Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1997-03

Analysis, experimental evaluation, and software upgrade for attitude estimation by the Shallow-Water AUV Navigation System (SANS)

Roberts, Ricky L.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/31936

Downloaded from NPS Archive: Calhoun

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ANALYSIS, EXPERIMENTAL EVALUATION, AND SOFTWARE UPGRADE FOR ATTITUDE ESTIMATION BY THE SHALLOW-WATER AUV NAVIGATION SYSTEM(SANS)

by

Ricky L. Roberts

March 1997

Thesis Co-Advisors: Robert B. McGhee
Eric Bachmann

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 3

19971121 123

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 reviewing instructions, 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
March, 1997

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
ANALYSIS, EXPERIMENTAL EVALUATION, AND SOFTWARE UPGRADE FOR ATTITUDE ESTIMATION BY THE SHALLOW-WATER AUV NAVIGATION SYSTEM (SANS)

5. FUNDING NUMBERS

6. AUTHOR(S)
Roberts, Ricky L.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Dept.
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

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 United States Government.

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The main problem addressed by this research is the lack of a small, low-cost integrated navigation system to accurately determine the position of an Autonomous Underwater Vehicle (AUV) during all phases of an underwater search or mapping mission. The approach taken utilized an evolving prototype, called the Shallow-Water AUV Navigation System (SANS), combining Global Positioning System (GPS), Inertial Measurement Unit (IMU), water speed, and magnetic heading information using Kalman, low-pass, and complimentary filtering techniques. In previous work, addition of a math coprocessor improved system update rate from 7 to 18 Hz, but revealed input/output coordination weaknesses in the software. The central focus of this thesis is on testing and programming improvements which resulted in reliable integrated operations and an increased processing speed of 40 Hz. This now allows the filter to perform in real time. A standardized tilt table evaluation and calibration procedure for the navigation filter also was developed.

The system was evaluated in dynamic tilt table experiments. Test results and qualitative error estimates using differential GPS suggest that submerged navigation with SANS for a period of several minutes will result in position estimation errors typically on the order of 10 meters rms, even in the presence of substantial ocean currents.

14. SUBJECT TERMS
Autonomous Underwater Vehicles, GPS/INS integration, navigation, NPS AUV, Low Pass Filtering, Complementary Filtering Kalman filtering, Attitude Estimation

15. NUMBER OF PAGES
209

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

ANALYSIS, EXPERIMENTAL EVALUATION, AND SOFTWARE UPGRADE FOR ATTITUDE ESTIMATION BY THE SHALLOW-WATER AUV NAVIGATION SYSTEM (SANS)

Ricky L. Roberts
Lieutenant-Commander, United States Navy
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1997

Author: [redacted]
Ricky L. Roberts

Approved by: [redacted]
Robert B. McGhee, Thesis Co-Advisor

[redacted]
Eric Bachmann, Thesis Co-Advisor

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

DTIC QUALITY INSPECTED 3

ABSTRACT

The main problem addressed by this research is the lack of a small, low-cost integrated navigation system to accurately determine the position of an Autonomous Underwater Vehicle (AUV) during all phases of an underwater search or mapping mission. The approach taken utilized an evolving prototype, called the Shallow-Water AUV Navigation System (SANS), combining Global Positioning System (GPS), Inertial Measurement Unit (IMU), water speed, and magnetic heading information using Kalman, low-pass, and complimentary filtering techniques. In previous work, addition of a math coprocessor improved system update rate from 7 to 18 Hz, but revealed input/output coordination weaknesses in the software. The central focus of this thesis is on testing and programming improvements which resulted in reliable integrated operations and an increased processing speed of 40 Hz. This now allows the filter to perform in real time. A standardized tilt table evaluation and calibration procedure for the navigation filter also was developed.

The system was evaluated in dynamic tilt table experiments. Test results and qualitative error estimates using differential GPS suggest that submerged navigation with SANS for a period of several minutes will result in position estimation errors typically on the order of 10 meters rms, even in the presence of substantial ocean currents.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. RESEARCH QUESTIONS 4

C. SCOPE, LIMITATIONS AND ASSUMPTIONS 5

D. ORGANIZATION OF THESIS 6

II. SURVEY OF RELATED WORK 7

A. INTRODUCTION 7

B. GPS NAVIGATION 8

C. INS NAVIGATION 12

D. INTEGRATED GPS/INS NAVIGATION 13

E. AUV SUBMERGED NAVIGATION 15

F. NAVIGATION FILTER THEORY 19

G. SUMMARY 22

III. SYSTEM HARDWARE CONFIGURATION 25

A. INTRODUCTION 25

B. HARDWARE DESCRIPTION 28

1. Computer 28
2. Inertial Measuring Unit 29
3. GPS/DGPS Receiver Pair 30
4. Compass 31
5. Other Components 32

C. SUMMARY 33

IV. SOFTWARE DEVELOPMENT 35

A. INTRODUCTION 35

B. SOFTWARE FILTER 38

C. IMPLEMENTATION DESCRIPTION 43

1. Compass Data 45
2. GPS Data 45
3. Inertial Sensor Data 46

4. Sampler 48
5. INS 52
6. Navigator 54
7. Communication Objects 56

D. SUMMARY 57

V. SYSTEM TESTING 59

A. INTRODUCTION 59

B. LOW PASS FILTER BIAS RESPONSE 59

C. FILTER TESTING METHODOLOGY 63

D. IMU TEST RESULTS 65

E. SUMMARY 76

VI. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK 77

A. CONCLUSIONS 77

B. RECOMMENDATIONS FOR FUTURE WORK 78

APPENDIX A: Real Time Navigation Source Code (C++) 81

C. TOWTYPES.H 81

D. TOEFISH.CPP 83

E. NAV.H 90

F. NAV.CPP 92

G. GPS.H 100

H. GPS.CPP 101

I. INS.CFG 102

J. INS.H 103

K. INS.CPP 105

L. SAM.CFG 116

M. SAMPLER.H 116

N. SAMPLER.CPP 119

O. COMPASS.H 124

P. COMPASS.CPP 125

Q. A2D.CFG 130

R. A2D.H 131

S. A2D.CPP 134

APPENDIX B: Serial Port Communications Source Code (C++) 155

T. GLOBALS.H 155

U. BUFFER.H 156

V. BUFFER.CPP 157

W. GPSBUFF.H 159

X. GPSBUFF.CPP 160

Y. COMPBUFF.H 163

Z. COMPBUFF.CPP 164

AA. SERIAL.H 167

AB. SERIAL.CPP 169

AC. GPSPORT.H 174

AD. GPSPORT.CPP 175

AE. COMPPORT.H 178

AF. COMPPORT.CPP 179

APPENDIX C: SANS TILT-TABLE TEST TUNING AND CALIBRATION PROCEDURE 183

A. Isolate Accelerometer Input From Integrator 183

B. Choose Initial Bias Weight (biasWght) 183

C. Determine Angular Rate Scale Factor 183

D. Adjust Gain Value Above Zero 183

E. Determine Accelerometer Scale Factor 183

F. Fine Tuning 183

LIST OF REFERENCES 185

INITIAL DISTRIBUTION LIST 189

LIST OF FIGURES

1. Discrete Low Pass Filter Block Diagram 21
2. Discrete Low Pass Filter Signal Flow Graph 21
3. Redesigned SANS Hardware Configuration (Walker 96) 26
4. SANS Hardware Configuration (Walker 96) 27
5. E.S.P. 486SLC DX2 50 MHz Computer (Walker 96) 28
6. Systron-Donner Inertial Measuring Unit (Bachmann 95) 30
7. ONCORE GPS/DGPS Receiver (Walker 96) 31
8. SANS Code Classes and Objects 37
9. SANS Software Navigation Filter (Bachmann 96) 40
10. Complementary Filter Feedback Loop for Euler Angle Estimation 42
11. SANS Data Flow Between Software Objects 44
12. Buffer Data Structures 46
13. samplerClass Summary 49
14. samplerClass Data Flow 50
15. Model of the A2D Sample Array (Walker 96) 51
16. insClass Summary 53
17. Navigation Class and Initialization Summary 55
18. Navigation Position Format Utilization (Bachmann 95) 57
19. Bias Filter Response to a Roll Rate Step Input of 10°/sec 60
20. Estimated Short Term Bias Response to a 45° Roll Completed in 4.5 Seconds 62
21. Estimated Long Term Bias Response to a 45° Roll Completed in 4.5 Seconds 62
22. Initial Pitch Test, $K_1 = 0.0$, $\tau = 1000$, qScale = 4.02, 10°/sec 67
23. Pitch Test, $K_1 = 0.0$, $\tau = 5000$, qScale = 4.02, 10°/sec 67
24. Initial Roll Test, $K_1 = 0.0$, $\tau = 1000$, pScale = 4.01, 10°/sec 68
25. Roll Test: $K_1 = 0.0$, $\tau = 1000$, pScale = 4.01, 40 °/sec 69
26. Roll Test: $K_1 = 0.01$, $\tau = 1000$, pScale = 4.01, 40 °/ sec 69
27. Roll Test: $K_1 = 0.05$, $\tau = 200$, pScale = 4.01, 40 °/sec 70
28. Roll Test: $K_1 = 0.05$, $\tau = 200$, pScale = 4.01, 10°/sec 71
29. Same as Previous, with yAccelScale = 1.405 71
30. Roll Test: $K_1 = 0.1$, $\tau = 1000$, 10 /sec, yAccelScale = 1.405 72
31. Roll Test: $K_1 = 0.1$, $\tau = 1000$, 10 /sec, yAccelScale = 1.317 73
32. Roll Test: $K_1 = 0.1$, $\tau = 1000$, pScale = 3.923, 10°/sec 73
33. Roll Test: $K_1 = 0.1$, $\tau = 1000$, yAccelScale = 1.347, 10°/sec 74
34. Roll Test: same as previous, with 45°/ sec vice 10 75
35. Roll Test: same as previous, with 90°/sec vice 45 75

LIST OF TABLES

1 Expected RMS GPS Accuracy Levels (Logsdon 92) 10
2 State Variables of the Kalman Filter (Bachmann 96) 39
3 A2D DC-to-Digital Conversion Mapping (Walker 96) 47

ACKNOWLEDGMENTS

The most important of a group of people whose support made completion of this work possible is my wife, Julie. She delivered and performed the lion's share of the early care for our first child, Tristan, while I was devoting long hours to my studies. Her support at home enabled me to remain focused and, along with the joy of Tristan's arrival and daily growth, have provided essential perspective and emotional release. Thank you, Julie.

My appreciation to Dr. McGhee can not be overemphasized. He is as fine a teacher as I have had the pleasure of learning from. His genuine concern for the students is obvious from his priority of learning over administrative research milestones. Dr. McGhee's real world perspective unfailingly kept me from losing sight of the real goals. His incredible patience is reflected in his achievement of taking a student with no real computer science or kinetics background, and very little electrical engineering background, and enabling me to not only understand the SANS filter, but be able to improve upon the software.

Eric Bachmann went far beyond providing the historical background to get me started and reviewing this thesis. He never hesitated to help with the myriad of issues which at any time could have overwhelmed me. Always ready to answer even the most mundane questions or to provide assistance on a major software design consideration, his efforts as Co-Advisor were crucial to successful completion.

Last, but on a routine, day-to-day basis, most important, was the assistance and friendship of Russ Whalen. From sharing office space and providing encouragement and counsel to unfailingly supplying computer expertise and hardware support, Russ is the cornerstone of the continued success of the SANS project. I only wish that as students we could provide something in return to him other than incessant demands on his time and expertise.

Dedication

For Julie and Trey

I. INTRODUCTION

A. BACKGROUND

Autonomous Underwater Vehicles (AUVs) are capable of a variety of overt and clandestine missions. Such vehicles have been proposed for inspection, mine countermeasures, survey, and observation. Recent research trends in underwater robotics have emphasized minimizing the need for human interaction by increasing AUV autonomy. (Yuh 95)

The NPS *Phoenix* AUV is an experimental vehicle designed primarily for research in support of shallow-water mine countermeasures and coastal environmental monitoring (Healey 93, 95, Brutzman 96). The clandestine nature of the missions for which Phoenix was designed necessitates minimum surfaced exposure time while in the operating area, the ability to submerge in order to investigate targets, and a navigation system that is accurate enough to allow target revisit if desired.

Many missions of the Phoenix class of vehicles can be separated into two distinct phases: transit and search. After being launched from an aircraft, submarine, or surface vessel, the AUV would execute a transit phase in order to arrive at the search area. Once established in the mission area, it would enter a search phase, which might include missions such as mine hunting, mapping, or environmental data collection. Navigation is one of the most important and difficult aspects of an AUV mission. Therefore, a robust, real-time navigation system is critical for a multi-mission capable AUV. Typically, a search phase would require more precise navigation than a transit phase. This could be accomplished by obtaining more frequent Global Positioning System (GPS) fixes, or by using Differential

GPS (DGPS) either in real-time if available, or after mission completion using post-processing (Walker 96). After the search is completed, the AUV would commence a second transit phase and return to a recovery position. Both kinds of mission phases would typically involve waypoint steering, and possibly obstacle avoidance.

An approach is described in Kwak (93) for determining the position of submerged detected objects by executing a "pop-up" maneuver to obtain a GPS fix. This fix is then extrapolated backwards to the submerged object location using recorded inertial data. Navigation accuracy during such a surfacing maneuver is strongly enhanced by the use of accurate depth information available from low-cost pressure cells (Kwak 93). However, this form of "aided" inertial navigation is not applicable to a surfaced or near surface AUV (Brown 92).

Continuously reliable GPS reception would not be possible unless the AUV were to be fitted with an extensible mast mounted antenna. Extending an antenna above the effects of wave action is not desirable for a military application and, at any rate, would probably be mechanically impractical for a small AUV. As a result, any system relying solely upon GPS would not be sufficiently robust to provide accurate navigation information during surfaced or near surface operations due to intermittent reception. Therefore, inertial navigation is needed between periods where continuous reliable reception of GPS satellite signals is not possible. (Bachmann 95)

Inertial navigation hardware is sometimes based on rotating gyros, which provide attitude information needed to stabilize a platform that holds acceleration sensors. The limiting factors to this approach include: high expense due to required precision, inordinate

power consumption, high failure rates, and acoustic and structure-borne noise (Cox 94). These factors counter the Phoenix AUV philosophy of providing a low cost, general purpose platform capable of long-term independent operation, despite relatively small vehicle size (McGhee 95). Additionally, the rotating gyros now installed in Phoenix are aging and mechanically unreliable. It is therefore desirable to find a solution to the AUV navigation and control problem not requiring such components.

In order to achieve robust navigation, the AUV should be capable of navigating with GPS and/or an Inertial Navigation System (INS). GPS is capable of highly accurate positioning when the AUV is surfaced, while an INS can be used for submerged navigation and periods between GPS satellite reception. In order to ensure accurate navigation for a wide variety of missions, GPS and INS components can be combined. A favorable analysis of this type of navigation system was conducted in McKeon (92). The hardware and software architecture required for a typical mapping scenario was evaluated in Norton (94).

Bachmann (95) made the architecture evaluated in Norton (94) a reality, and subsequently developed the first working prototype of the proposed Shallow-Water AUV Navigation System (SANS). The SANS was designed to overcome the problem of intermittent GPS satellite tracking. It is an experimental system that uses a low-cost, strapped-down inertial measurement unit (IMU), complemented with magnetic heading and water speed sensors, to enable inertial navigation between GPS fixes. This system is well suited for pop-up navigation. Finding this means of navigating near the sea surface provides a complete solution to the overall navigation problem associated with transiting an AUV to a shallow water work site, recording the position of detected submerged objects,

and then returning to a recovery site where stored mission data can be uploaded (McGhee 95).

Additionally, the navigation filter developed by McGhee (95) solves the problems of cost and power consumption by eliminating rotating gyros and replacing them with acceleration and angular rate sensors. This filter is implemented in SANS by Bachmann (95). One application of SANS is to upgrade the Phoenix navigation system. Others, particularly as component miniaturization continues, include marine mammal and diver navigation.

With the prototype SANS having achieved favorable results in open-water, at-sea test trials, Walker (96) advanced the SANS to another level of maturity, making it a truly integrated system ready for direct application to a real-world AUV. The physically redesigned system includes an on-board processor and consolidated the diverse components into a compact unit, while improving individual component reliability and performance. The research reported in this thesis continues the evolution of the SANS by incorporating software improvements to accommodate the dramatically improved processing speed, implementing a networking capability to monitor at-sea tests and prepare for installation into the AUV, and developing a standardized calibration procedure for the navigation filter.

B. RESEARCH QUESTIONS

This thesis will examine the following research topics:

- Evaluate the hardware and software architecture of the SANS.
- Develop a calibration procedure for the SANS navigation filter.

- Evaluate the performance of the SANS navigation filter in a laboratory environment.
- Evaluate the SANS hardware and software architecture for installation in Phoenix.

C. SCOPE, LIMITATIONS AND ASSUMPTIONS

This thesis reports part of the findings of the fifth year of research in an ongoing research project. The scope of this thesis is to evaluate SANS attitude estimation capabilities for eventual installation as a replacement for the older technology gyros now used on board the Phoenix AUV. The requirements for an ideal SANS described by Kwak (93) which impact this project are:

- Low power consumption. Operation from a small external battery pack for 12 hours is desirable.
- Limited exposure time. The amount of time that the GPS antenna is exposed in the search phase should be as short as possible. Up to 30 seconds of exposure is allowed, but less is better, and time between exposures should be maximized.
- Maintain clandestine operation. The GPS antenna should present a very small cross section when exposed and should not extend more than a few inches above the surface of the water.
- Maximize accuracy. During the search phase of the mission, system accuracy of 10 meters or better is required following postprocessing, both while submerged and surfaced.
- Total volume not to exceed 120 cubic inches. Elongated, streamlined packaging is preferred.

D. ORGANIZATION OF THESIS

The purpose of this thesis is to present the development of a prototype system intended to meet the mission requirements of the SANS. The term AUV is understood to include any small underwater vehicle (including human divers) which can easily carry such a compact device. The term "towfish" refers to the test vehicle used to evaluate the SANS during at-sea testing.

This thesis provides an evaluation of the hardware and software used to provide accurate navigation for the NPS AUV. The major thrust of the thesis is to evaluate the attitude estimation capabilities of the SANS both statically and dynamically in a laboratory environment.

Chapter II reviews previous work on this project as well as on GPS and INS navigation, AUV submerged navigation, and navigation filtering theory. Chapter III provides a summary description of both the original and current SANS prototype hardware. Chapter IV provides a detailed description of the software architecture, including the navigation filter. Particular emphasis is placed on changes, additions, and updates made to the C++ code in support of this portion of the project. Chapter V is a description of the experiment design and an analysis of the experimental results. Finally, Chapter VI presents the thesis conclusions and provides recommendations for future research.

II. SURVEY OF RELATED WORK

A. INTRODUCTION

Autonomous Underwater Vehicles (AUVs) have the potential to be used in an efficient and cost effective manner in a variety of missions involving military and non-military applications. Accurate navigation is one of the most important capabilities supporting AUV mission effectiveness. Many possible AUV missions, such as mine hunting, require a high degree of navigation accuracy. This chapter will discuss some of the possible AUV navigation solutions.

Navigation systems are generally categorized by whether they are based on external signal reception or internal sensors. External-signal-based navigation systems, such as Loran, Omega, and Global Positioning System (GPS), are limited to determining position only while the receiver is exposed to the signal. Loran and Omega are relatively inaccurate compared to GPS. While Loran covers most of the northern hemisphere, it has almost no coverage in the southern hemisphere (Bowditch 84). GPS provides an attractive, affordable system for the surfaced portion of an AUV mission because it is capable of world-wide coverage with a high degree of navigational accuracy.

Internal-sensor-based navigation can be implemented as a self-contained unit which can be composed of various types of equipment such as inertial measuring units (IMUs), acoustic transponders, or geophysical map comparison. All sensors are subject to some amount of error, which may compound to unacceptable levels for some AUV missions if not accounted for. Each of these components also has unique disadvantages. Acoustic transponders must be pre-deployed at precisely known locations and may require costly

maintenance. Geophysical map interrogation requires a precise bottom contour map be previously stored in the AUV's computer. IMU-based navigation is prone to sensor drift, which if left uncorrected, can become very large. However, it has advantages relative to the other navigation options due to a lack of dependence on external signals and no requirement to transmit any signals which might reveal its presence.

B. GPS NAVIGATION

The Navigation Satellite Timing and Ranging (NAVSTAR) Global Positioning System (GPS) is a space-based radio positioning, navigation and time-transfer system sponsored by the U.S. Department of Defense (DoD). It was originally intended to provide the military with precise navigation and timing capabilities (Parkinson 80). The system is designed to provide 24-hour, all-weather navigation anywhere on earth. It is comprised of 24 satellites in 22,200 km orbits that are inclined at 55° to the earth's spin axis, with 12 hour periods. The satellites broadcast two L-band frequencies: L1 (1575.4 MHz) and L2 (1227.6 MHz). Navigation and system data, predicted satellite position (ephemeris) information, atmospheric propagation correction data, satellite clock error information, and satellite health data are all superimposed on these two carrier frequencies. (Logsdon 92, Wooden 85)

There are two different navigation services available from the GPS satellites depending on the type of receiver being used: the Standard Positioning Service (SPS), and the Precise Positioning Service (PPS). The SPS is based on receiving the L1 carrier signal, which is broadcast with an intentional inaccuracy called Selective Availability (SA). SA limits world-wide navigation to 100 m horizontal accuracy with a 95% confidence level (Logsdon

92). PPS is based on the L2 signal. It is limited to U.S. and allied military, and specific non-military uses that are in the national interest. Access to PPS is restricted by use of special cryptographic equipment. PPS provides the highest stand alone accuracy: 16 m Spherical Error Probable (SEP), a velocity accuracy of 0.1 m/sec, and a timing accuracy of better than 100 nanoseconds. (Logsdon 92, Wooden 85)

Civilian customers have determined a way to improve the accuracy of the SPS in order to take full advantage of GPS precision without having access to cryptographic equipment. This method, called Differential GPS (DGPS), provides a way of working around the inaccuracies of the SPS. It may be used in real-time or during post-processing. The general idea is to place a receiver at a surveyed stationary site. The receiver is then able to determine the difference between its actual position and its computed GPS position, and broadcast the resulting pseudorange (distance to satellite) corrections to any DGPS capable receivers. Real-time differential processing can reduce the typical 100 m accuracy of the SPS to 2-4 m regardless of the status of SA (Logsdon 92). It is also possible to record the raw PPS or SPS GPS information for later comparison to a known geographical site using post-processing. Precise procedures can be used to reconstruct extremely accurate positioning information, typically in the submeter range. Table 1 shows a comparison of expected GPS accuracies.

The size and cost of GPS receivers have decreased drastically as GPS technology has matured. Miniaturization is continuously progressing while maintaining or increasing GPS receiver performance capability. Since as early as 1992, the GPS industry has been able to produce receivers that are essentially a single printed circuit board. Souen (92) reports

POSITIONING SERVICE	PPS (m)	SPS (m)
Non-Differential	16	100
Differential	2-4	2-4

TABLE 1: Expected RMS GPS Accuracy Levels (Logsdon 92)

that the Furuno GPS receiver module LGN-72 is an eight-channel receiver implemented on a single printed circuit board measuring 100 mm x 70 mm x 20 mm and requiring only 2 W of power.

There is currently a performance trade-off associated with the miniaturization of GPS receivers. For instance, Trimble offers the PC Card 110 GPS miniature receiver in the form of a Personal Computer Memory Card International (PCMCIA) interface. This credit card-sized device simply slides into any laptop, most palmtops, or pen-based computers compliant with PCMCIA (release 2.0). It is capable of tracking eight satellites using three channels. However, because it does not have an allocated channel for each of the satellites, it does not use a continuous tracking scheme. This degrades its acquisition time performance. In order to reduce receiver size, manufacturers often reduce the number of channels on the receiver. GPS receivers in this configuration are called "sequencing" receivers (Logsdon 92). Sequencing receivers utilize a time-sharing technique to "dwell" on each satellite for a brief interval before switching to the next satellite in the sequence. They have a typical acquisition time of about two minutes. Continuous tracking GPS receivers have typical acquisition times of about 30 seconds or less. However, their larger number of receiver channels results in a less compact size. Given this trade-off between

size and performance, the choice of GPS receiver must be made with the particular application in mind. A sequencing receiver offers an adequate compromise for applications such as mobile navigation that are not so dynamic. However, if the application requires a short time to initial acquisition, the most viable option is the continuous tracking receiver. GPS is an obvious choice for AUV navigation given the level of miniaturization and its excellent accuracy performance.

One manner of using GPS to locate an AUV is to place buoys with GPS receivers at appropriate locations. These buoys would translate the GPS signal and retransmit an underwater acoustic signal. The AUV would then determine its position via ranging and position fixes to the buoys. Youngberg (91) suggests that the GPS antenna, receiver, processing and control subsystem, acoustic transmitter, battery power, and homing beacon could all be contained in a buoy measuring 123 mm diameter x 910 mm long and weighing 5 - 15 kg. A simulation which showed the feasibility of this approach is presented in Leu (93). The simulation consisted of several sonobuoys spaced one kilometer apart. Due to uncertainties in buoy position caused by wave action and variations in altitude, the study proposed the use of Kalman filtering techniques to combine the outputs of an accelerometer and DGPS to enhance accuracy. Each GPS buoy would essentially act as a GPS satellite and broadcast its position via spread spectrum acoustic signals used by the AUV for ranging. This technique would eliminate the requirement to predeploy a surveyed transponder field.

Another possible method for using GPS to determine the AUV's position is to physically mount the GPS antenna and receiver on-board the AUV. For areas covered by

DGPS service, this has the advantage of making the system self-contained. One major concern would be that the GPS receiver would be unable to acquire satellites in a timely manner due to splash effects on the antenna. However, Norton (94) describes both static and dynamic test results which show that a submersible system is able to meet the accuracy and time requirements of the mission, even while being splashed by wave wash. Therefore, this method was adopted in the SANS configuration.

C. INS NAVIGATION

Inertial navigation is essentially a complex method of dead reckoning. Its purest form involves no outside references to fix position. All position data is calculated relative to a known starting point. An inertial navigation system (INS) continuously measures three mutually orthogonal acceleration components using accelerometers. These measurements are taken in short time increments and multiplied by elapsed time in order to determine an estimate of instantaneous velocity. The three-dimensional change in position can then be determined by integrating respective velocities over time. (Bachmann 95)

The primary drawback of any INS is the tendency for small sensor drift rates to accumulate as errors over time. Without outside references for correction, these errors grow relentlessly and eventually lead to large errors in the estimated position. Highly accurate inertial navigation systems can be constructed, but they are large, costly, and complex (Touhy 93). Size alone makes them unacceptable for the SANS. A compromise solution to meet SANS requirements is to integrate a low-cost, miniature INS with GPS. In such a system, GPS will provide the INS with the periodic position fixes necessary to correct slowly building INS errors.

The acceleration measurements required by an INS can be made by several types of IMUs. There are two fundamental categories: gimbaled and strapdown. Due to their large size and power requirements, gimbaled systems are not suitable for the SANS. In a strapdown unit, three mutually orthogonal accelerometers and three angular rate sensors are mounted parallel to the three body axes of the vehicle. Linear accelerations and rotational velocities are continuously measured. Strapdown systems are smaller and simpler than gimbaled systems, but necessitate much larger computational capabilities. (Logsdon 92)

D. INTEGRATED GPS/INS NAVIGATION

SPS mode GPS navigation could be used to adequately perform both the transit and search phases of an AUV mission. During surfaced transit phases, non-differential SPS, a water speed sensor, and a magnetic compass would provide the primary source of navigation data. In order to utilize GPS as a meaningful correction to a low-cost INS system, periods between GPS fixes during the transit phase must not exceed the time in which the INS error has accumulated to an amount comparable to the horizontal accuracy of SPS (100m) (Bachmann 95). The search or mapping phases of an AUV mission would require the vehicle to maintain a more accurate navigational picture, both submerged and on the surface. This would necessitate the use of periodic differentially corrected GPS information in order to keep the INS system accurate while submerged. This differential correction could be provided in real-time during overt missions along friendly shores where a DGPS reference signal is available, or during mission post-processing following a clandestine mission.

Integration of GPS and INS into a single system can produce continuously accurate navigational information even when using relatively low-cost components. This integration not only allows periodic reinitialization of the INS to correct accumulated errors, but can also (with the aid of Kalman filtering techniques) improve the performance of the INS between fixes. Complementary filtering of acceleration data with additional sensor information such as water speed and heading will further improve system accuracy. Overall, an integrated system will provide improved reliability, smaller navigation errors, and superior survivability. (Logsdon 92)

Kalman filtering is a method of combining all available sensor data, regardless of their precision, to estimate the current posture of a vehicle (Cox 90). The filter is actually a data-processing algorithm which minimizes the error of this estimate statistically using currently available sensor data and prior knowledge of system characteristics. Each piece of data is weighted relative to data from other system components based upon the expected accuracy of the measurement it represents. In a *complementary* filter, low-frequency data, which is trusted over the long term, and high-frequency data, which is trusted only in the short term, are used to "complement" each other providing a much better estimate than either can alone. (Brown 92)

Bachmann (95) demonstrated the use of the complementary filter technique by combining low-frequency orientation data from accelerometers and a magnetic compass with high-frequency angular rate information to estimate heading and attitude. Intermediate position results were obtained by integrating high-frequency water-speed data. GPS data was used to reinitialize the system each time a fix was obtained and to

develop an error bias, expressed as an apparent ocean current. The current was utilized to correct the system between GPS fixes. The concept of using a relatively inexpensive IMU with limited accuracy, coupled with differentially-corrected GPS, has proven to be a viable solution to the challenge of shallow-water AUV navigation. (Bachmann 95)

The above conclusion has been independently duplicated in Wolf (96). Utilizing an integrated GPS/INS system using the same Systron-Donner IMU used in SANS, but without incorporating DGPS, accuracies in attitude of better than 0.2° in roll and pitch and 0.3° in azimuth were achieved. Specific results from those tests, along with static tests indicating the SANS software filter (described in Chapter IV) response to IMU inputs are discussed further in Chapter VI, System Testing. (Wolf 96)

E. AUV SUBMERGED NAVIGATION

There are many techniques available for submerged navigation, including dead reckoning, inertial, electromagnetic, and acoustic navigation. With acoustic navigation, time of arrival and direction of propagation of acoustic waves are the two principal measurements made. A wide variety of acoustic navigation systems have been developed for underwater vehicle use. They are typically divided into long, short, and ultrashort baseline systems. All involve the use of acoustic beacons or receivers whose positions must be known to an accuracy somewhat better than the desired vehicle localization accuracy (Tuohy 93). Unfortunately, most acoustic navigation systems require major expeditions for their accurate set-up and periodic maintenance. This makes them expensive, and in many ways reduces the level of autonomy achievable by an AUV. Also, acoustic methods are affected by changes in the speed of sound in the ocean and suffer from

refraction and multipath propagation problems in restricted shallow water coastal and ice-covered areas. (Tuohy 93)

There are various alternative submerged navigation methods not dependent upon the aid of external signals. Charge Coupled Device cameras, laser scanning, or variations in the earth's magnetic field can aid in determining position (Bergem 93). Position can also be estimated by the double integration of acceleration as sensed by an IMU.

Doppler sonar or correlation velocity log sensors can be utilized to determine speed through the water or over the ground. Doppler velocity logs utilize the physics of frequency shifts in the sound waves of sources and receivers with relative radial motion. A critical assumption for two-way transmission in the ocean is that the sound scatterers, (small particles and plankton) uniformly populate the environment, and at the average move at the same horizontal velocity as the water. Correlation velocity logs, on the other hand, use reflections from the sea bottom, even at great depths, and on-board sensor arrays to detect forward and lateral motion occurring between sonar pings. (Gordon 96)

Doppler technology has been redesigned as the Acoustic Doppler Current Profiler (ADCP). The ADCP measures water velocity more accurately, and allows measurement in range cells over a depth profile. Throughout the 1980's, ADCPs were further improved by production of self-contained, vessel-mounted, and direct-reading models, and by the addition of broadband capability in 1991. Broadband ADCPs take advantage of having typically 100 times as much bandwidth for measuring velocity as the original, narrow-bandwidth models, reducing variance nearly 100 times. (Gordon 96)

Broadband Doppler processing computes the phase change of propagation time delay. Since longer propagation times provide greater accuracy, but incur phase changes beyond 360°, a mathematical autocorrelation function resolves ambiguity and allows transmission of a series of coded pulses within a single long pulse. Multiple beams are utilized to obtain velocity in three dimensions, under the assumption of uniform currents across layers of constant depth. Non-homogenous current layers produce large velocity errors. (Gordon 96)

ADCP single-ping random or short-term error may range from just a few mm/s to as much as 0.5 m/s, depending on internal factors such as frequency, depth cell size, number of pings averaged together, and beam geometry. Since this random error is uncorrelated from ping to ping, the standard deviation of the velocity error can be reduced by the square root of the number of pings through averaging. Although averaging can greatly reduce the relatively large, single-ping error, at a certain point it fails to improve on overall error as the random error becomes smaller than the bias. (Gordon 96)

The bias is typically less than 10 mm/s and depends on factors such as temperature, mean current speed, signal/noise ratio, and beam geometry. It is not yet possible to measure ADCP bias and calibrate or remove it in post-processing. External error factors include turbulence, internal waves, and ADCP motion, and can dominate internal errors. While the technology behind the ADCP is impressive and bears serious consideration for future small AUV navigation development, the combination of relative affordability and unpredictable bias make it a les attractive option for the SANS application. (Gordon 96)

For covert missions, an AUV may not be able to refer to external signals while submerged. In this case, the system must rely on some sort of dead reckoning. Modern dead reckoning systems typically use magnetic or gyroscopic heading sensors, and a bottom or water-locked velocity sensor (Grose 92). The presence of an ocean current will add a velocity component to the vehicle which is not detected by a water speed sensor. In the vicinity of the shore, ocean currents can exceed two knots (Tuohy 93). Using dead reckoning with currents which are relatively large in relation to the typical 4-6 knot speed of an AUV can produce extremely inaccurate results (Tuohy 93). This inaccuracy represents the central challenge of AUV submerged dead reckoning navigation.

There are many techniques for measuring acceleration and angular rates. These include using ring laser and fiber optic gyros, rotating mass gyros, vibratory rate sensors, and high performance IMUs. Inertial grade IMUs typically contain three angular rate sensors, three precision linear accelerometers and a three-axis magnetometer. The acceleration measurements required by an INS can be made by several types of IMUs. All of these sensors are subject to drift errors which relentlessly increase with time. High quality sensors are subject to less drift, but can cost up to $100,000 (Tuohy 93), making them unattractive for small AUVs.

McKeon (92) proposes a combination of GPS and INS to allow an AUV to determine position information. While submerged, the AUV uses a low-cost inertial navigation system. However, when on the surface, the vehicle has access to GPS information. GPS/ INS information could be combined with Kalman filter techniques to reduce errors during the next dive sequence as simulated in Nagengast (92) and demonstrated in McGhee (95).

The system described in McGhee (95) senses linear accelerations and angular rates with respective sensors and processes the data in a twelve state Kalman filter, resulting in an estimated position. A mechanical water speed sensor and a magnetic compass are added to complement acceleration and angular rate data and further enhance navigation accuracy. The twelve states can be divided into seven continuous-time states (three Euler angles, two horizontal velocities, two horizontal positions), two discrete-time states derived from the DGPS fixes (estimated east and north current), and three angular rate sensor bias estimates, (subtracted from the output of these sensors). The DGPS fixes occur aperiodically whenever the vehicle surfaces and is able to acquire a sufficient number of satellites. (Bachmann 96)

F. NAVIGATION FILTER THEORY

The inherent sensor measurement errors plaguing inertial measurement systems may be partially compensated for, but never eliminated. Drift is the tendency of bias errors in the angular rate sensors of the inertial platform to cause relentlessly increasing orientation measurement errors. The single integration of a bias-ridden angular rate signal will cause a steady build-up of error over time. This leads to an incorrect estimation of the body orientation relative to the earth-fixed coordinate system and a corresponding body position estimate error. Angular rate sensor biases typically change unpredictably over time, making a simple, complete compensation impossible. (Frey 96)

Standard inertial navigation procedures utilize fix updates if an alternative method of determining instantaneous orientation exists. Drift is compensated for by periodic adjustments of the inertial system to the external reference, returning the bias error

accumulation to zero. Short fix intervals then result in relatively insignificant bias errors. Higher quality angular rate sensors typically have lower bias errors and correspondingly longer fix intervals. (Frey 96)

Linear acceleration sensor drift errors are compounded by the double integration of the linear acceleration measurements to obtain position data. This results in a position estimate in error proportional to time-squared, rather than simply time. This error may be similarly compensated. However, given the same sensor quality, the fix interval needed to maintain comparable accuracy will be much shorter than that required for the angular rate sensor bias compensation alone. (Frey 96)

Discrete low pass filter theory provides a method for obtaining a rate bias estimate. Such filters may be represented by a signal-flow graph (SFG), which is a simplified version of a block diagram. The SFG was introduced by S. J. Mason for the cause-and-effect representation of linear systems that are modeled by algebraic equations (Kuo 95). A SFG may be defined as a graphical means of portraying the input-output relationships between the variables of a set of linear algebraic equations, or simply

$$output = \sum gain \times input \qquad \text{Eq (2.1)}$$

Corresponding block and signal flow graph diagrams for a single input discrete low pass filter are shown in Figures 1 and 2 below.

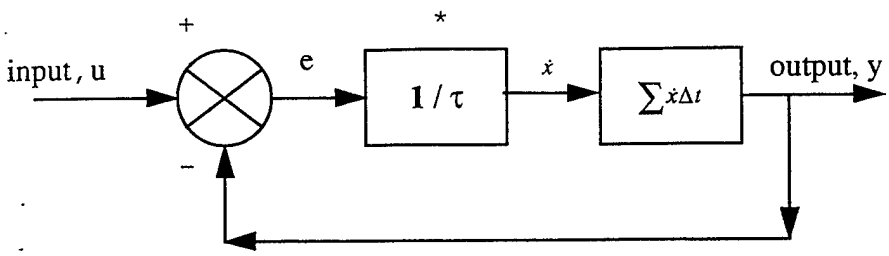

Figure 1: Discrete Low Pass Filter Block Diagram

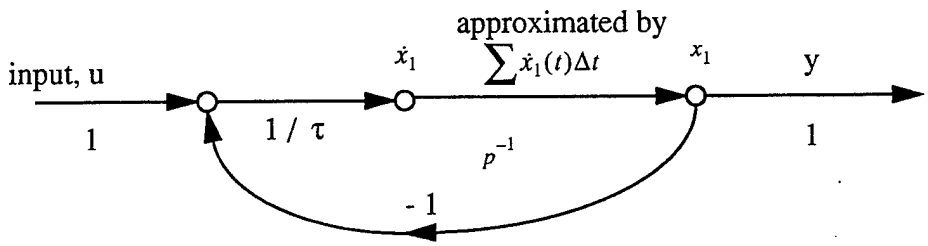

Figure 2: Discrete Low Pass Filter Signal Flow Graph

In this diagram, p^{-1} stands for the time domain integration operator, and tau is the relaxation time constant. Directly from Figure 2,

Eq (2.2)

$$x_1(t+\Delta t) = x_1(t) + \dot{x}(t)\Delta t$$

or

Eq (2.3)

$$\text{new output} = \text{old output} + \frac{\text{input} - \text{old output}}{\tau}\Delta t$$

This is the classic relationship describing a low pass filter (McGhee 96). Rewritten, Equation 2.3 becomes

Eq (2.4)

$$x_1(t+\Delta t) = x_1(t) + \frac{(input - x_1(t))\Delta t}{\tau}$$

Which can also be written

Eq (2.5)

$$x_1(t+\Delta t) = x_1(t) + input\frac{\Delta t}{\tau} - x_1(t)\frac{\Delta t}{\tau}$$

and, finally

Eq (2.6)

$$x_1(t+\Delta t) = x_1(t)\left(1-\frac{\Delta t}{\tau}\right) + input\frac{\Delta t}{\tau}$$

or, in more common terminology, and the terms used in the SANS code

Eq (2.7)

$$\text{new output} = \text{outputWeight} \times \text{old output} + \text{input} \times \text{sampleWeight}$$

The above general result can be applied to the SANS system for rate sensor bias estimation. In this case, the signal used for attitude estimation is the raw rate sensor reading with the estimated bias subtracted. An alternative formulation is to add the negative of the bias to the sensor reading. This formulation is derived similarly, and is implemented in the SANS code as,

Eq (2.8)

$$\text{new negative bias} = \text{biasWeight} \times \text{old negative bias} - \text{input} \times \text{sampleWeight}$$

In this form, the bias estimation integrator is initialized to a negative average value and the bias is then added to the sensor input.

G. SUMMARY

Many approaches to the problem of AUV navigation have been devised. New ones are still emerging and technological improvements are improving current approaches. Choices range from simple dead reckoning, to systems which use acoustic information from floating or stationary transponders, to complex systems which use sophisticated IMUs and GPS receivers combined with Kalman filtering techniques. Most of the described

approaches can only be used in very specialized applications. Most are also limited by dependence on previously deployed external means and by some requirement to actively exchange data with those means. The preferred method of many developers is the acoustic approach. However, most of these systems have a higher degree of complexity and dependence on external means than the system implemented in McGhee (95).

It can be seen that high accuracy and other design goals for an inertial navigation system are achievable. But clearly, the cost increases rapidly with the degree of sophistication and the desired precision. From this point of view the NPS Phoenix AUV, described in Healey (94), together with the SANS navigation system developed by Bachmann (95), McGhee (95), Steven (96), and Walker (96), promises to provide a very effective means for achievement of clandestine missions in shallow water by a small AUV.

The remainder of this thesis continues an ongoing experimental study pertaining to the development of the SANS system and associated problems. The current system under evaluation is of small physical size and relatively low cost. The IMU selected is representative and has limited accuracy, so additional water-speed and magnetic heading information is required. Accelerometers are used mainly to derive low frequency attitude information, and are not utilized for velocity or position estimation for periods of more than a few seconds.

Previous research on the prototype SANS has produced test results and qualitative error estimates which indicate that submerged navigation accuracy comparable to GPS surface navigation is attainable (Bachmann 95). The research goal of this thesis is to refine the hardware and software configuration to allow more accurate submerged navigation, and

to develop the SANS into a self contained system capable of being internally or externally attached to any AUV, delivering regular, accurate, real-time position updates.

III. SYSTEM HARDWARE CONFIGURATION

A. INTRODUCTION

Bachmann (95) describes the initial prototype in the ongoing development of the SANS. Walker (96) redesigned the original prototype to consolidate components in one integrated system. In addition, he presented an evaluation summary of the original prototype hardware, with particular emphasis on the noise characteristics of the Systron-Donner MotionPak IMU, which is retained in the SANS.

Figure 3 presents a block diagram for the hardware making up the redesigned SANS. Figure 4 presents a photograph of the SANS components fully assembled into their testing configuration. The project box in which the components are currently mounted is an interim solution. A more permanent, water-tight, streamlined housing is currently in development.

This configuration is significantly different from the previous prototype presented in Bachmann (95). The SANS components are no longer separated; all components are physically located in one self-contained package. When joined with its accompanying power source (a 12 VDC battery), the complete system can now be strapped-down to a tilt table or inserted into a towfish for at-sea testing. In its current configuration, the SANS has its processor and GPS/DGPS components "on-board," thus no longer requiring the transfer of sensor data via modem to an external processor or GPS/DGPS receiver. (Bachmann 95, Walker 96)

The SANS processor is linked with an external processor via a DOS TCP/IP network connection to allow for human monitoring and interaction during the course of an

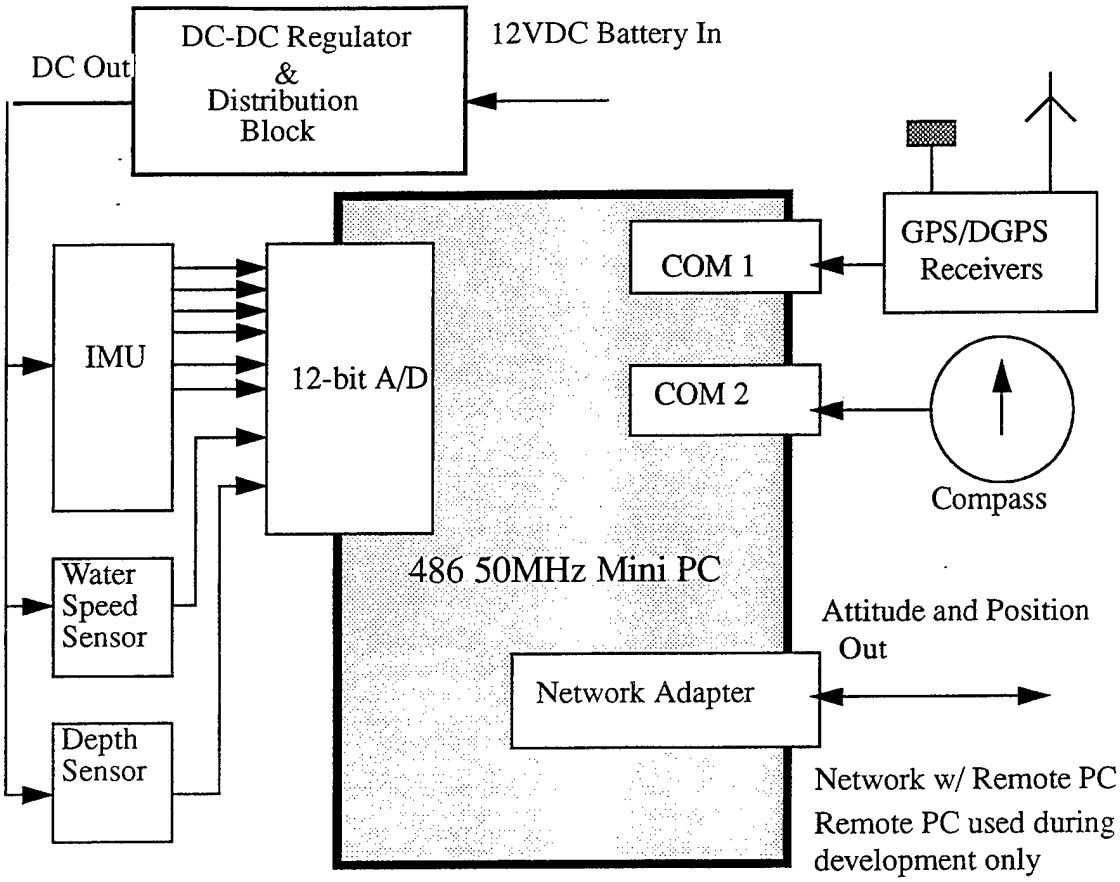

Figure 3: Redesigned SANS Hardware Configuration (Walker 96)

experiment. This external processor's only function is to maintain a remote control session with the SANS processor and receive its attitude and position updates. Unlike the original SANS proof of concept design presented in Bachmann (95), the SANS now maintains the capability to on-board process its own data and interface with any other higher-level processor via a network. This capability will directly enable smooth incorporation of SANS into the Phoenix architecture. This chapter will summarize the hardware component capabilities realized in Walker (96).

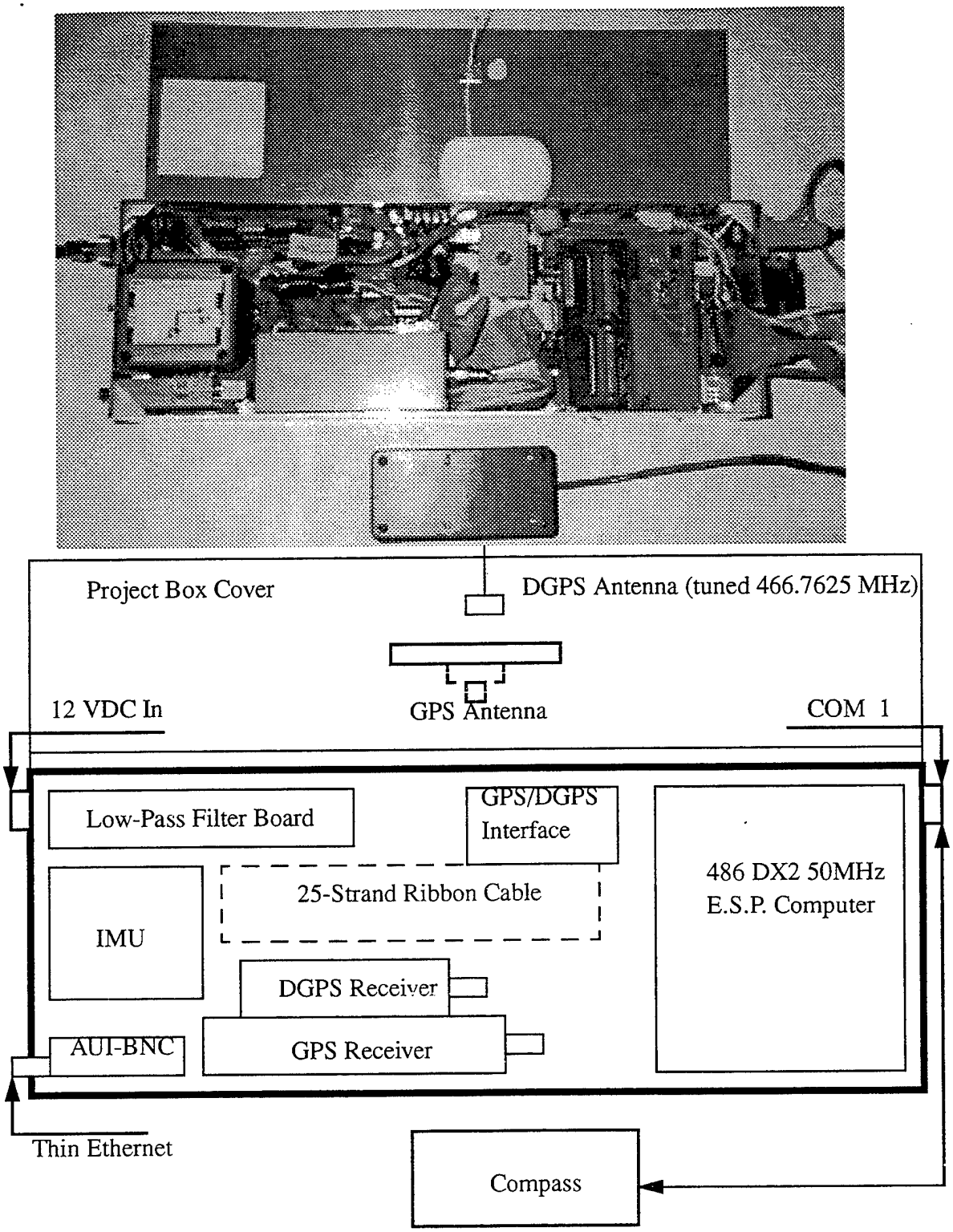

Figure 4: SANS Hardware Configuration (Walker 96)

B. HARDWARE DESCRIPTION

1. Computer

The on-board processor is an Extremely Small Package (E.S.P.) Cyrix 486SLC DX2 50 MHz computer, pictured in Figure 5. It is specifically designed to offer off-the-shelf PC-compatible solutions in space and/or power constrained environments. This particular E.S.P computer possesses a total of eight modules which perform various system tasks. Together, the processor and its accompanying modules provide a small, low-power system with system performance comparable to a standard, desk-top type system. (MAXUS 95, Walker 96)

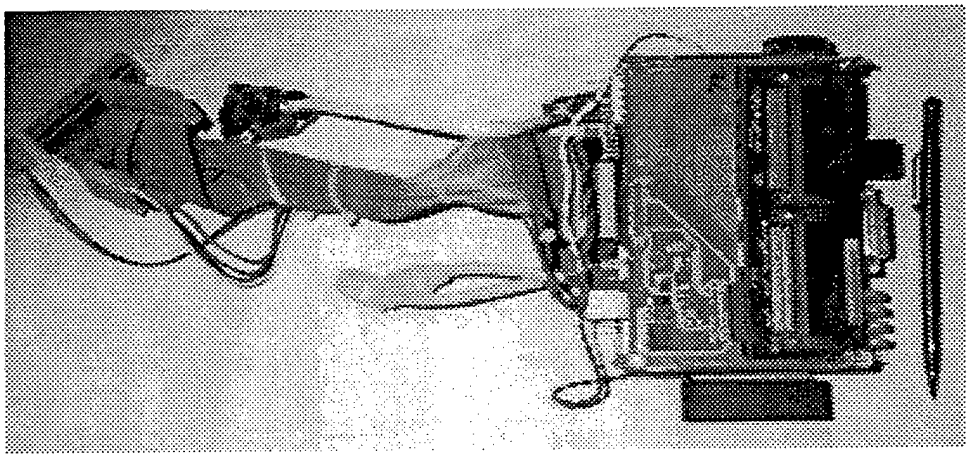

Figure 5: E.S.P. 486SLC DX2 50 MHz Computer (Walker 96)

The CPU Module provides the processing capability, the interface for a standard keyboard, the Flash PROM containing the system BIOS, and memory and bus controller logic. The DC-DC Power Module provides for all the system power requirements up to a maximum 35W total output. It accepts an unregulated 12 V DC and provides the required +5, +12, -12, and -28 V DC to power various system components and optional peripherals

(i.e., an external floppy and hard drive, as is used in the tilt-table test configuration). (Maxus 95, Walker 96)

The VGA Adapter Module provides the interface to operate an external VGA monitor. A PC I/O Module provides for two Serial ports and one parallel I/O port. It also provides two type-III PCMCIA sockets which conform to PCMCIA Release 2.01 standard. These two ports can be used for a variety of compatible devices (i.e., Ethernet Adapter, Modem, GPS Receiver, etc.). This module was included in the current design to provide additional secondary storage in the form of PCMCIA SRAM cards, as well as to enable possible future expansion. An Ethernet Module provides the SANS with an external ethernet interface. (Maxus 95, Walker 96)

The Analog to Digital (A/D) Module provides 8 differential or 16 single-ended input channels at 12-bit resolution. In its current configuration, the A/D module samples only 8 of the available 16 single-ended channels. It features a single-channel maximum sampling rate of 333 KHz, and an input range from +/- 1.25mV to +/-10V (MAXUS 95). The A/D module provides a 34-pin external connector (J3) to which developers can connect their input signals. (Walker 96)

The DRAM Module provides for high-speed (70ns) memory storage available in 2, 4, 6, 8,or 16MB capacities (MAXUS 95). This module is to the E.S.P. as a hard disk is to a standard desk-top PC. (Walker 96)

2. Inertial Measuring Unit

The inertial navigation component of the SANS is provided by a Systron-Donner Model MP-GCCCQAAB-100 "MotionPak" inertial sensing unit, pictured in Figure 6. This

self-contained unit provides analog measurements in three orthogonal axes of both linear acceleration and angular velocity. It consists of a cluster of three accelerometers and three "Gyrochip" angular rate sensors. (Walker 96)

Figure 6: Systron-Donner Inertial Measuring Unit (Bachmann 95)

3. GPS/DGPS Receiver Pair

The GPS/DGPS receiver used is the ONCORE 8-channel receiver which incorporates an imbedded DGPS capability (Oncore 95). The receiver is capable of tracking up to eight satellites simultaneously. It can provide position accuracy of better than 25 meters Spherical Error Probable (SEP) without Selective Availability (SA), and 100 meters (SEP) with SA. Typical Time-To-First-Fix is 18 seconds with a typical reacquisition time of 2.5 seconds (Oncore 95). This receiver meets or exceeds the capabilities of the receiver

described in Norton (94), which, under normal operating conditions, met the accuracy and time requirements of the SANS project. Norton (94) also demonstrated that a receiver with these qualities will perform well when using an antenna that is located on or near the sea surface, as is necessary during a clandestine mission. Figure 7 shows the ONCORE GPS/ DGPS receiver used in the SANS project. (Walker 96)

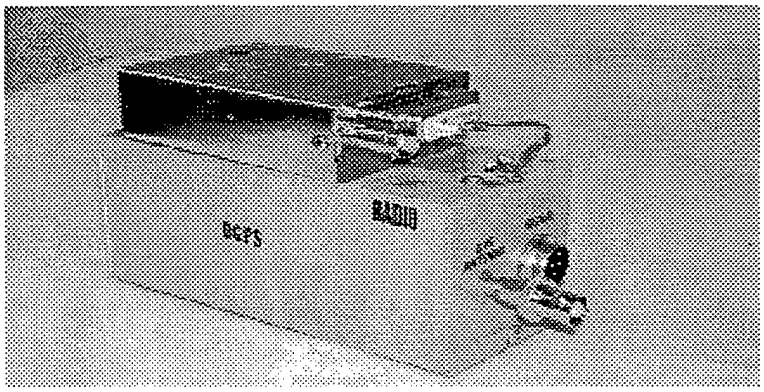

Figure 7: ONCORE GPS/DGPS Receiver (Walker 96)

4. Compass

The compass used in the SANS project is a Precision Navigation model TCM2 Electronic Compass Module. This compass does not employ the mechanical gimbal technology utilized in the compass described in Bachmann (95), but rather employs a three-axis magnetometer and a high-performance two-axis tilt sensor in a small form-factor (TCM2 95). The TCM2 compass is capable of providing readings of pitch, roll, and surrounding magnetic field strength in addition to heading. The TCM2 provides greater accuracy by calibrating (performed by the user) for distortion fields in all tilt orientations, providing an alarm when local magnetic anomalies are present, and giving out-of-range warnings when the unit is being tilted too far (TCM2 95). (Walker 96)

5. Other Components

The water speed sensor and the depth sensor are those described in Bachmann (95) and therefore are not depicted in Figure 5. The GPS antenna shown in Figure 5 is an active antenna, which was selected for its performance and low profile. Because the E.S.P. Ethernet module's output media type is AUI, a standard AUI-to-BNC media converter is employed to allow the use of durable RG-58 coax cable to span the roughly 100m distance required while pulling the towfish behind a towing vessel. The GPS/DGPS Interface box is nothing more than an adapter to interface the GPS receiver signal with the serial port of the E.S.P. computer. (Walker 96)

Based on the analysis given in Walker (96), the 2-pole anti-aliasing Bessel filters used in Bachmann (95) were replaced with new low-harmonic distortion filters. These come factory tuned to a user-specified corner frequency of 10 Hz, require no external components or adjustments, and operate with a dynamic input voltage range from non-critical +/- 5V to +/-18V power supplies (Frequency Devices 96). To implement these filters into the SANS, a double-sided printed circuit board was designed and machined to receive all six filter DIPs, as well as three quad op-amp LM324 DIPs configured as voltage-followers to provide input and output circuit protection. (Walker 96)

To provide for the requisite +/-15 VDC, a DATEL model BWR-15/330-D12 DC-DC Converter is used to convert the unregulated 12 VDC battery input into regulated +/-15 VDC needed to power the low-pass filter circuits and the IMU. This converter features over-current and short-circuit protection, a compact form-factor, and high reliability at a

minimum efficiency of 82%. It employs switching regulator technology, which minimizes heat generation and current usage. (DATEL 95, Walker 96)

Physically connecting the IMU, Low-pass Filters/DC-DC Converter PCB, the Analog-Digital Converter, input power, water speed sensor, and depth sensor, is a 25-strand flat ribbon cable. This type of cable was chosen to allow all system components to be easily interconnected. (Walker 96)

C. SUMMARY

The SANS design described in this chapter is significantly different from that described in Bachmann (95). The processing capability, along with the GPS/DGPS receiver, is now on-board the SANS, making it completely self-contained. The only external link is a DOS ethernet environment to a remote PC utilized for test monitoring purposes. The IMU sensor data, after low-pass filtering, along with water speed and depth data, are converted from analog to digital form, with 12-bit resolution, and then passed to the processor. GPS data is passed separately to the processor, which computes updated attitude and position information to be exported over an ethernet socket. The hardware for this version of the SANS was chosen to comply as far as possible with the requirements set forth in Kwak (93). Though there are many possible choices of hardware for each of the components in Figure 4, trade-offs between accuracy, size, power requirements, and cost have been considered. As further advances in miniaturization are made, accuracy will continue to increase while price and size decrease, thus making it easier to meet the challenges of the SANS baseline requirements. The next chapter of this thesis will describe the software which supports this hardware configuration.

IV. SOFTWARE DEVELOPMENT

A. INTRODUCTION

The purpose of the SANS software is to control some of the individual hardware components, to control input/output interface communications between the components, to assimilate all of the incoming data, and to implement a twelve state navigation filter. This chapter will review the software structure inherited from Bachmann (95) and Walker (96), and will concentrate particularly on the changes made to accommodate the greatly improved processing speed that Walker (96) made possible.

The code is written in C++ and is designed for use on an IBM-compatible personal computer using the Borland version 3.1 compiler under DOS 5.0. This project code choice has proven to complicate the integration of the hardware interfaces. Additionally, the dated software compiler formats and DOS system calls make the code specific to this application only and increases the difficulty of troubleshooting or implementing changes. Although most of the code is transportable to other C++ compiler environments, the interrupt processing and input/output communications control uses obsolete type declarations and function calls to the rapidly aging operating system.

This limitation could easily be resolved in future project work in either of two ways. Utilizing a Borland version 5.0 compiler with updated communications code would allow continued use of a traditional IBM-compatible environment. Converting those sections of code to be compatible with Unix environment compilers could also be implemented on the PC under the Linux operating environment.

The software design instantiates objects corresponding either to the individual hardware components or to the purpose accomplished, in a straightforward manner. The class and object relationships are shown in Figure 8. All of the concrete classes depicted are specifically instantiated by the class instance above them, in descending chronological order as the program is initiated. All are instantiated as a single object, named as shown. There is no need in this application for extensive polymorphism. The serialPortClass and bufferClass classes are abstract parent classes containing the common definitions and functions from which the specific compassPort, compassBuffer, gpsPort, and gpsBuffer classes inherit. The stampedSample object, defined in the main program's header file, contains the latest update of all pertinent navigation information. Therefore, it is the object which is passed between the class objects. Other objects which support the calculations are structures to hold such things as position in the various formats. For simplicity, they are not shown in Figure 8.

This architecture represents a substantial change from the original design, while retaining most of the functionality. As the project evolved, it was determined that much of the flexibility originally envisioned did not prove to be necessary. This includes features such as the capability to instantiate an array of serial ports, or a need for a wide variety of buffers for the data received through the serial ports.

The above features were included in the original object oriented design approach, but have been streamlined to a more specific, less complicated structure. Specifically, the portbank and bytebuffer classes have been removed. Only two serial ports are required, for the compass and gps interfaces, respectively. The serial port code was modified, and the

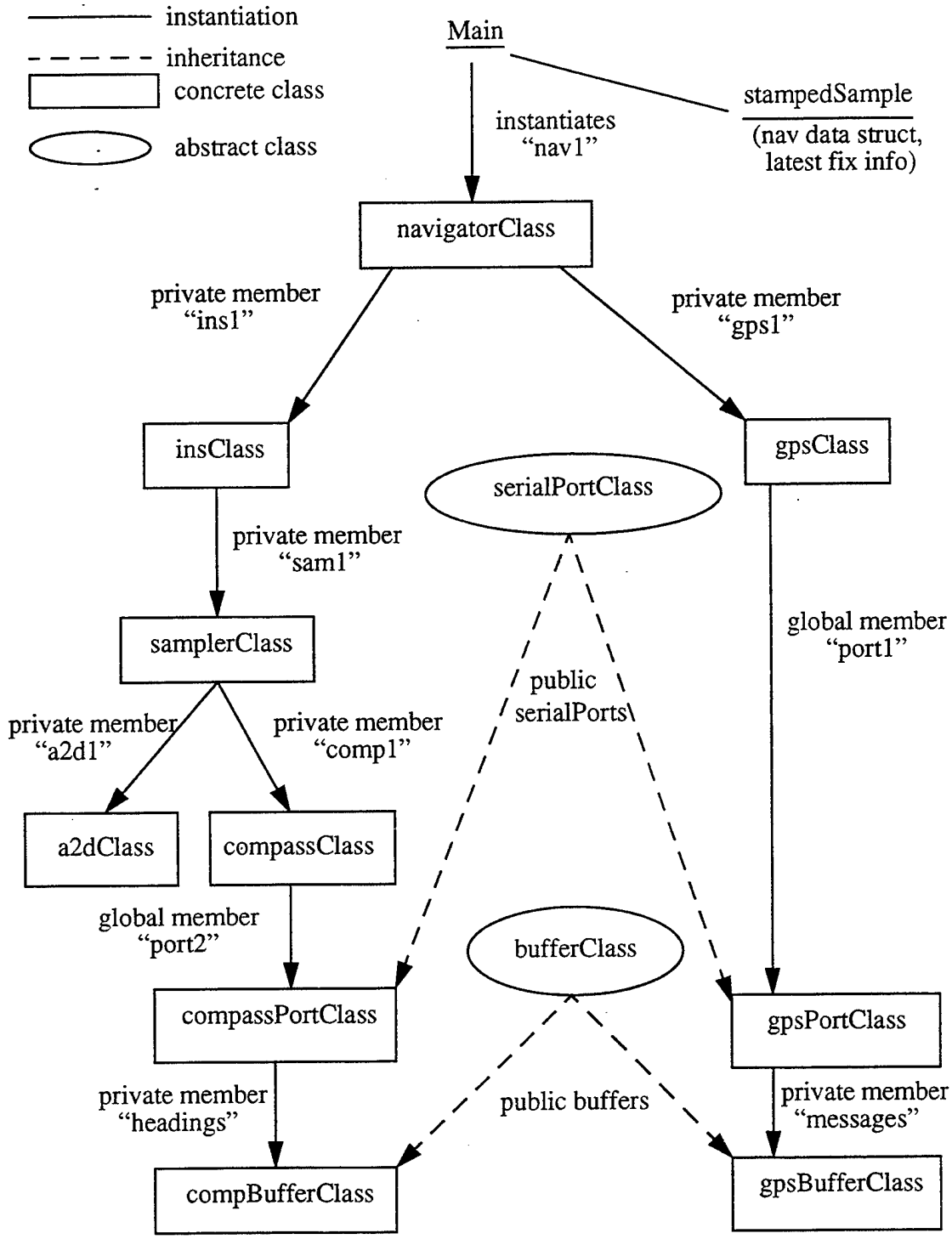

Figure 8: SANS Code Classes and Objects

buffered serial port class has been specialized to a compassPort class and a gpsPort class, while retaining the same basic function. This resulted in the compassPort and gpsPort classes representing a kind of serial port, similar to the way the compBuffer and gpsBuffer classes already were a kind of bytebuffer and continue to be a kind of buffer. This simplified the class membership hierarchy and variable passing across class lines. The specific nature of the application made efficiency a higher priority than general applicability.

Other improvements included the addition of configuration files containing such data as gain settings to allow repeated testing without the necessisity of recompiling after every change. The increased processing speed overwhelmed the DOS operating system's ability to print information to the user's screen in real time, so an interval was added that reduced screen output to a more usable human rate that also reduced input/output conflicts. All screen output and data writing to files were consolidated to single points to further simplify exchanges. And finally, some error checking was added to ensure such things as proper A/D converter channel initialization.

B. SOFTWARE FILTER

The purpose of the software filter is to utilize IMU, heading, and water-speed information to implement an INS, and then to integrate this with GPS information. This results in a single system which can produce continuously accurate navigational information in real time. The filter mitigates the effects of sensor inaccuracies (inherent, electronic noise, and transitory), ocean current (the largest single factor affecting AUV navigation), dynamic model uncertainty, measurement errors, and calculation errors.

Kalman filtering techniques are used to implement the INS using DGPS fixes as "error-free data". This allows periodic reinitialization of the INS to correct accumulated drift and development of error biases. All sensor data is logged in raw form for post-mission processing. (Bachmann 96)

Figure 9 is a data flow diagram for the SANS software filter. On this diagram, R represents a rotation matrix and T is a body rate to Euler rate transformation matrix. Table 2 gives the state variables for the navigation filter. The twelve state variables include the outputs of the three integrator blocks, the estimated current in north and east direction components, and the bias estimates for the angular rate readings. (Bachmann 96)

Euler Angles	Φ ,θ ,ψ
North & East Velocity	$\dot{x}_e$,$\dot{y}_e$
North & East Position	x_e ,y_e
Apparent Current	$\dot{x}_c$,$\dot{y}_c$
Angular Rate Bias Estimates	p_b ,q_b ,r_b

TABLE 2: State Variables of the Kalman Filter (Bachmann 96)

Ten of the state components are "continuous time": the three Euler angles (Φ, θ, ψ), two horizontal velocities ($\dot{x}_e, \dot{y}_e$), two horizontal positions (x_e , y_e), and three angular rate bias estimates. Continuous time integration is approximated by numerical integration, making these "continuous time" components discrete time values in the reality of the digital filter. This is necessary due to the minimum integration sampling time limitation of the computer and A/D hardware. The apparent ocean current values ($\dot{x}_c, \dot{y}_c$) are updated aperiodically as a result of both diving and wave action, which produce inherently discrete

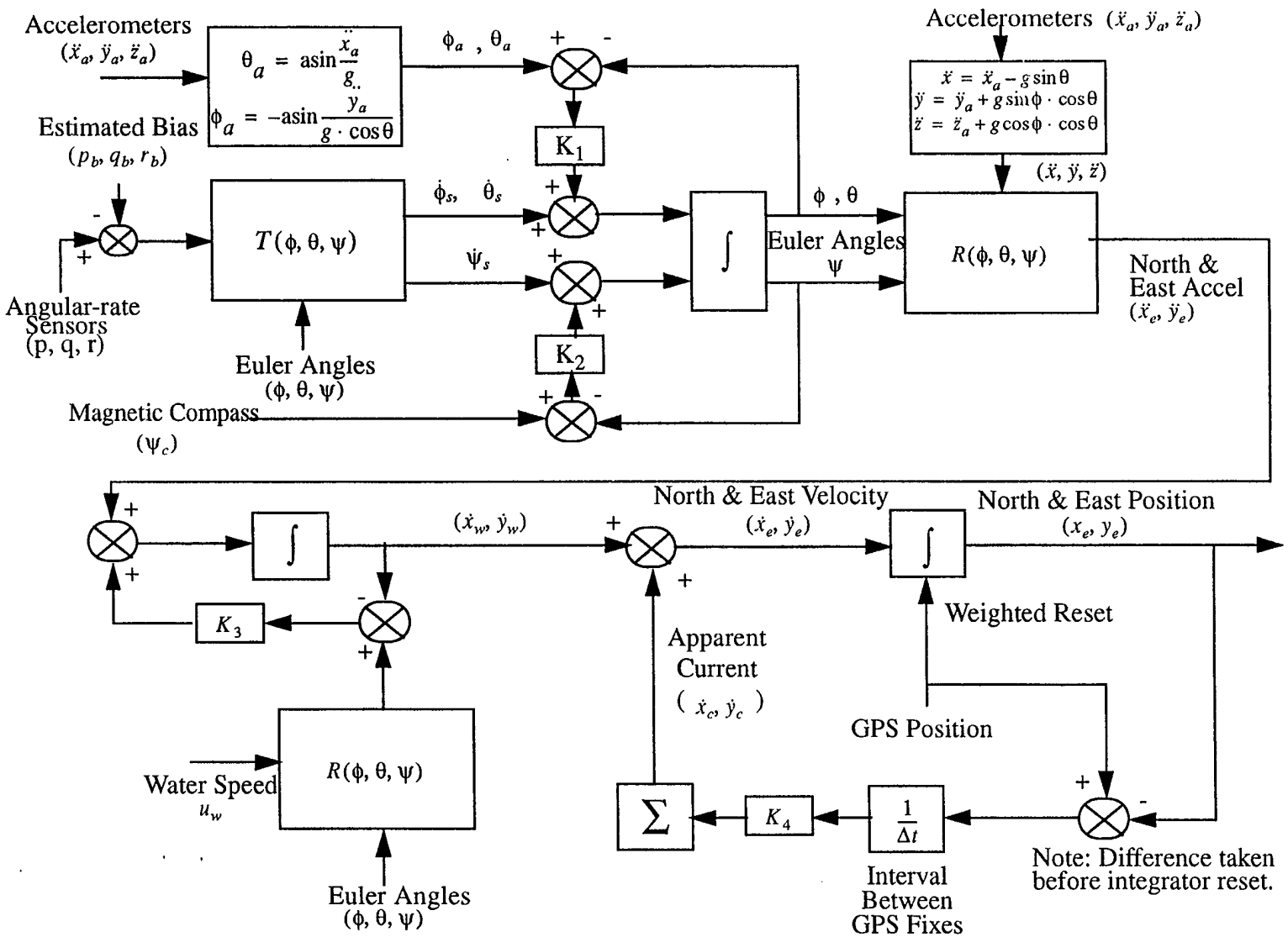

Figure 9: SANS Software Navigation Filter (Bachmann 96)

gps fix information. This discrete event dynamic system is well suited to application of Kalman filter theory to obtain optimal time-varying values for the gain matrices K_i in Figure 9. However, at the time of writing this thesis, there are inadequate statistics on DGPS noise and AUV maneuvering as needed by this approach. Therefore, bandwidth and steady-state error considerations were used to compute initial constant gains (Bachmann 95, McGhee 95), which were subsequently adjusted based on the results of experimental studies. (Bachmann 96)

One area for future project work involves obtaining the necessary statistical data needed for refinement of the aperiodic, gps update portion of the filter. The optimum reset weight for application to the final integrator block could then be determined. Additionally, application of the gps fix interval ($1/\Delta t$) just prior to K_4 is under consideration for removal.

The principal difference between the current filter and that described in Bachmann (95) regards the point in the filter process at which the apparent current error correction is made. The previous filter added the apparent current to the water speed. The difference between this value and the estimated north and east velocities was input to the north and east accelerations with a gain K_3. Poor initial sea test results in Bachmann (95) indicated this approach was possibly underdamped or even unstable. The present approach is to apply the apparent current as feedback to the output of the third integrator block, prior to input to the final, position integrator. (Bachmann 96)

The continuous state portion of Figure 9 shows that the Euler angle and linear velocity outputs are fed back to the corresponding integrator inputs. Thus, with diagonal gain matrices K_1, K_2, and K_3, each of these integrators is in fact a low pass filter for its respective inputs (Bachmann 96). Figure 10 isolates one feedback loop to help illustrate

this relationship. The integrator block is shown using s-domain (Laplace transform) notation. This approach prevents unlimited state estimate growth caused by unmodeled bias errors in state derivative inputs to the integrators (Bachmann 96). Complementary, low frequency information from an independent source (accelerometers) is also furnished to each integrator to correct for long-term decay of the state estimates resulting from this feedback (McGhee 95). The low frequency information sources include attitude estimates from accelerations sensed by the accelerometers ($\ddot{x}_a$, $\ddot{y}_a$, $\ddot{z}_a$), the magnetic compass readings (Ψ_c), and water speed (u_w). (Bachmann 96)

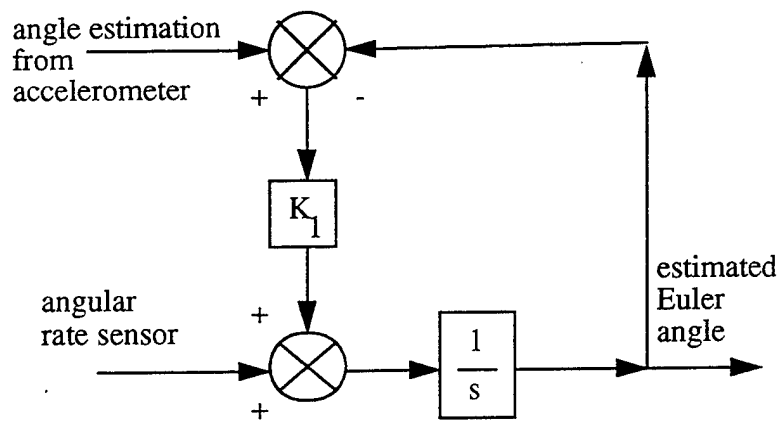

Figure 10: Complementary Filter Feedback Loop for Euler Angle Estimation

The IMU acceleration readings require correction in addition to filtering. The accelerometer data is utilized as an inclinometer, to determine how much of the specific force felt along each axis is due to gravity. Computed gravity is then subtracted from specific force readings of the accelerometers ($\ddot{x}$, $\ddot{y}$, $\ddot{z}$), to transform them into accelerations, prior to rotation into earth-fixed coordinate values ($\ddot{x}_e$, $\ddot{y}_e$, $\ddot{z}_e$). (Bachmann 96)

The rate sensor input in Figure 10 is added to accelerometer attitude estimates after the gain matrix is applied. This signal already has the estimated bias removed utilizing the low pass filter methodology derived in Chapter II and resulting in Equation 2.7. New biases are calculated on each filter cycle by the calculateBiasCorrections function of the insClass, and are applied to new navigational state information in the applyBiasCorrections function. Filter response to example and real world inputs will be discussed in detail in Chapter VI, System Testing.

C. IMPLEMENTATION DESCRIPTION

Figure 11 shows the revised data flow between software objects. The tasks performed by the SANS software can be divided into two basic categories. The primary tasks are related to calculating the current position and other navigational state information. This includes processing incoming GPS data, IMU data, water-speed, and heading information, and integrating all of this information through the navigational filter to obtain a fix. These tasks are performed by the gpsClass, insClass, and Navigator software objects respectively. The secondary tasks involve hardware interfacing, communications, data filtering and unit conversion. These basic but crucial tasks are handled by the Sampler, Buffer, compBuffer, gpsBuffer, A2D, Serial Port, compassPort, and gpsPort software objects. The *main program* serves to drive the other objects by continually querying the navigator for position updates and performs output to the user screen and data files from a single location. Real time navigation source code is provided in Appendix A. Supporting serial communication and other administrative function code is provided in Appendix B. The following summary

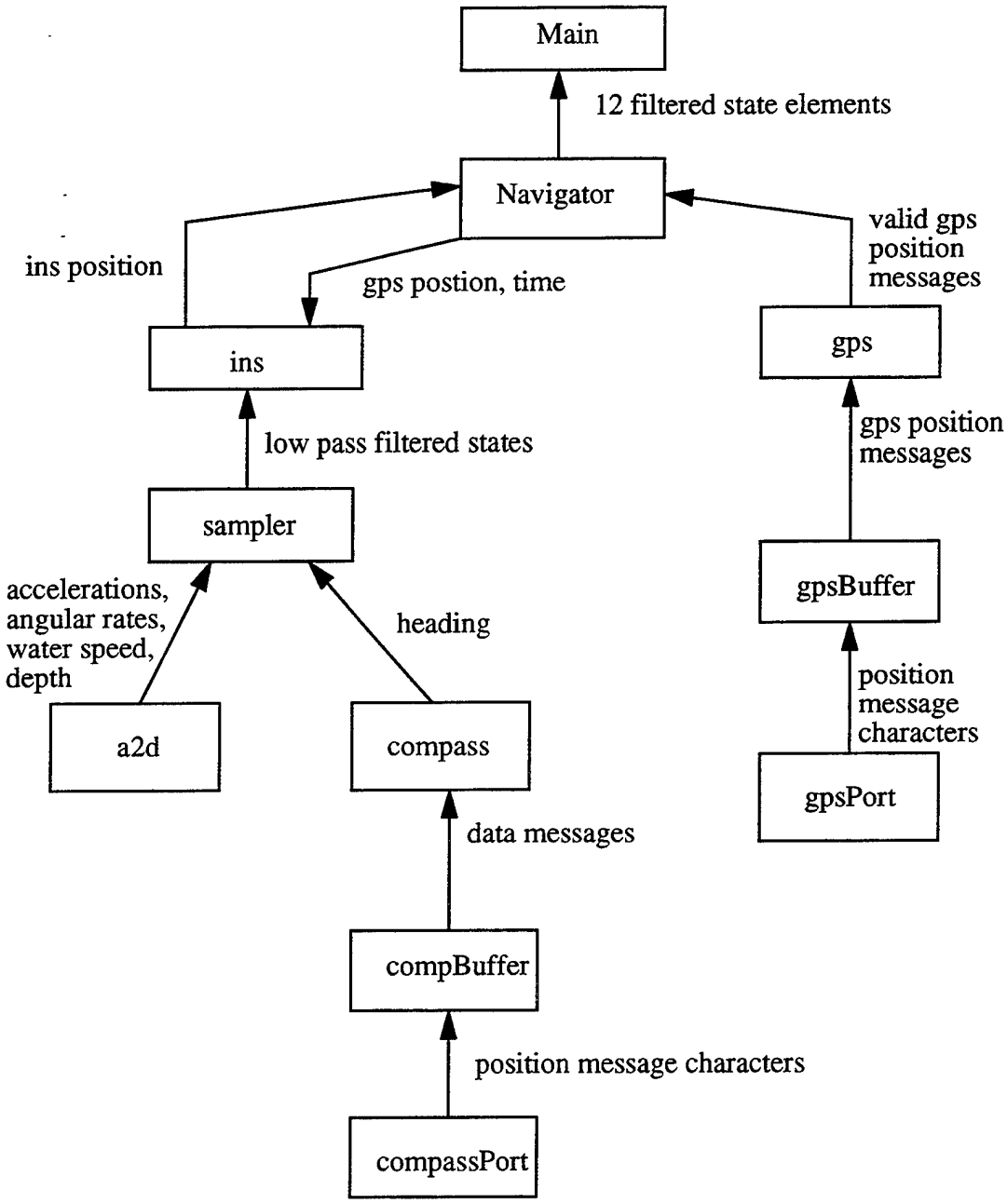

Figure 11: SANS Data Flow Between Software Objects

of the source code is presented bottom-up to illustrate construction of the navigation state from the individual data elements. (Bachmann 95)

1. Compass Data

The compassClass contains the code and member objects which implement reception of compass messages in a design similar to, with the exception of specific hardware details, the gpsClass. Private member compassPort instantiates a kind of serialPortClass object to allow data communication on COM2. CompassPort in turn has private member compBufferClass which provides a kind of bufferClass structure for temporary storage of incoming compass messages. Figure 12 illustrates the compBufferClass and gpsBufferClass data structures. The compassClass therefore contains code to communicate with the serial port, as well as to check the "checksum" and header of each compass message received. The samplerClass object instantiates compassClass object "comp1" and periodically interrogates comp1 to empty the buffer of information. (Bachmann 95, Walker 96)

2. GPS Data

The gpsClass, as previously mentioned, is similar in design to the compassClass, with differences driven by the different message formats, and it utilizes COM1. It obtains GPS position messages in the Motorola proprietary format (@@Ea). Before the code recognizes a GPS message as being valid, the message must pass three conditions; 1) it must have a valid checksum, 2) the fix must be based on at least 4 satellites, and 3) the differential bit in the message must be set (i.e., the fix must have the differential correction

applied to it). The navigatorClass instantiates gpsClass object "gps1" and interrogates gps1 to empty its buffer directly. (Bachmann 95, Walker 96)

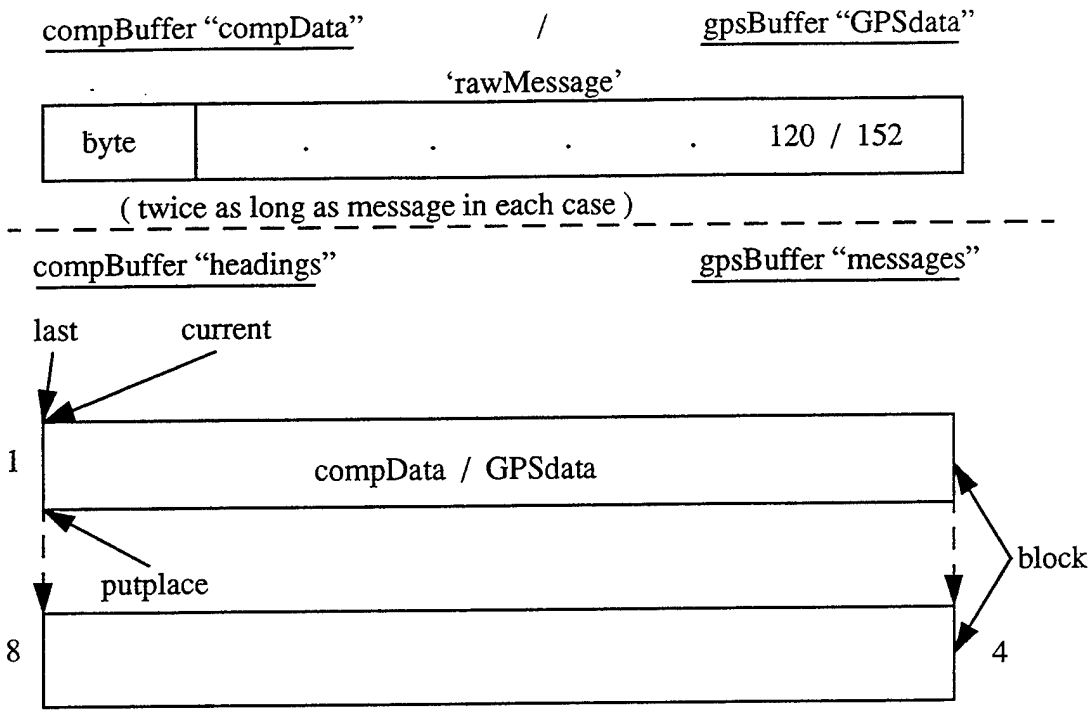

Figure 12: Buffer Data Structures

3. Inertial Sensor Data

Inertial sensor data passes through the new filter circuit board. From there, it is input directly to the A/D converter module in the processor.

a. A/D

The A/D module came with demonstration C source code provided by the unit manufacturer. Walker (96) modified the demo code and converted it to C++ for the SANS application. The a2dClass provides all of the requisite software operation for the A/D module in the E.S.P. computer, which is completely controlled through software. Control is maintained through the manipulation of the A/D Control Register and the A/D Status

Register. These registers are manipulated by writing to and reading from specific memory addresses. The a2dClass is designed with some degree of user flexibility. For instance, the user can choose between one of two base addresses. (Walker 96)

The SANS software only uses a few of the member functions in accomplishing its mission. Those member functions not directly utilized in this particular application are useful for troubleshooting, or allow a variety of options for specific applications. The following general discussion explains how the A/D module works in the SANS application. (Walker 96)

The A/D provides 12 bits of resolution, or $2^{12} = 4096$ discrete quantization levels. The A/D module may be employed in differential mode or single-ended mode. The SANS application employs the A2D in the single-ended mode of operation. The A2D samples the dual-ended swing of the IMU sensor signals, and represents these voltages as a digital value in the range 0 - 4095. A general A/D conversion table is provided as Table 3 to further illustrate how the sensor voltages are mapped over to their digital equivalents. (Walker 96)

Sensor DC Voltage	Converted Equivalent
+10 Volts	4095
+5 Volts	3071
0 Volts	2047
-5 Volts	1023
-10 Volts	0

TABLE 3: A2D DC-to-Digital Conversion Mapping (Walker 96)

When an a2dClass object is instantiated, the class constructor sets several default data member values, and then reads the A/D configuration file A2D.cfg. This configuration file provides simple user update of A/D module operation without recompiling the source code. The constructor initializes the system addresses, then initializes the A/D hardware using those variables that were loaded upon reading the configuration file. The a2dClass object is instantiated by the samplerClass object as "a2d1". It is a private data member of the samplerClass. (Walker 96)

The A/D module is set into operation by a call to the samplerClass function initSampler(). It utilizes a2dClass member functions to program the sequencer and tell it which channels to sample and in what order, resets the A/D First-In-First-Out (FIFO) to enable it to receive data, and then toggles the trigger bit in the A/D Control Register from a low to a high, which starts the A/D into operation. (Walker 96)

4. Sampler

The samplerClass object prepares raw IMU, heading, and water speed data for use by the INS. This preparation includes simple filtering, unit conversion, and time stamping. Figure 13 provides a summary of the principal class members and functions, with psuedocode descriptions of the principal methods. The samplerClass interface consists of a single method (getSample) which controls the data formatting and returns a formatted sample if valid raw data is available, and a negative response otherwise. (Bachmann 95)

Figure 14 provides an illustration of the process of obtaining samples from the A/D. During SANS operation, the samplerClass member function readSamples() is called repeatedly to retrieve inertial data from the A/D FIFO. It first checks to ensure that the

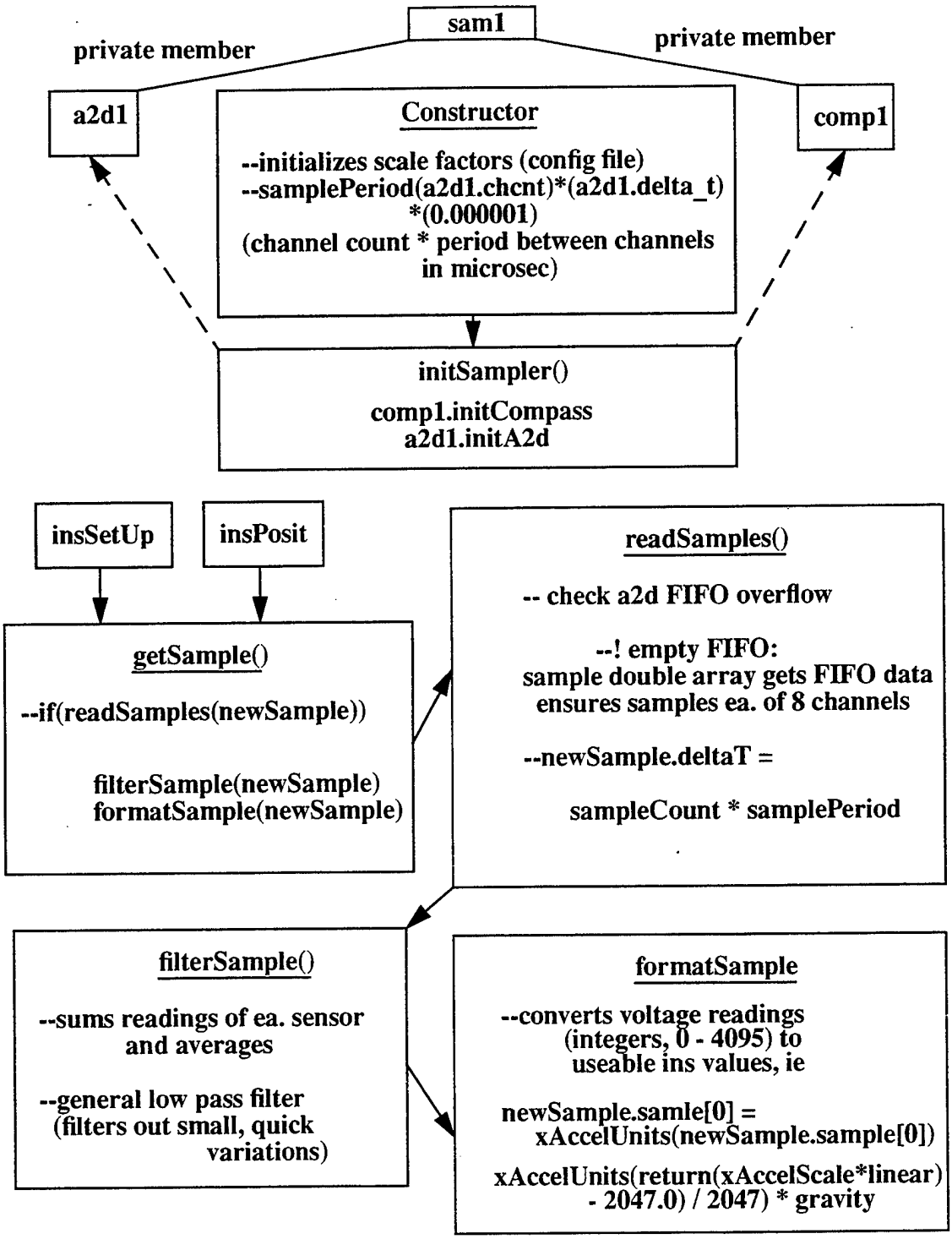

Figure 13: samplerClass Summary

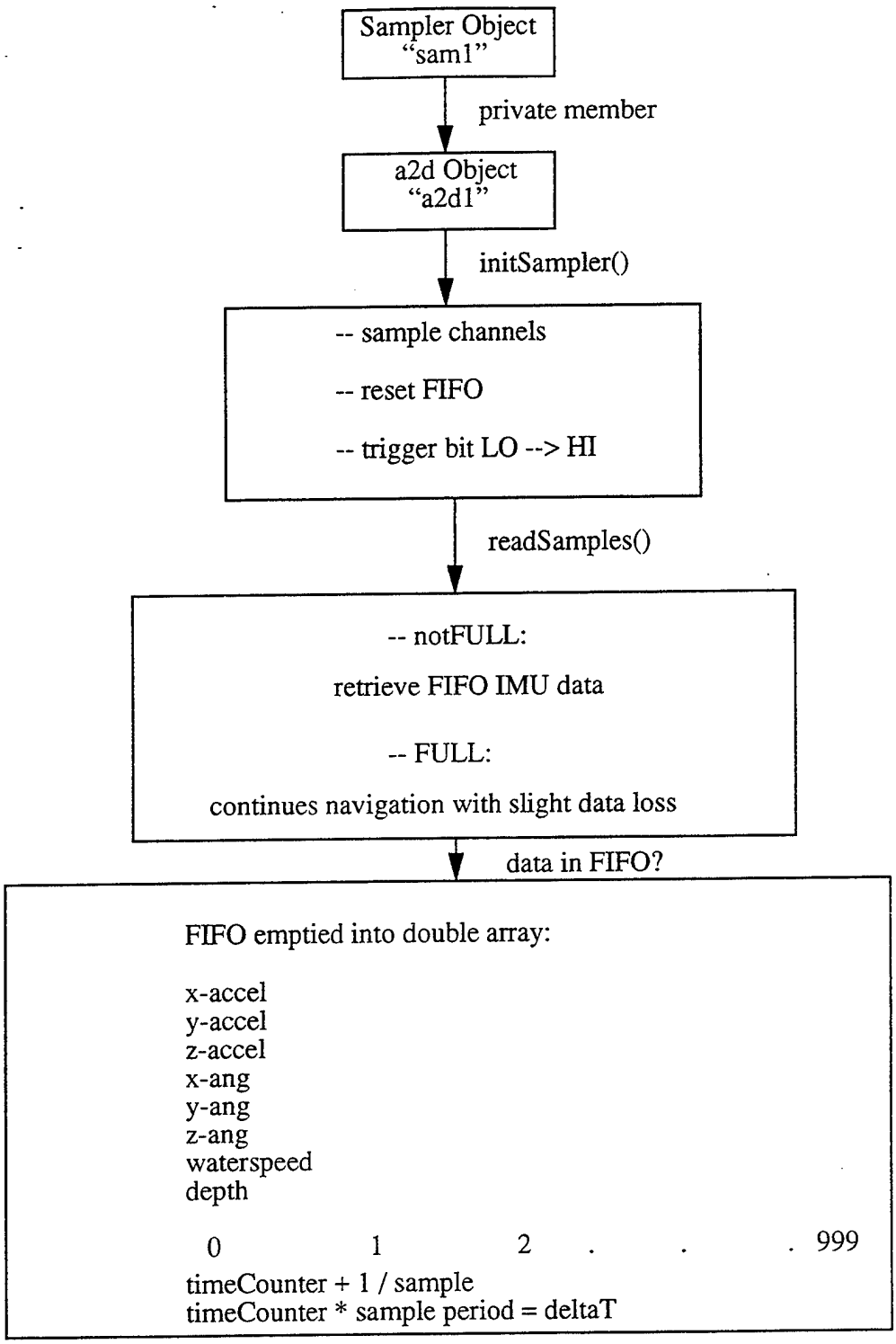

Figure 14: samplerClass Data Flow

FIFO is not FULL. If the FIFO ever gets filled without being immediately emptied, data will continue to push into the FIFO. There is no room for this additional data and all information from that point on will be lost. Preventing the FIFO from overflowing is critical for proper SANS operation. If this check is ever true, the SANS software has been rewritten to reinitialize the a2d and continue to navigate. One full FIFO plus the data received in the time since the overflow will be discarded. This will result in a very short period of lost data with a minimal impact on navigation accuracy.

To prevent FIFO overflow, one need only be mindful of the rate at which the A/D is sampling its inputs and be sure the A/D FIFO is emptied at the same rate or faster. If the FIFO does have data in it, this data is emptied from the FIFO and stored in a doubly-subscripted array with 8 rows and 1000 columns to coincide with storing up to 1000, 8 channel samples of sensor data. This type of data structure is used to temporarily store the data to enable access to a history of samples. Figure 15 presents a model of this array. (Walker 96)

x-acc	x-acc	. . .
y-acc	y-acc	. . .
z-acc	z-acc	. . .
x-ang	x-ang	. . .
y-ang	y-ang	. . .
z-ang	z-ang	. . .
waterspeed	waterspeed	. . .
depth	depth	. . .

0 1 999

Sample Number/Array Index

Figure 15: Model of the A2D Sample Array (Walker 96)

The first action taken by the Sampler when a packet is received is to time stamp it. Since the time difference between the eight samples contained in a single message packet is relatively small, the Sampler object then respectively averages the eight corresponding data variables contained in a packet. As the samples are emptied from the FIFO, the variable "timeCounter" is incremented once for every 8 samples. This variable is then multiplied by the sample period to calculate the "deltaT", or the time between adjacent samples. The samplerClass code then averages over all the samples received since the last sample was taken from this array. The averaged measurements which result represent a simple low-pass filtering of the samples. This has the effect of filtering out small fluctuations in the data. (Bachmann 95, Walker 96)

The integers contained in a sample are digital measurements of analog voltages output by the SANS sensors. Once these eight filtered measurements are obtained they are converted from voltages to units which are usable by the INS object (i.e., feet and radians). Finally, each of the measurements is checked to ensure that it is within the limits of the sensor from which it came. If any values fall outside the capabilities of the sensor from which it came, the entire packet is considered invalid and discarded. (Bachmann 95)

5. INS

The INS class implements the SANS inertial navigation. It is the most complex class in the software. It has been changed very little as the project has evolved. Figure 16 provides a summary of the principal member objects and functions which constitute the INS methods. The interface consists of three public methods. Each is directly involved in the implementation of the twelve-state Kalman filter. The primary method (insPosition)

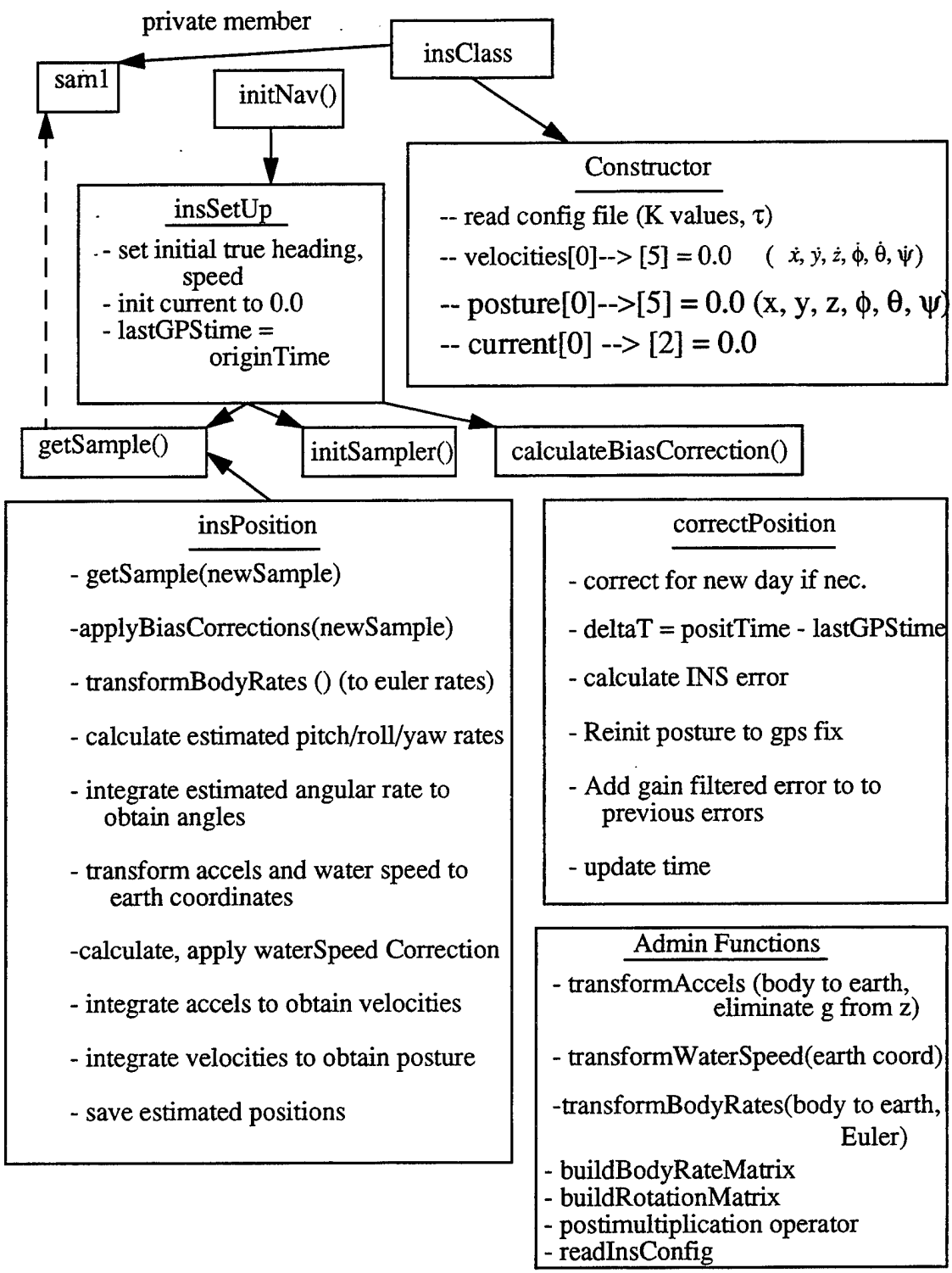

Figure 16: insClass Summary

combines all sensor information and uses the Kalman filter to produce a dead reckoning position estimate. The other methods support the primary method by performing one-time or periodic operations. Initialization of the INS is performed by the insSetUp method, which sets the INS posture at the grid coordinate origin, sets an initial heading and speed, and marks the beginning of the first integration intervals. The last public method of the class (correctPosition) inputs GPS information to reinitialize the INS position while determining a current and error correction bias. The INS class instantiates a samplerClass object "sam1", from which it obtains all sensor data except for GPS position fixes. (Bachmann 95)

6. Navigator

The navigatorClass acts as coordinator of all navigational information. As such it determines which source is currently providing the best information, converts various position formats from one format to another, and instantiates the GPS and INS objects "gps1" and "ins1". Like the insClass, this portion of the code has been changed very little as the project has evolved. Figure 17 provides a summary of the class members and functions that provide the principal navigation methods. The interface to the object is made up of two public methods. (Bachmann 95)

The main program instantiates navigatorClass object "nav1". The first method of the navigatorClass (initializeNavigator), initializes nav1, preparing it to begin providing the current position upon request. This method obtains an initial GPS fix for use as the origin of the grid used by the INS object to specify positions, and calls the initialization method of the INS. (Bachmann 95)

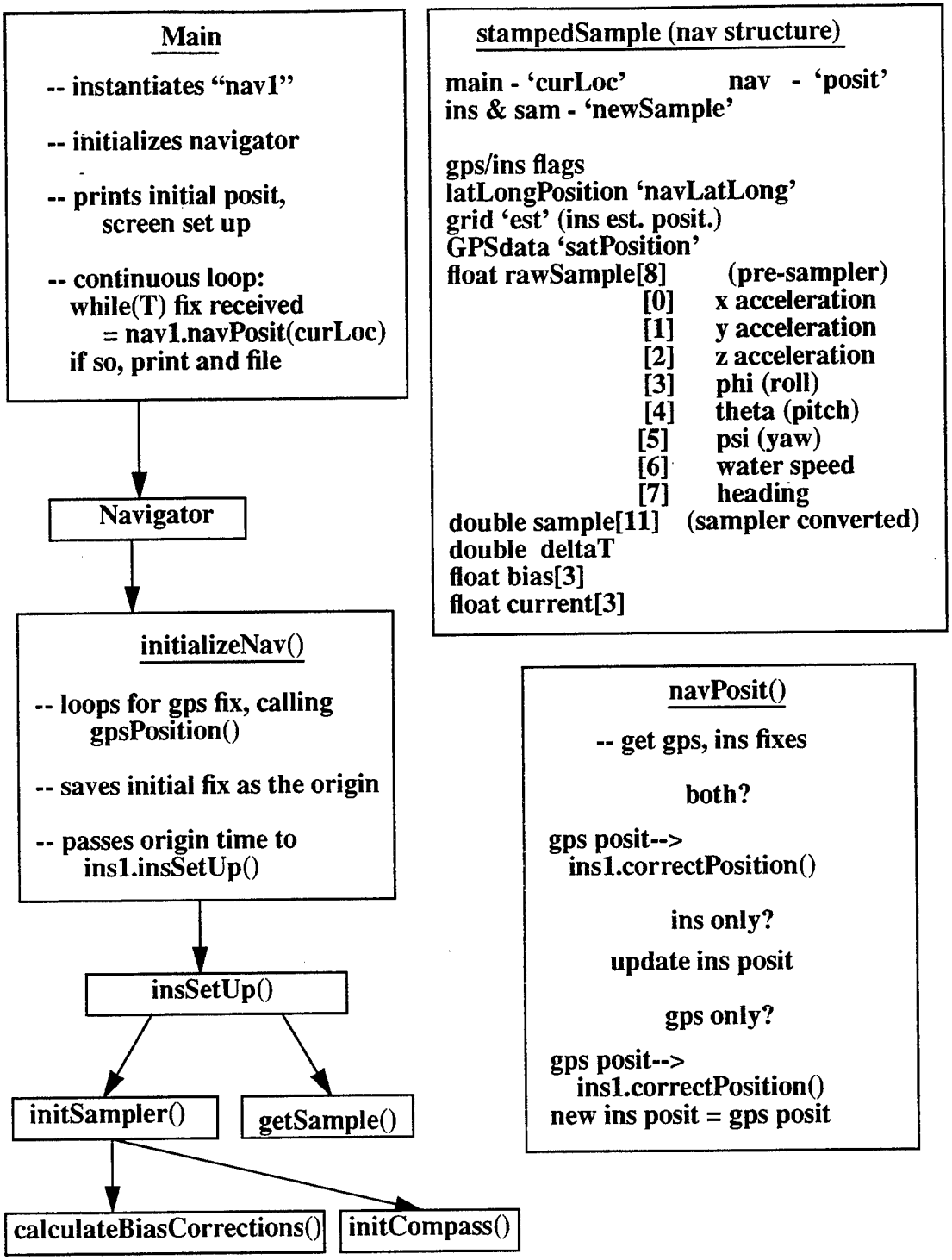

Figure 17: Navigation Class and Initialization Summary

The second navigator method (navPosit) drives both the GPS and INS objects, and provides the navigator's best estimate of current position in hours, minutes, seconds and milliseconds of latitude and longitude. Each time the method is invoked, it interfaces with the GPS and INS objects to determine if none, one, or both have an updated estimate of the current position. If no update is available, the navigator returns a negative reply indicating that it can not provide a position update. If only INS information is available, it is returned as the current estimated position. Whenever GPS information is available, it overrides the INS estimate of position. GPS information is also passed to the INS object as a reference for reinitialization and error estimation purposes. (Bachmann 95)

The navigator deals with three different position formats. GPS positions from the Motorola receiver are initially obtained entirely as latitude/longitude in milliseconds. INS positions are expressed in x-y grid coordinates based upon a navigator-stored origin. GPS positions must be converted to grid coordinates prior to utilization by the INS. The positions produced by the navigator are expressed in hours, minutes and seconds of latitude and longitude. A total of four methods are used to convert from one format to another. Figure 18 illustrates uses and conversions of the different position formats. (Bachmann 95)

7. Communication Objects

The bufferClass and serialPortClass objects are abstract parent classes from which specific instances are instantiated for the compassClass and gpsClass, respectively. As such, they contain the common class members and functions to support the routine but essential tasks of serial port communication and buffering received data.

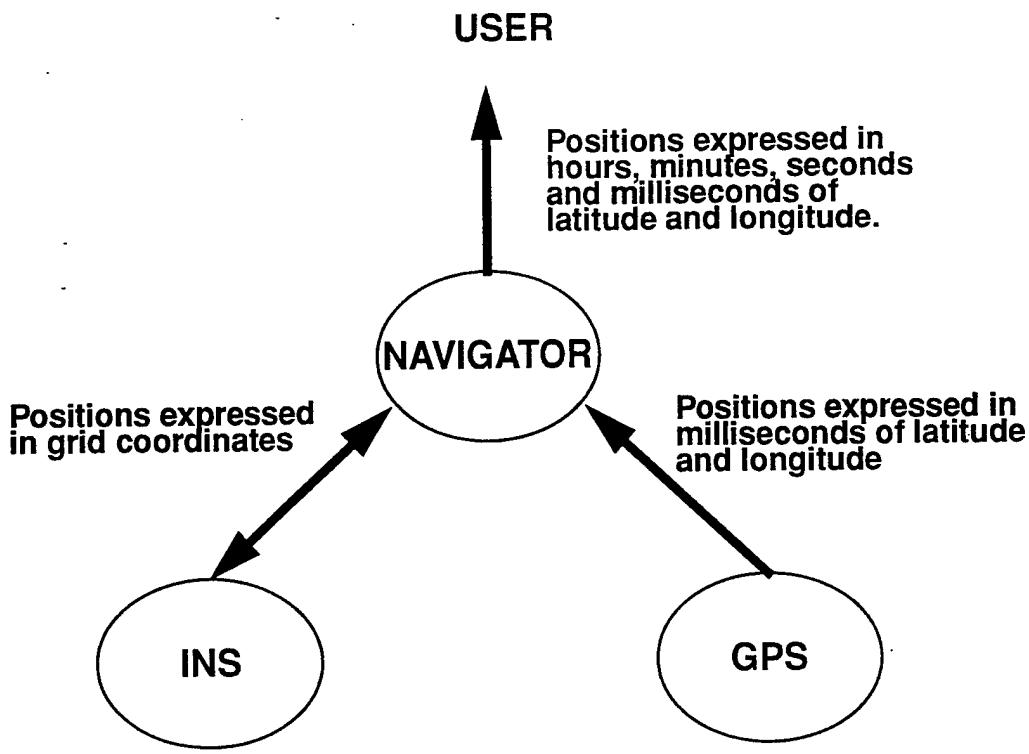

Figure 18: Navigation Position Format Utilization (Bachmann 95)

D. SUMMARY

The SANS software is designed to produce continuously accurate navigational information in real time. While submerged, IMU, heading and water-speed information are processed by the SANS Inertial Navigation System (INS) to produce a dead reckoning position estimation. This is integrated with DGPS information obtained during aperiodic surfacings using Kalman filtering techniques. The DGPS information is used to reset the position of the INS. It is also used to generate an apparent current vector to correct future INS position estimates. (Bachmann 95)

The software was implemented using object oriented paradigms. It was written in Borland version 3.1, C^{++} for use on an IBM-compatible processor. The primary tasks of the software are estimation of current position and communication. The former is handled by the Navigator, Sampler, a2d, Compass, INS, and GPS classes. The later is accomplished by the bufferClass, compBufferClass, gpsBufferClass, serialPortClass, compassPortClass, and gpsPortClass objects. (Bachmann 95)

The next chapter of this thesis will present the testing methodology and results for the tilt-table tests of the operational code.

V. SYSTEM TESTING

A. INTRODUCTION

This chapter presents both the testing methodology and the experimental results of the tilt-table testing used to determine the functionality and accuracy of the SANS attitude estimation. These tests focus on the operational C++ code, on determination of optimal gain settings for the attitude portion of the navigation filter, and on evaluation of the hardware accuracy and noise characteristics in a controlled environment. Factors which control attitude response include the K_1 gain value, the bias weight (biasWght), sample weight (sampleWght), and the x and y axis accelerometer scale factors.

As a reminder from Chapter II, the rate sensor input in Figure 11 has the estimated bias removed utilizing the low pass filter methodology resulting in Equation 2.7. Further background on low pass filter bias response is provided below in order to show the reasoning behind the testing methodology and to help explain the results.

B. LOW PASS FILTER BIAS RESPONSE

Applying Mason's formula to the signal-flow graph of Figure 2 from Chapter 2, in the s (Laplace transform) domain gives the transfer function of a low pass filter as

(Eq 5.1)

$$G(s) = \frac{\frac{1}{\tau s}}{1+\tau s} = \frac{1}{1+\tau s} = \frac{U(s)_{actual}}{U(s)_{commanded}} = \frac{L\{output\}}{L\{input\}} = \frac{Y(s)}{U(s)}$$

A typical tilt-table test of the attitude and angular rate sensors involves a step input of a constant roll rate to a commanded roll angle, for example, 10 degrees per second to an

angle of 45 degrees, resulting in a 4.5 second input. The filter bias estimation response can be determined from x(t) = u(t), leading, in the s domain, to X(s) = U(s) = 1 / s, and

$$Y(s) = G(s)X(s) = \frac{\frac{1}{s}}{1+\tau s} = \frac{1}{s(1+\tau s)} = \frac{1}{s\left(s+\frac{1}{\tau}\right)} \qquad \text{(Eq 5.2)}$$

Thus

$$y(t) = \frac{1}{\tau}\tau\left(1-e^{-\frac{t}{\tau}}\right) = 1-e^{-\frac{t}{\tau}} \qquad \text{(Eq 5.3)}$$

and

$$\dot{y}(t) = \frac{1}{\tau} \qquad \text{(Eq 5.4)}$$

which represents the bias filter output slope. Thus, the simplified response of the bias estimation to the initial roll input is an exponential rise beginning at the instant the input is initiated. After one time constant (τ), 63 percent of the input value has been reached. The output value gradually approaches the limit of the input as time continues. This is graphically represented in Figure 19.

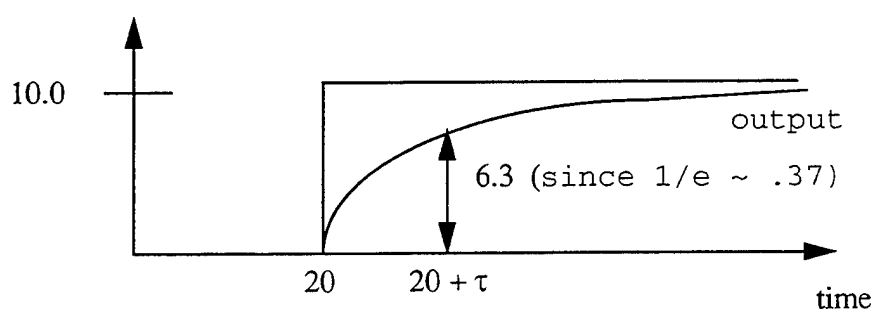

Figure 19: Bias Filter Response to a Roll Rate Step Input of 10°/sec

The developed testing procedure for the SANS allows approximately 20 seconds of initial stabilization time for the components and filter to "steady out". This was followed by an initial roll input, a similar stabilization period after the platform had reached the commanded angle, and then return to the zero position at the same rate. Typically, two of these cycles were performed under each testing condition.

To shift a unit step to start at 20 seconds

$$y(t) = \begin{cases} 0 & for \quad t < 20 \\ 1 - e^{-\frac{(t-20)}{\tau}} & for \quad t \geq 20 \end{cases} \qquad \text{(Eq 5.5)}$$

The example input pulse of 4.5 seconds can be written

$$x(t) = 10(u(t-20) - u(t-24.5)) \qquad \text{(Eq 5.6)}$$

giving

$$y(t) = \begin{cases} 0 & for \quad t < 20 \\ 10\left(1 - e^{-\frac{t-20}{\tau}}\right) & for \quad 20 \leq t \leq 24.5 \\ 10\left[\left(1 - e^{-\frac{t-20}{\tau}}\right) - \left(1 - e^{-\frac{t-24.5}{\tau}}\right)\right] & for \quad 24.5 \leq t \end{cases} \qquad \text{(Eq 5.7)}$$

Since

$$e^{x} = 1 + x + \frac{x^{2}}{2!} + \ldots \qquad \text{(Eq 5.8)}$$

then, for small x

$$y(t) \approx 10\left[1 - \left(1 - \frac{t-20}{1000}\right)\right] \approx 10\left(\frac{t-20}{1000}\right) \qquad for \qquad 20 \leq t \leq 24.5 \qquad \text{(Eq 5.9)}$$

for a time constant of 1000 seconds. This produces the first part of the response illustrated in Figure 20.

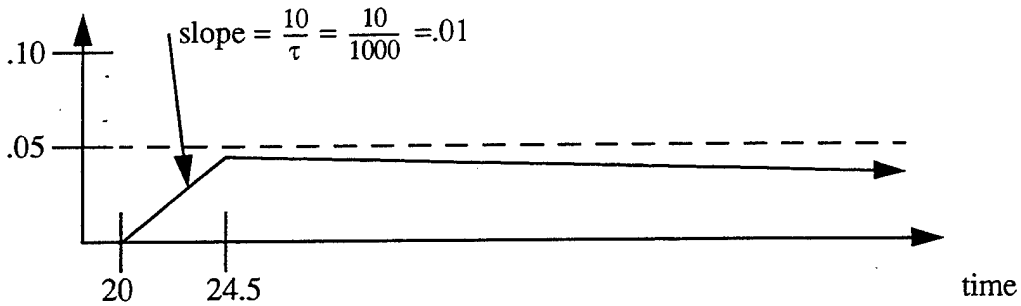

Figure 20: Estimated Short Term Bias Response to a 45 Roll Completed in 4.5 Seconds

For times equal to or greater than 24.5,

$$y(t) \approx 10\left(\frac{t-20}{1000} - \frac{t-24.5}{1000}\right)e^{-\frac{(t-24.5)}{1000}} \approx 0.045e^{-\frac{(t-24.5)}{1000}} \qquad \text{(Eq 5.10)}$$

This result is shown in Figure 21 on a longer scale to illustrate the gradual correction over time.

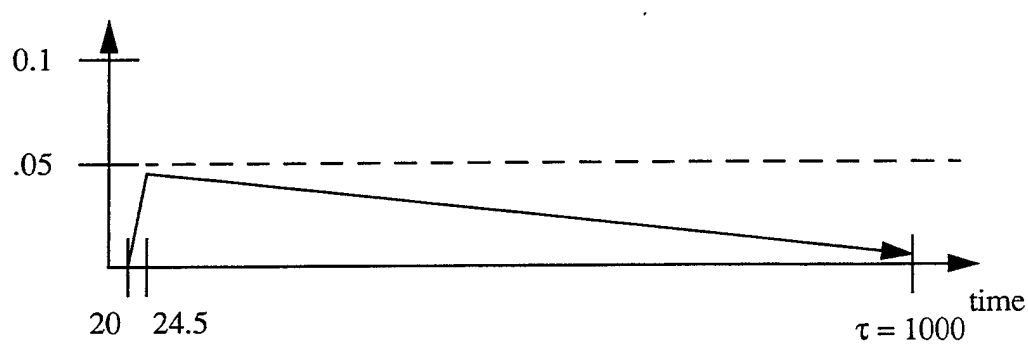

Figure 21: Estimated Long Term Bias Response to a 45° Roll Completed in 4.5 Seconds

Taken together, Figures 20 and 21 illustrate that the bias response of a low pass filter to a time-shifted step roll input is a rapid rise to the calculated value, followed over the

length of the relaxation time constant by a gradual correction to zero. Combining this response with the complementary filter design, incorporated as depicted in Figure 11, results in the time domain filter response including a time lag effect which barely sees minor transients. The initial response to a step change in attitude comes almost entirely from the angular rate sensors. Over time, input from the accelerometers takes over and compensates exactly for the decay of the rate sensors. The nature of this response influenced development of the testing methodology and is directly reflected in the following testing results.

C. FILTER TESTING METHODOLOGY

The tilt-table testing methodology has evolved through Bachmann (95) and Walker (96). Although basically unchanged from the method used in Walker (96), it is presented here in a standardized, sequential order with extensive background for the first time. It is also presented at Appendix D in a checklist format. The testing methodology is designed to separate the complementary effects of the filter and treat them individually before evaluating the entire filter process.

The SANS is mounted to the tilt-table described in (Bachmann 96) for a series of pitch and roll tests. If the unit is carefully leveled prior to testing, the actual commanded attitudes are extremely accurate in reference to real-world pitch and roll angles. Relative angle excursions are always extremely accurate on the tilt-table. The amount of the actual angle excursion is the important value for the testing. In other words, a valid 45 degree pitch from a beginning baseline of 2 degrees to 47 degrees, for example, is a successful test for the IMU. Once calibrated and installed in Phoenix, the SANS becomes the reference for

attitude determination. That is, if roll and pitch SANS outputs are zero, then this defines level orientation for Phoenix.

The general procedure is to allow a 15 to 20 second period for the sensors to initialize and stabilize after the filter code begins execution. This is followed by a pitch or roll excursion to 45 degrees at various commanded rates (consistent during each individual run). The unit is then tilted back to the zero position, followed by a roll excursion in the opposite direction, and then finally back again to zero. Each movement is followed by the stabilization period to allow observation of filter effects. Those cases where the excursions were both in the same direction reflected physical limitations as to how the SANS box could be mounted on the tilt-table. Maximum tilt rate was 90 degrees per second, but tests were normally conducted at either 10 or 45 degrees per second. These conditions are much more severe than those encountered by the SANS in the real world, with the possible exception of surfaced operations in a very heavy sea state, and therefore represent worst case performance for the filter.

In order to determine the rate sensor bias value, K_1 is set to zero to prevent accelerometer inputs from effecting the results. Therefore, only the high frequency angular rate and bias get to the first integrator. Any errors in attitude can then be attributed to the bias and scale factor. The appropriate initial angular rate scale factor (qScale for pitch, etc.) is then determined by taking the commanded tilt-table angles as truth. The scale factor adjusts the output of the IMU to the actual tilt results. Starting with a baseline of 1.0, it is possible to continuously apply the ratio of indicated and actual angles to the current setting in order to scale it to a proper value. For example, if the SANS says the unit pitched to 41°

when the actual pitch was 45°, the new scale factor is increased to 45/41 multiplied by the old scale factor.

The initial bias weight (biasWght) is chosen through a combination of project experience and filter theory considerations. Extensive simulation and tilt-table experiments can then refine the proper values prior to at-sea testing.

After setting the gain weight to some value other than zero, multiple test runs can refine the proper settings. The accelerometer scale factors are then adjusted in the same manner as the angular rate scale factors if indications show that the combined inputs result in inaccurate angle excursions. A complete tuning of one axis may take an extensive set of alternating adjustments to the various factors, as illustrated in the testing results provided.

D. IMU TEST RESULTS

The testing results included here utilized the current hardware configuration, along with the original code from Bachmann (95) only slightly modified to improve input/output rates. This resulted in update rates of approximately 18 Hz. The complete code revision described in this thesis resulted in an increased update rate of 40 Hz, making the filter real-time capable for the first time. That update rate unfortunately overwhelms the internal data storage of the SANS in the current configuration, so further testing will have to either be done at reduced rates or be conducted after new, larger storage cards (now available) have been obtained.

Figure 22 shows the initial pitch test run. Both K_1 values are set to 0.0, isolating the angle-rate input from the accelerometer input. Pitch was at a rate of 10 degrees per second.

The qScale value had already been adjusted to 4.02 to reflect 45° of pitch. When compared to previous project results (see Walker (96) and Bachmann (95)), the faster update rate significantly reduced initial overshoot of the final pitch angles. The stabilization periods following each pitch show that the effects of the filter cancel in that the initial slight overshoots gradually return to the proper value, regardless of pitch direction, as expected from the earlier explanation. In fact, for the pitch which is initiated at approximately 20 seconds, if no other pitch excursions occurred, the angle value would become essentially 45° by 1020 seconds ($20 + \tau$). The stabilization period is only a small fraction of the time constant, and the bias is subtracted from each new sample. Thus, the accumulated bias from the excursion is only partially corrected for, with a slope in the direction of the "correct" value.

Figure 23 shows a second pitch test with all values unchanged with the exception of τ, which increased from 1000 to 5000. Ideally, the filter should be initialized for a period of one time constant, however, the shorter stabilization periods here are sufficient to demonstrate filter behavior. The stabilization periods of Figure 23 show a flatter slope than those of Figure 22. This reduced slope shows that increasing τ minimizes the accumulated rate bias.

Turning to the roll axis, Figure 24 shows the initial roll test. The time constant τ has been reset to 1000 seconds. The roll rate is still 10 degrees per second and the initial scale factor (pScale) was set to be 4.01. The nearly identical scale factors show the IMU to be very consistent between axis. Otherwise, the roll results are similar to the pitch results.

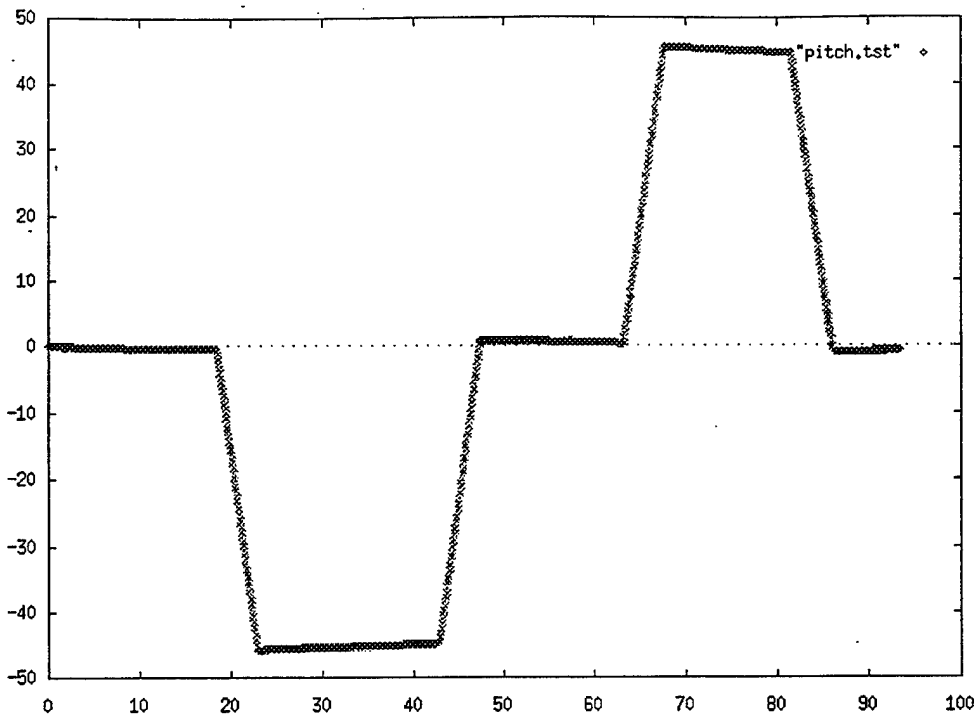

Figure 22: Initial Pitch Test, K_1 = 0.0, τ = 1000, qScale = 4.02, 10°/sec

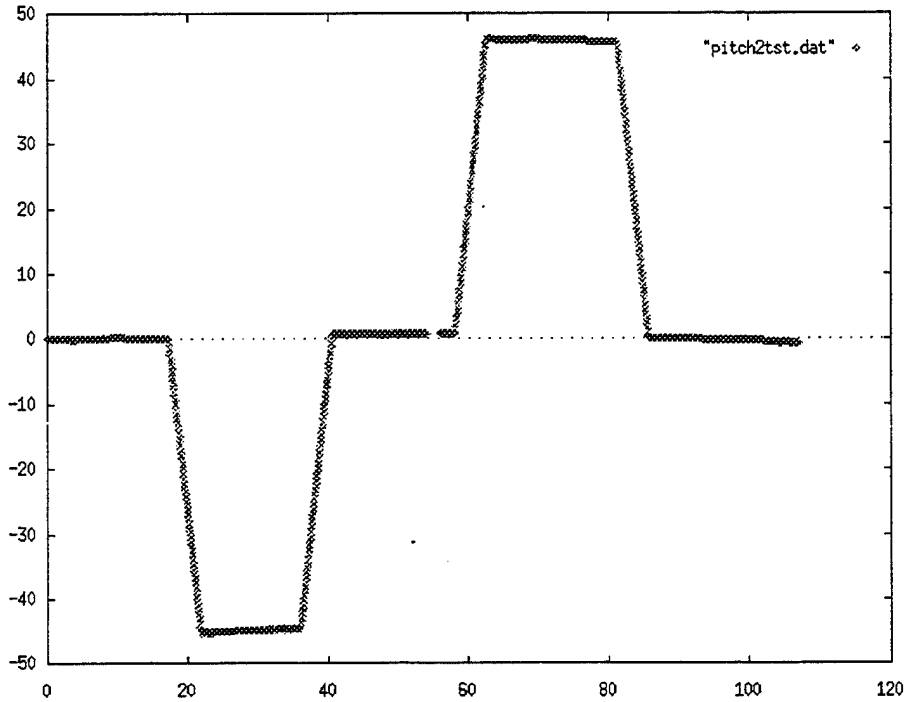

Figure 23: Pitch Test, K_1 = 0.0, t = 5000, qScale = 4.02, 10°/sec

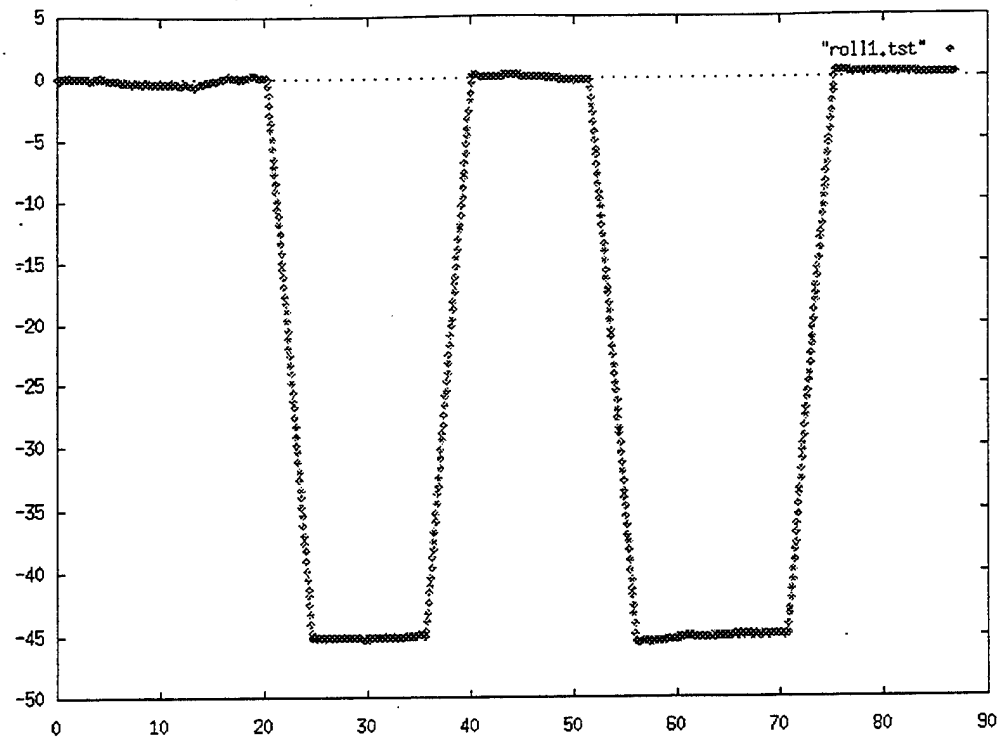

Figure 24: Initial Roll Test, K_1 = 0.0, τ = 1000, pScale = 4.01, 10°/sec

The roll rate was increased to 40 degrees per second for the second roll test. This becomes obvious with the more widely separated fix dots on the graph in Figure 25. Since the update may occur at any point during it's cycle (worst case immediately before the commanded angle is reached), more overshoot is possible, and in fact occurs. This leads to a more pronounced return effect during the stabilization periods.

The third roll test, shown at Figure 26, has identical settings to the previous test with the exception of K_1, which has been set to 0.01 to allow an accelerometer effect to return. This is what causes the wander in roll angle seen in the initial stabilization period. This effect is also present in the stabilization following the initial roll, but is less pronounced after the return to the zero position as the time grows closer to the initial time constant.

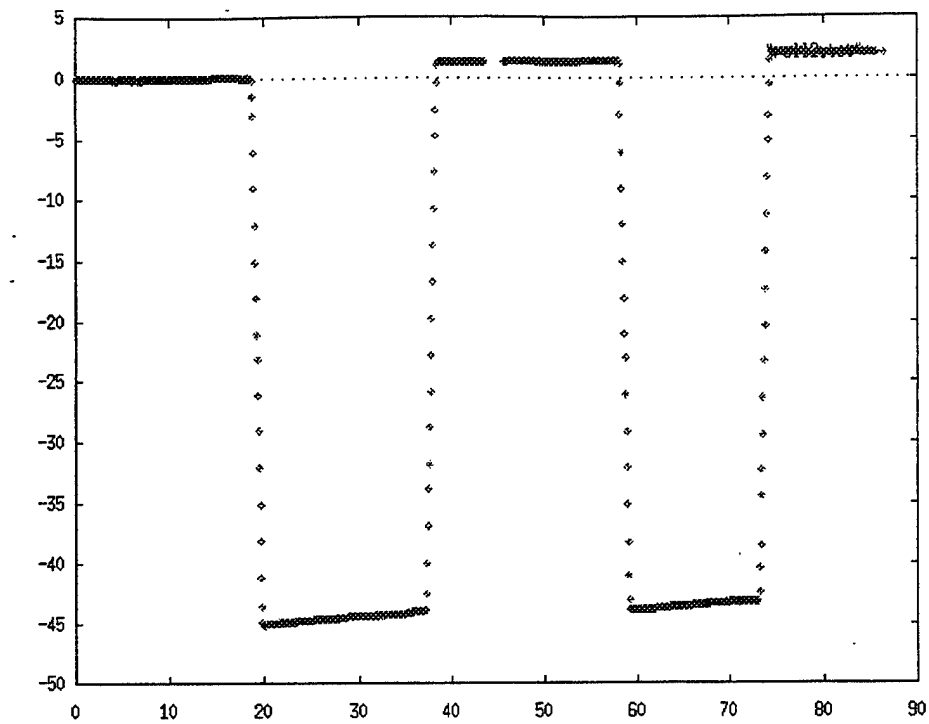

Figure 25: Roll Test: $K_1 = 0.0$, $\tau = 1000$, pScale = 4.01, 40 °/sec

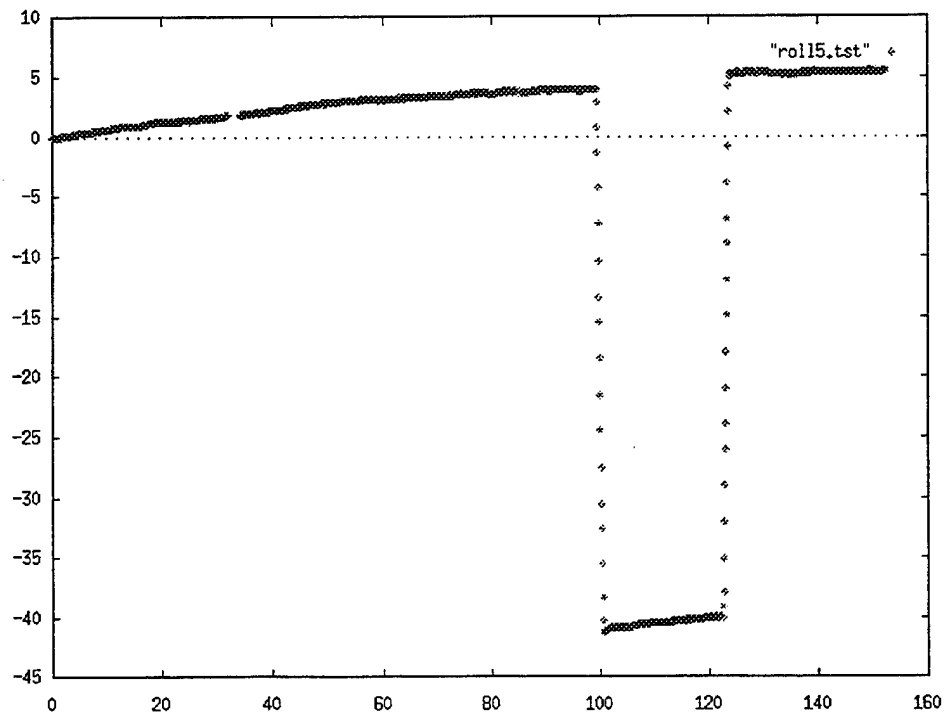

Figure 26: Roll Test: $K_1 = 0.01$, $\tau = 1000$, pScale = 4.01, 40 °/ sec

Increasing K_1 to 0.05 and reducing τ to 200 produces the results of Figure 27. These stabilization periods are characterized by more aggressive corrections to the "proper" angle. Both Figure 26 and 27 show the importance of increasing the filter update rate from the 18 Hz rate shown to the 40 Hz rate achieved in this thesis to prevent undershoot and overshoot due to sampling effects. The results of Figure 27 are essentially duplicated, although at a reduced roll rate of 10 degrees per second, in Figure 28.

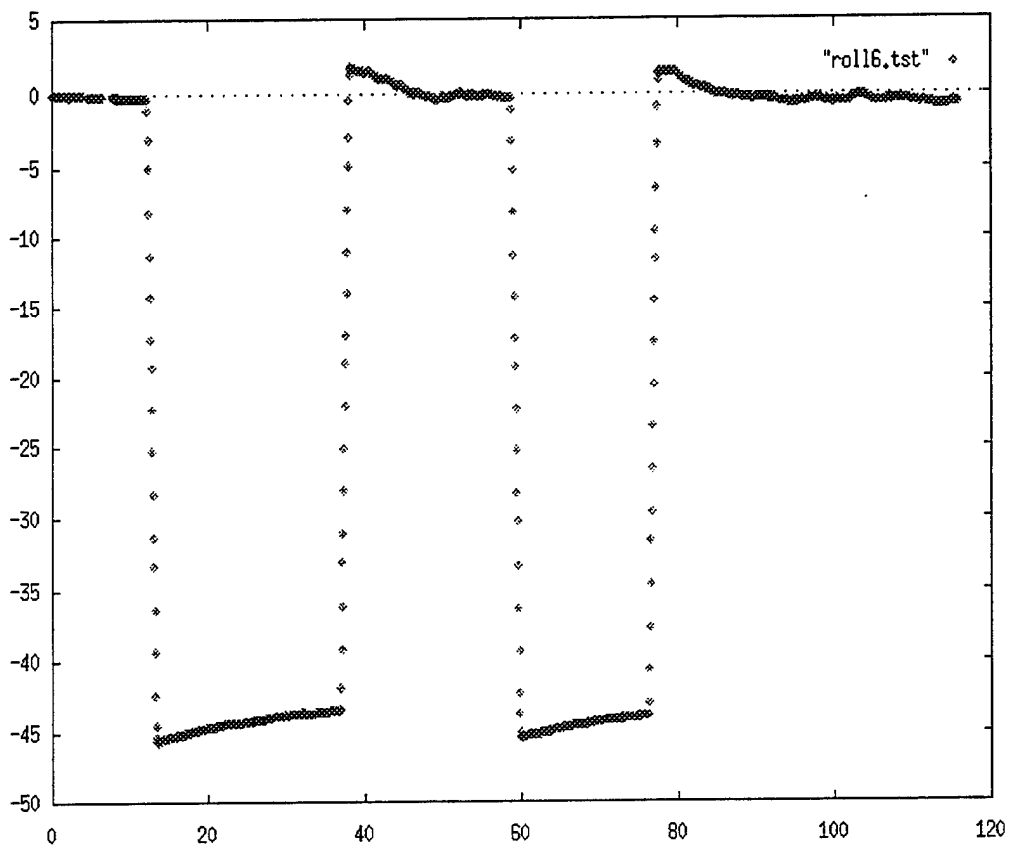

Figure 27: Roll Test: K_1 = 0.05, τ = 200, pScale = 4.01, 40 °/sec

The following roll test, Figure 29, shows the effect of varying the accelerometer scale factor (yAccelScale) from 1.34 to 1.405. The stabilization periods are flatter with respect

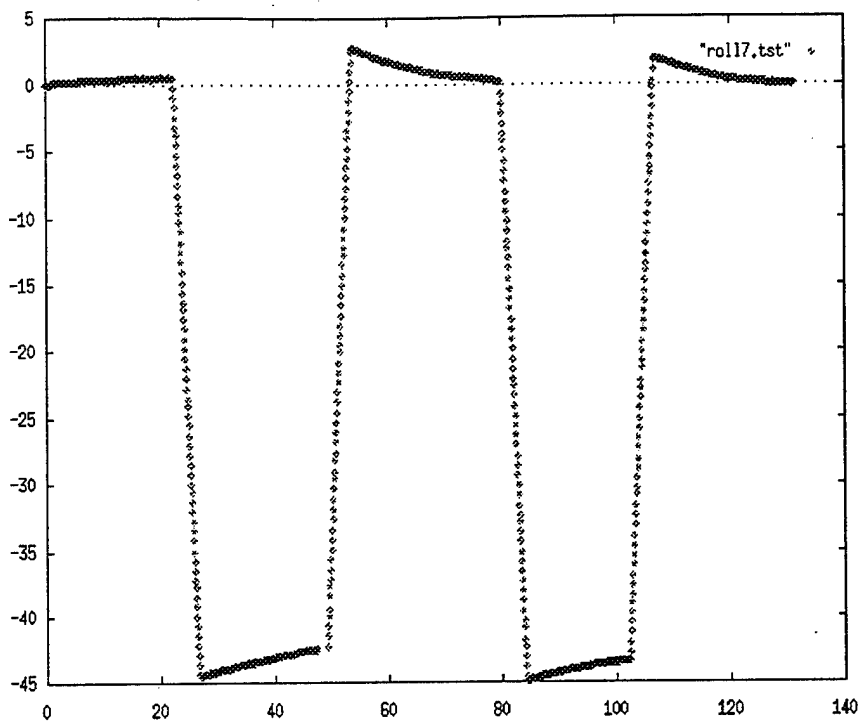

Figure 28: Roll Test: K_1 = 0.05, τ = 200, pScale = 4.01, 10°/sec

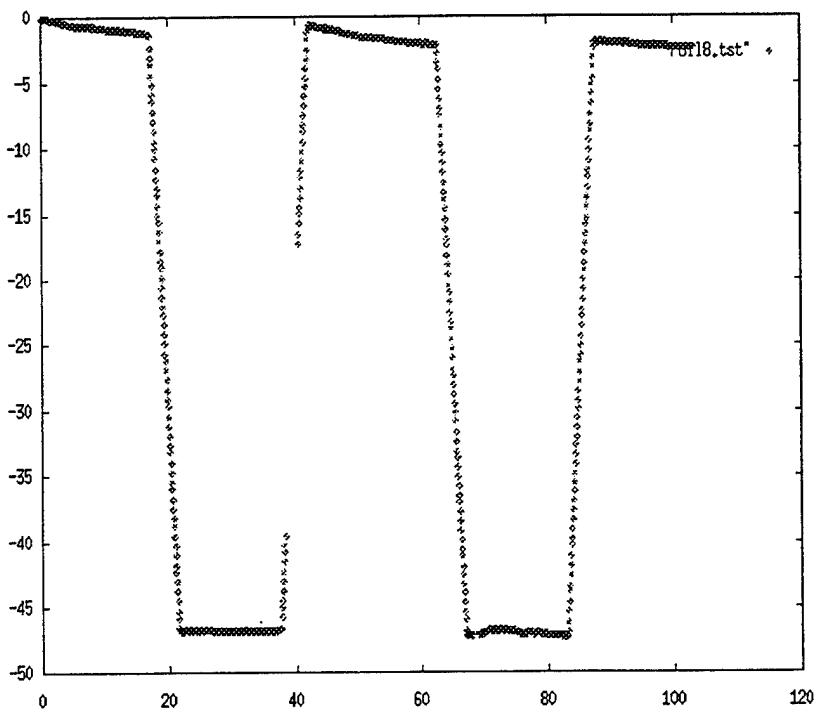

Figure 29: Same as Previous, with yAccelScale = 1.405

to the "correct" angle and the accelerometer effects less pronounced when compared to Figure 28.

Figure 30 returns the time constant to 1000 seconds while also doubling K_1. It was determined at this point that the yAccelScale value had been adjusted too high. Prior to the next roll test (Figure 31), it was adjusted by (45/48) * 1.405 since the unit computed an initial roll of 48° vice 45. Figure 31 shows a flatter response, but there is still some overshoot. The pScale was adjusted again for the test shown in Figure 32 by the amount (45/46) * 4.01. Finally, the yAccelScale was adjusted once again by (45/44) * 1.317 to produce the output in Figure 33. This sequence clearly illustrates the alternating, gradually

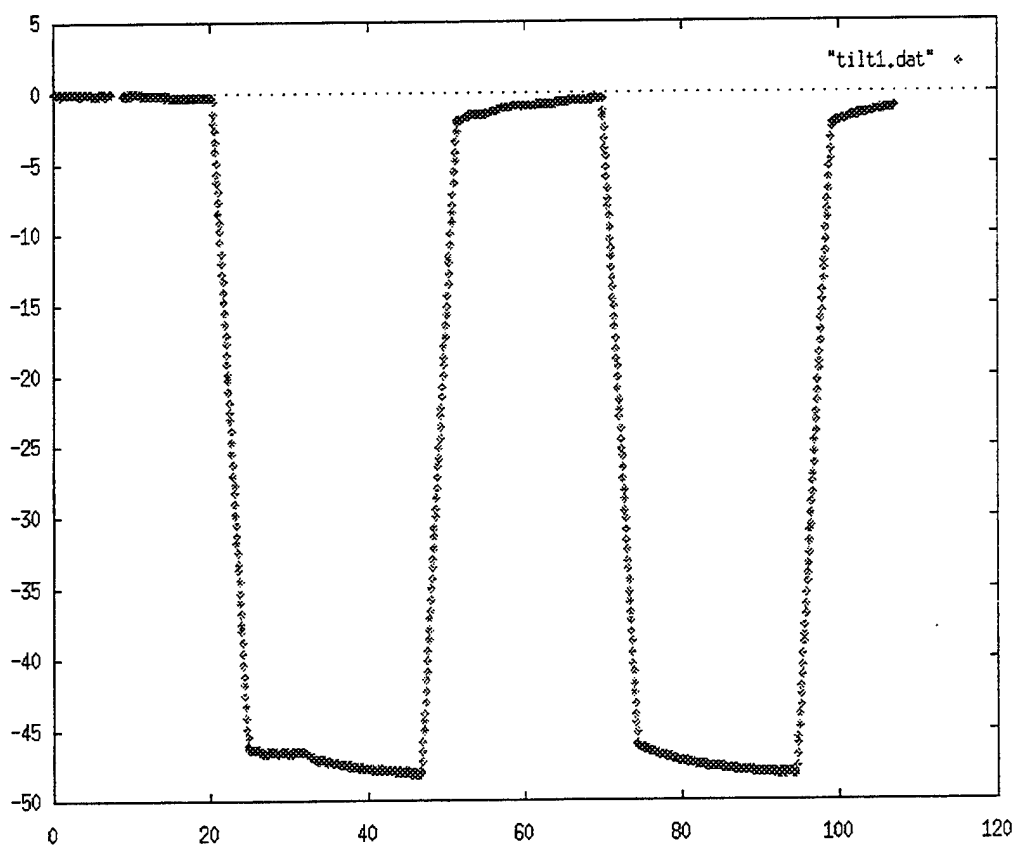

Figure 30: Roll Test: K_1 = 0.1, τ = 1000, 10°/sec, yAccelScale = 1.405

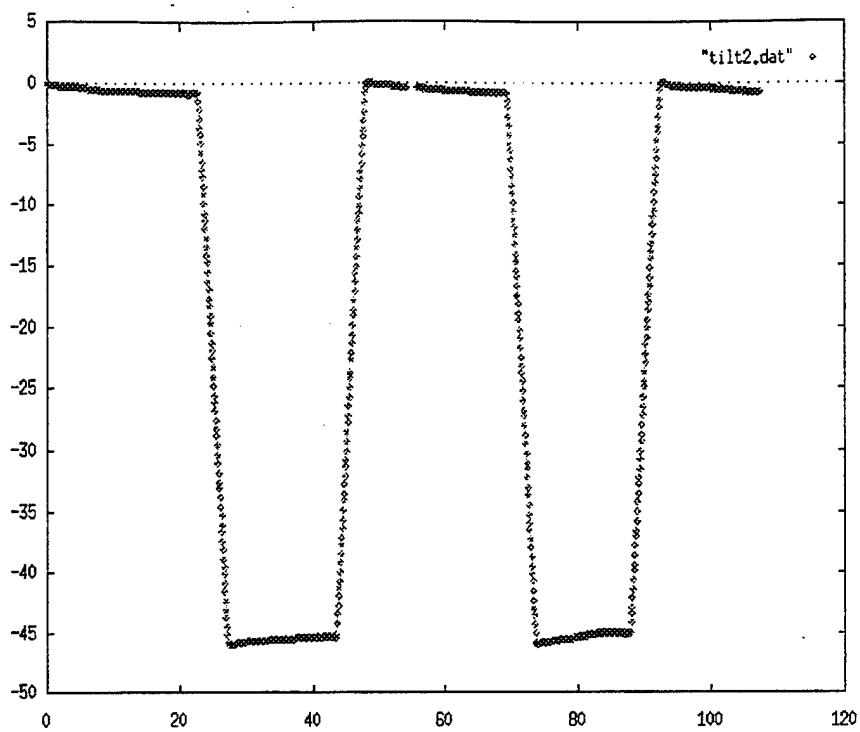

Figure 31: Roll Test: K_1 = 0.1, τ = 1000, 10°/sec, yAccelScale = 1.317

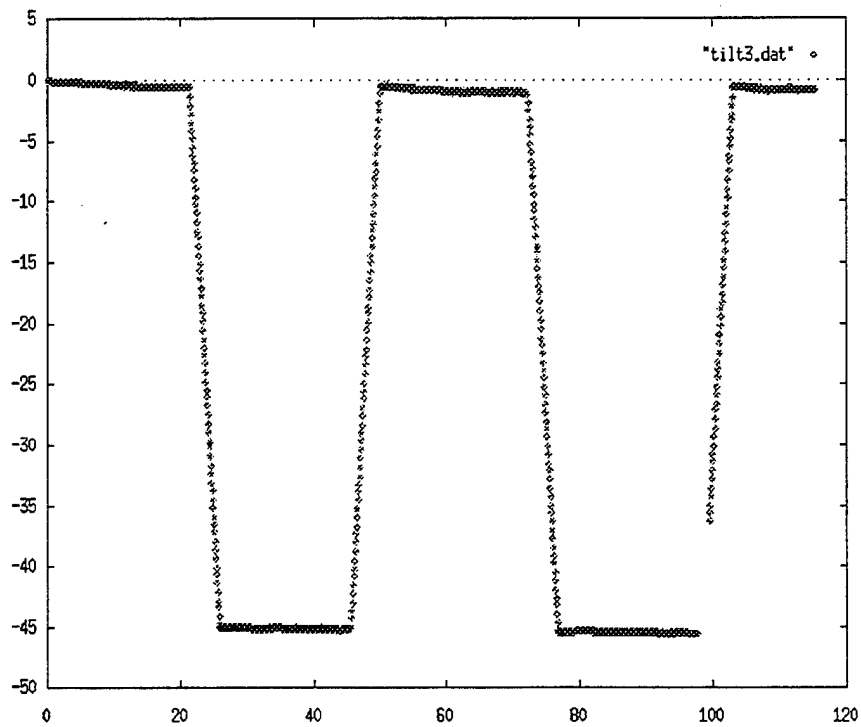

Figure 32: Roll Test: K_1 = 0.1, τ = 1000, pScale = 3.923, 10°/sec

finer tuning approach which must be taken in order to tune a filter of this complexity. Figure 33 clearly has the least overshoot/undershoot and the flattest stabilization periods while exhibiting a proper correction tendency before the next input is encountered.

Figures 34 and 35 are provided to illustrate filter response at the more radical rates of 45°/sec and 90°/sec. Although there is slightly more overshoot, as expected, even at these extremes, the filter behaves predictably and well within acceptable accuracy for the Phoenix or other small scale portable navigation applications.

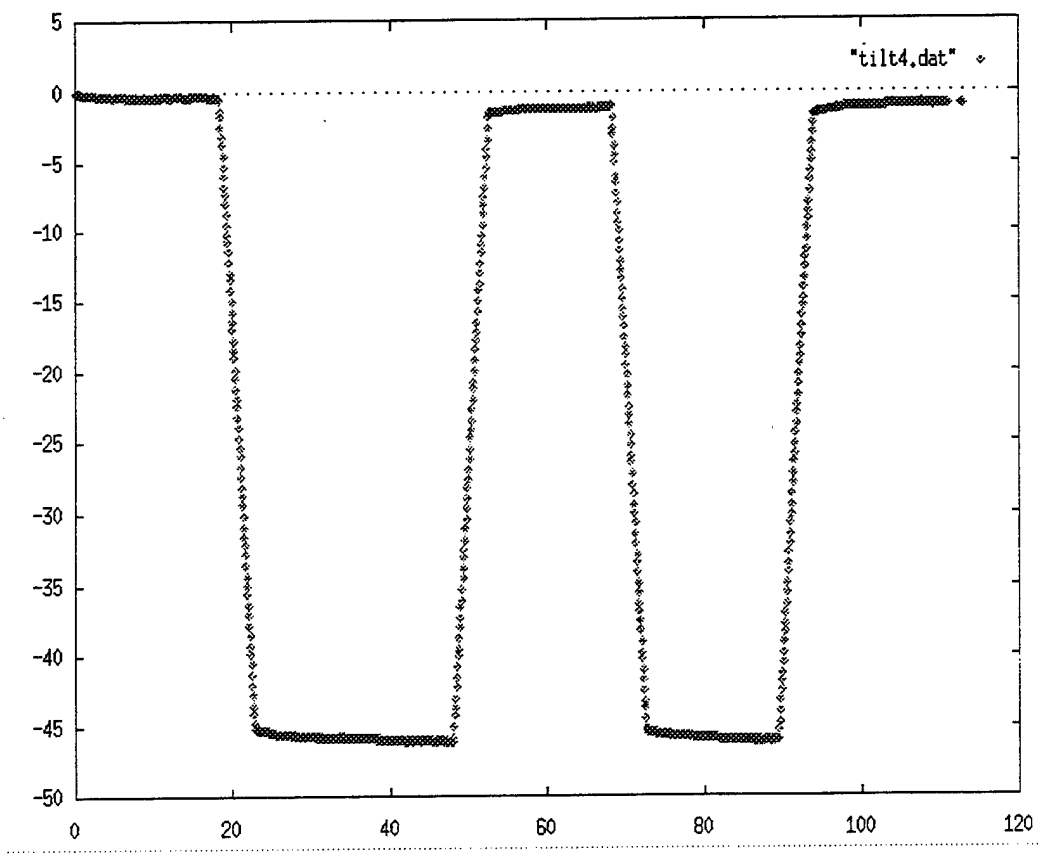

Figure 33: Roll Test: K_1 = 0.1, t = 1000, yAccelScale = 1.347, 10°/sec

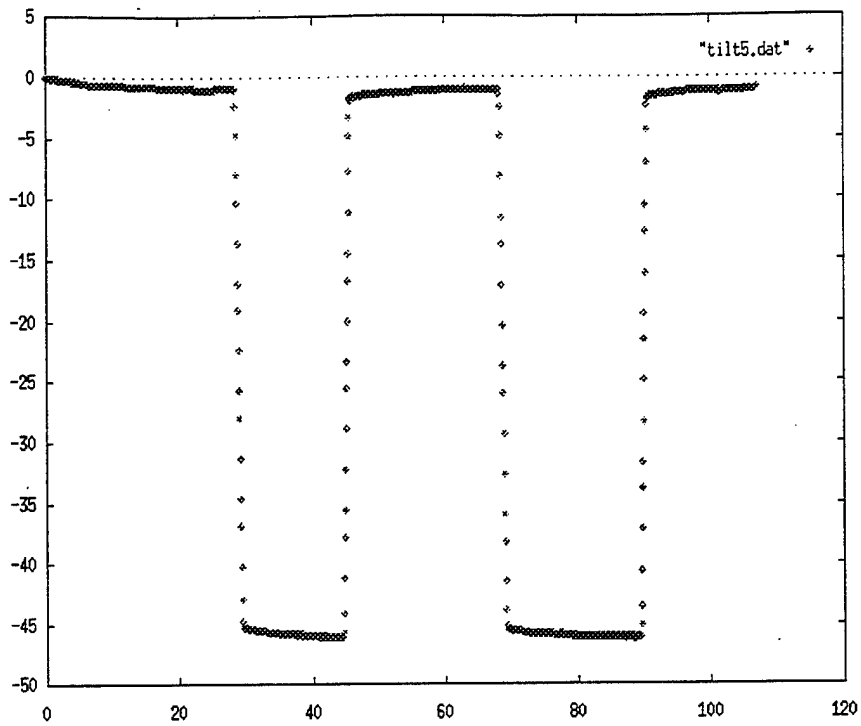

Figure 34: Roll Test: same as previous, with 45°/ sec vice 10

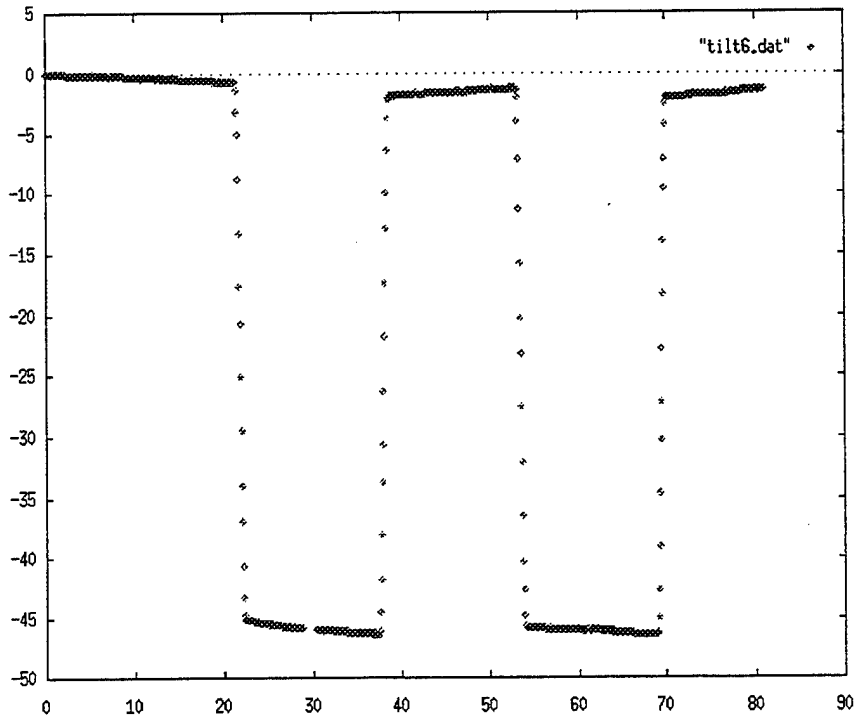

Figure 35: Roll Test: same as previous, with 90°/sec vice 45

E. SUMMARY

This chapter has provided a methodology for dynamic tilt-table testing with rationale and illustrative experimental results. Taken together, the results graphically show that the SANS design, code architecture, and filter implementation are performing as expected. Additionally, while room for some improvement remains, the sensor/filter combination is easily accurate enough to meet both the Phoenix AUV and other potential small scale portable navigation applications. It is important in reviewing the results presented to remember that these testing conditions are much more severe than are likely to be encountered in actual SANS operation except when surfaced in significant sea states. Other independent testing of the SANS approach (Henault 96) suggests that attitude estimation to an accuracy of a few tenths of degrees should be realized in normal operating conditions.

Addition of a math coprocessor to the E.S.P CPU module has increased performance dramatically and decreased the undersampling seen, as expected by Walker (96). Accompanying code revisions have resulted in a legitimate real-time navigation filter which is expected to improve accuracy even further. The final chapter of this thesis will review conclusions reached and recommendations for future project work.

VI. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

A. CONCLUSIONS

The research topics addressed by this thesis were: 1) evaluate the hardware and software architecture of the SANS, 2) develop a calibration procedure for the SANS navigation filter, 3) evaluate the specific performance of the SANS navigation filter, and 4) evaluate the SANS hardware and software architecture for installation in the Phoenix AUV. Each incremental step in the SANS project work has provided evolutionary improvement in capability and performance. Walker (96) built on the Bachmann (95) hardware prototype and provided the current hardware capability. This thesis has improved on the code architecture of Bachmann (95) to accommodate the greatly increased processing speeds resulting from the Walker (96) hardware configuration and addition of a math coprocessor.

A basic tilt-table testing methodology was utilized for an overall evaluation of the SANS attitude estimation pursuant to addressing the research issues. Combining the procedures used in Walker (96) and Bachmann (95) to produce a specific filter calibration procedure simultaneously addressed all of the topics in a general manner. The results showed that the filter is working correctly and as expected from the supporting theory. Furthermore, the real-time capability now makes SANS a bonafide option as a new navigation solution for Phoenix or alternative small scale portable navigation applications. The SANS project is now poised for meaningful at-sea trials to further validate the recent

improvements to allow further development of the linear velocity and position estimation portions of the filter.

B. RECOMMENDATIONS FOR FUTURE WORK

There remain many areas for further research on the SANS project. The next major step will be at-sea testing utilizing a tow fish as in Bachmann (95). Successful completion of these tests will make the SANS ready for adaptation and installation in Phoenix if it is chosen as the navigation solution. Incorporation into Phoenix is expected to be very straightforward. The ethernet connection can be utilized to pass the Phoenix "Officer of the Deck" software module the required navigation state elements. These elements are currently stored at each update and written to a data file. Compatibility issues should be limited to data communication between SANS and the Phoenix navigator software. In the meantime, purchase of a larger PCMCIA SRAM card will immediately alleviate the data storage problem encountered during laboratory testing resulting from the faster processing speeds.

Consideration should be given to updating the software utilized in SANS. Two approaches exist. The first is to update the DOS/BORLAND PC environment by upgrading to the latest versions. This option will entail rewriting some of the basic input/ output system function calls. The second option would be a complete rewrite to make the software compatible with the final Linux or LonWorks implementation option that is incorporated into Phoenix. Although more involved, this option is attractive because it prevents a proliferation of different operating systems within the same architecture.

Postprocessing of the navigation data file remains an unfinished area from Bachmann (95) and Walker (96). Test runs could be repeated multiple times to more easily optimize the Kalman filter gains. In a related matter, the incorporation of the aperiodic GPS updates into the overall Kalman filter scheme also still remains to be refined. The author hopes that the results presented in this thesis will prove to be valuable in this ongoing effort.

APPENDIX A: Real Time Navigation Source Code (C++)

A. TOWTYPES.H

```
#ifndef __TOETYPES_H
#define __TOETYPES_H

#include "globals.h"      // Types used by serial communications software

#define GPSBLOCKSIZE 76    // Size of Motorola @@Ea position message
#define PACKETSIZE 133    // Size of packet received via X-modem protocol
#define COMPSIZE 60

#define ONE_G 32.2185      // One g in feet per second
#define GRAVITY 32.2185    // In feet per second

#define TicksToSecs(x) ((double) ((10 * x) / 182))

typedef char  ONEBYTE;
typedef short TWOBYTE;
typedef long  FOURBYTE;

typedef unsigned char  UNSIGNED_ONEBYTE;
typedef unsigned short UNSIGNED_TWOBYTE;
typedef unsigned long  UNSIGNED_FOURBYTE;

struct latLongMilSec {      // Holds lat/long expressed in miliseconds
   long latitude;
   long longitude;
};

// Holds a latitude or longitude expressed in hours minutes and degrees
struct T_GEODETIC {
   TWOBYTE           degrees;
   UNSIGNED_TWOBYTE  minutes;
   double            seconds;
};

// Holds a latitude and longitude expressed as T_GEODETICs
struct latLongPosition {
   T_GEODETIC latitude;
   T_GEODETIC longitude;
};
struct grid {                              // Holds a grid position
   double x,y,z;
};

struct matrix {                            // 3 X 3 matrix
   float element[3][3];
};
```

```
struct vector {                                  // 3 X 1 matrix or vector
   float element[3];
};

// Oversize area to hold a GPS message
typedef BYTE GPSdata[2 * GPSBLOCKSIZE];

// Defines a type for holding compass messages
typedef BYTE compData[2 * COMPSIZE];

// Structure for passing around various types of INS information.
// The positions in the sample field of a stampedSample structure
   // sample[0]: x acceleration   gnuplot:  2
   // sample[1]: y acceleration             3
   // sample[2]: z acceleration             4
   // sample[3]: phi (roll)                 5
   // sample[4]: theta (pitch)              6
   // sample[5]: psi (yaw)                  7
   // sample[6]: water speed
   // sample[7]: heading

struct stampedSample {
   Boolean gpsFlag;              // True -- GPS fix obtained
   Boolean insFlag;              // True -- INS fix obtained
   latLongPosition navLatLong;   // posit in hours, mins, secs
   grid est;                     // position as estimated by the INS
   GPSdata satPosition;          // the latest GPS position
   float rawSample[8];          // Original readings for post processing
   double sample[11];            // sampler converted sample
   double deltaT;                // delta of the sample
   float bias[3];                // bias corrections
   float current[3];             // error correction current
};

#endif
```

B. TOEFISH.CPP

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <conio.h>
#include <dos.h>
#include <time.h>

#include "toetypes.h"
#include "nav.h"

extern compassPortClass port2;     // so breakhandler can call destructors
extern gpsPortClass port1;          // to insure cleanup on program exit

int  breakHandler(void);

void screenSetUp(void);

void printPosition (const latLongPosition&);

void positOut(stampedSample& posit);

// Write an INS packet and its timeStamp to the outPut file
void writeData(const stampedSample& drPosition, ofstream&);

// Write a GPS message to the outPut file.
void writeGpsData(const GPSdata& satPosition);


/*********************************************************************
   PROGRAM:  Main
   AUTHOR:   Eric Bachmann, Dave Gay, Rick Roberts
   DATE:     11 July 1995, last modified January 1997
   FUNCTION: Drives the navigator and its associated software.  Counts
             the positions & displays each to the screen. Exited only
             when control break (Ctrl c) is entered at the keyboard.
   RETURNS:  0
   CALLED BY: none
   CALLS:    initializeNavigator (nav.h)
             navPosit (nav.h)
             printPosition
             breakHandler
*********************************************************************/
int
main ()
{
   ctrlbrk(breakHandler);          // trap all breaks to release com ports
   setcbrk(1);                     // turn break checking on at all times
   char dataFile[] = "att.dat";
```

```
cout << "\nWriting attitude data to " << dataFile << endl;

// Instantiate the navigator (also private members gps1 & ins1)

navigatorClass nav1;

ofstream attitudeData(dataFile);

stampedSample curLoc;           // Lat/Long of most recent fix

Boolean  fixReceived(FALSE);    // True if a new fix was received
int      fixCount(0);           // Count of navigation fixes received
float    timeCount(0.0);        // Counter for screen output

cerr << "\nInitializing . . ." << endl;

nav1.initializeNavigator(curLoc);

// Check a2d initialization, channels off if y-accel != -32.2
while (curLoc.sample[2] <= -33.0 || curLoc.sample[2] >= -31.5) {
   cerr << "reinitializing for a2d channelization" << endl;
   nav1.initializeNavigator(curLoc);
   nav1.navPosit(curLoc);
}

clrscr();
gotoxy(1,6);
cerr << "Initialization Complete!" << endl;
cout << "Initial Position:" << endl;

// Print the initial position
cout << "latitude: " << curLoc.navLatLong.latitude.degrees << ':'
     << curLoc.navLatLong.latitude.minutes << ':'
     << curLoc.navLatLong.latitude.seconds << endl;
cout << "longitude: " << curLoc.navLatLong.longitude.degrees << ':'
     << curLoc.navLatLong.longitude.minutes << ':'
     << curLoc.navLatLong.longitude.seconds;

screenSetUp();

while (TRUE)  {            // Attempt to get a fix from the navigator
   fixReceived = nav1.navPosit(curLoc);

   if (fixReceived) {         // New fix received
      // Save fix info to the data file
      writeData(curLoc, attitudeData);
      // Print info to screen at designated print interval
      fixCount++;
      timeCount += curLoc.deltaT;
```

```
        if (timeCount >= 1.0) {
           gotoxy(9,11);
           cout << fixCount << endl;
           positOut(curLoc);
           timeCount = 0.0;
        }
     }
   }
}

/*********************************************************************
   PROGRAM:   printPosition
   AUTHOR:    Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Displays position to the screen
   RETURNS:   void
   CALLED BY: main
   CALLS:     none

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

void printPosition (const latLongPosition& posit)
{
   gotoxy(11,14);
   cout << posit.latitude.degrees << ':'<<
         posit.latitude.minutes << ':' << posit.latitude.seconds << endl;
   gotoxy(12,15);
   cout << posit.longitude.degrees << ':'<<
           posit.longitude.minutes << ':' << posit.longitude.seconds
        << endl;
}


/*********************************************************************
   PROGRAM:   breakHandler
   AUTHOR:    Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995
   FUNCTION:  Cleans up com ports upon program exit.
   RETURNS:   0
   CALLED BY: main
   CALLS:     compass port and gps port destructors

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

int breakHandler(void)
{
   port2.~compassPortClass();
   port1.~gpsPortClass();
   return 0;                        // keep the compiler happy
}
```

```
/*********************************************************************
    PROGRAM:    screenSetup
    AUTHOR:     Eric Bachmann, Randy Walker
    DATE:       12 May 1996
    FUNCTION:   Sets up the output screen
    RETURNS:    0
    CALLED BY:  main
    CALLS:      none

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

void screenSetUp(void)
{
   gotoxy(4,11);
   cout << "Fix ";

   gotoxy(1,14);
   cout << "Latitude: " << "\nLongitude: ";

   gotoxy(1,17);
   cout << "Roll: " << "\nPitch: ";

   gotoxy(1,25);
   cout << "deltaT: ";

   int col(45),row(1);

   gotoxy(col,row++);
   cout << "x accel: ";
   gotoxy(col,row++);
   cout << "y accel: ";
   gotoxy(col,row++);
   cout << "z accel: ";
   gotoxy(col,row++);
   cout << "phi dot: ";
   gotoxy(col,row++);
   cout << "theta dot: ";
   gotoxy(col,row++);
   cout << "psi dot: ";
   gotoxy(col,row++);
   cout << "water speed: ";
   gotoxy(col,row++);
   cout << "heading: ";

   col = 45;
   row = 12;

   gotoxy(col,row++);
   cout << "x: ";
   gotoxy(col,row++);
   cout << "y: ";
```

```
   gotoxy(col,row++);
   cout << "z: ";
   gotoxy(col,row++);
   cout << "phi: ";
   gotoxy(col,row++);
   cout << "theta: ";
   gotoxy(col,row++);
   cout << "psi: ";

   gotoxy(45,20);
   cout << "Bias Values";

   gotoxy(60,20);
   cout << "Current Values";
}


/*********************************************************************
   PROGRAM:    positOut
   AUTHOR:     Eric Bachmann
   DATE:       21 October 1996
   FUNCTION:   Updates the Screen
   RETURNS:    0
   CALLED BY:  main
   CALLS:      none

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

void positOut(stampedSample& posit)
{
   printPosition(posit.navLatLong);

   if (posit.gpsFlag) {
      gotoxy(20,11);
      cout << "GPS";
   }

   else {
      gotoxy(20,11);
      cout << "   ";
   }

   // Output the bias values
   for(int j = 3; j < 6; j++) {
      gotoxy(45,j+18);
      cout << posit.bias[j];
   }

   // Set output precision and fixed format
//cout.precision(6);
//cout.setf(ios::fixed);
```

```
   // Display linear accelrations and angular rates
   for (j = 0; j < 8; j++) {
      gotoxy(59,j+1);
      cout << posit.rawSample[j];
   }

   // Display time delta to the screen.
   gotoxy(9,25);
   cout << posit.deltaT;

   // Display roll and pitch
   gotoxy(8,17);
   cout << (posit.sample[3] * radToDeg);
   gotoxy(8,18);
   cout << (posit.sample[4] * radToDeg);

   // Display current location and posture
   for (j = 0; j < 6; j++) {
      gotoxy(52,j+12);
      cout << posit.sample[j];
   }

   // Display error current values
   for (j = 0; j < 3; j++) {
      gotoxy(60,j+21);
      cout << posit.current[j];
   }

   // Output the biases
   for (j = 3; j < 6; j++) {
      gotoxy(45,j+18);
      cout << posit.bias[j];
   }
}



/*********************************************************************
   PROGRAM:     writeData
   AUTHOR:      Eric Bachmann, Dave Gay
   DATE:        11 July 1995
   FUNCTION:    Writes the packet and the time stamp contained in a stamped
                sample to the out put file for post processing.
   RETURNS:     void
   CALLED BY:   navPosit (nav.cpp)
   CALLS:       None

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

```
void writeData(const stampedSample& drPosition, ofstream& attitudeData)
{
   static float elapsedTime(0.0);

   elapsedTime += drPosition.deltaT;

   // Output attitude data to a file
   attitudeData
               << elapsedTime << ' '
               << drPosition.sample[0] << ' '
               << -1.0 * drPosition.sample[1] << ' '
               << drPosition.sample[2] << ' '
               << (radToDeg * drPosition.sample[3]) << ' '
               << (radToDeg * drPosition.sample[4]) << ' '
               << (radToDeg * drPosition.sample[5]) << ' '
               << drPosition.sample[6] << ' '
               << (radToDeg * drPosition.sample[7]) << ' '
               << drPosition.current[0] << ' '
               << drPosition.current[1] <<endl;
}


/*********************************************************************
   PROGRAM:      writeGpsData
   AUTHOR:       Eric Bachmann, Dave Gay
   DATE:         11 July 1995
   FUNCTION:     Writes a raw GPS message to a binary output file for
                 post processing.
   RETURNS:      void
   CALLED BY:    navPosit (nav.cpp)
   CALLS:        None

*********************************************************************/
/*
void
navigator::writeGpsData(const GPSdata& satPosition)
{
   for( int j = 0; j < GPSBLOCKSIZE; j++) {
      putc(satPosition[j], rawData);
   }
}
*/

// end of file toefish.cpp
```

C. NAV.H

```
#ifndef _NAVIGATOR_H
#define _NAVIGATOR_H
#include <stdio.h>
#include <fstream.h>
#include <iostream.h>
#include <math.h>
#include <dos.h>
#include "toetypes.h"
#include "globals.h"
#include "gps.h"
#include "ins.h"

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

   CLASS:    navigatorClass
   AUTHOR:   Eric Bachmann, Dave Gay, Rick Roberts
   DATE:     11 July 1995, Modified January 1997
   FUNCTION: Combines GPS and INS information to return the current
             estimated position.

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

class navigatorClass {

   public:

      // Constructor, initializes object slots
      navigatorClass() : gpsSpeedSum(0.0), insSpeedSum(0.0)
         { cerr << "\nconstructing nav1" << endl; };

      ~navigatorClass() {}                    // Destructor

      // provides the navigator's best estimate of current position
      Boolean navPosit (stampedSample&);

      // Initialize the navigator
      Boolean initializeNavigator(stampedSample&);

     void userInitNav(stampedSample&);   // Allows user to initialize nav

   private:

      double gpsSpeed, insSpeed, gpsSpeedSum, insSpeedSum;

      insClass ins1;                  // ins object instance

      gpsClass gps1;                  // gps object instance

      // Obtains system time to utilize for origin
```

```
    double getSystemTime();

    latLongMilSec origin;            // lat-long of navigational origin

    // Returns the position in Miliseconds
    latLongMilSec getMilSec(const GPSdata&);

    // Returns the position in degrees, minutes, seconds and milisecs
    latLongMilSec latLongToMilSec(const latLongPosition&);

    // Convert position in milSec to degress, minutes, seconds and milsec
    latLongPosition milSecToLatLong(const latLongMilSec&);

    // Convert xy (grid) position to lat long
    latLongMilSec gridToMilSec(const grid&);

    // Converts lat/long to xy position
    grid milSecToGrid(const latLongMilSec&);

    // Parses and returns the time of a GPS message.
    double getGpsTime(const GPSdata& rawMessage);

    // Parses and returns the velocity in fps of a GPS message.
    double getGpsVelocity(const GPSdata& rawMessage);
};
#endif
```

D. NAV.CPP

```
#include <signal.h>
#include <dos.h>
#include <time.h>
#include "nav.h"

#define SIGFPE 8            // Floating point exception

/*********************************************************************
   PROGRAM:    navPosit

   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Provides the navigator's best estimate of current position.
               Attempts to obtain GPS and INS position fixes from the gps
               and ins objects and copies the most accurate fix available
               into the input argument 'navPosition'.  Sets a return
               flag to indicate whether a valid fix was obtained.
   RETURNS:    TRUE, a valid position fix is in the variable 'navPosition'.
               FALSE, otherwise.
   CALLED BY:  towfish.cpp (main)
   CALLS:      gpsPosition (gps.h)              gridToMilSec (nav.h)
               correctPosition (ins.h)          milSecToGrid (nav.h)
               insPosition (ins.h)              milSecToLatLong (nav.h)
               getMilSec (nav.h)                writeScriptPosit (nav.h)

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

void fpeNavPosit(int sig)
{if (sig == SIGFPE) cerr << "floating point error in navPosit\n";}

Boolean navigatorClass::navPosit (stampedSample& posit)
{
   signal (SIGFPE, fpeNavPosit);

   latLongMilSec gpsMilSec;       // the latest GPS position in milseconds

   latLongMilSec insMilSec;       // the latest INS position in milseconds

   // Attempt to get the INS and GPS positions
   posit.insFlag = ins1.insPosition(posit);
   posit.gpsFlag = gps1.gpsPosition(posit.satPosition);

   // INS and GPS positions obtained?
   if (posit.insFlag && posit.gpsFlag) {
      // Parse position from GPS messsage
      gpsMilSec = getMilSec(posit.satPosition);

      posit.est = milSecToGrid(gpsMilSec);
```

```
      // Pass GPS position to INS object for navigation corrections.
      ins1.correctPosition(posit, getGpsTime(posit.satPosition));

      // Convert position in milisec to latitude and longitude.
      posit.navLatLong = milSecToLatLong(gpsMilSec);

      return TRUE;
   }
   else {
      if (posit.insFlag) {                // Only INS position obtained?
          insMilSec = gridToMilSec(posit.est);
          posit.navLatLong = milSecToLatLong(insMilSec);
          insSpeed = posit.sample[6];
          return TRUE;
      }
      else {
         if (posit.gpsFlag) {             // Only GPS position obtained?
            // Parse position from GPS messsage
            gpsMilSec = getMilSec(posit.satPosition);
            posit.est = milSecToGrid(gpsMilSec);

           // Pass GPS position to INS object for navigation corrections.
            ins1.correctPosition(posit, getGpsTime(posit.satPosition));

            // Convert position in milisec to lat/long.
            posit.navLatLong =
               milSecToLatLong(getMilSec(posit.satPosition));

            return TRUE;
         }
         else {
            return FALSE; // No new position available
         }
      }
   }
}


/*********************************************************************
   PROGRAM:     initializeNavigator
   AUTHOR:      Eric Bachmann, Dave Gay, Rick Roberts
   DATE:        11 July 1995
   FUNCTION:    Obtains an initial GPS fix for use as a navigational origin
                for grid positions used by the INS object. Saves the origin
                and passes it to the INS object in latLong form.
   RETURNS:     TRUE
   CALLED BY:    towfish (main)
   CALLS:        gpsPosition (gps.cpp)          writeGpsData(nav.cpp)
                 correctPosition (ins.cpp)      getMilSec (nav.cpp)
                 writeInsData(nav.cpp)          milSecToGrid (nav.cpp)
*********************************************************************/
```

```
Boolean navigatorClass::initializeNavigator(stampedSample& posit)
{
   Boolean gpsFlag(FALSE);

   cerr << "Initializing Navigator." << endl;
   cerr << "   Initializing GPS." << endl;

   // Loop until an initial GPS fix is obtained.
   for (int i = 1 ; ((i < 100) && (gpsFlag == FALSE)) ; i++) {
      if (gps1.gpsPosition(posit.satPosition)) {
         gpsFlag = TRUE;
      }
      else {
         delay(500);
      }
   }
   if (gpsFlag == FALSE) {
      cerr << "\nWARNING: UNABLE TO OBTAIN INITIAL GPS FIX!" << endl;
      userInitNav(posit);
   }
   else {
      cerr << "   GPS initialization complete." << endl;

      // Save navigational origin for later grid position conversions.
      origin = getMilSec(posit.satPosition);

      // Pass time of first GPS fix to INS object initialization routine.
      ins1.insSetUp(getGpsTime(posit.satPosition), posit);
   }
   cerr << "Navigator initialization complete." << endl;

   return TRUE;
}
```

```
/*********************************************************************
   PROGRAM:      userInitNav
   AUTHOR:       Rick Roberts
   DATE:         03 November 1996
   FUNCTION:     Allows user to input current position and initialize
                 nav if no gps fix is available. (ie for testing)
   RETURNS:      void
   CALLED BY:    initializeNavigator
   CALLS:        getMilSec (nav.cpp), getSystemTime (nav.cpp)

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

```
void navigatorClass::userInitNav(stampedSample& posit)
{
   int choice;

   cerr << "\nEnter a 0 to enter posit and continue without GPS"
        << "\nEnter a 1 to continue without GPS or initial posit, or"
        << "\nEnter a 2 to exit: " << endl;
   cin >> choice;

   if (choice == 0) {
     cerr << "\nEnter current position in the following format: " << endl;
      cerr << "Latitude: (36, Enter, 35 Enter, 41.5 Enter)" << endl;
      cin  >> posit.navLatLong.latitude.degrees;
      cin  >> posit.navLatLong.latitude.minutes;
      cin  >> posit.navLatLong.latitude.seconds;
      cerr << "Longitude: (-121, Enter, 52, Enter, 30.2, Enter)" << endl;
      cin  >> posit.navLatLong.longitude.degrees;
      cin  >> posit.navLatLong.longitude.minutes;
      cin  >> posit.navLatLong.longitude.seconds;
   }
   else if (choice == 2) { exit(1); }

   // Save nav origin for later grid position conversions
   origin = latLongToMilSec(posit.navLatLong);

   // Pass system time of initialization to ins object
   ins1.insSetUp(getSystemTime(), posit);
}
/*********************************************************************
   PROGRAM:      latLongToMilSec
   AUTHOR:       Rick Roberts
   DATE:         22 January 1997
   FUNCTION:     Converts a position expressed in latitude and longitude
                degrees, minutes and seconds to mili seconds & returns it.
   RETURNS:      latLongMilSec
   CALLED BY:    userInitNav
   CALLS:        none

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

latLongMilSec navigatorClass::latLongToMilSec(const latLongPosition&
latLong)
{
   latLongMilSec milSec;
   double degrees, minutes, seconds;

   milSec.latitude = (latLong.latitude.degrees * DEGREES_TO_MSECS) +
                     (latLong.latitude.minutes * MINS_TO_MSECS) +
                     (latLong.latitude.seconds * 1000.0);

   milSec.longitude = (latLong.longitude.degrees * DEGREES_TO_MSECS) +
```

```
                    (latLong.longitude.minutes * MINS_TO_MSECS) +
                    (latLong.longitude.seconds * 1000.0);
   return milSec;
}


/*********************************************************************
   PROGRAM:      getSystemTime
   AUTHOR:       Rick Roberts
   DATE:         03 November 1996
   FUNCTION:     Obtains system time to utilize for origin.
   RETURNS:      double (origin time in seconds)
   CALLED BY:    userInitNav
   CALLS:        dos time function
*********************************************************************/

double navigatorClass::getSystemTime()
{
   dostime_t* sysTime;                    // pointer to dos time structure

   _dos_gettime(sysTime);

   return double((sysTime->hour * 3600.0) + (sysTime->minute * 60.0)
       + (sysTime->second));
}


/*********************************************************************
   PROGRAM:  getMilSec
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Extracts a position in miliseconds from a Motorola (@@Ba)
             position contained in the input argument 'rawMessage'.
   RETURNS:  The latitude and longitude in miliseconds.
   CALLED BY:   navPosit (nav.cpp)
                initializeNavigator (nav.cpp)
   CALLS:       none.
*********************************************************************/

latLongMilSec navigatorClass::getMilSec(const GPSdata& rawMessage)
{
   FOURBYTE temps4byte;
   latLongMilSec position;

   temps4byte        = rawMessage[15];
   temps4byte        = (temps4byte<<8) + rawMessage[16];
   temps4byte        = (temps4byte<<8) + rawMessage[17];
   temps4byte        = (temps4byte<<8) + rawMessage[18];

   position.latitude = temps4byte;
```

```
   temps4byte        = rawMessage[19];
   temps4byte        = (temps4byte<<8) + rawMessage[20];
   temps4byte        = (temps4byte<<8) + rawMessage[21];
   temps4byte        = (temps4byte<<8) + rawMessage[22];

   position.longitude = temps4byte;

   return position;
}

/*********************************************************************
   PROGRAM:  milSecToLatLong
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Converts a position expressed totally in miliseconds to
             degrees, minutes, seconds and miliseconds.
   RETURNS:  The position in degrees, minutes, seconds and miliseconds.
   CALLED BY:   navPosit (nav.cpp)
   CALLS:       none
*********************************************************************/

latLongPosition navigatorClass::milSecToLatLong(const latLongMilSec&
milSec)
{
   latLongPosition  position;

   double degrees, minutes;

   degrees = (double)milSec.latitude * MSECS_TO_DEGREES;
   position.latitude.degrees = (TWOBYTE)degrees;

   if(degrees < 0) {
      degrees = fabs(degrees);
   }
   minutes = (degrees - (TWOBYTE)degrees) * 60.0;
   position.latitude.minutes = (TWOBYTE)minutes;
   position.latitude.seconds = (minutes - (TWOBYTE)minutes) * 60.0;

   degrees = (double)milSec.longitude * MSECS_TO_DEGREES;
   position.longitude.degrees = (TWOBYTE)degrees;

   if(degrees < 0) {
      degrees = fabs(degrees);
   }
   minutes = (degrees - (TWOBYTE)degrees) * 60.0;
   position.longitude.minutes = (TWOBYTE)minutes;
   position.longitude.seconds = (minutes - (TWOBTE)minutes) * 60.0;

   return position;
}
```

```
/*********************************************************************
   PROGRAM:  gridToMilSec
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Convert a grid position to a latitude and longitude in mili-
             seconds and returns the result.
   RETURNS:  The latitude and longitude in miliseconds.
   CALLED BY: navPosit (nav.cpp)
   CALLS:     none
*********************************************************************/

void fpeGridToMilSec(int sig)
{if (sig == SIGFPE) cerr << "floating point error in gridToMilSec\n";}

latLongMilSec navigatorClass::gridToMilSec(const grid& posit)
{
   signal(SIGFPE, fpeGridToMilSec);
   latLongMilSec  latLong;

   // converts grid in ft to latitude
   latLong.latitude = origin.latitude + (posit.x / LatToFt);
   // converts grid in ft to longitude
   latLong.longitude = origin.longitude +
  HemisphereConversion * (posit.y / LongToFt);
   return latLong;
}


/*********************************************************************
   PROGRAM:  milSecToGrid
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Convert a latitude and longitude expressed in milseconds to
             a grid position in xy coordinates in feet from the origin.
   RETURNS:  The grid position
   CALLED BY: navPosit (nav.cpp), initializeNavigator (nav.cpp)
   CALLS:     none
   COMMENTS:  altitude is always assumed to be zero.
*********************************************************************/
grid navigatorClass::milSecToGrid(const latLongMilSec& posit)
{
   grid  position;

   position.x = (posit.latitude - origin.latitude) * LatToFt;
   position.y = HemisphereConversion *
  (posit.longitude - origin.longitude) * LongToFt;
   position.z = 0;

   return position;
}
```

```
/*********************************************************************
   PROGRAM:  getGpsTime
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Parse the time of a gps message.
   RETURNS:  The time of the gps message in seconds
   CALLED BY: navPosit (nav.cpp), initializeNavigator (nav.cpp)
   CALLS:     none
*********************************************************************/

double navigatorClass::getGpsTime(const GPSdata& rawMessage)
{
   UNSIGNED_ONEBYTE     tempchar, hours, minutes;
   UNSIGNED_FOURBYTE    tempu4byte;
   double seconds;

   hours   = rawMessage[8];
   minutes = rawMessage[9];

   tempchar         = rawMessage[10];
   tempu4byte       = rawMessage[11];
   tempu4byte       = (tempu4byte<<8) + rawMessage[12];
   tempu4byte       = (tempu4byte<<8) + rawMessage[13];
   tempu4byte       = (tempu4byte<<8) + rawMessage[14];
   seconds = (double)tempchar + (((double)tempu4byte)/1.0E+9);

   return hours * 3600.0 + minutes * 60.0 + seconds;
}


/*********************************************************************
   PROGRAM:  getGpsVelocity
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Parse the velocity out of a gps message.
   RETURNS:  The velocitiy of the gps message in feet per second
   CALLED BY: navPosit (nav.cpp), initializeNavigator (nav.cpp)
   CALLS:     none
*********************************************************************/

double navigatorClass::getGpsVelocity(const GPSdata& rawMessage)
{
   UNSIGNED_ONEBYTE tempchar=rawMessage[31];

   return (double)(3.2804 * ((tempchar << 8) + rawMessage[32]) / 100.00);
}
// end of file nav.cpp
```

E. GPS.H

```
#ifndef _GPS_H
#define _GPS_H

#include <iostream.h>
#include <fstream.h>
#include <conio.h>

#include "toetypes.h"
#include "globals.h"
#include "gpsPort.h"

/*********************************************************************
   CLASS:    gpsClass
   AUTHOR:   Eric Bachmann, Dave Gay, Rick Roberts
   DATE:     11 July 1995, last modified January 1997
   FUNCTION: Reads GPS messages from the GPS buffer. Checks for valid
             checksum and minimun number of satellites in view.

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

class gpsClass {

   public:

      // Class constructor and destructor
      gpsClass() { cerr << "\nconstructing gps1" << endl; };
      ~gpsClass() {}

      // returns the latest gps position and a flag
      Boolean gpsPosition(GPSdata&);

  private:

      // calculates the check sum of the message
      Boolean checkSumCheck(const GPSdata);

};

#endif
```

F. GPS.CPP

```
#include <math.h>
#include "gps.h"

// instantiates serial port communications on comm1, global to allow
// interrupt processing, cleanup to function properly
gpsPortClass port1;

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

   NAME:       gpsPosition
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Determines if an updated gps position message is available
               and copies it into the input argument 'rawMessage'. If the
              message has a valid checksum and was obtained with at least
             three satelites in view,  a 'TRUE' is returned to the caller,
               indicating that the message is valid.
   RETURNS:    TRUE, if a valid position message is contained in the
               input argument.
   CALLED BY:  navPosit (navigator.h)
   CALLS:      Get (buffer.h)
               checkSumCheck (gps.h)

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

Boolean gpsClass::gpsPosition(GPSdata& rawMessage)
{
   unsigned long Mask(4);
   if (port1.Get(rawMessage)) {
      // Check for a valid check sum and more the 3 satelites and DGPS
      return Boolean((checkSumCheck(rawMessage)) && (rawMessage[39] > 3)
                     && ((rawMessage[GPSBLOCKSIZE - 4] & Mask) == Mask));
   }
   else {
      return FALSE;              // No updated position is available.
   }
}
```

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

   PROGRAM:    checkSumCheck
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Takes an exclusive or of bytes 2 through 78 in a Motorola
               format (@@EA) position message and compares it to the
               checksum of the message.
   RETURNS:    TRUE, if the message contains a valid checksum
   CALLED BY:  gpsPosition (gps)
   CALLS:      none

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

Boolean gpsClass::checkSumCheck(const GPSdata newMessage)
{
   BYTE chkSum(0);

   for (int i = 2; i < GPSBLOCKSIZE - 3; i++) {
      chkSum ^= newMessage[i];
   }

   return Boolean(chkSum == newMessage[GPSBLOCKSIZE - 3]);
}
// end of file gps.cpp
```

G. INS.CFG

```
0.1     //Kone1
0.1     //Kone2
0.6     //Ktwo
0.5     //Kthree1
0.5     //Kthree2
0.5     //Kfour1
0.5     //Kfour2
1000    //tau
```

H. INS.H

```
#ifndef _INS_H
#define _INS_H

#include <time.h>
#include <math.h>
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <fstream.h>
#include <iostream.h>

#include "toetypes.h"
#include "globals.h"
#include "sampler.h"

/*********************************************************************
   CLASS:    insClass
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Takes in linear accelerations, angular rates, speed and
             heading information and uses Kalman filtering techniques to
             return a dead reconing position.

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

class insClass {

   public:

      insClass();                    // Constructor, initializes gains
      ~insClass() {}                 // destructor

      Boolean insPosition(stampedSample&);  // returns ins est. position

      // Updates the x, y and z of the vehicle posture
      void correctPosition(stampedSample&, double);

      // Sets posture to the origin and develops initial biases
      void insSetUp(double, stampedSample&);

   private:

     float posture[6];       // ins estimated posture (x y z phi theta psi)

     double velocities[6];  // ins estimated linear and angular velocities
                           // x-dot y-dot z-dot phi-dot theta-dot psi-dot

      float current[3];        // ins estimated error current
                               // (x-dot y-dot z-dot)
```

```
        float lastGPStime;       // time of last gps position fix
        int tau;                 // filter time constant

        samplerClass sam1;       // sampler instance

        matrix rotationMatrix;   // body to euler transformation matrix

        double biasCorrection[3]; // Software corrections,IMU rate sensors

        // Kalman filter gains.
        float Kone1, Kone2, Ktwo, Kthree1, Kthree2, Kfour1, Kfour2;

        // Transforms body coords to earth coords, removes gravity component
        void transformAccels (double[]);

        // Transforms water speed reading to x and y components
        void transformWaterSpeed (double, double[]);

        // Tranforms body euler rates to earth euler rates.
        void transformBodyRates (double[]);

        // Euler integrates the accelerations and updates the velocities
        void updateVelocities (stampedSample&);

        // Euler integrates the velocities and updates the posture
        void updatePosture (stampedSample&);

        // Builds the body to euler rate matrix
        matrix buildBodyRateMatrix();

        // Builds the body to earth rotation matrix
        void buildRotationMatrix();

        // Calculates the imu bias correction during set up
        void calculateBiasCorrections(stampedSample&);

        // Applies bias corrections to a sample
        void applyBiasCorrections(stampedSample&);

        // Reads filter constants from 'ins.cfg'
        void readInsConfigFile();
};

// Post multiply a matrix times a vector and return result.
vector operator* (matrix&, double[]);

#endif
```

I. INS.CPP

```
#include <iostream.h>
#include <signal.h>
#include "ins.h"
#define SIGFPE 8                    // Floating point exception

/*********************************************************************
   PROGRAM:    insClass (constructor)
   AUTHOR:     Eric Bachmann, Dave Gay, Rick Roberts
   DATE:       11 July 1995
   FUNCTION:   Constructor initializes kalman filter gains and linear and
               angular velocities.
   RETURNS:    nothing
   CALLED BY:  navigator class
   CALLS:      none

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

insClass::insClass() : Kone1(0.5), Kone2(0.5), Ktwo(0.6), Kthree1(0.5),
                       Kthree2(0.5), Kfour1(0.5), Kfour2(0.5), tau(1000)
{
   cerr << "\nconstructing ins1" << endl;

   readInsConfigFile();            // Read the config file

   velocities[0] = 0.0;            // ẋ dot
   velocities[1] = 0.0;            // y dot
   velocities[2] = 0.0;            // z dot
   velocities[3] = 0.0;            // phi dot
   velocities[4] = 0.0;            // theta dot
   velocities[5] = 0.0;            // psi dot

  posture[0] = 0.0;     // Set posture to straight and level at the origin.
   posture[1] = 0.0;
   posture[2] = 0.0;
   posture[3] = 0.0;
   posture[4] = 0.0;
   posture[5] = 0.0;

   current[0] = 0.0;      // Initialize error current to zero
   current[1] = 0.0;
   current[2] = 0.0;

}
```

```
/*********************************************************************
   PROGRAM: insPosit
   AUTHOR:  Eric Bachmann, Dave Gay
   DATE:    11 July 1995
   FUNCTION:  Make dead reckoning position estimation using kalman
              filtering. Inputs are linear accelerations, angular rates,
              speed and heading.  Primary input data is obtained from a
             sampler object via the getSample method. This data is stored
              in the sample filed of a stampedSample structure called
              newSample. The sample field is then used as a working
              variable as the linear accelerations and angular rates it
              contains are converted to earth coordinates and integrated
              to determine current velocities and posture. The data is
              complimentary filtered against itself, speed and magnetic
              heading.
   RETURNS:   position in grid coordinates as estimated by the INS
   CALLED BY:  navPosit (nav.cpp)
   CALLS:      getSample (sampler.cpp)
               findDeltaT (ins.cpp)
               transformBodyRates (ins.cpp)
               buildRotationMatrix (ins.cpp)
               transformAccels (ins)
               transformWaterSpeed (ins)

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

void fpeInsPosit(int sig)
{if (sig == SIGFPE) cerr << "floating point error in insPosit\n";}


Boolean insClass::insPosition(stampedSample& newSample)
{
   signal (SIGFPE, fpeInsPosit);

   double thetaA, phiA, xIncline, yIncline;       // Working variables
   double waterSpeedCorrection[3];          // Filter correction for drift
                                             // and water speed
   if (sam1.getSample(newSample)) {

      applyBiasCorrections(newSample);

      newSample.rawSample[0] = newSample.sample[0];
      newSample.rawSample[1] = newSample.sample[1];
      newSample.rawSample[2] = newSample.sample[2];
      newSample.rawSample[3] = newSample.sample[3];
      newSample.rawSample[4] = newSample.sample[4];
      newSample.rawSample[5] = newSample.sample[5];
      newSample.rawSample[6] = newSample.sample[6];
      newSample.rawSample[7] = newSample.sample[7];

      xIncline = newSample.sample[0] / GRAVITY;
```

```
yIncline = (newSample.sample[1] -
              (newSample.sample[5] * newSample.sample[6]))
                  / (GRAVITY * cos(posture[4]));

if (fabs(yIncline) > 1.0) {
   static int inclineCount(0);
   gotoxy(1,24);
   cerr << "Inclination errors: " << ++inclineCount << endl;
   return FALSE;
}

thetaA = asin(xIncline);          // Calculate low freq pitch and roll
phiA = -asin(yIncline);

// Transform body rates to euler rates.
transformBodyRates(newSample.sample);

// Calculate estimated roll rate (phi-dot).
velocities[3] = newSample.sample[3] + Kone1 * (phiA - posture[3]);
// Calculate estimated pitch rate (theta-dot).
velocities[4] = newSample.sample[4] + Kone2 * (thetaA - posture[4]);
// Calculate estimated heading rate (psi-dot).
velocities[5] =
   newSample.sample[5] + Ktwo * (newSample.sample[7] - posture[5]);

// integrate estimated angular rates to obtain angles
posture[3] += newSample.deltaT * velocities[3];// pitch rate to angle
posture[4] += newSample.deltaT * velocities[4];  // roll rate to angle
posture[5] += newSample.deltaT * velocities[5];  // yaw rate to angle

buildRotationMatrix();

// Transform accels to earth coordinates
transformAccels(newSample.sample);

// Transform water speed to earth coordinates
transformWaterSpeed(newSample.sample[6], waterSpeedCorrection);

// Subtract out previous velocity and apply statistical gain
waterSpeedCorrection[0] =
   Kthree1 * (waterSpeedCorrection[0] - velocities[0]);
waterSpeedCorrection[1] =
   Kthree2 * (waterSpeedCorrection[1] - velocities[1]);

// Determine filtered accelerations
newSample.sample[0] += waterSpeedCorrection[0];
newSample.sample[1] += waterSpeedCorrection[1];

// Integrate accelerations to obtain velocities
velocities[0] += newSample.sample[0] * newSample.deltaT;
velocities[1] += newSample.sample[1] * newSample.deltaT;
```

```
      velocities[2] += newSample.sample[2] * newSample.deltaT;

      // Integrate velocities to obtain posture
      posture[0] += (velocities[0] + current[0]) * newSample.deltaT;
      posture[1] += (velocities[1] + current[1]) * newSample.deltaT;
      posture[2] += velocities[2] * newSample.deltaT;

      newSample.sample[0] = posture[0];
      newSample.sample[1] = posture[1];
      newSample.sample[2] = posture[2];
      newSample.sample[3] = posture[3];
      newSample.sample[4] = posture[4];
      newSample.sample[5] = posture[5];

      newSample.est.x = posture[0];
      newSample.est.y = posture[1];
      newSample.est.z = posture[2];

      return TRUE;
   }
   else {
      return FALSE;          // New IMU information was unavailable.
   }
}


/*********************************************************************
   PROGRAM:  correctPosition
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Reinitializes the INS based on a known position and computes
             apparent current based on past accumulated errors of the INS.
              It is called by the navigator each time a new GPS (true) fix
              is obtained.
   RETURNS:        void
   CALLED BY: navPosit (nav)
   CALLS:     none

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

void
insClass::correctPosition(stampedSample& posit, double positTime)
{
   double deltaT;

  if (positTime < lastGPStime) {       // Correct for new day if necessary
      positTime += 86400;
   }

   deltaT = positTime - lastGPStime;     // Find time since last gps fix.
```

```
   // Determine INS error since last gps fix
   double deltaX = posit.est.x - posture[0];
   double deltaY = posit.est.y - posture[1];

   // Reinitialize posture to known position (gps fix)
   posture[0] = posit.est.x;
   posture[1] = posit.est.y;
  posture[2] = 0.0;                // Unit is assumed to be on the surface

   // Add gain filtered error to previous errors
   posit.current[0] = current[0] += Kfour1 * (deltaX / deltaT);
   posit.current[1] = current[1] += Kfour2 * (deltaY / deltaT);

   // Save the time of the gps fix for next calculation
   lastGPStime = positTime;
}


/*********************************************************************
   PROGRAM:  insSetUp
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Initializes the INS system. Sets the posture to the origin.
             Initializes the heading using magnetic compass information.
            Initializes the last GPS fix and last IMU information times.
   RETURNS:  void
   CALLED BY: initializeNavigator (nav)
   CALLS:     calculateBiasCorrections (ins)
              getSample (sampler)
              buildRotationMatrix (ins)
              transformWaterSpeed (ins)

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

void fpeInsSetUp(int sig)
{if (sig == SIGFPE) cerr << "floating point error in insSetUp\n";}

void insClass::insSetUp(double originTime, stampedSample& posit)
{
   cerr << "   Initializing INS." << endl;
   signal (SIGFPE, fpeInsSetUp);

   sam1.initSampler();                  // Initialize the sampler

   calculateBiasCorrections(posit);     // set imu biases

   posture[5] = posit.sample[7];        //set initial true heading

   buildRotationMatrix();               //set initial speed
   transformWaterSpeed(posit.sample[6], velocities);
```

```
   posit.current[0] = 0.0;
   posit.current[1] = 0.0;
   posit.current[2] = 0.0;

   lastGPStime = originTime;            // initialize times

   cerr << "   INS initialization complete." << endl;
}

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

   PROGRAM:  transformAccels
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Transforms linear accelerations from body coordinates to
             earth coordinates and removes the gravity component in the
             z direction.
   RETURNS:  void
   CALLED BY: navPosit
   CALLS:     none

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

void insClass::transformAccels (double newSample[])
{
   vector earthAccels;

   newSample[0] -= GRAVITY * sin(posture[4]);
   newSample[1] += GRAVITY * sin(posture[3]) * cos(posture[4]);
   newSample[2] += GRAVITY * cos(posture[3]) * cos(posture[4]);

   earthAccels = rotationMatrix * newSample;

   newSample[0] = earthAccels.element[0];
   newSample[1] = earthAccels.element[1];
   newSample[2] = earthAccels.element[2];
}


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

   PROGRAM:  transformWaterSpeed
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Transforms water speed into a vector in earth coordinates and
             returns them in the speedCorrection variable.
   RETURNS:   void
   CALLED BY: navPosit
   CALLS:     none
*********************************************************************/
```

```
void insClass::transformWaterSpeed (double waterSpeed, double
speedCorrection[])
{
   double water[3] = {waterSpeed, 0.0, 0.0};
   vector waterVelocities = rotationMatrix * water;

   speedCorrection [0] = waterVelocities.element[0];
   speedCorrection [1] = waterVelocities.element[1];
   speedCorrection [2] = waterVelocities.element[2];
}


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

   PROGRAM:     transformBodyRates
   AUTHOR:      Eric Bachmann, Dave Gay
   DATE:        11 July 1995
   FUNCTION:    Tranforms body euler rates to earth euler rates
   RETURNS:     none
   CALLED BY:   insPosit
   CALLS:       buildBodyRateMatrix

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

void insClass::transformBodyRates (double newSample[])
{
   vector earthRates = buildBodyRateMatrix() * &(newSample[3]);

   newSample[3] = earthRates.element[0];
   newSample[4] = earthRates.element[1];
   newSample[5] = earthRates.element[2];
}


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

   PROGRAM:    buildBodyRateMatrix
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Builds body to Euler rate translation matrix.
   RETURNS:    rate translation matrix
   CALLED BY:  insPosit
   CALLS:      none

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

matrix insClass::buildBodyRateMatrix()
{
   matrix rateTrans;

   float tth = tan(posture[4]),
```

```
        sphi = sin(posture[3]),
        cphi = cos(posture[3]),
         cth = cos(posture[4]);

   rateTrans.element[0][0] = 1.0;
   rateTrans.element[0][1] = tth * sphi;
   rateTrans.element[0][2] = tth * cphi;
   rateTrans.element[1][0] = 0.0;
   rateTrans.element[1][1] = cphi;
   rateTrans.element[1][2] = -sphi;
   rateTrans.element[2][0] = 0.0;
   rateTrans.element[2][1] = sphi / cth;
   rateTrans.element[2][2] = cphi / cth;

   return rateTrans;
}


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

   PROGRAM: buildRotationMatrix
   AUTHOR: Eric Bachmann, Dave Gay
   DATE:   11 July 1995
   FUNCTION:  Sets the body to earth coordinate rotation matrix.
   RETURNS:     void
   CALLED BY:   insPosit, insSetUp
   CALLS:       none

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

void insClass::buildRotationMatrix()
{
   float spsi = sin(posture[5]),
         cpsi = cos(posture[5]),
          sth = sin(posture[4]),
         sphi = sin(posture[3]),
         cphi = cos(posture[3]),
          cth = cos(posture[4]);

   rotationMatrix.element[0][0] = cpsi * cth;
   rotationMatrix.element[0][1] = (cpsi * sth * sphi) - (spsi * cphi);
   rotationMatrix.element[0][2] = (cpsi * sth * cphi) + (spsi * sphi);
   rotationMatrix.element[1][0] = spsi * cth;
   rotationMatrix.element[1][1] = (cpsi * cphi) + (spsi * sth * sphi);
   rotationMatrix.element[1][2] = (spsi * sth * cphi) - (cpsi * sphi);
   rotationMatrix.element[2][0] = -sth;
   rotationMatrix.element[2][1] = cth * sphi;
   rotationMatrix.element[2][2] = cth * cphi;
}
```

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

   PROGRAM:   postmultiplication operator *
   AUTHOR:    Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Post multiply a 3 X 3 matrix times a 3 X 1 vector and
              return the result.
   RETURNS:   3 X 1 vector
   CALLED BY:
   CALLS:     None

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

vector operator* (matrix& transform, double state[])
{
   vector result;

   for (int i = 0; i < 3; i++) {

      result.element[i] = 0.0;

      for (int j = 0; j < 3; j++) {

 result.element[i] += transform.element[i][j] * state[j];
      }
   }
   return result;
}


/*********************************************************************
   PROGRAM:   calculateBiasCorrections
   AUTHOR:    Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995
   FUNCTION:  Calculates the initial imu bias by averaging a number of
              imu readings.
   RETURNS:   none
   CALLED BY: insSetup
   CALLS:     none

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

void fpeCalculateBiasCorrections(int sig)
{if (sig == SIGFPE) cerr << "floating point error in
CalculateBiasCorrections\n";}


void insClass::calculateBiasCorrections(stampedSample& biasSample)
{
   signal (SIGFPE, fpeCalculateBiasCorrections);
```

```
   int biasNumber(tau/10);

   biasCorrection[0] = 0.0;                    // p roll rate
   biasCorrection[1] = 0.0;                    // q pitch rate
   biasCorrection[2] = 0.0;                    // r yaw rate

   for (int i = 0; i < biasNumber; i++) {

      // Find the average of the biasNumber packets
      while(!sam1.getSample(biasSample)) {/* */};
     biasCorrection[0] += biasSample.sample[3]/biasNumber; // roll-rate/
b#
    biasCorrection[1] += biasSample.sample[4]/biasNumber; // pitch-rate/
b#
    biasCorrection[2] += biasSample.sample[5]/biasNumber; // yaw-rate/b#
   }

   // set biasSample correction fields to new bias correction values
  // negative biasCorrection value is taken so biases are added to sensor
values
   biasSample.bias[3] = biasCorrection[0] = -(biasCorrection[0]);
   biasSample.bias[4] = biasCorrection[1] = -(biasCorrection[1]);
   biasSample.bias[5] = biasCorrection[2] = -(biasCorrection[2]);
}


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

   PROGRAM:  applyBiasCorrections
   AUTHOR:   Eric Bachmann, Dave Gay, Rick Roberts
   DATE:     11 July 1995
   FUNCTION: Applies updated bias corrections to a sample.
   RETURNS:  void
   CALLED BY: insPosit
   CALLS:     none

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

void insClass::applyBiasCorrections(stampedSample& posit)
{
   const float sampleWght(posit.deltaT/tau);
   const float biasWght(1 - sampleWght);

   //Calculate updated bias values
   biasCorrection[0] = (biasWght * biasCorrection[0])
                        - (sampleWght * posit.sample[3]);
   biasCorrection[1] = (biasWght * biasCorrection[1])
                        - (sampleWght * posit.sample[4]);
   biasCorrection[2] = (biasWght * biasCorrection[2])
                        - (sampleWght * posit.sample[5]);
```

```
  posit.sample[3] += biasCorrection[0];     //Apply the bias to the sample
   posit.sample[4] += biasCorrection[1];
   posit.sample[5] += biasCorrection[2];

  posit.bias[3] = biasCorrection[0];        //Save the bias to the sample
   posit.bias[4] = biasCorrection[1];
   posit.bias[5] = biasCorrection[2];
}

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

   PROGRAM:     readInsConfigFile
   AUTHOR:      Rick Roberts, Eric Bachmann
   DATE:        02 Nov 96
   FUNCTION:    Reads filter constants from 'ins.cfg'
   RETURNS:     void
   CALLED BY:   ins class constructor
   CALLS:       none

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

void insClass::readInsConfigFile()
{
   cerr << "Reading ins configuration file." << endl;

   ifstream insCfgFile("ins.cfg", ios::in);

   if(!insCfgFile) {
      cerr << "could not open ins configuration file!" << endl;
   }

   else {

      char comment[128];

      insCfgFile
                >> Kone1 >> comment
                >> Kone2 >> comment
                >> Ktwo >> comment
                >> Kthree1 >> comment
                >> Kthree2 >> comment
                >> Kfour1 >> comment
                >> Kfour2 >> comment
                >> tau >> comment;
   }
   insCfgFile.close();
}
// end of file ins.cpp
```

J. SAM.CFG

```
1.0     ;pScale(roll)
1.0     ;qScale(pitch)
1.0     ;rScale(yaw)
1.34    ;xAccelScale(pitch)
1.34    ;yAccelScale(roll)
1.34    ;zAccelScale(yaw)
1.827   ;waterSpeedScale
```

K. SAMPLER.H

```
#ifndef _SAMPLER_H
#define _SAMPLER_H

#include <time.h>
#include <math.h>
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <fstream.h>
#include <iostream.h>

#include "toetypes.h"
#include "globals.h"
#include "a2d.h"
#include "compass.h"

#define MAX_SAMPLE_NUM 1000
#define xyAccelLimit ONE_G          // Max accell in x and y direction
#define zAccelLimit 2 * ONE_G        // Max accel in z direction
#define rateLimit  0.872665         // Max rotational rate in radians
#define speedLimit 25.3              // Max water speed
#define headingLimit 2 * M_PI

const int INBUFFSIZE = 512;

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

   CLASS:     samplerClass
   AUTHOR:    Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995, last modified January 1997
   FUNCTION:  Formats, timestamps, low pass filters and limit checks IMU,
              water-speed and heading information.
   COMMENTS:  This class is extremely dependent upon the specific
              hardware configuration. It is designed to isolate the
              INS from these particulars.

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

```
class samplerClass {

   public:

      samplerClass();                    // Class constructor, destructor
      ~samplerClass() {}

      Boolean initSampler();             // Initializes Sampler

      // checks for the arrival of a new sample and formats it
      Boolean getSample(stampedSample&);

   private:

      float pScale;                      // roll
      float qScale;                      // pitch
      float rScale;                      // yaw

      float xAccelScale;                 // pitch
      float yAccelScale;                 // roll
      float zAccelScale;                 // yaw

      float waterSpeedScale;

      compassClass comp1;               // instantiate member compass object

      a2dClass a2d1;                     // instantiate member a2d object

      // stores incoming FIFO samples by channel
      float sample[MAX_SAMPLE_NUM][8];

      int subSampleIndex;                // counts channels

      int sampleIndex;                   // indexes samples' array

      int sampleCount;                   // counts samples

      float samplePeriod;

      Boolean readSamples(stampedSample& newSample);

      void filterSample(stampedSample& newSample);

      void formatSample(stampedSample& newSample);

      void increment(int& index)
         { if (++index == MAX_SAMPLE_NUM) index = 0;}

      void decrement(int& index)
         { if (--index < 0) index = MAX_SAMPLE_NUM - 1;}
```

```
        // Reads filter constants from 'sam.cfg'
        void readSamplerConfigFile();

        double pUnits(double angular)
        { return
          (pScale * (((angular-2047.0) / 2047.0 ) * 50.0) * (M_PI/180.0));}

        double qUnits(double angular)
        { return
          (qScale * (((angular-2047.0) / 2047.0 ) * 50.0) * (M_PI/180.0));}

        double rUnits(double angular)
        { return
          (rScale * (((angular-2047.0) / 2047.0 ) * 50.0) * (M_PI/180.0));}

        double xAccelUnits(double linear)
        { return (xAccelScale * ((linear-2047.0) / 2047.0 ) * GRAVITY);}

        double yAccelUnits(double linear)
        { return (yAccelScale * ((linear-2047.0) / 2047.0 ) * GRAVITY);}

        double zAccelUnits(double linear)
        { return
           (zAccelScale * ((linear-2047.0) / 2047.0) * (2.0 * GRAVITY));}

        double depthUnits(double depth)
        { return (((depth - 819.0) / (4095.0-819.0)) * 180.0);}

        double waterSpeedUnits(double speed) //feet per second
        { return (waterSpeedScale * ((speed - 2047.0) / 2048.0) * 25.3);}
};
#endif
```

L. SAMPLER.CPP

```
#include "sampler.h"

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

   PROGRAM:  samplerClass Constructor
   AUTHOR:   Eric Bachmann, Randy Walker, Rick Roberts
   DATE:     12 May 1995, last modified December 1996
   FUNCTION: Constructs sam1, initializes default config values, calls
             readSamplerConfigFile to read any updated values.
   RETURNS:  sam1
   CALLED BY: insSetUp (ins.cpp)
   CALLS:     readSamplerConfigFile

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

samplerClass::samplerClass()
                 : sampleIndex(0), subSampleIndex(0),
                   samplePeriod(a2d1.chcnt * a2d1.delta_t * 0.000001),
                   pScale(0.0), qScale(0.0), rScale(0.0),
                   xAccelScale(0.0), yAccelScale(0.0), zAccelScale(0.0),
                   waterSpeedScale(0.0)
{
   cerr << "\nconstructing sampler w/ a2d1, comp1" << endl;
   readSamplerConfigFile();
}

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

   PROGRAM:  initSampler
   AUTHOR:   Eric Bachmann, Randy Walker, Rick Roberts
   DATE:     12 May 1995
   FUNCTION: Instantiates the compass A2D objects.
   RETURNS:  TRUE
   CALLED BY: insSetUp (ins.cpp)
   CALLS:     initCompass(), A2D member functions

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

Boolean samplerClass::initSampler()
{
   cerr << "      Initializing Sampler" << endl;

   comp1.initCompass();

   cerr << "        Initializing A2D." << endl;

   a2d1.initA2d();
```

```
   cerr << "         A2D initialization complete." << endl;

   cerr << "       Sampler initialization complete." << endl;

   return TRUE;
}

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

   PROGRAM:  getSample
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Prepares raw sample data for use by the INS  object
   RETURNS:  TRUE, if a valid sample was obtained
   CALLED BY: insPosit (ins)
              insSetup (ins)
   CALLS:     readSamples (sampler)
              filterSample (sampler)
              formatSample (sampler)

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

Boolean samplerClass::getSample(stampedSample& newSample)
{
  if (readSamples(newSample)) {  // checks for the arrival of a new sample

      filterSample(newSample);

      formatSample(newSample);

      return TRUE;
   }

   return FALSE;                  // Sample packet not available
}

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

   PROGRAM:  readSamples
   AUTHOR:   Eric Bachmann, Randy Walker
   DATE:     12 May 1996
   FUNCTION: Retrieves all samples of the IMU, water speed, and depth
             that are present in the A2D FIFO until the FIFO is EMPTY.
             Calculates delta_t.
   RETURNS:  TRUE - There were new samples pulled from the FIFO
             FALSE - There were no new samples
   CALLED BY: getSample
   CALLS:     getFifoStatus(), getFifoData()

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

```
Boolean samplerClass::readSamples(stampedSample& newSample)
{
   static int overflowCount(0);
   if (a2d1.getFifoStatus() == FULL) {      // Did the FIFO overflow?
      gotoxy(1,19);
      cout << "FIFO Overflowed, #: " << ++overflowCount
           << " reiniting a2d" << endl;
      a2d1.reinitA2d();
      return FALSE;
   }

  if (a2d1.getFifoStatus() != EMPTY) {   // Does the FIFO have new samples?

     sampleCount = 0;                  // Counts the number of samples taken

      while (a2d1.getFifoStatus() != EMPTY) {  // Empty the FIFO

         sample[sampleIndex][subSampleIndex++] = a2d1.getFifoData();

         // Has it pulled one sample of each channel from the FIFO?
         if (subSampleIndex == 8) {
            subSampleIndex= 0;
            increment(sampleIndex);       // set to record next sample
            ++sampleCount;
         }
      }

      if (sampleCount > 0) {
         // calculate time delta
         newSample.deltaT = sampleCount * samplePeriod;
         return TRUE;
      }
      else {                              // No full samples
         return FALSE;
      }
   }

   else {                                 // No new samples
      return FALSE;
   }
}
```

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

   PROGRAM:   filterSample
   AUTHOR:    Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Low pass filters eight closely spaced sets of sensor
              readings by summing the readings of each sensor and computing
              the average.
   RETURNS:   void
   CALLED BY: getSample
   CALLS:     none

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

void samplerClass::filterSample(stampedSample& newSample)
{
   for (int i = 0; i < 8; i++) {
      newSample.sample[i] = 0;
   }

   int j(sampleIndex);

   for (i = 0; i < sampleCount; i++) {

      decrement(j);
      newSample.sample[0] += sample[j][0] / sampleCount;
      newSample.sample[1] += sample[j][1] / sampleCount;
      newSample.sample[2] += sample[j][2] / sampleCount;
      newSample.sample[3] += sample[j][3] / sampleCount;
      newSample.sample[4] += sample[j][4] / sampleCount;
      newSample.sample[5] += sample[j][5] / sampleCount;
      newSample.sample[6] += sample[j][6] / sampleCount;
      newSample.sample[7] += sample[j][7] / sampleCount;
   }
}


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

   PROGRAM:  formatSample
   AUTHOR:   Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Converts integers representing voltage readings into
             real world units which are useable by the INS.
   RETURNS:  void
   CALLED BY:getSample
   CALLS:    none

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

```
void samplerClass::formatSample (stampedSample& newSample)
{
   newSample.sample[0] = xAccelUnits(newSample.sample[0]);
   newSample.sample[1] = yAccelUnits(newSample.sample[1]);
   newSample.sample[2] = zAccelUnits(newSample.sample[2]);

   newSample.sample[3] = pUnits(newSample.sample[3]);
   newSample.sample[4] = qUnits(newSample.sample[4]);
   newSample.sample[5] = rUnits(newSample.sample[5]);

   newSample.sample[6] = waterSpeedUnits(newSample.sample[6]);
   newSample.sample[7] = comp1.getHeading();
}


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

   PROGRAM:     readSamplerConfigFile
   AUTHOR:      Rick Roberts, Eric Bachmann
   DATE:        02 Nov 96
   FUNCTION:    Reads filter constants from 'ins.cfg'
   RETURNS:     void
   CALLED BY:   ins class constructor
   CALLS:       none
   COMMENTS:    * Do not allow blanks in 'comment' section of sam.cfg *

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

void samplerClass::readSamplerConfigFile()
{
   FILE *samCfgFile;

   if ((samCfgFile = fopen("sam.cfg", "r")) == NULL){
      cerr << "could not open sampler configuration file!" << endl;
   }
   else {

      cerr << "\nReading Sampler configuration file." << endl;

      char line[128];

      fscanf(samCfgFile,"%f%s",&pScale,line);
      cerr << "pScale: " << pScale << endl;

      fscanf(samCfgFile,"%f%s",&qScale,line);
      cerr << "qScale: " << qScale << endl;

      fscanf(samCfgFile,"%f%s",&rScale,line);
      cerr << "rScale: " << rScale << endl;

      fscanf(samCfgFile,"%f%s",&xAccelScale,line);
```

```
      cerr << "xAccelScale: " << xAccelScale << endl;

      fscanf(samCfgFile,"%f%s",&yAccelScale,line);
      cerr << "yAccelScale: " << yAccelScale << endl;

      fscanf(samCfgFile,"%f%s",&zAccelScale,line);
      cerr << "zAccelScale: " << zAccelScale << endl;

      fscanf(samCfgFile,"%f%s",&waterSpeedScale,line);
      cerr << "waterSpeedScale: " << waterSpeedScale << endl;

   }
   fclose(samCfgFile);
}
// end of file sampler.cpp
```

M. COMPASS.H

```
#ifndef _COMPASS_H
#define _COMPASS_H

#include <iostream.h>
#include <fstream.h>
#include <conio.h>

#include "toetypes.h"
#include "globals.h"
#include "compport.h"

BYTE asciiToHex(BYTE);           // conversion function prototype

/*********************************************************************
   CLASS:    compassClass
   AUTHOR:   Eric Bachmann, Dave Gay, Rick Roberts
   DATE:     11 July 1995, last modified January 1997
   FUNCTION: Reads compass messages from the compass buffer.  Checks for
             valid checksum. Corrects heading for magnetic variation.
            Heading is continuous. There is no branch cut at 360 degrees.

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

class compassClass {

  public:

     // class constructor and destructor

     compassClass() : currentHeading(0.0)
        { cerr << "Compass constructed." << endl; }
```

```
    ~compassClass() {}

    float initCompass();                // initialize currentHeading

    float getHeading();                 // returns the latest heading

  private:

    // Maintains the most recently obtained heading.
    float currentHeading;

    // calculates the check sum of the message
    Boolean checkSumCheck(const compData);

    // Parses a selected field out of a compass message.
    float parseCompData(const compData, const BYTE);

    // Converts magnetic direction based on magnetic variation.
    float trueHeading(const float);

    // Returns the heading without branch cuts
    float continousHeading(const float);
};
#endif
```

N. COMPASS.CPP

```
#include <math.h>
#include <stdlib.h>
#include "compass.h"

// instantiates serial port communications on comm2, global to allow
// interrupt processing, cleanup to function correctly
compassPortClass port2;

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

   NAME:       initCompass
   AUTHOR:     Eric Bachmann, Dave Gay, Rick Roberts
   DATE:       11 July 1995
   FUNCTION:   Determines if a valid compass message is held in the
              compass buffer and initializes currentHeading to that value.
               Will attempt 10 times with a built in delay and then exit
               with a warning if a valid heading is not obtained.
   RETURNS:    currentHeading
   CALLED BY:  INSsetUp (ins.cpp)
   CALLS:      Get (buffer.h)              parseCompData (compass.cpp)
              checkSumCheck (gps.h)       continuousHeading (compass.cpp)
               trueHeading (compass.cpp)
*********************************************************************/
```

```
float compassClass::initCompass()
{
   cerr << "          Initializing Compass" << endl;

   Boolean compFlag(FALSE);
   float tempHeading;
   compData rawMessage;

   // try 10 times to get a valid message
   for (int i = 1 ; ((i < 10) && (compFlag == FALSE)); i++ ) {

    if ((port2.headings.Get(rawMessage)) && (checkSumCheck(rawMessage))){
        tempHeading = parseCompData(rawMessage, 'C') * degToRad;
        currentHeading = continousHeading(trueHeading(tempHeading));
        compFlag = TRUE;
     }
     else {                              // invalid message - delay
        delay(1000);
     }
   }

   if (compFlag == FALSE) {
      cerr << "\nWARNING: UNABLE TO OBTAIN INITIAL COMPASS HEADING!"
           << endl;
      delay(2000);
   }
   else {
      cerr << "          Compass initialization complete." << endl;
   }
   return currentHeading;
}


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

   NAME:       getHeading
   AUTHOR:     Eric Bachmann, Dave Gay, Rick Roberts
   DATE:       11 July 1995
   FUNCTION:   Determines if an updated compass message is available and
               copies it into the input argument 'rawMessage'. If the
               message has a valid checksum, currentHeading is returned
               to the caller, currentHeading is also the default return.
   RETURNS:    currentHeading
   CALLED BY:  navPosit (navigator.h)
   CALLS:      Get (buffer.h)
               checkSumCheck (compass.cpp)

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

float compassClass::getHeading()
{
```

```
   float tempHeading;
   Boolean checkSumFlag;
   compData rawMessage;

   if ((port2.headings.Get(rawMessage)) && (checkSumCheck(rawMessage))) {

      tempHeading = parseCompData(rawMessage, 'C') * degToRad;
      currentHeading = continousHeading(trueHeading(tempHeading));

      return currentHeading;
   }
   else {
       return currentHeading;    // No updated position is available.
   }
}


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

   NAME:       asciiToHex
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Administrative conversion function
   RETURNS:    Hex version of an ascii character
   CALLED BY:  checkSumCheck
   CALLS:      None

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

BYTE asciiToHex(BYTE letter)
{
   if (letter >= 'A') {
      return (letter - 'A' + 10);
   }
   else {
      return (letter - 48);
   }
}


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

   PROGRAM:    checkSumCheck
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Calculates the checksum of the compass message and
               compares it to the indicated checksum of the message.
   RETURNS:    TRUE, if the message contains a valid checksum
   CALLED BY:  initCompass, getHeading
   CALLS:      none
*********************************************************************/
```

```
Boolean compassClass::checkSumCheck(const compData newMessage)
{
   BYTE calChkSum(0);
   BYTE mesChkSum(0);

   for (int i = 1; newMessage[i] != '*'; i++) {

      calChkSum ^= newMessage[i];
   }

   mesChkSum = asciiToHex(newMessage[i+1]) * 16
                  + asciiToHex(newMessage[i+2]);

   return Boolean(calChkSum == mesChkSum);
}


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

   PROGRAM:    trueHeading
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Converts magnetic direction to true based on local
               magnetic variation.
   RETURNS:    true heading
   CALLED BY:  insPosit
               insSetUp
   CALLS:      none

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

float compassClass::trueHeading(const float magHeading)
{
   static double twoPi(2.0 * M_PI);
   double trueHeading = magHeading + RADIANMAGVAR;

   if (trueHeading > twoPi) {

      trueHeading -= twoPi;
   }

   return trueHeading;
}
```

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

   PROGRAM:    continousHeading
   AUTHOR:     Eric Bachmann
   DATE:       11 July 1995
  FUNCTION:  Maintains track of branch cuts & returns a continous heading.
   RETURNS:    continous true heading
   CALLED BY:  insPosit, insSetUp
   CALLS:      none

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

float compassClass::continousHeading(const float trueHeading)
{
   const float twoPi(2.0 * M_PI);
   static int branchCutCount(0);
   static float previousHeading(trueHeading);

   if ((4.71 < previousHeading) && (trueHeading < 1.57)){
     ++branchCutCount;          // Went through North in a right hand turn
   }
   else {
      if ((1.57 > previousHeading) && (trueHeading > 4.71)) {
        --branchCutCount;       // Went through North in a left hand turn
      }
   }

   previousHeading = trueHeading;

   return trueHeading + (branchCutCount * twoPi);
}


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

   PROGRAM:    parseCompData
   AUTHOR:     Eric Bachmann
   DATE:       11 July 1995
   FUNCTION:   Parses the heading out of a compass message.
   RETURNS:    the message heading as a float
   CALLED BY:  insPosit, insSetUp
   CALLS:      none

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

float compassClass::parseCompData(const compData rawMessage,
                                  const BYTE key)
{
   float dataSum(0);

   for(int j = 0; rawMessage[j] != key; j++){}
```

```
    j++;

    for(int i = 0; rawMessage[i + j] != '.'; i++){}

    switch (i) {

       case 3:

          dataSum = (rawMessage[j] - 48) * 100.0 +
                     (rawMessage[j+1] - 48) * 10.0 +
                    (rawMessage[j+2] - 48) + (rawMessage[j+4] - 48) * 0.1;
          break;

       case 2:

          dataSum = (rawMessage[j] - 48) * 10.0 +
                    (rawMessage[j+1] - 48) + (rawMessage[j+3] - 48) * 0.1;
          break;

       case 1:

          dataSum = (rawMessage[j] - 48) + (rawMessage[j+2] - 48) * 0.1;

          break;
    }

    return dataSum;
}
// end of file compass.cpp
```

O. A2D.CFG

```
8       ;seqcnt:number_of_seq_addresses_to_load
0       ;mode_sel:__DIFF=1__SE=0
1       ;mode_acdc:_Signal_coupling_select__DC=1__AC=0
8       ;chcnt:_____Number_of_channels_to_sequence_(hex,_1-F)
3125    ;delta_t:__Sample_rate_in_microsecs_3-8192
7       ;samprate:__Sample_rate_in_recurrent_mode__0(fast)-7(slow)
0       ;sampindex:_Which_channel_to_sample_in_recurrent_mode
0       0       0       0
1       1       0       0
2       2       0       0
3       3       0       0
4       4       0       0
5       5       0       0
6       6       0       0
7       7       0       0
8       8       0       0
9       A       2       0
```

```
A       5       2       0
B       A       2       0
C       5       2       0
D       A       2       0
E       5       2       0
F       A       2       0
```

P. A2D.H

```
#ifndef _A2D_H
#define _A2D_H

#include <dos.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <iostream.h>
#include <fstream.h>

//ESP A2D General Global Definitions
#define DEFBASE  0x100       // Base address SEL=1->0x300 & SEL=0->0x100
#define FIFOSIZE 1000        // FIFO size (MAX=1000 decimal)
#define MAXCHAN  0x10        // Max channels

//ESP A2D Status Register Definitions
//BASE+02h: 011D DDDD
#define INT_STAT 0x10     // 0001 0000 INTERRUPT STATUS   (1=IRQ Pending)
#define TRG_STAT 0x08     // 0000 1000 TRIGGER STATUS     (1=Triggered)

#define FULL     0x01      // 0000 0001 FIFO FULL        (001=Full)
#define HALF     0x05      // 0000 0101 FIFO HALF FULL   (101=Half Full)
#define EMPTY    0x06      // 0000 0110 FIFO EMPTY       (110=Empty)

//ESP A2D Control Register Definitions
//BASE+08h: DDDD DDDD
//BASE+09h: DDDD DDRR
#define GATE1OUT 0x0008   // 0000 0000 0000 1000 GATE1OUT (Always Driven)

#define TRG_POS  0x0010   // 0000 0000 0001 0000 TRIG POS (Trig on +|-)
#define SET_TRG  0x0020   // 0000 0000 0010 0000 TRIG SET (Active  LOW)
#define RST_TRG  0x0040   // 0000 0000 0100 0000 TRIG CLR (Active  LOW)
#define INT_EN   0x0080   // 0000 0000 1000 0000 IRQ ENAB (Active HIGH)

#define DIFF       0x0400  // 0000 0100 0000 0000 DIFF/SE  (1=DIFF 0=SE)
#define RMS        0x0800  // 0000 1000 0000 0000 RMS Mode (1=ON   0=OFF)

#define CAL        0x1000  // 0001 0000 0000 0000 CAL  Mode (1=ON   0=OFF)
#define PRG_SEQ    0x1000  // 0001 0000 0000 0000 SEQ  Mode (1=PRG  0=RUN)
```

```
#define ACDC        0x2000  // 0010 0000 0000 0000 ACDC Mode (1=DC   0=AC)
#define SAM_SEQ     0x4000  // 0100 0000 0000 0000 SAMP/SEQ  (1=SEQ  0=SAMP)
#define RST_FIFO    0x8000  // 1000 0000 0000 0000 FIFO Reset(1=EN   0=REW)

//ESP A2D Useful Definitions
#define CLRRATE  0xFFF8 // CLEAR RATE TO HIGHEST RATE

//Class Definition for the A2D Class
class a2dClass {

   public:

   a2dClass();                // reads a2d.cfg file, initializes hardware
   ~a2dClass() { lockTrigger(); }

   void readConfigFile();     // reads a2d.cfg file

   void initA2d();            // initializes the a2d
   void reinitA2d();         // reinitializes the a2d after FIFO overflow

   void initSysAddr(void);    // sets address mapping

   void initHardware(void);   // initializes the a2d control register

   // Print out the variable ctrlw, for debug purposes
   void printCtrlw(void);

   // Sets the A2D Control Register for Single-Ended mode
   void setSe(void);

   // Sets the A2D Control Register for Differential mode
   void setDiff(void);

   // Loads sequencer memory with channel data
   void setChannel(unsigned seq,unsigned ch,unsigned g10,unsigned g2);

   // Sets sequencer to program mode
   void setProgSeq(void);

   // Sets sequencer to run mode
   void setRunSeq(void);

   // Loads sequencer address counter with number of channels to scan.
   void setCount(unsigned nch);

   void setAcDc(unsigned acdc);   // sets AC or DC coupling

   void lockTrigger(void);        // prevents triggering

   void unlockTrigger(void);      // allows the trigger to function
```

```
// Toggle the trigger (software triggering)
void setTrigger(void);

void resetTrigger(void);        // clears the trigger

// Switches in the RMS measurement chip
void setRmsOn(void);

// Switches out RMS measurement chip
void setRmsOff(void);

// Sets the A2D module to sequencer mode
void setSequencer(void);

// Sets the A2D module to sampler mode
void setSamplerRate(unsigned);

// Set GATE1OUT bit of control word high
void gate1outOn(void);

// Set GATE1OUT bit of control word low
void gate1outOff(void);

// Sets timer channel 1 to square-wave input
void squareWaveTimer1(unsigned);

// Initialize the A2D timing using timer 2
void initTiming(unsigned dt);

void resetFifo(void);           // rewind FIFO to beginning of memory

void setFifo(void);             // enable FIFO to acquire data

unsigned getFifoStatus(void);   // returns the state of the FIFO

// Returns next data word stored in FIFO
signed getFifoData(void);

// Program timer channel 0 to set the desired interrupt rate
void setIntRate(unsigned intrate);

void intOff(void);              // locks out the interrupt request line

void intOn(void);               // enables system interrupt request

// Sets the trigger level; trigger level (0=-10V, 128=0V, 255=+10V)
void setTriggerLevel(unsigned tl);

// Sets falling or rising edge trigger
void setTriggerPosition(unsigned tp);
```

```
    void zeroOffset(void);          // calibrates zero offset error

    // Grounds the two differential inputs for zero adjust
    void grndInput(void);

    void freeInput(void);          // ungrounds the two differential inputs

    void zeroAdjust(void);          // adjust the trimmer on the PGA

    int chcnt;                      // Number of channels to sequence
    unsigned delta_t;               // period between channels

    private:

    unsigned ctrlw;             // Holds A2D Control Register update values
    unsigned seqcnt;              // Sequence Counter
    unsigned mode_sel;            // Single-ended or Differential
    unsigned mode_acdc;           // AC/DC Coupling
    unsigned samprate;            // Sample Rate in Recurrent Mode
    unsigned sampindex;         // Which Channel to Sample in Recurrent Mode
    unsigned seqaddr[MAXCHAN];  // Sequencer Address
    unsigned chan[MAXCHAN];     // Channel
    unsigned g10[MAXCHAN];      // x10 Gain
    unsigned g2[MAXCHAN];       // x2 Gain
};
#endif
```

Q. A2D.CPP

```
#include "a2d.h"

//ESP A2D Addresses
unsigned BASE    = DEFBASE;     // BASE I/O  ADDR        [BASE]   ()
unsigned FIFO    = 0x00;        // FIFO READ ADDR        [00-01]  (R)
unsigned MEM     = 0x00;        // SEQUENCER ADDR        [00-01]  (W)
unsigned STAT    = 0x02;        // STATUS REGISTER       [02]     (R)
unsigned COUNT   = 0x02;        // SEQUENCER ADDR PTR    [02]     (W)
unsigned TIMER0  = 0x04;        // TIMER 0               [04]     (R/W)
unsigned TIMER1  = 0x05;        // TIMER 1               [05]     (R/W)
unsigned TIMER2  = 0x06;        // TIMER 2               [06]     (R/W)
unsigned TIMERC  = 0x07;        // TIMER CONTROL WORD    [07]     (R/W)
unsigned CNTL    = 0x08;        // A2D CONTROL REGISTER  [08-09]  (W)
unsigned DAC     = 0x0C;        // DAC DATA              [0C]     (W)
```

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

// FUNCTION NAME: a2dClass()
// AUTHOR: Randy Walker
// DATE: 27 March 1996
// DESCRIPTION: Reads a2d.cfg file, initializes address map and hardware
// RETURNS: void
// CALLS: readConfigFile(), initSysAddr(), initHardware()
// CALLED BY: Object declaration
//
*********************************************************************

a2dClass::a2dClass(void)
{
   cerr << "constructing a2d1" << endl;

   ctrlw=0;
   seqcnt=1;
   mode_sel=0;
   mode_acdc=1;
   delta_t=3;
   chcnt=1;
   samprate=0;
   sampindex=0;
   readConfigFile();
   initSysAddr();
   initHardware();
}


//
*********************************************************************
// FUNCTION NAME: readConfigFile()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Reads the a2d.cfg file and sets variables
// RETURNS: void
// CALLS: none
// CALLED BY: a2d class constructor
//
*********************************************************************

void a2dClass::readConfigFile()
{
   FILE *configFile;
   char junk[128];

   if ((configFile = fopen("a2d.cfg", "r")) == NULL){
      fprintf(stderr, "Cannot open file A2D.CFG...\n");
      exit(1);
   }
```

```
    fscanf(configFile,"%x%s",&seqcnt,junk);
   if (seqcnt==0 || seqcnt>0x0F){ // seqcnt must be 1-F (15 max in seq mode)
      cout << "\nseqcnt out of range in A2D.CFG...\n";
      exit(1);
    }

    fscanf(configFile,"%d%s",&mode_sel,junk);
    if (mode_sel !=0 && mode_sel != 1){
      cout << "\nmode_sel out of range in A2D.CFG...\n";
      exit(1);
    }

    fscanf(configFile,"%d%s",&mode_acdc,junk);
    if (mode_acdc !=0 && mode_acdc != 1){
      cout << "\nmode_acdc out of range in A2D.CFG...\n";
      exit(1);
    }

    fscanf(configFile,"%x%s",&chcnt,junk);
   if (chcnt == 0 || chcnt > 0x0F){  //chcnt must be 1-F (15 max in seq mode)
      cout << "\nchcnt out of range in A2D.CFG...\n";
      exit(1);
    }

    fscanf(configFile,"%d%s",&delta_t,junk);
    if (delta_t < 3 || delta_t > 8192){
      cout << "\ndelta_t out of range in A2D.CFG...\n";
      exit(1);
    }

    if (delta_t < 6 && chcnt > 1){
      cout << "\ndelta_t must be > 6 for chcnt > 1...\n";
      exit(1);
    }

    fscanf(configFile,"%d%s",&samprate,junk);
    if (samprate > 7){
      cout << "\nsamprate out of range in A2D.CFG...\n";
      exit(1);
    }

    fscanf(configFile,"%x%s",&sampindex,junk);
    if (sampindex > 0x0F){
      cout << "\nsampindex out of range in A2D.CFG...\n";
      exit(1);
    }
    for (int i = 0; i < seqcnt; i++){
      fscanf(configFile,"%x%x%x%x",&seqaddr[i],&chan[i],&g10[i],&g2[i]);
    }
    fclose(configFile);
}
```

```
//*********************************************************************
// FUNCTION NAME: initSysAddr()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets system address mappings
// RETURNS: void
// CALLS: none
// CALLED BY: a2d class constructor
//*********************************************************************

void a2dClass::initSysAddr(void)
{
   //clear BASE
   FIFO    &= 0x0F;        // FIFO READ ADDRESS    [00,01] (R)
   MEM     &= 0x0F;        // SEQENCER MEM ADDRESS [00,01] (W)
   STAT    &= 0x0F;        // STATUS REGISTER      [02]    (R)
   COUNT   &= 0x0F;        // SEQENCER ADDRESS PTR [02]    (W)
   TIMER0  &= 0x0F;        // TIMER 0              [04]    (R/W)
   TIMER1  &= 0x0F;        // TIMER 1              [05]    (R/W)
   TIMER2  &= 0x0F;        // TIMER 2              [06]    (R/W)
   TIMERC  &= 0x0F;        // TIMER CONTROL WORD   [07]    (R/W)
   CNTL    &= 0x0F;        // CONTROL REGISTER     [08]    (R/W)
   DAC     &= 0x0F;        // DAC DATA             [0C]    (W)

   //set BASE
   FIFO    |= BASE;        // FIFO READ ADDRESS    [00,01] (R)
   MEM     |= BASE;        // SEQENCER MEM ADDRESS [00,01] (W)
   STAT    |= BASE;        // STATUS REGISTER      [02]    (R)
   COUNT   |= BASE;        // SEQENCER ADDRESS PTR [02]    (W)
   TIMER0  |= BASE;        // TIMER 0              [04]    (R/W)
   TIMER1  |= BASE;        // TIMER 1              [05]    (R/W)
   TIMER2  |= BASE;        // TIMER 2              [06]    (R/W)
   TIMERC  |= BASE;        // TIMER CONTROL WORD   [07]    (R/W)
   CNTL    |= BASE;        // CONTROL REGISTER     [08]    (R/W)
   DAC     |= BASE;        // DAC DATA             [0C]    (W)
}


//*********************************************************************
// FUNCTION NAME: initA2d()
// AUTHOR: Rick Roberts
// DATE: 13 November 1996
// DESCRIPTION: Performs necessary steps for initialization of the a2d
//              or to reinitialize if acceleration parameters are in
//              error due to a poor initial data transfer.
// RETURNS: void
// CALLS: setRmsOff(), setSequencer(), lockTrigger(), resetFifo(),
// unlockTrigger(), and setTrigger(), all in a2d.cpp
// CALLED BY: sampler class constructor
//*********************************************************************
```

```
void a2dClass::initA2d(void)
{
   setRmsOff();
   setSequencer();
   lockTrigger();
   resetFifo();
   setFifo();
   unlockTrigger();
   setTrigger();
}


//*******************************************************************
// FUNCTION NAME: reinitA2d()
// AUTHOR: Rick Roberts
// DATE: 13 November 1996
// DESCRIPTION: Performs necessary steps for reinitialization of the a2d
//              or to reinitialize if acceleration parameters are in
//              error due to a poor initial data transfer.
// RETURNS: void
// CALLS: readConfigFile(), initSysAddr(), initHardware(),2
//        setRmsOff(), setSequencer(), lockTrigger(), resetFifo(),
//        unlockTrigger(), and setTrigger(), all in a2d.cpp
// CALLED BY: sampler class readSamples if a2d FIFO has overflowed
//*******************************************************************

void a2dClass::reinitA2d(void)
{
   readConfigFile();
   initSysAddr();
   initHardware();
   setRmsOff();
   setSequencer();
   lockTrigger();
   resetFifo();
   setFifo();
   unlockTrigger();
   setTrigger();
}

//*******************************************************************
// FUNCTION NAME: initHardware()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D Control Register to 0020 and sets the data
//              member, ctrlw=0060; initializes the module setup for
//              software triggering of the A2D.  Programs each channel.
// RETURNS: void
// CALLS: outpw()
// CALLED BY: a2d class constructor
//*******************************************************************
```

```
void a2dClass::initHardware(void)
{
   outpw(CNTL,SET_TRG);
   ctrlw = SET_TRG|RST_TRG;

   if (mode_sel == 0)
      setSe();

   else
      setDiff();

   for(int i = 0;i < chcnt;i++){
      setChannel(seqaddr[i],chan[i],g10[i],g2[i]);
   }
   setAcDc(mode_acdc);
   initTiming(delta_t);
   setCount(chcnt);
}


//*********************************************************************
// FUNCTION NAME: printCtrlw()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Print A2D control register var, ctrlw.
//              The variable is used to set a byte in the
//              ESP A2D control register at BASE + 08h/09h
//              Used during application code debug
// RETURNS: void
// CALLS: none
// CALLED BY: none
//*********************************************************************

void a2dClass::printCtrlw(void)
{
   printf("ctrlw: %04x\t", ctrlw);
   for (int i=0x00; i < 0x10; i++){
      printf("%i",((ctrlw>>0x0F-i) & 1));
      if ((i+1)%4==0)
         printf(" ");
   }
}
```

```
//*********************************************************************
// FUNCTION NAME: setSe()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets ctrlw for single ended mode and writes ctrlw to
//              A2D Control Register
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
//*********************************************************************

void a2dClass::setSe(void)
{
   ctrlw &= ~DIFF;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setDiff()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets ctrlw for differential mode and writes ctrlw to
//              A2D Control Register
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
//*********************************************************************

void a2dClass::setDiff(void)
{
   ctrlw |= DIFF;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setChannel()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Loads sequencer memory with channel data
// CALLS: progSeq(), outpw(), runSeq()
// CALLED BY: initHardware()
// VARIABLES: seq - sequencer number
//            ch  - channel number
//            g10 - x10 gain value
//            g2  - x2 gain value
//*********************************************************************
```

```
void a2dClass::setChannel(unsigned seq,unsigned ch,unsigned g10,
                          unsigned g2)
{
   unsigned d = 0;

   setProgSeq();            // set sequencer program mode
   outpw(COUNT,seq);        // set sequencer address

   //load sequencer memory
   d |= ch<<8;              // channel
   d |= (g2<<12);           // gain X2
   d |= (g10<<14);          // gain X10
   outpw(MEM,d);            // load sequencer

   setRunSeq();             // set sequencer run mode
}


//*********************************************************************
// FUNCTION NAME: setProgSeg()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets sequencer to program mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: setChannel()
//*********************************************************************

void a2dClass::setProgSeq(void)
{
   ctrlw |= PRG_SEQ;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setRunSeq()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets sequencer to run mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::setRunSeq(void)
{
   ctrlw &= ~PRG_SEQ;
   outpw(CNTL,ctrlw);
}
```

```
//*********************************************************************
// FUNCTION NAME: setCount()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Loads sequencer address counter with number of channels
//              to scan.
// RETURNS: void
// CALLS: outpw(), setProgSeq(), setRunSeq()
// CALLED BY: initHardware()
// VARIABLES: nch - number of channels to sequence
//*********************************************************************

void a2dClass::setCount(unsigned nch)
{
   nch=nch<<4;              // put in upper nibble
   outpw(COUNT,nch);        // out to register
   setProgSeq();            // reset sequencer
   setRunSeq();             // put it in run mode
}


//*********************************************************************
// FUNCTION NAME: setAcDc()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets AC or DC Coupling
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
// VARIABLES: acdc - holds coupling value
//*********************************************************************

void a2dClass::setAcDc(unsigned acdc)
{
   if (acdc)
      ctrlw |= ACDC;       // acdc=1 -> DC
   else
      ctrlw &= ~ACDC;      // acdc=0 -> AC
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: lockTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Prevents triggering
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************
```

```
void a2dClass::lockTrigger(void)
{
   ctrlw &= ~RST_TRG;
   outpw(CNTL,ctrlw);
}

//*********************************************************************
// FUNCTION NAME: unlockTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Allow the triger to function
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::unlockTrigger(void)
{
   ctrlw |= RST_TRG|SET_TRG;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Toggle the trigger (software triggering)
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::setTrigger(void)
{
   outpw(CNTL,ctrlw&~SET_TRG|RST_TRG);
   outpw(CNTL,ctrlw| SET_TRG|RST_TRG);
}


//*********************************************************************
// FUNCTION NAME: resetTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Clears the trigger
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************
```

```
void a2dClass::resetTrigger(void)
{
   outpw(CNTL,ctrlw|SET_TRG&~RST_TRG);
   outpw(CNTL,ctrlw|SET_TRG| RST_TRG);
}

//*********************************************************************
// FUNCTION NAME: setRmsOn()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Switches in the RMS measurement chip
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::setRmsOn(void)
{
   ctrlw |= RMS;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setRmsOff()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Switches out RMS measurement chip
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::setRmsOff(void)
{
   ctrlw &= ~RMS;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setSequencer()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D module to sequencer mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************
```

```
void a2dClass::setSequencer(void)
{
   ctrlw |= SAM_SEQ;
   outpw(CNTL,ctrlw);
}


//*******************************************************************
// FUNCTION NAME: setSamplerRate()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D module to sampler mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
// VARIABLES: rate - sampler rate
//*******************************************************************

void a2dClass::setSamplerRate(unsigned rate)
{
   ctrlw &= ~SAM_SEQ;    //Set to sampler mode
   ctrlw &= CLRRATE;     //Clear previous rate to 000
   ctrlw |= rate;        //Set new rate
   outpw(CNTL,ctrlw);    //Set Control Word
}


//*******************************************************************
// FUNCTION NAME: gate1outOn()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Set GATE1OUT bit of control word high
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*******************************************************************

void a2dClass::gate1outOn(void)
{
   ctrlw |= GATE1OUT;
   outpw(CNTL,ctrlw);
}
```

```
//*********************************************************************
// FUNCTION NAME: gate1outOff()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Set GATE1OUT bit of control word low
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::gate1outOff(void)
{
   ctrlw &= ~GATE1OUT;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: squareWaveTimer1()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets timer channel 1 to square-wave input
// RETURNS: void
// CALLS: outp()
// CALLED BY: main
// VARIABLES: dt-micro seconds per period (1 to 8192)
//                 assuming 8 MHz clock input
//             ch-timer channel 1
//             ph-local variable
//             pl-local variable
//*********************************************************************

void a2dClass::squareWaveTimer1(unsigned dt)
{
   char    ph,pl;

   pl = (dt*8)&0xFF;        // 8 CLOCKS PER uS
   ph = (dt*8)>>8;

   outp(TIMERC,0x76);       // initialize timer
   outp(TIMER1,pl);         // dt uS delay
   outp(TIMER1,ph);         // with 8 MHz clock
}
```

```
//*********************************************************************
// FUNCTION NAME: initTiming()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Initialize the A2D timing using timer 2
// RETURNS: void
// CALLS: outp()
// CALLED BY: initHardware()
// VARIABLES: dt - number of micro seconds (3 to 2730)
//*********************************************************************

void a2dClass::initTiming(unsigned dt)
{
   char    ph,pl;

   pl = (dt*8)&0xFF;     // 8 CLOCKS PER uS
   ph = (dt*8)>>8;

   outp(TIMERC,0xB6);    // initialize timer2
   outp(TIMER2,pl);      // dt uS delay
   outp(TIMER2,ph);      // with 8 MHz clock
}


//*********************************************************************
// FUNCTION NAME: resetFifo()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Rewind FIFO to beginning of memory
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::resetFifo(void)
{
   ctrlw &= ~RST_FIFO;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: setFifo()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Enable FIFO to acquire data
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************
```

```
void a2dClass::setFifo(void)
{
   ctrlw |= RST_FIFO;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: getFifoStatus()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Returns FIFO status
// RETURNS: RETURNS: 6 - empty
//                   5 - half full
//                   1 - full
// CALLS: inpw()
// CALLED BY: main
//*********************************************************************

unsigned a2dClass::getFifoStatus(void)
{
   return (inpw(STAT)&7);
}


//*********************************************************************
// FUNCTION NAME: getFifoData()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Returns next data word stored in FIFO
// RETURNS: 16bits of data. Lower 12 are A2D data
// CALLS: inpw()
// CALLED BY: a2d class constructor
//*********************************************************************

signed a2dClass::getFifoData(void)
{
   return (inpw(FIFO)&0x0FFF);     //Get data and mask upper nibble
}

//*********************************************************************
// FUNCTION NAME: setIntRate()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Program timer channel 0 to set the desired interrupt rate
// RETURNS: void
// CALLS: outp()
// CALLED BY: main
// VARIABLES: intrate-micro secs per period (1 to 8192)
//            assuming 8 MHz clock input
//*********************************************************************
```

```
void a2dClass::setIntRate(unsigned intrate)
{
   outp(TIMERC,0x36);                // Set timer 0 to mode 3
   outp(TIMER0,(intrate*8)&0xFF);    // Load Least Significant Byte
   outp(TIMER0,(intrate*8)>>8);      // Load Most Significant Byte
}

//*******************************************************************
// FUNCTION NAME: intOff()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Locksout the interupt request line
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*******************************************************************

void a2dClass::intOff(void)
{
   ctrlw &= ~INT_EN;          // INT_EN is active high
   outpw(CNTL,ctrlw);
}


//*******************************************************************
// FUNCTION NAME: intOn()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Enables system interuppt request
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*******************************************************************

void a2dClass::intOn(void)
{
   ctrlw |= INT_EN;          // INT_EN is active high
   outpw(CNTL,ctrlw);
}

//*******************************************************************
// FUNCTION NAME: setTriggerLevel()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets the trigger level
// RETURNS: void
// CALLS: outp()
// CALLED BY: main
// VARIABLES: tl-trigger level  (0=-10V, 128=0V, 255=+10V)
//*******************************************************************
```

```
void a2dClass::setTriggerLevel(unsigned tl)
{
   outp(DAC,tl);
}


//*********************************************************************
// FUNCTION NAME: setTriggerPosition()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Sets falling or rising edge trigger
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
// VARIABLES: tp: 0=falling, 1=rising
//*********************************************************************

void a2dClass::setTriggerPosition(unsigned tp)
{
   ctrlw &= ~TRG_POS;        //Clear previous TRG_POS
   ctrlw |= (tp)?TRG_POS:0; //Evaluate tp and set ctrlw
   outp(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: zeroOffset()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Calibrates zero offset error
// RETURNS: void
// CALLS: outpw()
// CALLED BY: a2d class constructor
//*********************************************************************

void a2dClass::zeroOffset(void)
{
   unsigned d=0,i,g2,g10;
   float    sum;
   float    offsetErr[4][4];
   float    bits[4][4];
   unsigned gains10[4] = {1,10,100,100};
   unsigned gains2[4]  = {1, 2,  4,  8};

   clrscr();
   printf("\n\tG10\tG2\t OFFSET\t\t BITS");

   for(g10 = 0; g10 < 4; g10++)
      for(g2 = 0; g2 < 4; g2++)
         printf("\n\t%d\t%d\t+X.XXXXXX\t+XX.X",g10,g2);
```

```
    setRmsOff();
    setAcDc(0);
    setSequencer();
    initTiming(3);
    setChannel(0,0,g10,g2);
    grndInput();
    delay(5);                //Let new gain values stabilize

    while (!kbhit()){
       for (g10 = 0; g10 < 4; g10++){
          for (g2 = 0; g2 < 4; g2++){
             setChannel(0,0,g10,g2);
             grndInput();
             lockTrigger();
             resetFifo();
             setFifo();
             unlockTrigger();
             setTrigger();
             delay(1);
             while (getFifoStatus() != FULL);
                lockTrigger();

             for (i = 0, sum = 0.0; i < FIFOSIZE; i++){
                d=getFifoData();
                sum+=(float)d*10/2048;
             }
             offsetErr[g10][g2]=((float)(sum/FIFOSIZE)-10)/
                                   (float)(gains10[g10]*gains2[g2]);
             bits[g10][g2] =
             (float)(offsetErr[g10][g2]*4096/20*gains10[g10]*gains2[g2]);
          }
          clrscr();
          printf("\n\tG10\tG2\t OFFSET\t\t BITS");
          for (g10 = 0; g10 < 4; g10++){
             for (g2 = 0; g2 < 4; g2++){
                printf("\n\t%d\t%d\t%+1.6f\t%+04.1f",g10,g2,
                                    offsetErr[g10][g2],bits[g10][g2]);
             }
          }
       }
       freeInput();
       getch();
    }
}
```

```
//*********************************************************************
// FUNCTION NAME: grndInput()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Grounds the two diff input for zero adjust
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::grndInput(void)
{
   ctrlw |= CAL;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: freeInput()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Ungrounds the two diff inputs
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::freeInput(void)
{
   ctrlw &= ~CAL;
   outpw(CNTL,ctrlw);
}


//*********************************************************************
// FUNCTION NAME: zeroAdjust()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Adjust the trimmer on the PGA
// RETURNS: void
// CALLS: outpw()
// CALLED BY: main
//*********************************************************************

void a2dClass::zeroAdjust(void)
{
   int      i;
   unsigned d;
   float    sum,offsetErr;

   clrscr();
```

```
    printf("\n\nADJUST THE TRIM POT FOR 0.0 OFFSET\n\n");

    setRmsOff();
    setAcDc(0);
    setSequencer();
    initTiming(3);

    while(!kbhit()){
       setChannel(0,0,3,3);
       grndInput();
       lockTrigger();
       resetFifo();
       setFifo();
       unlockTrigger();
       setTrigger();
       while(getFifoStatus() != FULL);
       lockTrigger();

       for (i = 0, sum = 0.0; i < FIFOSIZE; i++) {
          d = getFifoData();
          sum += (float)d*10/2048;
       }

       offsetErr=((float)(sum/FIFOSIZE)-10)/8000.0;

       printf("\tTHE MEASURED DC OFFSET IS: %+8.6f\r",offsetErr);
    }

    freeInput();
    getch();
}
// end of file a2d.cpp
```

APPENDIX B: Serial Port Communications Source Code (C++)

A. GLOBALS.H

```
#ifndef _GLOBALS_H
#define _GLOBALS_H

#include <dos.h>

// types
typedef  unsigned charBYTE;
typedef  unsigned short WORD;
typedef  unsigned long  DWORD;

#define  MEM(seg,ofs)     (*((BYTE far*)MK_FP(seg,ofs)))
#define  MEMW(seg,ofs)    (*((WORD far*)MK_FP(seg,ofs)))

enum Boolean   {FALSE, TRUE};

// basic bit twiddles
#define  set(bit)              (1<<bit)
#define  setb(data,bit)        (data | set(bit))
#define  clrb(data,bit)        (data & !set(bit))
#define  setbit(data,bit)      (data = setb(data,bit))
#define  clrbit(data,bit)      (data = clrb(data,bit))

// specific to ports
#define  setportbit(reg,bit)  (outportb(reg,setb(inportb(reg),bit)))
#define  clrportbit(reg,bit)  (outportb(reg,clrb(inportb(reg),bit)))

// navigation conversion factors and useful global variables
#define MSECS_TO_DEGREES (1.0/(1000.0 * 3600.0))  // time conversion
factors
#define DEGREES_TO_MSECS 3600000.0
#define MINS_TO_MSECS 60000.0

// Conversion constants for location of 36:35:42.2N and 121:52:28.7W
#define LatToFt 0.10134                // converts degrees Latitude to ft
#define LongToFt 0.08156              // converts degrees Longitude to ft
#define HemisphereConversion -1        // -1 if west of of Greenwich

#define RADIANMAGVAR 0.261799 // Local Magnetic variation in radians

#define radToDeg  (180.0/M_PI)
#define degToRad  (M_PI/180.0)


#endif
```

B. BUFFER.H

```
#ifndef _BUFFER_H
#define _BUFFER_H

#include "toetypes.h"
#include "globals.h"

#define ONE (unsigned short)1

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

   CLASS:       bufferClass
   AUTHOR:      Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:        11 July 1995
   FUNCTION:    Base class for use as a polymorphic reference in the
                serial port code which defines a buffer to be used in
                serial port communications.

*********************************************************************

class bufferClass  {

   public:

      // Constructor
      bufferClass(WORD sz);
      ~bufferClass() {}

      // Checks for the arrival of new characters in the buffer
      Boolean hasData() { return Boolean(putPtr != getPtr); }

      // How much of the Buffer is used (rounded percentage 0 - 100)
      int capacityUsed();

      Boolean Get(BYTE&);          // read from the buffer
      void Add(BYTE);              // write to the buffer

   protected:

      // Increment the pointer to next position
      void  inc(WORD& index) { if (++index == size) index = 0; }

      WORD before(WORD index)              // decrement the pointer
         { return ((index == 0) ? size - ONE : index - ONE);}
      WORD getPtr;         // Location of unread data
      WORD putPtr;         // Location to read data to
      WORD size;           // Size of the buffer in bytes
      BYTE* buf;
};
#endif
```

C. BUFFER.CPP

```
#include <iostream.h>
#include <stdio.h>

#include "globals.h"
#include "buffer.h"

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

// FUNCTION NAME:  bufferClass constructor
// AUTHOR:         Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
// DATE:           11 July 1995
// DESCRIPTION:    Instantiates a buffer
// RETURNS:        void
// CALLS:          none
// CALLED BY:      compBuffer, GPSbuffer, bufferedSerialPort constructors
//
*********************************************************************

bufferClass::bufferClass(WORD sz) : getPtr(0), putPtr(0), size(sz)
{
   buf = new BYTE[size];
}

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

// FUNCTION NAME:  capacityUsed()
// AUTHOR:         Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
// DATE:           11 July 1995
// DESCRIPTION:    Returns the rounded percentage of the buffer used.
// RETURNS:        void
// CALLS:          none
// CALLED BY:      bufferedSerialPort::processInterrupt
//
*********************************************************************

int bufferClass::capacityUsed()
{
   int cap = (putPtr + size) % size - getPtr;
   return 100 * cap / size;
}
```

```
//*********************************************************************
// FUNCTION NAME:  Get
// AUTHOR:         Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
// DATE:           11 July 1995
// DESCRIPTION:    Reads a character from the buffer
// RETURNS:        Boolean
// CALLS:          hasData()
// CALLED BY:      GPSbufferClass, compBufferClass
//
*********************************************************************

Boolean bufferClass::Get(BYTE& data)
{
   if (hasData())  {
      data = buf[getPtr];
      inc(getPtr);
      return TRUE;
   }
   return FALSE;
}

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

// FUNCTION NAME:  Add
// AUTHOR:         Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
// DATE:           11 July 1995
// DESCRIPTION:    Writes a character to the buffer and checks for buffer
//                 overflow
// RETURNS:        void
// CALLS:          hasData
// CALLED BY:      GPSbufferClass, compBufferClass
//
*********************************************************************

void bufferClass::Add(BYTE ch)
{
   buf[putPtr] = ch;
   inc(putPtr);
   if (!hasData()) { // if no data after adding data, it overflowed
      cerr << "\nError: byteBuffer overflow\n";
   }
}
// end of file buffer.cpp
```

D. GPSBUFF.H

```
#ifndef _GPSBUFF_H
#define _GPSBUFF_H

#include "globals.h"
#include "toetypes.h"
#include "buffer.h"

#define  GPSBLOCKS      4
#define  LINE_FEED      10
#define  CARR_RETURN    13

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

  Class buffers GPS position messages via serial port communications.
  Uses a multiple buffer system in which each buffer is capable of
  holding a single position message. Buffers are filled and processed
  sequentially in a round robin fashion. Messages are checked for
  validity only upon attempted reads from the buffer.

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

class gpsBufferClass : public bufferClass  {

   public:

      gpsBufferClass(BYTE GPSblocks = GPSBLOCKS);
      ~gpsBufferClass() { delete [] block; }

      Boolean  hasData();                // a complete structure is ready
      Boolean  Get(BYTE&)    { return FALSE; }
      Boolean  Get(GPSdata);             // fill in a complete structure
      void     Add(BYTE ch);            // build the structure byte by byte

   protected:

      Boolean  validHeader(GPSdata);    // check a block for valid header
      GPSdata  *block;                  // hold the buffered GPS data
      WORD     current, last;           // current and last GPS block in
use
      BYTE     *putPlace;               // for the next character received
};

#endif
```

E. GPSBUFF.CPP

```
#include <iostream.h>
#include <stdio.h>

#include "gpsbuff.h"

/
*********************************************************************
   PROGRAM:    gpsBuffer (Constructor)
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Allocates message buffers, indicate that no data has been
               received by equalizing current and last and set position
               into which initial character will be read.
   RETURNS:    nothing.
   CALLED BY:  navigator class (nav.h)
   CALLS:      none.

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

gpsBufferClass::gpsBufferClass(BYTE GPSblocks) : current(0), last(0),
                 bufferClass(GPSblocks) // Call to base class constructor
{
   cerr << "constructing gpsBuffer" << endl;
   block = new GPSdata[GPSblocks]; // Create an array of GPSdata elements
   putPlace = &(block[current][0]); // Set the place for first character
}


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

   PROGRAM:    Add
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Interrupt driven routine which writes incoming characters
               into the gps buffers
   RETURNS:    nothing.
   CALLED BY:  interupt driven by bufferedSerialPort
   CALLS:      none.

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

void gpsBufferClass::Add(BYTE data)
{
   static BYTE lastChar(data);       // Holds last for <cr> <lf> detection
   static Boolean lfFlag = FALSE;    // True when message end is detected

   if (lfFlag && (data == '@')) {    // Is a new message starting?
      last = current;            // Set last to buffer with newest message.
      inc(current);                 // Set current to the next buffer
```

```
      // Set putPlace to the beginning of the next buffer.
      putPlace = &(block[current][0]);
      lfFlag = FALSE;                    // reset for end of next message.
   }

   *putPlace++ = data;                 // Write character into the buffer.

   //Has the end of a message been received?
   if ((lastChar == CARR_RETURN) && (data == LINE_FEED)) {
      lfFlag = TRUE;
   }
   lastChar = data;              //Save last character for <cr> <lf>
detection
}


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

   PROGRAM:   Get
   AUTHOR:    Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Checks to see if a new message has arrived, copies it into
              the input argument data and returns a flag to indicate
              whether a new message was received.
   RETURNS:   TRUE, if a new valid position has been received.
              FALSE, otherwise
   CALLED BY: navPosit (nav.cpp)
              initializeNavigator (nav.cpp)
   CALLS:     gpsBufferClass::hasData

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

Boolean gpsBufferClass::Get(GPSdata data)
{
   if (hasData())  {                // Has a new valid message been
received.
      // Copy the message out of the buffer.
      memcpy (data, block + last, GPSBLOCKSIZE);
      last = current;         // Indicate that this message has been read.
      return TRUE;
   }
   else {
      return FALSE;
   }
}
```

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

   PROGRAM:    hasData
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Determines whether a new message has been received and
               checks to see if it has a valid header.
   RETURNS:    TRUE, if a new valid message has been received.
   CALLED BY:  gpsBufferClass::Get (buffer.cpp)
   CALLS:      validHeader (buffer.cpp)
*********************************************************************/

Boolean gpsBufferClass::hasData()
{
   // Has a new message with a valid header been received
   if (last != current) {
      if (validHeader(block[last])) {
         return TRUE;
      }
      else {
         return FALSE;
      }
   }
   return FALSE;
}


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

   PROGRAM:    validHeader
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Checks to see if a message has the proper header for a
               Motorola position message. (@@Ea)
   RETURNS:    TRUE, if the header is valid. FALSE, otherwise.
   CALLED BY:  gpsBufferClass::hasData (buffer.cpp)
   CALLS:      none.

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

Boolean gpsBufferClass::validHeader(GPSdata dataPtr)
{
   if ((dataPtr[0] == '@') && (dataPtr[1] == '@') &&
       (dataPtr[2] == 'E') && (dataPtr[3] == 'a'))  {
      return TRUE;
   }
   else {
      return FALSE;
  }
}
// end of file gpsbuff.cpp
```

F. COMPBUFF.H

```
#ifndef __COMPBUFF_H
#define __COMPBUFF_H

#include "toetypes.h"
#include "globals.h"
#include "buffer.h"

#define  COMPBLOCKS      8
#define  LINE_FEED      10
#define  CARR_RETURN    13
#define  g              103
#define  o              111


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

  Class buffers COMPASS messages received via serial port communications.
  Uses a multiple buffer system in which each buffer is capable of
  holding a single message. Buffers are filled and processed
  sequentially in a round robin fashion. Messages are checked for
  validity only upon attempted reads from the buffer.

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

class compBufferClass : public bufferClass  {

public:

   compBufferClass(BYTE compBlocks = COMPBLOCKS);

   ~compBufferClass() {delete [] block;}

   Boolean  hasData();                    // a complete structure is ready
   Boolean  Get(BYTE&) {return FALSE;}    // satisfy inheritence
   Boolean  Get(compData);              // get a complete structure filled in
   void     Add(BYTE ch);                // build the structure byte by byte

protected:                                // for inheritance

   Boolean  validHeader(compData);       // check a block for valid header
   compData *block;                      // points to array of compass msgs
   WORD     current, last;             // current and last comp block in use

   BYTE     *putPlace;                   // for the next character received
};

#endif
```

G. COMPBUFF.CPP

```
#include <iostream.h>
#include <stdio.h>

#include "compbuff.h"

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

   PROGRAM:  compBuffer (Constructor)
   AUTHOR:   Eric Bachmann, Randy Walker
   DATE:     28 April 1996
   FUNCTION: Allocates message buffers, indicates that no data has been
             received by equalizing current and last and sets the position
              into which initial character will be read.

   RETURNS:  nothing.

   CALLED BY: compassClass (compass.h)

   CALLS:     none.

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


compBufferClass::compBufferClass(BYTE compBlocks):  current(0), last(0),
                bufferClass(compBlocks) // Call to base class constructor
{
   cerr << "compBuffer constructor called" << endl;

   block = new compData[compBlocks]; // Create array of message buffers
   putPlace = &(block[current][0]);  // Set position for first character

   cerr << "compBuffer constructed." << endl;
}



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

   PROGRAM:   compBuffer::Add
   AUTHOR:    Eric Bachmann, Randy Walker
   DATE:      28 April 1996
   FUNCTION:  Interrupt driven routine which writes incoming characters
              into the compass message buffers
   RETURNS:   nothing.
   CALLED BY: interrupt driven by compassPort
   CALLS:     none.
*********************************************************************/
```

```
void compBufferClass::Add(BYTE data){

   static Boolean lfFlag = FALSE;          //True, if message end detected
   static int messageCount(0);     // Counts characters in current message

   if (lfFlag && (data == '$')) {          // Is a new message starting?

      last = current;            // Set last to buffer with newest message.
      inc(current);               // Set current to the next buffer

      // Set putPlace to the beginning of the next buffer.
      putPlace = &(block[current][0]);
      lfFlag = FALSE;                     // reset for end of next message.
   }

   *putPlace++ = data;                  // Write character into the buffer.
   messageCount++;

   //Has the end of a message been received (<cr><lf>)?
   if (data == LINE_FEED) {
      lfFlag = TRUE;
   }
}

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

   PROGRAM:    compBuffer::Get
   AUTHOR:     Eric Bachmann, Randy Walker
   DATE:       28 April 1996
   FUNCTION:   Checks to see if a new message has arrived, copies it
               into the input argument data and returns a flag to indicate
               whether a new message was received.
   RETURNS:    TRUE, if a new valid position has been received.
               FALSE, otherwise
   CALLED BY:  compass.cpp
   CALLS:      compBuffer::hasData

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

Boolean compBufferClass::Get(compData data)
{
   if (hasData())  {            // Has a new valid message been received.
      // Copy the message out of the buffer.
      memcpy (data, block + last, COMPSIZE);
      last = current;          // Indicate that this message has been read.
      return TRUE;
   }
   else {
      return FALSE;
   }
}
```

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

   PROGRAM:    compBuffer::hasData
   AUTHOR:     Eric Bachmann, Randy Walker
   DATE:       28 April 1996
   FUNCTION:   Determines whether a new message has been received and
               checks to see if it has a valid header.
   RETURNS:    TRUE, if a new valid message has been received.
   CALLED BY:  compBuffer::Get
   CALLS:      validHeader (compBuffer.cpp)
*********************************************************************/

Boolean compBufferClass::hasData()
{
   if ((last != current) && (validHeader(block[last]))) {
      return TRUE;
   }
   else {
      return FALSE;
   }
}


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

   PROGRAM:     validHeader
   AUTHOR:      Eric Bachmann, Dave Gay
   DATE:        11 July 1995
   FUNCTION:    Checks to see if a message has the proper header for a
                compass message. ($C)
   RETURNS:     TRUE, if the header is valid. FALSE, otherwise.
   CALLED BY:   compBuffer::hasData
   CALLS:       none.
*********************************************************************/

Boolean compBufferClass::validHeader(compData dataPtr)
{
  if ((dataPtr[0] == '$') && (dataPtr[1] == 'C')) {
     return TRUE;
  }
  else {
     return FALSE;
  }
}
//end of file compbuff.cpp
```

H. SERIAL.H

```
#ifndef _SERIAL_H
#define _SERIAL_H

#include <dos.h>
#include <stdio.h>
#include "globals.h"

#define     ALMOST_FULL    80    // % full to turn off DTR (user defines)

// leave the following alone - hardware specific

enum  COMport         {COM1=1, COM2, COM3, COM4};
enum  BaudRate        {b300, b1200, b2400, b4800, b9600};
enum  ParityType      {ERROR=-1, NOPARITY, ODD, EVEN};
enum  handShake       {NONE, RTS_CTS, XON_XOFF};
enum  Shake           {off, on};
enum  interruptType   {rx_rdy, tx_rdy, line_stat, modem_stat};

#define  BIOSMEMSEG  0x40
#define  DLAB        0x80
#define  IRQPORT     0x21
#define  EOI         outportb(0x20,0x20)

#define  COM1base    MEMW(BIOSMEMSEG,0)
#define  COM2base    MEMW(BIOSMEMSEG,2)

#define  TX          (portBase)
#define  RX          (portBase)
#define  IER         (portBase+1)
#define  IIR         (portBase+2)
#define  LCR         (portBase+3)
#define  MCR         (portBase+4)
#define  LSR         (portBase+5)
#define  MSR         (portBase+6)
#define  LO_LATCH    (portBase)
#define  HI_LATCH    (portBase+1)

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

   CLASS:       serialPortClass
   AUTHOR:      Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:        11 July 1995, last modified January 1997
   FUNCTION:    Parent class, defines a simple serial port.

*********************************************************************/
class serialPortClass  {

   public:
```

```
        serialPortClass(COMport port, BaudRate baud, ParityType parity,
                        BYTE wordlen, BYTE stopbits, handShake hs);
        ~serialPortClass() {}

        Boolean     Send(BYTE data);
        Boolean     Get(BYTE& data);

        inline Boolean dataReady();
        Boolean statusChanged()
           { return Boolean((ifportbit(MSR,0) || ifportbit(MSR,1))); }

         // the rest are only if handshake is specified as RTS_CTS
         Boolean     isCTSon()            { return ifportbit(MSR,4); }
         Boolean     isDSRon()            { return ifportbit(MSR,5); }

         void        setDTRon()           { setportbit(MCR,0); }
         void        setDTRoff()          { clrportbit(MCR,0); }
         void        toggleDTR();

         void        setRTSon()           { setportbit(MCR,1); }
         void        setRTSoff()          { clrportbit(MCR,1); }
         void        toggleRTS();

    protected:

        WORD            portBase;
        handShake       ShakeType;
        Shake           DTRstate, RTSstate;

        inline Boolean    ifportbit(WORD, BYTE);
        inline void       toggle(Shake&);

};
#endif
```

I. SERIAL.CPP

```
#include <iostream.h>
#include <stdio.h>
#include "serial.h"

//   Usage Note: Because of the interrupt handlers used, you MUST call
//   your compassPort & gpsPort objects port2 & port1 so the
//   right handler gets called and can properly service the interrupt.

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

   PROGRAM:   serialPortClass (Constructor)
   AUTHOR:    Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995, last modified January 1997
   FUNCTION:  Initializes one of the Serial Ports.
       1) Determines the base I/O port address for the given COM port
       2) Sets the 8259 IRQ mask value
       3) Initializes the port parameters - baud, parity, etc.
       4) Calls the routine to initialize interrupt handling
       5) Enables DTR and RTS, indicating ready to go

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

serialPortClass::serialPortClass(COMport port, BaudRate speed,
                    ParityType parity, BYTE wordlen,
                    BYTE stopbits, handShake hs)     :
                    DTRstate(off), RTSstate(off), ShakeType(hs)
{
   cerr << "serialPort constructor called" << endl;
   delay(500);

   switch (port)  {                    // initialize appropriate port base
  case COM1:  portBase = COM1base;
                              break;
  case COM2:  portBase = COM2base;
                              break;
   }  // switch

   const WORD     bauddiv[] = {0x180, 0x60, 0x30, 0x18, 0xC};

   // Change 1
   outportb(IER,0);                          // disable UART interrupts
   (void)inportb(LSR);
   (void)inportb(MSR);
   (void)inportb(IIR);
   (void)inportb(RX);
   outportb(LCR,DLAB);  // set DLAB so can set baud rate (read only port)
   outportb(LO_LATCH,bauddiv[speed] & 0xFF);
   outportb(HI_LATCH,(bauddiv[speed] & 0xFF00) >> 8);
   setportbit(MCR,3);                        // turn OUT2 on
```

```
   BYTE  opt = 0;
   if (parity != NOPARITY)  {
      setbit(opt,3);        // enable parity
      if (parity == EVEN)   // set even parity bit. if odd, leave bit 0
         setbit(opt,4);
   }
   // now set the word length. len of 5 sets both bits 0 and 1 to
   // 0, 6 sets to 01, 7 to 10 and 8 to 11
   opt |= wordlen-5;
   opt |= --stopbits << 2;
   outportb(LCR,opt);

   if (ShakeType == RTS_CTS)  {
      setDTRon();
      setRTSon();
   }
   cerr << "serialPort constructed" << endl;
}


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

   PROGRAM:   Get
   AUTHOR:    Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Gets a byte from the port. Returns true if there's one
              there, and fills in the byte parameter. If there's no
              character, the parameter is left alone, and false is
              returned.

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

Boolean serialPortClass::Get(BYTE& data)
{
   if (dataReady())  {              // make sure there's a char there
      data = inportb(RX);           // read character from 8250
      return TRUE;
   }
   else
      return FALSE;
}
```

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

   PROGRAM:    Send
   AUTHOR:     Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Outputs a single character to the port. Returns Boolean
               status indicating whether successful

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

Boolean serialPortClass::Send(BYTE data)
{
   while (!(ifportbit(LSR,5))) {};                // wait until THR ready

   switch (ShakeType)  {

      case NONE:
                              outportb(TX,data);
                              return TRUE;

      case RTS_CTS:
                              if (isCTSon() && isDSRon())  {
                                 outportb(TX,data);
                                 return TRUE;
     }
                              else {
                                 return FALSE;
     }

//      case XON_XOFF:                      // add this later if needed
        default:
                              break;
   }
   return FALSE;
}


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

   PROGRAM:  dataReady
   AUTHOR:   Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Checks port to see if a character has arrived.

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

```
inline Boolean serialPortClass::dataReady()
{
/*      Un-commenting this code increases transmission errors, but this
        code is useful for troubleshooting, so is retained if needed
   if (ifportbit(LSR,1)) {
      cerr <<"\nOverrun Error\n";
   }
   if (ifportbit(LSR,2)) {
     cerr <<"\nParity Error\n";
   }
   if (ifportbit(LSR,3)) {
     cerr <<"\nFraming Error\n";
   }
*/
   return ifportbit(LSR,0);
}

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

   PROGRAM:  ifportbit
   AUTHOR:   Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: Checks for byte on inportb register

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

inline Boolean serialPortClass::ifportbit(WORD reg, BYTE bit)
{
   BYTE on = inportb(reg);
        on &= set(bit);
   return Boolean(on == set(bit));
}

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

   PROGRAM:  toggleDTR
   AUTHOR:   Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: toggles Data Transmit Ready if RTS_CTS is off
*********************************************************************/

void serialPortClass::toggleDTR()
{
   if (ShakeType != RTS_CTS)
      return;
   if (DTRstate == off)
      setDTRon();
   else
     setDTRoff();
  toggle(DTRstate);
}
```

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

   PROGRAM:  toggleRTS
   AUTHOR:   Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: toggle Ready to Send (RTS) if RTS_CTS is on.

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

void serialPortClass::toggleRTS()
{
   if (ShakeType != RTS_CTS)
      return;
   if (RTSstate == off)
      setRTSon();
   else
      setRTSoff();
   toggle(RTSstate);
}


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

   PROGRAM:  toggle
   AUTHOR:   Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:     11 July 1995
   FUNCTION: toggles value of the input variable

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

inline void serialPortClass::toggle(Shake& h)
{
   if (h == off)
       h =  on;
   else
       h = off;
}
// end of file serial.cpp
```

J. GPSPORT.H

```
#ifndef _GPSPORT_H
#define _GPSPORT_H

#include <dos.h>
#include <stdio.h>
#include "toetypes.h"
#include "globals.h"
#include "serial.h"
#include "gpsbuff.h"

// this is the type for a standard interrupt handler
typedef void interrupt (IntHandlerType)(...);

// com handler to interface with processInterrupt
void interrupt COM1handler(...);

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

   CLASS:gpsPortClass
   AUTHOR:Rick Roberts
   DATE:28 January 1997
   FUNCTION:    Defines a buffered serial port which is interrupt driven
                on receive, and buffers all incoming characters in the
                gps buffer

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

class gpsPortClass : public serialPortClass  {

   public:

      gpsPortClass(COMport portnum = COM1, BYTE irq = 4,
                      BaudRate speed = b9600,
                      ParityType parity = NOPARITY, BYTE wordlen = 8,
                      BYTE stopbits = 1,
                      handShake hs = XON_XOFF);

      ~gpsPortClass();

      Boolean Get(GPSdata& data);      // buffered version
      void  processInterrupt();        // buffers the incoming character

   protected:

      gpsBufferClass messages;

      BYTE irqbit;    // Value to allow enable PIC interrupts for COM port
      BYTE origirq;    // keep the original 8259 mask register value
      BYTE comint;
```

```
      IntHandlerType *origcomint;  // keep original vector for restoring
                                   // later

      // this allows the actual handler to access processInterrupt()
      friend IntHandlerType COM2handler;

};

extern gpsPortClass port1;

#endif
```

K. GPSPORT.CPP

```
#include <iostream.h>
#include <stdio.h>
#include "gpsPort.h"

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

   PROGRAM:   gpsPortClass (Constructor)
   AUTHOR:    Rick Roberts
   DATE:      28 January 1997
   FUNCTION:  Initializes the interrupts for the gps Serial Port.
        1) takes over the original COM interrupt
        2) sets the port bits, parity, and stop bit
        3) enables interrupts on the 8250 (async chip)
        4) enables the async interrupt on the 8259 PIC

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

gpsPortClass::gpsPortClass(COMport portnum, BYTE irq, BaudRate baud,
                           ParityType parity, BYTE wordlen,
                           BYTE  stopbits, handShake hs)    :
                       serialPortClass(portnum, baud, parity, wordlen,
                                       stopbits, hs),
                         irqbit(irq), comint(irqbit+8)
{
   cerr << "gpsPort constructor called" << endl;

   if (ShakeType == RTS_CTS)  {  // turn it off first, it was enabled
      setDTRoff();               // in the base class
      setRTSoff();
   }

   origcomint = getvect(comint);          // remember the original vector
```

```
   setvect(comint,COM1handler);            // point to the new handler

   setportbit(MCR,3);             // turn OUT2 on
   disable();     // disable all interrupts - critical section
   setportbit(IER,rx_rdy);        // enable ints on receive only
   origirq = inportb(IRQPORT);    // remember how it was
   clrportbit(IRQPORT,irqbit);    // enable COM ints

   if (ShakeType == RTS_CTS)  {
      setDTRon();
      setRTSon();
   }
   enable();

   EOI;
   cerr << "exiting gpsPort constructor" << endl;
}


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

   PROGRAM:    ~gpsPortClass
   AUTHOR:     Rick Roberts, Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:       28 January 1997
   FUNCTION:   Resets the interrupts.
     1) disables the 8250 (async chip)
     2) disables the interrupt chip for async int
     3) resets the 8259 PIC

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

gpsPortClass::~gpsPortClass()
{
   setvect(comint,origcomint);    // set the interrupt vector back
   outportb(IER,0);               // disable further UART interrupts
   outportb(MCR,0);               // turn everything off
   outportb(IRQPORT,origirq);
   EOI;
}


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

   PROGRAM:    Get
   AUTHOR:     Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Calls Get based on buffer type

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

```
Boolean gpsPortClass::Get(GPSdata& data)
{
   return messages.Get(data);
}

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

   PROGRAM:     showPorts
   AUTHOR:      Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:        11 July 1995
   FUNCTION:    Prints interrupt vector addresses.  This function is for
                trouble shooting, it is not called in the code.

*********************************************************************/
/*
int showPorts()
{
   BYTE* p = (BYTE*)COM2base;
   p += 5;
   fprintf(stderr,"%X   ",*p++);
   fprintf(stderr,"%X\n",*p++);
   fprintf(stderr,"IRQPORT = %X", inportb(IRQPORT));
   return 0;
}
*/

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

   PROGRAM:     COM1handler
   AUTHOR:      Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:        11 July 1995, last modified January 1997
   FUNCTION:    Specific interrupt handler which maps each interrupt to
                the proper ISR.

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

void interrupt COM1handler(...)
{
   port1.processInterrupt();
   EOI;
}

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

   PROGRAM:   processInterrupt
   AUTHOR:    Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995
   FUNCTION:  Calls the ISR based upon buffer type

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

```
void gpsPortClass::processInterrupt()
{
   if (dataReady())  {                    // make sure there's a char there
      BYTE data = inportb(RX);         // read character from 8250
       messages.Add(data);
   if (ShakeType == RTS_CTS && messages.capacityUsed() > ALMOST_FULL)
       setDTRoff();
   }
}
// end of file gpsport.cpp
```

L. COMPPORT.H

```
#ifndef _COMPORT_H
#define _COMPORT_H

#include <dos.h>
#include <stdio.h>
#include "toetypes.h"
#include "globals.h"
#include "serial.h"
#include "compbuff.h"

// this is the type for a standard interrupt handler
typedef void interrupt (IntHandlerType)(...);

// com handler to interface with processInterrupt
void interrupt COM2handler(...);

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

   CLASS:        compassPortClass
   AUTHOR:       Rick Roberts
   DATE:         28 January 1997
   FUNCTION:     Defines a buffered serial port which is interrupt driven
                 on receive, and buffers all incoming characters in the
                 compass buffer

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

class compassPortClass : public serialPortClass  {

   friend compassClass;

   public:

      compassPortClass(COMport portnum = COM2, BYTE irq = 3,
                        BaudRate speed = b9600,
                       ParityType parity = NOPARITY, BYTE wordlen = 8,
                        BYTE stopbits = 1, handShake hs = NONE);
```

```
      ~compassPortClass();

      Boolean     Get(BYTE& data);      // buffered version

      void processInterrupt();          // buffers the incoming character


   private:

      compBufferClass headings;

      BYTE irqbit;    // Value to allow enable PIC interrupts for COM port
      BYTE origirq;   // keep the original 8259 mask register value
      BYTE comint;

      IntHandlerType *origcomint; // keep original vector for restoring
                                  // later

      // this allows the actual handler to access processInterrupt()
      friend IntHandlerType COM2handler;
};

extern compassPortClass port2;

#endif
```

M. COMPPORT.CPP

```
#include <iostream.h>
#include "compport.h"

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

   PROGRAM:   compassPortClass (Constructor)
   AUTHOR:    Rick Roberts
   DATE:      28 January 1997
   FUNCTION:  Initializes the interrupts for the compass Serial Port.
      1) takes over the original COM interrupt
      2) sets the port bits, parity, and stop bit
      3) enables interrupts on the 8250 (async chip)
      4) enables the async interrupt on the 8259 PIC

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

compassPortClass::compassPortClass(COMport portnum, BYTE irq,
                                   BaudRate baud, ParityType
                                   parity, BYTE wordlen, BYTE
                                   stopbits, handShake hs) :
                serialPortClass(portnum, baud, parity, wordlen,
                                stopbits, hs)
```

```
{
   cerr << "compassPort constructor called" << endl;

   irqbit = irq;
   comint = irqbit + 8;

   if (ShakeType == RTS_CTS)  {   // turn it off first, it was enabled
      setDTRoff();                // in the base class
      setRTSoff();
   }

   origcomint = getvect(comint);          // remember the original vector

   setvect(comint,COM2handler);            // point to the new handler

   setportbit(MCR,3);            // turn OUT2 on
   disable();     // disable all interrupts - critical section
   setportbit(IER,rx_rdy);       // enable ints on receive only
   origirq = inportb(IRQPORT);   // remember how it was
   clrportbit(IRQPORT,irqbit);   // enable COM ints

   if (ShakeType == RTS_CTS)  {
      setDTRon();
      setRTSon();
   }
   enable();

   EOI;
   cerr << "exiting compassPort constructor" << endl;
}

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

   PROGRAM:   ~compassPort
   AUTHOR:    Rick Roberts, Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:      28 January 1997
   FUNCTION:  Resets the interrupts.
      1) disables the 8250 (async chip)
      2) disables the interrupt chip for async int
      3) resets the 8259 PIC

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

compassPortClass::~compassPortClass()
{
   setvect(comint,origcomint);   // set the interrupt vector back
   outportb(IER,0);              // disable further UART interrupts
   outportb(MCR,0);              // turn everything off
   outportb(IRQPORT,origirq);
   EOI;
}
```

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

   PROGRAM:   Get
   AUTHOR:    Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:      11 July 1995
   FUNCTION:  Calls Get based on buffer type

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

Boolean compassPortClass::Get(BYTE& data)
{
   return headings.Get(data);
}


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

   PROGRAM:    showPorts
   AUTHOR:     Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Prints interrupt vector addresses.  This function is for
               trouble shooting and is not called from the code.

*********************************************************************/
/*
int showPorts()
{
   BYTE* p = (BYTE*)COM2base;
   p += 5;
   fprintf(stderr,"%X   ",*p++);
   fprintf(stderr,"%X\n",*p++);
   fprintf(stderr,"IRQPORT = %X", inportb(IRQPORT));
   return 0;
}
*/

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

   PROGRAM:     COM2handler
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:11 July 1995, last modified January 1997
   FUNCTION:    Specific interrupt handler which maps each interrupt to
                the proper ISR.

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

void interrupt COM2handler(...)
{
   port2.processInterrupt();
   EOI;
}
```

```
/*********************************************************************
   PROGRAM:   processInterrupt
   AUTHOR:    Frank Kelbe, Eric Bachmann, Dave Gay, Rick Roberts
   DATE:      11 July 1995
   FUNCTION:  Calls the ISR based upon buffer type

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

void compassPortClass::processInterrupt()
{
   if (dataReady())  {                    // make sure there's a char there
      BYTE data = inportb(RX);         // read character from 8250
      headings.Add(data);
   if (ShakeType == RTS_CTS && headings.capacityUsed() > ALMOST_FULL)
      setDTRoff();
   }
}
// end of file compport.cpp
```

APPENDIX C. SANS TILT-TABLE TEST TUNING AND CALIBRATION PROCEDURE

1. **Isolate Accelerometer Input From Integrator**

   --Set K_I to zero.

   --Only angular rate scale factor and bias effects will be reflected in error

2. **Choose Initial Bias Weight (biasWght)**

   --Using project experience, background theory

3. **Determine Angular Rate Scale Factor**

   --Baseline setting is 1.0.

   --Adjust by determining SANS output vs. actual angle excursion.

   --Apply ratio to current scale factor to obtain corrected scale factor.

   --Commanded tilt table angles taken as truth

   --Scale factor adjusts the output of the IMU to actual tilt results.

   --pScale (roll), qScale (pitch) rScale (yaw)

4. **Adjust Gain Value Above Zero**

   --Re-includes accelerometer input to filter

5. **Determine Accelerometer Scale Factor**

   --Same process as angular rate scale factor

   --xAccelScale (pitch), yAccelScale (roll), zAccelScale(yaw)

6. **Fine Tuning**

   --Adjust various factors from 1-5 above

LIST OF REFERENCES

Bachmann, E.R. and Gay, D., *Design and Evaluation of an Integrated GPS/INS System for Shallow-water AUV Navigation (SANS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1995.

Bachmann, E.R., McGhee, R.B., Whalen, R.H., Steven, R., Walker, R.G., Clynch, J.R., Healey, A.J., and Yun, X.P., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," *Proceedings of the 1996 IEEE Symposium on Autonomous Underwater Vehicle Technology*, Monterey, California, June, 1996, pp. 268-275.

Bennamoun, M., Boashash, B., Farugi, F., and Dunbar, M., "The Development of an Integrated GPS/INS/Sonar Navigation System for Autonomous Underwater Vehicle Navigation," *Proceedings of the 1996 IEEE Symposium on Autonomous Underwater Vehicle Technology*, June 2-6, 1996, Monterey, California, pp. 256-261.

Bergem, O., "A Multibeam Sonar Based Positioning System for an AUV," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology*, Durham, New Hampshire, September 27-29, 1993, pp. 291-299.

Bowditch, N., *American Practical Navigator, Vol. 1 and 2*, Defense Mapping Agency Hydrographic/Topographic Center, 1984.

Brown, R.G and Hwang, P.Y.C., *Introduction to Random Signals and Applied Kalman Filters*, 2nd Edition, John Wiley and Sons, New York, 1992.

Brutzman, D.P., *A Virtual World for an Autonomous Underwater Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, December, 1994. Available at *http://www.sh.nps.navy.mil/ubrutzman/dissertation.*

Brutzman, D.P., Burns, M., Campbell, M., Davis, D.T., Healey, A.J., Holden, M., Leonhardt, B., Marco, D., McClarin, D., McGhee, R.B. and Whalen, R.,"NPS Pheonix AUV Software Integration and In-Water Testing," *Proceedings of the 1996 IEEE Symposium on Autonomous Underwater Vehicle Technology*, Monterey, California, June, 1996.

Cox, I.J. and Wilfong, G.T., *Autonomous Robot Vehicles*, Springer-Verlag, New York, 1990.

Cox, R., Wei, S., "Advances in the State of the Art for AUV Inertial Sensors and Navigation Systems," *IEEE Journal of Oceanic Engineering*, October 1995, Number 4, pp. 361-366.

Craig, J.J., *Introduction to Robotics Mechanics and Control*, 2nd Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

DATEL BWR Series Data Sheet, DATEL, Inc., September, 1993.

Davidson, S.L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

Frequency Devices DP74 Series Data Sheet, Frequency Devices, January, 1996.

Frey, W., III, *Application of Inertial Sensors and Flux-Gate Magnetometer to Real-Time Human Body Motion Capture*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1996.

Fu, K.S., Gonzalez, R.C. and Lee, C.S.G., *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill, Inc., New York, 1987.

Gordon, R.L., *Acoustic Doppler Current Profiler, Principles of Operation, A Practical Primer*, 2nd Edition, RD Instruments, San Diego, Californiz, January 8, 1996.

Healey, A.J. "Evaluation of the NPS PHOENIX Autonomous Underwater Vehicle Hybrid Control System," *Proceedings of ACC'95 Conference*, Seattle, Washington, June, 1995.

Healey, A.J. and Lienard, D., "Multivariable Sliding Mode Control for Autonomous Diving and Steering of Unmanned Underwater Vehicles," *IEEE Journal of Oceanic Engineering*, vol. 18 no. 3, July, 1993.

Kuo, B.C., *Automatic Control Systems*, 7th Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

Kwak, S.H., Stevens, C.D., Clynch, J.R., McGhee, R.B., and Whalen, R.H., "An Experimental Investigation of GPS/INS Integration for Small AUV Navigation," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology (UUST)*, September 27-29, 1993, Durham, New Hampshire, pp. 239-251.

Leu, C.T., Chao, J.J., and Lee, T.S., "GPS Based Underwater Positioning - A System Design," *Proceedings of The Institute of Navigation GPS-93*, Salt Lake City, Utah, September 22-24, 1993, pp. 745-754.

Logsdon, T., *The Navstar Global Positioning System*, Van Nostrand Reinhold, New York, 1992.

Matthews, M.B., *A Description of the Hardware and Software Interface to the Systron-Donner MotionPak Inertial Sensor Unit for ROV Tiberon*, Monterey Bay Aquarium Research Institute Internal Correspondence, Monterey Bay Aquarium Research Institute, March 6, 1995.

MAXUS E.S.P 386sx/486slc Scamp II User's Manual, Maxus Electronics Corp., July, 1995.

McGhee, R.B., Clynch, J.R., Healey, S.H., Kwak, S.H., Brutzman, D.P., Yun, X.P., Norton, N.A., Whalen, R.H., Bachmann, E.R., Gay, D.L. and Schubert, W.R., "An Experimental Study of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," *Proceedings of the Ninth International Symposium on Unmanned Untethered Submersible Technology (UUST)*, September 25-27, 1995, Durham, New Hampshire.

McGhee, R.B., *CS4910 Lecture Notes: Derivation of SANS Filter Equations*, Naval Postgraduate School, Monterey, California, February, 1996.

McKeon, J.B., *Integration of GPS into a Small Underwater Navigation System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1992.

Nagengast, S., *Correction of Inertial Measurements Using GPS Update for Underwater Navigation*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

Norton, N.A., *Evaluation of Hardware and Software for a Small Autonomous Underwater Vehicle Navigation System (SANS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1994.

Oncore User's Guide, Motorola Inc., August 1995.

Parkinson, B.W., "Overview," *Global Positioning System*, Vol. 1, The Institute of Navigation, Washington, D.C., 1980, pp. 1-2.

Souen, K., and Nishida, T., "The World's Smallest 8-Channel GPS Receiver," *Proceedings of The Institute of Navigation GPS-92*, Albuquerque, New Mexico, September 16-18 1992, pp. 707-713.

Steven, R., *Simulation-Based Validation of Navigation Filter Software for a Shallow Water AUV Navigation System (SANS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1996.

Systron-Donner Model MP-GCCCQAAB MotionPaK IMU, Systron-Donner, Inc., Concord, California.

TCM2 Electronic Compass Module User's Manual, Precision Navigation, Inc., June, 1995.

Tuohy, S.T., Patrikalakis, N.M., Leonard, J.J., Bellingham, J.G., and Chryssostomidis, C., "AUV Navigation Using Geophysical Maps with Uncertainty," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology (UUST)*, Durham, New Hampshire, September 27-29 1993, pp. 265-276.

Walker, R.G., *Design and Evaluation of an Integrated, Self-Contained GPS/INS Shallow-Water AUV Navigation System (SANS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, June, 1996.

White, D.G. and Psota, F., "A Precision Navigation System for Autonomous Undersea Vehicles," *Proceedings of the 1996 IEEE Symposium on Autonomous Underwater Vehicle Technology*, June 2-6, 1996, Monterey, California, pp. 262-267.

Wolf, R., Hein, G.W., Eissfeller, B., and Loehnert, E., "An Integrated Low Cost GPS/INS Attitude Determination and Position Location System," Institute of Geodesy and Navigation (IfEN), University FAF Munich, D-85577, Neubiberg, 1996.

Wooden, W. H., "NAVSTAR Global Positioning System: 1985," *Proceedings of the First International Symposium on Precise Positioning with the Global Positioning System*, April 1985, pp. 23-32.

Youngberg, J.W., "A Novel Method for Extending GPS to Underwater Applications," *NAVIGATION, Journal of The Institute of Navigation*, vol. 38, no. 3, Fall 1991.

Yuh, J., *Underwater Robotic Vehicles: Design and Control*, TSI Press, Albuquerque, New Mexico, 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 013 2
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

4. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Dr. Robert B. McGhee, Code CS/Mz 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Eric Bachmann, Code CS/Bc 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. LCDR Ricky Roberts 2
   1081 Calma Dr.
   Chula Vista, CA 91910

8. Russ Whalen, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

9. Dr. Anthony J. Healey, Code ME/Hy 1
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

10. ECJ6-NP 1
HQ USEUCOM
Unit 30400 Box 1000
APO AE 09128