55

GUIDANCE, NAVIGATION AND CONTROL OF A ROBOTIC FISH

by

Mohan Gurunathan

Submitted to the Department of Electrical Engineering and Computer Science, in partial fulfillment of the requirements for the degrees of
BACHELOR OF SCIENCE IN ELECTRICAL SCIENCE AND ENGINEERING
and
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1998

Author: ....
Mohan Gurunathan
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Certified by: .... [illegible]
Jamie M. Anderson
Senior Member Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by: .... [illegible]
John J. Leonard
Professor in Ocean Engineering, Massachusetts Institute of Technology
Thesis Supervisor

Accepted by: .... [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Eng

GUIDANCE, NAVIGATION AND CONTROL OF A ROBOTIC FISH

by

Mohan Gurunathan

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1998, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science.

Abstract

This thesis describes the development of specialized electronics for a robotic fish at the Charles Stark Draper Laboratory in Cambridge, MA. The construction of this prototype vehicle has required the expertise and interaction of several different disciplines, including mechanical, electrical and ocean engineering. In particular, the complete electrical system of the vehicle is responsible for gathering data from sensors, controlling actuators which drive the tail, keeping track of the current state of the vehicle (including its current position), monitoring its safety, and communicating with the outside world. This thesis focuses on a small subset of the electrical system that is concerned with navigation, guidance and control. These issues encompass the vehicle's knowledge of its position at a given time, and its ability to coordinate actions so as to intelligently change its position. Here, an effort is described to develop a groundwork for the guidance, navigation and control electronic subsystem of the Vorticity Control Unmanned Undersea Vehicle (VCUUV) at Draper Lab.

Thesis Supervisor:	Jamie M. Anderson
Title:	Senior Member Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor:	John J. Leonard
Title:	Assistant Professor of Ocean Engineering, Massachusetts Institute of Technology

Acknowledgments

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under Internal Research and Development funding.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

Mohan Gurunathan

Many a time I went to fellow students in the Unmanned Vehicle Lab to verify my sanity (or just to distract me from going insane); all of you who helped, one time or another, include Jamie Cho, Alan DiPietro, Ryan Norris, Tom Trapp, Rusty Sammon, Long Phan, Bill Kaliardos, Paul Marquardt, John Thele, Chuck Tung, Jonah Peskin and Chris Gadda. In particular, Jamie Cho taught me a great deal about electronics and software when I was just a young tyke starting out on this project. Long and Rusty are experts in applied stress-relief. Jonah and Chris continue to work on the fish and I wish you best of luck.

Thank you in particular Mark Little for your help in decision-making, debugging, and regularly tolerating my "pessimism." Pete Kerrebrock was always willing share his engineering expertise and ingenuity in *any* discussion, to help solve all sorts of problems. Thank you Jamie Anderson for giving me the opportunity to work on this project and for your sensitivity to my own goals and pursuits over the last year. I have learnt a great deal this year through my research, as well as my interactions with everyone in the lab.

Elsewhere at Draper Lab, Tom Thorvaldssen and Pat Brown were gracious enough to sit down with me and explain some basics of strapdown inertial systems; Pat also donated some code to my cause. Your selfless help was much appreciated.

Many thanks to M.I.T. for helping to perfect the art of the all-nighter, without which I would never have finished this thesis.

This thesis is devoted to my parents for their continuous love, support and inspiration of 22 years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 The VCUUV project

The Charles Stark Draper Laboratory is currently in the process of developing an autonomous free-swimming robot fish to investigate the feasibility of fish-swimming motion as compared to conventional underwater propulsion techniques. This project, known as the *Vorticity Control Unmanned Underwater Vehicle* (VCUUV), is the successor to a 1996 MIT project called *Robotuna.* Robotuna consisted of a flexible-hull vehicle shaped like a tuna that was suspended in a tank by an overhead towing sled, which allowed movement of the fish in one dimension. The hull could be articulated to imitate the motions of a fish, and measurements of propulsive efficiencies could be taken while towing the fish through the water. In particular, the project demonstrated that a man-made vehicle could obtain increased propulsive efficiency via fish-like propulsion. Furthermore, Robotuna suggested that drag reduction and greatly increased maneuverability may also be achieved by the same swimming motion.

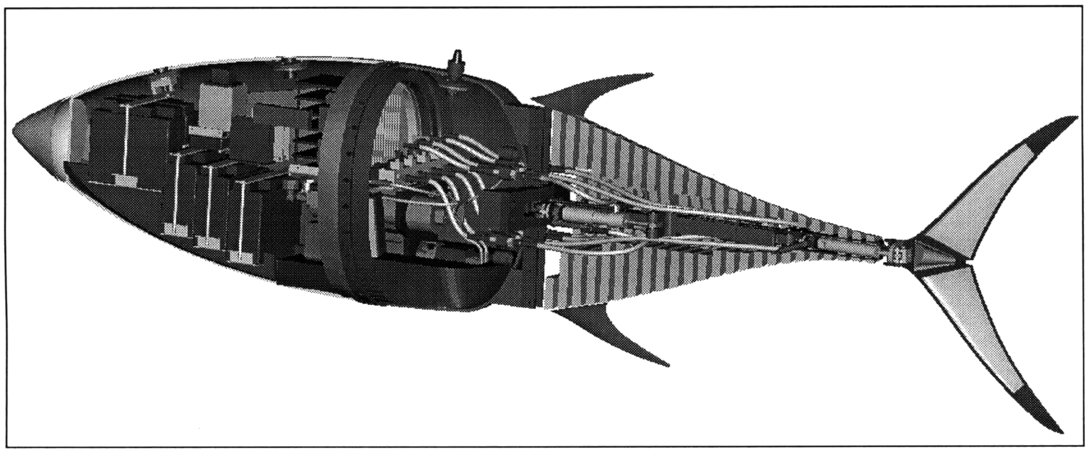


Figure 1.1: Vorticity Control Unmanned Undersea Vehicle, CAD model

The VCUUV is the next step in the study of fish propulsion. The primary goal of the project is to develop a vehicle that can be used to investigate self-propelled swimming and maneuvering. Unlike Robotuna, which was constrained by its towing sled to move along one axis, the VCUUV will be free-swimming and autonomous, features which present many additional challenges in its design. All computers, power supplies, sensors and actuators must be self-contained within the vehicle, placing a fundamental space constraint on the number of components and the heat they can dissipate. Since the vehicle is battery powered, an effort towards power conservation must be made in electronics design in order to maximize the mission duration. The vehicle itself must be neutrally buoyant, and its rigid body dynamics (i.e. centers of gravity, buoyancy, etc.) must be well known and characterized.

Some possible applications of such a vehicle include delivering underwater payloads in hard-to-navigate areas on the ocean floor, or removing mines from underwater minefields. However, these applications are far in the future; the goal for this project is simply a proof of concept: to demonstrate that a man-made vehicle can be made to swim like a fish. If VCUUV lives up to its designers' hopes, it will be more efficient and *much* more maneuverable than propeller-driven subs of comparable size [2].

1.2 VCUUV Electronics Subsystem

In this modern era of high-technology, a good electronics design is essential to the successful performance of any vehicle. An autonomous vehicle such as the VCUUV presents some very unique challenges, since it must eventually function with no human intervention – requiring sophisticated and robust development in the areas of safety and guidance.

The complete high-level design of the VCUUV electronics subsystem was done by Jamie Cho, and is the subject of his Master's thesis, "Electronic Subsystems of a Free-Swimming Robotic Fish [3]." The result of his efforts are summarized in the VCUUV architecture diagram in Figure 1.2. Jamie was responsible for specifying all of the electronic hardware necessary to make the fish operational; in essence, his work defines the capabilities of the current design. Despite various superficial changes, the general architecture has remained similar to his initial conception. His work included choosing the sensors and hardware responsible for the Navigation Subsystem of the electronics, which is pictured in the lower part of Figure 1.2.

The VCUUV uses a distributed-processing architecture. There are four separate micro-controllers that control and interact with various local subsystems of the vehicle. In addition, communication must exist between all of the processors. This occurs through the central "host" processor, the 486 on the PC-104 stack. All coordination of the activities and information from the other processors is handled by software running on the 486.

The remaining processors include a PIC microcontroller, a TI-C44 DSP (Digital Signal Processor), and a Tattletale microcontroller. Each device is assigned a specific region of responsibility. The PIC holds a vigil on the internal safety of the fish's electronics, by monitoring leak sensors, temperature and battery monitors. The DSP is dedicated to closed-loop motion control of the four hydraulic cylinders that make the tail move in a "fish-like" manner. The Tattletale is the central hub of the Navigation Subsystem. This subsystem is the primary focus of Chapter 3.

Various individuals were responsible for different sections of the electronics design. Jamie Cho designed the architecture, Chris Gadda and Jonah Peskin worked on the fish's safety features as well as software to actuate the tail. Mark Little, a Draper Lab engineer, designed a power management board and has helped integrate all of the electronics into a tightly functioning system. This work still continues, particularly in the software development arena, where much remains to be done. Many fragments of test code need to be combined and refined in order to truly make the fish an "autonomous vehicle."

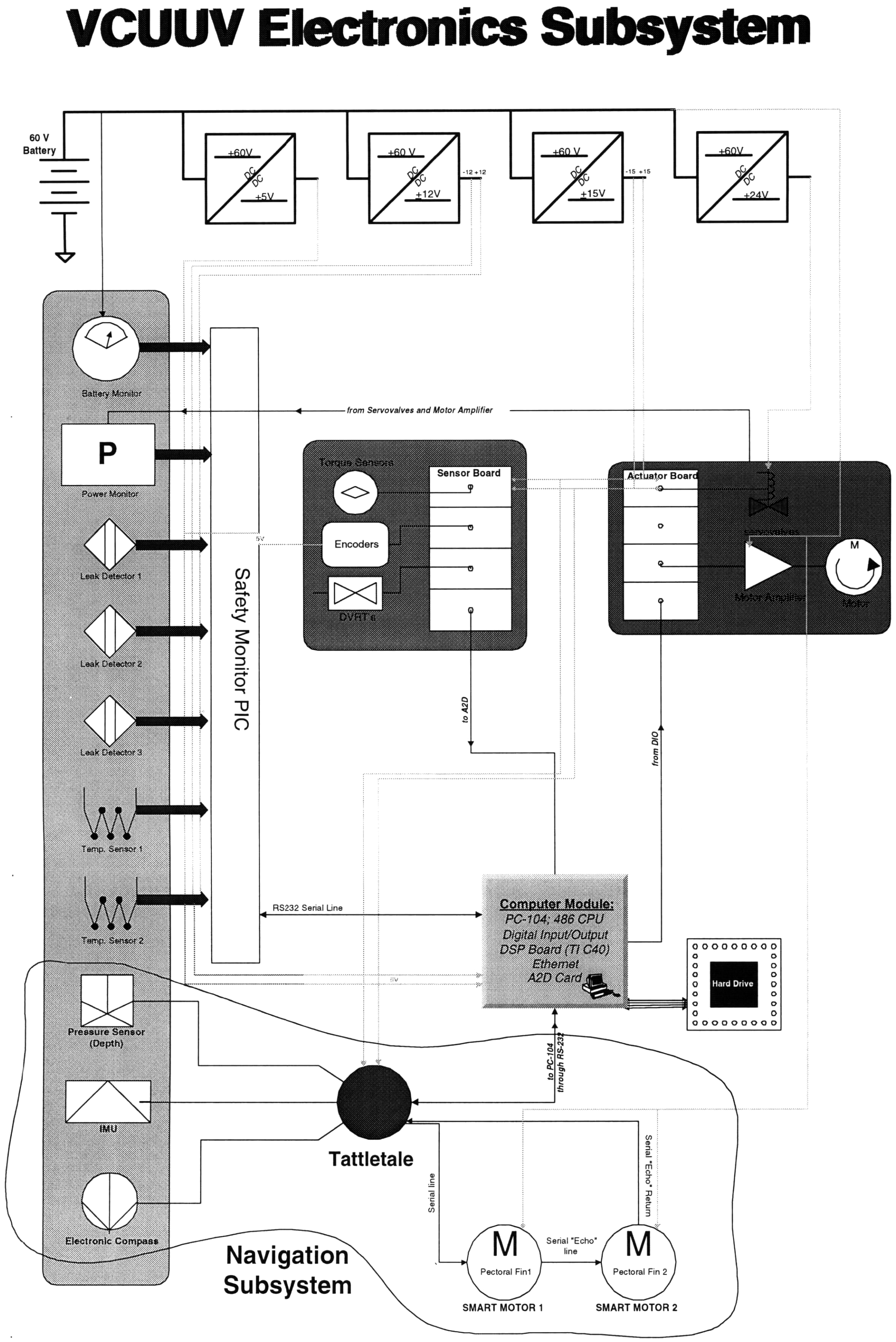


Figure 1.2: Complete VCUUV Electronics Diagram

The Navigation Subsystem was largely developed externally to the rest of the vehicle. The electronics and software design for the Tattletale controller were designed using a separate PC as the host computer; similarly, capabilities to communicate with the 486 through a serial port were developed (in software) on a laptop that operates almost identically to the 486 on the fish. At the publication of this thesis, the navigation work has been successfully demonstrated outside of the fish; current work involves integrating this small system into the already bustling electronic chassis.

1.3 Objectives of this Thesis

The VCUUV has been in development for over a year and a half, and its mechanical and electrical systems have recently been completed. The next stage is a series of experimental swims in a large tank at the University of New Hampshire, where data reflecting the swimming behaviors of the fish can be collected and analyzed. What remains to be proven is: will flapping the tail make the fish swim as expected?

Until this point, therefore, creating a vehicle that can propel itself via fish-like motion has been the primary concern. Navigating such a vehicle has been a secondary goal; but if the fish is demonstrated to swim well, guidance and control will become the next biggest challenge.

This thesis documents the design of the VCUUV guidance and control electronics, from initial conception to their current state. Hopefully the reader will get a sense not only of the system's capabilities, but the challenges that constrain these capabilities. Chapter 2 presents research done into general guidance and navigation principles, and how they might be implemented on the VCUUV. Much of this work has been done without the benefit of a proper testbed – i.e., a swimming tank in which to run navigation experiments – and so is described in terms of theory and speculation. This is intended to provide direction for future team members who might wish to continue development of the Navigation Subsystem. Chapter 3 describes actual electronics that have been designed and built, including sensors, hardware and software development. Chapter 4 analyzes the performance of the existing system, using experimental data collected from the electronics. Chapter 5 explores a simplified modeling and control experiment to regulate the fish's depth during a swim. Chapter 6 summarizes the research to date and makes recommendations for future work.

The work presented in this thesis is intended to lay a groundwork for the guidance and navigation of the VCUUV; however, by no means have all the capabilities been explored. Several of the ideas proposed here cannot be completed until the dynamics and swimming behaviors of the fish are better characterized. Numerous mathematical models should eventually be created in order to design well-constructed feedback routines to navigate the fish. With this in consideration, the central aim of this thesis is to thoroughly examine the fish's navigational potential based on the current electronics. From this point many possibilities may grow – largely through continued software development.

Chapter 2 Guidance and Navigation

Technically, navigation is the process of determining one's location with respect to some reference (usually a coordinate system). Guidance involves acting in ways to bring one to a desired location. Therefore navigation is a critical part of achieving guidance: it is the feedback element. Most methods of vehicle guidance depend on the ability to navigate, since a vehicle cannot accurately move from place to place without knowing that it has arrived at each place, and with what accuracy [7].

This chapter introduces some very general principles of navigation, in order to acquaint the reader with the issues involved. Special attention is given to *inertial navigation*, a dead-reckoning technique which estimates vehicle position by integrating incremental changes in the vehicle's accelerations and rotations. Inertial navigation is a very sophisticated technique which is discussed in deep mathematical rigor in many texts; this chapter will give a general discussion of its fundamentals and possible applications to the VCUUV. Chapter 4 presents some of the math and examines the VCUUV inertial system in greater detail.

2.1 Dead-Reckoning

The term dead-reckoning can be applied to any vehicular navigation system which attempts to estimate its position using only information that is internal to the vehicle itself. A good example of this is the odometer in an automobile, which counts the turning of the car's wheels. For travel in a given direction, the odometer reading gives the car's distance in miles from the starting point. The driver might also determine his position by reading a mile marker on the highway, but this would not be considered dead-reckoning: the distinction is that the odometer is a sensor (technically, an "encoder") which provides information about an internal property of the vehicle, versus the mile marker which is a source of information from the external world.

Other examples of non-dead-reckoning sensors are compasses and altimeters, which rely on external phenomena – the earth's magnetic field, and the atmospheric pressure – for their measurements. The importance of this classification is clearly seen if the external world were to go completely haywire; for instance if the earth's magnetic field were to reverse itself, compass measurements would become duplicitous. In the example above, if someone were to deceptively re-paint all of the highway mile markers, the only trustworthy indication of position would be on the odometer. In a sense, dead-reckoning is equivalent to self-sufficiency. The only reliable sensors *under all external circumstances* are instruments whose measurements do not rely on the constancy of the external world.

Unfortunately, few vehicles can afford to rely on dead-reckoning and ignore the external world for very long. This is mainly because dead-reckoning techniques are imperfect. Returning to the car/odometer example: after the car travels in a straight line for a long time, the odometer reading will eventually accumulate errors due to effects such as wheel slippage and tire wear. Sooner or later the driver will need to look at a mile marker to figure the actual distance traveled, since the odometer reading will have "drifted" from the correct distance. In general, this is an unfortunate property of dead-reckoning techniques: after some amount of time, the dead-reckoned position estimate will have strayed from the true position due to shortcomings of the sensors. At such a time, the position estimate must be revised by glancing at the external world for supplementary information.

Often, complete navigation and guidance systems are based on such a fusion of dead-reckoning and external information. Dead-reckoning is used to track position for short intervals of time, and in between, periodic corrections are provided by external sensors. The dead-reckoning information is often called "short-term" or "high-frequency" data since it is guaranteed accurate for a bounded intervals of time, and the external information is called "long-term" or "low-frequency" data since it is used less often, and only to correct the drifting errors of the dead-reckoned estimate. Ideally, a navigation system will be as self-sufficient as possible and rely almost entirely on dead-reckoning sensors; however, this is only possible with near-perfect sensors, which come at prohibitive cost. The greatest example of this is the ballistic missile, which can guide itself to a target thousands of miles away with no communication or observation of the external world, using inertial dead-reckoning techniques alone.

2.2 Inertial Guidance Theory

The ballistic missile provides a good first example of inertial dead-reckoning. For vehicles that travel in three dimensions and have no spinning wheels, encoder-type sensors (such as odometers) cannot be used to track position. Vehicles traveling through space must rely on a completely different type of dead-reckoning: inertial dead-reckoning. This is built on the fundamental principle that bodies of finite mass experience forces when they are accelerated (in an inertial reference frame). These forces can be measured, and therefore the accelerations can be measured. Similarly, when bodies are rotated around an axis, centripetal forces arise that can be measured, which reflect the direction and speed of the rotation. These two principles form the basis for inertial guidance theory, which states that the changing position and orientation (or *attitude*) of a vehicle can be calculated through measurements of its accelerations and angular rates of rotation. Essentially, by adding up – or integrating – incremental accelerations, velocity information is obtained; and integrating velocity gives position. The angular rates are needed to determine which directions the accelerations are being measured in. The instruments needed to measure linear acceleration and angular velocities are, respectively, accelerometers and gyroscopes [7].

Accelerometers and gyroscopes have been used in inertial navigation systems since the 1950's. They began to gain popularity as guidance instruments for the early ballistic missiles of World War II. Inertial navigation was not possible until these instruments were made much more precise than they had been; during WWII, Charles Stark Draper of MIT pioneered the movement which made gyroscopes and accelerometers orders of magnitude more accurate than the crude pre-war devices. Draper continued to push the envelope through the Apollo era, where he built the inertial instruments that took the first astronauts to the moon – and into the present day, where inertial instruments are so precise and so well characterized that they can pilot a missile into your backyard launched from any point on the surface of the earth.

Many types of accelerometers and gyroscopes exist, from simple springs and spinning masses, to vibrating quartz tuning forks and electrostatically suspended spinning beryllium spheres. This thesis will not go into the details of these various incarnations, but will suffice to say that in general they obey the basic law of cost versus accuracy and complexity. There are sometimes other tradeoffs that make certain instruments more suitable for certain applications – such as varying susceptibilities to different types of instrument errors – but the cost/accuracy one is the most predominant.

A full inertial guidance system uses three accelerometers and three gyroscopes, and is thus known as a six degree of freedom (6-DOF) system. The accelerometers are mounted with their sensitive axes pointing in three orthogonal directions, as shown in Figure 2.1. The gyros are oriented so as to measure angular rates around the accelerometer axes.

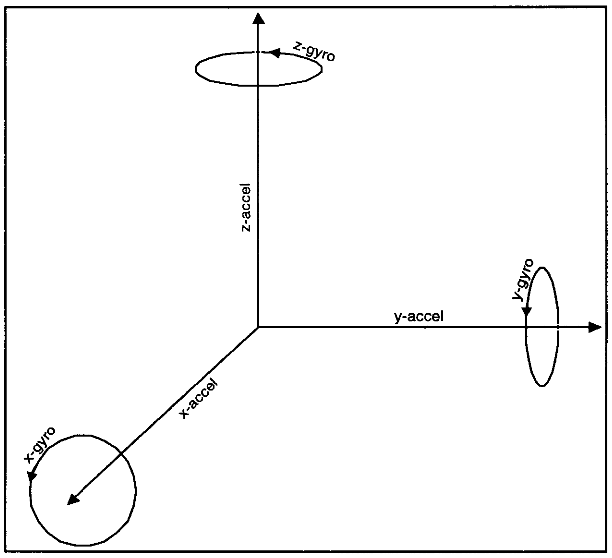


Figure 2.1: Accelerometer and Gyro sensitive axes in a 6-DOF system

The raw information in an inertial system consists of accelerations and angular rates. In order to compute our position as a function of time, we need to know our starting position, velocity, and orientation. This is a consequence of acceleration being the second-derivative of position (leaving position and velocity as the unknown initial conditions), and angular rate the first-derivative of angular position.

2.3 Inertial Systems

2.3.1 Platform vs. Strapdown

There are essentially two varieties of inertial systems: platform or strapdown. The difference is that in a platform system, the gyros and accelerometers are mounted on a platform which is mechanically "isolated" from the rotational movements of the vehicle through a set of gimbals. In other words, the attitude of the platform (pitch, roll and yaw angles) can change independently of the vehicle's attitude. In a strapdown system, the inertial components are rigidly mounted (or "strapped down") to the vehicle's body, and are forced to experience all attitude changes with the vehicle. The mechanics of each system are illustrated in Figures 2.2 and 2.3. This results in a fundamental difference of operation between the two types of systems, as well as a tradeoff between size, cost, power consumption, error performance, and mechanical versus computational complexity [7].

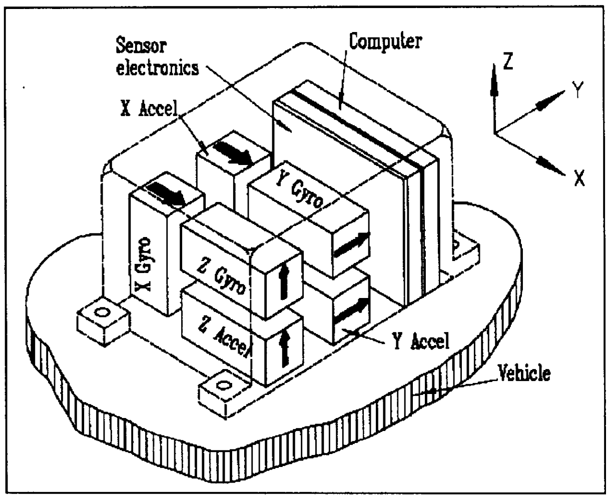


Figure 2.2: Strapdown inertial system
(taken from Lawrence [7])

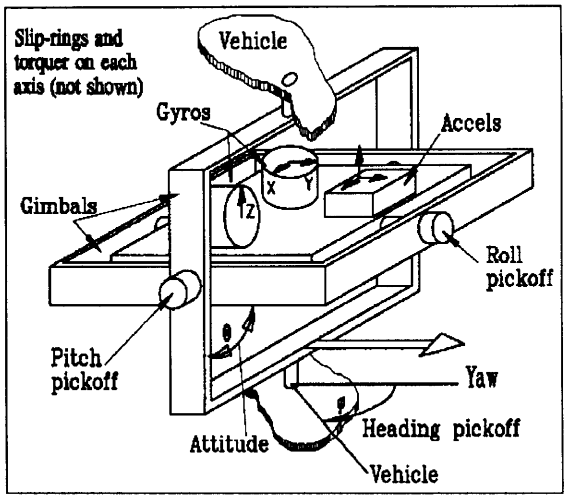
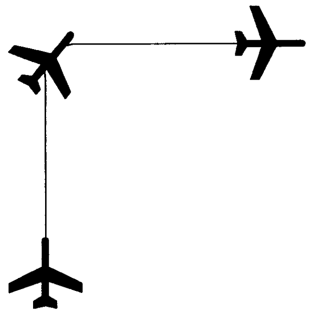


Figure 2.3: Platform inertial system
(taken from Lawrence [7])

The primary difference between platform and strapdown systems is how they solve the problem of the changing orientations of the vehicle. A platform system is mechanically designed so that *the orientation of the platform in inertial space is constant.* This is arranged by a feedback system with the rate gyros: when the gyros sense that the attitude of the vehicle is changing, they send command signals to torque motors which counter this change in attitude to keep the platform's spatial orientation fixed. Thus if the vehicle were to turn completely upside down, the platform would spin 180° to maintain its original orientation. The advantage of this in navigation is best demonstrated through an example.



Consider the following (two-dimensional) scenario: a plane with an inertial navigation system (INS) accelerates forward, makes a 90° turn to the right, and accelerates forward again. If the INS is a strapdown system, the first acceleration is picked up on the x-axis accelerometer, then a positive turning rate is seen on the yaw-rate gyro, and then another acceleration is seen on the x-accelerometer. In a platform system, the first acceleration would also be picked up by the x-accelerometer, but the yaw-rate gyro would command the platform to *counter-rotate* 90° while the plane executes its turn, so that the second acceleration is picked up by the y-accelerometer. In this way the platform's orientation in space always remains fixed (i.e. x- and y-accelerometers can always thought to be pointing north and east, respectively).

Given these sensor outputs, how would we compute the plane's position? In the platform system, this is easy: we have acceleration signals in the x-direction and the y-direction, and can integrate these twice to compute an (x,y) position in space – a simple task for an onboard computer. Note that the change of attitude has already been taken care of by the mechanical compensation of the platform. In the strapdown system, however, things are not so simple. The gyro outputs must be fed to the computer, and must be dealt with by number-crunching rather than mechanics. The onboard computer has the additional burden of computing continuous coordinate transformations in order to determine which direction in space the x- and y- accelerometers are pointing at all times. In essence, a strapdown computer must emulate through software the mechanics of a fixed-attitude platform – a "virtual platform" must be implemented. The process of implementing this virtual platform is also known as *resolving* the strapdown measurements into a reference frame, and is discussed in detail in Chapter 4.

An added complication is the presence of gravity. To an accelerometer, the force of gravity is indistinguishable from physical acceleration: a stationary accelerometer lying on a table with its axis aligned with gravity will read 1*g* of acceleration. Thus any inertial system must know exactly what the magnitude and direction of gravity are at all times, in order to subtract it out from the reading. In a platform system, this is relatively simple, since the platform can be initially aligned perpendicular to gravity: then the z-axis will always see a 1*g* offset (since the platform's orientation will stay fixed)[1]. In a strapdown system, the situation is again more complex: the orientation of the accelerometers will constantly be changing with respect to gravity. Based on the computer's current estimate of this orientation, it must subtract out the appropriate component of gravity from each of the accelerometer readings. In precision inertial systems (strapdown or platform), compensating for gravity is a major problem due to many non-uniform properties of the earth's gravitational field. Complex gravity models are needed that account for the earth's bulge near the equator, its varying mass distribution, and so on.

The fundamental tradeoff between choosing a platform or strapdown system is

[1] However, if the vehicle travels large distances over the earth (such as a ballistic missile), the platform will no longer be perpendicular to gravity (if aligned in the U.S. and traveled to China, the platform will be upside down!). The technique for continually compensating for the earth's curvature during travel is called Schuler tuning, and is a detail not discussed in this thesis.

computational versus mechanical complexity. Either a precisely constructed (and thus more fragile) gimbaled structure with an analog attitude-control system is necessary, or a relatively fast computer with the sophisticated software necessary to carry out all the strapdown computations (in addition to other tasks it might need to perform, such as vehicle control). Strapdown components are cheaper, lighter and more mechanically robust than their platform cousins. Strapdown gyroscopes, however, require a much greater dynamic input range than platform gyros, since platform gyros only experience incremental rotations. This is one of the reasons why platform systems have superior accuracy over strapdown: characterizing their gyros completely (errors, temperature stability, etc.) and ensuring linearity is much easier over a small dynamic range than the large input range of strapdown gyros. As it turns out, inertial navigation accuracy over time all depends on how well every aspect of the inertial instruments themselves can be characterized.

2.3.2 Drift

Since it is a dead-reckoning technique, inertial navigation experiences drift. The nature of this drift is amplified by the mathematics involved: accelerations are integrated twice to determine position. Consider a small constant error in an acceleration measurement, known as a *bias*. Over time, this is integrated twice and leads to a quadratic growth in error versus time (i.e., error proportional to time-squared)! Gyro errors, as it turns out, cause even worse overall error in the position estimate, since they distort the attitude of the vehicle, and skew the accelerations from the actual directions they are measured in. Thus gyro error begets acceleration error which in turn begets quadratic position error – so in the end, gyro biases cause position drift that grows roughly proportional to the cube of time. In order for an inertial system to navigate correctly for any useful amount of time, all of its errors must be well known and compensated for. The best inertial systems characterize all of the instruments to a very high order, including scale factors, temperature and bias stability, noise characteristics, angular misalignments, nonlinearities, gravity-dependent errors, and many more very subtle sources of error. Nothing is left to chance. Specification sheets for a single accelerometer or gyro can span several hundred pages[2]! Furthermore, systems that navigate large distances across the surface of the earth need a good model of its gravity field, as well as its size, shape, turning rate, etc. Once these parameters are known, and *assuming the errors are systematic and repeatable*, instrument readings can be compensated in software to give the most precise acceleration and angular rate values possible – therefore the lowest drift.

2.3.3 External Sensors

Some external sensors that are typically used to augment inertial guidance are compasses, velocity sensors (i.e. air-speed detectors), radar or sonar. More often, a precise fix on location can come from an external reference such as the Global Positioning System (GPS), or by star-sighting (which has been used by sailors for centuries). Systems which use a combination of INS and external navigation aids are known as *integrated* systems.

[2] As one can imagine, inertial testing equipment is extremely sophisticated and well-controlled, including instruments such as precision centrifuges, rate tables, vibration ("shaker") tables, dividing heads, etc.

However, in some applications – particularly military, where GPS or radio might be jammed by an enemy – reliance on external sources is undesirable. In these cases, the vehicle's navigation over time is only as good as the sensors it carries with it: this is true "dead-reckoning." For this reason many military vehicles (submarines, ballistic missiles, air and space-craft) are designed with million-dollar inertial systems that have incredibly low drift rates. These are vehicles designed to rely on nothing but themselves to bring them to their targets.

2.4 Uses of Inertial Information in Guidance and Control

2.4.1 Short-term attitude stabilization and alignment

Not all guidance, navigation and control systems use inertial instruments to continuously compute an estimate of the system's position. In fact, some systems use gyros and accelerometers for short-term, local maneuvers, without much care for the long-term position of the vehicle. For instance, the VCUUV (which carries a small strapdown inertial system) might use feedback from gyroscopes to determine whether it is rolling or yawing undesirably, and adjust fin angles and the tail-bias[3] to stabilize its attitude (or "swim straight"). Such momentary attitude-stabilization maneuvers are easier for a computer to implement than a full navigation algorithm which continuously calculates the vehicle's position and attitude in real-time. Furthermore, since the drift of a long-term navigation algorithm will get very bad after some length of time, a series of short-term maneuvers might be a more intelligent method of guidance. Consider the following thought experiment, where we assume the fish has the capability to brake and to turn 90° in place[4].

We desire the fish to swim straight during a rectangular lap around the pool, without rolling or yawing during the mission. If a continuous, long-term navigation routine is used, the estimate of the vehicle's attitude would slowly drift during the whole mission; by the time it completes one lap, it may have rolled over on its back, and yawed enough to miss its destination by a significant margin. However, using a short-term attitude-stabilization[5] scheme offers the possibility to zero-out the accumulated gyro errors at regular intervals. This is done as follows:

This fish swims from the starting corner to the second using a short-term feedback scheme from two gyros to stabilize roll and yaw. The fish brakes to a halt at the second corner, and

[3] The fish can control roll-movement by holding its pectoral fins at unequal angles. Yawing can be controlled by swimming with the tail centered slightly to the right or left of the central axis of the fish body, or applying a "tail-bias."

[4] These assumptions are not true at this time but may be implemented in the future. In any case, they clarify the thought experiment.

[5] In fact, many modern airplane autopilot systems implement attitude-stabilization using feedback with strapdown gyros. Using gyros with sufficiently low drift, the plane can maintain a fixed roll, pitch and yaw for long periods of time. Such arrangements are called Attitude and Heading Reference Systems (AHRS).

due to an accumulation of errors in the feedback from the gyros, it has rolled and yawed slightly (same as in the long-term navigation scenario). However, if we know the fish has successfully braked to a halt, we can determine its actual attitude (relative to the vertical) by measuring the tug of gravity on each of the accelerometers. This is known as re-aligning the IMU. At each corner, the fish can brake to a stop, and replace the estimate of its attitude (which has drifted) with an exact attitude measurement from the accelerometers. This chain of short-term controlled runs should exhibit less drift than a system which is guided by a continuous estimate of position and attitude. The tradeoff is that we no longer have available an absolute estimate of our position (only an estimate relative to the corner where we last re-aligned the system). But if an absolute estimate will inevitably drift so far to be worthless, what good is it?

Thus, *whenever we know the vehicle is not accelerating*, we can re-align our knowledge of attitude by measuring the "tilt" of the IMU, or the effect of gravity on the accelerometers. Tilt re-alignment is very important, because in inertial systems the direction of gravity must always be known. The alignment algorithm requires some way to know the vehicle is not accelerating, i.e. information that is external to the inertial system (once again, the accelerometers cannot distinguish between gravity and true acceleration; something else needs to provide the knowledge that the vehicle is not accelerating). One possibility is that the fish decides that it is "stationary" – and thus not accelerating – when it has applied its brakes for a given amount of time. Another algorithm might calculate the variance of accelerometer signals taken over a period of one second. If the vehicle is truly experiencing motion accelerations, this variance is likely to be high (since motion accelerations are usually "jerky" or momentary in nature, rarely constant). If however the vehicle is not accelerating, all non-zero forces on the accelerometers must result from components of gravity which is static, so the variance will be low. From these accelerometer readings, the fish can exactly determine its present tilt and re-align itself with the direction of gravity.

These short-term techniques are a variation on traditional inertial navigation, which usually calculates the full set of position, velocity and attitude information in real-time within the feedback loop. However, in systems where the computer is not fast enough to compute all the strapdown calculations, or where the IMU's bias and noise characteristics are very poor, such short-term techniques might be advisable. They improve drift performance by re-zeroing the attitude error whenever the vehicle can be brought to a stop (or, any known state of acceleration). This method inherently relies on having a system that is able to make sophisticated and intelligent decisions independently of the inertial information, such as: when has the vehicle stopped accelerating?

Such "sophisticated" decision making, however, is really a simplified version of a very rigorous mathematical technique that is used to significantly reduce errors in inertial systems. Consider the task of deciding how long to apply braking before the vehicle has come to a stop. This will involve some knowledge of the mass of the vehicle, the hydrodynamic forces of the "brakes" in water, etc. Taking this idea to its logical extreme, suppose we can mathematically characterize not only the fish's braking abilities, but *all* aspects of its motion in water: the thrust and angular accelerations produced by swimming the tail at a certain amplitude and frequency, the lift on the pectoral fins, etc. With this mathematical model of the vehicle (an open-loop model), we can run a real-time simulation of the fish's motion as it swims. If we give the fish a command to beat five strokes of its tail with a negative pectoral fin angle, the mathematical model will generate a prediction in real-time of how far it will travel, and how deep it will dive. At the same time, the IMU will provide information on the observed motion of the vehicle. We now have predicted and observed motion information, and there exists a mathematical technique to fuse these separate estimates in an optimal way to generate the best combined estimate. This is known as Kalman filtering (or optimal-observer design), and is the subject of the next section.

2.4.2 Kalman Filter

The Kalman filter is a mathematical tool used to combine several imperfect measurements of a quantity into an "optimal" estimate of the quantity. It is often used in inertial navigation to generate a position estimate using measurements from inertial sensors and predictions from an open-loop model. In general, the Kalman filter is a tool for "fusing" or mixing data from different sources. Therefore it can also be used to combine data from inertial systems, GPS, altimeters, and any number of other sources, in an optimal fashion. It is called a *filter* because in combining the various measurements, it reduces the measurement errors or "noise" that are associated with each individual source of data, in a statistical sense.

Kalman filtering is an elaborate mathematical topic, so we will only describe it in generalities here. In order for the filter to be effective, the correlation between the various sources of data must be known. This essentially means that in order for the filter to combine two "noisy" estimates to generate a better estimate, the filter must know if the noise (or measurement error) on the first source is correlated with the noise on the second source. There are various issues with determining this correlation, injecting "stabilizing noise" and such; these are ignored in this general discussion. For more information, the reader is referred to Appendix A of Titterton and Weston [13].

The filter can be formulated in a number of ways, but the most intuitive is as a state-space observer. In a state-space system, an observer is a controller which is based on a model of the *plant* or process of interest. The observer essentially runs a real-time simulation of the process to predict what the output of the process will be, and then compares its prediction with the measured output of the process. The difference between the predicted and measured quantities is used in a feedback routine to correct the subsequent error of the predictive model.

In the VCUUV, the Kalman filter might be used as follows. A linearized model of the full hydrodynamic system is developed and written in state-space form. Since this Kalman filter is concerned with navigation, the state variables should describe the position and attitude of the vehicle. Thus, there should be at least 6 state variables, corresponding to three angles (attitude) and three positions. The state vector will probably have an additional 3 state variables for the first derivatives of the position variables. The system will also have inputs and outputs. The inputs to the system might include quantities such as tail beat frequency, amplitude, and pectoral fin angle. The outputs would include measurements that can be *observed*, such as IMU, compass and pressure sensor readings (as will be discussed in the next chapter, these are the basic navigational sensors for the VCUUV). A proposed Kalman filter architecture for the VCUUV, assuming a linear model can be developed for the plant, is shown in Figure 2.2 [13].

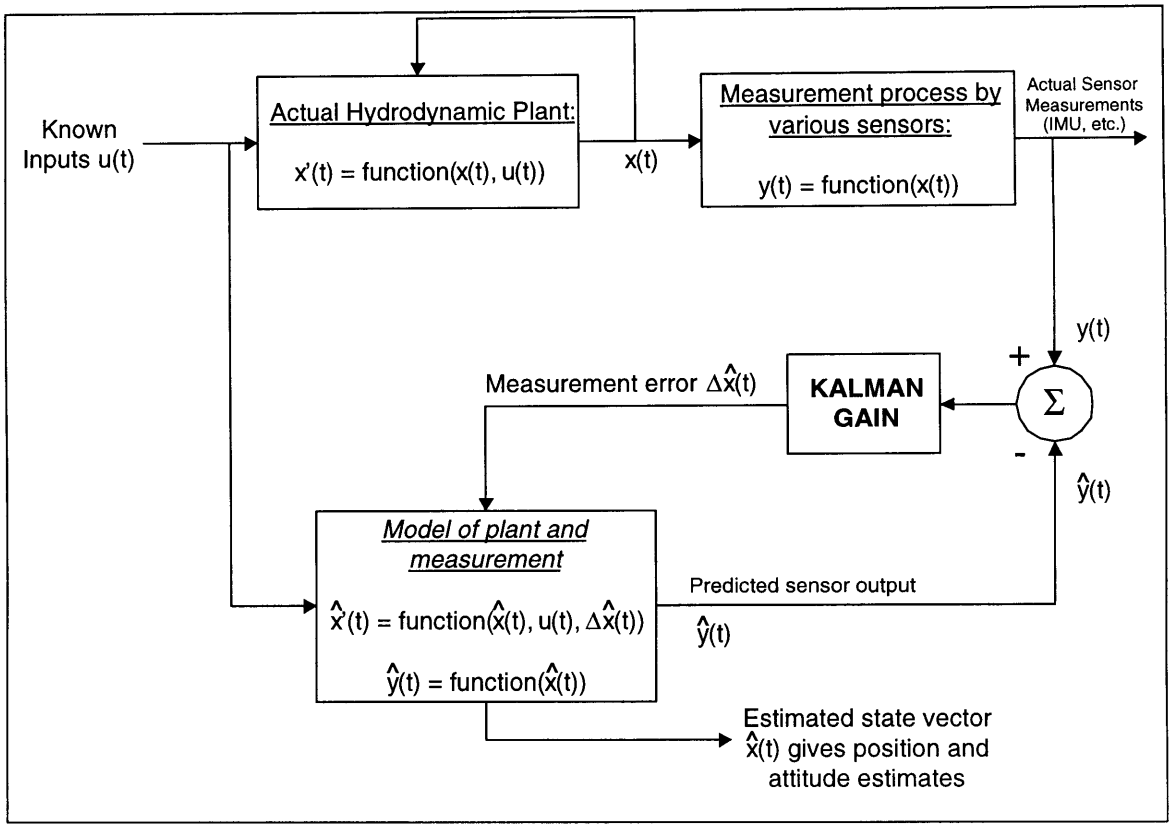

Known Inputs u(t)
Actual Hydrodynamic Plant:
x'(t) = function(x(t), u(t))
x(t)
Measurement process by various sensors:
y(t) = function(x(t))
Actual Sensor Measurements (IMU, etc.)
y(t)
+
Σ
-
ŷ(t)
KALMAN GAIN
Measurement error Δx̂(t)
Model of plant and measurement
x̂'(t) = function(x̂(t), u(t), Δx̂(t))
ŷ(t) = function(x̂(t))
Predicted sensor output
ŷ(t)
Estimated state vector x̂(t) gives position and attitude estimates

Figure 2.2: General Kalman filter architecture for VCUUV navigation

Here, the state vector $\mathbf{x}(t)$ contains the variables describing the true position, velocity, and angular rotation rates of the vehicle. The input vector $\mathbf{u}(t)$ contains variables describing the fish's tail and pectoral fin actuation. The output vector $\mathbf{y}(t)$ are the "noisy" measurements available to the external user, such as IMU and compass readings. The whole task of the filter is to fuse the observed measurements $\mathbf{y}(t)$ with the predicted response of the process model, in order to create an estimate of the state vector $\hat{\mathbf{x}}(t)$.

Note the block labeled "Kalman Gain." In control-system terminology, this is sometimes called the observer gain, but when this matrix is chosen in a specifically optimal way it is called the Kalman gain. This is a special gain matrix that is determined by various correleations between the "noise" characteristics of each of the sensor measurements. The details of choosing the Kalman gain matrix are beyond the intended scope of this thesis, but are widely available in the literature.

In presenting this generalized (linear) Kalman filter for the VCUUV we have left out many details, including specific choice of state variables, input and output variables, and the process model. Work on such a filter for the VCUUV has not yet begun, but the subject is presented here as a suggestion for future work along with the generalized algorithm. More complexity arises when we consider that hydrodynamic processes we wish to model are extremely nonlinear, and linearizing them around a nominal point of operation would destroy many useful properties of the equations. To deal with this, a variation on the above known as the *extended Kalman filter* must be employed. The details of the EKF are left for the reader to investigate.

2.5 Summary

The VCUUV is equipped with several navigational instruments, including a small strapdown IMU (this will be discussed in more detail in following chapters). This chapter has presented some of the theory behind using these sensors in a collective fashion to estimate a vehicle's position over time.

The observations of this chapter all reinforce the basic fact that inertial navigation is ideal for short-time or high-frequency navigation, and poor as a long-term navigational aid. Several algorithms specific to improving the drift performance of an inertial system were also discussed. Hopefully, these algorithms will eventually be developed for practical use on the VCUUV.

Chapter 3 Guidance Data Acquisition

All systems that interact with the external world require the power of observation: eyes and ears in humans, sensors in machines. In particular, most vehicle guidance and navigation systems rely heavily on information fed back from sensors. Sensor information generates an estimate of position, which in turn compels actions to bring the position to the desired state. Thus sensor accuracy is an important concern, and the circuitry that gathers data from the sensors becomes a critical path in a large control loop. Typically this data gathering might include scaling, level-shifting, filtering, and analog-to-digital (A-to-D) conversion of the sensor signal – all operations which are intrinsically imperfect. The goal of any good data acquisition system is to collect sensor data while corrupting this data as little as possible[6].

This chapter describes the sensors used for guidance and navigation on the VCUUV, as well as the design of custom hardware and software to interact with these sensors. This "Navigation Subsystem" is shown in relation to the entire electronics architecture in Figure 1.2. We include control of the pectoral fins as part of this subsystem, for several reasons. The fish will use its pectoral fins to control its depth, which is a navigational aim (at least in one dimension). Furthermore, control of the pectoral fins occurs entirely through the same local processing unit which talks to all of the navigation sensors, the Tattletale. Grouping all of the Tattletale's functional responsibilities as the "Navigation Subsystem" is a convenient categorization, and also sets the stage for an experiment in closed-loop depth control which will be discussed in Section 5.2.

3.1 Tattletale

The heart of the Navigation Subsystem is the **Tattletale Model 8** (TT8) board. This is a dedicated microcontroller board that is purchased commercially from Onset Computers, Inc. The TT8 is a credit-card sized board that comes complete with switching-voltage supplies, digital I/O, a 12-bit 8-channel A-to-D converter, a 68332 microprocessor, two RS-232 serial ports, and other features. Along with it is also provided a software development package that allows the user to write and compile C-programs directly onto the Tattletale, for ease of system development and debugging.

The Tattletale was chosen largely because of its on-board A-to-D converter and serial ports. These features give the TT8 the functional capabilities for talking to the navigation sensors, pectoral fin motors and the host computer (486). Furthermore, the TT8 provides the convenience of C-programmability, and can run programs stored in non-volatile Flash memory from powerup (a necessity).

The navigation electronics consist largely of signal-conditioning circuitry for analog data from

[6] This is particularly true for inertial systems, where errors in data acquisition get amplified through integration to become serious drift and random-walk errors.

the sensors. These include amplifiers, level-shifters and pre-sampling filters. Due to the tight space constraints of the fish's electronic chassis, these circuits were built by hand on protoboards that could be "piggy-backed" on top of the TT8 through header pins. The compact assembly of stacked boards is shown in Figure 3.1.

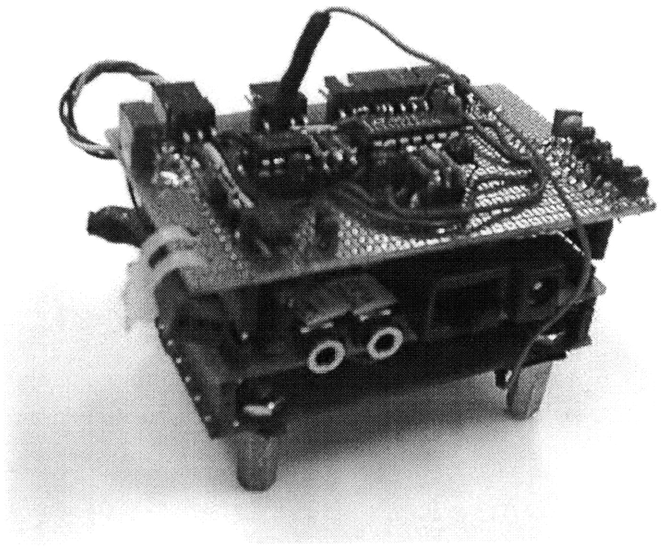


Figure 3.1: Tattletale Model 8 with custom circuitry

Details on critical issues such as programming and setting the TT8 up for operation can be found in Appendix B. A full schematic and interconnections diagram of the navigation circuitry are provided in Appendix A.

3.2 Sensors

The choice of sensors to aid the fish's guidance was made early in the design phase of the VCUUV. Given that this is a marine vehicle, several useful navigational aids were automatically eliminated, such as GPS and triangulation with radio beacons[7] – both of which are common tools for many terrestrial or airborne autonomous vehicles. The final suite of sensors includes a pressure sensor, a magnetic compass, and an Inertial Measurement Unit (IMU). The IMU is by far the most elaborate and expensive of the sensors, and its potential uses in guidance and attitude-stabilization is discussed at length in Chapter 2.

Other sensors that were considered included hydro-velocity flowmeters and proximity detectors (for obstacle avoidance). It was decided that neither of these would be very effective in a vehicle of this size. However, based on the current work some new ideas have come up that may help improve the navigational capabilities of the next generation fish; these are presented along with other conclusions in Chapter 6.

[7] Radio waves in general do not penetrate water more than a few feet.

3.2.1 Depth Pressure Transducer

The simplest method for making a depth measurement in an underwater vehicle is to measure the water pressure at that level. The sensor chosen for this task was the Model 93-015S Sealed Gauge pressure sensor, made by EG&G IC Sensors. The sensor has a linear range of 0-15 psi (corresponding to 33.75 feet of water), but can safely be loaded with three times that pressure. "Sealed gauge" means that the sensor will give its reading with respect to one atmosphere: at sea level the sensor should nominally give a reading of zero.

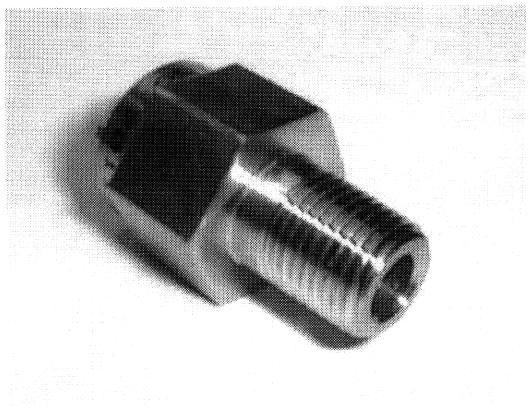


Figure 3.2: Model 93-015S Sensor

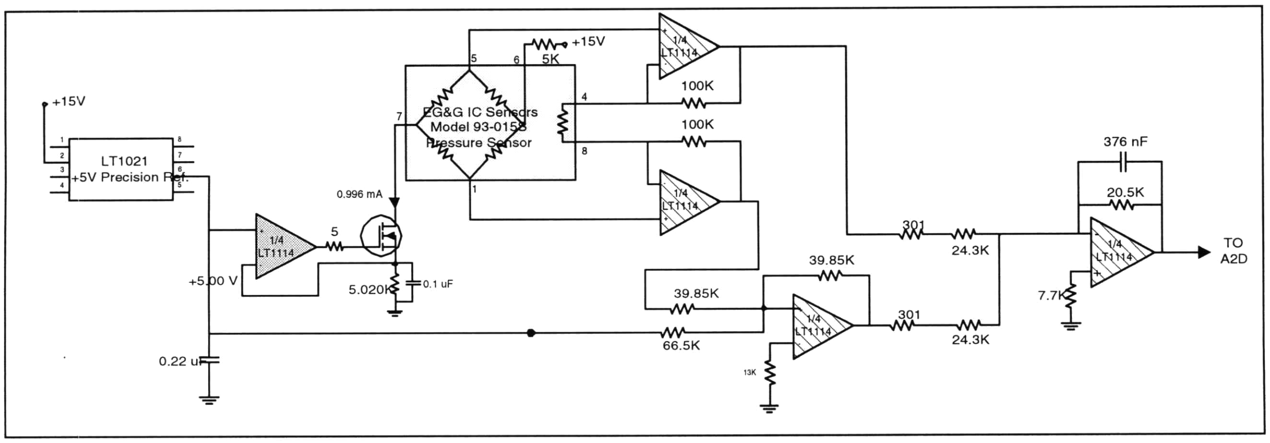


Figure 3.3: Depth pressure transducer circuitry

The sensor consists of a Wheatstone bridge with a pressure-dependent resistive element. To get a reading, the bridge is excited with a constant current of 0.996 mA, and a differential voltage read across the remaining two terminals. This voltage is further amplified and level-shifted by several op-amps before being delivered to the A-to-D converter. The depth-sensor circuitry is shown in Figure 3.3.

One concern with using a pressure sensor is that of daily air pressure variation: the weight of the atmosphere is never the same from day to day, and it is reflected in water pressure readings as an additive offset. Since it is just an offset error, we can calibrate the measurement by subtracting out the atmospheric pressure. For instance, take the fish to the surface, take a pressure reading, and subtract that reading from all readings henceforward (so that the surface reading becomes zero). This should be done before every new day that the fish goes swimming. Another more difficult problem is that the density of water will vary depending on its temperature, introducing variation into the pressure-vs.-depth scale factor. This problem might be fixed with an external temperature sensor to compensate the pressure readings. However, since the VCUUV does not need (extremely) precise knowledge of its depth, this problem was not tackled.

3.2.2 Compass

An electronic compass is used to obtain measurements of heading, or azimuth. The model used is the Vector 2XG, made by Precision Navigation Inc. This sensor is built upon the principle of a two-axis magnetometer, which measures the strength of a magnetic field in two orthogonal directions. In order to measure the earth's field and compute heading accurately, such a magnetometer must remain parallel to the earth's surface. On the V2XG, the coils which measure field strength are each mounted in free-swinging gimbals, so that they will remain parallel to the earth even if the compass is tilted (up to 15°).

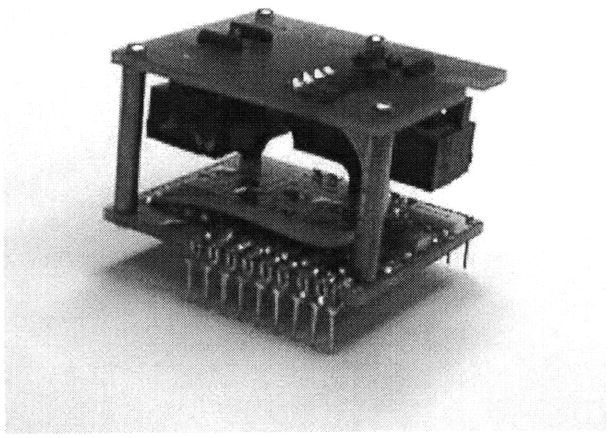


Figure 3.4: Vector 2XG Compass

Obtaining a reading from the compass is a purely digital task; the timing requirements are shown in Figure 3.5. The compass obeys a serial protocol known as QSPI, where it can act as either a "Master" or "Slave." We use it in Slave mode. In this mode the compass is polled for a reading, takes 80-100 milliseconds to make its measurement, and sends a signal high when the measurement is ready. Then the host system can serially clock out the 16 bits that carry the heading information. The compass interface is easily implemented using the TPU digital I/O

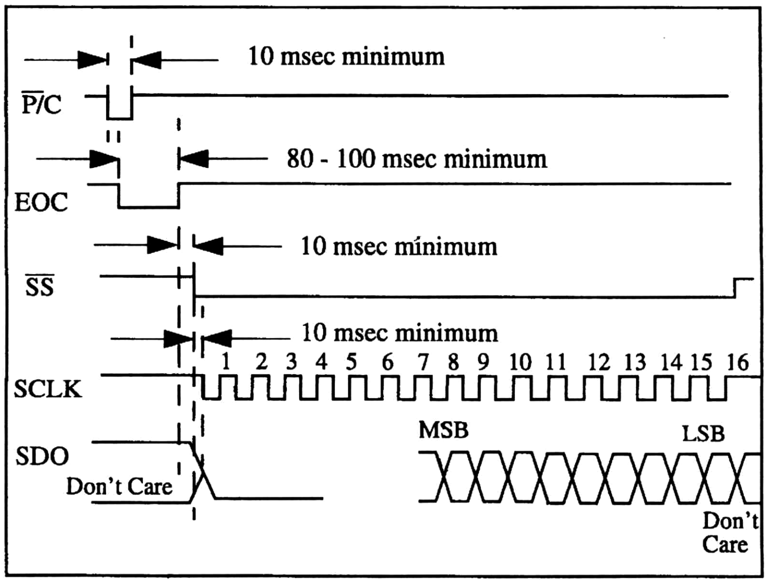


Figure 3.5: Compass timing diagram (from Vector 2XG manual)

lines on the Tattletale. On average, following all timing constraints, taking one reading requires about 200 milliseconds (during most of this time the compass is "thinking" and the host system can switch to other tasks), which allows a 5 Hz sample rate for compass data.

There are several problems associated with using the compass. First of all, the same gimbals which allow the coils to stay parallel to the earth will rock back and forth in response to vehicle acceleration: once again, gravity – which tugs at the gimbals – is indistinguishable from the accelerations of the vehicle. This may add significant errors to compass readings when the fish accelerates. The only possible future fix for this problem is changing the V2XG out for a three-axis magnetometer, which can compensate for tilt by measuring magnetic field in a third orthogonal direction, rather than using gimbals.

The compass also suffers when the fields around it are distorted by ferromagnetic materials (i.e. iron, steel, permanent magnets). Ferromagnetic distortion comes in two varieties: hard-iron and soft-iron distortions. Soft-iron is iron that cannot be permanently magnetized; this iron creates a non-uniform field distortion that varies with the strength and direction of the ambient fields. Thus, soft-iron distortion cannot be corrected. Hard-iron distortion is caused by types of iron and steel which retain a permanent magnetization. This is easily corrected for, since the distortion appears simply as an additive offset to the ambient (earth's) magnetic field. To calibrate the compass for hard-iron distortion within the fish, the CAL pin is pulsed once, then the fish (with the compass inside it) is swung 180°, and the CAL pin is pulsed again[8]. Experiments demonstrated that calibration was very effective in correcting local magnetic distortions near the fish. This calibration should be repeated every time the compass is powered up.

3.2.3 Inertial Measurement Unit

The IMU is the gem of the VCUUV navigation sensors. The device chosen for the VCUUV is built by Systron Donner and is known as the MotionPak. Inside this "black box" are the sensitive inertial components introduced in Chapter 2: three accelerometers and three rate gyros, mounted in an orthogonal set of axes. As the inertial components are rigidly fixed inside of the box, and the box will be rigidly mounted to the VCUUV, this is a strapdown inertial system. The case of the IMU serves to protect the sensitive instruments from dust and water, and can be connected to ground to shield the instruments from external radiation.

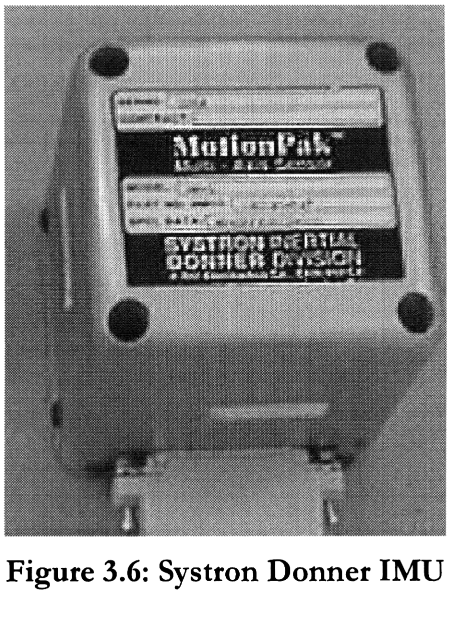


Figure 3.6: Systron Donner IMU

Systron Donner delivers each IMU with an individual calibration sheet, which lists characteristics (scale factors, biases, temperature

[8] Cruder calibration can be achieved by swinging the fish less than 180° between pulsing CAL. Calibration can be erased by pulsing CAL twice without moving the fish.

coefficients, etc.) specific to that particular device. The calibration sheet information for the fish IMU is shown in Table 3.1. We will continue to refer to the specs on this sheet throughout this section.

	Angular X- Axis	**Angular Y-Axis**	**Angular Z-Axis**	**Linear X-Axis**	**Linear Y-Axis**	**Linear Z-Axis**
Range	±200°/s	±200°/s	±500°/s	±5 g	±5 g	±5 g
Scale Factor	12.551 mV/°/s	12.518 mV/°/s	5.000 mV/°/s	1.503 V/g	1.503 V/g	1.504 V/g
Bias (@22°C) Temp Perform.	+0.16°/s < 3°/s	+0.07°/s < 3°/s	+0.09°/s < 3°/s	+3.20 mg -8 mg/°C	+5.19 mg -25 mg/°C	+3.38 mg -4 mg/°C
Alignment error	0.24°	0.64°	0.41°	0.11°	0.12°	0.15°
Bandwidth	78 Hz	75 Hz	74 Hz	901 Hz	885 Hz	981 Hz
Damping	0.69	0.68	0.66	0.89	0.83	0.92
Noise(10-100Hz)	1 mVRMS	1 mVRMS	.4 mVRMS	.7mVRMS	.9mVRMS	.7mVRMS
Output Impedance	0 Ω	0 Ω	0 Ω	500 Ω	508 Ω	513 Ω

Table 3.1: MotionPak Calibration Sheet

From the calibration sheet alone we observe that the Systron Donner is an low-quality inertial unit that is not intended for standalone use. To begin with, higher-end systems will have much lengthier calibration sheets which describe sensor accuracies and imperfections to a much higher order. For instance, error models for military-quality gyros include at least 10 "classical, gravity dependent terms, [as well as] additional terms describing the error contribution of temperature effects, float motion, magnetic effects, power supply variations," etc. etc. (Mackenzie [8], p. 376). Furthermore, note the coarse temperature performance spec on the bias of three angular channels: < 3°/s. This means that if the gyros outputs are integrated and the nominal bias is as given, the integrated angle value can still drift as much as 3° every second because of temperature variation! Compared to most military systems, which guarantee gyro biases to fractions of a degree per *hour*, this is an awful spec. Remember also that gyro drift rate is the key limitation on inertial system performance. In practice, using temperature compensation techniques that are explained in Chapter 4, the Systron Donner gyros can achieve a worst-case overall drift rate of about 1°/min., which is still not quite good enough for standalone navigation. This inertial system, ideally, should rely heavily on external sensor data to keep the position estimate from drifting too far over time – or restrict itself to very short-term navigational missions.

The IMU has seven outputs, all analog: three from accelerometers, three from rate gyros, and one from an internal temperature sensor. Each accelerometer output acts as a dependent voltage source whose voltage is proportional to the acceleration along one axis. It is also important to note that there is a finite output impedance on each acceleration channel; the value of this impedance is provided on the IMU calibration sheet. The gyro outputs also behave as dependent voltage sources, but there is zero output impedance on these. Refer to Figure 3.7.

The purpose of the temperature sensor is to allow dynamic temperature compensation of the accelerometer and gyro outputs. Instruments such as gyros and accelerometers output a small DC voltage even when there is no motion input; this is the *bias.* Furthermore, gyro and accelerometer biases will change with temperature (approximately linearly in the Systron Donner); note the specifications "bias" and "temperature performance" on the calibration sheet. The temperature sensor allows us to compute in software the expected bias of each inertial instrument at the current temperature, and dynamically subtract this bias to obtain a zero-offset reading from each instrument. The temperature sensor in the Systron Donner IMU is a dependent current source which delivers a current (in microamps) directly equal to the

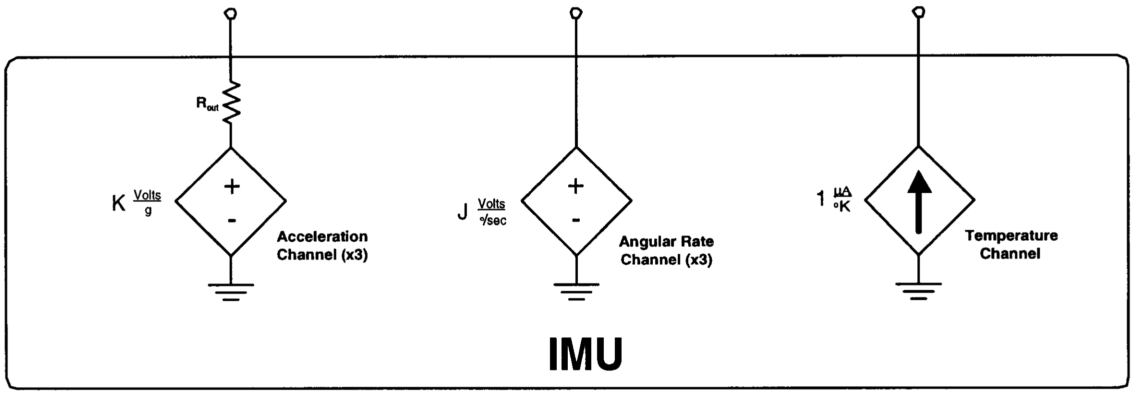

R_{out}
$K\ \frac{\text{Volts}}{g}$
+
-
Acceleration Channel (x3)
$J\ \frac{\text{Volts}}{°/\text{sec}}$
+
-
Angular Rate Channel (x3)
$1\ \frac{\mu A}{°K}$
Temperature Channel
IMU

Figure 3.7: IMU Outputs

IMU's temperature (in Kelvin)[9].

The IMU outputs are scaled and low-pass filtered using the active filter circuitry shown in Figure 3.8. Scaling is necessary to convert the full-scale range of the gyros and accelerometers to the full-scale input range of the Tattletale's A-to-D converter (which is ±2.5V). Low-pass filtering ("anti-aliasing") must be performed prior to sampling the analog signal and converting it to discrete-time information. This is a subject that will be explored extensively in the next section.

3.4 IMU Data Acquisition

3.4.1 Filter design for sampling

In an inertial system that uses a digital computer to execute the navigation algorithm, there is the inherent issue of converting analog data from gyros and accelerometers into digital data that the computer can work with. Several immediate concerns arise: these include choosing the sample rate or control bandwidth, and designing filters to avoid aliasing.

[9] When plugged in, the IMU dissipates about 8 watts of power, and thus "warms up" to some steady-state temperature (which is a function of the ambient temperature as well as the material it is mounted to).

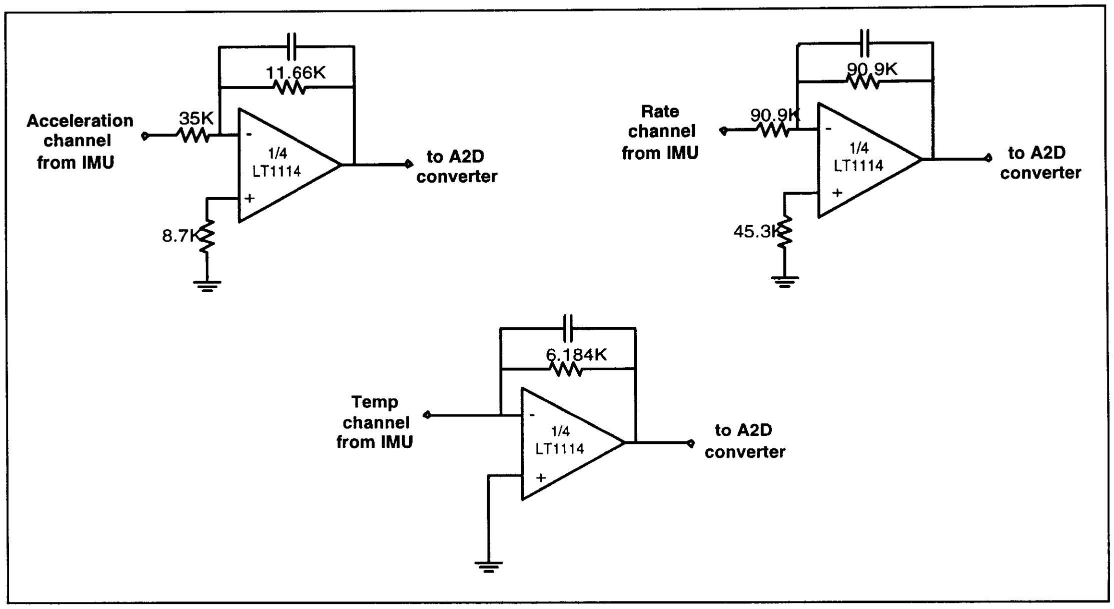


Figure 3.8: IMU Scaling and filtering circuitry

In electromechanical control systems, there is no precise science to choosing a suitable sampling rate. There are however some rules of thumb commonly preached, such as "the sample rate should be at least 20 times above the highest frequency that the mechanics are expected to follow [5]." Considering that the VCUUV is not expected to make extremely rapid movements in the water (the tail beat-frequency is nominally 1.5 Hz), a sampling rate of 50 Hz was deemed more than adequate. In fact, this initial choice was made with the awareness that the VCUUV computer might not have enough computing power to close a control loop at 50 Hz, but designing the system to have a sample rate surplus is always better than needing more control bandwidth later.

The problem of aliasing is described as follows: when analog data is sampled at discrete intervals, no one can predict the value of the analog signal between each sample. If the signal's spectrum is rich in high frequencies, it is likely that there are some high frequency "wiggles" occurring between each sample; but if the signal is concentrated in low frequencies, it is likely to meander slowly during the time between samples. The Nyquist criterion tells us that we can (in theory) exactly determine what happens in between the samples (or *perfectly reconstruct* the analog signal from its discrete samples) if and only if the analog signal is "bandlimited" to frequencies which are below one-half the sampling frequency. In other words, if the data is sampled at a frequency f_s, any frequency components in the spectrum of the analog signal above $f_s/2$ must be removed or filtered before digitizing the signal[10]. Otherwise, the sampled high frequencies will "alias" down to corrupt the low-frequency components of the digital data. Thus for a sample rate of 50 Hz, we must design low-pass filters which sufficiently destroy any components of the IMU signals above 25 Hz – while passing the range of DC-25 Hz with good fidelity.

Specifically, a good low-pass analog filter can cut the bandwidth of the analog signal to the

[10] $\frac{f_s}{2}$ is known as the "Nyquist frequency."

appropriate size, so that all frequencies at or above the Nyquist frequency are sufficiently attenuated. Usually the cutoff of the low-pass filter must be chose well below the Nyquist frequency, since it is impossible to have a perfect "brick-wall" low-pass filter. However, this is not the end-all solution, since using analog filters for anti-aliasing is often problematic.

One of the fundamental problems is that good analog filters are hard to build. Simple first and second-order RC op-amp filters are often used, but their rolloff is pitifully sluggish, and they don't provide enough attenuation at the Nyquist frequency. Attempting to remedy this by lowering the filter cutoff compromises the filter passband; that is, the low-frequency data of interest is corrupted by the filter. Higher order filters with sharper cutoffs usually involve several op-amps, and a careful choice of resistors and capacitors to set the filter bandwidth. Tolerances in these components can change the shape of the filter dramatically. It is possible to buy good Butterworth, Chebyshev, and other analog filter types on a chip, but these have other problems. They are often built with switched-capacitor technology which allows changing the filter cutoff with a digital clock input, but also involves increased noise generation, and DC offset errors.

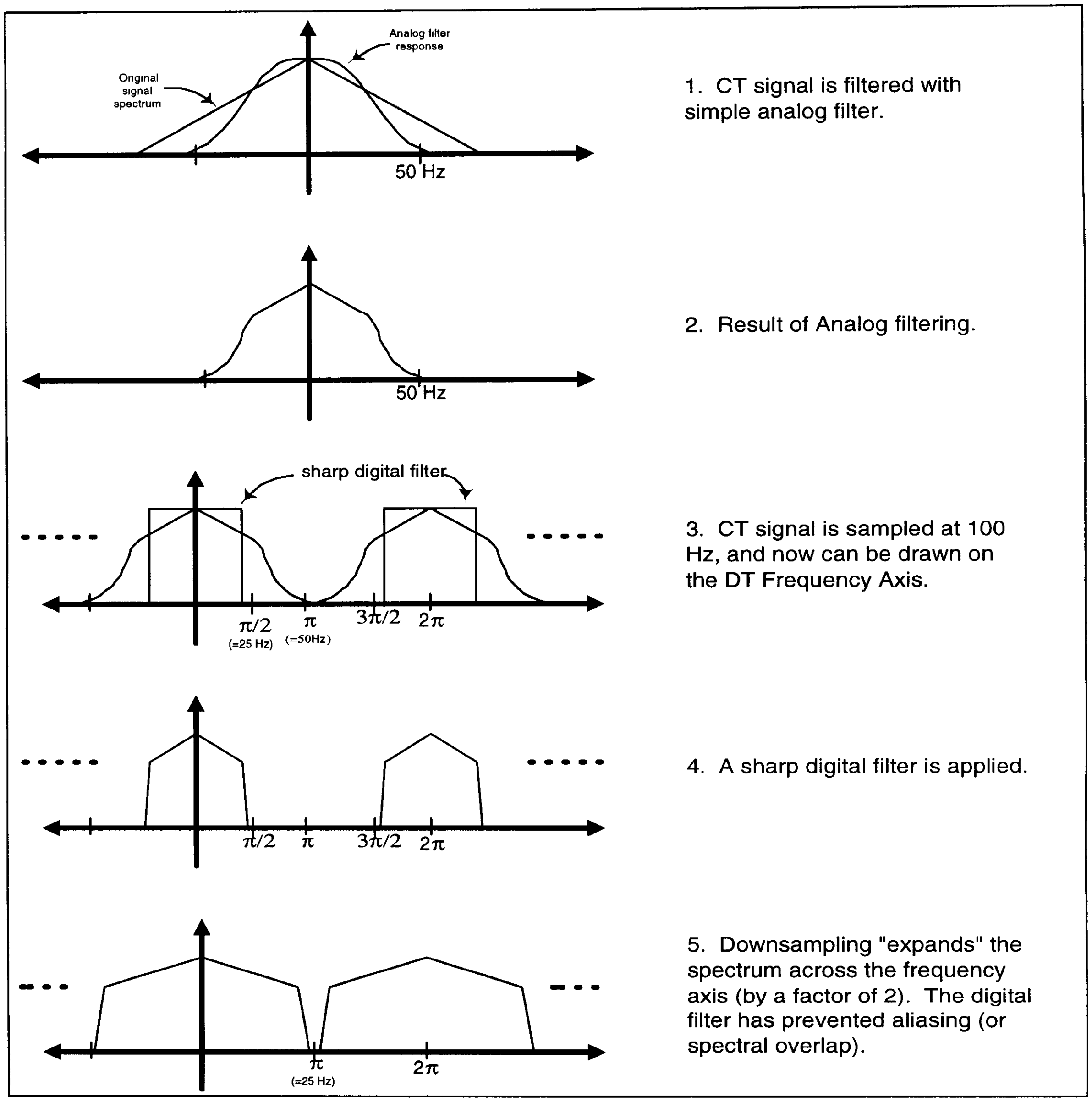
In order to overcome the difficulties of building good analog anti-alias filters, a technique is often used that also incorporates digital filtering [9]. In a nutshell, this is how it works:

Consider initially sampling the signal at higher than 50 Hz, say 100 Hz; at this rate, we can use a simple analog filter, perhaps 1 pole at 20 Hz, to bandlimit the signal to the new Nyquist frequency, 50 Hz. The desired passband of 0-25 Hz is not badly distorted by this filter. Now we have twice as much data as we wanted (100 Hz over 50 Hz), so we would like to keep only one out of every two samples. To prevent aliasing when we throw out these samples, however, we need to use a sharply-designed digital filter to cut the effective digital bandwidth[11] to 25 Hz, and then throw away every second sample (or *decimate* the 100 Hz data by a factor of 2). The steps involved are oversampling, digital filtering and decimation, and their effect on a signal in the frequency domain is depicted in Figure 3.9.

This technique uses the best features of both analog and digital filters. Analog filters work nicely when they don't have to be precise in cutoff or steep in rolloff; the digital filter picks up the slack, since its cutoff and sharpness can be precisely controlled. Of course, digital filters have their own problems as well, such as finite-precision errors – quantization and roundoff. An advantage to using the digital filter as the final filtering step is that sometimes noise is introduced into the extra circuit paths of analog filter hardware, and this noise can be attenuated by the digital filter which follows. However, this advantage should be weighed against the amount of finite-precision "noise" that might be generated by the digital technique itself[12].

[11] Note that cutting the "effective digital bandwidth" to 25 Hz means designing a digital filter with a cutoff at $\pi/2$, since in the discrete-time world, the frequency domain is "wrapped" around the unit circle, and any frequency (in radians) is always referenced to the sampling frequency (100 Hz, which converts to 2π radians). This differs from the continuous-time (analog) world, where frequency is absolute.

[12] In the implementation of the digital filter on the Tattletale, finite-precision noise was considered to be minimal, since all the computation was done using double-precision (8-byte) floats ("doubles").



1. CT signal is filtered with simple analog filter.

2. Result of Analog filtering.

3. CT signal is sampled at 100 Hz, and now can be drawn on the DT Frequency Axis.

4. A sharp digital filter is applied.

5. Downsampling "expands" the spectrum across the frequency axis (by a factor of 2). The digital filter has prevented aliasing (or spectral overlap).

Figure 3.9: Oversampling, filtering and decimation in the frequency domain.

The analog filters used are illustrated in Figure 3.8. They are simple first-order active RC filters; the more interesting task was choosing the best digital filter for the job. Quite a few digital filter designs are available, and numerous tradeoffs exist among them in issues such as rolloff, group delay and computational time. People have devoted entire textbooks to this subject, so we will suffice here with two tables of pros and cons between choosing Finite Impulse Response (FIR) versus Infinite Impulse Response (IIR) filters. For more information, a good place to start is Oppenheim's *Discrete-Time Signal Processing* [9].

FIR PROS	FIR CONS
Implemented through convolution of a finite sequence of samples with the signal. For most FIR impulse response shapes (box, triangular, etc.), all integer coefficients can often be chosen, making the convolution computationally efficient.	Length of FIR impulse response in most cases has to be quite long ("higher order") to implement filters with sharp cutoffs. Long impulse responses result in convolution becoming more computationally taxing.
For symmetrical impulse responses, FIR has linear phase (constant group delay). These constant delay filters are easily designed using the window method, for example.	For long impulse responses (sharp cutoffs), the delay, though constant, will be rather long (specifically, half the length of the impulse response). This is not acceptable in real-time control loops.
When filtering with FIR before downsampling, sometimes a more efficient algorithm can be used which allows skipping unnecessary steps in the convolution sum.	In most cases, FIR is more computationally intensive overall than an IIR filter with similar filter shape/sharpness.

Table 3.2: FIR digital filter pros/cons

IIR PROS	IIR CONS
Implemented through computation of a difference equation using delayed values of the input (x[n]) and output (y[n]). Thus each filter computation involves only a few multiplications and additions, and no convolution. Usually *much* faster to implement than a similar FIR filter.	IIR requires floating point addition and multiplication operations, since the coefficients are non-integer. In some systems (i.e. no hardware floating point unit) this may take an excessive amount of time, depending on filter type/order.
Though the group delay is non-linear, it can often be acceptable in the frequency range of interest (i.e., the low-frequency passband) in a control loop.	IIR filters (Butterworth, Chebyshev, etc.) typically have grossly non-linear phase responses/group delay, and thus unequally disperse the frequency content of the signal.
Flexibility to design filters with well-known attributes, such as Butterworth, Chebyshev, Elliptic.	When filtering with IIR before downsampling, no steps can be skipped; all samples must be computed, even those that will be discarded.

Table 3.3: IIR digital filter pros/cons

The final digital filter design was a third-order Chebyshev type II IIR filter. The coefficients and characteristics of this filter were designed using MATLAB. The filter's discrete-time frequency response is shown in Figure 3.10; recall that we desired a fairly sharp cutoff at $\pi/2$. This filter satisfies that requirement quite well.

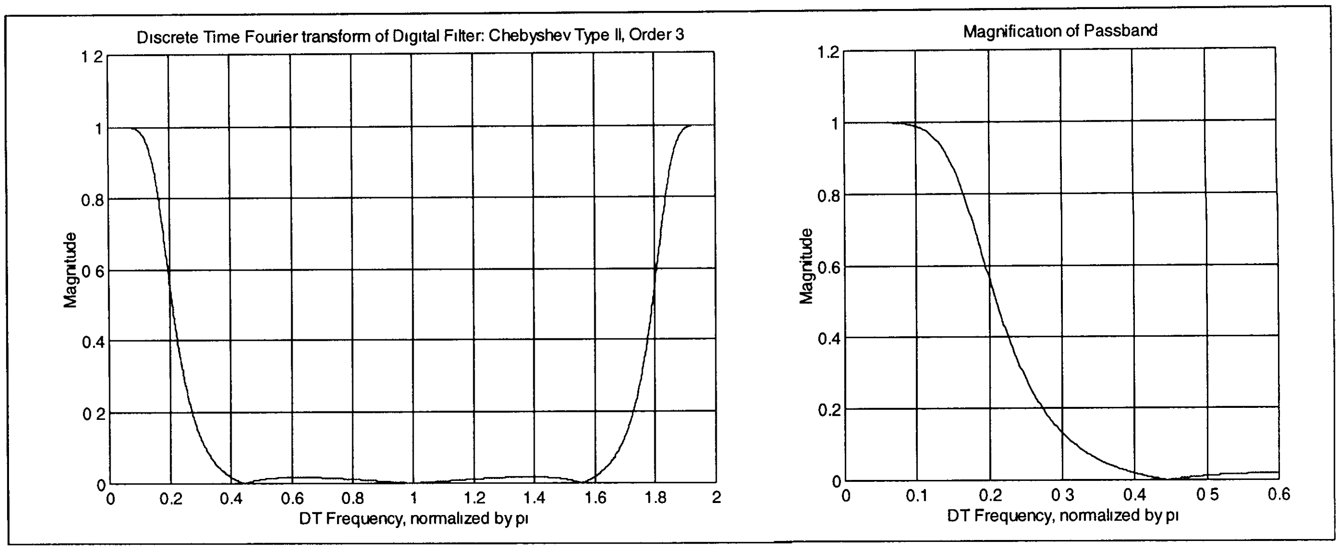


Figure 3.10: Tattletale digital filter design

A few notes on choosing this filter: several other types of IIR filters were considered, including a Butterworth and Chebyshev type I. The Butterworth is known for its maximally smooth frequency response, and the Chebyshev type I for its nice rolloff characteristics. However, the Chebyshev I exhibits ripples in the passband, which is very undesirable since we require a flat passband. The ripple can be reduced only in exchange for a very slow rolloff. The Chebyshev type II, on the other hand, exhibits stopband ripple, which is not so bad for our purposes. For comparable filter designs of third-order, the Chebyshev type II appeared to be the best at attenuating all frequencies above $\pi/2$ without sacrificing too much of the passband (see Figure 3.11).

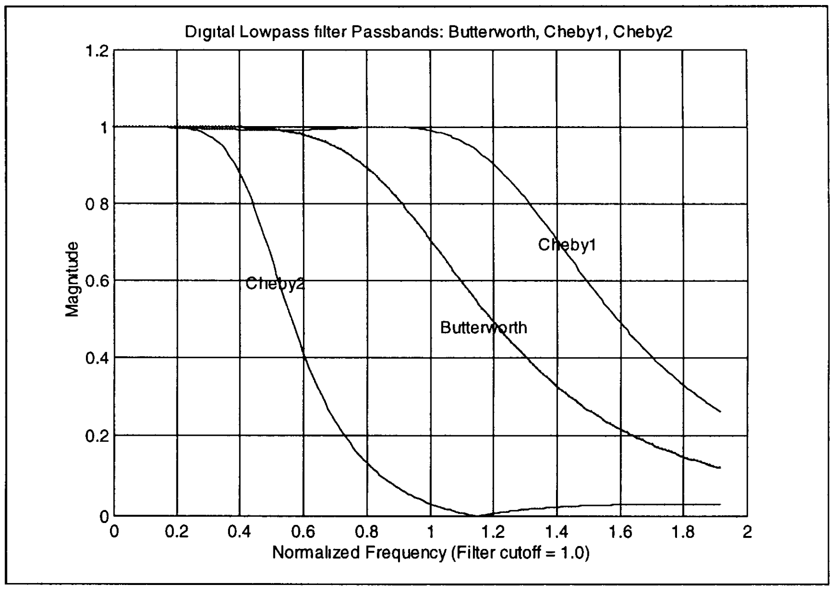


Figure 3.11: Various types of IIR filter passbands

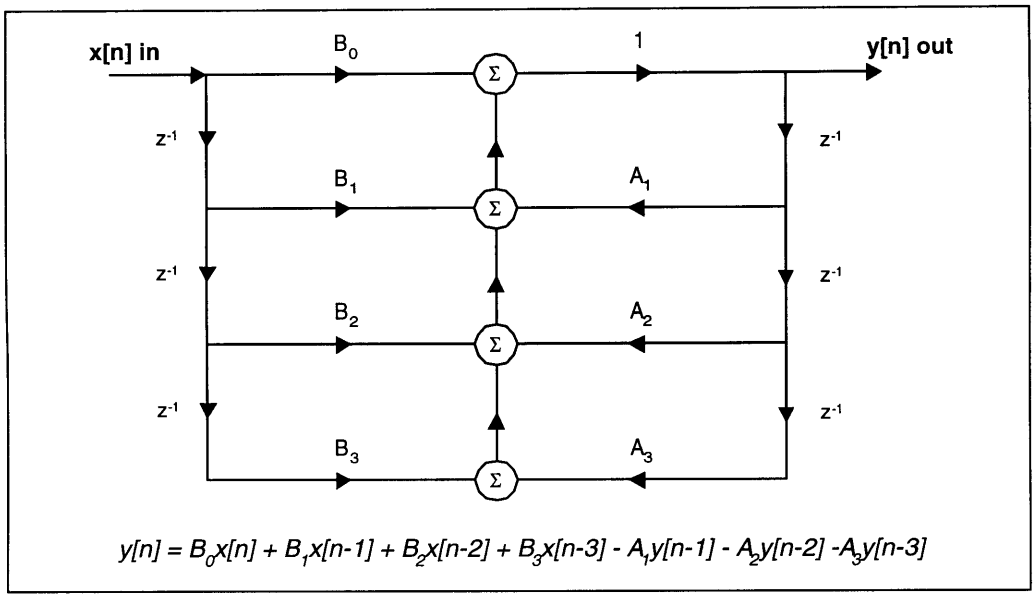

x[n] in
y[n] out
B_0
1
B_1
A_1
B_2
A_2
B_3
A_3
z^{-1}
Σ
$y[n] = B_0x[n] + B_1x[n-1] + B_2x[n-2] + B_3x[n-3] - A_1y[n-1] - A_2y[n-2] - A_3y[n-3]$

Figure 3.12: Direct Form I implementation of 3rd order IIR filter

The Chebyshev type II filter is implemented through the solution of a difference equation. Figure 3.12 shows the generalized third order difference equation, and the "computational structure" used to carry out the filtering. Note that each output y[n] is computed as a weighted sum of the current input x[n], the previous three inputs x[n-1], x[n-2], x[n-2], and the previous three outputs y[n-1], y[n-2], y[n-3]. The coefficients A_i and B_i (i = 0,1,2,3) necessary to make this a Chebyshev type II filter were obtained using MATLAB. The actual software used to compute the difference equation will be revisited in section 3.5.1.

How well did the composite filtering scheme work? The following simple experiment demonstrated its effectiveness. When the IMU is sitting upon a piece of foam (similar to the material it is mounted on within the VCUUV), the mass of the IMU coupled with the compliance of the foam forms a second-order system, that exhibits a resonance frequency (see Figure 3.13). This was demonstrated by looking at a filtered analog output of the IMU with a scope, and lightly tapping the IMU: as it stopped vibrating, a damped sinusoid was clearly seen on the scope, with a frequency of roughly 40 Hz. However, observing the outputs of the digital filter, there is practically no response to the 40 Hz excitation, which is convincing evidence that the digital filter is working as promised to cutoff sharply at 25 Hz. If it didn't, the 40 Hz would alias down to appear as a 10 Hz disturbance.

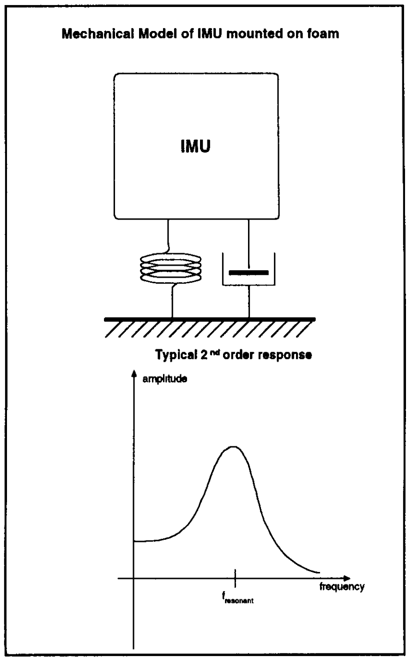

Mechanical Model of IMU mounted on foam
IMU
Typical 2nd order response
amplitude
frequency
$f_{resonant}$

Figure 3.13: Mechanics of IMU mounting

A serious issue which has not yet been mentioned is that of filter phase. Remember that the IMU, pressure sensor and compass data will eventually be used for real-time navigation and control of the fish. If the filtering is part of a large control loop, the phase lag of the filters can introduce enough delay into the loop to cause instability. It is thus necessary to try to minimize all sources of phase lag in the system. These include not only the analog and digital filters, but also the data-transmission and processing delays to and from the 486 (see Figure 3.14). Unfortunately, it is too early to determine the specific navigation and control techniques that the VCUUV will use, and so no rigorous bound can be placed on the total allowable delay in the control loop. Creating such a bound will involve characterizing not only the response of the navigation electronics, but the hydrodynamic behavior of the VCUUV itself – a challenging task in system ID that will probably commence several months from now.

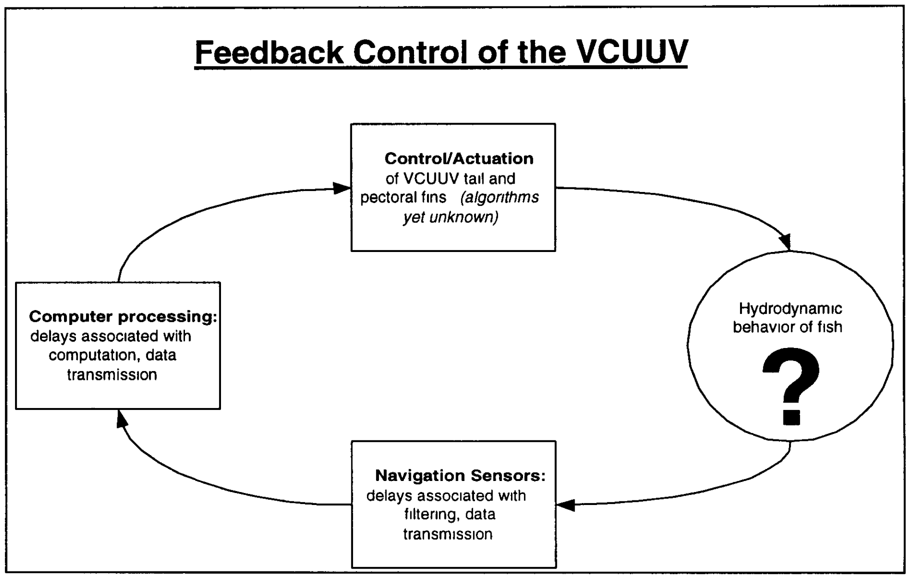

Feedback Control of the VCUUV
Control/Actuation of VCUUV tail and pectoral fins (algorithms yet unknown)
Hydrodynamic behavior of fish
?
Navigation Sensors: delays associated with filtering, data transmission
Computer processing: delays associated with computation, data transmission

Figure 3.14: Theoretical control loop for navigation

3.4.2 Precision Circuitry

Hopefully it has been clear how important it is to obtain the most precise analog-to-digital conversion of the IMU signals as possible. The op amps chosen to scale and filter the IMU outputs (shown in Figure 3.8) were LT1114s, a precision low voltage-offset device made by Linear Technologies. Care was taken to use 1% precision resistors in each amplifier[13]; where possible, the gains were measured using a signal generator and precision voltmeter. Note the resistor from the non-inverting terminal to ground: this resistor value is set to the parallel resistance of the other two, and eliminates offset due to input current bias.

3.4.3 Electromagnetic Interference (EMI)

Historically, the Unmanned Vehicle laboratory at Draper Lab has had a losing battle with EMI. A number of otherwise well-designed vehicles ultimately failed to meet their goals because of unanticipated EMI which plagued the electronics and caused a range of failures. In designing the VCUUV electronics, considerable efforts were made to minimize this potential show-

[13] In retrospect, 0.1% resistors would have been more appropriate for the desired precision.

stopper.

EMI is often given the ubiquitous label of "noise," which includes such effects as antenna radiation, magnetic/inductive pickup, and capacitive coupling. In the VCUUV, there are many sources of EMI, and several different ways in which they affect signals.

The main source of EMI on the VCUUV is its switching power converters. These efficient supplies operate in principle by rapidly pulsing charge onto a capacitor through an inductor: the fast switching of current through an inductive medium tends to generate a whole spectrum of random electromagnetic junk. This electronic litter is very proficient at jumping from place to place within the electronic chassis, corrupting analog and digital signal lines alike. The VCUUV also contain four microprocessors (486, DSP, PIC and 68332-Tattletale), all running at speeds in the Megahertz range, which are very likely candidates for generating EMI.

Debugging noise problems often seems like black magic; techniques which seem to work in some cases fall short in others. The first step is to diagnose where noise tends to appear, since it shows up in very selective places. For instance, electrical signals that come out of a high source impedance often pick up transients from fast switching digital signals that run physically nearby. This is because the small capacitance between the wires combines with the large source impedance to form a high-pass filter, bringing high-frequency transients over to the line. As for magnetic pickup, circuitry with "ground loops[14]" or large stray inductances are susceptible. Almost *anything* might show a particular affection for picking up antenna radiation.

In the VCUUV navigation hardware, the greatest concern regarding noise is that *the IMU signals remain pristine.* A good deal of noise was originally seen on the IMU signals, as well as all other signals that ran near the Tattletale: this was attributed to its processor (running at 32 MHz) and its on-board switching power converters. Capacitive noise coupling (described above) was also found between the Tattletale's serial port and the IMU signals: millivolt-level bursts of noise were seen on IMU signals that coincided with each data packet being sent to the serial port at 115Kbaud.

The IMU signals were somewhat cleaned up by adding 200 nF capacitors to ground directly at the A-to-D inputs. This is very effective against capacitive coupling because the high-frequency noise is shunted to ground through the large capacitor (which is greater than the stray coupling capacitance). This technique is a little risky, since it is usually a bad idea to drive a large capacitance directly from an op-amp output (due to phase margin and stability problems); but for this particular op-amp (LT1114) and the gains involved, it turned out to work quite well. In general, a small resistor between the op amp output and the capacitor is a good idea.

In order to further protect the clarity of the IMU signals, each signal wire from the IMU was shielded with a grounded metal sheath. Care was taken to connect the shields to only one ground point (at the IMU), thus avoiding ground loops (see Figure 3.15). In principle, these shields "swallow" electromagnetic fields and divert them from signal lines. Following the same idea, large squares of grounded metal foil ("ground-planes") were added to empty spaces on the PC board.

[14] Ground loops are completely independent paths to ground from the <u>same</u> electrical node.

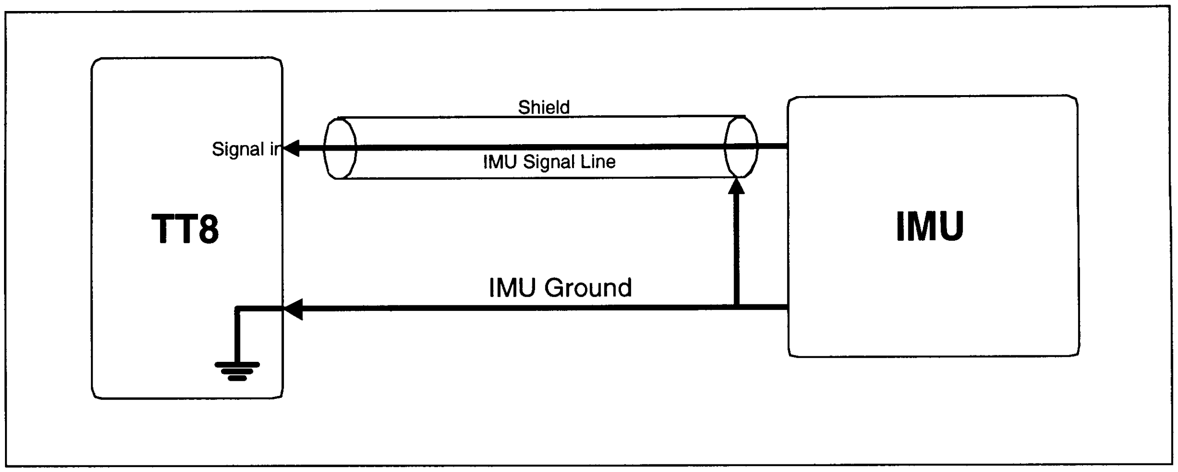


Figure 3.15: Shielding of each IMU signal

Using these techniques, IMU signal noise at the A-to-D converter was reduced from about 10 mV peak-to-peak to about 2 mVpp, which is about two A-to-D quanta. After sampling, the digital filter further cleaned up this noise.

3.5 Software architecture

Making the Tattletale perform all of the duties described above required the creation of well-crafted program code. Specifically, this software is *real-time*: it is responsible for sampling and processing data at very precise intervals at time, and cannot afford to fall behind. The major challenge in designing such software is having absolute knowledge of how long each process takes to execute, and determining the *bandwidth* available to each process. Overall bandwidth can be used most efficiently by implementing a multitasking scheme which never allows the processor to "sit idle." On the TT8, this multitasking was achieved through an interrupt handler to manage strictly periodic events (sampling, digital filtering, packet transmission), and in between, a loop to handle slower, asynchronous tasks (processing 486 commands, reading the compass, sending Smart Motor commands). These issues are discussed in detail below. The actual program code is provided in Appendix C.

3.5.1 Sampling and Digital Filtering

As described in Section 3.4.1, the IMU and pressure sensor data is sampled by the Tattletale at 100 Hz, and then digitally filtered before downsampling the data to 50 Hz. In order to sample the data precisely every 10 milliseconds (100 Hz), the Tattletale's processor provides a convenient mechanism called the Periodic Interrupt Timer (PITR), which can be programmed to generate hardware interrupts at very accurate intervals of time. Once this interrupt timer is set up to generate interrupts at 100 Hz, an interrupt handler can be written to sample the data each time it is called. The digital filtering also takes place within the handler, immediately after a new set of samples arrives.

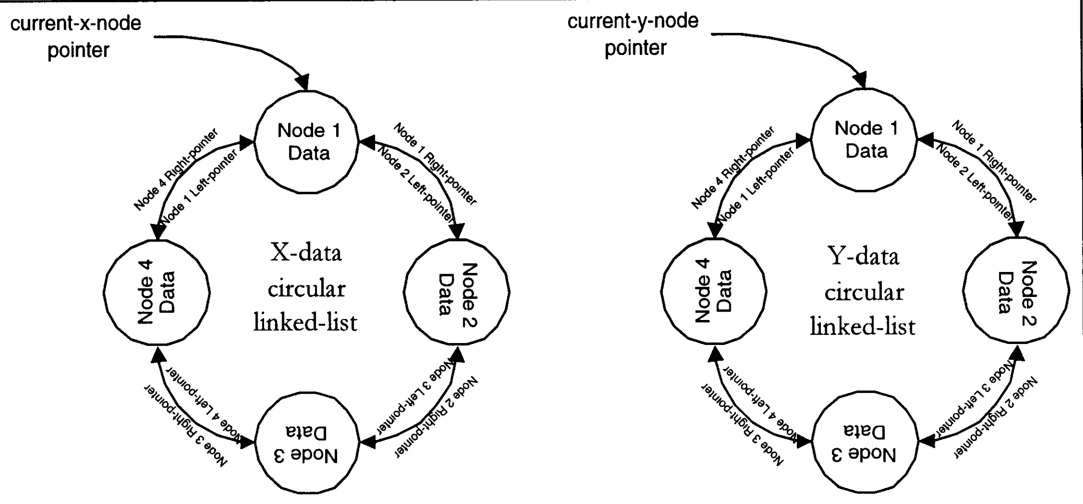
Sampling is accomplished by direct access to a peripheral on the Tattletale CPU called the Queued Serial Module (QSM). The QSM can talk to devices using a serial protocol known as the Queued Serial Peripheral Interface (QSPI). This protocol allows a processor to quickly obtain data from a list or "queue" of devices that support QSPI. Thus, when the eight separate channels of the Tattletale's A-to-D converter are placed as separate elements of the "queue," and a QSPI start bit is set high, it rapidly selects each channel in turn, collects the data and stores it in memory. Now the data has been captured, and is ready to be digitally filtered.

One of the most challenging tasks in implementing the composite filtering scheme was developing a digital filter that would run fast enough to allow real-time processing of the data, concurrent with all the other tasks of the Tattletale. Eight channels of data (seven IMU signals, one pressure signal) must be sampled every 10 milliseconds, and each run through an IIR filtering structure (remember Figure 3.12) so that a new filter output is computed for every sample. If we want the Tattletale to have time to do anything more than just sample and filter, then, the filtering has to occupy far less than 10 milliseconds. Originally, designing a filter to run faster than 10 milliseconds was very difficult, especially considering that the Tattletale's CPU has no floating-point unit: this means that all floating-point calculations (necessary to implement a digital IIR filter) are done through software emulation, which is notoriously slow[15]. After doubling the original CPU clock rate and some optimization of the code, the filtering time for each set of 8 new samples was brought down to about 4 milliseconds. This leaves about 6 milliseconds out of every 10 for the Tattletale to perform other tasks, or about 60% bandwidth. The issue of available bandwidth is very important, and will be returned to later.

To implement the digital filter, a software construction was used that mimics the IIR filtering structure of Figure 3.12. Each of the delayed inputs and outputs are stored in two circular linked-lists, structures in memory where "nodes" store data and also point to adjacent nodes (see Figure 3.16). As new data (x[n]) arrives, the pointers shift to replace the x[n-3] data with x[n], and a new output y[n] is computed from the values in both linked-lists. This cycle of pushing in a new input, pushing out old filter history and computing a new output is repeated 100 times a second on each of the 8 channels of data.

The whole point of the digital filter is allow downsampling to 50 Hz; that is, to allow discarding one of every two samples that are filtered. Thus while the interrupt handler is called at 100 Hz, it only transmits the filtered data to the 486 *every other* time it is called – 50 Hz.

[15] Also, programming the TT8 in C resulted in fairly inefficient code generated by the compiler and assembler; this is the tradeoff for the convenience of using C.



These two circular linked lists hold the current x- and y-data and history. When a new input x[n] is sampled, it is stored at the current-x-node location, replacing the previous x[n-3]. Next, a current output y[n] is computed based on x[n] and the delayed values of x and y. This new y[n] is stored at the current-y-node location. Then the current-x- and current-y-node pointers both shift to the left, by following the "Node-left" pointer: what were "current" filter values have become filter history. Finally, the new x[n] is sampled, stored in the new current-x-node location, and the next y[n] computed; the cycle repeats.

Figure 3.16: Software IIR Filter, storage of delayed elements

3.5.2 Compass Software

Since the compass takes up to 100 milliseconds to compute each new heading, it is impossible to sample it at 100 Hz with the other sensors. It must be dealt with asynchronously. The compass is interfaced through TPU digital I/O lines from the Tattletale, which can be set high or low through software. A sequence of "compass events" is established where each event is either the proper time to set or clear a TPU line, or a wait state. Cycling through the 12 compass events corresponds to playing the digital "song-and-dance" with the compass that is necessary to obtain a reading. This technique amounts to a method of multitasking, where the Tattletale won't waste time (or get "blocked") while waiting for the compass to compute its heading: it checks the compass event status periodically, and moves on to its other duties when the compass is still thinking. The compass readings arrive about every 200 milliseconds, or 5 Hz. This data is then placed into a "data packet" with the IMU and depth sensor data, and sent to the 486.

3.5.3 Talking to the 486

A data packet is built during every other call to the interrupt handler, and sent out through the Tattletale's serial port. Each packet contains the following sequential information:

DATA	NUMBER OF BYTES
Sample Number	4
X-Accel	4
Y-Accel	4
Z-Accel	4
X-Rot	4
Y-Rot	4
Z-Rot	4
Temp	4
Depth	4
Compass	2
Collision Flag	1
Extra Flag	1

Table 3.4: TT8 Data Packet Structure

Thus a total of 40 bytes per packet, or 320 data bits that need to be sent to the 486[16].

The first question is: how much time do we have to serially transmit this data? Remember that the building and transmission of each packet occurs during the interrupt handler, which has a total of 10 milliseconds execution time before the next handler is called. Also, 4 milliseconds of this time are already occupied by sampling and filtering activity. A data rate of 115.2 Kbps was settled on[17], at which speed it takes about 2.8 milliseconds to send one data packet.

Using such a fast rate for serial transmission can sometimes be risky. If the data were not being sent along such a short distance of wire, it would not be wise to run at this speed. In fact, the RS-232 serial protocol is only guaranteed accurate up to about 26 Kbaud, but it is often run much faster in special circumstances – such as this one.

In order to allow the 486 to synchronize to the incoming packets, each packet is pre-pended with a special header byte that the 486 can identify. Unfortunately, there are some problems with this scheme. It is impossible to find a "unique" header byte, meaning a byte that will never be seen in the data packet itself, since most of the data bytes can take on the full range of byte values (0-255). Though seldom, the 486 sometimes synchronizes on the wrong byte and translates a packet of "garbage." Another packet-reception problem occurs when the 486 serial

[16] The compass data, which arrives more slowly than the other data, is set to a dummy value when unavailable.
[17] Unlike the 486, the Tattletale cannot operate at 115.2 Kbaud exactly; the closest it can come is 111.1Kbaud. Serial transmission, however, is typically tolerant of baud rate mismatches up to 5%, and this is only a 3.6% error.

input buffer overflows (i.e. the 486 cannot read the packets fast enough to keep data moving through the buffer), and packets are truncated or filled out with garbage[18]. To abate these problems, the sample-number quantity was always made an even value, so the header byte which precedes the sample-number (which has an odd ASCII value) can never be confused with its neighbor. A *checksum* was also added to the end of each packet: this is a two-byte quantity that equals the sum of all the data bytes in the packet. The 486 computes its own checksum of the data it receives, and if this checksum doesn't equal the Tattletale's checksum, decides that the packet is corrupted, and throws it out. In the end, data transmission was observed to be over 99.99% error free.

It is important to mention that no handshaking occurs between the 486 and the Tattletale during packet transmissions; the TT8 does all the talking. Even at 115.2 Kbps, the TT8 has virtually no extra time to wait for 486 acknowledgments of packets received – and absolutely *no* time to re-transmit any packets that were received incorrectly. The only information the 486 can send to the TT8 are the following commands:

1) Reset the compass
2) Perform compass hard-iron calibration
3) "Ping:" respond if you are alive
4) Start sampling and sending data at 50 Hz
5) Move the Smart Motors to commanded position
6) Shut down, reboot and reinitialize the system

These commands are sent to the TT8 in a 7-byte packet built by the 486 with the following structure:

Byte #	Contents
0	Header byte (ASCII 'q')
1	Reset Compass, or pulse CAL pin on Compass
2	Home the fins, or start sampling the sensors, or reply to a "Ping," or stop sampling and reboot the whole system
3	Left Smart Motor Most Significant Byte
4	Left Smart Motor Least Significant Byte
5	Right Smart Motor MSB
6	Right Smart Motor LSB

Table 3.5: 486 Data Packet Structure

Notice that bytes 1 and 2 can be coded to send any of several different messages to the TT8. This functionality can be expanded further in the future if necessary. The Smart Motors can take possible commands of +25000 to -25000, and so 2 bytes are required per command. If no Smart Motor movement is required, the value of 30000 ("MOTOR_DO_NOTHING") can be sent to the TT8. Transmission of this packet appeared to be extremely reliable.

[18] These errors occur particularly when the 486 has to do something time-consuming between reading TT8 packets, such as print data to the screen.

3.5.4 Smart Motor Control

The VCUUV has two pectoral fins, one on each side of the body. These act as control surfaces that allow the fish to rise or dive while it is moving forward. The pectoral fins are driven by an off-the-shelf motion control package known as *Smart Motors*[19]. Smart Motors are manufactured by Animatics, and are used in many industrial and robotics applications which require position or velocity servomotors. They offer excellent programmable versatility, and can communicate with a host system using a fairly simple serial string protocol. Each pectoral fin is driven by a Smart Motor through a worm gear assembly (see Figure 3.17).

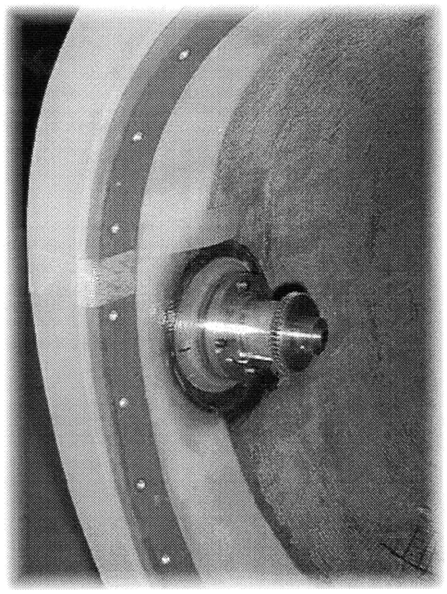


Figure 3.17: Pectoral Fin Gear Assembly

A Smart Motor is a servomotor with a built-in microprocessor, digitally tunable PID filter[20], and serial port all in one. The device can be commanded to servo to a velocity or position command, with a resolution of 2000 counts/revolution. Best of all, any of a small army of servo parameters can be changed on-the-fly through the motor's serial port: these include the P, I, and D gains, maximum current draw, maximum steady-state error, and many more.

A typical serial command sent to the Smart Motors might be the following string:

"~0A=1000 ~0V=900000 ~1P=200 ~2P=300 ~0G"

The prefixes "~0", "~1" or "~2" are addressing operators, which respectively address both motors, motor #1, or motor #2. The "A=" command sets initial/final acceleration/deceleration, "V=" sets the cruising velocity, and "P=" the final position command. Once a Smart Motor receives the GO command "G", it computes a trapezoidal velocity profile from those three parameters, and follows this profile to its destination.

The motors are wired in what is called "Echo mode," where the host system sends a serial string to the first motor which is echoed to the second motor, and eventually echoed back to the host system. Using addresses to identify each motor, it is possible to communicate with and control them individually.

When the fish is first powered up, the motors have no way of knowing the current angle of the pectoral fins. Thus after every powerup, a ritual called "homing" must be performed. This is made possible by a small magnet on each worm gear which trips a magnetic reed switch buried on the opposite side of the gear: the fixed location of the switch serves as a reference (or

[19] The Smart Motor mechanics and assembly were specified by Steve Bellio, a Draper engineer.

[20] PID control (Proportional-plus-Integral-plus-Derivative) is usually the best choice of feedback compensation for motors. The proportional gain attributes for the overall "stiffness" of the motor response, the integral term kills any steady-state error, and the derivative acts as a "shock absorber" for high-frequency disturbances.

"homing") location. The status of the reed switches are read by the TT8's digital TPU inputs. To home a pectoral fin, the Smart Motor turns the fin until the reed switch is tripped by the magnet. Since there is about a 20° arc within which the switch can be tripped by the magnet, the homing algorithm finds the two edges of this arc, averages them to find a center point, and calls this center point "home."

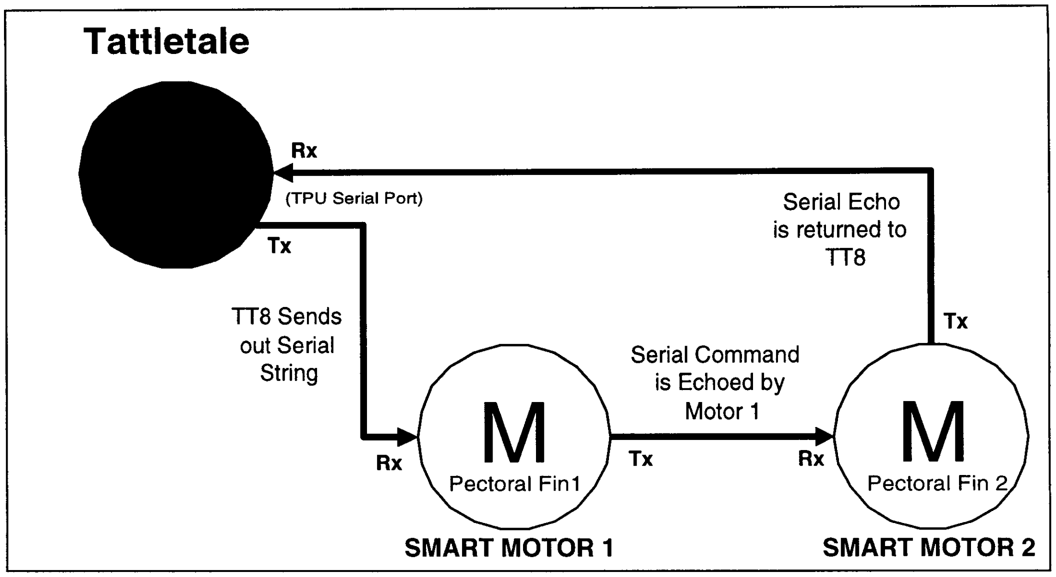


Figure 3.18: "Echo" mode for TT8-to-Smart Motor communication

3.5.5 Bandwidth Issues

The Tattletale has a variety of tasks to perform at specific times, and the way it distributes its computational abilities over these tasks might be called *bandwidth management.* Certain duties such as sampling, filtering and data transmission take a fixed amount of time that is known, but others (such as compass management and Smart Motor control) are largely a function of other devices. An important question is: what is the maximum rate that the Tattletale can send commands to the Smart Motors? This question is motivated by the ultimate desire for depth control: in order to establish a feedback loop to maintain a depth in the pool, we need to know the delay between when the Tattletale receives a smart-motor command from the 486, and when this command is actually performed by the motors.

We begin with the assumption that the longest periodicity the Tattletale follows is obtaining a reading from the compass, which occurs about once every 200 milliseconds. Due to the compass event-multitasking scheme, we can put an upper bound of 5 milliseconds of actual processor time devoted to the compass. During this 200 milliseconds, also, the interrupt handler is called 20 times. On half of those calls the handler consumes about 4 milliseconds of processor time (sampling and filtering), and on the other half an additional 3 milliseconds is used to transmit a data packet (refer to Figure 3.19). Adding another 10 milliseconds or so for processor overhead in saving registers, etc., we have:

$$5ms + 10 \times 4ms + 10 \times 7ms + 10ms = 125ms$$

or 75 of every 200 milliseconds left over to control the Smart Motors. In other words, the Tattletale can devote about 37% of its bandwidth to Smart Motor control.

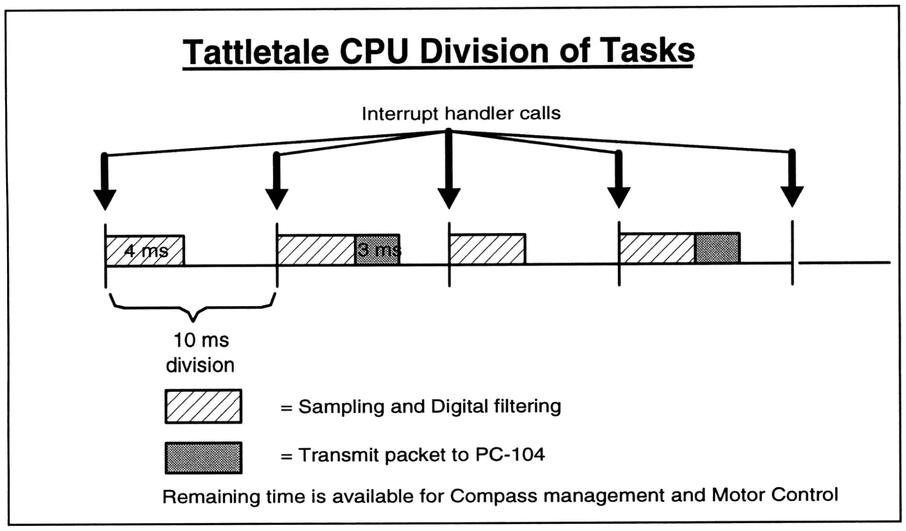

Tattletale CPU Division of Tasks
Interrupt handler calls
4 ms
3 ms
10 ms division
= Sampling and Digital filtering
= Transmit packet to PC-104
Remaining time is available for Compass management and Motor Control

Figure 3.19: TT8 Bandwidth division

As described in Section 3.5.4, each Smart Motor command is a string of ASCII characters sent serially to the motors at 38.4 Kbaud. A typical motor string command might be "~1P=100 ~2P=100 ~0G ", commanding each motor to go to position 100. At 38.4 Kbaud these strings would be sent quite fast, but the limiting factor becomes a 2 millisecond delay between characters that is necessary for the Smart Motors to read a string without errors. Thus this string should take about 40 milliseconds of the Tattletale's processor time to prepare and send (adding a few milliseconds for overhead). With the 37% bandwidth available, this time stretches out to almost 110 milliseconds per Smart Motor command.

This back-of-the-envelope calculation implies that the maximum rate we will be able to control the Smart Motors is just under 10 commands per second. However, this does not consider the delays that occur in the filtering process, or at the 486 end. These delays amount to a filter group delay of a few tens of milliseconds, and a data-transmission (from the 486) delay of about 30 milliseconds – resulting in a more realistic control rate of 5 commands per second. Even though our navigation data is arriving at 50 Hz, an 5 Hz control rate should be adequate for the pectoral fins, since the VCUUV is not expected to exhibit high-frequency behavior in its depth. This can be accomplished by averaging every group of 10 samples that arrive at 50 Hz to generate 5 Hz data.

Design of an actual depth control loop using this information will be revisited in Chapter 5.

3.5.6 QNX Software

The 486 processor on the PC-104 stack is running the QNX real-time operating system. In order to communicate with the TT8, a C-program was written which is able to write and read bytes from the 486's serial port. This program can instruct the Tattletale to do such tasks as reset or calibrate the compass, sample and return data to the 486 at 50 Hz, and home the pectoral fins or set them to a desired angle of attack. When sensor data arrives at the 486, the program can display the data to the screen, log it to memory (and eventually the hard drive), and/or use it in a feedback loop to set the pectoral fin angles through the Tattletale. Some primitive code was also written to integrate IMU gyro readings, giving the user a dynamic look at attitude drift; however, this function can be refined much further. The main challenges of maintaining an error-free communication link with the TT8 have already been tackled. What is remains is to continue developing software that takes advantage of the data acquired and sent back from the TT8 to perform more complex navigational routines.

3.6 Summary

The electronics described in this chapter were developed externally to the fish, and perform admirably in that setting. Pressure sensor and compass data can be accurately acquired, pectoral fin control is robust, and TT8-to-486 communication is efficient and reliable. The performance of the inertial system in particular will be discussed in more detail in the next chapter. Current work is directed at integrating this subsystem into the fish's electrically noisy environment, and protecting these circuits from a range of EMI-induced failures.

Chapter 4 VCUUV Inertial System

Having introduced the basics of inertial navigation systems and the particular IMU used on the fish, the next step is to characterize the performance possible with this instrument.

The first task is to characterize the IMU's drift behavior. It has already been mentioned that the drift of inertial position estimates is dominated by gyro (or attitude) errors. Thus, the performance of an inertial system is usually described by the performance of its gyros, and in particular, by a bound on the gyro error. This "gyro drift[21]" is specified in degrees per second (degrees per hour for more accurate instruments), and the lower the value, the better the accuracy of the computed position over time. However, it is still important to characterize accelerometer biases as well.

The gyro drift in an inertial system usually defines its performance over time, and therefore defines the range of applications that system is suited to. Figure 4.1, taken from Titterton and Weston [13], shows the different accuracies necessary for various inertial navigation tasks. For comparison, note that the earth rotates at about 15 °/hour.

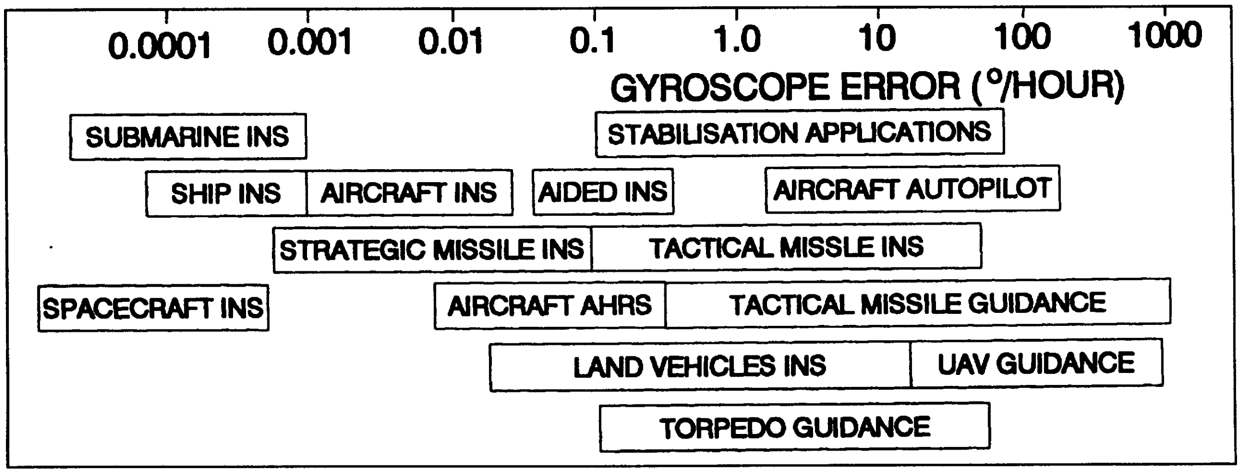


Figure 4.1: Gyro drift performance for various applications

An effort can be made to measure and characterize both accelerometer and gyro bias/drift, and thus compensate for it in software. This is the subject of Section 4.2. However, even after that compensation, drift will occur, due to the integration of electrical noise (producing "random-walking" of the angle/velocity), imperfections in the IMU, and quantization errors. In any inertial system (which includes the IMU, data acquisition circuitry, and computer) there is some fundamental limit to how well you can characterize the sensor errors, how susceptible they are to external phenomena (i.e. magnetic fields, cosmic rays, etc.), and how repeatable (or stable) these errors are between each run. Ultimately, this limit defines the

[21] In the literature, "drift" is used ambiguously to describe either position estimate drift, gyro error, or both. Additionally, the terms drift and bias are used interchangeably.

"drift" of the compensated system.

The performance of the VCUUV inertial system is tested at the end of this chapter, using a MATLAB routine to post-process data acquired and from the sensors and hardware.

4.1 IMU Limitations

The IMU sensors are solid-state, vibrating quartz accelerometers and gyros. These are commonly used in low-quality strapdown systems intended for use with GPS or similar external sensors. While they are very rugged, they usually are produced with biases of in the range of 0.1 to 1 °/s.

Systron Donner provides a CAD drawing of the IMU internals, showing the locations of the sensors within the case. This information is important to compensate against an error known as *size effect*: when the instruments are not all located at the center of the IMU, compensation is necessary for lever-arm (or centripetal) forces on the accelerometers. However, the Systron Donner CAD sheet is deficient in that it only gives the sensor locations in two dimensions, not three! Refer to Table 3.1, and notice the Alignment error parameter. These are the angles made between the sensitive axes of the sensors and the true orthogonal directions; alignment error information can be used to compensate for cross-coupled accelerations and rotations between the channels. However, to describe alignment error completely, *two* such angles are needed! One angle only provides the alignment within a cone of error. In general, several of Systron Donner's specifications for this IMU are incomplete and therefore useless. The size effect and alignment uncertainties associated with the Systron Donner calibration information are pictured in Figure 4.2.

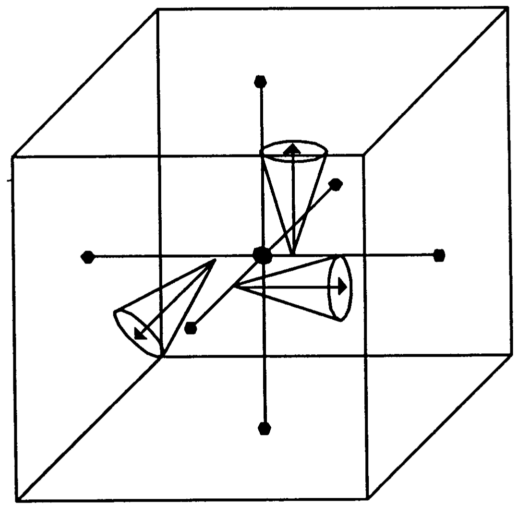


Figure 4.2: Illustration of size effect and alignment uncertainties in IMU

Obviously, the Systron Donner IMU is not intended for stand-alone navigation; it is a good instrument for integrating into a combined GPS/INS navigation system. On the VCUUV, it might perform well for short test runs (less than thirty seconds) around the tank. Its ultimate value might be found in attitude stabilization rather than complete navigation of the fish.

4.2 IMU Temperature Compensation

While we cannot compensate for many of the IMU's shortcomings, we can attack its greatest source of error: bias. While (coarse) bias values and temperature coefficients are provided for each channel on the calibration sheet, better accuracy can be achieved by measuring the biases ourselves, using the actual hardware that will eventually acquire the IMU data. For instance, by hooking the IMU up to the data acquisition system and measuring how the biases change with temperature, we are calibrating for a *composite* bias error that is associated with the IMU as well as the op amps and A-to-D converter that measure the IMU signal. The biases can also be measured in terms of A-to-D quanta, so software compensation need not worry about unit conversion to volts or degrees Celsius. This is sometimes called a "hardware in-the-loop" calibration.

The goal is to characterize the bias (or "zero-input offset") of each channel as a linear function of temperature:

$$Bias = A_1(temp) + A_2$$

where A_1 and A_2 are the parameters of interest: respectively, the temperature-dependent and fixed components of the bias. Determining A_1 and A_2 is fairly simple for the gyro channels. The Tattletale samples and digitally filters the gyro readings while the IMU is placed on a stable, vibration-free surface that is kept *absolutely still* – thus, there is zero motion input. The IMU is allowed to warm up, and is cycled through several slow heating and cooling cycles (using a heat gun) while data (in A-to-D units[22]) is being sent to the 486 and logged to the hard drive. In order to improve the signal-to-noise ratio (since the biases are on the same order of magnitude as the noise on the signals), the logged data is averaged and decimated in groups of 100 – thus at 50 Hz, one value is logged every 2 seconds. The resulting data can be plotted versus the temperature measurements to give a temperature-vs.-bias plot, as shown in Figure 4.3. Usually, this procedure is repeated a handful of times to check for consistency. Each channel's plot from each run is then fitted with a least-squares linear estimate (using 'polyfit' in MATLAB), to give coefficients A_1 and A_2; finally, these parameters are averaged for each channel across the runs to give more accurate values for A_1 and A_2.

The measurement of A_1 and A_2 for the acceleration channels uses the same procedure, with one added consideration. Remember that the bias is the zero-input response; for the gyros, we can achieve zero input by keeping them still. But the accelerometers respond to gravity, so in order to have zero input, we must have the accelerometer's sensitive axis perpendicular to gravity while taking data. This was done very carefully by mounting the IMU on an extremely level granite calibration table, whose surface plane was checked to be perpendicular to gravity using a precision level. Note that we can only measure biases on two acceleration channels at a time this way, because if the x- and y- channels are perpendicular to gravity, the z-channel is necessarily parallel to gravity. After making sure the zero-input condition is satisfied, the IMU is again subjected to heating and cooling, and

[22] Even though the data is in A-to-D units, it is still processed by the digital filter. This is why the data assumes non-integer value in the plots in Figure 4.3.

the acceleration measurements logged and averaged. Following this, least-squares linear fitting and averaging across experiments produce values for A_1 and A_2.

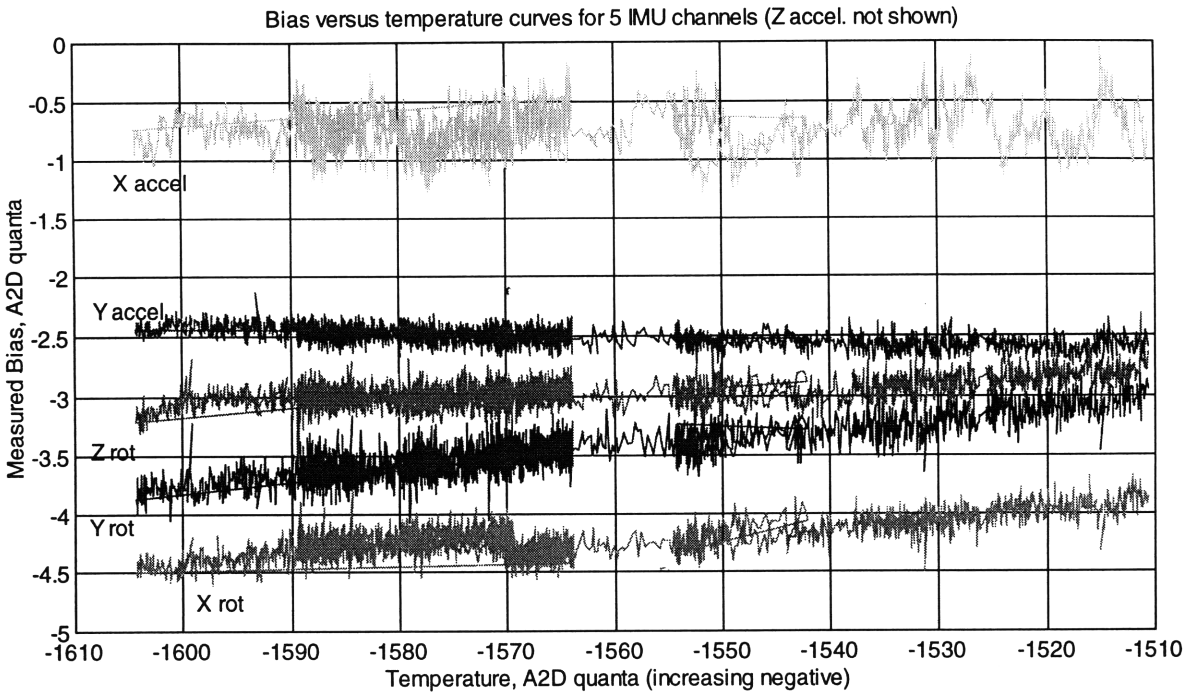


Figure 4.3: Temperature vs. Bias curves

The bias plots in Figure 4.3 appear linear in general, but some have some slight nonlinearity associated with them. For this system, it was decided that a linear fit was good enough and that higher-order fitting wouldn't be much more effective, considering that nonlinear error was slight compared to the amount of noise on the signals.

Once the parameters A_1 and A_2 have been determined for each IMU channel, they are entered into a software algorithm on the TT8 that subtracts the bias from each channel immediately before sending the IMU data to the PC-104 stack.

A result of the temperature compensation, observed experimentally, is the improvement of the gyro drift rate from the coarse bound of 3°/s to an approximate bound of 1°/min. – almost 200 times better drift performance. This is still not standalone quality for long missions, but at least may allow satisfactory standalone performance for short runs.

4.3 Determining Position and Attitude

The next task is to solve the navigation algorithm. This section is somewhat mathematically oriented; much of the background material comes from Titterton and Weston [13], an extraordinarily thorough and up-to-date volume on strapdown inertial technology.

4.3.1 Strapdown Computational Architecture

A strapdown system consists of accelerometers and gyros, support electronics, and the computing devices that continuously implement the navigation algorithm (see Figure 4.4). The algorithm itself is split into two stages: first attitude is computed, then position and velocity. The reason for this hierarchy is that vehicle attitude must first be known in order to resolve the measured accelerations into the correct directions. In other words, the accelerations cannot be integrated to find velocity and position until we know which directions the accelerations were measured in.

Often strapdown systems will use a separate "attitude computer" and "navigation computer" for the two stages. This is a more conceptual organization of the algorithm, and makes the software design easier at the expense of more hardware.

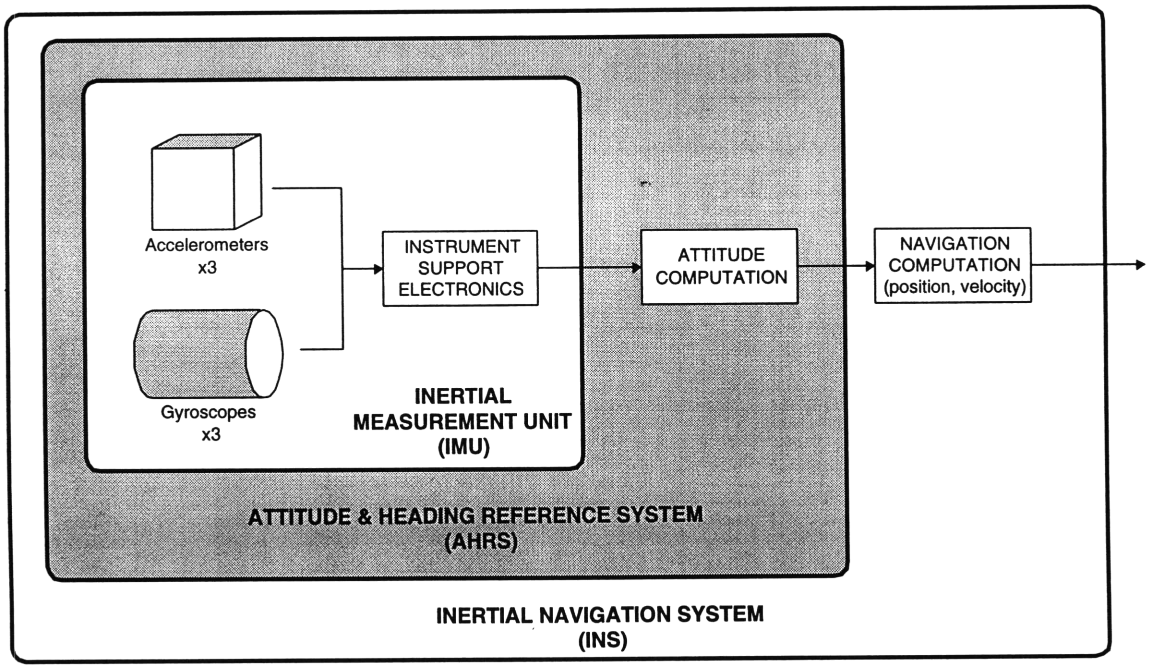

Accelerometers
x3
Gyroscopes
x3
INSTRUMENT SUPPORT ELECTRONICS
INERTIAL MEASUREMENT UNIT (IMU)
ATTITUDE COMPUTATION
NAVIGATION COMPUTATION (position, velocity)
ATTITUDE & HEADING REFERENCE SYSTEM (AHRS)
INERTIAL NAVIGATION SYSTEM (INS)

Figure 4.4: Strapdown architecture
(from Titterton and Weston [13])

The actual algorithm used to compute attitude, position and velocity is known as the *mechanization*. A mechanization describes the sequence of computations needed to convert raw IMU signals (measured in the "body frame") to movements in a given reference frame. An example is shown in Figure 4.5. The diagram also includes certain corrections that need to be made for variations in gravity, rotations of the earth, and Coriolis forces. These are subtleties of inertial navigation that for the most part we will choose to ignore.

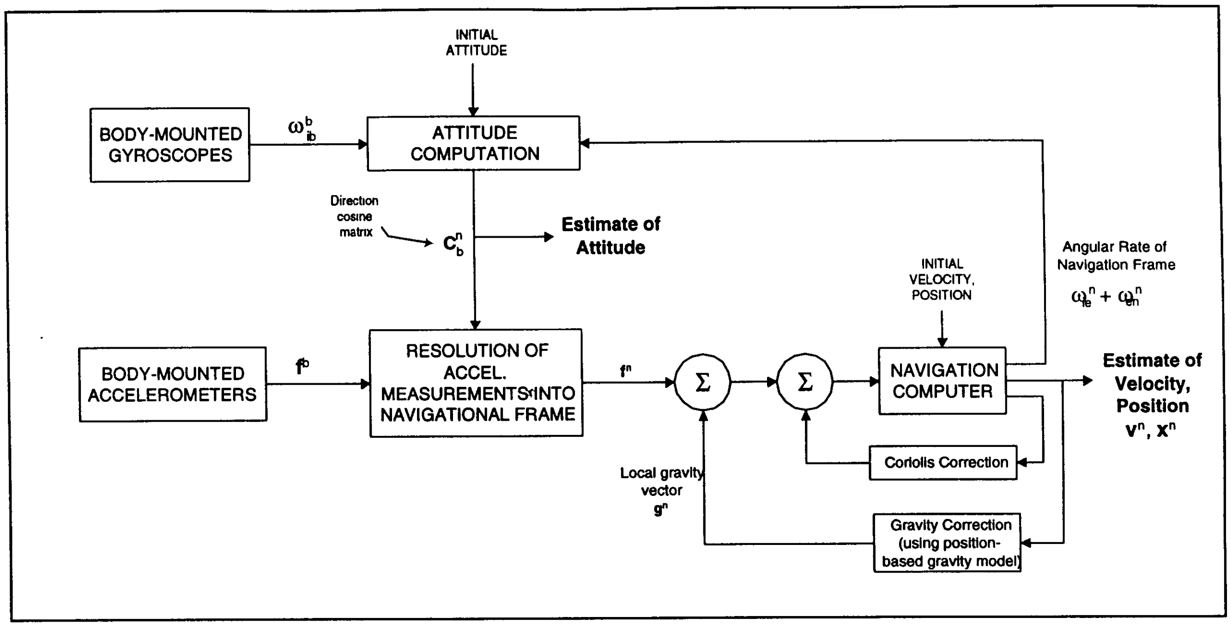

INITIAL ATTITUDE
BODY-MOUNTED GYROSCOPES
ω_{ib}^{b}
ATTITUDE COMPUTATION
Direction cosine matrix
C_b^n
Estimate of Attitude
INITIAL VELOCITY, POSITION
Angular Rate of Navigation Frame
$\omega_{ie}^{n} + \omega_{en}^{n}$
BODY-MOUNTED ACCELEROMETERS
f^b
RESOLUTION OF ACCEL. MEASUREMENTS INTO NAVIGATIONAL FRAME
f^n
Σ
Σ
NAVIGATION COMPUTER
Estimate of Velocity, Position
v^n, x^n
Coriolis Correction
Local gravity vector
g^n
Gravity Correction (using position-based gravity model)

Figure 4.5: Mechanization from body to navigation frame
(from Titterton and Weston [13])

4.3.2 Reference Frame versus Body Frame

There are six or seven different Cartesian coordinate systems (or "frames") that are defined for different applications of inertial navigation. In strapdown systems, the one frame that is always of interest is the *body frame*. This is defined by a set of x, y, and z coordinates that are based on the vehicle and travel with the vehicle – essentially, these axes are aligned with the accelerometers on the vehicle's IMU. Thus the body frame of axes is equivalent to the orthogonal axes of the vehicle's IMU.

As the vehicle travels in space, the body frame changes attitude and distance with respect to another frame of axes, the *reference frame*. Reference frames can be any one of several common choices; usually, they involve the fixing of the center of the earth, or the stars, as a reference origin. The reference frame most useful for local navigation over small (essentially flat) regions of the earth, such as the VCUUV will perform, is called the *navigational* frame. This frame has its origin at some point on the earth, with the axes pointing north, east, and down.

During vehicle travel, the body frame axes change attitude with respect to the navigational frame. The critical part of the strapdown computation is to be able to track this change in attitude. This is because before the acceleration signals can be double-integrated to provide position, they must be correctly resolved into the north, east, and down directions of the reference frame. Resolving the acceleration vector (and compensating for gravity) requires precise knowledge of the vehicle's changing attitude.

For this reason, the strapdown algorithm always involves updating the current vehicle attitude as the first step of every position update. This is also a large part of why attitude (or gyro) errors are the limiting factor in inertial system performance. Gyro error not only contributes to attitude errors that increase proportional to time, but they result in resolution of accelerations into incorrect directions – and it can be shown that this causes position estimates to drift roughly proportional to time cubed[23].

Perhaps the trickiest part of the strapdown computation is the representation of attitude. Knowing the vehicle's attitude means knowing the relationship between the vehicle's body frame axes and the axes of the reference frame. This relationship must be updated using angular rates arrive from the gyros – but the catch is, attitude cannot be characterized by integrating each of the angular rates alone to get three angles, because *rotations do not commute.* In other words, rolling an object 90° and then pitching it 90° (in body axes) will produce a different attitude than pitching it 90° first and then rolling it, even though separately integrating the resulting gyro signals would produce the same "attitude" in each case. Changing attitude must be updated by considering a small set of three orthogonal unit vectors which represent the body frame, and how these vectors are *each* rotated by *each* of the gyro signals relative to the three orthogonal unit vectors that define the reference frame. The geometry can get quite complicated when working this problem in three dimensions, so only the fundamentals will be discussed here.

There are several different ways attitude can be represented in navigational systems: these include direction cosine matrices, Euler angles, and quaternions. The direction cosine matrix is a three-by-three matrix that evolves with time as the vehicle attitude changes, where the (i,j) element of the matrix represents the cosine of the angle between the ith axis of the reference frame and the jth axis of the body frame. For instance, if the body-frame attitude coincides with the reference frame axes, the direction cosine matrix is simply the identity matrix. In essence, this matrix describes (as a function of time) how each body-frame axis can be *resolved* (or projected) into the reference-frame axes, and thus provides a complete picture of attitude. Euler angles are a more intuitive approach for some, since they involve three angles (evolving in time) which describe successive rotations of the reference frame that are needed to make it coincide with the body frame. However, Euler rotations must occur in the correct order, since as mentioned before, rotations do not commute! Quaternions are a more complicated system involving four parameters instead of three, and have the advantage that their equations have no singularities – a problem seen in both direction cosine matrix and Euler angle navigation. Such singularities, which occur during particular attitude maneuvers, can be disastrous in spacecraft and other high-importance navigation tasks[24].

The chosen method for the processing the VCUUV navigation data is the direction cosine method. This is a popular algorithm, and we do not expect singularities with the fairly standard changes in attitude that the fish will experience. The basic idea is the following:

[23] The derivation of this is beyond the scope of this thesis, but can be found in Titterton and Weston [13].

[24] It is interesting to note the analogies here between strapdown and platform systems. Quaternions, a four-parameter strapdown algorithm, prevent singularities in the attitude computation; whereas in a platform system, a mechanical assembly with four gimbals instead of three completely prevents a possible mechanical failure called gimbal lock! Quaternions are the computational analog to a set of four gimbals.

For a set of x, y, and z accelerations that arrive at (discrete) time interval *k*, these accelerations can be resolved into the axes of the reference frame by multiplying by the direction cosine matrix at time *k*:

$$\begin{bmatrix} x^n \\ y^n \\ z^n \end{bmatrix} = \mathbf{C}_b^n \begin{bmatrix} x^b \\ y^b \\ z^b \end{bmatrix}$$

where the superscripts *n* and *b* refer to navigational and body frames respectively, and the direction cosine matrix **C** is designated to make the transformation between these frames. Once this relationship is established, all that remains is to determine the time-evolution of the direction cosine matrix, or equivalently, the changing attitude of the vehicle over time. Since we are simply post-processing the inertial data in this section, we will compute the complete attitude path before resolving all of the acceleration vectors.

In continuous time, the direction cosine matrix evolves as a function of the current attitude and the input rate values from the gyroscopes, as follows:

$$\dot{\mathbf{C}}_b^n(t) = \mathbf{C}_b^n(t)\Omega_{nb}^b(t) \quad \text{where} \quad \Omega_{nb}^b = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$$

However, since the gyro inputs (notated as $\omega_{(axis)}$ above) in all strapdown systems arrive as discrete-time values to a digital computer, a slightly different analysis is used. The discrete-time (DT) evolution of the direction cosine attitude matrix can be shown to be (see Titterton and Weston [13], pp. 294-97):

$$\mathbf{C}_{k+1} = \mathbf{C}_k \mathbf{A}_k \quad \text{where} \quad \mathbf{A}_k = \mathbf{I}_{3x3} + \frac{\sin\sigma}{\sigma}[\sigma\times] + \frac{1-\cos\sigma}{\sigma^2}[\sigma\times]^2$$

with the input matrix of gyro signals (σX) similar to Ω above:

$$[\sigma\times] = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \bullet \Delta t \quad \text{and} \quad \sigma \equiv \Delta t \bullet \sqrt{\omega_x{}^2 + \omega_y{}^2 + \omega_z{}^2}$$

where Δt is the sample period $\left(\frac{1}{50Hz}\right)$.

There is one immediate problem: the evolution equation requires an initial condition for the matrix $\mathbf{C}_0$; in other words, the initial attitude of the vehicle must be supplied! This is known as *alignment*. Alignment is a critical matter, since if the initial attitude is wrong, all the subsequent attitude calculations will drift even worse, and the position drift will be obscene. The best method of determining the initial attitude is to recall the alignment technique discussed in Section 2.4; using the accelerometers to measure the gravity vector, and back-tracking to determine the "tilt" of the IMU with respect to gravity.

4.3.3 Alignment

In platform systems there is a mechanical procedure used to align the initial platform attitude, called gyrocompassing. There is an analogous analytic procedure for strapdown systems. This essentially involves determining attitude by measuring the tug of gravity on the accelerometers, using the prior assumption that no true acceleration is being experienced.

Note that there are a few things we ignore while aligning the VCUUV inertial system. Gyrocompassing usually involves measuring the rotation of the earth, to align the attitude in the azimuth (compass heading) plane. However, we have chosen to ignore the earth's rotation, as well as Coriolis forces, etc.; the earth rotates at $0.004^\circ/\mathrm{s}$, a rate which is considered beneath the resolution of our gyros. Therefore the only alignment we are truly concerned with is alignment with the vertical: that is, making sure we know the attitude of the IMU relative to the direction of gravity. The azimuth alignment, which determines where north and east are, is considered unimportant for the local navigation the fish will implement. When navigating in a local environment (i.e. a pool), we can reference a "relative azimuth" by calling some direction "north" and its perpendicular "east." Eventually, the compass can be used to augment this and bound drift errors in relative azimuth.

Alignment is equivalent to determining the initial direction cosine matrix. If the IMU is stationary, the force of gravity measured by each accelerometer is the ***measured gravity vector*** $\begin{bmatrix} g_x & g_y & g_z \end{bmatrix}^{\mathrm{T}}$. This vector is measured in the body frame, and is related to the gravity field by a direction cosine matrix transformation:

$$\mathbf{C}_b^n \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -9.80 \end{bmatrix}$$

where -9.80 is the value of the gravitational acceleration ($\mathrm{m/s^2}$) in the reference frame[25]. The task is to determine $\mathbf{C}_b^n$.

[25] This is very close to the gravitational acceleration in Boston, MA. Gravity is assumed to be aligned with the vertical, or very close to it.

Multiplying both sides by the matrix inverse gives us

$$\begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} = \mathbf{C}_b^{n^{-1}} \begin{bmatrix} 0 \\ 0 \\ -9.80 \end{bmatrix}$$

One very nice property of all direction cosine matrices is that they are orthogonal. This means that each column is of unit magnitude, and the dot-product of any two columns is zero: the columns form an orthogonal unit-vector subspace. In addition, this means that the inverse of a direction cosine matrix is simply the transpose. Thus if

$$\mathbf{C}_b^n = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}, \quad \text{then} \quad \mathbf{C}_b^{n^{-1}} = \mathbf{C}_b^{n^{\mathrm{T}}} = \begin{bmatrix} c_{11} & c_{21} & c_{31} \\ c_{12} & c_{22} & c_{32} \\ c_{13} & c_{23} & c_{33} \end{bmatrix}.$$

From the previous equation, then, it is easy to solve for the last column of $\mathbf{C}_b^{n^{\mathrm{T}}}$, i.e. the terms c_{31}, c_{32}, and c_{33}:

$$c_{31} = {}^{-g_x}\!/_{9.8}, \quad c_{32} = {}^{-g_y}\!/_{9.8}, \quad c_{33} = {}^{-g_z}\!/_{9.8}$$

This is the important part of the alignment: the part of the matrix which aligns the body frame level with respect to gravity in the reference frame. The rest of the matrix terms deal with azimuth alignment, which we are unconcerned with. Therefore, the matrix is filled out using the Gram-Schmidt orthogonalization process, which constructs two unit-magnitude vectors orthogonal to [c_{31} c_{32} c_{33}] by taking the reference frame axes ([1 0 0], [0 1 0]), projecting them onto [c_{31} c_{32} c_{33}], and subtracting these projections from the original reference frame axes. This is a standard technique discussed in most linear algebra textbooks [12]. Having determined $\mathbf{C}_b^n$, the alignment is complete.

4.4 Cart Tests

The direction cosine algorithm is easily coded into MATLAB. To test it, a number of short trips were taken while logging IMU data on a portable cart around Draper Lab. These tests ranged from 30 to 40 seconds, and involved straight runs, turns, rectangular paths, and up and down in the elevators. Some data analysis of this IMU data will be attempted using the MATLAB code developed.

The code for the MATLAB algorithm is provided in Appendix C. The program begins by taking a long vector of repeating IMU "packets" and unraveling it into vectors for each channel (accelerometer and gyro) of IMU information. The data[26] is then multiplied by appropriate scale factors to change the numbers into engineering units (m/s^2, and rad/s). Next, the initial attitude is determined using the alignment technique just described. The initial attitude takes form as the first direction cosine matrix $\mathbf{C}_0$.

Using the three vectors of gyro information, the entire sequence of direction cosine matrices is computed using the recursive algorithm described above – that is 1500 matrices for a 30 second run! The sequence of matrices can be plotted to visualize the changes in attitude. Finally, position can be computed as follows. Each triad of accelerometer measurements at time *k* is multiplied by the direction cosine matrix of the same time: this resolves all the body-frame measurements into the reference frame axes. Now, the value of gravity can be removed from the resolved z-axis, since in the reference frame, the z-axis coincides with the direction of gravity! Having resolved the accelerations into the reference frame, position in the reference frame is easy to compute: just double-integrate each of the resolved acceleration channels. This of course requires the initial conditions of position and velocity, but since the cart test began at a standstill, the initial conditions are zero. The resolved accelerations in x, y, and z can be integrated separately, since accelerations in different directions, unlike rotations, *do* commute.

4.4.1 Results

Six cart tests were done in all. Three are presented here: a straight run, a rectangular trajectory, and a trip in an elevator. The following pages show 3 plots for each experiment:

- **Attitude plot**: To understand these graphs, imagine the body axes of the IMU as an orthogonal triad of unit vectors. This plot shows the endpoints of the body axes plotted with respect to the three-dimensional reference frame. (It is easier to visualize this changing attitude in MATLAB because as the axes move with time, each endpoint is a tracked with a different color asterisk.) The direction of gravity is along the reference frame z-axis. Because of our alignment procedure which ignores azimuth, the body-attitude at time zero is always perfectly aligned with the reference frame axes.
- **Resolved accelerations plot**: These are the accelerations measured by the IMU and resolved using the direction cosine matrix. Thus the accelerations pictured should (theoretically) be directed along the axes of the reference frame. Notice that they are somewhat noisy; this is largely because the tests were conducted on a cart that was quite prone to vibration. Theoretically, these vibrations should be seen equally to any other motion input, but in some cases (as we will see) they are an undesirable disturbance to the system.
- **Position plot**: The separate X, Y, and Z coordinates of the position are shown versus time. These graphs are just the double-integrations of the resolved accelerations in the previous plots. Zero initial position and velocity are assumed for all the experiments.

[26] Remember that the data has already been temperature-compensated by the Tattletale.

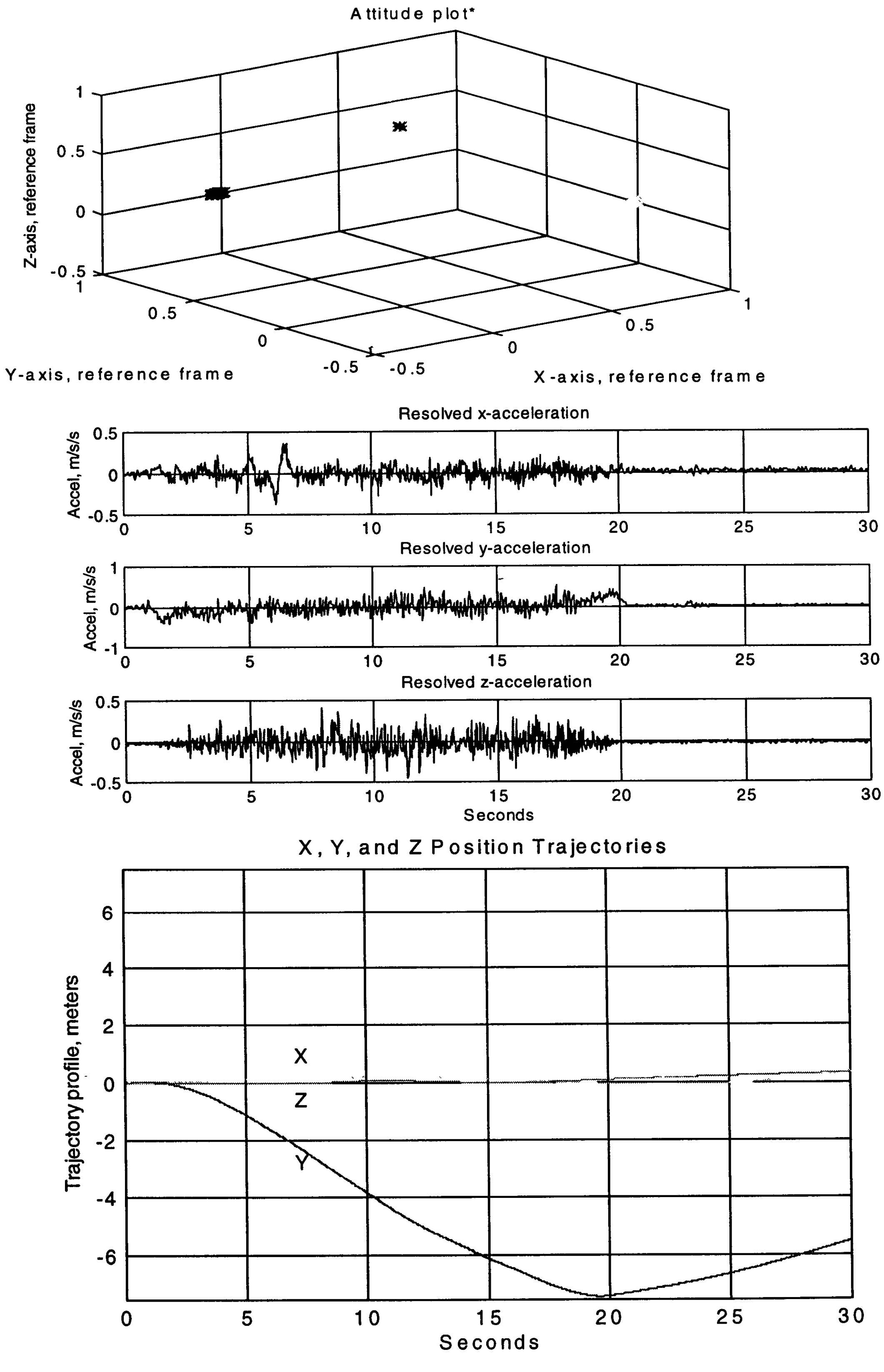


Figure 4.6: Post-processed IMU data — Straight run

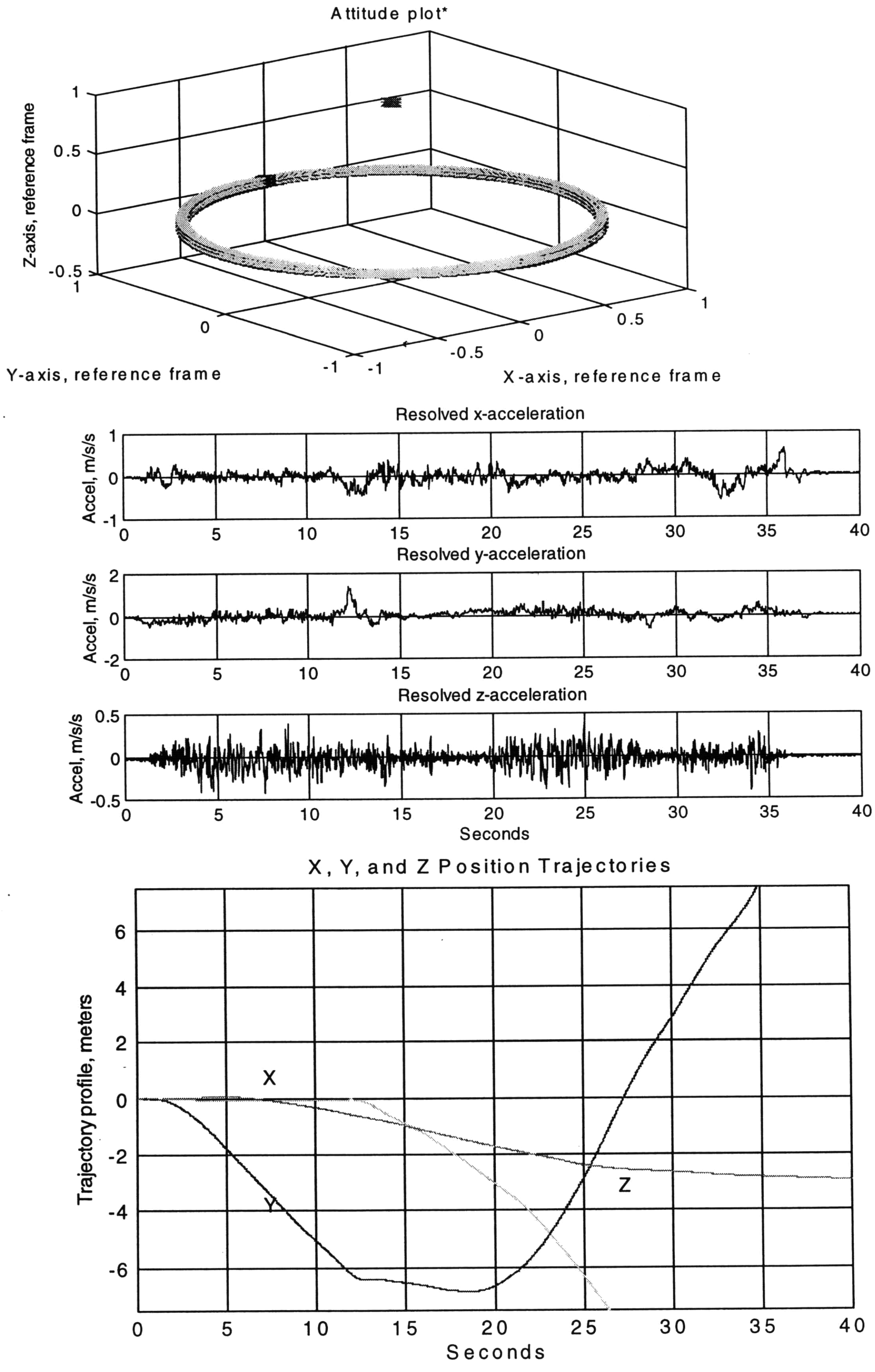
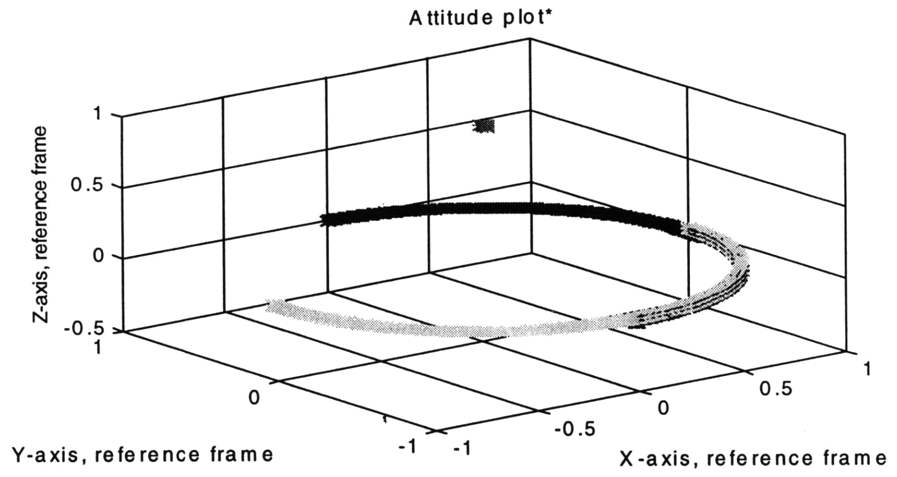
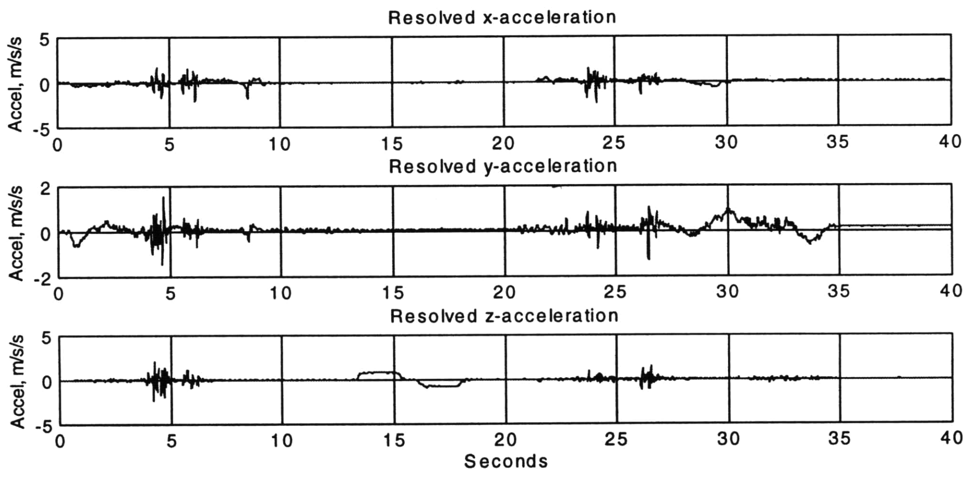


Figure 4.7: Post-processed IMU data – Rectangle





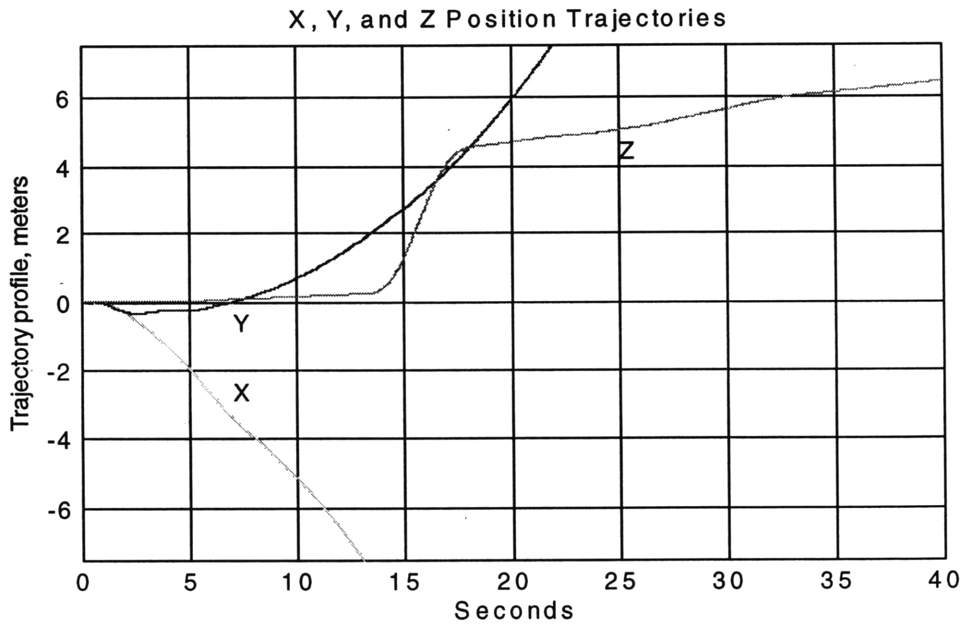


Figure 4.8: Post-processed IMU data – Elevator

Below we describe the experiments that generated these plots, and make some qualitative observations on the system performance. These experiments themselves were somewhat qualitative, and were performed in order to get a "feel" for the inertial system.

4.4.1.1 Straight Run

The straight run experiment simply involved rolling the cart (initially at rest) in a straight line for approximately 8.5 meters and then decelerating to a stop. The IMU was positioned so that the direction of the straight line was aligned with the IMU's Y accelerometer axis.

The attitude plot shows three roughly static points; each corresponds to the endpoint of a body-frame vector. The orthogonal body-axis set can be imagined by connecting each point to the origin with a vector. Notice that the points do meander a small amount due to the slight angular wandering of the cart, but in general, the data correctly shows that no angular rotations are experienced during this experiment.

The position plot shows that the Y coordinate begins to increase negatively, and gets to about 7.5 meters before it turns back up again. This motion reflects the straight line run, and can be explained as follows. As the cart is accelerated from rest, the Y coordinate displays a quadratic position change. The cart steady states to about a constant velocity, and so we see a linear ramp in position between 5-13 seconds. At about 13 seconds, there seems to be a slight "kink" in the graph; this is where tiny errors have been integrated to the point where they begin to blow up, in a positive direction. Thus, the graph never makes it all the way down to the actual 8.5 meters traveled, since the drift takes over at around 13 seconds in the Y direction.

The X coordinate motion appears to be fairly well behaved; since virtually no motion occurred in the X axis, this plot appears to be correct. Similarly, the Z coordinate hardly moves, since there is no motion input. In this experiment both X and Z seem to be fairly well compensated against drift.

The conclusion from this experiment is that estimating position using the IMU alone can be done fairly accurately for a maximum of about 13 seconds.

4.4.1.2 Rectangle

This experiment was also performed in the X-Y plane (i.e., no vertical motion except perhaps vibrations). It consisted of rolling the cart in a rectangle of approximate dimensions 6 meters by 2 meters. At each corner of the rectangle, the cart was rotated 90° before traveling along the next side.

Rotation of 90° at each corner results in the entire IMU being rotated a full 360° (in the yaw plane) over the course of the experiment. The attitude plot reflects this fact: the X and Y body vectors of the IMU have each traced out the circle in the yaw plane of the reference frame. Meanwhile, the Z vector has not changed attitude, since no pitching or rolling occurred (except perhaps due to vibrations).

The position plot shows a linear ramp of the Y position coordinate until about 12 seconds; then the cart is rotated and linear position commences along the X axis. However, drift takes over before these two sides of the rectangle have been traversed; by 20 seconds, the estimates are worthless. The Z direction appears to have drifted more in this experiment than the last; about 2 meters within 20 seconds. However the overall Z drift is less than that seen in the X and Y directions. Overall, the position estimate again seems to be reasonably good up to about 13 seconds from the start of the experiment.

4.4.1.3 Elevator

Finally, we desired an experiment to test motion along the Z axis. The elevator experiment involved first rotating the cart 90°, walking about 1 meter straight into an elevator, traveling up one floor, backing out of the elevator and rotating backwards 90° (so that our net rotation in the yaw plane is 180°).

The attitude plot clearly shows the X and Y body vectors each rotating 180° in the yaw plane of the reference frame. The Z vector, as expected, is stationary. In general, the attitude plots we have seen so far have been quite accurate and shown little drift. This indicates that the gyro bias-temperature compensation was done successfully.

The position plot lends some insight to the a serious limitation of the system. Look at the graph in the middle of the page, of resolved accelerations. The Z direction travel of the elevator is clearly seen between about 14 to 18 seconds; a very nice trapezoidal acceleration and deceleration profile. However, we also see some very large and high frequency acceleration spikes occurring at 5 seconds and 26 seconds; what are these? What happened in the experiment was that the cart was seriously jolted when crossing the threshold into and out of the elevator – consequently, the large spikes of acceleration. While ideally the IMU should see these spikes as normal motion input, we see that at around 5 seconds (at the first jolt) the X and Y position begin to drift like mad. The plausible explanation is the following: when the cart was jolted, the IMU bounced around and changed attitude very rapidly. The change in rotation (or angular acceleration) was fast enough that the low-pass anti-aliasing filters corrupted the high-frequency motion information, resulting in a perceived gyro error. This gyro error subsequently caused errors in the attitude computation, and attitude error will quickly lead to extreme drift in position. The fact that only the X and Y coordinates were affected – notice that the Z coordinate correctly tracks the elevator motion, despite some drift – implies that the high frequency attitude change occurred mostly through yawing.

The observation, therefore, is that the position estimate can be severely affected by sudden motion shocks to the system, due to the corruption of high-frequency motion by the anti-aliasing filters, and perhaps also from gyro errors induced by conical vibrations.

4.5 Conclusions

A lot of insight on the system's performance was gained from the cart tests. One qualitative observation was that the measured accelerations for various maneuvers were not as great as expected; for instance, during all of the tests rarely did we see accelerations of above 1*g*. The conclusion is that the IMU full-scale range of ±5*g* on each accelerometer is probably excessive; and since our circuitry scales the signals to use this full range, we can probably significantly increase the signal-to-noise ratio (SNR) and resolution of or accelerometer readings by not using the full range. This is definitely an improvement worth making to the system in the future. It involves replacing a precision resistor in the scaling circuit of Figure 3.8 to change the gain, and recalibrating the temperature-bias compensation and scaling factors in both the TT8 and QNX software (more bias testing would probably need to be done). This improvement could potentially improve the drift performance quite significantly.

We also learned from these tests that large and sudden *g*-shocks and vibrations in the system cause errors which can cause the position error to blow up prematurely, even if the accelerations are all under 5*g*. The reason for this, again, is probably in the anti-alias filtering which destroys all high-frequency components of the measured accelerations upon sampling. Furthermore, out-of-phase vibrations around two axes (a motion called "coning") can occur, which causes false rotational input to the gyros; this throws off the attitude estimate. As a general rule, this system was not designed for and does not deal well with high-frequency (>20 Hz) motions.

Finally, we have demonstrated that on a typical run, inertial dead-reckoning alone on the VCUUV can provide a reasonably good position estimate for about a maximum of about 15 seconds. Immediately after this time, however, position errors blow up rapidly, as expected. What seems to be much more stable and accurate than position estimation was attitude estimation. The plots above indicate that tracking the fish's attitude might be achieved with good accuracy over a period of several minutes or more.

Was the temperature compensation worthwhile and/or successful? A similar inertial system can be presented for comparison. The Unmanned Aerial Vehicle(UAV) group at Draper Lab has developed an autonomous helicopter that uses the same Systron Donner IMU as the VCUUV. In this system, standalone IMU navigation can not be relied on for more than 5 seconds: it is usually corrected at 5-10 Hz using GPS and altimeter readings in a Kalman filter. Unlike the system described in this thesis, the helicopter inertial system performs no temperature compensation of the bias. The VCUUV system has demonstrated position drift improvement of perhaps a factor of two over that of the copter. Of course, the fish has no GPS capability, and its position error can only be bounded in the z-direction using the pressure sensor. This unfortunate fact means that until a new means is found to correct the position drift in x- and y-channels, only short navigational missions of under 20 seconds are feasible. Attitude stabilization may still be achieved for longer periods of time, however.

Chapter 5 Modeling, Simulation and Control

Within the VCUUV, there are two levels of control: controlling the tail to swim in a "fishlike" manner, and controlling the larger vehicle to swim to a desired location in the pool. The latter task, also known as *guidance*, is the main concern of this chapter. In general, guidance and control of any vehicle is an extremely complicated problem, involving a considerable amount of modeling and statistical analysis. This chapter attacks only a small piece of the puzzle, by developing a simple model to analyze the behavior of the VCUUV in two dimensions.

One important issue addressed here is the fact that vertical forces on the fish can only be generated by changing the "angle of attack" of the pectoral fins, which act as control surfaces. The upshot of this is that fish cannot move in the *z*-direction, or "dive-plane," without first having some horizontal velocity. Much of the work in this chapter, therefore, is directed towards modeling the relationships between horizontal motion, vertical motion, and pectoral fin angle of the VCUUV. A simple depth controller is designed and implemented as well.

5.1 Model Development

In the course of this research, some steps were taken towards creating a dynamic mathematical model of the fish. For simplicity, this model ignores the third dimension and is developed in a two-dimensional pool (see Figure 5.1), also known as the *dive-plane*. Its primary usefulness will be shown in analyzing the depth control capabilities of the VCUUV.

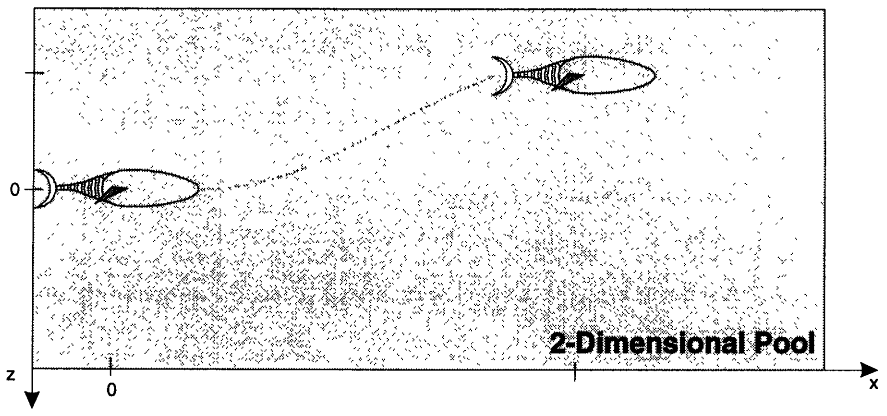


Figure 5.1: 2D setting for model development (dive-plane)

The model assumes that there are two physical means of controlling the movement of the fish: the propulsive force from swinging the tail, and the angle of attack of the pectoral fins. The tail thrust is modeled as a propulsive force in the positive x-direction (in Newtons) that can be directly controlled[27]; this force will be called $\mathbf{F_x}$. Any vertical forces on the fish are created by angling the pectoral fin, which acts as a control surface. These forces will be characterized later. Finally, note that the position of the fish is designated by coordinates (x,z), with z defined positive in the *down* direction.

To model the motion of the fish in water, we turn to classical hydrodynamics. In general, the three-dimensional motion of an underwater vehicle is characterized by state equations which involve the continuous-time evolution of 6 state variables: velocity in x, y, and z, directions, plus pitch, roll and yaw rates. However, since we are dealing with a two-dimensional system, we do not deal with roll, yaw, or velocity in the y-direction[28]. Thus we are left with three state variables of interest: horizontal velocity, vertical velocity, and pitch rate $(\boldsymbol{u, w, q})$. These are velocity terms (linear and angular); in addition, we would like our state-vector to have positional variables (essentially the integrals of u, w, and q), so we add three more variables $(\boldsymbol{x, z, \theta})$. As explained above, the two controlling inputs are the horizontal thrust $\mathbf{F_x}$, and the pectoral fin angle (which we call ϕ).

Thus the general state-space representation of the system is of the form:

$$\begin{bmatrix} \dot{u} \\ \dot{w} \\ \dot{q} \\ \dot{x} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = f\,(u, w, q, x, z, \theta, F_x, \phi)$$

This form describes the rate of change of the state variables as a function of the current state variables and the control inputs; in other words, the time-evolution of the system as the control inputs are varied.

Using the decoupled hydrodynamic model from [1], we can fill in the equations that make up this state-space system. Unfortunately, we find that the actual system is extremely nonlinear, even after a number of simplifications are applied. The state equations for the system are given in Figure 5.2.

[27] While the tail-motion/thrust relationship has not yet been characterized, that is one of the eventual goals of the VCUUV project.

[28] In removing these, we are "decoupling" the longitudinal-plane equations from the lateral-plane equations. The simplifications necessary to do this are outlined in a paper by Jamie M. Anderson, entitled "Unmanned Undersea Vehicle Model Analysis for Low speed Controller Development [1]."

$$\dot{u} = \left(\frac{1}{m - X_{\dot{u}}}\right)\left[\left(X_{wq} - m\right)wq + \left(X_{u|u|}\right)u|u| + F_{xthrust} + (\pi\rho A)u^2 \underbrace{\left(\tan^{-1}\frac{w}{u} + \phi\right)}_{\alpha}\sin\left(\tan^{-1}\frac{w}{u}\right)\right]$$

$$\dot{w} = \left(\frac{1}{m - Z_{\dot{w}}}\right)\left[\left(Z_q + m\right)uq + \left(Z_w\right)uw - (\pi\rho A)u^2 \underbrace{\left(\tan^{-1}\frac{w}{u} + \phi\right)}_{\alpha}\cos\left(\tan^{-1}\frac{w}{u}\right)\right]$$

$$\dot{q} = \left(\frac{1}{I_y - M_{\dot{q}}}\right)\left[\left(M_q\right)uq + \left(M_w\right)uw - \left(z_G W - z_B B\right)\sin\theta + (\pi\rho A)u^2 \underbrace{\left(\tan^{-1}\frac{w}{u} + \phi\right)}_{\alpha}\right]$$

$$\dot{x} = u\cos\theta + w\sin\theta$$

$$\dot{z} = -u\sin\theta + w\cos\theta$$

$$\dot{\theta} = q$$

Figure 5.2: Nonlinear state-space model for dive-plane analysis

These equations are arrived at through the decoupling of the lateral (*x*-*z*) plane equations from the general three-dimensional hydrodynamic equations that describe the motion of a rigid-body in water. Note is that *u*, *w*, and *q* are in vehicle *body-coordinates*, and *x*, *z*, and θ are in *global coordinates* (see Figure 5.3). Thus, the *x*-*z*-θ variables represent not only a time-integration of the *u*-*w*-*q* velocity variables, but a coordinate transformation as well. Note also the numerous constants throughout the equations, that correspond to vehicle mass, moment of inertia, center of gravity, etc., as well as skin-friction and drag coefficients. We will not dwell on the meanings of these coefficients, but trust that their derivation is accurate. The values chosen for these constants (derived by Jamie Anderson, group manager) are available in Appendix D.

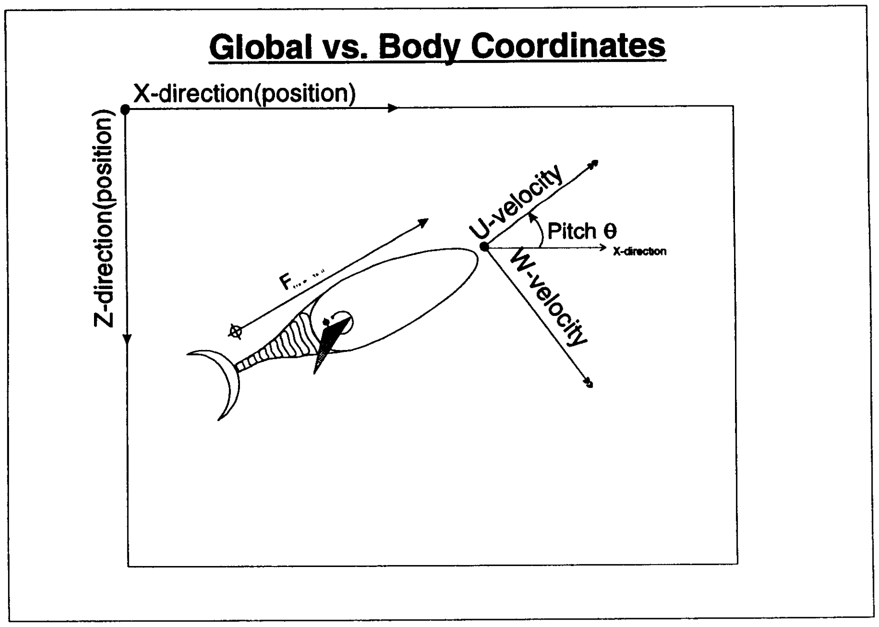


Figure 5.3: Coordinate systems in used in dive-plane model

In developing these equations several simplifications were made, including:

- no fin drag model
- neglect viscosity effect except for forward, simple drag
- grossly simplified propulsor model
- no pitch damping
- prolate ellipsoid estimates for inertial coefficients (not including caudal fin)

The highly nonlinear model was further analyzed using SIMULINK[29]. The state equations of the model were incorporated into a SIMULINK model (pictured in Figure 5.4), which allows a user to run a real-time simulation of the two-dimensional fish. The simulation also drives an animation of the fish swimming through the pool as the inputs (F_x, ϕ) are manipulated manually. The animation shows the (x, z) position of the fish, as well as its pitch angle θ.

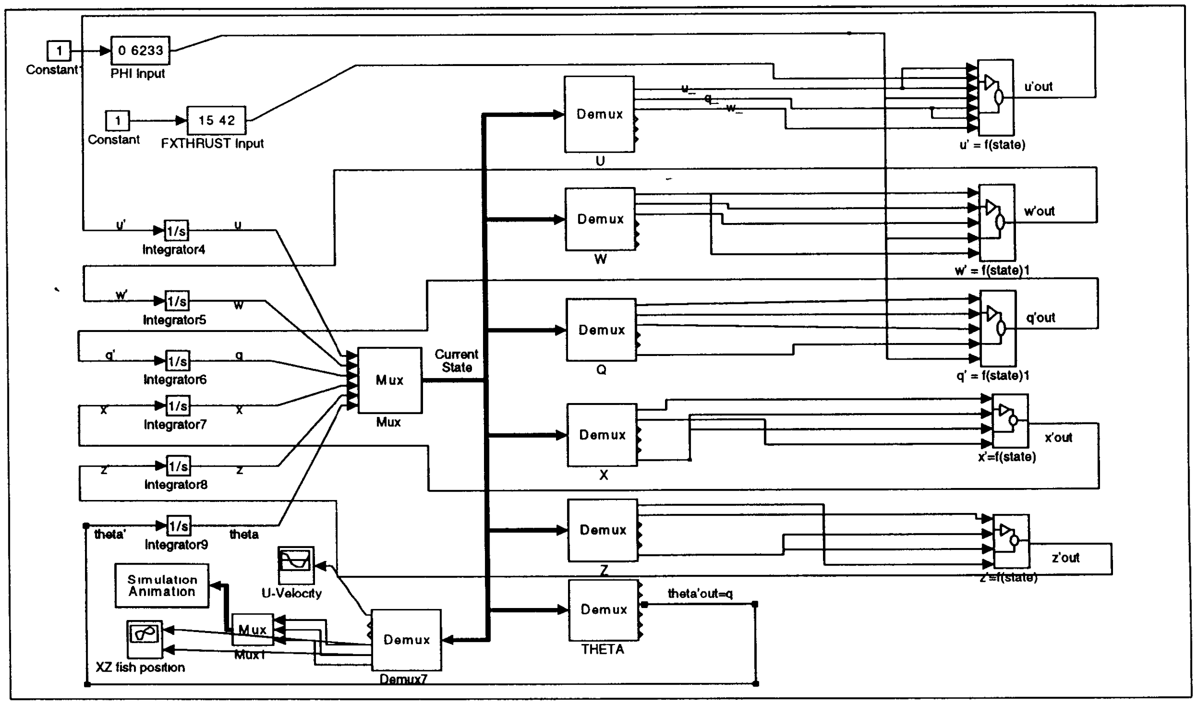


Figure 5.4: SIMULINK block diagram

Using the SIMULINK model, various "virtual" experiments could be run. By simulating trial runs with various thrusts and pectoral fin angles, some of the following important questions could be investigated (qualitatively, in some cases):

- What is the steepest slope ($\Delta z/\Delta x$) at which the fish can dive?
- Is the dive slope a function of forward velocity or only of pectoral fin angle?
- At what angle do the pectoral fins stop providing useful lift, or stall?
- How stable is the fish's pitch during rising or climbing? How much pitch oscillation occurs when the pectoral fin is "stepped" from 0° to some positive angle of attack?

[29] SIMULINK is an add-on to the MATLAB software package which allows very sophisticated nonlinear systems to be modeled and simulated using a very easy block-diagram interface.

These questions were addressed as follows. The simulation experiment involved applying a constant tail thrust with zero angle of attack until the fish's horizontal velocity steady-stated (due to drag forces countering the thrust). Then the pectoral fin angle was "stepped" to some value, and the slope of the resulting dive and oscillations in pitch were observed. Thus the dive-slope is characterized as a function of both the forward velocity of the fish (which, at steady-state, is a direct function of the thrust from the tail) and the pectoral fin angle of attack. The following data were obtained:

Constant thrust F_x, newtons	Steady-state horizontal velocity, m/s	Fin Angle of Attack, radians	Slope of dive, $\Delta z/\Delta x$ (unitless)
5	1.25	0.1	0.064
10	1.75	0.1	0.088
16[30]	2.06	0.1	0.112
20	2.15	0.1	0.136
5	1.25	0.2	0.120
10	1.75	0.2	0.160
16	2.06	0.2	0.200
20	2.15	0.2	0.233

Table 5.1: Dive-slope versus forward speed and angle of attack

The data clearly shows that dive-slope is an increasing function of both angle of attack and forward velocity. In other words, the amount of dive Δz the fish can achieve in a horizontal distance Δx can be increased either by swimming faster, using a greater angle of attack on the pectoral fins, or both.

Furthermore, the SIMULINK animation predicted that upon stepping the fin angle, the fish can experience large pitch oscillations on the order of tens of degrees before settling. This result is somewhat surprising. Along with the other interesting observations presented here, it should soon be tested against experimental data from actual swim tests.

5.2 Depth Control Loop

We can apply the insight gained from the simulated fish's dive-plane behavior to a simple controller design. The design goal is a feedback controller that will regulate the fish's depth through readings from the pressure sensor[31]. The pressure sensor readings alone are used for feedback since the navigation software for the IMU is not yet complete. Furthermore, the pressure sensor is a far more effective sensor for depth measurements, since it does not provide a measurement that drifts with time. The controller is developed around the simulated fish's behavior at a nominal speed of 4 knots.

[30] 16 Newtons of tail thrust results in the nominal forward velocity specified for the VCUUV, 4 knots (2.06 m/s).

[31] Though the controller will be implemented in discrete time (DT), it will be designed in a continuous time (CT) setting. Observation from the SIMULINK model suggests that our DT control rate is sufficiently fast (compared to the dynamics of the depth behavior) to justify a CT analysis.

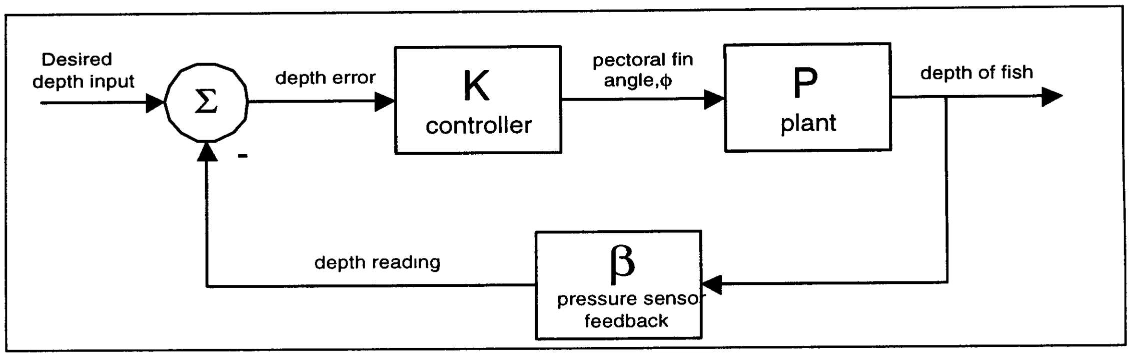


Figure 5.5: Depth feedback loop

The control system is single-input, single-output (SISO), and takes the general form shown in Figure 5.5. Essentially, we want to design a good controller **K** that suits the properties of the plant, **P**. The plant represents the (linearized) input-output relationship from a pectoral fin angle (with both fins set at the same angle of attack ϕ) to the depth (z-coordinate) of the fish.

The plant was characterized using the SIMULINK model. It was observed that a fixed angle of attack eventually results in a fixed-slope dive (i.e., fixed $\Delta z/\Delta x$), while traveling horizontally at its nominal velocity of 4 knots. Furthermore, the relationship between angle of attack and dive-slope ($\Delta z/\Delta x$) – at the nominal velocity – is very nearly linear. This observation comes from a two-dimensional "slice" of the data from Table 5.1, and is plotted in Figure 5.6.

Since the fish responds to a constant fin angle by a linear "ramp" in depth, the plant acts like an integrator. Specifically, the transfer function of the plant is approximately $\frac{1.8335}{s}$, where 1.8335 is the slope of the plot in Figure 5.6 (in units of radian^{-1}) times the nominal velocity (4 knots, or 2.06 m/s). In other words, 1.8335 represents the dive rate in m/s per radian of pectoral fin angle, while swimming forward at 4 knots.

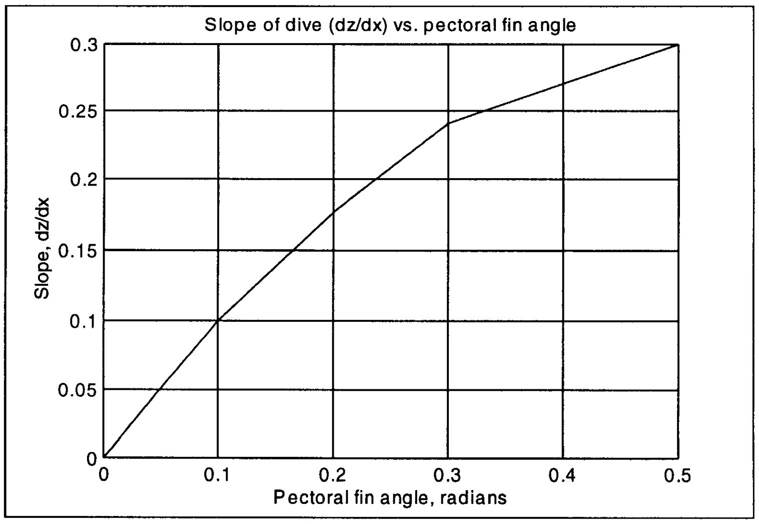


Figure 5.6: Dive slope versus fin angle, at nominal velocity (4 knots)

It is known that the fin will become ineffective (or stall) with an angle of more than ±15°, or ±0.2618 radians. Thus the controller should not allow the pectoral fin to exceed this range. Proportional control was chosen, since depth control did not need to be extremely precise, nor was high-frequency disturbance a major concern (ruling out integral and derivative control, respectively). Furthermore, since the angular range is so limited, fancy control schemes are really unnecessary. This means that K becomes a scalar proportional gain. The value of K is chosen as follows: with $\beta=1$ (which can be arranged in software), the closed loop transfer function is $\frac{1.8335K}{s+1.8335K}$. This is a first-order response which has a bandwidth of $\omega=1.8335K$. K was thus chosen by recognizing that with a 5 Hz digital control rate of the Smart Motors, we can approximately control a continuous time bandwidth of 2.5 Hz (according to Nyquist). Setting $1.8335K = 2.5*2\pi$, we arrive at K=8.57.

A number of shortcuts were taken in this analysis. To begin with, the derivation of K was made by recognizing at the last minute that we are using a CT analysis for a DT system. Furthermore, the "control" of the Smart Motors is difficult to characterize, because the motors are so "smart." Sending a command to the motors does not generate an instantaneous movement (as the model assumes), but rather it is an input to another feedback loop (the Smart Motor PID loop) which can be approximated as a source of delay. Several other small delays in the loop related to data transmission and processing were ignored as well. In the end, the assumptions were justified by the fact that the overall depth control is expected to be a very low-frequency control task, and that the value of K was not extremely critical – especially considering that the pectoral fins are constrained to such a small angular range anyhow.

The final feedback design is shown in Figure 5.7.

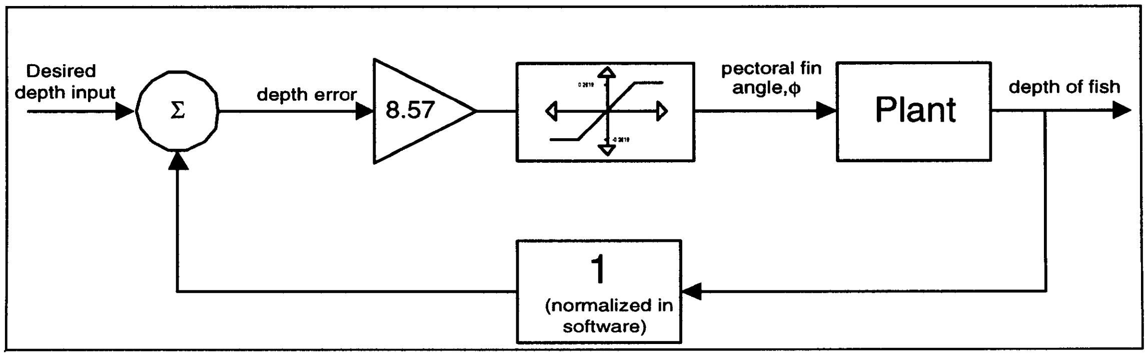


Figure 5.7: Final feedback controller

Finally, remember that the TT8 is sampling depth information at 50 Hz, while Smart Motor control occurs at 5 Hz. The simplest solution for this control loop was to average every 10 samples of depth information to generate the lower frequency 5 Hz data.

To summarize the control loop: the TT8 samples depth information from the pressure transducer at 50 Hz. This data is sent to the 486 and processed by the QNX program. The program which averages each group of 10 samples, determines the depth error (difference between actual and desired) and computes a new pectoral fin angle. The pectoral fin angle is sent back to the TT8, which sends serial commands to the Smart Motors.

5.3 Summary

By no means does this chapter even begin to describe all of the "Feedback and Control" issues surrounding the VCUUV; it is merely one tip of a many-tipped iceberg. The motivation for the SIMULINK model development is to gain a better understanding to the VCUUV's behavior in the dive-plane only. Depth control is a sufficiently important guidance task that it deserves a separate analysis. At present, it is probably the easiest dimension of the fish's motion to put into simulation, since the full three-dimensional dynamic model of the fish is expected to be quite complex.

Unfortunately, the depth control loop has not yet had a chance to be tested in water. On land, the loop was demonstrated by applying a variable source of pressure to the sensor to simulate changing depths in water. As the pressure was varied, the pectoral fins responded by setting the appropriate angle of attack; however, for most cases (where the depth error is greater than a foot or so), the fins sat at either the positive or negative limit of 15°.

Future work should include measuring the dive-slope vs. pectoral fin and forward speed relationship, and comparing it to the SIMULINK results. The SIMULINK model should be revised and updated as new data arrives from the field and from others who are working on hydrodynamic modeling of the VCUUV.

Chapter 6 Conclusions

"So long, and thanks for all the fish." -- *Douglas Adams*

6.1 Summary of Thesis

This thesis has studied guidance and navigation of the VCUUV from several different theoretical and practical perspectives. The work discussed so far includes the following:

- Research on general guidance and navigation theory, and potential application of these ideas to VCUUV navigation.
- Design of electronics to gather data from navigation sensors. In some respects, the electronics design constrains the navigational capabilities of the vehicle.
- Performance characterization of the VCUUV inertial system (which collectively includes IMU, electronics, and software).
- Modeling and simulation applied to studying the behavior of the VCUUV in the dive-plane.

6.2 Current State of the VCUUV

The fish has recently gone swimming in a large tank at the University of New Hampshire. It has demonstrated the capability to swim in a straight line, execute simple turns, and swim in a circle. The primary goal to "swim in a fish-like manner" seems within grasp.

The next focus is on guidance and navigation of the VCUUV, and so the work described in this thesis will soon be put to the test. While the navigation electronics have been successfully demonstrated outside of the vehicle, it has experienced a few problems operating inside of the fish, attributed largely to EMI induced failures. Current work is directed at seamlessly integrating the navigation electronics into the electrically noisy environment of the VCUUV.

As mentioned several times in the preceding chapters, a great deal of work remains in mathematically modeling the fish and its hydrodynamic behavior. Since the dynamics are expected to be very complex, most of this modeling will be based on observations of how the actual vehicle swims – in other words, an exercise in *system identification.* Inertial measurements from the IMU might actually be very useful in creating such a model.

6.3 Lessons from the Existing Work

Anyone who has designed something will admit to wishing he had done something differently at the beginning; this is because design itself is a powerful learning experience. A lot of insight was gained while designing the navigation electronics. Hopefully these lessons can be applied if and when a second-generation VCUUV is built. Some of the ideas and observations collected during the course of this project are presented below.

The first potential improvement to the current navigation electronics might be to design and build a special precision low-noise analog-to-digital (A-to-D) board specifically to sample IMU signals, with at least 16 bits of resolution per channel. Electrical noise is a very troublesome problem throughout the VCUUV; primarily noise generated by the power converters. This EMI couples to virtually every signal, even shielded lines, and has caused several mysterious failures. The best form of noise-protection for the IMU signals is to convert the analog signals to digital data *before* the information is sent over cables that can pick up EMI. Thus this special sampling board would have to be mounted *very* close to the IMU, right near the connector, and be shielded itself – perhaps in a metal cage that was connected to the shielded IMU casing. Furthermore, the board would need analog scaling and filtering circuits as well as the A-to-D converter, so components would have be of very small size, probably surface mount; building and mounting such a board would present a number of electrical and mechanical challenges but certainly is not an impossible task.

Such a board would seemingly "waste" seven input channels on the Tattletale's A-to-D converter, but could potentially improve noise problems by orders of magnitude. A suitable A-to-D converter would have 8 channels of bipolar inputs, preferably differential (since the IMU provides differential outputs), that can transmit the digitized data via a serial link. The serial data can then go to a TPU channel on the Tattletale, which can be programmed to transmit and read serial information. At the moment, all TPU inputs are being used (for compass and Smart Motor homing), but detection of the Smart Motor homing sensors can easily be switched over to the analog A-to-D inputs which have now been freed up.

A 16-bit A-to-D converter is desirable because signal quantization was clearly evident with the TT8's 12-bit conversions. While quantization never caused any serious problems in the existing system, a few more bits of resolution would improve signal quality significantly.

A related revision would be to change the gains of the precision IMU scaling circuits (and of course, software constants to reflect the hardware change). As it is, the circuit scales the accelerometer and gyro signals so that their full-scale ranges correspond to the full-scale input of the A-to-D converter, ±2.5V. However, it seems that only a fraction of the IMU's full-scale ranges will ever be used; for instance, when will the accelerometers ever see a ±5g input? From experimental observations, it appears that the full-scale ranges of the IMU are a bit extreme, and that we could increase resolution by looking only at a smaller dynamic range: for instance, ±3.5g of the acceleration channels. More importantly than resolution, we would gain a much better signal-to-noise ratio (SNR), since the relative scaling of the signals would be larger compared to the fixed amplitude of the noise. This change would involve modifying the resistor ratios in the scaling circuits, changing some software constants, and re-measuring the temperature/bias parameters (as explained in Chapter 4). In the next-generation vehicle, this is certainly worth consideration.

6.4 Recommendations for the Future Generation VCUUV

If the VCUUV proves successful, it may lead to the design and development of a new generation of robotic fish which are more maneuverable and efficient than propeller-driven subs of comparable size. In this case, many improvements can be made on the existing VCUUV design, particularly in the areas of guidance and navigation.

A main finding of this thesis is that the present inertial system can only track position for a very short time without external correction. Since the VCUUV is a prototype vehicle and the first of its kind, an inertial system of this quality may be useful for short-term tasks such as attitude stabilization, velocity measurement, and direct acceleration and rotation rate measurements during various maneuvers. It is quite clear, however, that in a future generation of this vehicle estimation of position will be a *much* more critical task, and so a better navigational system must be designed. Below we discuss some possibilities for improvement.

In general, strapdown IMUs of similar size, weight and cost to the Systron Donner have similar drift performance, since they are constructed using similar solid-state inertial sensor technologies. However, a number of 6-DOF IMUs are available with digital outputs, a feature which has significant advantages over the analog outputs of the Systron Donner. In the digital IMUs, the gyro and accelerometer signals are already digitzed within an electrically shielded case, preventing external EMI from corrupting the measurements. This also eliminates the hassle of designing analog scaling, filtering and sampling circuitry. The inertial measurements are often delivered from the IMU over a standard interface such as RS-232, and in some cases they have already been digitally compensated for bias and temperature-dependent errors within the IMU. More advanced IMUs are designed to plug directly into a standard GPS receiver for periodic corrections, and can even implement a Kalman filtering algorithm! Such features eliminate a great deal of design effort and external hardware, but do not come without cost. A very good IMU made by Boeing, with most of these features, costs nearly double the price of a Systron Donner[32]. In many applications, however, the price is well worth the savings in time, effort and cost of external components associated with a custom system.

We still have not solved the drift problem on the VCUUV, since any IMU of this size and cost range will not be suitable for standalone navigation. *There is no getting around the fact that a future generation VCUUV will need better external navigation aids.* The present vehicle has only the pressure sensor and compass, which can only bound drift of the position estimate in depth and azimuth respectively. Access to an external correction source such as GPS is vital if such a system is to navigate correctly for any useful period of time. A possible solution is to have a small GPS antenna which floats on the surface, attached to the fish through a cable. This, however, has the disadvantage of a long wire continuously dangling between the fish

[32] Other digitally-interfaced IMUs are made by Crossbow and Litton.

and the surface of the water. Another possibility is to set up a local system of sonar beacons in the vicinity where the fish will need to navigate. By calculating the time delays between synchronized "pings" transmitted by each beacon, the fish can triangulate its position. An on-board sonar system[33] might also be used by the fish to "map" its environment, and to measure its own movement by the relative movement of the environment around it. The bottom line is that if low-quality strapdown inertial systems are used, better methods of externally correcting the inertial system are necessary.

Sonar in general is a very useful tool for underwater systems, and should definitely find its place in a future VCUUV. The present vehicle has no means of detecting or avoiding obstacles, which should be a major concern for any autonomous vehicle! While sonar environment-mapping would require very sophisticated hardware, obstacle detection is a much more feasible and essential application of sonar.

The last recommendation, not necessarily related to guidance and navigation, also involves the use of sonar – for communication. Sonar modems can be purchased which allow digital information to be modulated as sound and transmitted through the water. Next-generation VCUUVs would greatly benefit from the ability to communicate with a ground station (or other robotic fish!) through a wire-free channel.

In conclusion, future robotic fish will need a sophisticated combination of instruments in order to truly demonstrate intelligent action and autonomy in their underwater environment. The navigational system described in this thesis may be sufficient for the prototype VCUUV, but is most instructive in exposing the limitations of standalone inertial navigation using a small low-cost inertial system. The future breed of robotic fish will certainly need to take much greater advantage of external navigational aids.

[33] Perhaps using a phased-array scanning sonar system, such as those manufactured by the Interphase Corpotaion.

References

1. Anderson, Jamie M., "Unmanned Undersea Vehicle Model Analysis for Low Speed Controller Development," Technical Memorandum, Naval Undersea Warfare Center Division, Newport, Rhode Island, 30 Sept. 1992.

2. Anderson, Jamie M., Kerrebrock, Peter, Triantafyllou, Michael S., "Concept Design of a Flexible-Hull Unmanned Undersea Vehicle," Draper Lab Document ID# CSDL-P-3555, April 1997.

3. Cho, Jamie L., "Electronic Subsystems of a Free-Swimming Robotic Fish," Master of Science Thesis, Massachusetts Institute of Technology, December 1997.

4. Eduardo, Nebot, Salah, Sukkarieh, and Hugh, Durrant-Whyte, "Inertial Navigation aided with GPS information," Dept. of Mechanical and Mechatronic Engineering, University of Sydney, Australia, http://mecharea.me.su.oz.au/people/nebot/papers/ m2v_ins_gps/m2v_ins_gps.htm.

5. Horowitz, Paul, and Hill, Winfield, "The Art of Electronics," Second Edition, Cambridge University Press, 1989.

6. Karu, Zoher Z., "Signals and Systems Made Ridiculously Simple," ZiZi Press, Cambridge, MA, 1995.

7. Lawrence, Anthony A., "Modern Inertial Technology: Navigation, Guidance and Control," Springer-Verlag, New York, 1993.

8. Mackenzie, Donald, "Inventing Accuracy: A Historical Sociology of Nuclear Missilie Guidance", The MIT Press, Cambridge, MA, 1990.

9. Oppenheim, A.V., and Schafer, Ronald W., "Discrete-Time Signal Processing," Prentice-Hall, Englewood Cliffs, NJ, 1989.

10. Savant, C.J., Howard, R.C., Solloway, C.B., and Savant, C.A., "Principles of Inertial Navigation," McGraw-Hill, New York, 1961.

11. Siebert, William M., "Circuits, Signals, and Systems," McGraw-Hill, New York, 1986.

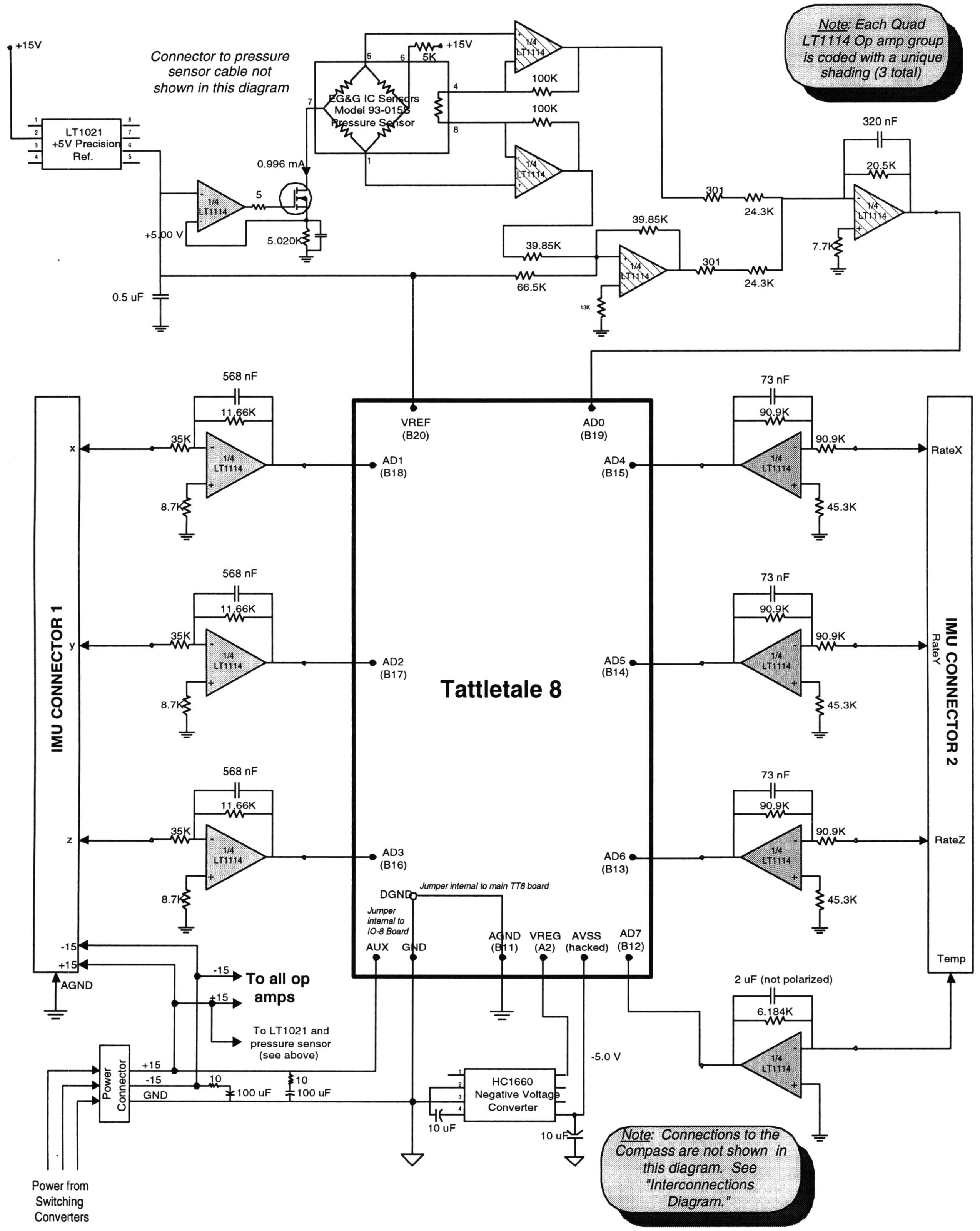
12. Strang, Gilbert, "Introduction to Linear Algebra," Wellesley-Cambridge Press, Wellesley, MA, 1993.

13. Titterton, D.H., and Weston, J.L., "Strapdown Inertial Navigation Technology," Peter Peregrinus Ltd., London, 1997.

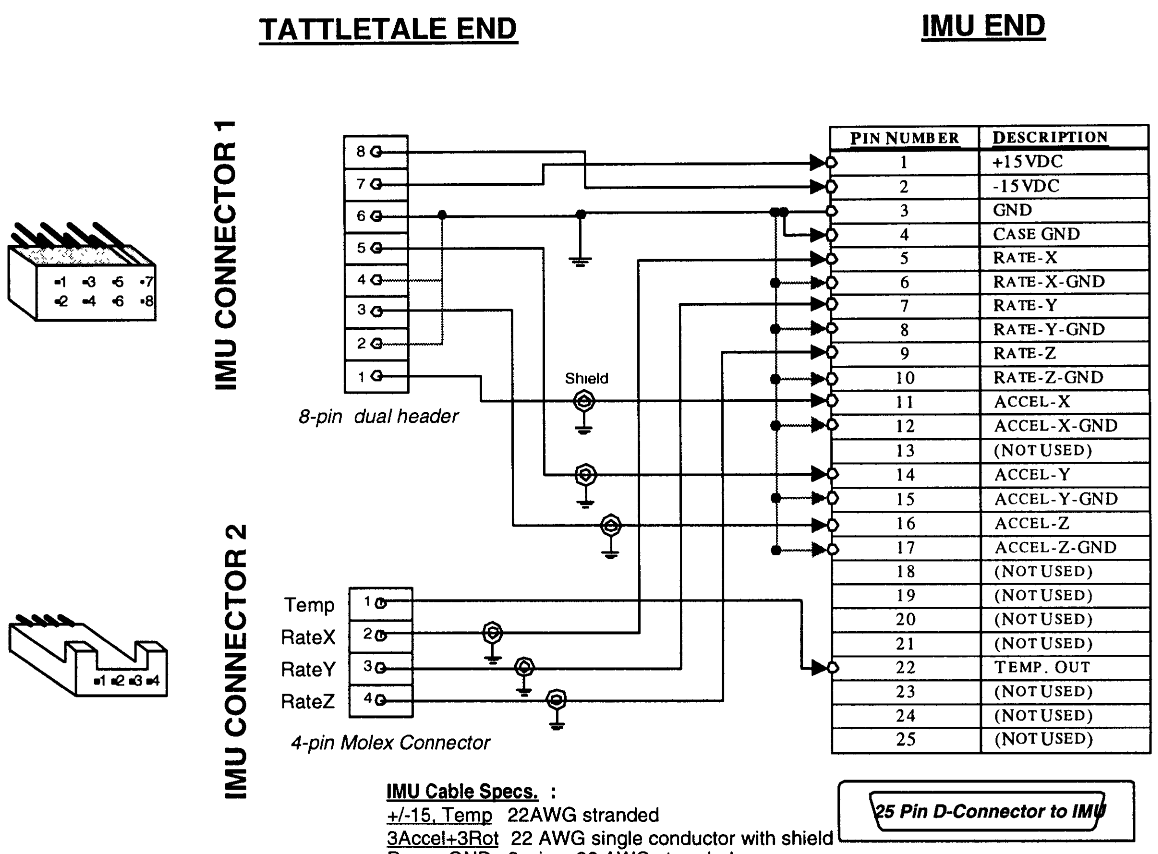
14. Trott, Christian A., "Electronics Design for an Autonomous Helicopter," Bachelor and Master of Science Thesis, Massachusetts Institute of Technology, June 1997.

Appendix A: Circuit Schematics

A.1 TT8 Surrounding circuitry


Note: Each Quad LT1114 Op amp group is coded with a unique shading (3 total)
Connector to pressure sensor cable not shown in this diagram
+15V
LT1021 +5V Precision Ref.
1/4 LT1114
EG&G IC Sensors Model 93-0155 Pressure Sensor
5K
100K
100K
320 nF
20.5K
301
24.3K
39.85K
39.85K
66.5K
13K
7.7K
0.996 mA
5
5.020K
+5.00 V
0.5 uF
568 nF
11.66K
35K
8.7K
x
y
z
IMU CONNECTOR 1
VREF (B20)
AD0 (B19)
AD1 (B18)
AD2 (B17)
AD3 (B16)
AD4 (B15)
AD5 (B14)
AD6 (B13)
Tattletale 8
73 nF
90.9K
45.3K
RateX
RateY
RateZ
Temp
IMU CONNECTOR 2
DGND
Jumper internal to main TT8 board
Jumper internal to IO-8 Board
AUX
GND
AGND (B11)
VREG (A2)
AVSS (hacked)
AD7 (B12)
2 uF (not polarized)
6.184K
-15
+15
AGND
To all op amps
To LT1021 and pressure sensor (see above)
Power Connector
10
100 uF
HC1660 Negative Voltage Converter
-5.0 V
10 uF
Power from Switching Converters
Note: Connections to the Compass are not shown in this diagram. See "Interconnections Diagram."

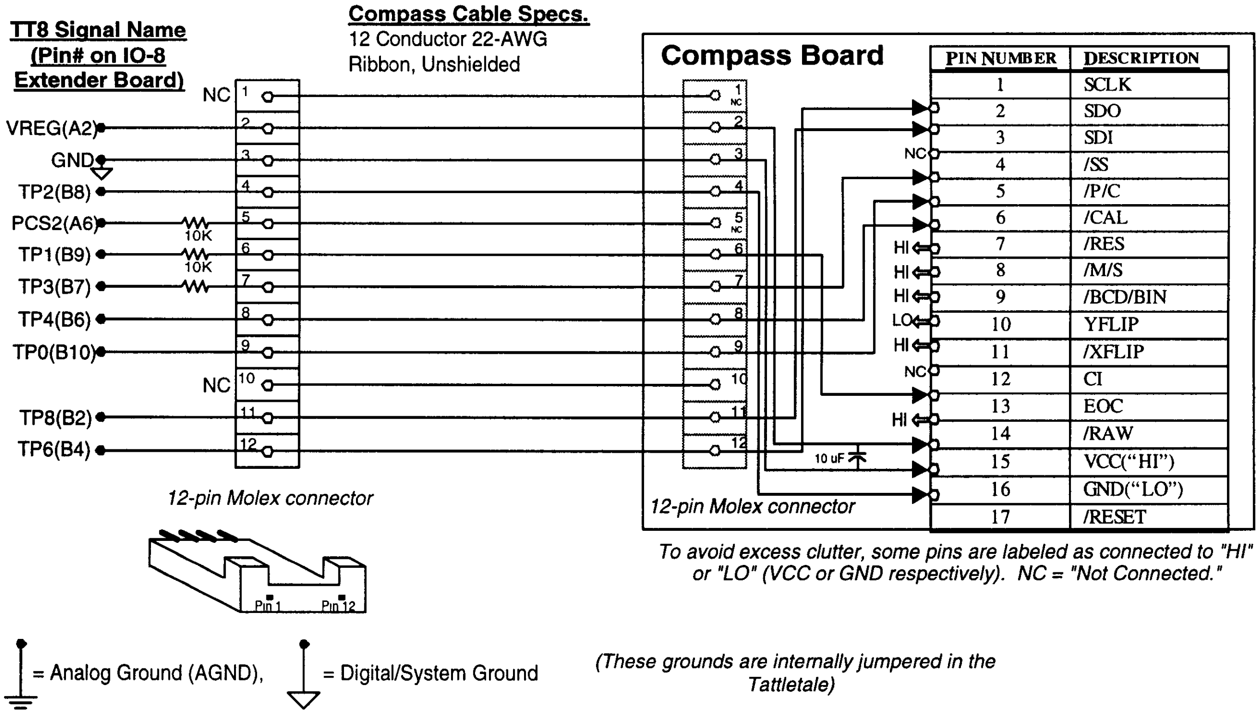
A.2 Interconnections Diagram



PIN NUMBER	DESCRIPTION
1	+15VDC
2	-15VDC
3	GND
4	CASE GND
5	RATE-X
6	RATE-X-GND
7	RATE-Y
8	RATE-Y-GND
9	RATE-Z
10	RATE-Z-GND
11	ACCEL-X
12	ACCEL-X-GND
13	(NOT USED)
14	ACCEL-Y
15	ACCEL-Y-GND
16	ACCEL-Z
17	ACCEL-Z-GND
18	(NOT USED)
19	(NOT USED)
20	(NOT USED)
21	(NOT USED)
22	TEMP. OUT
23	(NOT USED)
24	(NOT USED)
25	(NOT USED)

IMU Cable Specs. :
+/-15, Temp 22AWG stranded
3Accel+3Rot 22 AWG single conductor with shield
Power GND 3 wires 22 AWG stranded

Compass Cable Specs.
12 Conductor 22-AWG
Ribbon, Unshielded



PIN NUMBER	DESCRIPTION
1	SCLK
2	SDO
3	SDI
4	/SS
5	/P/C
6	/CAL
7	/RES
8	/M/S
9	/BCD/BIN
10	YFLIP
11	/XFLIP
12	CI
13	EOC
14	/RAW
15	VCC("HI")
16	GND("LO")
17	/RESET

To avoid excess clutter, some pins are labeled as connected to "HI" or "LO" (VCC or GND respectively). NC = "Not Connected."

(These grounds are internally jumpered in the Tattletale)

Appendix B: Tattletale 8 Documentation and Troubleshooting

B.1 General Construction

The TT8 is provided along with a choice of two prototyping boards that are necessary to interface with the TT8. One is the large PR-8 board, and the other is the smaller (credit-card sized) IO-8 board. For space considerations we have chosen to use the IO-8 board, which is compactly "piggy-backed" on top of the TT8 microprocessor board. The IO-8 allows access to a number of signal lines from the TT8, such as the A-to-D inputs and some of the TPU digital I/O pins. As we will explain later, the choice of the IO-8 (which does not allow access to *all* the TT8's signals, unlike the PR-8) necessitates a "hack" be done on the TT8 board.

The IO-8 consists of header-pins to piggy-back it to the TT8 as well as a blank prototyping space. This is where we begin to build our interface/signal-conditioning circuitry. As it turned out, this space was not large enough to contain all the circuitry. The solution was to add a *third* piggy-backed board, which was created from a sheet of standard prototyping board. This third board attaches on top of the IO-8 through two rows of header that pass all of the TT8 signal lines to the third board.

The TT8 also talks with a host computer through an RS-232 port located on the IO-8 board. Furthermore, the IO-8 has a number of screw-terminal blocks which can be used to connect external wires to the assembly (we will use them for connectors).

If all of this circuitry is someday put on a custom PCB, many things may change. It will be impossible to use the IO-8 prototyping space. The IO-8 and its internal connections (which are vital to TT8 operation) should be re-created on the custom board, as well as the navigation circuitry. The custom board can then plug into the TT8 just as the original IO-8 did, through header pins. For maximum versatility, the custom board might instead expand upon the more flexible PR-8 prototyping board, which attaches to the TT8 through a "squishybus" interface (see the Tattletale manual for details).

B.2 Power Requirements

The TT8, circuitry and associated sensors all run off of ±15 Volts and ground, which are supplied from an external source. This supply should be accurate to at least 2% and relatively noise free.

Power enters the board through a three-pronged Molex connector on the third piggy-backed board. From here it is routed to the IO-8 through the intermediate power connector, and the IO-8 powers the TT8, IMU, etc. At the input to the TT8 assembly, the ±15V lines are filtered with 10uF filter caps.

B.3 Schematic Documentation

This section dissects the schematics provided in Appendix A.

B.3.1 Tattletale

The large box in the center of the schematic represents the Tattletale. The circuitry outside this box is the added signal conditioning circuitry, which connects to various input and output pins on the IO-8 board. Remember that all of this circuitry is divided between the IO-8 prototyping space and the space on third piggy-backed board. Note that all connections to the IO-8 in the schematic are labeled by two identifiers: a signal name (i.e. VREF or AD0), and a pin number (which refers to the pin number on the IO-8 board).

Note the important exception to this: the AVSS signal at the bottom right of the rectangle in the schematic drawing has no pin number. This is because on the IO-8 board, we do not normally have access to the AVSS signal (we would normally need to use the much larger PR-8 board). In order to save space, we are using the IO-8 board and have "hacked" a connection directly to the TT8. This goes along with the detailed description of the "A-to-D hacks" explained below; please refer to that discussion.

B.3.2 General A-to-D Operation

The Tattletale 8 uses the MAX186 12-bit, 8-channel serially-accessed A-to-D converter to sample analog data. The IMU data we will be reading has two important properties: 1) it is bipolar, and 2) it has precise scale factors associated with it. We must make sure we are sensitive to these properties.

Two chips in the schematic are necessary for precision and bipolar A-to-D operation. One is the LT1021-CCN8-5 in the upper left corner of the schematic. This takes a +15V input and provides a very accurate +5.00V reference (0.05% accuracy). This precision reference allows the A-to-D to make accurate conversions. The second is the ICL7660s negative voltage converter in the bottom center of the schematic, made by Harris. This takes the regulated 5 Volt supply (generated on the TT8, called "VREG"), and flips its polarity. This provides the proper negative supply (-5V) for bipolar operation.

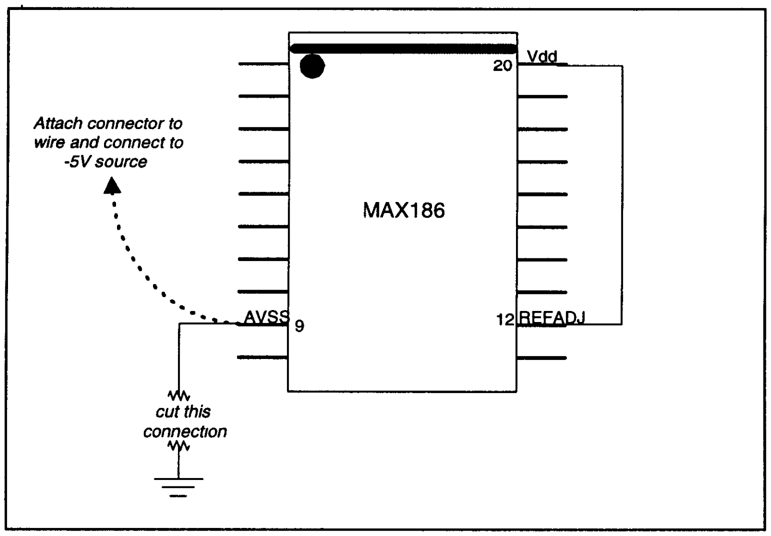
With this external circuitry the A-to-D is configured to have its bipolar input range between +2.5V and - 2.5. These limits correspond to digital values of +2048 and -2048, respectively (a 12-bit range).

B.3.2.1 A-to-D Modifications

To make use of the precision and bipolar features, several board-level changes are necessary. To begin with, the Tattletale 8 manual indicates that in order to get bipolar readings *a trace must be cut* on the main microcontroller board (see details in Section 7.1 of the manual). Cutting this trace separates the pin called "AVSS" from ground. AVSS is pin 9 of the MAX186 (a surface-mount chip), and is the negative power input to the A-to-D converter. Thus after cutting the trace we must supply a -5V source to AVSS (which is generated by our ICL7660s voltage inverter chip). Unfortunately, with the IO-8 board we do not have access to the AVSS signal! (Though the PR-8 prototyping board can access AVSS, we do not want to use it because of its much greater size.) So, we must "hack" a connection to pin 9 of the MAX186 chip. This is done by *carefully* soldering a wire to the pin (it is tiny!), checking the connection under a magnifier, and then creating a strain-relief to keep the delicate connection from breaking (either epoxy or some other method is necessary). The other end of the wire is then connected to a single pin connection which can plug into the ICL7660s's -5V output.

The second hack is necessary in order to supply a precise reference to the A-to-D converter. The converter comes with an internal precision reference of ± 2.048 volts, but this turns out to be an inconvenient value. Thus we provide a +5.00 precision reference with our LT1021 chip, to the signal VREF (pin B20 on the IO-8) board. In order to use this reference, however, we must tie the pin on the MAX186 called "REFADJ" to Vdd (5 Volts). And once again, we don't have access to REFADJ on the IO-8 board. So this is done through another *careful* hack directly on the Tattletale board: pin 20 of the MAX186 chip (Vdd) must be connected to pin 12 of the MAX186 (REFADJ). This hack is similar to the AVSS one, but this time there is no wire hanging off of the board; the hack is internal to the TT8.

These hacks are illustrated below:



Note that these hacks are being done on a surface mount board with many fine traces and pins, and mistakes may cause the Tattletale to be ruined.

For clarification of these hacks, study the Tattletale 8 manual: Chapter 7, and the MAX186 datasheet in the appendix.

B.3.3 IMU Circuit

The interface to the IMU is through two connectors on either side of the schematic in A.1. IMU Connector 1 brings ±15V power and ground to the IMU, and brings the x, y, and z linear acceleration signals back to the IO-8. IMU Connector 2 brings the three angular-rotation signals and the IMU temperature signal back to the IO-8.

These are the seven raw signals from the IMU. The three linear acceleration signals on the left pass through op-amp circuits that scale them by a factor of -1/3, and filter them with a single pole at about 25 Hz. The three rate signals on the right go through op-amp circuits that scale them by a factor of -1 and also filter them with one pole at 25 Hz. Finally, the temperature signal, which is a current-signal, is buffered, filtered (1 pole at 12 Hz), and converted to a voltage signal by the op-amp at the bottom right.

> *A note about scale factors*: While 1% resistors are used, scale factors in the op-amp circuits will necessarily have some error. Thus after constructing these circuits, it is wise to precisely measure the gains of each op-amp circuit, and to make calculations in software so as to correct for gain errors. This technique was used whenever possible.

The seven op-amps used are from two LT1114 quads. (The spare op-amp is used in the Pressure Circuit; see below.) After this signal conditioning, the seven IMU signals are passed to seven A-to-D inputs on the IO-8 board (AD1-AD7).

There is an additional issue with grounding. The IMU has a number of ground returns: the power ground, case ground, and then six individual signal grounds. The reason for separate signal grounds is that there is current that returns through the power ground line. The IMU draws about 250 mA from the +15V supply, but only sinks 190 mA into the -15V line; thus the difference of 60 mA returns to the power supply through the ground line. This causes an "IxR" offset in the ground line, especially if it is a long wire. In order to prevent this offset from adding to signal bias, each signal is provided with its own ground. These grounds are all connected to power ground *internally at the IMU*; however, the extra grounds are provided so that the circuitry can make a differential measurement, so that this 60 mA will not return partially in each of the 6 lines and cause an IxR offset. Due to complexity of making so many differential measurements, we have not taken advantage of the separate grounds. Given the shortness of the cable (about 8 inches), and the fact that a well-conducting ground wire is being used, the IxR bias should be insignificant; thus we have simply considered the voltage drop across the power ground wire to be zero. The case is also grounded to suppress noise around the IMU.

B.3.4 Pressure Sensor Circuit

The pressure sensor used is a Model 93-015S (sealed-gauge) sensor made by EG&G IC Sensors. It measures pressure *with respect to one atmosphere*. There is usually a bias, since the internal one-atmosphere reference has some error to it; this can be accounted for in software. The maximum pressure reading guaranteed accurate is 15 p.s.i., but it can safely go to about 45 p.s.i.

The circuit occupies the upper part of the schematic. The sensor itself is in the gray box, and it consists of a pressure-sensitive Wheatstone bridge. Our circuit uses five op-amps: four from an LT1114 quad, and one ¼LT1114 left-over from the IMU circuit.

The leftmost op-amp is used as a current-sink. The pressure sensor requires a that a precise 0.996 mA current be drawn through it. Thus, one end of the sensor (pin 6) is connected to +15V (through a 5K resistor which is necessary to establish the correct output level), and the other end (pin 7) is connected to the drain of a MOSFET. Through feedback, the op-amp drives the MOSFET to precisely draw $\frac{+5.00V}{5020\Omega} = 0.996$ mA.

The outputs of the sensor are differential, and come from pins 1 and 5. These are passed to a pair of op-amps which calibrate the values with respect to an internal reference resistor (pins 4 and 8). The last two op-amps do level-shifting, scaling and filtering (again one pole at 25 Hz). The final output is passed to an A-to-D input (AD0), and has the following characteristics:

♦ Zero-pressure (w.r.t. 1 atmosphere) output, assuming zero bias:	+2.5V
♦ 45 p.s.i. pressure output, assuming linearity and no bias:	-2.5V

This corresponds to the full range (+2048 to -2048) of our 12-bit A-to-D converter. 15 p.s.i., which corresponds to 33.75 feet of water, falls at +0.8333V (digital value of +683). Thus our A-to-D has a full range of 0-45 p.s.i. in this circuit – if the pressure sensor were linear above 15 p.s.i. (which it is not guaranteed; but it is guaranteed not to break up to 45 p.s.i.) Note the assumption of zero-bias; in reality, the zero-pressure output was a bit less that +2.5V, since the one-atmosphere reference in the sensor was a bit low (this also varies day-to-day). This can be corrected for in software. (However, that if we were working at higher altitudes, with lower atmospheric pressures, this circuit might have to be redesigned since the A-to-D might saturate at +2048 at the surface...)

B.3.5 Electronic Compass

The compass used is the V2XG, manufactured by Vector. It is powered from VREG (+5) from the Tattletale. The compass requires no signal-conditioning; it is a purely digital instrument. It is up to software to actuate digital signals in the appropriate manner to recover data from the compass. See the Vector manual for timing details.

B.4 Programming the TT8

Programs for the TT8 are written in C and compiled into Tattletale object files (.RHX and .AHX) using the Aztec C compiler. A "makefile" is used to compile and link the files correctly.

Programs can be loaded into the TT8 in either .RHX or .AHX format. .RHX files are loaded in Tattletale RAM, and are lost from memory as soon as the TT8 is powered down. .AHX are intended to be finalized programs that boot up automatically every powerup, and are stored in flash memory. Flash memory can be reprogrammed many times over. Programs are loaded into the TT8 from a host computer running the *Crosscut* terminal program. The TT8 must be plugged into the computer's serial port, and 'crosscut' must be run. Then when the TT8 is powered *and the gray switch on the piggy-back board is up*, the <TOM> monitor interface[34] should appear in the Crosscut window. *When using Crosscut, make sure the switch is up before powering up the TT8 (you will see a red LED light up on the TT8)!* This bypasses any stored program on powerup and jumps to the <TOM> monitor program stored in TT8 ROM. Once the <TOM> monitor is running, you can use Crosscut to download .RHX and .AHX files to the TT8. Once a .AHX file is loaded, power down the TT8 and *push the gray switch down to allow the flash program to automatically run on the next TT8 powerup.*

The source code for the latest version of the Tattletale software is located in <c:/tt8/mohan/final/pitr.c>. To compile and link this file, go to the directory and type 'make.'

B.5 Parts List

Part Number	Part Description	Quantity	Unit Cost
Tattletale Model 8	Data Logger/Controller	1	$500.00
LT1114ACN	Quad Precision Op Amp, DIP	3	$12.75
LT1021 CCN8-5	5V Precision Reference, DIP	1	$6.38
ICL7660s (Harris)	Voltage Inverter, DIP	1	$1.95
BS170 (National)	MOSFET, TO-92	1	$1.00

+ Handful of Resistors (1%), Capacitors, LEDs.

Sensor	Type	Function	Vendor	Approx. Cost
IMU	MotionPak	Inertial Guidance	Systron Donner	$13,800
Pressure Sensor	Model 93-015S	Depth	EG&G IC Sensors	$100
Compass	V2XG	Heading	Vector	$100

[34] The <TOM> monitor is a mini-operating system that the Tattletale runs when it is powered up with the gray switch in the up position. The monitor is necessary to load new programs into the TT8's RAM or flash memory..

Appendix C: Code Listing

C.1 Tattletale Code

The following code is written in C, and is burned into the flash memory on the TT8 so as to automatically boot on powerup. The TT8 code includes three separate C programs and two custom headers:

- C-files: pitr.c (main file which #includes the others), motors.c, compass.c
- Headers: bias.h, coeffs.h

PITR.C

```
/*** A2D sampling using the Periodic Interrupt Timer (PIT)
       and the QSPI ***/ /*** Mohan Gurunathan Last Edit: 5/1/98
                                                    5 March 1998
                                                    11 March 1998
                                                    20 March 1998   ***/
/*** 100 Hz Version ***/

/***** What is going on in this program:

       EXTAL is 40 kHz.  The PIT is clocked at this rate.
       PTP is the periodic timer prescaler value, which either
       multiplies the PITR period by 512 (if MODCLK=0) or by
       1 (if MODCLK=1).
       Interrupt frequency is calculated as follows:

       PIT Frequency = XTAL Freq. /(PITM * PTPvalue * 4)

       In this program we will make PTP 1 by setting MODCLK to 1.
       We will write the value 100 into the PITM(Per.Interr.Timer
       Modulus.)  this gives a freq of 100 Hz (=40K /4 /100).
       The interrupt priority and vectoring are set in the
       the values PIRQL and PIV in the PICR control register.
       We will leave PIRQL at 1, where it is initialized.
       We will leave the exception vector number
       as previously defined (vector 80), PIT_INT_VECTOR.


Note: PIT_INT_REQ_LEVEL is previously defined as 1.


Data filtering also occurs as follows:

*** data filtering program uses 3rd-order Chebyshev Type II;

*** difference equation is implemented as
  follows:

  Ay*y[n] + By*y[n-1] + Cy*y[n-2] + Dy*y[n-3] =
  Ax*x[n] + Bx*x[n-1] + Cx*x[n-2] + Dx*x[n-3]

  where Ai,Bi,Ci, and Di are the difference equation
  coefficients for i=x and i=y, and Ay
  is assumed to be 1 [after normalization].

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

#include <stdio.h>
#include <stdlib.h>
#include <tt8libnew.h>
#include <qsm332.h>
```

```
#include <sim332.h>
#include <tpu332.h>
#include <math.h>


/*** Coefficients for data filtering: 3rd order ChebyII filter***/
#include "coeffs.h"
#include "bias.h"      /*** Temperature-bias compensation for the IMU ***/

/*****   Definitions to make code more readable ******/
#define CXND (current_x_node->data)
#define CYND (current_y_node->data)
#define CXNRD (current_x_node->right->data)
#define CYNRD (current_y_node->right->data)
#define CXNLD (current_x_node->left->data)
#define CYNLD (current_y_node->left->data)
#define CXNRRD (current_x_node->right->right->data)
#define CYNRRD (current_y_node->right->right->data)

/*****  Collision flags  ***************/       /** these should appear in QNX code
also... ***/
#define XCOLL   1
#define YCOLL   2
#define ZCOLL   3
#define NO_COLLISION 4


/*****  QNX based definitions (for QNX sending packets)*****/   /** these should appear in
QNX code also... ***/
#define QNX_PACKET_SIZE 7                       /** in bytes (uchars)... ***/
#define MOTOR_DO_NOTHING 30000    /*** long (2-byte) word: appropriate string is "u0" ***/
#define QNX_PACKET_HEADER 'q'
#define RESET_COMPASS 'h'
#define COMPASS_CAL_PULSE 'a'
#define START_NAV 'r'
#define STOP_NAV 'w'
#define TT8_PACKET_HEADER1 'p'
/***#define TT8_PACKET_HEADER2 ((uchar)15)***/
#define TT8_PING 'n'
#define TT8_PACKET_SIZE 42   /** includes CHECKSUM, but not header ***/
#define TT8_HOME_YOUR_FINS 'g'
/**#define TT8_SET_COLLISION_FLAG 'c'***/  /*** to be implemented later **/


/***Compass definitions ****/
/*** TPU Pin Assignments ****/
#define V2XG_RESET 2
#define V2XG_EOC 1
#define V2XG_SS 3
#define V2XG_CAL 4
#define V2XG_PC 0
#define V2XG_SDO  8
#define V2XG_SCLK 6
#define HIGH 1
#define LO 0
#define NO_COMPASS_DATA 999

/***Motor Definitions ******/
#define MOTOR_CHAN_Tx 13
#define MOTOR_CHAN_Rx 14
#define MOTOR_BAUD 38400
#define QSIZE 1024
#define MAX_COMMAND_LENGTH 20
#define RIGHT_FIN_HOME 7         /*** TPU CHANNEL ***/
#define LEFT_FIN_HOME 5        /*** TPU CHANNEL ***/
#define BAUD_RATE 111000        /*** for talking to QNX: actual is
                                             115.2Kbaud but the closest TT8 can
                                              come is 111.1KBaud
                                             (3.5% error is tolerable***/


/**** QNX packet structure: one type of packet; 7 bytes (THIS STRUCTURE IS NOT USED)
*****/
struct QNX_packet {
  uchar QNX_header;
  uchar reset_or_cal_compass;
  uchar nav_start_stop_or_ping;
  uchar left_motor_MSByte;
  uchar left_motor_LSByte;
  uchar right_motor_MSByte;
  uchar right_motor_LSByte;
};
typedef struct QNX_packet QNX_packet;



/*****   Packet structure for TT8 Data: one type of packet; 40 bytes (without header and
checksum) ******/
struct packet {
  ulong samplenum;
  double xval;
  double yval;
  double zval;
  double xrot;
  double yrot;
  double zrot;
  double temp;
  double depth;
  long compass;
  short flag1;
  short collflag;
};
typedef struct packet packet;


typedef struct loop {
  packet  data;
  struct loop *right;
  struct loop *left;
};
typedef struct loop NODE;
typedef NODE *NODEPTR;




/*** Function prototypes: ***/
int setup_system(void);         /*****  Set CPU freq and Baud rate ****/
void inthandler(void);   /*****  100 Hz periodic interrupts for sampling ***/
double twos_comp(short);       /*****  Twos complement function   ****/
int spi_setup(void);           /*****  Set-up the QSPI Serial interface (A2Ds) ****/
void create_loops(void);       /*****  Set-up IIR filtering structure ****/
void initloops (NODEPTR, NODEPTR);      /*****  Erase all IIR filter values  ****/
void compute_filter (void);             /*****  Update the output of digital filter ****/
void transmit_packet_to_PC104(void);    /*****  Sends the current_y_node packet to QNX
****/
uchar* listen_to_PC104(void);           /*****  Listen for commands from QNX on serial
line ****/
void TT8_comm_init(void);               /*****  Setup TT8 to communicate with PC104 ****/
void execute_PC104_command(void);       /*****  get PC104 command; decode it; execute it
****/
void move_motors(long, long);           /*****  set motor position after getting PC104
command ****/

/**Compass***/
/*** Motors ****/


/*** Global var.s ****/
/*register*/ NODEPTR basenodex, basenodey, current_x_node, current_y_node;
/*register*/ ulong calls=1;
ulong oldcalls = 0;
ushort toggle = 0;
ushort oldclk=0;
ushort compass_flag = 0;
long compass_reading=NO_COMPASS_DATA;
long collision_threshold = 700;

uchar motor_init[] =
"Z\nZ\nZ
\n~0SADDR1\n~1ECHO\n~1SLEEP\n~0SADDR2\n~2ECHO\n~1WAKE\n~0E=9000\n~0AMPS=1000\n~0KL=22\nF\n
~0BAUD38400\n";

uchar QNX_receive[QNX_PACKET_SIZE];


/*** put an error flag here... ***/

#include "motors.c"
#include "compass.c"
```

```
/***********************************************************/
/***************************  MAIN  **********************/
/***********************************************************/

int main(void)
{
  ushort n = 1;
  char conv[8] = {0,4,1,5,2,6,3,7};
  static ExcCFrame framebuf;  /** frame pointer for the handler routine
                               to find its way back to the function
                               ***/


  InitTT8(NO_WATCHDOG, TT8_TPU);
  setup_system();             /*** CPU rate, baud rates, etc. ***/
  create_loops();       /*** IIR filtering structure creation **/
  initloops(current_x_node, current_y_node);
  TT8_comm_init();

  /*************** create interrupt handler routine: *****************/
  InstallHandler(inthandler, PIT_INT_VECTOR, &framebuf);

  DelayMilliSecs (100);    /*** Wait for motors to be powered up ***/

  /*****  init motors and home ****/
  if (!init_motors())
      {      /** printf ("Motor init failed\n");**/
              /**exit(1);        /*** Error initializing the motors ***/
              while(1) {PutStr ("Motor init failed... \n");}
      }

if (motors_are_there())  /** checks if motors are connected ***/
 {
 /**home_fin(2);
    home_fin(1);
    home_fin(2);
    home_fin(1);
    offset_home(); **/    /*** removed automatic home: 5/1/98 ***/
 }


  /*** init compass (reset it; it still needs to be calibrated) ***/

  init_compass();    /*** we should see the green LED go on now:  ***/

  /**other possible interrupts? -- emergency shutdown/memory cleanup;
    restart/reset/recalibrate;  PECTORAL FIN CONTROLLERS ***/


  /***** set up the QSPI queue:  ****/
  spi_setup();



/******** Idle Housekeeping Loop Begins here: *******/


  while (1)

   {
       compass_process();
       execute_PC104_command();
   }

 return(0);       /*** never should get to this point... ***/

}                        /**** end of main *****/




void inthandler(void)      /*** interrupt handler ****/
{
. short temp;
```

```
  ulong t;
  short i;


  current_x_node = current_x_node->left;        /*** Advance IIR filter pointers **/
  current_y_node = current_y_node->left;


  /*** get A2D READINGS:  ***/

  _SPCR1->SPE = 1;  /* start the read */
  while (_SPSR->SPIF != 1);


  /***** PERFORM THE SAMPLING USING QSPI ****/
  CXND.temp = twos_comp((short)(((SPIRCV[0]) <<1) >> 4));
  CXND.xval = twos_comp((short) (((SPIRCV[2]) <<1) >> 4));
  CXND.yval = twos_comp((short) (((SPIRCV[3]) <<1) >> 4));
  CXND.zval = twos_comp((short) (((SPIRCV[4]) <<1) >> 4));
  CXND.xrot = twos_comp((short) (((SPIRCV[5]) <<1) >> 4));
  CXND.yrot = twos_comp((short) (((SPIRCV[6]) <<1) >> 4));
  CXND.zrot = twos_comp((short) (((SPIRCV[7]) <<1) >> 4));
  CXND.depth = twos_comp((short)(((SPIRCV[1]) <<1) >>4));


compute_filter();            /**** DIGITALLY FILTER THE DATA ****/


/************ COLLISION DETECTION  ***************/
/**** We do this here because after filtering, a collision
value may be swallowed by the filter *****/

  if ((CXND.xval > collision_threshold) || (CXND.xval < (-collision_threshold)))
      {
       CYND.collflag = XCOLL;
       CYNLD.collflag = XCOLL;
      }
  else if ((CXND.yval > collision_threshold) || (CXND.yval < (-collision_threshold)))
      {
       CYND.collflag = YCOLL;
       CYNLD.collflag = YCOLL;
      }
  else if ((CXND.zval > collision_threshold+100) || (CXND.zval < -
(collision_threshold+100)))
      {
       CYND.collflag = ZCOLL;
       CYNLD.collflag = ZCOLL;

       }

/*** ONCE A COLLISION IS DETECTED, THE COLLISION FLAG IS
PERMANENTLY SET TO A VALUE LESS THAN 4, UNTIL THE TT8 IS REBOOTED;
4 means "NO_COLLISION".  WE MIGHT WANT TO CHANGE THIS LATER SO
THE COLLISION FLAG RESETS TO "NO_COLLISION" AFTER BROADCASTING
"XCOLL," "YCOLL," OR "ZCOLL" FOR SOME TIME. ***/


  if (toggle == 1)      /*** Every other call to handler, Xmit data ***/

   {
   /***************** TRANSMIT DATA: *********************/
      CYND.samplenum = calls;   /** only even numbers sent, thus we can
                                    send an unambiguous packet header ***/

          /***** now transmit the packet CYND: *****/


    transmit_packet_to_PC104();

    toggle = 0;
  }  /** end of "if toggle" ***/

  else toggle = 1;

  calls++;                      /**  Update # of calls to handler **/

  _SPSR->SPIF = 0;         /*** clear finished flag  ***/

}
```

```
double twos_comp(short num)
{
  double ans;
  if (num >= 2048)
    {ans = num - 4096;}           /** two's complement correction **/
  else ans = num;
  return (ans);
}




int setup_system(void)
{
  /** SET cpu FREQUENCY and serial baud rate:***/
  SimSetFSys (32000000);
  SerSetBaud (BAUD_RATE, 32000000);          /*** 115200... ***/
  DelayMilliSecs (100);  /**** Just in case system needs to settle ***/

}



int spi_setup(void)
{
  ushort index;

  Max186PowerUp();
  Max186Setup(pdExtClk, extComp);

  *SPCR0  =     8                /*8 = 1 MHz 4 = 2 MHz */        // baud rate (2 is max)
     |   (0 << 10)                       // bits per transfer (0 = 16)
     |   M_MSTR  &       SET             // 1 = Master, 0 = Slave
     |   M_WOMQ  &       CLR             // 1 = open drain, 0 = cmos
     |   M_CPOL  &       CLR             // 1 = inactive SCK high
     |   M_CPHA  &       CLR             // 1 = chg lding edge, capt folng
     ;

  *SPCR2 = 0;                    /** clear register; NEWQP is at $0***/
  *SPCR2 = 7 << 8                        /***  8 channels on the queue ***/
    | M_SPIFIE & CLR    /** disable finished interrupts **/
    | M_WREN & CLR               /** no wrap around  ***/
    | M_WRTO & CLR;              /** no wrapping to ***/


  *SPCR3 = M_LOOPQ & CLR | M_HMIE & CLR | M_HALT & CLR;

  *SPCR1 = 0 | 16 << 8 | M_SPE & CLR; /** check the delays here later **/

  _SPSR->SPIF = 0;

  /** set up the QSPI command and transmit RAM: ***/

  SPIXMT[0] = 0x87;
  SPIXMT[1] = 0xC7;
  SPIXMT[2] = 0x97;
  SPIXMT[3] = 0xD7;
  SPIXMT[4] = 0xA7;
  SPIXMT[5] = 0xE7;
  SPIXMT[6] = 0xB7;
  SPIXMT[7] = 0xF7;

  /*** later add pressure sensor(done) and compass ***/

  for (index = 0; index < 8; index++)
    {
      SPICMD[index] = M_CONT & CLR /** SET***/      |
        M_BITSE & SET   |
        M_DT & CLR      |
        M_DSCK & SET;
    }

/****  Set up Periodic Interrupt Timer (PITR)  *****/
  _PICR->PIV = PIT_INT_VECTOR; /*** defaulted to 80 ***/
  _PICR->PIRQL = PIT_INT_REQ_LEVEL;  /*** defaulted to 1, i think ***/
}
```

```
void initloops (NODEPTR x_node, NODEPTR y_node)
{
  packet clean ={0,0,0,0,0,0,0,0,0,0, 123, NO_COLLISION};
  /**** 123 is just a dummy value for flag1... ***/
  x_node->data = clean;
  x_node->right->data = clean;
  x_node->left->data = clean;
  x_node->right->right->data = clean;
  y_node->data = clean;
  y_node->right->data = clean;
  y_node->left->data = clean;
  y_node->right->right->data = clean;
}



void create_loops(void)
{
       basenodex = malloc(sizeof(NODE));
       current_x_node = basenodex;
       basenodex->right = malloc(sizeof(NODE));
       basenodex->left = malloc(sizeof(NODE));
       basenodex->right->right = malloc(sizeof(NODE));
       basenodex->left->left = basenodex->right->right;
       basenodex->left->right = basenodex;
       basenodex->right->left = basenodex;
       basenodex->right->right->right = basenodex->left;
       basenodex->right->right->left = basenodex->right;


       basenodey = malloc(sizeof(NODE));
       current_y_node = basenodey;
       basenodey->right = malloc(sizeof(NODE));
       basenodey->left = malloc(sizeof(NODE));
       basenodey->right->right = malloc(sizeof(NODE));
       basenodey->left->left = basenodey->right->right;
       basenodey->left->right = basenodey;
       basenodey->right->left = basenodey;
       basenodey->right->right->right = basenodey->left;
       basenodey->right->right->left = basenodey->right;

}


void compute_filter(void)
{

/*** This computation is done all using global variables... ***/

  CYND.temp = Ax*(CXND.temp) + Bx*(CXNRD.temp) + Cx*(CXNRRD.temp) + Dx*(CXNLD.temp) -
By*(CYNRD.temp) - Cy*(CYNRRD.temp) - Dy*(CYNLD.temp);
  CYND.xval = Ax*(CXND.xval) + Bx*(CXNRD.xval) + Cx*(CXNRRD.xval) + Dx*(CXNLD.xval) -
By*(CYNRD.xval) - Cy*(CYNRRD.xval) - Dy*(CYNLD.xval);
  CYND.yval = Ax*(CXND.yval) + Bx*(CXNRD.yval) + Cx*(CXNRRD.yval) + Dx*(CXNLD.yval) -
By*(CYNRD.yval) - Cy*(CYNRRD.yval) - Dy*(CYNLD.yval);
  CYND.zval = Ax*(CXND.zval) + Bx*(CXNRD.zval) + Cx*(CXNRRD.zval) + Dx*(CXNLD.zval) -
By*(CYNRD.zval) - Cy*(CYNRRD.zval) - Dy*(CYNLD.zval);
  CYND.xrot = Ax*(CXND.xrot) + Bx*(CXNRD.xrot) + Cx*(CXNRRD.xrot) + Dx*(CXNLD.xrot) -
By*(CYNRD.xrot) - Cy*(CYNRRD.xrot) - Dy*(CYNLD.xrot);
  CYND.yrot = Ax*(CXND.yrot) + Bx*(CXNRD.yrot) + Cx*(CXNRRD.yrot) + Dx*(CXNLD.yrot) -
By*(CYNRD.yrot) - Cy*(CYNRRD.yrot) - Dy*(CYNLD.yrot);
  CYND.zrot = Ax*(CXND.zrot) + Bx*(CXNRD.zrot) + Cx*(CXNRRD.zrot) + Dx*(CXNLD.zrot) -
By*(CYNRD.zrot) - Cy*(CYNRRD.zrot) - Dy*(CYNLD.zrot);
  CYND.depth = Ax*(CXND.depth) + Bx*(CXNRD.depth) + Cx*(CXNRRD.depth) + Dx*(CXNLD.depth) -
By*(CYNRD.depth) - Cy*(CYNRRD.depth) - Dy*(CYNLD.depth);

    /**** see if compass data is ready... ******/
if (compass_flag == 12)
       {
        CYND.compass = compass_reading;     /*** Put compass data in packet ***/
        compass_flag = 0;                   /*** Go to start of compass_process ***/
       }
```

```
	else CYND.compass = NO_COMPASS_DATA;	/** No compass data ready; insert dummy
value **/

/**** Remember: collflag value is set AND LOCKED in handler if collision occurs ****/
/**** flag1 as of yet has no meaning: reserved for TT8 status message ***/
/**** Compass data arrives about one-tenth the rate packet transmission:
(compass data at about 5 Hz); and is sent about once every 10 transmissions.
Otherwise the dummy value NO_COMPASS_DATA is sent. ***/
/**** remember to subtract biases here, or on QNX side.... ****/

}



/******* Communications functions for the TT8<--->PC104 ******/


void transmit_packet_to_PC104(void)
{
	uchar* char_ptr;
	uchar send_string[TT8_PACKET_SIZE];
	short i;
	float dummy_float;
	double tt = CYND.temp;
	long cksum = 0;   /*** 4 byte value, we only use first two bytes ***/

/*** Data in packet is 40 bytes, + 1 byte sent as a header byte,
+ 2 bytes sent as a checksum, which is a long value that is the
sum of the IMU data fields (which are the most likely to be corrupted) ***/
/*** Note QNX stores floats and longs in "Big Indian" format, so we
must byte-reverse the TT8 values ****/
/*** for the Digitally filtered values, we convert the doubles
to floats so they can be sent in 4 bytes ***/
/** WE ALSO TEMPERATURE COMPENSATE THE VALUES HERE, though
	some fine-tuning of gyro compensation is also done on the
	QNX side  ***/

	char_ptr = (uchar*)(&(CYND.samplenum));
	send_string[0] = char_ptr[3];
	send_string[1] = char_ptr[2];
	send_string[2] = char_ptr[1];
	send_string[3] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.xval - Mx*tt - BBx);
	char_ptr = (uchar*)(&dummy_float);
	send_string[4] = char_ptr[3];
	send_string[5] = char_ptr[2];
	send_string[6] = char_ptr[1];
	send_string[7] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.yval - My*tt - BBy);
	/***char_ptr = (uchar*)(&dummy_float);***/
	send_string[8] = char_ptr[3];
	send_string[9] = char_ptr[2];
	send_string[10] = char_ptr[1];
	send_string[11] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.zval - Mz*tt - BBz);
	/***char_ptr = (uchar*)(&dummy_float);***/
	send_string[12] = char_ptr[3];
	send_string[13] = char_ptr[2];
	send_string[14] = char_ptr[1];
	send_string[15] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.xrot - Mxr*tt - BBxr);
	/***char_ptr = (uchar*)(&dummy_float);***/
	send_string[16] = char_ptr[3];
	send_string[17] = char_ptr[2];
	send_string[18] = char_ptr[1];
	send_string[19] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.yrot - Myr*tt - BByr);
	/***char_ptr = (uchar*)(&dummy_float);***/
	send_string[20] = char_ptr[3];
	send_string[21] = char_ptr[2];
	send_string[22] = char_ptr[1];
	send_string[23] = char_ptr[0];
	cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
	dummy_float = (float)(CYND.zrot - Mzr*tt - BBzr);
```

```
        /***char_ptr = (uchar*)(&dummy_float);***/
        send_string[24] = char_ptr[3];
        send_string[25] = char_ptr[2];
        send_string[26] = char_ptr[1];
        send_string[27] = char_ptr[0];
        cksum += (char_ptr[3]+char_ptr[2]+char_ptr[1]+char_ptr[0]);
        dummy_float = (float)(CYND.temp);
        /***char_ptr = (uchar*)(&dummy_float);***/
        send_string[28] = char_ptr[3];
        send_string[29] = char_ptr[2];
        send_string[30] = char_ptr[1];
        send_string[31] = char_ptr[0];
        dummy_float = (float)(CYND.depth);
        /***char_ptr = (uchar*)(&dummy_float);***/
        send_string[32] = char_ptr[3];
        send_string[33] = char_ptr[2];
        send_string[34] = char_ptr[1];
        send_string[35] = char_ptr[0];
        /**** ONLY 2 bytes for compass: ****/
        send_string[36] = CYND.compass>>8;
        send_string[37] = CYND.compass%256;
        /**** One-byte shorts ****/
        send_string[38] = CYND.flag1;
        send_string[39] = CYND.collflag;
        /*** CHECKSUM: 2 bytes ***/
        char_ptr = (uchar*)(&cksum);
        send_string[40] = char_ptr[3];
        send_string[41] = char_ptr[2];

        SerPutByte (TT8_PACKET_HEADER1);
        for (i=0; i<TT8_PACKET_SIZE; i++)
        {
                SerPutByte (send_string[i]);
                /*** eventually use PutStr; this is slow ***/
        }


}



uchar* listen_to_PC104(void)
{
        short i;

        if (!SerByteAvail()) {return(NULL);}
           /***else printf ("Getting data...\n");***/
        QNX_receive[0] = SerGetByte();
        if (QNX_receive[0] != QNX_PACKET_HEADER)  /*** look for header**/
                {  return (NULL); }              /** if header is not there**/
        for (i=1; i<QNX_PACKET_SIZE; i++)
        {
          QNX_receive[i] = SerGetByte();          /** add timeout **/
        }
        return (QNX_receive);
}


/***** QNX packet format:  7 bytes
        Byte 0:         HEADER Byte 'q'
        Byte 1:         Reset Compass or Pulse Cal low on Compass
        Byte 2:         Start NAV or Stop NAV, home fins, or TT8 ping;
        Byte 3-4:       Left Motor Most Sig BYTE, LS BYTE
        Byte 5-6:       Right Motor MSByte, LSByte

WE SHOULD ADD MORE OPTIONS TO FIRST FEW BYTES, AND
WE SHOULD LEAVE BYTES 3-6 FOR GENERAL DATA USAGE SO PC104 CAN
SET VARIOUS VALUES ON THE TT8:  For instance,
1)  Collision detection thresholds:  Right now they are set for
1.75 G's in any direction (A2D value of +/- 700)

MUCH more functionality can be made out of the first three bytes,
if desired.  The last 4 MUST be reserved for motor use.

****/
```

```
void TT8_comm_init(void)
{
       ptr buffer;

       buffer = malloc(2048*sizeof(uchar));

        SerSetInBuf(buffer, (long)(sizeof(buffer)));
        DelayMilliSecs(10);
        SerInFlush();
        /**** set baud rate *****/
        /**** What else?  *****/
}


void execute_PC104_command(void)
{
        uchar* dat;
        long left_motor, right_motor;

        dat = listen_to_PC104();
        if (dat == NULL) return();     /*** NO PC104 command waiting ****/
           /**** remember: static arrays are free'd at the end of function! ****/
        if (dat[1] == RESET_COMPASS)
        {
           /**printf ("Compass being reset... \n");**/
           init_compass();
        }

        if (dat[1] == COMPASS_CAL_PULSE)
        {
           compass_cal_pulse();
           /**printf ("Cal Pulse sent to compass...\n");**/
        }

        if (dat[2] == TT8_PING)
        {
           PutStr("TT8 is alive and bitchin'\n");    /*** add varicus status messages ***/
           if (!motors_are_there())
           {
               PutStr("Please connect the Smart motors and Reset the TT8\n");
           }
           else
           PutStr("TT8 is hungry for some action\n");

           /*** read motor positions... ***/
        }

        if (dat[2] == TT8_HOME_YOUR_FINS)
        {   /*** Only home the fins if we are not sampling... ***/
              if (*PITR != 0x064)
              {
                if (motors_are_there())
                {
                  home_fin(2);
                  home_fin(1);
                  offset_home();
                }
              }
        }

        if (dat[2] == START_NAV)
        {
           *PITR = 0x064;     /*MODCLK=0, 0x13 = 19 in decimal,
                       thus freq. is 40000/(512*19) = 1.02Hz
                       0x014 = 500Hz,  0x113 = 1.02Hz
                        0x064 = 100Hz *//**** start interrupt handler... ****/
        }
        if (dat[2] == STOP_NAV)
        {
           *PITR = 0;       /*** stop sampling ****/
           exit(1);         /*** COMPLETELY restart the Tattletale:
                               init motors, reset compass, etc... ***/
        }


        left_motor = ((int)(dat[3]))*256 + ((int)(dat[4]));
        right_motor = ((int)(dat[5]))*256 + (int)(dat[6]);

        move_motors(left_motor, right_motor);
}


/**** can't do this until motors are initialized! *****/

void move_motors (long left, long right)
{
        char f[10], g[10];

        if (left != MOTOR_DO_NOTHING)
        {
           sprintf (f, "~2P=%ld G", left);
           motor_send (f);
        }
        if (right != MOTOR_DO_NOTHING)
        {
           sprintf (g, "~1P=%ld G", right);
           motor_send (g);
        }
}
```

MOTORS.C

```
/***** Motor function prototypes *****/


void motor_string_convert (uchar*);  /*****  obsolete function?  ******/
void SM_send_string(uchar*);         /*****  Send uchar string to motors  *****/
long SM_send_and_read(uchar*);       /*****  Send uchar string; listen for response **/
void home_fin (int);                 /*****  Bring the fin to switch-homing position **/
void put_str (uchar*, const char*);  /***    converts char strings to uchar strings ***/
void motor_send(const char*);        /*****  HIGH LEVEL function for sending "strings"**/
long motor_send_and_read(const char*);/*****  HIGH LEVEL function for send_and_read  **/
void finish_moving(void);            /*****  Wait for fin to finish moving ****/
void offset_home (void);             /*****  Move both fins to horiz. AFTER home_fin **/
void Wait5ms(void);                  /*****  Delay between sending characters ****/
int init_motors(void);               /*****  Setup Smartmotor communications ****/
int motors_are_there(void);          /*****  Checks if motors are connected *****/
/*** warning motors_are_there function checks if the proportional constant
in the motors, KP, is = 500 (default value).  be aware of this if you ever decide to
change
the prop. gain in the smart motors. ***/


/*************** MOTOR FUNCTIONS ***********************/



int init_motors(void)
{
       /*** This fct. sets up the TPU Serial Ports to talk to the motors ***/
  ptr Txbuf;
  ptr Rxbuf;


  if ((Txbuf = malloc(QSIZE+TSER_MIN_MEM))==0)
    {
       printf ("\nError allocating Tx memory.\n");
       return(0);
    }
  if ((Rxbuf = malloc(QSIZE+TSER_MIN_MEM))==0)
    {
       printf ("\nError allocating Rx memory.\n");
       return(0);
    }
  if (TSerOpen(MOTOR_CHAN_Tx, HighPrior, 1, Txbuf, QSIZE, 9600, 'N',8,1) != tsOK)
/***????***/
       {
              printf ("\nError Opening the TPU Serial port on channel %d\n\n",
MOTOR_CHAN_Tx);
              return(0);
       }
  if (TSerOpen(MOTOR_CHAN_Rx, HighPrior, 0, Rxbuf, QSIZE, MOTOR_BAUD, 'N',8,1) != tsOK)
       {      printf ("\nError Opening the TPU Serial port on channel %d\n\n",
MOTOR_CHAN_Rx);
              return(0);
       }
  TSerInFlush(MOTOR_CHAN_Tx);
  TSerInFlush(MOTOR_CHAN_Rx);
  SM_send_string(motor_init);
  DelayMilliSecs(500);
  TSerInFlush(MOTOR_CHAN_Rx);
  TSerClose(MOTOR_CHAN_Tx);
  if (TSerOpen(MOTOR_CHAN_Tx, HighPrior, 1, Txbuf, QSIZE, MOTOR_BAUD, 'N',8,1) != tsOK)
       {
              printf ("\nError Opening the TPU Serial port on channel %d\n\n",
MOTOR_CHAN_Tx);
              return(0);
       }
  TSerInFlush(MOTOR_CHAN_Tx);
  TSerInFlush(MOTOR_CHAN_Rx);
  return(1);                     /*** Motor Initialization OK  ****/

}


void SM_send_string(uchar* string)
```

```
{
  int i;

  for (i=0; string[i] != '\0'; i++)
               {
                  if (string[i] == 126)
                  { i++;
                    string[i] = (string[i]) + 80;}
                   TSerPutByte (MOTOR_CHAN_Tx, (int)(string[i]));
                   Wait5ms();
               }
 TSerPutByte (MOTOR_CHAN_Tx, 13);      /** Send carriage return **/
 TSerInFlush (MOTOR_CHAN_Rx);       /*** FLUSHES ECHOED CHARACTERS ***/

}


long SM_send_and_read(uchar* string)
{
/**** This function sends a string to Motors and reads the returning
       VALUE sent from motor.  Only ONE command that requires a
       response should be sent, addressed to a SINGLE motor.  ****/

 int i, j;
 uchar echo[80]="";
 uchar receive[80]="";
 char receive2[80]="";
 long num;

/******  SEND   **********/

 for (i=0; string[i] != '\0'; i++)
               {
                  if (string[i] == 126)
                  { i++;
                    string[i] = (string[i]) + 80;}
                  TSerPutByte (MOTOR_CHAN_Tx, (int)(string[i]));
                  Wait5ms();   /*** NEED 4 OR 5 ms HERE FOR MOTORS ***/
               }
 TSerPutByte (MOTOR_CHAN_Tx, 13);         /** Send carriage return **/

 /*** transmitted characters will be echoed; total i+1 transmitted ***/


/********* RECEIVE ***********/
 for (j=0; j < (i+2); j++)
 {
     if (TSerByteAvail(MOTOR_CHAN_Rx))
         {

           echo[j] = (uchar)(TSerGetByte(MOTOR_CHAN_Rx));
         }
 }
  echo[j] = '\0';
  j=0;
  Wait5ms();
  Wait5ms();            /** probably is not necessary **/
  while (TSerByteAvail(MOTOR_CHAN_Rx))
     {
            receive[j] = (uchar)(TSerGetByte(MOTOR_CHAN_Rx));
            j++;
     }
  receive[j] = '\0';
  j=0;
  while (receive2[j]=(char)receive[j]) ++j;  /** copy string..? **/
  num = strtol(receive2, NULL, 10);    /*** convert received string to long ***/

 TSerInFlush(MOTOR_CHAN_Rx);      /** flush buffer now that we've read everything in it
**/
 return (num);

}


/**** The proper calling sequence for the homing functions is: ****
home_fin(1);
home_fin(2);
offset_home();
```

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


void home_fin (int fin)
{
   short TPU_fin_chan;
   char f[10];
   uchar command[40];
   long a, hit1, hit2, hit3, mid, offset=0;

   /*** Turn up the integrator limit in motors: ****/
   motor_send ("~0 KL=700 F ");

   /**** Choose fin to home: *****/

   if (fin==1)
       {
         TPU_fin_chan = RIGHT_FIN_HOME;
         motor_send ("~1 MP V=0 G\n");
       }
     else
       {
         TPU_fin_chan = LEFT_FIN_HOME;
         motor_send ("~2 MP V=0 G\n");
       }


   /******* Initial crude homing:  ******************/

   motor_send ("O0 D=200000 V=920000 A=750\n");

   a = TPUGetPin(TPU_fin_chan);         /***** See if home switch has been tripped ***/

   if (a == 1) hit1 = motor_send_and_read("RP");   /*** Switch has been tripped ***/
   if (a == 0)                            /*** a=0: Switch has NOT been tripped ***/
       {
          motor_send("G");              /** start moving "G" ***/
          while (a == 0)                /** while switch has NOT been tripped... ***/
            {
              a = TPUGetPin (TPU_fin_chan); /** keep looking at switch ***/
            }

           /***** a=1: switch has tripped; read position ********/
           hit1 = motor_send_and_read("RP");          /** hit1 = switch tripped position
***/
           motor_send ("X");                          /*** stop moving ****/
           sprintf (f, "P=%ld G", hit1);              /** create string to hit1 position
**/
           motor_send  (f);                           /*** move to hit1 position  ****/
           finish_moving();                           /*** wait to arrive there   ****/
       }


    /****  Precise homing begins here:  ****/

   motor_send ("V=430000 A=550 D=-50000");       /*** move slowly ***/
   motor_send ("G");
   a = TPUGetPin (TPU_fin_chan);
   while (a == 1)                               /*** still withing homing hysteresis ***/
        {
           a = TPUGetPin(TPU_fin_chan);         /*** find left edge of switch limit  ***/
        }

      /*** hit edge of hysteresis limit ***/
      hit2 = motor_send_and_read ("RP");        /**  hit2 = left edge position  **/
      motor_send ("X");                         /**  STOP   ***/
      motor_send ("V=894784 A=750\n");         /***  FAST move again ***/
      sprintf (f, "P=%ld G", hit1);             /**  create string to hit1 position **/
      motor_send (f);                           /*** go back to  hit1 position  ****/
      finish_moving();                          /*** wait to arrive there   ****/
      DelayMilliSecs(1);

      motor_send ("V=430000 A=550 D=50000");     /*** move slowly ***/
   motor_send ("G");
   a = TPUGetPin (TPU_fin_chan);
   while (a == 1)                               /*** still within homing hysteresis ***/
      {
        a = TPUGetPin(TPU_fin_chan);           /**** find other edge of switch ***/
      }
   /**** hit edge of hysteresis limit ***/
   hit3 = motor_send_and_read ("RP");            /**  hit3 = "RP"   **/
   motor_send ("X");                             /**  STOP  ***/
   mid = (hit2 + hit3)/2;                        /**  find average of switch positions ***/

   motor_send ("V=894784 A=950\n");              /*** FAST move again ***/

   sprintf (f, "P=%ld G", mid);                  /**  create string to mid position **/
   motor_send (f);                               /*** move to mid position  ****/
   finish_moving();                              /*** wait to arrive there   ****/
/**   printf ("\n\n\n\t\tMOTOR IS HOMED!!!\n\n"); ***/


   motor_send ("OO");            /** set home as new origin ***/

/*** use this function on EACH fin, THEN invoke offset_home **/
}



void offset_home (void)
{
      /*** This function sends the fins from their switch
         homed position to a horizontal position. ***/
       /*** Smart Motor 1 is on the Fish's left fin ***/
          motor_send ("~1P=-25750 ");
          motor_send ("~2P=-47975 ");
          motor_send ("~0V=1000000 A=950 ");
          motor_send ("~0G ");
          motor_send ("~1 "); finish_moving();
          motor_send ("~2 "); finish_moving();
          DelayMilliSecs (250);          /*** let motor settle ***/
          motor_send ("~0 KL=22 F A=880");
          motor_send ("~0 O0 P=0 ");
}


/***** I DO NOT KNOW WHY THIS FUNCTION (put_str) IS NECESSARY, BUT SMART
MOTOR CONTROL LOCKS UP WITHOUT IT.  IT IS BASICALLY A STRING
TRANSLATION FUNCTION FROM CHARs TO UCHARs.  I'm PRETTY SURE
IT IS NOT A TIMING ISSUE, BECAUSE REPLACING THIS FUNCTION WITH
A SMALL DELAY DOES NOT WORK.  SO, LEAVE IT ALONE! ****/
void put_str (uchar* command, const char* string)
{
   int i=0;

   while (*(string+i) != '\0')
   {    *(command+i) = (uchar)(*(string+i));
        i++;
   }
   *(command+i)='\0';
}


void motor_send(const char* string)
{
        uchar command[40];

        put_str (command, string);
        SM_send_string(command);
}



long motor_send_and_read(const char* string)
{
        uchar command[40];
        long ans;


        put_str (command, string);
        ans = SM_send_and_read (command);
        return (ans);
}



void finish_moving(void)
{
/**** FCT. TO WAIT TILL MOTOR IS FINISHED MOVING (WATCH RBT) ****/
```

```
/*** This function works on whatever fin has been already selected... ***/

while (motor_send_and_read("RBt") != 0);
}


void Wait5ms(void)
{
   auto long i;
   for (i=0; i<5600; i++);
}


int motors_are_there (void)
{
       long val;


       val = motor_send_and_read ("~1RKP");    /*** get the motor to report some value
***/
       if (val != 500)                         /*** the default value ***/
              return (0);                     /*** motors not connected ***/
       return(1);                              /*** motors are there ***/
}
```

COMPASS.C

```
/*** Compass function prototypes *****/

void init_compass(void);          /*****  Reset the V2XG compass   *****/
void compass_process(void);       /*****  Compass DataAcq process *****/
void Wait2ms(void);               /*****  Wait about 2 msecs.      *****//**COMPASS**/
void compass_cal_pulse(void);      /*****  Pulse Cal pin low ****/



/****** COMPASS FUNCTIONS:  ********/


void init_compass(void)
{
  TPUSetPin(V2XG_PC, 1);
  TPUSetPin(V2XG_SS, 1);
  TPUSetPin(V2XG_CAL, 1);
  TPUSetPin(V2XG_RESET, 1);
  TPUSetPin(V2XG_SCLK, 1);

  DelayMilliSecs (300);
  TPUSetPin(V2XG_RESET, 0);     /*** Pulse RESET low ****/
  DelayMilliSecs (300);
  TPUSetPin(V2XG_RESET, 1);
  DelayMilliSecs (300);
}



/*** Here is the multi-tasking compass event loop that is looped over
   and over, while being interrupted by the sampling handler.  ***/

void compass_process(void)  {

int bit[9], i;

          switch (compass_flag) {
               case 0:   oldcalls = calls;
                         compass_flag = 1;
                         break;
               case 1:   if (calls > oldcalls)         /** Wait for one interrupt cycle**/
                            {compass_flag = 2;}
                         break;
               case 2:   TPUSetPin(V2XG_PC, 0);        /** Set PC low **/
                         compass_flag = 3;
                         break;
               case 3:   if (TPUGetPin(V2XG_EOC) == 0) /** Wait for EOC low **/
                            {compass_flag = 4;}
                         break;
               case 4:   if (TPUGetPin(V2XG_EOC) == 1) /** Wait for EOC high **/
                            {  TPUSetPin(V2XG_PC, 1);  /** If EOC==1 raise PC **/
                              compass_flag = 5;
                            }
                          break;
               case 5:   oldcalls = calls;
                         compass_flag = 6;
                         break;
               case 6:   if (calls > oldcalls + 1)       /** Wait for one cycle (10ms) **/
                            {compass_flag = 7;}
                         break;
               case 7:   TPUSetPin(V2XG_SS, 0);        /** Set SS low **/
                         compass_flag = 8;
                         break;
               case 8:   oldcalls = calls;
                         compass_flag = 9;
                         break;
               case 9:   if (calls > oldcalls + 1)         /** Wait for one interrupt
cycle**/
                            {compass_flag = 10;}
                         break;
               case 10:  for (i=0; i<7; ++i)           /** ACQUIRE COMPASS DATA **/
                         {
                              TPUSetPin(V2XG_SCLK, LO);   /** Clock out 7 dummy bits **/
                              TPUSetPin(V2XG_SCLK, 1);
                         }
                         compass_flag = 11;
```

```
                              break;
               case 11:  for (i=0;i<9;++i)              /*** Clock out 9 data bits **/
                         {
                              TPUSetPin(V2XG_SCLK, LO);
                              bit[i] = TPUGetPin(V2XG_SDO);
                              TPUSetPin(V2XG_SCLK, 1);
                         }

                         compass_reading = 256*bit[0] + 128*bit[1] + 64*bit[2] + 32*bit[3]
+ 16*bit[4] + 8*bit[5] + 4*bit[6] + 2*bit[7] + bit[8];
                         TPUSetPin(V2XG_SS, 1);          /** Raise SS high **/
                         compass_flag = 12;              /** Set compass flag to data
ready**/
                         break;
               case 12:  break;

               } /*** end of switch ***/
       return;
}


void Wait2ms(void)
{
   auto long i;
  /** THis function waits about 2 milliseconds of delay at 32 MHz **/

   for (i=0;i<2750;i++);
}


void compass_cal_pulse(void)
{
       TPUSetPin (V2XG_CAL, 0);
       Wait5ms();
       Wait5ms();
       Wait5ms();
       TPUSetPin (V2XG_CAL, 1);
       compass_flag = 0;
}
```

BIAS.H

```
/*** bias.h     header file

This file contains Temperature compensation for each of the channels.
The temperature compensation includes a term which is proportional
to the current temperature (called "M_") and a constant term (called
"BB_"), thus a linear model of bias vs. temperature.  These terms
are derived from experimental measurements and MATLAB linear fits.
****/

/*** Temperature dependent biases: *****/

#define Mx -0.00079476
#define My -0.00213411
#define Mz -0.000319362
#define Mxr 0.005790534
#define Myr 0.0079685214
#define Mzr 0.00237073

/*** Constant offset ***/
#define BBx -1.8814
#define BBy -5.839556
#define BBz -1.87283
#define BBxr 4.885233
#define BByr 9.000
#define BBzr 0.745244
```

COEFFS.H

```
/*** coeffs.h   header file

 The (uncommented) coefficients at the top are the ones
 being used.  These are for:
 Chebyshev type II order 3, cutoff pi/2.5, -35 dB stapband ripple ***/
  /**** Passband gain is 1 ****/

#define Ay 1
#define By -1.86198578362921
#define Cy 1.29421395659688
#define Dy -0.31036146437923
#define Ax 0.03687717667614
#define Bx 0.02405617761808
#define Cx 0.02405617761808
#define Dx 0.03687717667614
```

C.2 QNX Code

This is the QNX program that runs on the 486, which is responsible for communicating with the Tattletale. It uses a menu interface to send various messages to the TT8 or to receive messages and/or data packet information from the TT8.

Some notes:

- The code shown here is only as recent as 5/12/98 and has probably been modified on the system by now.
- This code uses 'switch' commands, which should be replaced with 'if-else' commands for bug-free operation on the VCUUV computer.
- The code here has a slight error in the log-data-and-maintain-depth operations; this should be fixed in the code operating on the fish.

QNX2.C

```
/**************** QNX Comm port test code 2, Mohan G. 4/8/98  *****************/
/****** Last update 5/12/98: fully functional Tattletale communication
    program ******/

#include <stdio.h>
#include <ioctl.h>
#include <fcntl.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <sys/qnxterm.h>
#include <time.h>

/**** still need to implement/fix:

1) Checksum's in both directions
2) HUGE arrays if necessary
3) Multiprocessing structures...
4) Logging AND depth control simultaneously...
5) Manual control while logging

***/

/**** Notes: QNX stores all the 4 byte values in reverse format to the
TT8.  I'm not sure how this affects right shifting, but I think
right-shifting operates in reverse on both platforms, so the end
value after the bit-shift remains the same for a given direction shift
for both QNX and TT8. **/

/*** Note:  when Input serial Buffer is full, QNX does NOT accept new data
into the buffer until data is removed from the front of the buffer. ***/

/*** Notes:  I had a lot of trouble getting the QNX serial port FIFO to keep
from overunning with the data arriving at 50 Hz packets, 40 bytes/packet
(2kB per second data).  When the serial port FIFO gets full, it doesn't accept
any more characters until space is freed up.  If you are trying to read packets
and the buffer gets full momentarily, this will result in packet errors, since
bytes will be missed.  For instance, trying to packet_print data to
the screen in-between reading packets is enough of a delay to make the
buffer fill up and incur packet errors.
So, QNX has to continually read its serial port to clear
incoming data, but it is hard to do this quickly enough if other processes
are running in this code as well.
```

```
I am implementing check-sum error detection on incoming packets.  Since any errors
almost always affect the last half of a packet, the checksum is the sum of bytes
of only the xrot, yrot, and zrot fields.  In a mission that logs data, the mission
data can be printed to a file at the end.  If packet_errors occur, then the
current packet written is just the previous packet, and packet_errors is incremented.
*****/


/**************        DEFINEs        ******************/

#include "imuc.h"                   /**** Coefficients to convert IMU values to
engineering values ***/
#define COMM_PORT "/dev/ser1"       /**** QNX Serial port Device file (FIFO)     ****/
#define PC104_BUFFER_SIZE 4096      /****  This is not used yet ***/
#define BAUD_RATE B115200           /**** B115200, 57600, 38400, 9600 ***/
#define WAITCOUNT 42000                     /**** Used to implement a fixed-length wait
****/
#define PI 3.141592

        /*** Note the "delay(ms)" function has high overhead for small delay (i.e. 1-5 ms
        delays take much longer than 1-5 ms, accuracy is poor for these small delays).
        That is why we use a for-loop and count WAITCOUNT to implement a small delay. ***/

    /*** note:  WAITCOUNT may have to be modified depending
            on the platform this program runs on, since it
            is used to implement a fixed-time delay (and
            different systems count at different speeds...) ***/
    /*** On this 100MHz 486 PC, a WAITCOUNT of 40000 results in a delay
        of 4 Msecs. (thus 10000 counts per msec.) ***/


/***** Common TT8 definitions  ******/
#define TT8_PACKET_SIZE 42                      /*** forty bytes of data sent from
TT8 (NOT including headers)+2 checksum ***/
#define QNX_PACKET_SIZE 7                   /*** in bytes (uchars), INCLUDING header
byte ***/
#define MOTOR_DO_NOTHING 30000                  /*** long (2-byte) word: appropriate
string is "u0" ***/

#define QNX_PACKET_HEADER 'q'                   /*** header byte that QNX sends to
TT8 to identify a packet ***/
#define RESET_COMPASS 'h'
#define COMPASS_CAL_PULSE 'a'
#define START_NAV 'r'
#define STOP_NAV 'w'
#define TT8_PING 'n'
#define TT8_HOME_YOUR_FINS 'g'
#define TT8_PACKETS_PER_SEC 50

#define TT8_PACKET_HEADER1 'p'                  /*** header byte1 that TT8 sends to
QNX ***/
#define NO_COMPASS_DATA 999
#define XCOLL 1
#define YCOLL 2
#define ZCOLL 3
#define NO_COLLISION 4

/**** Mission types *****/
#define NO_MISSION 0
#define LOG_DATA_ONLY 1
#define DEPTH_CONTROL_ONLY 2
#define LOG_AND_DEPTH 3

#define RADIANS_TO_MOTOR_COMMAND 15915.494  /*** = (180/pi)*(25000/90) ****/
#define MAX_PECTORAL_FIN_ANGLE 0.2618       /*** in radians (this is 15 degrees)***/
#define MIN_PECTORAL_FIN_ANGLE -0.2618              /*** in radians ***/

#define DEPTH_METERS_PER_QUANTA 0.007534424     /*** convert A2D Quanta to depth in meters
***/


/*** typedef's to ensure compatibility with TT8 code ****/
typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned long ulong;
typedef char* ptr;
typedef void (*vfptr)(void);
```

```
/*****   Packet structure for TT8 Data: one type of packet; 40 bytes  ******/
struct packet {
  ulong samplenum;
  float xval;
  float yval;
  float zval;
  float xrot;
  float yrot;
  float zrot;
  float temp;
  float depth;
  long compass;
  short flag1;
  short collflag;
};
typedef struct packet packet;



/**** Internal mission-planning structure *****/

struct mission {
   ulong mission_num;
   int mission_type;          /*** see define's above ***/
   ulong mission_length;      /*** in seconds ***/
   float depth_to_maintain;
};
typedef struct mission mission;

/*** later we can construct arrays of missions to be scheduled, etc. ... *****/
/*** we should put filenames to log data as part of mission structure, later... ***/

/************** function prototypes ******************/


void comm_init(void);
void write_datachar_to_port(char*);
int send_packet_to_TT8(uchar*);
uchar* build_packet(char, char, long, long);
void reset_compass(void);
void cal_compass(void);
void TT8_ping(void);
void get_motor_commands(void);
void Waitcount(void);
int get_TT8_packet(packet*);          /** returns 1 if successful, 0 if not ***/
void send_TT8_start_NAV(void);
void send_TT8_stop_NAV(void);
void packet_print (packet);
void set_zero_depth(void);
int log_data_to_file (void);
void view_instructions(void);
void lookat_compass(void);
void log_biases(void);
void send_TT8_home_fins(void);
int define_mission (mission);
int execute_mission(mission);
int log_data_to_memory(long);
int maintain_depth(float, long);
int log_and_depth_control (float, long);
void show_relative_attitude (void);


/**************  Global variables    *****************/
static int comm_port;
static int comm_wait;
static int buffer_size;               /*** probably get rid of this... ***/
uchar send_packet[QNX_PACKET_SIZE] = {QNX_PACKET_HEADER,'0','0','u','0','u','0'};   /***
default packet does nothing ***/
packet get_packet;                           /**** used in option 6 ****/
float ZERO_DEPTH_VALUE = 2030.0;             /**** default depth reading for surface ***/
long LAST_GOOD_COMPASS_DATA=0;
double xsum=0, ysum=0, xrsum=0, yrsum=0, zrsum=0;
double CUR_TEMP=-1544;

   /*** Important Globals... for mission execution ***/
packet* GET_PACKETS;                  /*** This is an array which will be dynamically
allocated
                                                     to log the incoming data ****/
mission current_mission;
```

```
int packet_errors;               /*** counts number of packets missed from TT8 ***/

/**** STILL HAVE TO IMPLEMENT CHECKSUM ****/



void main()
{
  unsigned char c;
  unsigned char receive_buffer[4096];
  int i=0, j=0, b=0;
  long left_motor, right_motor, a;
  long read_secs;
  int h=0;


/**  term_load(); **/    /** screen control functions that DON'T WORK! ***/
/**  term_clear (TERM_CLS_SCR); ***/

  current_mission.mission_type = NO_MISSION;
  current_mission.mission_num = 0;
  current_mission.mission_length = 0;
  current_mission.depth_to_maintain = ZERO_DEPTH_VALUE;

  comm_init();

  printf ("Comm port 1 has been initialized. \n\n");
  printf ("CLOCKS_PER_SEC is:\t%ld\n", CLOCKS_PER_SEC);
  printf (" (for use in clock function\n");
  printf ("Sizeof(int) is: %%d  char(255) is %c\n", sizeof(int), 255);
printf ("Warning: This program does NOT implement any handshaking\n");
printf ("from the TT8 except for the 'TT8 Ping' menu option.\n");
printf ("Checksum error correction is being implemented on packets.\n");
    set_zero_depth();
    set_zero_depth();

    /**** Implement MENU: *****/
 j=0;
 while (1) {
    j++;
    fflush (stdin);
    printf ("\n**************** MENU[%d] ******************\n\n", j);
    printf ("0 -- View OPERATING INSTRUCTIONS\n");
    printf ("1 -- Reset Compass\n2 -- Calibrate Compass\n3 -- TT8 ping -- Don't invoke
when logging data!\n");
    printf ("4 -- Send Single Motor Commands\n5 -- Start TT8 sending NAV data\n");
    printf ("6 -- Read TT8 packets for a while to screen\n7 -- Home the Fish's Pectoral
Fins (Do this BEFORE option 5 always!)\n");
    printf ("8 -- Stop TT8 sending data; RESET the Tattletale\n");
/**    printf ("9 -- ");***/
    printf ("A -- Set current depth as surface (zero feet)\n");
    printf ("B -- \n");
    printf ("C -- Read asynchrounous COMPASS packets to screen for a while\n");
    printf ("D -- Log biases for 30 minutes: Mohan only!\n");
    printf ("E -- Define a navigation mission plan\n");
    printf ("F -- Execute th current mission\n");
    printf ("G -- Show vehicle attitude relative to current\n");
    printf ("\nChoose:\t");

    c=getchar();

    switch(c)  {
           case '0':     view_instructions();
                         break;
           case '1':     reset_compass();
                         break;
           case '2':     cal_compass();
                         break;
           case '3':     TT8_ping();
                         break;
           case '4':     get_motor_commands();
                         break;
           case '5':     send_TT8_start_NAV();
                         break;
           case '6':     printf ("\nHow many seconds?\t");
                         scanf ("%ld", &read_secs);
                         tcflush (comm_port, TCIFLUSH);
                         for (i=0;i<read_secs*50;i++)
                         {
                         if(get_TT8_packet(&get_packet)!=1) /**break**/;
```

```
                              packet_print (get_packet);
                              delay(20);
                              tcflush (comm_port, TCIFLUSH);
                              }
                              break;
               case '7':      send_TT8_home_fins(); break;
               case '8':      send_TT8_stop_NAV(); break;
               case 'A':      set_zero_depth(); break;
               case 'B':      /**if (log_data_to_file() == 0)
                              printf ("Data logged INCORRECTLY.\n\n");**/
                              break;
               case 'C':      lookat_compass();
                              break;
               case 'D':      log_biases(); break;
               case 'E':      define_mission(current_mission); break;
               case 'F':      execute_mission(current_mission); break;
               case 'G':      show_relative_attitude(); break;

/*** define a mission ***/
/*** execute a mission ***/
/*** write mission data to file ***/


/*** start TT8 sending data ***/
/*** start QNX logging Data (for how many seconds) ***/
/*** start QNX gathering pressure data ****/
/**** start QNX controlling fins based on pressure data ***/
/*** start QNX controlling motors while logging data ***/
/*** home fins ***/
/*** set session parameters: ignore collisions, etc. ***/

            }

}  /*** end of while ***/



    /**** When NAV is running, we should not send a packet more than
    5-7 Hz frequnecy.  When NAV is not running, this should be like
    20 Hz MAX.   See if we can have this interrupt-driven. ****/


/**************** program to send various data packets to the TT8 ******/



}


void comm_init(void)
{
       struct termios comm_termios;

       /*** set up serial port ****/
/***   fcntl (comm_port, F_SETFL, O_NONBLOCK);***/   /*** this causes QNX to reboot at
the end... ***/
       comm_port = open(COMM_PORT, /**O_NONBLOCK|**/O_RDWR);
       buffer_size = PC104_BUFFER_SIZE;

/**~   comm_wait = 0;
**/
       if (comm_port == -1)
       {
              printf ("error opening serial port\n");
              exit(1);
       }

       tcgetattr(comm_port, &comm_termios);
       comm_termios.c_cflag = /**BAUD_RATE|***/CS8|CREAD|CLOCAL;
       comm_termios.c_iflag = 0;
       comm_termios.c_lflag = 0;
       comm_termios.c_oflag = 0;
       comm_termios.c_cc[VEOF] = 1;
       comm_termios.c_cc[VEOL] = 0;
       cfsetispeed(&comm_termios, BAUD_RATE);
       cfsetospeed(&comm_termios, BAUD_RATE);
       tcsetattr(comm_port, TCSANOW, &comm_termios);

       tcflush(comm_port, TCIOFLUSH);
       tcflow (comm_port, TCOONHW);
```

```
        /**tcdrain(comm_port);***/

/**comm_send...initialization packet***/

}

void write_datachar_to_port (char* data)
{
        write (comm_port, data, 1);    /** send one byte beginning at address data ***/
        Waitcount();    /*** about a 4 millisecond delay ***/
/**     delay(1);  /*** A DELAY IS NECESSARY FOR TT8 TO READ OK; MAYBE CAN BE <1; check
later... ***/
        /**** 'delay' command needs to be replaced with something more reasonable ***/
        /*** too much overhead in 'delay' command, it's inaccurate ***/
        tcdrain (comm_port);    /*** tcflow, tcdrain?? **/

/**     printf ("Char sent...");  **/
}



int send_packet_to_TT8 (uchar* packet_address)
{
    int i;

    for (i=0; i<QNX_PACKET_SIZE; i++)
    {
       write_datachar_to_port (packet_address+i);
    }
    /*** have a response ready for TT8ping; return TT8 value***/
    return(0);
}

.
/*** This is the 7-byte packet structure that QNX sends to the TT8: it is simply an array
of uchars ***/

uchar* build_packet (char compass, char nav_home_or_ping, long left_motor, long
right_motor)
{
    send_packet[0] = QNX_PACKET_HEADER;
    send_packet[1] = compass; /** RESET or CAL ***/
    send_packet[2] = nav_home_or_ping;
    send_packet[3] = (uchar) (left_motor>>8);
    send_packet[4] = (uchar) (left_motor%256);
    send_packet[5] = (uchar) (right_motor>>8);
    send_packet[6] = (uchar) (right_motor%256);
    return (send_packet);
}


void reset_compass (void)
{

    build_packet (RESET_COMPASS, '0', MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);

    send_packet_to_TT8 (send_packet);
    delay (200);
    printf ("\nCompass Reset\n");
}


void cal_compass(void)
{
    fflush(stdin);
    printf ("\nKeep Compass in one direction: press enter");
    build_packet (COMPASS_CAL_PULSE, '0', MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);
    getchar();
    send_packet_to_TT8 (send_packet);
    delay (200);     /*** if cal_pulsed too fast, compass will hang... ***/
    printf ("\nNow turn compass to face 180 degrees from before: press enter");
    getchar();
    send_packet_to_TT8 (send_packet);
    delay(100);
    printf ("\n\nCompass calibrated\n\n");
}


void send_TT8_start_NAV(void)
```

```
{
    build_packet ('0', START_NAV, MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);
    tcflush(comm_port, TCIOFLUSH);
    send_packet_to_TT8 (send_packet);
    delay(50);
    set_zero_depth();
    set_zero_depth();
    set_zero_depth();

}


void send_TT8_home_fins(void)
{
        build_packet ('0', TT8_HOME_YOUR_FINS, MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);
        tcflush(comm_port, TCIOFLUSH);
        send_packet_to_TT8 (send_packet);
}



void send_TT8_stop_NAV(void)
{
    build_packet ('0', STOP_NAV, MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);
    tcflush(comm_port, TCIOFLUSH);
    send_packet_to_TT8 (send_packet);
    printf ("\n\nTT8 is now resetting itself; give it a few seconds.\n\n");
    delay(800);
}


void TT8_ping(void)
{
    char ping_receive[80];
    int i;

    tcflush (comm_port, TCIFLUSH);

    printf ("\nSending TT8 ping..\n.\n.\n");
    build_packet ('0', TT8_PING, MOTOR_DO_NOTHING, MOTOR_DO_NOTHING);
    send_packet_to_TT8 (send_packet);
    delay(100);                 /*** wait for TT8 to respond ***/
    i = dev_read (comm_port, ping_receive, 80,5,1,1,0,0);
    if (i<=0) {printf ("No response from TT8...\n\n"); return;}
    ping_receive[i] = '\0';
    printf ("TT8's response is:\t%s\n", ping_receive);
}

void get_motor_commands(void)
{
    long left_motor;
    long right_motor;
    char junk;

    printf ("\nEnter left motor command (-25000 to +25000, 30000 does nothing):\t");
    scanf ("%ld", &left_motor);
    if ((left_motor > 25000) && (left_motor != 30000)) left_motor=25000;
    if (left_motor < -25000) left_motor=-25000;
    printf ("\nEnter right motor command (-25000 to +25000, 30000 does nothing):\t");
    scanf ("%ld", &right_motor);
    if ((right_motor > 25000) && (right_motor != 30000)) right_motor=25000;
    if (right_motor < -25000) right_motor=-25000;
    build_packet ('0','0',left_motor,right_motor);
    tcflush (comm_port, TCIFLUSH);   /*** we DON't want to have to do this! ***/
/**    read(comm_port, &junk, 1); ***/
    send_packet_to_TT8 (send_packet);
    printf ("\n\nMotor commands sent to TT8.\n\n");
}


/**** THIS FUNCTION IS VERY IMPORTANT: CALLED BY MOST OTHER
       DATA GATHERING FUNCTIONS ********/

int get_TT8_packet(packet* packet_addr)
{
/*** This function places the packet into the packet space pointed
to by packet_addr; returns 1 if successful, 0 if unsuccessful ***/

    int i=-1;
    uchar receive[TT8_PACKET_SIZE];
    uchar header1 = 'o', header2 = 'o';
    ulong* longptr;
    ulong sample=0;
    float* floatptr;
    float floatval;
    long j=0, wt=0;
    char cksum[4];
    long* cksumptr;
    long cksumrcv = 0;


  while (header1 != TT8_PACKET_HEADER1)
    {
      i = dev_read (comm_port, &header1, 1, 1, 1, 1, 0, 0); /** use dev_read to do
timeout... **/
      if (i<=0) {printf ("Packet not found\n\n"); return(0);}
    }

/**** here is dev_read:  ****/

        if (i=dev_read(comm_port, receive, TT8_PACKET_SIZE, TT8_PACKET_SIZE, 1, 1, 0, 0)
== -1)
        {   printf ("FUCKING error\n"); return(0);}

/***    else printf ("\n\n%d Bytes gotten\n\n", i); ****/

        /**** CHECKSUM ERROR CORRECTION: ***/
        cksum[0] = receive[40];
        cksum[1] =  receive[41];
        cksum[2] = 0;
        cksum[3] = 0;
        cksumptr = (long*)(cksum);
        for (i=0;i<28; i++)
        {
          cksumrcv += receive[i];
        }
        if (*(cksumptr) != cksumrcv)      /** CHECKSUM error... **/
        {
          printf ("Checksum error: calc   %ld    rcv   %ld \n", *cksumptr, cksumrcv);
          printf ("receive[40] is:%d\treceive[41] is:%d\n", receive[40], receive[41]);
          return (0);                     /** exit without reading packet data ***/
        }
        longptr = (ulong*)receive;
        sample = *longptr;
        packet_addr->samplenum = sample/2;
        for (i=4; i<=32; i=i+4)
        {
          /** read 8 float values, 4 bytes each ***/
            floatptr = (float*)(receive+i);
            floatval = *floatptr;
            switch(i) {
                /******* IMU data is stored in engineering units *********/
                        case 4: packet_addr->xval = floatval; break;
                        case 8: packet_addr->yval = floatval; break;
                        case 12: packet_addr->zval = floatval; break;
                        case 16: packet_addr->xrot = floatval-(0.000765*CUR_TEMP) - 1.2682;
break;
                        case 20: packet_addr->yrot = floatval - (0.00104*CUR_TEMP) -
1.7367; break;
                        case 24: packet_addr->zrot = floatval - ( (-0.00048127)*CUR_TEMP) +
0.422866; break;
                        case 28: packet_addr->temp = floatval; CUR_TEMP = floatval; break;
                        case 32: packet_addr->depth = ZERO_DEPTH_VALUE - floatval; break;
                /*** note depth is positive DOWN; zero at the surface ***/
                        }
        }
        packet_addr->compass = receive[36]*256 +  receive[37];
        packet_addr->flag1 = receive[38];
        packet_addr->collflag = receive[39];
        xsum+= packet_addr->xval*.02;  /** These are little integration **/
        ysum+= packet_addr->yval*.02;  /** hacks, please ignore ***/
        xrsum+=packet_addr->xrot*.02;
        yrsum+=packet_addr->yrot*.02;
        zrsum+=packet_addr->zrot*.02;

/** FILTER OUT DUMMY COMPASS VALUES: ***/

        if (packet_addr->compass > 359)  /** includes NO_COMPASS_VALUE=999 **/
            {packet_addr->compass = LAST_GOOD_COMPASS_DATA;}
        else {LAST_GOOD_COMPASS_DATA = packet_addr->compass;}
        return(1);

}


void Waitcount(void)
{       /*** NOTE THIS FUNCTION IS SYSTEM DEPENDENT!!! ***/
/***     talking to Tattletale may not work if the
         value of WAITCOUNT is not sufficient to implement
         a 4 millisecond delay. ****/

    int i;

    for (i=0;i<WAITCOUNT; i++);

}


void packet_print (packet p)
{

/***    term_clear (TERM_CLS_SCR); ***/
    printf ("--------------------------------------------\n");
    printf ("\nSamplenum:\t\t%ld\n", p.samplenum);
    printf ("Raw X Accel:\t%lf\tsum:  %lf\n", p.xval, xsum);
    printf ("Raw Y Accel:\t%lf\tsum:  %lf\n", p.yval, ysum);
    printf ("Raw Z Accel:\t%lf\n", p.zval);
    printf ("Raw X Rate:\t\t%lf\tsum:  %lf\n", p.xrot, xrsum);
    printf ("Raw Y Rate:\t\t%lf\tsum:  %lf\n", p.yrot, yrsum);
    printf ("Raw Z Rate:\t\t%lf\tsum:  %lf\n", p.zrot, zrsum);
    printf ("Raw IMU temp:\t%lf\n", p.temp);
    printf ("Raw depth:\t\t%lf\n", p.depth);
    printf ("Compass:\t\t%ld\n", p.compass);
    printf ("Flag1:\t\t\t%d\n", p.flag1);
    printf ("CollFlag:\t\t%d\n\n", p.collflag);
    printf ("--------------------------------------------");
    printf ("\n\n\n\n\n\n\n\n");
/**    delay(1);           /** get rid of this!****/
}



/*** Need to study:

1) Interrupt driven stuff
2) writing and reading to serial ports in QNX; what happens when full
3) how to best allocate bandwidth -- multitasking
****/

/*** need to figure out: why isn't compass working,
                        why isn't collision flags working
ÿ                       why is pressure value erratic overly sensitive
                        why does xval have a large bias (when connected
                                                  or floating ) **/


void set_zero_depth (void)
{
    packet now;

    tcflush(comm_port, TCIFLUSH);
    if (get_TT8_packet(&now) == 1)
        {
            printf ("\nCurrent Raw depth is %4.5lf (A2D quanta); setting this as the
surface value.\n", now.depth);
            ZERO_DEPTH_VALUE = ZERO_DEPTH_VALUE - now.depth;
        }
    else
        {
            printf ("No current value available for current depth.\nUsing last value
set:\tRaw surface value is %4.5lf\n\n", ZERO_DEPTH_VALUE);
        }
}


int log_data_to_file (void)
{
    int secs;
    FILE *fp;
    char filename[10];
    int data_points, i;
    packet get_it;

    printf ("How many seconds do you want to log data for?\t");
    scanf ("%d", &secs);
    data_points = secs*50;
    printf ("\nEnter the filename to store IMU data in:\t");
    scanf ("%s", filename);
    if ( (fp=fopen(filename, "w")) == NULL)
    {
       printf ("File not open-able.\n\n");
       return (0);
    }

    tcflush (comm_port, TCIFLUSH);
    for (i=0; i<data_points; i++)
    {
       if (get_TT8_packet(&get_it) == 0)
       {
          fclose (fp);
          printf ("Packet not gotten correctly\n\n");
          return (0);
        }

       if (fprintf (fp, "%ld\n%f\n%f\n%f\n%f\n%f\n%f\n%f\n", get_it.samplenum,
get_it.temp, get_it.xval, get_it.yval, get_it.zval, get_it.xrot, get_it.yrot, get_it.zrot)
< 0)
       {
          fclose (fp);
          printf ("Packet not written to correctly\n\n");
          return (0);
       }

    }

    fflush (fp);
    fclose (fp);
    return (1);
}


void view_instructions(void)
{

    printf ("\n\n\n\n\nHere are some rules for this menu.\n\n");
    printf ("Rule 1 -- When you first run this program, use option 3 several\n");
    printf ("times to verify the TT8 is connected, working and happy.");
    printf ("If you don't see 'TT8 is alive; TT8 is hungry for action'\n");
    printf ("SOMETHINGÿIS AWRY:  see Mohan.\n\n");
    printf ("Rule 2 -- Next thing to try is to Reset the Compass, option 1.\n\n");
    printf ("Rule 3 -- Now, Calibrate the Compass, option 2.  Note that this is\n");
    printf ("a 'hard-iron' calibration, which requires you to turn the whole fish\n");
    printf ("around in its place between hitting 'Enter' a second time.\n\n");
    printf ("Rule 4 -- Use the 'send single motor' command to test out the
communication\n");
    printf ("through the TT8 to the Smart Motors.  Test this for a few values and\n");
    printf ("visually verify the motors go to the correct angles.  +25000 corresponds\n");
    printf ("to 90 degrees positive angle, -25000 to 90 degrees down.\n\n");
    printf ("\n\nEnter to continue...\n\n"); getchar(); getchar();
    printf ("Rule 5 -- Now we are ready to get data.  The Tattletale is NOT\n");
    printf ("talking to its sensors (IMU, depth and comapss) UNTIL you tell it\n");
    printf ("to 'Start NAV data.'  Once you do this, you can verify that it is\n");
    printf ("talking to it's sensors by a periodic green LED pulse (about 3 Hz)\n");
    printf ("on the compass board.  If this does not happen SOMETHING is AWRY.  Talk\n");
    printf ("to Mohan.\n\n");
    printf ("Rule 6 -- NEVER: Ping the TT8 while it is sending data. May corrupt
data.\n");
    printf ("                 Try to Reset or Calibrate compass while TT8 is sending
data.\n");
    printf ("???\n");
    printf ("More options to be described later: logging data; viewing real-time;\n");
    printf ("Logging data in background, Multitasking with motor control; \n");
    printf ("implementing a depth-control loop.\n\n\n\n");



    printf ("Enter to continue...\n");
    getchar();
}


void lookat_compass (void)
{
    int i, num;
    packet p;

    p.compass = NO_COMPASS_DATA;
    printf ("\nHow many compass samples do you want to see?\t");
    scanf ("%d", &num);
    printf ("\n\n");
    tcflush (comm_port, TCIFLUSH);
    for (i=0;i<num;i++)
    {
        p.compass = NO_COMPASS_DATA;  /** figure out a better way to do this ***/
        while (p.compass == NO_COMPASS_DATA)
        {
            if (get_TT8_packet(&p) == 0) return;
        }
        printf ("Heading:\t%ld\n", p.compass);
    }
    printf ("\n\n");

}


void log_biases(void)
{

    FILE* fp;
    char fnm[15];
    long i,j;
    packet get_it[100];
    packet avg;
    float tempsum=0, xsum=0, ysum=0, zsum=0, xrsum=0, yrsum=0, zrsum=0;

   sprintf (fnm, "BIASLOG");
   printf ("\nOpening file...\n");
   if ( (fp=fopen("BIASLOG", "w")) == NULL)
       {
         printf ("File can't be opened\n");
         return;
       }
    printf ("File opened successfully\n");
    printf ("Writing IMU data...\n");
    fflush(fp);

    for (i=0; i<10000; i++)
        {
            printf ("\nWriting to file[%d]:\t%s\n", i, fnm);
            printf ("Temp is\t%f\n", avg.temp);
            tcflush (comm_port, TCIFLUSH);
            for (j=0; j<100; j++)
            {
                while (get_TT8_packet(get_it+j) == 0)
                {
                    printf ("Packet errors...\n");
                }
            }
            tempsum=0;xsum=0;ysum=0;zsum=0;xrsum=0;yrsum=0;zrsum=0;
            for (j=0; j<100; j++)
            {
                tempsum+=get_it[j].temp;
                xsum+=get_it[j].xval;
                ysum+=get_it[j].yval;
                zsum+=get_it[j].zval;
                xrsum+=get_it[j].xrot;
                yrsum+=get_it[j].yrot;
                zrsum+=get_it[j].zrot;
            }
            avg.temp=tempsum/100;
            avg.xval=xsum/100;
            avg.yval=ysum/100;
            avg.zval=zsum/100;
            avg.xrot=xrsum/100;
            avg.yrot=yrsum/100;
            avg.zrot=zrsum/100;
            if (fprintf (fp, "%f\n%f\n%f\n%f\n%f\n%f\n", avg.temp, avg.xval, avg.yval,
avg.xrot, avg.yrot, avg.zrot) < 0)
                {
                    fclose (fp);
                    printf ("Packet not written to correctly\n\n");
                    return;
                }
            fflush(fp);
        }

        fflush (fp);
        fclose (fp);

}


/***** WE SHOULD IMPLEMENT CHECKSUM IN BOTH DIRECTIONS *****/
/***** TT8 should NOT mess-UP MOTOR COMMANDS!!! ****/




int define_mission (mission MS)
/**** defines the current_mission; actually needs no arguments ***/
{
        printf ("\nWhat type of mission?\n0-NO MISSION\t1-LOG DATA ONLY\t2-DEPTH CONTROL
ONLY\t3-LOG AND DEPTH\nChoose:\t");
        scanf ("%d", &(current_mission.mission_type));
        if (current_mission.mission_type == NO_MISSION) {printf ("Goodbye...\n");
return(0);}
        if (current_mission.mission_type == LOG_DATA_ONLY)
        {
          printf ("\n\nHow many seconds do you want to log data for?\t");
          scanf ("%ld", &(current_mission.mission_length));
          printf ("\n\n"); return(1);
        }
        if (current_mission.mission_type == DEPTH_CONTROL_ONLY)
        {
          printf ("How many seconds do want depth control for?\t");
          scanf ("%ld", &(current_mission.mission_length));
          printf ("\nWhat depth (in meters) do you want to maintain?\t");
          scanf ("%f", &(current_mission.depth_to_maintain));
          printf ("\n\n"); return(1);
        }
        if (current_mission.mission_type == LOG_AND_DEPTH)
        {
          printf ("How long do you want to log data and control depth for?\t");
          scanf ("%ld", &(current_mission.mission_length));
          printf ("\nWhat depth (in meters) do you want to maintain?\t");
          scanf ("%f", &(current_mission.depth_to_maintain));
          printf ("\n\n"); return(1);
        }
  printf ("Huh?\n");
  return(1);
}



/*********** Execute a mission ***********/

int execute_mission (mission M)
{
        /**** returns 1 if mission successful, 0 if unsuccessful ***/
        long pnum;
        char ans;
        char fnm[10];
        FILE* fp;
        long i;

        /***** make sure tt8 is sending packets ****/

        packet_errors = 0;
        M.mission_num++;
        printf ("\nMISSION NUMBER %ld:\t", M.mission_num);
        if (M.mission_type == LOG_DATA_ONLY) printf ("Log Data Only for %ld seconds\n",
M.mission_length);
        if (M.mission_type == DEPTH_CONTROL_ONLY) printf ("Depth Control Only for %ld
seconds\n", M.mission_length);
        if (M.mission_type == LOG_AND_DEPTH) printf ("Log Data and Control Depth for %ld
seconds\n", M.mission_length);
```

```
.       pnum = TT8_PACKETS_PER_SEC * M.mission_length;    /*** number of packets to get
***/

        if ((M.mission_type == NO_MISSION))
        {
         printf ("\nNo current mission has been defined, or mission length is set to 0\n");
          return(0);
        }

        if (M.mission_type == LOG_DATA_ONLY)
        {
                printf ("Finding memory storage to log data...\n");
                GET_PACKETS = malloc(pnum*sizeof(packet));
                if (GET_PACKETS==NULL)        /*** is this right? ****/
                {
                        printf ("Memory allocation failed.  Mission being cancelled...\n");
                        return(0);
                }
                printf ("Memory allocation successful.  Beginning data logging for %ld
seconds:\n\n", M.mission_length);
                /******* put data logging function here *******/
                log_data_to_memory(pnum);
                /*** returned from mission... ****/
                printf ("\n%d packet errors occurred during logging.\n", packet_errors);
                printf ("Write data to file(y/n)?\t");
                fflush(stdin);
                scanf ("%c", &ans);
                if ((ans=='y') || (ans=='Y'))
                {
                  getf: printf ("\nAll logged data will be dumped into file: Enter
filename:\t");
                  scanf ("%s", fnm);
                  printf ("\nOpening file...\n");
                  if ( (fp=fopen(fnm, "w")) == NULL)
                        {
                          printf ("File can't be opened\n");
                          goto getf;
                        }
                  printf ("File opened successfully\n");
                  printf ("Writing IMU data...\n");
                  for (i=0; i<pnum; i++)
                   {
                        if (fprintf(fp, "%ld\n%f\n%f\n%f\n%f\n%f\n%f\n%f\n%ld\n",
GET_PACKETS[i].samplenum, GET_PACKETS[i].temp, GET_PACKETS[i].xval, GET_PACKETS[i].yval,
GET_PACKETS[i].zval, GET_PACKETS[i].xrot, GET_PACKETS[i].yrot, GET_PACKETS[i].zrot,
GET_PACKETS[i].compass) < 0)
                        {
                            fclose (fp);
                            printf ("Packet %ld not written to correctly\n\n", i);
                            free (GET_PACKETS);
                            return(0);
                        }
                   }
                  fflush(fp);
                  fclose (fp);
                  free (GET_PACKETS);
                  printf ("\nFile write successful.\nPress Enter...\n"); getchar();
                  return(1);
                }
             free(GET_PACKETS);
             printf ("\nLogged Data is thrown away...\nPress Enter...\n"); getchar();
             return(1);
        }


        if (M.mission_type == DEPTH_CONTROL_ONLY)
        {
                printf ("Beginning a mission to maintain depth to %lf meters, for %ld
seconds...\n", M.depth_to_maintain, M.mission_length);

                maintain_depth(M.depth_to_maintain, pnum);
                /***** Return from maintaining depth *****/
                printf ("\n%d packet receive errors occurred during depth control loop.\n",
packet_errors);
                printf ("\nType enter to continue...\n\n"); fflush(stdin); getchar();
                printf ("Returning to main menu...\n");
                free (GET_PACKETS);    /*** this shouldn't be necessary ***/
                return(1);
        }
        if (M.mission_type == LOG_AND_DEPTH)
        {
                printf ("Beginning a mission to log data and maintain a depth of %lf meters
for %ld seconds...\n", M.depth_to_maintain, M.mission_length);
                printf ("Finding memory storage to log data...\n");
                GET_PACKETS = malloc(pnum*sizeof(packet));
                if (GET_PACKETS==NULL)          /*** is this right? ****/
                {
                        printf ("Memory allocation failed.  Mission being cancelled...\n");
                        return(0);
                }
                printf ("Memory allocation successful.  Beginning data logging for %ld
seconds:\n\n", M.mission_length);

                log_and_depth_control(M.depth_to_maintain, pnum);
                /**** Return from this function  ***/
                printf ("%d packet receive errors occurred during logging and control
loop.\n", packet_errors);
                printf ("Write data to file(y/n)?\t");
                fflush(stdin);
                scanf ("%c", &ans);
                if ((ans=='y') || (ans=='Y'))
                {
                  gf: printf ("\nAll logged data will be dumped into file: Enter
filename:\t");
                  scanf ("%s", fnm);
                  printf ("\nOpening file...\n");
                  if ( (fp=fopen(fnm, "w")) == NULL)
                        {
                          printf ("File can't be opened\n");
                          goto gf;
                        }
                  printf ("File opened successfully\n");
                  printf ("Writing IMU data...\n");
                  for (i=0; i<pnum; i++)
                   {
                        if (fprintf(fp, "%ld\n%f\n%f\n%f\n%f\n%f\n%f\n%f\n",
GET_PACKETS[i].samplenum, GET_PACKETS[i].temp, GET_PACKETS[i].xval, GET_PACKETS[i].yval,
GET_PACKETS[i].zval, GET_PACKETS[i].xrot, GET_PACKETS[i].yrot, GET_PACKETS[i].zrot) < 0)
                        {
                            fclose (fp);
                            printf ("Packet %ld not written to correctly\n\n", i);
                            free (GET_PACKETS);
                            return(0);
                        }
                   }
                  fflush(fp);
                  fclose (fp);
                  free (GET_PACKETS);
                  printf ("\nFile write successful.\nPress Enter...\n"); getchar();
                  return(1);
             }
             free(GET_PACKETS);
             printf ("\nLogged Data is thrown away...\nPress Enter...\n"); getchar();
             return(1);
          }
   printf ("HuhHuh?\n");    /**** shouldn't be here ***/
   return(0);
}


/***** This function does not yet work… needs minor changes *****/
int log_and_depth_control (float ref_depth, long num_of_packets)
{
        long i,j,k;
        float avg_depth=0, errr;
        float PROP_GAIN=8.2;     /*** prop. gain for control loop; first-order guess ***/
        float command;

        j=num_of_packets/10;
        tcflush (comm_port, TCIOFLUSH);         /*** flush input and output serial FIFOs
***/
        for (i=0; i<j; i++)
        {
                avg_depth=0;
                for (k=0; k<10; k++)
                {
                    if (get_TT8_packet(GET_PACKETS+((10*i)+k)) != 0)
                    {
                          packet_errors++;
                          GET_PACKETS[i] = GET_PACKETS[i-1];
                    }                                        /***what if collision?
***/
                    avg_depth += GET_PACKETS[i].depth;        /*** this is in meters
***/
              }
         avg_depth = avg_depth/10;
         errr = avg_depth-ref_depth;                /*** this is in meters ***/
         command = (PROP_GAIN * errr);              /*** this should be in radians ***/
         if (command > MAX_PECTORAL_FIN_ANGLE) command = MAX_PECTORAL_FIN_ANGLE;
         if (command < MIN_PECTORAL_FIN_ANGLE) command = MIN_PECTORAL_FIN_ANGLE;
         command = command * RADIANS_TO_MOTOR_COMMAND;
         tcflush (comm_port, TCIOFLUSH);  /*** we don't want to do this! ***/
         build_packet ('0','0',command, command);   /** both fins do same thing **/
         send_packet_to_TT8 (send_packet);
         printf (".");                              /*** JUST A TEMPORARY CHECK TO MAKE
SURE WE DON't LOCK UP ***/
        }
        return(1);                   /*** successful ***/
}



int log_data_to_memory (long num_of_packets)
{
        long i;

        tcflush (comm_port, TCIFLUSH);
        for (i=0; i<num_of_packets; i++)
        {
          if (get_TT8_packet(GET_PACKETS+i) != 1)    /*** unsuccessful get ***/
              {
                packet_errors++;
                GET_PACKETS[i] = GET_PACKETS[i-1]; /*** replace bad packets with last
good packet... ***/
              }
        }
        return(1);
}


/*** depth loop is implemented at 5 Hz, initially: we take an average
depth every 10 incoming packets and use this as the fed-back depth value
****/



int maintain_depth(float ref_depth, long num_of_packets)
{
        long i,j,k;
        packet ptemp;
        float avg_depth=0, errr;
        float PROP_GAIN=8.2;     /*** prop. gain for control loop; first-order guess ***/
        float command;

        j=num_of_packets/15;
        tcflush (comm_port, TCIOFLUSH);        /*** flush input and output serial FIFOs
***/
        for (i=0; i<j; i++)
        {
              avg_depth=0;
              for (k=0; k<15; k++)
              {
                    if (get_TT8_packet(&ptemp) == 0)
                    {
                          packet_errors++;
                          /***old packet should still be in ptemp ***/
                    }                                        /***what if collision?
***/
                    avg_depth += ptemp.depth;          /*** this is in meters ***/
              }
        avg_depth = (avg_depth*DEPTH_METERS_PER_QUANTA)/15;
        errr = avg_depth-ref_depth;                /*** this is in meters ***/
        command = (PROP_GAIN * errr);              /*** this should be in radians ***/
        if (command > MAX_PECTORAL_FIN_ANGLE) command = MAX_PECTORAL_FIN_ANGLE;
        if (command < MIN_PECTORAL_FIN_ANGLE) command = MIN_PECTORAL_FIN_ANGLE;
        command = command * RADIANS_TO_MOTOR_COMMAND;
        build_packet ('0','0',command, command);   /** both fins do same thing **/
        tcflush (comm_port, TCIFLUSH);      /*** see note below ****/
        send_packet_to_TT8 (send_packet);
        printf ("avg depth is: %lf meters\n", avg_depth);
      }
      return(1);                    /*** successful ***/
}


/***** Note eventually we don't want to have to flush the serial inport, but
this is a hack.  I couldn't find a better way (for the moment) to allow
writing serial output when the input buffer is 'clogged.'  Later we
might have a separate process to constantly be 'cleaning' the input
buffer while the data is either logged, used for depth control, or both. ***/



void show_relative_attitude(void)
{
    packet getp, lastp;
    double curpitch=0;
    double curroll=0;
    double curyaw=0;

    while (!get_TT8_packet(&lastp));
    while (1)
    {
       if (get_TT8_packet(&getp) != 1)
           { getp = lastp; }
       curpitch += (((getp.xrot+lastp.xrot)/2) * Axr * 180)/(50*PI);
       curroll += (((getp.yrot+lastp.yrot)/2) * Ayr * 180)/(50*PI);
       curyaw += (((getp.zrot+lastp.zrot)/2) * Azr * 180)/(50*PI);
       printf ("Pitch: %lf\tRoll: %lf\tYaw: %lf\tTemp: %lf\n", curpitch, curroll, curyaw,
getp.temp);
       lastp = getp;
     }
}
```

C.3 MATLAB Code

This MATLAB code is used to post-process data logged by the Navigation Subsystem. It implements the strapdown algorithm on IMU data as follows:

1) First compute an alignment (or the "tilt" of the system relative to the gravity vector).
2) Compute the time-evolution of the direction cosine matrices (attitude).
3) Resolve all the accelerations into the reference frame; remove gravity from the z-axis.
4) Double integrate the resolved accelerations to track position.

These various steps are then graphically plotted to visualize position and attitude.

NAVIGATE.M

```
function [xval, yval, zval, xres,yres,zres, XPOS, YPOS, ZPOS] = attitude(vect)

%NAVIGATE.M---  Calculate direction cosine matrices for Attitude,
%              then calculate position trajectory and generate plots.
%               Mohan Gurunathan - 5/12/98
%
%  This program calculates the attitude of the vehicle using direction
%  cosines.  Then accelerations are resolved and position computed.
%  The outputs are resolved accelerations and positions.
%  The input is a repeating vector in the following format:
%
%  samplenum, temp, xval, yval, zval, xrot, yrot, zrot, compass.
%
%  Each of these is a packet of info at 50 Hz.
%  This program does not use the compass info yet; eventually can
%  use them to determine actual azimuth.
%
%  Initial attitude (ALIGNING the system) is determined by measuring
%  gravity vector with the accelerometers.  In order for this to be
%  accurate, the vehicle should be not have any motion-accelerations
%  when the data begins (for at least 1/2 second) so the initial
%  attitude can be computed by measuring the gravity vector with the
%  accelerometers.


load INSgains.mat;


N=9;    %INSERT NUMBER OF DATA PER PACKET HERE

samps = length(vect)/N;              % number of samples of data
secs = samps/50;             % number of seconds of data
dt = 1/50;

disp ('Data is the following number of seconds long:')
disp (secs);
disp (''); disp ('');


%  Separate long vector into components:
%-------------------------------------------
sampnum = vect(1:N:length(vect));
temp = vect(2:N:length(vect));
xval = vect(3:N:length(vect));
yval = vect(4:N:length(vect));
zval = vect(5:N:length(vect));
xrot = vect(6:N:length(vect));
yrot = vect(7:N:length(vect));
zrot = vect(8:N:length(vect));
compss = vect(9:N:length(vect));


%SCALE THESE to engineering units, i.e. m/s/s and delta theta's (radians):
```

```
%----------------------------------------------------------------------
xval = xval*Ax;
yval = yval*Ay;
zval = zval*Az;
xrot = xrot*Axr*dt;
yrot = yrot*Ayr*dt;
zrot = zrot*Azr*dt;

% later can change this scaling for a trapezoidal integration algorithm...?


% DETERMINE INITIAL ATTITUDE (or C at time 0) using Accelerometer data:
% This assumes the vehicle experiences no physical acceleration in the first
% 25 samples (1/2 second) when data logging is started, to measure gravity.
%  Average first 20 values of xaccel, yaccel, zaccel to get
%    initial gravity vector g0:
% we can later determine how many to average by seeing how long the
%   averaged acceleration vector magnitude is valued at about 9.8

%  [0 0 1] = C(1)*g(0); solve this to find initial cosine matrix (C),  MUST
%  BE SOLVED SO THAT C is orthogonal in columns...?

initxval = mean (xval(1:20));
inityval = mean (yval(1:20));
initzval = mean (zval(1:20));
initmag = sqrt (initxval^2 + inityval^2 + initzval^2);
initxval = initxval/initmag; inityval = inityval/initmag; initzval=initzval/initmag;
%(NORMALIZE THE SIZE!--same as dividing by 9.8, if vehicle has not moved)


% If we want to watch the time evolution of matrix C let us define C as 9
% SEPARATE vectors in time:
% C = [c11 c12 c13; c21 c22 c23; c31 c32 c33]  --> each at any given time k
%
% so above for initial C matrix, we find c11(1), c12(1)...c33(1)
%



c31(1) = -initxval;
c32(1) = -inityval;
c33(1) = -initzval;

% now we perform gram-schmidt orthogonalization:
AA = [c31(1) c32(1) c33(1)]';
BB = [0 1 0]';
CC = [1 0 0]';

BB = BB - ((AA'*BB)/(AA'*AA))*AA;
CC = CC - ((AA'*CC)/(AA'*AA))*AA - ((BB'*CC)/(BB'*BB))*BB;


%  normalize length:
BBmag = sqrt (BB(1)^2 + BB(2)^2 + BB(3)^2);
CCmag = sqrt (CC(1)^2 + CC(2)^2 + CC(3)^2);
BB = BB/BBmag;
CC = CC/CCmag;

% now we have orthoganalized the system and determined initial attitude,
% except for ignoring absolute azimuth (which we can later correct with
% compass measurements if desired)

c11(1) = CC(1); c12(1) = CC(2); c13(1) = CC(3);
c21(1) = BB(1); c22(1) = BB(2); c23(1) = BB(3);

% Initial attitude is complete!



% Loop to continually figure out the direction cosine matrix, C(2) to C(samps)
% using recursion:  C(k+1) = C(k)*A(k) (see page 297 in Titterton and Weston)
% first try uses rectangular algorithm  5/9/98 (later make trapezoidal)
%


oldC=[c11(1) c12(1) c13(1); c21(1) c22(1) c23(1); c31(1) c32(1) c33(1)];
disp ('Initial attitude')
disp (oldC);

%return;
```

```
%oldC = eye(3);   % initial attitude is same as reference for testing


disp ('Evolving the direction cosine matrix...');
for i=2:samps,

% Create skew-symmetric matrix from body rotation inputs
sigmax = [0, -zrot(i), yrot(i); zrot(i), 0, -xrot(i); -yrot(i), xrot(i), 0];
magsigma = sqrt (xrot(i)^2 + yrot(i)^2 + zrot(i)^2);

A      =      eye(3)      +      (((sin(magsigma))/magsigma)*sigmax)      +      ((1-
cos(magsigma))/(magsigma^2))*((sigmax)^2);
newC = oldC * A;

c11(i) = newC(1,1);
c12(i) = newC(1,2);
c13(i) = newC(1,3);
c21(i) = newC(2,1);
c22(i) = newC(2,2);
c23(i) = newC(2,3);
c31(i) = newC(3,1);
c32(i) = newC(3,2);
c33(i) = newC(3,3);

oldC = newC;
end;

end

figure(1);
clg;

plot3 (c11(1),c21(1),c31(1),'y');
hold on;

% PLOT ATTITUDE
%---------------------------------
plot3 (c11, c21, c31,'y*');
plot3 (c12,c22,c32,'r*');
plot3 (c13,c23,c33,'g*');
title ('Attitude plot*');
grid;
xlabel ('X-axis, reference frame');
ylabel ('Y-axis, reference frame');
zlabel ('Z-axis, reference frame');


disp ('Calculating the position trajectory...')

%START CALCULATING THE TRAJECTORY IN X-Y-Z coordinates...
% First, resolve the accelerations using the direct-cosine matrix at time i:
%-------------------------------------------------------------
for i=1:samps,
Cmatrix = [c11(i) c12(i) c13(i); c21(i) c22(i) c23(i); c31(i) c32(i) c33(i)];
resolved = Cmatrix* [xval(i);yval(i);zval(i)];
xres(i) = resolved(1);
yres(i) = resolved(2);
zres(i) = resolved(3) + 9.822;    %compensate for gravity;
end;


% Double integrate the resolved accelerations
% assumes zero initial position and velocity here -- easy to change later...
%------------------------------------------------
XPOS = integ(integ(xres));
YPOS = integ(integ(yres));
ZPOS = integ(integ(zres));


disp ('Plotting...');

%RESOLVED ACCELERATION PLOTS
%---------------------------
figure(2);clg;
subplot (3,1,1);
plot ((1:samps)/50, xres,'b');grid;
title ('Resolved x-acceleration');
ylabel ('Accel, m/s/s');
```

```
subplot (3,1,2);
plot((1:samps)/50, yres(1:samps),'b');grid;
title ('Resolved y-acceleration');
ylabel ('Accel, m/s/s');
subplot (3,1,3);
plot ((1:samps)/50, zres,'b');grid;
title ('Resolved z-acceleration');
xlabel ('Seconds'); ylabel ('Accel, m/s/s');


%POSITION PLOTS
%-------------------------------------
figure(3); clg;
plot ((1:samps)/50, XPOS, (1:samps)/50, YPOS, (1:samps)/50, ZPOS); grid;
title ('X, Y, and Z Position Trajectories');
grid;
axis ([0 secs -7.5 +7.5]);
xlabel ('Seconds');
ylabel ('Trajectory profile, meters');
grid;
text (7, XPOS(350)+0.8, 'X');
text (7, YPOS(350)-0.6, 'Y');
text (7, ZPOS(350)-0.6, 'Z');


% END

% EXPERIMENTAL VARIATIONS TO TRY:
%
% 1)  This algorithm appears to be a rectangular integration
%       algorithm..?  Could better accuracy be gained by modifying it
%       to a trapezoidal, or even an Euler integration method?
%
% 2)  Perhaps the tilt alignment could be recomputed any time
%       the windowed variance of the accelerations was below
%       some nominal value, suggesting that no accelerations
%       were occurring (since accelerations are spiky and
%       produce larger variances).  "Windowed variance" means
%       if we are at time k, compute the variance for the last
%       n samples, i.e. the variance of each acceleration between
%       [k-n] and [k].
%
% 3)  The integration might be done more smoothly by re-designing this
%       algorithm in SIMULINK.
%
```

INTEG.M

```
function vout=integ(vect)
%       Integrates function numerically at 50 Hz
%       Mohan Gurunathan
%       5/6/98
%


K = length(vect);
sm=0;

for i=1:K,
        sm=sm+vect(i);
        vout(i)=sm;
end

vout=(vout')./50;

end;
```

Appendix D: Hydrodynamic Coefficients

The following coefficients were used in the SIMULINK model described in Section 4.1. They are for the most part standard hydrodynamic coefficients representing physical characteristics of the fish such as mass, moment of inertia, center of gravity, center of buoyancy, drag and skin-friction coefficients, etc. These numbers are the first crude estimates produced for the fish, and are the only ones made available at the present time. Therefore they may need to be revised in future models.

$$m = 136.36kg.$$
$$X_{\dot{u}} = -10.9kg.$$
$$X_{wq} = -122.4kg.$$
$$X_{|u|u} = -3.9\tfrac{kg}{m}$$
$$\rho = 1000\tfrac{kg}{m^2}$$
$$A = 105.5in^2 = 0.06806m^2$$
$$Z_{\dot{w}} = -122.4kg.$$
$$Z_q = 10.9kg.$$
$$Z_w = -300\tfrac{kg}{m}$$
$$I_y = 48.5kg \cdot m^2$$
$$M_{\dot{q}} = -29.1kg \cdot m^2$$
$$M_q = -100\tfrac{kg}{m}$$
$$M_w = 111.5kg.$$
$$z_G = +0.20m.$$
$$z_B = -0.20m.$$
$$W = B = 300lb. = 1336.4N$$

Inputs: $F_{xthrust}, \phi$

State Vars: u, w, q, x, z, θ