# Vehicle Data Logging Device

a sponsored project.

Design Document

faculty sponsored by: Dr Yiannis Papelis. Visiting Professor. UCF. CECS.

EEL4914 • Senior Design I • Group 17
Kyle Fiducia • Joshua Lance Mahaz • Graham Smith

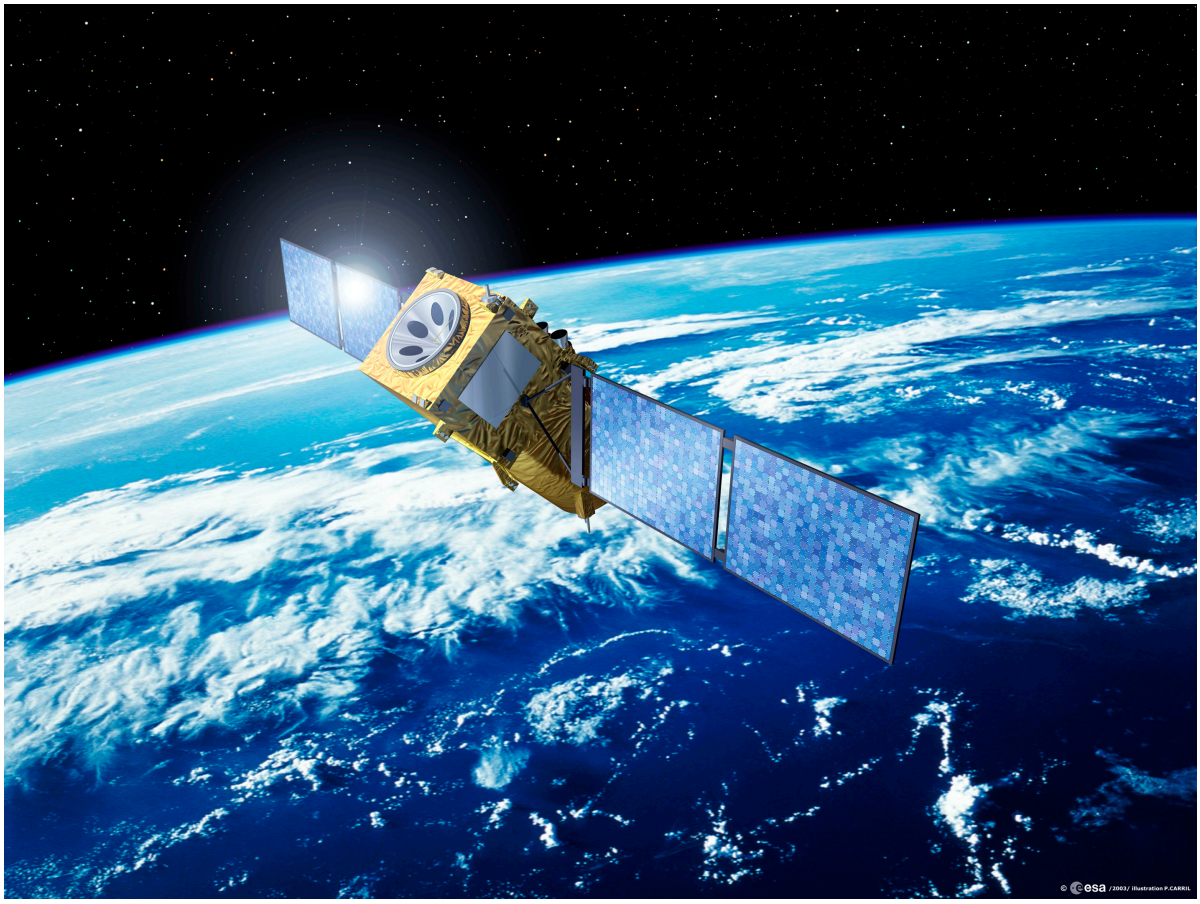



Photo:ESA-P.Carril

4 December 2006

University of Central Florida • College of Engineering and Computer Science

# 1. Project Definition and Specifications

## 1.1. Executive Summary

The Vehicle Data Logging device is a senior design project sponsored by Dr. Yiannis Papelis from the University of Central Florida's College of Electrical and Computer Science. As outlined by Dr. Papelis, the project's intention is to build a compact hardware device that logs data acquired by various sensors on a passenger vehicle. Vehicle data that is to be logged includes: geographic location, engine RPM, throttle position, accelerations, speed, yaw rate, and forward looking video. The device must include some method of configuration as well as an easy way to retrieve the data. The device must meet budget restrictions to allow for widespread data gathering usage. Finally, the device must be portable and easy to install.

## 1.2. Motivation

The motivation for Dr. Papelis' sponsoring of a senior design group was to provide real world data for input into his research. His research is motivated by the 45,000 automobile accident deaths each year, of which 90% involve driver error. The device he is requesting for us to design will log extensive amounts of data for studying driver behavior and developing metrics for driver performance. The sponsored project was attractive to us for a multitude of reasons. We knew we would be building a solid-state device that uses common standard components in use today. The experience working with standard hardware will be much more versatile than learning a highly specialized task to complete a project. Although each group member has different reasons for working on the project, we agreed that using microcontrollers, GPS hardware, removable media, and other input devices would be interesting for each of us. On top of having most of the guidelines set for us, we also had a sponsor who was intelligent and familiar with most of what we would be doing. It is also encouraging that if the design is good enough, the device could be actually used after the project is done, rather than just a proof of concept.

## 1.3. Objectives

The device is rather simple, and certainly not novel. There are many logging devices on the market currently. Our logging device should be able to log GPS data, OBDII data, video, and other optional inputs in attempts to gain as much information as possible about the driving conditions in the car in which the device is installed. We are also to minimize the cost in order to facilitate large numbers of these devices so more data can be acquired from more drivers. The device must also be small, portable, and easy to install. Of the many devices on the market that log OBDII data and GPS data, none of the devices log all of the required data all at once, and most of the devices that are close are prohibitively expensive. The project budget for the device is $400.

## 1.4. Requirements

The device must fulfill the OBDII, GPS and accelerometer data logging in a simple and inexpensive design and allow for convenient retrieval of the logged data. The final result must not be a bunch of loosely arranged parts but a reliable, rugged, custom printed circuit board. Again, total unit cost must not exceed $400.

## 1.5. Goals

The project must meet all of the requirements and still have room for expansion. The device must work well, which means it must have excellent error handling and fail-safes. The device must be well constructed, solid, rugged and small. We also aspire to keep the device's cost to half of the estimated budget of $400. In addition to all of the requirements, Dr. Papelis has also included optional sensors that would add to the cost and complexity of the project. It is our goal to include all of those optional sensors. We realize some of the sensors may cost as much as our entire budget. For those sensors we aim to allow for an expansion port so the sensor may be added in future just by plugging it in. At that point no further PCB modifications will be necessary, maybe just small changes in the firmware on the microcontroller. Dr. Papelis also suggested we have some sort of configuration method. We do not think a web server user interface is a feasible option since this device will rarely have network connectivity. We do intend to have a robust configuration ability as well as error handling for an un-configured device. We also hope to add many error-handling capabilities such as video overwriting when the device begins to run out of storage media and a user interface to show system status.

### *1.4.1. Milestones*

We plan to develop the device in a series of milestones. The first milestone was to acquire the majority of the components, dated mid-November. We were actually on time with that milestone if not a bit ahead. We have acquired all of the components and have begun testing them for unforeseen complications. The next milestone is a working prototype. We have set a date of mid-December to have a working prototype. The prototype must have all the devices hooked up simultaneously and working properly. This prototype must log data and perform all of its other required tasks successfully, however the software does not have to be complete or perfect yet. Once we have a working prototype we can test for problems with the components or circuit. After the working prototype is approved, we can begin development on the custom printed circuit board (PCB). Since the final version of the device is going to be fabricated onto a single PCB we will have to make sure the design is finished and working well ahead of the deadline in case there are problems. We expect to be contracting the PCB manufacturing in late January of 2007. While waiting for the hardware to be completed the software designer(s) will continue to work on making the software more robust. Once the hardware team finishes the final version of the PCB, the software team should be finalizing the device's firmware. We expect this date to be around April of 2007.

## 1.6. Specifications

Below you will find a chart of required data and frequency of capture.

Figure 1.5.1. - Project Specifications

| Data | Guidelines |
|---|---|
| Vehicle Position | (> 1 Hz) |
| Vehicle Velocity | (> 1 Hz) |
| Throttle Position | (> 1 Hz) |
| Engine RPM | (> 1 Hz) |
| Lateral Acceleration | (> 10 Hz) |

| | |
|---|---|
| Longitudinal Acceleration | (> 10 Hz) |
| Forward Looking Video Images | (> 5 Hz) |
| Yaw rate (optional) | (> 5 Hz) |
| Following distance (optional) | (> 1 Hz) |
| User level flags (optional) | undefined |
| Cost Total | $400 |

Project Specification Chart

These specifications have been given as guidelines only. The success of the project is not based on any one of these specifications. Based on research done to this point we do not see the need for 5Hz video especially when coupled with the storage requirement and negative effect on longevity of the device. The device could log data for days more with a lower video frame rate. If video is needed at such a high frame rate we may step down the resolution to accommodate the increase in video capture frequency.

## 1.7. Budget

The desired cost was set in the specifications of the project to be about $400. While deciding on hardware we actually saw many ways to cut the costs of the final device substantially, but we also saw increases in the amount of work needed to build the device. Based on discussions with Dr. Papelis, we decided to forgo the costs savings in favor of expediting the device's production. The project is to be paid for entirely by Dr. Papelis leaving the students without a financial obligation to the project. However, due to delays in ordering and receiving components, we have purchased some of the components ourselves to save time and frustration. Dr. Papelis already had some of the components that we needed, some we ordered from various Internet stores, and others we found at local stores and from personal supplies. For example, for testing, we used our own OBDII cable to interface with the car, and used our own solder, LEDs and capacitors. We also purchased many tools and hardware that will never be used in the production unit, used solely for development, testing, and debugging. Waiting to be provided with these components would offer us substantial time delays, with no benefits. Referring to the following chart you can see we are significantly over budget, however, the cost of the device should not include the SD module, the yaw rate gyro is an optional component that does not have to be included. As we finalize the design we will need fewer miscellaneous components and be able to save money buying in bulk.

Figure 1.6.1. - Budget Diagram

| Part | Actual Cost | Our Cost | # Purchased | Total Spent |
|---|---|---|---|---|
| GPS Device | $80.00 | $0.00 | 2.0 | $0.00 |
| CMOS Image Sensor | $40.00 | $40.00 | 2.0 | $80.00 |
| 2-Axis Accelerometer | $40.00 | $40.00 | 2.0 | $80.00 |
| Yaw Rate Gyro | $50.00 | $50.00 | 2.0 | $100.00 |
| ELM327 | $35.00 | $0.00 | 0.0 | $0.00 |
| DOSonChip | $40.00 | $40.00 | 2.0 | $80.00 |

| | | | | |
|---|---|---|---|---|
| UI LEDs (4) | $10.00 | $10.00 | 0.0 | $0.00 |
| 2GB SD Card | $43.00 | $43.00 | 1.0 | $43.00 |
| Rabbit 3220 Core | $80.00 | $80.00 | 1.0 | $80.00 |
| Misc Components | ?? | $251.60 | 1.0 | $251.60 |
| Enclosure | $10.00 | $10.00 | 0.0 | $0.00 |
| Mounting Method | $5.00 | $5.00 | 0.0 | $0.00 |
| Parts Totals | $433.00 | $569.60 | 13.0 | $714.60 |

* Items highlighted are estimations, as we have not yet completed development.

Budget Diagram

Leaving out the optional yaw rate gyro, and the SD card, which isn't part of the device, brings the parts costs to about $340. That is still before fabrication, and not including the miscellaneous components that will be needed for final fabrication. Leaving us enough to pay for fabrication

Figure 1.6.2. - Device Reproduction Costs

| Number of Boards | 2 | 4 | 6 | 8 | 10 | 15 |
|---|---|---|---|---|---|---|
| PCB Fabrication | $222.00 | $256.00 | $282.00 | $308.00 | $334.00 | $375.00 |
| Cost/Board | $111.00 | $64.00 | $47.00 | $38.50 | $33.40 | $25.00 |
| Totals | $544.00 | $497.00 | $480.00 | $471.50 | $466.40 | $458.00 |
| w/o opt components | $451.00 | $404.00 | $387.00 | $378.50 | $373.40 | $365.00 |

Device Reproduction Costs

As you can see from the table above, we can get the printed circuit board cost down to less than $25, without components, if we want to make at least fifteen boards. If we factor in cost savings at this number, we should meet the budget requirements if we do not include optional components. It may also be beneficial to rebuild the circuit and software to utilize a less expensive microcontroller. We will explore other microcontrollers in the next section. We still have not considered the additional components costs such as capacitors, resistors, level shifters and interfaces, but do not expect the costs involved in such components to be significant enough to list. A complete parts list would span quite a number of pages and will need further consideration.

## 1.8. Task Breakdown

Since this is a group project, each member will have to contribute something to the project. Based on each group members' interests and desires, we have broken up the tasks accordingly. Josh Mahaz our Computer Engineer will be handling most of the software design; Graham Smith one of our Electrical Engineers will be handling the power system and hardware communications; Kyle Fiducia will be handling most of the hardware design and serve as group nagging-pain-in-the-rear. The flowchart below breaks down not only the project assignments, but also gives an idea of the order they will be accomplished.

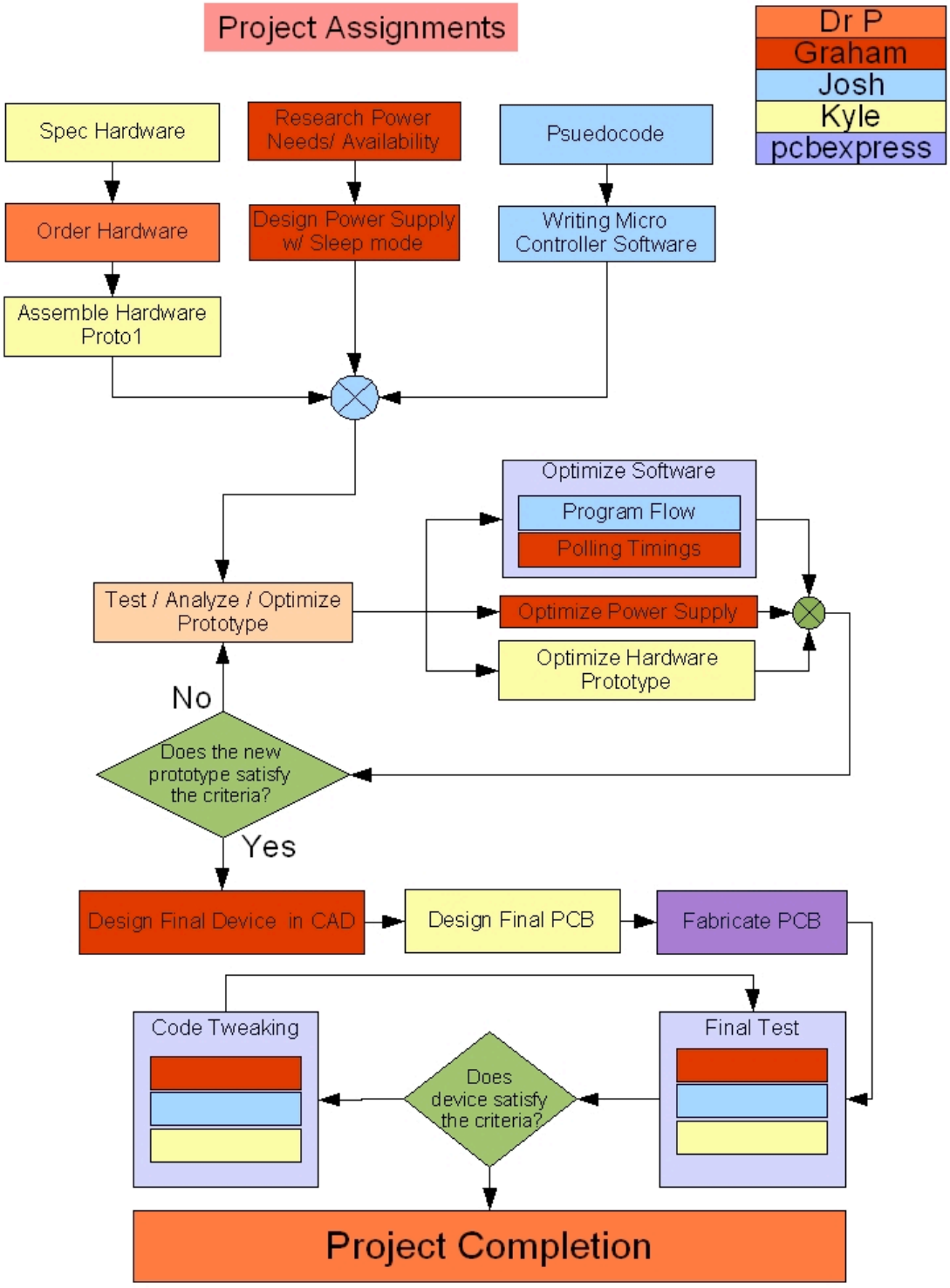



Figure 1.7.1. - Project Assignments Flowchart

# 2. Hardware

## 2.1. Main CPU (Microcontroller)

Our project requires gathering data from multiple different sources. These sources all have their own communications protocols, handshakes and voltage levels. The data sources will also need to be samples at varying rate. All of these data samples must be stored for retrieval in order to be useful. To collect and store all this data we need something to handle all these operations. The most logical choice is a microcontroller. The microcontroller is the most important piece of hardware in the device's design. This is what will make this device useful, rather than just a collection of hardware; therefore it must be carefully considered.

Factors to consider when deciding on a microcontroller:

- Speed
- Input/Output Lines
- Communications Protocols
- Power Consumption
- Size
- Voltage Tolerances
- Cost

### *2.1.1. Speed*

Speed for microcontrollers is rated in numbers much smaller than personal computers. Somewhere in the range of 20-50 MHz is what will be looking for. The speed is a limiting factor because we need to make sure that we can gather and write all of the required data at the specified polling rates. As we recall from the project description we have many different data sources each with their own framing and handshaking. So we need to make sure all of the data capturing can go on, including enough time for framing and writing the data within the specified polling interval. For instance, at 1Hz we need to make sure we can capture all the data, format it for the data storage and then tell our storage hardware to write the data. We must also allow time for encapsulation into packets and transmission as all of our data has specified maximum baud rates and required framing. Our major limiting data source is the camera. Although we are capturing sub-VGA we will want to allow for future high resolutions and possibly higher frequency video capture than the specified 1Hz. Here is a breakdown to give you an idea of how the processor speeds come into play.

The video is captured at 320x240, which is a low resolution. Even at this low resolution we still have 76,800 pixels. If this image were a 16-bit color image (approx 65k colors) that would equate to 1,228,800 bits. Luckily our built-in JPEG encoder will reduce this 150KB image down to around 10KB before delivering to our microcontroller. Even with such excellent compression we still have 10KB to worry about. However, we have been asked to allow for higher resolution capture in future, so we are going to make sure we can capture at full VGA (640x480). After JPEG compression, which is the most efficient way to extract the data, we still can have about 30KB of data. So we need to have a minimum data rate of 30 KBps which is 240 Kbps. This however is a bare minimum, as we still need to allow for framing and procedural clock cycles. We estimate about 10% overhead to allow for framing. With the slowest bit rate this would take about .07 seconds. As you can see we are well within our data transmission constraints even

considering this is asynchronous only and we will still need to transmit to the storage media.

### *2.1.2. Input & Output Data Lines*

The next factor is the number of data lines. We will need to allow for quite a number of IO (Input/Output) devices each with a different number of IO pin requirements. As we can see we will need a minimum of 32 IO Pins. Unfortunately no matter how many times you go over the spec sheets, there will always be some surprises so even though we were conservative in our estimates, we may want to opt for even more IO lines than we think we need, or more microcontrollers.

Figure 2.1.1. - IO Lines Chart

| Device | Communication | Data Pins | Other | Total Req |
|---|---|---|---|---|
| GPS Device | Serial Data | 2 | 0 | 2 |
| CMOS Image Sensor | $I^2C$ Data | 2 | 2 (Power Seq.) | 2 |
| 2-Axis Accelerometer | PWM | 2 | 0 | 2 |
| Yaw Rate Gyro | SPI/Parallel | 3 | 1 | 4 |
| ELM327 | Serial | 2 | 2 (Busy/RTS) | 4 |
| Diagnostics Port | Serial | 2 | 0 | 2 |
| Future Expansion | Serial | 4 | 2 | 6 |
| Input Total | | | | 22 |
| DOSonChip | UART/SPI | 6 | 0 | 6 |
| UI LEDs (4) | PWM | 0 | 4 | 4 |
| Output Total | | | | 10 |
| Input Total | | | | 32 |

IO Lines Chart

### *2.1.3. Interface Support (Communications Protocols)*

Even if we have plenty of IO lines to interface with our devices we have to be sure these IO lines can communicate with each device, as you can see from Figure zeJK3im4 each device has its own communication method and there are many different methods used. Since it is easier to find microcontrollers that support more data communication protocols than it is to find hardware devices that have all the same features with a different protocol we will opt to find match the microcontroller with the devices rather than the devices with the microcontroller. This also opens up a wider selection for future expandability. You can refer to the communication protocol section to find out more about each protocol. So far we find that we have selected excellent choices for microcontrollers as they all support all of our devices.

### *2.1.4. Power Consumption*

The next factor is the power consumption. The microcontroller power consumption is not really a significant factor in our design while the car is running, as we will be pulling power for the device from the car battery. While the car is running the alternator will be generating plenty of power to run all of our microcontrollers and devices. The device will pull its power from the OBDII port, which is specified to be switch with the ignition. Since not all manufacturers comply with the standard, not all ports have switched power, and will require our device to sense the car's ignition state, switch off the peripherals, and go into low power mode. The device will need to continually poll the car's ignition state for instructions to turn back on. We don't want a device to know when to turn itself off but never turn itself back on. If the device does not have a low power mode it must run at a low enough wattage to not significantly drain the car battery. We do not want the user to come back to a dead battery.

### *2.1.5. Physical Size*

Physical size constrains were not given in the project specifications. It was however one of our goals to make the device as compact as possible. Since smaller parts are available at no additional costs we are going to use the smaller parts and make the device as compact as possible.

### *2.1.6. Voltage Tolerances*

Voltage tolerances are actually quite important. Each peripheral device seems to communicate at a different voltage and interoperability without extra hardware makes the design much easier. Adjusting voltage levels to power the devices is quite easy; adjusting the voltage levels of the communications protocols requires more complicated hardware. Since the RS232 port has never gained a standardized voltage specification each device uses its own voltage, which is rarely the same as the device with which you are trying to interface. Unfortunately all of the devices are similar in that they are tolerant of 5V on their inputs, so none of the candidate microcontrollers gain an advantage in this respect.

### *2.1.7. Cost*

Cost is generally a significant factor when designing a device. In fact we initially looked for the lowest cost hardware possible, which is why we were looking at the PIC microcontrollers. Dr. Papelis urged us to use the hardware he already had and was familiar with, which was a Rabbit 3365. We later decided that this hardware, although seemed quite useful, as it had build in removable media, was also very limited in its use. Some of it's IO pins were in use by the Ethernet port which we would have no use for, and it's built-in XD media card, although fast and low power, was limited to 128MB. We had already found solutions capable of addressing 2GB which seemed much more appropriate for a data logger with video capture. The Rabbit 3365 also carries a tag of about $90 verses the PIC of about $9. However, once we factor in the development environment and development board, the costs quickly level out considering Dr. Papelis already owns a Rabbit development kit. The main reason we decided on the Rabbit was the ease of use and availability, if we were to design this for mass production, the $80 Rabbit 3220 would be far too expensive, we would again look at the PIC microcontrollers that would significantly reduce the costs.

### *2.1.8 Software Environment*

Each microcontroller seems to each have its own programming environment. The Rabbit's Dynamic C environment seemed quite easy to use. Dr. Papelis, our sponsor, also has experience using this software as well as programming the Rabbit micro controllers. The PIC environment was quite complicated and had a much steeper learning curve. It is possible to compile the C language into a compatible assembly language for the PIC as well as write in assembly, which is not recommended. However the Rabbit works right out of the box and does not require a separate programmer to load the boot loader, the basic operating system of the PIC.

### *2.1.9. Microcontroller Decision*

As of this writing we have decided on the Rabbit 3220 microcontroller for use in our project. It has plenty of IO pins, supports all of the necessary protocols, has sample code for those protocols, is easy to get started with, and we already have the development board.

## 2.2. Yaw Rate Gyro

The yaw rate gyro was requested provided the costs remained under budget. Since we were able to find some great parts whose total cost was well under the budget limitation, we decided to go ahead and spec the device to include an angular rate sensor. With this device we opted for a digital output to again reduce the calculations and hardware needed. With an analog sensor we would have needed an analog to digital converter either built in to the microcontroller or added to the circuit. Delegating as many processes to integrated circuits keep our jobs easier and our circuit less complicated meaning less room for error. The costs are marginally if at all higher for these slightly more complex chips.

This device is probably the most advanced piece of hardware we will use on this project. The yaw rate sensor is actually a microelectromechanical (MEMS) gyroscope, and works similarly to other gyroscopes. You may think gyroscopes deal with only rotating masses, but there are many kinds of gyroscopes. All the different types are beyond the scope of this project so we will focus on the vibrating structure gyroscope, which is what is used in the MEMS device. As you know from physics rotating masses tend to rotate in the same plane, simple kids' toys such as the top demonstrate this principal. With significant enough angular momentum the top continues to rotate about its axis, as it gradually slows down it loses its angular momentum and begins to topple.

A top is a simple gyroscope.

Photo: Juni

The vibrating structure gyroscope works similarly but instead of a rotating mass there is a vibrating mass. Just as a rotating mass has angular momentum, vibrating structures tend to continue vibrating in the same plane even when the support is rotated. The momentum of the vibrating structure exerts a force on the support as it rotates this force can correlated to the amount of rotation based on the mass and vibration frequency. The resulting information about the change in rotation gives you a rather simple attitude indicator for planes, which is critical for flight. In fact insects use a very similar method to maintain stable flight with their halteres. The halteres look like two small antennae

with knobs on the ends, as the insect flies the halteres are flapped rapidly. The insect has sensors at the base of each halter to sense a change in attitude, which can be quickly corrected by the rest of the flight equipment, mainly the wings. The devices are rated in degrees per second. We expect the yaw rate gyro with more like 80 deg/sec to be better suited to this application, however without knowing what to expect or finding sample data, we will opt for the most versatile sensor which we can configure later.

### *2.2.1. Cost*

The cost of this device was concerning, we were given a budget of only $400 and to use 12.5% of that on a single sensor that isn't all that important seems like a waste. Luckily the final design will be rather modular, and we can leave the availability there but not put the device on the board at very little extra cost. If the design ends up being produced in larger quantities and the yaw rate is not found to be very useful, it will be easy to make the boards with or without the sensor and save $50.

### *2.2.2. Interface*

As with our other devices, of main concern to us was the interface to the microcontroller. We wanted to maintain digital outputs to the microcontroller so the microcontroller had less processor overhead in simple conversions. The base model digital output yaw rate gyro from Analog Devices is the ADIS16100. The 16100 has SPI output rather than analog output, it also has half of the number of pins of the analog ADXRS401 which is packages in a BGA32 footprint. The ADIS16100 is about $15 more than the analog counterpart which makes it not only expensive but it is also complicated. From start to finish this device does not make for an easy addition. The SPI protocol is the most complex to work with for reasons discussed in the interface section. On top of the protocol issues, it is also extremely difficult to work with because of its packaging. The ADIS16100 comes in an LGA16 package. Which makes this 8mm by 8mm device with 16 pins, quite difficult to hand solder for prototyping. As of this writing no evaluation board was available from Analog Devices.

## 2.3 Accelerometer

When deciding on an accelerometer our main deciding factor was again on digital output. As previously discussed we would prefer the device to do the analog to digital conversion and use the pulse width detection of our microcontroller. Of secondary importance was the range of the device. Car magazines such as Car & Driver do certain tests on skid pads. A skid pad is a large area of smooth pavement used for various tests. The test important for us was the lateral acceleration test. For the lateral acceleration test the car is driven in a circle about 300 feet in diameter and driven faster and faster until it begins to slid. This measures the car's road-holding ability. Even high performance cars do not score higher than 1.2 Gs, where a G is equivalent to 9.8 meters/second. Based on our knowledge of the cars in these tests and the cars this device is aimed for, we did not this it was necessary for the accelerometer to measure any more than 1.2g. Therefore we decided on a +/-1.2g accelerometer so the full resolution would be used and a higher accuracy could be gained. To give you an idea of the lateral acceleration values for some cars, here is a chart compiled from tests done with a commercially available acceleration meter of in a controlled environment with friends' cars.

| Figure 2.3.1. - Skidpad Tests | |
|---|---|
| Car | Lateral acceleration (g) |
| Toyota Supra | 1.07 |
| Acura Type R | 1.05 |
| Toyota Honda S2000 | 0.96 |
| Nissan 300ZX | 0.91 |
| Mitsubishi Lancer | 0.90 |
| Ford Mustang | 0.87 |
| Hyundai Tiburon | 0.66 |

Skidpad tests for some high performance cars. Results should exceed normal conditions for device use.

The accelerometer is a very simple MEMS device based on the simple relationship force is equal to mass multiplied by acceleration (F=m*a). If every engineer doesn't have that memorized by their first physics class they are headed for trouble. Usually the devices consist of no more than a well-quantified mass or cantilever beam and some sensing circuitry. The amount force the mass exerts divided by the mass will give the device's acceleration.

For our device we are going to two accelerometers to give both forward acceleration as well as lateral acceleration. We have not been specifically briefed in exactly how Dr. Papelis intends to use all of this information, but we can assume based on the acceleration data he can infer how people approach stoplights, turns, and et cetera.

## 2.4 Global Positioning System

The GPS device gives the location of the vehicle on the surface of the earth. This is really the most valuable data to be logged. In addition to providing the most useful information about the vehicle, the GPS unit will also provide a highly accurate clock for use in making sure the data is properly correlated to the time. There is a plethora of GPS units on the market today. There are only a few different GPS engine manufacturers though. The GPS engine is the actual chipset that does the calculations. Of those there are really only four popular engines, the Garmin, SiRF III, Trimble, and Sony engine. Each device supports different position calculation methods, varying by manufacture date and price. The devices also vary in the number of satellites they can track and their acquisition times. For our purposes almost any of these devices will work just fine. The main requirement for us is that whichever device we chose, we need proper documentation in order to use it. Of secondary importance, since most devices come with this, is a housing and mounting method. We would prefer to buy the device already in a waterproof magnetic housing with a built in antenna rather than trying to waterproof and secure our electronic components ourselves.

### *2.4.1. Resolution*

Most all of the devices have similar accuracies of about +/- 5m. Some devices have support for various advanced computation methods such as Wide Area Augmentation Systems (WAAS) and Differential Global Positioning System (DGPS). The device does not require such advanced systems although if they are included at a

comparable price it is preferred. How the more advanced systems work and their effects are beyond the scope of this document, we just need to know they offer improved accuracy.

### *2.4.2. Update Frequency*

Almost all of these devices update at 1Hz. Garmin has a special unit that updates at 5Hz, which is useful, but probably not for our application. We feel such a high update rate will be a waste of space on the storage media and far exceeds the requirements of the specifications. Higher update rates may also push the processing limits of our microcontroller beyond what we have planned for.

### *2.4.3. Antenna Configuration*

Most devices either have a built in patch antenna or a connector for an external antenna. Due to losses faced in the waveguide to the GPS unit from the antenna, the built in patch antennas are probably better for non-amplified applications. Other factors to consider are if you would prefer to have a $10 antenna exposed on the roof of the car or the $80 GPS device with the built-in antenna. From a technical standpoint, there is little difference, just preference.

### *2.4.4. Time to First Fix (TFF)*

Our device must be capable of a fairly quick TTF. Without reference data such as time and relative position it can be quite difficult for a GPS receiver to calculate where it is or which satellites to expect. If the GPS device knows approximately where it is it can compare the code it is receiving against the C/A code of the most likely satellites. This small bit of data can shave about 30 seconds off the fix time. Most devices use a small battery such as a watch battery to power the memory in order to retain this information while the device is disconnected.

### *2.4.5. Physical Layout*

Our main concern for the GPS device is that it is either surface mountable on our PCB with an external antenna or it comes in a waterproof magnetic housing suitable for mounting on the roof of a car. This is a concern because the roof of the car and other objects obstructs GPS signals. Operating the GPS unit with signal blockage can significantly lengthen the TFF. Even glass can block the signals of satellites and reduce the reliability and accuracy of the data being logged.

### *2.4.6. Cost*

GPS units are usually in the range of $50-90 depending on the packaging. Surface mountable GPS engines are at the low end and devices with a housing and mounting at the upper end. Depending on how the devices are to be used, it may end up being more cost effective to permanently mount the GPS antenna in the vehicles, and to just remove the main box containing all the expensive hardware. This will allow for a quick installation of the device without dangling wires as well as keep the costs low.

## 2.5. Video Capture

One of the specifications was for video capture. Due to the popularity and abundance of cameras built in to cellular phones, CMOS image sensors have become inexpensive and high quality. After having looked at quite a number of sensors, we decided on a TransChip 5747.

### *2.5.1. Resolution*

The camera resolutions are usually rated in mega-pixels (MP), but for our application we are not concerned with mega-pixels, we are more interested in the smaller sizes. We would prefer VGA (640x480 pixels) or lower resolution. The reason for this is based solely on the available memory space. Rough estimations of storage capacities suggest we can store about 2275 images at 2MP on a 2GB storage card. That may seem like a lot of pictures, and it is if you are taking pictures of friends and family, but when you think of image capture at 1Hz, that wouldn't even last an hour. At lower resolutions like VGA, which to give you a relative idea is 0.3MP, you can get more like 68,000 images, and at sub-VGA you can get more like 2 million images. For our application we would rather have more images per pixel rather than more pixels per image. At these sizes at 1Hz we can go for more than 2 days straight. Usually images are quite recognizable even at low resolutions. To give you another relative reference DVD quality images are only 720x480 pixels, and most people will agree the images on DVDs are far better than standard television, which is still quite recognizable.

### *2.5.2. Frame Rate*

Based on the possible need for higher frame rates the image capture device needs to have the ability for much higher frame rates. Dr. Papelis said he might want up to and beyond 5Hz for video depending on the speed and location. Keeping in mind our TC5747, at the maximum frame rate, 40fps, we expect the microcontroller to run out of clock cycles to process the capture and storage long before the camera's reaches its maximum frame rate.

### *2.5.3. Configurability*

In order to control the output resolution the camera will need to have some sort of configuration ability. Configuration by UART or $I^2C$ is quite common; we need to be able to down-sample the images and set the output format. The TC5747's resolution, sampling, output, and rotation can be configured by UART or $I^2C$. This is quite handy, as you will see when we get to interface.

### *2.5.4. Focal Length*

The focal length is actually a property of the lens, but most image sensors come with the lens since they are rarely interchangeable or standard. Together, the focal length and sensor size make up the field of view with the relationship: $2 \times \tan^{-1}( L / (2 \times F) )$ where L is the length of the frame and F is the focal length of the lens. We would like a focal length that allows for a wide field of view (FOV) like 160º. A larger FOV would allow for the camera to not only capture what was going on in front of the car, but also inside the vehicle. This would allow for logging not only road conditions, but also the

conditions influencing the driver. The images would be able to log most distractions such as cell phone use, drinking coffee, putting on makeup, etc.

### *2.5.5. Aperture & Focal Range*

The focal range measures the distances within which the camera can focus, for the camera we chose, which is fairly standard, the focal range is 40cm to infinity, meaning this is ideal to use for our applications as long as we don't want to see anything that is closer to the lens than 40cm (15.75 inches). The aperture is also an important feature of the lens. The aperture is measured in F-stops and is a measure of the amount of light reaching the sensor. Lowering the F-stop will increase the aperture size and increase the amount of light hitting the sensor assuming the same shutter speed. This will also decrease our depth of field. For our purposes we would prefer a larger depth of field, meaning more of the image would be in focus. Otherwise just what the camera is focused on and few centimeters closer and father would be in focus. A shallow depth of field would leave the driver completely out of focus if the camera were focusing on the car in front; obviously not optimal for our applications. Also to consider is that if we opt for too high of an F-stop, there will be much less light entering the lens at any given point in time. Less light coming through the lens means we would have to leave the shutter open longer for a properly exposed image. Having to leave the shutter open longer for images taken on the highway would lead to blurry images, probably too blurry to be useful. We think F2.4 will be an acceptable balance to maintain proper depth of field and shutter speed and still maintain a properly exposed image.

### *2.5.6. Image Sensor Type*

Image sensors come in basically two flavors; the CCD and the CMOS. CCD is an acronym for charge coupled device, and CMOS for complementary meta-oxide semiconductor. Those both don't mean much really even what you know what that means, but without going into specifics, as that is far beyond the scope of this document, they are two different imaging technologies. Each technology has its individual benefits; the CCD has superior image quality, particularly in low light applications and the CMOS sensors require much less space and power. Although we would like to have the camera work well in low light conditions, space and power are certainly factors, and image quality certainly isn't. We have no reason to pursue the CCD technology for our purposes.

### *2.5.7. Compression*

Compression ended up being one of the largest deciding factors concerning the image capture device. Very similar to delegating the analog to digital conversions on the sensors, having the functionality built in to the camera frees the microcontroller of the extra compression load. The compression load would probably take too many processor cycles forcing us to add additional circuitry adding to the complexity of the circuit. Although just storing raw data for off-device processing is an option, the compression allows for smaller images to be stored and extends the longevity of our storage media. The compression technology, JPEG, is familiar to most people and can save a considerable amount of space, particularly when considering the sheer number of images the device is intended to capture. Since we are trying to keep the circuit as simple and

useful as possible, the built-in JPEG compression is a major advantage for the image sensor hardware. Such an advantage we are going to consider it a requirement.

### *2.5.8. Interface*

The TC5747 offers just about every possible interface method, which was one of its biggest assets, its flexibility. Because our microcontroller has a limited number of IO pins, that is a general constraint facing all of our devices. The TC5747 camera allows for not only controlling the camera over the $I^2C$ bus but we can also off-load our compressed JPEG data with the same IO pins. Using only two IO pins makes the camera extremely flexible and leaves more IO pins open for future expansion, it also means we have fewer pins to hook up and worry about connecting which decreases the complexity of the circuit yet again.

### *2.5.9. Cost*

Although there are some less expensive cameras on the market, unfortunately the less expansive cameras were produced in bulk for a specific purpose and manufacturer and have little to no documentation, not designed for OEM applications. The cost is about $40 for the TC5747.

## 2.6. OBDII Interface

The OBDII port offers many different bits of information about the vehicle it is on. You can generally retrieve information such as the fuel pressure, airflow at the mass air flow sensor, intake temperature, coolant temperature, throttle position, speed, RPM, fuel level, oxygen sensor data and much more. Not all of this information will be of use to us though. We are mainly concerned with accessing the throttle position, speed, and RPM, if these are supported. One some vehicles tested, throttle position information was not provided, even when we asked nicely. If we have additional time it may be nice to allow for complete access to the OBDII data through the device, and allow for reading the check engine trouble code light.

### *2.6.1. Compatibility*

This OBDII is the second version of a generic term referring to on board automotive diagnostics. All vehicles after 1996 were required to have this technology and the connector must be located within 3 feet of the steering wheel. However standardized this may sound, it really isn't. The only standard aspect of this technology is the connector. Through this connector you can access any one of the five different protocols in use today. Most manufacturers usually stick to just one of the protocols, but we have seen certain models use a different protocol than the manufacturers other cars. Because of all these incompatibilities we need a circuit that intelligently figures out which protocol the car has and uses it. Luckily ELM electronics has turned that complex task into a simple integrated circuit (IC). Below is a chart that shows the different protocols in use by common manufacturers in the Unites States.

Figure 2.6.1. - ELM Chip Compatibility

| Manufacturer | J1850 | J1850 | ISO9141 | ISO15765 |
|---|---|---|---|---|
| Protocol | PWM | VPW | ISO14230 | CAN |

| ELM Model * | (ELM320) | (ELM322) | (ELM323) | (ELM327) |
|---|---|---|---|---|
| Acura | | | X | |
| New Acuras | | | | X |
| Chrysler | | X | X | |
| New Chryslers | | | | X |
| Ford | X | | | |
| New Fords | | | | X |
| General Motors | | X | | |
| New GMs | | | | X |
| Honda | | | X | |
| Saturn | | X | | |
| Subaru | | | X | |
| Suzuki | | | X | |
| Toyota | | | X | |
| New Toyotas | | | | X |
| Volkswagen | | | X | |
| * This is the minimum chip required, the ELM327 covers all protocols. | | | | |

ELM Chip Compatibility

### *2.6.2. Ease of Use*

The ELM327 is the only integrated circuit ELM makes that supports all of the protocols; it scans each method for readable data to determine which protocol the car is using. The ELM327 communicates using the simple AT command set. This is probably the simplest piece of hardware we have to work with, particularly because of the excellent documentation. There are actually other methods of accessing the OBDII data without the fairly expensive ELM chip, but the additional circuitry and time is not worth reinventing the wheel for marginal savings. The ELM327 is an easy answer to interfacing with just about every car in the United States built after 1996.

### *2.6.3. Availability*

As with all of our other component availability is not always a certainty, and costs are always a top priority. In this instance Dr. Papelis already has a number of the fairly expensive ELM327 chips and some of the supporting ICs for the interface circuit. Dr. Papelis is also well versed in how to use the ELM chips if we were to run into trouble. One of the group members is also familiar with the ELM chips and has software and hardware compatible with the ELM chips to interface with the car's OBDII for debugging and testing purposes. For those reasons and the groups' previous knowledge of the popularity of the ELM chip, we have not explored the other interface methods.

## 2.7. Laser Range Finder

As we saw with the CMOS image sensors becoming popular in consumer technology, laser range finders are also becoming less expensive do to integration into consumer electronics. This time, however, it is the golf industry driving the prices down. Laser range finders are used in the golf industry to tell golfers the distance to the pin or sand bunker, etc. As the devices become more popular the prices go down. As of this writing you can find an inexpensive laser range finder for about $130. This was encouraging for us, as following distance, to us, is what would make our device different

from every other GPS logging device on the market. Unfortunately after hours of looking for OEM parts manufacturers we could not come up with such an inexpensive unit. The least expensive we could find was about $600. We had thought about disassembling the inexpensive consumer model for the hardware, but without proper documentation the device would be a project in itself to reverse engineer. What we have decided to do is to allow for the future expansion of our device to accommodate a laser range finder by leaving an expansion port. We expect the devices to become more commercially available for prototyping and hobby projects, and as that happens the prices will fall into the realm of possibility. Our expansion port can then be used to expand the functionality of the device. We will give a basic outline of the hardware to be used with our device.

### *2.7.1. Range*

Since the range finder will need to track following distances of the cars in front, and possibly behind, it will first need to be mounted in a place free of obstructions. We expect somewhere on the roof, trunk, hood, or in the grill of the car. Obviously this makes for a more complicated mounting method. To add to the complexity, the device will also have to be level so it points into the cars, not the ground or sky. We expect the range required to be at a minimum 1m (meters), and maximum of about 100m. This should allow for accurate following distances from the highway, to how close drivers pull up to the person in front at the stop lights.

### *2.7.2. Availability*

As previously discussed the availability is quite difficult to predict, the devices are currently available at higher costs than our entire project budget. We expect prices to drop rapidly in future.

### *2.7.3. Accuracy*

The precise accuracy is not critical, but would be nice, as higher accuracies would allow us to also calculate other drivers' speeds and at what speed differentials drivers decide to pass each other. The accuracy is also important to reduce noise. A more accurate device we would expect to have less noise, and of the noise it does not filter, we should be able to filter the rest with software. We should explore noise more, as it is an important topic for the range finders.

Lets define car A as the car with our device installed in it, and car B as the car in front of car A. As car A and B approach a red light the following distance gradually decreases. When the light turns green and both cars pull away the distance between them increases, then when car B decides to turn, the distance decreases and suddenly drastically increases to the maximum range, as there is no one now in front of car A. That is one aspect of noise. Another aspect of noise occurs when car A goes around a bend in the road; the laser beam will bounce off of a multitude of items, guard rails, oncoming traffic, construction barricades, etc. This will result in erratic and unreliable results from the range finder. The software will have to interpret this data coupled with accelerometer readings and chose to either ignore the data or not. Unfortunately, without adding significantly to the complexity of the logging device we cannot have a tracking system to point the laser beam to track the vehicle in front of you while you turn or to account for unexpected road conditions, such as hills.

### *2.7.4. Power Consumption*

Depending on the final device that becomes available power consumption may be a factor. We should not overload the voltage regulators or the OBDII port with which we are getting power for the rest of the components. Judging by what current commercially available units use for batteries, we do not expect the power consumption to be a problem, but certainly worth looking into to prevent damage to other components.

### *2.7.5. Cost*

As discussed the current costs prevent this sensor from being practical. As we have explored all of the accompanying issues of this sensor we see it becomes more and more costly to incorporate and get reliable useful data. However, if the costs for the devices were to come down significantly, it would be possible to add this sensor along with its accompanying hardware to provide useful, reliable data.

### *2.7.6. Physical Size*

In addition to the many other problems with this sensor, we also have to not only worry about its physical size, but the physical size of the harware to accompany the range finder if we decide to add hardware tracking to it. We could quite easily add servos to track the car in front, but as we add more hardware the device gets larger and larger. Then we also have to consider where we will be able to mount this that would be stable and secure, the wind on the roof of a car may actually be too much for the mounting method if the device has too much drag.

### *2.7.5. Interface*

As previously discussed we will allow for future expansion from the microcontroller to the rangefinder. So far we have seen most devices communicate with serial UART data, which is simple and easy to work with. However we do realize that this could be as EIA232, RS232, CMOS, or TTL levels, therefore we may want to add a level shifter that has a hardware enable and disable to protect our circuit. Also, we must consider the possibility of wanting to add the vehicle tracking system in future. Although we will probably have plenty of I/O pins to control the system, we do not feel that the microcontroller is powerful enough to do such complicated calculations in addition to all of the other processes it needs to execute per second. Based on this extra load we feel accommodating the possibility is a fruitless effort as the circuit will need to be redesigned for either more processors, or just more powerful processors to handle the extra load.

### *2.7.6. FCC Regulation*

As you may be aware, the FCC has regulations on practically every frequency of the electromagnetic spectrum. Lasers are a particularly stringent topic because of their ability to inflict damage to human vision. Whichever device we decide to use, it must have FCC approval for its use, which will further hinder this sensor’s possibility of seeing the light of day.

## 2.8. Microwave Range Finder

The microwave range finder was explored as an alternative to the laser based range-finding systems. We found they are only more expensive and less likely to ever become widely available and inexpensive. They also have their own set of FCC regulations as well as the possibility of interfering with police radar systems that use the same technology and frequencies.

## 2.9. Power Supply System

The main power required to run the CPU and all of the devices attached to it will be taken from the host car's OBDII port. The OBDII specifications require that pin 16 on the J1962 (OBDII) connecter be battery positive, while pin 4 is chassis ground. Battery voltage to the connector is defined to be switched with the vehicle's ignition, however our research has shown that this can vary among manufacturers, so our device must be tolerant of non-standard vehicles to prevent draining the battery and logging a motionless vehicle. The car's battery, which is a nominal 12V DC, should provide us with about 13.8V DC while running due to the alternator charging. The battery voltage may sag or spike during events such as the car being started or the lights or air conditioning being switched on, as these devices cause a very short but heavy start-up demand from the power system. Our device must be capable of protecting itself from voltage sags, spikes, reverse current, and sudden power off.

The heart of the device's power system will be three LP2960 low dropout linear voltage regulators from National Semiconductor. Linear voltage regulators were chosen over more efficient switching regulators due to their lower noise and better ripple rejection that will prevent voltage fluctuations that are possible in an automotive situation. Many of our devices explicitly specify they require conditioned (clean) power. The unique features that this particular series of regulators incorporate that make it ideal for this application include:

- Fixed outputs of 3.3V and 5V, and adjustable output available
  - Our devices have multiple voltage requirements
- 500mA internally limited current
  - Over-current protection, like a fuse
- Logic level shutdown
  - Intelligent power control
- Status flag pin
  - Communicates status to microcontroller
- Shut down pin
  - Controllable via the microcontroller
- Reverse polarity protection
- 30 V maximum input
  - Plenty of tolerance for voltage spikes
- Narrow SO16 package that is small, but not too small
  - Surface mountable device keeps PCB small
- Automatic thermal limiting
  - Will shut down before heat damages components

A possible option that we are keeping open for development is a battery backup system that would run on two 18650 Li-Ion cells that would charge while the device is powered by the vehicle and provide power once the engine is turned off or the OBDII cable is unplugged in order to let the device finish its operations and then safely power down. This could be easily extended to allow the device to run solely on battery power if necessary, or when using in a covert non-OBDII setup. The following diagram shows a high-level function-block diagram for the complete power system.

Power System Block Diagram

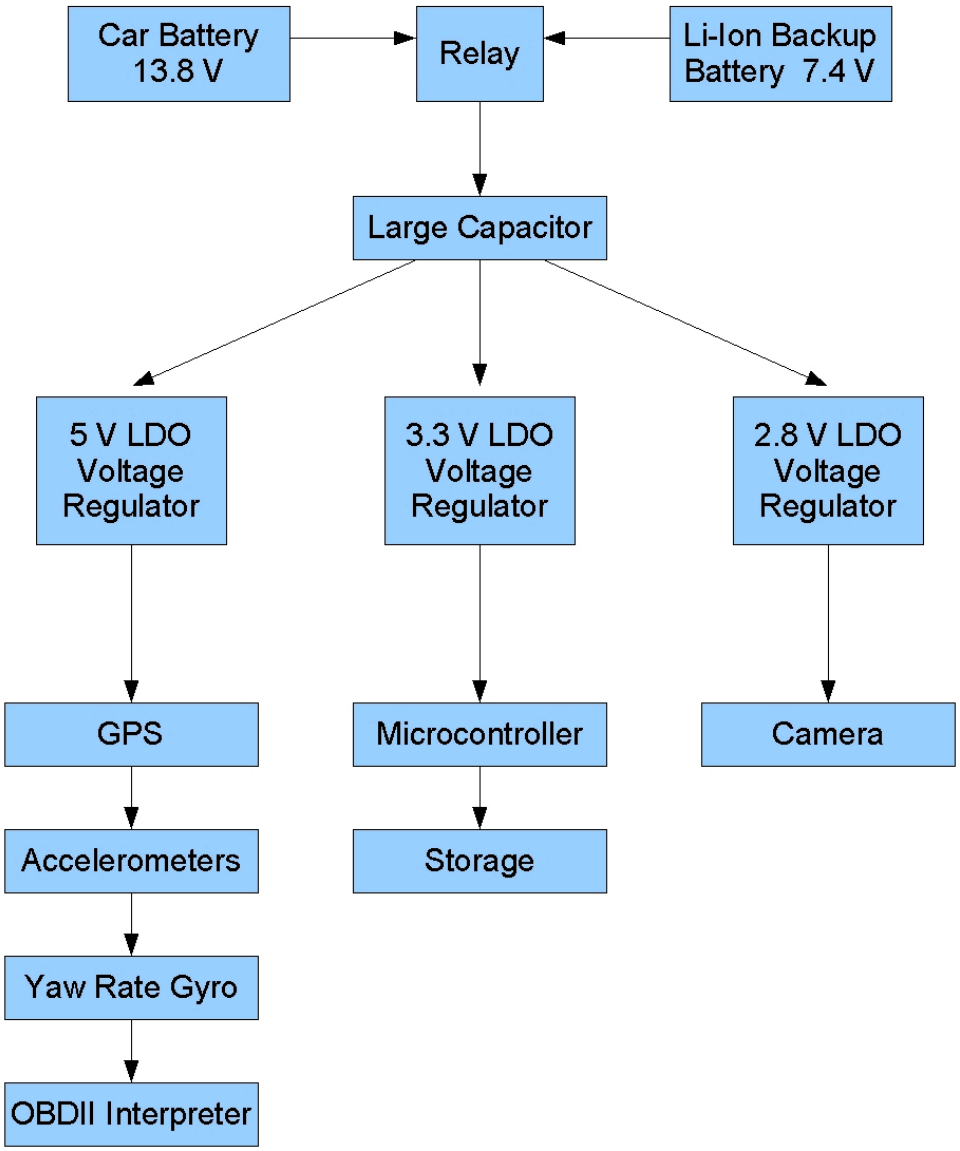



Figure 2.9.1. - Power System Block Diagram

### *2.9.1. Efficiency*

The relatively low currents required and ample power availability while the engine is running decreases the need for the device to be extremely efficient while normally logging data. The combined power for all components running at their typical values only adds up to 1448.5 mW. Power can be found with the equation:

*Power = Voltage x Current*

Figure 2.9.2. - Component Power Requirements

| Part | Voltage (V) | Current (mA) | Power (mW) |
|---|---|---|---|
| Microcontroller | 3.3 | 255.0 | 841.5 |
| Storage | 3.3 | 5.0 | 16.5 |
| Yaw Rate Gyro | 5.0 | 40.0 | 200.0 |
| Accelerometers | 5.0 | 0.7 | 3.5 |
| OBDII Chip | 5.0 | 9.0 | 45.0 |
| GPS | 5.0 | 60.0 | 300.0 |
| Camera | 2.8 | 15.0 | 42.0 |
| Total | 29.4 | 384.7 | 1448.5 |

Component Power Requirements

The 13.8 volts could be stepped down by a switching regulator more efficiently than a linear regulator, however, the additional features provided by the LP2960 more than make up for the decreased efficiency. When running off of Li-Ion power, the efficiency increases as the 7.2 V provided is closer to the 2.8 V, 3.3 V, and 5 V needed by the components. The LP2960 will also provide features that are necessary to enable a low power standby in which efficiency is of the utmost importance.

Since our devices require three different voltages, we must use three separate regulators to supply them. The Rabbit microcontroller and DOSonCHIP storage device both require 3.3 V at a combined current of 260 mA. The yaw rate gyro, accelerometers, OBDII interpreter, and GPS consume 109.7 mA at 5 V. The video camera will be on its own adjustable voltage regulator set at 2.8 V while drawing 15 mA. The dropout voltage is the minimum difference between the input and output voltages to maintain voltage regulation. The dropout voltage for the LP2960 ranges from approximately 0.07 V, for the 2.8V regulator, to approximately 0.3 V for the 3.3V regulator. The dropout voltage, for a 5V regulator, is plotted versus load in the graph below from the National Semiconductor datasheet. The 3.3V and 2.8V regulators have a similar plot.

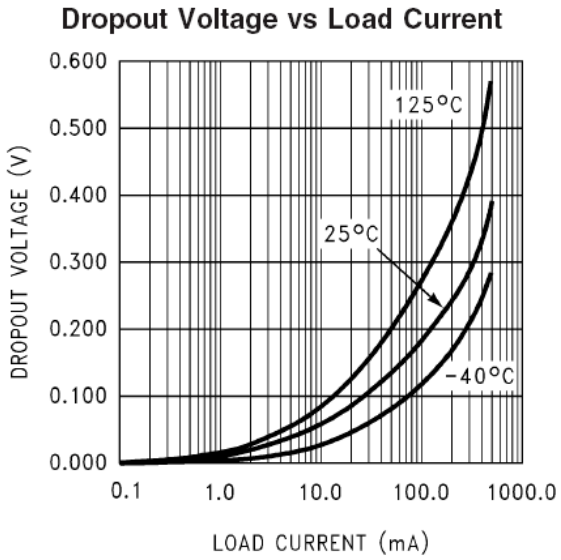



Figure 2.9.3. – Voltage Regulator Dropout Voltage. Reprinted with permission.
-National Semiconductor LP2960 Datasheet

The 5V regulator will be the first regulator to dropout during a sag in voltage due to the vehicles inability to keep the voltage stable. During the worst-case scenario of junction temperature approaching 125 C the dropout voltage for a 109.7 mA load would be around 0.29 V. So the minimum input voltage needed to run the device with all input components would be 5.29V. We do not expect the voltage of any battery in good condition to drop this low even during cranking (car ignition). The minimum input voltage required to run just the microcontroller and DOSonCHIP, at normal clock rates, would be 3.6V. If the voltage ever does drop below the 5.29V, we still have some room until the microcontroller shuts off in order to tell the microcontroller about the dire power situation.

Figure 2.9.4. - Component Power Requirements

| Part | Voltage (V) | Minimum Input Voltage (V) | Drop-Out Voltage |
|---|---|---|---|
| Microcontroller | 3.3 | 3.6 | 0.3 |
| Storage | 3.3 | 3.6 | 0.3 |
| Yaw Rate Gyro | 5.0 | 5.3 | 0.3 |
| Accelerometers | 5.0 | 5.3 | 0.3 |
| OBDII Interpreter | 5.0 | 5.3 | 0.3 |
| GPS | 5.0 | 5.3 | 0.3 |
| Camera | 2.8 | 2.9 | 0.1 |

Component Power Requirements

In order to estimate the overall efficiency of the device we must figure out the ground pin current in the regulators. Since we are not driving the regulators near their maximum current capacity the ground pin currents remain at quite reasonable levels. Estimated from the chart in the datasheet provided by National Semiconductor, we find

that the 5V regulator will have the largest ground pin current at around 7.5mA. The 3.3V regulator will lose about 3mA, and the 2.8V regulator, about 1mA.

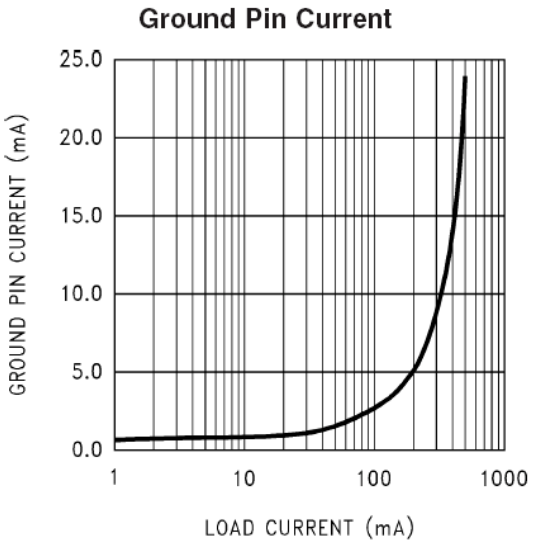



Figure 2.9.5. – Voltage Regulator Ground pin current.
Reprinted with permission - National Semiconductor LP2960 Datasheet

Now that we have the total currents being drawn by each regulator the total efficiency can be obtained by dividing the output power by the input power.

$$\text{Efficiency} := \frac{\text{Output\_Power}}{\text{Input\_Power}}$$

We see that the efficiency of the LDO regulator declines steadily as the output voltage drops farther below the input voltage. The ground pin current from all three regulators only had a small impact on the overall efficiency by contributing 10.6 mA toward the losses. The main cause of waste in the system stems from the relatively large difference between the input and output voltages present. This waste is reduced when voltage is supplied by two 3.7 V nominal Li-Ion cells in series, for a total of 7.4 V.

Figure 2.9.6. - Power and Efficiencies

| Part | Input Voltage (V) | Output Voltage (V) | Input Current (mA) | Output Current (mA) | Input Power (mW) | Output Power (mW) | Efficiency |
|---|---|---|---|---|---|---|---|
| 5 V Regulator | 13.8 | 5 | 112.7 | 109.7 | 1555 | 548.5 | 35.3 |
| 3.3 V Regulator | 13.8 | 3.3 | 267.5 | 260 | 3692 | 858 | 23.2 |
| 2.8 V Regulator | 13.8 | 2.8 | 15.1 | 15 | 208.4 | 42 | 20.2 |

| | | | | | |
|---|---|---|---|---|---|
| Total | 395.3 | 384.7 | 5455 | 1449 | 26.6 |

Power and Efficiencies

With the Li-Ion cells providing power the net system efficiency approaches fifty percent. At an input current of only 395.3mA two 2400mAh 18650 cells should last for approximately 6 hours of data logging.

### *2.9.2. Low Power Mode*

A low power mode is necessary for when the device is still receiving power through the OBDII connector and the car is turned off, to prevent draining the car battery down. Since the OBDII standard calls for the battery positive to be switched with the ignition the device should log data whenever it is receiving power. However, we have found that many manufacturers do not adhere to the standard, which requires us to implement a low power mode which won't quickly deplete the car battery, yet will monitor for a sign to power up and start logging data again.

We will implement the microcontroller's built-in leads for a small 3V lithium battery, which keeps the real time clock running and contents of SRAM preserved while the external power is disconnected. To keep the microcontroller in a standby state while it is still powered and the car is turned off we will utilize a feature of the Rabbit 3220 called "sleepy mode". We can enter this mode from software based on any number of factors, including: RPM at zero, GPS reported speed at zero for a set length of time, or any other indicator we choose. When in sleepy mode the Rabbit switches from its main oscillator to a 32.768kHz oscillator. The reduced clock frequency cuts the power consumption from 841.5mW down to 0.33mW allowing the device to run in the background waiting for a signal to turn on without depleting the car's battery.

The device's ability to enter a standby mode will have the ability to be overridden by the user with a three way switch on the external case of the device, allowing the device it to be placed in either 'on', 'off', or 'auto' modes. The 'on' mode will have the device continuously log data while it has some form of power. Of course 'off' will have all devices powered down. Only a 12μA current pulled from the onboard watch battery to maintain the SRAM and RTC would be present. The 'auto' mode is what most users will use. The auto mode will start logging only when the car is running and place the system in a low power standby once the car's engine has shut down. While in low-power mode the device will wait for the engine to be started before it powers up and starts logging again.

If future testing deems it necessary to further reduce the standby current draw, the Rabbit microcontroller can use a clock divider to further reduce power draw. The 32.768kHz oscillator can be reduced by a factor of 2, 4, 8, or 16. This will place the device in an ultra low power mode.

### *2.9.3. Brown Out Protection*

Since the voltage output from the car's electrical system is not always going to be stable, we decided it was necessary to provide for possible brown outs. These are most likely to occur in short bursts when high current devices like the lights and air conditioning are switched on. An even more important concern is the possible extended brown out condition created by trying to start the engine with a weak or depleted battery.

For a brown out to actually impact our device the input voltage must drop to the highest minimum required voltage for each regulator. As described in the previous section, that would be the minimum input voltage of 5.29V at the 5V regulator. Until the car battery voltage drops to this point the device will remain unaffected. In the event that the input voltage drops below that 5.29V threshold the output voltage will begin decreasing until it reaches 4.75V, a five percent deviation from its regulated 5V calibration. At this point the regulator will throw an error flag at its output pin with may be used to shutdown the regulator and alert the microcontroller if necessary. This should not be a problem for the ELM327, GPS, or ADXL213, as they will turn off and then back on when power is restored and continue to operate normally. The ADIS16100 yaw rate gyro may need to be reinitialized pending tests, as the SPI protocol it utilizes is more complex than the others. The important part is that the error reporting pin should be tied to the regulator force shutdown pin so that the components are never feed less that 4.75V, which could possibly result in damage to them. In order to prevent the false "in regulation" reading at input voltages less than 1.3 V we will use pull up resistors connected to the output. The timing diagram shown here was taken from the LP2960 datasheet provided by Nation Semiconductor.

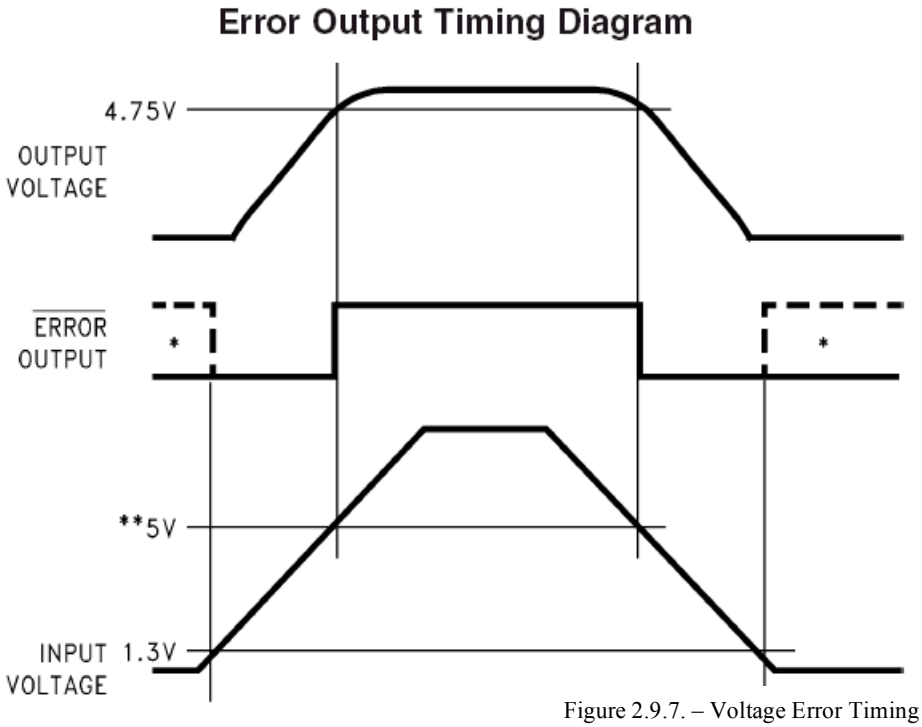



Figure 2.9.7. – Voltage Error Timing Diagram. Reprinted with permission. -National Semiconductor

In order to smooth out abrupt but short duration brown out situations a simple large capacitor will be connected from the input voltage to ground. Since the voltage regulators can disable themselves until their output voltage reaches five percent of the typical value the relatively long charge up time of the capacitor will not adversely affect the circuit. The larger the value chosen for the capacitor, the longer the device is protected from brown outs. In the sudden absence of power a 2200uF capacitor will run

the entire device for .0736 seconds till the regulators see 5.29V and the 5V rail goes out of regulation. It will last longer in a realistic situation where voltage is decreased and not just disconnected. Shown below is a MATLAB simulation of capacitor values ranging from 1µF to 10000µF and the approximate runtimes until an output voltage reaches 5.29V.

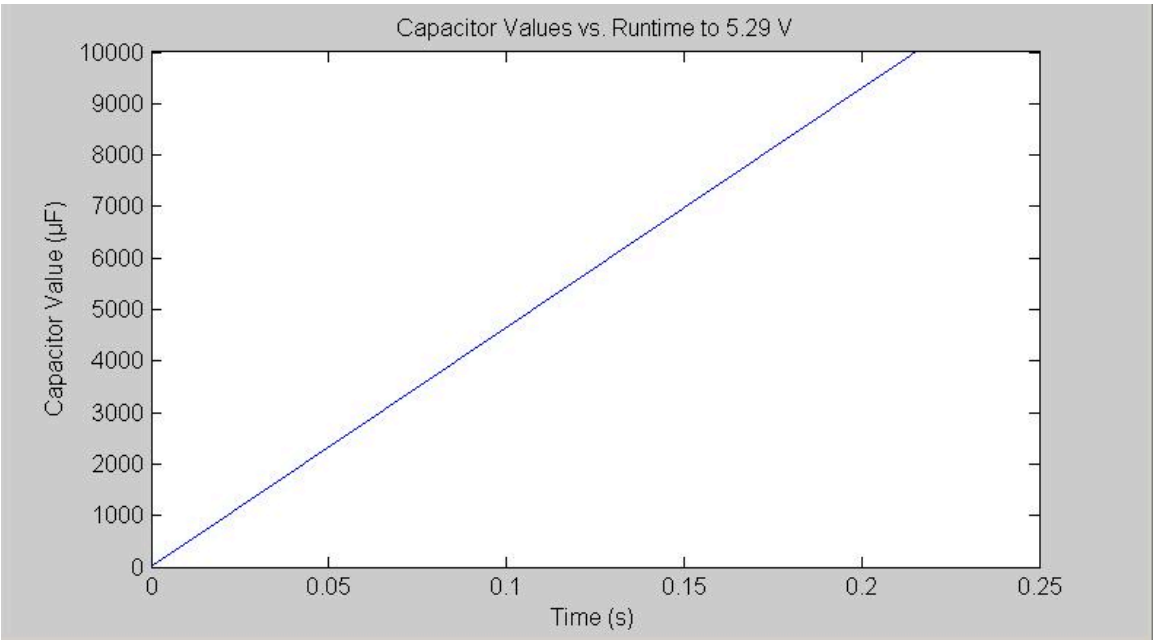



Figure 2.9.8. – Graph of runtimes from 13.8 V to 5.29 V

Based on this information we will begin testing with a 4700µF 35V electrolytic capacitor. A capacitor of this value should yield a, worst-case, runtime of 0.101 seconds following a total loss of power. After the first 101 ms the 5V regulator would shut down, powering off the GPS, accelerometers, yaw rate gyro, and OBDII interpreter. This would leave just the microcontroller, storage, and camera running at a current draw of 282.6mA. At which point the microcontroller would sense the 5V regulator's error pin go low and signal the storage device to finish its last command and halt the rest of the components. It would have about 19ms to do this based on a 4700µF capacitor. If this is not enough time to prevent file corruption on the flash card the capacitor value can be increased. The graph for capacitor values up to 10000µF is shown below.

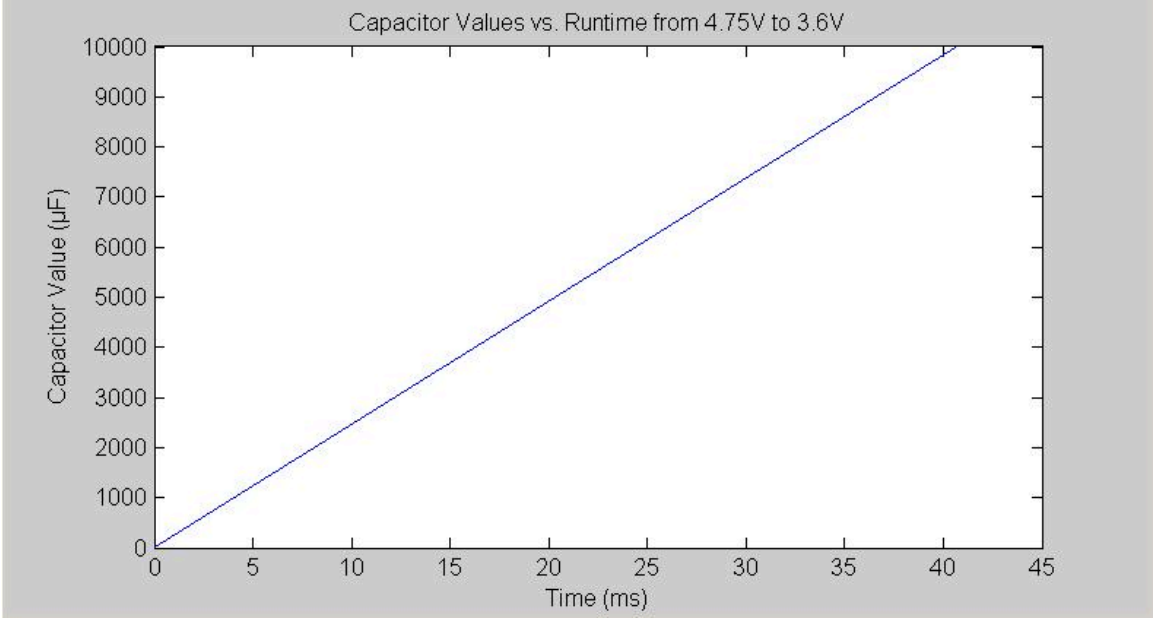



Figure 2.9.9. – Graph Capacitor values for runtimes from 4.75 V to 3.6 V

### *2.9.4. Battery Backup System*

An option that we would like to pursue at the end of this project is a Li-Ion battery backup system to run the device without requiring power from the vehicle. This would be useful for many different situations, including:

- Non standard OBDII connector that lacks a 12V wire
- Brown out protection while starting an engine with a failing car battery
- Protection from any loss of power that would affect logging
- Covert applications such as a roof mounted magnetic puck with no wires

Li-Ion cells are lightweight, with high energy densities, even discharge curves, and no memory effect, making them ideal for this application. At 3.7 V nominal per cell, only 2 cells in series would be required to power the entire device. The 18650 cell is used in most laptop battery packs, making them inexpensive and highly available. The two 2400 mAh cells in series will be enough for six hours of runtime before requiring a recharge. A constant current/constant voltage Li-Ion changer IC like the Linear Technology LTC4054 chip can easily handle recharging and trickle charging once power is reconnected.

If the cells are to be used as brown out protection in conjunction with the large capacitor at the input to the voltage regulators, it will need to use a relay to switch the voltage source from the car battery to the Li-Ion cells. This switching point could be any set reference voltage that indicates the car battery voltage is dipping dangerously low, such as 9 V. At this point the relay would fire and the cells would take over powering the device until the external power is restored. Below is a circuit diagram incorporating brown out protection using a Li-Ion battery backup.

## Circuit Schematic of Power Input Brown Out Protection

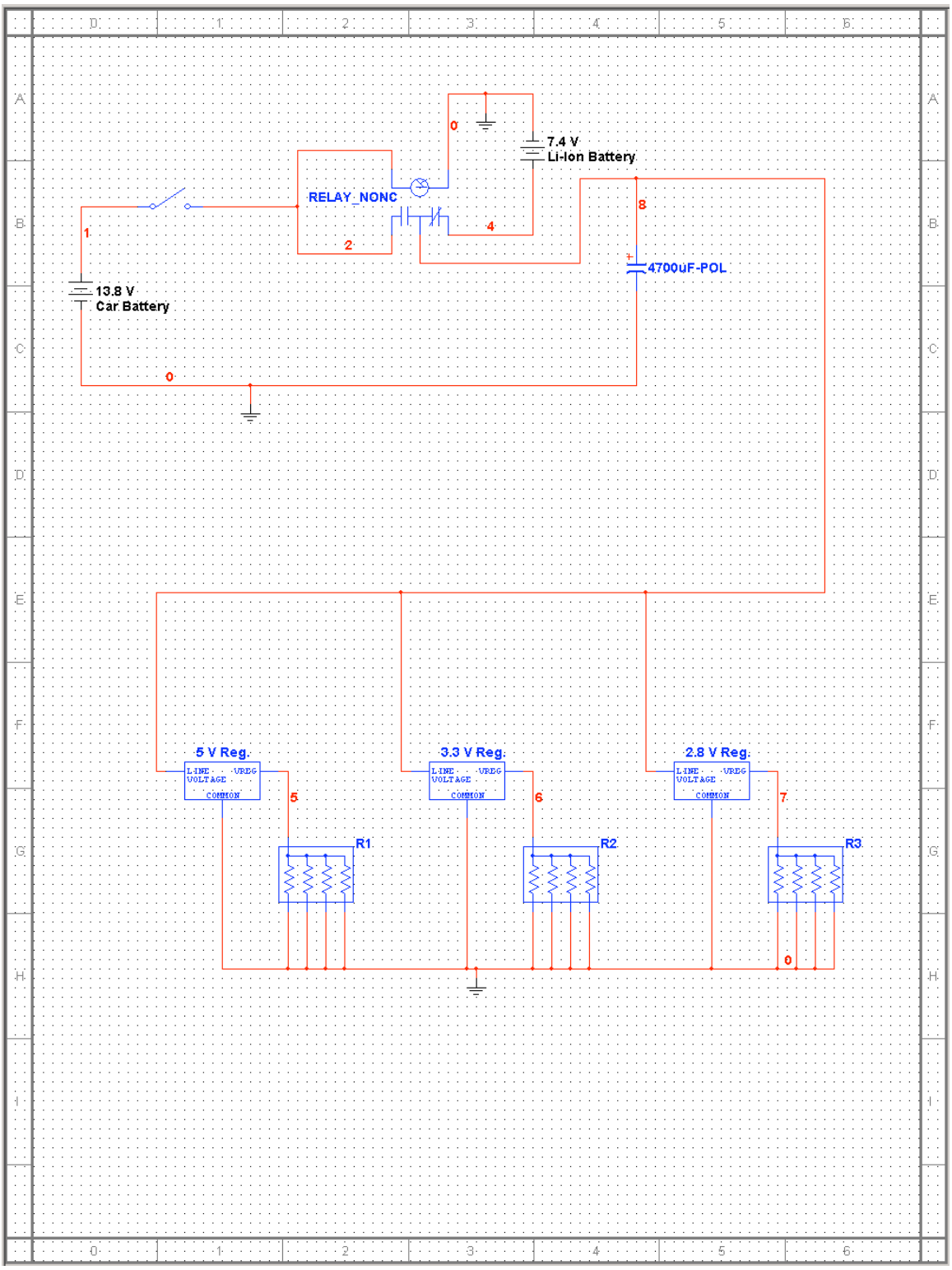



Figure 2.9.10. – Circuit Schematic of power input brown out protection

### *2.9.5. Thermal Considerations*

Keeping in mind this device will be used in a vehicle that can become quite hot in the sun, we thought it was necessary to look at the thermal aspects of the design. Linear voltage regulators can produce a significant amount of heat while running. The heat produced increases as current and voltage difference between input and output increase. The only thermal paths available to dissipate the heat are to the atmosphere and to the ground pins tied to the copper circuit board. Our initial testing demonstrated that the heat buildup in the regulators was not a significant concern, as the temperatures remained reasonably low. We attribute the low temperatures to the fact that the regulators are overbuilt for the circuit that they are being used in.

When a working prototype is built we will test it for heat buildup in the enclosure. If the working temperature of the regulators has risen to an unacceptable level we may have to add heatsinks to the chips, mount a fan to the enclosure, or add additional regulators to reduce the current through each one.

All of the devices have operating ranges tolerant of the conditions we expect in the vehicle.

## 2.10. Data Storage Device

The basic premise of this data logger is to log the data it accumulates. In order to log all of this data for easy retrieval we need a data storage method. This is another aspect of our device that is benefited by the popularity of consumer electronics. Digital cameras and portable MP3 players have ushered in cheap, fast, and larger storage medias.

### *2.10.1. File System Format*

The most obvious file format to use would be the same file format readable by about 95% of today's personal computers, Microsoft's FAT. There are a lot of file systems out there, Microsoft has a few, Apple has a few, UNIX has a few, Sun has a few, even Google has their own file system. But we want one that is readable easily by personal computers, and because of the popularity of Microsoft's Windows, the FAT system has been made readable to most Windows, Apple, UNIX, Sun, etc, systems. Within Microsoft's FAT, there is a choice between FAT12, FAT16, and FAT32, however FAT12 is fairly archaic, developed in 1977, and only addresses 32MB. FAT16 on the other hand will address 2GB and 4GB depending on the implementation; FAT32 will address up to 8TB. Obviously we need at least FAT16 support. We may need more addressing depending on storage media and sizes used. Currently compact flash has a maximum capacity of 12GB although the CF specification can support up to 137GB.

### *2.10.2 Crash Protection*

Crash protection is important as we do not want data to be lost if the device is improperly shut down. We do expect to design an intelligent power supply to keep all critical components running until they cleanly shut down, but in the event they do not, we do not want lost or corrupted data.

### *2.10.3 Data Integrity Verification*

Data integrity deals with not only corrupted data due to crashing and improper shut downs, it also deals with data being corrupted in transfer. Usually some sort of

checksum is used to check the data, but we can also use the microcontroller to read back the data that was just written to make sure they are the same.

### *2.10.4 Physical Media Format*

Physical media format is a large topic considering the number of media formats currently on the market. However from looking at the following chart, a few of them can be quickly ruled out.

Figure 2.10.1. - Storage Media Comparison

| Media Type -> | CompactFlash | | SmartMedia | MMC | |
|---|---|---|---|---|---|
| Varieties | I | II | | MMC | RS-MMC |
| Maximum storage capacity, MB | 8000 | 12000 | 128 | 4096 | 512 |
| Theoretical maximum capacity | 137 GB | 137 GB | | 128 GB | |
| Data read speed, MB/s | 40 | 40 | 2 | | |
| Data write speed, MB/s | 40 | 40 | | | |
| Read/write cycles | | | 1,000,000 | 1,000,000 | |

| Media Type -> | Memory Stick | | | |
|---|---|---|---|---|
| Varieties | Standard | Pro | Pro Duo | Micro |
| Maximum storage capacity, MB | 128 | 4096 | 4096 | |
| Theoretical maximum capacity | 128 MB | 32 GB | | |
| Data read speed, MB/s | 2.5 | 20 | 20 | 20 |
| Data write speed, MB/s | 1.8 | 1.8 | 10 | |
| Read/write cycles | | | | |

| Media Type -> | xD | | | Secure Digital | | |
|---|---|---|---|---|---|---|
| Varieties | | Type M | Type H | SD | miniSD | microSD |
| Maximum storage capacity, MB | 512 | 2048 | 1024 | 4096 | 2048 | 2048 |
| Theoretical maximum capacity | 512 MB | 8 GB | 8 GB | 128 GB | | |
| Data read speed, MB/s | 5 | 4 | 15 | 20 | | |
| Data write speed, MB/s | 3 | 2.5 | 9 | 20 | | |
| Read/write cycles | | | | | | |

Storage Media Comparison

The ones that can be ruled out are: SmartMedia, which has a maximum capacity of 128MB; and MMC, which has a maximum number of read/writes. The xD cards are also fairly limited in capacity right now and have relatively low read/write speeds. Not many vendors provide hardware for Sony's proprietary Memory Stick technology so that not only increases the costs but also decreases the likelihood of being able to find hardware in this format. So that leaves us with two options left, the CompactFlash and the Secure Digital, both of which have great read/write speeds.

#### 2.10.4.1. Physical Size

The Secure Digital cards are quite a bit smaller than the Compact Flash cards, however even though size is a constraint on our device, the difference the card will make in the overall size of the design is not worth worrying about if it allows us to use better technology.

2.10.4.2. Availability

Both the Secure Digital and Compact Flash cards are readily available, even a local retailers, at comparable prices. Both storage cards are available in sizes that are great for data logging purposes, and both are proven in the industry to be reliable data storage devices. The hardware used to interface with them although not available at local retailers, is also fairly easily to find for both formats. Based on the number of components for each format, it appears that the Secure Digital format is far more popular.

2.10.4.3. Power Consumption

Neither the Secure Digital nor the CompactFlash use much power, but our research has shown the Secure Digital to still use less power than the CompactFlash. Based on the differences between the two, which is not much, power consumption will not be a deciding factor.

Both the Secure Digital and CompactFlash formats are viable storage media formats, but because the Secure Digital is smaller, requires less power, and has more hardware easily available for it, we feel the Secure Digital format is the best choice for our device.

We decided upon a device from Wearable, Inc called the DOSonCHIP, which is available in a package that interfaces with an SD card. This chip handles all the low level file system commands and allows for higher level commands that we are familiar with. For example, we can tell the DOSonCHIP module "md A:\NEWDIR" and the device will create a new directory on the root of the removable media card called "NEWDIR." The device will them send back a result code to tell us whether the command succeeded or failed and, if it failed, why.

# 3. Software

## 3.1. Programming Language

### *3.1.1. Preface*

Dynamic C is a development platform for the Rabbit microprocessor, which bundles all of the development tools into a single application. It was developed by Z-World and Rabbit microprocessors and comes bundled with all Rabbit kits. Dynamic C uses the C programming language as a backbone with a few exterior variations and numerous behind the scene differences that allow a high level language such as C to be used on a low level platform such as a Rabbit microprocessor. Its key features include cheap price, clean simplistic development environment, powerful debugging tools, function chaining, and multitasking. Dynamic C also includes numerous pre-written libraries and sample programs allowing for quick and easy device interfacing and software development.

### *3.1.2. Rationale*

A development platform specially designed for Rabbit microprocessors that is shipped with all Rabbit development kits as well as can be individually purchased

through Rabbit and a handful of other vendors for around one hundred dollars. Being left with only one option for a development environment at first look may seem like a hindrance. But after working with Dynamic C and becoming familiar with its rich featured editor and debugger, huge selection of libraries and code examples, efficient compiler, and lighting fast PC to rabbit memory transfer rates we quickly learned it was an extremely easy to use and powerful application that would easily satisfy all of our requirements.

### *3.1.3. Included Tools*

With Dynamic C programmers don't have to worry about obtaining separate tools and licenses before they can begin development because the software comes bundled with an editor, compiler, linker, loader, and debugger.

#### *3.1.3.1. Editor*

The Dynamic C editor uses a Notepad-type interface matched with a rich feature set to provide an exemplary editor. Features include:

- Syntax highlighting which differentiates different source code elements by displaying them as different colors. Users can easily edit the default color scheme of syntax highlighting to better match an editor scheme they have grown accustomed to.

- Code templates allow users to quickly insert common code sections by simply right clicking in the editor environment. A window then prompts for a code section desired for insertion, such as a default if statement or comment box. Through the template options menu users can edit, create, and delete their own code sections to speed up development.

- Grep utility allows users to search for variables and text strings in a file as well as its directories, making searching for data in thousands of lines of code drastically faster.

- Column Mode gives users the ability to select sections of code based on their column location, after selection users can copy, paste, cut, shift code left/right.

- Bookmarks can be set at any location in a body of code by pressing ctrl-shift-# and with ctrl-# the page will jump to user set bookmark locations.

- Bracket matching is done via keystroke ‘ctrl-[‘ before the bracket you wish to find the delimiter for, once pressed the page will jump to the desired location.

#### *3.1.3.2. Compiler, Linker, and Loader*

To improve ease of use, compiling, linking, and loading are executed through one function. The compiler itself has a tough job in Dynamic C because compiling for a microprocessor and a PC are quite different. For example with a PC a compiler can assume the existence of an operating system and a clean slate ready for the program, but

for a microprocessor this can not be assumed, error flags might be thrown due to power cycles, watch dogs, and/or system errors. Dynamic C's compiler also handles function prototypes differently, instead of numerous include files; function prototypes are handled within the source libraries. It also differs in that it compiles the entire program from the source code and libraries, unlike typical compilers, which use separate modules, which are then linked.

#### *3.1.3.3. Debugger*

Dynamic C's debugger comes with all the standard debugging tools which will be briefly described below, all of which can be toggled on and off through the project options menu.

- Printf() displays user defined messages to the stdio window, serial port, or file valuable to show application progress without introducing interrupts.

- Breakpoints can be created anywhere in the program by the user, upon execution the program will run at full speed till it reaches the breakpoint. At which point a system screen shot is presented in the debug window and users can choose other debugging tools or continue program execution.

- Single stepping is one of the available debugging tools after a breakpoint has been reached, the name says it all, it allows users to advance through the C or assembly code one line at a time. And just how it sounds, it is very slow; it can also cause problems with any external device the Rabbit is attempting to talk to.

- Watched expressions can be placed on any C expression and allow the user to review the value of the watched expression after the program has finished execution.

- Memory Dumps present the data located at an address to the user, this is handy in ensuring strings and variables are sending and receiving the expected values and ensures memory remains accurate.

- MAP Files are generated post program execution and provides a summary of the entire programs memory usage, particularly useful when attempting to optimize memory efficiency.

- Assert Macros allow programmers to state a required condition, if the condition is not met at the time of the assert macro the program is halted and an error message will be returned.

- Execution Trace provides a means of post execution program flow checking, if a program is deviating from the expected path at an unknown point enabling execution tracing and reviewing the post execution report will show exactly where the program has a bug.

### *3.1.4. Dynamic C over and above C*

As stated in section 3.1.3.2 it is impossible to take C code written for a PC and directly load it into a microprocessor. Code manipulation to handle differences such as read only memory and assembly language is required to make it microprocessor friendly. Listed below are other features added to Dynamic C to ensure a clean microprocessor interface, make development less tedious, and in some cases, just make C porting for microprocessors possible.

- Function chaining enables declared blocks of code, usually initializations and recovery routines, to be called upon by multiple functions. When called, all portions of the function chain will execute, initializing the system or returning it to a safe state after a system error is encountered.

- Mixed Assembly and C is supported in dynamic C. Users only have to enclose their assembly code in the #asm and #endasm tags to program in the assembly environment.

- Protected variables are those that are defined by a prefaced protected tag followed by the type and variable name. Protected variables are stored within the rabbit's battery-backed random access memory, which can easily be recalled if power loss or other critical failure occurs.

### *3.1.5. Dynamic C Pitfalls*

Differences between Dynamic C and C are outlined below, it is important to consider these differences before beginning software development.

- Assigning a value to a variable during its initialization will cause it to be stored as a read only item in memory. To prevent this from happening do not assign a value until required later in the program.

- Static variables are defined by #GLOBAL INIT, but unlike normal C where these values are automatically set to zero, Dynamic C does not automatically assign values to these variables because of the possibility they will be receiving data from the system's battery-backed RAM.

- The #include directive is not used with Dynamic C, in its place is the #use directive, refer to 3.3.2 for further detail.

### *3.1.6. Multitasking*

Multitasking is a single processor non-parallel system attempting to artificially create parallel processing. Of course a single processor, no matter what the speed, will never be able to execute more then one instruction at a time. Instead the system uses delays in one task to allocate time to another task speeding up the overall application execution time.

These delays are ones set by programmers so that devices are polled at a staggered pace. It would be very simple to just put each polling task in one after the other with no delays and no consideration for the tasks around it, but this would result in a vast amount of resources wasted on tasks like polling the GPS numerous times per second when really our accelerometers and yaw rate gyro needed those resources. Or for instance if our camera system was polling and storing data to the Secure Digital card as fast as the processor can run even two gigabytes of space it would be filled in less then an hour. So to prevent the devices that only need to be polled occasionally from draining resources and filling memory we set a delay so that it only executes as much as needed.

#### *3.1.6.1. Multitasking Methods*

Dynamic C has two methods of multitasking, preemptive, which uses slice statements, and cooperative, where tasks share the processor when they are in a waiting state. Both methods of multitasking could satisfy the software requirements of our project, but due to the less intrusive behavior and more simplistic nature of cooperative multitasking, it will be used for our program.

Cooperative multitasking is the less intrusive of the two multitasking methods. Programmers declare a main system loop and then costates within it. The costate coordinates the sharing of processor power during delays. On the first run of a costate all statements are executed until a waitfor command is encountered, the system then breaks to work on other tasks. It will return after the allotted time has passed to finish execution. The figure below illustrates the program flow of cooperative multitasking operations.

**Costate Flow Control**

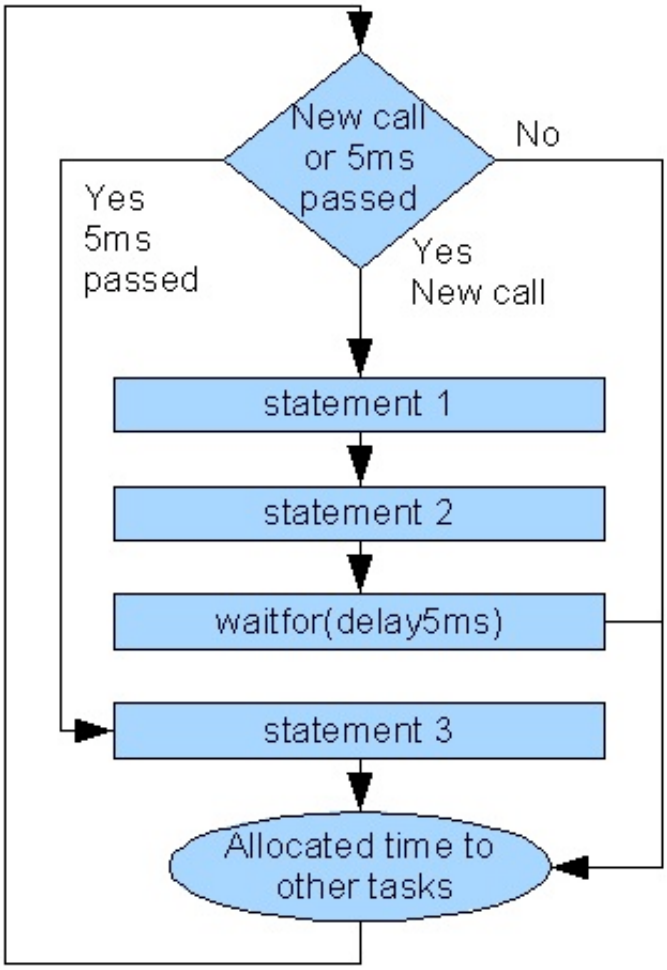



Figure 3.3.1. - Program flow for a costate containing statement.

For the current build of our software we have the GPS and accelerometer sensor working in separate costates. The accelerometer requires polling much more often than the GPS, so the system executes the first portion on the GPS instructions then polls the accelerometer several times before returning to poll the GPS again.

### *3.1.6.2. Modifying Costate Flow*

Besides simply initializing costates to run as outlined above, we will also be taking advantage of Dynamic C's other costate initialization options that allow a costate to only run one time unless it is re-flagged to execute again. Cooperative multitasking also has an abort option, which exits the costate and will never re-enter it.

The single costate initialization is done by declaring a named costate as init_on. Once this is done, it will only complete one full execution and then will be ignored until it is specifically flagged to be reset. We will be implementing this use of costates for our boot cycle. When initial power is given to the system, a diagnostic cosate will be run to

verify all devices are operational. After it has completed successfully it will remain dormant unless there is a call for a system reset. At this time it will be flagged to execute once again.

The 'yield' command allows the costate to operate without delay by skipping its lower portion. This method is needed if we are unable to poll a sensor quickly enough.

#### *3.1.6.3. Importance of Multitasking*

The multitasking concept is particularly important for our project due to the large number of external sensors we are using. The more sensors that require delays, the more efficient costates will make the program. This is because without multitasking one has to spend large amounts of time dealing with the tedious task of finding the perfect mixture of looping individual functions and intermingling other tasks at just the right time to get the desired polling rate. If a sensor's polling rate needs to be tweaked, the entire code would have to be modified. Multitasking saves us this potential heartache because delay modification is as easy as changing a single number in the polling function. That function's polling rate is modified without detrimentally affecting any other functions. It is also a possible future upgrade for the program to self modify its timing delays for different devices. This would allow in certain conditions, such as an emergency mode, all the sensors to poll twice as fast as normal in order to acquire as much data from the vehicle as possible in a short burst. Without the simplistic delay modification capabilities, performing this task would be nearly impossible.

### *3.1.7. Libraries*

Another benefit of Dynamic C is its dozens of communication and protocol libraries. Within each section devoted to the software for the sensors is a section discussing the libraries directly working with the sensors. As an example, within the GPS section the GPS.LIB is outlined, including an overview, explanation of the functions we are using, and any notes about the library. But the basic library that the GPS.LIB uses to serially communicate with the GPS is not covered, that is what this section is for. A dissertation of the basic libraries and their functions within Dynamic C that we will be using for our project is included. These libraries control the heart of the interfacing by setting baud rates, manipulating the status of ports, and controlling data through said ports. Having these libraries and functions bundled with Dynamic C drastically cuts development time. An experiment with so many sensor interfaces in such a short period of time would be impossible to accomplish if all the interface libraries had to be written before development began.

#### *3.1.7.1. Libraries of Interest*

RS232.LIB: This library establishes the functions to serially send and receive single and multiple characters of data through a RS232 converter.
SYSIO.LIB: A library at the heart of the Rabbit Module, it defines the systems registers as well as handles prep work for quadrature decoding.
SYS.LIB: Home of many of Dynamic C's data structure algorithms, program override commands, and hardware modification.
The functions from these libraries slated for use in the project are listed below in the table below, which outlines function name, description, sending parameters, and returned data.

Core Libraries:

| Library | Functions | Description | Parameter | Return on success | Return on failure |
|---|---|---|---|---|---|
| RS232.LIB | Int ser$open(long) | Open serial port & set baud rate | Baud rate | 1 | 0 |
| | Char ser$getc() | Pull character from serial port | - | Next char | -1 |
| | Int ser$putc() | Write character to serial port | Character | 1 | 0 |
| | Int ser$puts() | Sends a string over serial port | String | Chars sent | 0 |
| | Void ser$close() | Close serial port | - | 1 | 1 |
| | *$=port A,B,C,or D | | | | |
| | | | | | |
| SYSIO.lib | Int RdPortI(int) | Read internal I/O port specified | Address of internal port | Integer pulled from port | |

Figure 3.1.2. - Outlines main functions used from the main libraries.

### *3.1.8. WatchDog*

The Rabbit CPU has a built in watchdog to monitor the hardware for any occurrences of infinite loops or hardware faults. If one is encountered the system will be reset and the boot sequence will handle re-initialization of the system. The Rabbit also has ten virtual watchdog timers that require user initialization and periodic stepping. At this time we will be using only the single hardware watchdog unless a problem arises.

## 3.2. Memory

### *3.2.1 Rabbit Memory Management*

Our Rabbit's processor works with a logical address space of 16bit/64k with segments blocks of 4k, Dynamic C on the other hand, with the help of on chip memory management units works with physical address space of 20bit/1M with banks of 256k.

#### *3.2.1.1 Memory Management Unit*

On the Rabbit module the on chip memory management unit (MMU) handles the conversion of logical 16-bit addresses to physical 20-bit addresses. The logical space that the memory management unit works with is divided into four sections: xmem, stack, data, and base. The user can modify the address ranges of the four sections by changing the values within XPC, STACKSEG, and DATASEG. But changing the values at these locations can make the system extremely volatile. We do not think it will be needed for our application, seeing as how the majority of our memory accessing, which would require memory restructuring, will be on our main storage DOSonCHIP device.

#### *3.2.1.2 Memory Interface Unit*

After the MMU has performed the address conversion, the Memory Interface Unit (MIU) controls data access. There are five registers under the MIU but only four of them are of interest to our project: MB0CR, MB1CR, MB2CR, and MB3CR. Each of these has the capability to enable wait states, chip select, write enable, output enable, line usage, and write protection for its associated 256K block of memory. The chart below shows the values to set the different registers at to enable different settings.

| Memory Block Settings | | |
|---|---|---|
| **Bit(s)** | **Value** | **Description** |
| 7, 6 | 0 | 4 wait states |
| | 1 | 2 wait states |
| | 10 | 1 wait states |
| | 11 | 0 wait states |
| 5 | 1 | Invert address A19 |
| 4 | 1 | Invert address A18 |
| 3 | 1 | Write-protect memory |
| 2 | 0 | Use /OE0, /WE0 |
| | 1 | Use /OE1, /WE1 |
| 1, 0 | 0 | Use /CS0 |
| | 1 | Use /CS1 |
| | 1x | Use /CS2 |

Figure 3.2.1. - Memory Block Settings

### *3.2.2. On Board Memory vs. DOSonCHIP*

After reviewing direct memory modification in Dynamic C we learned it was both complicated and dangerous. Placing and retrieving a file in a set location is not as easy as just stating place/fetch file at this address. The programmer has to work with the address converters, which are usually transparent, to hunt down their data. The dangerous aspect comes into play when memory-mapping modifications are performed. The Rabbit's highest level of flash memory is unstable and only guaranteed to work for 100,000 writing cycles, if a programmer inadvertently addressed this memory location for the application to use normally, in a weeks time the memory could fail and destroy the device. In the Rabbit memory section above, the memory system is discussed because it will be used by the program for storing temporary data at its default settings. However, the methods of manually accessing and modifying memory or changing the memory space layout are not discussed because there is a better means of storing our data. Aside from looking for a simple non-volatile memory system, our device is also required have a large amount of external removable memory because we will have so many devices polling, in some cases very quickly. All the returned data has to be stored somewhere with a lot of space that can be easily removed for reviewing. Storing to the on board memory of the Rabbit would work, but would fill up its small memory cache very fast, or in our case a very short drive. Then you would have to worry about either removing the system or bringing a computer to it to download the data. Otherwise, if the device is shut down before the information is pulled, the only data that would remain is in the battery-backed sections. Instead of dealing with all this, we are using a DOSonCHIP module that

uses a Secure Digital card to store all of our data. Using a Secure Digital card gives us easy to use, non-volatile, and vast storage space compared to the Rabbit's built in memory.

### *3.2.3. DOSonCHIP Memory System/FAT32*

The DOSonCHIP memory management system is handled by its own on board File Allocation Table (FAT), which has support for both FAT32 and FAT16. Exactly how it interfaces with the FAT system is undocumented so we are only able to discuss the FAT system itself. The FAT32 file system has three sections; first is the boot sector, which contains file system information, locations of other system files, and operating system boot loader code. Second is the FAT region, that contains the file allocation table, which tells the system which clusters are allocated to data and which allocated to directories. Third is the data region, where the data and root directories are written. Actually filling the data region with the FAT file system works by breaking its available memory up into predetermined blocks ranging in size from 2K to 32K. When a file is written to the device it begins to fill one of these blocks, if the file exceeds a single box the location of the address of the next block is written in the file allocation table. This is done because the next block for the file might not necessarily be written in the next logical block in memory. There are many differences between the FAT32 and FAT16 system, but much of them are not applicable because they deal with exceeding the 2-gigabyte threshold. The difference that does apply is the fact that the FAT32 and FAT16 have different capabilities when it comes to the block sizes discussed above. A FAT32 is able to map a larger number of blocks, with a larger number of smaller block sizes files that are less likely to leave unused block space, thus being more space efficient. This was the deciding factor in choosing the FAT32.

## 3.3. Device Software Interface

Within this section each device, its required libraries and functions, and pseudo code, will be analyzed. The pseudo code is written for its initial test phase where only one sensor at a time is connected and tested. Once each has passed the initial phase the code will be slightly modified for the final build. This includes moving all initialization commands from the multitasking loops to within a boot sequence function. This way all the devices will be initialized once instead of during each iteration of their multitasking loop. Another change will be removing the Secure Digital storage commands from all of the multitasking loops and moving them to a single multitasking loop dedicated solely to storage of all accumulated data in the polling master cycle. These are the only two modifications required to transform the presented pseudo code, if it initially works properly, into the final software build.

### *3.3.1. Dual Axis Accelerometer*

The dual axis accelerometer will be communicating with the Rabbit module via pulse width modulation (PWM). To decode a pulse width modulated input, quadrature decoding matched with Dynamic C's included R3000.lib file is required. The functions of interest within this library are as follows:

#### *3.3.1.1. R3000.LIB Function Descriptions*

-void qd_init(int iplevel)
Uses SYS.lib's SetVectIntern, qd_isr, and WrPortI to enable the quadrature encoder, its corresponding port (F), and specifies address space to pull initial test data. If test ran successfully the error flags and high counters are reset and program flow is passed back to the main function.

-long qd_read(int channel)
Polls the desired one of two channels on the quadrature enabled port repeatedly until it either receives a reading or an error flag. If error flag received it re-polls, else if it receives a high signal from the accelerometer it adds one to the temporary counter. If at that instant the accelerometer is sending a low signal zero is added. Because of the fact that the running total of highs is only manipulated on calls of qd_read many calls are needed in a cycle to get an accurate reading.

-void qd_zero(int channel)
Resets the corresponding channel's temporary counter back to zero.

#### *3.3.1.2. Translating Raw Data to Acceleration*

To translate the highs and lows being received from the accelerometer into actual acceleration data you must find the percentage of highs you are polling from the device. Then with the aid of a reference chart from “Analog Device” you can find the current acceleration. The acceleration sent as a function of the PWM duty cycle. Fifty percent duty cycle indicates no net acceleration, positive or negative accelerations are given by: Acceleration (g) = (Change in duty cycle)/(30%).

#### *3.3.1.3. Pseudo Code and Flow Chart*

-initialize reading to accept  returned data from polling the accelerometer
-initialize avgDown to store the average amount of time the accelerometer sends a low signal
-initialize avgUp to store the average amount of time the accelerometer sends a high signal
-initialize UpCounter to store the amount of high signals the accelerometer sends
-initializes DownCounter to store the amount of low signals the accelerometer sends
-initializes Lcounter to store the amount of iterations Multitasking Loop One has performed
-initialize quadrature decoder and port F to receive PWM data
-initialize array ACCarray with “Analog Devices” acceleration reference chart
-reset channel one's high counter
-set all variables equal to zero so as not to corrupt all the associated counters

Loop Forever:
Multitasking Loop One: This loop will execute every twenty milliseconds, which is set by the delay at the beginning of the loop.

-insert twenty millisecond delay

-set reading equal to quadrature decoded data polled from port F with R3000.LIB qd_read
IF reading is less then zero an error in polling has occurred
-reset channel one's high counter which will reinitialize it back onto a valid polling frequency
-set reading equal to quadrature decoded data polled from port F with R3000.LIB qd_read

-increment Lcounter to have a running total of amount of times Multitasking Loop One has run

-set UpCounter equal to reading
-set DownCounter equal to Lcounter minus UpCounter

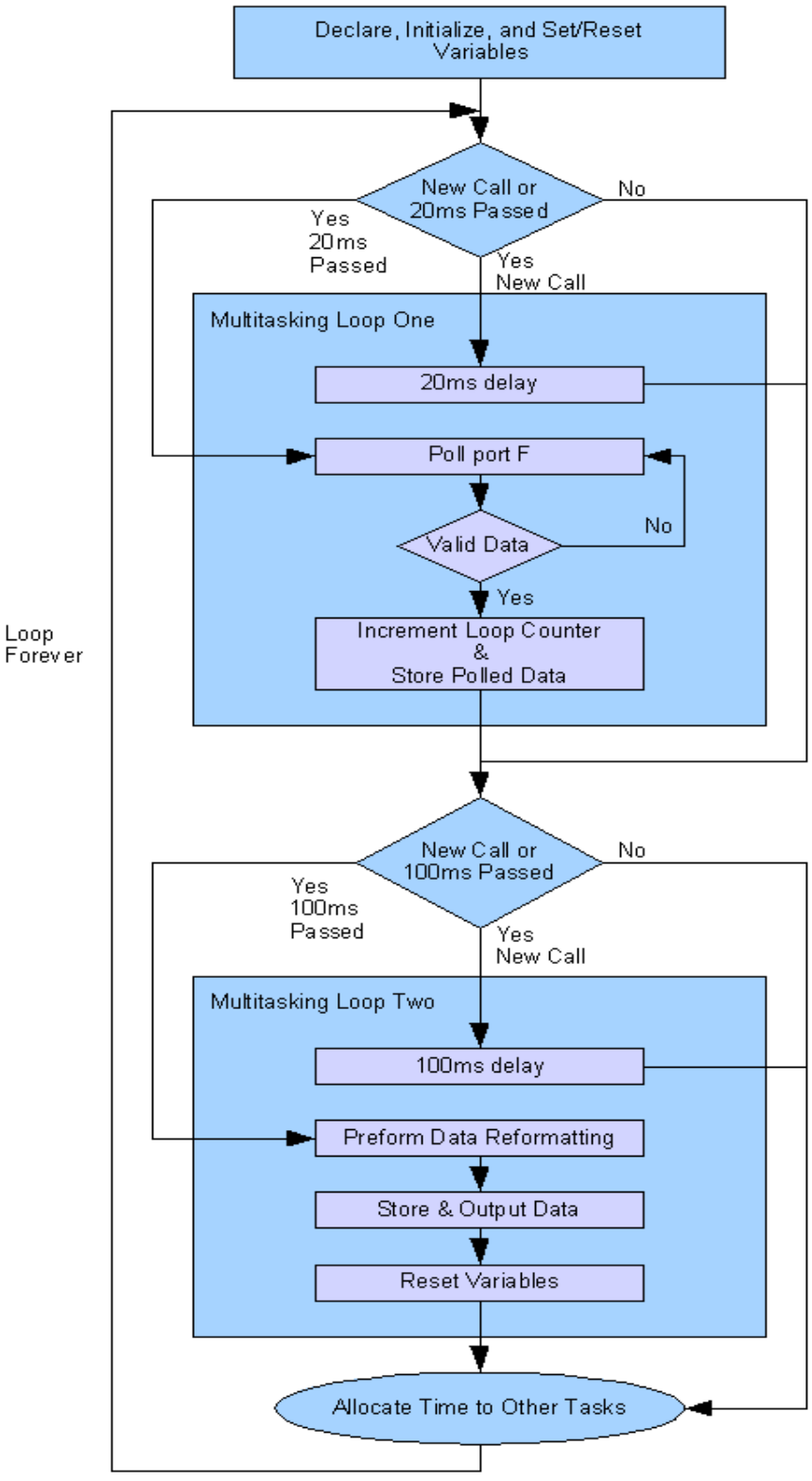
End Multitasking Loop One
Multitasking Loop Two: This loop will run every one hundred milliseconds, which is set by the delay at the beginning of the loop.

-insert one hundred millisecond delay
-divide UpCounter by Lcounter and assign result to avgUp
-divide DownCounter by Lcounter and assign result to avgDown
-using avgUp ACCarry pull associated acceleration data converting and result store in accelRate
-store accelRate to Secure Digital card
-output avgUp to Dynamic C display window to view validity of data *For Testing*
-reset Lcounter, UpCounter, and DownCounter to zero so a new acceleration rate can be found for the next cycle
-reset channel one's high counter

End Multitasking Loop Two
End Loop Forever

Below is a flow chart illustrating accelerometer pseudo code.

**Figure 3.3.1. - Accelerometer data polling, conversion, and storage.**



#### *3.3.1.4. Pseudo Code Notes*

The time delays set for each of the multitasking loops as of now are arbitrary, based on preliminary testing of the device set at these delay rates the outputted percentages are as expected, but far from precise. Attempts made to lessen delays to get faster polling and overall shorter period were not successful, data outputted looked very similar to the original one and five hundred millisecond delays within the pseudo code. We have surmised that our problem is not coming from the delays we are setting but possibly the natural delay required for the output to be sent to the PC for us to view for testing. It was decided further testing would be conducted using the Secure Digital memory unit and we would no longer need the data to be sent to the PC. Once this is done we will continue to test and tweak our delays to get accurate acceleration rates.

### *3.3.2. Yaw Rate Gyroscope*

The yaw rate gyro will be communicating with the Rabbit via Serial Peripheral Interface (SPI). Communicating with SPI requires a master and a slave; the Rabbit will be acting as the master tasked with clock generation and data flow. The yaw rate gyro on the other hand will be the slave and control serially shifting data in and out. The Rabbit requires the inclusion of the SPI.LIB, the functions of interest within this library are in the next section.

#### *3.3.2.1. SPI.LIB Function Descriptions*

void SPIinit(void)

Using the previously user defined port (A, B, C, and/or D), enables the desired port for UART output and sets the second half of the port as the SPI bit rate. Once complete it initializes itself as either a master or slave based on user definitions previous to the call. We will never be setting it as slave because it prevents us from having other devices connect to it properly. As a master, it prepares the clock for master mode and passes control back.

Int SPIWrRd(void *pointer, void *pointer, int)

This function handles data flow through the Rabbit using the three parameters passed to it. Parameter one is the address of the data the system has to send. Parameter two is the address of where the received data needs to be saved. The third parameter is the size of the data. The system first stores the data in from the function in the appropriate locations, then pulls anything at the address of parameter one. Once completed data is again sent and received.

Int SPIWrite(void *pointer, int)

This function controls sending data through the SPI port. Parameter one is the address of the data to be sent through the SPI and parameter two is the size.

Int SPIRead(void *pointer, int)

This function controls reading data in from the SPI port. Parameter one is the address of the data to be stored from the SPI and parameter two is the size.

The user sets the addresses mentioned above so a safe address for storage on the Rabbit module will have to located. The address sent to the yaw rate gyro controls different functions, a zero address sent to the system will initialize the gyroscope, where as a one address will initialize the temperature sensor.

#### *3.3.2.2. Translating SPI Data Flow to Yaw Rate*

To translate data flow through the system into the yaw rate the size of the data being stored must be analyzed. The size of data received from the yaw rate gyro is directly proportional to the yaw rate itself. So the conversion is as easy as getting the size of each block of data the Rabbit is being instructed to store each cycle.

#### *3.3.2.3. Pseudo Code and Flow Chart*

-initialize the use of SPI.LIB
-initialize dataSize to store data size of data received from yaw rate gyro
-initialize YRGerr to store error flag for yaw rate gyro
-initialize pointer safAddress pointed at a safe address location to store received data
-initialize array YRGarray with “Analog Devices” yaw rate gyro reference chart for any conversions necessary
-set all variables to zero to prevent use of possible old data at memory locations
-set port to be used for SPI communication
-set system for SPI master mode
-initialize desired port for UART output and system for master mode
-using a separate line on the yaw rate gyro discussed in the hardware section send logical high to begin yaw rate gyro self-test function and warm up the unit mechanically, repeat twice to ensure system mechanically ready

Loop Forever:
Multitasking Loop One: This loop will execute every two hundred and fifty milliseconds, which is set by the delay the beginning of the loop.

-insert two hundred and fifty millisecond delay
-using write/read function send pointer safAddress, zero, and zero to initialize gyroscope functionality and receive data saved to memory location safAddress
-compute the size of data located at safAddress and store to dataSize
-convert dataSize using YRGarray and store to dataSize
-store dataSize to Secure Digital card

End Multitasking Loop One
End Loop Forever

Below is a flow chart illustrating yaw rate gyroscope pseudo code.

**Yaw rate gyroscope data pulling, conversion, and storage**.

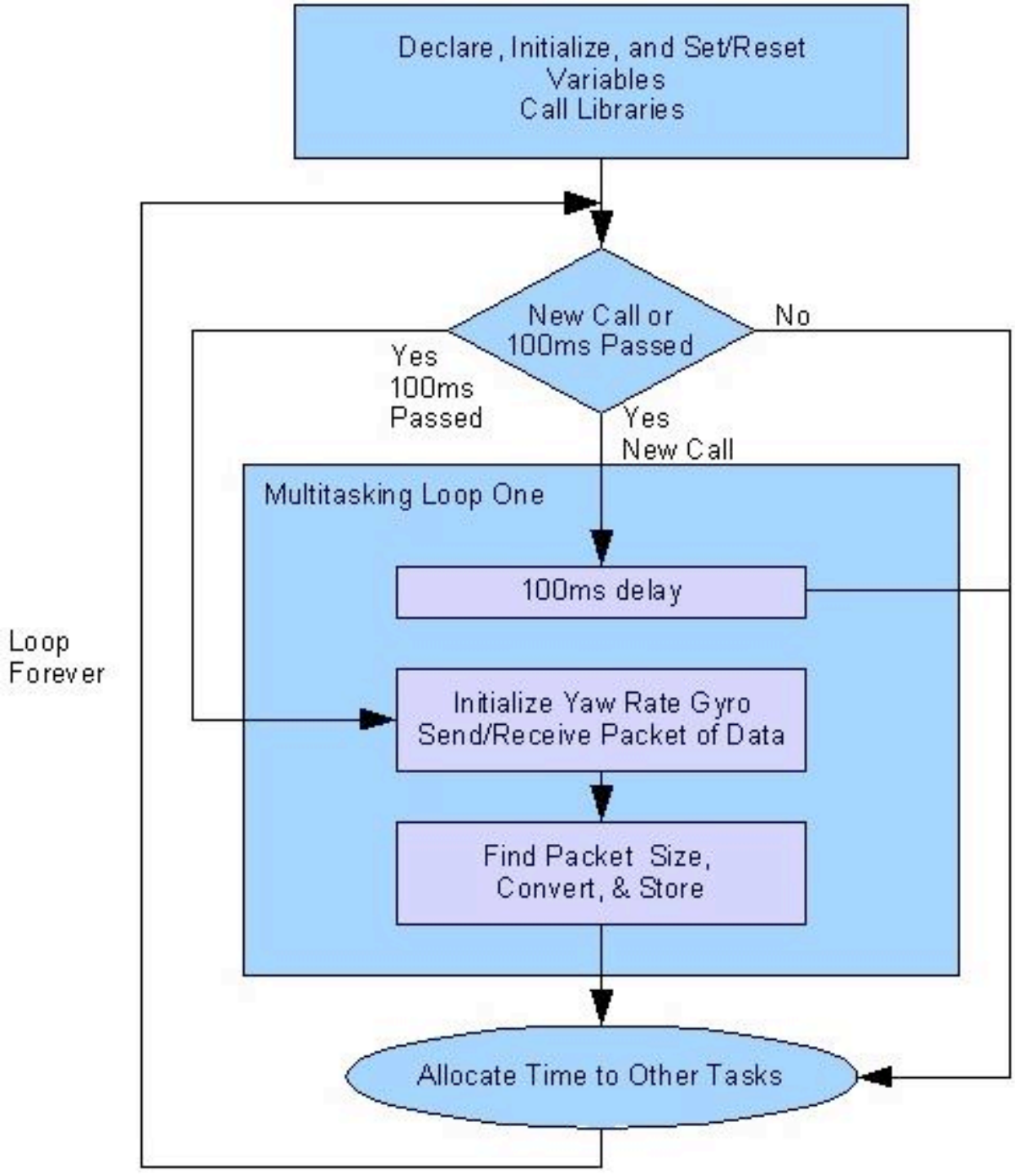



Figure 3.3.1. - Illustrates yaw rate gyroscope data pulling, conversion, and storage.

### *3.3.2.4. Pseudo Code Notes*

This device has yet to be installed and tested, so the time delay for multitasking was arbitrarily picked to be the same as the accelerometers writing cycle delay of one hundred milliseconds. You may have noticed that a port was not declared for the yaw rate gyro. We have not yet decided which of the four possible ports to use.

### *3.3.3. Global Positioning System:*

The Garmin 18 GPS unit communicates using the National Marine Electronics Association's NMEA-0183 standard for marine electronics devices, though other standards exist the Rabbit, controller comes with libraries specifically for NMEA-0183.

The NMEA-0183 protocol works by sending an ASCII string containing six parts. The first part is the dollar sign symbol which denotes the beginning of the sentence, second is a two character talker ID, third is a three character sentence ID, fourth is comma delineated data fields which contain all the relevant sentence data, fifth is a asterisk marking the end of the comma delimited section, sixth is a two-digit checksum, and seventh is a carriage return and line feed.
The entire sentence contains no more then eighty-two characters, including the dollar sign, carriage return, and line feed. Below is a table of the different talker and sentence IDs and what they represent.

Figure 3.3.2. - GPS ID Table

| Talker ID Description | | Sentence ID Description | |
|---|---|---|---|
| GP | Global Positioning System Receiver | GGA | GPS Fix Data |
| LC | Loran-C Receiver | GLL | Geographic Position: Longitude and Latitude |
| OM | Omega Navigation Receiver | RMC | Recommended Minimum Specific GPS/Transit Data |
| II | Integrated Instrumentation | | |

GPS Talker and Sentence IDs

This eighty-two character max length is helpful when it comes to error checking, if a sentence stream surpasses this eighty-two char limit there was an error in processing. The system either missed the carriage return or the device is corrupted, the likelihood of a corrupted system is rare, so the course of action will be to re-poll a new sentence and check for appropriate sentence length. Before the ASCII sentence containing data from the GPS is routed to the Rabbit it first must run through an RS232 converter to ensure the correct voltage boundaries are set and the data is in the correct format for a Rabbit module to receive. To quickly and easily decode the ASCII sentences Dynamic C's

GPS.LIB was used. The functions of interest to us within this library are in the next section.

#### *3.3.3.1. GPS.LIB Function Descriptions*

int gps_parse_coordinate(char *coord, int *degrees, float *minutes)
A sub-function of the three below. It converts the coordinates sent to it by *coord into degrees and minutes.

int gps_get_position(GPSPosition *newpos, char *sentence)
GPSPosition *newpos: declares the structure for this execution of gps_get_position() to fill char *sentence: incoming data from the GPS unit. This function is able to decode any of the three sentence ID types in figure 3.3.2, GPS for its GPS position data. This is done by first parsing the sentence ID out of the string and sorting the data to a function based on its ID so that it may handle the different formats of the different sentence types. After sorting gps_parse_coordinates is used to convert format, then the GPS data is stored into the library's GPSPosition structure seen below. On successful execution a zero is returned, a negative one denotes parsing error, and negative two denotes an invalid sentence. The following is the struct that the data is stored in.

```
Typedef struct{
int lat_degrees;
int lon_degrees;
float lat_minutes;
float lon_minutes;
char lat_direction;
char lon_direction;
} GPSPosition;
```

int gps_get_utc(struct tm *newtime, char *sentence)
struct tm*newtime: declares the structure where the UTC time data is stored.
char *sentence: incoming data from the GPS unit.
This function decodes sentences with the sentence ID RMC and stores the time data into predefined structure for time stamp storage. On successful execution a zero is returned, a negative one denotes a parsing error, and a negative two denotes an invalid sentence.

float gps_ground_distance(GPSPosition *a, GPSPosition *b)
GPSPosition *a: pointer to point a
GPSPosition *b: pointer to point b
gps_ground_distance computes the ground distance between two points by using pointer a, pointer b, and predefined earth radius variable. Once calculation is complete the distance is returned to the main function via a return type of float.

The program we write will pull characters from the port that the GPS is connected to one character at a time, filling a string called sentence with the data. Once a complete sentence is created and verified Gps_get_position decodes the sentence polled from the GPS unit into its basic elements of lat_degrees, lon_degrees, lat_minutes, lon_minutes, lat_direction, and lon_direction. It then loads those values into defined structure called

GPSPosition. Gps_get_utc works exactly like the previous function though instead of decoding GPS data, it decodes time from the data pulled from the satellite. Gps_ground_distance works by accepting two GPS coordinates and computing the distance between them using a spherical model of the earth.

#### *3.3.3.2. GPS Data Translation*

The data gps_get_position function decodes does not require any data manipulation, calling the functions from the GPS.LIB automatically fills a structure, after which pulling coordinates is as easy as saying longitude = GPSposition.lon_degrees. This pulls the most recent longitude and stores it in a variable we define for output or storage. The function gps_get_utc. on the other hand, does require a small amount of manipulation to convert its numerical based date into character based. This will be done by defining character arrays filled with the days of the week and another with the months of the year, we will then use these arrays to cross reference the gps_get_utc return data.

#### *3.3.3.3. Pseudo Code and Flow Chart*

-initialize the use of GPS.LIB
-define a max sentence size of 100 characters to validate sentence length, as stated above a sentence should never run over eighty-two characters, if it does the system must be reset because an error was encountered
-Initialize arrays to store the days of the week to convert numerical data from gps_get_utc into a character based date
-initialize arrays to store the months of the year to convert numerical data from gps_get_utc into a character based date
-initialize a new variable using GPS.LIB's GPSPosition structure called curPos to store current positions gps_get_positions pulls from GPS
-initialize a new variable using GPS.LIB's tm structure called curTime to store current time returned from gps_get_utc call to GPS
-initialize string variable sentence using max sentence size
-initialize variable charChecker to analyze individual characters in string sentence to check for a carriage return or new line
-initialize sting curDirecton to store current direction for latitude and longitude
-initialize noCord to store flag if there was no coordinate fetched yet
-initialize variable stringPosPointer to point at the location in the sentence array is of concern
-initialize variable curLong to store the current longitude pulled from GPS
-initialize variable curLat to store the current latitude pulled from GPS
-initialize array curTimeStamp to store the current time stamp from the GPS

-using RS232.LIB function serCopen() set baud rate to 4800
-set all variables to zero to prevent possibility of preexisting data at memory locations

Loop Forever:
Multitasking Loop One: This loop will execute every thirty seconds which is set by the delay at the beginning of the loop

-insert thirty second delay
-set noCord to one so that the follow loop continues to execute until a coordinate is fetched

Loop as long as noCord remains equal to one
-assign character fetched with RS232.LIB's serCgetc function that pulls a single character from the serial input line to charChecker

IF charChecker is equal to carriage return or newline proceed with coordinate pulling
    -set value in sentence at stringPosPoint to zero for subsequent coordinate pulling
    -set stringPosPoint to zero for subsequent coordinate pulling
    IF call of gps_get_position to fill curPos executes without error proceed with storing data
        -store current longitude direction from curPos into curDirection at position zero
        -store current latitude direction from curPos into curDirection at position one
        -store current longitude from curPos into curLong
        -store current latitude from curPos into curLat
        -set noCord to zero to show coordinate successfully fetched
    IF call of gps_get_utc to fill curTime executes without error proceed with storing data
        -store current time stamp from curTime into curTimeStamp
        -using days of week and months of year array reference arrays convert data in curTimeStamp from numerical to characters
    -store variables holding current position and current time into Secure Digital card
ELSE IF charChecker is valid date greater then zero
    -store data in charChecker in sentence at the location of stringPosPointer
    -increment stringPosPointer to point at the next location in sentence
    IF stringPosPointer equals max sentence
        -error has occurred reset stringPosPoint back to zero so can reset data pulling

End onCord verification Loop
End Multitasking Loop One
End Loop Forever
Below is a flow chart illustrating GPS pseudo code.

## GPS Pseudo Code Flow Chart

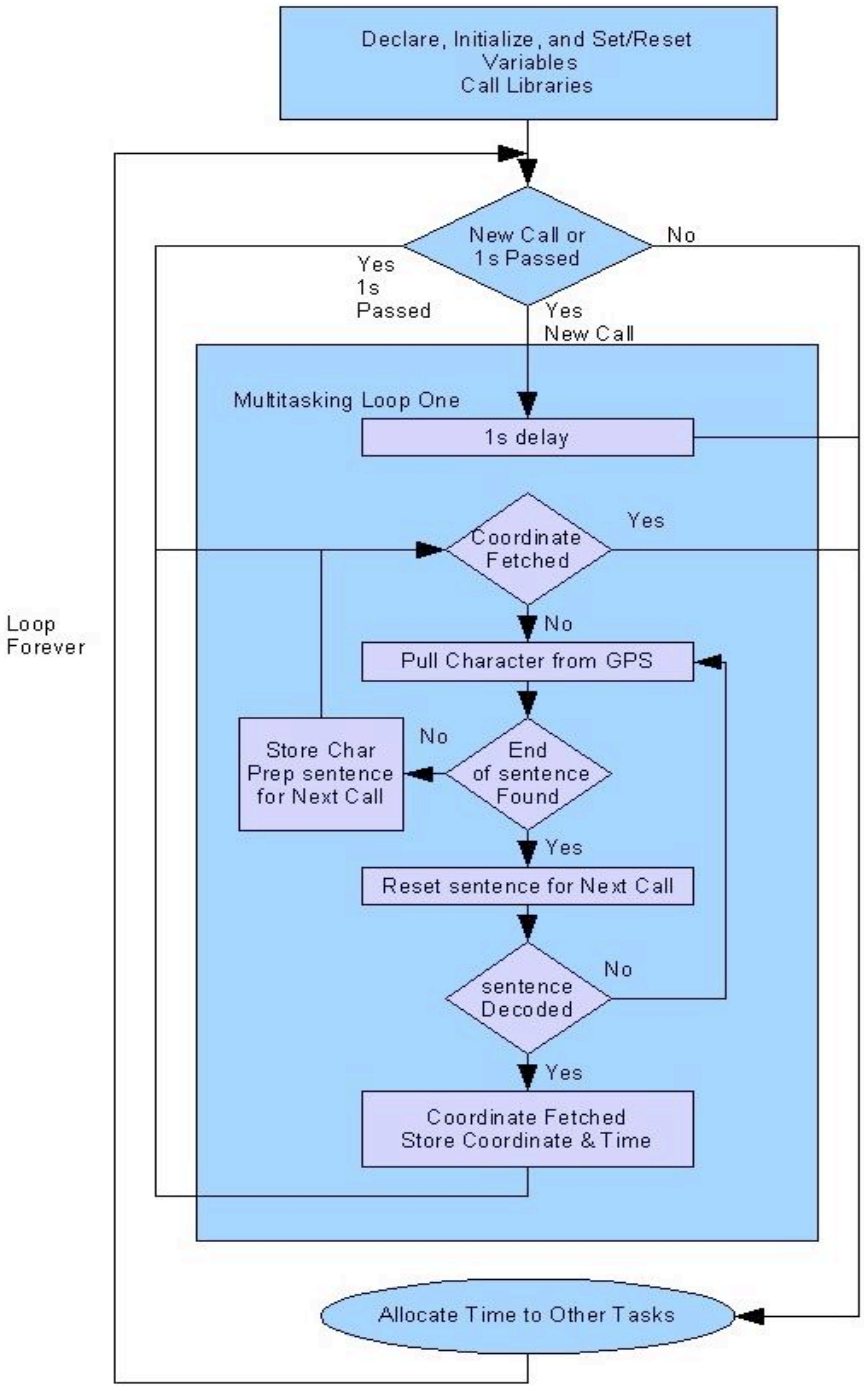



Figure 3.3.3. - Illustrates pulling data, error checking, and storing data.

#### *3.3.3.4. Pseudo Code Notes*

This device has been installed and tested, the data output has passed a reality check so the delay for multitasking was purposefully set to a large interval. This keeps it out of the way while we attempt to install and calibrate other sensors that have yet to be verified and require more attention and polling.

### *3.3.4. Camera*

Our TransChip TC5747 CMOS camera module will interface to the Rabbit through an inter-integrated circuit (I2C) serial bus. The camera is very small and efficient, as it was developed for use in cell phones. It is also extremely flexible in the available protocols for data transfer and control lines. Features that we will take advantage of include:

- Small size
- Low power consumption at 42mW
- Built in still and video JPEG encoder
- Flexibility in the control and data lines available
- Adjustable down sampling and digital zoom
- Adjustable frame rate
- Automatic exposure, white balance, and anti-flicker

We will begin testing the camera with only the I2C two-wire bus. This simple protocol will allow us to setup the camera's control registers and download the compressed JPEG images to the microcontroller for storage. However, we believe that the I2C serial bus may be too slow to support VGA resolution images at greater that one frame per second sampling rate, as it is limited to a bit-rate of 400kbps. If we determine after testing that the I2C bus is a bottleneck when increasing the frame rate beyond one frame per second we will have to implement a parallel bus to allow future expandability and higher bandwidths.

The I2C bus is a very simple protocol in that it only needs a clock wire (SCL), and a data wire (SDA). It requires devices that wish to communicate be arranged in a master/slave arrangement. The master will provide the clock and initiate the communication while the slave monitors the data line for data prefaced by its seven bit address. In this case the Rabbit will act as the master and the camera will be the slave. To communicate, the Rabbit will first establish and stabilize the clock on the SCL line of 400khz. Then it will send out a start bit on the data line followed by the seven bit address of the slave device. The next bit sent will determine if the slave is designated to send or to receive data. By sending a 1 the master is saying that it will be sending out one byte of data for the addressed slave. On the other hand, a 0 means that it is expecting the slave to send data back. On our setup the camera module has been designated the seven bit address 0x47, it can be designated read or write by appending a zero or a one to the end of the address according to the following table of values.

Figure 3.3.4. - Camera Address Modes

| Action | Hex | Binary |
|---|---|---|

| | | |
|---|---|---|
| Address | 0x47 | 1000111 |
| Slave will receive | 0x8E | 10001110 |
| Slave will transmit | 0x8F | 10001111 |

Camera Address Modes

Master Transmitting

The master will begin transmitting with the start bit followed by the address of the intended slave. Once the proper slave address is sent followed by the write bit (0), the slave sends out an ACK bit to acknowledge receipt of the command. The master is now free to send data one byte at a time as long as the slave sends an ACK bit after each received byte. When the master has finished sending all data, it will wait for the final ACK bit and then issue the stop command.

Slave Transmitting

The master will send out the start bit and the slave address followed by a one to signify the slave's permission to send data. The master will then issue the ACK bit and the slave will send one byte of data. This pattern will continue until the master fails to follow each byte with an ACK bit, this means that the transmission is complete and the master will issue the stop command.

#### *3.3.4.1. Power Sequencing:*

The camera module requires two additional lines for power up sequencing and enabling low power modes. These pins, PS1 and PS2, must both be set to 0 during startup until the clock stabilizes, then PS2 is set to 1 for full operation. A table showing the operational modes is shown below.

Figure 3.3.5. - Camera Operational Modes

| Mode | PS1 | PS2 |
|---|---|---|
| Startup | 0 | 0 |
| Full Operation | 0 | 1 |
| Sleep | 1 | 1 |
| Power Down | 1 | 0 |

Camera Operation Modes

A low power bypass mode is enabled by sending the serial low power mode over the data lines, and then setting PS1 to 1 and PS2 to 1 (sleep mode). The camera can then be woken up over the serial I2C bus.

### *3.3.4.2. I2C.LIB Function Descriptions*

The dynamic C environment includes a library called I2C.LIB, which contains many useful functions for using the I2C bus on the Rabbit microcontroller.

int i2c_init();
This function initializes the I2C protocol and defines the default I/O pins. We will be using Port D for the I2C clock and data lines. Rabbit pin PD6 will serve as the SCL clock line, and pin PD7 will serve as the SDA data line.

int i2c_wSCL_H();
This function will manually set the defined SCL clock output pin to high, and wait a predefined amount of time for the slave to stretch it if necessary. If the slave stretches the clock too long the function returns a -1.

int i2c_start_tx();
This function will initiate I2C data transmission by sending a start bit (S) and waiting for a set delay to see if the slave is stretching the clock. It returns 0 for a success and a -1 if the slave stretched the clock for too long.

int i2c_send_ack();
This function sends the ACK bit to acknowledge that data was successfully sent. It returns 0 for a success and a -1 if the slave stretched the clock for too long.

int i2c_read_char(char *ch);
This function will read in eight bits of data from the slave device (camera). It will tolerate the slave stretching the clock to allow communication with slower devices. The parameter “char *ch” is the character return buffer. The function returns 0 for a success and a -1 if the slave stretched the clock for too long.

i2c_check_ack();
This function checks to see if the slave sent an ACK bit on the clock pulse. It will return a 0 for ACK received, 1 for NAK received and -1 if the slave tried to stretch the clock for too long.

int i2c_write_char(char d);
This function is obviously used the write char d to the slave. It will return 0 for a success, 1 for a NAK, and -1 if the slave tried to stretch the clock for too long.

void i2c_stop_tx();
This function issues the stop command (S).

iint i2c_wr_wait(char d);
This function will try to write char d to the slave until it responds or the predefined value “Max i2cRetries” is reached. It returns 0 for a success, and a -1 to indicate a failure after too many retry attempts.

### *3.3.4.3. Image Translation*

The bulk of the work for data formatting and compression is performed by the microcontroller on-board the camera module. It will adjust many settings automatically such as the white balance, exposure, and anti-flicker. It also allows adjustment of the frame rate and down sampling by setting the control registers over the I2C bus. The raw images it captures are compressed into JPEG image files and stored on the module's memory until they are transferred off over the I2C or parallel data bus. Our only job is to download the formatted data from the camera and send it to the DOSonCHIP for storage onto the SD card.

### *3.3.4.4. Pseudo Code and Flow Chart*

-Initialize variables
-Check user specified settings on the configuration file
-Store settings and send as I2C commands after power sequencing
-Start power up sequencing
    -Set PD6 to 0
    -Set PD7 to 0
    -RESET_N to 1
    -Wait for 100 ms
    -Set PD7 to 1
    -Reset N to 0
-if i2c_init();
    -continue
    -else try again
-Setup the camera settings registers
-if i2c_start_tx();
    -continue
    -else try again
-Send slave address
    -i2c_wr_wait(0x8E);
-char d = first letter of camera command
-i2c_wr_wait(char d);
-i2c_check_ack();
-char d = second letter of camera command
-i2c_wr_wait(char d);
-Repeat the entire first setup command is sent to the camera
-i2c_stop_tx();
-Repeat above process to send commands based on user configuration file if present
    -Send command to set camera to VGA capture
    -Send command to set camera to 1 frame per second
    -Send command to set camera to JPEG compression

Loop Forever:

Multitasking Loop One: This loop will execute every thirty seconds which is set by the delay at the beginning of the loop

-insert one second delay
-Start pulling JPEG images off of camera
-i2c_read_char(char *ch);
-place char *ch in an array to be stored on the Rabbit flash memory
-check to see if char received is an ancillary character
    -if not, continue to append incoming data to the array
    -if the received character is ancillary, break
-The complete JPEG has been received
-Get time from RTC
-Send command to DOSonCHIP to make file named with the time
-Append to the file with the JPEG data stored in flash
-Repeat until all data is written
-Wait until next frame is ready
-Pull new JPEG image off camera
-Repeat until terminated
-Power camera down
    -Set PD6 to 1
    -Set PD7 to 0
    -Disable clock crystal

End Multitasking Loop One
End Loop Forever

Shown below is a flow chart illustrating camera pseudo code.

**Camera Pseudo Code Flow Chart**

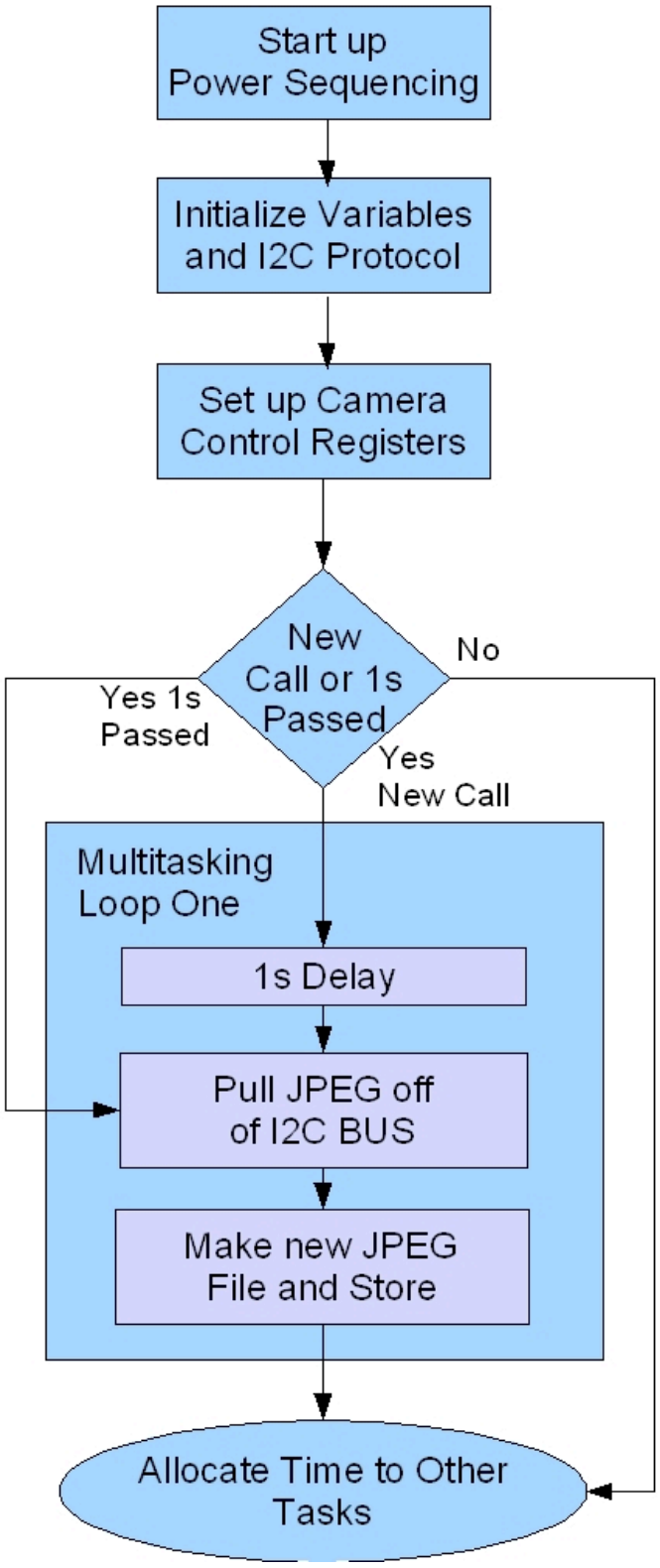



Figure 3.3.6. - Illustrates camera initialization, image downloading and storage.

### *3.3.4.5. Pseudo Code Notes*

This device has not yet been tested as a custom PCB must be milled to hold the extremely small, five mil pitch, 24-pin connector. This is only a rough outline of how we intend the code to be written for actual testing.

### *3.3.5. On Board Diagnostic System*

Data pulled from the vehicle's on board diagnostic (OBD) computer will perform two interface changes on its way to the Rabbit module. It first passes through an ELM 327 which handles converting all of the popular OBD-II interfaces (ISO 15765-4, SAE J1850 PWM, SAE J1850 VPW, ISO 9141-2, and 14230-4) data outputs to the ASCII character standard. The device is extremely handy in that it auto detects which OBD-II interface it is dealing with and handles all the grunt work associated with converting it to the RS232 standard. This will allow our device to be installed in a majority of vehicles in the world without any hardware or software modifications. Next we will route the signal through an MAX3232 converter to make the signal Rabbit friendly. Dynamic C does not come bundled with an OBD.LIB file to handle OBD-II/ELM 327 data polling so we will be writing it ourselves. The OBD.LIB will handle sending the correct PID codes for data we need from the OBD, receiving the OBD's response, and converting the response into the correct format. The PID codes we will be using are listed below.

OBD PID codes

| PID code | Desired Data | OBD Return Range | Post Conversion |
|---|---|---|---|
| OC | Engine RPM | 0-3FFF(HEX) | 1-16383rpm |
| OD | Car Speed | 0-FF(HEX) | 0-255km/h |
| 11 | Throttle Position | 0-64(HEX) | 0-100% |

Figure 3.3.7. – Outline of the OBD PID codes we will be using to communicate and pull data from the OBD system.

With the OBD.LIB library handling data fetching and conversion, the multitasking loop is left to initializing the correct port, setting the port's baud rate, sending any setting changes to the ELM 327, calling the OBD.LIB functions, and handling temporary storage for returned values.

### *3.3.5.1. OBD Data Translation*

After OBD-II data has run through the ELM 327 and MAX3232 filters it still must be converted one more time because it arrives at the Rabbit as HEX. To convert the data, we must pull one character at a time and perform a mathematical operation to arrive at the correct HEX value.

### *3.3.5.2 OBD.LIB: Pseudo Code and Flow Chart*

-define library as OBD.LIB
-declare getRPM function header

```
/*****************************************************************/
/*      int getRPM() will send the return engine RPM PID code to the OBD      */
/*      over the corresponding port, after a short delay loop will then pull  */
/*      data from the port, perform the required data conversions, and         */
/*      return the engine RPM                                                  */
/*****************************************************************/
```

-declare function 'int getRPM()'
    -initialize string rpmPID to hold the PID code for RPM fetching
    -initialize int rpmOBD to store the returned RPMs from OBD
    -set rpmPID to '01 OC\r' ASCII string required to tell the OBD to return RPM
    -send rpmPID over port B for OBD to process using RS232.LIB serBwrite()
    -short for loop to give OBD processing time
    -loop four times pulling characters from OBD using RS232.LIB serBgetc()
    converting to Dec and adding to rpmOBD
-return rpmOBD

-declare getSpeed function header

```
/*****************************************************************/
/*      int getSpeed() will send the return vehicle speed code to the OBD      */
/*      over the corresponding port, after a short delay loop will then pull  */
/*      data from the port, perform the required data conversions, and         */
/*      return the vehicle speed                                               */
/*****************************************************************/
```

-declare function 'int getSpeed()'
    -initialize string speedPID to hold the PID code for speed fetching
    -initialize int speedOBD to store the returned speed from OBD
    -set rpmPID to '01 OD\r' ASCII string required to tell the OBD to return vehicle speed
    -send speedPID over port B for OBD to process using RS232.LIB serBwrite()
    -short For-loop to give OBD processing time
    -loop two times pulling characters from OBD using RS232.LIB serBgetc() converting to Dec and adding to speedOBD
-return speedOBD

-declare getThrot function header

```
/*****************************************************************/
/*      int getThrot() will send the return vehicle throttle position code to  */
/*      the OBD over the corresponding port, after a short delay loop will     */
/*      then pull data from the port, perform the required data conversions,   */
/*      and return the vehicle throttle position                               */
/*****************************************************************/
```

-declare function 'int getThrot()'
    -initialize string throtPID to hold the PID code for throttle position fetching

-initialize int throtOBD to store the returned throttle position from OBD
-set throtPID to '01 11\r' ASCII string required to tell the OBD to return vehicle throttle position
-send throtPID over port B for OBD to process using RS232.LIB serBwrite()
-short For-loop to give OBD processing time
-loop two times pulling characters from OBD using RS232.LIB serBgetc() converting to Dec and adding to throtOBD
-return throtOBD

The three OBD.LIB functions are identical with only a few small exceptions, the type of data being pulled from OBD, the PID code, and how many return characters there are, so only one generalized flow chart is presented below.

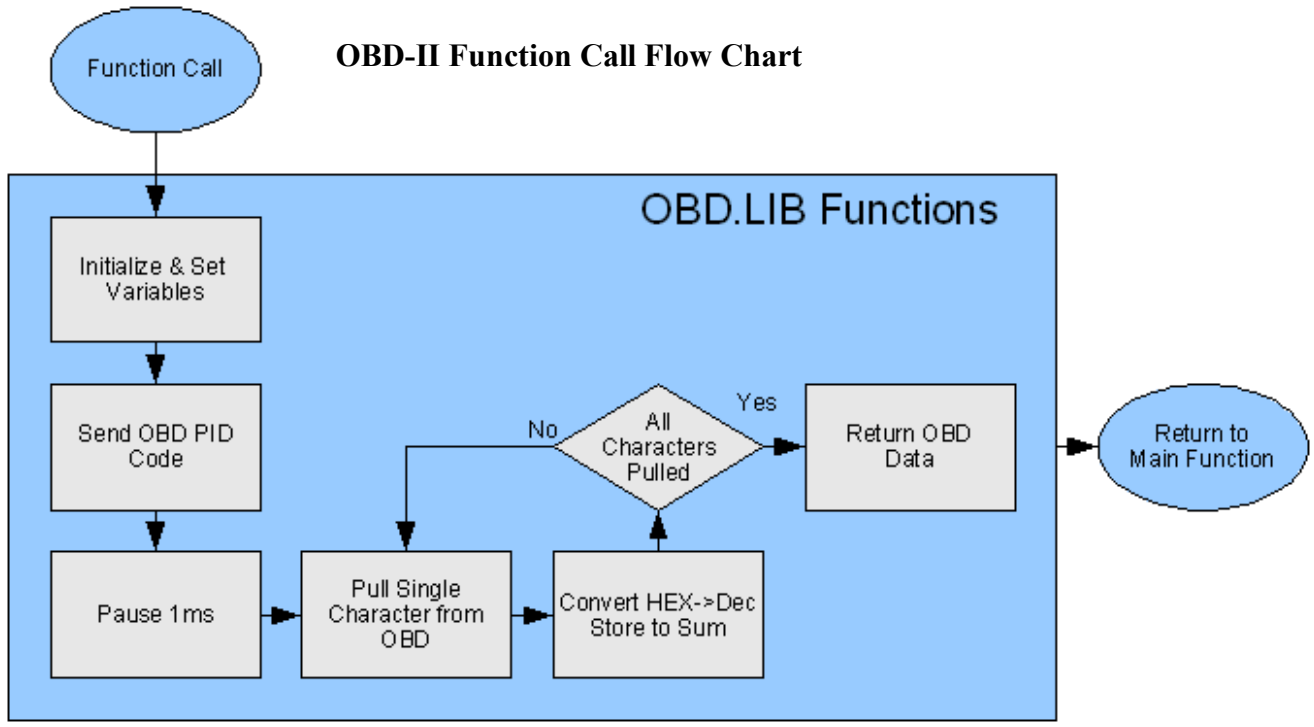



Figure 3.3.8. - Illustrates the generalized functions of the OBD.LIB library for fetching, converting, and returning data.

### *3.3.5.3. OBD: Pseudo Code and Flow Chart*

-initialize the use of OBD.LIB
-initialize integer variable vehicleRPM to store the vehicles RPM sent from OBD
-initialize integer variable vehicleSpeed to store the vehicles speed sent from OBD
-initialize integer variable vehicleThrot to store the vehicles throttle position sent from OBD

-using RS232.LIB function serBopen() set baud rate to 9600 the ELM 327 unmodified default

-set all variables to zero to prevent possibility of preexisting data at memory locations

Loop Forever:
Multitasking Loop One: This loop will execute every five hundred milliseconds which is set by the delay at the beginning of the loop

-insert five hundred millisecond delay
-call getRPM() assign return value to vehicleRPM
-call getSpeed() assign return value to vehicleSpeed
-call getThrot() assign return value to vehicleThrot
-store vehicleRPM, vehicleSpeed, and vehicleThrot into Secure Digital card

End Multitasking Loop One
End Loop Forever

Figure below illustrates the OBD pseudo code presented.

**OBD-II Data Acquisition and Storage Flow Chart**

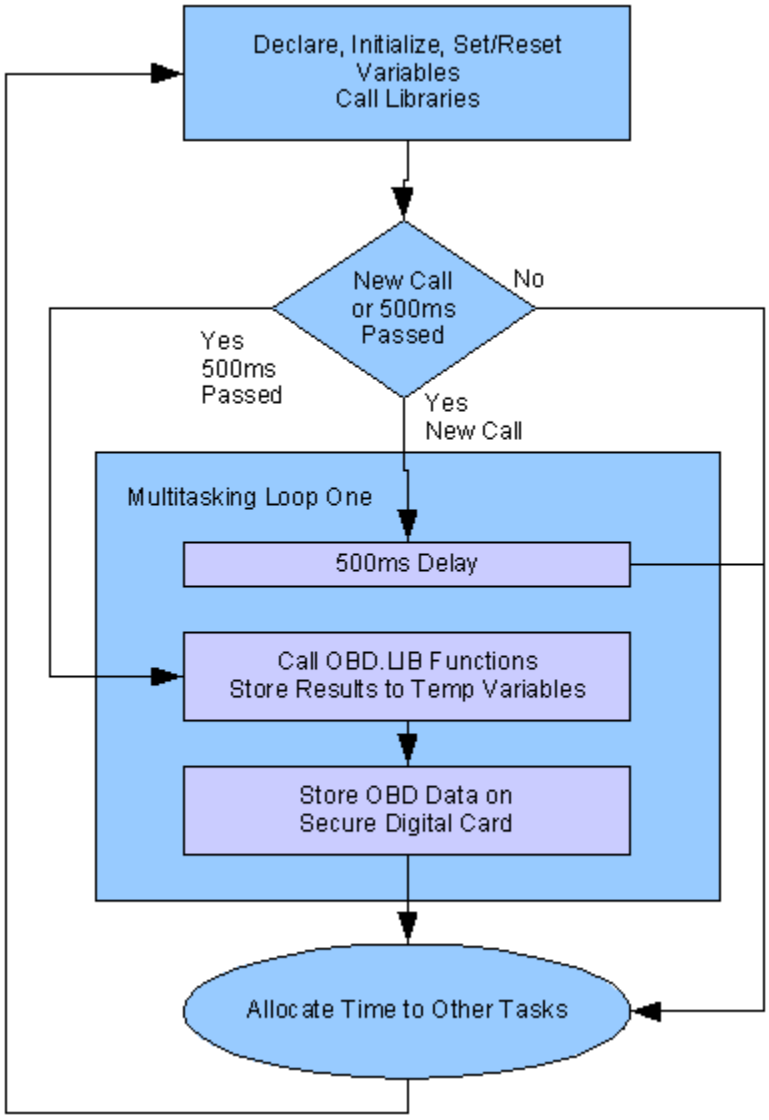



Figure 3.3.9. - Illustrates OBD data acquisition and storage.

### *3.3.5.4. Pseudo Code Notes*

The simple OBD pseudo code is a resultant of off loading all of the data fetching leg work to the OBD library. The extremely user friendly interface of the ELM 327 also plays a large roll in that its default settings work perfectly for our set up, negating the need for any type interface modification calls.

### *3.3.6. DOSonCHIP*

In each sensor's pseudo code section the data storage is termed: "store variables into Secure Digital card." Of course the storing routine is more complicated then this but to prevent redundancy, discussion of it was reserved for this section.

Communication between the Rabbit and DOSonCHIP module will be via universal asynchronous receiver/transmitter (UART), a type of serial communication. We will be using Dynamic C's RS232.LIB to handle serial communication over the appropriate port. Before we can begin data transmission for storage the baud rate must first be set by sending two sequential carriage returns. The DOSonCHIP unit will receive the first and calculate the baud rate, after receiving the second it calculates the baud rate again and compares the two. If they match the device preps itself for communication and sends a ready prompt to the Rabbit. Once the ready prompt is received the system can proceed with sending information polled from the different sensors. The DOSonCHIP commands that we will be using to send this data and their description are as follows:

Figure 3.3.10. - DOSonCHIP Commands

| Command | Description | Returns |
|---|---|---|
| ow{#1}{A:\TripLogs\Data.TXT} | Opens the existing file Data.TXT within the directory A:\TripLogs\, and sets write pointer to the end of the file for easy appending. | Which of the four possible sessions it is working in (for this case one) and any errors encountered when attempting to execute the command |
| w{#1}{string} | Appends data contained in the string to the file opened by the ow command for session one. | Datainput identifier along with any errors that occurred during execution. |
| Q{#1} | Quit and close the file session one is currently working with. | Any errors encounter during execution |

For the first build of our device sensors will have their own corresponding data files on the DOSonCHIP. This way each can load data onto the Secure Digital card during its multitasking loop. After successfully accomplishing this task we will move on to the project requirement of one single file with data logs from every device. We will do this by refraining from executing a data upload until every sensor is polled and data is available from each sensor. Then one string will be formulated with each sensor's data to be sent over the serial line to the DOSonCHIP. Below is an example of how we plan on structuring the data within the Secure Digital card.

Figure 3.3.11. - Secure Digital Data File Layout

| Time | AccelX m/s | AccelY m/s | Yaw Rate % | Speed (MPH) | RPM | GPS Coordinate |
|---|---|---|---|---|---|---|
| 0.500011574 | 0.00 | 0.00 | 0 | 0 | 1200 | 28.47N & 81.44W |
| 0.500023148 | 0.00 | 0.00 | 0 | 0 | 1225 | 28.47N & 81.44W |
| 0.500034722 | 0.00 | 0.00 | 0 | 0 | 1200 | 28.47N & 81.44W |
| 0.500046296 | 0.50 | 0.00 | 0 | 1 | 1225 | 28.47N & 81.44W |
| 0.500057870 | 0.75 | 0.00 | 0 | 2 | 1250 | 28.47N & 81.44W |
| 0.500069444 | 1.15 | 0.12 | 2 | 3 | 1250 | 28.47N & 81.44W |
| 0.500081019 | 1.50 | 0.13 | 4 | 4 | 1275 | 28.47N & 81.44W |
| 0.500092593 | 1.55 | 0.15 | 7 | 5 | 1300 | 28.47N & 81.44W |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.500104167 | 1.50 | 0.17 | 8 | 6 | 1325 | 28.47N & 81.44W |
| 0.500115741 | 1.48 | 0.19 | 10 | 7 | 1250 | 28.48N & 81.43W |

#### *3.3.6.1. Pseudo Code and Flow Chart*

-initialize sensorData a string that we will construct with all the different sensor data
-initialize sensorDataArray to store all sensor data before the array in converted into a string
-initialize errFlagArray array to store any errors encountered

Multitasking Loop One:
-set two millisecond delay so write function performed every millisecond
-set sensorDataArray element one to current time
-set sensorDataArray element two to accelerometer x-axis data
-set sensorDataArray element three to accelerometer y-axis data
-set sensorDataArray element four to yaw rate data
-set sensorDataArray element five to speed from OBD
-set sensorDataArray element six to RPM from OBD
-set sensorDataArray element seven to present GPS coordinate

-compress array sensorDataArray into a string with tab characters as delimiters

-set baud rate on available serial port D to 115200
-use RS232.LIB's serDputc to send the carriage return character twice
-use RS232.LIB's serDgetc to pull a character from the receiving line of the DOSonCHIP and store to first element of errFlagArray

IF element one of errFlagArray is ready prompt
    -send open file for writing command of desired file and location
    -receive response from DOSonCHIP store to errFlagArray element two
    -send write to file command along with sensorData to be appended at the end of the file
    -receive response from DOSonCHIP store to errFlagArray element three
    -send quit/close command
    -receive response from DOSonCHIP store to errFlagArray element four
IF error flag received
    -flash LED to denote error during write cycle
    -return write failure to application and errFlagArray
End Multitasking Loop One

Figure below illustrates DOSonCHIP pseudo code presented.

**DOSonCHIP Communications and File Writing Flow Chart**

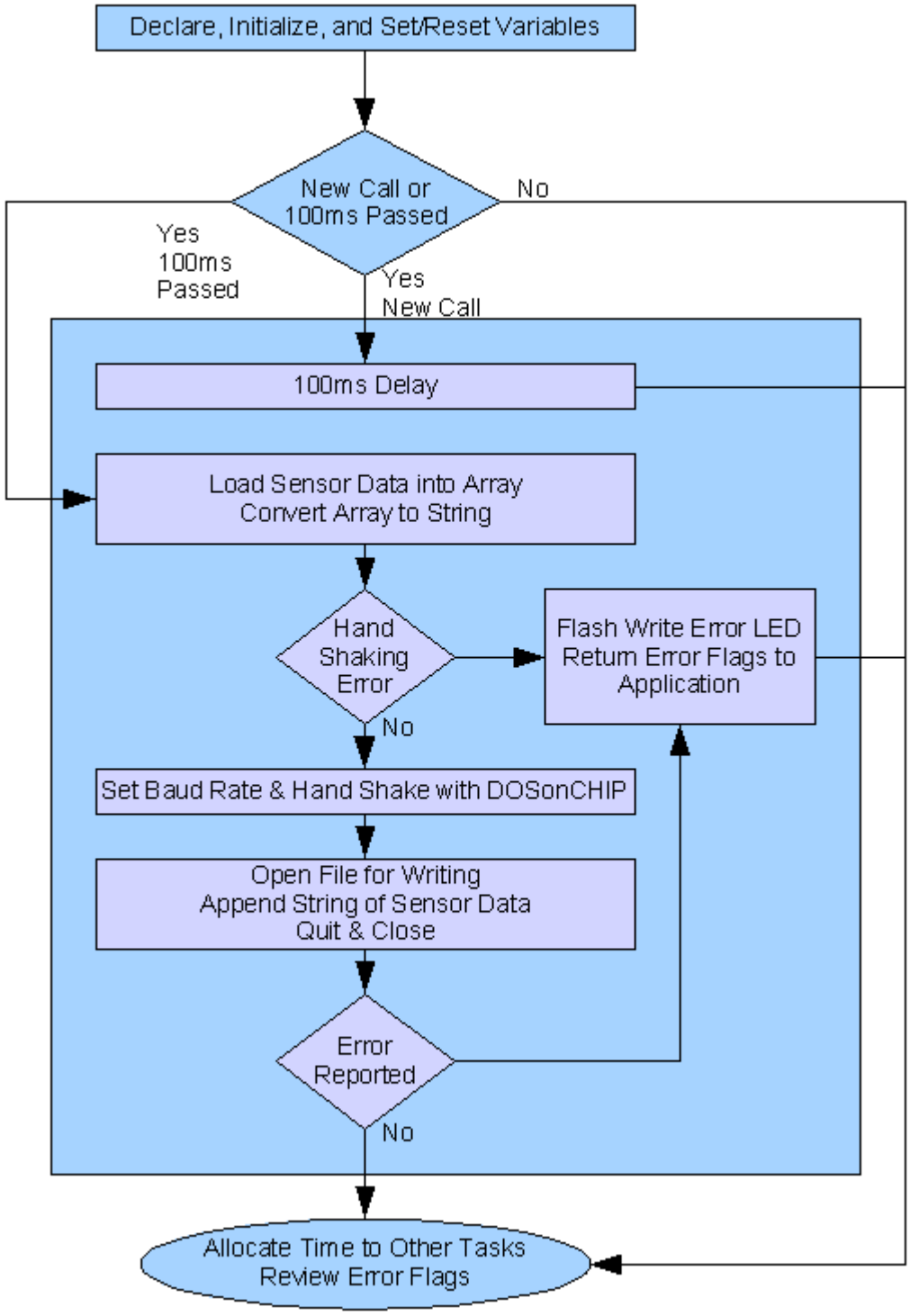



Figure 3.3.12. - Illustrates data conversion, DOSonCHIP error checking, and file manipulation.

#### *3.3.6.2. Pseudo Code Notes*

The pseudo code outlined above will be for the final build's data file format, which is illustrated in the Secure Digital Data file table. The format will be achieved by the routine that converts sensor data in an array to a string delimited with tab characters. When this is written to a text file within the DOSonCHIP it will create a table like format that will be easy to read. During initial device testing all sensors will have their own single file. This way if a device is not working as we wished and corrupting its data file, it will not propagate into other sensor's data files. The only difference in the pseudo code and flow chart for the testing storage routine and the final storage routine is the exclusion of the array compression function. This code will be called by all sensor multitasking loops.

## 3.4. Software Routines

### *3.4.1. Boot Sequence*

When power is applied to the Rabbit it will need to begin a sensor and DOSonCHIP test sequence to verify all devices are online and communicating properly before the master program begins. All of our data will be stored on our non-volatile Secure Digital card so no effort will need to be put into pulling data from battery backed ram after cold boot. First to be initialized and tested and will be the DOSonCHIP memory system, the most critical external device. After communication protocols like the ones outlined in the DOSonCHIP software interface section are completed, the system will then conduct an echo ping test. Echo pinging involves setting the device to echo back all received data. After data sent is matched with the data received a new log file using a generic name matched with the Rabbit's data and time stamp appended to the end will be created in the appropriate directory. Echo mode will then be shut off and initialization and testing can move to the next device. If the DOSonCHIP fails the boot tests and recovery attempts the system is put in a critical state because no data can be logged, in effect making the entire device useless. If this happens the system will signal a red LED to be flashed repeatedly and remain idle, indicating a critical error. Next to be initialized will be the sensor systems; each will have its communication protocols initialized (outlined in each devices software interface section) and be echo pinged to ensure operation. If all attempts to remedy a failed sensor boot (outlined in boot sequence pseudo code) are unsuccessful an error log will be created on the Secure Digital card saving system information and details available on the failed boot, the system will flash a yellow LED to indicate an unsuccessful boot sequence, the multitasking loop for the failed device will be disabled and the program will begin down a sensor. The entire boot sequence will be housed in a multitasking loop set to run once on initialization, it then will remain dormant until the system is reset.

#### *3.4.1.1. Pseudo Code and Flow Chart*

/**********Accelerometer**********/
-initialize array ACCarray with "Analog Devices" acceleration reference chart
/**********End Accelerometer**********/

/**********Yaw Rate Gyro**********/
-initialize the use of SPI.LIB
-initialize pointer safAddress pointed at a safe address location to store received data
-initialize array YRGarray with “Analog Devices” yaw rate gyro reference chart for any conversions necessary
/**********End Yaw Rate Gyro**********/

/**********GPS**********/
-initialize the use of GPS.LIB
-define a max sentence size of 100 characters to validate sentence length, as stated above a sentence should never run over eighty-two characters, if it does the system must be reset because an error was encountered.
-initialize arrays to store the days of the week to convert numerical data from gps_get_utc into character based date
-initialize arrays to store the months of the year to convert numerical data from gps_get_utc into character based date
-initialize a new variable using GPS.LIB's GPSPosition structure called curPos to store current positions gps_get_positions pulls from GPS
-initialize a new variable using GPS.LIB's tm structure called curTime to store current time returned from gps_get_utc call to GPS
-initialize string variable sentence using max sentence size
-initialize variable charChecker do analyze individual characters in string sentence to check for a carriage return or new line
-initialize sting curDirecton to store current direction for latitude and longitude
-initialize noCord to store flag if there was no coordinate fetched yet
-initialize variable stringPosPointer to point at which location in the sentence array is of concern
-initialize variable curLong to store the current longitude pulled from GPS
-initialize variable curLat to store the current latitude pulled from GPS
-initialize array curTimeStampe to store the current time stamp from the GPS
/**********END GPS**********/

/**********OBD**********/
-initialize the use of OBD.LIB
/**********END OBD**********/

/**********Camera**********/
-initialize the use of I2C.LIB
/**********END Camera**********/

/**********Universal**********/
-initialize character bootTestc to store return values from devices to confirm proper functioning

-initialize string bootTests to store return values from devices to confirm proper functioning
-initialize integer bootTesti to store return values from devices to confirm proper functioning
-initialize integer breakTest to store flag for repeated attempts to initialize device
-initialize string saveName to store the name of the current data logging file being used
-initialize integer deviceOnline to store flag confirming DOSonCHIP working correctly
-set all initialized variables to zero
/**********End Universal**********/

/**********DOSonCHIP Boot Sequence**********/
Do while breakTest is less then or equal to four
-set baud rate on available serial port D
-use RS232.LIB's serDputc to send the carriage return character twice
-use RS232.LIB's serDgetc to pull a character from the receiving line of the DOSonCHIP and store to bootTestc
IF bootTestc is equal to ready prompt
-use RS232.LIB's serDputs('e1') initializing DOSonCHIP echo mode to confirm data sent matches data received
    -use RS232.LIB's serDputc('T') sending single character T to DOSonCHIP
    -use RS232.LIB's serDgetc() storing result to bootTestc
    IF bootTestc equals 'T' DOSonCHIP confirmed proceed with program
        -create string containing 'datalogger' appended with the time and date from Rabbit, store in saveName, and create the file on the DOSonCHIP
        -use RS232.LIB's serDputs('e0') turning off DOSonCHIP echo mode
        -set deviceOnline to one confirming DOSonCHIP online
        -set breakTest equal to four breaking out of test/initialization routine
-increment breakTest by one
End do while loop

IF deviceOnline equals zero, the DOSonCHIP is not online critical halt
    -flash red LED
    -set all multitasking loops not to run
    -halt program
Else reset testing variables for next device
    -set breakTest to zero
    -set bootTestc to zero
    -set bootTesti to zero
    -set bootTests to zero
    -set deviceOnline to zero
/**********End DOSonCHIP Boot Sequence**********/

/**********Accelerometer Boot Sequence**********/
Do while breakTest is less then or equal to four

-initialize quadrature decoder and port F to receive PWM data
-reset channel one's high counter for Port F using R3000.LIB qd_zero ensuring clean slate before polling begins

-set bootTesti equal to quadrature decoded data polled from port F using R3000.LIB qd_read

IF bootTesti is less then zero an error in polling has occurred
    -reset channel one's high counter which will reinitialize it back onto a valid polling frequency using qd_zero
    -set bootTesti equal to quadrature decoded data polled from port F using qd_read
IF bootTesti is equal to zero or one the accelerometer is being polled correctly and is inifitalize
        -set breakTest equal to four breaking out of test/initialization routine
        -set deviceOnline equal to one indicating sucessful device boot

-increment breakTest by one
End do while loop

IF deviceOnline equals zero, the Accelerometer is not online
    -flash yellow LED
    -set accelerometer multitasking loop not to run
    -create error report file on DOSonCHIP reporting accelerometer boot failure

/* Reset universal variables for next device*/
-reset channel one's high counter using qd_zero
-set breakTest to zero
-set bootTestc to zero
-set bootTesti to zero
-set bootTests to zero
-set deviceOnline to zero
/**********End Accelerometer Boot Sequence**********/

/**********Yaw Rate Gyro Boot Sequence**********/
Do while breakTest is less then or equal to four
-set clocked serial port E to be used for SPI communication
-set system for SPI master mode
-using a separate line on the yaw rate gyro discussed in the hardware section send logical high to begin yaw rate gyro self-test function and warm up the unit mechanically, repeat twice to ensure system mechanically ready
-using write/read function send pointer safAddress, zero, and zero to initialize gyroscope functionality and has received data saved to memory location safAddress
-compute the size of data located at safAddress and store to bootTesti

IF bootTesti is valid data, Yaw Rate Gyro is initialized

-set breakTest equal to four breaking out of test/initialization routine
-set deviceOnline equal to one telling system sucesffuly booted

-increment breakTest by one
End do while loop

IF deviceOnline equals zero, the Yaw Rate Gyro is not online
-flash yellow LED
-set yaw rate gyro multitasking loop not to run
-create error report file on DOSonCHIP reporting yaw rate gyro boot failure

/* Reset universal variables for next device*/
-set breakTest to zero
-set bootTestc to zero
-set bootTesti to zero
-set bootTests to zero
-set deviceOnline to zero
/**********End Yaw Rate Gyro Boot Sequence**********/

/**********GPS Boot Sequence**********/
-using RS232.LIB function serCopen() set baud rate to 4800
-set noCord to one so that the follow loop continues to execute until a coordinate is fetched

Loop as long as noCord remains equal to one
IF bootTesti is equal greater than or equal to five hundred GPS has failed to obtain lock
-set noCord to zero breaking loop without deviceOnline being set
-assign character fetched with RS232.LIB's serCgetc function that pulls a single character from the serial input line to charChecker

IF charChecker is equal to carriage return or newline GPS initalized and functioning properly
-set value in sentence at stringPosPoint to zero for subsequent coordinate pulling
-set stringPosPoint to zero for subsequent coordinate pulling
-set deviceOnline equal to one
IF call of gps_get_utc to fill curTime executes without error proceed with storing data
-store current time stamp from curTime into curTimeStamp
-using days of week and months of year array reference arrays convert data in curTimeStamp from numerical to characters
-use GPS time to set the Rabbit's internal timer
-set noCord to zero to break testing loop
ELSE IF charChecker is valid date greater then zero
-store data in charChecker in sentence at the location of stringPosPointer

```
        -increment stringPosPointer to point at the next location in sentence
        IF stringPosPointer equals max sentence
                -error has occurred reset stringPosPoint back to zero so can reset data
pulling
-increment bootTesti
End onCord verification Loop

IF deviceOnline equals zero, the GPS is not online
        -flash yellow LED
        -set GPS multitasking loop not to run
        -create error report file on DOSonCHIP reporting GPS boot failure

/* Reset universal variables for next device*/
-set breakTest to zero
-set bootTestc to zero
-set bootTesti to zero
-set bootTests to zero
-set deviceOnline to zero
/**********************End GPS Boot Sequence**********************/



/**********************OBD Boot Sequence**********************/

Do while breakTest is less than or equal to four
-using RS232.LIB function serBopen() set baud rate to 9600 the ELM 327 unmodified
default

        -call getRPM() assign return value to bootTesti
        IF bootTesti is valid data
                -set breakTest to one breaking loop
                -set deviceOnline equal to one showing OBD working properly
        Else continue boot attempts
                -increment breakTest
End do while loop


IF deviceOnline equals zero, the OBD is not online
        -flash yellow LED
        -set OBD multitasking loop not to run
        -create error report file on DOSonCHIP reporting OBD boot failure

/* Reset universal variables for next device*/
-set breakTest to zero
-set bootTestc to zero
-set bootTesti to zero
-set bootTests to zero
```

-set deviceOnline to zero
/**********End OBD Boot Sequence**********/

/**********Camera Boot Sequence**********/
Do while breakTest is less than or equal to four

-Check user specified settings on the config file
-Store settings and send as I2C commands after power sequencing
-Start power up sequencing
    -Set PD6 to 0
    -Set PD7 to 0
    -RESET_N to 1
    -Wait for 100 ms
    -Set PD7 to 1
    -Reset N to 0
-if i2c_init();
    -continue
    -else try again
-Setup the camera settings registers
-if i2c_start_tx();
    -continue
    -else try again
-Send slave address
    -i2c_wr_wait(0x8E);

-char d = first letter of camera command
-i2c_wr_wait(char d);
-i2c_check_ack();
-char d = second letter of camera command
-i2c_wr_wait(char d);
-Repeat the entire first setup command is sent to the camera
-i2c_stop_tx();
-Repeat above process to send commands based on user config file if present
    -Send command to set camera to VGA capture
    -Send command to set camera to 1 frame per second
    -Send command to set camera to JPEG compression

-call i2c_read_char(char *ch) to download picture from camera
-place char *ch in bootTests to temporaly store image data
-check to see if char received is an ancillary character
    IF no ancillary characters returned
        -continue to append incoming data to the array
        -after completion set deviceOnline to one signaling successful boot
        -set breakTest equal to four

-increment breakTest for another iteration of boot sequence if first failed and corrupted data returned

End do while loop

IF deviceOnline equals zero, the camera is not online
    -flash yellow LED
    -set camera multitasking loop not to run
    -create error report file on DOSonCHIP reporting camera boot failure

/* Reset universal variables for next device*/
-set breakTest to zero
-set bootTestc to zero
-set bootTesti to zero
-set bootTests to zero
-set deviceOnline to zero
/**********End Camera Boot Sequence**********/

-Proceed with main program

Below is a flow chart of the pseudo code:

**Boot Sequencing Flow Chart**

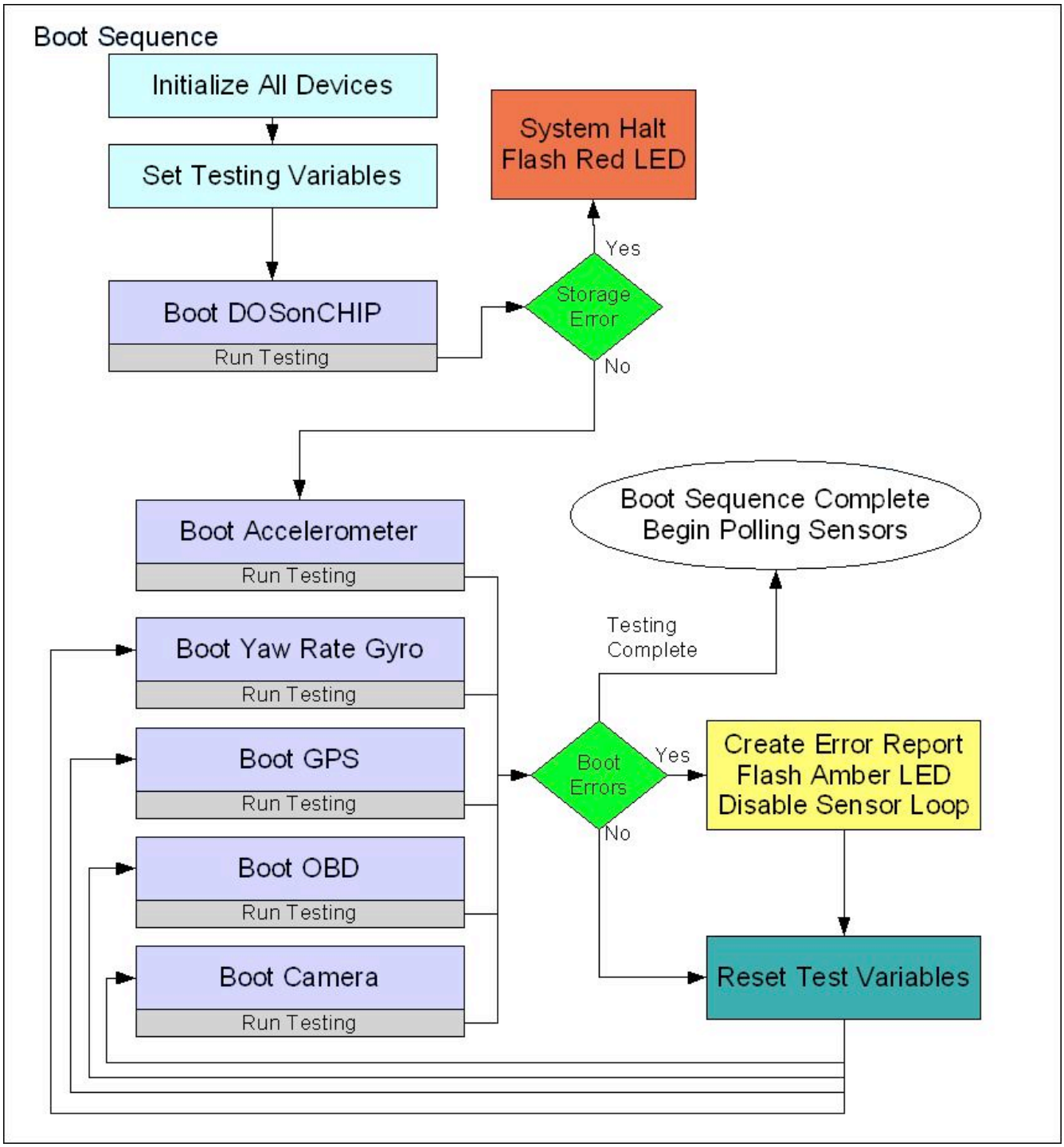



Figure 3.4.1. - Illustrates boot sequencing discussed in pseudo code above.

### *3.4.2. Emergency Mode*

An optional addition to the project is an emergency mode, an idea touched on in the multitasking section. The concept is that if the sensors pick up on a sudden or erratic change in driving behavior the timing delays for the different sensors would be reduced. This would be accomplished with the addition of emergency flag variables in the multitasking loops for the accelerometer, yaw rate gyroscope, and OBD. A sensor's emergency flag would be activated when its return data surpasses a defined threshold. So if the accelerometer reads a sudden deceleration, or the yaw rate gyroscope reports an erratic jerk, or the OBD shows throttle position at 100%, its associated flag would be assigned the value one. Within the emergency mode multitasking loop would be a wait for emergency flag statement, so when the emergency flag is given the value one the emergency mode multitasking loop would immediately begin execution. Within the emergency mode multitasking loop, post wait statement, would be the reassigning of the critical sensors loop delays, below which would be another wait for statement with a fifteen second delay. For fifteen seconds the system will run at its maximum processing power to bring in as much critical data as possible. Allowing for later data examination to present a clear picture of what happened to cause an accident or a near miss. After fifteen seconds the system would come back into the emergency mode multitasking loop where the critical sensor loop delays and emergency flag would be set back to their defaults and emergency mode would be put back into standby.

#### *3.4.2.1. Pseudo Code and Flow Chart*

-initialize integer variable emergFlag to store the emergency mode flag
-initialize integer variable delayAccel to store the multitasking loop delay for the accelerometer
-initialize integer variable delayYaw to store the multitasking loop delay for the yaw rate gyroscope
-initialize integer variable delayOBD to store the multitasking loop delay for the OBD
-initialize integer variable delayDOS to store the multitasking loop delay for the DOSonCHIP
-initialize integer variable delayCam to store the multitasking loop delay for the camera

-set all variables to zero to prevent possibility of preexisting data at memory locations

Loop Forever:
Multitasking Loop One: This loop will execute immediately after emergFlag is set to one

-insert wait for command using emergFlag as a parameter
-set delayAccel to 50 representing a 50ms re-poll delay, twice as fast as the default
-set delayYaw to 50 representing a 50ms re-poll delay, twice as fast as the default
-set delayOBD to 250 representing a 250ms re-poll delay, twice as fast as the default
-set delayDOS to 50 representing a 50ms re-poll delay, twice as fast as the default
-set delayCam to 500 representing a 500ms re-poll delay, twice as fast as the default

-set 15 second delay
-set delayAccel to 100 representing a 50ms re-poll delay, twice as fast as the default
-set delayYaw to 100 representing a 50ms re-poll delay, twice as fast as the default
-set delayOBD to 500 representing a 250ms re-poll delay, twice as fast as the default
-set delayDos to 100 representing a 250ms re-poll delay, twice as fast as the default
-set delayCam to 1000 representing a 1s re-poll delay, twice as fast as the default
-set emergFlag to zero completing the system transition to normal mode

End Multitasking Loop One
End Loop Forever
The figure below illustrates the emergency mode pseudo code.

**Program Flow Control Including Emergency Mode**

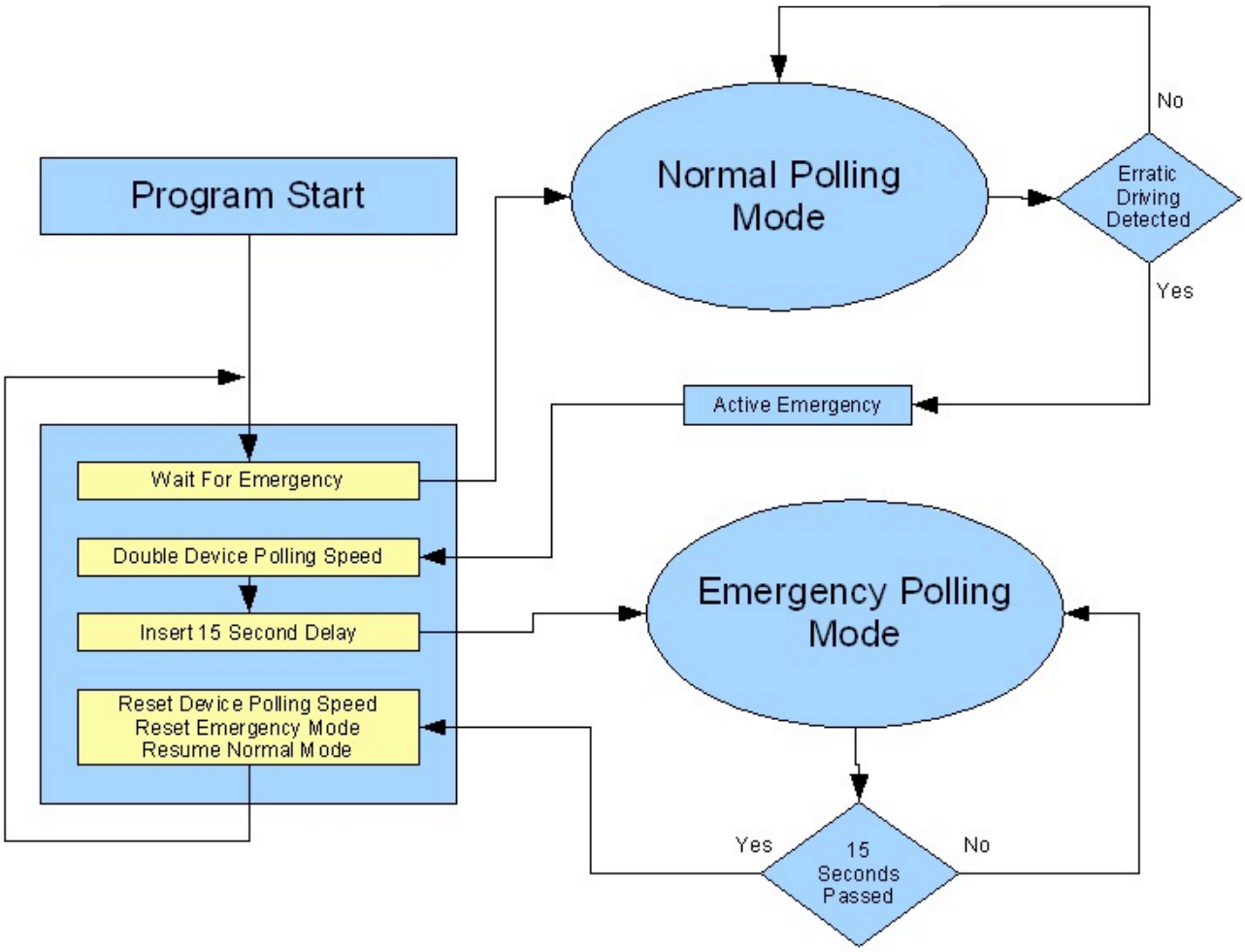



Figure 3.4.2. - Illustrates program flow as system switches in and out of emergency mode.

### *3.4.3. Shutdown Sequence:*

Preliminary designs for system shutdown will quite simply be cutting off power to the Rabbit and all of our external sensors. The Rabbit 3200 module lacks any built in soft shutdown capability so without any manipulation it will be shutdown in the middle of code execution with no warning. As of now there are no plans to do anything more then let the Rabbit hard shut down. The only concern resultant of a hard shut down is the effect on our data file on the Secure Digital card if power is cut amidst a write call. This will be heavily tested to ensure that our saved data will not be entirely corrupted. If only the last entry is affected we can easily delete it and lose that last tenth of a second of data. But if the entire file is lost it would be disastrous in the scope of the project.

If testing proves hard shutdowns are too risky a method of soft shutdown will have to be found. One possibility would be a battery backup unit for the Rabbit to sustain power for a few seconds after initial power is cut. This would allow it time to detect it is no longer able to contact any of its external sensors, at which point it would stop its main program execution in anticipation of shutdown. Another option would be to have the Rabbit take special note of the vehicles RPMs. If they dip below the point where the vehicle is likely not moving, the Rabbit could freeze Secure Digital access until RPMs rose. This would ensure that if a shut down does occur; it would at least not be in the process of writing to the Secure Digital card.

### *3.4.4. Master System Outline and Flow Chart*

As stated in section 3.3 the pseudo code presented for each sensor is its initial test code. Once operation is individually verified all sensors will be incorporated into a single device. The pseudo code manipulations to make this possible will involve moving all initializations and testing to the boot sequence outlined in section 3.4.1. This boot sequence will handle boot up and verify the different devices are operating at intended. Once the boot sequence has completed, program flow will go to polling and storage procedures. While the system is polling the different sensors the Rabbit module will be monitoring all the multitasking loops, switching processing power to each sensor when it is that sensor's turn to poll. After every sensor has completed a polling cycle and has data ready for storage, a master cycle has been completed. Processing will then divert to storing the gathered data to the DOSonCHIP, after which the system will start the entire process over again. In an attempt to refrain from repeatedly presenting the same code this section does not include pseudo code. All pseudo code and been presented in its individual sections. What will be presented is a master flow chart illustrating what a majority of the software sections have been building up to, the big picture for the entire program.

**Master Program Flow Control**

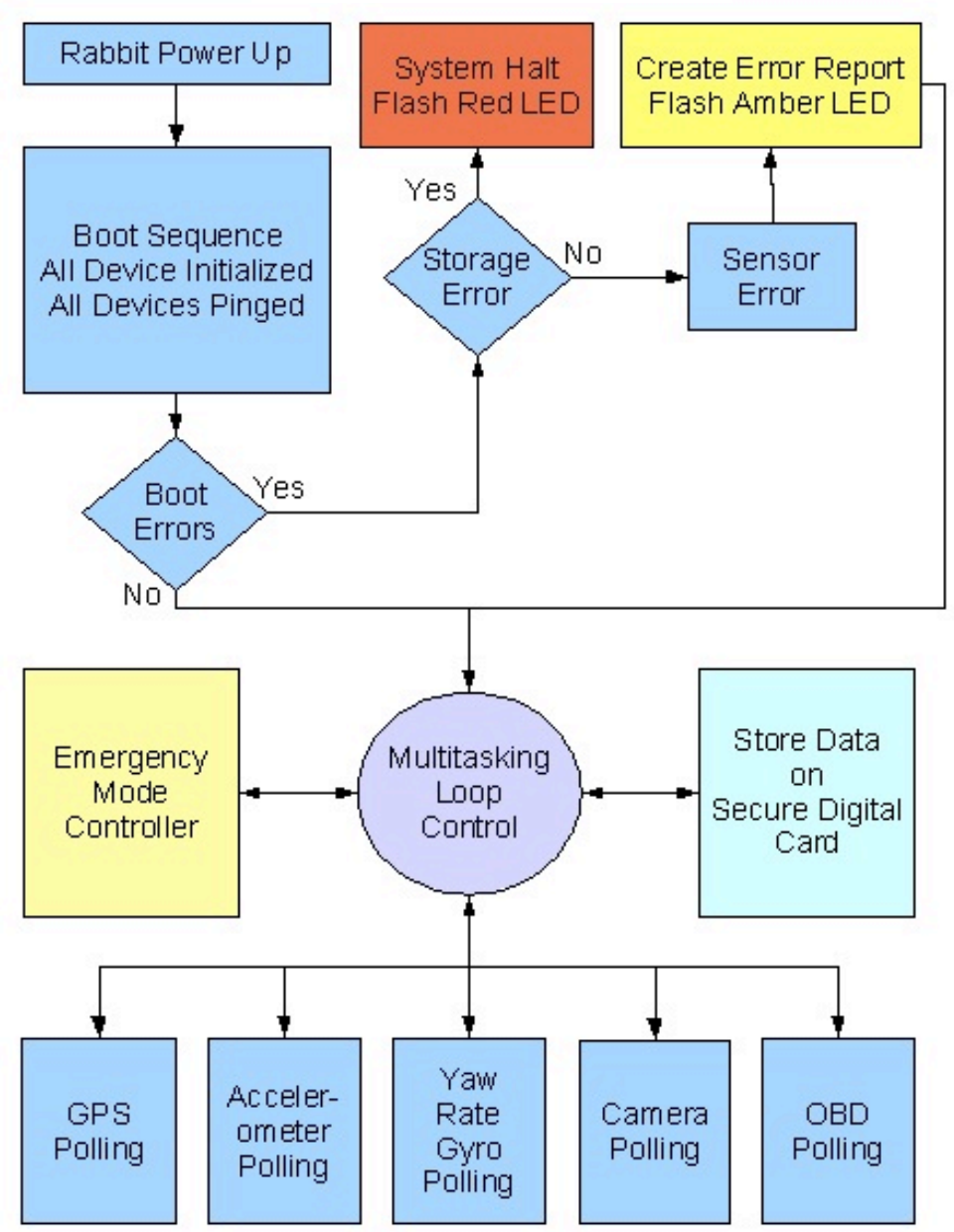



Figure 3.4.3. - Illustrates the big picture overview for the entire software suite.

# 4. Communication Protocols

Almost every peripheral device we chose to use communicates with a different protocol, therefore are required to use almost the entire gamut of communications protocols in order to gather all of the required data. In addition to the different pin configurations and power requirements for each protocol, there is also a vastly different command set and structure used to communicate with the devices. The Rabbit microcontroller we are using has built in capabilities that help use each protocol, but some of these are limited to certain ports on certain output pins to work properly. We have carefully chosen our devices to work within the capabilities of the Rabbit to ensure

that we have enough of the correct data ports to use. Shown below is a table describing the unique needs of all of the peripheral devices hooked up to our microcontroller.

Figure 4.1.1 All Components Data Lines Specifications

| Component | Protocol | Type | Voltage Swing (V) | Voltage Leveling Required? |
|---|---|---|---|---|
| GPS | EIA232 | Serial UART | 0 - 5 | Yes |
| DOSonCHIP | EIA232 | Serial UART | 0 - 3.3 | No |
| | SPI | Clocked Serial | 0 - 3.3 | No |
| Accelerometer | PWM | Duty Cycle | 0 - 5 | Yes |
| Yaw Rate Gyro | SPI | Clocked Serial | 0 - 5 | Yes |
| OBDII Interpreter | EIA232 | Serial UART | 0 - 5 | Yes |
| Camera | $I^2C$ | Clocked Serial | 0 - 2.8 | No |
| | 8 Bit Bus | Parallel | 0 - 2.8 | No |

All Components Data Lines Specs

The components that we have selected each have different communications needs that will need to be filled by the microcontroller's I/O ports. Many of these ports have been set aside by the manufacturer as dedicated to certain communications protocols, other pins have been left available as general purpose I/O pins. Show below is a diagram showing the configuration that we have chosen to meet all of the peripheral devices protocols by using the ports on the Rabbit that will work the best.

**Mircocontoller I/O port to device interconnections.**

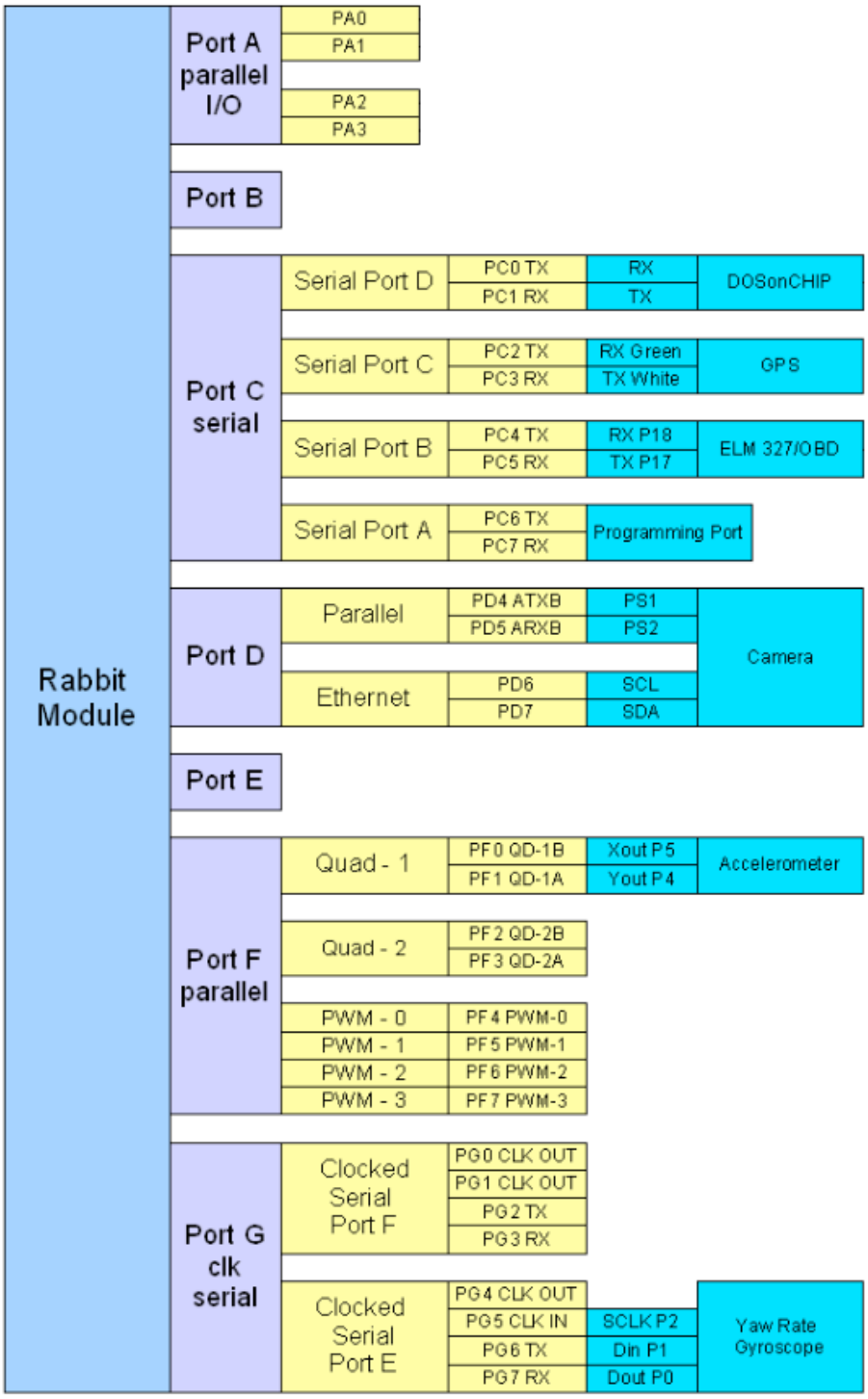



Figure 4.1.2. – Mircocontoller I/O ports to devices interconnections.

## 4.1. Serial

### *4.1.1. RS232*

The RS232 protocol for serial data communications never became standardized. The protocol has evolved to adapt to new technology so much that there are many different variants and flavors of RS232 in use today. Because the RS232 protocol's data transmission has never been explicitly defined, ambiguities continue to cause confusion and ambiguities. Technically speaking, most references to RS232 actually are speaking of EIA232. The positive voltages are not well defined in the standards in order to allow flexibility. Some of our devices need 3.3V for a positive to register, and some need 5V. Our solution to the voltage level problem is to utilize the MAX3232 voltage-leveling chip from Maxim. This inexpensive integrated circuit takes care of converting levels between the Tx and Rx pins so that we will never damage a serial port or get garbage data due to a voltage level mismatch. We will actually use these leveling chips for other protocols that have signal voltages above the 3.3V that the Rabbit microcontroller needs to communicate with. The Rabbit does not like voltages on its IO pins higher than 3.3V.

The most popular protocol for communications with our components is serial UART, or universal asynchronous receiver/transmitter. This protocol uses two wires for serial communication and requires that both devices be independently set to the same baud rate, number of data bits, stop bits, and parity. Our OBDII interpreter and storage controller both use two additional lines, CTS and RTS, for hardware flow control. The ready to sent (RTS) line is used to get attention when a peripheral device has data to send. When the host device is ready to receive the data, it brings the clear to send (CTS) line low. This handshaking ensures that data will flow smoothly across the bus. After a successful handshake the devices may begin transmitting data at the agreed predetermined bit-rate.

Figure 4.1.3. - Serial Components

| Component | Baud Rate | Hardware Flow Control? |
|---|---|---|
| GPS | 9,600 | No |
| DOSonCHIP | Auto Detect | Yes |
| OBDII Interpreter | 38,400 or 9,600 | Yes |

Serial Components Transmission Speed

## 4.2 Clocked Serial

### *4.2.1. $I^2C$*

We have several devices that communicate with clocked serial protocols that are a bit more complicated than serial UART. The two protocols used are SPI (serial peripheral interface bus) and $I^2C$ (inter-integrated circuit). These use a clock to synchronize and time data transactions. The $I^2C$ bus uses only two wires, a SCL clock wire, and a SDA data wire in a master/slave arrangement. $I^2C$ is also an addressed bus, where each device has a unique address. The master provides the clock and initiates the communication while the slave monitors the data line for data prefaced by the device's seven-bit address. In this case the Rabbit will act as the master and the camera will be the slave. To communicate the Rabbit will first establish and stabilize the clock on the SCL line at 400 kHz. Then it will send out a start bit on the data line followed by the seven-bit address of

the slave device. The next bit sent will determine if the slave is designated to send or to receive data. By sending a 1 the master is saying that it will be sending out one byte of data for the addressed slave. On the other hand, a 0 means that it is expecting the slave to send data back. In our circuit the camera module has been designated the permanent seven bit address 0x47, it can be designated read or write by appending a zero or a one to the end of the address. All of these data bits are sent on the high clock pulse, which allows the camera to temporarily ask for a short pause by holding the SCL clock line active low. The Rabbit will wait a short while for the clock to go high, if it is held low for too long a timeout error will be reported and no data sent. The $I^2C$ protocol is simple and only requires two data wires be run to the device, but it is limited to 400 kbps, which limits its usefulness to us when a higher bandwidth is needed to increase the frame rate above 1 frame per send at VGA, or 3 frames per second at QVGA.

### *4.2.2. SPI*

The SPI protocol is used by our yaw rate gyro and is an optional protocol for the DOSonCHIP storage controller. In order to save one clocked serial port on our microcontroller we will be using serial UART to communicate with the DOSonCHIP. SPI differs from $I^2C$ in that SPI allows full duplex communications due to its one extra data line. To ability send and received data simultaneously is at the expense of one extra required wire, but allows for much faster serial data transfer. The yaw rate will provide the SCLK clock signal that will control the rate at which bits will be sent over the two data lines. We will need to determine by testing the device if a fourth wire is required for a CS chip select to initiate data transfer, it may not be needed as it will by the only device on the bus. Communications begin by checking the value of the DIN data in pin on the first falling edge of SCLK. If it is a logic 1 the gyro will be put in a state to accept new values for its control registers. These values are clocked in on the rising edge of SCLK over the next 12 clocks. At the same time, yaw rate data is being clocked out on the falling edge of SLCK on the DOUT data out pin for the next 16 clock pulses. If the value at DIN had been a logic 0 the control registers would have remained unchanged and the yaw rate information clocked out just the same.

## 4.3. PWM

Pulse width modulation (PWM) is only used by our dual-axis accelerometer. It is a very simple protocol that alters the duty cycle of the signal to communicate its information. It allows for only one-way communication to take place. The data can be extracted from the duty cycle by computing how much higher or lower than a fifty percent duty cycle is present. If a duty cycle is at sixty percent there is a ten percent change in the positive axis. We can then obtain acceleration with the formula 1g = 30% change. So a ten percent change would indicate a acceleration of 1/3 g in the positive axis direction. We will use the built in pulse width detection capabilities of the Rabbit microcontroller to measure the two PWM signals from the accelerometer X and Y axes. Below is an example of data sent using PWM. Pay attention to the duty cycles as that is the useful data, a higher duty cycle means a longer pulse. The example data given is normal to the force of earth's gravity. The vertical axis would yield a duty cycle equal to 1g, obviously.

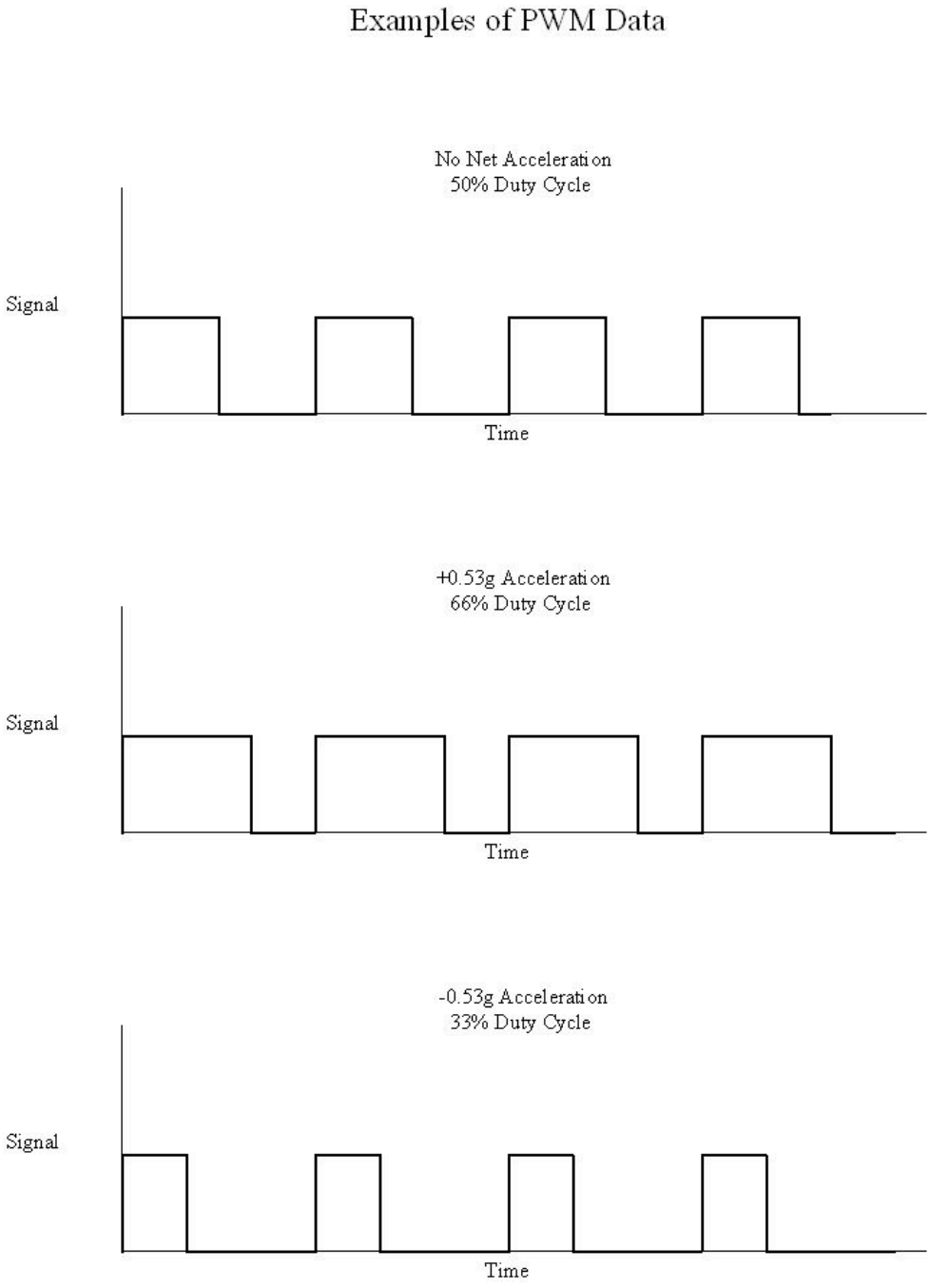



Figure 4.1.4. – Comparison timing charts showing Pulse Width Modulated signals.

# 5. Device Design Considerations

When designing any device there are many different considerations that are not explicitly set out in the guidelines that you figure out along the way. Many times you don't figure these out until after the final circuit is built and you realize you forgot something as silly as a status indicator to signify the device is working. Since Dr. Papelis did not specify much more than the basic functionality, we had to think ahead to make sure the device would be well designed enough to be usable by people other than the soon-to-be engineers that built it. Usually that starts with a user interface.

## 5.1. User Interface

The user interface is one of the most important aspects of the hardware, as eventually the device will have a user. The user will need to know critical information about the status of the device. The user will also need to be reminded if and when something is not set up properly. We will start from the simplest problems and get into progressively more complex problems while maintaining the same level of simplicity with the user interface.

The most common and likely user interface is a series of light emitting diodes (LEDs) that are labeled and flash at different rates to indicated different statuses. Since we are working from most simple to more complex we start with a simple power indicator. Once the device is connected to the OBDII bus, which is where the unit pulls power from, it will indicate it whether or not it has power by illuminating the power LED. If the car's ignition is switched to off, the device should not have power. If for some reason the device does have power but the car's ignition is off the device will indicate it is in a low-power mode by blinking quickly every four seconds. If the device has power and the car is running the device will show a solid power light.

Once the device has power it can begin to power up its subsystems, of primary importance is the data storage device and the data storage media. First the device will check to make sure the storage card is present and able to be written to. If the storage card is not present the storage media LED will flash rapidly, about 4 times per second. If the storage media is inserted but not writable, the storage media indicator LED will flash rapidly, about 6 times per second. If for some reason the card is present but is full, or will shortly be full (less than 50MB available), the storage media LED will flash at about 2 times per second. This test is not just run once at start-up, the device will continue to make sure the media is not full throughout operation and indicate low storage capacity when the situation arises. This way the user can glance at the system to make sure everything is working properly and be confident it will continue to work for a while after the user checks the status.

After the device has power and is able to write to the storage media, it will try and acquire a GPS lock, first the GPS unit must be connected. If the GPS unit is not connected the GPS status LED will flash rapidly, about three times per second. If the GPS unit is connected but is still waiting to acquire a GPS lock, the LED will flash slowly, about once per second. When the device acquires a GPS lock from one or more satellites the GPS indicator light will remain solid.

Next is to make sure the OBDII is communicating properly. We do not really need to make sure the cable is connected, without it being connected, the device will not power at all. We do need to make sure the device can find the correct OBDII protocol and

retrieve data. Once the ELM chip has determined the proper communication method and begins polling for RPM and throttle position data the light will remain solid. If for some reason the data is unavailable or is invalid, the OBDII LED indicator will flash with different codes, four short bursts followed by a pause if the RPM is not being properly read, and two short bursts if the throttle position cannot be read. If neither the RPM nor the throttle position data is readable, the OBDII LED indicator will flash constantly about four times per second to indicate a problem.

The camera is the next device on the priority list and is next on the list for our startup-tests. If the camera is not connected the camera indicator LED will flash about twice per second for about 10 seconds, after ten seconds the device will assume you have no intention of connecting the camera, the light will blink quickly once every four seconds. The camera sub-routine is now, and for the entire time the device is on, until it is reset, disabled. No images will be captured or stored. The log file will indicate the camera was not connected. The mode allows the processor to forgo the clock cycles necessary to store image data and prevents the card from filling up with blank image files. If you want to use the camera you must plug it in and reboot the device. Once you have plugged in the camera to the device and powered the device by starting the car's engine, the device will check communication with the camera, if all goes well, the camera indicator LED will glow solid. If there is a communication problem with the camera the LED indicator will flash rapidly six times per second. The device will keep testing the communication with the camera until it is either successful or 10 seconds has passed, at which time the device will act like the camera is not connected and ignore it.

If there are any devices plugged into the other expansion ports software must be written to support the future device. In addition to adding the data retrieval and storage code, there must also be some user interface code written to support the expansion port LED indicator. This way the new device may also enjoy the same error checking and status indicators the rest of the devices have.

When all currently implemented systems are operational and functioning properly all indicator lights should be lit and solid. If the expansion port is not in use the LED indicator should not be light, this merely means that it is not in use, which is correct. If correctly programmed, when the expansion port is in use the LED indicator should be lit and solid.

## 5.2. User Configuration

In addition to the user needing to know what is going on with the device, the user should be able to configure the device based on the individual logging requirements. Some devices may need to poll images more or less than others; some may want more accurate accelerometer readings. Since the accelerometer readings are an average over the polling frequency, as the frequency increases the accuracy also increases. However all these polling frequencies need to have the ability to be configured easily. Configuration is a significant factor in device design, it must be simple and easy, and fault tolerant.

Our device configuration is to be done with a text file located on the storage media. We think this would be the easiest way to configure the device, as a text file can be easily edited on just about any personal computer. The text file is also easily read by our device and does not require there to be any hardware changes, jumpers or dipswitches to be set. The text file also offers the needed fault tolerance. Fault tolerance is much like

the error checking done with the user interface. After all the device checks are done, we need to make sure the device has a configuration file loaded and that it is readable and not corrupted. If for some reason the configuration file is missing or damaged, the default configuration can be loaded from the firmware stored on the device. The backup configuration file can be also be changed by flashing the microcontroller, which is a more technical process, but doable. Once the new file is restored or the version on the card is loaded successfully the device can load its polling frequencies and begin to log data.

## 5.3. Physical Size

While we are talking about physical size, lets outline the physical layout of the device. The device has three main components: the central processing unit, the camera, and the GPS device. The central processing unit (CPU) houses all of the main circuitry, including the microcontroller, the storage media, user interface, OBDII decoder, accelerometers, and yaw rate sensor. This device should be located near the OBDII port, which should be no more than thirty-six inches from the steering wheel. The CPU should be placed on the floor for a flat, level surface with a low center of gravity to avoid exaggerated motions from the suspension. This device we expect to be no larger than two inches by five inches by ten inches, which is a conservative estimate. This device will have to have enough surface area for the various plugs for the additional components and user interface LED lights.

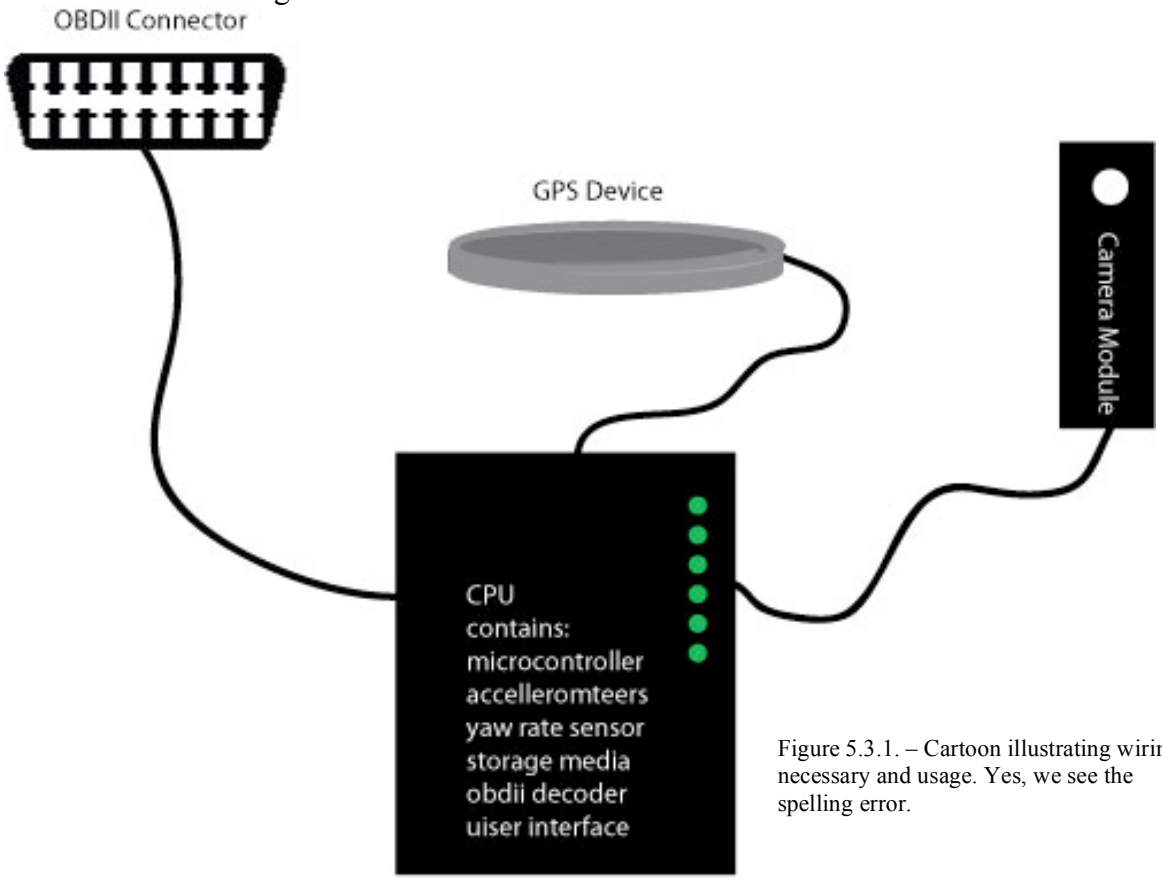



Figure 5.3.1. – Cartoon illustrating wiring necessary and usage. Yes, we see the spelling error.

Connected to the CPU is the camera module and GPS device. The GPS device should be either a round or square "puck," which has a magnetic base and waterproof housing, which is to be mounted on the roof. The GPS device should be about half an inch thick and 3 inches in diameter or 2 inches square. This device contains the GPS antenna, which is why it must have a clear view of the sky. Due to the device's small size, which forces the antenna to also be small, the GPS device must also have a large ground plane to help focus the signal. The need for a ground plane and clear view of the sky make the roof an ideal place to mount the GPS device.

The camera module we expect to be no larger than a pack of gum and use a Velcro strap as a mounting to either the passenger seat headrest or the rear-view mirror. Wherever is most convenient of a mounting place that has a good view of the road and even the driver if possible, but does not impede or distract the driver, is a where the camera should be mounted.

## 5.4. Logical Size

Logical size refers to the size of the logical device, the storage media. The device uses a commonly available memory storage media called Secure Digital, abbreviated as SD card. Secure Digital media cards can be found in sizes from about 64MB all the way up to 4GB. When deciding on our removable storage the logical size of the removable storage was certainly of extreme importance. The storage media interface chip, called the DOSonCHIP is reported to address cards larger than the largest currently available size. Although when we asked the company to speak on this topic we received a conflicting answer. Regardless of whether or not the company knows what they are talking about we see no reason the card will not be able to address at least 2GB which is adequate for our use. When deciding on a removable media storage card size we mainly focused on the number of pictures we will need to capture, as text logging takes significantly less space than images.

The table below shows the average file size at each resolution. Sample images were taken in different conditions with different content to try and represent the varying compression ratios in JPEG images. JPEG, which stands for Joint Photographic Experts Group, is a standard method of image compression commonly used today. We will be using JPEG compression with our logged images.

image file size chart

| | vga | qvga | 160x120 |
|---|---|---|---|
| sample 1 | 45.2 KB | 14.3 KB | 7.8 KB |
| sample 2 | 37.1 KB | 12.3 KB | 5.4 KB |
| sample 3 | 18.6 KB | 6.9 KB | 7.1 KB |
| sample 4 | 28.3 KB | 10.3 KB | 6.8 KB |
| avg file size | 32.3 KB | 11.0 KB | 6.8 KB |

| | Approx File Size (KB) | Approx Number of Images/Capacity (MB) | | | | |
|---|---|---|---|---|---|---|
| | | 128 | 512 | 1024 | 2048 | 4096 |
| VGA | 32.3 | 4057.96 | 16231.83 | 32463.65 | 64927.31 | 129854.61 |
| QVGA | 10.9525 | 11967.31 | 47869.25 | 95738.51 | 191477.01 | 382954.03 |
| 160x120 | 6.77 | 19360.71 | 77442.84 | 154885.67 | 309771.34 | 619542.69 |
| | | Hours of Image Capture @ 1Hz | | | | |
| | VGA | 1.13 | 4.51 | 9.02 | 18.04 | 36.07 |
| | QVGA | 3.32 | 13.30 | 26.59 | 53.19 | 106.38 |
| | 160x120 | 5.38 | 21.51 | 43.02 | 86.05 | 172.10 |

Chart showing number of images that will fit on different sized media cards.

Based on the chart above, you can select your media card size based on how long you want the device to last before you have to remove, backup, and erase the media card. We expect the device to be most useful if it does not have to be constantly maintained, which means the longer it can last, the better. We also expect the usual capture size will be QVGA, or 320x240 pixels. Based on this assumption and a moderately priced SD card, the 2GB card, our device will last for about 50 hours. Please remember, this chart is only an estimate, based on image complexity the numbers can vary drastically. The chart also does not account for any file format overhead such as minimum block size or FAT table overhead. The chart only estimates images and does not include the text data which is small in comparison but critically important to the purpose of the device.

## 5.5. External Antenna Expandability

The idea of moving the GPS engine to the CPU box and just having an antenna port on the CPU box with an inexpensive patch antenna running to the roof has been discussed before. Really it depends on how the device is to be used and how likely the components are to be lost. Replacing a $10 antenna is much less expensive than replacing a $90 GPS unit quite obviously. If the GPS engine were moved to the CPU box, all of the expensive parts would be locked inside the car. I would not expect there to be a problem theft of vandalism, but it is a point that should be raised, as the design change is rather simple, but much easier to do before production than after.

## 5.6. Future Expandability

We have tried to make sure the final device design is somewhat future-proofed. Larger storage cards should work when they become available, there is an extra expansion port for additional devices, IO pins on the microcontroller left open for future use and a spare status LED to be used by the additional device. In trying to think ahead of how we might expand the design, we have found there are features we would like to incorporate in to this version but feel there is simply no time. We would prefer to meet the requirements and the deadline with a working version rather than have a spectacular improvement on the requirements but run the risk of it not being finished. Therefore, we have decided to catalog our design improvement ideas in case the device is finished early, or someone decides to build upon the device after it proves its utility.

We have so far limited the configurability of the device to a text file stored on the storage media. In future we expect to have data polling rates be dynamic based on environment variables such as speed or acceleration. The idea is to prevent 90 images of the car in front at a stoplight. If the car has stopped moving, the camera can capture

images once every 30 seconds, or just wait until the car starts moving again. Also if the car begins rapid negative acceleration, as would occur in an emergency stop, the image capture rate could increase to catch the entire experience and driver reaction. The OBDII and accelerometer polling could also increase to show the driver's reaction to try and determine what evasive maneuvers or behaviors that attributed to the emergency situation.

Both of these ideas may require a bit more processing power from the microcontroller. Since we have not yet completed the project, we do not know how efficient the Dynamic C will be when it gets converted to machine language. We may be able to implement a much more simple method of controlling capture rates to extend the longevity of the device by adding a control switch to the camera module. This module should be within the car operator's immediate reach and could be adjusted just as easily as the mirrors or stereo.

After we begin initial tests we will also be able to determine the necessity of images at night. We expect to find night images particularly challenging for our image capture device and would not be surprised of the images turn out too blurry or underexposed to be useful. We can use information provided by the GPS system to anticipate loss of daylight and shut down the image capture accordingly.

Better data acquisition is always an important facet of this device, but equally as important is collecting the data. Since the device does not necessarily work for a prescribed period of time, and each device will have a different run-time based on usage, backing up and clearing the contents of memory for each device will become a task all by itself. For this portion of the device support we also have some ideas to make the process more streamlined. An auto-run script could be included on the storage media, which will execute when it is inserted into Windows based PC. The script could automatically zip the data contents, naming them appropriately, and readying them for upload. The newly created zip file backup up to the users computer and the contents of the media card cleared. The script could then open an ftp session to a central server and upload the data. Obviously if the user is uploading 2GB of information, this process would take a long time, a progress meter would accompany the upload, meanwhile the card could be placed back into the logging device. Ideally the text portion would be uploaded before the images, as the upload time would be much less, and the text data is more important per byte than the images.

### *5.6.1. Expansion Port Protocol*

Without knowing the device to be used in the future expansion port, we are going to make it a standard serial port and have a built in level converter to protect our circuit. This will make communication fairly standardized, so if the new device is RS232 it should work without modification, if it is a serial UART device, all it will need is a simple level shifter to attain proper voltage swing. Leaving this as a standardized port will make is as versatile as possible for future use while still leaving it directly connected to the microcontroller.

## 5.7. Software Update Loading Port

In addition to all of the many interface ports that will be in use by the sensors, we will also need to make sure one is reserved to upgrade the software. This port must not be

in use, and should have the ability to load software to the microcontroller, as we do not want to make updating the software a difficult task that involves hardware changes, or setting jumpers. Software can be loaded using a standard RS232 (serial) port from a PC using the Rabbit's Dynamic C software environment.

## 5.8. Diagnostics Port

We would like for the software update port to double as a diagnostic port, using usually unconnected pins to just output straight serial data that give information about what the microcontroller is doing. This port may require jumpers to activate in hardware and software so additional clock cycles are not wasted when diagnostics are not necessary.

## 5.9. Mounting Location

As previously discussed, each module will need to be mounted in a specific place for optimal effectiveness. Mounting location was a consideration in the design of the device and led us to the three-module design of the system.

### *5.9.1. Ease and Speed of Installation*

The device must not be complicated to install or use, that is a primary goal. If the device is too much work or too complicated, it will not get used, just the same as any other device on the market. We have tried to make the device as simple as possible while still accomplishing its rather complex set of tasks. The device's function does require a certain level of complexity. The device has to be connected to the OBDII port. That port is always near the steering wheel. The device has to have a clear, unobstructed, view of the sky, which is usually found on the roof of the vehicle. The device must be able to look out the front windshield to capture images, which would put the camera in a passenger position. Finally, the device needs to be isolated from excessive false inputs to the accelerometers caused by the suspension. The best place to isolate the device is at the lowest point possible, the vehicle's floor. All of these different requirements do not leave any room for compromise. However, there are some components without restrictions. The microcontroller and accompanying circuitry does not require a specific mounting place, the OBDII circuitry does not require a certain mounting place, and the SD card module does not require any certain mounting place. The SD module does need to be accessible to the user and easily accessed. Based on these requirements we decided that as much of the device as possible would go into the floor module. Since that includes the microcontroller, we named this the CPU module. That only leaves the components that are required to be somewhere else left to add. We have decided to extract power from the OBDII port to avoid excessive number of cables running through the vehicle. The rest of the devices are also detachable from the device to ease setup and installation. The modularity of this design also allows the user to leave components disconnected if they do not wish to use them.

### *5.9.2. OBDII Accessibility*

As previously mentioned, there are certain restrictions imposed upon the OBDII standard. One of which is where the interface port is located. Although there are no strict definitions as to where specifically it has to be mounted, only that it must be within

thirty-six inches of the steering wheel. Assuming manufacturers adhere to this standard, the user should have few complications finding the vehicle's OBDII interface or accessing it from within the vehicle.

### *5.9.3. Wire Routing*

As discussed we tried to keep the number of wires to a minimum to ease the installation procedure. In addition to making the device easier to install we also need to keep the vehicle free of obstacles and tangles. For this reason we have also designed a proper cable routing method to keep the wires out of the passenger and driver's view and travel path.

Figure 5.9.1. – Car interior showing OBDII port location.

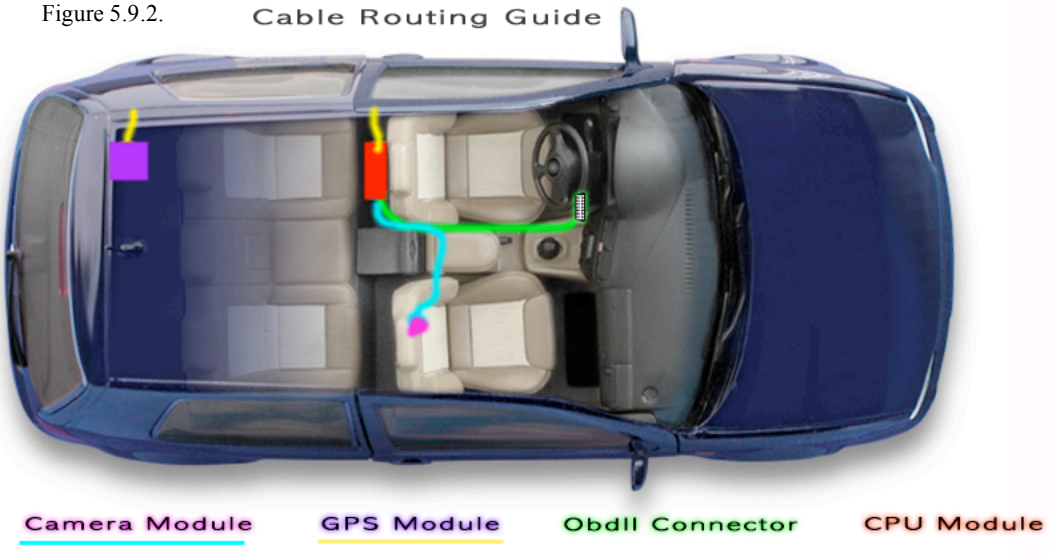
* Photo: stmongomery

As shown in the above image, the OBDII connector will be located somewhere close to the steering wheel. When routing the cable make sure that the cable is secured under the dash to avoid stepping on it when trying to apply the brake or accelerator. Then run the cable along the floor adjacent to the seat while being sure not to interfere with the seat track.

The main CPU device should rest on the floor in the back seat to avoid accidental abuse and facilitate easy wire routing for the remainder of the devices. In addition, since the CPU module contains all of the accelerometers and the yaw rate sensor, it should be at the lowest part of the car possible to avoid exaggerate input from the suspension. If there are going to be passengers in your car, the passengers should take care not to disturb the device as and kicking, stomping, or poking, will be logged on the sensors. As we can see from the next diagram, the camera module can be placed on the headrest of

the passenger seat, provided no one is sitting there, or alternately on the rear-view mirror as long as the driver's view remains unobstructed. Finally, the GPS module can be run to the roof, the wire can travel up the B-pillar of the vehicle and be closed in the door. The B-pillar of the car is the part of the car the driver's side door is latched to. Not to be confused with the A-pillar where the door is hinged. In the diagram the devices with wire a different color from the device (also underlined) signifies they are optional components, the only component that is not optional is the OBDII connector, as it supplies the power for the entire device and the components.

Figure 5.9.2.




* Illustration used and edited with license

### *5.9.4. Low Center of Gravity*

The importance of clean data has been discussed before, even in relation to data exaggerated by the suspension, but we will now take a closer look as to why this is important. When a car goes through a turn the suspension expands and compresses to absorb the forces exerted on the car in an attempt to make the ride more comfortable. In some cars, the ride is softer than others. Usually a softer ride is associated with more body roll. The following picture examples significant amounts of body roll.

Figure 5.9.3. – Car showing extreme body roll.


* Photo: VWVortex

In the picture above the driver side of the car's suspension is fully compressing while the passenger side suspension is fully extending. The body roll on the car is so extreme that the car's tire is actually coming off of the road. In the picture to the right the car's suspension is much stiffer, which means there is much less body roll during cornering. The accelerometers and yaw rate sensors will record the movements and forces caused by cornering. In order to maximize the validity of our data we need to minimize false inputs exaggerated or minimized by the different suspensions of various cars. The best way to accomplish this task is to put the sensors as close to the plane of the suspension as possible. The following diagram illustrates how the movement readings will be affected based on the location of the sensors. The blue line represents the plane that contains the suspension components at each wheel. The black line is perpendicular to that, and the red line is perpendicular to the road. The farther away from the blue line you get, the more extreme the movement is based on the saw suspension compression and expansion. As you can see, having the device lower in the car will render more reliable results with fewer false inputs.

Figure 5.9.4. – Car showing lmited body roll.

* Photo: Tinou Bao

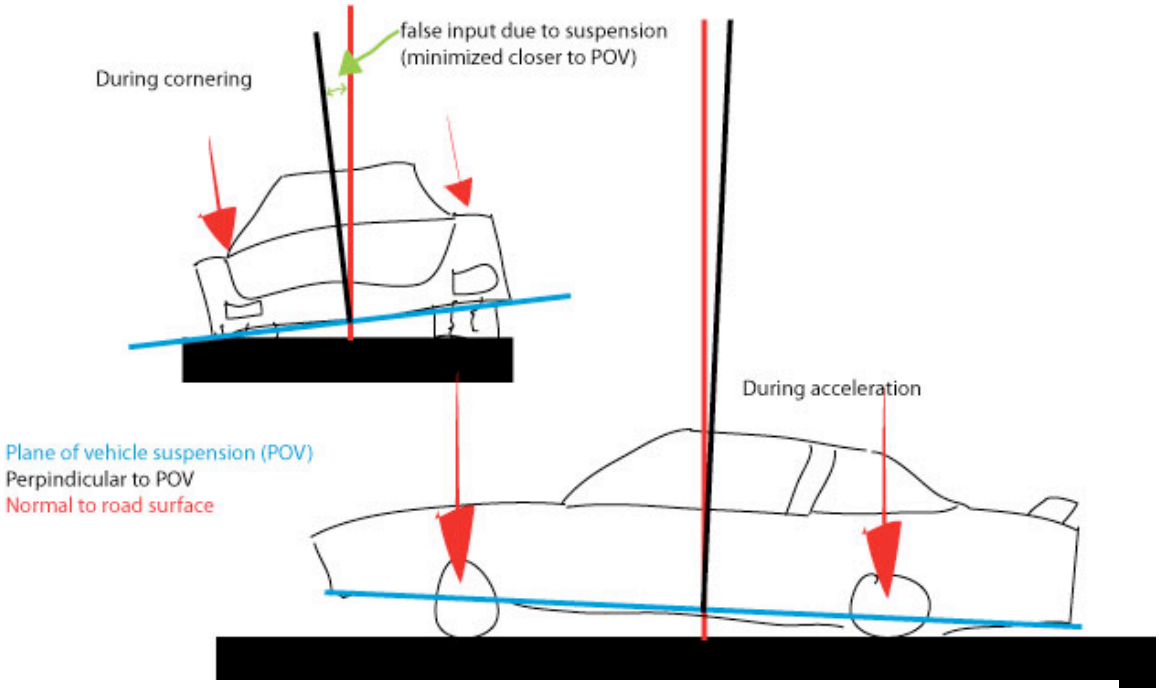



Figure 5.9.5. – Cartoon illustrating how the placement of the accelerometers could potentially affect the acceleration readings due to the vehicle's suspension.

### *5.9.5. Level Surface*

A level surface is also important to the installation. Some devices can be haphazardly installed in any orientation, however, due to the positioning of the accelerometers on the circuit board, the CPU module must be mounted based on the following guidelines. Currently the device has accelerometers for only two axes, the project guidelines require recording longitudinal and lateral accelerations, if the accelerometers are not aligned properly, they will not get accurate or complete data. For example, if the device is mounted vertically, perpendicular to how it was designed, it would have one axis in the direction of gravity rather than in the direction of longitudinal or lateral acceleration. There is no way to extract this missing data if the device is mounted in this manner. If the device is rotated horizontally, rather than facing straight forward as it was designed to be, a braking force will be recorded in both axes, rather than just longitudinal. Software can be used to interpret the data, but will add complexity and introduces the possibility of errors.

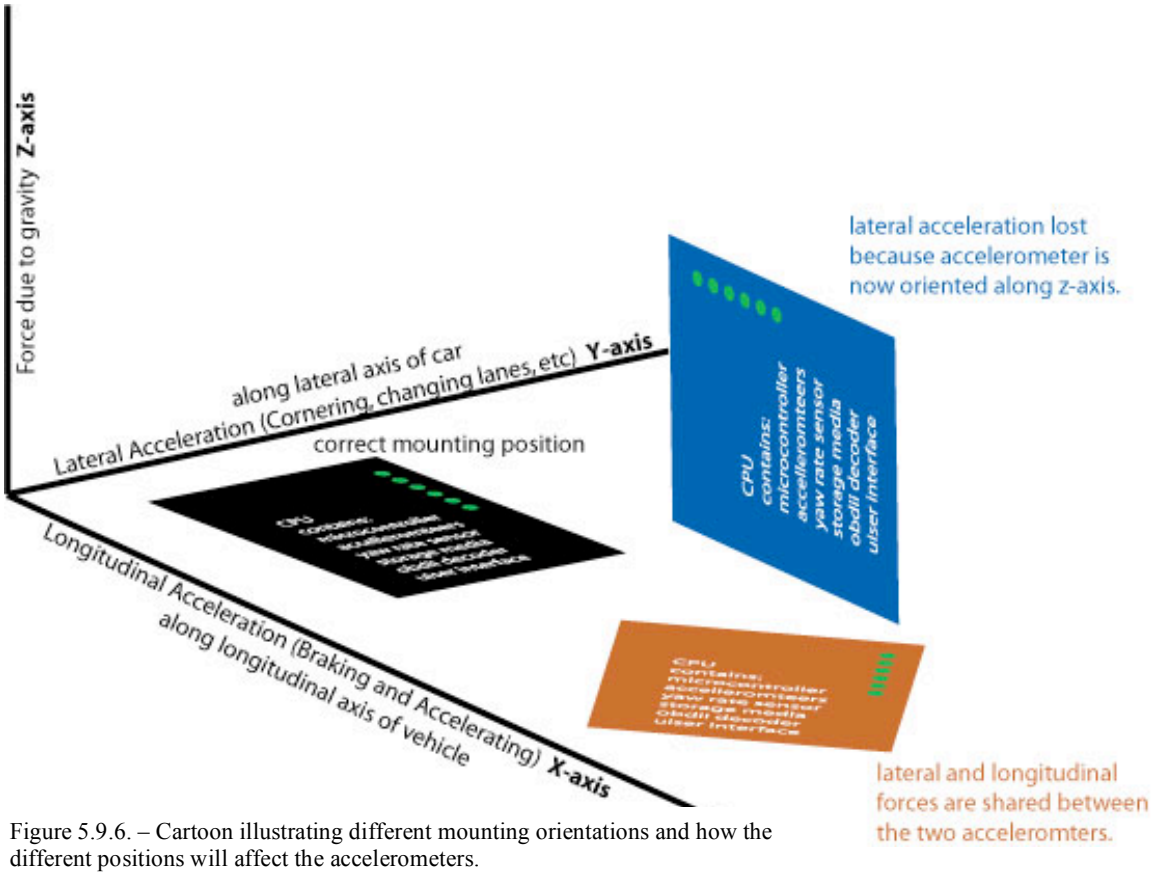



Figure 5.9.6. – Cartoon illustrating different mounting orientations and how the different positions will affect the accelerometers.

In future we may find we will have to add a third axis in order to calculate the starting position of the device and compensate for inaccurate readings due to a non-level mounting. Consistently level mountings will ensure the recorded readings have relevancy to the collected data from other units.

### *5.9.6. Camera Field of View*

Ideally we would like to know what is going on in all directions around the car, and also be able to interpret this data by machine rather than needing a human to examine the images to extract information. Unfortunately, computers cannot yet recognize data in images accurately enough, and the cost of cameras in all directions would be too high. Even if the cameras were less expensive we would need a much more powerful microprocessor and much larger storage device to accomplish this task. The next best alternative to the ideal situation is to have the camera with a large field of view mounting in the proper place. As we recall from the discussion in section 2.5.4. a larger field of view will allow us to capture more of the scene than a smaller field of view. Since the purpose of the device is to log data, the more data the better. A larger field of view means more data. Once we have the field of view set appropriately, which is controlled by the lens' focal length, we need to position the camera. The optional requirement for project was to capture forward-looking video. In addition we feel it is also important to capture the profile shot of the driver in order to log the conditions present in the passenger compartment of the vehicle. For this we expect to need the camera on the headrest of the passenger seat pointed at a slight angle towards the driver.

### *5.9.7. Device Accessibility*

Although we need the device to be somewhat insulated from human interaction, to avoid bumping and moving the device, we also need it to be easily accessible to the user. The user will need to periodically check the user interface to make sure there are no errors, as well as remove the data storage media in order to backup the data, erase the media card and reinstall the media card in the device. The device must also be easy to remove in case the user wants to have passengers that may interfere with or step on the device.

### *5.9.8. Mounting Method*

The mounting method has so far been one of the least thought about facets of the device's design; we originally thought it would be easily figured out "later." However, the more thought we give this enigma, the more daunting the task becomes. We must account for all sorts of different mounting surfaces, with any amount of dirt, grime, and debris. We expect to be mounting the device on the floor of different cars; each floor could offer us a new surface to mount on, carpet, metal, rubber and possibly more. Adhesives are most likely out of the question because of their degree of permanence an inability to be reused. We cannot use any sort of permanent mounting like a metal bracket, even though the device needs to be held securely, as it would damage the vehicle. We felt the most likely option to yield acceptable results would be fabric or plastic hook and loop fasteners. We would need the hook side of this duo to attach to the fabric loops commonly found in the car interior upholsteries. We expect the plastic version to be more durable and also offer a stronger, more secure bond. Unfortunately we do not expect regular hook and loop fasteners to provide adequate security in more plush carpeting. For the plush carpets that are a cut pile rather than the loop style fabric, we need a series of spike that will rest amongst the pile and prevent movement, while not damaging the upholstery. We feel if we can find plastic hook fasteners that are long enough it will suit both styles of carpeting. For rubber and vinyl floor-mats we are still

unsure of the best way to affix the device to prevent movement, but will wait to see if we need to worry about this possibility considering how few cars have a flooring that is not a type of carpet. For the very rare bare metal floors we can use small rare earth magnets, which are extremely strong, to secure our device.

# 6. About Us

## 6.1. Facilities and Equipment

The first stage of this project will be completed in the personal homes of the group members. Our own personal computers will be used for all coding and most of our own tools will be used to create prototypes. Dr Papelis provided some developments kits. Initial testing of the device will be performed in our own vehicles (2000 Nissan Xterra, 2005 Mitsubishi Lancer). Once initial testing is completed circuit board fabrication will be outsourced to "PCBExpress." Final assembly will be conducted in our own homes and then final test will be conducted in our sponsor's vehicle of choice.

## 6.2 Summary and Conclusions

Our project is progressing according to schedule. We do not envision any problems that will keep us from completing it on time.

## 6.3. Project Personnel

# Kyle Fiducia

Kyle has been working in technology field since 1999 when he worked as a network administrator. Since then he has launched three successful personally owned businesses. He has also done consulting work to launch two other businesses. All businesses are still running successfully. Kyle also has an interest in hardware devices and hopes to further his knowledge in hardware development. Kyle will be graduating from the University of Central Florida with a bachelor's degree in electrical engineering in May 2007. After which he plans on attending graduate school at the University of Central Florida.

# Joshua Mahaz

Joshua has been involved in web based software development since 2004, specializing in PHP, MySQL, and AJAX. He also owns and has lightly developed software for Basic Stamp Microcontrollers, through this project he hopes to gain a much firmer footing in developing software for microcontrollers as well as touch on developing and integrating hardware devices for them.

After Joshua graduates in August 2007 with a bachelor's degree in computer engineering, he will be enrolling in graduate school where he plans on furthering his studies in software engineering and microcontroller devices.

# Graham Smith

Graham Smith's main focus in this project is the power supply and backup systems because he has experience with power electronics and Li-Ion cells. He also has experience with circuit layout and soldering components. Graham will graduate from UCF in May 2007 with a BSEE, and then continue on to graduate school to pursue a masters in electrical engineering.

# 7. Works Referenced:

## OBD

http://en.wikipedia.org/wiki/OBD-II_PIDs
http://www.elmelectronics.com/connect.html
http://en.wikipedia.org/wiki/On_Board_Diagnostics

## I2C

http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf
http://en.wikipedia.org/wiki/I%C2%B2C

## SPI

**http://embedded.com/showArticle.jhtml?articleID=9900483**
http://en.wikipedia.org/wiki/Serial_Peripheral_Interface

## Camera

http://www.sparkfun.com/commerce/product_info.php?products_id=7906
**http://www.acroname.com/robotics/info/articles/ramcam/ramcam.html**
**http://www.chipdesignmag.com/display.php?articleId=16&issueId=3**
**http://www.electronic-kits-and-projects.com/kit-files/ovt/OV7620_OV7120_v1.2whole.pdf**

## Voltage Regulators

**http://www.national.com/pf/LP/LP2960.html**

## Rabbit

**Yahoo Rabbit Group**
http://tech.groups.yahoo.com/group/rabbit-semi/

**Rabbit/dynamic C**
http://www.rabbitsemiconductor.com/
**http://www.impulse-corp.co.uk/index.html**

**GPS**
http://www.garmin.com/

## Yaw Rate Gyro

http://en.wikipedia.org/wiki/MEMS

## Storage Medias

http://www.dalsa.com/shared/content/Photonics_Spectra_CCDvsCMOS_Litwiller.pdf
http://www.compactflash.org/faqs/faq.htm

# Copyright permissions:

## National Semiconductor Datasheet Diagrams

**"Access and Use of the Site**. You are granted permission to access and use the Site, including permission to download, copy, and display materials contained on the Site, on a limited basis for internal, noncommercial, informational purposes only and strictly subject to these Terms."
http://www.national.com/webteam/site_terms_of_use.html

## Images

Carril, P. "ADM." Photograph. *ESA Multimedia Gallery* < http://www.esa.int/esa-mmg/mmg.pl?b=b&type=I&collection=Spacecraft%20Engineering&start=5>

ESA: "Most images have been released publicly from ESA. You may use ESA images or videos for educational or informational purposes."

Juni. "Top" Flickr. < http://en.wikipedia.org/wiki/Image:Japanese_top.jpg>
tinou bao. "Zoom zoom" Flickr. <http://www.flickr.com/photos/tinou/page5/>

Photographs used with permission under Creative Commons license.

VWVortex: Materials subject to the following: (1) you can use the Materials only for personal, informational, internal, non-commercial purposes;

Images that were licensed do not require citing in terms of use agreement. License was purchased.