Application of the Kalman Filter to Rocket Apogee Detection

By
David W. Schultz
NAR 63255

Contents

Abstract

Commercial altimeters used for deployment exhibit less than perfect performance in deploying at apogee. The techniques used are capable of getting in the neighborhood of apogee but rarely get it exactly. Sometimes the errors are very large (several seconds) which results in higher dynamic loads on the recovery system.

The Kalman filter is applied to this problem with excellent results. A filter is first designed and used on pre-recorded flight data to verify its operation and performance. A filter is then designed for and used in an altimeter to demonstrate that a Kalman filter is a good fit to commonly available flight hardware.

The filter provides excellent results:

1. Accurate apogee deployment time.
2. Apogee determination immune to transonic effects.
3. Provides reasonable accuracy at altitudes exceeding the pressure sensor range by falling back to using only the acceleration measurement.
4. Accelerations exceeding the accelerometer's range do not effect accuracy of apogee time.
5. Works in a common low end micro-controller.

Introduction

In my research and development report presented at NARAM 44, I described the development of a Kalman filter designed for use with pressure altitude data. The filter was tested using previously recorded flight data and showed good results at detecting the time of apogee. For future work I stated that I wanted to extend the filter to also use acceleration measurements and implement the filter on flight hardware. This paper describes those efforts.

My goal for this project is to show that a Kalman filter does a better job of detecting apogee than other algorithms. I also want to show that even though the Kalman filter requires considerably more computation than typical apogee detection algorithms, it can be used in low end micro-controllers typical of what is used in commercial altimeters.

Background

The algorithms currently used by commercial deployment altimeters give variable results. Sometimes they work quite well but other times they do not. As demonstrated by the flight data from my first report. Two flights in particular convinced me that better methods were required. The first was a level 3 certification flight by Dave Schaefer. His rocket used a RRC2 (barometric) and a RDAS (accelerometer) altimeters for deployment. The RRC2 deployed 2 seconds past apogee and the RDAS did not fire either of its ejection charges until after the rocket had landed. The second flight was of my model of a PAC-3 missile on a K motor. This rocket used two accelerometer based altimeters (AltAcc and RDAS) which both deployed quite late. This flight exhibited a pronounced pitch oscillation which was the primary cause of the late deployment.

My first report discussed in some detail the problems associated with using a barometric pressure measurement to detect apogee so I will not repeat that here. I will instead focus on the problems with using an accelerometer.

The method used to detect apogee using an accelerometer is very simple. Integrate the measured acceleration during flight (after removing the constant effects of gravity) to get velocity. The time when the velocity changes from positive (upward motion) to negative (downward motion) is used to determine the moment of apogee. The major problems with this method are sensor offset and the treatment of two vector quantities as scalar.

Sensor Offset

The integrating acceleration algorithm is very sensitive to any bias errors. This is because the error accumulates with each iteration thus increasing the magnitude of the error with time. The primary source of bias is from the determination of the force of gravity.

The zero offset of acceleration sensors used in altimeters (Analog Devices ADXL150 or equivalent) changes with temperature and the age of the sensor. Therefore it is not possible to use a preprogrammed value for the offset. The offset must be measured after the altimeter is turned on and prior to the beginning of flight. Most commercial altimeters perform a running measurement of this offset that ends when the flight begins.

Offset error will be introduced if the rocket is not setup and launched from a perfectly vertical direction. The maximum error in the offset (assuming the 20 degree angle from vertical maximum in the safety code is not exceeded) will be 6%. Offset errors can also be introduced if the sensor is not perfectly aligned with the long axis of the rocket. These errors are typically small as a 5 degree error in alignment results in a 0.38% error in the measurement.

The sensitivity of the sensor (unit of acceleration per volt of output) also varies with temperature but does not effect the performance of the altimeters. This is because the altimeters never convert their measurements into engineering units (meters/sec/sec for example) and leave them in the form of Analog to Digital Converter (ADC) counts. Because the criteria for apogee is that the integrated value returns to zero (where it started) the actual magnitude of each measurement is not important just so long as they are consistent throughout the flight.

The 1G offset must be measured with great precision because small errors in the offset cause large errors in apogee time. For example, the Analog Devices ADXL150 accelerometer is commonly used in altimeters. This device has a nominal sensitivity of 38mV/G. When combined with a 10 bit ADC (a common resolution) there is a +/- ½ LSB uncertainty in the measured value or 5000mV/1024 counts / 38mV/G or +/- .064G uncertainty in the offset. This doesn't seem like much but because of the integration process, it accumulates to significant values. In a typical 20 second flight, the uncertainty in velocity resulting from the offset uncertainty is +/- .064G * 32ft/sec/sec * 20 sec = +/- 41 ft/sec. Because drag is low, the primary acceleration on the rocket is the 1G from gravity so this translates into a greater than +/- 1 second uncertainty in the apogee time.

The offset can be measured with greater resolution by averaging many samples together. This works because the accelerometer generates a significant amount of random noise which dithers the ADC reading around the true value. See Appendix D for more discussion of noise.

Non Vertical Flight

The primary source of error is from non vertical flight paths. This cannot be compensated for without the use of a full blown inertial measurement system (three axis measurement of acceleration and rotation). An article in *High Power Rocketry*[1] discusses this problem but the treatment is limited. A program was created to simulate high angle flights but the program starts its simulation after motor burnout. Flight tests were conducted but the recorded data was compared with the simulation and no attempt was made to measure apogee detection performance.

The problems associated with non vertical flight are the result of treating two vector quantities (magnitude and direction) as scalar values. The two vectors are the measured acceleration and gravity.

---

[1] Cumming, Duncan, "Non Vertical Flight and the Combridge IA-X95 Accelerometer", *High Power Rocketry*, August 1996, pp 33-37.

Gravity always acts in the same direction: down. But the rocket is not constrained to keep its acceleration in a perfectly up direction. The result is that the two vectors are acting in different directions.

This causes problems when the time comes for the altimeter to perform its integration. It needs to integrate the measured acceleration but must first subtract the acceleration due to gravity from this. These are both vectors but because their relative directions are not known, they are treated as though their directions were identical. This results in errors in the integrated value. There are two ways to look at this situation.

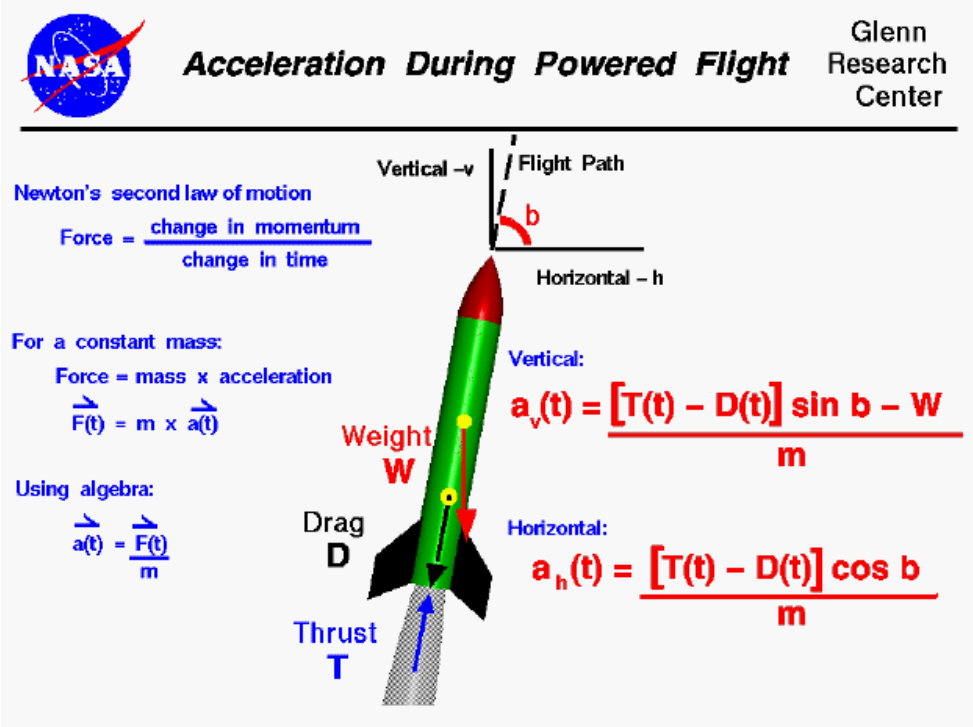

Figure 1, Acceleration vectors. From NASA web page: http://www.grc.nasa.gov/WWW/K-12/airplane/rktapow.html

The first viewpoint is that the altimeter is attempting to determine the time of apogee. In this case the pre-launch value for the 1 gravity offset is correct. But the measured acceleration is not necessarily aligned with the vertical. This misalignment results in integration errors.

Figure 1 shows that the measured acceleration (thrust minus drag) must be corrected by the sine of the angle of flight. Since the altimeter has no knowledge of this angle, it assumes a value of 90 degrees (vertical). Non vertical flight will result in the value used being too large.

During the boost phase of the flight, thrust exceeds drag. Therefore if the flight angle is non vertical, the value integrated is too large and the integrated velocity will be too high. Because the velocity is larger than it should be, it will take longer for it to decrease to zero in the coast phase resulting a late deployment.

During the coast phase, there is no thrust. Now if the flight is non vertical, too much acceleration from drag will be subtracted from the velocity integral. This tends to result in the velocity integral reaching zero (and apogee) before it should.

Because the error in the boost phase tends to result in late deployment and coast phase errors lead to early deployment you might assume that they cancel each other. They do not. The total error from each phase is a complex function of the measured acceleration and flight angle. Only if these match each other exactly will they cancel out.

During most flights the rocket is observed to "arc over" and is typically in a horizontal attitude at apogee. Obviously this means that there is a large difference in the direction of measured acceleration and gravity. But the errors introduced by this effect are moderated because the rocket is slowing down. Because drag is proportional to the square of velocity, the resulting drag is very small as the rocket approaches apogee. Even though the magnitude of the angle error is very large, the drag is very small resulting in a small net error in the velocity integration.

The other point of view is that the altimeter is attempting to find the time when the rockets velocity is at zero. Now it is the measured acceleration that is correctly aligned but gravity that needs to be corrected by the sine of the flight angle.

If the flight is non vertical, the value used by the altimeter for gravity will be larger than it should be because it is not corrected for the flight angle. This results in the velocity integral reaching zero before the rockets velocity reaches zero. (Because of the trajectory, this is actually a minimum velocity instead of zero.) Thus deployment is always early. This analysis is confirmed by Table 2 in another *High Power Rocketry* article[2]. This table shows for various launch angles the minimum velocity and time, velocity at apogee and time, and velocity and time of altimeter apogee. It shows that for angles of 20 degrees from vertical or less that the difference in velocity at apogee from the minimum during flight is very small. It is hard to tell the significance of the altimeter data because the conditions were not fully specified. The article does not indicate if the altimeter 1G offset is determined with the rocket at the launch angle or not.

---

[2] Scott Bartell and Konrad Hambrick, "Rocketry Electronics Revisited", *High Power Rocketry*, May 2000, pg. 36.

Rockets which exhibit pitch oscillations during flight will create large errors in the velocity integral. This is demonstrated by the data from the flight of my PAC-3 rocket. (See previous R&D report.) Both altimeters were several seconds late.

Error Summary

The acceleration measurement is prone to a wide range of errors. These accumulate in the velocity integral and never vanish. Some of the error sources can be controlled (offset, and alignment) but there is no way to control the effects of non vertical flight. An acceleration based altimeter simply cannot detect apogee. The best it can provide is an estimate that is in the neighborhood of apogee.

The accelerometer based altimeter is adversely effected by non vertical flight but it doesn't have any trouble with supersonic flights. The barometric sensor actually measures the rockets altitude and is thus perfect for determining apogee. But it has severe problems with supersonic flights. Perhaps they could both be used to utilize their strengths and minimize their problems.

The obvious technique is to use the accelerometer to estimate velocity and use the velocity to prevent the barometric system from functioning until the velocity had decreased well below Mach 1. But this method throws away valuable information: we know something about how the rocket is moving. It must follow the basic laws of dynamic motion.

The Kalman filter combines a model of a dynamic system with measurements of that system to produce an optimal estimate of what the system is doing. The dynamic model is used to relate the measurements and effectively as a third measurement.

Kalman Filter

The Kalman filter has a long history of application to rocket and missile guidance dating back to the 1960' s. While the theory behind the filter is fairly complex, the basic idea is quite simple: combine a model of a dynamic system with noisy measurements of that system to produce an optimal estimate of what the dynamic system is doing (its state). A full understanding of the Kalman filter requires knowledge of random variables and state variable control theory. Happily, these were the only non-elective courses required by Oklahoma State University for my Masters degree in Electrical Engineering.

The Kalman filter using just a pressure measurement I developed in my previous report showed a tendency to be slightly early in its apogee time. This was the result of using a constant acceleration model when the acceleration was constantly decreasing as velocity (and the resulting drag) decreased near apogee. The filter was constantly trying to catch up with the actual acceleration. The filter response was slowed because the model noise value used was made very small to prevent transonic pressure effects from causing an early deployment.

One way to improve this would be to include drag in the dynamic model. But this introduces problems of its own. The first is that the Kalman filter only works with linear systems and because drag depends on velocity squared, this makes the model non-linear. There are extended versions of the Kalman filter for non-linear systems but they are fairly complex. Using the extended Kalman filter also requires much more computation because the gains no longer converge to constant values and is beyond the capabilities of most inexpensive micro-controllers.

Because many recording altimeters have both pressure and acceleration sensors, I have decided to use both measurements in the Kalman filter. I believe that this will provide better performance than the pressure only filter.

The first step in producing a Kalman filter is to derive a model of the system in question. In the case of a rocket in flight, I have chosen a simple model which assumes that the acceleration of the rocket is constant. The acceleration is actually never constant but near apogee where velocity is low and drag is minimal, it is very nearly constant.

The model used is:

$$a_{t+\Delta t} = a_t$$
$$v_{t+\Delta t} = v_t + a_t * \Delta t$$
$$s_{t+\Delta t} = s_t + v_t * \Delta t + a_t * \frac{\Delta t^2}{2}$$

Where:

a is acceleration
v is velocity
s is position
t is time
) t is the time between samples

a, v, and s constitute the state of the system.

The Kalman filter equations[3] are:

System model:
$$x_k = \Phi_{k-1} x_{k-1} + w_{k-1}$$

Measurement model:
$$z_k = H_k x_k + v_k$$

x_k is the state of the system at step k
M_{k-1} is a matrix which transforms the system state at time k-1 to time k

w_k represents the noise in the system model
z_k is the measurement
H_k transforms the system state into the measurement
v_k is the measurement noise

The measurement model reflects the fact that it might not be possible to directly measure the system states but that there is a way to transform the system states into the measurement. It also includes the measurement noise.

---

[3] Gelb, A. Editor. "Applied Optimal Estimation", The M.I.T. Press Cambridge MA, 1974, pg. 110

These equations are the basis for the actual Kalman filter. The filter equations are usually expressed as though there are two steps in the process. This is reflected in the use of (-) and (+) to modify the step k. The first step uses the state transition matrix Φ to extrapolate the previous system state to the current time.

State estimate extrapolation:

$$\hat{x}_{k(-)} = \Phi_{k-1} x_{k-1(+)}$$

State estimate update:

$$\hat{x}_{k(+)} = \hat{x}_{k(-)} + K_k \left[z_k - H_k \hat{x}_{k(-)} \right]$$

The state extrapolation predicts the current value of the state using the previous state and the state transition matrix. The update uses the difference between what was measured (z_k) and what was expected ($H_k x_{k(-)}$) multiplied by a gain matrix to form a correction that is applied to the system state. This is usually called a predictor- corrector form.

This seems pretty simple but the computation of the gain matrix is anything but simple. It is constrained to have a value which minimizes the variance in the system state error which leads to considerable computation.

Error covariance extrapolation

$$P_{k(-)} = \Phi_{k-1} P_{k-1(+)} \Phi_{k-1}^T + Q_{k-1}$$

Error covariance update

$$P_{k(+)} = \left[I - K_k H_k \right] P_{k(-)}$$

Kalman gain matrix

$$K_k = P_{k(-)} H_k^T \left[H_k P_{k(-)} H_k^T + R_k \right]^{-1}$$

(Note: The superscript “T” indicates a matrix transpose operation and a superscript “-1" is matrix inversion.)

This introduces several new matrices. The P matrix holds the estimate of the noise statistics (covariance) for the system state, Q contains the model noise covariance, R contains the measurement noise covariance, and K is the Kalman gain matrix. The filter estimates the noise statistics of the state estimate so that it can be used in determining the gains.

As an example of why this is done, consider the case where you have two noisy measurements of the position of a vehicle. You would like to combine these two

measurements in such a way as to provide the best estimate of the position. In the absence of any additional information, the best you can do is to average the two measurements. But if you have some knowledge of the noise in each measurement you can do better. By weighting the measurement with less noise more highly, you get a better estimate. This is what the Kalman filter is doing but in a more complex situation. It uses the estimate of the variance in the state and the variance in the measurements to determine how to combine the two to provide the best estimate. In this case "best" means that the least squared error is minimized.

The equations were not too bad until the last one. This includes a matrix inversion which is fairly time consuming. The number of operations required to compute a matrix inverse is proportional to the cube of the matrix dimension. So a 4X4 inversion takes eight times as long as a 2X2 inversion.

These equations are for the general case so now I will fill in the values for this particular case.

$$x_k = \begin{pmatrix} s_k \\ v_k \\ a_k \end{pmatrix}$$

$$\Phi_k = \begin{pmatrix} 1 & T & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{pmatrix}$$

$$H_k = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The H matrix is very simple because the measurements match two of the system states exactly. This is handy because it simplifies things. Because it is a 2x3 matrix, the matrix that must be inverted in the Kalman gain equation is only a 2x2 matrix so the inversion is quite easy. Because it is mostly zeros, the calculations are greatly simplified. It is even easier when only the pressure measurement is being used as the matrix to be inverted is then a 1X1 and the inversion is trivial. If I were using a program like MATLAB to perform my calculations this wouldn't be important as MATLAB handles matrixes quite

simply. But I am coding the filter directly in C (or assembly) so keeping things simple is definitely a plus.

One question which must be answered is if the systems states can be determined (are observable) with the chosen measurements. The criteria for this is that the matrix[4]:

$$\left[H^T \middle| \Phi^T H^T \middle| ... \middle| \left(\Phi^T\right)^{n-1} H^T \right]$$

(where n is the number of states) must be non-singular. This means that the matrix is of rank n.

$$\left[\begin{matrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{matrix} \middle| \begin{bmatrix} 1 & 0 & 0 \\ T & 1 & 0 \\ \frac{T^2}{2} & T & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \middle| \begin{bmatrix} 1 & 0 & 0 \\ T & 1 & 0 \\ \frac{T^2}{2} & T & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ T & 1 & 0 \\ \frac{T^2}{2} & T & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \middle| \right]$$

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & T & 0 & 2T & 0 \\ 0 & 1 & \frac{T^2}{2} & 1 & 2T^2 & 1 \end{bmatrix}$$

Which is of rank 3 and non-singular. (Rank can be determined by using linear operations on the rows. When you have reduced as many rows to all zero values as you can, the number of remaining non-zero rows determines the rank.)

If only the acceleration measure were being used then the measurement matrix H would be:

$$H = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

And the observability test results in:

---

[4] Gelb, A. Editor. "Applied Optimal Estimation", The M.I.T. Press Cambridge MA, 1974, pg. 68

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Which is of rank 1 and thus the system states are not observable.

A similar exercise for just the pressure measurement results in:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & T & 2T \\ 0 & \frac{T^2}{2} & 2T^2 \end{bmatrix}$$

Which is of rank 3 and the system states are observable.

The observability test shows that a filter using just the pressure measurement or pressure and acceleration measurements will work and be able to measure all of the systems states. A filter using just an acceleration measurement cannot determine the velocity and position states and thus is not suitable for use.

The model covariance matrix is simplified by assuming that all of the model noise is in the assumption that acceleration is constant. This is the one aspect of the Kalman filter I am unsure about. The texts I have looked at do not discuss model noise much nor do they provide good examples.

$$Q_k = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_m^2 \end{pmatrix}$$

The measurement covariance is also pretty simple. The off diagonal elements are zero because the measurements are independent of each other.

$$R_k = \begin{pmatrix} \sigma_s^2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_a^2 \end{pmatrix}$$

While I kept the subscript for these matrixes, except for the system state they are all constant. Because they are constant the Kalman gain will converge to a constant value. This allows for the gain to be pre-computed and then only the state extrapolation and update equations need to be used when running the filter. This is what allows a micro-controller to be able to use a Kalman filter.

Testing Using Recorded Data

In order to test the performance of the new Kalman filter, I modified my existing Kalman filter code to include the acceleration measurement. See Appendix A for the program listing. I then tested this filter on a number of previously recorded data sets. The performance of the filter was astonishingly good. As an example, Figure 2 is a plot of data from the flight of my PAC-3 rocket. This data set is particularly useful because the flight exhibited pitch oscillation, pressure anomalies, and deployment was very late which gives the filter uncorrupted data well past apogee.

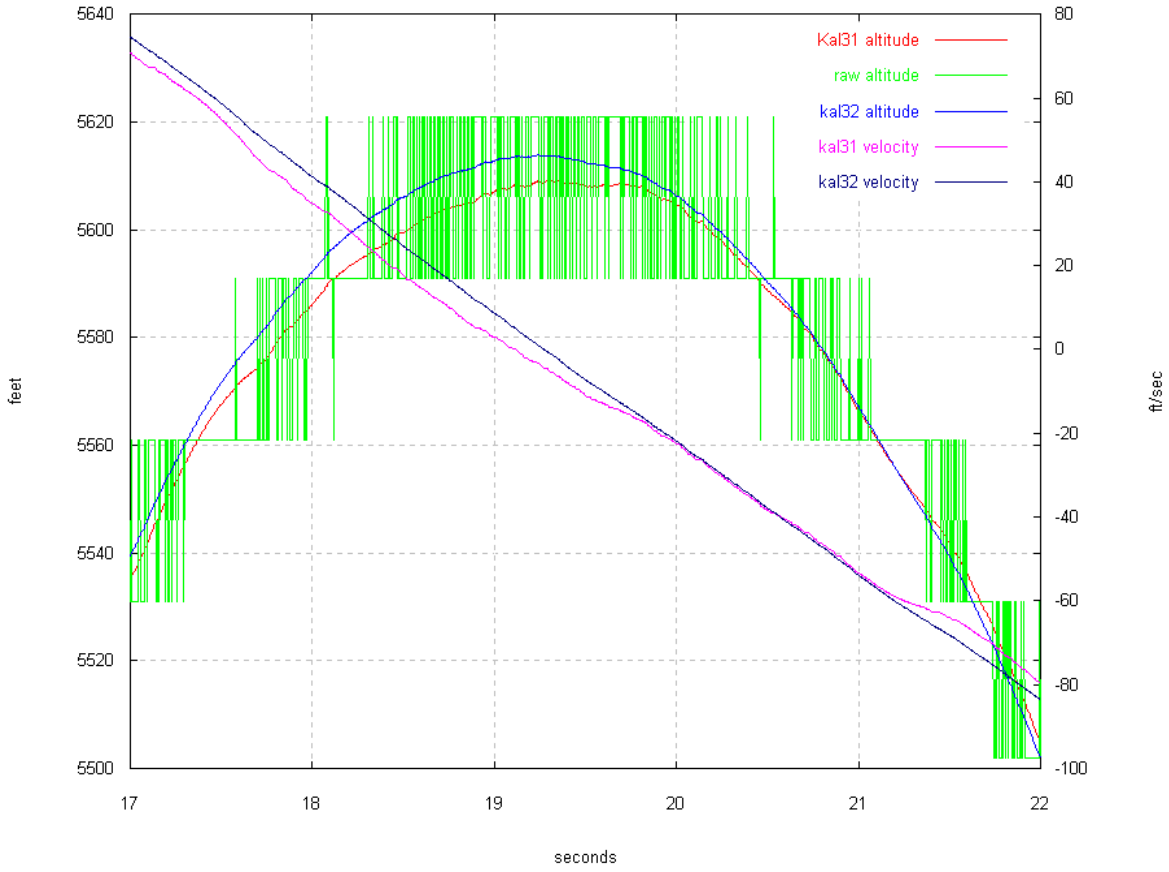

Figure 2. Kalman filtered data from PAC-3 flight

This plot is fairly busy as it has a lot of information. The noisy green lines are the raw barometric altitude without any filtering. This is in the plot as a reference. The red line is the altitude filtered through the pressure only Kalman filter (the ‘31" stands for a three state, single measurement filter). The blue line is from the new filter. (‘32" stands for three state, two measurement filter) The purple line is the velocity estimate from the pressure only filter and the black line is from the new filter.

The new filter produces a much smoother result than the old filter. Which is interesting because the value I used for the model noise in the new filter is much larger. This would tend to cause the Kalman gains to favor the measurements more which would result in less smooth results. But because the new filter includes the acceleration measurement, the results are better. To try and understand what is happening, here are the gains used by the filter.

	Pressure altitude	Acceleration
position	0.004587	0.004520
velocity	0.000216	0.000036
acceleration	0.002018	0.095108

The two columns are for the corrections based on the difference between the estimated and measured values. Each row shows the gain which is multiplied by this difference and then applied to the estimated value.

As can be seen, the gain for the acceleration correction to the estimated acceleration is by far the largest. This directly reflects the relationship between the acceleration measurement noise and the model noise used. Note that the gains for correcting the velocity estimate are quite small. This is the cause for the smoothness of the curve because the filter is placing a high level of confidence in the model.

The Kalman filter also produces an estimate of the noise statistics for the state variables.

Altitude	Velocity	Acceleration
1.016 feet	0.635 feet/sec	1.850 feet/sec/sec

This is the standard deviation and is much better than the noise in the sensor data. Which I measured at 15 feet for the altitude and 6 feet/sec/sec for the acceleration.

The improvement in apogee performance using this filter over the pressure only filter is very small. Both do a good job in determining the time of apogee. The main difference in performance is in estimating velocity. Figure 3 shows the velocity estimate for both filters from launch to apogee.

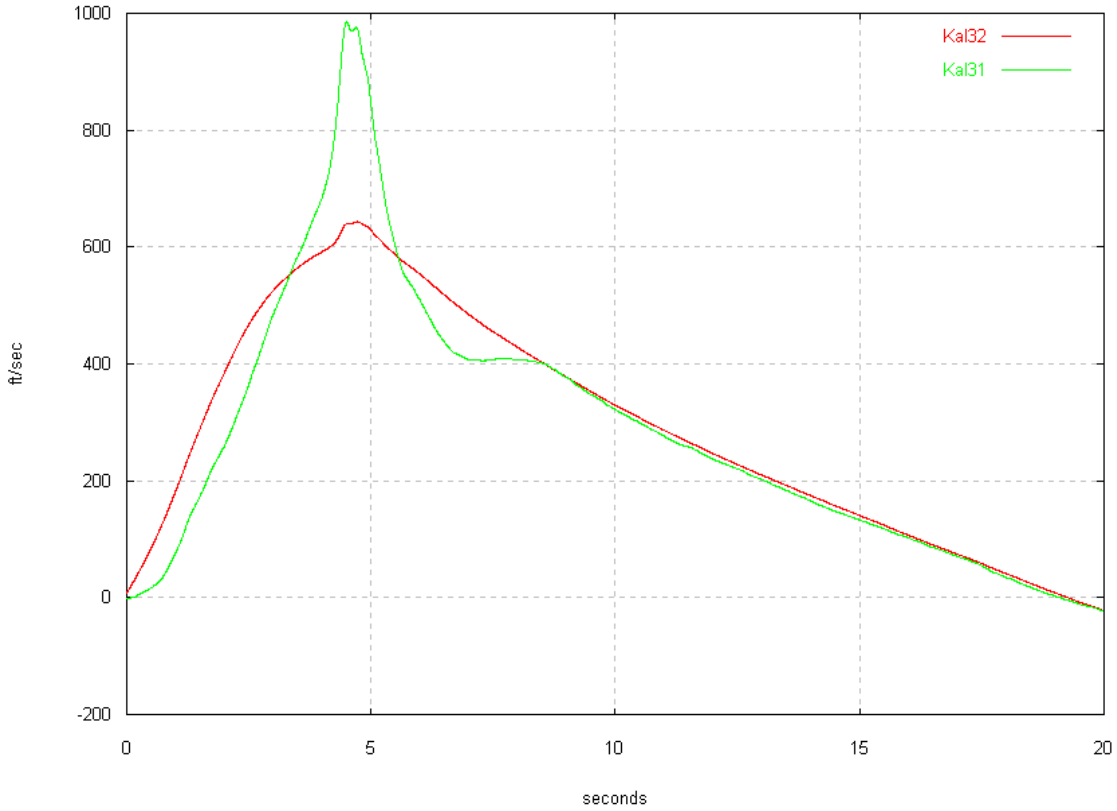

Figure 3, Velocity estimate from pressure only Kalman filter (Kal31) and pressure plus acceleration Kalman filter

The velocity estimate is greatly improved by including the acceleration measurement in the early part of the flight. This is expected because the primary limitation of the dynamic model is that acceleration is constant and this is the period when acceleration is changing the most.

I used this filter on data from many different flights and the results all look about the same. One thing that becomes apparent when looking at these plots is that estimating how well the filter is performing in determining the time of apogee is simply a problem in deciding when apogee really happens. The only data available to make that determination is the pressure data and it has its limitations. Deciding the precise moment of apogee given the high amount of sensor noise is very difficult. But as near as I am able to judge, the filter hits the precise moment of apogee every time.

Another data set that is very interesting to look at is from a very high performance hybrid motor powered rocket. The data was recorded by an RDAS which was also recording GPS (Global Positioning System) data. The things that make this flight interesting are the high level of noise in the acceleration data, velocity exceeding Mach 1, and altitude exceeding the range of the pressure sensor.

Transonic flight results in anomalous static pressure readings and I had to think about what to do about them. I finally decided to use the Kalman velocity estimate to cause the filter to ignore pressure readings within a limited range of velocities. Within this velocity range, the difference between measured and estimated altitude is not used to update the Kalman filter state estimates. This prevents the transonic anomalies from corrupting the velocity and acceleration state estimates. In addition the pressure altitude measurement is copied directly to the altitude state estimate. This is so that when the velocity falls outside of this velocity range there will not be a large difference between estimated and measured altitude. The plot of the data from this flight (Figure 4) clearly shows this behavior.

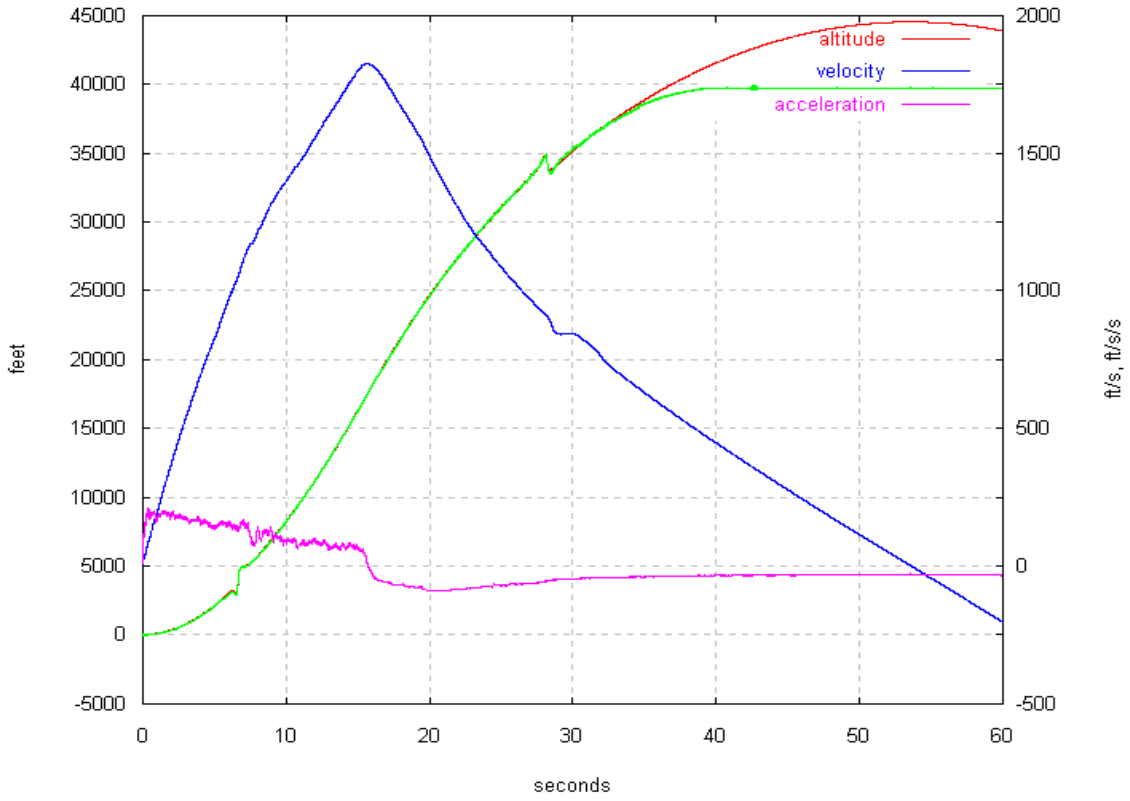

Figure 4, High Performance Flight Data

Also obvious in this plot is that the range of the pressure sensor was exceeded which resulted in the measured altitude never going over 40,000 feet. The filter code includes a test to determine if the pressure is lower than can be reliably measured and if it is, the pressure reading is ignored. At this point the filter effectively falls back to being a simple integrating accelerometer. But it has the advantage of having most of the errors from early in the flight being corrected by the altitude data. This code uses an arbitrary altitude as the threshold but actual flight hardware can take into account the characteristics of the measurement system to derive an objective threshold for ignoring pressure altitude data.

Because this data set includes GPS data it is possible to compare the apogee times. Plotting the GPS altitude data (Figure 5) shows that the maximum GPS reported altitude happened at a little past 54 seconds into the flight. The Kalman filter apogee happens at 53.6 seconds. Given the uncertainties in the GPS altitude latency, I consider this to be an excellent result.

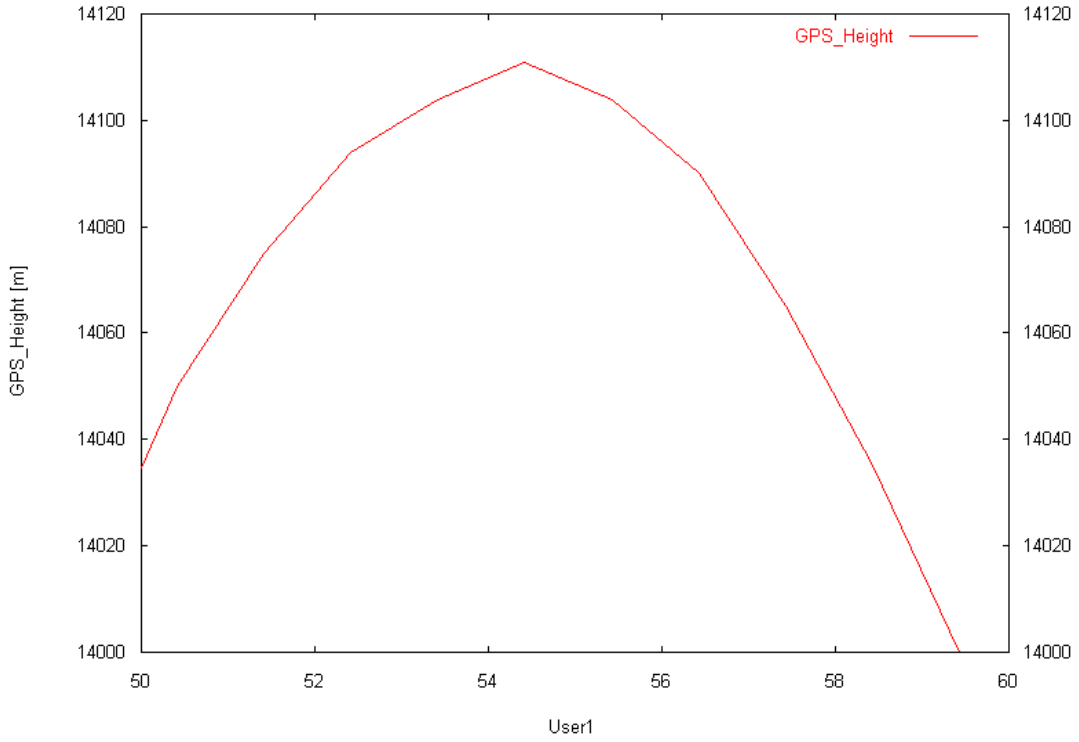

Figure 5, GPS data from high performance flight.

This flight used a hybrid motor which generated a large amount of vibration. Because of flaws in the design of the RDAS, this vibration generated large amounts of noise in the acceleration data. While the Kalman filter cleaned up this noise fairly well, it is not designed or intended for this purpose. It is vital that proper pre-sample filtering be used to prevent aliasing from corrupting the acceleration data into uselessness. See Appendix C for a discussion of sensor noise.

Kalman filter for the AltAcc

The filter works quite well with the RDAS but the RDAS samples its data at 200SPS and uses a 10 bit ADC. What would happen if a lower sample rate and 8 bit ADC were used? Fortunately, I also had an AltAcc on board the PAC-3 which used an 8 bit ADC and stores data at 16 SPS. After some quick changes to the Kalman filter code, I processed the AltAcc data. (See Figures 6 and 7)

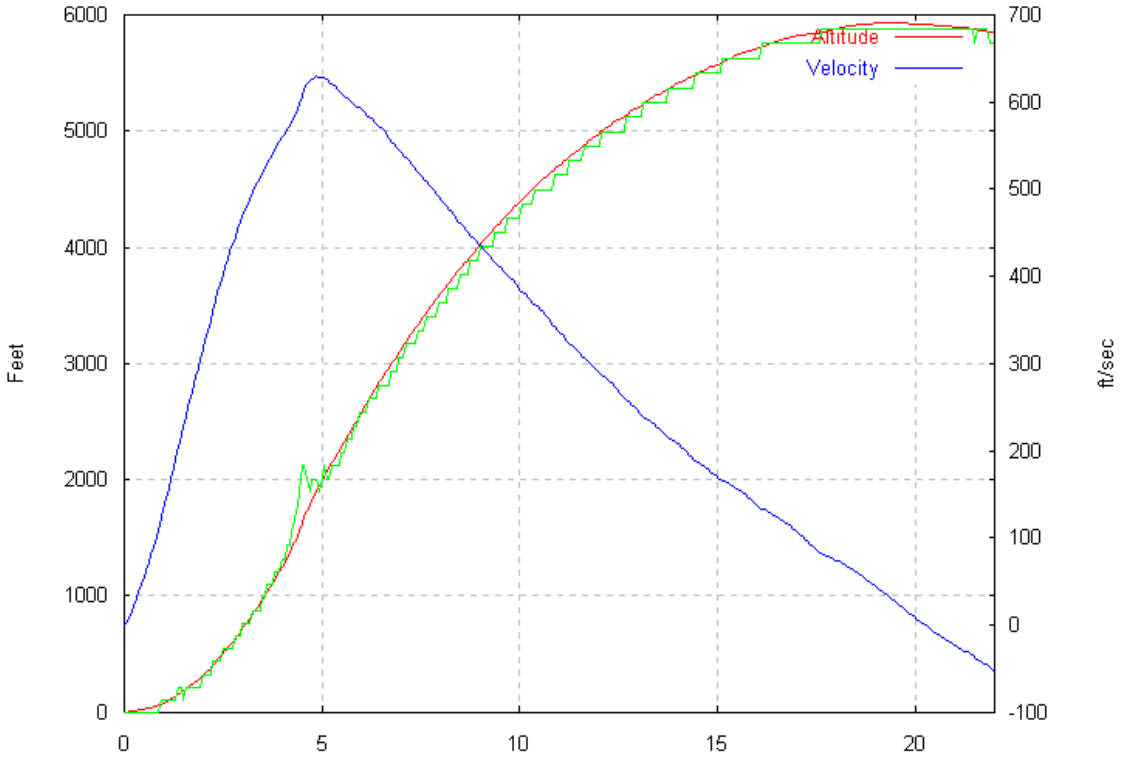

Figure 6, Kalman filtered AltAcc data

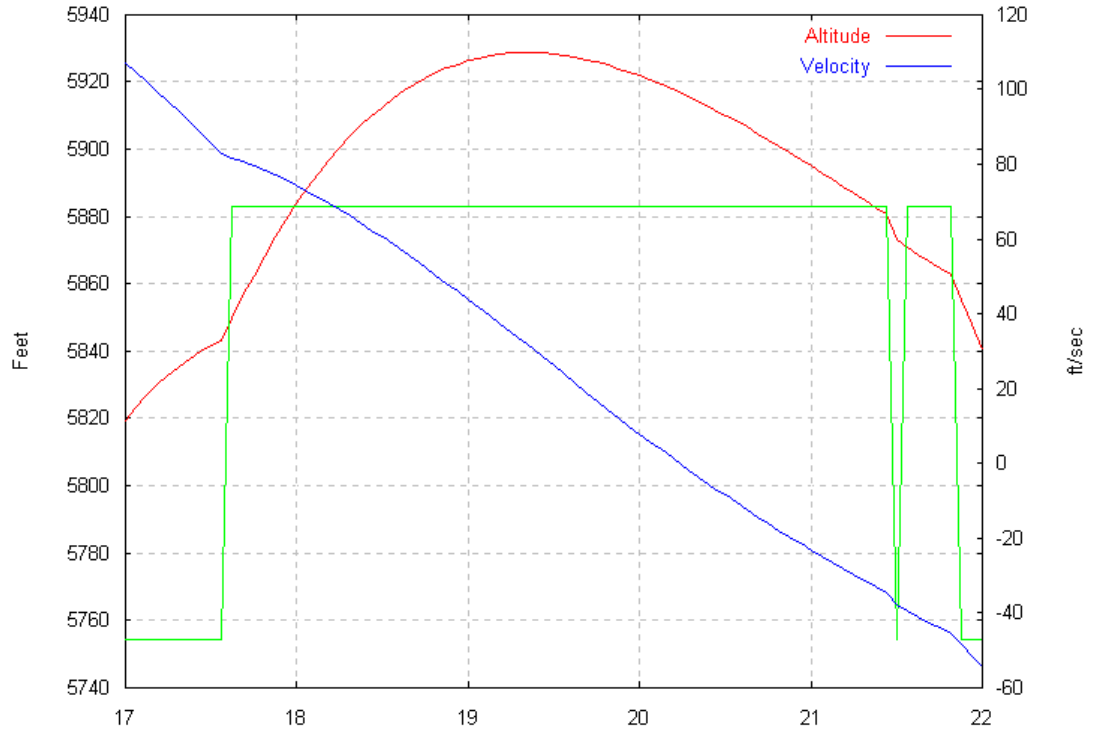

Figure 7, Kalman filtered AltAcc data near apogee

The data shows that the velocity goes negative at 20.25 seconds into flight. This time is different than the result from the RDAS data but the two altimeters are running on different clocks that are not synchronized. The only way to check on this is to look for the same event that appears in both data sets. For this I am using the apogee ejection event. The AltAcc reported that this happened at 23.875 seconds. The acceleration spike for this event appears at 23.475 seconds in the RDAS data. So the correction factor is about 0.4 seconds and this makes the AltAcc apogee time be 19.85 seconds based on the Kalman filter results.

This is not too bad but isn't that great. The difference in time is largely due to the lack of noise in the AltAcc pressure data. To help demonstrate this, I processed the RDAS data set to change its sample rate to 20 SPS.

Decreasing the sample rate by an integer factor is fairly simple. Just delete samples. To cut the sample rate in half, delete every other sample. This technique is referred to as "decimation" after the old Roman punishment for a legion that had disgraced itself. (Pick every tenth man in the legion and kill him.) Before reducing the sample rate by decimation you must first run the data through a low pass filter to prevent aliasing. But because the RDAS does not include an anti-aliasing filter, I chose not to do this step.

I decimated the RDAS data to a sample rate of 20 SPS and then process it using the Kalman filter. The results are shown in Figure 8. Apogee is still detected at 19.25 seconds.

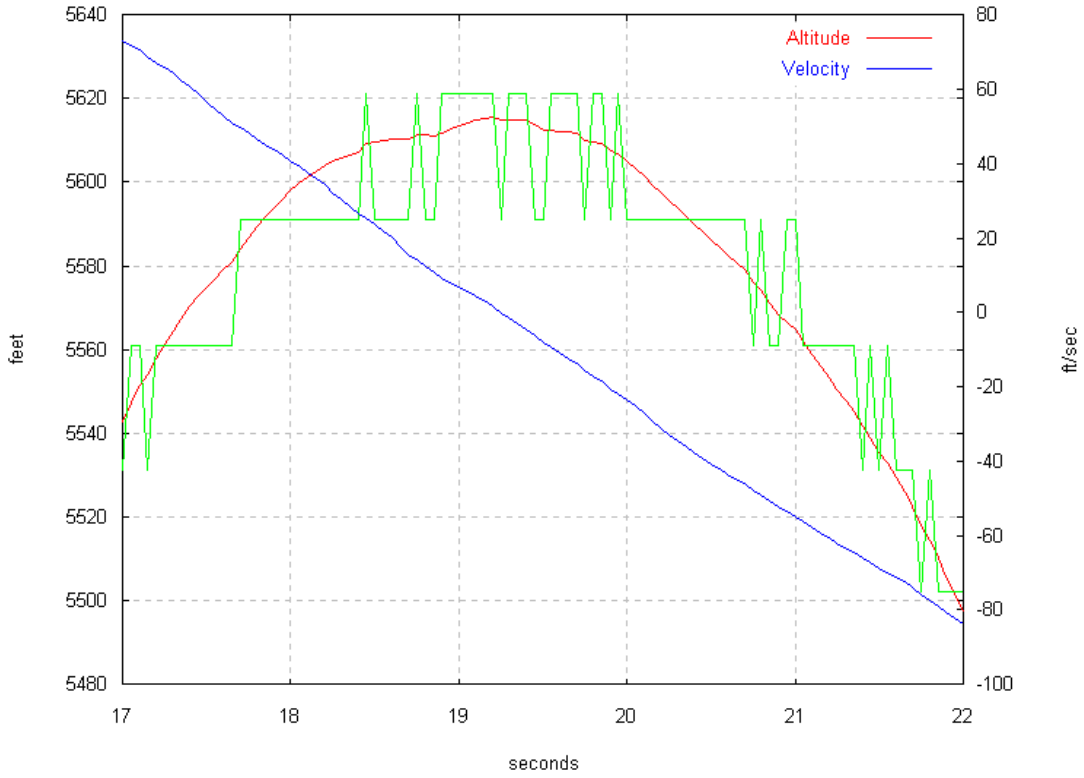

Figure 8, Kalman filtered RDAS data decimated to 20SPS.

A local flyer had a problem on a flight where he forgot to change the acceleration range on his AltAcc2C from +/-25G to +/-50G. The flight used an Aerotech I435 motor in a lightweight carbon fiber rocket. This resulted in the acceleration exceeding the range of the accelerometer and a very early deployment. I processed this data using a Kalman filter to see how it would perform. (See Figure 9.) While it isn't possible to judge the filters performance to apogee because of the very early deployment, it is possible to see how well it is tracking altitude. When deployment happens at 10 seconds into the flight, the filtered altitude is tracking the measured altitude nicely which indicates that the filter has recovered from the large measurement errors caused by the clipped acceleration data. From this I conclude that the filter can recover from badly clipped acceleration data so long as it has a little bit of time with good pressure and acceleration data to recover.

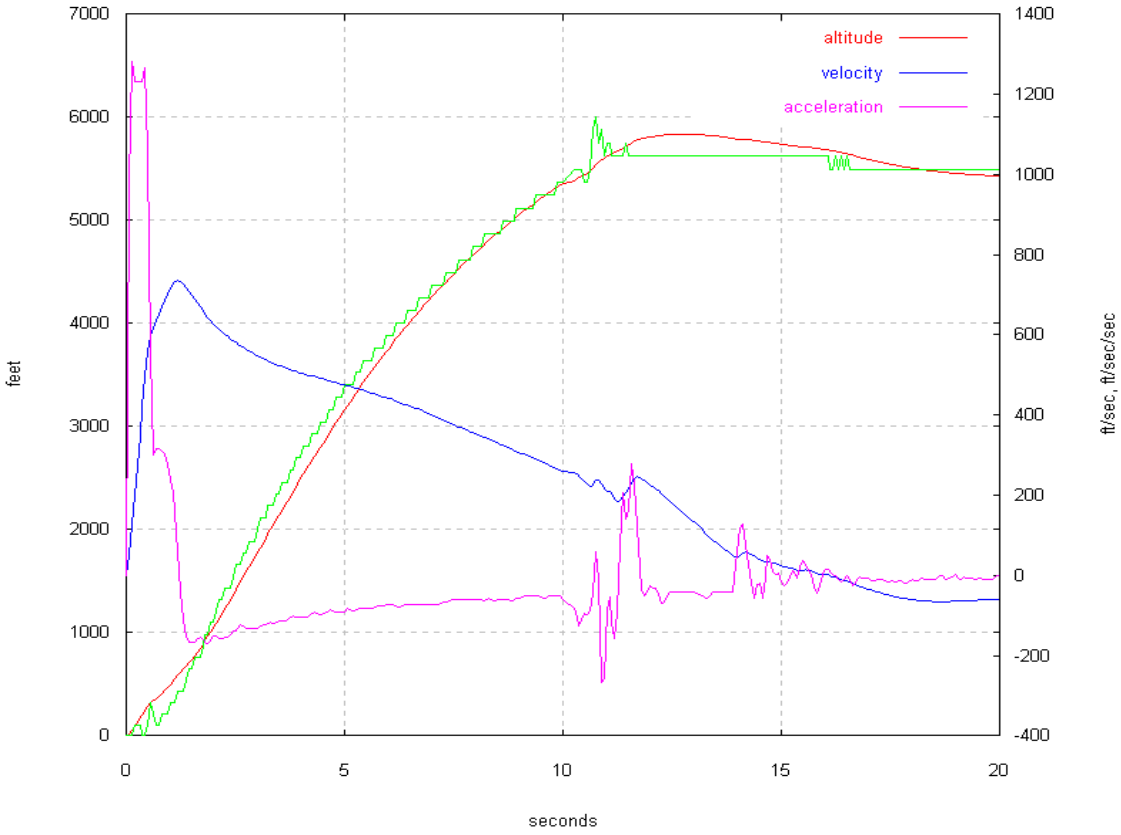

Figure 9, AltAcc data from flight that exceeded acceleration measurement range.

The Kalman filter demonstrated excellent performance when processing prerecorded flight data. It had no difficulty with transonic effects, high altitudes, pitch oscillation, or clipped acceleration. It was obviously quite capable of accurately determining apogee time with recorded flight data. The next step was to implement a filter in flight hardware.

Design and Development of Flight Software

I decided to use the Roctronics hardware of Robert Dehate for the first flight hardware implementation of the Kalman filter. There were several reasons for this decision.

1) The hardware design was complete and I only needed to purchase circuit boards from Robert and parts to build a working altimeter.
2) The micro-controller used was typical of what is used in commercial altimeters. Therefore, if it worked, it would demonstrate the filters suitability for commercial products.
3) The 16F628 micro-controller had sufficient program space and execution speed to run the filter. Program space turns out to be the limiting factor.

I chose to use Microchip's MPLAB Integrated Development Environment to develop the code because it targeted the desired processor and was free. Besides offering the usual editing and assembly functions, the MPLAB IDE also offered simulation capabilities which were very useful.

The volume of code required was quite large so I used the separate assembly and link features of MPLAB. I have included a listing of only the Kalman filter code in Appendix B. The remainder of the code is included separately in computer readable format. The program includes routines for continuity checking the ejection charge outputs, beeping out peak altitude, storing data to serial EEPROM, and serial communications to upload flight data. The final code required about 1500 words of code space. This is more than some older PIC micro-controllers have but most of the modern parts have 2K or more of code space.

The 16F628 micro-controller can run with clock speeds up to 20MHz but I decided to run it at 4MHz to demonstrate that it can easily run this code.

I wrote several other programs to go with this code. There are programs to compute the Kalman gains, the pressure to altitude conversion tables, and to process the flight data into a format suitable for use with plotting software.

Kalman Filter code

The heart of the software is the Kalman filter code. It calls routines to get samples of the pressure and acceleration converted to appropriate units and then performs the Kalman filter operations. Because micro-controllers do not have floating point number operations, the selection of number representation is a crucial step in writing the code.

I chose to use a fixed point number representation for most numbers using 16 bits for the integer portion and 16 bits for the fractional portion. This allowed for a reasonable range and sufficient precision. This then required a 32 bit multiply.

Fortunately, Microchip makes freely available the code for a 32 bit by 16 bit signed multiply routine which is perfect. A full 32 X 32 multiply was not needed because of the characteristics of the Kalman gains.

The Kalman gains are positive and have small values. I therefore chose to represent these as 16 bit numbers that were entirely fraction bits. This limits the gains to values less than one half but I felt that this was not a serious limitation. The gains will only exceed one half at very low sample rates and with very high model noise values.

The result of the 32X16 bit multiply is a 48 bit number. This must be truncated to a 32 bit number again with the binary point in the correct location. Because the numbers input to the multiplication have a total of 32 fractional bits, the result has 32 fractional bits. So we only keep the most significant 32 bits of the result and we are back to having 16 bits for the integer part and 16 for the fraction. Fixed point numbers are very useful if the dynamic range required is limited but you do have to keep track of the binary point.

By carefully choosing the sample rate I can use simple bit shifts for some of the required multiplications. The state extrapolation step requires several multiplies by the sample time. By using a sample rate that is a power of two the multiplies become simple shifts which are much faster than the full multiply.

A total of 8 multiplies are required to process each sample. Two are needed to convert the acceleration and pressure measurements into engineering units. These values cannot be left as ADC counts because the units must be consistent for the filter to work correctly. This causes a problem with the pressure to altitude conversion.

The relationship between pressure and altitude is highly non-linear. The standard atmosphere model is:

$$T = 15.04 - 0.00649h$$

$$p = 101.29 * \left[\frac{T + 273.1}{288.08} \right]^{5.256}$$

Where h is altitude in meters (less than 11,000m), T is temperature in Celsius, and p is pressure in kilo-Pascals.[5]

[5] http://www.grc.nasa.gov/WWW/K-12/airplane/atmosmet.html

Using this model requires raising a number to the 1/5.256 power to convert pressure to altitude. This function is not available in any of the code available from Microchip and even if it were, the code size and execution time would make it a poor fit for this project. Therefore I chose to utilize a piecewise linear interpolation scheme. In this I broke down the pressure range into 16 segments corresponding to the upper 4 bits of the 12 bit ADC result. Within each of these ranges I computed a slope and offset to compute the altitude. This is not the best method but it is computationally simple.

I decided to use metric units so that the maximum altitude would be 32,767 meters. Because the number representation has 16 fractional bits, I do not lose a significant amount of resolution by this choice.

The conversion to engineering units requires one multiply for the pressure and one for the acceleration. The Kalman filter requires six more multiplies to apply the gains. These operations consume the majority of the time required to process each sample.

A Kalman filter using only the pressure measurement would not have to convert the ADC reading into altitude. This results in odd units for velocity and acceleration (ADC counts/second and ADC counts/sec/sec) but it is not a problem. Thus this filter would only require three multiplies for the Kalman gains. This cuts the execution time significantly and allows the code to run at a higher sample rate.

Launch detect

Launch detection is quite simple. Because the Kalman filter is running continuously and providing a good velocity estimate, the velocity is used to determine when liftoff occurs. 15 meters/second was chosen as a reasonable value to indicate that the rocket had started its flight. This level will not be reached during normal handling of the rocket unless the orientation of the rocket is changed significantly. To help protect against orientation changes, acceleration is checked and if it is not greater than 10 meters/sec/sec (about 1G) then the altimeter does not switch to the flight state.

Accelerometer Offset

The one thing that could result in the velocity prematurely exceeding 15 m/s is drift in the accelerometer offset. Because the offset varies with temperature, measuring the offset once and then storing it is not sufficient. The offset must be measured continuously until launch is detected. The raw ADC acceleration reading is processed using a simple recursive filter and every two seconds the filtered value is stored in a double buffered manor. Once launch is detected, this buffering stops.

The double buffer is required because a single buffered system might be corrupted by the initial part of the motor thrust. There will be a period prior to launch detect when the

acceleration has increased above the 1G resting value and that cannot be allowed to corrupt the 1G offset. Determining the 1G offset is crucial to the performance of the altimeter because it is subtracted from every acceleration measurement prior to being used by the filter.

Pre-launch altitude is also double buffered but it is not critical to the altimeter performance. It is only used when computing the maximum AGL altitude after the flight is complete.

Altimeter state machine

The primary code loop of the altimeter is organized as a state machine. A variable is used to hold the current state and this variable is used with a lookup table to jump to the code required for each state. The states are:

State	Description	Waiting for Event
0	Pre-launch	Launch detect
1	In flight (launch detected and in flight)	Apogee
2	apogee (apogee detected, this state is used to time the duration of the pyro output)	Pyro output timer expiration
3	post apogee - not used	
4	drogue (Under drogue and waiting for the main altitude)	Main parachute altitude
5	main (Altitude for main deployment has occured. Used to time the main pyro output.)	Pyro output timer expiration
6	postmain (Under main parachute and waiting to land.)	Zero velocity (on ground)
7	ground (On the ground and the flight is over.)	No more events

Firmware Testing

Testing of the flight firmware began with the simulator included with the MPLAB programming environment. Using this to simulate a flight is not really possible because of

limitations in feeding realistic ADC values to the simulator and the speed of execution. It was still very useful in testing various blocks of code to verify that they were behaving in a reasonable manor.

After moving to the hardware I was able to make even more progress although it is difficult to debug something that is limited to beeping for its output. This is where the ability to store data really paid off. The Roctronics processor card design includes a 256 kilobit EEPROM for storing data

I started a debugging log to track what I had done, the problems encountered, and the fixes made. I did this in the hope of preventing a repeat of previous mistakes. It is included in machine readable form with this report.

After I had located many bugs I finally reached the point where the altimeter was able to pass a simple bench test where the altimeter is held upside down and then quickly inverted to simulate a launch. Once I reached this point it was time to move on to flight tests.

Flight Tests

The first flight test used a modified Aerotech Initiator flying on a G64. This flight revealed a problem in the Kalman filter code where I had changed the sample rate but not changed a key section of the code. While making some additional changes to the data storage code I discovered that the code was taking much longer to execute than I had thought. Because of this I had to drop the sample rate from 128 SPS to 64 SPS. No significant degradation in performance was expected. This code revision also resulted in a significant change to the data storage code. As a compromise between getting high sample rates and also covering the entire flight I changed to a dual rate system for data storage. During the first 16 seconds of flight data is stored at 64 SPS. After that the rate drops to 2 SPS.

The next flight test used the same rocket configuration. This revealed a problem in the piecewise linear interpolation code used to convert the pressure measurement to altitude. The problem was not in the flight code but in a companion program used to compute the table entries. I had sufficient confidence in the code at this point that the next flight would have an ejection charge controlled by the altimeter.

The third flight test used the same modified Initiator but I used a F39 motor with a delay long enough so that it would be past apogee and an ejection charge was connected to the altimeter. Visually the flight was perfect with ejection exactly at apogee. The recorded data (see Figures 10 and 11) shows this clearly.

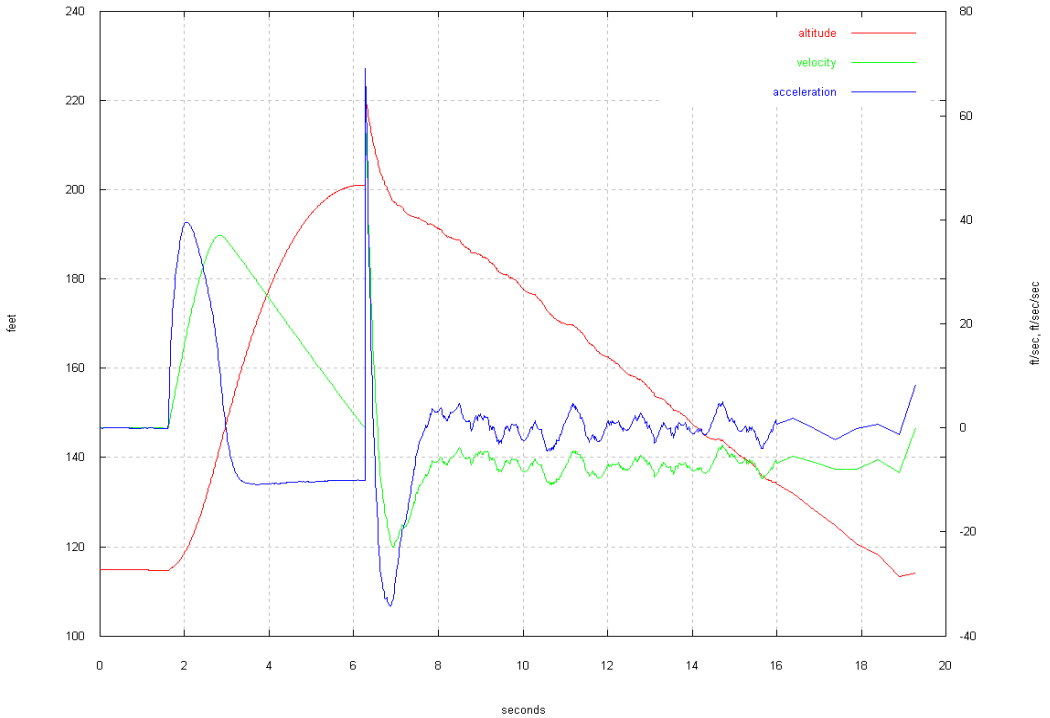

Figure 10, F39 flight test data

Th

e filtered acceleration looks to be smoothed a bit too much in this plot so I changed the Kalman gains for the next flight to decrease this smoothing.

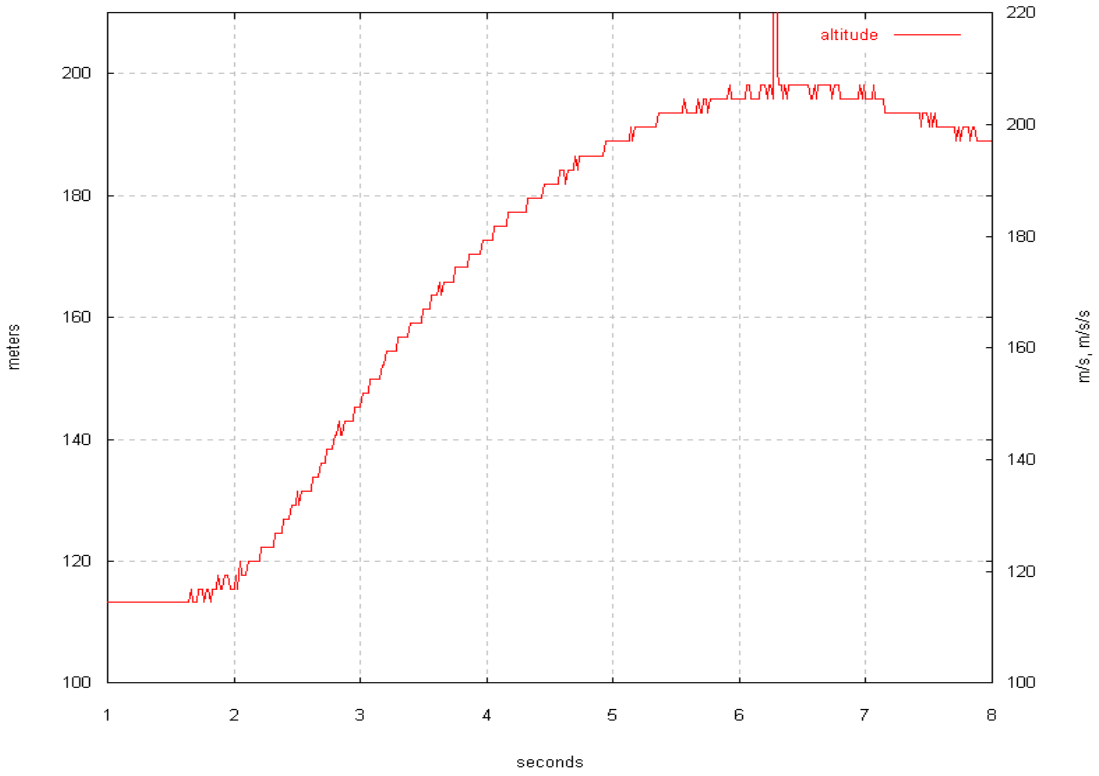

Figure 11, F39 flight test. Unfiltered altitude data.

The large spike in the data at apogee (Figure 11) was the result of the battery voltage collapsing under the load of the electric match. Because of this I modified the altimeter hardware by adding a 1000 ohm resistor and 1000uF capacitor to provide power to the electric match.

The next flight used a G80 motor. This flight was also perfect except that it was harder to see visually if deployment happened at apogee because of the increased altitude. Data is shown in Figures 12 and 13.

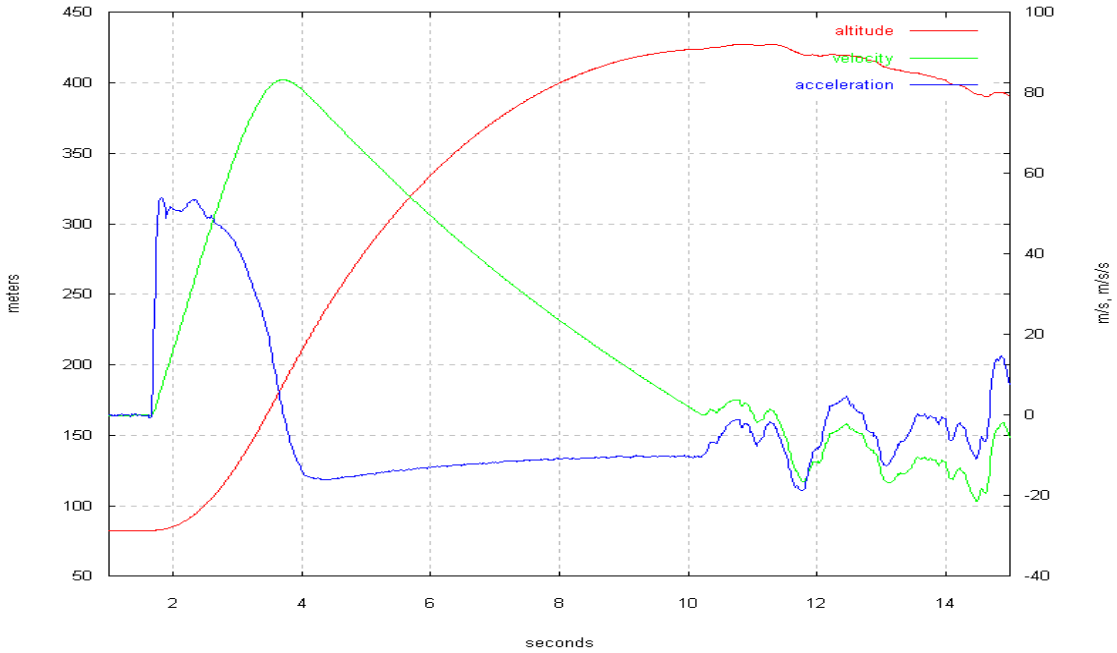

Figure 12, G80 flight test data

The filtered acceleration data for this flight looks much more like the unfiltered data. This isn't a big deal since the unfiltered data is always available. But it shows that the gains can be changed to a level that provides very little filtering of the acceleration and still provide excellent apogee timing. It also will help remove transonic pressure effects because the velocity will match the actual speed more closely.

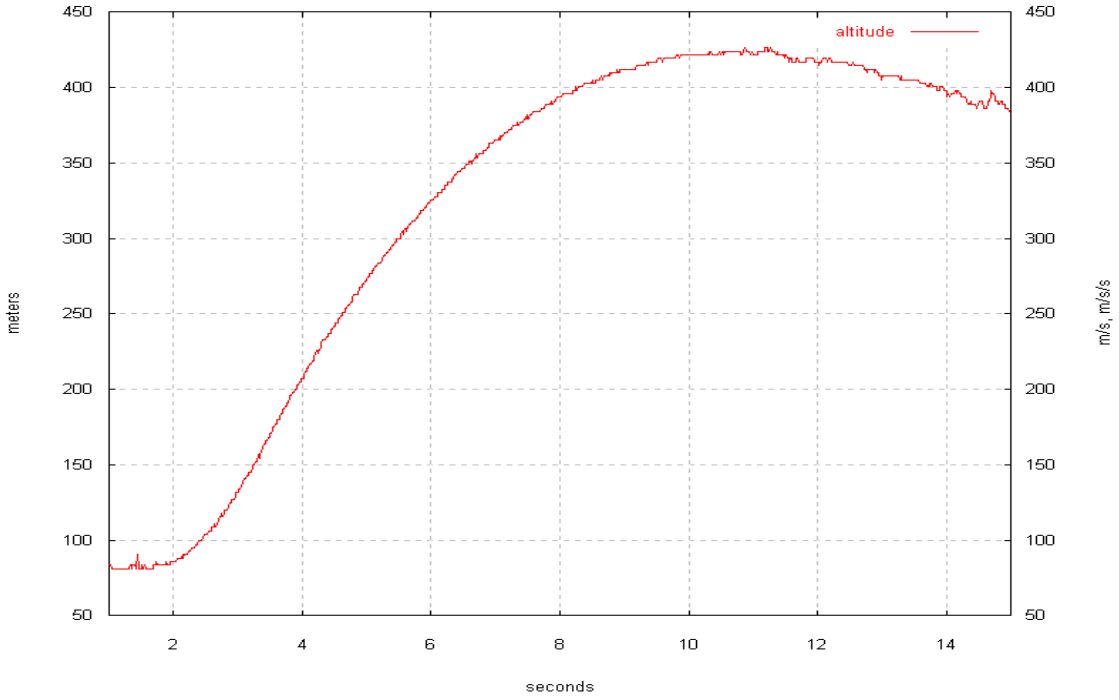

Figure 13, G80 flight test data. Unfiltered altitude.

The altitude data (Figure 13) shows a slight increase in altitude after the ejection charge was fired. This was the result of the ejection event pushing the payload bay higher and not because of an early ejection.

This flight verifies that the Kalman filter software is working and producing the expected results. Some portions of the code have not been tested that deal with high speed and altitude flights. I arranged for the altimeter to ride on a rocket that should have tested these portions of the code but the rocket motor suffered a nozzle failure and only achieved an altitude of 6,000 feet. An operational error resulted in the altimeter detecting liftoff as the rocket was raised to the vertical launch position so no flight data was recorded.

But even without testing these portions of the code, it is obvious that the Kalman filter is functioning as designed and providing excellent apogee detection.

Kalman Filter "Lite"

I had not planned on producing a version of flight code for a pressure data only filter because I felt that the full version accomplished my goal of demonstrating that the Kalman

filter could be run on a common micro-controller. But then I had an idea for reducing the computational load even more.

The primary computational load from the filter is the result of having to perform eight 32X16 multiplies. Using only the pressure measurement reduces this to three but I wondered if I could eliminate those last three multiplies. The filter has shown itself to be very stable and robust so I decided to see what would happen if I rounded the Kalman gains to the nearest power of two. This would let me use bit shifts instead of multiplies and really speed up the code.

A quick check on recorded data showed that performance was not seriously effected so I proceeded with producing flight code. This mostly involved hacking out bits of the existing code that were no longer required. I discovered that certain sections of the code were hogging most of the execution time and it wasn't the Kalman filter.

The micro-controller must communicate with two devices by serial interfaces (the ADC and data EEPROM). The serial conversions are done in software ("bit banged") and are therefore a lot slower than hardware. A lot of time was being wasted in these routines.

These routines were based on sample code provided by Robert DeHate for the Roctronics hardware. I converted the "C" code he provided into assembly language while trying to keep the same timing. Once they were working I didn't pay much attention to them.

I checked the data sheets for the parts and removed unnecessary delays. But even with these changes the Kalman filter code still required far less time than the rest of the code. The final version of the Kalman filter "lite" code requires about 200 microseconds (or 200 instructions) to execute. The remainder of the code (ADC conversion, data storage, etc.) requires another 3.4 milliseconds.

I flight tested the firmware twice. The first flight was data only and the second flight had the altimeter deploy the parachute at apogee.

The first flight used a G64. Analysis of the data revealed two problems. The first was that the Kalman gains were too high. I traced this to a flaw in the program to compute the gains that I thought I had fixed. The gains sometimes appear to converge to stable values after only a few iterations. But if more iterations are run they will change and then converge on the final values. I added code to make sure that at least 1000 iterations are performed and this resulted in much better gains. The high gains made the filter very sensitive as shown in Figure 14.

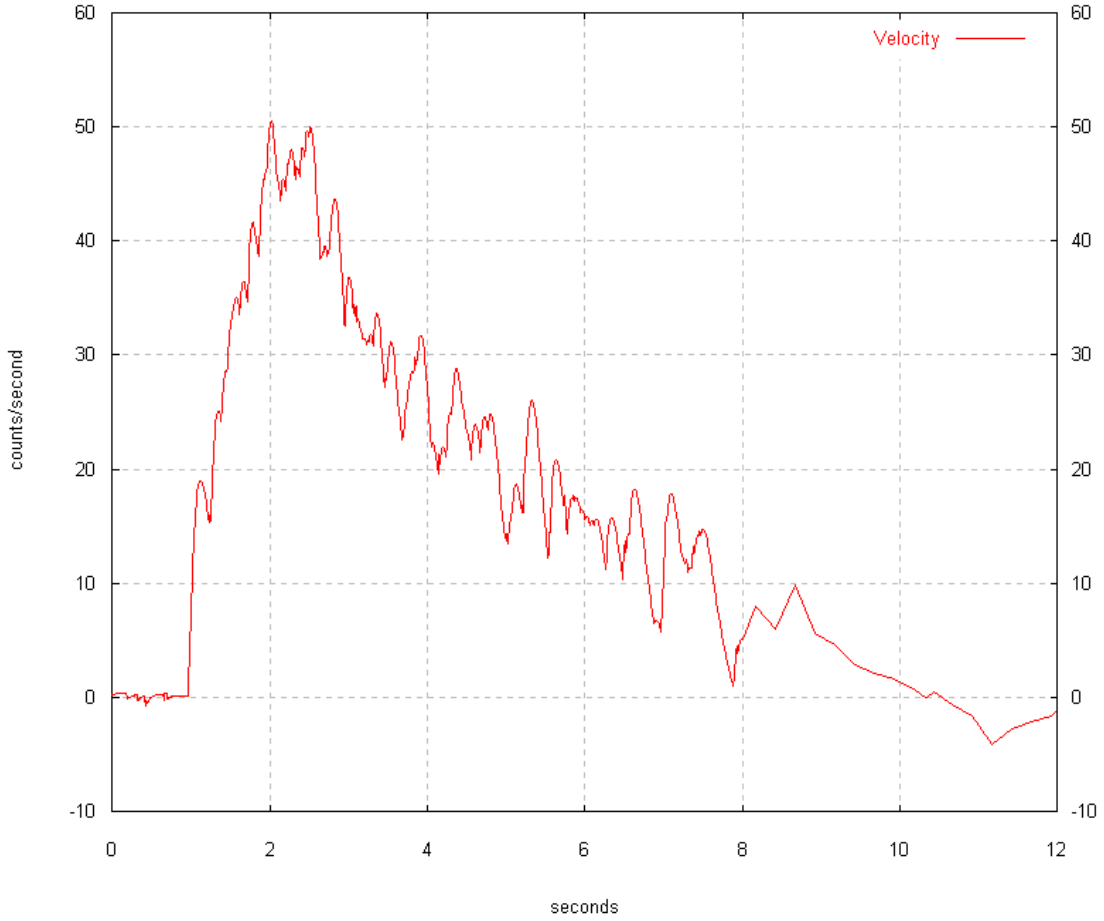

Figure 14, Kalman filter "lite" velocity plot from first flight test.

The velocity very nearly dropped below zero at eight seconds. If it had, it would result in a very early apogee event. The filter did get the correct apogee time but this was obviously not acceptable. Incorrect Kalman gains were part of the problem and the other part was that the data from the pressure sensor was terrible.

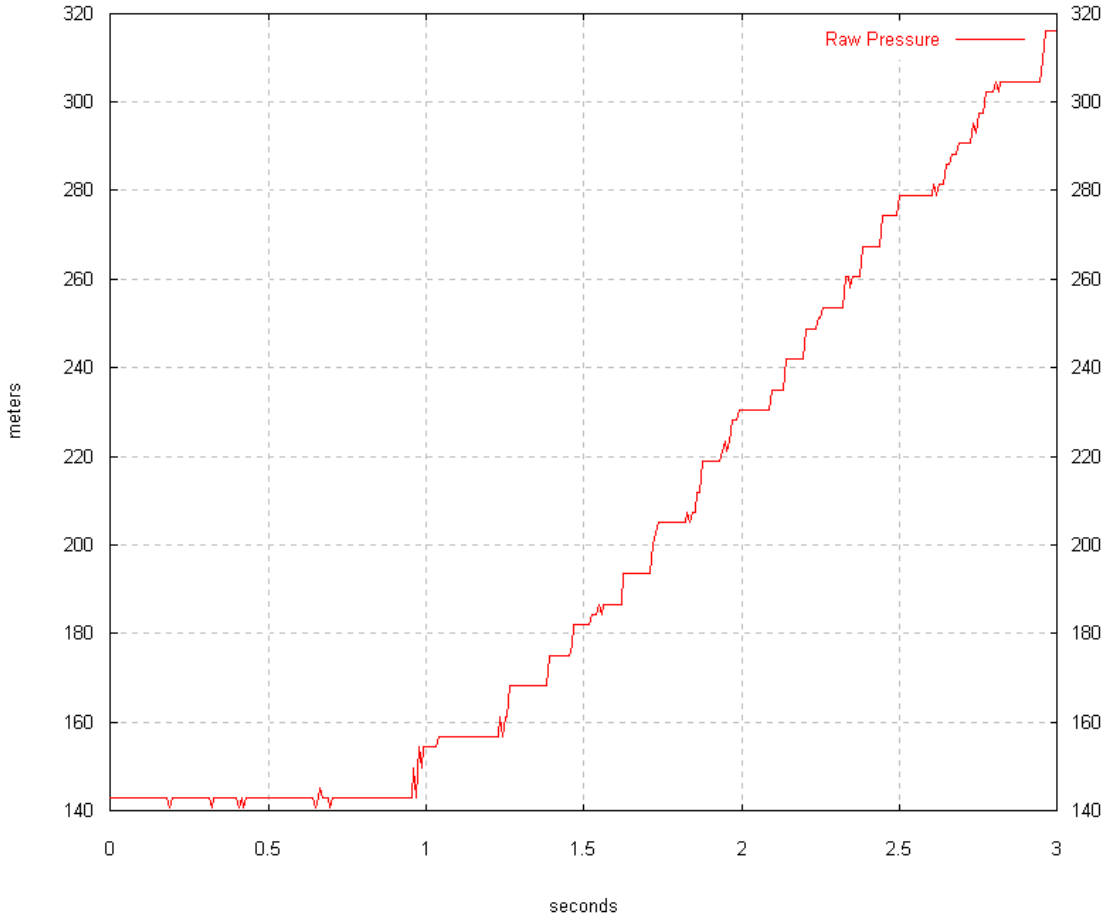

Figure 15, Unfiltered pressure data

The pressure data (Figure 15) shows that the ADC is skipping codes resulting in very jumpy data. Even worse is that it sticks to one value for significant periods of time. Stable values indicate zero velocity and that is where the filter goes. Earlier flights did not show this problem so it must be the result of the changes I made to the serial communication routines for the ADC. Which is very strange because all of the timing is well within the specifications of the ADC.

Resolving this problem is difficult because I cannot easily simulate the pressure profile of a flight. The best that I can do is to pull a vacuum using a FoodSaver. After playing around a little bit, I decided to go ahead with another flight test. This time flying on an F39 and with an apogee deployment charge.

This flight was nearly vertical and I was very disappointed to see the rocket do a tail slide for just a moment before the ejection event. The altimeter had deployed the parachute late. Analysis of the data showed that while the change to the Kalman gains helped a lot, the pressure data was still messed up and caused the late deployment.

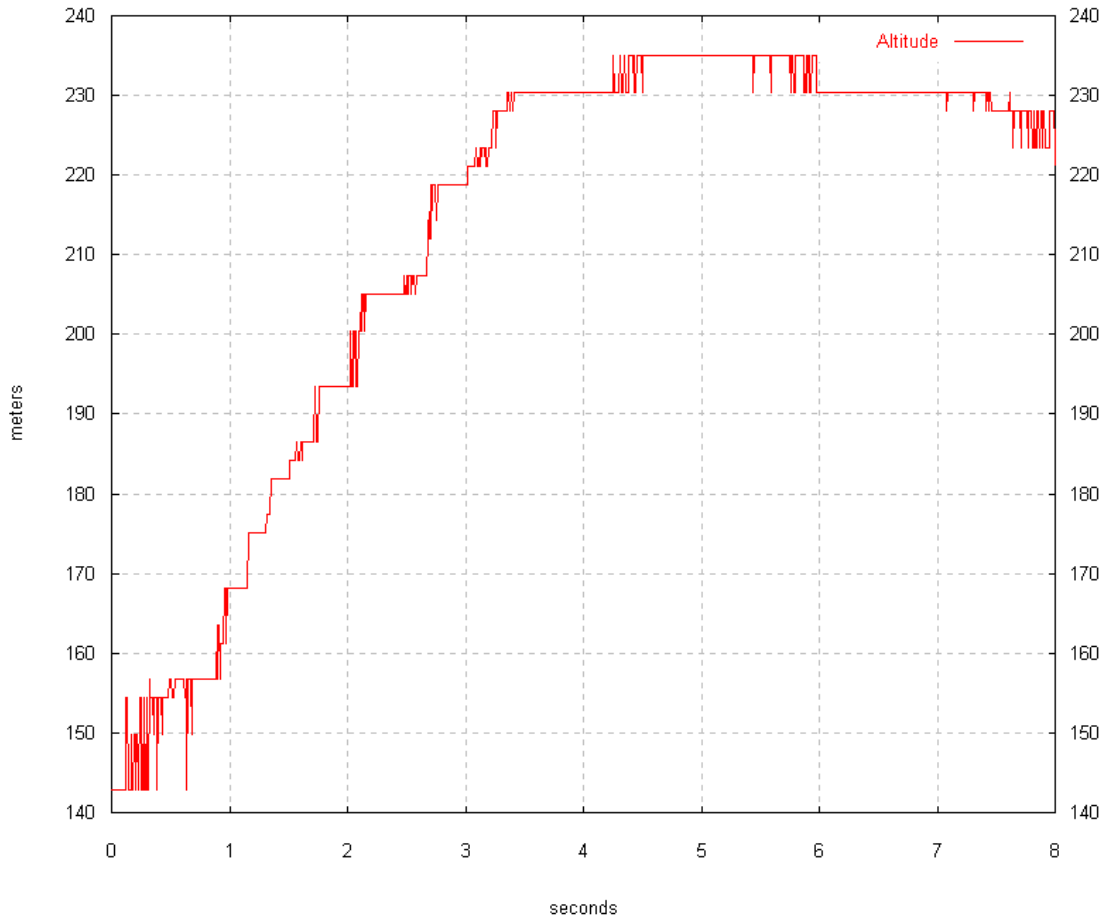

Figure 16, Unfiltered pressure data

If anything, this data (Figure 16) is worse. I suspect that there is some aspect of the ADC's operation that is not covered in the data sheets. I do not like it that I do not understand the problem in the ADC code but at this point the only solution is to return to my original slow version.

But the Kalman filter still exhibited reasonable performance even with this miserable data. It was late but not excessively late. I am certain that if the data was of the same quality as that shown in Figure 11, deployment would have been exactly at apogee.

Because of the way that pre-flight data buffering is performed, launch detect is at the 128 data sample point or 1 second for this flight. Launch detect was about 1 second after first motion. Launch detect is relatively slow because low model noise and the resulting low Kalman gains are required to prevent Mach transition and other pressure data anomalies from causing early launch detect.

Conclusion

The Kalman filter demonstrated excellent performance with recorded data. Even with data sets that cause problems for barometric and acceleration based altimeters. The filters velocity estimate consistently and accurately indicates the time of apogee. Adding the acceleration measurement to the filter provided a slightly better apogee time. Additional advantages are a quicker launch detect and the ability to fall back to acceleration measurements when the pressure sensor range is exceeded with minimal loss of performance.

A common micro-controller was easily able to handle the required computations of both a two measurement and single measurement filter. Thus demonstrating that the Kalman filter can provide better performance at apogee detection than existing apogee algorithms and that it can be used in hardware typical of commercial altimeters.

Appendix A
Kalman filter program for RDAS flight data

```
/*


 NAME
      rkal32.c -  A third order Kalman filter for
                  RDAS interpreted data files.

 SYNOPSIS
      rkal32 [model] <infile >outfile

                  model is an optional parameter to specify the ratio
                  between the variance for the acceleration measurement
                  (fixed) and the model (variable). Default value for
                  this is that the model is about 100 times better
                  than the measurement.

 DESCRIPTION:
      Performs Kalman filtering on standard input
      data using a third order, or constant acceleration,
      propagation model.


      The standard input data is of the form:

      Column 1: Time of the measurement (seconds)
      Column 2: Acceleration Measurement (feet/sec/sec)
      Column 3: Pressure Measurement (altitude in feet)

      The standard output data is of the form:

      Column 1: Time of the estimate/measurement
      Column 2: Pressure Measurement
      Column 3: Acceleration Measurement
      Column 4: Position estimate
      Column 5: Rate estimate
                  Column 6: Acceleration estimate

 AUTHOR
                    Hacked by David Schultz


 REPORTING BUGS AND REVISIONS
 mailto:david.schultz@earthlink.net

 COMPILING
   cc -o rkal32.exe rkal32.c

*/



#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define  ALTITUDESIGMA        15.0
#define  ACCELERATIONSIGMA     6.0
#define  MODELSIGMA                    0.6

double altitude_variance = ALTITUDESIGMA*ALTITUDESIGMA;
double acceleration_variance = ACCELERATIONSIGMA*ACCELERATIONSIGMA;
double model_variance  = MODELSIGMA*MODELSIGMA;

main(int argc, char ** argv)
{
                  char              buf[512];
                  int     i, j, k, notdone;

                  double alt_inovation, accel_inovation;
                  double time, accel, pressure;
                  double last_time, last_pressure;
                  double det;
                  double            est[3] = { 0, 0, 0 };
                  double  estp[3] = {0, 0, 0 };
                  double            pest[3][3]    = { 2, 0, 0,
                                                      0, 9, 0,
                                                      0, 0, 9 };
                  double  pestp[3][3]   = { 0, 0, 0,
                                                      0, 0, 0,
                                                      0, 0, 0 };
                  double            phi[3][3]     = { 1, 0, 0,
                                                      0, 1, 0,
                                                      0, 0, 1.0 };
                  double  phit[3][3]    = { 1, 0, 0,
                                                      0, 1, 0,
                                                      0, 0, 1.0 };
```

```
                double  kgain[3][2] = { 0.01, 0.01,
                                                        0.01, 0.01,
                                        0.01, 0.01 };
                double  lastkgain[3][2], dt;
                double          term[3][3];

                /* check for model variance parameter */
                if (argc == 2)
                {
                                double temp;
                                temp = atof(argv[1]);
                                model_variance = acceleration_variance/temp;
                }

/* Initialize */

                /* Skip text at start of file. */

                while(1)
                {
                                if(gets(buf) == NULL)
                                {
                                                fprintf(stderr, "No data in file\n");
                                                exit(1);
                                }
                                else
                                {
                                                if( strstr(buf, "-2.000"))
                                                                break;
                                }
                }
                sscanf(buf, "%lf %lf %lf", &time, &accel, &pressure);

                est[0] = pressure;
                last_time = time;

                if(gets(buf) == NULL)
                {
                                fprintf(stderr, "No data\n");
                                exit(1);
                }
                sscanf(buf, "%lf %lf %lf", &time, &accel, &pressure);

        dt = time - last_time;

                last_time = time;
/*
 Fill in state transition matrix and its transpose
*/
        phi[0][1]    = dt;
        phi[1][2]    = dt;
        phi[0][2]    = dt*dt/2.0;
        phit[1][0]    = dt;
        phit[2][1]    = dt;
        phit[2][0]    = dt*dt/2.0;


        /* Compute the Kalman gain matrix. */
        for( i = 0; i <= 2; i++)
                        for( j = 0; j <=1; j++)
                           lastkgain[i][j] = kgain[i][j];
        k = 0;

        while(1)
                        {
                         /* Propagate state covariance */

                         term[0][0] = phi[0][0] * pest[0][0] + phi[0][1] * pest[1][0] + phi[0][2] * pest[2][0];
                         term[0][1] = phi[0][0] * pest[0][1] + phi[0][1] * pest[1][1] + phi[0][2] * pest[2][1];
                         term[0][2] = phi[0][0] * pest[0][2] + phi[0][1] * pest[1][2] + phi[0][2] * pest[2][2];
                         term[1][0] = phi[1][0] * pest[0][0] + phi[1][1] * pest[1][0] + phi[1][2] * pest[2][0];
                         term[1][1] = phi[1][0] * pest[0][1] + phi[1][1] * pest[1][1] + phi[1][2] * pest[2][1];
                         term[1][2] = phi[1][0] * pest[0][2] + phi[1][1] * pest[1][2] + phi[1][2] * pest[2][2];
                         term[2][0] = phi[2][0] * pest[0][0] + phi[2][1] * pest[1][0] + phi[2][2] * pest[2][0];
                         term[2][1] = phi[2][0] * pest[0][1] + phi[2][1] * pest[1][1] + phi[2][2] * pest[2][1];
                         term[2][2] = phi[2][0] * pest[0][2] + phi[2][1] * pest[1][2] + phi[2][2] * pest[2][2];

                         pestp[0][0] = term[0][0] * phit[0][0] + term[0][1] * phit[1][0] + term[0][2] * phit[2][0];
                         pestp[0][1] = term[0][0] * phit[0][1] + term[0][1] * phit[1][1] + term[0][2] * phit[2][1];
                         pestp[0][2] = term[0][0] * phit[0][2] + term[0][1] * phit[1][2] + term[0][2] * phit[2][2];
                         pestp[1][0] = term[1][0] * phit[0][0] + term[1][1] * phit[1][0] + term[1][2] * phit[2][0];
                         pestp[1][1] = term[1][0] * phit[0][1] + term[1][1] * phit[1][1] + term[1][2] * phit[2][1];
                         pestp[1][2] = term[1][0] * phit[0][2] + term[1][1] * phit[1][2] + term[1][2] * phit[2][2];
                         pestp[2][0] = term[2][0] * phit[0][0] + term[2][1] * phit[1][0] + term[2][2] * phit[2][0];
                         pestp[2][1] = term[2][0] * phit[0][1] + term[2][1] * phit[1][1] + term[2][2] * phit[2][1];
                         pestp[2][2] = term[2][0] * phit[0][2] + term[2][1] * phit[1][2] + term[2][2] * phit[2][2];
```

```
                pestp[2][2] = pestp[2][2] + model_variance;
                /*
                 Calculate Kalman Gain
                */
                det = (pestp[0][0]+altitude_variance)*(pestp[2][2] + acceleration_variance) - pestp[2][0] * pestp[0][2];


                kgain[0][0] = (pestp[0][0] * (pestp[2][2] + acceleration_variance) - pestp[0][2] * pestp[2][0])/det;

                kgain[0][1] = (pestp[0][0] * (-pestp[0][2]) + pestp[0][2] * (pestp[0][0] + altitude_variance))/det;

                kgain[1][0] = (pestp[1][0] * (pestp[2][2] + acceleration_variance) - pestp[1][2] * pestp[2][0])/det;

                kgain[1][1] = (pestp[1][0] * (-pestp[0][2]) + pestp[1][2] * (pestp[0][0] + altitude_variance))/det;

                kgain[2][0] = (pestp[2][0] * (pestp[2][2] + acceleration_variance) - pestp[2][2] * pestp[2][0])/det;

                kgain[2][1] = (pestp[2][0] * (-pestp[0][2]) + pestp[2][2] * (pestp[0][0] + altitude_variance))/det;

    pest[0][0] = pestp[0][0] * (1.0 - kgain[0][0]) - kgain[0][1]*pestp[2][0];
    pest[0][1] = pestp[0][1] * (1.0 - kgain[0][0]) - kgain[0][1]*pestp[2][1];
                pest[0][2] = pestp[0][2] * (1.0 - kgain[0][0]) - kgain[0][1]*pestp[2][2];
    pest[1][0] = pestp[0][0] * (-kgain[1][0]) + pestp[1][0] - kgain[1][1]*pestp[2][0];
    pest[1][1] = pestp[0][1] * (-kgain[1][0]) + pestp[1][1] - kgain[1][1]*pestp[2][1];
                pest[1][2] = pestp[0][2] * (-kgain[1][0]) + pestp[1][2] - kgain[1][1]*pestp[2][2];
    pest[2][0] = (1.0 - kgain[2][1]) * pestp[2][0] - kgain[2][0] * pestp[2][0];
    pest[2][1] = (1.0 - kgain[2][1]) * pestp[2][1] - kgain[2][0] * pestp[2][1];
                pest[2][2] = (1.0 - kgain[2][1]) * pestp[2][2] - kgain[2][0] * pestp[2][2];

                /* Check for convergance. Criteria is less than .001% change from last
                 * time through the mill.
                 */
                notdone = 0;
                k++;
                for( i = 0; i <= 2; i++)
                  for( j = 0; j <= 1; j++)
                    {
                      if( (kgain[i][j] - lastkgain[i][j])/lastkgain[i][j] > 0.00001)
                                notdone++;
                      lastkgain[i][j] = kgain[i][j];
                    }
                if( notdone )
                  continue;
                else
                  break;
                }
    printf("Input noise values used (standard deviation):\n");
    printf("#Altitude       - %15f feet\n", sqrt(altitude_variance));
    printf("#Acceleration   - %15f feet/sec/sec\n", sqrt(acceleration_variance));
    printf("#Model noise    - %15f feet/sec/sec\n#\n", sqrt(model_variance));
    printf("#Kalman gains converged after %d iterations.\n#", k);
    for( i = 0; i <= 2; i++)
                for( j = 0; j <= 1; j++)
                  printf("%15f ", kgain[i][j]);
    printf("\n#\n");
    printf("#Estimated output first order statistics (standard deviation):\n");
    printf("#Altitude     - %15f feet\n", sqrt(pest[0][0]));
    printf("#Velocity     - %15f feet/sec\n", sqrt(pest[1][1]));
    printf("#Acceleration - %15f feet/sec/sec\n", sqrt(pest[2][2]));

    /* Now run the Kalman filter on the data using previously
     * determined gains.
     */
/*
                Output header for data.
*/
                printf("#\n# Output from rkal32:\n# A third order Kalman filter using acceleration and pressure measurements\n");
                printf("#     Time          Press. Alt.     Acceleration    Est Pos         Est Rate        Est Accel\n#\n");

    while( gets(buf) != NULL)
                {
                  sscanf(buf, "%lf %lf %lf", &time, &accel, &pressure);

                  /* remove offset and convert from G'  s to ft/sec/sec */
                  accel = (accel-1.0)*32.17417;

                  /* sanity check on time */
                  if( last_time >= time )
                    {
                      fprintf(stderr, "Time does not increase.\n");
                      exit(1);
                    }

                  /* Compute the innovations */

                  alt_inovation = pressure - estp[0];
                  accel_inovation = accel - estp[2];
```

```
                /* Experimental code to modify Mach transition pressure
                 * disturbances.
                 */

                if( abs(alt_inovation) > 100 )
                  {
                    /* We have a large error in altitude. Now see how fast we are
                     * going.
                     */
                    if( estp[1] > 900 && estp[1] < 1200 )
                                {
                                 /* Somewhere in the neighborhood of Mach 1. Now check to
                                  * see if we are slowing down.
                                  */
                                 //                                    if( estp[2] < 0 )
                                   {
                                   /*
                                    * OK, now what do we do? Assume that velocity and
                                    * acceleration estimates are accurate. Adjust current
                                    * altitude estimate to be the same as the measured
                                    * altitude.
                                    */
                                   est[0] = pressure;
                                   alt_inovation = 0;
                                   }
                                }
                  }
                /* Simple check for over-range on pressure measurement.
                 * This is just hacked in based on a single data set. Actual
                 * flight software needs something more sophisticated.
                 */
                if( pressure > 37000)
                  alt_inovation = 0;


                /* Propagate state */

                estp[0] = phi[0][0] * est[0] + phi[0][1] * est[1] + phi[0][2] * est[2];
                estp[1] = phi[1][0] * est[0] + phi[1][1] * est[1] + phi[1][2] * est[2];
                estp[2] = phi[2][0] * est[0] + phi[2][1] * est[1] + phi[2][2] * est[2];


/*
 Update state
*/
        est[0] = estp[0] + kgain[0][0] * alt_inovation + kgain[0][1] * accel_inovation;

        est[1] = estp[1] + kgain[1][0] * alt_inovation + kgain[1][1] * accel_inovation;

        est[2] = estp[2] + kgain[2][0] * alt_inovation + kgain[2][1] * accel_inovation;


/*
 Output
*/

                printf("%15f %15f %15f %15f %15f %15f\n", time, pressure, accel, est[0], est[1], est[2]);

                last_time = time;

                }
}
```

Appendix B
Kalman filter code for PIC 16F628 altimeter

```
include p16f628.inc
include data.inc
include adc.inc
include eedata.inc

global      KALMAN_STEP,kalman_init,set_alt_gains
extern      FXM3216S, get_altitude, get_acceleration, eeread
extern      beep_alt

;; Copyright 2002 David W. Schultz
;;
;; Quick and dirty version of Kalman filter for PIC16 processors.
;; Uses 32X16 multiply routine extracted from Microchip application
;; note.
;;
;; There may be something else hiding in the multiply code but
;; I think I found all of them. That adds up to 32 locations.
;;
;; This version incorporates both pressure and acceleration
;; measurements. It uses ~330 words of code space. Execution time
;; is dominated by the 32x16 bit multiplies which require ~400 machine
;; cycles each. With 6 multiplies that means 2400 machine cycles.
;; Double that to allow for everything else that goes on and you have
;; a requirement for ~5000 machine cycles per sample. Since the PIC
;; has four clock cycles per machine cycle that it 20,000 clocks. A
;; 4MHz clock would allow a maximum of 200 samples/second. So there is
;; plenty of margin when running at 128 samples/second. Even more at
;; 64SPS which is what this code is set to.
;;
;; The primary limitation is code and data space availability. The
;; 16F84 has 1024 words code space and 68 bytes data space. This code
;; takes 309 words of code space. Could be pretty
;; tight. The 16F628 which has 2K of code space would be a better
;; choice. Especially since it is pin compatible with the 16F84, has
;; more features, more code space, more data space, and is cheaper.
;;
;; There are some fairly obvious opportunities to tighten up this code.
;; I was primarily interested in getting it all down.
;;
;;
;; Changing the sample rate requires changing the number of bits
;; to shift in the state update code. It is currently set for
;; 64 SPS
;;
;; Kalman gains are computed by a "C" application and are determined
;; by sample rate, measurement noise, and model noise. To compute
;; gains, select sample rate, measure sensor noise
;; (standard deviation), and select measurement/model noise ratio. I
;; recomend a ratio of 10 to 100. Input parameters into kgain32.exe and
;; note results. Gains shown here are based on 64 SPS, noise of 15'
;; for pressure altitude, 6ft/sec/sec for acceleration, and  a noise
;; ratio of 100
;;
;; Initialization:
;;
;; Clear VELB0-VELB3 and ACCELB0-ACCELB3. Read altitude value and load
;; into ALTB0-3. Then you can start running the filter.
;;
;; If you start the filter with random garbage for the intial values
;; you will get random garbage out for quite a while.
;;
;; A note on number representation.
;;
;; Altitude, velocity, and acceleration are stored in 16.16 fixed
;; point format. The two most significant bytes represent the integer
;; portion of the number and the two LSB'  s the fractional portion. It'  s
;; just like base ten if you are missing 8 fingers. :-)
;;
;; The Kalman gains are only 16 bit values because the integer part is
;; zero. This allows use of a 32X16 multiply with 48 bit result instead
;; of a 32x32 bit multiply with 64 bit result. It shouldn'  t be a
;; problem but there is a chance that at very low sample rate, one
;; of the gains could be larger than one. You have two choices:                1)
;; increase sample rate. 2) Modify number representation and code.
;;
;;
;; Buyer beware.
;; Batteries not included.
;; Your mileage may vary.
;;


code

;; Initialization of the Kalman filter is pretty simple. Just set
;; the velocity and acceleration to zero. Then get the current altitude
```

```
                ;; from the pressure sensor to set altitude.
                ;;
kalman_init:
                clrf        VELB0
                clrf        VELB1
                clrf        VELB2
                clrf        VELB3
                clrf        ACCELB0
                clrf        ACCELB1
                clrf        ACCELB2
                clrf        ACCELB3


                ;; Read current altitude

                call        get_altitude    ; result is in T0-3

                movf        T0,W                    ; copy to altitude state variable
                movwf       ALTB0
                movf        T1,W
                movwf       ALTB1
                movf        T2,W
                movwf       ALTB2
                movf        T3,W
                movwf       ALTB3
                ;;
                ;; Initialize accelerometer averages
                ;;
                movlw       SECH0                   ; read value from ADC
                movwf       T0
                call        readADC


                movf        T0,W
                bsf         STATUS,RP0              ; date page 1
                movwf       AVE_ACCEL0B0
                movwf       AVE_ACCEL1B0
                movwf       AVE_ACCEL2B0
                bcf         STATUS,RP0              ; data page 0
                movf        T1,W
                bsf         STATUS,RP0              ; date page 1
                movwf       AVE_ACCEL0B1
                movwf       AVE_ACCEL1B1
                movwf       AVE_ACCEL2B1
                clrf        AVE_ACCEL0B2
                clrf        AVE_ACCEL1B2
                clrf        AVE_ACCEL2B2
                clrf        AVE_ACCEL0B3
                clrf        AVE_ACCEL1B3
                clrf        AVE_ACCEL2B3
                bcf         STATUS,RP0              ; data page 0

                ;; Set Kalman gains

                movlw       K1_HI
                call        eeread
                movwf       K1B0
                movlw       K1_LO
                call        eeread
                movwf       K1B1

                movlw       K2_HI
                call        eeread
                movwf       K2B0
                movlw       K2_LO
                call        eeread
                movwf       K2B1

                movlw       K3_HI
                call        eeread
                movwf       K3B0
                movlw       K3_LO
                call        eeread
                movwf       K3B1

                movlw       K4_HI
                call        eeread
                movwf       K4B0
                movlw       K4_LO
                call        eeread
                movwf       K4B1

                movlw       K5_HI
                call        eeread
                movwf       K5B0
                movlw       K5_LO
                call        eeread
```

```
                movwf   K5B1

                movlw   K6_HI
                call    eeread
                movwf   K6B0
                movlw   K6_LO
                call    eeread
                movwf   K6B1

                ;; set timer tick to zero
                clrf    TICKB0
                clrf    TICKB1
                clrf    TICKB2

                return

;;
;;              Set Kalman gains to the alternate, pressure sensor only, gain set.
;;              This is called at apogee since the orientation of the acclerometer
;;              becomes pretty random at that point.
;;

set_alt_gains:

                movlw   ALTK1_HI
                call    eeread
                movwf   K1B0
                movlw   ALTK1_LO
                call    eeread
                movwf   K1B1

                movlw   ALTK2_HI
                call    eeread
                movwf   K2B0
                movlw   ALTK2_LO
                call    eeread
                movwf   K2B1

                movlw   ALTK3_HI
                call    eeread
                movwf   K3B0
                movlw   ALTK3_LO
                call    eeread
                movwf   K3B1

                clrf    K4B0
                clrf    K4B1
                clrf    K5B0
                clrf    K5B1
                clrf    K6B0
                clrf    K6B1

                return


                ;;  Multiply/accumulate code. This routine is used by the state
                ;;  correction code.
                ;;
                ;;  Before calling, T0-T3 should contain the inovation. BARG should
                ;;  be loaded with the Kalman gain, and FSR should point to the
                ;;  location to add the result to.
                ;;
                ;; NOTE: MSB of multi-precision numbers is in xxxB0 which has the
                ;; lowest address. FSR should point to the LSB (xxxB3) of the target.
                ;;
                ;;
KALMAC:         movf    T0,W                    ; Copy T0-T3 to AARG in preperation for multiply
                movwf   AARGB0
                movf    T1,W
                movwf   AARGB1
                movf    T2,W
                movwf   AARGB2
                movf    T3,W
                movwf   AARGB3

                call    FXM3216S    ; Perform the signed multiply

                ;; crap! I hate the fact that this stupid chip doesn'  t have an add
                ;; with carry

                movf    AARGB3,W    ; add upper 32 bits of result to altitude
                addwf   INDF,F

                decf    FSR,F                   ; point at next byte
                movf    AARGB2,W
                btfsc   STATUS,C    ; handle the carry
                incfsz  AARGB2,W
```

```
                addwf   INDF,F

                decf    FSR,F
                movf    AARGB1,W
                btfsc   STATUS,C
                incfsz  AARGB1,W
                addwf   INDF,F

                decf    FSR,F
                movf    AARGB0,W
                btfsc   STATUS,C
                incfsz  AARGB0,W
                addwf   INDF,F
                retlw   0



                ;;
                ;;
                ;; Arithmetic right shift 32 bit T0-T3, LOOPCOUNT places
                ;;
RSTEMP:         movf    T0,W                    ; Copy MSB to use for sign extension
                movwf   SIGN
RSLOOP:         rlf     SIGN,W                  ; Pick up the sign bit in C
                rrf     T0,F
                rrf     T1,F
                rrf     T2,F
                rrf     T3,F
                decfsz  LOOPCOUNT,F
                goto    RSLOOP
                retlw   0

                ;;
                ;; Begin state update
                ;;
                ;; Start by computing Xt+1 = X + V*T + A*T*T/2
                ;;
                ;; Since T is a fractional power of two, we do this by shifting bits
                ;;
                ;; First make a temporary copy of the last velocity estimate and
                ;; divide the copy by 64.
                ;;
KALMAN_STEP:            movf    VELB3,W                 ; Copy velocity to temp location
                movwf   T3
                movf    VELB2,W
                movwf   T2
                movf    VELB1,W
                movwf   T1
                movf    VELB0,W
                movwf   T0
                movlw   6                       ;  load count for right shift
                movwf   LOOPCOUNT
                call    RSTEMP                  ; shift temp right 6 bits

                ;; Now add it to the last altitude estimate (32 bit add)
                ;;
                movf    T3,W                    ; add temp to altitude
                addwf   ALTB3,F

                movf    T2,W
                btfsc   STATUS,C    ; handle the carry
                incfsz  T2,W
                addwf   ALTB2,F

                movf    T1,W
                btfsc   STATUS,C
                incfsz  T1,W
                addwf   ALTB1,F

                movf    T0,W
                btfsc   STATUS,C
                incfsz  T0,W
                addwf   ALTB0,F

                ;; Now make a temporary copy of the last acceleration estimate
                ;; and divide by 64*64*2 or shift right 13 bits
                ;;

                movf    ACCELB3,W ; Copy acceleration to temp location
                movwf   T3
                movf    ACCELB2,W
                movwf   T2
                movf    ACCELB1,W
                movwf   T1
                movf    ACCELB0,W
                movwf   T0
```

```
movlw   13                      ;  load count for right shift
movwf   LOOPCOUNT
call    RSTEMP                  ; shift temp right 13 bits

;; Now add it into the last altitude estimate (32 bit add)
;;
movf    T3,W                    ; add temp to altitude
addwf   ALTB3,F

movf    T2,W
btfsc   STATUS,C    ; handle the carry
incfsz  T2,W
addwf   ALTB2,F

movf    T1,W
btfsc   STATUS,C
incfsz  T1,W
addwf   ALTB1,F

movf    T0,W
btfsc   STATUS,C
incfsz  T0,W
addwf   ALTB0,F

;; State update of altitude is now complete
;;
;; Now we update the velocity estimate
;;
;; Make a temporary copy of the last acceleration estimate and
;; multiply by T, which is the same as shifting it right
;; 6 bits
;;
movf    ACCELB3,W ; Copy acceleration to temp location
movwf   T3
movf    ACCELB2,W
movwf   T2
movf    ACCELB1,W
movwf   T1
movf    ACCELB0,W
movwf   T0
movlw   6                       ;  load count for right shift
movwf   LOOPCOUNT
call    RSTEMP                  ; shift temp right 6 bits

;; Now add it to the velocity estimate
;;
movf    T3,W                    ; add temp to altitude
addwf   VELB3,F

movf    T2,W
btfsc   STATUS,C    ; handle the carry
incfsz  T2,W
addwf   VELB2,F

movf    T1,W
btfsc   STATUS,C
incfsz  T1,W
addwf   VELB1,F

movf    T0,W
btfsc   STATUS,C
incfsz  T0,W
addwf   VELB0,F


;; This completes the state update for velocity
;;
;; Acceleration is assumed to be constant so no update of it
;;  is required. Move on to the state correction.
;;
;; Begin state correction
;; A miracle occurs and the latest measured altitude (in meters)
;; appears in T0-T3 as a 16.16 fixed point number.

call    get_altitude    ; This takes care of the miracle

;;
;; First up is a sanity check. The code that reads the pressure
;; should set the carry flag if the range of the sensor is exceeded.
;; This is best checked by comparing the ADC value so we don'  t do it
;; here.
;; Therefore, we check the carry flag and if it is set, we skip all of
;; the pressure altitude inovation code.

btfsc   STATUS,C
goto    A_INOVAT    ;  carry set, move on to altitude inovation
```

```
                ;; Now we check to see if we are in the Mach transition region.
                ;; Mach 1 = ~331m/s

                movlw   #1                      ; MSB should be = 1
                xorwf   VELB0,W
                btfss   STATUS,Z
                goto    P_INOVAT    ; less than 256 or greater than 512 m/s, so go on
                ;; Now we know that our velocity is in the range of 256 to 512 m/s
                ;; An arbitrary range of Mach 1 +/- 75m/s is selected as the criteria.
                ;;  This allows us to complete the test with only one more comparison
                ;; since 331 m/s - 75 m/s = 256 m/s. So we just check VELB1 to see if
                ;; it is greater than 150.

                movf    VELB1,W
                addlw   (0x100 - 0x96)
                btfsc   STATUS,C    ;
                goto    P_INOVAT    ; result was positive so carry on

                ;; Copy measured pressure to estimate and then skip to acceleration
                ;; inovation part of code

                movf    T0,W
                movwf   ALTB0
                movf    T1,W
                movwf   ALTB1
                movf    T2,W
                movwf   ALTB2
                movf    T3,W
                movwf   ALTB3
                goto    A_INOVAT

                ;; Pressure inovation correction
                ;;
                ;; First step is to subtract altitude from pressure reading to
                ;; get the inovation. Since we need this value for three computations,
                ;; it is left in T0-T3 for later use. Do not mess with T0-T3!

P_INOVAT:
                movf    ALTB3,W
                subwf   T3,F

                movf    ALTB2,W
                btfss   STATUS,C
                incfsz  ALTB2,W
                subwf   T2,F

                movf    ALTB1,W
                btfss   STATUS,C
                incfsz  ALTB1,W
                subwf   T1,F

                movf    ALTB0,W
                btfss   STATUS,C
                incfsz  ALTB0,W
                subwf   T0,F

                ;; Altitude correction
                ;; Multiply the inovation (currently in T0-T3) by the appropriate
                ;; Kalman gain and add the altitude estimate
                ;; This is done by the KALMAC subroutine which copies to invation
                ;; from T0-T3 to AARG and then calls the 32X16 multiply routine. It
                ;; then adds the upeer 32 bits of the result to the altitude estimate.
                ;; We use the upper 32 bits because we have multiplied two number with
                ;; 16 bits each to the right of the binary point. Therefore the result
                ;; has 32 bits to the right of the binary point and we are dropping
                ;; the least significant 16 bits of the fractional part.
                ;;

                movf    K1B0,W          ;  Load Kalman gain
                movwf   BARGB0
                movf    K1B1,W
                movwf   BARGB1

                movlw   ALTB3
                movwf   FSR

                call    KALMAC          ; Perform the multiply/accumulate

                ;; Velocity correction

                movf    K2B0,W          ;  Load Kalman gain
                movwf   BARGB0
                movf    K2B1,W
                movwf   BARGB1

                movlw   VELB3
                movwf   FSR
```

```
                call    KALMAC          ; Perform multiply/accumulate

                ;;  Acceleration correction

                movf    K3B0,W          ; Load Kalman gain
                movwf   BARGB0
                movf    K3B1,W
                movwf   BARGB1

                movlw   ACCELB3
                movwf   FSR

                call    KALMAC

                ;; Once again a miracle occurs. This time the current acceleration
                ;; measurement (In m/s/s) appears in T0-T3. This value must be offset
                ;; so that while at rest on the pad, the acceleration is ZERO. It
                ;; is very important that this value be as close to zero as possible.
                ;; Fractional bits are really good so don'  t throw them away.
                ;;
                ;; This is just like the pressure inovation code except that we use
                ;; different gains.

                ;; First step is to subtract acceleration from measurement to
                ;; get the inovation. This is left in T0-T3 for later use.

A_INOVAT:
                call    get_acceleration ; handle the miracle bit

                movf    ACCELB3,W
                subwf   T3,F

                movf    ACCELB2,W
                btfss   STATUS,C
                incfsz  ACCELB2,W
                subwf   T2,F

                movf    ACCELB1,W
                btfss   STATUS,C
                incfsz  ACCELB1,W
                subwf   T1,F

                movf    ACCELB0,W
                btfss   STATUS,C
                incfsz  ACCELB0,W
                subwf   T0,F

                ;; Altitude correction

                movf    K4B0,W          ;  Load Kalman gain
                movwf   BARGB0
                movf    K4B1,W
                movwf   BARGB1

                movlw   ALTB3
                movwf   FSR

                call    KALMAC          ; Perform the multiply/accumulate

                ;; Velocity correction

                movf    K5B0,W          ;  Load Kalman gain
                movwf   BARGB0
                movf    K5B1,W
                movwf   BARGB1

                movlw   VELB3
                movwf   FSR

                call    KALMAC          ; Perform multiply/accumulate

                ;;  Acceleration correction

                movf    K6B0,W          ; Load Kalman gain
                movwf   BARGB0
                movf    K6B1,W
                movwf   BARGB1

                movlw   ACCELB3
                movwf   FSR

                call    KALMAC

                ;; Fine'
                ;; Done
                ;; Compleate
```

```
return
end
```

Appendix C
On Noise

The sensors used for pressure and acceleration data in altimeters generate random noise internally and also respond to external noise. This is both good and bad.

The good part is that internally generated random noise can be used as a dithering signal to increase the resolution of the measurement system. This reduces the quantization noise resulting from the ADC converting the continuous analog voltage produced by the sensor into a discrete digital value.

The signal produced by the sensor can be viewed as being composed of two parts. The first part being the true measurement and the second part being zero mean random noise. If the random noise is larger than the resolution of the ADC system, it will cause the ADC results to be dithered.

If, for example, the true value was exactly halfway between 512 and 513 counts in the ADC, the dithering would result in half of the converted values being 512 and half being 513. Averaging a large number of samples would result in the true value of 512.5

This technique is vital in measuring the 1G offset of the accelerometer for the velocity integration technique. It isn't quite as vital for a Kalman filter based altimeter as the pressure measurement will tend to correct for slight errors in accelerometer offset. But since this is easy to do, it should be done anyway.

Dithering of the pressure data is useful as well because if it were not present, the altitude data would rise in a strict stair-step fashion. The Kalman filter would then try to converge on each of these relatively constant values. With dithering, the Kalman filter produces a much smoother output. This can be clearly seen when comparing the Kalman filter output of the RDAS and AltAcc data.

While the random sensor noise is quite useful, noise generated externally and measured by the sensors can be extremely disruptive. The best example of this is the high vibration levels created by the Hypertek hybrid motors. This vibration has caused considerable difficulty for acceleration based altimeters. Particularly the RDAS.

The problem in the RDAS results from the complete lack of a pre-sample filter for the accelerometer. The ADXL150 sensor used has a built-in low pass filter but its 3dB point (bandwidth) is 1000Hz and the maximum sample rate of the RDAS is 200Hz. Sampling theory shows that the sampled signal must be band limited so that it has no frequency content in excess of one half the sampling frequency. This is also known as the Nyquist frequency.

If the signal does have frequency content greater than the Nyquist frequency, it will be aliased so that it appears to have a frequency lower than one half the sampling frequency.

For example, at the RDAS sample rate of 200SPS, a 102Hz sine wave would appear as a 98 Hz sine wave after sampling. There is absolutely no way to correct this after sampling.

This sort of aliasing is very bad for altimeters. In the worst case, the motor would vibrate at a frequency very close to the sampling frequency. This would cause the vibration to be aliased to a frequency near 0 Hz, or constant. If the vibration had sufficient amplitude, it would severely impact the operation of the altimeter. An example of this problem is shown in Figure 14.

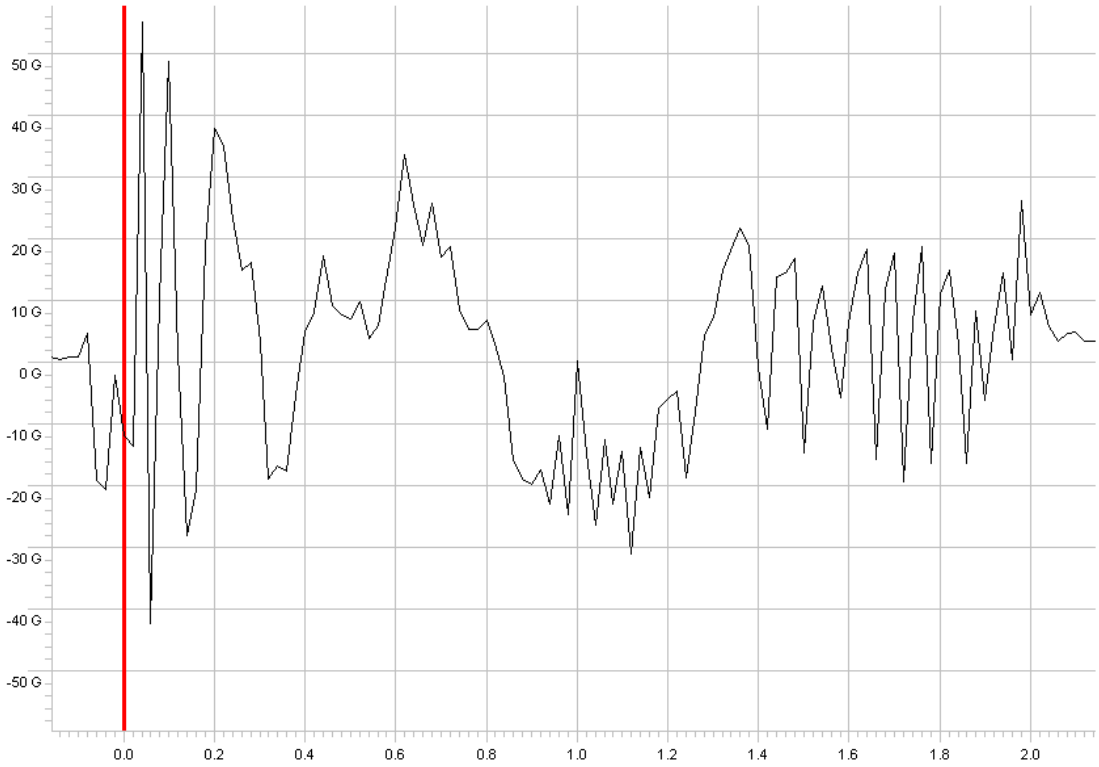

Figure 14, RDAS data from hybrid flight exhibiting extreme aliasing

This flight resulted in very early ejection from the RDAS because of aliasing. The RDAS had been configured to record data at 50SPS and this also apparently changed the sample rate for the apogee algorithm.

Figures 15 and 16 show the frequency content in 200SPS data recorded by an RDAS for a hybrid and composite motor. The composite motor acceleration data has almost no energy above 0Hz but the hybrid has energy all the way up to (and above) 100Hz.

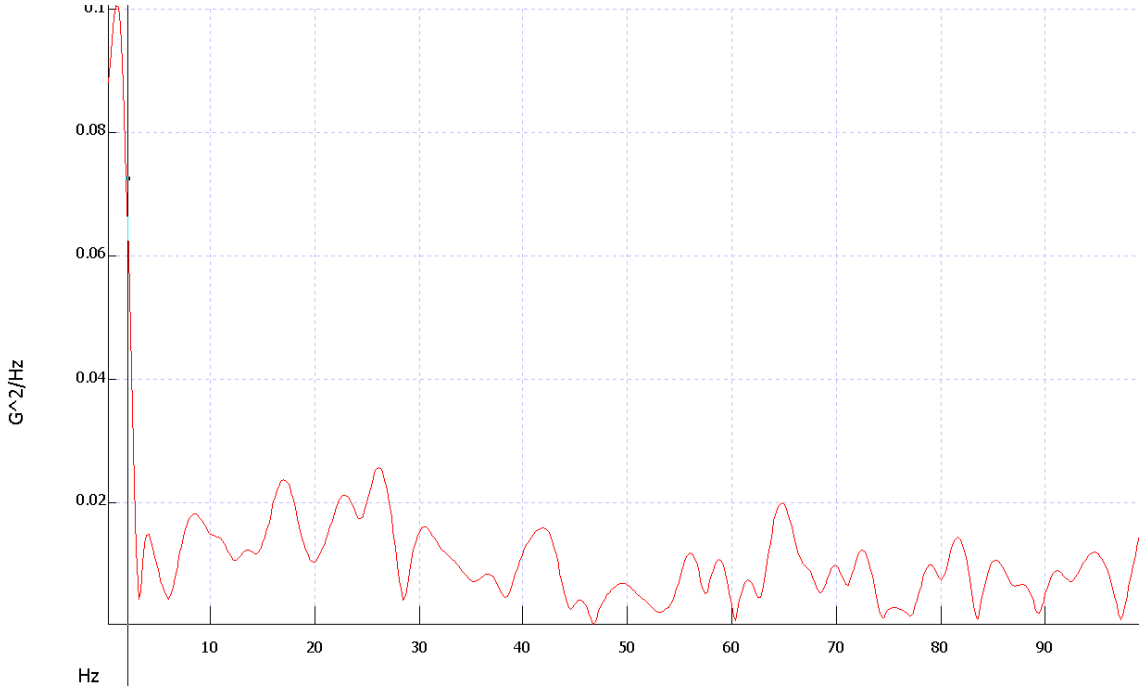

Figure 15, PSD for 'M' class composite

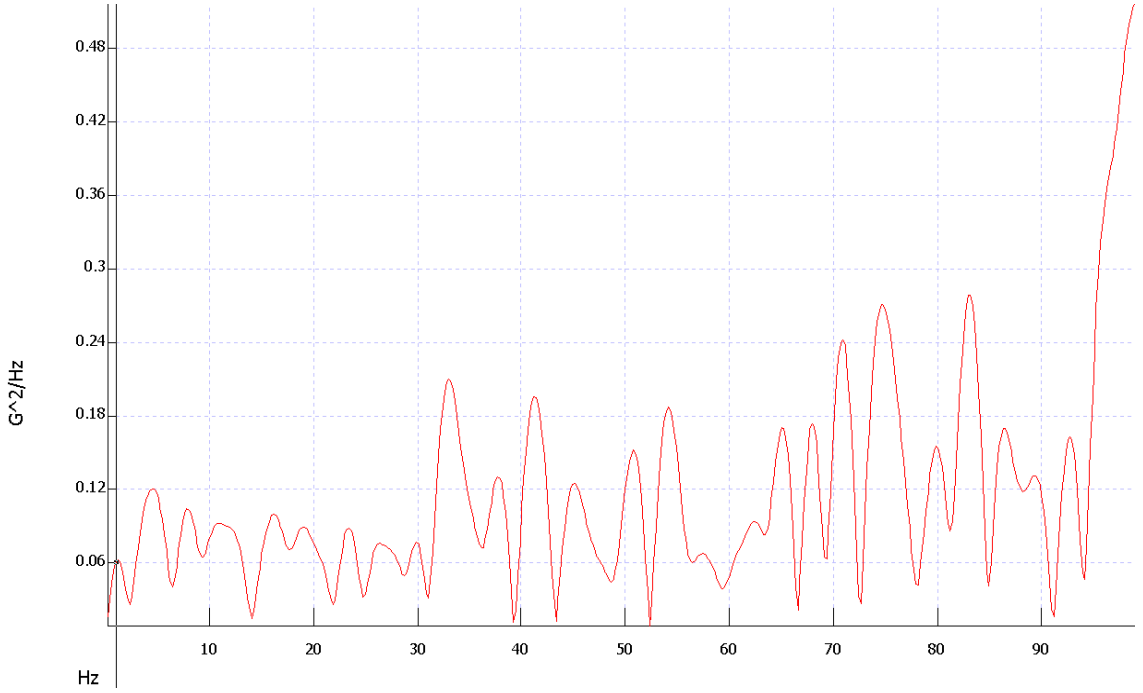

Figure 16, PSD for 'K' class hybrid

These figures are plots of the power spectral density (PSD) for the signals. This shows the power in the signal for a given frequency bandwidth. The large rise at 100Hz for the hybrid data is possibly partly an artifact from the PSD computation.

The composite motor clearly has almost all of its signal power at DC while the hybrid spreads its power over the entire 100Hz bandwidth.

The obvious solution for this is to include a good analog low pass filter between the accelerometer and the ADC. However this would also result in removing so much of the random sensor noise that dithering would no longer occur. This would result in a +/- ½ bit uncertainty in the 1G offset.

There is a way to both prevent aliasing and keep the dithering that is quite simple. The first step is to low pass filter the accelerometer data to a frequency that leaves enough random noise in the signal for dithering to happen. This is typically much greater than the filter frequency required to eliminate aliasing and depends on the resolution of the ADC. The altimeter then samples the signal at a much higher rate than normal. If the pre-sample filter had a bandwidth of 250Hz then a sample rate of 1000SPS would be suitable. After sampling, the altimeter would then pass the data through a simple digital filter to reduce the bandwidth below the Nyquist frequency of the main sampling loop.

While there are plenty of fancy digital filters available, most require a lot of computation. But there is a simple filter that is equivalent to a basic analog RC filter. Its performance is not the best but it is extremely simple.

$$x_{n+1} = (1-a)x_n + a \cdot s$$

Where

s is the new sample
x is the filtered value
a is the filter gain (0<a<1)

The multiplications can be reduced to simple bit shifts by choosing the filter gain to be a power of 2.

The relationship between the 3db frequency and the filter gain is:

$$(1-a) = e^{-2\pi f_c}$$

Where f_c is the 3dB frequency expressed as a fraction of the sample rate.

Appendix D
Static Port Time Lag

Altimeters are usually installed into an electronics bay that is vented to the atmosphere through one or more holes. Because there is a closed cavity with vents to the outside, there is a delay between changes in the static pressure at the vent holes and the cavity pressure. This delay can result in late deployment if proper care is not exercised.

A formula to calculate this delay is:[6]

$$T = \frac{L\left(\frac{D}{d}\right)^2}{4000}$$

Where:

L is the length of the altimeter bay in inches
D is the diameter of the altimeter bay in inches
d is the diameter of the vent hole in inches
T is the delay in seconds

This formula assumes a cylindrical altimeter bay and a single vent hole.

My flight test vehicle has an altimeter bay that is approximately 12" long and 2.5" diameter with two 0.25" vent holes. (Equivalent to one vent hole with a diameter of 0.35".) The time delay for this bay is 0.15 seconds. Note that this delay is about the same as the apogee time difference between the pressure only and pressure plus acceleration Kalman filters.

[6] Bob Krech, message of 20 July 2004 on Rocketry Online

Appendix E
Rule 63.5 Information

Objective: Development Kalman filter for pressure and acceleration data. Demonstrate its effectiveness and suitability for use in altimeters.

Approach: Write software to process pre-recorded altimeter data (primarily RDAS). Then build an altimeter and write software for it followed by flight tests. Lacking any other objective measure of the time of apogee, barometric pressure altitude data is used.

Previous R&D Reports: "Barometric Apogee Detection Using the Kalman Filter" presented at NARAM 44.

Equipment used:

Personal computer
Warp13 PIC programmer
Assorted tools, soldering iron, etc.

Software tools used:

MPLAB - Microchip development environment for PIC microcontrollers
djgpp - GNU based C compiler for Microsoft Windows
HyperTerminal - for downloading flight data
RDAS user interface
GNUplot - plotting data

Facilities used: Dallas Area Rocket Society launch sites and equipment.

Cost of project:

Parts for Altimeter	$124
Rocket motors and other launch supplies	~$40 (all motors were from existing stock and none were purchased for this project)
Altimeter bay for Initiator	built from parts on hand. Value unknown.

See report body for data collected, results, and conclusions.

Further work: I believe that I have completed what I set out to do. Which was to demonstrate a better method of altimeter apogee detection. While an extended Kalman filter using a better dynamic model could be developed, I do not think that the marginal improvement in performance would justify the effort. I will use my results as the basis for an altimeter of my own design.