MEMS Inertial Navigation System

Jordan Crittenden (jsc59) Parker Evans (pae26)

May 8, 2008

Abstract

Our goal was to design and construct a simple Inertial Navigation System using micro-electromechanical systems (MEMS) sensors. The sensors are processed by a microcontroller and external floating point unit to estimate current state. Although we made good progress, significant difficulties along the way prevented us from developing a system that lived up to our initial expectations. We were, however, able to obtain modest results with the system we developed and, given more time, would have been able to improve upon them.

1 Background

1.1 Inertial Navigation

An Inertial Navigation System (INS) is a device which computes the real-time state of a moving vehicle using motion sensors. By state we mean the position, velocity, and orientation of the vehicle. An INS can be used either as an isolated unit to provide continuous state information to a pilot, or in conjuction with a control system for autonomous movement. INSs are widely used in military and commercial projects, and have been under constant development and revision for several decades, but it is only recently that the technology has been accessible to hobbyists.

An INS can be logically decomposed into an Inertial Measurement Unit (IMU), which measures instantaneous accelerations or velocities in the body frame, and a state update system which uses the IMU values to update the position, velocity, and orientation of the vehicle in the navigation frame. Coordinate frames will be discussed shortly.

1.2 MEMS Sensors

There are several different ways to design an IMU, and the particular design chosen is often the driving factor in the cost of the entire INS. The recent proliferation of MEMS sensors has enabled the construction of low-cost IMUs with a tradeoff of reduced accuracy.

MEMS IMUs generally require three orthogonal measurements of acceleration and three measurements of rotational velocity around the same orthogonal axes. Accelerometers and gyroscopes respectively can produce these measurements. Both use micromachined surface capacitors to measure forces, whether due to an acceleration or the Coriolis effect.

1.3 State Update

As mentioned above, measurements from the IMU are referenced to the body frame, which is the coordinate system whose origin is located at the center of mass of the vehicle and whose orientation matches that of the vehicle. To compute the state of the vehicle, then, these values must be tranformed into the navigation frame, which is an earth-based coordinate system under flatness assumptions (a good simplification for local movement). The body and navigation frames are shown in Figure 1.

The coordinate transformation process is complicated, but a good discussion is given in Shin [3]. Eskin [1] gives an example simplified implementation. In addition to converting the measurements into the navigation frame, the accelerations and velocities must be integrated to compute the state values. A summary of the entire transformation/integration process is given diagrammatically by Shin, and is shown in Figure 2.

2 Component Selection

In order to design the INS hardware, we had to make choices about which components to use. There is no single correct way to construct an inertial system, and cost-performance tradeoffs must be considered.

2.1 Sensors

One of our primary goals for the INS was to use MEMS components. There are many choices for the particular MEMS sensors. We considered the following paramters before deciding on the particular sensors.

- **Number of Measurement Axes.** Many manufacturers have created two or three axis accelerometers and gyroscopes. These are tremendously useful as they reduce the required number of chips and supporting passive components, and eliminate misalignment errors.
- **Breakout Boards.** To eliminate the difficulty of soldering various packages and to speed up prototyping time, it is convenient for the sensors to come mounted on a breakout board. These boards also usually provide the necessary filtering capacitors.
- **Source Voltage.** The input voltage to the sensors must be within the range of our source, and it is most convenient if it is exactly equal to the source voltage. Our primary source voltage was 5V from a regulator. Many sensors require a nominal 3.3V or 3V input.
- **Type of Output.** MEMS sensors generally provide either digital or voltage proportional analog output. Digital outputs are further available in either PWM, SPI, or I^2C formats. The latter two formats were less desirable for us because they do not support arbitrary resolution.
- **Cost.** We wanted to keep costs as low as possible. One can pay hundreds of dollars for more accurate sensors, but these are out of our budget range.
- **Bandwidth.** Different sensors have different bandwidths. For moderate to very accurate systems, the bandwidth is an important factor to consider. However, for our project, this constraint was less important than the others.
- **Manufacturer.** Not all manufacturers provide the same quality products and datasheets.

After considering various products, we chose two breakout boards available from Sparkfun Electronics (TM), a hobbyist electronics depot. The first breakout board includes both a three axis accelerometer from Analog Devices (ADXL330) and a two axis gyroscope from InvenSense (IDG300). The board accepts a 3.3V input and produces five voltage proportional analog outputs. The board costs approximately $100. The second breakout board features a single axis gyroscope from Analog Devices (ADXRS300) which completes the orthogonal

set. The board accepts a 5V input and produces a voltage proportional analog output, at an approximate cost of $70.

2.2 Analog to Digital Converter

Having chosen sensors with analog output, we needed to be able to convert the signals to digital values. Most microcontrollers provide an analog to digital converter (ADC) for precisely this purpose. However, the resolution of these devices is usually 10 or 12 bits. We wanted higher precision than this. We decided on 16 bits because it fits conveniently in exactly two bytes, or two 8-bit ports on a microcontroller.

ADCs have similar parameters to those enumerated above. We eventually decided on the AD976 from Analog Devices which features 5V input, parallel output, and up to 100KSPS (kilosamples per second). Most high resolution ADCs use SPI or I^2C output to reduce the number of required microcontroller GPIO pins. We initially sampled such a component, but the interface turned out to be more hassle than it was worth. Parallel output can also be sampled more quickly than the serial alternatives, which is appropriate for slower processors.

2.3 Microcontroller

The workhorse processing unit for any embedded project is the microcontroller. An INS is particularly processing intensive application. At each timestep, the signals must be read and modified according to the current calibration. Then they must be fed into the state update method, which performs significant mathematical calculation, including several full matrix multiplications and trigonometric computations. For the greatest accuracy, all of these computation must be performed on 32 bit floating point values. Thus a fast, powerful microcontroller is ideal.

The vast array of available microcontrollers does include many 32 bit, 100+ MHz, floating point capable units. However, code complexity and development costs often scale proportionally to the power of the microcontroller. Furthermore, neither of the authors have had experience with these very powerful devices. For these reasons, we eventually decided to use a familiar, cheap, battle-tested, but less-powerful microcontroller: the 8 bit Atmel ATMega32. The ATMega32 is a 16MHz device with no hardware floating point. Thus, to allow real-time state update, an external floating point processor was necessary.

2.4 Floating Point

Compared to the wide variety of available microcontrollers, floating point coprocessors are relatively uncommon. However, the uM-FPU from Micromega is cheaply available and provides a rich set of IEEE 754 floating point math operations. In addition, the FPU features

macro operations such as matrix multiplication and FFT, and allows user programmable functions such that significant processing can be offloaded from the microcontroller.

2.5 False Starts and Future Changes

We tried several seemingly promising paths before settling on the hardware configuration given in the previous sections. These false starts are instructive enough that we will mention them here.

The first ADC that we tried to use was a 14 bit SPI device. In retrospect, there is probably nothing wrong with this choice aside from the slight speed decrease from the serial communication and some small hassles with filtering capacitors. However, at the time we had no experience with SPI devices and were simultaneously attempting to use an unfamiliar microcontroller platform. The result was that we had limited success performing conversions and communicating with the device. Thus we abandoned the SPI device in favor of a parallel interface.

We considered several microcontrollers before choosing the ATMega32. Concern about 32 bit processing and hardware floating point led us to experiment with two 32 bit micros. The first was a relatively new Atmel device, the AVR32UC3. A sample development board was available to us at no cost. Unfortunately, the chip was so new that at the time, one could not even purchase the device as a standalone IC. Documentation was also sparse and incomplete. So we tabled the chip. The second 32 bit micro we tried was the Phillips (now NXP) LCP2119 with ARM7 core. One of the authors had a development board for this device prior to starting research. The development board, however, was created by a foreign manufacturer, resulting in poor documentation and limited community support. Furthermore, the device was difficult to program and the development tools were not polished. So we gave up on this chip too.

Looking back, there are still many changes that we ought to make to the current design. Perhaps the easiest change with the biggest payoff would be to add an op amp to the output of the analog multiplexer before the signal enters the ADC. Although the ADC is sourced by only a nominal 5V, the valid range for the input voltage is allowed to range from -10V to +10V. That is, the output of the ADC attains its minimum value (-(2^{15}) in two's complement) when the input voltage is -10V and its maximum value (2^{15} in two's complement) when the input voltage is +10V. Since the peak range for most of our input signals was only 0V to +3.3V, we were using less than a quarter of the available range, thus wasting a full 2 bits of resolution. This problem could be partially alleviated by amplifying the input signal to a wider range. Another desirable change would be to replace the current gyroscopes with three matched gyroscopes with 3.3V source input. The discrepancy between the parameters for the dual-axis gyro and the single-axis gyro lead to very different performances. Finally, given more time we would have like to design a PCB using the bare sensor components instead of breakout boards to facilitate a more solid interface between the sensors and the

rest of the components.

3 Hardware Design

Once we had selected the components, designing the sensor PCB was straightforward. The sensor breakout boards were placed as closely as possible to each other and to the center of the PCB to minimize errors associated with shifted origins. The sensor outputs were connected as inputs to an analog multiplexer (ADG608), and the output of the multiplexer was passed to the ADC. Supporting resistors and capacitors were placed around the ADC as prescribed by the datasheet. 5V and 3.3V voltage regulators with appropriate filter capacitors served as the inputs to the sensor boards, the multiplexer, and the ADC. Finally, a barrel plug was added for the voltage source and a prototyping area was created for any quick fixes. The schematic and PCB layout are shown in the Appendices.

Rather than design a new prototype board for the ATMega32, we decided to borrow the well tested design due to Bruce Land [2]. The protoboard is small and provides access to each 8 bit port. In addition, serial communication from the UART is supported. The protoboard was mounted into a whiteboard and the appropriate connections were made to both the FPU and sensor board. A picture of the setup is given in Figure 3. Given more time, we would have like to mount the device in a project enclosure for easier handling and better protection against electrostatic discharge.

4 Software Design

Since we offloaded much of the computation to user functions on the FPU, the software design is logically divided into two pieces - the code on the microcontroller and the code on the FPU. The relationship between these pieces of code is summarized in Figure 4.

4.1 ATMega32 Code

The code running on the microcontroller is responsible for interfacing with the sensor board and the floating point unit. All state structures and update calculations are offloaded to the FPU. The code contains a single task running at $100Hz$ (using a millisecond timer interrupt triggered by timer 2) which is responsible for reading each of the six sensor values into memory. This is done by 1) selectings the appropriate multiplexer channel on the sensor board and then 2) driving the control lines of the ADC (the control sequence is replicated from the ADC data sheet) once for each sensor. Once we are done reading all of the sensor values we transfer the data to the appropriate registers of the FPU, in addition to the time interval since the last read (in our case a constant 0.01s), and then call a user function on the FPU to update the state of the system (position, velocity, and a matrix representing orientation) given the current accelerations and angular rates from the sensors.

The communication between the FPU and the microcontroller is done via SPI. Micromega provides a fully featured API which does the actual low level communication between the microcontroller and the FPU and provides convenient high level functions and named constants.

Because the state update calculations are all done on the FPU, the only time consuming code running on the microcontroller is the sensor readings, which include several delays and takes roughly 100 microseconds to run, and the data transfer to the FPU. This transfer involved moving seven 32 bit floating point numbers or 28 bytes of data plus the necessary control bytes (about 12 bytes) for a total of about 40 bytes plus a function call to update the state which is an extra two bytes. 42 bytes takes about $84\mu s$ assuming the SPI is running at the 5MHz continuous transfer speed. This means that we are well within our limit of $10ms$ between tasks.

Apart from reading the sensors values and transferring them to the FPU, the only other tasks that the microcontroller has is initializing the state of the FPU and INS at reset, which involves a call to an initialize function on the FPU, and to initialize the sensor board into a known state (the ADC requires one conversion operation before the outputs are determinate).

All code for the ATMega32 was written in AVR Studio 4 Version 4.13 Build 557 with GCC extensions provided by WinAVR 20070525.

4.2 FPU Code

The Micromega FPU provides a wealth of functionality to speed up the calculations required by this project. It allows us to do full 32 bit IEEE 754 floating point calculations much faster than on the ATMega32 and to offload significant processing to ensure that the microcontroller operates in real time and controls the system correctly. The FPU runs at 29.48MHz, nearly twice the speed of the microcontroller (16MHz). The accuracy gained from computing with floating point instead of fixed point is arguably negligible considering the accuracy of our sensors, but was a nice addition and gave us experience interfacing with a coprocessor.

We also took advantage of the Integrated Development Environment provided by Micromega to compile some of our C code math statements into FPU assembly and to provide a framework for writing our own assembly. One of the key features in debugging the FPU code was the debug interface between the serial pins on the chip and this program.

We also made extensive use of the FPU's hardware support for matrix operations. The state update routine uses several matrices and vectors to convert between reference frames and it was very convenient to be able to code these with few instructions. Initialization is also made easier by the matrix operation that does an element-wise set (we initialized many vectors and matrices to zero at reset). Helpful math routines such as those for cosine and sine were

also provided by the FPU.

The FPU code is a direct translation of code written in C during the early stages of the project. The C code was written for the PC and tested for correctness. The mathematics in the code relies heavily the papers by Shin and Eskin. Small changes were made for the FPU version to take advantage of the available floating point registers.

According to on-chip timing, the total state update on the FPU takes less than $4ms$. Although this is a significant amount of time, we are afforded a window of $10ms$ to do this computation, so we are well within our time allowance.

5 Conversions and Calibration

In order to perform the state update calculations, the sensor readings must be converted into quantities with the appropriate units, namely meters per second squared for accelerations and radians per second for angular velocities. The output from the ADC, however, is simply an integer between 0 and 2^{16}. The conversion formula from the ADC reading to the physical quantity is given by the affine map

$$p = \alpha\left(q - \beta\right) + \epsilon \tag{1}$$

where q is the value from the ADC, β is the *bias*, α is the *scale factor* or *gain*, ϵ is additive noise, and p is the resulting physical quantity. There are several ways to find these values, which we now discuss.

5.1 Theoretical Biases and Gains

The gains and biases can be found theoretically from the datasheet specifications. The biases can be computed from the zero level output voltage (or *offset voltage*) of the devices. The gains, on the other hand are functions of the devices' sensitivites. The datasheet for the accelerometers reports the expected sensitivity in terms of millivolts of change in the output voltage per g of acceleration. The gyroscope sensitivities are reported in millivolts of change in the voltage per degree/s. From these we can compute the ideal biases and gains in terms of ADC levels as shown in Table 1.

Sensor	**Zero Offset**	**Ideal Bias**	**Sensitivity**	**(Ideal Gain)$^{-1}$**
x/y/z Accelerometer	1.65V	5406.72	300 $\frac{mV}{g}$	100.27746 $\frac{\text{levels}}{m/s^2}$
x/y Gyroscope	1.65V	5406.72	2 $\frac{mV}{^\circ/s}$	375.49362 $\frac{\text{levels}}{rad/s}$
z Gyroscope	2.5V	8192	5 $\frac{mV}{^\circ/s}$	938.73405 $\frac{\text{levels}}{rad/s}$

Table 1: Ideal biases and gains for the three sensor packages

Sensor	Experimental Bias	$(\text{Experimental Gain})^{-1}$
x Acceleration	5440	113.9526 $\frac{\text{levels}}{m/s^2}$
y Acceleration	5481.8	111.1882 $\frac{\text{levels}}{m/s^2}$
z Acceleration	4824.5	71.6598 $\frac{\text{levels}}{m/s^2}$
x Gyroscope	4862.6	429.6229 $\frac{\text{levels}}{rad/s}$
y Gyroscope	4863.7	389.2020 $\frac{\text{levels}}{rad/s}$
z Gyroscope	8223.9	828.2950 $\frac{\text{levels}}{rad/s}$

Table 2: Experimental biases and gains for the six sensors

5.2 Experimental Calibration via Constant Sources

Of course, while the ideal biases and gains above are useful starting points, the sensors will not adhere to theoretical values. Therefore, it is necessary to perform calibration to find experimental values for the parameters. The calibration of the accelerometers can be performed by logging the output of the sensors when the IMU is positioned in each of the orientations shown in Figure 5.

In each of the orientations, we make use of gravity as a fixed acceleration. The reading from a sensor when its axis is completely orthogonal to gravity should correspond to the sensor bias. And by finding the magnitude of the difference between the maximum and minimum readings of the sensor, we can compute the scale factor. This assumes that there are no other appreciable forces imposed on the sensor, so we must try to restrict any motion to purely rotational while logging. The output from such a log are shown in Figure 6, and the resulting experimental biases and gains are listed in Table 2.

5.3 Experimental Calibration via Integration

Finding the biases for the gyroscopes can be done simply by logging the sensor output for the stationary device. The analogous method for finding the gains for the gyroscopes, however, would require that we be able to rotate the sensor about each of the orthogonal axes at a known, constant rate. Unfortunately, there is no freely available constant rotation source analogous to the freely available constant acceleration due to gravity. We could try to rotate the sensor by hand, but we would certainly not acheive a constant rate. So the gyroscope gains cannot be found using this approach.

An alternative method for calibrating the gains of the sensors makes use of the linearity of integration. Assuming we have already computed the sensors biases and we ignore noise, Equation 1 simply becomes $p = \alpha q$. If p is an angular velocity, $p = \theta'$, then $\Delta\theta = \int \alpha q \, dt$, and we can find α using

$$\alpha = \frac{\Delta\theta}{\int q \, dt}$$

If p is a linear acceleration, $p = x''$, then $\Delta x = \iint \alpha q \, dt$, which we solve for α to get

$$\alpha = \frac{\Delta x}{\iint q \, dt}$$

Therefore, to find the scale factors, we can log the sensor data while the sensor is moved a known distance, say $1m$, or rotated by through a known angle, say $90°$, along or about a single axis. We can then find the gain by dividing the known distance or angle by the sum of the logged measurements (or in the case of an acceleration, the sum of the cumulative sum of the measurements). This process is not as accurate as the method above, so we only used it to find the gyro gains. The results are listed in Table 2.

6 Results

To ascertain the success of the device as an inertial navigation system, we conducted a number of experiments. The most informative results are given here.

6.1 Experiment 1 - Stationary Device

The first experiment one should carry out when testing an INS is to log the computed position for the stationary device. This test is useful for estimating the drift error rate. A log of the three position coordinates in the navigation frame for the first 2s after boot up is given in Figure 8 [x=blue, y=green, z=red].

These results are somewhat encouraging. We see that after 2s, all of the coordinate positions has drifted by less than 5m. And in the first second, the drift is less than 1m. Since GPS update rates are approximately 1s, this means that if we were to add a GPS unit and Kalman filtering to the INS, the stationary accuracy would deviate by only about 1m.

This experiment also serves as a sanity check that the basic integration algorithm is working correctly. Each of the coordinate positions follows a quadratic curve, which is exactly what should happen under constant acceleration. From this we deduce that at each timestep, the sensor measurement differs from the bias value by a small amount, leading to compounded velocity, and a quadratic drift.

6.2 Experiment 2 - Single Axis Oscillation

The stationary results just reported could have been coincidental. Perhaps the sensors aren't really working and we are just sending a constant value to the update algorithm at each timestep. This would produce the same results and we would be none the wiser. Therefore, we performed an experiment to rule out this hypothesis. While again logging the reported

positions, we moved the device up and down along the z-axis in a roughly sinusoidal pattern with amplitude approximately equal to 5cm. The results of this log are given in Figure 9.

We see that the estimated z-position (the red line) appears to be a superposition of a quadratic with a sinusoidal oscillation. Moreover, the oscillation appears to have approximately 5cm amplitude. We interpret this result as showing the actual oscillations superimposed on the drift error. So we are convinced that the state update did pick up on the movement.

6.3 Experiment 3 - Prerotation vs. Postrotation

Having established that the accelerometers are truly affecting the state estimate, we performed a final test to see if the gyroscopes and rotational update calculations were working. The test makes use of the following fact: the state update algorithm assumes that, at initialization, the sensors are oriented such that z-accel points directly up (opposite gravity). The experiment works by first logging the positions when the sensor is rotated out of this assumed initial orientation before the system is booted up (specifically we rotated it upside down). We expect in this case that the system will think that the sensor is being accelerated strongly in the +z-direction (because gravity is acting opposite from the assumed orientation). We then create another position log where we first boot up the system, and then rotate the sensor as before. We expect in this case that the state update will take into account the angular velocities reported by the gyros and thus will realize that the sensor has been rotated. If this works, we should not see the same large positional displacements as before. Ideally, the INS will report that the sensor has not moved at all. The actual results are shown in Figure 10.

As expected, when the device is prerotated, the z-position grows very quickly, reaching nearly 200m in 5s. This represents the effect of opposite-from-normal gravity plus the usual bias drift error. When the rotation is performed after boot-up, we see that, although not nearly stationary, the INS does not think that it has moved as far in the z-direction. It now reports a displacement of closer to 70m in the +z-direction, which can be almost fully explained by drift. So it seems that the gyroscopes and rotational update calculations are working.

7 Conclusions and Future Work

Our initial goals at the inception of the project were very ambitious. Needless to say, we did not meet all of them. We were, however, able to make a sensor system that performs the necessary measurements for inertial navigation, and does significant calculation on those values to obtain new state (all at a rate of one hundred times per second). The code to update the state of the system appears to be working correctly on our floating point unit (as can be seen in the results) and with more robust sensor calibration and error correction and perhaps faster sampling time and sensor filtering, we feel that the system could perform

at a level consistent with other low cost dead reckoning systems. Some improvements and possible future work are detailed below.

7.1 Sensor Reading and Calibration Improvements

Our sensor calibration, while extensive, could certainly have been improved. The system could have performed self-calibration at start up to ensure more accurate biases and scale factors by computing time averages and taking advantage of the accelerometers self test feature. In addition, a temperature sensor could have been used in conjunction with the sensors' stated skew factors to dynamically update scale factors and biases during runtime. The sensor reading method could have also been improved. We neglected to do software filtering of our sensor readings in the interest of time but feel that this could have provided some noise immunity for our system. The sensor readings have occasional spikes which may cause the state to drift more than necessary. Some improvement may have been derived by increasing the sampling rate and averaging consecutive samples, as described by Eskin [1]. A simple Guassian filter may also have reduced the sensor noise.

The 100Hz sampling frequency may have been one of the biggest weaknesses in our design. We sample at a much lower frequency than related systems and we sample all of our sensors at once. While the sampling frequency was limited by the speed of our state update code, we feel that with a better understanding of the sampling frequency requirements of the sensors, and with code optimizations (and perhaps hardware improvements), we could have increased the sampling frequency and possibly the accuracy of our system.

7.2 GPS Integration and Kalman Filtering

The method of state update that we employed falls in the class of dead reckoning integration. That is, there is no feedback of position, velocity, or orientation error. Such a technique falls victim to integration drift and compounding errors. To alleviate these shortcomings, advanced navigation systems generally integrate a GPS unit. GPS devices have limited accuracy but are not vulnerable to drift. Thus the combination of a GPS unit with an inertial measurement unit can be quite powerful, allowing the IMU to provide improved resolution while avoiding drift.

The mathematical mechinism for the feedback control is the Kalman filter, which combines multiple estimates of a signal to reduce the expected error from the true value. The theory of Kalman filtering is well established and has been successfully applied to INS design. We had hoped to explore this integration, but time did not permit. On a related note, we had originally intended to incorporate duplicate MEMS sensors along each axis. These extra measurements could be utilized by the Kalman filter to further reduce error.

7.3 Remote Monitoring

Another initial goal was to allow the INS to be monitored remotely via wireless communication. This would have allowed the INS to be mounted on a flight vehicle while maintaining contact with a base station. The addition of this component to the project would not, in principle, be very difficult, and would have been our next step. Again, time ran out.

7.4 Autonomous Control

Ultimately, our goal was to use the INS as a platform for autonomous control. We planned to purchase a remote control helicopter and develop real time autopilot algorithms. Helicopters are notoriously difficult to fly and require constant adjustment from the pilot. The hope was that a system could be developed which would remove the complex human control requirements and provide a simple interface for waypoint navigation. We realize now that this goal was far beyond our reach from the beginning, even if we had not encountered any of the problems that we did. Nevertheless, the goal is attainable, and we still maintain an interest in this application.

Appendix - Some Lessons Learned

In any self-guided independent study, it is inevitable that difficulties will be encountered that were not expected. However, the frequency and magnitude of these difficulties can be minimized. Here are some lessons we learned and would like to pass on to future students performing an independent study. Some of these suggestions are specific to electrical engineering, but many apply to any field.

1. **Set Realistic Goals.** Our original goal was to build an inertial navigation system incorporating MEMS sensors, GPS, and wireless communication and use it to fly a scale helicopter. In hindsight, that goal was absurdly optimistic. Realize that you have other demanding classes and that things will go wrong and set your goals accordingly.

2. **Plan Ahead.** It is tempting to dive in right away building your device or programming. While this is fine for a prototype, the final product requires copious planning. For example, we designed a board an 8x2 parallel digital output without considering how this would get connected to our microcontroller. That decision led to hours of frustration which could have been avoided by a different layout.

3. **Read All Datasheets.** You won't know to pull the start conversion bit high on the ADC to enable the output pins if you don't read the datasheet. That was at least eight hours wasted. Read the whole datasheet - just do it. Pay special attention to the timing diagrams and reference circuits. They are there to help you.

4. **Start Simple.** In your head, you know how the whole symphony will play out. But when you end up with a cacophony, don't start blindly telling the instruments to do different things. Focus on a single component, make sure it is correct, and move on to the next. It's simple math - if you have six components working together, there are $2^6 = 64$ ways it could be failing, and if you change something and it doesn't fix the problem, you haven't narrowed anything down. If you focus on a single component, it either works or it doesn't, and it's usually easy to figure out which.

5. **Allocate a 'Class' Time.** Many independent studies don't have a weekly meeting time. This makes it easy to go weeks without working. So set a time each week when you have to be working and work some accountability into it, so you can't get away with not showing up.

6. **Use One Power Supply and Connect All Grounds.** Voltage is always referenced. Make everything in your circuit referenced to the same ground. If you don't, strange things will happen and you won't know why. I promise.

7. **Include a Power Switch.** It's always useful to be able to turn off your device. The best way to do this is NOT to pull out the power each time. Add a switch and save some tears.

8. **Minimize Wires.** Every wire is an opportunity for a faulty connection. So try to minimize the number and length of all wires. A good way to do this is to design a PCB and get it fabricated.

9. **Don't be Afraid to Spend Money.** If you have funding and spending money will make a problem go away, spend the money. We made a PCB but found out that some of the holes were too small and didn't fit nice headers. So we ordered a new one with bigger holes. 50 bucks - totally worth it.

10. **Remember the Rule of Pi.** Be pessimistic when estimating the time to get something done or working. A good rule of thumb is to guess how long something will take if everything you can think of goes wrong. Then multiply by pi.

Appendix - Code

Mega32 Code:

```
#include <avr/io.h>
#include <stdio.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "fpu.h"

#define inb(sfr)          _SFR_BYTE(sfr)
#define cbi(sfr,bit)     (_SFR_BYTE(sfr) &= ~_BV(bit))
#define sbi(sfr,bit)     (_SFR_BYTE(sfr) |= _BV(bit))

//ADC data read
#define ADCData (((unsigned int)_SFR_BYTE(PINA)) << 8) | ((unsigned int)_SFR_BYTE(PINC))

//Blinky task variables
#define BLINK_TIME 250
unsigned char volatile blink_count;
unsigned char toggle;

//INS Sensor stuff (sample time gives 100 samples per second)
#define SAMPLE_TIME 10
unsigned char volatile sample_count;
unsigned int cur_x_accel, cur_y_accel, cur_z_accel;
unsigned int cur_x_rate, cur_y_rate, cur_z_rate;
float dtheta[3];
float dv[3];

//FPU Functions
#define ins_initialize   5
#define ins_update_state 6

//Initialization
void initialize(void);
void blink_task(void);
void sample_task(void);
unsigned int readtask(unsigned char channel);

/*
 * Timer interrupt code for state update
 * Running at 1000 Hz
 */
ISR(SIG_OUTPUT_COMPARE2)
{
  if (sample_count > 0) sample_count--;
  if (blink_count > 0) blink_count--;
}

int main(void)
{
  initialize();

  while(1)
  {
    if (sample_count == 0) sample_task();
    if (blink_count == 0)
{
  toggle = !toggle;
  if (toggle) blink_task();
  else blink_count = BLINK_TIME;
    }
  }

  return 0;
}
```

```
/*
 * Blink a light and also print our current state.
 */
void blink_task(void)
{
  blink_count = BLINK_TIME;

  if (PIND & 0x04) cbi(PORTD,2);
  else             sbi(PORTD,2);
}

void sample_task(void)
{
  sample_count = SAMPLE_TIME;

  //Sample the 6 sensors
  cur_x_accel = readtask(0);
  _delay_us(10);
  cur_y_accel = readtask(1);
  _delay_us(10);
  cur_z_accel = readtask(2);
  _delay_us(10);
  cur_x_rate  = readtask(3);
  _delay_us(10);
  cur_y_rate  = readtask(4);
  _delay_us(10);
  cur_z_rate  = readtask(5);
  _delay_us(10);

  //Put these values in arrays for block transfer
  dv[0] = (float)cur_x_accel;
  dv[1] = (float)cur_y_accel;
  dv[2] = (float)cur_z_accel;
  dtheta[0] = (float)cur_x_rate;
  dtheta[1] = (float)cur_y_rate;
  dtheta[2] = (float)cur_z_rate;

  //Block transfer sensor readings to fpu
  fpu_write2(SELECTX,80);
  fpu_wrblk(3,dv);
  fpu_write2(SELECTX,71);
  fpu_wrblk(3,dtheta);
  fpu_writeFloatReg(0.01,83);    //Register number of dt
  fpu_wait();

  //Now update the INS state
  fpu_fcall(ins_update_state);
  fpu_wait();
}

// channel must be 0-7
unsigned int readtask(unsigned char channel)
{
  if (channel & 0x04) sbi(PORTD,5); else cbi(PORTD,5);
  if (channel & 0x02) sbi(PORTD,4); else cbi(PORTD,4);
  if (channel & 0x01) sbi(PORTD,3); else cbi(PORTD,3);
  _delay_us(5);                     // wait for analog mux to settle
  cbi(PORTD,6);                     // start conversion
  _delay_us(1);                     // wait for the conversion to start
  sbi(PORTD,6);                     // pull back high
  loop_until_bit_is_set(PIND,7);   // spin on busy bit

  return ADCData;
}

void initialize(void)
{
```

```
  // reset FPU and check synchronization
  if (fpu_reset() == SYNC_CHAR)
  {
  }
  else
  {
    return;
  }

  //Init timer0 for millisecond time base
  TCNT2 = 0; //clear the timer0 register
  OCR2  = 249; //set the timer0 compare value
  TCCR2 = _BV(WGM21) |
          _BV(CS22);  //CTC on OCR0, clock/64
  TIMSK = _BV(OCIE2); //enable the timer0 interrupt

  //Initialize ADC interface ports
  DDRA = 0x00;
  DDRC = 0x00;
  cbi(DDRD,7);
  sbi(DDRD,6);
  sbi(DDRD,5);
  sbi(DDRD,4);
  sbi(DDRD,3);
  sbi(PORTD,6);

  //Initialize blinky port
  sbi(DDRD,2);
  sbi(PORTD,2);
  blink_count = BLINK_TIME;
  toggle = 0;

  //Initialize sensor sampling
  sample_count = SAMPLE_TIME;

  //Start the ISRs
  sei();

  //Set up the INS state
  fpu_fcall(ins_initialize);
  fpu_wait();

  readtask(0);
}
```

FPU Code:

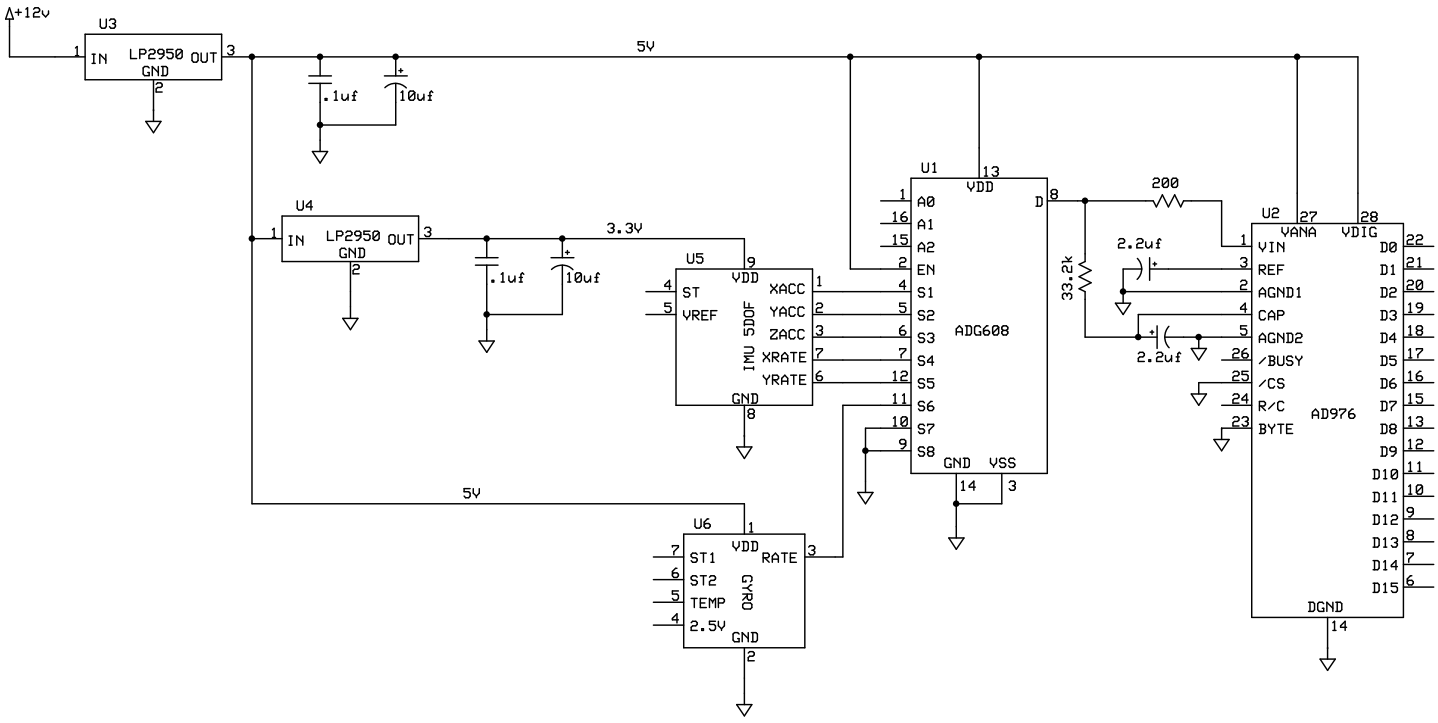
```
'This is the current state of the system
rx      equ F1
ry      equ F2
rz      equ F3
vx      equ F4
vy      equ F5
vz      equ F6
Cbn_0_0 equ F7
Cbn_0_1 equ F8
Cbn_0_2 equ F9
Cbn_1_0 equ F10
Cbn_1_1 equ F11
Cbn_1_2 equ F12
Cbn_2_0 equ F13
Cbn_2_1 equ F14
Cbn_2_2 equ F15
q_0     equ F16
q_1     equ F17
q_2     equ F18
q_3     equ F19
```

```
'Temporary and intermediate variables
qum_0_0 equ F20
qum_0_1 equ F21
qum_0_2 equ F22
qum_0_3 equ F23
qum_1_0 equ F24
qum_1_1 equ F25
qum_1_2 equ F26
qum_1_3 equ F27
qum_2_0 equ F28
qum_2_1 equ F29
qum_2_2 equ F30
qum_2_3 equ F31
qum_3_0 equ F32
qum_3_1 equ F33
qum_3_2 equ F34
qum_3_3 equ F35
scul_0_0 equ F37
scul_0_1 equ F38
scul_0_2 equ F39
scul_1_0 equ F40
scul_1_1 equ F41
scul_1_2 equ F42
scul_2_0 equ F43
scul_2_1 equ F44
scul_2_2 equ F45
pq_0 equ F46
pq_1 equ F47
pq_2 equ F48
pq_3 equ F49
pvn_0 equ F50
pvn_1 equ F51
pvn_2 equ F52
t3_1_0 equ F53
t3_1_1 equ F54
t3_1_2 equ F55
t3_2_0 equ F56
t3_2_1 equ F57
t3_2_2 equ F58
euler_0 equ F59
euler_1 equ F60
euler_2 equ F61
dtheta_0 equ F71
dtheta_1 equ F72
dtheta_2 equ F73
mag equ F74
s equ F75
c equ F76
sdx equ F77
sdy equ F78
sdz equ F79
dv_0 equ F80
dv_1 equ F81
dv_2 equ F82
dt equ F83
q00 equ F84
q11 equ F85
q22 equ F86
q33 equ F87
q01 equ F88
q02 equ F89
q03 equ F90
q12 equ F91
q13 equ F92
q23 equ F93

#FUNCTION 0 euler_angles_to_body_to_nav_matrix
```

```
Cbn_0_0 = cos(euler_1)*cos(euler_2)
Cbn_0_1 = (sin(euler_0)*sin(euler_1)*cos(euler_2)) - (cos(euler_0)*sin(euler_2))
Cbn_0_2 = (sin(euler_0)*sin(euler_2)) + (cos(euler_0)*sin(euler_1)*cos(euler_2))
Cbn_1_0 = cos(euler_1)*sin(euler_2)
Cbn_1_1 = (cos(euler_0)*cos(euler_2)) + (sin(euler_0)*sin(euler_1)*sin(euler_2))
Cbn_1_2 = (cos(euler_0)*sin(euler_1)*sin(euler_2)) - (sin(euler_0)*cos(euler_2))
Cbn_2_0 = -sin(euler_1)
Cbn_2_1 = sin(euler_0)*cos(euler_1)
Cbn_2_2 = cos(euler_0)*cos(euler_1)

#FUNCTION 1 body_to_nav_matrix_to_quaternion
mag = sqrt(1 + Cbn_0_0 + Cbn_1_1 + Cbn_2_2)
q_0 = 0.5 * (Cbn_2_1 - Cbn_1_2) / mag
q_1 = 0.5 * (Cbn_0_2 - Cbn_2_0) / mag
q_2 = 0.5 * (Cbn_1_0 - Cbn_0_1) / mag
q_3 = 0.5 * mag

#FUNCTION 2 sculling_matrix
scul_0_0 = 1
scul_1_1 = 1
scul_2_2 = 1
scul_0_1 = 0.5 * dtheta_2
scul_0_2 = -0.5 * dtheta_1
scul_1_0 = -0.5 * dtheta_2
scul_1_2 = 0.5 * dtheta_0
scul_2_0 = 0.5 * dtheta_1
scul_2_1 = -0.5 * dtheta_0

#FUNCTION 3 quaternion_update_matrix_eskin
mag = sqrt((dtheta_0*dtheta_0) + (dtheta_1*dtheta_1) + (dtheta_2*dtheta_2))
#ASM
    SELECTA, 74        ;Select mag
    FCMPI,   0         ;Check if mag is 0
    BRA, EQ, _szero
    COPYA,   75        ;Copy mag to s
    SELECTA, 75        ;Select s
    FDIVI,   2         ;Divide by 2
    SIN                ;Take the sine
    FDIV,    74        ;Divide the result by mag
    BRA,     _exit     ;Done
_szero:
    SELECTA, 75        ;Select s
    FSETI,   0         ;Set s = 0
_exit:
#ENDASM
c = cos(mag/2)
sdx = s*dtheta_0
sdy = s*dtheta_1
sdz = s*dtheta_2
qum_0_0 = c
qum_0_1 = sdz
qum_0_2 = -sdy
qum_0_3 = sdx
qum_1_0 = -sdz
qum_1_1 = c
qum_1_2 = sdx
qum_1_3 = sdy
qum_2_0 = sdy
qum_2_1 = -sdx
qum_2_2 = c
qum_2_3 = sdz
qum_3_0 = -sdx
qum_3_1 = -sdy
qum_3_2 = -sdz
qum_3_3 = c

#FUNCTION 4 quaternion_to_body_to_nav_matrix_shin
q00 = q_0*q_0
```

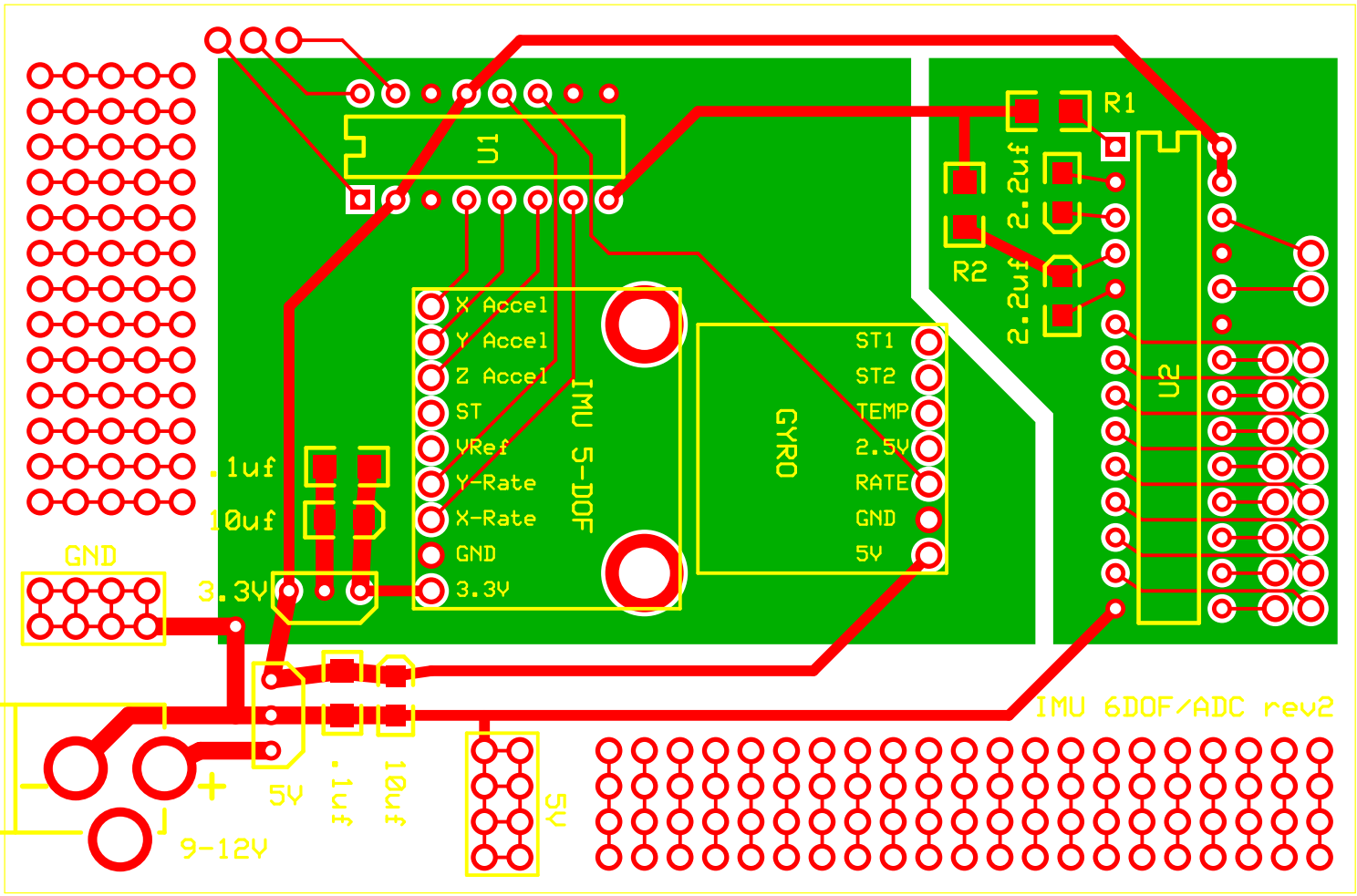
```
q11 = q_1*q_1
q22 = q_2*q_2
q33 = q_3*q_3
q01 = q_0*q_1
q02 = q_0*q_2
q03 = q_0*q_3
q12 = q_1*q_2
q13 = q_1*q_3
q23 = q_2*q_3
Cbn_0_0 = q00-q11-q22+q33
Cbn_0_1 = 2*(q01-q23)
Cbn_0_2 = 2*(q02-q13)
Cbn_1_0 = 2*(q01+q23)
Cbn_1_1 = q11-q00-q22+q33
Cbn_1_2 = 2*(q12-q03)
Cbn_2_0 = 2*(q02-q13)
Cbn_2_1 = 2*(q12+q03)
Cbn_2_2 = q22-q00-q11+q33

#FUNCTION 5 initialize
#ASM
    FWRITE0,0.0       ;Reg[0] = 0
    SELECTMA,20,4,4   ;A = qum
    MOP,0             ;A = 0  element-wise set
    SELECTMA,71,3,1   ;A = dtheta
    MOP,0             ;A = 0  element-wise set
#ENDASM
@sculling_matrix
#ASM
    FWRITE0,0.0        ;Reg[0] = 0
    SELECTMA,46,4,1    ;A = pq
    MOP,0              ;A = 0  element-wise set
    SELECTMA,50,3,1    ;A = pvn
    MOP,0              ;A = 0  element-wise set
    SELECTMA,53,3,1    ;A = t3_1
    MOP,0              ;A = 0  element-wise set
    SELECTMA,56,3,1    ;A = t3_2
    MOP,0              ;A = 0  element-wise set
    SELECTMA,59,3,1    ;A = euler
    MOP,0              ;A = 0  element-wise set
    SELECTMA,1,3,1     ;A = rn
    MOP,0              ;A = 0  element-wise set
    SELECTMA,4,3,1     ;A = vn
    MOP,0              ;A = 0  element-wise set
#ENDASM
@euler_angles_to_body_to_nav_matrix
@body_to_nav_matrix_to_quaternion

#FUNCTION 6 update_state
#ASM
    TRACEREG,1
    TRACEREG,2
    TRACEREG,3
#ENDASM

'Convert the accelerations and angular velocities (scale and bias)
dtheta_0 = (dtheta_0 - 4850.1)*0.0026632
dtheta_1 = (dtheta_1 - 4760.6)*0.0026632
dtheta_2 = (dtheta_2 - 8219.5)*0.0010653
dv_0 = (dv_0 - 5444.0)*0.0089039
dv_1 = (dv_1 - 5505.8)*0.0089609
dv_2 = (dv_2 - 5146.9)*0.0094717

'Get delta angle and delta velocity
#ASM
    LOAD,83           ;Reg[0]=dt
    SELECTMA,71,3,1   ;A = dtheta
    MOP,4             ;A = A * dt
```

```
    SELECTMA,80,3,1  ;A = dv
    MOP,4            ;A = A * dt
#ENDASM

'Update Quaternion using Eskin's method
@quaternion_update_matrix_eskin
#ASM
    SELECTMA,16,4,1  ;Select 4x1 q vector as A
    SELECTMB,20,4,4  ;Select 4x4 qum matrix as B
    SELECTMC,46,4,1  ;Select 4x1 pq vector as C
    MOP,26           ;pq = q (copy)
    MOP,16           ;A = BC
#ENDASM

'Update body-to-navigation matrix
@quaternion_to_body_to_nav_matrix_shin

'Update sculling matrix
@sculling_matrix

'Convert dv from body frame to navigation frame
#ASM
    SELECTMA,53,3,1  ;Select 3x1 t3_1 vector as A
    SELECTMB,37,3,3  ;Select 3x3 scul matrix as B
    SELECTMC,80,3,1  ;Select 3x1 dv vector as C
    MOP,16           ;A = BC
    SELECTMA,80,3,1  ;Select 3x1 dv vector as A
    SELECTMB,7,3,3   ;Select 3x3 Cbn matrix as B
    SELECTMC,53,3,1  ;Select 3x1 t3_1 vector as C
    MOP,16           ;A = BC
#ENDASM

'Add gravity from this
dv_2 = dv_2 + (9.80321 * dt)

'Update velocity
#ASM
    SELECTMA,4,3,1   ;Select 3x1 vn vector as A
    SELECTMB,80,3,1  ;Select 3x1 dv vector as B
    SELECTMC,50,3,1  ;Select 3x1 pvn vector as C
    MOP,26           ;pvn = vn  (copy)
    MOP,9            ;A = A+B  (element-wise add)
#ENDASM

'Update position
#ASM
    SELECTMA,53,3,1  ;Select 3x1 t3_1 vector as A
    SELECTMB,50,3,1  ;Select 3x1 pvn vector as B
    SELECTMC,4,3,1   ;Select 3x1 vn vector as C
    MOP,29           ;t3_1 = vn
    MOP,9            ;A = A+B
    FWRITE0,0.5      ;Reg[0] = 0.5
    MOP,4            ;A = A*0.5
    LOAD,83          ;Reg[0] = dt
    MOP,4            ;A = A*dt
    SELECTMA,56,3,1  ;Select 3x1 t3_2 vector as A
    SELECTMB,1,3,1   ;Select 3x1 rn vector as B
    SELECTMC,53,3,1  ;Select 3x1 t3_1 vector as C
    MOP,29           ;t3_2 = t3_1
    MOP,9            ;A = A+B
    MOP,25           ;rn = t3_2
#ENDASM

#END
```



Cornell University		
IMU 6DOF with ADC		
Jordan	Rev 1.0 10/12/2007	Page # or name

C:\Documents and Settings\Jordan\My Documents\cornell\heli\schematics\IMU 6DOF with ADC.sch - Sheet1



C:\Documents and Settings\Jordan\My Documents\cornell\heli\schematics\IMU 6DOF with ADC rev2.pcb (Silkscreen, Top layer, Bottom layer)

References

[1] Maksim Eskin. Design of an inertial navigation unit using mems sensors. Master's thesis, Cornell University, 2006.

[2] Bruce Land. Atmega32 protoboard. *http://www.nbb.cornell.edu/neurobio/land/PROJECTS/Protoboar*

[3] Eun-Hwan Shin. *Accuarcy improvement of low cost INS/GPS for land applications*. PhD thesis, Geomatics Engineering, University of Calgary, 2001.

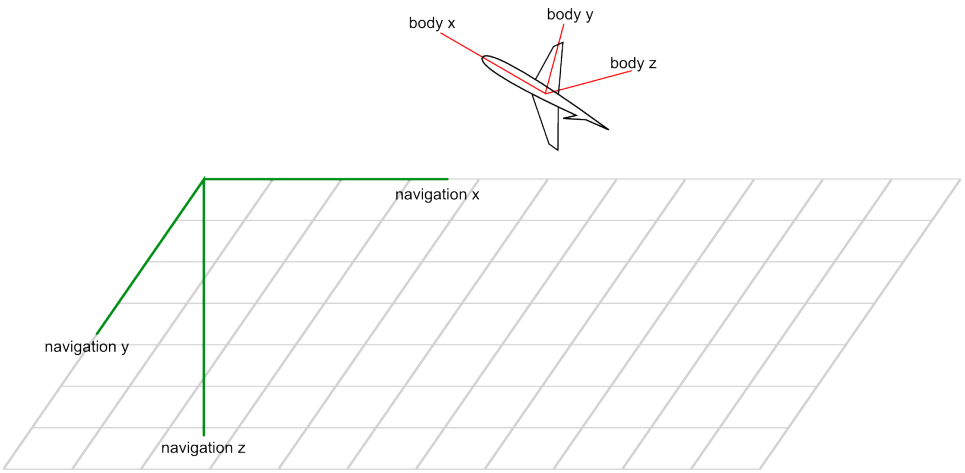


Figure 1: Body and Navigation frames

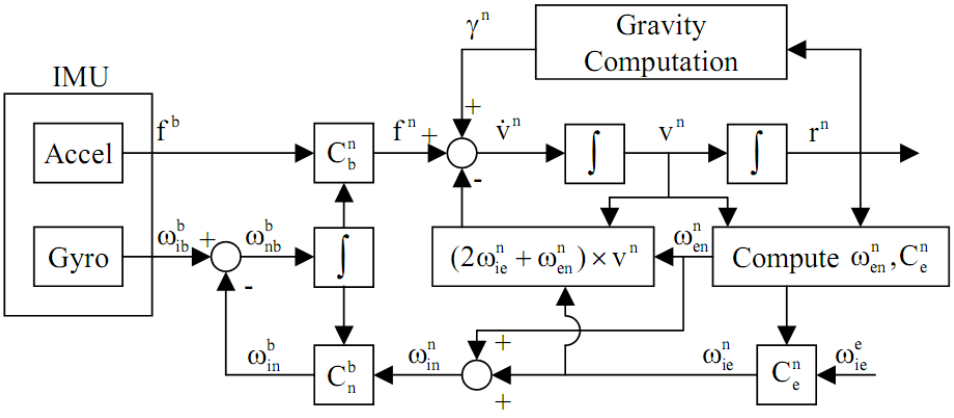


Figure 2: State update network [credit: Shin[3]]

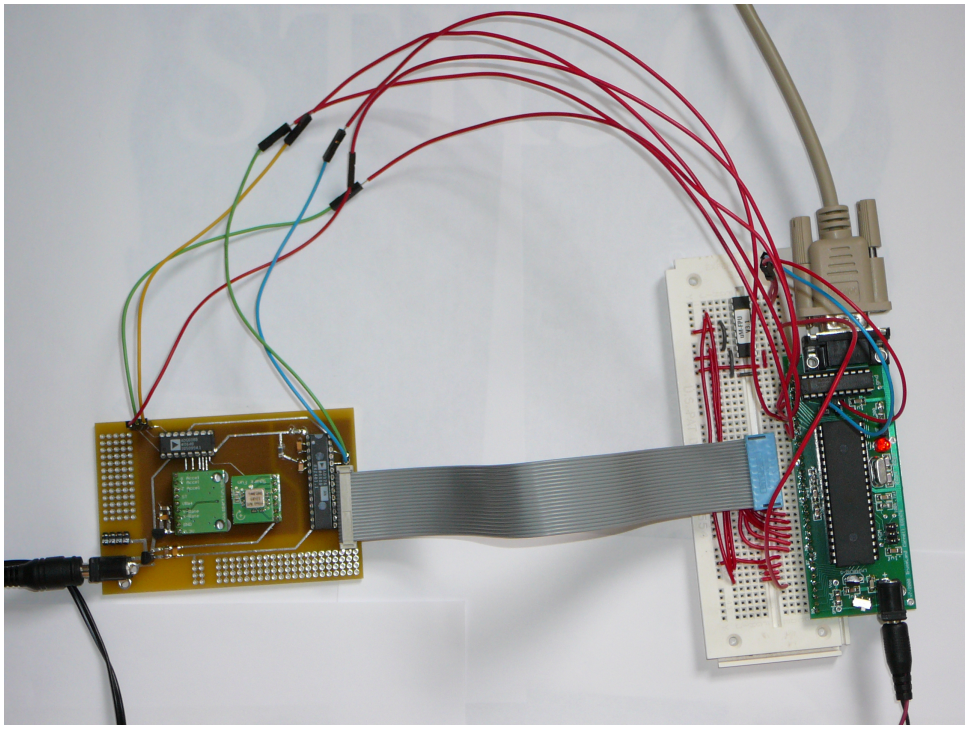


Figure 3: Hardware Setup

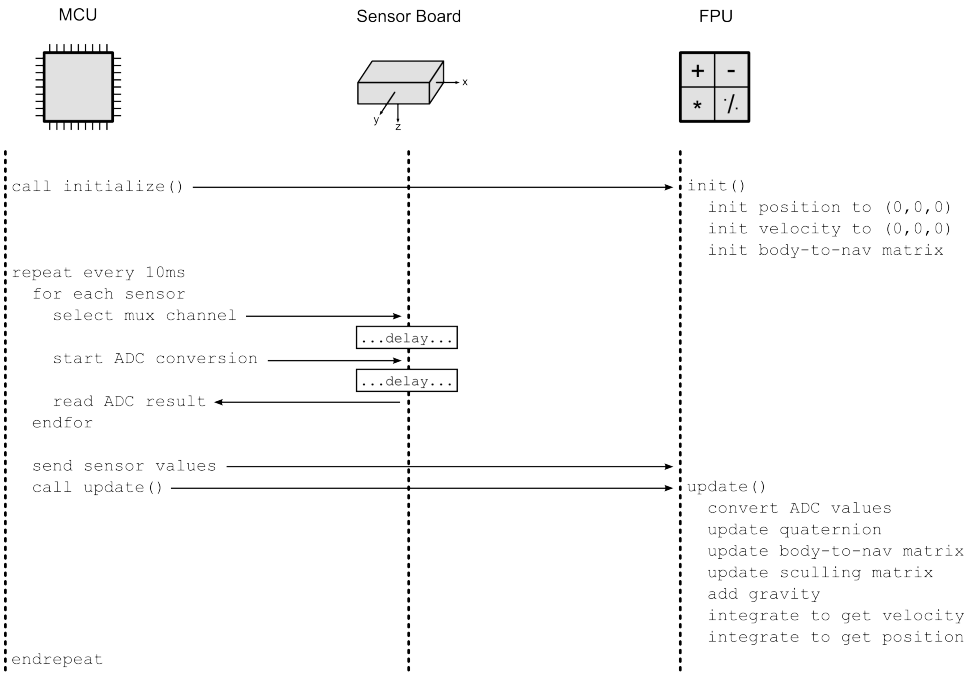


Figure 4: Code summary flow

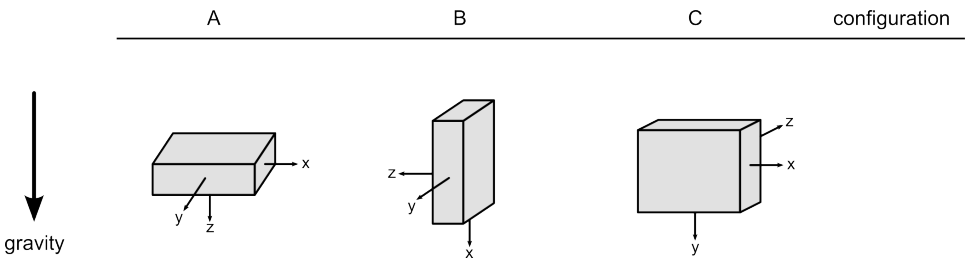


Figure 5: Setup for accelerometer calibration

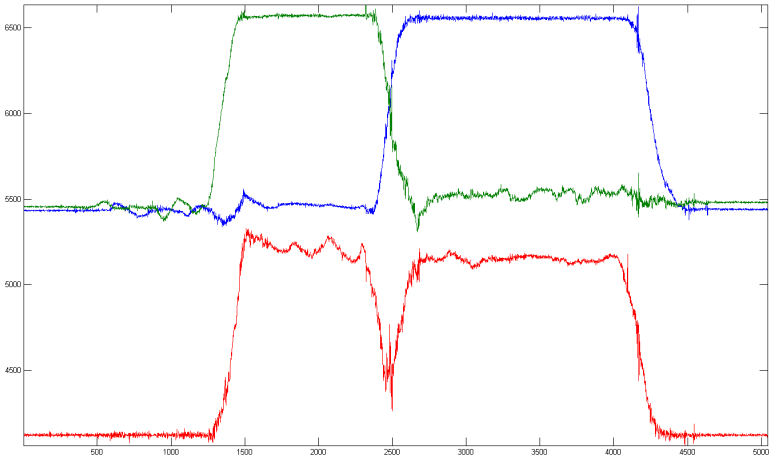


Figure 6: Calibration data for the accelerometers

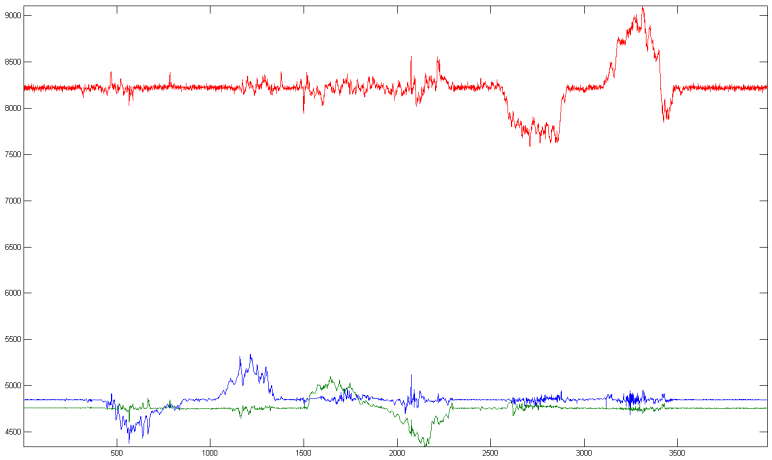


Figure 7: Calibration data for the gyroscopes

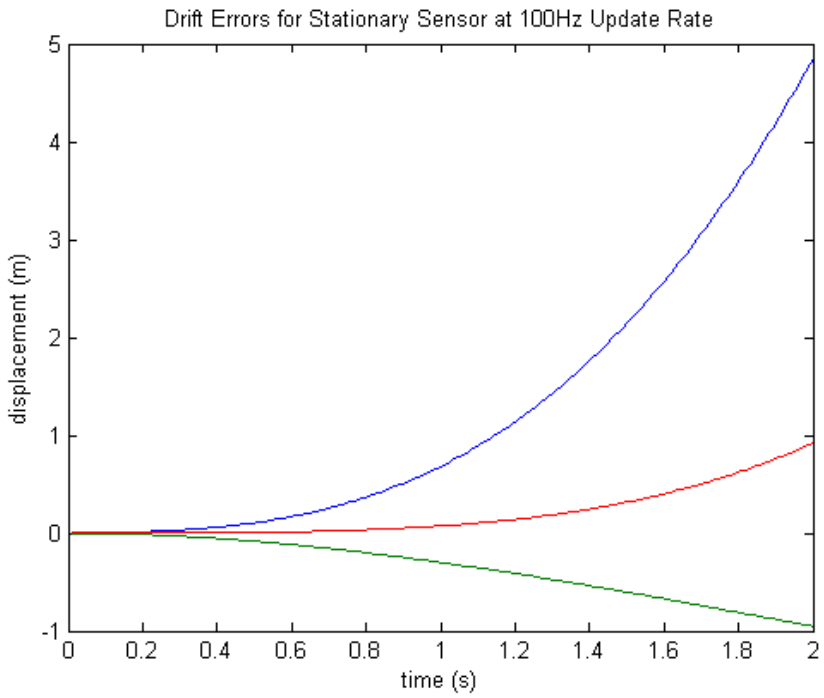


Figure 8: Stationary device log

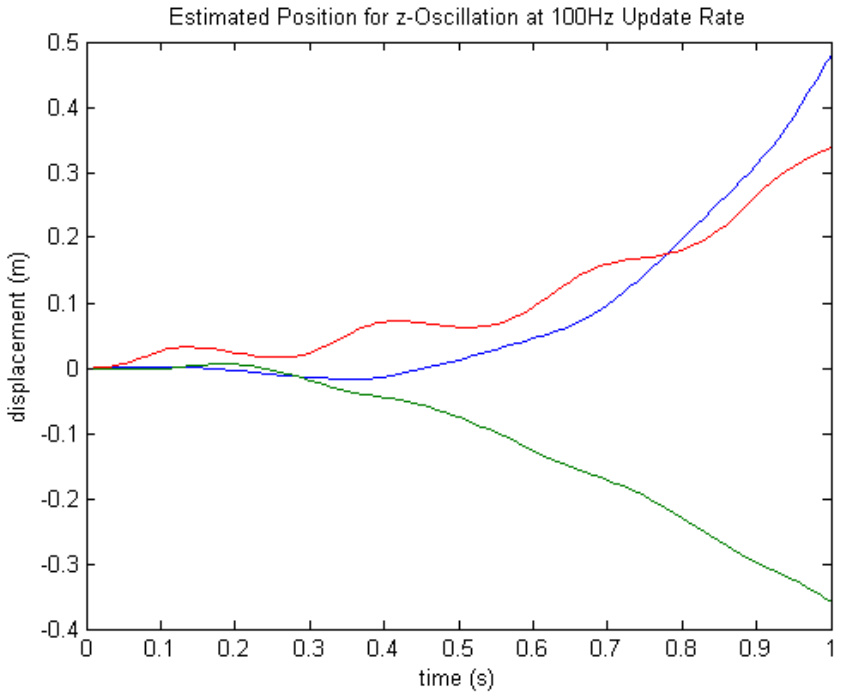


Figure 9: Oscillating device log

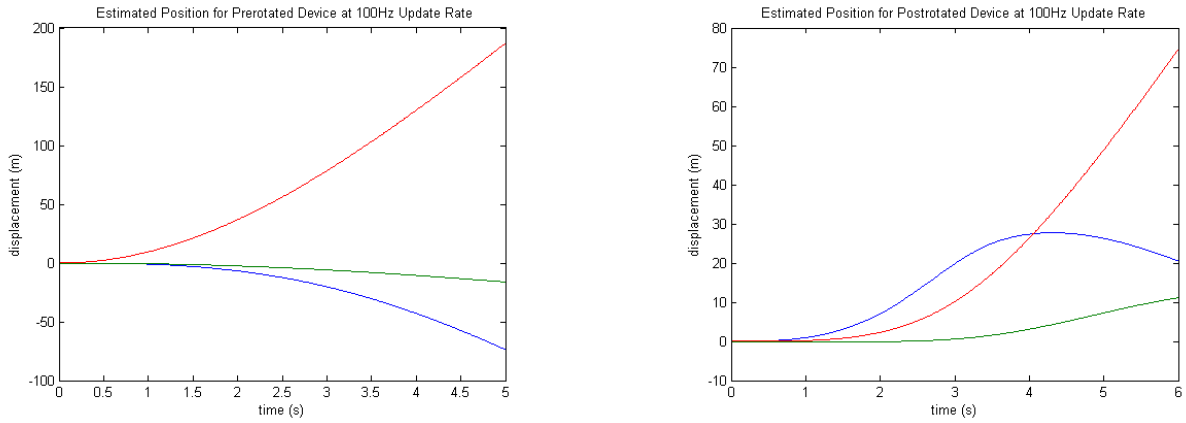


Figure 10: Prerotation (left) vs. Postrotation (right) logs