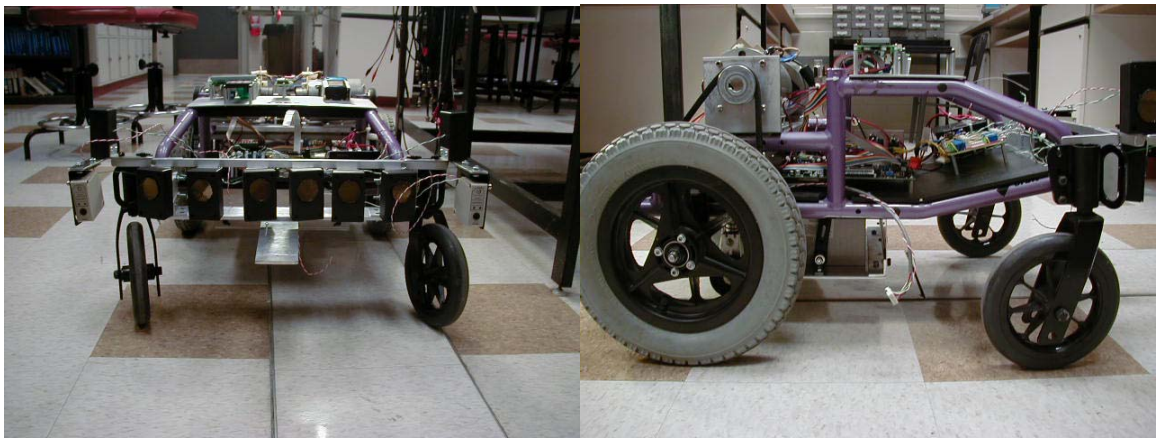
Designing a Dead-Reckoning Positioning System for the Whitney Autonomous Vehicle Project

By Alan Ghelberg, BR '02
Electrical Engineering & Economics
Spring 2002



Abstract:

Currently GPS systems available are subject to inaccuracy and use signals that are easily obstructed by buildings, tree canopies, etc. However, a dead reckoning positioning system is independent of external signals, and therefore may complement the GPS to create a more robust hybrid positioning system. Seeking to create such a system for the Whitney Autonomous Vehicle Project, I first experimented with an accelerometer / compass arrangement, but found it too inaccurate for the purposes of the project. Then, by implementing an odometry / compass – based system, I was successful in obtaining a relatively accurate position reading.

Introduction

Today, many vehicle navigation systems employ *Global Positioning System* (GPS) sensing to ascertain position. By picking up radio signals form various satellites orbiting earth, a GPS receiver is able to calculate its position. However accuracy is relatively limited, as most commercially available systems can generally only find position within 20-30 feet. Moreover, if the "line of sight" from the receiver to the satellites is blocked, the signal degrades, and may become unusable. In real-world applications this is a common occurrence since a vehicle may move under a tree canopy, behind a building, or any number of things that will block vision to the sky. The Whitney Autonomous Vehicle, if working off of GPS sensing alone would run into this problem often, navigating the dense urban terrain of New Haven. Also, GPS is useless indoors as there will be no signal. Therefore, I sought out to design another mode of ascertaining position that would not suffer from these problems.

A *dead reckoning* positioning system is defined as "a method of surveying that measures distance and direction from one point to the next along a travel path." Essentially by repeatedly recording the distance traveled and the direction, one may iteratively calculate approximate position. A method such as this relies on local sensing, and therefore is not subject to external obstruction. It may also be designed cheaply and compactly. For these reason I decided that dead reckoning would be a good method to supplement GPS.

To implement a dead reckoning system, I first arrived at the idea of using acceleration and compass data as inputs to a micro-controller, where I would perform the positioning calculations. After some exploration, it soon became apparent that due to the double integration, error would build up too rapidly, resulting in an unusable system. From there, I looked for a method that would not suffer this same problem, and came to the idea of replacing the accelerometer with an odometer, thereby acquiring distance directly. This proved to work much better, and in fact was very successful. My test runs resulted in errors of roughly 5% over a 40-meter travel distance.

This *methods* section of this report will look at how I implemented both of these dead reckoning systems, in terms of both hardware and software. The *results* section will present how I tested the system, and an analysis of the results. The *discussion*, will investigate possible uses for the system and as well as such issues as manufacturability.

Methods

First try: Compass/Accelerometer-based system

Over the past few years, accelerometers have increased in quality, while decreasing in price. Accelerometers are also very small and entirely self-contained. For these reasons, I felt that they would provide a good means of calculating distance traveled, and therefore in the first configuration, I employed a compass and an accelerometer as my two sensors.

Equipment

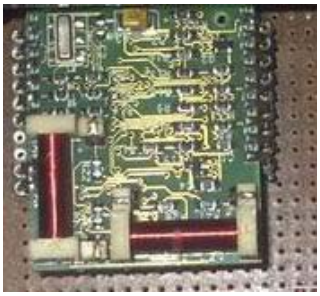
PIC 16F877 Microcontroller

The PIC 16F877 microcontroller, manufactured by Microchip, is a 20 MHz processor with built-in features such as analog-digital (A/D) conversion, a counter, etc. Its versatility, low-cost, and low power consumption make it ideal for embedded applications. In my project, the PIC provided a mean of interfacing with external devices, performing calculations, and communicating with the main computer.

Vector 2x Digital Compass

The Vector 2x compass module, manufactured by Precision Navigation Instruments, Inc. is a low cost digital compass. By making use of the earth's magnetic field information

sensed by two perpendicularly-mounted magnetometers, the Vector uses on-board processing to calculate heading. Heading is reported as an angle from 0° to 359°, and the signal is specified to be accurate within 2°. The compass is sensitive to tilt, as 1° degree of tilt can lead to about 2 degrees of heading error. Therefore the Vector 2x is really only suited for use on relatively flat ground. Furthermore, the compass is very sensitive to external magnetic fields from nearby electrical equipment, which cause large errors. The Vector 2x has a refresh rate (calculates a new heading) of roughly 5 Hz.

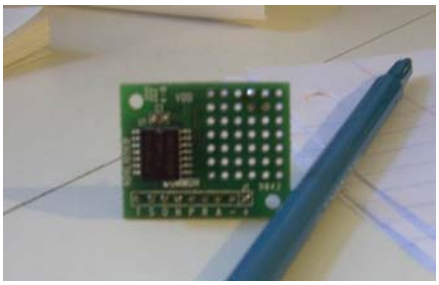


The Vector 2x Digital Compass

The Vector 2x communicates to the microcontroller via a serial peripheral interface (SPI) link. Originally, there was significant difficulty getting this working, because we were trying to use the built-in SPI functions on the PIC chip. This led to shaky performance, as the compass would sometimes put out garbage data. I therefore rewrote the code, using a method that does not employ the SPI functions. This has proven much more robust. I include this code in the appendix (comp5.h). Also, there are two points in the communication that require a delay, the duration of which the user's manual says, "depends on your system." In our case the communication worked with the delays set to 5 milliseconds; however, experimentation may be necessary to find the proper delay.

ADXL105 Accelerometer

Analog Devices' ADXL105 Accelerometer is a relatively high-accuracy MEMS (micro-electro-mechanical systems) accelerometer embedded on a tiny microchip. It determines the amount of acceleration felt along in a certain direction. For a range of –5 to +5 Gs (1 G = acceleration due to gravity = 9.8 m/s^2), the ADXL105 outputs a voltage from 0 to 5 Volts. The voltage output is roughly proportional to acceleration, with 0 V representing –5 Gs and 5V representing +5 Gs.



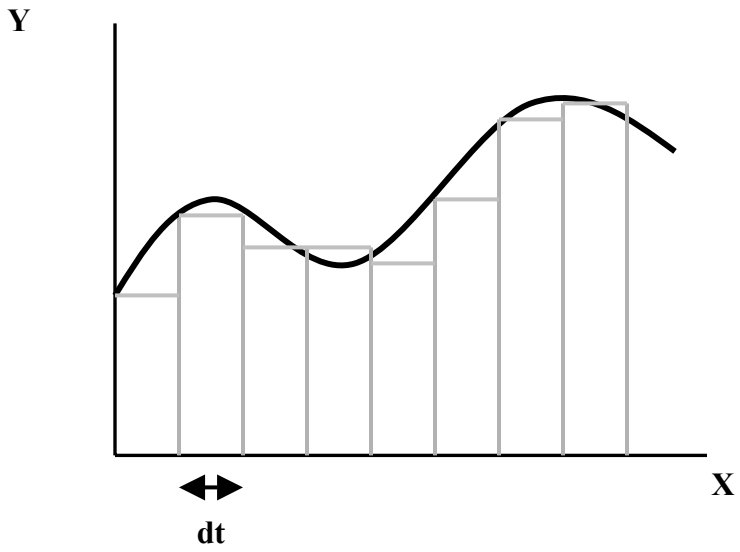
The compact ADXL105 Accelerometer

Principles of Operation

The accelerometer/compass system takes the accelerometer data into the PIC and then double-integrates the acceleration to arrive at position since:

$$a = \frac{d^2x}{dt^2} \Leftrightarrow x = \iint a\,dt$$

Where *a* is acceleration in one direction and x is displacement in that direction. In actuality, it is necessary to perform an approximate integration since the data from the accelerometer is not continuous (it can only be sampled by the A/D converter every 20 microseconds). This is done by taking time slices of the data.

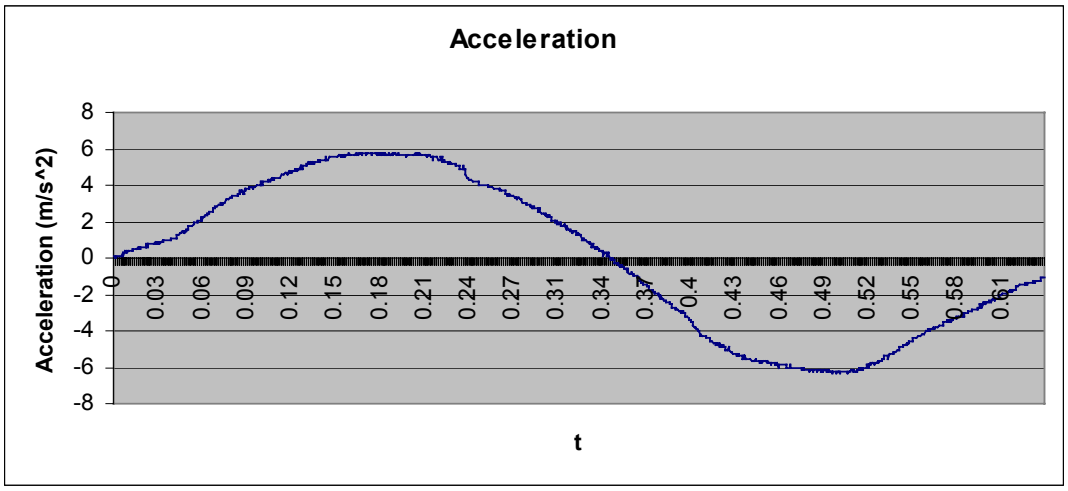


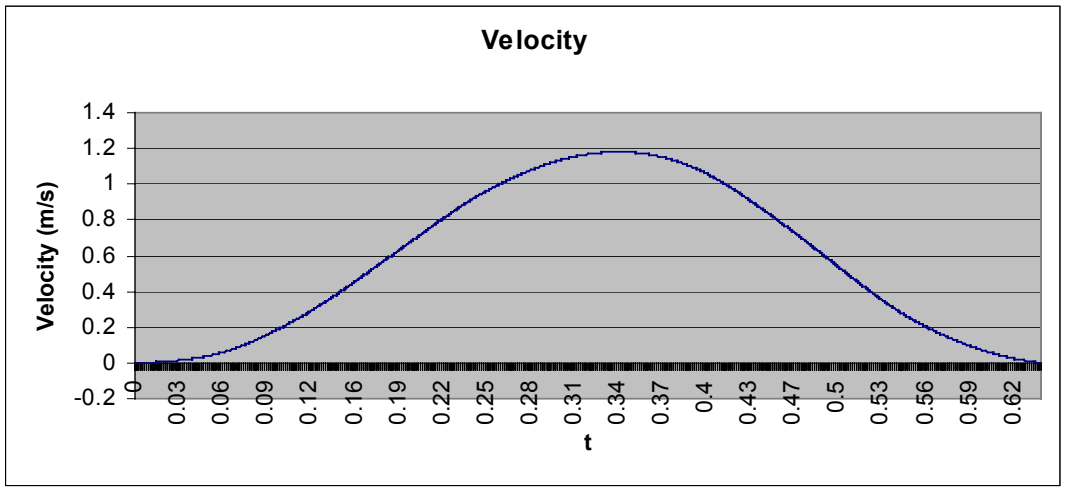
Approximate Integration

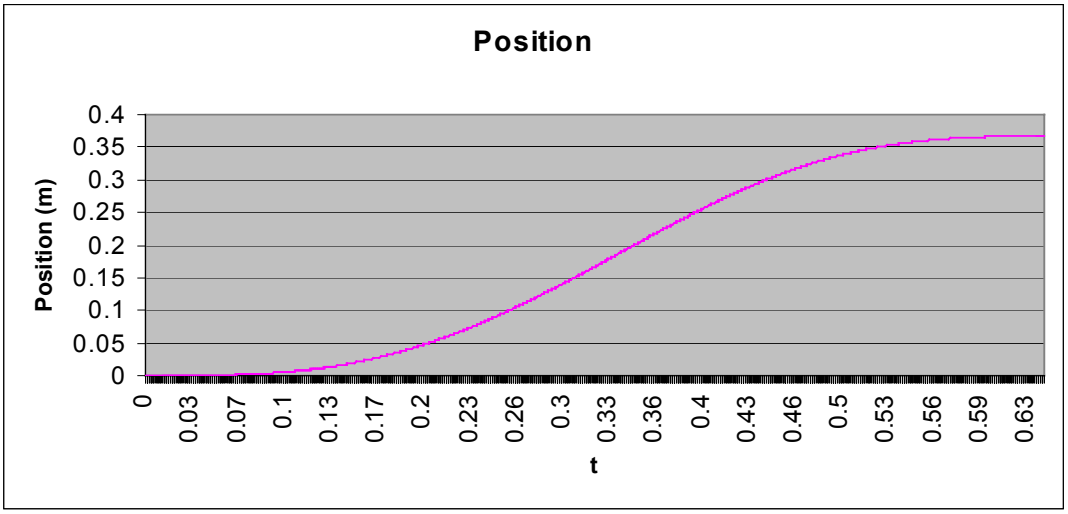
The area of the rectangles under the curve approximates the actual area under the curve; the smaller the time slices *dt*, the more accurate the approximation. In the accelerometer system, the two equations for the double integration are:

$$v_f = v_i + a\,dt$$
$$x_f = x_i + v\,dt$$

By performing these operations every time slice, we move from acceleration to velocity, and then velocity to distance. We must assume initial values for velocity and position (here assumed to be zero). The first of these graphs is of data collected from an oscilloscope when I pushed the accelerometer a certain distance. I then imported the data into Microsoft Excel, converted the acceleration voltage data into actual acceleration, and then performed these calculations to obtain velocity and finally position.







Discretely-calculated double Integration

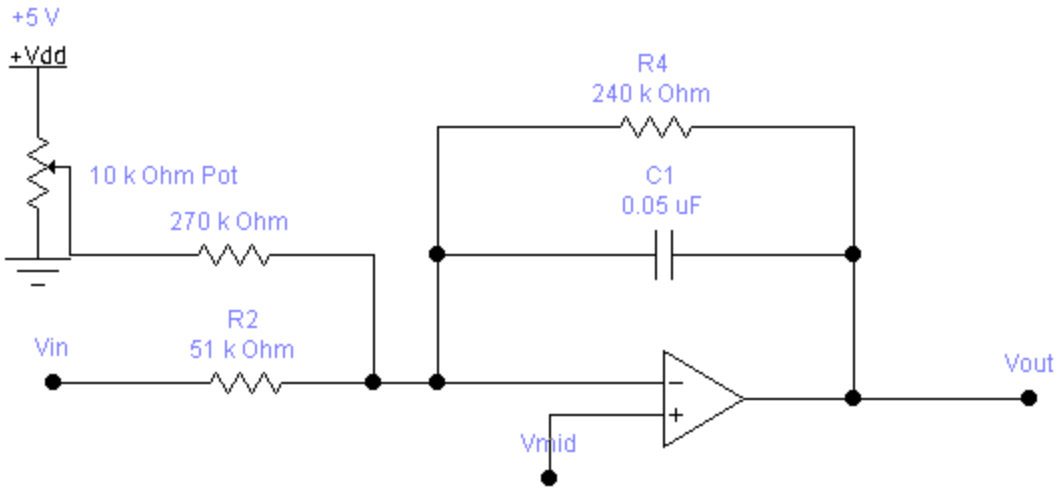
This initial test was very encouraging, since I moved the accelerometer 37 cm, and the calculations resulted with a distance of 36.7 cm. However, when actually calculating on the PIC in real-time, and over a longer duration, the accuracy suffered severely.

After calculating distance, it is possible to calculate x/y position given that you know the direction you have traveled in. However, since I did not ultimately implement two-dimensional positioning with this arrangement, I will save the explanation of calculating x/y position for later in the report.

Implementation

The first thing I did when implementing the system was to look for a way to get clean useful data out of the accelerometer. There were two problems with the accelerometer output. First of all, the ADXL105 was specified to have a resolution of 2mG (milliGs) over a range from –5 G to +5 G. This means that there should be (10 G) / (2 mG) = 5000 possible outputs. However, the PIC has a built in 10-bit A/D converter, meaning that only $2^{10} = 1024$ values may be represented. Therefore in order to make full use of the resolution, it was necessary to limit the range of accelerations that the device could measure. Given the operation specifications of the Whitney Autonomous Vehicle – it could not drive extremely fast – I assumed that it would not undergo accelerations of more than 1 G. I implemented a circuit to amplify the output so that roughly –1 G to +1 G acceleration would map to 0 V to 5 V. This theoretically allows the PIC to use the maximum 2 mG resolution.

The second problem with the accelerometer output was excessive noise. Once again, though, the Whitney could not feel rapid acceleration changes. Therefore, I could use a low pass filter to remove the high-frequency noise, while retaining the important information. I accomplished both the amplification and the filtering with this circuit:



Accelerometer Circuit

In this circuit V_{mid} is a voltage output from the accelerometer that is roughly 2.5 Volts and is therefore the midpoint of the 0 V to 5 V range. This centers the outputs around 2.5 Volts. The amplification provided by the circuit is equal to:

$$\frac{R_f}{R_i} = \frac{240k\Omega}{51k\Omega} \approx 4.7$$

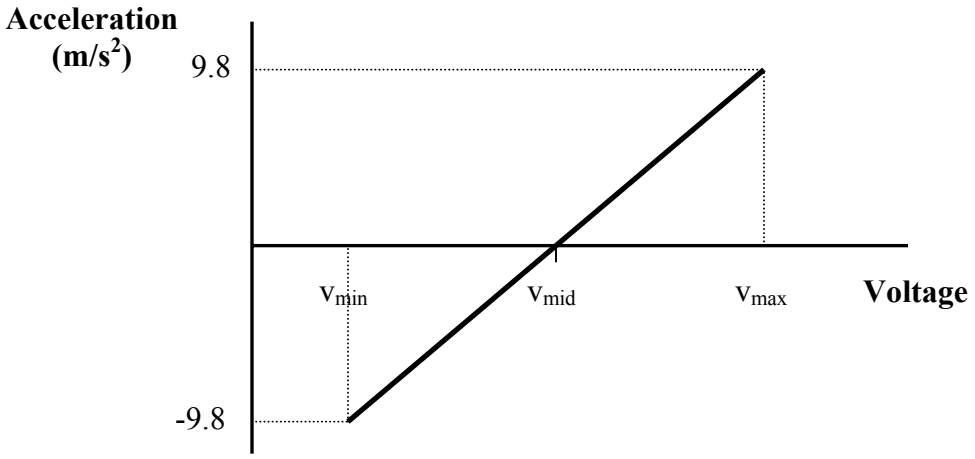
The cutoff frequency for the low-pass filter is:

$$f_{cutoff} = \frac{1}{2\pi R_f C} = \frac{1}{2\pi(240k\Omega)(.05\mu F)} \approx 13.3Hz$$

The reason for including the potentiometer is to compensate for what is referred to as zero-G bias. Even if the accelerometer feels no acceleration, it will not output exactly 2.5 V. There will be a slight bias that varies from one device to the next and also depends on

temperature. Therefore, by adjusting his potentiometer, the user can calibrate the unit. The zero-G bias is also somewhat compensated for in code. When the unit initializes, it samples assumes it is under no acceleration and samples 64 times and takes the average. This average value is then the zero-G level.

Once the acceleration voltage data is in the PIC, it must be transformed into actual acceleration data before calculating position. We calibrate by using gravity as a reference; if we point the accelerometer down it feels –1 G acceleration and if we point it upwards it feels +1 G. Assuming that the accelerometer output is linear with acceleration, and knowing that 1 G = 9.8 m/s^2, we can convert the voltage to acceleration



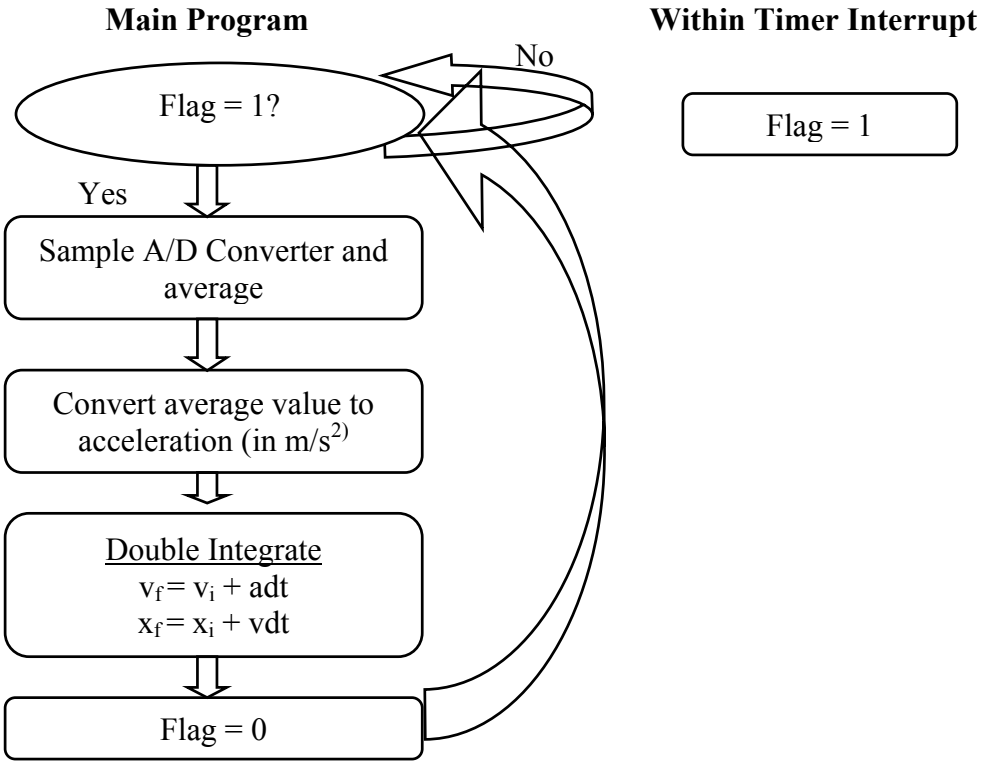
Voltage – Acceleration Relationship

$$\textbf{Acceleration} = \left(\frac{9.8-(-9.8)}{V_{max}-V_{min}}\right)\times \textbf{Voltage}$$

Now after the PIC calculates acceleration, it must then do the numerical integration with time slices as stated previously. This is implemented through the use of a timer interrupt. This interrupt occurs at a regular interval (in this case 819.2us), and

within this interval all the calculations must be completed. The more frequently the interrupt occurs, the better the accuracy will be. To implement this all on the PIC, I initially tried to actually do all calculations within the interrupt, however this caused the PIC to output nonsense data. The problem is that the PIC has poor interrupt handling capabilities, and cannot call functions from the interrupt. Therefore, to get the program to work, I simply set a flag in the interrupt that enables a function in the main program. In this way, the function is only called once every time slice.

Within this time slice, the PIC samples the A/D converter – in fact it improves accuracy by over-sampling (samples 8 times and takes the average). Then the PIC converts the voltages to acceleration, and performs the double integration.



Accel_7.c Flowchart

In reality, this algorithm is complicated by the fact that it is difficult to represent the values with the necessary accuracy. To use the PIC for floating point numbers would be too slow, so it is necessary to represent the numbers in a different manner. If, when multiplying by *dt*, the program multiplies by the time slice (.0008192 s), the system would be useless, because the calculation $v_f = v_i + a*dt$ would always equal zero. Therefore I change the time scale to ms, so *dt* is now (.8192ms). This, however leads to rapid data overflow, since it has the effect of multiplying velocity by 1,000 and acceleration by 1,000,000. Therefore counters are necessary to make the program work. Whenever position traveled goes above 16,000 the program increments an overflow counter.

After coding the program, I experimented with the accelerometer system to test its effectiveness in determining distance traveled. The system behaved extremely poorly. There were some errors in the acceleration signal due to a number of factors such as:

- Limited resolution
    - The accelerometer has a specified *resolution* of 2 mG, however within the system it seemed more like 10 mG, perhaps due to noise in the PIC A/D converter.

- Nonlinearity
    - Although the accelerometer ideally provides a voltage output directly proportional to acceleration, it is not perfect. Therefore there are non-linearities in the output that are difficult to account for.

- Zero-G Bias
    - As stated previously, the accelerometer output is not centered perfectly at 2.5 V, and this bias shifts with temperature. It is difficult to completely subtract this out of the system.

- Gravity
    - When perfectly flat, the accelerometer will only sense acceleration due to movement. However, when tilted, it begins to sense acceleration due to gravity, which could be very high. Therefore, if the system is stationary, but tilted, it will conclude that it is moving very fast.

The errors from these factors accumulate extremely rapidly due to the double-integration. If there was even a small error in the accelerometer signal, this would then lead into a larger error in velocity, which would in turn lead to a much larger distance error. This rapid increase in error over time is called "drift," as the calculated distance drifts far away from the actual distance traveled. Due to the double-integration, this drift accumulates exponentially with time.

Second try: Compass/Odometry-based system

Given the failings of the accelerometer system, I moved away from any integration and arrived at odometry as a means of calculating distance traveled. By putting an extra wheel on the vehicle and measuring rotations, it is possible to directly calculate distance traveled.

Equipment

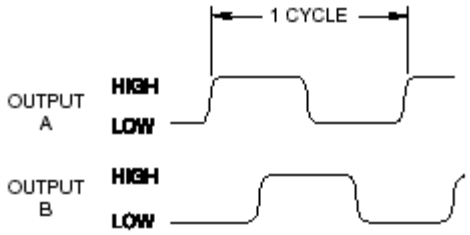
The odometry-based system uses two of the same devices as the previous method. This system still uses the Vector 2x compass for directional information, and uses the PIC16F877 microcontroller for some processing. The new pieces of equipment are:

Grayhill 63R128 Optical Encoder

The odometry system uses the optical encoder to count the number turns a wheel makes – or more precisely the number of fractions of turns a wheel makes. The encoder used in this project was the Grayhill 63R128, which has a resolution of 128 counts/revolution.

The Grayhill 63R128 Optical Encoder

Essentially, the encoder works by having a series of (in this case 256) internal light and dark bands. As the shaft rotates, an optical sensor pulses whenever the band underneath it goes from light to dark, or vice-versa. There are actually two outputs for the encoder, A & B, where A leads B by 90° for clockwise rotation of the shaft.

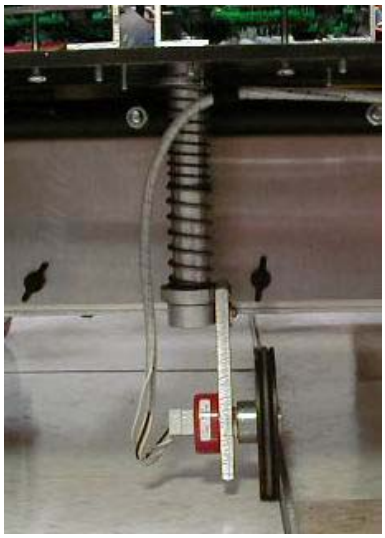


Optical Encoder Outputs

By using the information provided by both outputs, one may ascertain in which direction the shaft is turning. For instance, if A goes high and B is still low, then the shaft is turning clockwise.

Additional Tracking Wheel and Mount

The tracking wheel is an additional wheel mounted on the vehicle to gather the odometry data. The additional wheel was mounted because, since it is not a driving wheel, it is only reacting to the vehicle's motion and therefore is less likely to slip.



The Tracking Wheel Assemblage

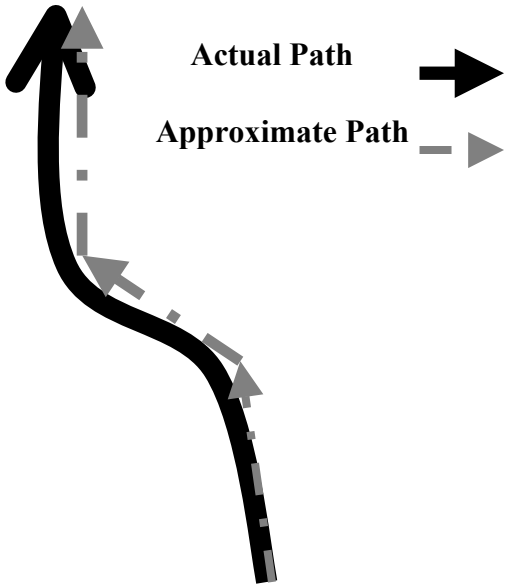
The wheel is 7.7 cm in diameter, and is mounted on springs to keep constant contact with the ground. It is mounted directly between the two rear wheels, so that it measures the actual distance traveled, even while the vehicle is turning.

EBX Main Computer

The EBX is the center of Whitney's intelligence. Essentially a small form-factor computer, it is powered by a 300 MHz National Geode chip. It can communicate with peripherals, such as the PIC microcontrollers, through an RS232 serial data link. For the positioning calculations, the EBX performs the more difficult computation so as to not tax the PIC's very limited resources.

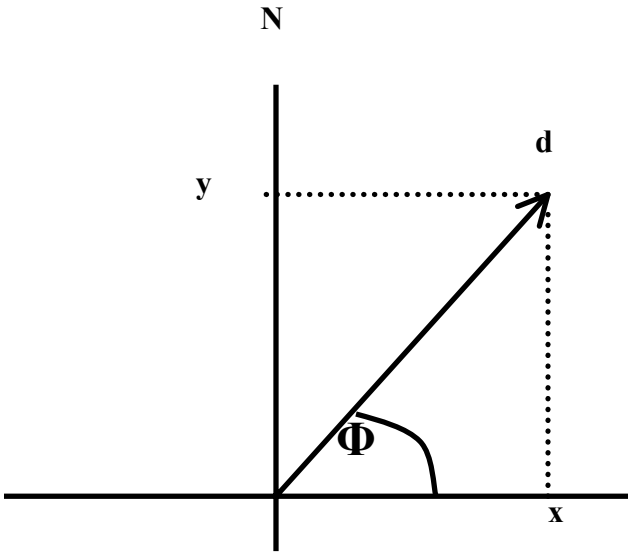
Principles of Operation

The general ideas governing the odometry positioning system are similar to those governing the accelerometer system. The Vector 2x compass outputs data to the PIC roughly every 200 ms. In between compass data refreshes, the PIC counts the number of wheel rotations, or fraction of wheel rotations from the encoder. Therefore the system has information on distance and direction. By assuming that all the distance traveled over the 200 ms period was in the direction specified by the compass, it makes a linear approximation of the vehicle's actual movement.



Vehicle Movement Approximation

Because the Whitney Vehicle moves relatively slowly, the linear approximation proves to be a relatively good proxy for the actual motion. Given the distance and heading information it is now possible to calculate 2-dimensional position.



x/y Positioning Graph

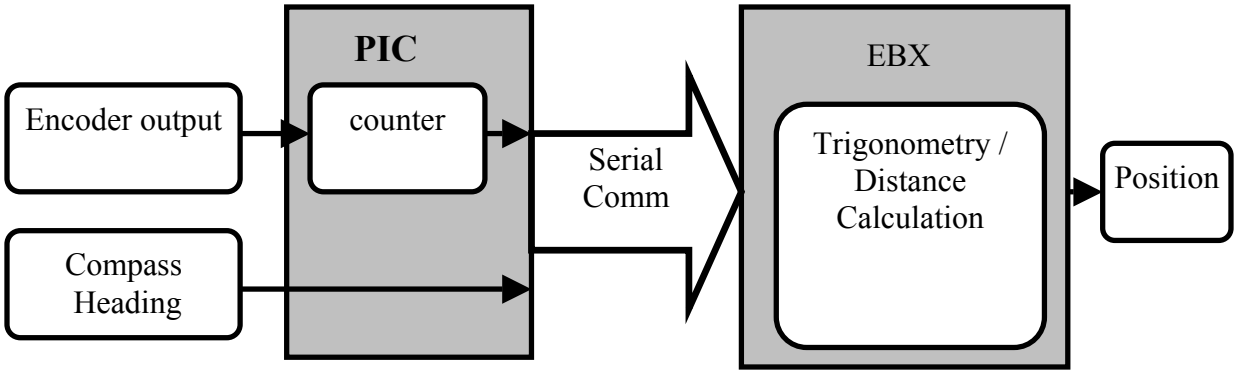
Since the compass gives directional heading as angle clockwise from y-axis with the positive y-axis defined as due north, the angle Φ in this graph is actually (90° - compass heading). To get the x and y positions:

$$\mathbf{x_f = x_i + d\cos\Phi}$$
$$\mathbf{y_f = y_i + d\sin\Phi}$$

where **d** is the distance traveled over the time period. If x and y are initially defined as zero, this will iteratively give the vehicle's position.

Implementation

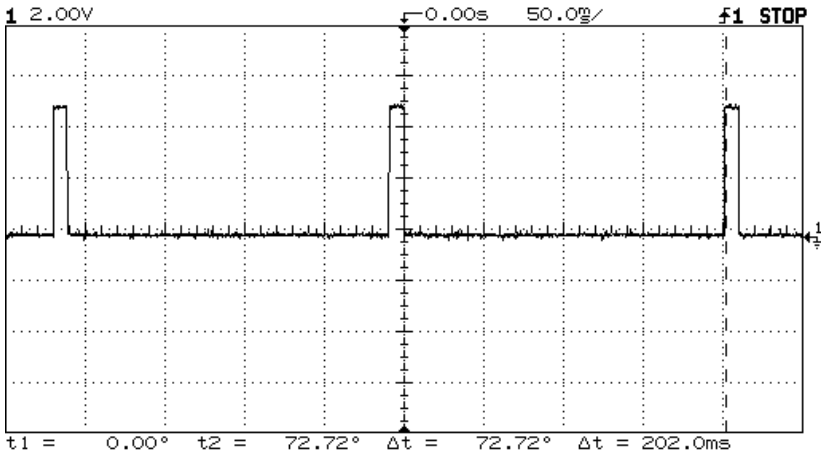
The principle guiding the implementation of this algorithm was to move the more difficult calculations to the EBX, where they would take less time and be more accurate (i.e. double-precision floating point numbers). The PIC is used solely for data acquisition.



PIC / EBX Division of Labor

PIC Portion of System

Once asking the Vector 2x compass for heading information, it takes the compass 202 ms to return the information. Therefore, the compass refresh rate is roughly 5 Hz.



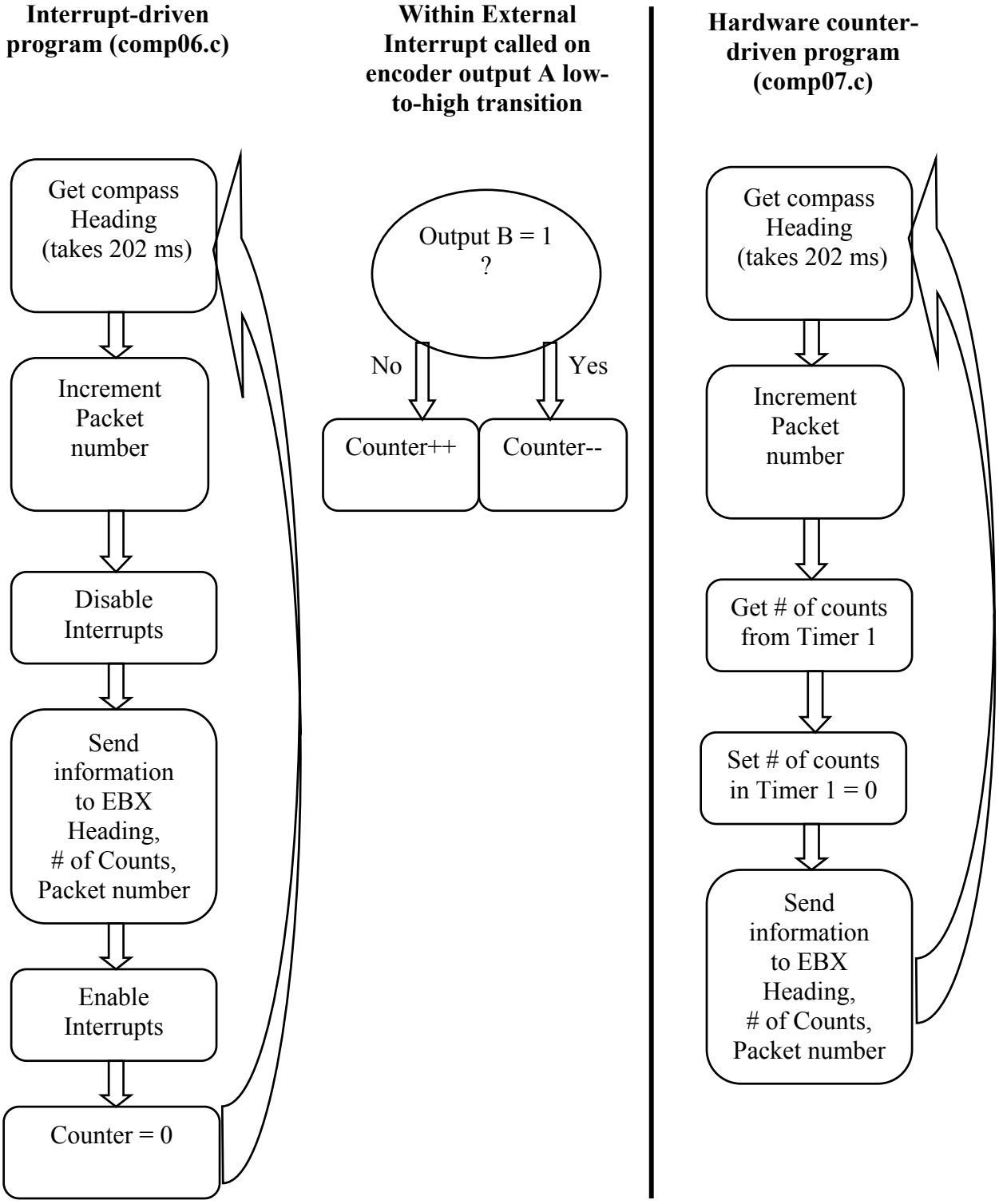
Compass Timing on Oscilloscope

Between these compass refreshes, the PIC counts the number of "ticks" (encoder counts) that the optical encoder outputs. Once it receives the new compass data, the PIC sends the heading and number of ticks to the EBX via the serial connection.

Initially I counted the ticks by connecting the encoder's output A to the PIC's external interrupt pin. This way, the program interrupts whenever the A output transitions from low to high, 128 times per rotation. Within the interrupt, the program would check the B output and decide whether to increment or decrement a counter based on the direction of rotation. However, this caused a problem because the RS232 link is very time-sensitive. Whenever the PIC was interrupted during communication with the EBX, the link would fail and incorrect information was sent. I solved this by disabling

interrupts during the RS232 communication; however, this causes some counts to be missed.

I also re-implemented the program, using the PIC's on board hardware counter to count the number of ticks during each time period, thereby avoiding the use of any interrupts at all. This implementation is somewhat deficient however, in that it is unable to discern forward from reverse motion. Both programs also increment another variable called packnum, each cycle. This numbers each data packet sent to the EBX, so that the EBX does not double-count any data. Also, to ease in communication, the data packets are all set to a fixed length of 17 bytes.

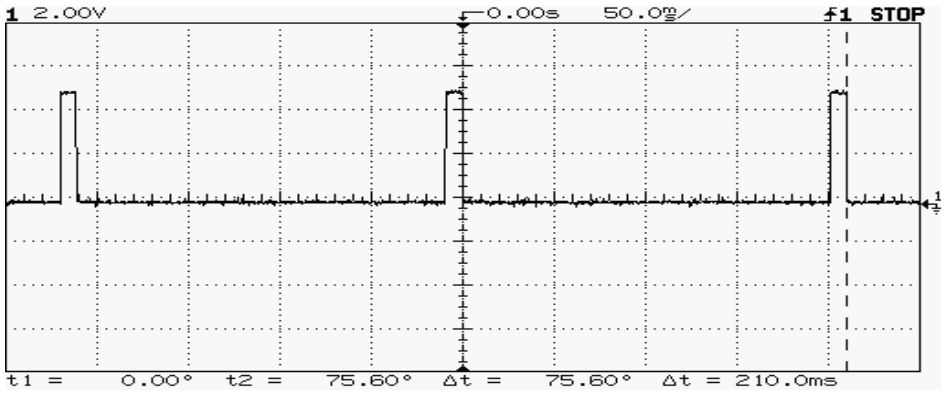


Flowcharts for both implementations of Compass/Odometer sampling program

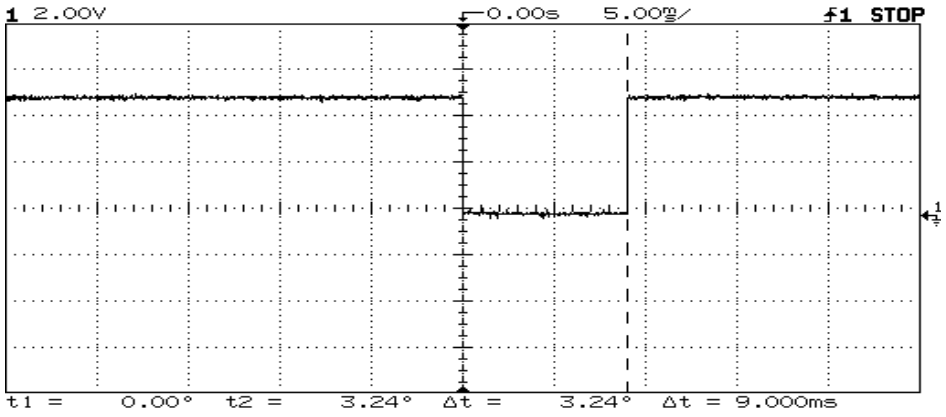
Each of these implementations has both an advantage and a disadvantage. The counter-based program does not work if the vehicle is moving in reverse. The interrupt-driven program will miss some counts. In theory, since the PIC is sending 17 bytes at 19.2 kbits/sec, the interrupt disabling should occur for:

$$17\text{bytes} \times \frac{8\text{bits}}{1\text{byte}} \times \frac{1\text{second}}{19200\text{bits}} = 7.1\text{ms}$$

plus some time to disable and then re-enable the interrupts. The first of these graphs shows the overall loop duration of the program, while the second shows the actual length of time for which interrupts are disabled.



Total Loop Duration Of Program



Duration of Interrupt Disabling during RS232 communication

The total loop takes 210ms to complete, of which the interrupts are actually disabled for 9ms. Therefore the system is missing counts for (9 / 210) = 4.3% of the time. This can be somewhat corrected for by just taking this into account and multiplying the number of counts by a scaling factor 1/(1-.043) = 1.045. This essentially assumes that the vehicle is moving at a constant speed over each 210ms period, which is actually quite a realistic estimation. Since this is an accurate adjustment, I feel it is worthwhile to use the interrupt-driven program so that it will still function with reverse motion.

EBX Portion of System

The job of the EBX is to calculate position based on the data it receives from the PIC. The first part of this is actually receiving the information from the PIC. This is not a trivial task, since the two devices are asynchronous, and it is crucial not to miss any data packets.

As data comes in to the EBX from the PIC it is stored in a buffer. If the EBX program is to pull data from the buffer before a packet has finished sending, then it loses the packet. Therefore the method used to collect the data is to poll the buffer every 50ms, and then check if it contains 17 bytes of data (each packet is a fixed length of 17). The program only checks the buffer every 50ms so as to not eat up too many system resources. If there are not 17 bytes in the buffer, then the program loops through. If there are 17 bytes, then the program pulls that data from the buffer. The information from the packet (direction, distance, packet number) is then extracted and the program

makes sure that the packet number is different from the previous packet, to ensure that there is no double-counting of data.

If the data packet is new, then the program updates its positioning data with the new information. It also keeps track of total counts and overall distance traveled. To obtain distance:

$$\mathbf{Distance} = \frac{\pi \times \mathbf{diameter}}{\#\mathbf{counts/rotation}} \times (\#\mathbf{counts})$$

Given the previous calculation for the interrupt-driven program, there should be 128 * (1 - .043) = 122.5 counts/rotation. I tested this by spinning the wheel 10 times and then dividing the counts by 10. I did this 20 times and the average was 122 counts/rotation – almost exactly the calculated value. For this value the distance traveled per count equals:

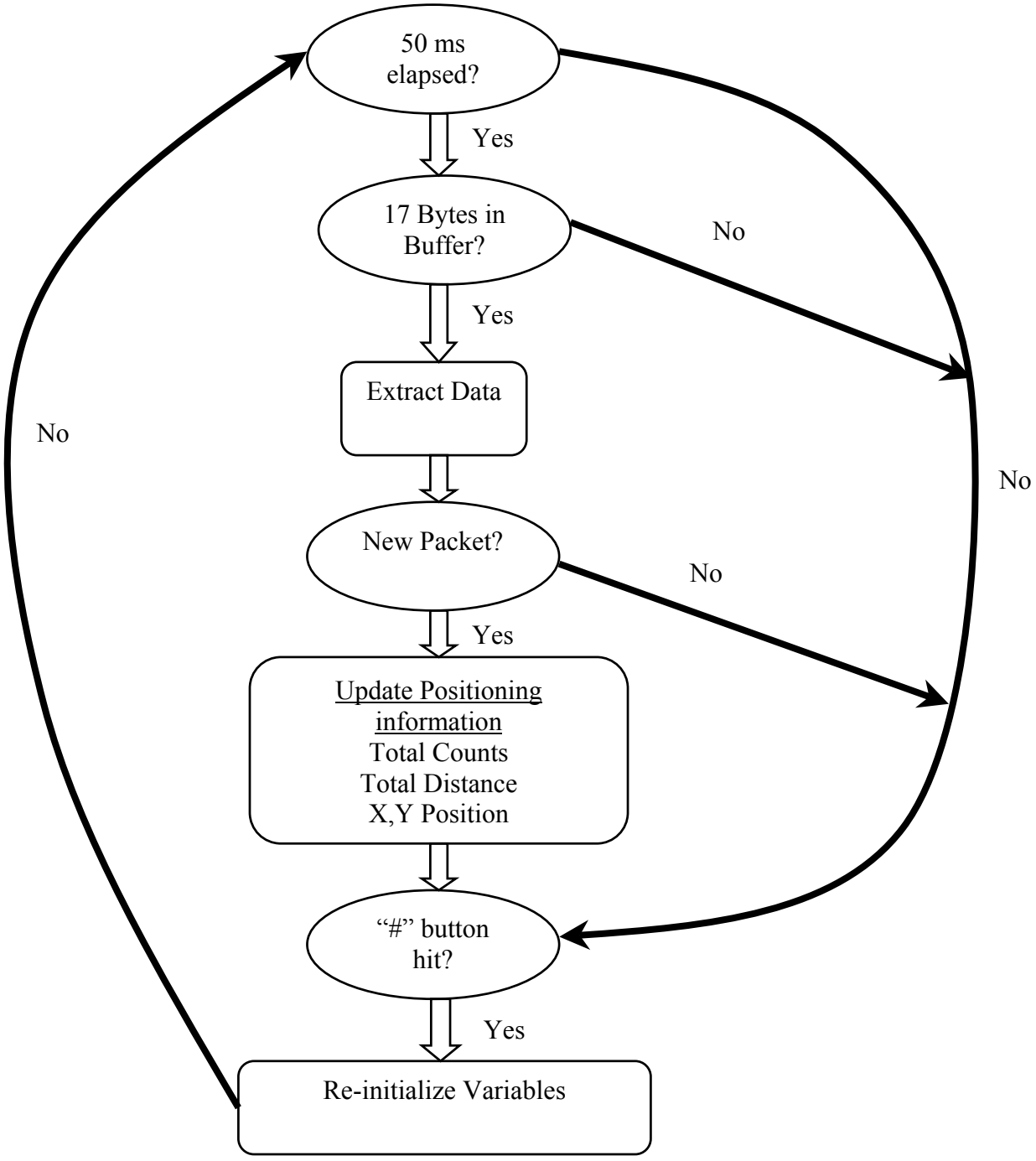
$$\mathbf{Distance/count} = \frac{\pi \times .077\mathbf{m}}{122\mathbf{counts/rotation}} = .001983\mathbf{m/count}$$

Performing the same experiment on the non-interrupt system, I found an average counts/rotation of 126.55. This is not the full 128 counts because one or two counts can be missed during the time lag between reading the counter and resetting it to zero. The distance per count for this system is therefore .001911m/count.

From the actual distance data and the compass data, the program calculates x/y position using the formulas stated previously. The angles must be converted to radians, for the C program to perform the trigonometry operations. For each new packet, all navigation data is updated and then printed to the vehicle's LCD screen. The program

also provides the ability for user reset by pressing the "#" button on the vehicle keypad, as it checks for the key hit every cycle.

EBX Positioning Program (Pos.c)



EBX Positioning Program: Pos.c

Results

After the final implementation of the compass and odometry-based positioning system I began testing to ascertain system performance. The tests provided positive results, as the position estimation remained relatively accurate even over long distances.

The first trials were to test both inputs to the system, the accelerometer and odometer, in isolation and therefore get an idea of the reliability of each on its own. I tested the compass for accuracy and also its susceptibility to magnetic fields. As far as accuracy was concerned, I looked for self-consistency. By taking a reading, turning the unit 180°, and looking at the new output, I tested whether the Vector 2x was accurate in keeping a constant reference frame. Ideally, the two readings should be precisely 180° separated from each other. Four tests resulted in errors off this ideal ranging from 0° to 12°. The compass therefore has some inherent inaccuracies.

The second compass trial tested the effects of external magnetic fields on the compass reading. Moving Whitney down the Becton hallway in a constant direction, I noticed that the compass readings varied up to 50° for the same actual heading. This error was by far the greatest when passing an electrical equipment cabinet, as the large magnetic field totally invalidated the reading.

The odometer tests consisted of pushing the vehicle exactly 10 meters and then checking the reading that the EBX outputs. These tests showed that the odometer is extremely accurate.

Trial	Odometer Output	Error (m)	Error (%)
1	10.02	0.02	0.2%
2	10.04	0.04	0.4%
3	10.03	0.03	0.3%
4	9.99	0.01	0.1%
5	10.02	0.02	0.2%

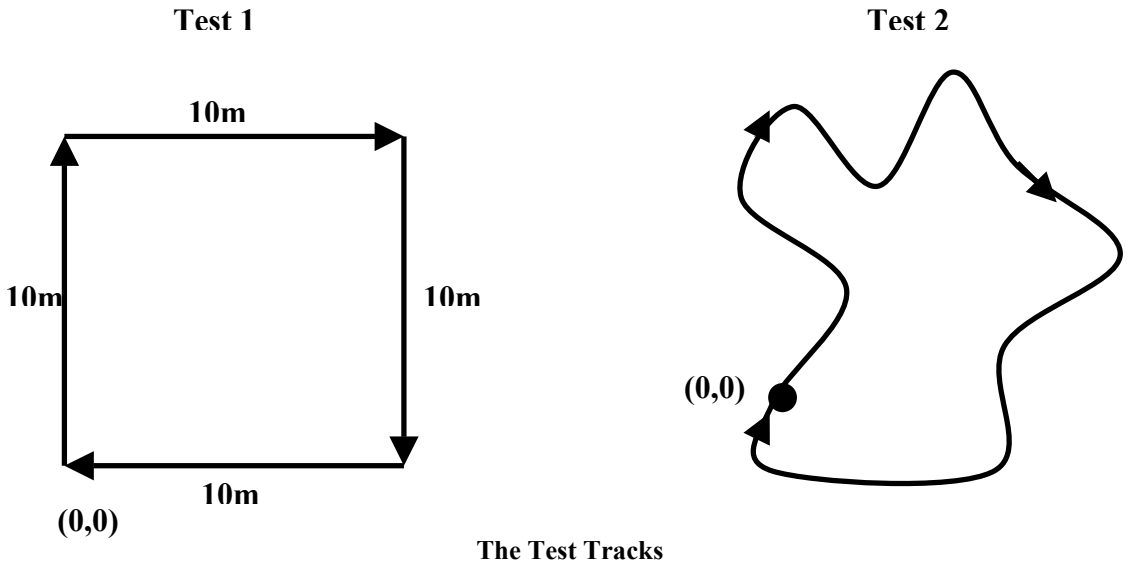
Odometer Performance

The odometer error measures in fractions of percents. Slippage proved to be an insignificant issue.

Due to the large indoor magnetic fields, I decided to conduct the overall system performance tests outdoors on Becton plaza. This however did not entirely remove the issue of magnetic fields, as compass readings were still off by up to 15º at times. Becton plaza runs directly over Becton Lab, so the field due to electrical equipment may still have been causing the issues.

I used two methods to test the positioning system. Both tests involved rolling Whitney from an initial starting point, and arriving at the starting point at the end of the path. Since the vehicle starts at (0,0), it should ideally end at (0,0). Any deviation from this is due to the error of the system.

The difference between the two tests was that one involved long straight paths with sharp turns while the other involved freeform vehicle movement. Theoretically, the system should be more accurate for the former, since it will be performing a linear approximation of a straight path.



The Test Tracks

The second test consisted of moving the vehicle in curves rather than straight lines. In both these tests, error is measured as the absolute distance from the final calculated position to the origin.

$$\text{Error} = \sqrt{(x_f)^2 + (y_f)^2}$$

Percentage error is the ratio of error to total distance traveled, which is determined by the odometer.

Results: Test 1

Trial	Distance Traveled (m)	Final Position (x,y)	Error (m)	Error (%)
1	39.86	(-2.3, 0.7)	2.40	6.03%
2	39.85	(-2.6, 0.5)	2.65	6.64%
3	40.06	(-2.4, -0.7)	2.50	6.24%
4	40.05	(-2.3, -0.8)	2.44	6.08%
5	39.83	(1.5, -2.1)	2.31	5.80%

Results: Test 2

Trial	Distance Traveled (m)	Final Position (x,y)	Error (m)	Error (%)
1	32.62	(-0.6,0.8)	1.00	3.07%
2	40.27	(-1.5, -0.8)	1.70	4.22%
3	20.37	(0.3, -0.9)	0.95	4.66%
4	43.74	(1.9, -3.7)	4.16	9.51%
5	40.38	(1.5, -2.1)	2.58	6.39%

It is interesting to note that the errors for the random path are not higher than those of the straight path. The average percent error for test 1 is 6.16 %, while it is only 5.57 % for test 2. This goes against the intuition that a more curved path will result in greater inaccuracy. However, the results of the second test have a much higher standard deviation – 2.51 % as opposed to .31 %.

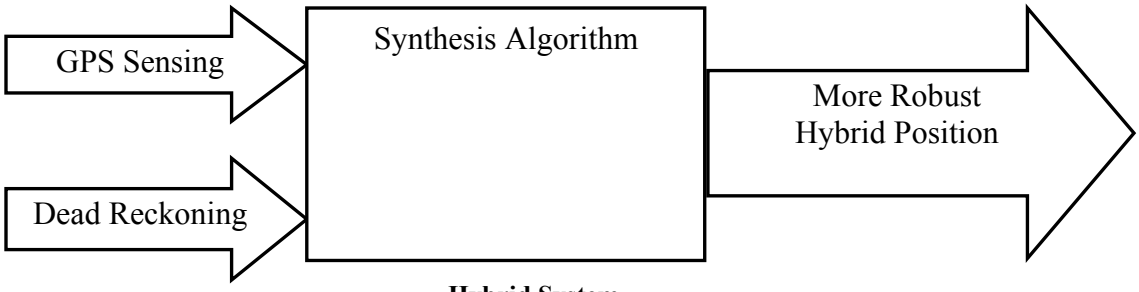
These results imply that the vast majority of the error was due to external magnetic field interference. When moving over a fixed course, the compass would feel the same distortions each time, resulting in a similar error. Note that the test 1 results are all incorrect in almost exactly the same way. On the other hand, moving the vehicle randomly causes very different interference patterns, and therefore a higher variance in results. The errors in this system are roughly what we would expect to see with the magnetic field disturbances on Becton Plaza. Under these conditions, the dead reckoning positioning system performed quite well, but it could certainly perform much better in an area more isolated from external fields.

Conclusion

Initially, while beginning the accelerometer-based system, I held high hopes for its possible uses in navigation. However, it soon became apparent that a system using double-integration would lead to huge errors. By simplifying the system and obtaining distance traveled through the most direct means possible, I was able to achieve greater success than with the more complex system. Through odometry, I was able to obtain extremely accurate distance information, and the limiting factor in the system became the compass as opposed to the accelerometer.

The odometry/compass-based dead reckoning system proved to be a very able positioning system that could successfully be used to supplement GPS under certain conditions. Even with external magnetic field interference, the system behaves well, accruing only about 2.5 meters of error over a 40 meter run. While the error tends to increase linearly over distance traveled, this system can still outperform GPS over a relatively long duration. Given that a GPS only has an accuracy of 6 to 9 meters, the dead reckoning system can travel over 120 meters before it becomes less accurate than the GPS.

The optimal navigation configuration is a hybrid between GPS and dead reckoning. Such a hybrid system collects data from the two modes of positioning as inputs, and uses some intelligent algorithm to decide how heavily to weigh each piece of information. The GPS can be used to periodically correct the local sensing system if it has accumulated too much error.



Hybrid System

An example of an intelligent synthesis algorithm would be to use the information of how many satellites the GPS sees to determine how heavily to weigh the two systems. If there is clear sight of many satellites, the hybrid could look exclusively at GPS; if many satellites are obstructed, the intelligence would turn over to dead reckoning. This kind of redundancy achieves a much more robust system, since each positioning method complements the other's weaknesses.

Either the hybrid system or dead reckoning alone lends itself to many possible applications. The system could benefit any wheeled vehicle that needs to know its position in order to guide itself autonomously. For instance, as the project's origins indicate, it could be used as an aid to the handicapped in wheelchairs. Robots equipped with such technology could navigate through tunnels or any difficult to reach area for search and rescue operations. Robots could use this positioning to perform farming tasks or lawn-mowing in an autonomous manner. Eventually self-guiding cars will likely use hybrid positioning such as this in order to navigate.

Given that the instruments used in the final positioning system are the odometer itself, the PIC chip, and the Vector 2x compass, the system itself could be manufactured relatively efficiently and inexpensively. It does not require much in the way of hand calibration. Each part is standard and mass produced, except for the wheel, which easily

could be. The whole unit could be designed as a small self-contained package, which links to a main computer system. To keep costs down, the system could sacrifice some quality in the optical encoder, since very high resolution is not really necessary. One of the main problems that would occur in manufacturing is the fact that not all Vector 2x compasses are specified to work the same way. As discussed previously, there are certain parts of the communication protocol that are said to vary from compass to compass. Therefore, another digital compass may be a better choice for a mass-produced system.

In terms of ethics, when selling a positioning device such as this, it is extremely important not to misrepresent its performance. One must clearly state the system's limitations, or else users may rely on it to perform tasks that it cannot do. For instance the facts that the system performs poorly on steep inclines and in heavy magnetic fields must be made clear to any customer. Otherwise, the customer may use it believing it is reliable, and end up colliding into another object, harming their vehicle or themselves. One must keep these ethical issues in mind when marketing a device as crucial to many systems as a navigation tool.

A compass and odometry-based dead reckoning system is built of easily-available devices and can be manufactured at a low cost. For the dollar, it provides useful information reliably, especially under low external magnetic field conditions. It complements the Global Positioning System very well. Overall, it is an interesting and practical tool, for the Whitney autonomous vehicle project as well as many other applications.

Special thanks to:

Professor Roman Kuc,
for providing guidance and ideas throughout the entire project.

Ed Jackson,
for helping me polish my ideas and iron out my bugs.

Andy Nelson,
for helping design and build the odometer wheel mount.

Michael Liu,
for aiding in the communication between PIC and EBX.

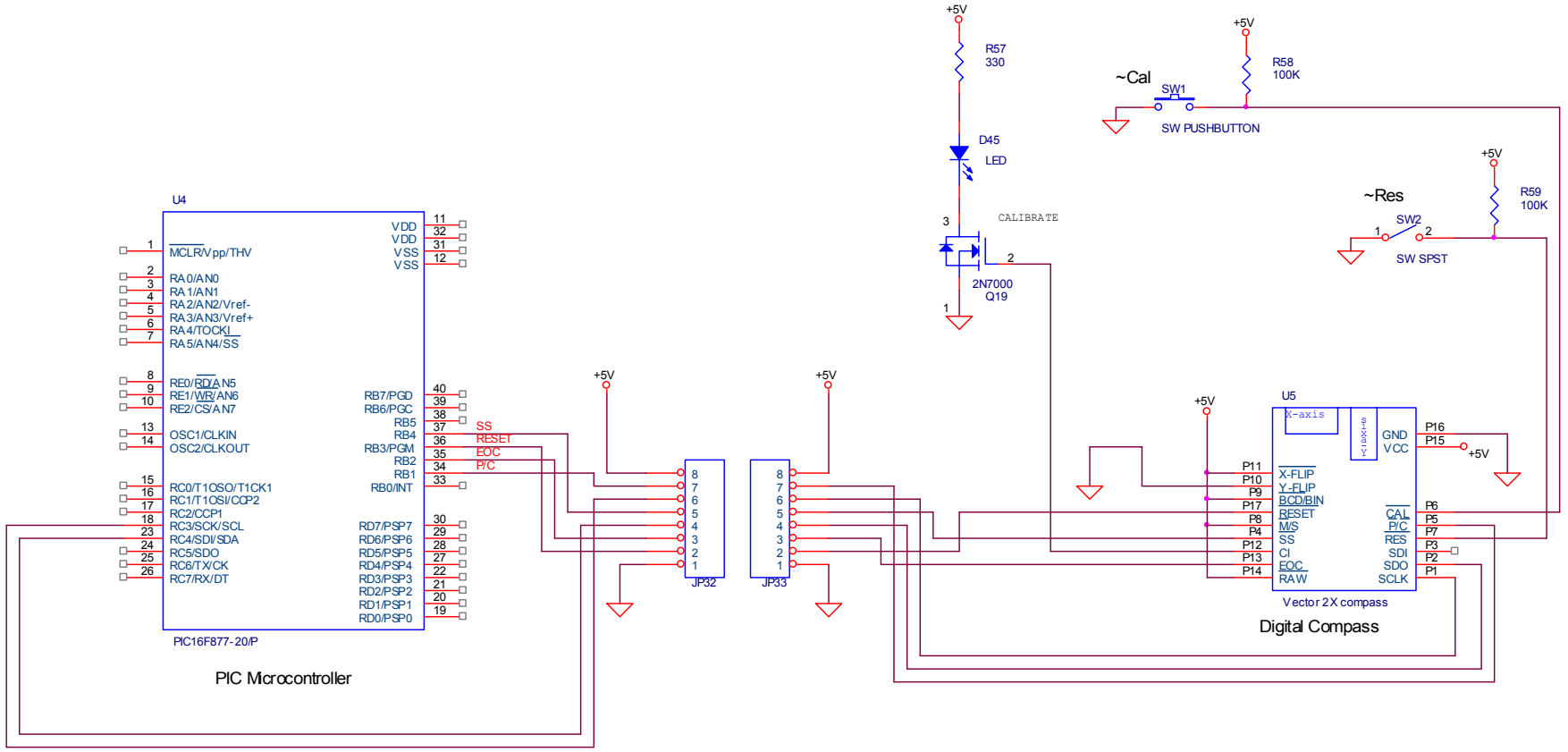
Appendix A: Circuit Diagrams

Compass

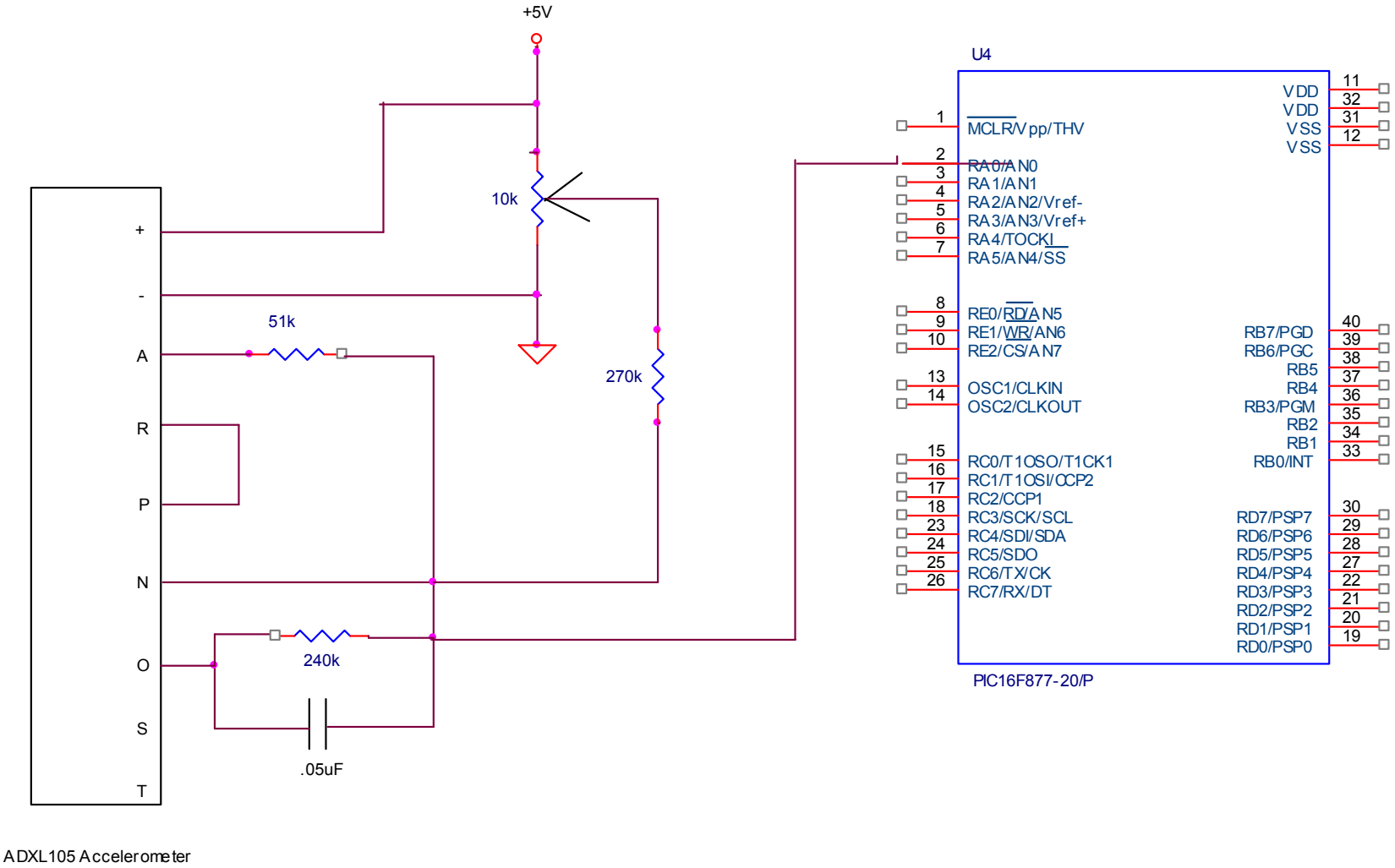
Accelerometer

Optical Encoder

Vector 2x Digital Compass Circuit

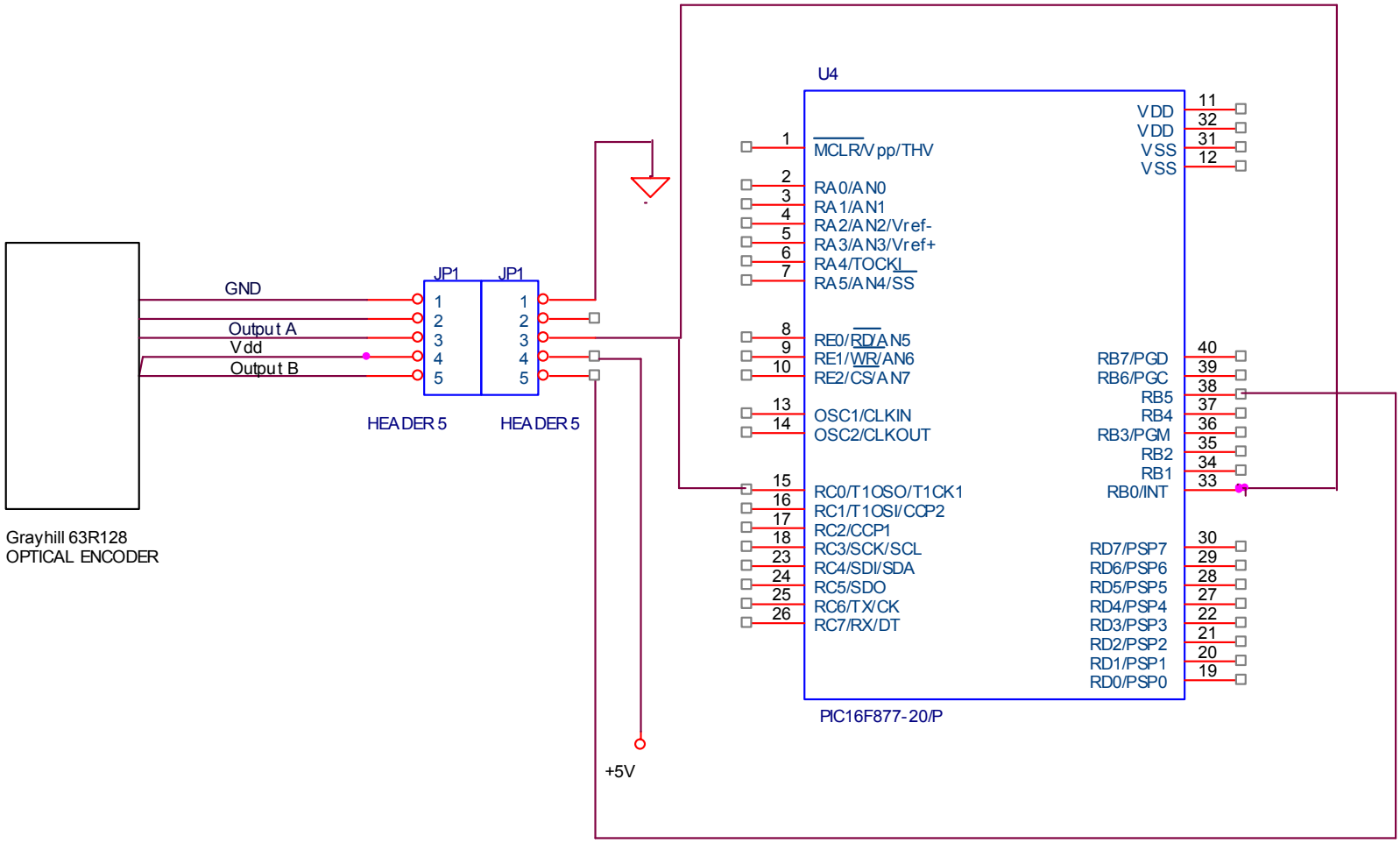


Accelerometer Circuit



ADXL105 Accelerometer

Optical Encoder Circuit



Appendix B: Programs

Comp5.h

Alan.h

Accel_7.c

Comp06.c

Comp07.c

Pos.c

Comp5.h

```
/*=====================================================================
==
•  First PIC16F877 program for the Whitney chair robot functions
•  This file contains the basic functions needed to get the robot to
   function.
*
•  Rev 0 2/14/2002 EWJ JS
*/

#include "16f877.h"
#device PIC16F877

#use delay (clock=10000000) // Robot with 20 MHz crystal oscillator
                           // Use HS selection in MPLAB ICD
                           // This sets up correct delay times

// Define serial communication to computer

#use rs232(baud=19200, xmit=PIN_c6, parity=N, bits=8)


#define ALL_OUT 0          // Used for data direction register
#define ALL_IN  0xff       // Used for data direction register

/*
•  ---------------------------------------------------------------------
   ------
•  functions
•  ---------------------------------------------------------------------
   ------
*/

long int read_bit(long int d)
      {
      delay_us(1);
  output_low(pin_c3);
  delay_us(1);
  output_high(pin_c3);
  d = (d<<1) | (input(pin_c4));
  return(d);
  }


//----- Reset vector compass
void reset_compass()
{
  output_high(pin_b1);    // Make P/C high
  output_high(pin_b4);    // Make sure ss high
  output_low(pin_b3);     // bring reset low
  delay_ms(20);           // Delay at least 10 ms
  output_high(pin_b3);    // return high
  delay_ms(1000);         // Delay at least 500 ms
```

```
}

//----- Get heading
long int get_compass()
{
  int i;
  long int heading;
  output_low(pin_b1);     // Pulse P/C low
  delay_ms(15);           // Delay at least 10 ms
  output_high(pin_b1);
  while(!input(pin_b2)){} // Wait for EOC to go high
  delay_ms(15);           // Delay at least 10 ms
  output_low(pin_b4);     // take SS low
  delay_ms(5);

  for (i=0;i<=7;++i)
        {
            heading = read_bit(heading);
    }
  delay_ms(5);
  for (i=0;i<=7;++i)
        {
            heading = read_bit(heading);
    }
  output_high(pin_b4);    // take SS high
  return(heading);
}
```

Alan.h

```
////////////////////////////////////////////////////////////////////////
/////
////                             alan.h
////
////////////////////////////////////////////////////////////////////////
/////
// 4/13/02
// Rev 0

#include "16f877.h"
#device PIC16F877

#use delay (clock=20000000) // Robot with 20 MHz crystal oscillator
                           // Use HS selection in MPLAB ICD
                           // This sets up correct delay times


/* Set Up LCD ------------------------------------------------------
-- */
// As defined in the following structure the pin connection is as
follows:
//     D0  D0
//     D1  D1
//     D2  D2
//     D3  D3
//     D4  D4
//     D5  D5
//     D6  D6
//     D7  D7

//     E0  RS
//     E1  R/W
//     E2  E

#byte PortD = 8              // LCD connected to port D

#define E   pin_E2
#define RW  pin_E1
#define RS  pin_E0
#define all_out 0x00
#define all_in 0xff

byte lcd_busy_flag()
{
  byte di;
  output_low(RS);            // Set to instruction
  output_high(RW);           // Set to read
  delay_cycles(1);           // Clock / 4 for each cycle .2uS @ 20 MHz
  output_high(E);            // enable display
  delay_cycles(3);
  di=portD & 0x80;
  output_low(E);             // when enable brought read over
  return di;
}
```

```
void lcd_write_byte(byte a,byte n)
{
  set_tris_D(all_in);
  while (lcd_busy_flag() > 0);
  set_tris_D(all_out);
  output_bit(RS,a);         // Set to write instruction or data
  output_low(RW);           // Set to write
  delay_cycles(1);
  PortD = n;                // put data out
  output_high(E);           // enable display
  delay_cycles(3);
  output_low(E);            // when enable brought low data written
}

void init_LCD()
{
  lcd_write_byte(0,0x30);   // software reset sequence
  delay_ms(5);
  lcd_write_byte(0,0x30);
  delay_us(200);
  lcd_write_byte(0,0x30);   // 8 bit mode
  lcd_write_byte(0,0x38);   // 8 bit 2 lines (1x16 is really 2x8)
  lcd_write_byte(0,0x08);   // Display off
  lcd_write_byte(0,0x01);   // Clear display
  delay_us(50);
  lcd_write_byte(0,0x06);   // Moving cursor not display
  lcd_write_byte(0,0x0c);   // Display on no cursor
  lcd_write_byte(0,0x02);   // Return Home
}

void lcd_goto( byte x, byte y)
{
   byte address;

   if(y!=1)
     address=0x40;
   else
     address=0;
   address += x-1;
   lcd_write_byte(0,0x80|address);
   delay_us(50);
}

void lcd_putc( char c)
{
   switch I
   {
     case '@':
       lcd_write_byte(0,1);
       break;
     case '\a':
       lcd_goto(1,1);
       break;
     case '\b':
       lcd_goto(1,2);
       break;
```

```
     default:
       lcd_write_byte(1,c);
       break;
   }
}

/* Set Up PWM ------------------------------------------------------
-- */
void init_PWM()
{
  setup_ccp1(CCP_PWM);   // Configure CCP1 as a PWM
  setup_ccp2(CCP_PWM);   // Configure CCP2 as a PWM

  // The cycle time will be (1/clock)*4*t2div*(period+1)
  setup_timer_2(T2_DIV_BY_16, 255, 16);
  // 1/20000000 * 4 * 16 = 3.2uS timer 2 rate
  // overflow = 3.2uS * 255 =.816mS
  // interrupt every .816mS * 16 = .013 sec (If timer2 software
interrupt enabled)
}

/* Setup A/D -------------------------------------------------------
---- */
void init_AD()
{
  setup_adc(adc_clock_div_32);
  setup_adc_ports(RA0_RA1_RA3_ANALOG);
}

/* Input A/D -------------------------------------------------------
---- */
long int in_ad(byte ch)
{
  set_adc_channel(ch);
  delay_us(100);
  return read_adc();
}

/* Timer interrupt setup-------------------------------------------
---- */
void init_Timer0()
{
  set_rtcc(0);           // initialize timer0
  // select internal .2uS clock with prescaler set to divide by 16 =
3.2uS
  setup_counters(RTCC_INTERNAL, RTCC_DIV_16);
}
```

Accel_7.c

```
/*====================================================================
==
•  Accelerometer Software
•  by Alan Ghelberg
•  Converts Acceleration to Distance
•  Rev 1 5/5/02
*
*/

#include "alan.h"

/*
•  --------------------------------------------------------------------
   ------
•  Definitions
•  --------------------------------------------------------------------
   ------
*/

//Multiplier calculated as (slope of Voltage-Accleration (in cm)
Graph)*timeslce (in ms)

#define multiplier .4 // ((431-(-525))/1960)*.8192


/*
•  --------------------------------------------------------------------
   ------
•  Global Variables
•  --------------------------------------------------------------------
   ------
*/
  // Zero-G Bias, Acceleration, velocity, velocity counter,
  // velocity differential term, position counter
  signed long zeroG, accel, vel, velCounter, velDif, posCounter;
  long int pos;          //position

  short flag1, test;  //flag set in interrupt to run main, test flag
for interrupt duration
  byte count1; //counter to decide when to divide vel by 1000 and reset
to zero

  char StringOut[55];  // LCD out string
  byte in_index;
  byte out_index;
  byte i, outflag, CharNum;

/*
•  --------------------------------------------------------------------
   ------
•  Interrupts
```

```
•  ------------------------------------------------------------------
   ------
*/
// Specify Timer 0 as interrupt
// An interrupt occurs when timer0 overflows
// timer0 setup is init_Timer0() from yale.h
//
// If more tasks are to be done during the interrupt then the □llotted
time, then a flag can
// be used to divide the interrupt into time slices.
// Clock to timer0 is 20MHz/4 or .2 uS
// Div by 16 prescaler or 3.2 uS
// Since timer0 must go from 0  to 256 or 819.2 uS

#INT_RTCC               // This must be just before clock_isr()
clock_isr()
{

//---------------------------
// Test code to create high/low signal output for scope examination of
timing
// It is helpful in an interrupt driven system.  This signal can be
used along with others
// for each task to verify that all code is executing in the □llotted
time.
// This code can be eliminated
  if (test)             // Each time interrupt is called C7 is output
high or low
  {                     // so for this timing 409.6 uS high and 409.6
uS low
    output_low(pin_c7);
    test=0;
  }
  else
  {
    output_high(pin_c7);
    test=1;
  }
//---------------------------
// Get A/D reading

  //output_high(pin_c6);  // Test code to time A/D with scope on C6
high during conversion

  flag1 = 1;   //set flag high in interrupt to run functions in main

  //output_low(pin_c6);   // Test code to time A/D with scope return
low

//---------------------------
// Output one character to LCD display if new message in buffer

  //output_high(pin_c5);  // Test code to time LCD out with scope on C5
high during output

  if (outflag)
  {
```

```
    lcd_putc(StringOut[out_index]);       // send to LCD
    if (out_index < CharNum) out_index++; // Check if last character in
message
    else
    {
      outflag = 0;                        // If last one reset flags
for new message
      out_index = 0;
    }
  }
  //output_low(pin_c5);   // Test code to time LCD out with scope
}

/*
•  ---------------------------------------------------------------------
   ------
•  functions
•  ---------------------------------------------------------------------
   ------
*/

//---------------------------
// Initialize LCD message buffer
void init_string()
{
  for (i=0; i<55; i++)
  {
    StringOut[i]=' ';
  }
}

//---------------------------
// Setup buffer for output to display
void LCDmessage(char s)
{
  if (in_index < 55)            // Don't do if over buffer limit
  {
    if (s == '\r')             // If end of transmission get number of
characters
    {
      CharNum = in_index-1;     // Get total number of characters
      in_index = 0;             // Reset for next time
      outflag = 1;              // We have complete message. Set flag
so output can happen
    }
    else                        // Not finished yet so add character to
buffer
    {
      StringOut[in_index]=s;    // Build up string for sending
      in_index++;
    }
  }
  else
  {
    CharNum = 48;               // Beyond buffer size with no end so
cut to 48
```

```
    in_index = 1;
    outflag = 1;                // Set flag so we send what we have
  }
}

//This function initializes the accelerometer variables for maximum
accuracy
//It finds the zeroG bias by taking 64 samples and averaging

void init_accel()
{
      // Assume vehicle is stopped and remember zeroG voltage
      for (i=0; i<64; i++)
       {
        zeroG = zeroG + read_adc();
        delay_ms(20);
   }
      zeroG = zeroG>>6;  //shift right by 6 same as divide by 64
}

/*
•  ------------------------------------------------------------------
   ------
•  Main Program
•  ------------------------------------------------------------------
   ------
*/

main()
{
  init_AD();                        // Set up A/d converter
  set_adc_channel(0);                                 // Set up
Channel 0
  delay_us(100);
  init_string();
  init_LCD();                       // Set up LCD display
  init_Timer0();
  outflag=0;
  flag1 = 0;
  in_index=1;
  zeroG = 0;
  vel = 0;
  velCounter = 0;
  pos = 0;
  posCounter = 0;
  count1 = 0;
  init_accel();


  // Enable interrupts on timer 0
  enable_interrupts(RTCC_ZERO);

  // Activate enabled interrupts
  enable_interrupts(GLOBAL);
```

```
  if (!outflag) printf(LCDmessage,"@\r");
                                    // @ clears the display
                                    // \a is start of first LCD line
                                    // \b is start of second LCD line
                                    // \r is end of transmission MUST be
included
  while(outflag){}                  // wait for last message if needed

  if (!outflag) printf(LCDmessage,"\a**Welcome EE350**\b 16F877 Robot
\r");
  while(outflag){}
  delay_ms(1500);                   // delay time for welcome message
  printf(LCDmessage,"@\r");
  delay_ms(100);

  while(1)
  {
  if (flag1 == 1)
  {
      // Input acceleration A/D input on channel 0 from 0 to 5V on A/D
input
      // Oversample 8 times and take average
      accel = 0;
      for (i=0; i<8; i++)
       {
        accel = accel + read_adc();
       }

      //take average and subtract out the zero G bias, centering around
zero

      accel = (accel>>3) - zeroG;

      //assume velocity remains under 32 cm/s, else overflow problem

      vel = vel + multiplier * (accel);


      pos = pos + .8192 * vel;   //differential term for velocity(vel *
time interval (in ms))


      //System to protect from position overflows.  Position will
overflow regularly,
      //since it is measured in millionths of centimeters (to keep
accuracy in integration
      //Position overflow counter should be incremented if velocity
goes above 16000
      //It should be decremented if velocity goes below -16000
      //Therefore the actual position is (1/1000000)*(16000 velcounter
+ vel) cm


      if ((pos >= -16000) && (pos <= 16000)){}  //position does not
overflow
      else
      {
```

```
        if (pos > 16000)                          //overflow increment
        {
          pos = pos - 16000;
          posCounter++;}
            else
            {pos = pos + 16000;              //overflow decrement
            posCounter--;}
      }


      flag1 = 0;                                  //reset flag to zero so
code does not rerun
                                          //until next interrupt
  };

  //output values to LCD
  if (!outflag) printf(LCDmessage,"\aacc=%6ld vel=%6ld\bpC=%6ld
pos=%6ld\r", accel, vel, posCounter, pos);

  }
}
```

Comp06.c

```
/*=====================================================================
==
* First program to implement positioning on PIC using compass data and
* odometer data.  This program uses interrupts to count odometer
cycles.
* 5/9/02 Alan Ghelberg
*/

#include "comp5.h"   /* include file */.
#include "stdlib.h"   /* for absolute value function */

long int degrees;                    //heading
signed long counter =0;              //odometer Count
long int count =0;                   //used as absolute value of counter
int packnum=0;                       //data packet numbering
short countsign;                     //the sign of counter (+/-)

/*
* ---------------------------------------------------------------------
-----
* Main Program
* ---------------------------------------------------------------------
-----
*/

/*
* ---------------------------------------------------------------------
-----
* Interrupts
* ---------------------------------------------------------------------
-----
*/


#INT_EXT
void odometer()
{
  if (input(pin_b5))                 //Wheel counterclockwise --->
reverse
    {counter--;}
  else
    {counter++;}                     //clockwise ---> forward
}




main()
{

  enable_interrupts(INT_EXT);              //enable external interrupt
  enable_interrupts(GLOBAL);               //enable global interrupt
  reset_compass();
  while(1)          // Endless loop
```

```
    {
    degrees = get_compass();              //poll compass for heading
    packnum++;                            //increment packet number

    //counter is broken up into sign and absolute value so that a
fixed-
    // width packet may be sent
    if (counter >= 0)         //countsign is 1 if count is positive
      {countsign = 0;}  // Print for positive case
    else                  //countsign is 0 if count is negative
      {countsign = 1;}
    count = abs(counter);
    counter = 0;                          //reset odometer counter


    disable_interrupts(GLOBAL);           //disable interrupts for
RS232
    printf("(%3lu|%1u|%5lu|%3u)",degrees, countsign, count,packnum);
    enable_interrupts(GLOBAL);            //re-enable interrupts

  }
}
```

Comp07.c

```
/*====================================================================
==
* Second program to implement positioning on PIC using compass data and
* odometer data.  This program uses the PIC's internal counter to count
* odometer cycles.  Therefore, it only works for forward motion.
* 5/9/02 Alan Ghelberg
*/

#include "comp5.h"   /* include file */.
#include "stdlib.h"   /* for absolute value function */

long int degrees;             //heading
long int counter =0;          //odometer counter

int packnum=0;                //data packet number

/*
* --------------------------------------------------------------------
-----
* Main Program
* --------------------------------------------------------------------
-----
*/



main()
{
  setup_timer_1(T1_External);             //Initialize timer to count
external source

  reset_compass();                        // Initialize compass
  while(1)          // Endless loop
    {
    degrees = get_compass();              // Poll compass for heading
    packnum++;                            // Increment Packet number
    counter = get_timer1();               // Pull odometer count from
counter
    set_timer1(0);                        // reset count to zero

    printf("(%3lu|0|%5lu|%3u)",degrees, counter,packnum);   // Print
for positive case

  }
}
```

Pos.c

```
/* Position Calculation Program for EBX
                              */
/* This prgram recieves compass and odometry data from a PIC and calculates x/y
position            */
/* Alan Ghelberg 5/8/02
                              */
/* RS232 protocol initially from Get Compass SubProgram Mike Liu 4/13/02
             */



                                                                          */
/* COM1: Compass/Odometer
                     */

/* COM2: LCD Display
                              */

/* COM3: Motor Control
                     */

/* COM4: GPS/Sonar
                              */



/* Main Header File that contains other include files
                     */
#include "full.h"

/* GLOBAL VARIABLE DEFINITIONS
                     */
double total_distance;
double x_coordinate;
double y_coordinate;
signed long total_count;

/* EXTERNAL VARIABLE DECLARATIONS
                     */
extern double total_distance;
extern double x_coordinate;
extern double y_coordinate;
extern long total_count;

void main(void)
{

       int Seg1, Seg2, Seg3, j, i, sec=0, hund=0;
       unsigned long newtime=0,oldtime=0;
       unsigned long deg;
       short sign;
       signed long int count;
       unsigned int packnum;
       unsigned int oldpacknum = 0;
       char string[50]="";
       char *strptr;
       char LCDstring[21];
       char far *Ptr1, *Ptr2, *Ptr3;
       struct time timebuf;
```

```
/* !!! SETUP Simultaneous COM Port Stuff !!! */

       SioPorts(4,4,0,PC_PORTS);
       SioUART(COM1,0x03f8);
       SioUART(COM2,0x02f8);
       SioUART(COM3,0x03e8);
       SioUART(COM4,0x02e8);
       SioIRQ(COM1,4);
       SioIRQ(COM2,3);
       SioIRQ(COM3,11);
       SioIRQ(COM4,10);

/* !!! SETUP COM1 !!! */
       // setup 128 byte receive buffer
       Ptr1 = (char far *)RxBuffer1;
       Seg1 = FP_SEG(Ptr1) + ((FP_OFF(Ptr1)+15)>>4);
       SioRxBuf(COM1,Seg1,Size128);
                                                    // setup 128 byte transmit
buffer
       Ptr1 = (char far *)TxBuffer1;
       Seg1 = FP_SEG(Ptr1) + ((FP_OFF(Ptr1)+15)>>4);
       SioTxBuf(COM1,Seg1,Size128);
                                                    // set port parmameters &
reset port
       SioParms(COM1,NoParity,OneStopBit,WordLength8);
       SioReset(COM1,Baud19200);


/* !!! SETUP COM2 !!! */
                                                    /* setup 128 byte receive
buffer */
       Ptr2 = (char far *)RxBuffer2;
       Seg2 = FP_SEG(Ptr2) + ((FP_OFF(Ptr2)+15)>>4);
       SioRxBuf(COM2,Seg2,Size128);
                                                    /* setup 128 byte transmit
buffer */
       Ptr2 = (char far *)TxBuffer2;
       Seg2 = FP_SEG(Ptr2) + ((FP_OFF(Ptr2)+15)>>4);
       SioTxBuf(COM2,Seg2,Size128);
                                                    /* set port parmameters &
reset port */
       SioParms(COM2,NoParity,OneStopBit,WordLength8);
       SioReset(COM2,Baud19200);

/* !!! SETUP COM3 !!! */
                                                     /* setup 128 byte receive
buffer */
       Ptr3 = (char far *)RxBuffer3;
       Seg3 = FP_SEG(Ptr3) + ((FP_OFF(Ptr3)+15)>>4);
       SioRxBuf(COM3,Seg3,Size128);
                                                    /* setup 128 byte transmit
buffer */
       Ptr3 = (char far *)TxBuffer3;
       Seg3 = FP_SEG(Ptr3) + ((FP_OFF(Ptr3)+15)>>4);
       SioTxBuf(COM3,Seg3,Size128);
                                                     /* set port parmameters &
reset port */
       SioParms(COM3,NoParity,OneStopBit,WordLength8);
       SioReset(COM3,Baud19200);


       fflush(stdin);
```

```
                                                        // Set auto line
wrapping on
       SioPutc(COM2,'\xfe');

       SioPutc(COM2,'\x43');
                                                        // Set auto scroll
on
       SioPutc(COM2,'\xfe');
       SioPutc(COM2,'\x51');
                                                        // Set underline
cursor off
       SioPutc(COM2,'\xfe');
       SioPutc(COM2,'\x4b');
                                                        // Clear Screen
       SioPutc(COM2,'\f');

// Initialize variables to zero

       // INITIALIZE VARIABLES
       total_distance=0;
       x_coordinate=0;
       y_coordinate=0;
       total_count=0;
       oldtime = hund;

       while (1)
              {

              gettime(&timebuf);
              hund = timebuf.ti_hund;
              newtime = hund;

              if ((newtime > oldtime+5) || (newtime < oldtime))  //Only run
every 50ms or whenever hundredths overflows
                     {
                     oldtime = newtime;
                     if (SioRxQue(COM1) == 17)                  //Only run if
whole 17byte packet is in bufffer
                            {
                            SioGets(COM1, string, 34);          // get packet
from buffer

                            sscanf(string, "(%3lu|%hd|%5ld|%3u)", &deg, &sign,
&count, &packnum); //Extract info from packet

                            if (packnum != oldpacknum)          //If packet is
new
                                   {
                                   printf("String = |%s|\n",string);
                                   oldpacknum = packnum;               // reset
oldpacknum to new packnum

                                   if (sign)                           //if sign
is one odometry count is negative
                                          count = count * -1;

                                   total_count = total_count + count;  // update
total odometer count
                                   update_position(count,deg);             //
call update position function

                                   //Print all information to string
```

```
                                    sprintf(LCDstring,"Count = %5ld\nDir  =
%3lu\n(X,Y) = (%2.1f,%2.1f)\nTotal = %2.2f\n",
                                          total_count, deg, x_coordinate,
y_coordinate, total_distance);

                                    //Print to monitor

                                    printf("|%s|",LCDstring);

                                    //Print to LCD display

                                    SioPuts(COM2,LCDstring,strlen(LCDstring));


                                    SioRxClear(COM1);
      //Clear Buffer
                                    strcpy(string,"");                //Clear
String
                                    }
                              }

                  }
            if ((i=SioGetc(COM2,0)) > -1)                           /* was key
pressed ? */
                  {
                  switch (i)
                        {
                        case 'O':                                         // "#"
Pressed?
                              total_distance=0;
      //reset all variables
                              x_coordinate=0;
                              y_coordinate=0;
                              total_count=0;
                              SioPutc(COM2,'\f');
                        break;
                        }
                  }



            if (kbhit())                                            //CTRL+z
exits program
                  {
                  j = getch();
                  if ((char) j == CTLZ)
                        {
                        SioPutc(COM1,'\f');
                        SioDone(COM1);
                        SioDone(COM2);
                        SioDone(COM3);
                        SioDone(COM4);
                        exit(0);
                        }
                  return;
                  }
            }

      return;
}
```

```
/*  Function: updatePosition

                                    */
/*  Input:   current odometer reading, current compass reading
                                     */
/*  Output: void

                                                                  */
/*  Purpose: Subfunction to update global position variables
                                    */
/*                                   double total_distance, x_coordinate,
y_coordinate;                                                     */
/*
*/
void update_position(signed long int count, long int degrees) {

       double distance;
       float theta;
       distance = .001983 * count; //convert count into distance measurement
       in meters
       theta = 90 - degrees;                                      //shift reference
frame such that 0 degrees is y-axis
       theta= .01745*theta;
                                                          //convert to radians
       x_coordinate = x_coordinate + distance*cos(theta);    //recalculate x
       y_coordinate = y_coordinate + distance*sin(theta);    //recalculate y
       total_distance += distance;

}
```

Appendix C: Weekly Reports

Hybrid GPS/Accelerometer Navigation System

Alan Ghelberg
January 29, 2002

Why do we need an additional navigation method?

Currently our navigation system includes a GPS system. However, problems with GPS include the facts that:

- Signal is easily degraded or lost due to obstructions such as buildings or trees
- Excessive granularity of reading, i.e. often cannot obtain position more precisely than within 20 feet

Given these limitations, it is useful to supplement GPS-based navigation with some mode of local sensing. A system using accelerometers has the following advantages:

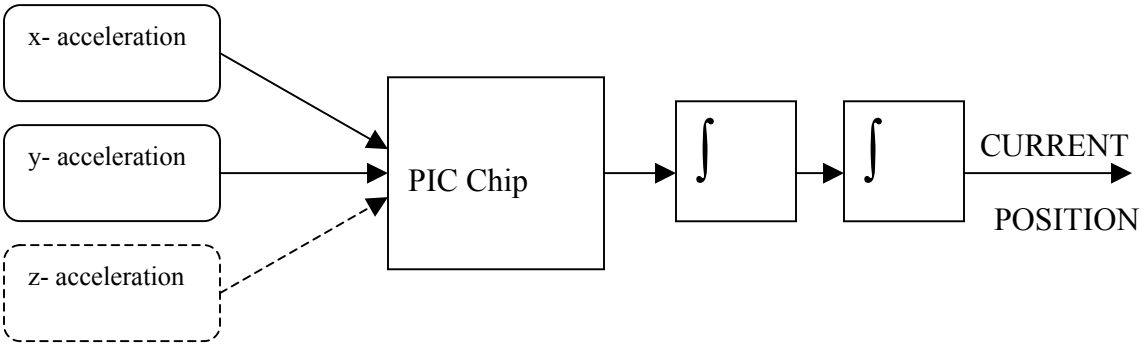
- Entirely self-contained and therefore not subject to external obstructions
- Possible to obtain high accuracy for limited time periods
- Not subject to wheel slippage error, such as velocity measurements

Given these facts, an ideal navigation system will contain both GPS sensing and the local sensing provided by accelerometers. This hybrid navigation system will intelligently switch between both position acquisition techniques. For instance, if the system feels that not enough GPS satellites are available it may switch to local sensing.

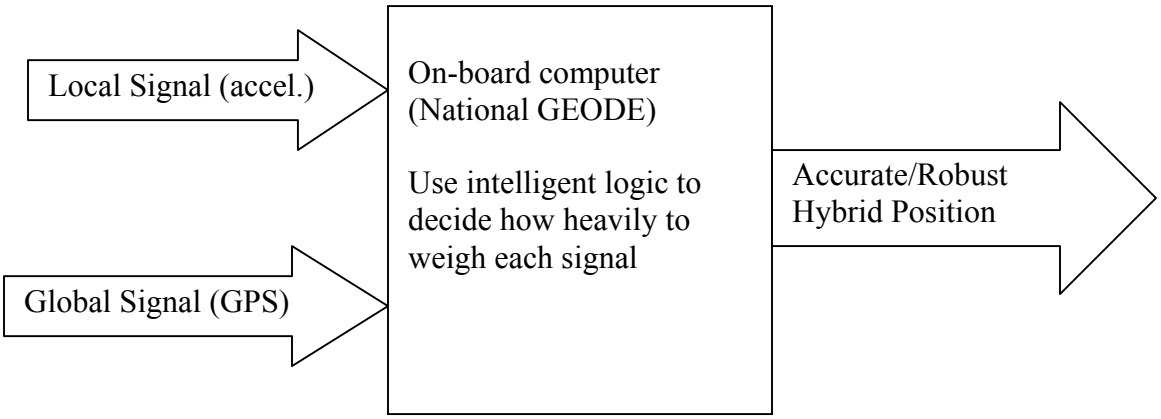
Idea behind Accelerometer-based navigation:

I will use at least two accelerometers, one for the x-direction, and one for the y-direction (and possibly a third z-component as well). By taking the acceleration signals and double-integrating them (on a PIC), I will obtain position. The accelerometers may be mounted on a gyroscope in order to keep a constant reference frame. If the accelerometers are not on a gyroscope, the reference frame will be constantly changing, which must be accounted for.

Accelerometer System



Hybrid System



Next Steps

- Research and experiments on current accelerometers (Analog Devices) ADXL05EM-1 to determine suitability for task
- Explore the pros and cons of using gyroscope system. Decide if necessary or not.

Mobile Wheelchair Project – Hybrid Navigation System
Week 2 Handout

This Week

- Experimented with current accelerometers (ADXL05EM-1)
    - Give a response in acceleration Range we will look at
    - Need more tests to determine accuracy within this range

- Explored issues to be dealt with when using accelerometers
    - Granularity of acceleration data (10mG)
    - Noise
    - Stationary Drift (Even when not moving, an acceleration signal occurs
        - This Drift changes with temperature, etc.
- Nonlinearity in accelerometer signal

Next Week

- Explore means of compensating for these issues
- Use low pass filter to filter out noise
- Figure out how to remove stationary drift
- Investigate precise nature of nonlinearity, so can be compensated for on PIC

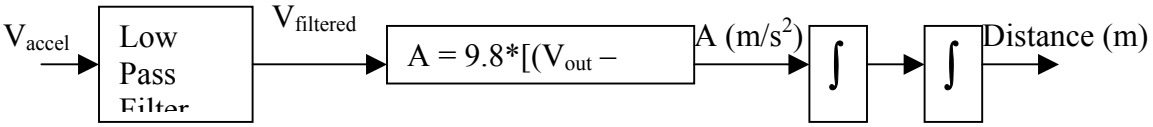
YAV

Alan Ghelberg
Week 3

This Week

- Attempts to improve accelerometer signal
- Signal was full of high-frequency noise (so applied low-pass filter)

- Tested use of accelerometer to obtain position
- Ran a controlled test by using one accelerometer and moving it in the direction of the accelerometer's sensitivity
- Take accelerometer's voltage output and transform into actual acceleration reading
    - First test +g and –g voltage and then assume linearity between these points
        - Acceleration = 9.8*[(Vout – 2.5)/.5]
- Moved accelerometer straight 37 cm, and read on oscilloscope
    - Imported onto computer, then calculate for position (take numerical integrals)
        - $V_f = V_I + Adt$
        - $X_f = X_I + Vdt$



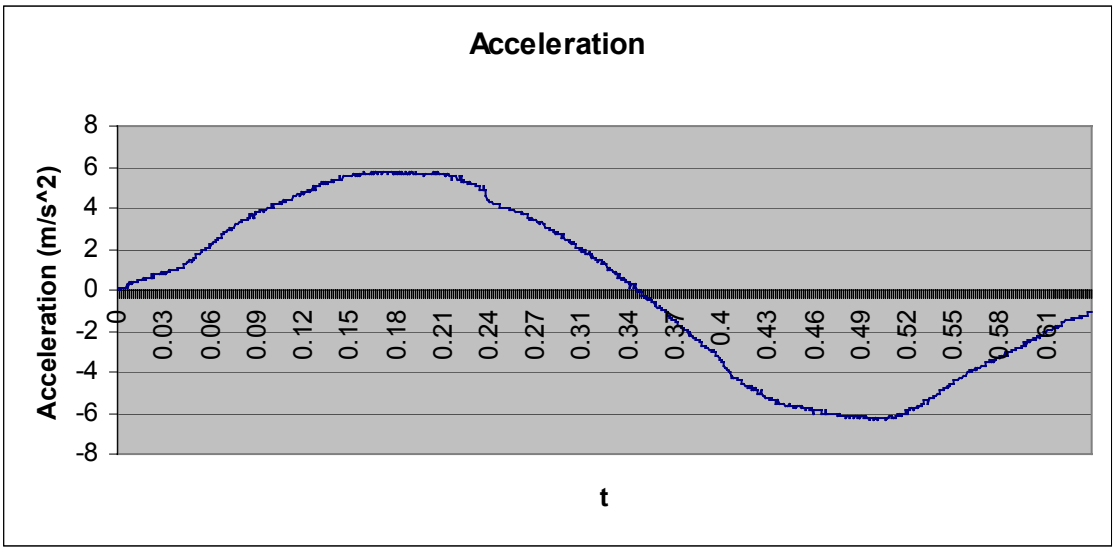
- Results
    - Total distance traveled was determined to be 36.7 cm
- Good results considering not much compensation for signal errors

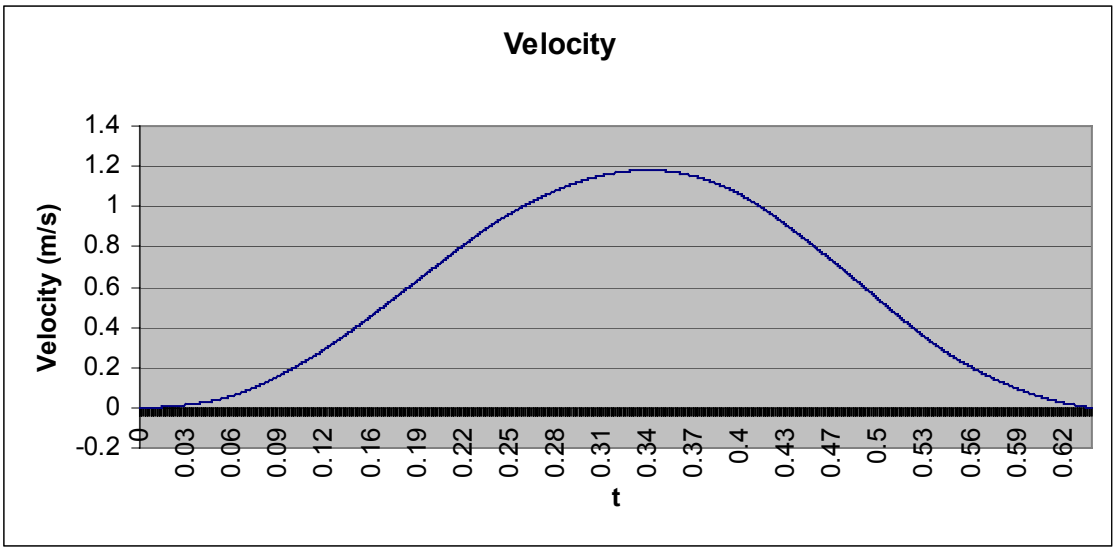
Issues

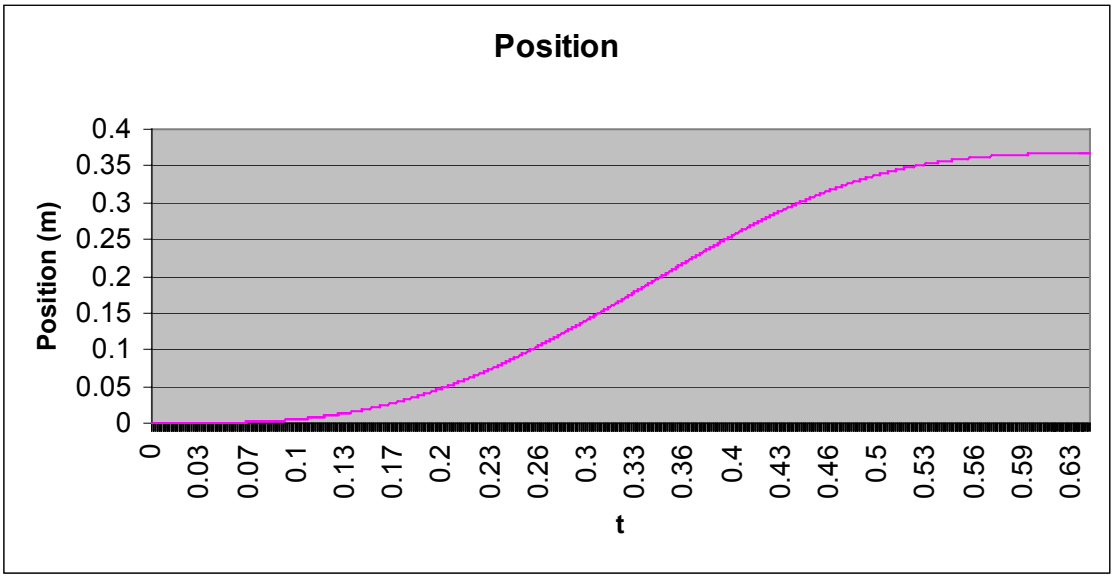
- Need to determine more methods to improve accelerometer signal

Next week

- Code positioning algorithm for PIC
- Switch to using ADXL05 accelerometer
    - Newer and more accurate
    - Must amplify signal over certain range to take advantage of greater resolution
- Test compass as a means of detecting direction
- Develop algorithm for combining acceleration/compass direction to calculate 2-d position


Acceleration
Acceleration (m/s^2)
8
6
4
2
0
-2
-4
-6
-8
0
0.03
0.06
0.09
0.12
0.15
0.18
0.21
0.24
0.27
0.31
0.34
0.37
0.4
0.43
0.46
0.49
0.52
0.55
0.58
0.61
t

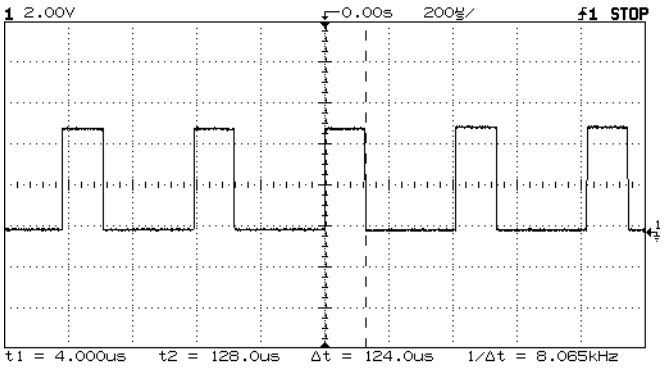

Velocity
Velocity (m/s)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.03
0.06
0.09
0.12
0.16
0.19
0.22
0.25
0.28
0.31
0.34
0.37
0.4
0.43
0.47
0.5
0.53
0.56
0.59
0.62
t


Position
Position (m)
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
0.03
0.07
0.1
0.13
0.17
0.2
0.23
0.26
0.3
0.33
0.36
0.4
0.43
0.46
0.5
0.53
0.56
0.59
0.63
t

Whitney
Alan Ghelberg
Week 4

This Week

- Implemented a velocity-calculating program on PIC
- From AD converter, acceleration signal is scaled from 0 to 1024
    - Want to rescale to get acceleration data
        - Subtract out Zero-G bias (averaged over 10 samples)
        - Extrapolate multiplying factor to get acceleration value
- $V_f = V_I + Adt$
- Use interrupts to call subroutine at known interval dt
    - Must therefore make sure that calculations within this interval complete before next interrupt is called
    - Use timing diagrams



Issues

- Currently error increases very rapidly
    - Must explore means of reducing error, such as over-sampling
- The program currently runs slowly, which doesn't present such a problem now but as complexity increases, will need to be streamlined. Otherwise the time intervals will be too long and accuracy will suffer.
    - Currently using floating-point numbers because I need decimal accuracy, but this is difficult for PIC to process.
        - I eventually need to convert to integers, but will need a means of keeping accuracy, such as basing times on 1 ms
    -

Next week

- Calculate Position on PIC
- Increase accuracy of program
- Begin integrating Compass into system

Project Whitney
Alan Ghelberg
Week 5

This Week / Issues:

This week I did more work on the implementation of the position-sensing algorithm on the PIC chip. There was work to be done in terms of working around the PIC's limitations in order to get the program somewhat operational. The PIC could not handle any math done within an interrupt call, as I had previously implemented the code. To fix this I only set a flag within the interrupt to 1, which tells the code in the main program to run (so that it only runs once every interrupt)

I also developed a method of handling numbers that will allow the level of accuracy I need in my numbers, without using floating point. The issue is that the PIC takes in a voltage value form the A/D converter which ranges from zero to 1027. Then the program has to implement the algorithm that:

- $V_f = V_I + A\Delta t$
- $X_f = X_I + V\Delta t$

However, since the time scale is small, on the order of 1 millisecond, then for each time slice, multiplying by Δt will be equivalent to dividing by 1000 and therefore destroy the accuracy, since I am not using decimals. The solution is to let $\Delta t = 1$, and keep in mind that the actual velocity value is $1/1000^{th}$ of the stored value and the actual position is $1/1000000^{th}$ of the stored value. The problem that this creates is that the values will overflow all the time since they are long signed integers which can range from –32768 to +32768. Therefore, within 32 time increments, the variables will overflow. The method I've decided to use to correct for this is to increment a counter to count the number of overflows. In this case:

- $V = m * 32768 + n * 1$ and
- $X_f = (s * 32768 + t * 1) + (m * 32768 + n * 1)\Delta t$

where m,n,s, and t are integers themselves ranging from –32768 to +32768. Therefore the calculation of X_f can be factored out into:

- $s_f = s_I + m$ and
- $t_f = t_I + n$

I also need to account for when s overflows, by putting some conditions into the code. Using these algorithms will not require any multiplication or division, only addition and subtraction. This will allow each cycle to be completed more quickly.

Next week

- Finish coding linear position calculation on PIC
- Start using new accelerometer (ADXL105)
    - Filtering and rescaling of signal
- Begin integrating Compass into system

Project Whitney
Alan Ghelberg
Weeks 6 & 7

This Week:

This week I completed a program for linear position sensing on the PIC. Also, I began using the ADXL105 accelerometer instead of the ADXL05. The ADXL105 accelerometer has a resolution of 2 mG as opposed to 10 mG for the ADXL05. This should hopefully the system's accuracy. However, the PIC's AD converter is 10-bit, (0 to 1023) and the accelerometer's range is –5G to +5G. The effective resolution of the system would still only be 10G/1024 ≅ 10mG. Therefore I limited the accelerometer's range to only –1G to +1G by amplifying the signal, since the wheelchair should not undergo any accelerations outside this range. The result is that the system should theoretically make use of the 2mG resolution.

Issues / Next Steps:

Using the accelerometer causes several problems that are difficult to avoid. Inherently the system has some accuracy problems due to limited resolution, noise, etc.. Because of the double-integration, errors in the signal are compounded very rapidly. Even if the integration from acceleration to velocity leads to relatively small error, the error in calculating position may be to great. I have and will try more methods to minimize these problems. My current system uses oversampling as well as some filtration, but the errors are still significant.

Another problem with the use of accelerometers is that they are highly sensitive to any tilt. The current system uses one accelerometer oriented in the x-direction. However, if the wheelchair were to tilt down, the accelerometer would feel a strong acceleration due to gravity, which is not related at all to the actual wheelchair movement. As I show on the next page, if I use another accelerometer oriented vertically, it is theoretically possible to compensate for this, however in practice this would add another degree of inaccuracy into the system.

Given these limitations of the accelerometer-based system, I want to explore using velocity (as obtained from wheel rotation), along with the compass, in order to calculate position. This method, although subject to slippage, will likely be more robust. This week I intend to implement such a system and test it's reliability.

Project Whitney
Alan Ghelberg
Week 8

This Week:

This week I began work on a new navigation method which will use wheel rotation instead of acceleration in order to determine position. Wheel rotation will be determined either by using a strip of light/dark bands placed around the wheel, along with an IR LED and photo-detector to sense the bands. Therefore, by using the detector as an input to the PIC I can count the number of pulses and translate this into distance traveled.

The other means of measuring wheel rotation is to use an optical encoder, which Ed has ordered a sample of. This is essentially a self-contained box which attaches to the axle and counts turns. The advantages of this are that first, the encoder is enclosed and therefore not subject to interference from sunlight. Second, the resolution is very high on these devices, such as 128 steps per rotation. Lastly, these encoders can tell forward from reverse motion.

If the system gets a reliable angular turn count, along with compass data, then the x/y movement can be determined. The system counts wheel turns and whenever the direction from the compass refreshes (4 or 5 times per second), it is assumed that those turns occurred in the direction specified by the compass. Then the process begins again and the new movement is added to the previous.

This method should provide a relatively robust approximation of the actual vehicle movement, and hopefully slippage will not be huge factor. The strength of this method as opposed to the use of accelerometers is that no integration is necessary and therefore errors should be accumulated at a much slower rate.

Next Steps:

In the next few days I hope to implement a trial version of the system I have described. The PIC will take data from the compass and for wheel rotation, and send it to the main computer via RS232, where the actual navigation calculations will occur.

Project Whitney
Alan Ghelberg
Week 9

This Week:

This week I began the new navigation system using odometry along with the compass. Ed ordered an optical encoder which I will use for the odometry, the Grayhill 63R128. This encoder is highly accurate, with up to 128 cycles per revolution. It is also very durable with a life of 300 million revolutions.

Andy N. and I began work on the third wheel upon which the odometer will be mounted. It will have a spring to ensure that the wheel is always in contact with the ground.

I coded a test program which sends the compass heading, along with the odometry data to the computer via RS 232. The PIC counts how many encoder cycles occur in between each compass refresh. Since the encoder is not yet hooked up, I am simulating this with timer overflow interrupts, instead of external interrupts triggered by the encoder.

Problems:

I have run into the problem however that by adding these interrupts, the compass readings are sometimes garbage. This is due to the fact that the communication between compass and PIC is time sensitive, and the interrupts are throwing off the timing. I will need to find ways to make the compass code more resistant to the timing errors. The problem is worse in the test code than it will be in the final code because the interrupt is being called more often. Currently it is called 152 times per second. With the encoder at 32 cycles / revolution and a roughly 3 inch diameter wheel, assuming a maximum speed of 10 miles/hour the rate will be:

10 mph * 5280 feet/mile * 12 in/foot * 1/3600 hrs/sec = 176 in/sec

176 in/sec / (2 * 3 * 3.14 in) = 9 cycles/sec

This interrupt rate should affect the compass communication far less, but must still be considered.

Next Week:

In the next few week I will mount the tracking wheel and encoder onto the robot. I will finish the code for the PIC, and then use the information the PIC sends to the main machine to calculate the robot's position.

Project Whitney
Alan Ghelberg
Week 10

This Week:

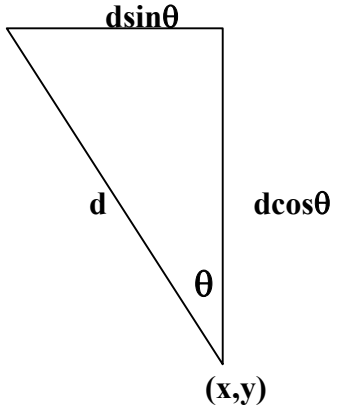
This week I progressed further in implementing the odometry and compass-based navigation. I implemented the odometer algorithm on the PIC such that it can tell forward motion from reverse. The way this works is that the optical encoder has two outputs, A and B, which are separated by a 90 degree phase shift. Therefore, by calling an interrupt on A, and checking B in the interrupt, one can tell forward motion from reverse.

After running into some trouble concerning the external interrupt pins on the PIC, I was able to code a program that sends the main computer the odometer and compass data via RS232.

I also worked with Andy N. on the additional motion tracking wheel for the vehicle. It is mounted on a spring so that it maintains constant contact with the ground and resists slippage.

Next Week:

Now, with the PIC sending correct data to the main computer, I want to write a navigation program on the EBX, that will actually track this position. It will convert the # of cycles from the encoder into distance. Then, given an initial position x,y over a certain time the compass records an angle of θ, and the wheel travels a distance d.



Therfore the final position over one period is:

- $x = x + d\sin\theta$
- $y = y + d\cos\theta$

By repeating this algorithm over relatively small time slices, the system should be able to approximate position.

Project Whitney
Alan Ghelberg
Week 11

This Week:

This week Mike and I were able to get the compass/odometer PIC to interface with the EBX computer, and have the readings display on the LCD screen. From there, we attempted to have the robot turn to a certain direction. Un fortunately, we were not able to get the motor controllers to work properly and as such weren't able to get it moving straight.

Almost completed wheel mount for odometer.

Getting the compass and odometry readings onto the EBX brings me one step closer to calculating position, however certain problems remain.

Problems:

Difficulties getting serial input working properly on EBX.

- Needed to set up asynchronous communication and can't lose any data packets.

Problems with interrupts on PIC chip.

Currently the odometry system works by calling interrupts on the optical encoder output. This may call several hundred interrupts per second. Although each interrupt only last about 6 microseconds, they caused severe problems on the RS232 connection. I was able to fix this by disabling the interrupts during the communication (which lasts about 7 ms of the 200 ms total cycle). This brings about the problem of missed "clicks." This should not be too severe since the interrupts will only be disabled 7 ms/ 200 ms = 3.5% of the time. This may be helped by some extrapolation to fill the time in.

Beyond messing up serial communication, the interrupt is also hurting the communication with the compass, which is very time sensitive. This is odd considering how short each interrupt really is, but there are certain compass readings which are huge outliers.

Possible solutions:

- Throw away compass readings deemed to be garbage
- Try to disable some interrupts during compass SPI communication.

Problems with compass

It has become apparent that the vector 2x compass has some problems that will be difficult to work around. As previously stated, the lack of robustness in communication protocol makes it difficult to use interrupts. Also the compass is relatively inaccurate, a problem which is exacerbated by the presence of external magnetic fields. A trial test found a difference in reading of over 15 degrees while pointed in the same direction within the same room. There are also problems with tilting and slow refresh rate. These limitations may make it difficult to obtain an accurate position reading over an extended period of time.

Next Week:

- Mount wheel
- Finish position sensing algorithm
- Finish turn to specified heading program