Project Code: FJL-CIT2

Automated GPS Mapping of Road Roughness

A Major Qualifying Project Report
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

Project Team:

Carlton Stedman II
sageman@wpi.edu

Andrew DeMarco
ademarco@wpi.edu

Advisors:

Fred Looft, PhD
fjlooft@ece.wpi.edu

Fabio Carrera, PhD
carrera@wpi.edu

1 June 2007

ABSTRACT

Many roads across the United States are in poor condition, which can lead to unnecessary accidents and repair costs. In order to alleviate these problems, our group built a second generation road roughness detector that could identify these troubled roads. With the aid of GPS and a mapping program, city officials would now be able to keep track of the condition of their roads. Through examining the first generation design and researching applicable topics, our team created a more compact product that was easy to use.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 INTRODUCTION

According to TRIP, a nonprofit organization that promotes transportation policies, thirty-five percent of major roads in the United States are in "poor to mediocre condition" and that millions of Americans are affected by these conditions every day (Key Facts, page 2). One particular impact is that many motorists have to pay for vehicle repairs due to damage caused by rough roads. Even worse, the substandard conditions of roads have been shown to result in unsafe conditions for drivers, often playing a primary role in traffic related accidents and deaths.

A paper published by TRIP claims that motorists throughout the nation pay an extra fifty-four billion dollars a year in vehicle repairs, due to driving on roads which have not been properly maintained (Key Facts, page 1). This is, on average, roughly 275 dollars per motorist. In urban areas, motorists are paying even more, approximately 401 dollars (Rough Ride Ahead, page 2). The TRIP paper also noted that traffic congestion due to accidents caused by poor road conditions cost American motorists 63.1 billion dollars in wasted time and fuel. Vehicle crashes cost drivers 230 billion dollars a year, which includes medical costs, insurance costs, legal costs, and workplace costs (Key Facts, page 1).

The worst problem concerning the condition of roads is the possible danger imposed on motorists. In the same TRIP paper mentioned above, it was determined that in approximately one out of every three traffic related fatalities, roadway conditions played a significant role. In 2004, over fourteen thousand deaths resulted largely due to roads that had not been maintained (Key Facts, page 2). With appropriate road improvements, traffic fatalities and accidents can be significantly reduced. According to the Federal Highway Administration, 100 million dollars spent on these improvements will save 145 lives over a ten year period (Key Facts, page 2).

The number of roads in poor condition has grown each year since 1998. In particular, urban environments have some of the worst road conditions in the country. Many major cities aim to maintain seventy-five percent of its roads in good condition. However, only thirteen of the nation's urban areas with a population of 500,000 or more have at least fifty percent of their roads maintained in fair or better condition (Rough Ride in the City, page 2). As an example, forty-nine percent of the roads in Boston are considered to be in substandard condition (Rough Ride Ahead, page 2). With significant increases in traffic expected in the future, it is almost certain that road conditions will continue to deteriorate if nothing is done.

Even though poor road conditions are a major problem, the ability of cities to find problem areas in their roads is not well developed. Many municipalities, such as those throughout Massachusetts, rely on motorists to report problems in the roads, for instance potholes. These cities need to be able to easily and inexpensively determine the condition of their roads. This requires measuring the roughness of a road and coordinating the reading with a geographical location.

The goal of our project was to design an inexpensive, small, low power, and autonomous system that logs data for road roughness measurements, which are coordinated with geographical locations. The device can be placed inside any type of municipal vehicle without being obstructive. It continually monitors the road condition as the vehicle drives. After the system records data, it is plugged into a computer and the data is uploaded. This data is overlaid onto a map which shows where rough spots in the road are located and how relatively rough they are.

1.1 Report Summary

This introduction has provided an overview of the state and implications of the road roughness problem. Additionally, it has been described how our project provides a solution to this problem. The report will continue on into a background section, providing an understanding of the history of the road condition problem, as well as other important related areas. A problem statement section will then follow, describing the goals, objectives, and tasks of the project, in addition to providing a schedule. Next, in a methodology section, an approach to accomplishing our project will be described. A system design section will then present detailed descriptions of how our system works, followed by results. A conclusion section to this report provides an overall summary and analysis of the project work.

2 BACKGROUND

In the section, background material will be presented to help further understand the problem of rough roads in the United States. It will detail how cities across the nation currently measure the roughness of roads and the problems that go along with these methods. Our project will improve upon these methods to make it easier and more efficient to determine which roads need to be fixed before serious damage occurs. The theory of how the project operates will also be discussed in this section.

2.1 Current Methods of Measuring Road Roughness

In order for cities to improve their rough roads, it needs to be determined which roads actually need to be fixed. One simple method is for people to call and complain about rough roads in their area. A more useful method is physically determining the roughness of the road. There are various pieces of equipment used in measuring the roughness of pavement.

2.1.1 Profilograph

One device is a profilograph, which is used to measure the longitudinal profile of the pavement. A profile is a two dimensional slice of the road surface, taken along a continuous imaginary line. This profile shows the roughness and texture of the road (Sayers, page 2). Figure 1 shows an example of a road profile.

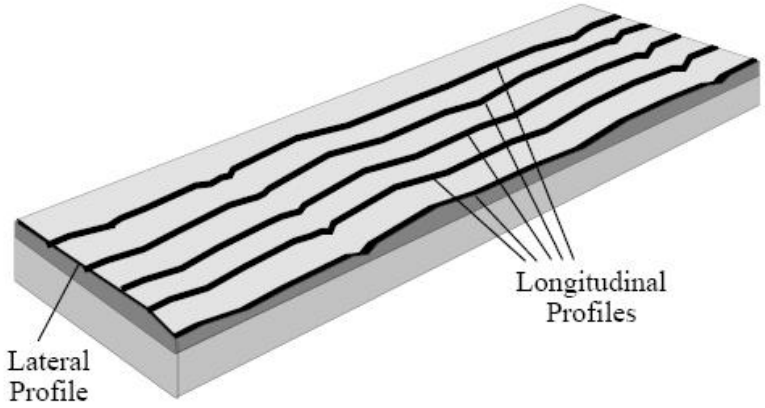


Figure 1: Profile of Road Surface (Sayers, page 2)

Profilographs come in a variety of forms with different number of wheels. They are manually operated by moving the device across the road at walking speed. There are two major types of profilographs. The first is known as the California type profilograph. It consists of four to twelve wheels that are attached to the ends of a 25 foot truss, as well as wheels at the centerline of the truss. The other model is known as the Rainhart profilograph. This particular type is similar to the California profilograph. The main difference is that this type uses twelve wheels arranged in four groups of three. This arrangement allows for twelve longitudinal profiles at once as opposed to only three, as with the California type (Budras, para. 6). These two types are shown below in Figure 2.

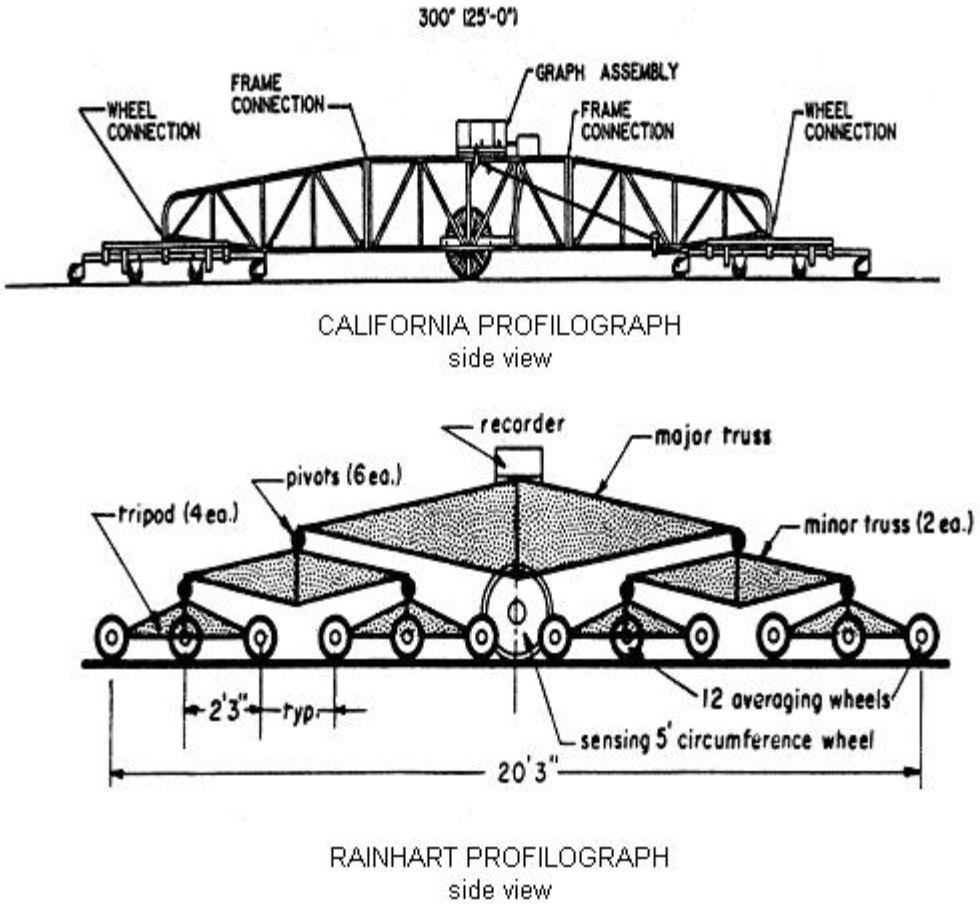


Figure 2: Sketches of California Profilograph and Rainhart Profilograph (Budras, para. 10)

There are many problems with this method of determining road roughness. One problem is the device is quite large. It is 25 feet long and a challenge to maneuver. Another problem is the device is unusable for large stretches of roads. Since the device must be manually operated, only a small section of road can be analyzed at a time.

2.1.2 Response-Type Road Roughness Measuring System

Another piece of equipment used to measure the roughness of pavement is response-type road roughness measuring system (RTRRMS). This device determines road roughness by measuring the response of a vehicle to the road texture. Unlike profilographs, this device can operate at highway speeds. The most commonly used version of the RTRRMS is the Mays Ride Meter. A diagram of the Mays meter can be found in Figure 3. This meter measures the space between the attached axle housing and the car itself. It consists of a transducer, a distance measuring instrument, and a pavement condition recorder. The transducer is used in this system to convert the movement into an electrical signal. The recorder processes information received from the transducer, the keyboard, and the distance measuring instrument, which acts like an electronic odometer. The output of this meter is relative motion (in inches) over a distance (Budras, para. 10).

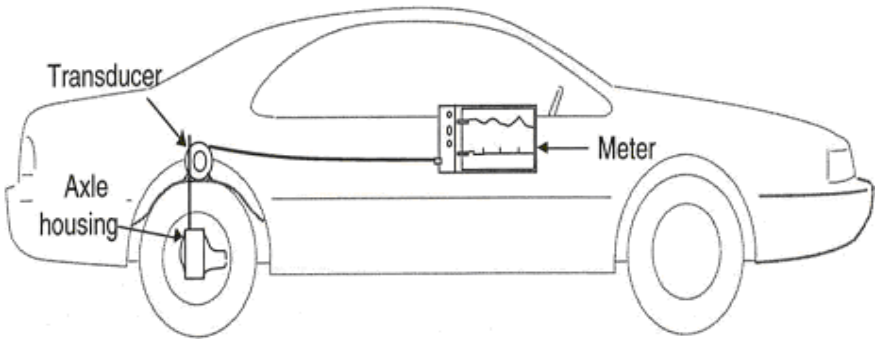


Figure 3: A Car with a Mays Meter (Budras, para. 16)

There are some problems associated with this type of device. One problem is that the road profile is not being directly measured. The vehicle response to the road is being measured. This measurement is not reliable because of its dependence on the vehicle. One vehicle could be different from another. Another problem is the processing is done all inside the vehicle. This takes up at lot of room in the vehicle and requires another person to control the meter.

2.1.3 Road Roughness Profiling Device

A third device used to measure road roughness is known as a road roughness profiling device, which measures the longitudinal profile of the road. An inertial profilometer is a commonly used profiling device in the United States. A diagram of the inertial profilometer is located below in Figure 4 (Budras, para. 16). This particular profilometer uses an accelerometer to create an inertial reference that defines the height of the accelerometer located on the vehicle. A height sensor is used to determine the height to the pavement surface from the vehicle. A laser is most commonly used for the height sensor. This height and the reference level are used to measure and compute the longitudinal profile of the pavement (Achieving, para. 10). The processing is done on a computer located inside the vehicle. The computations are performed in real time as the vehicle is moving. It can operate at speeds between 10 and 70 miles per hour.

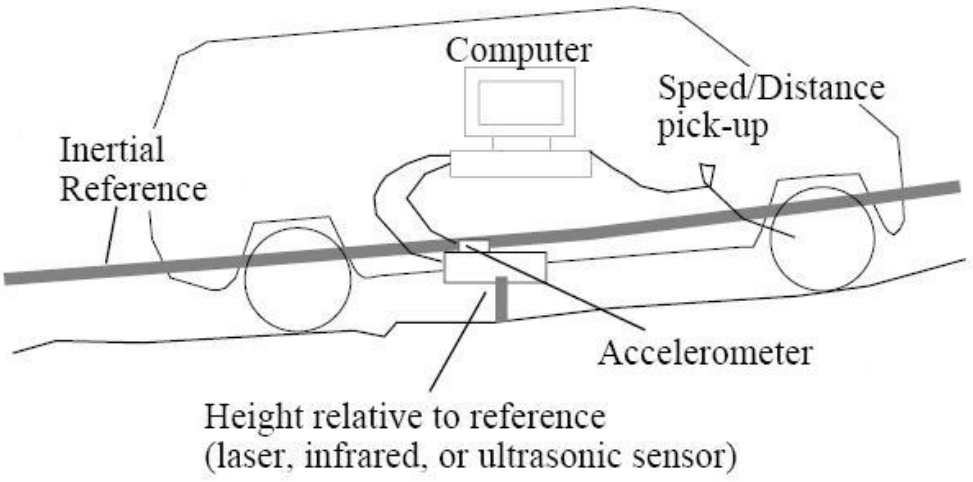


Figure 4: A Van Equipped with an Inertial Profilometer (Sayers, page 6)

As with the previous devices, the inertial profilometer has certain drawbacks. One potential problem is the laser used in the height sensor. Since road surfaces present many dynamically-changing targets including tarmac, concrete, yellow striping, and white striping, the distance provided by the laser might not be that accurate (Laser, para. 1). Another problem is size. In order to process the data, a computer needs to be installed inside the vehicle. This wastes space and requires an additional person to operate.

There are also non-contact lightweight profiling devices used to determine the roughness of the road. They are similar to the previous profiling devices, but are much smaller and lighter. However, these devices can only operate at speeds between 8 and 25 miles per hour (Budras, para. 24).

2.1.4 Multi-Laser Profiler

A fourth piece of equipment used to measure the roughness of pavement is the Multi-Laser Profiler (MLP). Figure 5 shows a vehicle that contains the MLP system. This system measures longitudinal profiles of the road surface by using lasers. It is useful for monitoring large amounts of roads (370 miles) and works well at highway speeds (18 to 75 miles per hour). The MLP comes with a computer system that handles data acquisition and analysis. This system also offers additional features, such as GPS, digital mapping system, and defect logging, which logs the road defects while driving (Budras, para. 25).

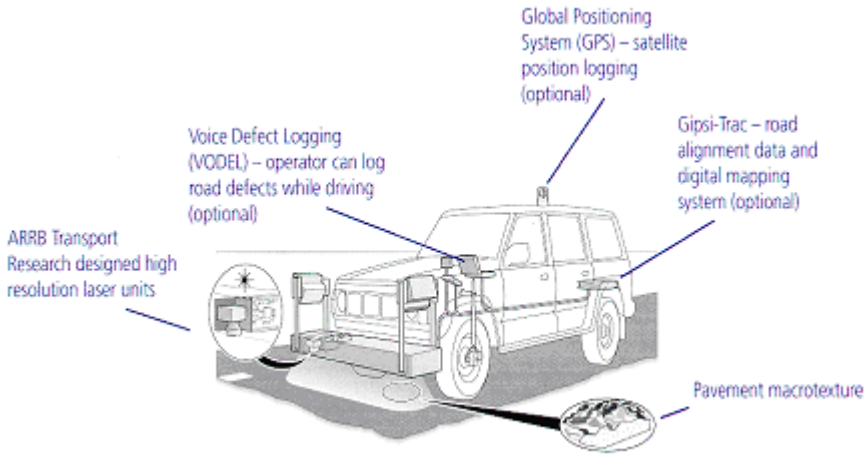


Figure 5: Multi-Laser Profiler Vehicle (Budras, para. 26)

One drawback to the MLP system is it requires a two person operation. One person must drive, while the other takes care of the computer. Another problem with this device is the price. It is very expensive, with an approximate cost of $75,000 (Budras, para. 25).

2.1.5 Road Roughness Indices

All the above devices will measure a longitudinal profile of the road. A profile measurement is a series of numbers that represent elevation relative to a reference. The data will then need to be analyzed to relate it to road roughness. The most common way to interpret these profiles is to reduce them down to summary roughness indices. Since there is no set standard, there are various procedures to reduce the

profiles.

One index used is known as a profile index. It is a number that is calculated from the numbers of the profile. The first step in reducing these values is creating an outline trace. This step averages out any spikes and other inconsistent data. These deviations are often caused by dirt, rocks, or grooves in the road. The next step in reducing the profile data is adding a blanking band. This band, often 5 mm wide, is placed on the trace to cover as much as the profile as possible. The remaining data above and below the blanking band will be balanced. This step removes any minor changes in the road smoothness (minimal roughness) (Achieving, para. 15).

The remaining profile data is used to determine the profile index. Figure 6 shows the profile index determination method for a particular stretch of road. The heights of the deviations above and below the blanking band are measured in millimeters and totaled. The deviations are only counted if they measure more than 0.6 mm and maintain this minimum for more than 0.6 m on the road. The sum of heights in a particular section is the profile index for that segment. The index is in units of mm/km (Achieving, para. 15).

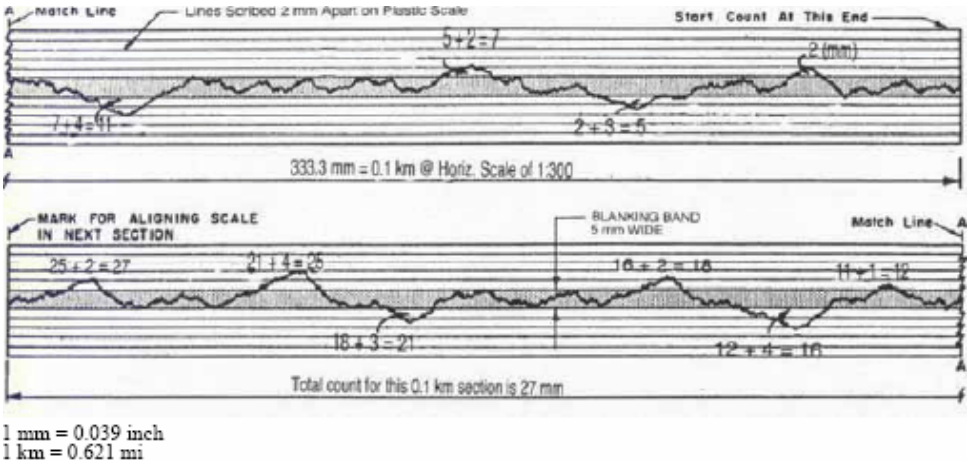


Figure 6: Determining the Profile Index from a Profile (Achieving, para. 15)

Another example of an index used is the International Roughness Index (IRI). The IRI is determined by a mathematical model known as the quarter car model. The model is simulated on the measured profile to calculate the suspension deflection of a mechanical system similar to that of a car (Sayers, page 45). This simulation set up can be seen in Figure 7.

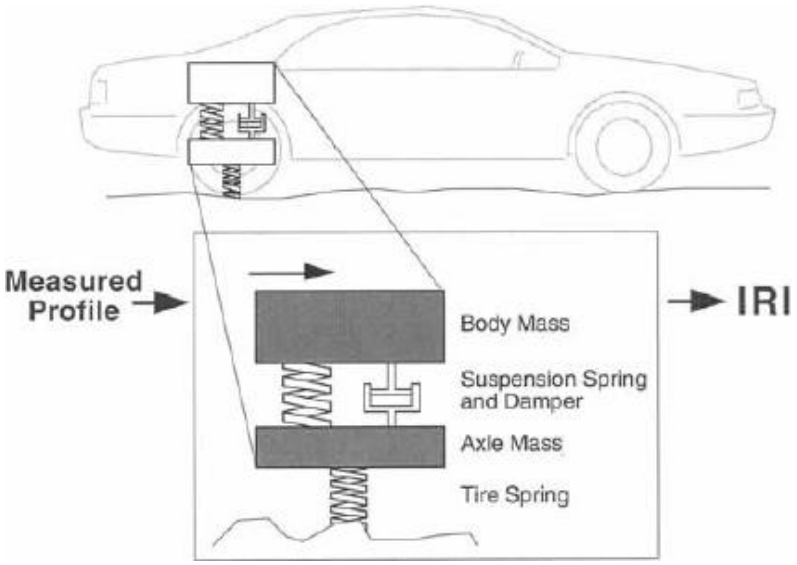


Figure 7: Quarter Car Model (Achieving, para. 17)

The totaled data from this simulation is divided by the distance traveled by the system during the measurement. This calculated value is used as the IRI. The units of the IRI are m/km or mm/m. Figure 8 shows IRI ranges of various roughnesses of roads (Sayers, page 45).

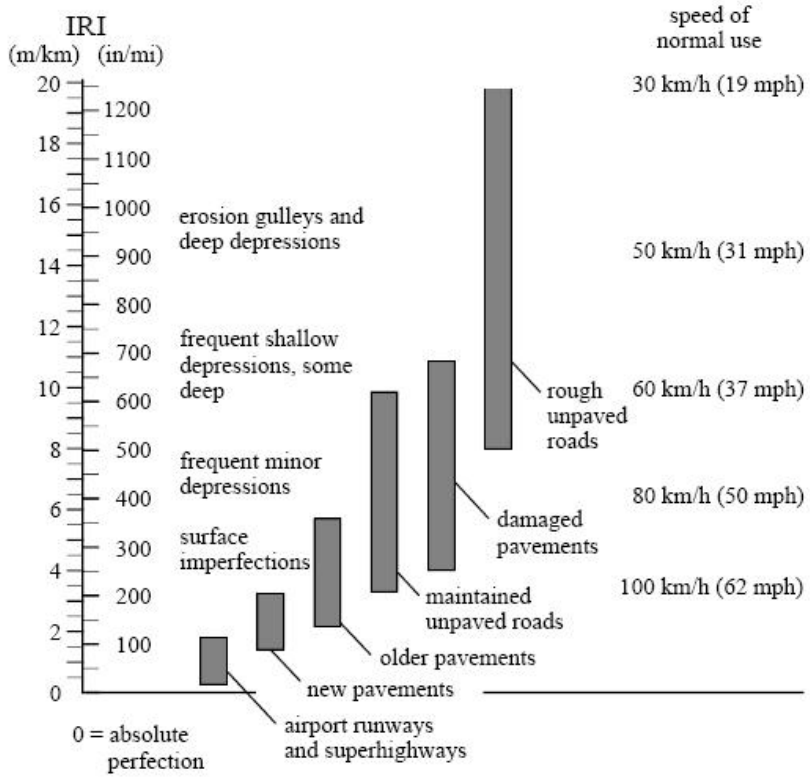


Figure 8: Graph of IRI Ranges (Sayers, page 48)

2.2 Past WPI Road Roughness Projects

This road roughness project has already been examined at WPI. A previous team had designed a device that could determine when the vehicle hit a pothole. Their design is shown below in Figure 9. This was achieved by using an accelerometer and a peak detector circuit. Whenever the accelerometer outputted a voltage above a certain threshold voltage, this data was saved to an onboard memory card. In addition to the accelerometer reading, the location of the pothole was saved into memory. A GPS module was used to determine the location. After the data was collected, it was uploaded to a mapping program. For each GPS location, a colored dot was placed on a map of the area driven. The color represented the severity of the pothole. With this map, cites could easily determine where potholes were located, and decide if they needed to be repaired.

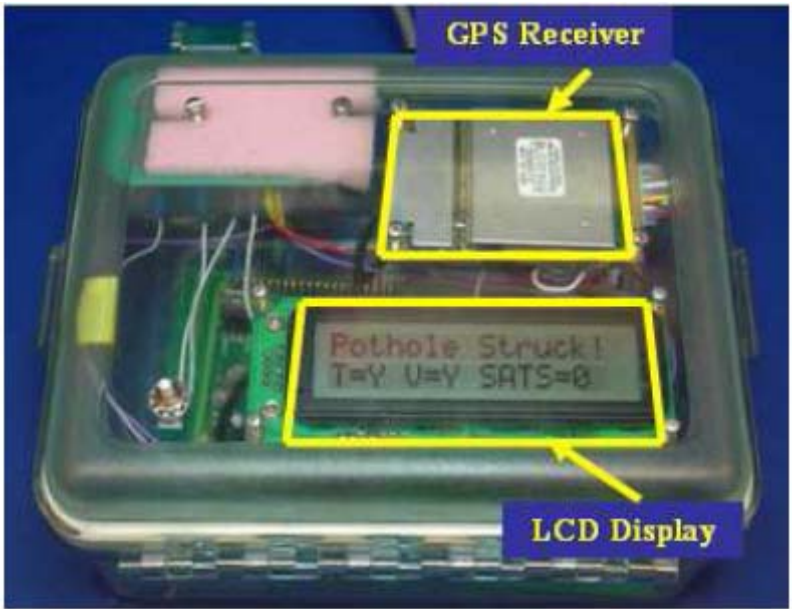


Figure 9: Pothole Detector (Angelini, page 40)

There were some drawbacks to this project. One problem was power. Since the device used a substantial amount of power, it had to be plugged into the cigarette lighter outlet to provide power. This limits the number of locations in the vehicle where the device can be placed. Another problem was the accelerometer had to be placed outside the vehicle to achieve accurate readings. The long wire needed to connect the accelerometer to the device unit is undesirable. This designed could be improved upon in several areas, such as power and size.

2.3 Accelerometer Basics

Defined, acceleration is the change of velocity in respect to time. Often, when describing the force exerted due to acceleration, the "g" is used as a unit of measurement. As a unit, a "g" is equal to the sea level gravity of the Earth, approximately 32.2 ft/s^2 or 9.81 m/s^2. Thus, the inherent acceleration experienced due to the gravitational field of the Earth is simply one g. In fact, rarely does an individual experience more than two g's of acceleration, and at seven times the acceleration of gravity a person will typically fall unconscious. Some reference levels for different g values are displayed in the table below.

Acceleration in Human Terms

- **What are some "g" reference points?**

Description	"g" level
Earth's gravity	1g
Passenger car in corner	2g
Bumps in road	2g
Indy car driver in corner	3g
Bobsled rider in corner	5g
Human unconsciousness	7g
Space shuttle	10g

Table 1: Reference "g" Points

A key note concerning acceleration is that it is a vector. Relative to a given axis, an acceleration has both a magnitude and a direction. Acceleration can therefore have negative values, where the sign simply indicates the direction of the acceleration. Due to being a vector, acceleration can be used to find the angle of tilt with respect to a given axis. Additionally, the change in acceleration can be used to determine how an object is moving. The former concerns static acceleration, while the latter pertains to dynamic acceleration. The acceleration vector can be used to measure tilt and analyze object movement.

2.3.1 Introduction to Accelerometers

The accelerometer is an electromechanical device which is designed to measure acceleration. Accelerometers can be constructed in a number of ways, making use of different phenomena and electrical effects. Due to the distinctions in accelerometer design, there are variable characteristics for accelerometers. There are many applications for accelerometers, including tilt sensors, vibration sensors and inputs to control systems. A multitude of designs, characteristics and applications exist for accelerometers.

2.3.2 Accelerometer Theory of Operation

Accelerometers can be constructed in many different ways, exploiting different electrical effects. In the past, accelerometers were designed mostly using piezoelectric properties of materials. Now, with the advent of MEMS technology, different, smaller accelerometers are being designed. Different construction techniques, pertaining to piezoelectric and variable capacitance effects are the primary types of designs for accelerometers.

Earlier accelerometers were designed based on the piezoelectric effect. These devices contained piezoelectric materials which, when stressed by acceleration forces, causes a voltage to be generated. By measuring this voltage, the acceleration can be obtained. Other designs are based on the piezoresistive effect, which is similar, with the resistance of the material changing as stress is applied via acceleration forces. Piezoelectric and piezoresistive effects have been used to design accelerometers.

A more recent design used in microelectromechanical systems (MEMS) concerns changing the capacitance of a material within the accelerometer. As a MEMS device, the accelerometer typically consists of a cantilever "proof mass" beam and some fixed plates. In a typical micromachined

accelerometer, the proof mass is on the order of 0.1 micrograms. The beam has a conductor which can move between the fixed plates to vary the capacitance. As a mechanical stress is applied via acceleration forces, the beam moves between the fixed plates and the capacitance is varied. An example of this implementation is that of Analog Devices, as pictured below.

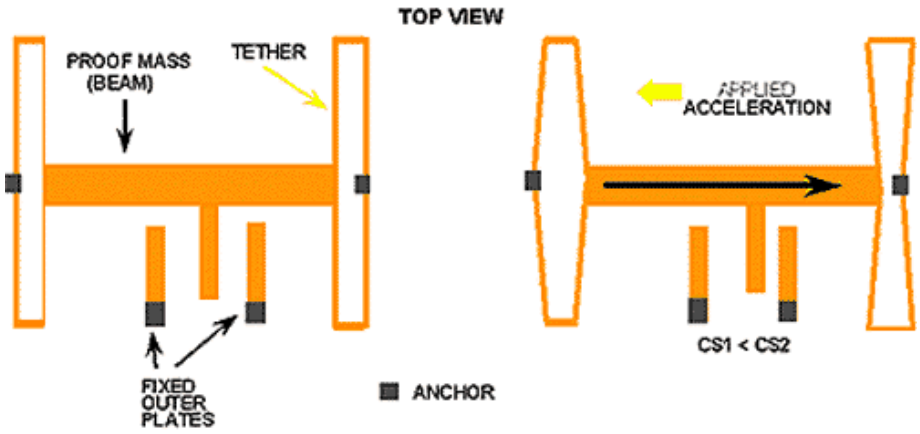


Figure 10: Analog Devices MEMS Accelerometer Implementation (Accelerometer Design, page 1)

The "fixed outer plates" seen in the figure above are set to create a differential capacitor. This differential capacitor measures the deflection induced by an applied acceleration. The changing capacitance varies the voltage output, a signal proportional to acceleration. In the case of an analog accelerometer, the amplitude of the output voltage signal is proportional to acceleration. The MEMS surface mount accelerometers are tiny compared to the larger piezoelectric accelerometers of the past, making use of a variable capacitor to measure the acceleration applied.

2.3.3 Accelerometer Characteristics

Many different characteristics differentiate the varied selection of accelerometers on the market. Both analog and digital accelerometers are developed, for different application integration. The number of axes, swing, sensitivity and bandwidth of accelerometers are primary characteristics. Additionally, as lower power applications are becoming more and more prevalent, accelerometers which operate on lower voltages and consume less current are becoming available.

One of the first decisions, when integrating accelerometers into a design, is often the choice between analog and digital. With digital outputs, typically a pulse width modulated output varies frequency and voltage high time to be proportional with acceleration. An analog outputting accelerometer, however, simply changes the amplitude of the output signal, proportional to acceleration. In general, it can be easier to work with an analog accelerometer if an analog to digital converter (ADC) is available. Otherwise, a digital accelerometer is necessary, which can bring along heavy computations is a PWM input is not available and a general purpose digital input must be used. The choice between analog and digital accelerometers is largely mitigated by the hardware available, with analog accelerometers typically considered easier to work with, if hardware permits.

Many characteristics of accelerometers depend on the application. These include number of axes, swing, sensitivity and bandwidth. Accelerometers can be single-, dual- or triple-axis, providing one, two

or three dimensions of measurement, respectively. Swing refers to the range of the acceleration that can be measured, with low-g accelerometers typically measuring ±1 to ±10 g's or more, and high-g accelerometers measuring up to ±100 g's or more. The sensitivity is a measurement of a change in the output signal, based on a given change in acceleration; generally, a higher sensitivity is better, since it can be easier to measure and allows for more precise readings. A measurement of how frequently a reliable reading from the accelerometer can be taken, bandwidth requirements can range from less than 20 Hz to several hundred. The number of axes, swing, sensitivity and bandwidth requirements vary from application to application.

As low power applications are growing in number, many accelerometers are designed for different levels of power consumption. Although some applications do not worry as much about power consumption, other applications, such as battery-operated devices, can make use of low voltage accelerometers. There are accelerometers designed to run at three volts or less. Additionally, current draw is important in lowering power consumption. Many newer accelerometers have current consumption on the order of hundreds of microamps or less. There are many new accelerometers designed to be optimal in low power applications where low voltage and current consumption are key.

Below is an abbreviated listing of some features for a selection of Analog Devices MEMS accelerometers.

ADXL Low-g Accelerometer Selection Table

Part#	# of Axes	Range	Sensitivity	Sensitivity Accuracy (%)	Output Type	**Max Band Width (kHz)	Voltage Supply (V)	Supply Current (mA)
ADXL103	1	±1.7g	1000 mV/g	±4	Analog	2.5	5 (3 to 6)	0.7
ADXL203	2	±1.7g	1000 mV/g	±4	Analog	2.5	5 (3 to 6)	0.7
ADXL204	2	±1.7g	620 mV/g	±4	Analog	2.5	3.3 (3 to 6)	0.5
ADXL213	2	±1.2g	30 %/g	±10	PWM	2.5	5 (3 to 6)	0.7
ADXL330	3	±3g	300 mV/g	±10	Analog	1.6(XY 0.55(Z	1.8 to 3.6	0.18 @1.8V
ADXL320	2	±5g	174 mV/g	±10	Analog	2.5	2.4 to 6	0.5
ADXL321	2	±18	57 mV/g	±10	Analog	2.5	2.4 to 6	0.5
ADXL322	2	±2g	420 mV/g	±10	Analog	2.5	2.4 to 6	0.5
ADXL323	2	±3g	300 mV/g	±10	Analog	1.6	1.8 to 5.25	0.18 @1.8V

ADXL High-g Accelerometer Selection Table

Part#	# of Axes	Range	Sensitivity	Sensitivity Accuracy (%)	Output Type	**Max Band Width (kHz)	Voltage Supply (V)	Supply Current (mA)
ADXL78	1	±35g ±50g ±70g	55 mV/g 38 mV/g 27 mV/g	±5	Analog	0.4	4.75 to 5.25	1.3
ADXL193	1	±120g ±250g	18 mV/g 8 mV/g	±5	Analog	0.4	4.75 to 5.25	1.5
ADXL278	2	±35g/±35g ±50g/±50g ±70g/±35g	55/55 mV/g 38/38 mV/g 27/55 mV/g	±5	Analog	0.4	4.75 to 5.25	2.2

Table 2: Key Features of Analog Devices MEMS Accelerometers (iMEMS, para. 3)

As seen in the previous figure, there are many characteristics associated with accelerometers. These include the choice between analog and digital, characteristics such as number of axes, swing (range), sensitivity and bandwidth, as well as power requirements.

2.3.4 Accelerometer Applications

There are many applications for accelerometers, ranging from tilt and vibration sensors, to dynamic measurement for use in a control system. The applications are based on the fact that acceleration is a vector and the accelerometer measures acceleration relative to its own "axis of sensitivity". The acceleration vector has a magnitude and direction relative to the axis of sensitivity of the accelerometer. When an accelerometer is used to measure dynamic acceleration, it can be used as an input into a control system, which may then correct the system based on changing conditions. By measuring the acceleration

vector, accelerometers can be used as tilt sensors. Accelerometers are also used in vibration measuring applications.

Accelerometers are used as inputs to a control system and in tilt sensing applications. In this case, the dynamic acceleration is monitored, which relates to the changing velocity and position of the system. The accelerometer readings are fed back into a control system which may then provide corrective measures affecting the system. Accelerometers can also measure tilt. A tilt sensor works by varying the outputs of a dual axis accelerometer. The outputs are proportional to the angle of tilt, thus, the tilt is measured. Accelerometers can be used as inputs fed into control systems and as tilt sensors.

Another use for accelerometers is in vibration sensors. This is because an accelerometer can be used to isolate mechanical vibrations from external sources. This is done by performing analysis on recorded results in the frequency domain to separate different sources. Additionally, filtering can be used to remove the effect of sources at particular frequencies to zone in on the vibrations from only one source.

There are many applications of accelerometers. Common applications include measuring dynamic behavior of a system to provide feedback to a control, as well as tilt sensing using dual axis accelerometers. Accelerometers are also used to measure vibrations in conjunction with a filter to isolate specific sources of vibration.

2.3.5 Accelerometer Summary

Accelerometers measure the acceleration vector. This is accomplished by making use of the piezoelectric phenomena or using a voltage producing differential capacitance MEMS device, among other ways. The choice of analog or digital output accelerometers, as well as considering the number of axes, swing, sensitivity, bandwidth and power requirements depend on the application and a wide range of all the characteristics is present for a varied selection of accelerometers. Used as inputs to control systems, tilt and vibration sensors, as well as in many other functions, there are countless applications for accelerometers.

2.4 GPS Basics

The Global Positioning System (GPS) is a navigation system that uses satellites and radio signals to determine locations on the Earth. This system was created by the United States Department of Defense by launching the first satellite in 1978. The system's official name was NAVSTAR GPS (Navigation Signal Timing and Ranging Global Positioning System). GPS is available to the civilian population as a public good (Global, para. 2).

By 1994, there were 24 GPS satellites orbiting the Earth at an altitude of 12,600 miles. Only 21 satellites are active at a time. The other three are used as spares. As the years went on, more satellites were added. Each satellite is built to last about 10 years, with replacements constantly being built. The satellites are spaced so that at any location on Earth, at least four are visible in the sky. Figure 11 shows the satellite positions with respect to the Earth (What is GPS, para. 6).

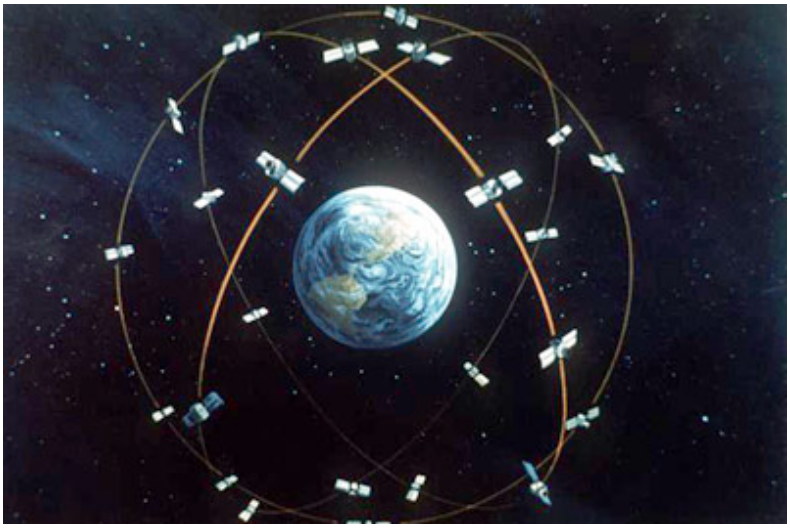


Figure 11: Satellite Positions (How GPS Receivers Work, para. 5)

2.4.1 GPS Signals

Two radio signals are transmitted by the GPS satellites that are used to determine location. They are known as L1 and L2. L1 uses the frequency of 1575.42 MHz, whereas L2 uses the frequency 1227.60 MHz. Civilian GPS receivers use the L1 signal. These signals travel by line of sight and will not travel through solid objects (Global, para. 13). Three different types of data are sent in these two radio signals. The first piece of data is known as the almanac. The almanac contains information about the status of the satellite, as well as the current time and date. The second piece of data is known as the ephemeris. The ephemeris contains orbital information about the satellites, such as where the satellite should be at any time throughout the day. This allows the GPS receiver to calculate the satellites' position at any time. The last piece of data is a pseudorandom code. This data is a unique code that allows the satellite to be identified. It is used as an I.D. (What is GPS, para. 9).

2.4.2 Triangulation

A GPS receiver calculates its location by using a technique called triangulation. Triangulation is a method of determining the relative position of an object by using triangles and geometry. In order to triangulate its location, the receiver first has to determine the distance from itself to a satellite. This is accomplished by using the travel time of the radio signals. The receiver measures the time delay between when the satellite sent out the signal to when the signal was received. This delay is used to determine the distance. The receiver also knows the position of the satellite from the ephemeris data sent in the signal. From these two pieces of information, the time delay and satellite position, it is known that the receiver's position must lie on the surface of an imaginary sphere with a radius equal to the distance between the receiver and the satellite. This process is repeated at least twice to create three or more of these imaginary spheres. The receiver is located at the intersection point of the spheres (How GPS Works, para. 1).

2.4.3 GPS Inaccuracies

Unfortunately, since the satellite clocks are not precise, there will be errors in the GPS calculations. The receiver can only roughly estimate its distance from the satellite. Figure 12 demonstrates this problem. The dotted lines represent the actual location of the spheres and the solid lines represent the

estimated value. Since the three spheres do not intersect at one point, the GPS receiver slightly alters the size of the spheres until an intersection is found. This can cause inaccuracies up to several meters (GPS, para. 1).

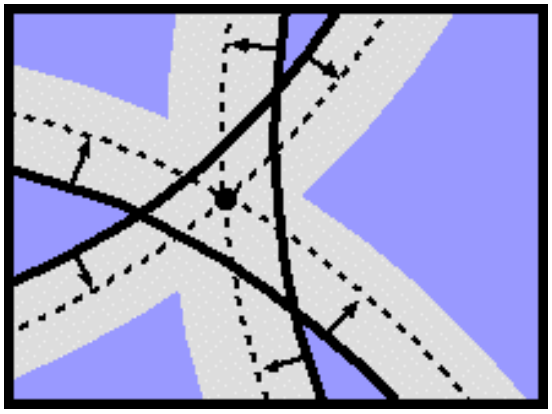


Figure 12: Inaccuracies in Sphere Sizes (GPS, para. 5)

There are several other problems that can affect the accuracy of the GPS receiver. One potential problem is signal scattering. This happens if the satellite signal is reflected off of a large object, such as a building. This will cause the signal to be slowed down and increase the delay time between the receiver and the satellite. Another problem is troposphere and ionosphere delays. When the signal passes through the atmosphere, the troposphere and ionosphere layers will cause the signal to slow down. Again, this will increase the calculated signal delay. A third possible problem is errors in the ephemeris data. Since the ephemeris data is sent out every 12.5 minutes, the GPS receiver could be using outdated data. The satellite might not be in that specified location at that time. This will affect the triangulation process (What is GPS, para. 13).

2.4.4 Techniques to Improve Accuracy

Many solutions exist to help reduce these inaccuracies. One solution is Differential GPS (DGPS). This system improves the accuracy of the GPS receiver by adding a local reference station to compute corrections to GPS parameters, error sources, and resultant positions. Since the reference station knows its exact location, it can determine errors in the satellite signal. It accomplishes this by comparing the ranges of the signals received by the receiver to the actual ranges calculated from the reference station. These corrections are then sent to the receiver to remove most of the inaccuracies (Differential, para. 1). Another solution is Wide Area Augmentation System (WAAS). As with DGPS, reference stations are used to calculate the difference between the GPS signal and the actual signal range calculated by the station. Two master stations collect this data then broadcast it through one or two satellites with a fixed position over the equator. GPS receivers compatible with WAAS can read this signal and correct its errors (What is WAAS, para. 3).

2.4.5 GPS Summary

For our project, we will be using a GPS module in our design. This module contains a digital signal processor, real-time clock, UART, and memory all in one small package. This package is ideal for

our purposes because it performs all the previous mentioned GPS tasks in a compact size. It will track up to twelve satellites, determine its location using triangulation, and save the results to memory. It is ideal for embedded projects.

2.5 Background Summary

This chapter discussed the current methods cities use to measure the roughness of roads. Many different devices were shown, each with their own drawbacks. The major problems were size and cost. Most of the devices required a separate vehicle in which to operate. This increases the cost significantly, which may scare away potential city investments in that product. Our device will try to overcome these problems to make it more appealing to the cities. Also, the theory behind accelerometers and GPS receivers were described. This will help us in our product design.

3 METHODS

This section is concerned with defining the methods followed to accomplish our project. First, the specifications required for the project will be described. Second, the process in which the project was accomplished will be detailed.

3.1 Specifications

Once we had finished researching the topic and past devices, a list of specifications for our project was devised. These specifications acted as a guide for our design and component selection. Every decision made throughout the projects took these provisions into consideration.

The first specification was the device had to be inexpensive. Since most cities do not have a lot of spending money, a low cost device would be ideal. City officials would be more apt to purchase a product with a reasonable price. In addition, the cost needs to be low so the cities can afford multiple units. More units will enable the city to better track the condition of their roads, since more roads will be traveled. For our project, we tried to stay under $250 for a total cost.

Another specification was the device needed to be small. Since this device would be placed inside municipal vehicles, minimizing the occupied space would be optimal. The device should not be intrusive and easily stored on a seat or windshield. The device should be no bigger than 6.5 in. (L) x 7.5 in. (W) x 3.5 in. (H), which was the size of the previous WPI road roughness project.

A third specification was the device needed to be in one enclosure. One problem with the previous WPI pothole detector project was the accelerometer had to be placed outside the vehicle. Our design needed to bring the device into one small enclosure.

A fourth specification was low power. Since the device was designed to run on batteries, a low power system would be the best. This would extend the life of the batteries. Fewer batteries would need to be purchased, which would cut down on the maintenance cost. A target power level for our device was 100mA or less.

Another specification for our device was reliability. Before cities make an investment in our device, they need to know if they can depend on the product. It should be able to be simply placed inside the vehicle and forgotten about until the data needs to be uploaded to a computer. Also, minimum maintenance should be required.

A sixth specification was recording length. We needed to make sure that the memory would be able to hold all the data during a recording period. For our design, we decided to have a recording length of one week. The device would be placed inside a municipal vehicle at the beginning of the work week and would record data while the vehicle drove around. At the end of the work week, the data would be uploaded to a computer and analyzed.

Another specification was accuracy. We needed to decide how accurate the device should be. Since we were measuring the roughness of the road, it was not necessary to achieve an exact location. Finding the general area would be sufficient. It was decided that an accuracy of up to 15m would be acceptable. This value was around the average accuracy for GPS modules without error correction capabilities.

3.2 Process

The diagram in Figure 14 shows the process used to accomplish the project. First, we came up with a project goal (Chapter 1). This goal was the basis for our project. Next, background research was conducted to learn about previous roughness detection methods (Chapter 2). Various devices were examined to determine what improvements were necessary. Afterwards, a list of specifications was developed (Chapter 3). These provisions aided us in our decisions made throughout the project. Then, the design process began. We broke down the system into modules and designed each components (Chapter 4). The component design was broken into three parts, shown in Figure 13. First the requirements and specifications for each component were developed. Next, the various options for each component were compared to choose the best option for the project. Then, each component was tested individuality to verify functionality.

Figure 13: Component Design Block

Once each module had been tested, they were all placed together on a soldered breadboard. The entire system was then tested as a whole to verify the hardware and software functioned correctly. This testing included a field test to see if the device was able to locate the rough spots in the road. Once the first prototype was working, we then designed a PCB. After various revisions, the PCB was made, populated, and tested. The testing was similar to that of the breadboard prototype. Once the PCB was working, an enclosure was chosen to house the board. Then, a final field test was performed.

3.3 Summary

After completing the specifications and process, we were able to start designing the device. The specifications gave us a guideline of what to look for in components. We can rule out components that do not fit our expectations. Also, the described process gave us a method of tackling our design. Following the steps will assure a successful project.

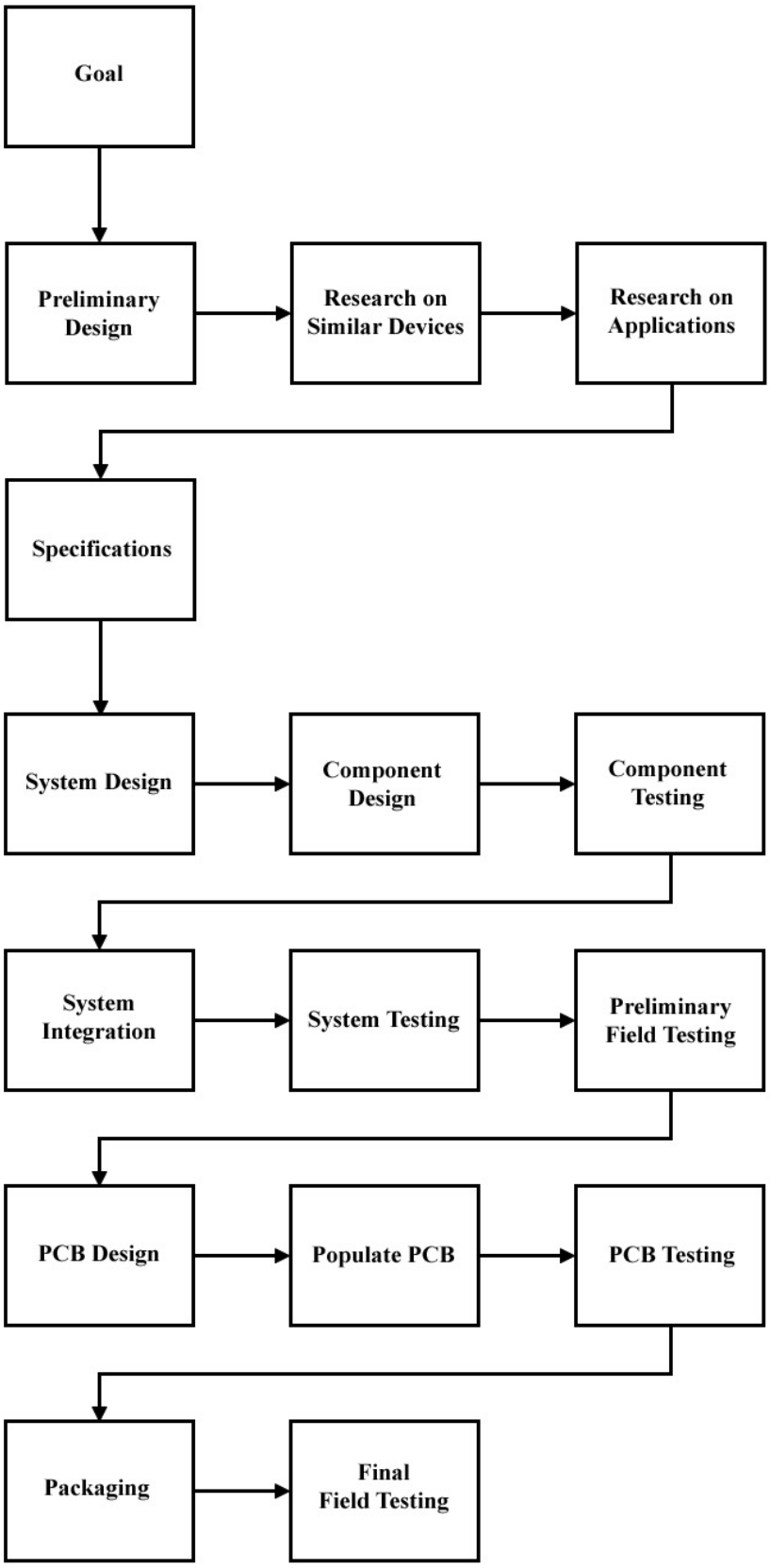


Figure 14: Process Block Diagram

4 SYSTEM DESIGN

This section will present the overall system design of the project. Figure 15 shows the block diagram for the system. The overall design was broken down into several subsections. These sections include the power module, accelerometer module, GPS module, microcontroller module, memory module, USB interface module, and LCD module.

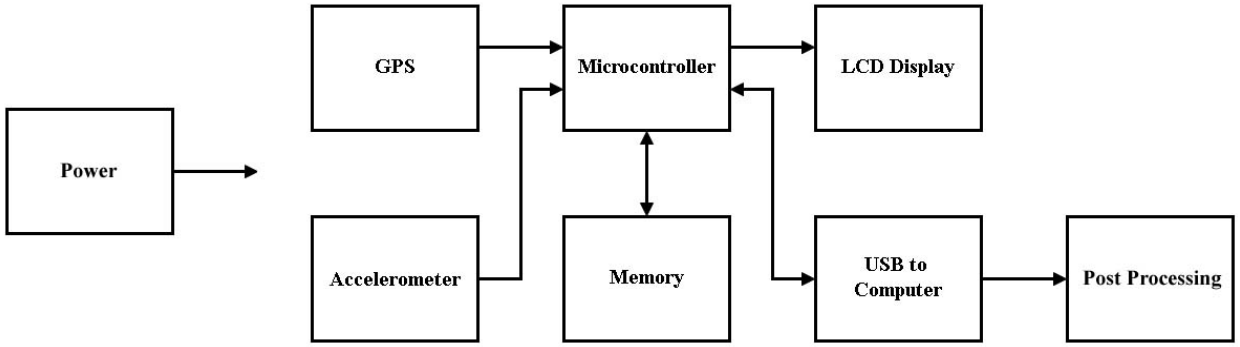


Figure 15: System Block Diagram

4.1 Power Module

The power module was the first module of our design. The purpose of this module was to provide power to each component of the project. A more detailed breakdown of the power module is shown below in Figure 16.

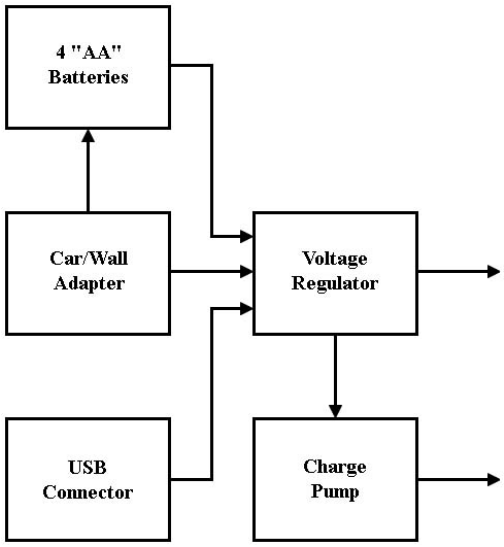


Figure 16: Power Module

The main source of power for the device was provided by batteries. Batteries were chosen to help keep the device more autonomous and offer flexibility in placement inside the vehicle. With batteries, the device did not need to be located near a cigarette lighter to provide power. In addition, the types of batteries were chosen to be nickel metal hydride (NiMH). By using rechargeable batteries, the cost of replacing batteries would be greatly reduced. A total of four "AA" NiMH batteries were used to power the device. A battery charger circuit was designed to recharge the batteries. The MAX712 was chosen for this circuit. The MAX712 was a fast charge control IC which charged by forcing a constant current into the batteries. Charging began when power from a wall or car adapter was applied to the charging circuit.

Another subsection of the power module was the car or wall adapter. The main purpose of this part was to charge the batteries. Whenever an adapter was plugged into the device, the batteries would start the charging process. While in charging mode, the car or wall adapter became the main source of power for the device. MOSFETS were used to disconnect the batteries and allow the adapter to provide power. This was done so the batteries would charge faster and more efficiently.

The USB connector was another method of powering the device. Whenever a USB cable was connected to upload the stored data, the same MOSFETS mentioned above would disconnect the batteries. The USB was then the main source of power for the device. This method was chosen to help extend the life of the batteries.

The main system voltage subsystem provided a regulated voltage to power the digital and analog devices of the embedded system, with the exception of the LCD. The systems which it drove included the microcontroller, GPS, accelerometer, and serial flash memory. These were all powered with 3.3V. The LCD required 5V, which was provided by a charge pump.

A number of design choices were possible for the main system voltage subsystem. The main decision was between a linear and switched-mode regulation topology.

Switched-mode power supplies, or SMPS systems, regulate from DC to DC via step-up, step-down or step-up/step-down topologies. SMPS converters such as a buck or a buck-boost allow for high efficiency and a wider operating range, even less than the output voltage, in the case of step-up/step-down topologies.

The main disadvantages of SMPS systems over LDOs are added complexity, increased size, high part count, higher cost, lower stability, and increased quiescent current consumption. The increase in complexity, size and cost are from the fact that switch-mode regulation uses a controller IC to drive an external switching device, usually a MOSFET, which charges inductors and capacitors to set a stable output voltage. Stability is an especial concern with switch-mode regulation, due to the fact that in order to stabilize the feedback compensation loop, application-specific external discrete components must be chosen. Once properly compensated, SMPS systems can be just as stable as LDOs, but compensation is a difficult process. Additionally, there is typically much more output ripple, due to the switched-mode design, causing more noise to be injected into the supply lines. The relatively high quiescent current consumption, on the order of 20 mA or more, can make the SMPS less efficient than LDOs in low conversion ratios (such as going from 4V to 3.3V).

Linear-mode regulation works by effectively shunting excess voltage to provide a stable output voltage. A Zener reference pulls the input voltage down and an internal MOSFET is switched on and off with a PWM signal to provide regulated output voltage. The use of linear regulators is very straightforward, typically requiring only some input and output capacitance to stabilize the output voltage during input line transients and output load transients. An important characteristic of linear-mode

regulators is the dropout voltage, defined as the minimum difference in input and output voltages required for regulation. Most modern linear regulators are described as low-dropout voltage regulators, or LDOs (Kolanko, para. 1).

The main advantages of LDOs over SMPS systems are simplicity, small size, low cost, low part count, low electro-magnetic interface (EMI), and very low quiescent current consumption. The EMI could get coupled into signals, especially analog ones, like the accelerometer, and the clock like of the microcontroller, causing stability issues. Additionally, the inductor that SMPS systems require could itself be a cause of EMI. LDOs require no inductor, just a few capacitors, which can be very small. The main draw for low-power design is the very low quiescent current consumption used by LDOs, which, when used with low conversion ratios, can actually make LDOs more efficient than SMPS systems. An LDO was chosen to power the device.

4.1.1 Power Module Requirements

A source chooser was first designed for the power module. The purpose of the source chooser subsystem was to dynamically switch the power source, which supplied the input voltage to the main system and LCD driving voltage subsystems. The design goals for this subsystem were as follows:

- Minimize losses in general, but especially from the battery
- Use the battery as the input voltage supply only when no other sources are present
- Disconnect the battery from the input voltage supply while it is being charged
- Support power over USB
- Allow powering via the car/wall adapter
- Provide a constant input voltage supply, even during source switching
- Prioritize the source to use as input voltage supply when more than one is present

The priority of the sources to use was as follows:

1) Car/wall adapter
2) USB
3) Batteries

Therefore, when no other sources were present, the batteries were used. If the USB became present, the batteries would be disconnected and the input voltage would be drawn from the USB. If the car/wall adapter became present, the batteries would be disconnected, and input voltage would be drawn from the car/wall while the batteries were being charged. Additionally, the USB would be disconnected from the input voltage supply if all three of the sources were present, thus prioritizing the car/wall adapter over the USB.

4.1.2 Power Module Circuit Description

First, the battery charger circuit will be described. The charger circuit is shown below in Figure 17. The main component of the circuit was the MAX712. This controller was designed to fast charge Nickel Metal Hydride (NiMH) batteries from a DC source by injecting a large current. The MAX712 was always in one of two states, fast charge or trickle charge. The device entered fast charge when either a DC source was applied, or when the batteries were installed (DC source already connected). The device was

able to terminate fast charge three different ways. It could terminate fast charge by detecting zero voltage slope, battery temperature, and a timer. Once full charge was determined, the current was reduced to trickle charge.

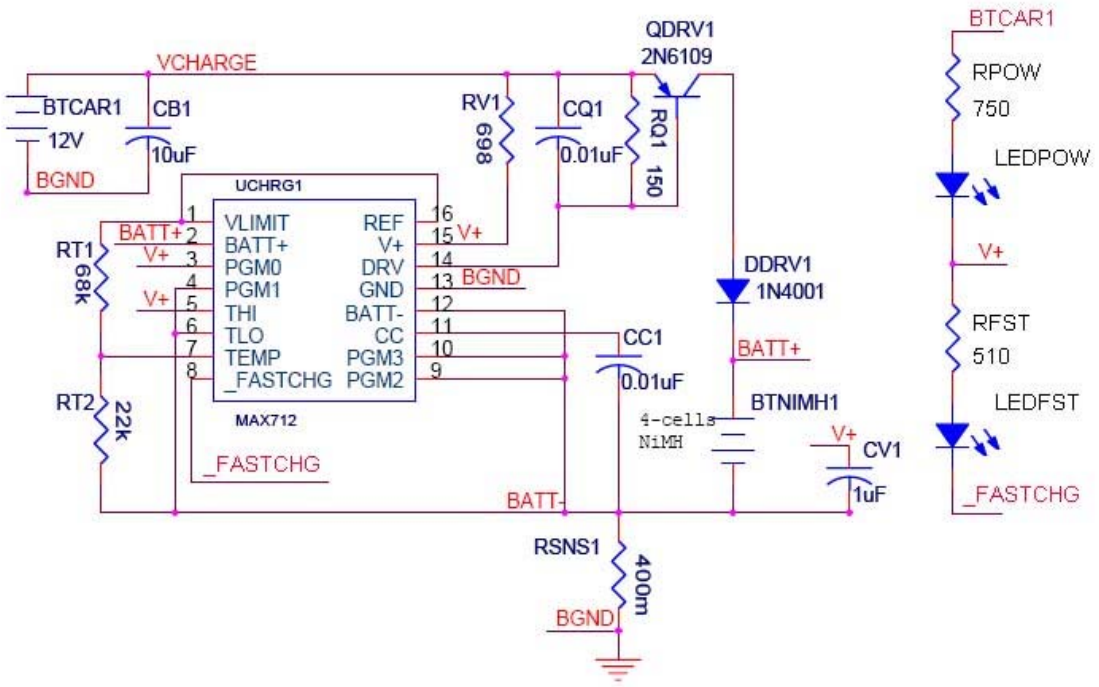


Figure 17: Battery Charger Circuit

The battery charger circuit was powered by 12V DC. This voltage was provided by the car battery or a wall adapter (BTCAR1). The capacitor, CB1, was included to help filter the input voltage. The actual MAX712 device was powered by an internal +5V shunt regulator (V+ pin). The resistor, RV1, was chosen to be 698Ω to limit the current into V+ to 10mA. This was the recommended current given in the datasheet.

When the 12V was applied to the circuit, CV1 charged through RV1. When the capacitor reached 5V, the shunt regulator adjusted V+ to 5V, which started fast charge. The DRV pin sunk current for driving the PNP current source. The diode was used to prevent current from flowing into the DRV pin when the 12V was not connected.

The values for CQ1 and RQ1 were chosen by selecting CC1, which was a compensation capacitor for the input. CC1 was chosen to be 0.01µF, which gave a bandwidth for the current regulation loop equal to180kHz (BW = gm/CC1, where gm = 0.0018A/V). From this bandwidth, RQ1 was chosen to be 150Ω and CQ1 to be 0.01µF. Also, this bandwidth needed to be less than the pole frequency of the PNP. The pole frequency was found by dividing the gain-bandwidth product by the DC current gain. For the 2N6109, the pole frequency was 333MHz (10MHz/30). This value was less than the bandwidth.

The diode needed to be able to handle the fast charge current. Since the batteries were 2500mAh and the charge rate was C/4, the expected current was 0.625A (2500mAh/4h). The 1N4001 diode was

chosen because it could handle up to 1.0A of current. From this current, the sense resistor, RSNS1, was chosen to be 0.4Ω (0.25V/0.625A).

One method of detecting full charge is the battery temperature. However, for our design, we decided not to use this feature. To disable this method of detection, THI was connected to V+ and TLO was connected to BATT-. Also, a 68kΩ resistor from REF to TEMP, and a 22kΩ resistor from BATT- to TEMP were needed.

The methods of detection used in this circuit were zero voltage slope and a timer. The MAX712 has an internal ADC that determines if the battery voltage is rising, falling, or the same. To implement the zero voltage slope and timer features, pins PGM0 – PGM3 were used. PGM0 and PGM1 set the number of cells to be charged. Since four batteries were being charged, PGM0 was connected to V+ and PGM1 was connected to BATT-. PGM2 and PGM3 set the maximum time allowed for fast charge. Since the charge rate was C/4, the maximum time allowed (264 minutes) was used. Also these pins were used to enable zero voltage slope detection. Both PGM2 and PGM3 were connected to BATT-.

The output pin, _FASTCHG, was used to determine when power was applied to the circuit, and when fast charge was occurring. This pin sunk current when the device was in fast charge. When trickle charge began, the pin stopped sinking current. For testing purposes, two LEDs were used to signal when 12V was applied (LEDPOW), and when fast charge was taking place (LEDFST).

Next, the input power circuitry will be described. A schematic for the power circuitry input stage can be seen below.

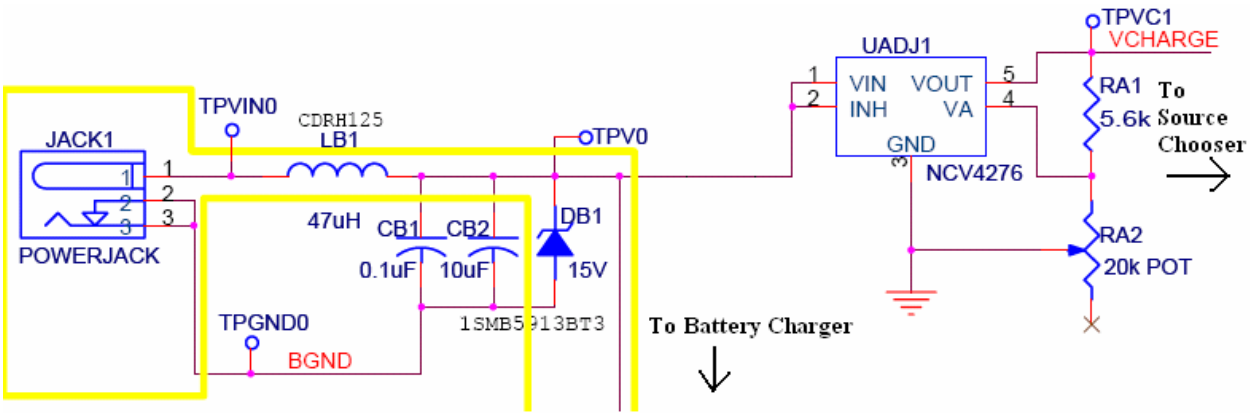


Figure 18: Input Power Circuit

A power jack connector was used to plug in both the car and wall adapter. This circuitry used a very low DC resistance (DCR) power inductor from Sumida, the CDRH125 with 47µH of inductance. This inductor was chosen due to very low DCR, small surface mount package with some shielding, and low cost. It was rated for 1.8A, which was more than enough to power the battery charging circuitry, and had a low DCR of typically 58mΩ (maximum of 78mΩ). The part can be used in automotive environments.

Two input capacitors were used on the input line to filter the signal. A 0.1µF ceramic capacitor was used to filter out high noise, and a 10µF tantalum was used to improve dynamic line response, such as on a line transient. Additionally, a 15V Zener diode in parallel with the capacitors provided a very fast response to over-voltage conditions. The Zener used was the 1SMB5913BT3 from ON Semiconductor. It

is intended for automotive applications and rated to withstand up to 200V, and up to 3W continuous DC power. On a transient event, which the Zener was intended for, the power would only be surged, so the power handling capability was much higher. The Zener had a breakdown voltage of 15.2V. This Zener was chosen due to its small SMB surface mount package, 15.2V breakdown voltage, power rating, and 1μA leakage current.

The node marked with the test point, TPV0, was used to power the battery charger circuitry. The rest of the board could not handle the relatively high voltages that the power jack would draw. Therefore, a pre-regulator was used to step the voltage down from the car voltage (nominally 13.2V) or wall voltage (nominally 12V) to approximately 5V. An adjustable regulator could be used to compensate for losses that VCHARGE was subject to in the source chooser circuitry. However, the adjustable version of the regulator we wanted was not available, so we are currently using a 5V output version.

The NCV4276 was used, due to previous experience with the part and its excellent characteristics. The main characteristics of interest for the pre-regulator were the following:

- Surface mount package
- High thermal dissipation
- Large input voltage range
- Excellent transient response

The NCV4276 was available in a DPAK-5 surface mount package, which, compared to the LDO and other parts, was enormous, but relative to the overall size of the board, was not too large. This package can dissipate a lot of heat, making it ideal for automotive applications where the input voltage could be typically 14V and spike as high as 40V. Additionally, a 1" spreader board was built on the PCB to help dissipate heat, as instructed in the data sheet. This brings the junction-to-ambient thermal resistance down to 46.8 degrees C/W, which was very low, meaning there was a lot of thermal dissipation. The maximum power dissipation was calculated, assuming an absolute worst quiescent current of 100mA. This was very unlikely (depended on ambient temperature due to BJT passive element). With an average load current requirement of 50mA (plus 70mA with backlight enabled), a 14V input, and a 5V output, the dissipation was calculated as seen below:

$$Pd = (14V - 5V) * (100mA + 50mA + 70mA) = 1.98W$$

At this power dissipation, the calculated temperature in the junction would be approximately 92.7 degrees C above ambient, which with an ambient temperature of 25 degrees C would yield approximately 117.7 degrees C. This was below the rated 150 degrees C junction temperature, so the part would act fine.

In reality, this was a worst case scenario, since the quiescent current was likely to not be nearly as high. The car battery voltage should be nominally 13.2V, the average load current could be less, and the backlight current would be zero with the backlight turned of, as would be typical operation.

We had run the NCV4276 part with a 30V input and maximum load at room temperature with no issues. The NCV4276 could accept input voltages as high as 45V. It could source up to 400mA of current. The transient response was optimized to respond very quickly to an input voltage transient or an output load transient. This made the NCV4276 work very well for our application as a pre-regulator.

For our application, the NCV4276 may be overkill. A smaller regulator may be able to handle the

thermal dissipation and use a MOSFET passive element, thus having a much lower Iq which does not rise up to as high as 100mA as temperature increases. Additionally, more heatsinking, including a larger copper pour area on the PCB, could provide better thermal characteristics. A step-down SMPS regulator could also be considered, in lieu of a pre-regulator/LDO combination, which would largely avoid any power dissipation and thermal issues due to the very high power transfer efficiency. However, this would increase the overall size of the system, since a power inductor would be required, which is approximately the size of the DPAK-5 the NCV4276 uses.

Finally, the source chooser will be described. The schematic is shown below.

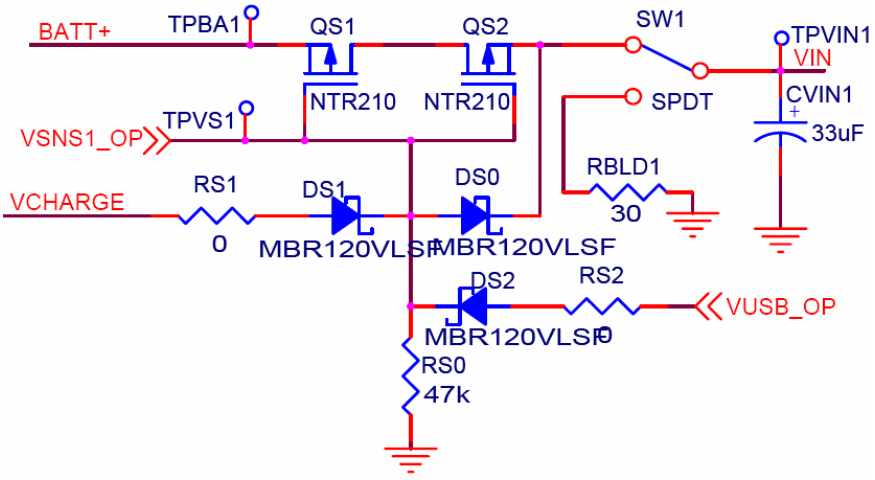


Figure 19: Source Chooser Circuit

When no other source was present, the batteries were used as the input voltage supply. In this case, the gates of both QS1 and QS2 were pulled to ground through RS0. Since QS1 and QS2 were PMOS FETs, they were each in the active region of operation, effectively acting as a resistor. This resistance was equal to the r_DS(on) of the PMOS FETs. The NTR2101P from ON Semiconductor was a very low r_DS(on) power PMOS, with a specified r_DS(on) of approximately 39mΩ and 4.5V Vsg. Two were used in series, due to the parasitic drain-to-source diode in power MOSFETs. By putting two in series, current was blocked in both directions, allowing the battery to be fully disconnected from the input voltage supply when another source was present. However, the equivalent resistance was raised to approximately 78mΩ. Our Vsg was typically 4.8V, which would mean that r_DS(on) was actually less. However, going with a worst case calculation and assuming r_DS(on) was effectively 78mΩ for the two in series, the calculated losses for a typical 50 mA load is as follows:

$$Vds = r\_DS(on) * Ids = 78m\Omega * 50\text{mA} = 3.90\text{mV}$$

When another source was present, either DS1 or DS2 was forward biased. The USB voltage was approximately 5V. The diodes were all low forward-voltage Schottkey diodes. This was to minimize the voltage losses, since a typical diode could have a Vf as high as 0.7V. The MBR120VLSF from ON Semiconductor was specified to have a maximum Vf of 0.275V at room temperature with a 100mA load. The measured voltage drop was actually on the order of 100 to 200mV. With the two diodes in series, this

corresponded to a measured voltage drop of less than 400mV.

The NCV551 from ON Semiconductor was chosen as the LDO. This part was designed for harsh automotive environments and had an incredibly low dropout voltage of only 40mV and a maximum of 150mV for the 3.3V version. The quiescent current consumption (Iq) was ultra low, typically at only 4.0µA. The part could source up to 150mA of current, which was more than enough for our application. The input voltage could be as high as 12V, which was much higher than necessary for our application. The extremely low dropout voltage and Iq, as well as very small TSOP-5 surface mount package, made the NCV551 our choice for LDO.

The LDO schematic is shown in Figure 20. The input capacitor, CLDO1, was a 10µF tantalum surface mount capacitor. This was recommended in the NCV551 datasheet and was used to quell input line transients. The output, VOUT, originally had no output capacitor, since the DREF and AREF lines had capacitors. However, on the PCB, an output capacitor was added, 4.7µF (tantalum) to improve load transient response. The two 0Ω resistors, RZ4 and RZ5, separated the digital and analog 3.3V references, DREF and AREF, respectively. This provided isolation between the digital and analog sources, which decreased digital/analog noise coupling.

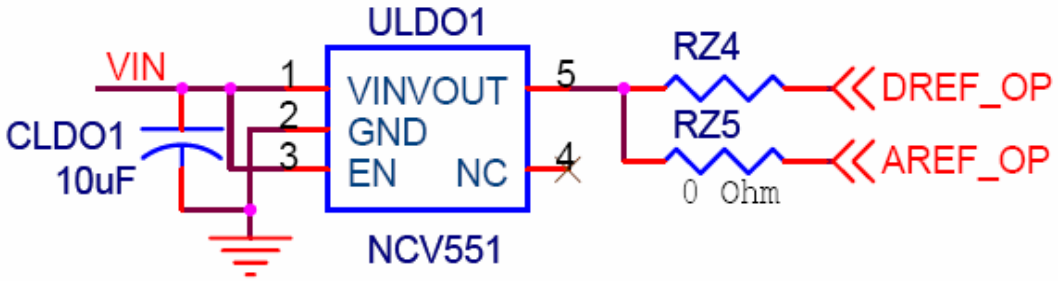


Figure 20: LDO Schematic

The LCD required a 5V reference. While running off of 4 NiMH batteries, the maximum voltage was expected to be 4.8V, and this decreases as the batteries are discharged. Therefore, some sort of voltage step-up was required to drive the LCDs. A SMPS converter was one choice, but a charge pump regulator was decided on for the following reasons: ease of use, low parts count, low cost, small size, and no inductors. The final point, that a charge pump requires no inductors, was important to reduce electro-magnetic interference (EMI), size, cost, and complexity.

The MAX619 charge pump doubler/regulator was chosen due to high availability, low cost, small SOIC 8-pin surface-mount package, low pin count, low part count, and integrated 5V regulator (as compared to simple charge pump ICs). The MAX619 regulated to 5V from inputs ranging from 2.5V to 5V, and sources up to 50mA. It doubled the input voltage (and halved the current) by using just two switching capacitors. The capacitors were charged up in parallel and then switched to series to get double the input voltage. This voltage was then regulated to 5V. The MAX619 can provide a stable 5V output with an input voltage of only 2.5V. The power was regulated, not just the voltage, as with a linear voltage regulator. Therefore, for the maximum output current of 50mA, the input current would have to be 100mA at an input voltage of 2.5V, not counting quiescent current losses.

A schematic for the charge pump circuit is shown in Figure 21. The switching capacitors, CCP1 and CCP2 were 0.22µF surface mount ceramic capacitors. These were suggested in the application note

of the MAX619 datasheet. The input and output capacitors, CCP3 and CCP4, were 10µF tantalum surface mount capacitors, which were used to quell input line transient and output load transients, respectively. The surface mount 47kΩ resistor, RCP1, was used to pull down the SHDN pin, which enabled the part. The SHDN pin, when low, put the MAX619 in a low-power, non-regulating state, like the EN pin on other parts. This pin was also connected to the microcontroller. The intent was that the microcontroller can turn off the charge pump regulator when the LCD was not required in order to reduce power consumption.

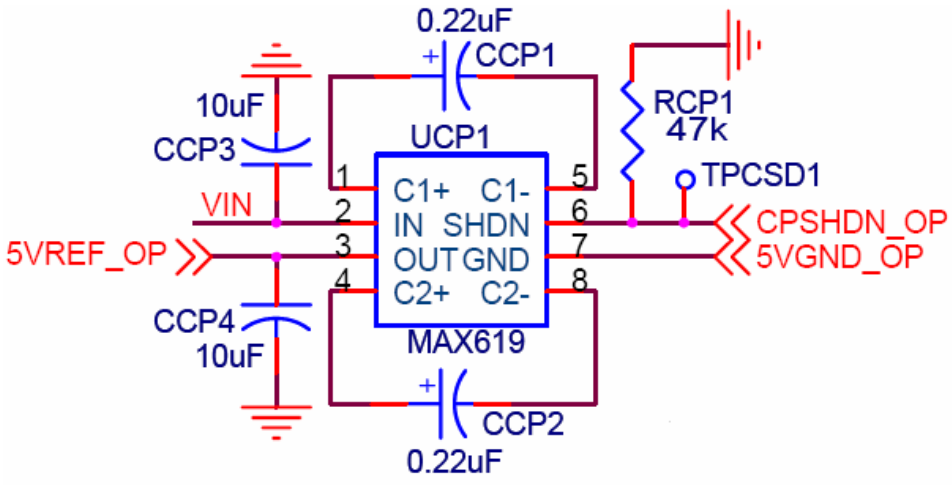


Figure 21: Charge Pump Regulator Schematic

4.1.3 Power Module Testing

To test the charger circuit, we first discharged the batteries for a set amount of time. This way we knew how many milliamp-hours had be discharged. In the experiment, the voltage was regulated to 3.26V across a load of 62.2Ω. From these two values, the current flowing through the load was determined to be 52.4mA. The batteries were discharged for 32.25 hours. This corresponded to 1690mAh drained from the batteries.

To charge the batteries, 12V from the laboratory DC power supply was applied to the circuit with the batteries already connected. This set up can be seen in Figure 22. Both the power LED and fast charge LED turned on. The circuit remained in fast charge for approximately two hours. During this time, the current was 630mA. When trickle charge mode was entered, the current was reduced to 170mA. This current remained constant for the remaining 144 minutes of the charge time. Fast charge was most likely terminated early to prevent damage to the batteries. With trickle charge, the battery voltage could slowly reach its maximum without overshooting the value.

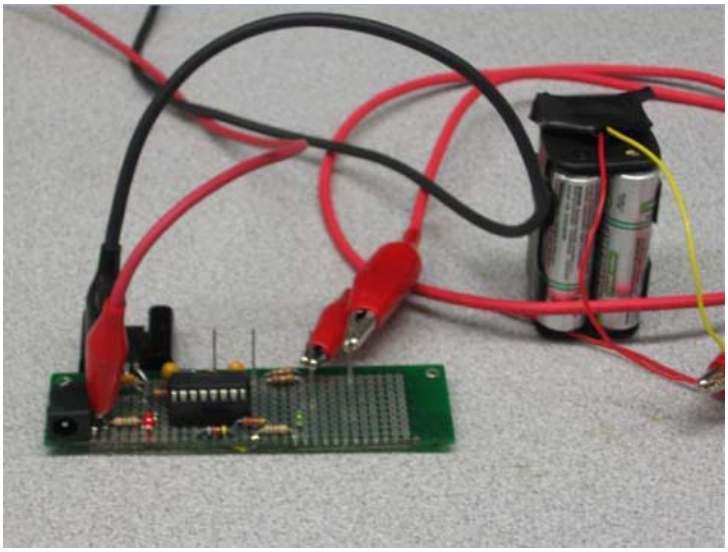


Figure 22: Testing the Charger Circuit

During the two hours of fast charge, approximately 1260mAh was injected into the batteries (2hrs*630mA). For the remaining time spent in trickle charge, around 408mAh was injected (2.4hrs*170mA). After the total 426 charge time, approximately 1668mAh was restored into the batteries. This circuit charged the batteries to 98.7% of the original milliamp-hours.

In addition, the temperature of the PNP case was monitored to ensure that the transistor could handle the power. During the entire 426 minutes, the maximum temperature was 105°C. At this temperature, the PNP can dissipate up to 15W. Since our circuit would never reach that point (7.56W max), the 2N6109 PNP was suitable for this design.

The results from the battery charger circuit were fairly reasonable. All the components met the required specifications in the MAX712 datasheet. Also, the batteries were almost fully charged to their original state. One reason for not fully recharging was that the charging efficiency can be as low as 80%, depending on the battery. The efficiency with which electrical energy was converted to chemical energy in the batteries was not the same as the power conversion efficiency of the MAX712. To fully charge the batteries, more time would be needed.

Next, the LDO was analyzed. The efficiency of the NCV551 was calculated with three and four NiMH batteries, with a nominal voltage of 3.6V and 4.8V, respectively. The average load current was assumed to be 50 mA, which was calculated as a typical value. Therefore, the following math was valid.

$$\eta\_LDO = \frac{3.3V(50mA)}{3.6V(50mA + 0.004mA)} = 91.7\%$$

$$\eta\_LDO = \frac{3.3V(50mA)}{4.8V(50mA + 0.004mA)} = 68.7\%$$

Clearly, the use of only three NiMH batteries, coupled with the low conversion ratio and ultra-low Iq, made the LDO more efficient than most SMPS systems. However, in our system, we are currently using four NiMH batteries. To use three, we'd have to redesign charging circuitry. Also, the runtime is

slightly longer with four batteries, even though the efficiency is not as high, due to the discharge curves of NiMH batteries. This is shown in the below figure.

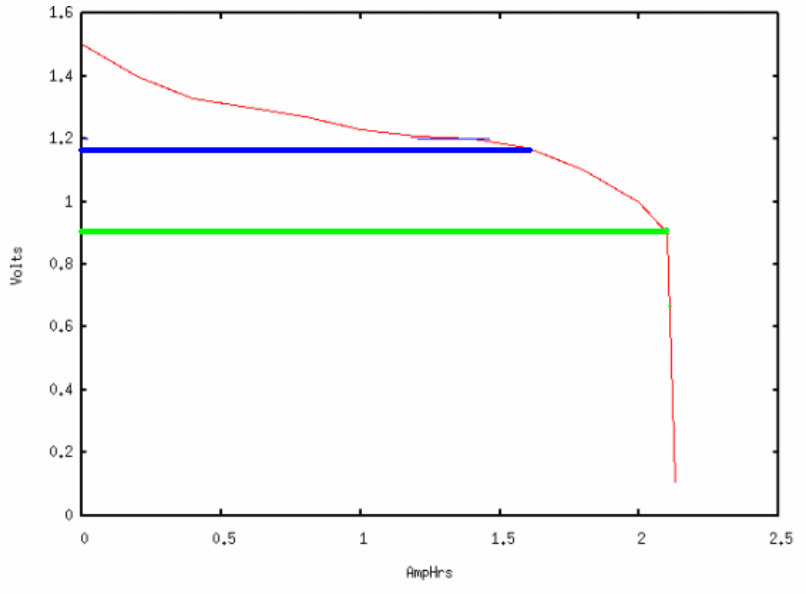


Figure 23: Battery Discharge Curve

In the above figure, the red line represents the discharge curve of one NiMH battery. The value graphed is the average of four batteries discharge curve. The batteries where measured at different intervals to generate the curve. The blue line represents the needed voltage if three batteries were used, of approximately 1.15V per battery to get 3.3V regulated with 150mV dropout. This was the worst case. The green line represented the needed voltage if four batteries were used, of approximately 0.86V per battery.

Results of this analysis show that using three batteries would provide an effective charge of only 1.65Ah. Using four batteries would provide an effective charge of 2.15Ah. This means that, even with the efficiency hit, the four batteries would last 1.3 times as long as just three batteries.

4.2 Accelerometer Module

The next module of the project was the accelerometer module. The complete module block diagram can be found in Figure 24. The accelerometer was used to measure the dynamic acceleration resulting from vibrations caused by the rough road conditions. The accelerometer output was ratiometric, which meant that output voltage varied proportionally with the changing acceleration.

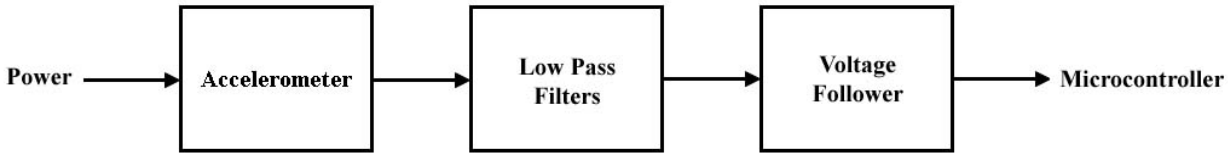


Figure 24: Accelerometer Module

4.2.1 Accelerometer Requirements

We decided to choose an accelerometer that had three axes. This would enable the user to place the device in any position inside the vehicle. The magnitude vector from the vibration could then be found by simply calculating the resultant vector from all three axes. The accelerometer should also have an output range of around ±2 – 3 g. In the background section, it was shown that the average g force experienced from a rough road was approximately 2 g's. The low number of g's also would increase the sensitivity of the output.

The accelerometer should minimize the interference from the vehicle vibration. A filter system was added to the accelerometer output to remove any vibration contributed by the vehicle. A simple RC low pass filter was used for each axis. The filters were designed to block high frequencies (greater than 20 Hz) from the vehicle. These high frequencies could potentially have a negative impact on the readings. The vibrations experienced by the rough roads would have a smaller frequency.

The accelerometer should also be small and low power. A small accelerometer would reduce the size of the overall device. The size should be less than 0.5 in. x 0.5 in. Low power would help to extend the life of the batteries. A power less than 5mW would be ideal.

From the above information, a list of specifications could be derived. This list is shown below.

- Accurately output magnitude no matter what the accelerometer orientation
- High sensitivity
- Minimize interference
- Small
- Low power

4.2.2 Accelerometer Selection

In order to meet our goals for the project, the correct accelerometer needed to be chosen. To make the best decision, a trade analysis was performed. Three accelerometers were examined in different areas of importance and assigned a rating. The higher the rating, the better suited the accelerometer was for our project. The examined areas are listed below. The scale used for each area ranged from 1 to 5 (low to high).

Power: Power was an important specification in choosing the correct accelerometer. Since the device was running on batteries, low power was required for our accelerometer. The metrics used to rate the accelerometers are as follows:

- Less than 1 mW => 5
- Between 1 and 5 mW => 4
- Between 5 and 9 mW => 3
- Between 10 and 14 mW => 2
- 15 mW or more => 1

Cost: Cost was another important specification that was examined. To keep the price of the device low, a less expensive accelerometer would be the best. The metrics used to rate the accelerometers are as follows:

- Between $0 and $4.99 => 5
- Between $5 and $9.99 => 4
- Between $10 and $14.99 => 3
- Between $15 and $19.99 => 2
- More than $20 => 1

Size: Another important consideration for the accelerometer was size. In order to minimize the overall size of the device, the accelerometer needed to be small. The metrics used to rate the accelerometers are as follows (decided from maximum dimension):

- Between 0 and 5.9 mm (length and width) => 5
- Between 6 and 10.9 mm (length and width) => 4
- Between 11 and 15.9 mm (length and width) => 3
- Between 16 and 20.9 mm (length and width) => 2
- More than 21 mm (length and width) => 1

Sensitivity: The sensitivity of the accelerometer was examined. The sensitivity was the scale factor for the accelerometer output. The larger the sensitivity, the larger the output range would be for our device. The metrics used to rate the accelerometers are as follows:

- More than 500 mV/g => 5
- Between 500 and 401 mV/g => 4
- Between 400 and 301 mV/g => 3
- Between 300 and 201 mV/g => 2
- 200 mV/g or less => 1

Axes: The number of axes for the accelerometer was an important consideration. More axes meant that the impact on the output from the orientation of the accelerometer would be decreased. The more axes it had, the more flexibility in placement of the device inside the vehicle. The metrics used to rate the accelerometers are as follows:

- Three => 5
- Two => 3
- One => 1

Supply Voltage: Supply Voltage was another consideration. To reduce the power consumed by the device, an accelerometer with a low supply voltage would be ideal. The metrics used to rate the accelerometers are as follows:

- Less than 3 V => 5
- Between 3 and 3.9 V => 4
- Between 4 and 4.9 V => 3
- Between 5 and 5.9 V => 2
- 6 V or more => 1

Maximum Rating: The maximum 'g' rating for the accelerometers were examined. The more g's the accelerometer could withstand, the longer it would last. The metrics used to rate the accelerometers are as follows:

- More than 10,000 g's => 5
- Between 10,000 and 5,000 g's => 4
- Between 4,999 and 1,000 g's => 3
- Between 999 and 100 g's => 2
- Less than 100 g's => 1

Temperature: The final consideration for the accelerometer was operating temperature. In order for the device to properly operate, the accelerometer needed to be able to operate over a wide range of temperatures. The metrics used to rate the accelerometers are as follows (decided from cold temperature):

- Between -50 and 90 °C => 5
- Between -40 and 80 °C => 4
- Between -30 and 70 °C => 3
- Between -20 and 60 °C => 2
- Between -10 and 50 °C => 1

After coming up with the accelerometer categories, all three accerelometers were examined. For each category, the relevant information was listed. The three accelerometers were the MMA1250 (made by Freescale Semiconductor), the ADXL320, and the ADXL330 (both made by Analog Devices). The information for each accelerometer is shown below:

MMA1250:
Power: 2.1 mA at 5.0 V => 10.5 mW
Cost: $11.51 from Digikey
Size: 10.67 mm x 10.45 mm (L x W)
Sensitivity: 400 mV/g
Axes: 1 (Z)
Supply Voltage: 5.0 V
Maximum Rating: 1500 g's (Powered); 2000 g's (Unpowered)
Temperature: -40 to 105 °C

ADXL320:
Power: 0.48 mA at 3.3 V => 1.58 mW
Cost: $8.60 from Digikey
Size: 4 mm x 4 mm (L x W)
Sensitivity: 174 mV/g
Axes: 2 (X, Y)
Supply Voltage: 3.3 V
Maximum Rating: 10,000 g's (Powered); 10,000 g's (Unpowered)
Temperature: -55 to 125 °C

ADXL330:
Power: 0.32 mA at 3.3 V => 1.06 mW
Cost: $11.58 from Digikey
Size: 4 mm x 4 mm (L x W)
Sensitivity: 300 mV/g
Axes: 3 (X, Y, Z)
Supply Voltage: 3.3 V
Maximum Rating: 10,000 g's (Powered); 10,000 g's (Unpowered)
Temperature: -55 to 125 °C

With all the specifications listed for each accelerometer, they could then be rated and compared. The importance of the category is shown next to the name in the parentheses. The scale used for each category ranged from 1 to 5 (low to high). The results are shown in Table 3. The raw scores were computed by adding the values in each row. The weighted score was computed by scaling each value and finding the sum. The accelerometer with the highest score was the ADXL330. This accelerometer was the best choice for our project.

Categories	**Freescale**	**ADXL320**	**ADXL330**
Power (5)	2	4	4
Cost (5)	3	4	3
Size (4)	4	5	5
Sensitivity (4)	4	1	3
Axes (4)	1	3	5
Supply Voltage (5)	2	4	4
Max. Rating (3)	3	4	4
Temperature (2)	4	5	5
Raw Score	23	30	33
Weighted Score	88	118	129

Table 3: Accelerometer Selection Analysis

4.2.3 Accelerometer Circuit Description

The circuit is shown below in Figure 25.

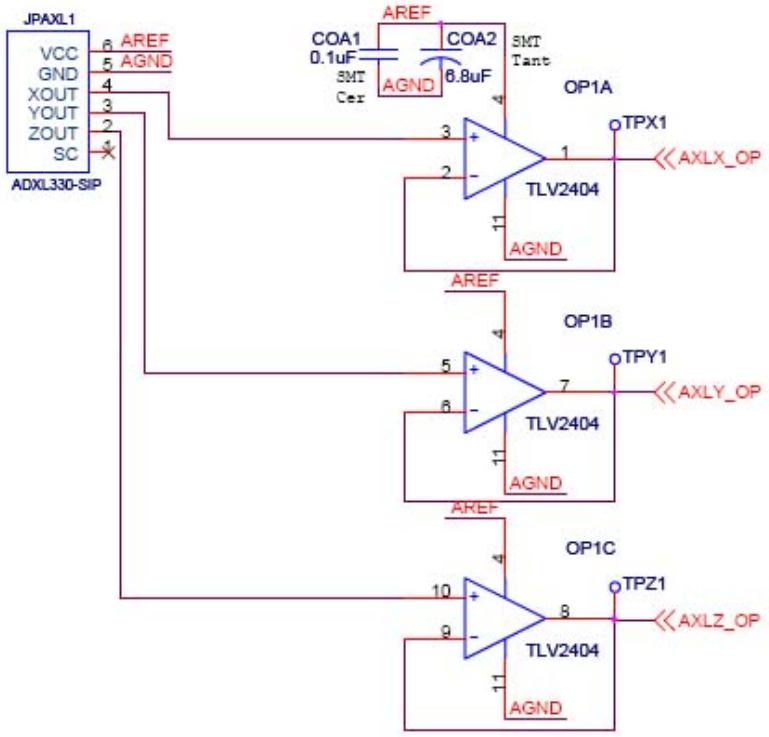


Figure 25: Accelerometer Circuit

The accelerometer output a voltage related to the g-level sensed. When an axis of the accelerometer was experiencing zero g's, the output voltage was equivalent to VDD/2, or 1.65V. If the axis was rotated, then the output changed according to the sensitivity of the accelerometer. For the ADXL330, the sensitivity was approximately 330mV/g. This meant that if the axis was experiencing one g, then its output was 1.65V (zero g) plus 0.330V (one g), or 1.98V. If the axis was experiencing negative one g, then the output was 1.65V (zero g) minus 0.330V (negative one g), or 1.32V. Since the accelerometer could range from ±3g, the output range per axis was 0.66 V to 2.64V.

A single pole RC low pass filter system was used to filter out unwanted vibrations from the vehicle. The cutoff for the filters was chosen to be 20Hz. A low frequency was selected because defects in the road would not vibrate the vehicle too rapidly. Since the accelerometer had 32kΩ resistors built inside the unit, only the capacitor values had to be selected. The values were chosen by the following equation:

· $C = 1 / [2\pi Rf]$
· $C = 1/ [2\pi * 32000 * 20]$
· $C = 0.249\ \mu F$

The final value for the capacitors was selected to be 0.22µF.

A voltage follower was included after the low pass filter to serve as a buffer for the ADC of the microcontroller. A TLV2402 op-amp was used in this design. A buffer was added because the

microcontroller had a low input impedance. The buffer will counteract any loading effects.

Since the accelerometer was so small, it was difficult to solder. To overcome this problem, a breakout board from Sparkfun was purchased. The board came with an accelerometer already soldered. The board can be seen in Figure 26.

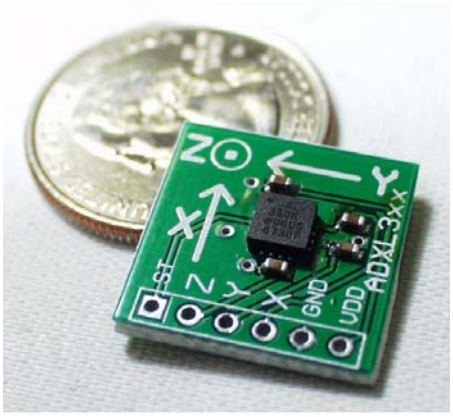


Figure 26: Accelerometer Breakout Board (Triple Axis, page 1)

4.2.4 Accelerometer Testing

First, to verify that the filter system functioned correctly, Bode plots were graphed. For this experiment, the accelerometer was removed and replaced with the function generator. The input signal from the generator was a $1.96V_{(pk\text{-}pk)}$ sine wave with a bias of 1.65V. The frequency of the signal was varied and the output voltages were measured. First, only the low pass filter was tested. The Bode plot is shown below in Figure 27. For low frequencies, the gain of the system was around one. Also, the 3-dB frequency was measured to be approximately 21.012Hz. This value was 5.06% higher than the expected frequency of 20Hz.

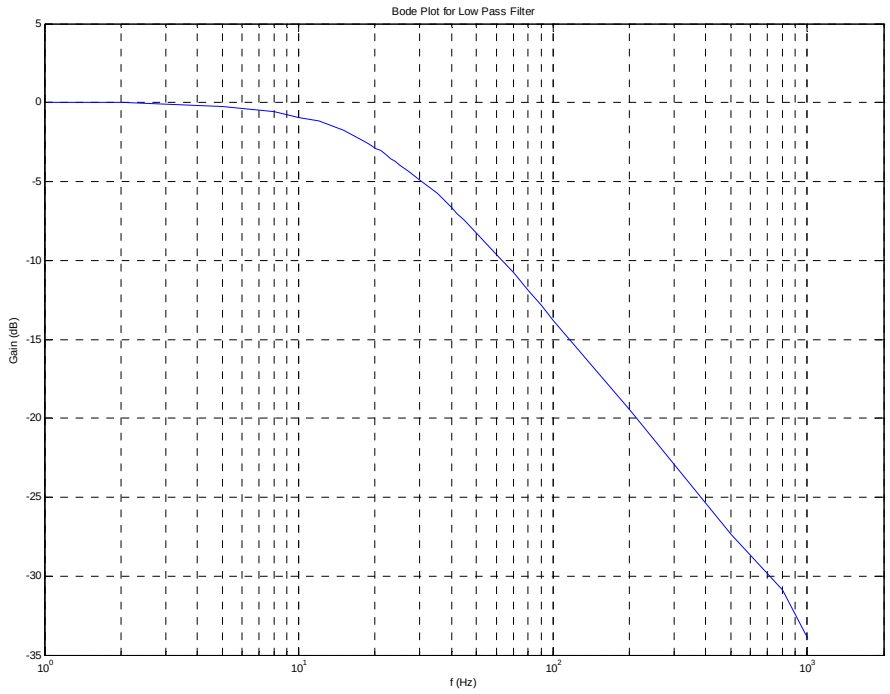


Figure 27: Bode Plot for Low Pass Filter

Next, the voltage follower circuit was tested. Again, the frequency of the input signal was varied from 1Hz to 1kHz, and the output was measured. The Bode plot is shown below in Figure 28. For low frequencies, the gain of the follower was one. However, as the frequencies exceeded 200Hz, the performance of the op-amp deteriorated. For a gain of one, the 3-dB frequency of the op-amp is 5.5kHz, so this was not the problem. The real issue was the slew rate. The slew rate for this op-amp is 2.5V/ms. At around 285Hz, the slope of the input signal reached 2.5V/ms. As the frequency increased, so did the input signal slope. Since this value exceeded the slew rate, the op-amp's performance decreased.

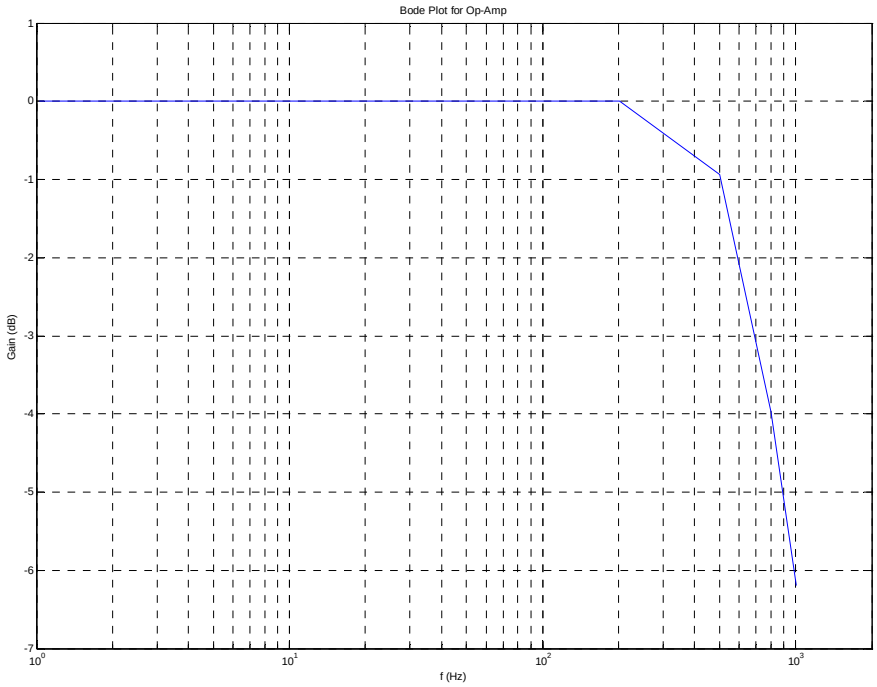


Figure 28: Bode Plot for Voltage Follower

Finally, the entire system was tested (low pass filter and voltage follower). The frequency of the same input signal was varied from 1Hz to 1kHz, and the output voltage was measured. The Bode plot is shown below in Figure 29. The measured data corresponded to the expected results. For low frequencies, the gain was one. The 3-dB frequency was measured to be 21.012Hz. For frequencies near 1kHz, the output was affected due to the decreased op-amp performance.

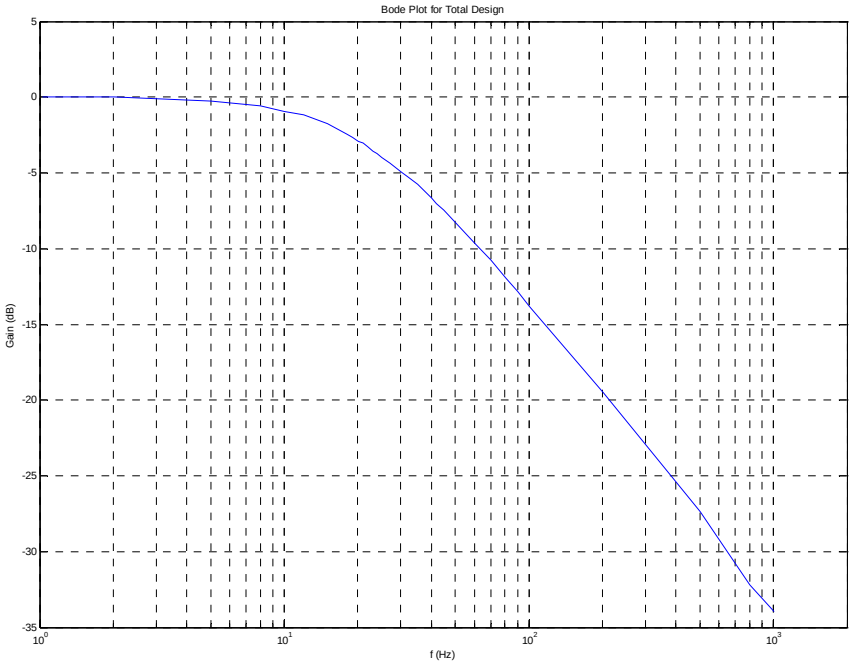


Figure 29: Bode Plot for Total Design

Next, the accelerometer was tested in various positions to verify that the expected output was the same as the actual output. By simply rotating the accelerometer, it was expected that the resultant vector would equal one g (0.330V). This resultant vector was calculated by taking the square root of the squares of the axes' output. All possible orientations were examined. Figure 30 shows the circuit that was tested.

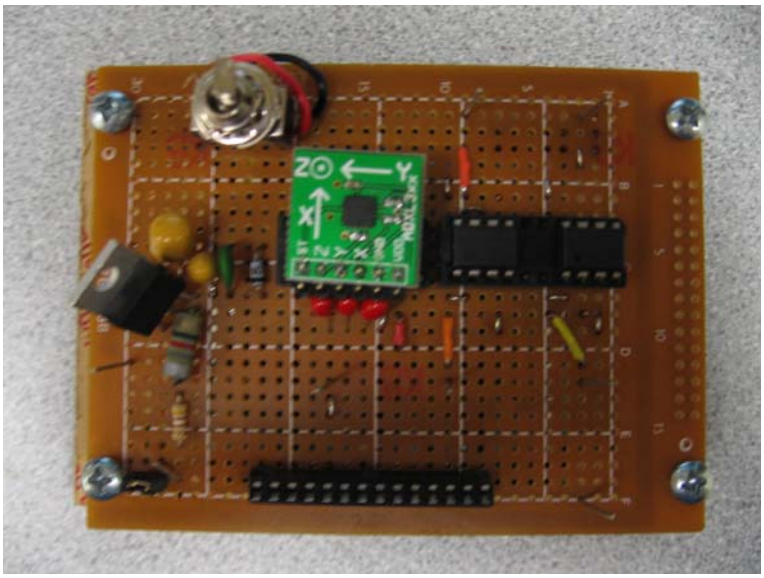


Figure 30: Soldered Accelerometer Circuit

First, the accelerometer was rotated around the X axis. By using this axis as the center, it was expected that its output would remain constant. The accelerometer was rotated 360° in increments of 45°. The output data can be seen in Table 4 and Table 5. Table 4 shows the measured data that was outputted from the accelerometer. These values included the DC offset voltage of VDD/2 (1.65V).

To measure the number of g's, the bias was removed from the output voltages. These adjusted values are shown in Table 5. From this data, the amount of force experienced can be determined (in volt form). The resultant vectors were calculated from these values. All of the magnitudes were around the expected 0.330V (one g). An error column in Table 5 shows how far the measured magnitudes were away from the expected 0.330V. Also, there was some additional error by assuming the DC bias level was VDD/2. In reality, the bias level for all the accelerometers will not be this value. Since our roughness data does not have to be too accurate (less than 10 % error), this assumption is acceptable.

Actual Accelerometer Output Voltages				
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude (V)**
0	1.66	1.66	2.0	3.084023
45	1.64	1.9	1.9	3.147952
90	1.62	1.96	1.64	3.025822
135	1.62	1.8	1.4	2.797213
180	1.64	1.6	1.34	2.65428
225	1.68	1.38	1.46	2.618855
270	1.68	1.3	1.68	2.708284
315	1.68	1.4	1.9	2.896964

Table 4: Actual Output Voltages (Rotate Around X Axis)

Adjusted Accelerometer Output Voltages					
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude(V)**	**Error (%)**
0	0.01	0.01	0.35	0.350286	6.147151
45	-0.01	0.25	0.25	0.353695	7.180237
90	-0.03	0.31	-0.01	0.311609	5.573112
135	-0.03	0.15	-0.25	0.293087	11.18575
180	-0.01	-0.05	-0.31	0.314166	4.798315
225	0.03	-0.27	-0.19	0.331512	0.458088
270	0.03	-0.35	0.03	0.352562	6.836985
315	0.03	-0.25	0.25	0.354824	7.522394

Table 5: Adjusted Output Voltages (Rotate Around X Axis)

Next, the accelerometer was rotated around the Y axis. Again, the accelerometer was rotated 360° in increments of 45°. The output data can be seen in Table 6 and Table 7. Table 6 shows the actual voltage outputted by the accelerometer. Table 7 shows these values without the DC bias. As previously done, the magnitudes were calculated. The results were fairly accurate. Most were within 2% of the expected 0.330V.

Actual Accelerometer Output Voltages				
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude (V)**
0	1.66	1.66	2.0	3.084023
45	1.4	1.62	1.86	2.836195
90	1.32	1.62	1.66	2.668782
135	1.42	1.62	1.42	2.580155
180	1.64	1.64	1.34	2.678582
225	1.9	1.66	1.42	2.895168
270	1.98	1.66	1.68	3.081947
315	1.8	1.66	1.94	3.123972

Table 6: Actual Output Voltages (Rotate Around Y Axis)

Adjusted Accelerometer Output Voltages					
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude (V)**	**Error (%)**
0	0.01	0.01	0.35	0.350286	6.147151
45	-0.25	-0.03	0.21	0.327872	0.644871
90	-0.33	-0.03	0.01	0.331512	0.458088
135	-0.23	-0.03	-0.23	0.32665	1.015255
180	-0.01	-0.01	-0.31	0.310322	5.962905
225	0.25	0.01	-0.23	0.339853	2.98573
270	0.33	0.01	0.03	0.331512	0.458088
315	0.15	0.01	0.29	0.32665	1.015255

Table 7: Adjusted Output Voltages (Rotate Around Y Axis)

Finally, the accelerometer was rotated around the Z axis. The accelerometer was rotated 360° in increments of 45°. The output data can be seen in Table 8 and Table 9. Table 8 shows the actual voltage outputted by the accelerometer, and Table 9 shows these values without the DC bias. The magnitudes of the resultant vectors were calculated. The results were similar to the expected 0.330V. About half of the magnitudes were below 3% in error.

Actual Accelerometer Output Voltages				
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude (V)**
0	1.98	1.66	1.66	3.071091
45	1.88	1.86	1.66	3.122435
90	1.64	1.96	1.68	3.058366
135	1.36	1.8	1.68	2.812828
180	1.3	1.62	1.66	2.658947
225	1.42	1.42	1.66	2.605456
270	1.68	1.3	1.66	2.695923
315	1.9	1.42	1.68	2.906682

Table 8: Actual Output Voltages (Rotate Around Z Axis)

Adjusted Accelerometer Output Voltages					
Position (°)	**Xout (V)**	**Yout (V)**	**Zout (V)**	**Magnitude (V)**	**Error (%)**
0	0.33	0.01	0.01	0.330303	0.091785
45	0.23	0.21	0.01	0.311609	5.573112
90	-0.01	0.31	0.03	0.311609	5.573112
135	-0.29	0.15	0.03	0.327872	0.644871
180	-0.35	-0.03	0.01	0.351426	6.492627
225	-0.23	-0.23	0.01	0.325423	1.38703
270	0.03	-0.35	0.01	0.351426	6.492627
315	0.25	-0.23	0.03	0.341028	3.341775

Table 9: Adjusted Output Voltages (Rotate Around Z Axis)

The results from the initial accelerometer tests were reasonable. The Bode plots were close to the expected results. For low frequencies, the gain was equal to one. Also, the 3-dB frequency was close to the expected 20Hz. The results from the orientation tests were similar to the expected output. In all the different orientations, the accelerometer experienced a force of approximately one g (0.330V). The average error for all the positions was 4.08%. These results increase our confidence that the data from the car test will be accurate. The next step was to place the accelerometer in the car and measure the vibrations from the road.

After the initial testing, the accelerometer was interfaced with the microcontroller through the ADC, channels 0-3. Results in the following graphs, Figure 31, Figure 32, Figure 33, Figure 34, Figure 35, and Figure 36, show readings, on a scale from 0 to 4095, of the different axes. The data shows that each axis can be independently modified.

ADC12MEM1	2067
ADC12MEM1*3.3/4...	1.66571426
ADC12MEM2	2103
ADC12MEM2*3.3/4...	1.69472527
⊟ axl	<struct>
⊟ x	<struct>
value	2500
max	2500
min	2082
⊟ y	<struct>
value	2067
max	2074
min	1646
⊟ z	<struct>
value	2103
max	2701
min	1644

Figure 31: Accelerometer, X,- +1g

ADC12MEM1	2075
ADC12MEM1*3.3/4...	1.67216122
ADC12MEM2	3095
ADC12MEM2*3.3/4...	2.49413919
⊟ axl	<struct>
⊟ x	<struct>
value	1667
max	2500
min	1667
⊟ y	<struct>
value	2075
max	2075
min	1646
⊟ z	<struct>
value	3095
max	3095
min	1644

Figure 32: Accelerometer, X, -1g

ADC12MEM1	2509
ADC12MEM1*3.3/4...	2.0219047
ADC12MEM2	3107
ADC12MEM2*3.3/4...	2.50380945
⊟ axl	<struct>
⊟ x	<struct>
value	2078
max	2500
min	1667
⊟ y	<struct>
value	2509
max	2509
min	1646
⊟ z	<struct>
value	2040
max	3107
min	1642

Figure 33: Accelerometer, Y, +1g

ADC12MEM1	1655
ADC12MEM1*3.3/4...	1.33369958
ADC12MEM2	2040
ADC12MEM2*3.3/4...	1.64395606
⊟ axl	<struct>
⊟ x	<struct>
value	2067
max	2500
min	1667
⊟ y	<struct>
value	1655
max	3142
min	1646
⊟ z	<struct>
value	2040
max	3107
min	1642

Figure 34: Accelerometer, Y, -1g

ADC12MEM1	2073
ADC12MEM1*3.3/4...	1.67054939
ADC12MEM2	1644
ADC12MEM2*3.3/4...	1.32483506
⊟ axl	<struct>
⊟ x	<struct>
value	2082
max	2083
min	2082
⊟ y	<struct>
value	2073
max	2074
min	2073
⊟ z	<struct>
value	3107
max	3107
min	1644

Figure 35: Accelerometer, Z, shaken

ADC12MEM1	2071
ADC12MEM1*3.3/4...	1.66893768
ADC12MEM2	1644
ADC12MEM2*3.3/4...	1.32483506
⊟ axl	<struct>
⊟ x	<struct>
value	2077
max	2500
min	1667
⊟ y	<struct>
value	2071
max	3142
min	1646
⊟ z	<struct>
value	1644
max	3107
min	1642

Figure 36: Accelerometer, Z, -1g

This data was collected on the PCB in the in-circuit debugger of IAR Embedded Workbench. The data shows that the accelerometer can be read through the ADC channels of the microcontroller. In each case, the PCB was rotated to intentionally affect one axis. It was then rotated 180° on the same axis. The results are "+1g" and "-1g" readings.

4.3 GPS Module

The purpose of this module was to accurately locate the rough spots measured by the accelerometer. This was accomplished by recording the longitude and latitude positions every few seconds. The information was sent to the GPS from satellites via an antenna. The type of information received was date, time, and orbital information of the satellites. This information was used to determine the location. The output from the GPS module was a National Marine Electronics Association (NMEA) string. The relevant information contained in these strings was date, time, longitude, and latitude. This data was then sent to the microprocessor to be stored.

4.3.1 GPS Module Requirements

The main requirement for the GPS was low power. The GPS module would require more power than any other module. Since the overall power draw of the device was desired to be less than 100mW, the selected GPS module could not use more than this amount. A power of 50mW or less would be ideal. The lower power would greatly help extend the life of the batteries.

Another important area was accuracy. As previously stated, the accuracy did not have to be too exact in determining the location. Since our device would locate a general area of road roughness, an accuracy of 5m was not too far off from an accuracy of 10m. Most GPS modules have an accuracy of 15m or less (without error correction), so a distance of 15m or less would be acceptable.

A third requirement would be inexpensive. Similar to power, the GPS module cost more than any other module. To keep the cost of the overall device down, a GPS module with a cost between $50 and $100 would be desired. Size would also be another concern. A smaller GPS module would reduce the overall size of the device.

From the above information, a list of specifications could be derived. This list is shown below.

- Low Power
- Moderately accurate
- Inexpensive
- Small

4.3.2 GPS Module Selection

As with the accelerometer, a trade analysis was performed to determine the right GPS module. Four modules were examined in different areas of importance and assigned a rating. The higher the rating, the better suited the GPS module was for our project. The examined areas are shown below. The scale used for each category ranged from 1 to 5 (low to high).

Power: Power was the most important specification in choosing the correct GPS module. Since the GPS used the most power, finding a low power module would be preferred. The metrics used to rate the modules are as follows:

- Between 0 and 49 mW => 5
- Between 50 and 99 mW => 4
- Between 100 and 149 mW => 3
- Between 150 and 199 mW => 2
- 200 mW or more => 1

Cost: Cost was another important specification that was examined. An inexpensive GPS module would help reduce the overall cost of the device. The metrics used to rate the modules are as follows:

- Less then $50 => 5
- Between $50 and $64 => 4
- Between $65 and $79 => 3
- Between $80 and $94 => 2
- $94 or More => 1

Size: Another important consideration for the GPS was size. In order to minimize the overall size of the device, the module needed to be small. The metrics used to rate the GPS modules are as follows (decided from maximum dimension):

- Between 0 and 29 mm (length and width) => 5
- Between 30 and 39 mm (length and width) => 4
- Between 40 and 49 mm (length and width) => 3
- Between 50 and 59 mm (length and width) => 2
- 60 mm or More (length and width) => 1

Accuracy: The accuracy of the accelerometer was examined. In order to properly locate the rough spots in the road, the GPS coordinates need to be as accurate as possible. The metrics used to rate the modules are as follows:

- Less than 0.5 meter error => 5
- Between 0.5 and 1 meter error => 4
- Between 1.1 and 5 meter error => 3
- Between 5.1 and 10 meter error => 2
- More than 10 meter error => 1

Performance: The performance of the GPS module was an important consideration. The performance was measured by the GPS cold start time. This was the time needed by the GPS to reacquire the needed information to determine its location. This information included the current time and the orbits of the satellites. The metrics used to rate the GPS modules are as follows:

- Less then 25 seconds => 5
- Between 25 and 49 seconds => 4
- Between 50 and 74 seconds => 4
- Between 75 and 99 seconds => 4
- 100 seconds or More => 1

Supply Voltage: Supply Voltage was another consideration. To reduce the power consumed by the device, a GPS module with a low supply voltage would be the best choice. The metrics used to rate the modules are as follows:

- Less than 3 V => 5
- Between 3 and 3.9 V => 4
- Between 4 and 4.9 V => 3
- Between 5 and 5.9 V => 2
- 6 V or more => 1

Temperature: The final consideration for the GPS modules was operating temperature. In order for the device to properly operate, the GPS needed to be able to operate over a wide range of temperatures. The metrics used to rate the modules are as follows (decided from cold temperature):

- Between -50 and 90 °C => 5
- Between -40 and 80 °C => 4
- Between -30 and 70 °C => 3
- Between -20 and 60 °C => 2
- Between -10 and 50 °C => 1

After selecting the categories, all four GPS modules were examined. For each category, the relevant information was listed. The four modules were the 25LP LVS (made by Garmin), the Lassen iQ (made by Trimble), the PG-31 (made by Laipac), and the RGPSM002 (made by Semtech). The information for each GPS module is shown below:

25LP LVS:
Power: 115 mA at 5.0 V => 575 mW
Cost: $99.99
Size: 46.5 mm x 69.9 mm x 11.4 mm (L x W x H)
Accuracy: 15 meters (non WAAS)
Performance: 45 seconds
Supply Voltage: 5.0 V
Temperature: -30 to 85 °C

Lassen iQ:
Power: 27 mA at 3.3 V => 89.1 mW
Cost: $49.95 from Sparkfun
Size: 26 mm x 26 mm x 6 mm (L x W x H)
Accuracy: 8 meters (90% of time) (non WAAS)
Performance: 84 seconds (90% of time)
Supply Voltage: 3.3 V
Temperature: -40 to 85 °C

PG-31:
Power: 70 mA at 3.3 V => 231 mW
Cost: $55.00
Size: 30.6 mm x 26 mm x 9.8 mm (L x W x H)
Accuracy: 25 meters (non WAAS)
Performance: 45 seconds
Supply Voltage: 3.3 V
Temperature: -40 to 85 °C

RGPSM002:
Power: 19 mA at 3.3 V => 62.7 mW
Cost: $68.75 from Digikey
Size: 26.59 mm x 31.59 mm x 11.2 mm (L x W x H)
Accuracy: 5 meters (50% of time) (non WAAS)
Performance: 120 seconds (50% of time)
Supply Voltage: 3.3 V
Temperature: -40 to 85 °C

With all the specifications listed for each GPS module, they could then be rated and compared. Again, the importance of the category is shown next to the name in the parentheses. The scale used for each category ranged from 1 to 5 (low to high). The results are shown in Table 10. The raw scores were computed by adding the values in each row. The weighted score was computed by scaling each value and finding the sum. The module with the highest score was the Lassen iQ. This GPS module was the best choice for our project. The Lassen iQ module can be seen in Figure 37.

Categories	**25LP LVS**	**Lassen iQ**	**PG-31**	**RGPSM002**
Power (5)	1	4	1	4
Cost (5)	1	5	4	3
Size (4)	1	5	4	4
Accuracy (3)	1	2	1	3
Performance (3)	4	2	4	1
Supply Voltage (5)	2	4	4	4
Temperature (2)	3	4	4	4
Raw Score	13	26	22	23
Weighted Score	45	105	84	91

Table 10: GPS Module Selection Analysis

Figure 37: Lassen iQ GPS Module (Lassen iQ GPS, page 23)

4.3.3 GPS Module Circuit Description

The GPS circuit is shown below in Figure 38.

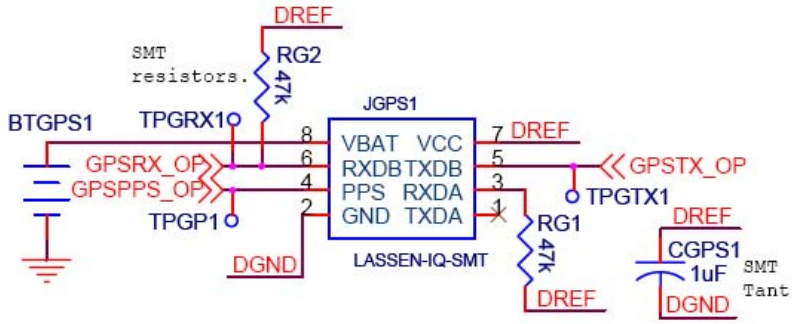


Figure 38: GPS Module Circuit

The Lassen iQ GPS module ran off a 3.3V supply provided by the voltage regulator. A lithium cell battery (BTGPS1) was included as a back up power supply. The battery kept the module's RAM memory alive and powered the real time clock when the receiver's main power source was turned off.

Two serial ports were available with this GPS module. Serial port B (Port 2) was used to transmit data to the microcontroller (TXDB) and receive data from the microcontroller (RXDB). Both pins were able to be connected directly to the microcontroller UART. Serial port A (Port 1) was unused. In order to prevent damage to the unused serial port, the receive pin (RXDA) was tied to the 3.3V supply via a pull up resistor. The transmit pin (TXDA) was able to be left floating.

4.3.4 GPS Module Testing

A special breakout board designed for the Lassen iQ was used to verify that the GPS was functioning. The board was purchased from Sparkfun and can be seen in
Figure 39. The RS-232 output was used to verify that the GPS module did output the expected NMEA strings (Port 2).

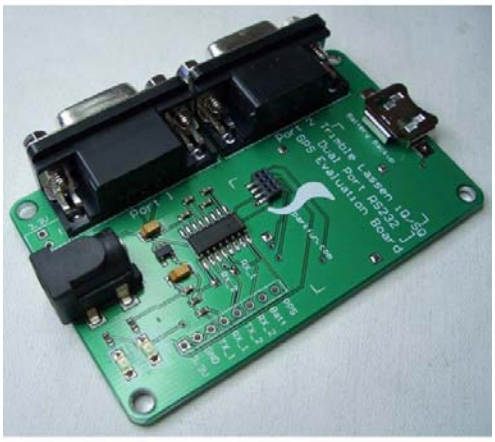


Figure 39: Lassen iQ Evaluation Board (Lassen iQ Evaluation, page 1)

4.4 Microcontroller Module

The purpose of the microcontroller module was to interface all of the embedded system hardware and run the embedded system software. There were various ways that this module could be implemented. A microcontroller, microprocessor, and FPGA were all considered.

A microcontroller combines a processing unit with many interfaces and built-in RAM and ROM memory all on one chip. They are available with 8, 16, and 32-bit support with kilobytes of RAM and ROM running at tens or a couple of hundred of MHz for the more robust and expensive chips. The software can be written in a variety of languages, including BASIC and assembly, with C being the most common. At the high-end, ARM microcontrollers, available from many different manufacturers, currently dominate the market. Some low-power consuming examples include the TI MSP430, Atmel AVR, and Microchip PICs. Since one of our primary objectives was to minimize power consumption, a low-power microcontroller was ideal.

A microprocessor would be used in a single board computer (SBC), where there is separate RAM, ROM, and interface IC's connected to the microprocessor. These also are available with 8, 16, 32, and 64-bit support with kilobytes to megabytes, or even gigabytes, of RAM and ROM running from a few MHz to over a GHz. There are numerous languages in which to write the software, but C and C++ are most popular, with Java making some headway and BASIC still a popular choice. Some examples include the old 8051 to the more modern chips available from companies such as Freescale, IBM, and Intel. For our system, new microprocessor SBC systems would be overkill, and the old chips do not provide as much functionality as a microcontroller. A microcontroller, with features such as ROM, RAM, and serial interfaces on-chip, is much smaller than a microprocessor or SBC system. Additionally, both the old, less feature-full, and newer microprocessors use much more power than modern low-power microcontrollers. Therefore, microprocessor systems were ruled out at the beginning.

Another possible implementation would be to use a FPGA. A FPGA is essentially programmable hardware. In this system, hardware is written in a hardware description language such as VHDL or Verilog to interface all the units together. The interfacing would be fairly easy, but some aspects are more straightforward in a software programming language, such as instructing the chip to run a sequence of commands in order. However, with the prevalence of soft cores, where a microprocessor can be run on

the FPGA chip itself, one could get the best of both worlds.

The main reason we went with a microcontroller was that drivers to interface with certain hardware, especially the memory, were available as C libraries. Porting the code over to run in hardware initially seemed more difficult to do than modifying the library to work for our particular architecture. In the end, porting the code to the FPGA and using a soft core seemed like it would take more time and be less straightforward than implementing the system using a microcontroller. We decided to use a low-power microcontroller for our design.

4.4.1 Microcontroller Requirements

We came up with a list of requirements for the microcontroller module based on the overall design requirements. They are listed below.

- Ultra-low power consumption
- Economical performance
- Interfaces for all devices
- In-circuit debugging support
- Extensive documentation
- High-level language support
- Low cost

The requirements for in-circuit debugging, extensive documentation, and high-level language support were practical in nature. Without these features, it would be very difficult to work with the microcontroller module. Based on these requirements, some specifications were derived. The following paragraphs outline the process arriving at the specifications from the requirements.

Minimization of power consumption was the primary design goal. Most modern microcontrollers are designed to minimize power consumption, and this includes running off of a lower voltage than the typical 5V supply used in the past. The power saving microcontrollers can operate on a couple of milliamps or less. Additionally, power-saving modes are desired to minimize power consumption during down time. A specification of drawing under 1mA during power saving modes was compatible with modern low-power microcontroller specifications.

Running at a lower voltage level meant the other devices would need to be compatible with the logic levels of the microcontroller; otherwise, unnecessary complexity and cost would have to be added by introducing logic level converters. The 3.3V logic level was the next step down from 5V in common use, which included a wide array of devices compatible with this logic. Stepping down to 3.0V, 2.7 V, or lower would make it more difficult to find compatible devices, especially a GPS module.

For our system, we did not require a high level of processing performance. It simply needed to take data, perform minimal processing on-board, and record it. More intense post-processing used the power of the PC, allowing for a microcontroller which did not require much performance. For this reason, low performance microcontrollers (by modern standards), running at under 20 MIPS, typically at under 20 MHz, could be used. This additionally meant that a full 32-bit microcontroller would be overkill and 16 or even 8-bit microcontrollers were suitable.

To interface with all the devices, we needed serial ports. These could be emulated in “bit banging” drivers in software, but hardware drivers greatly increase performance and decrease the power

consumption by decreasing the amount of work required from the microcontroller's processing unit itself. We needed at least two hardware serial ports, since we would only be using two serial devices at a time. Additionally, based on the interfaces of the other devices, these ports needed to be compatible with both the asynchronous UART and synchronous SPI.

The accelerometer module chosen provided an analog output. In order to interface with the microcontroller, an analog-to-digital converter (ADC) was required. The accelerometer used was triple-axis, so at least three ADC channels were required.

In order to do in-circuit programming and debugging, an interface to achieve this was required. The JTAG interface is nearly ubiquitous for this use. We needed support for JTAG or some equivalent interface.

We decided to program in C due to the availability of a C library for the memory and extensive experience in C. Therefore, we needed a microcontroller with a C compiler. In order to perform the in-circuit debugging, a full-featured IDE was also required, so we could step through the C code and view the values of memory locations and variables. On such IDE is IAR Embedded Workbench. The derived specifications are listed below.

- Power-saving modes to draw < 1 mA
- Runs off of 3.3V logic
- 5-20 MIPS performance
- 8- or 16-bit architecture
- At least two hardware serial ports including SPI and UART
- 3 or more channel on-board ADC
- JTAG or equivalent interface for in-circuit programming and debugging
- C compiler and IDE availability
- Less than $10

4.4.2 Microcontroller Selection

Based on the specifications, a number of different microcontrollers were identified. These included the general groups of Microchip PIC18F, Atmel AVR ATmega, and TI MSP430.

The PIC18F was ruled out early due to weak support of in-circuit programming and debugging, C compilers, and full-featured IDEs. Also, personal experience with tools for performing these tasks with PIC18F chips led us away from them.

The main choice was between the ATmega and MSP430 series. They both operate from 1.8 to 3.6V, have standby and sleep modes that draw hundreds of microamps, and implement brown out protection and reset. Additionally, both are RISC architectures with C compilers available. Full-featured IDEs with focus on in-circuit debugging are available and well-developed for each series. Also, they have on-board oscillators. Lastly, both have extensive documentation.

There are some important architectural differences between the ATmega and MSP430. These are listed in Table 11 below.

	ATmega	MSP430
Bus width	8-bit	16-bit
ADC	10-bit	10-,12- and 16-bit
USART	Up to 3	Up to 2

Table 11: Microcontroller Module Selection

In the end, the major deciding factor was that the ECE department was switching over to the MSP430 microcontrollers from the PICs. The addition of more support and a development environment already installed pushed the MSP430 over the ATmega. The MSP430F169 was chosen due to its large programming size (60KB), two UART/SPI ports, internal 8 MHz clock, built-in brownout safety features, 8 channels of 12-bit ADCs, internal temperature sensor, small size, incredibly low power consumption, and low cost.

4.4.3 Microcontroller Circuit Description

The schematic for the microcontroller voltage sources is shown in Figure 40.

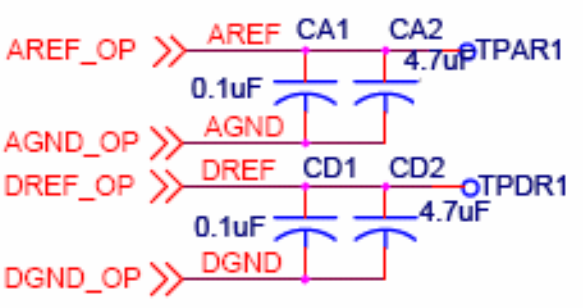


Figure 40: Microcontroller Source Schematic

The microcontroller had two voltage sources: analog and digital. The analog reference, AREF, was connected to the AVCC input. The analog ground, AGND, was connected to the AVSS input. The digital reference, DREF, was connected to the DVCC input. The digital ground, DGND, was connected to the DVSS input. The analog and digital grounds were isolated planes on the PCB.

The capacitors, CA1, CA2, CD1, and CD2 acted as decoupling capacitors to reduce noise and transient disruption of the references. CA1 and CD1 were 0.1µF ceramic surface mount capacitors used to block high frequency noise. Ceramic capacitors were chosen due to their low equivalent series resistance (ESR), a key element in increasing reference stability. Additionally, they were offered in very small and inexpensive surface mount packages which were relatively invariable over time and temperature. One downfall of ceramic capacitors was their mechanical fragility, so the PCB could not be heavily mechanically stressed or the capacitors could easily crack. In our application, this was a non-issue. The capacitors, CA2 and CD2 were 4.7µF tantalum surface mount capacitors used to reduce transient disruption of the references. Tantalum capacitors were chosen due to there small, surface mount packages and relative invariability over time and temperature.

In Figure 41, the schematic for the main microcontroller circuitry is shown.

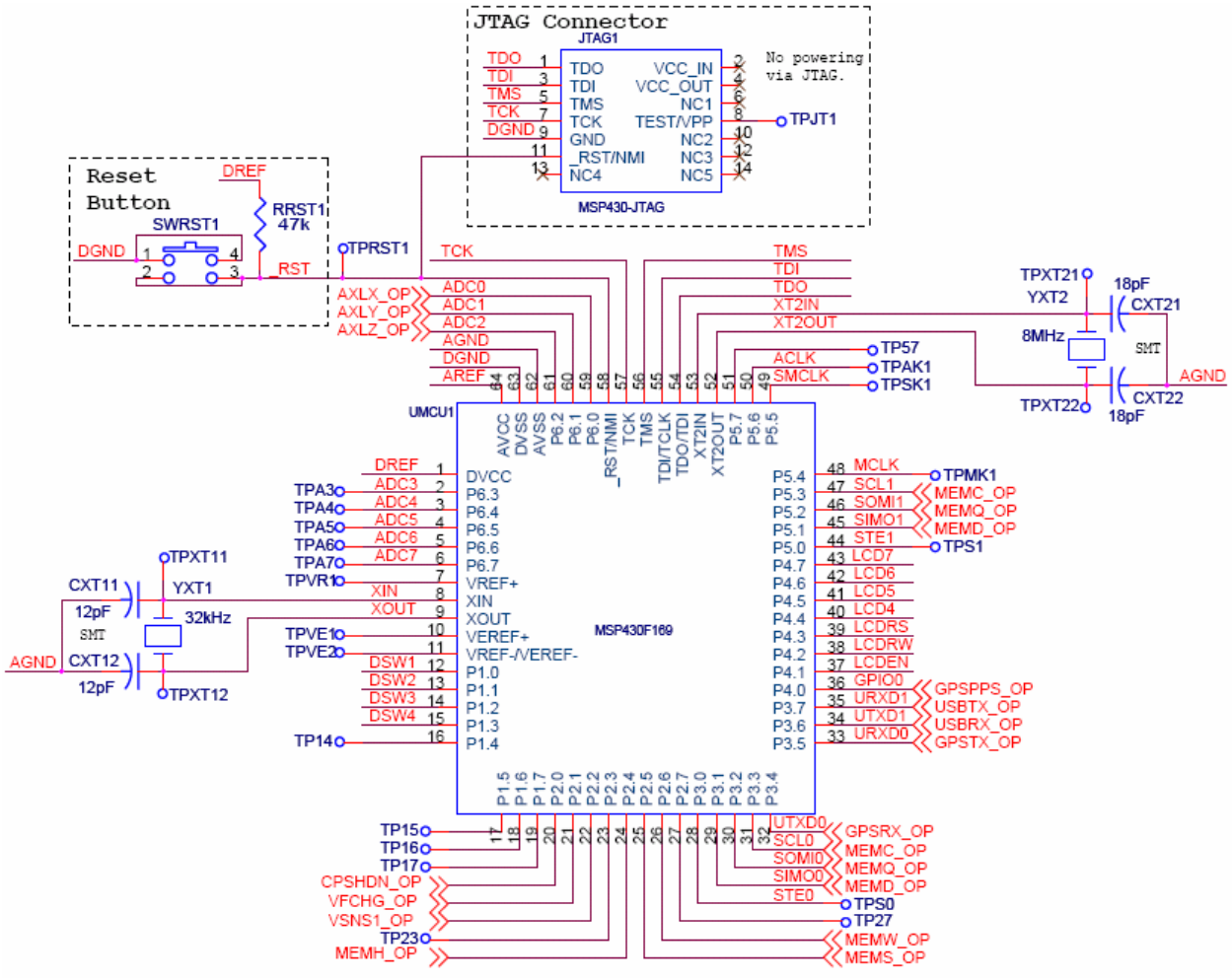


Figure 41: Microcontroller Schematic

Shown in the figure above are the interconnections of other modules with the microcontroller. The labels (in red) on each pin of the microcontroller correspond to labels found in other module schematics.

The JTAG connector was used to program and debug the microcontroller. In-circuit debugging was possible using IAR Embedded Workbench with the JTAG connector. The TDO, TDI, TMS, and TCK pins of the JTAG connector connected to the corresponding pins on the microcontroller. These were used for JTAG communication.

The reset button was a simple surface-mount push button, the FSMJSMA by Alcoswitch, as shown in Figure 42. This was used to minimize the size of the button, which would not be used frequently. The logic was active-low, connecting to the _RST line of the microcontroller. A 47kΩ surface-mount resistor pull up to the 3.3V digital reference was used to keep the microcontroller from not resetting by default. When the button was pushed, the _RST line was shorted to ground, causing the microcontroller to reset.

Figure 42: Alcoswitch FSMJSMA Push Button (Tact Switches, page 1)

The crystals, YXT1 and YXT2, together with their load capacitors, CXT11 and CXT12, and CXT21 and CXT22, provided external clocks to use in the microcontroller. YXT1 was a small, 6-mm "can"-type through-hole crystal, like the one seen in Figure 43. This was chosen due to the small size and mechanical strength of through-hole connections, as well as low cost. YXT1 oscillated at 32kHz with 12pF ceramic surface-mount load capacitors attached. This crystal was used to drive the internal hardware timer, Timer A, of the microcontroller.

Figure 43: 6mm Crystal (Crystal, page 1)

The crystal, YXT2 was a low-profile though-hole HC49/S crystal, like the one seen in Figure 44. This was chosen due to the low-profile, small board space requirement, mechanical strength of through-hole components, low cost, and ease of soldering (versus leadless surface mount crystals). YXT2 oscillated at 8MHz with 18pF ceramic surface-mount load capacitors attached. This crystal was used to drive the fast clock needed for 1Mbaud communication by the USB and the fast SPI communication used by the serial flash memory (SFM).

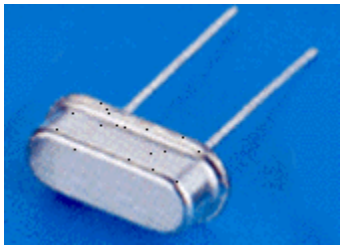


Figure 44: HC49/S Crystal (Quarts Crystals, page 1)

The figure below shows the DIP switch connection to the microcontroller.

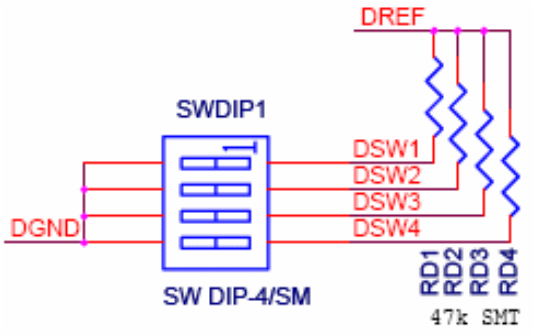


Figure 45: Microcontroller DIP Switch Schematic

The DIP switch was used to select the operating mode of the microcontroller. Originally, a small, low-profile surface mount 4-pole DIP switch was used, but this proved to be too small to use effectively. This was the A6H-4101 model by Omron Electronics, shown in Figure 46. A standard, low-profile through-hole 4-pole DIP switch is now used, such as the GDS04 available by Tyco. An 8-pole version of this switch is shown in Figure 47.

Figure 46: A6H-4101 DIP Switch
(Half-Pitch Dip, page 1)

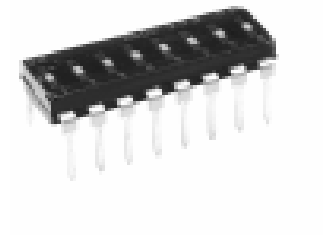


Figure 47: GDS04 DIP Switch
(DIP Switches, page 1)

4.4.4 Microcontroller Testing

The microcontroller was tested by seeing if IAR Embedded Workbench could identify, program and perform in-circuit debugging on it. This works. An example of in-circuit debugging is shown in Figure 48.

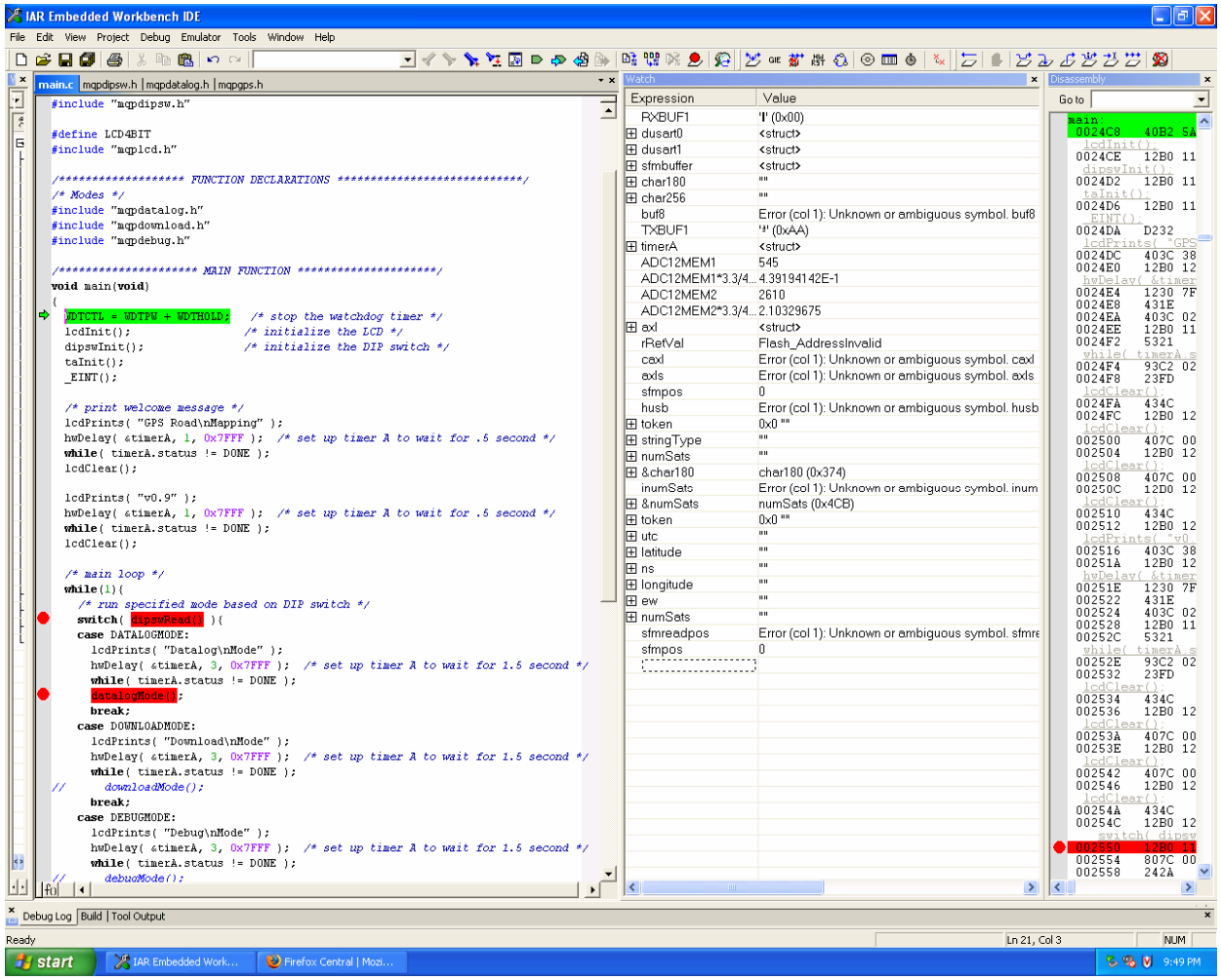


Figure 48: In-Circuit Debugging in IAR Embedded Workbench

4.5 Memory Module

The purpose of the memory module was to provide storage for data recordings. First, a number of design decisions needed to be made. These include the volatility, interface, and type of the memory.

The memory could be either volatile or non-volatile. Volatile memory would require constant powering of the memory because when power is lost, volatile memory is erased. A separate battery can be used to keep the volatile memory powered even when the rest of the device is not powered; however, this increases the size, cost, and complexity. Non-volatile memory does not require power in order to maintain the values stored in memory. Since our device is designed to run for up to a week, potentially losing that much data is undesirable. Therefore, we decided to use non-volatile memory.

There are a number of interfaces for memory. These can be generally broken down into two categories, based on their addressability: parallel and serial. Parallel-addressable memory is very simple to address, send data to, and receive data from. However, a large number of address lines is required. This increases the size of the memory, as well as uses many pins of the processing device. Additionally, special circuitry is needed to multiplex the address and data lines with tri-state buffers. Serial-addressable memory removes all these problems, but can be harder to interface, since special commands must be

issued serially to the memory in order to select an address and transfer data. This issue is too often lost since many serial-addressable memory devices have software libraries written by the developers. This greatly increases their ease of use. Due to the decreased pin count, smaller size, and lack of necessary special multiplexing and buffering circuitry, we decided to use serially-addressable memory.

Having decided on non-volatile memory, there are a number of different memory types which could be used. The options we considered included non-volatile random access memory (NVRAM), electronically-erasable programmable read-only memory (EEPROM), and Flash. NVRAM works by storing data magnetically. This is a low-power and newer solution, but is much more expensive than EEPROM or Flash. Also it is not available in large capacities as current Flash. Therefore, NVRAM was ruled out as a viable option. EEPROM allows many, practically limitless writes and is very inexpensive, but is not available in as large capacities as modern Flash. Additionally, it requires a higher voltage to program than to operate. Flash is available in very large capacities and can be programmed with the same voltage as the supply. Unfortunately, it suffers from limited erase/write cycles, typically more than EEPROM. Due to the increased capacity and the ability to program with the supply voltage, we decided to use a Flash memory.

A block diagram for the memory module is shown in Figure 49 below.

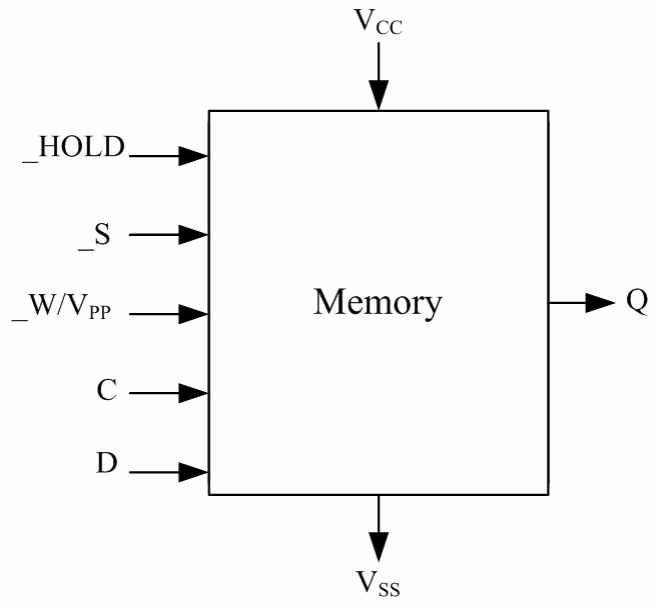


Figure 49: Memory Module Block Diagram

As seen in Figure 49, the memory module had a positive voltage supply, control pins for the memory device (_HOLD, _S, _W/V_{PP}), SPI clock, input and output (C, D, Q), and a return path to ground. These pins are described in Table 12.

V_{CC}	Positive voltage supply input.
_HOLD	Active-low memory hold. When set (low) the memory is disabled. To enable the memory, a high signal must be applied.
_S	Flash memory select line, active-low. When set (low) the memory is de-selected. To select the memory, a high signal must be applied.
_W/V_{PP}	Active-low write protect. When set (low) the memory can be written to. To disable writing, a high signal must be applied.
C	SPI clock input.
D	SPI data input.
Q	SPI data output.
V_{SS}	Return path to ground.

Table 12: Memory Module Symbol Description

4.5.1 Memory Requirements

The next step was to decide on the specifications for our serially-addressable Flash memory. The key characteristics included the following: capacity, erase/write cycles, voltage level, and serial interface.

The capacity needed was determined by the size of the data we were storing, and the fact that we wanted to design a system to record a week's worth of data. The data being stored in each reading included the following, with data size.

- Longitude – 8 bytes
- Latitude – 8 bytes
- Time – 6 bytes
- X-Axis maximum – 12 bits
- X-Axis minimum – 12 bits
- Y-Axis maximum – 12 bits
- Y-Axis minimum – 12 bits
- Z-Axis maximum – 12 bits
- Z-Axis minimum – 12 bits

The total was 31 bytes per reading, or 248 bits. If a reading was stored every two seconds, then a weeks worth of data, accounting for 604800 seconds, required 74995200 bits, uncompressed. This would require at least 75Mbits of storage. With compression, this could be reduced. This amount of storage is more than was easily available in serial flash memory devices when we first began looking into them. This figure assumed 24 hour/day recording, 7 days a week. If, instead, it was assumed data was only recorded during the work week of Monday through Friday, and that when immobile, the system did not record data (so that only approximately 20 hours per day is recorded), then only 100 hours, or 360000 seconds, need to be recorded. This would require approximately 45Mbits of storage. Therefore, we decided on using the largest available serial flash memory at the time of 64Mbit.

The number of erase/write cycles was an important characteristic of Flash memory devices. This value can range from 10,000 to over 1,000,000. Using 45 Mbits per week would effectively required one full erase/write cycle per week, in the worst case, with no compression used and without a rotating memory use algorithm. This meant that there would be 52 erase/write cycles per year. Even with only 10,000 erase/write cycles, the 64Mbit memory would last approximately 192 years. With 1,000,000 erase/write cycles, as is becoming common, the memory would theoretically last for about 19,200 years.

Unfortunately, due to end of life failure, it would be likely that the memory would become corrupt for other reasons past the 10 years of typical expected lifetime, before reaching the maximum number of erase/write cycles. Therefore, this statistic was weighed low in decided on a serial flash memory.

Since we were trying to minimize our power consumption, a lower voltage memory device was to be used. We decided to look for a device which could operate at 3.3V to meet this requirement. Additionally, it would need to be able to be programmed at this supply voltage. The main reason for this voltage choice was practical: the next step after 5V logic is down to 3.3V logic. Operating lower than 3.3V is possible, but since we could not find any GPS units at the time which would operate lower than this voltage and we wanted the majority of the system to run off of the same voltage, 3.3V was the obvious choice.

A number of serial interfaces for memory exist, but the most common two are I2C and SPI. SPI is a synchronous version of UART, requiring three lines for shared clock, transmit and receive. I2C is another synchronous serial interface, but only uses two lines, for a shared clock and a bidirectional data line for both transmit and receive. Both interfaces are straight-forward to implement on most modern microcontrollers and typically have hardware interfaces available. In the end, the choice was based on availability. For the capacity we wanted, there were many more SPI devices available at the time.

The results of the above process produced the following specifications for the memory module:

- Flash
- 64Mbit or Higher capacity
- Runs off of 3.3V logic and supply for programming
- SPI interface for serial-addressability

4.5.2 Memory Selection

We considered a number of different manufacturers of Flash memory with the previous specifications, including STMicroelectronics, Atmel, and SST. In the end, we decided to use memory from ST Microelectronics due to the vast documentation and freely-available, widely-documented C libraries for interfacing with the memory. The libraries were written in such a way so that only a minimum amount of application-specific code is required to interface with the memory device.

4.5.4 Memory Circuit Description

The schematic for the memory circuitry is shown in Figure 50.

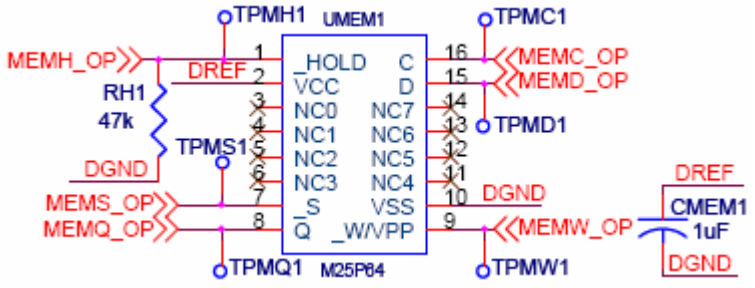


Figure 50: Memory Circuit Schematic

The M25P64 is a 64 Mbit SPI flash memory. It can run off of a 2.7V to 3.6V supply, making it ideal for our 3.3V supply. The M25P64 can operate at up to 50MHz, with an external synchronization clock signal applied to the C pin (MEMC). The device is rated for over 20 years of data retention and over 100,000 write/erase cycles. It features a 512kB sector that can be erased in one instruction and a fast bulk erase mode to erase the entire memory. Additionally, the M25P64 has very low power consumption, requiring only 100µA maximum of quiescent current, uses 8mA during a read and 15mA during a write. The M25P64 is available in a wide SO16 package as shown in Figure 51.

Figure 51: M25P64 SO16-wide Packaging (M25P64, page 1)

The capacitor, CMEM1, was a 1µF surface-mount tantalum capacitor. It was used to decouple the 3.3V digital supply line for the memory. It was located very close to the M25P64 chip on the PCB. VCC was connected to the digital supply reference and VSS to the digital ground plane, which was isolated from the other planes to reduce noise.

The resistor, RH1 was a 47kΩ surface-mount resistor used to pull the active-low _HOLD signal low to the digital ground, DGND, by default. This put the M25P64 in low power mode, disabling the part. This acted just like an enable pin on other ICs. The signal, _HOLD was connected to pin 2.4 on the microcontroller.

The active-low signal, _S, was used to select the memory. This was intended in case multiple SPI devices were connected together on the same bus. The signal, _S was connected to pin 2.5 on the microcontroller.

The active-low signal, _W/VPP, was used to enable writing to the memory. When low, data could be written to the memory; otherwise, it was write protected. It had a secondary function for fast page programming which was unused in our application. The signal, _W/VPP was connected to pin 2.6 on the microcontroller. Combined, _HOLD, _S and _W/VPP made up the control signals for the M25P64, and was connected to pins 2.4-2.6 on the microcontroller for coding convenience.

The data input, D, was connected to both SIMO0 (pin 3.1) and SIMO1 (pin 5.1) on the microcontroller. The data output, Q, was connected to both SOMI0 (pin 3.2) and SOMI1 (pin 5.2) on the microcontroller. The C signal was the SPI clock synchronization input. It was connected to both SCL0 (pin 3.3) and SCL1 (pin 5.3) on the microcontroller. These three pins, D, Q and C, made up the 3-wire SPI bus connecting to ports 3 and 5 (SPI0 and SPI1) on the microcontroller.

4.5.5 Memory Testing

Testing was performed by attempting to read the device and manufacturer identification from the M25P64 chip. This was successful, returning the “Flash_Success” value. This value is printed on the LCD. The first read always comes up as “Flash_Error”, but the second as “Flash_Success”. This value was parsed with the flash error to ASCII string converter function, which returned the string, “Flash – Success” when the “Flash_Success” error (an integer) was passed in. This was the case whether SPI0 or

SPI1 was used. A picture of the LCD showing this message is shown in Figure 52.

Figure 52: Memory Testing - "Flash_Success"

4.6 USB Interface Module

The purpose of the USB interface module was to provide input and output to a PC over a USB connection. The USB interface module allowed for transferring the data from the embedded system's internal storage to a PC. A program on the PC-side could also be used to communicate with the USB to clear memory, request data transfer, and provide a debugging interface. Additionally, it powered the device via the USB connection, in lieu of using the batteries.

A number of different interfaces could have been used to communicate between the embedded system and a PC. The main choices included RS-232 serial ports, USB, and Ethernet. RS-232 serial ports are old technology. Although they are very simple to interface and write drivers for, the ports are disappearing in modern PCs. This reason alone made RS-232 serial ports a poor decision. USB is ubiquitous, included in all PCs and even PDAs. There are many easy-to-use integrated circuits to help implement USB in a system. Receiving power over USB is trivial, since a 5V line is always present in every USB cable. Ethernet is nearly in every PC, but is less common in other devices. Additionally, using Ethernet is more difficult, potentially requiring implementing layers of the TCP/IP or UDP communication standards, and requires more work on the PC –side. Typically, more components are required to implement Ethernet, as well as more software. Also, powering over Ethernet (PoE) requires isolation transformers and more complicated circuitry. Due to the ubiquity, ease of implementation, and simplicity in powering from the interface, USB was chosen.

There are two main ways of implementing USB: implementing the interface directly or using a bridge. A direct implementation would require a great deal of knowledge about the USB standard and require implementing the USB stack in software on the embedded system side, as well as writing drivers on the PC side. A bridge, however, acts as a "black box", with a connection to the embedded system on one side, and a USB connection to the PC on the other side. The PC drivers are often provided by the manufacturers' of the bridge, making implementation on the PC-side very simple as well. Due to ease of

implementation, a USB bridge was chosen.

USB bridges that connect to microcontroller or microprocessor typically use the UART interface. A USB-UART bridge was used in our design. A simplified block diagram for the USB module is shown in Figure 53. It only shows the used pins from the USB-UART IC.

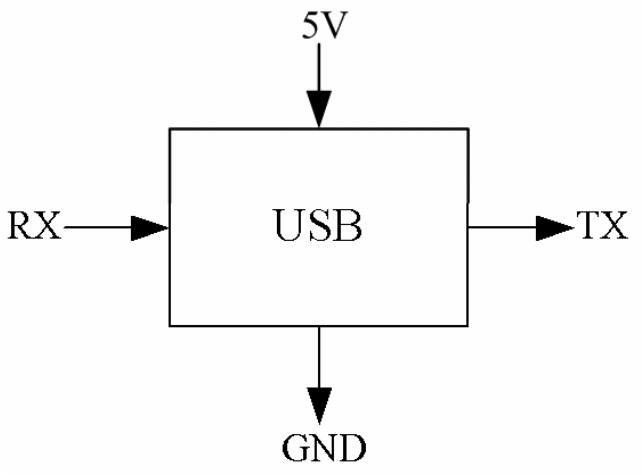


Figure 53: USB Module Block Diagram

As seen in Figure 53, the USB module has connections for input voltage, UART and ground. The pins are described in Table 13.

5V	Positive voltage supply input from the USB bus.
RX	UART receive input.
TX	UART transmit output.
GND	Return path to ground.

Table 13: USB Module Symbol Description

4.6.1 USB Module Requirements

The USB-UART bridge had a number of specifications used in determining which part to use. Firstly, it had to be compatible with the rest of the system, so it needed to be compatible with 3.3 V logic. Additionally, it should be able to be powered directly off of the USB. The USB 5 V line would also be used to power the rest of the system when it was available. Also, the USB-UART bridge needed to be able to communicate over UART with the embedded system, meaning it would have to have baud rates which the microcontroller could handle. The part also needed to be compatible with all current USB implementations, including USB 2.0, 1.1 and 1.0, even though it will not be transferring anywhere near the USB 2.0 maximum, due to limitations of the microcontroller. Finally, the driver must be well-documented with ease of use a desire.

The specifications for the USB-UART bridge are listed below.

- Compatible with 3.3V logic
- Can be powered off of USB
- UART communication with embedded system
- Compatible with USB 2.0, 1.1, 1.0
- Well-documented driver

4.6.2 USB Module Selection

There were a number of possibilities for USB-UART bridges. The devices considered included one device by FTDI, the FT232RL and two by SiLabs, the CP2102 and CP2103. In the end, the CP2102 was decided upon due to lowest cost, having more buffer space than the FT232RL, and not needing the extra features of the FT232RL and CP2103, which offer additional general purpose digital input and output pins. Additionally, a very small module was found for the CP2102.

In our design, we actually used a pre-built module for the CP2102 available from 4D Systems. This was due to the fact that our prototype and final PCB were hand-soldered, and the CP2102 is not available in a package designed for hand-soldering. The package used by the CP2102, like the accelerometer package, is designed for reflow soldering. Due to the complexity and our inexperience with reflow soldering, it was decided that a hand-solderable module would be a better solution. This module, the CP2102-microUSB, interfaces the CP2102 with a USB connector for plugging in a standard USB cable. The breakout board fits beneath the USB connector itself, making the module as small as possible. A picture of the breakout board can be seen in Figure 54 below.

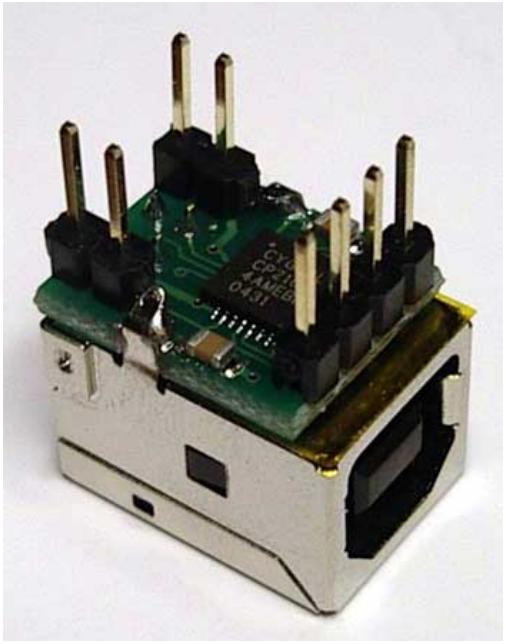


Figure 54: 4D Systems CP2102-microUSB Module (Micro-USB Module, page 1)

4.6.3 USB Module Circuit Description

The schematic for the USB-UART bridge is shown in Figure 55.

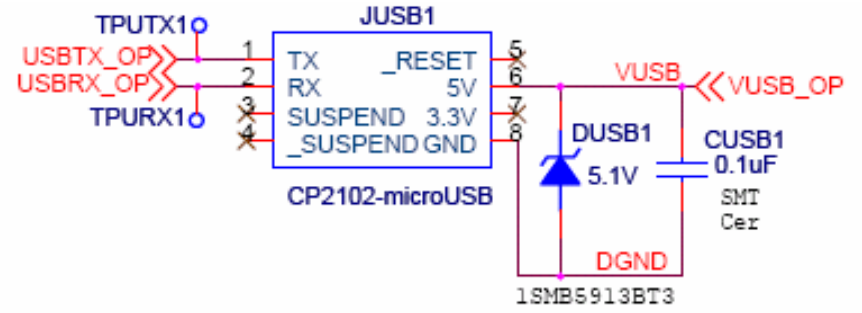


Figure 55: USB-UART Bridge Circuit

The CP2102-microUSB module was used. Some test point connections were included for easier PCB probing. The UART lines, TX, and RX were connected to URXD1 and UTXD1 on the microcontroller, respectively, corresponding to pins 3.7 and 3.6. The suspend, reset, and 3.3V output signals were unused.

The 5V output signal, labeled VUSB, was an input to the source chooser circuitry. A ceramic 0.1µF surface-mount capacitor was placed on this output to ground to provide decoupling. This helped with high-frequency noise and transients on the USB's 5V line. Additionally, a Zener diode was placed in parallel with the capacitor, to help quickly pull down transients on the line. In design, a 5.1V Zener was used. Later on, it was decided that this breakdown voltage was too close to the 5V output of the USB. A Zener with a larger breakdown voltage, perhaps 6 or 7V, should be used. On the PCB, the Zener was not populated.

The 1SMB5913BT3 was chosen here for the same reasons expressed in the power input stage description. The Zener is an ultra-fast, small surface mount diode in a SMB package with relatively high thermal dissipation ability.

4.6.5 USB Module Testing

Text was sent from the microcontroller to a PC to test the USB circuitry. A HyperTerminal client was used on the PC side, configured to receive input from a USB connection at almost 1 Mbaud. This tested the USB hardware and API, which worked. Results are shown in Figure 56.

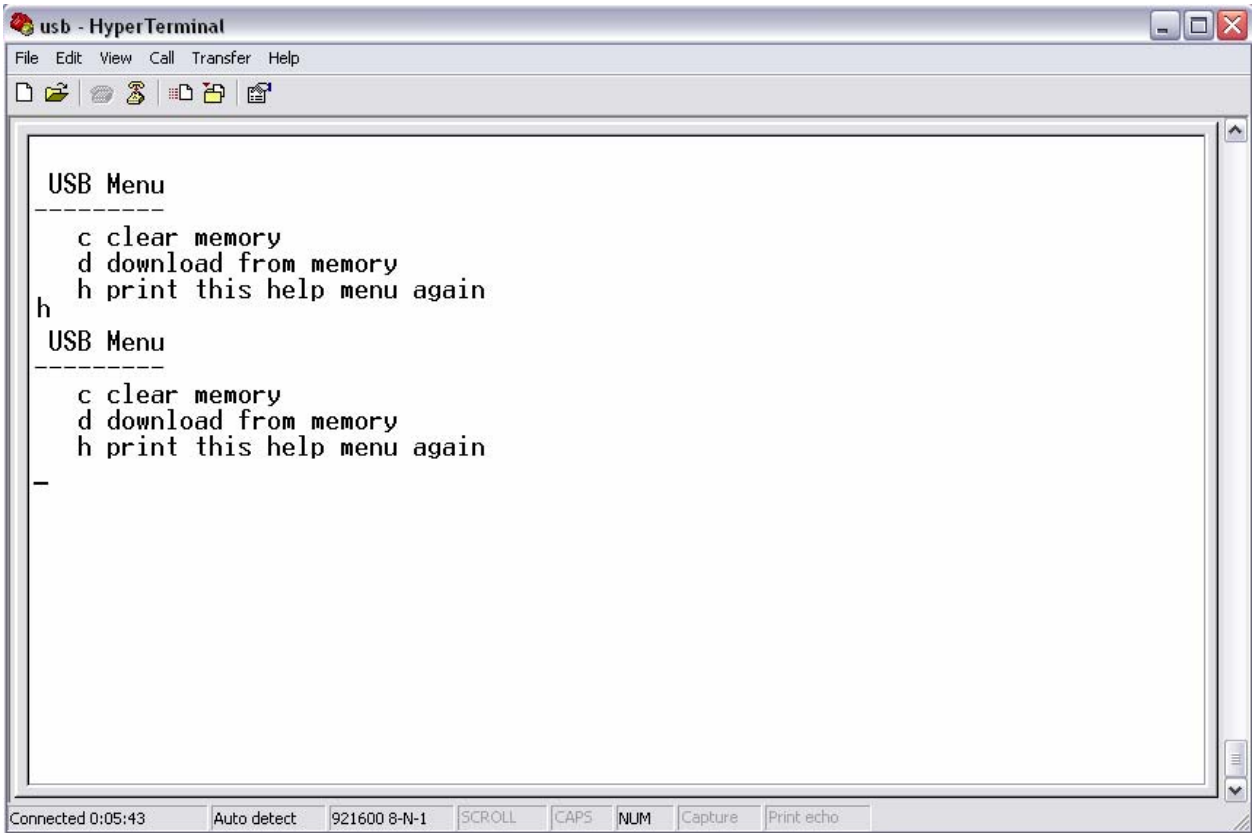


Figure 56: Testing Results of USB Module

As shown in the above figure, text was sent from the microcontroller via the UART interface to the CP2102 USB-UART bridge, where it was sent via USB to the PC and read in HyperTerminal. The figure shows the connection was set up for 921600 baud eight-none-one connection. The results show the initial printing of a menu via a call to usbPrintMenu, and then the re-printing, upon receiving the 'h' character.

4.7 LCD Module

The purpose of the LCD module was to act as a display for the embedded system. There were a number of choices for displays, including a set of LEDs, LED segment display, a LCD, and an OLED/PLED display.

LEDs and LED segments are an older technology and are therefore widely available and inexpensive. Both LEDs and LED segment displays have the advantage of being very easy to read at different angles and from a distance. Another advantage is the ability to operate off of lower voltages, which is compatible with the voltage used to drive the rest of the system. However, they have the disadvantage of large power consumption, on the order of 5-20 mA per LED or LED segment. A segment display is limited to the characters it can display, being just hexadecimal values. The user interface for LEDs is even worse, being completely binary. In order to minimize losses, an LED driver would need to be used to regulate the current, adding to the complexity and total cost.

Although a newer technology than LEDs, LCDs have been around for a long time and are also

widely available. LCDs use much less power than LEDs, on the order of 2 mA for a character display module, without a backlight. The backlight, being a LED itself, can use 20 mA or more. However, in many applications, a backlight would not be necessary. A disadvantage is that it is very difficult to find LCD modules which use lower voltages. Therefore, it would be necessary to generate a higher voltage for the LCD than for the rest of the system. In practice, this is not overly difficult, due to the availability of easy to use charge pump regulators. However, the LCD can be controlled by a lower voltage, compatible with 3.3V logic, even though the supply for the LCD is a higher voltage. This means that a level-converter is not necessary. An LCD has a lower contrast ratio and viewing angle than LEDs or OLED/PLED displays, making an LCD much less readable. LCDs have tremendously longer response time, but the response time is not very important, since the display will not be updated at very high frequencies. The one caveat is that the response time of LCDs is dependent on temperature, so the display could greatly slow down at lower temperatures.

OLED/PLED displays are a newer technology, and are much less available, especially a year ago when this project began. Currently, many new products are in the process of coming out, and their pricing is competitive with LCDs. They are very bright and easy to read, like LEDs, with a contrast ratio two orders of magnitude more than an LCD. The view angle is over twice the range of an LCD, and the response time is over five orders of magnitude greater than an LCD. Additionally, being a solid-state device, the response time of OLED/PLED displays is not very dependent on temperature. OLED/PLED displays consume much less power than LEDs. In fact, the supply current required is approximately one-third that required for similar LCD modules. However, they still require 10-30 mA of current to light the LEDs, in order to make the device readable. Therefore, they require a higher amount of power than a LCD without the backlight turned on. With the backlight turned on, the power consumption of the OLED/PLED module is slightly lower than a LCD. Additionally, there are OLED/PLED displays which are pin-for-pin compatible with the most common character LCDs, making them a drop-in replacement. OLED/PLED displays can typically operate over the same range of supply voltages as LCDs.

A trade analysis of the possible display choices is shown in Table 14 below. The three choices were LEDs, including segments, LCDs and OLED/PLED displays. They were compared on availability, power consumption, brightness, user interface, complexity to implement, and cost. The scale used for each category ranged from 1 to 5 (low to high).

Categories	**LEDs**	**LCD**	**OLED/PLED**
Availability (5)	5	5	1
Power Consumption (5)	1	5*	3
Brightness (2)	5	2	4
User Interface (4)	2	5	5
Complexity (3)	2	3	3
Cost (4)	5	3	3
Raw Score	20	23	19
Weighted Score	74	95	69

* Without the backlight

Table 14: LCD Module Selection

From trade analysis, it was decided that a LCD screen would be the most suitable display. However, it is worth noting that when OLED/PLED displays become more available, they could be a better choice for this project, if a backlit LCD was used. Using a LCD without a backlight is the best choice, due to the lower power consumption.

A block diagram for the LCD module is shown in Figure 57.

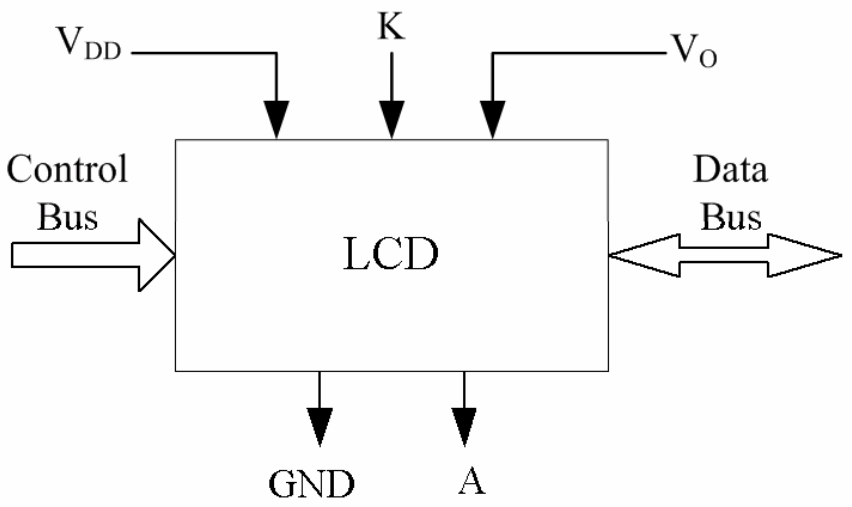


Figure 57: LCD Module Block Diagram

As seen in Figure 57, the LCD module has two supply inputs, a control bus input, a bidirectional data bus, a return path to ground and an internal LED for backlighting. The signals seen in the block diagram are described in Table 15 below.

V_{DD}	Positive voltage supply for logic.
V_O	Positive voltage supply for lighting.
K	Cathode of LED for backlighting.
Control Bus	Inputs to control functionality of the LCD, including the following: RW – read/write pin RS – reset/set pin EN – active low enable
Data Bus	Bidirectional pins for sending data to or receiving data from the LCD. DB7-DB0 for 8-bit data bus DB7-DB4 for 4-bit data bus
GND	Return path to ground.
A	Anode of LED.

Table 15: LCD Module Symbol Description

The logic supply, V_{DD} is specified to be 5.0V with respect to GND. Since the main system voltage is only 3.3V, and the batteries only supply a nominal 4.8V, a step-up regulator is required to drive the LCD. This is accomplished by using a 5.0V regulating charge pump doubler. The IC doubles the input voltage and then regulates it down to 5.0V using a shunt regulator, as found in linear regulation schemes. This provides a stable 5.0V for the LCD. More information is available in the power section.

The voltage supply for LCD lighting, V_O, is set by using a potentiometer between V_{DD} and GND. This value is temperature-dependent, but a typical value of around 2.0V from V_{DD} to V_O is suggested for a wide operation below and above room temperature. In order to simplify the process of setting this value and allow for field calibration, a trimpot is used.

The cathode and anode of an internal LED can be optionally used to turn on the backlight of the LCD. A current-limiting resistor is required between the cathode and V_{DD} in order to set the current for the LED. Positive display LCDs do not require a backlight to be visible, but negative display LCDs do. In practice, a potentiometer can be used in lieu of a discrete resistor to allow for an adjustable amount of backlighting.

The control and data bus are connected from the LCD to the microcontroller. Simple digital input/output pins on the microcontroller are used. The control bus contains three pins to enable the LCD (EN), and set the function (RW and RS). The data bus can be operated in either 8-bit or 4-bit mode. In 8-bit mode, an entire byte is sent in parallel. In 4-bit mode, the high nybble is sent, followed by the low nybble.

4.7.1 LCD Module Requirements

The LCD module acted as a visual output for the roughness detector. The LCD could be used to show pertinent run-time information, including the following:

- Mode of operation
- Number of satellites present
- When data is recorded or uploaded
- Powering mode
- Debugging information

4.7.2 LCD Module Selection

The next choice was on the type of LCD to use. There are a number of design choices, listed below:

- Character or graphics display
- Serial module, parallel module or software-driven
- Positive or negative display
- Size

The display itself could be character or graphics based. Character devices are very simple to use and operate. Graphics displays are more complicated to use and more expensive than character devices. For this project, it was realized that all of the information to be displayed could be simple text. This led to deciding on a character display.

When using LCDs, they can be controlled in software or hardware. In hardware, LCD modules are used, where the module receives commands and then translates them into logic to control the many pins of a LCD. In software, the code controls the logic to drive the LCD directly. The downfall of the software mode is that it requires much more processing time and can use many I/O pins. Additionally, the software is much more complex to interface straight to a LCD, instead of using a LCD module. However, a LCD is less expensive than getting a whole LCD module. LCD modules can be controlled via a serial or

parallel interface. Serial modules use very few I/O pins but can require more difficult software to interface and cost more than parallel modules. Parallel modules require more I/O pins than a serial module, but are relatively inexpensive and simple to interface. We decided to use a parallel LCD module, due to the lower cost than a serial module, the expansive documentation and ease of interfacing.

LCD modules may or may not require a backlight. LCDs requiring a backlight are known as negative displays, whereas LCDs that don't require backlights are known as positive displays. The advantage of negative displays is that they are much brighter than positive displays, can be read in the dark, and are much easier to read. However, due to requiring a backlight, negative displays draw more current than a positive display. Positive displays can be harder to read than negative displays and cannot be read in the dark, but consume less power than a negative display. Positive displays can have an optional backlight which allows them to be read in the dark. A positive display was decided upon, with optional backlight, due to the large power savings.

LCD modules come in many sizes. We did not need to display a lot of information, so a 8x2 or 16x2 module would be able to display all the necessary information. By using a parallel module, we could simply plug in a bigger screen and only make slight modifications to the code to make it work out-of-the-box.

The LCD has to work with the rest of the system. This means that it must be able to be interfaced with the microcontroller on a logic level. The most common and most well-documented parallel LCD module is the HD44780, which was chosen due to its documentation. This is compatible with 3.3V logic but requires a 5V supply. Since the batteries for our system cannot provide 5V, some step-up regulation would be required. The choice of a charge pump to provide a regulated 5V for the LCD supply is explained in the power supply section.

The requirements for the LCD are as follows.

- Parallel HD44780-compatible module
- Positive display with optional backlight
- 8x2 or 16x2 text display

The company, CrystalFontz, makes many different HD44780-compatible LCD modules. We chose to go with CrystalFontz because of previous experience using their LCDs, expansive documentation, and availability. Any HD44780-compatible LCD module will work with our hardware design, but for testing purposes, we chose the CFAH0802A-YMI-JP 8x2 negative module from CrystalFontz and the CFAH1602A-YYH-JP 16x2 positive module with optional backlight. The CFAH0802A-YMI-JP can be seen in Figure 58 and the CFAH1602A-YYH-JP can be seen in Figure 59.

Figure 58: CFAH0802A-YMI-JP LCD Module (CFAH0802A, page 1)

Figure 59: CFAH1602A-YYH-JP LCD Module (CFAH1602A, page 1)

4.7.3 LCD Module Circuit Description

The schematic for the LCD circuitry is shown in Figure 60.

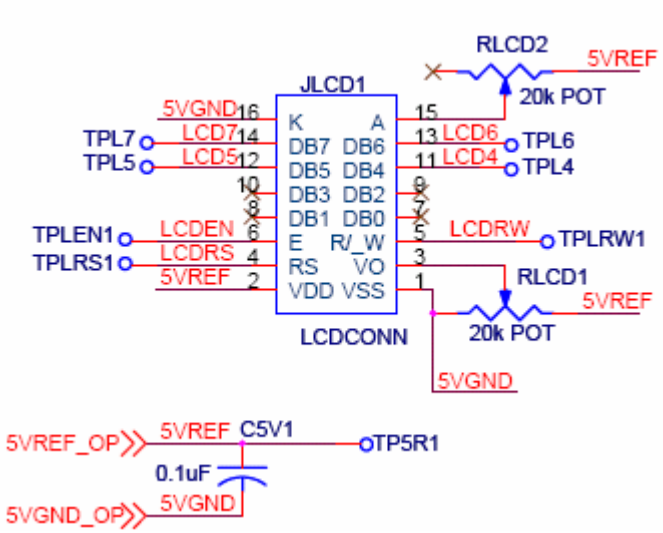


Figure 60: LCD Circuitry Schematic

The connector, JLCD1, was a small through-hole male connector. There was a cable from this connector, mounted on the PCB, to the LCD. The LCD was used in 4-bit mode, with data bus lines DB7-DB4 connected to the microcontroller at pins 4.7-4.4. The control lines, enable (EN), read/write (R/_W) and data/command mode set (RS) were connected to the microcontroller on pins 4.1-4.3 respectively. With this configuration, the entire LCD used only one of the six available ports on the microcontroller (with pin 4.0 unused by the LCD).

The LCD was powered by the 5V reference from the charge pump regulator. The ceramic 0.1µF capacitor, C5V1, acted as a decoupling capacitor to compensate for high frequency transient events and noise, causing a more stable 5V to be applied to the LCD.

The input voltage, VDD was tied to the 5V reference which was the output of the charge pump regulator. This was the logic voltage input of the LCD. The ground, VSS was tied to the 5V ground plane, which was isolated from the other ground planes. RLCD1 was a 20kΩ 12-turn side-adjust miniature

through-hole potentiometer which was used to see the display voltage of the LCD, VO. This was set so that the voltage, VDD-VO was approximately 4.0V at 25 °C. If a different ambient temperature was present, this potentiometer could be turned to allow for the correct VDD-VO, based on temperature. The brightness of the backlight of the LCD could be controlled with RLCD2. RLCD2 was a 20kΩ 12-turn side-adjust miniature through-hole potentiometer. This type of potentiometer was used to minimize the cost and improve mechanical connection, as compared to a surface-mount potentiometer. It provided an easy-to-use side adjustment in a small package and a large enough range of resistance values to accurately and easily trim into a selected value. The resistor, RLCD2, limited the current going into the backlighting LED of the LCD. This was typically 70mA for CrystalFontz LCDs with a backlight. This value could be adjusted to increase brightness or could be turned down to decrease backlight brightness, while greatly increasing battery life. Both RLCD1 and RLCD2 were PV37P potentiometers, made by Murata, as shown in Figure 61.

Figure 61: PV37P Potentiometer (Trimmer, page 4)

4.7.4 LCD Module Testing

The LCD was tested by writing the LCD API and a sample application for the microcontroller to display text on the LCD. This worked well. The results are shown in Figure 62.

Figure 62: LCD Module Testing Results

4.8 Summary

In this chapter, each module was shown how they were designed. First, the specifications for each module were derived. Each module was then designed to fit those specifications (as best as we could). The circuits were then shown with an explanation of why we made those decisions. Finally, each module was tested to verify that our design worked.

5 SOFTWARE DESIGN

This chapter explains the software design in detail. First, the data structures used will be explained. Libraries and APIs designed will be described next. A section discussing modes of operation will follow. The last section talks about the code for Google Earth.

5.1 Data Structures

A number of data structures were used to organize the software design. This section will explain each of them.

5.1.1 IOdevice

The *IOdevice* data structure was the top-view encapsulation used for interfacing serial devices, including the GPS, memory, and USB bridge devices. The structure contained the following information:

- Transmission buffer
- Reception buffer
- Receive function
- Send function
- Port

The transmission and reception buffers were instances of *IObuffer*. They were used as temporary storage which was accessed by the programmer through the APIs. They provided buffering for serial communication. They allowed the programmer to access information through the serial device without having to directly read from or write to internal registers, which are actually used to send or receive information. Additionally, they provided an abstraction to allow the receive and send functions to work in a general case.

The receive and send functions were handled as function pointers. This allowed for generalization, where each device could have a specific send or receive function associated with it. This was important, especially for the memory device, which could be operated on either USART port, and required distinct functions for handling transmission and reception depending on which port was being used. By providing function pointers, the programmer could simply call the send or receive function without caring which true function was actually being called.

The port was an instance of *IOport*. This was used to distinguish which port a device was currently operating on. The MSP430 only allowed for one set of transmit and receive interrupts per USART port, whether it was being operated in UART or SPI mode.

There were two *IOdevice* instances used, one for each of the two USART ports: *dusart0* and *dusart1*. The GPS, memory, and USB bridge were implemented as handles (pointers) to *IOdevices*: *hgps*, *hsfm*, and *husb* respectfully. As the hardware was configured, *hgps* always pointed to *dusart0*, since the GPS device was always connected to the first UART. Likewise, *husb* always pointed to *dusart1*, since the USB bridge was always connected to the second UART. The serial flash memory (SFM), with handle *hsfm*, would point to either *dusart0* or *dusart1*, depending on whether it was using SPI0 or SPI1.

One main reason for this type of implementation was the MSP430's interrupt vector

implementation. Each serial port (USART0 and USART1) had only two interrupt vectors, transmission and reception. Both UART0 and SPI0 (USART0) used the same interrupt vectors, as did UART1 and SPI1 (USART1). Thus, in order to run the appropriate function when the USART0 interrupt was called, the program either needed to know which mode it was in and have an ability to check this within the interrupt, or some abstraction was needed. The *IOdevice* and function pointers provided this abstraction. For example, during a USART0 transmit interrupt, just the receive function pointer for *dusart0* was called. As the function pointer was de-referenced, either the GPS or memory receive functions were called, depending on the mode of operation.

5.1.2 IObuffer

The *IObuffer* structure was an encapsulation of a buffer to be used for *IOdevice* instances. The *IObuffer* structure had the following distinct elements:

- Character buffer
- Length of used buffer
- Maximum size of buffer
- Status
- Position in buffer (transmitting) or stop byte (receiving)

The character buffer was implemented as a character pointer which was pseudo-dynamically allocated. In reality, there was a number of static character arrays allocated. This character buffer was simply a pointer to one of the static character arrays. Implementing in this fashion meant that as an operating mode was changed, a different *IOdevice* could point to one of the now unused static character arrays. This saved memory, since not every device required its own static memory space. Additionally, it required less processing and was more optimized than trying to do real dynamic allocation, through using *malloc* and *free*. Also, it required no garbage collection.

The length of the used buffer and maximum size of the buffer were used to manage storage in the buffer. Other routines could use these to make sure there was no buffer overflow.

The status flag was an instance of *IOstate*. It was used effectively as a semaphore to tell whether or not the buffer was free.

A union was used to store either the position in the buffer, when transmitting, or an optional stop byte, when receiving. When transmitting, the position was used to identify where the next character of the buffer should be read from. The stop byte could be used to stop reception upon receiving a specific character. This was used as a serial message delimiter.

5.1.3 IOstate

IOstate was implemented as an enumeration which could have two states: *free* and *used*. The purpose of this enumeration was to act as a status flag or semaphore. The semaphore was used to allow or disallow access to a specific buffer. This allowed for resource availability checking, so multiple interrupts did not access the same buffer at the same time.

Additionally, the status was used to check if an expected interrupt was finished. For example, when the GPS was read from, the state was first set as *used*. Then, once the reception from the GPS had completed or overflowed, the state was set to *free*. The main application used a wait loop to check if the status flag was *free*. If it was still *used*, then the application would do something else.

5.1.4 IOport

IOport was implemented as an enumeration which could have one of four values: UART0, UART1, SPI0, SPI1. The purpose was to identify which serial device was being used by an *IOdevice*.

5.1.5 AXLint

AXLint was a type definition for the values read in from the accelerometer via the ADC. For our application, this was an unsigned integer, providing 16 bits. In reality, the ADC was only 12 bits, but the next step down, a short integer, would only provide 8 bits.

5.1.6 AXLaxis

The *AXLaxis* structure encapsulated a single axis of an accelerometer. This structure could be application specific. For our application, it was decided that we would record the worst readings during a time frame or a rolling average value during the time frame. Therefore, the *AXLaxis* structure held the following data:

- Current/latest reading
- Rolling average or maximum and minimum values read

The current accelerometer reading was implemented as an *AXLint* type. It held the latest read in conversion value from the ADC. The other element of the structure was implemented as a union which either held a rolling average or two values: the maximum and minimum values read. In each case, the values stored were of type *AXLint*.

The values were automatically updated on each ADC conversion with code in the accelerometer API. By changing the data stored in this structure and minimal accelerometer API code, other data could be stored in addition to the latest reading, such as an RMS value.

5.1.7 AXL3

The *AXL3* structure was used to encapsulate an entire triple-axis accelerometer. It had three *AXLaxis* instances, one for each of the three axes: X, Y and Z. Multiple AXL3 instances could be used for a multiple accelerometer system, with minor modification to the accelerometer API.

5.1.8 Timer

The *Timer* structure was used by the timer API. It provided a generalized encapsulation for MSP430 hardware timers. The *Timer* structure had the following elements:

- *Start* function
- *Stop* function
- Number of iterations to count
- Current number of iterations run
- *Timer* offset
- *Timer* status

The *start* and *stop* functions were implemented as function pointers. They were only called internally. This allowed for an abstraction of the timer. The user could simply call for a hardware delay on

a specific timer, using *hwDelay* of the timer API. The appropriate start and stop functions were called by dereferencing the *start* and *stop* function pointers.

The number of iterations to count was set in the main application. This told the timer how many times it should count up and overflow. By setting this value and the timer offset, the hardware delay could be set. The current number of iterations run was an internal value to keep track of how many timer overflows had taken place. The timer offset could be set to achieve hardware delays which were not a multiple of the timer overflow time. For example, if the timer was set to overflow every one second, the offset could be set to achieve a hardware delay of just 500ms. Likewise, with a combination of timer offset and number of iterations to count, a 2.5 second hardware delay could be set.

The timer status value was implemented as a *TimerStatus*. The value of the timer status told if the timer had finished or not.

There could be multiple timers, depending on the architecture and application. For our application, only *Timer A* from the MSP430 was used. One instance of *Timer*, *timerA*, interfaced with this hardware timer.

5.1.9 TimerStatus

The *TimerStatus* enumeration was used to check the status of a timer. It could have one of two values: done and running. When the timer was counting, it would be set as "running". When the timer had finished counting, the status would change to "done". The intent was for the main application to use a wait loop to check the status of the timer.

5.2 Libraries and APIs

A number of libraries and APIs were designed. They are explained in this section.

5.2.1 Accelerometer

The accelerometer was driven by on-demand IO. This meant that the accelerometer was only read when a specific function was called, in lieu of interrupt-driven IO. The header file for the accelerometer API included a number of application specific constants. The values could be changed and the source re-compiled if, for example, the accelerometers were not connected to channels 0, 1 and 2 of the ADC.

The accelerometer API included the four following functions:

- *axlInit*
- *axlReset*
- *axlConvert*
- *axlRead*

The *axlInit* function was used to initialize the ADC channels for the accelerometer. For this application, channels 0, 1 and 2 were set up for measurement with respect to AVss, the MSP430's analog ground. This meant that the voltage could not be higher than AVcc, the MSP430's analog power. Since the accelerometer and the MSP430's analog power shared the same source, this was not an issue. At the end of the *axlInit* function, ADC conversion was enabled. Additionally, *axlReset* was called.

The *axlReset* function initialized the accelerometer device, an instance of AXL3. For this application, each of the three axes had their current value set to zero, maximum value set to zero, and minimum value set to INTMAX. This ensured that the first reading would replace the current value and

both the maximum and minimum values. This function could be called at the end of each datalogging period, to clear the accelerometer device values.

The *axlConvert* function was called internally to read a conversion from the ADC into ADC memory registers. The ADC memory registers, ADC12MEMx, where x is the channel, stored the newly converted values. This function then went into a wait loop to hang until the conversion was done.

The *axlRead* function was called externally by the programmer in order to request a new accelerometer reading. It called the *axlConvert* function to issue a conversion and then stored the new data in the accelerometer device structure. It checked if the new value was a new maximum or minimum, and updated the appropriate field in the accelerometer device structure.

5.2.2 DIP Switch

The DIP switch API provided a simple interface to reading from the DIP switch. The header file for the DIP switch API included a number of application specific constants, including the port and bits to use for the DIP switch. These values could be changed if a larger or smaller DIP switch was used, or if the DIP switch was connected to a different port. In our application, the DIP switch was configured for Port 1, bits 0-3 (P1.0, P1.1, P1.2, P1.3). Additionally, there were a number of constants defined for the different modes of operation.

The DIP switch API had two functions:

- *dipswInit*
- *dipswRead*

The *dipswInit* function set up the application specific port for input by the DIP switch. The *dipswRead* function could be called by the programmer to immediately read from the DIP switch. It returned the byte read in. The intent was to use the *dipswRead* function to read from the DIP switch and compare the returned value against one of the defined constants for mode of operation.

5.2.3 GPS

The GPS API provided interrupt-driven IO code for communicating with the GPS device. The GPS API had the following five functions:

- *gpsInit*
- *gpsEnable*
- *gpsDisable*
- *gpsRecv*
- *gpsSend*

The *gpsInit* function initialized USART0 for GPS communication. The USART0 was set up for a 4800 baud 8-none-1 UART connection with no flow control. The serial device was configured for interrupt-driven IO. Additionally, the function set up the *dusart0 IOdevice* structure for using the GPS on USART0. This included setting the receive and send function pointers to *gpsRecv* and *gpsSend*, respectively. Also, the stopbyte was set to “0x0A”, which corresponded to the end of a NMEA sentence. This meant that data would be read into the receive buffer until the end of the NMEA sentence was reached.

The *gpsEnable* and *gpsDisable* functions were used to enable and disable the interrupts for the GPS device. These were used so that each of the interrupt-driven devices could be individually enabled or disabled for interrupt service.

The *gpsRecv* function was used to receive and buffer characters from the GPS via a UART connection. For our application, this meant reading from Port 2 of the Lassen IQ, which was configured for NMEA output. The receive function also did boundary checking on the receive buffer, so that buffer overflow did not occur. When a stopbyte was reached or an overflow has occurred, the status flag was freed.

The *gpsSend* function was used to transmit buffered characters to the GPS via a UART connection. The intent would be to connect to Port 1 of the Lassen IQ, which is configured for TSIP interface. This port was used to configure the Lassen IQ. To actually implement this, another initialization function would be required, since the TSIP interface uses 9600 baud, not 4800 baud. For our application, the default configuration of the Lassen IQ was all that was needed, so this functionality was not implemented.

5.2.4 LCD

The LCD API was used to interface with a HD44780-compatible LCD character module. The API could use both the 8-bit or 4-bit interfaces. An application specific constant, LCD8BIT, could be defined to use the 8-bit code; otherwise, the 4-bit code would be used. The application code could be written once and then simply by setting or not setting LCD8BIT, the appropriate interface code would be compiled. This presented a layer of abstraction, since the main application programmer did not need to know or care if the actual hardware implementation was using eight or four data lines. In fact, this hardware implementation could change, with no change in application code.

Additionally, a number of application specific constants were defined to specify which port of the MSP430 the LCD is using. When set for 8-bit mode, this would encompass the entire port. When set for 4-bit mode, the high nybble would be used. In our application, Port 4 of the MSP430 was used with the 4-bit interface. Therefore, pins P4.4, P4.5, P4.6 and P4.7 were used. The LCD API had nine functions:

- *lcdInit*
- *lcdClock*
- *lcdWrite*
- *lcdCmd*
- *lcdLine1*
- *lcdLine2*
- *lcdClear*
- *lcdPrint*
- *lcdPrints*

Of these nine, only *lcdInit*, *lcdPrints*, *lcdClear*, and perhaps *lcdPrint* would typically be called in the main application. The functions, *lcdClock*, *lcdWrite*, *lcdCmd*, *lcdLine1*, and *lcdLine2* were internal calls.

The *lcdInit* function initialized the LCD for either 8-bit or 4-bit communication, as appropriate. The initialization routine for a 4-bit interface is shown in Figure 63. The 8-bit initialization was very

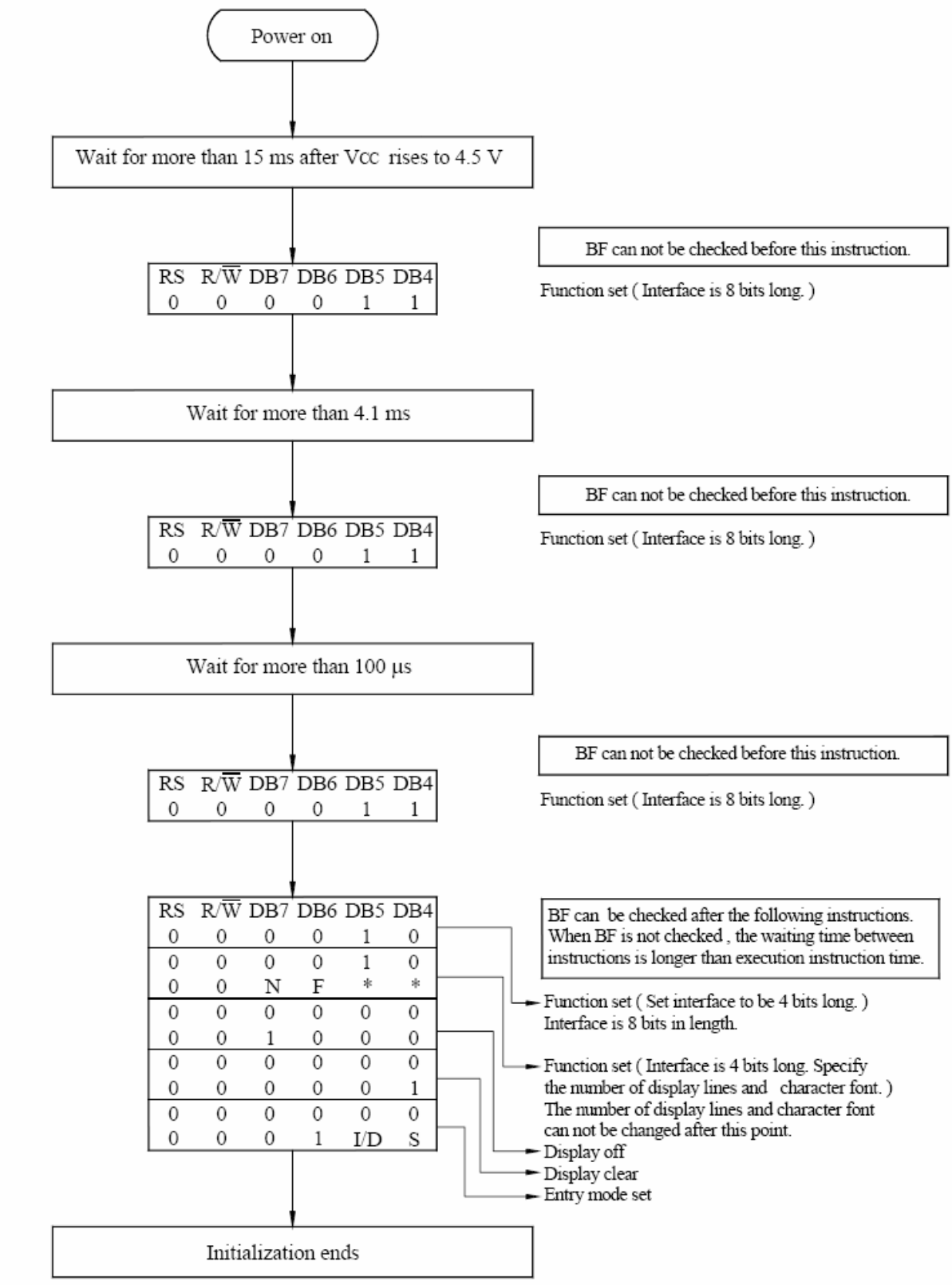


Figure 63: LCD Initialization (4-bit) (CFAH1602A-YYH-JP, page 17)

similar. The delays were implemented through software delays, using the *swDelay* function from the timer API. Each of the lines specifying bit values were implemented through calls to *lcdCmd*. The display clear was implemented through a call to *lcdClear*.

Once the data bits were set on the LCD, the LCD needed to have the enable flashed in order to read in and latch the current values. This was accomplished with the *lcdClock* function. The *lcdClock* function cleared the enable, waited 4ms, set the enable, waited 2ms, and then cleared the enable again. This forced the LCD to latch the current data bits' values.

The *lcdWrite* function was used to write data onto the data lines of the LCD and cause the LCD to read in and latch this data. An 8-bit value was passed into the function, which got written to the appropriate pins of the MSP430 which were connected to the LCD. Then, with a call to *lcdClock*, the data was latched in the LCD. In 8-bit mode, the data was written all at once. In 4-bit mode, only the high nybble was written.

The *lcdCmd* function was used to send a command to the LCD. It used the same code, whether a 8-bit or 4-bit interface was used. The RS and RW inputs of the LCD were both set low, to specify that a command was being entered. A 8-bit value was passed into the function, which would be written to the data lines with *lcdWrite*. This value specified which function was called. A list of commands is shown in Table 16. In 4-bit mode, two calls to *lcdCmd* were required, once for the high nybble and then for the low nybble.

Instruction	Instruction Code										Description	Execution time (fosc=270Khz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "00H" to DDRAM and set DDRAM address to "00H" from AC	1.53ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.53ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction and enable the shift of entire display.	39μ s
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Set display (D), cursor (C), and blinking of cursor (B) on/off control bit.	39μ s
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39μ s
Function Set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL:8-bit/4-bit), numbers of display line (N:2-line/1-line)and, display font type (F:5×11 dots/5×8 dots)	39μ s
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39μ s
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39μ s
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0μ s
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43μ s
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43μ s

Table 16: LCD Commands (CFAH1602A-YYH-JP, page 13)

The *lcdLine1* and *lcdLine2* functions were used to set the LCD cursor to the beginning of row one or row two, respectively. They were implemented as defined macros, making appropriate calls to *lcdCmd*. Likewise, *lcdClear* was a defined macro, making appropriate calls to *lcdCmd* and then *lcdLine1*. The *lcdClear* function cleared the LCD and set the cursor back to the first position of row one.

The *lcdPrint* function printed an ASCII character on the LCD. First R/W was set low to enable writing to the LCD, and then RS was set high to specify that the data was not a command, but should be printed to the display. Then, the HD44780-compatible LCDs simply needed an ASCII value on their data lines to print the character on the screen. In 8-bit mode, one call to *lcdWrite* was made to print the whole 8-bits passed into the function. In 4-bit mode, first the high nybble and then the low nybble were passed into calls to *lcdWrite*.

The *lcdPrints* function was used to print a string starting at the cursor on the LCD. It used the same code regardless of whether an 8-bit or 4-bit interface was used. The entire string was printed to the LCD. This might cause the string to wrap around and start overwriting characters. This was left to the responsibility of the application programmer. When the newline character was encountered, *lcdLine2* was called. Again, it was up to the application programmer to know if there was too much data to display on one line before the newline character was called; no row wrapping was implemented. Each character was passed into a call to *lcdPrint*.

5.2.5 Memory

The memory API provided an interface to the serial flash memory (SFM) device connected through the SPI interface. The code worked with a library provided by STMicroelectronics. The library had to be modified to work with the MSP430 and for our specific application. The modifications included writing six functions, as follows:

- *SelectSlave*
- *DeSelectSlave*
- *EnableTrans*
- *DisableTrans*
- *EnableRcv*
- *DisableRcv*

SelectSlave and *DeSelectSlave* functions toggled the active-low select line (S) of the SFM device. The *EnableTrans* and *DisableTrans* functions toggled the active-low hold line (HOLD) of the SFM device. The *EnableRcv* and *DisableRcv* functions toggled the active-low write enable line (W) of the SFM device. These functions were used internally by the STMicroelectronics library.

In the STMicroelectronics library, the function Serialize was used as a general purpose function to send data to and receive data from the SFM device. All the other functions to perform specific tasks, such as writing, reading, erasing a sector, et cetera, worked by going through the Serialize function. The Serialize function had to be modified to work with the MSP430 and in our particular application. The code was modified to use the *IOdevice* for the SFM device (*hsfm*). This allowed a level of abstraction, so that the same library code would work, regardless of what SPI port the SFM device was connected to.

The memory API included ten functions, as follows:

- *sfmInit0*
- *sfmInit1*
- *sfmEnable*
- *sfmDisable*
- *sfmRecv*
- *sfmSend*
- *sfmPost*
- *sfmErase*
- *sfmBuffer*
- *sfmFlush*

The *sfmInit0* and *sfmInit1* functions were used to initialize the SFM device for SPI0 and SPI1, respectively. Both functions set up the MSP430's port 2, bits 2 through 4 to the SFM device's active-low hold (HOLD), select (S), and write enable (W). Either SPI0 or SPI1 was configured for interrupt-driven IO using the external 8 MHz crystal with an 8-bit SPI interface. The actual clock that drove the SFM device was divided by two, so a 4 MHz clock was used. Either *dusart0* or *dusart1*, corresponding to the *IOdevice* for USART0 or USART1, was configured to use *sfmRecv* and *sfmSend* for the receive and send functions. Also, transmit and receive buffers were set up. An integer, *sfmpos*, was initialized. The purpose of *sfmpos* was to hold the current position in the memory for writing. There was an additional buffer set up, *sfmbuffer*, of type *IObuffer*. The *IOdevice* handle for the SFM device, *hsfm*, was set to point to either *dusart0* or *dusart1*.

The buffers internal to the *IOdevice* were used in the Serialize code, but the *sfmbuffer* was used to store what would be sent, before it was sent. When data was being sent, it was transferred to the internal buffers of the *IOdevice*. The intent was to allow for buffering of a lot of data in the *sfmbuffer* using *sfmBuffer*, before the data was flushed out, using *sfmFlush*. After initialized for the appropriate port, the rest of the memory API and STMicroelectronics' library were generalized to work for either port.

After initialization, *sfmpos* could be set to the next free area of the SFM device. This was accomplished by scanning the memory and finding the first location with a block of 0xFF values. This was the value given when the memory was erased. By searching for a contiguous block, the occasional 0xFF would be ignored.

The *sfmEnable* and *sfmDisable* functions enabled and disabled the interrupts for the SFM device. This allowed for independently selecting which interrupts were allowed to run, instead of globally allowing all interrupts. The global interrupt flag must also be set in order for the USB interrupts to be triggered, by using the _EINT macro.

The *sfmRecv* function was used to receive data from the SFM device. By looking at the port value stored in the *IOdevice* which *hsfm* points to, either RXBUF0 or RXBUF1 (for SPI0 and SPI1) was read from and added to the receive buffer. Boundary checking ensured that the buffer did not overflow. This function would be called when the receive interrupt was triggered for the port the SFM device was using.

The *sfmSend* function was used to send data to the SFM device. By looking at the port value stored in the *IOdevice* which *hsfm* pointed to, either TXBUF0 or TXBUF1 (for SPI0 and SPI1) was given the next value in the transmit buffer. Once the transmit buffer was empty, the transmit status flag was set

to *free*, and the index and length were reset to zero. This function would be called when the transmit interrupt was triggered for the port the SFM device was using.

The *sfmPost* function was used for a Power-On Self Test (POST) for the SFM device. First, the manufacturer ID was read from the SFM device using the Flash function, provided by the STMicroelectronics' library. The Flash function was used for all of the commands to send and receive data between the MSP430 and SFM device. If this was not the value specified for the M25P64, then Flash_Error was returned. Otherwise, Flash_Success was returned. A loop, waiting for Flash_Success was used to make sure the right device was used. This loop timed out after a number of iterations, to make sure the program was not held up in this loop forever.

The *sfmErase* function was used to erase the entirety of the SFM device. After a countdown from five, displayed on the LCD screen, the Flash function was called to erase the entire memory.

The *sfmBuffer* function was used to buffer data sent to the SFM device into the *sfmbuffer IObuffer*. The *sfmBuffer* function provided boundary checking so that buffer overflow did not occur. If the buffer was full, SFMFULL was returned; otherwise, SFMNFULL was returned. This allowed the main application program to keep adding data to the buffer until it was full, and then send it all along to the SFM device, using *sfmFlush*.

The *sfmFlush* function was used to send along all of the data in *sfmbuffer* to the SFM device. The Flash function was used to program the SFM device with the data in *sfmbuffer* starting at position stored in *sfmpos*. The position, *sfmpos* was then updated.

5.2.6 Timer

The timer API provided an interface to software and hardware delays. It included the *Timer* and *TimerStatus* data structures. The timer was interrupt-driven. An interrupt was called every time that the timer overflowed. The interrupt was associated with the specific hardware timer. For example, in our application, only *Timer A* was used; the *Timer A* interrupt was called whenever its value overflows.

There were six functions in the timer API, as follows:

- *taInit*
- *taStart*
- *taStop*
- *timerA_interrupt*
- *hwDelay*
- *swDelay*

The *taInit* function was called to initialize *Timer A*. The function associated *Timer A* with the *timerA* instance of the *Timer* structure, pointing the *start* and *stop* functions to *taStart* and *taStop*, respectively.

The *taStart*, *taStop* and *timerA_interrupt* functions were internally called to interface with *Timer A*. The *taStart* function set the timer A offset register, TAR, to the timer offset, and set up the timer A control register, TACTL, to use ACLK as a reference, be interrupt-driven, and continuously count. Upon calling the *taStart* function, *Timer A* began to count. It was set to overflow after one second.

When *Timer A* overflowed, the *timerA_interrupt* function was called automatically. This function incremented the timer iteration counter and checked to see if counting was done. If so, *taStop* was called.

Otherwise, *Timer A* was instructed that the interrupt has been processed and it should continue counting.

The *taStop* function was called when the timer had finished counting. This set TACTL to stop counting and set the timer status to done.

The application programmer interface to the hardware timers was through the function, *hwDelay*. This function had a number of arguments, as follows:

- Pointer to timer to use
- Number of iterations to count
- Timer offset to use

The pointer to the timer to use was available so that *hwDelay* could work with any *Timer* instance. The number of iterations to count and timer offset to use were set in the de-referenced timer. Then, the *start* function was called. Since the *start* function was a function pointer and the *Timer* passed in was a pointer, the *hwDelay* function would work for an arbitrary number of *Timer* instances.

The direct advantage of the hardware timer was that it was much more accurate than the software timer. The software timer was directly proportional to the clock frequency, which changed rapidly on the MSP430 in order to optimize power consumption. An indirect advantage was due to the interrupt-driven nature of the MSP430's hardware timers. The application programmer could set up a timer to start running and then perform some other task while waiting for the status value to be "done". This was used in a number of places in our application code.

The software delay function, *swDelay*, was used to make a software delay. It had two arguments, as follows:

- Delay length
- Delay multiplier

The actual delay was intended to be the delay length multiplied by the delay multiplier. There were a few experimentally determined constants defined to set up a half-second (DHSEC), millisecond (DMSEC), and 100-µsecond (DHUSEC) delay multipliers. For example, if a 20ms delay was intended, then the delay length would be set to 20 and the delay multiplier would be set to the millisecond constant (DMSEC).

The delay was accomplished by simply running through two nested loops. Therefore, the accuracy of the timer was not very high. This, coupled with the changing frequency of the MSP430, made the overall accuracy of the software delay poor. It was not intended to be used when an accurate delay was required. In our application, we used the software delays only for LCD initialization and in few other places, in order to get delays on the order of hundreds of microseconds, or less than one hundred milliseconds. These were situations where we did not yet have, or want, interrupts enabled, and thus could not use the interrupt-driven hardware timers.

5.2.7 USART

The USART API provided generalized interrupt code for use with UART0, UART1, SPI0, and SPI1. The MSP430 implemented a shared set of interrupt vectors for UART0 and SPI0, as well as for UART1 and SPI1. There were two interrupts, the transmit and receive interrupts. The USART API provided four interrupt service routines (ISRs), as follows:

- *usart0_tx*
- *usart0_rx*
- *usart1_tx*
- *usart1_rx*

The *usart0_tx* and *usart0_rx* ISRs were shared between UART0 and SPI0 for transmit and receive interrupts, respectively. The *usart1_tx* and *usart1_rx* ISRs were shared between UART1 and SPI1 for transmit and receive interrupts, respectively. In each case, the *IOdevice*, *dusart0* or *dusart1*, called the send or receive function that it was pointing to. This was the purpose of the function pointers, to allow this simple redirection in the ISRs.

In the receive interrupts, there was also a line to set a random scratch variable to the value in RXBUF0 or RXBUF1. The purpose of this was to actively pull out the value in RXBUF0 or RXBUF1, which told the MSP430 that the ISR had been processed.

In any case, the global interrupt enable (GIE) must be set to allow interrupts to be triggered. The macro, _EINT, was used to set this flag. Likewise, the macro, _DINT, was used to clear this flag, disabling all interrupts.

5.2.8 USB

The USB API was used to interface with the USB-UART bridge. The USB API provided nine functions, as follows:

- *usbInit*
- *usbEnable*
- *usbDisable*
- *usbRecv*
- *usbSend*
- *usbPost*
- *usbPrint*
- *usbPrintMenu*
- *usbDump*

The *usbInit* function initialized the USB device. The result was that USART1 was set up for a 921600 baud 8-none-1 UART connection. The external 8 MHz crystal was used to clock the USB communication. The connection was set up with no flow control and was interrupt-driven. The *IOdevice*, *dusart1*, was configured to use *usbRecv* and *usbSend* for receive and send functions. The newline character was set as the stop byte for receiving. The intent was to enter commands via HyperTerminal over the USB connection. The command would be terminated with a newline. The *IOdevice* handle for the USB device, *husb*, was set to point to *dusart1*.

The *usbEnable* and *usbDisable* functions were used to enable and disable the interrupts for the USB device. This was independent of any other interrupts. The global interrupt flag must also be set in order for the USB interrupts to be triggered, by using the _EINT macro.

The *usbRecv* function was used to receive data from the USB-UART bridge. The intent was to send data from a PC through a USB connection to the bridge. This function received that data. The data in RXBUF1 was stored in the USB receive buffer. Boundary checking was done to ensure buffer overflow did not occur.

The *usbSend* function was used to send data to the USB-UART bridge. The intent was to send data from the microcontroller to the bridge where it was automatically forwarded along to the PC through a USB connection. The current byte in the transmit buffer was stored to TXBUF1, which caused it to be sent via the UART connection. The position in the transmit buffer was updated. If the transmit buffer had been emptied, the transmit status flag was set to free. The intent was for the main application program to wait until the transmit flag was free to ensure all the data has been sent.

The *usbPost* function was used as a Power-On Self Test (POST) for the USB device. As implemented, it used *usbPrint* to send the text message, "GPS Road Roughness", and version information. The intent was for HyperTerminal to be running on the PC, where this message would be displayed.

The *usbPrint* function was used to send characters to the PC through the USB connection. A string was passed into this function which was copied into the transmit buffer. No checks were done to see if this would overflow the transmit buffer. Such functionality was up to the main application programmer. To initialize transfer, the first character was put in TXBUF1, which caused the USB interrupt to trigger. In the interrupt, through successive calls to *usbSend*, the characters were iterated through, sending the entire string.

The *usbPrintMenu* function was used to print a pre-defined menu over the USB connection. The intent was to use HyperTerminal to see the menu and enter in commands. The menu printed is as below:

```
USB Menu
---------
  c clear memory
  d download from memory
  h print this help menu again
```

The main application program handled the processing of data received, corresponding to this menu.

The *usbDump* function was used to send the entire contents of the SFM device over the USB connection. The intent was to use HyperTerminal on the PC with logging in order to store the data read from the SFM device. The *usbDump* function iteratively went through the entire SFM memory, filling up the USB transmit buffer and sending that data along. The function assumed the global interrupt flag (GIE) was set before being called.

5.3 Operating Modes

The system was designed to run in one of a number of modes. The general structure of the datalog and download modes was very similar, as shown in Figure 64. Each mode began with an initialization routine. Then, the loop function for the mode was entered and repeated. The loop would run endlessly unless the system detected that a new mode had been selected. When this happened, the mode terminated

by running a kill function and returned to the main program event loop.

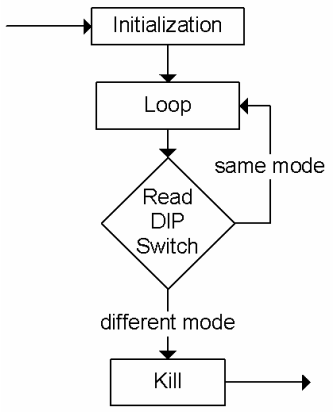


Figure 64: Operation mode general structure

Three modes are implemented, as follows:

- Datalog Mode
- Delete Mode
- Download Mode

5.3.1 Datalog Mode

The Datalog Mode was used for collecting data. In this mode, the GPS and accelerometer were read from, and data was stored in the serial flash memory (SFM). The USB was unused. The LCD was used to provide some pertinent runtime information.

The *datalogMode* function was called to start Datalog Mode. In this function, *datalogInit* was first called to initialize needed resources for this mode. Then, a while loop kept running the *datalogLoop* function. At the start of each loop, it checked the DIP switch with *dipswRead* to make sure the mode was still set for datalogging. If the mode was changed, the loop exited and *datalogKill* was called. Control was then transferred back to the main program.

The *datalogInit* function started by disabling interrupts. This was to quell spurious interrupts which could interfere with device and resource initialization. The GPS was first configured for UART0 with a call to *gpsInit*. Then, the SFM was configured for SPI1 with a call to *sfmInit1*. This was followed by calling *sfmPost* and seeing the result of a power-on self test for the SFM. If the result was not successful, an error message was displayed on the LCD, and the system was halted with a call to exit. This required a reset. When *sfmPost* was successful, the accelerometer was next initialized with a call to *axlInit*. The function ended by clearing the LCD.

The *datalogLoop* function began by starting *Timer A* with a 2.5 second hardware delay, using *hwDelay*. The GPS device's interrupts were then enabled with *gpsEnable*, and the accelerometer structure values were reset, by calling *axlReset*. While waiting for the timer to be finished, the accelerometer was continuously sampled by calls to *axlRead*. The result was that the accelerometer structure would have the maximum and minimum values read for the time period of roughly 2.5 seconds, between successive GPS reads.

The GPS device's interrupts were then disabled after the timer was done with a call to

gpsDisable, and the GPS receive buffer's length was checked to determine if any data has been received. If so, the string was parsed to find out what type of NMEA string had been received. In the case of a "GPGGA" string, the string was parsed to find the UTC time, latitude, north/south indicator, longitude, east/west indicator and number of satellites. The number of satellites was printed to the LCD, and if at least four satellites were present, the data was recorded. Recorded data included UTC time, latitude, north/south indicator, longitude, east/west indicator, and accelerometer maximum and minimum values for the X-axis, Y-axis, and Z-axis. The location in the SFM was then updated.

The *datalogKill* function freed up the buffers for the *dusart0* and *dusart1* functions through pointer nullification.

5.3.2 Delete Mode

The Delete Mode was used to clear the entire contents of the serial flash memory (SFM). This "mode" was an exception to the general structure shown in Figure 64. In this mode, the SFM was just initialized and a power on self test was performed on it.

If successful, there was a count down visible on the LCD from 5 to 0, with a one second pause between each number. When 0 was reached, the entire memory was erased with one bulk erase instruction passed to the Flash function. The microcontroller then intentionally hung until a reset was performed to ensure that this function was not accidentally set with the DIP switch and continuously run, which would erase the memory over and over again. This would be possibly damaging for a flash memory structure, since it has a limited number of writes.

If the power on self test was not successful, then the LCD printed an error message that the memory may be bad and hangs, waiting for a reset. Again, this was intentional, to avoid selecting this mode by accident.

5.3.3 Download Mode

The Download Mode was used to transfer data from the embedded system to a PC. In this mode, the serial flash memory (SFM) was read from, and data was sent over the USB connection via the USB-UART bridge. The GPS and accelerometer were unused. The LCD was used to provide some pertinent runtime information.

The *downloadMode* function was called to start Download Mode. In this function, *downloadInit* was first called to initialize needed resources for this mode. Then, a while loop kept running the *downloadLoop* function. At the start of each loop, it checked the DIP switch with *dipswRead* to make sure the mode was still set for downloading. If the mode was changed, the loop exited and *downloadKill* was called. Control was then passed back to the main program.

The *downloadInit* function started by disabling interrupts. This was to quell spurious interrupts which could interfere with device and resource initialization. The USB-UART bridge was first configured for UART1 communication. The power-on self-test for the USB was then run, followed by printing a menu over the USB. The intent was to connect to the embedded system via HyperTerminal, where this menu would be displayed, and commands could be entered. Then, the SFM was configured for SPI0 with a call to *sfmInit0*. This was followed by calling *sfmPost* and seeing the result of a power-on self test for the SFM. If the result was not successful, an error message was displayed on the LCD, and the system was halted with a call to exit, which required a reset. When *sfmPost* was successful, the position to start reading from, *sfmreadpos*, was set, and initialization was finished.

The *downloadLoop* function began by setting up *Timer A* for a 2.5 second hardware delay, through a call to *hwDelay*. The USB device's interrupts were enabled with a call to *usbEnable*, and the global interrupt enable (GIE) flag was set with a call to _EINT. A loop waited for either a USB command to be entered or for the timer to time out. Once this happened, the USB receive buffer was looked at to see if any data was received. If so, it was parsed to figure out which command was given. Three commands were implemented, as follows:

- 'c': Clear memory device
- 'd': Download entire memory device
- 'h': Print menu again

If the 'c' command was entered, the entire SFM was cleared by calling *sfmErase*. When the 'd' command was entered, the function *usbDump* was called, to get the entire contents of the SFM and print them out over the USB connection. The intent was to have HyperTerminal running on the PC, and to use the logging feature to record all of the data. When the 'h' command was entered, the function, *usbPrintMenu*, was called, in order to display the menu again.

The *downloadKill* function freed up the buffers for the *dusart0* and *dusart1* functions through pointer nullification.

5.4 MATLAB Code to Create .kml File

The mapping program chosen for this project was Google Earth. This section discusses the MATLAB code that creates a .kml file for Google Earth. When this program was run, four statements were displayed. The first told the user to enter an input file. This refers to the ASCII text file that was created in HyperTerminal. The file was read into MATLAB using the *fread* function. The second statement asked to assign a name for the new .kml file. The third and fourth statements asked the user to input a test name and description. Once the file was read into MATLAB, the array containing the inputted data was parsed using *strtok*. A while loop was used to separate each section of the string into different variables. For example, all the latitude readings were stored into the *lat* variable. Once separated, the longitude, latitude, and time was converted into a format that would be recognized by Google Earth. Also, the output magnitude from the accelerometer was calculated by taking the square roots of the squares of each axis output (both maximum and minimum). The maximum value between the two was chosen.

Next, the .kml file was created. The *diary* function was used to create a new file to save the program output. The .kml file was created using all disp commands to print out each line. The standard .kml file format was used (as shown on Google Earth's website). Also, the different pin colors were chosen by 'if' statements. The program examined the accelerometer magnitudes and decided which color to assign. When the program was complete, the new .kml file appeared in the current MATLAB directory.

5.5 Summary

In this chapter, all the software written for the device was explained. The data structures created for the project were described, along with the libraries and APIs that were designed. Also, the various modes of operation were detailed. They were Datalog, Delete, and Download. In addition, the code written for Google Earth was explained. This code converted the stored data into a .kml to display in Google Earth.

6 SYSTEM INTEGRATION AND TESTING

This chapter will present the integration of the individual modules and final testing. First, the modules were combined on a soldered protoboard and tested to prove functionality. Once the prototype was working, the PCB was build and tested. In addition, a run time analysis of the modules will be derived, and a Google Earth test will be presented.

6.1 Soldered Prototype

This section will discuss the building and testing of the soldered prototype. The prototype was built on a solderable protoboard. Sockets were used for the USB-UART bridge, LCD, and memory. A different USB module was used at this time from Sparkfun, as seen in
Figure 65. The USB module used was the CP2102, which was similar to the CP2102-microUSB module that was eventually used. A socket was constructed out of other sockets, cut in half, and aligned in a square, for the microcontroller breakout board. This board can be seen in
Figure 66, also from Sparkfun.

Figure 65: Sparkfun CP2102 Module
(Breakout Board, page 1)

Figure 66: Sparkfun MSP430F169 Breakout Board (Header Board, page 1)

The GPS was initially attached via a socket; however, this did not work very well. The first revision of the PCB was cut up, and the section for the GPS unit was used and connected to the socket on the protoboard. This can be seen in Figure 67.

Figure 67: GPS Connector for Prototype

The prototype was built up, module-by-module, iteratively testing each part. The microcontroller was first tested. Then, the LCD API was written and tested. During testing, a logic probe was used to make sure each line got the right signals, at the right time. This also required writing software delay code.

Next, general interrupt-driven UART code was written and tested. This was tested by using code for the USB, which was very straightforward. Code for the GPS was written and tested to make sure that strings could be read from the GPS and parsed. A sample string was used to test against the parsing code. The parsing code broke the string up into string type, UTC time, latitude, north/south identifier, longitude, east/west identifier and number of satellites.

Next, SPI code was written and tested. It was tested by hooking up a function generator to the input of one SPI port and setting up the output of the port to echo it. By looking at the function generator output and SPI port output on an oscilloscope, it was shown that the synchronous serial connection worked.

Afterwards, a full SFM API was built up and tested with the SPI code. This required reading documentation on the STMicroelectronic's libraries and modifying them to work with the MSP430. Then, code was written, tested, and debugged to get the device and manufacturer ID to return successfully. Upon this, code was written to store a specific string in the memory and then read back that location. This was tested in the debugger and worked. Memory erasing was then also tested, after having written data to the SFM. This was tested and worked for both SPI0 and SPI1 ports. Below are images of the completed prototype.

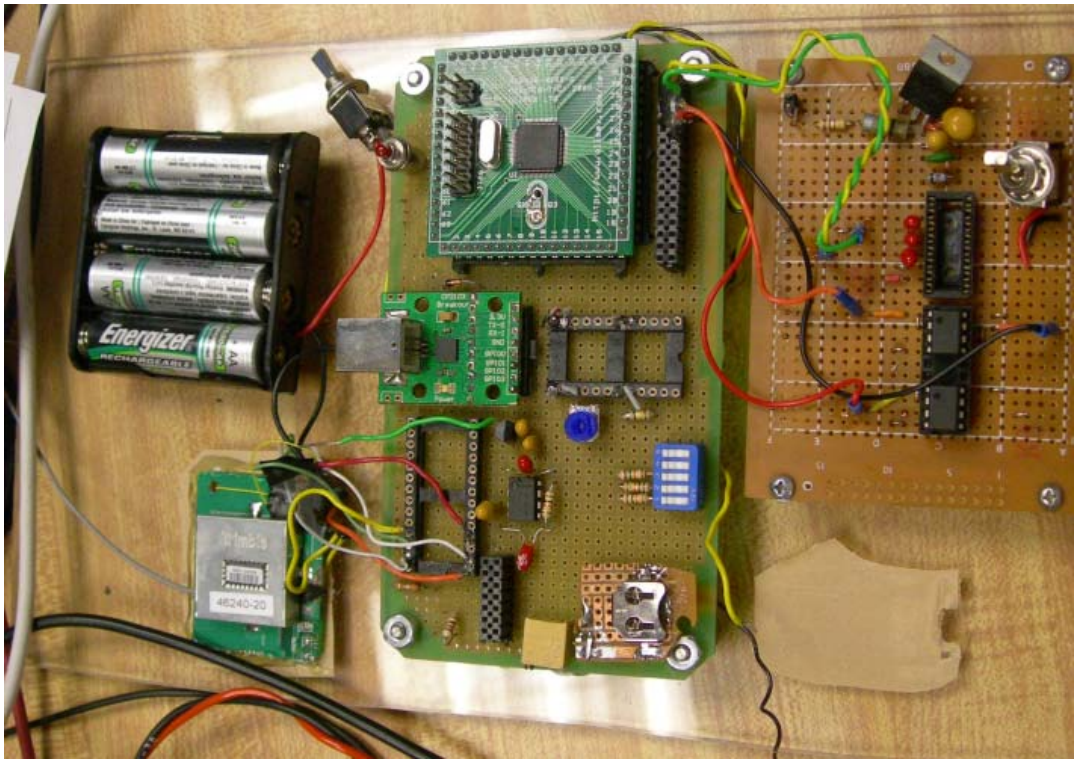


Figure 68: Image of Completed Prototype (Top View)

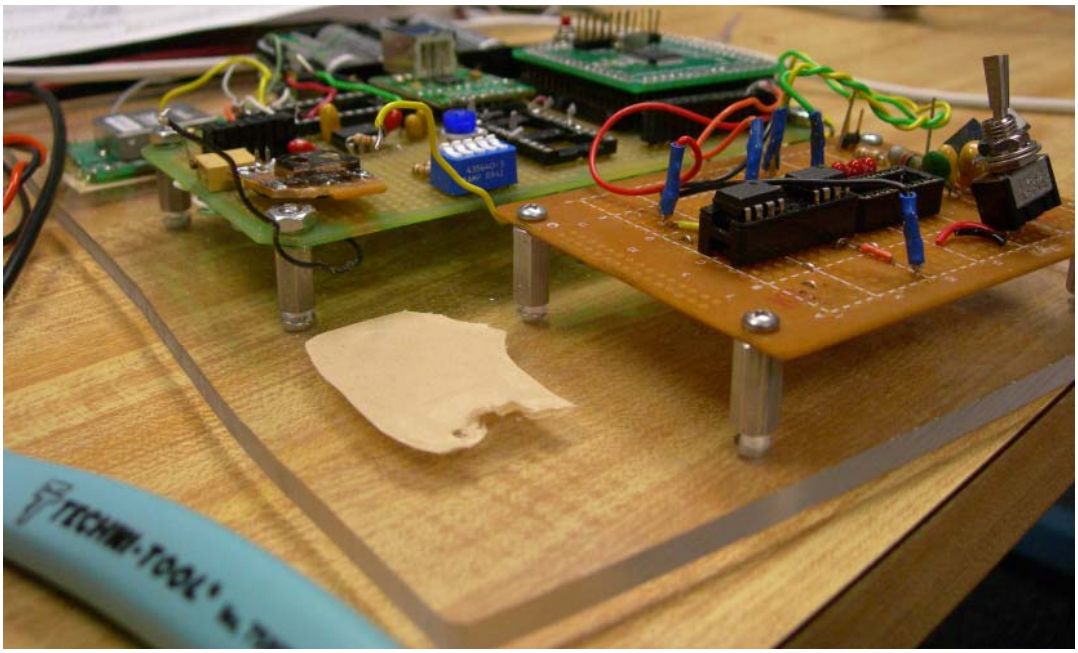


Figure 69: Image of Completed Prototype (Side View)

6.3 PCB

This section will discuss the building and testing of the PCB. The PCB was designed after having built the above working prototype. Some design decisions were made:

- 2-layer board
- 1 oz copper
- Minimize board space
- Use thick traces for power paths
- Separate ground planes for analog, digital, 5V and power
- Use 0805 components when possible
- Use mostly surface mount components

A 2-layer board is standard. We decided to use only two layers to minimize cost. For the same budgetary reason, 1 oz copper was to be used. Additionally, the board space was to be minimized. The first revision of the PCB had components on both sides in order to minimize space. Unfortunately, this was overly ambitious, and did not work.

Power paths require thick traces. Since a standard trace width is 12 mils, a power path would have 50, 60 or up to 75 mils, where possible. This included input voltage, output voltage rails, and ground rails, where needed (no direct path to ground plane). This figure was chosen because it would work with 1 oz copper for fairly high currents, much higher than the currents on the board. However, the traces were thickened to reduce trace resistance. Additionally, they would not heat up as much with the higher currents, especially in the battery charger circuitry where up to 1 or even 2A could be used. Therefore, the thicker traces help with thermal relief. A copper pour heatsink was also attached to the pre-regulator's tab in order to provide thermal relief. Also, there was room for a heatsink to be screwed on to the power BJT in the battery charger circuitry.

To minimize ground corruption noise, separate ground planes were used with analog, digital, 5V, and power ground. The ground planes connected to each other to provide a common return, but only through a thin, 12 mil trace. This helped provide some ground isolation. The ground planes were only on the bottom side of the board for the second board revision, due to issues with putting grounds on both sides that was found on the first revision.

Initially, there were planes on both sides of the board. This was overly ambitious, and there was not enough thermal clearance in some areas, causing difficult to find and fix plane-to-plane and plane-to-pin shorts. This made the first revision of the board unusable.

It was decided to use a design which focused on using surface mount components. The reason for this was that surface mount components are typically much smaller than through-hole components. Additionally, it allowed for more room for routing, since traces could run on the layer below surface mount components, whereas with through-hole components, there would be many holes on the bottom layer that the routing would have to avoid. Also, surface mount components, with shorter and smaller leads, could have less parasitic elements, such as decreased pin inductance, than through-hole components.

There were many surface mount package sizes. It was decided to use 0805 packages, where available, for resistors and capacitors. The needed power rating for each resistor was checked, and, where

needed, 1206 or larger resistors were used. A 2W wire-wound resistor was used for RQ1 in the battery charger circuitry. In reality, since this 2W was calculated as a worst case and was not constant, we could have relied upon the pulse power rating and used a smaller resistor. The wire-wound was chosen for its smaller size. A different resistor might be a better choice, to minimize the inductance of this resistor; wire-wounds are known for having a high amount of parasitic inductance.

Additionally, the needed voltage rating of each capacitor was checked. In some cases, 1206 or larger capacitors were required. The power supply input capacitor, rated for 33µF, for example, required a larger package. Tantalum capacitors were used for capacitor values rated for 1µF and greater, due to their excellent invariability with time and temperature. Ceramic capacitors were used for small capacitor values, such as 0.1µF decoupling capacitors, due to the very low equivalent series resistance (ESR), which aids in blocking higher frequency noise and has excellent thermal characteristics.

Decoupling capacitors were placed as close as possible between each module and the input pins. Capacitors in general were placed as close as possible to the pins they were connected to. This often reduces the effects of trace resistance and inductance, and provides a more stable voltage. Often, a 0.1µF ceramic was placed in parallel with a larger tantalum capacitor to help block out high frequency noise.

The program, PADS (version 2005), by Mentor Graphics, was used for the PCB layout and routing. The routing strategy was configured to route the power nets first, then ground, and then the rest. This was done to reduce the resistance, especially with connections to power and ground, in order to increase efficiency by reducing power losses through trace resistance. Additionally, smaller traces mean less trace inductance, which means less parasitic effects, especially on analog circuitry. The decoupling capacitors, right next to the part they were decoupling, also helped. The Auto-Router was configured to route in this fashion and provide a high level of routing optimization, while maintaining enough clearance from pin-to-trace, trace-to-trace, and trace-to-via. This optimization included trying to reduce the number of vias, not by increasing the length of traces. Trace length minimization first, and then via minimization was the goal.

The resulting second revision PCB worked well, but still has not been completely debugged. The working features are outlined in the following list.

- The pre-regulator system works completely.
- There is clean power regulation of 3.3V from the LDO and 5V from the charge pump, whether the USB, wall, car or four NiMH batteries are used.
- The power source switching system has been tested and works. There are no unwanted voltages feeding back in; there is a clean “break before make” of sources.
- The battery charger seems to draw current through the wall/car adapter and put a voltage onto the batteries when they are present, indicating that charging should work.
- The microcontroller can be programmed and debugged, in-circuit.
- The hardware timer of the microcontroller works, which means the 32kHz crystal works.
- The reset button and DIP switch works completely.
- The LCD module works completely.
- The USB module works completely.
- The accelerometer module works completely.
- The memory can be identified, read, and erased from either SPI0 or SPI1.
- Strings can be read and parsed from the GPS.

There are a number of issues that have not been debugged, which are outlined below:

- The GPS does cannot find satellites to “lock on” to.
- The writing functionality of the memory has not been tested.
- The Download Mode code only works 100% when the Datalog Mode code is commented out. This likely indicates that the code is running close to or over the 60KB limit of the MSP430.
- The battery charger circuitry has not been extensively tested.

Below are images of the second PCB revision. The first image, Figure 70, shows the PCB layout. The second image, Figure 71, shows the assembled PCB inside the enclosure. The third image, Figure 72, shows the test setup for the PCB.

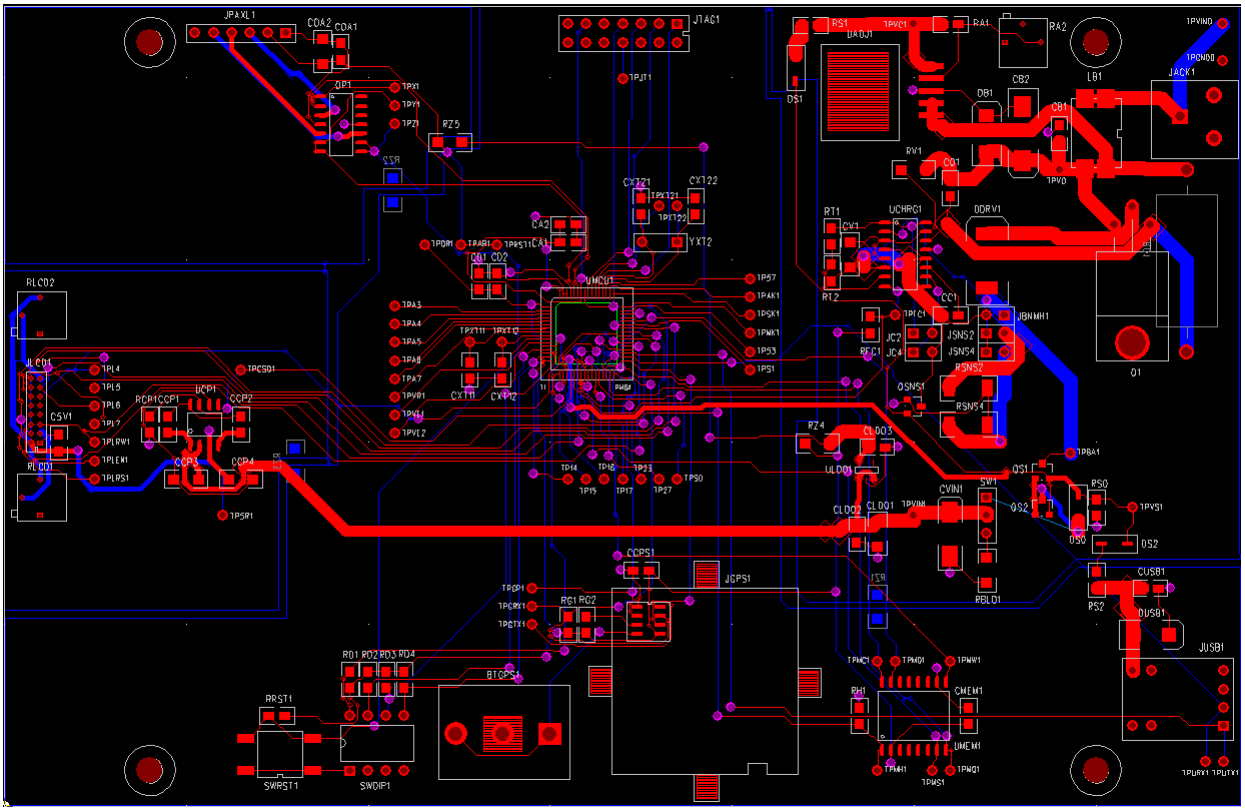


Figure 70: Second PCB Revision

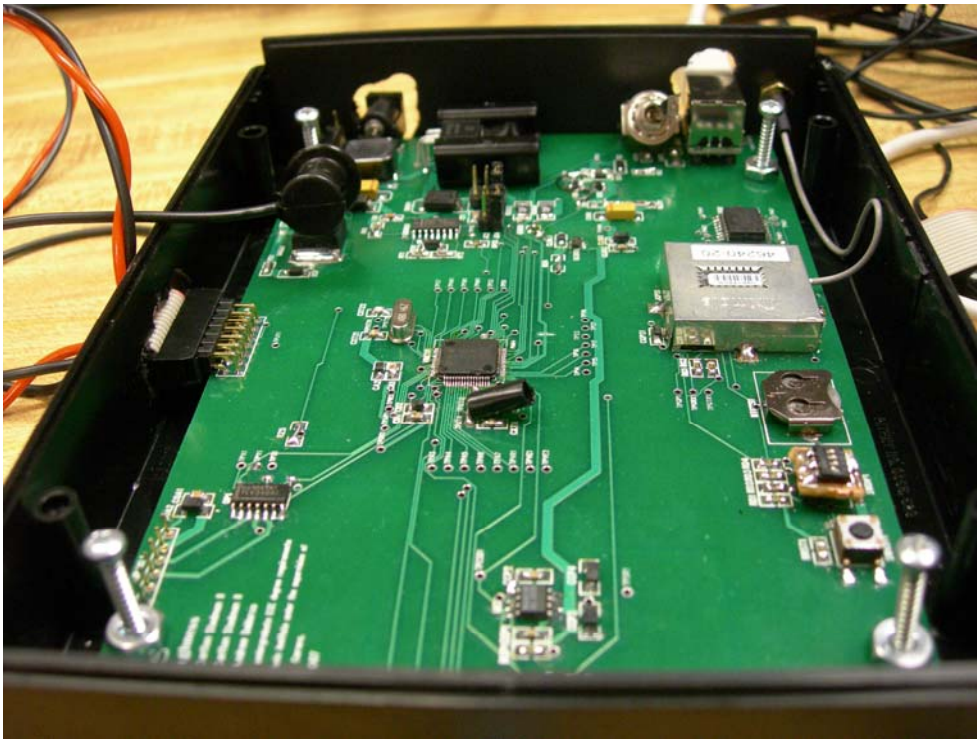


Figure 71: Assembled PCB inside Enclosure

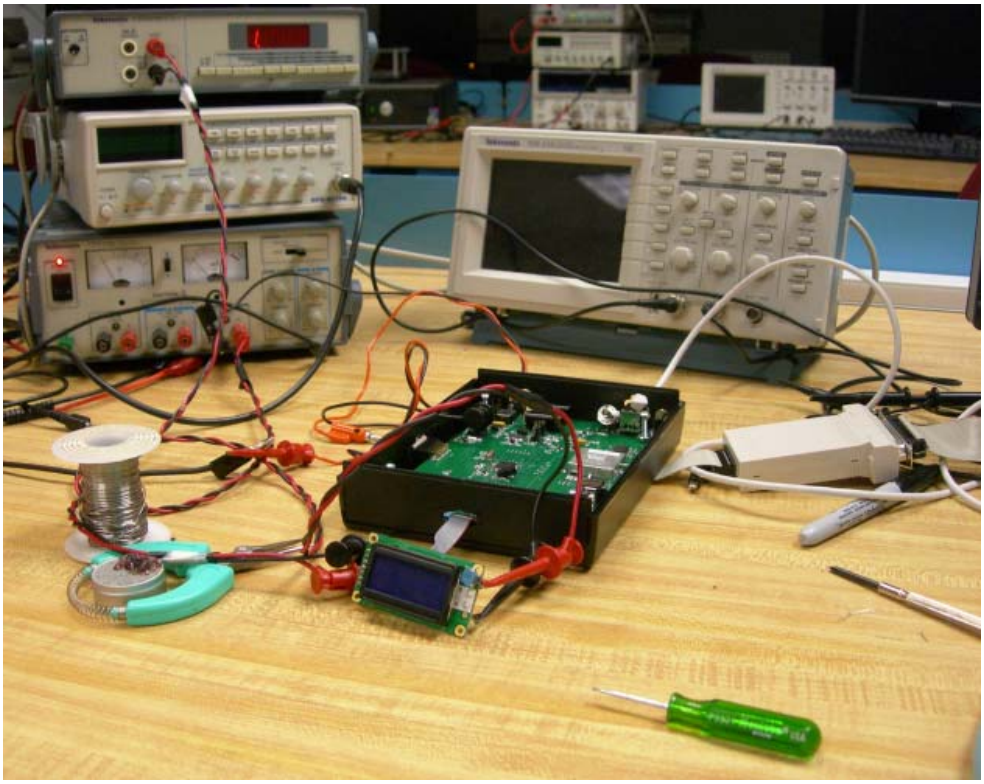


Figure 72: Test Setup for PCB

6.3 Enclosure

The enclosure chosen was the LH57-130 from PacTec. The dimensions were 7.2 in. x 5.5 in. 1.5 in. The second PCB revision was designed to specifically fit the enclosure. Some modifications were made to allow connections from the PCB to the outside of the enclosure. Holes were drilled in the sides to accommodate the GPS antenna, JTAG connector, LCD module, USB connector, and the power adapter. This enclosure can be seen in Figure 72.

6.4 Run Time Analysis

This section examined various run times for the device. This was accomplished by finding the currents used by all the components in the design. The components that were analyzed were the battery charger, voltage regulator, accelerometer, LCD, microcontroller, GPS, memory, and the USB to UART bridge. Also, the accelerometer block contained three op-amps.

Since the exact usage of the device cannot be predicted, several scenarios were examined. They predicted the worst case run times for the device. The first scenario was if all the components were running in normal mode and were never turned off. These current values were the maximum amount the components could draw. Table 17 shows the maximum currents for all the components when the device was in use. All the data was taken from the component datasheets.

Module	Current
Microprocessor	2.64mA
LCD[1]	2.28mA
Voltage Regulator	8µA
Charge Pump	150µA
Battery Charger	5µA
Accelerometer	0.32mA
Op-Amp	0.95µA
GPS[2]	27.02mA
Memory[3]	101.2µA
USB to UART	330µA

Table 17: Currents for Normal Mode

Notes:

[1] This current value for the LCD was a combination of the module running with and without the backlight. Without the backlight, the LCD used 1.2mA. With the backlight, the LCD used 130mA. Since the backlight drew too much current for our application, it was assumed that the backlight would only be on for 30 seconds every hour. To find the current for this assumption, the following steps were taken:

· Percent of time backlight is on = 30sec/3600sec = 0.0083
· Current used for backlight = 0.0083*130mA = 1.08mA
· Total current for LCD = 1.2mA+1.08mA = 1.28mA

[2] The current for the GPS was a combination of current drawn (27mA) and current for the battery backup (20µA).

[3] This current for the memory took into account both standby time and write time. The read time current was supplied by the USB. The standby current was 100µA and the current for a write was 15mA. To determine the current for the memory, the following steps were taken:

· Since data was stored in a buffer, a write to memory only occurred when the buffer was full. This was approximately every 18 seconds. A write lasted approximately 1.4ms.
· Number of writes per hour = 3600sec/18sec = 200 writes
· Length of writes = 200*1.4ms = 0.28sec
· Percent of time for writing = 0.28sec/3600sec = $7.78*10^{-5}$
· Current used for writing = $7.78*10^{-5}$ * 15mA = 1.167μA
· Percent of time in standby = (3600sec-0.28sec)/3600sec = 0.9999
· Current used in standby = 0.9999*100μA = 99.99μA
· Total current for memory = 1.167μA+99.99μA = 101.2μA

The total current drawn by all the components was 32.855mA. This current determined how long the unit could run on 4 "AA" batteries (2500mAh). Since the battery capacities were not perfect and the total voltage could not drop below 3.45V, it was assumed that only 2000mAh was drawn from the batteries. The total run time for this scenario was 60.87 hours (2300mAh/38.603mA). This was approximately two and a half days.

Another scenario was analyzed in order to extend the run time for the roughness detector. This time, all the components were fully running for only 10 hours a day (approximately one work day). For the other 14 hours, the components were in standby mode to conserve power. To determine the power for this scenario, the total current for standby mode needed to be calculated in addition to normal running mode. Table 18 shows the minimum currents for the components when the device was in standby mode.

Module	Current
Microprocessor	1.1μA
LCD[1]	2.28mA
Voltage Regulator	8μA
Charge Pump	150μA
Battery Charger	5μA
Accelerometer[2]	≈ 0
Op-Amp[2]	≈ 0
GPS[3]	20μA
Memory	100μA
USB to UART	330μA

Table 18: Currents for Standby Mode

Notes:
[1] The current for the LCD was calculated the same way as in the previous scenario.

[2] The currents drawn by the accelerometer and op-amp were assumed to be negligible.

[3] The GPS current only took into account battery backup.

The total current drawn by all the components in standby mode was 2.894mA. The current for normal mode was previously found to be 32.855mA. To determine the run time for this scenario, two situations were analyzed. The first situation was if the operator forgot to turn the unit off at the end of the day. If this occurred, then the unit was automatically turned off after 30 minutes of inactivity. The steps to determine the total run time were as follows:

· *Current used in normal mode = [(10hrs+0.5hrs)/24hrs]*32.855mA = 14.374mA*
· *Current used in standby mode = [(14hrs-0.5hrs)/24hrs]*2.894mA = 1.628mA*
· *Total current = 14.374mA+1.628mA = 16.002mA*
· *Run Time = 2000mAh/16.002 = 124.98hrs*

The total run time for the first situation was 124.98 hours. This was approximately 5.21 days. The second situation for this scenario was if the operator did turn the power off at the end of the day. The steps to determine the total run time were as follows:

· *Current used in normal mode = (10hrs/24hrs)*32.855mA = 13.690mA*
· *Current used in standby mode = (14hrs/24hrs)*2.894mA = 1.688mA*
· *Total current = 13.690mA+1.688mA = 15.378mA*
· *Run Time = 2000mAh/15.378 = 130.06hrs*

The total run time for the second situation was 130.06 hours. This is approximately 5.42 days. The run times for both situations were fairly close. It did not make a big difference if the detector was not turned off at the end of the day.

These two scenarios give a sense of how long the device can run on 4 "AA" batteries. If the device was always left on and never set to standby mode, it would only last approximately 2.5 days. However, if standby mode was used when the vehicle was not in use, the run time would be doubled. This would be the preferred method of operation. The device would be able to run for the entire work week without needing to be recharged.

6.4 Google Earth Test

Since our team did not perform a road test, we were not able to produce accurate results. However, doctored data was used to simulate a road test. We were able to create dummy data by using GPS values found by the previous WPI road roughness project. Since we did perform a road test with just the accelerometer in a car, we had an idea of what the accelerometer values should be. To get this accelerometer data, we used the PicoScope oscilloscope to record the accelerometer output from the soldered test board in Figure 30. The output was between 0.3 – 0.45 V. This corresponded to a g value around 1 (0.330 V is one g). A sample string is shown below.

```
152540,4215.5430,N,07149.3225,W,2195,2016,2096,1920,2225,2236;
```

The fields are as follows: time, latitude, north or south hemisphere, longitude, east or west hemisphere, x-axis maximum, x-axis minimum, y-axis maximum, y-axis minimum, z-axis maximum, and z-axis minimum. The accelerometer values are expressed in 12-bit values from the ADC.

With the input text file created, we ran the MATLAB code to create the .kml file. The data was saved to the specified file. Once MATLAB was complete, the .kml file could be opened in Google Earth. The output can be seen below.

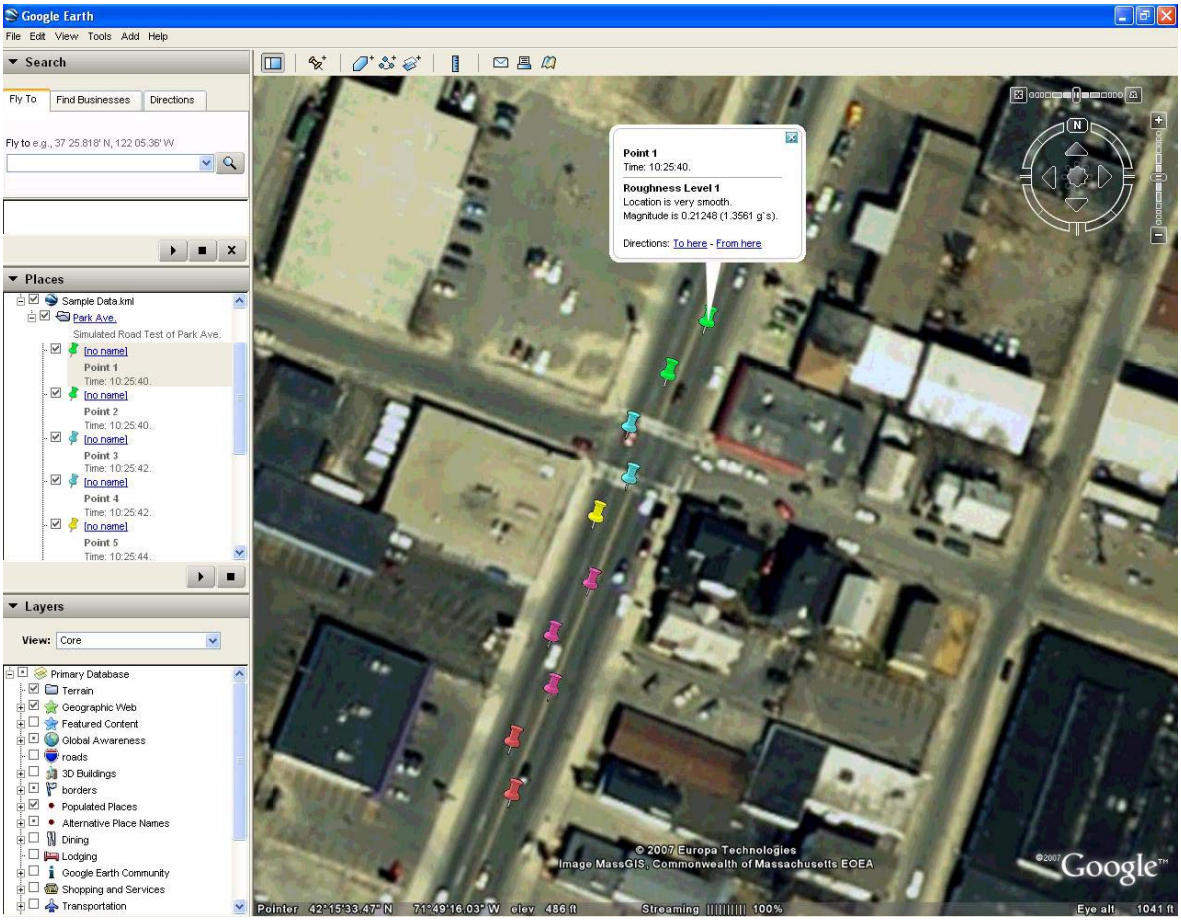


Figure 73: Google Earth Sample

The road roughness was classified into five categories: very smooth, smooth, average, rough, very rough. The magnitude of 0.3 – 0.45V was divided into five sections to obtain the levels. The higher the number, the rougher the road was. Each roughness level has an associated colored push pin. For a very smooth point, a green pin was assigned. For a smooth point, a light blue pin was assigned. For an average point, a yellow pin was assigned. For a rough point, a pink pin was assigned. For a very rough point, a red pin was assigned.

Also, when each push pin was selected, a balloon with relevant information was displayed. The information was the point number, time, roughness level, and magnitude. The magnitude was expressed in both volts and g's.

6.5 Summary

This chapter discussed the building of the prototype and PCB design. The building process was documented step by step. Even though the prototype was built up, no field test was performed. Due to the problems with the first PCB revision, there was not enough time to test the second PCB on the road. However, the mapping program was still tested using dummy data. With GPS data from a previous project and accelerometer data from an initial road test, a sample text file was created to test Google Earth. The output was as expected. The only task needed to complete this project was to drive around with the second PCB and see if the rough areas show up in Google Earth.

7 CONCLUSIONS AND RECOMMENDATIONS

This chapter summarizes the completed project. The overall goal of the project was to design a small, low power, and inexpensive system that would record the roughness of the road and coordinate it with a GPS location. This data would then be uploaded to a computer and overlaid on a map, indicating the magnitude of roughness. The projected needed to be simple and easy so a city could monitor the condition of their roads.

7.1 Summary of Project Design

To accomplish our project, we followed a process that led us from the project goal to the final PCB testing. This process can be seen in Figure 14. The overall design can be broken down into various modules.

The first module was power. This module provided power to the device with three options. The main source of power was four "AA" NiMH batteries that were charged every week. An alternate source of power was the car adapter. When the car adapter was plugged in, the device ran off the car, while charging the batteries. The third power source was the USB connection, which provided 5 V when attached.

The next module was the accelerometer. This module detected vibrations for the road and outputted a voltage proportional to the vibration. This information was used to determine the roughness of the road. The output voltage from all three axes were used to determine the overall magnitude.

The third module was the GPS. This module was responsible for determining the location of the data points collected from the accelerometer. Standard NMEA strings were outputted from the GPS. The information in these strings included latitude, longitude, and time. These readings were coordinated with the accelerometer readings.

The fourth module was the microcontroller. This module controlled the device and provided on board processing. The microcontroller inputted the accelerometer readings and corresponding GPS locations, and stored them in the memory. The on board processing included buffering the accelerometer readings and choosing the worst case for each GPS reading.

The next module was the memory. This module was responsible for storing the accelerometer and GPS data collected while driving. At the end of the week, the data was uploaded to a computer and the memory was cleared.

The sixth module was the USB interface. This module dealt with transferring data between the device and the computer. When all the data had been collected, the USB was connected to the device, and the USB mode was selected. This started the data upload to the computer via HyperTerminal. The data was stored in an ASCII text file, which was used to import the data into Google Earth.

The last module was the LCD. This module served as the visual output of the device. It aided in debugging the system and outputted various information, such as the number of satellites, mode of operation, when data is recorded or uploaded, and power mode.

Once each module was tested, they were interfaced with each other and tested as a whole. Software was also written for testing the prototype. The final system was tested on the road to verify the functionality.

7.2 Future Recommendations

There are several improvements that can be made to this project. Once improvement would be to use multiple accelerometers located in the vehicle. Since our project only uses one accelerometer, the output magnitude is only taking into account the vibration from one part of the car. For a better output reading, an accelerometer at each wheel would provide better information for determining the vibration magnitude.

Another improvement would be to have a real time output to a PDA or laptop. With this addition, the roughness can be plotted on the map as the vehicle is driving. This would aid in debugging. If a rough section of road was driven over, the user could instantly confirm if the device was working properly.

A third improvement would be to upload the data to a USB flash drive. This would simplify the project even more. If a flash drive was used, then a USB cable and computer would not be needed to upload the data from the memory. With the press of a button, the data would be saved into a file on the flash drive.

7.3 Conclusions

We were able to successfully accomplish our goal for the project. Most of the specifications described in Chapter 3 were met. The device was small (7 in. x 5.5 in. x 1.5 in.). This was lower than the specification of 7 in. x 5.5 in. x 1.5 in. It was also inexpensive (approximately $150). The predicted price was $250. The low power specification was not met. The device drew 32.9mA at 3.3V, which corresponds to a power of 108.57mW. Our goal was to have 100mW. The device was all in one enclosure and was accurate to at least 15 m. The memory could store more than one week's worth of data, which surpassed the specification. However, the device was not too reliable. There were bit errors in the GPS strings that affected the MATLAB parsing. Once these errors were deleted from the text file, the .kml file was created successfully. When the second PCB is field tested, then we will know if the project was a complete success. Overall, this system could help cities locate roads in need of repairs before more damage occurs to vehicles.

8 REFERENCES

This chapter lists the references used in this report. Each reference corresponds to the appropriate citation in the main text. Also the datasheets used will be listed.

8.1 Works Cited

"Accelerometer Design and Applications." (n.d.). *Analog Devices.* Retrieved September 15, 2006, from http://www.analog.com/en/content/0,2886,764%255F800%255F122115,00.html.

"Achieving a High Level of Smoothness in Concrete Pavements Without Sacrificing Long Term Performance." (n.d.). *Federal Highway Administration*. Retrieved October 8, 2006, from http://www.fhwa.dot.gov/pavement/pccp/pubs/05068/02chapter2.cfm.

Angelini, Nicholas, Matt Gdula, Craig Shevlin, and Jose Brache. "Mapping City Potholes." (2006, April 27). *Worcester Polytechnic Institute*. Retrieved September 16, 2006, from http://www.wpi.edu/Pubs/E-project/Available/E-project-042706-141742/unrestricted/Final_Pothole_Report.pdf.

"Breakout Board for CP2102 USB to Serial." (n.d.). *Sparkfun.* Retrieved November 12, 2006, from http://www.sparkfun.com/commerce/product_info.php?products_id=198.

Budras, Joseph. "A Synopsis on the Current Equipment Used for Measuring Pavement Smoothness." (2001, August). *Federal Highway Administration*. Retrieved October 8, 2006, from http://www.fhwa.dot.gov/pavement/smoothness/rough.cfm#iii.

"CFAH0802A Color Standard LCD Modules." (n.d.). *Crystalfontz.* Retrieved November 2, 2006, from http://www.crystalfontz.com/products/0802a-color/index.html#CFAH0802AYMIJP.

"CFAH1602A Color Standard LCD Modules." (n.d.). *Crystalfontz.* Retrieved November 2, 2006, from http://www.crystalfontz.com/products/1602a/index.html#CFAH1602AYYHJP.

"CFAH1602A-YYH-JP." (n.d.). *Crystalfontz.* Retrieved November 2, 2006, from http://www.crystalfontz.com/products/1602a/CFAH1602AYYHJP.PDF.

"Crystal." (n.d.). *Pic Fun.* Retrieved April 24, 2007, from http://www.picfun.com/module/crystal.jpg.

"Differential GPS." (n.d.). *Wikipedia*. Retrieved October 10, 2006, from http://en.wikipedia.org/wiki/Differential_GPS.

"DIP Switches, Low Profile, Slide Actuator, Through Hole and Surface Mount." (2004, September). *Tyco Electronics.* Retrieved March 15, 2007, from http://ecommas.tycoelectronics.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Catalog+Page%7F1308390_0904_A3_A4%7F1104%7Fpdf%7FEnglish%7FENG_CAT_1308390_0904_A3_A4_1104.pdf.

"Global Positioning System." (n.d.). *Wikipedia*. Retrieved October 10, 2006, from http://en.wikipedia.org/wiki/Gps.

"GPS in More Detail." (n.d.). *Smithsonian National Air and Space Museum*. Retrieved October 10, 2006, from http://www.nasm.si.edu/gps/spheres.html.

"Half-Pitch DIP Switch." (2005, March). *Omron Electronic Components LLC.* Retrieved March 15, 2007, from http://oeiwcsnts1.omron.com/ocb_pdfcatal.nsf/PDFLookupByUniqueID/1488E4BE7C28FFC486256FC7005E9221/$File/D22A6H0305.pdf?OpenElement.

"Header Board for MSP430F169." (n.d.) *Sparkfun.* Retrieved September 14, 2006, from http://www.sparkfun.com/commerce/product_info.php?products_id=48.

"How GPS Receivers Work." (n.d.). *Howstuffworks.* Retrieved October 10, 2006, from http://www.howstuffworks.com/gps.htm.

"How GPS Works." (n.d.). *Trimble*. Retrieved October 10, 2006, from http://www.trimble.com/gps/howgps.shtml.

"iMEMS Accelerometers." (n.d.). *Analog Devices Inc.* Retrieved October 16, 2006, from http://www.analog.com/en/subCat/0,2879,764%255F800%255F0%255F%255F0%255F,00.html.

"Key Facts About America's Road and Bridge Conditions and Federal Funding." (2006, March). *TRIP*. Retrieved October 8, 2006, from http://www.tripnet.org/NationalFactSheetMarch2006.pdf.

Kolanko, Frank. "Linear Regulators vs. Switchers for Automotive Applications." (2006, June 12). *Automotive Design Line.* Retrieved March 26, 2006, from http://www.automotivedesignline.com/189400333;jsessionid=VC1GHESMHQSTOQSNDLQCKH0CJUNN2JVN?printableArticle=true.

"Laser Sensors Used for Road Profiling." (n.d.). *Acuity*. Retrieved October 16, 2006, from http://www.acuityresearch.com/products/ar600/common-applications-road-profiling.shtml.

"Lassen iQ Evaluation Board – RS232." (n.d.). *Sparkfun.* Retrieved October 10, 2006, from http://www.sparkfun.com/commerce/product_info.php?products_id=167#.

"Lassen iQ GPS Receiver." (2005, February). *Sparkfun*. Retrieved September 24, 2006, from http://www.sparkfun.com/datasheets/GPS/Lassen%20iQ_Reference%20Manual.pdf.

"M25P64." (2006, September). *STMicroelectronics.* Retrieved September 15, 2006, from http://www.st.com/stonline/products/literature/ds/10987.pdf.

"Micro-USB Module." (n.d.). *Dontronics.* Retrieved March 10, 2007, from http://www.dontronics-shop.com/product.php?productid=16141.

"Quartz Crystals, Crystal Clock Oscillators, Crystal Filters." (n.d.). *Component Marketing Services, Inc*. Retrieved April 24, 2007, from http://www.cms-mkt.com/images/stest.gif.

"Rough Ride Ahead: Metro Areas With the Roughest Rides and Strategies to Make our Roads Smoother." (2005, May). *TRIP*. Retrieved October 8, 2006, from http://www.tripnet.org/RoughRoadsReport052605.pdf.

"Rough Ride in the City: Metro Areas With the Roughest Rides and Strategies to Make our Roads Smoother." (2006, October). *TRIP*. Retrieved October 816, 2006, from http://www.tripnet.org/RoughRideReportOct2006.pdf.

Sayers, Michael W. and Steven M. Karamihas. "The Little Book of Profiling." (1998, September). *University of Michigan*. Retrieved October 8, 2006, from http://www.umtri.umich.edu/content/LittleBook98R.pdf.

"Tact Switches." (2004, January). *Anglia Components Ltd.* Retrieved April 24, 2007, from http://www.anglia.com/tyco/datasheets/6053.pdf.

"Trimmer Potentiometers / Rotary Position Sensors." (2007, March 20). *Murata Manufacturing Co.* Retrieved April 15, 2007, from http://www.murata.com/catalog/r50e15.pdf.

"Triple Axis Accelerometer Breakout – ADXL330." (n.d.).*Sparkfun*. Retrieved April 12, 2007, from http://www.sparkfun.com/commerce/product_info.php?products_id=692.

"What is GPS?" (n.d.). *Garmin International Inc*. Retrieved September 16, 2006, from http://www8.garmin.com/aboutGPS/.

"What is WAAS?" (n.d.). *Garmin International Inc.* Retrieved September 16, 2006, from http://www8.garmin.com/aboutGPS/waas.html.

8.2 Datasheets

Accelerometer: ADXL330
Datasheet: http://www.analog.com/UploadedFiles/Data_Sheets/ADXL330.pdf

Battery Charger: MAX712
Datasheet: http://datasheets.maxim-ic.com/en/ds/MAX712-MAX713.pdf

Charge Pump: MAX619
Datasheet: http://datasheets.maxim-ic.com/en/ds/MAX619.pdf

GPS: Lassen iQ
Datasheet: http://www.sparkfun.com/datasheets/GPS/Lassen%20iQ_Reference%20Manual.pdf

LCD: CFAH1602A-YYH-JP
Datasheet: http://www.crystalfontz.com/products/1602a/CFAH1602AYYHJP.PDF

Low-Dropout Voltage Regulator: NCV551
Datasheet: http://www.onsemi.com/pub/Collateral/NCP551-D.PDF

Memory: M25P64
Datasheet: http://www.st.com/stonline/books/pdf/docs/10987.pdf

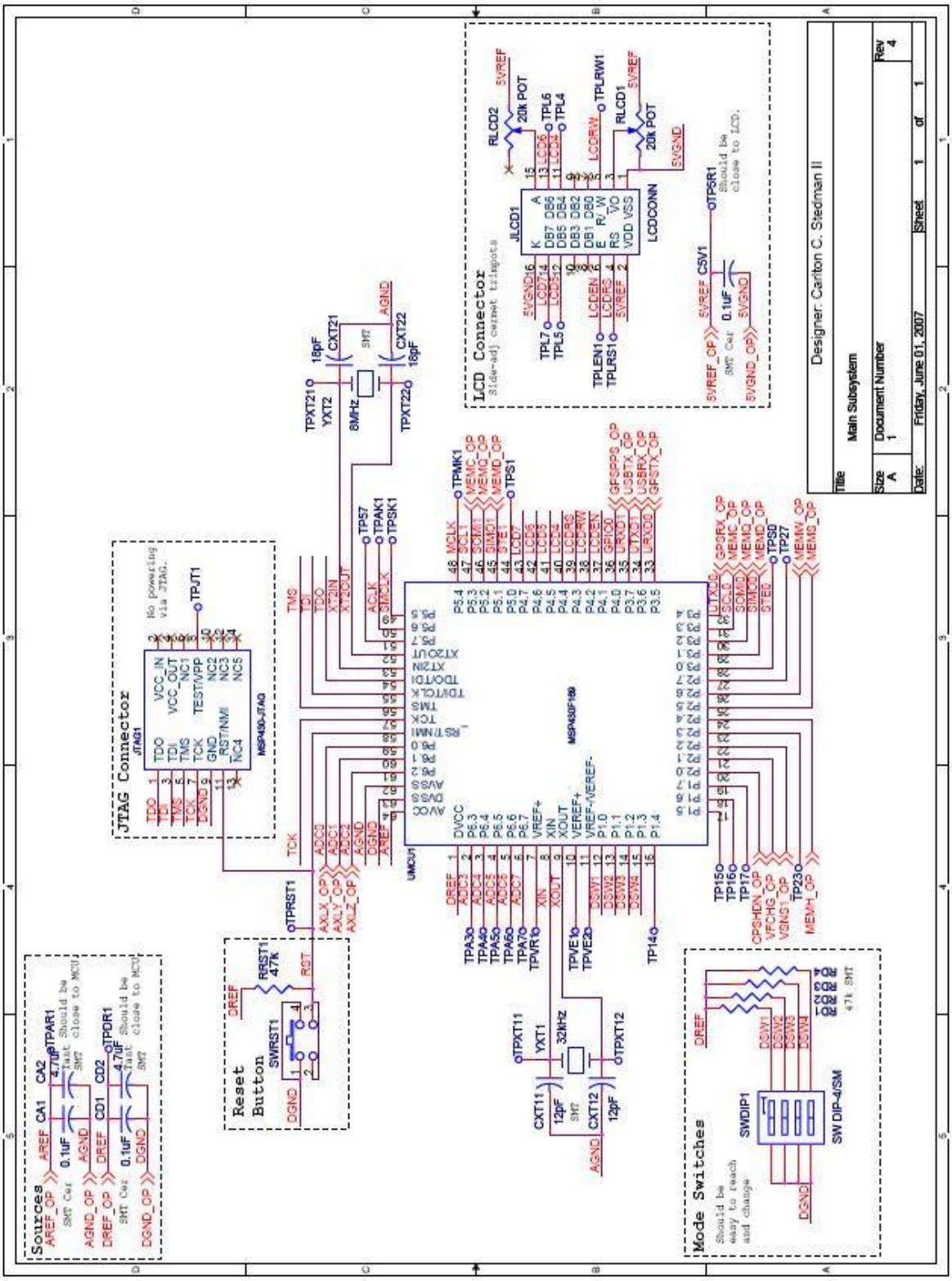
Microprocessor: MSP430F169
Datasheet: http://focus.ti.com/lit/ds/symlink/msp430f169.pdf

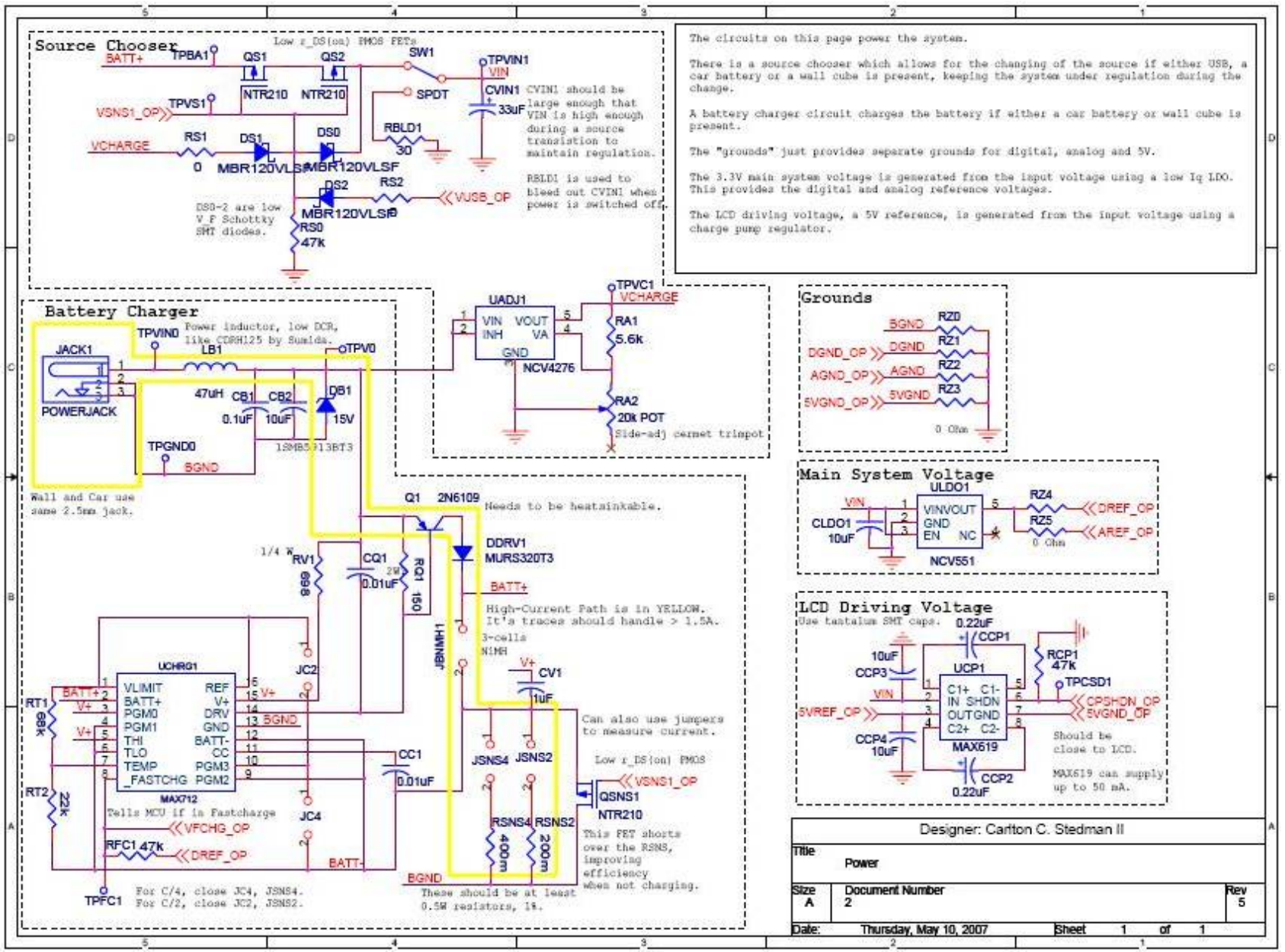
Op-Amp: TLV2404
Datasheet: http://focus.ti.com/lit/ds/symlink/tlv2404.pdf

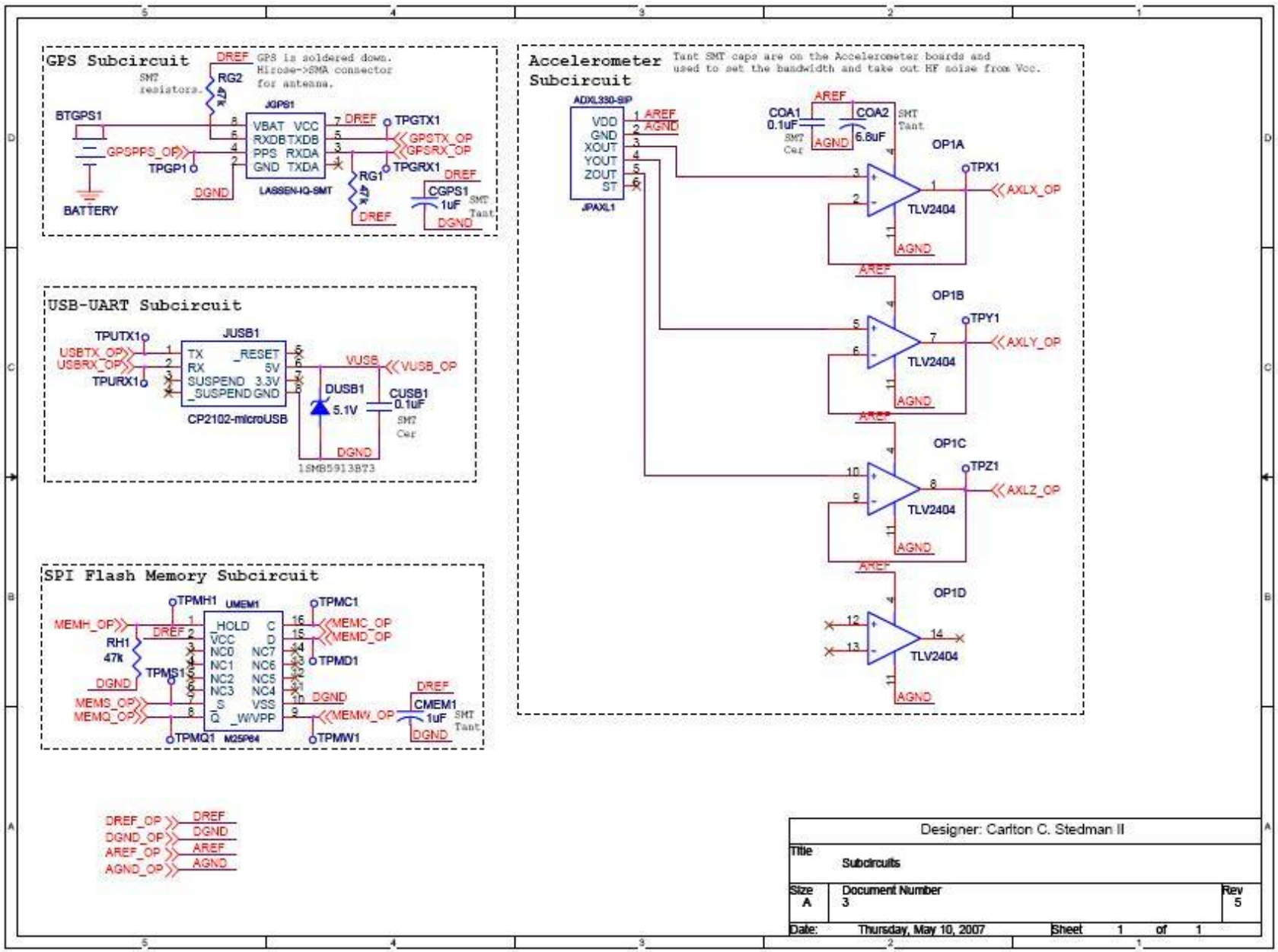
USB to UART: CP2102
Datasheet:
http://www2.silabs.com/public/documents/tpub_doc/dsheet/Microcontrollers/Interface/en/CP2102.pdf

APPENDIX A: SCHEMATICS

This appendix contains the schematics for the device.




Source Chooser
Low r_DS(on) PMOS FETs
BATT+
TPBA1
QS1
QS2
NTR210
NTR210
SW1
SPDT
TPVIN1
VIN
CVIN1
33uF
CVIN1 should be large enough that VIN is high enough during a source translation to maintain regulation.
TPVS1
VSNS1_OP
VCHARGE
RS1
0
DS1
MBR120VLSF
DS0
MBR120VLSF
RBLD1
30
RBLD1 is used to bleed out CVIN1 when power is switched off.
DS2
MBR120VLSF
RS2
0
VUSB_OP
DS0-2 are low V_F Schottky SMT diodes.
RS0
47k
The circuits on this page power the system.
There is a source chooser which allows for the changing of the source if either USB, a car battery or a wall cube is present, keeping the system under regulation during the charge.
A battery charger circuit charges the battery if either a car battery or wall cube is present.
The "grounds" just provides separate grounds for digital, analog and 5V.
The 3.3V main system voltage is generated from the input voltage using a low Iq LDO. This provides the digital and analog reference voltages.
The LCD driving voltage, a 5V reference, is generated from the input voltage using a charge pump regulator.
Battery Charger
JACK1
POWERJACK
Wall and Car use same 2.5mm jack.
TPVIN0
Power inductor, low DCR, like CDRH125 by Sumida.
LB1
47uH
CB1
0.1uF
CB2
10uF
DB1
15V
1SMB5913BT3
TPGND0
BGND
UADJ1
VIN
VOUT
INH
VA
GND
NCV4276
TPVC1
VCHARGE
RA1
5.6k
RA2
20k POT
Side-adj cermet trimpot
Grounds
BGND
RZ0
DGND_OP
DGND
RZ1
AGND_OP
AGND
RZ2
5VGND_OP
5VGND
RZ3
0 Ohm
Main System Voltage
ULDO1
VIN
CLDO1
10uF
VINVOUT
GND
EN
NC
NCV551
RZ4
DREF_OP
RZ5
0 Ohm
AREF_OP
LCD Driving Voltage
Use tantalum SMT caps.
0.22uF
CCP1
10uF
CCP3
VIN
5VREF_OP
UCP1
C1+ C1-
IN SHDN
OUTGND
C2+ C2-
MAX619
CCP4
10uF
CCP2
0.22uF
RCP1
47k
TPCSD1
CPSHDN_OP
5VGND_OP
Should be close to LCD.
MAX619 can supply up to 50 mA.
Q1
2N6109
Needs to be heatsinkable.
DDRV1
MURS320T3
BATT+
High-Current Path is in YELLOW.
It's traces should handle > 1.5A.
3-cells NiMH
JBNMH1
1/4 W
RV1
698
CQ1
2W
0.01uF
RQ1
150
V+
CV1
1uF
UCHRG1
VLIMIT
BATT+
PGM0
PGM1
THI
TLO
TEMP
_FASTCHG
REF
V+
DRV
GND
BATT-
CC
PGM3
PGM2
MAX712
Tells MCU if is Fastcharge
VFCHG_OP
RFC1 47k
DREF_OP
RT1
68k
RT2
22k
JC2
JC4
CC1
0.01uF
Can also use jumpers to measure current.
JSNS4
JSNS2
Low r_DS(on) PMOS
VSNS1_OP
QSNS1
NTR210
RSNS4
400m
RSNS2
200m
This FET shorts over the RSNS, improving efficiency when not charging.
These should be at least 0.5W resistors, 1%.
TPFC1
For C/4, close JC4, JSNS4.
For C/2, close JC2, JSNS2.
Designer: Carlton C. Stedman II
Title
Power
Size
A
Document Number
2
Rev
5
Date:
Thursday, May 10, 2007
Sheet
1
of
1


GPS Subcircuit
GPS is soldered down.
Hirose->SMA connector for antenna.
SMT resistors.
RG2
DREF
JGPS1
BTGPS1
VBAT VCC
RXDB TXDB
PPS RXDA
GND TXDA
LASSEN-IQ-SMT
TPGTX1
TPGRX1
GPSTX_OP
GPSRX_OP
GPSPPS_OP
TPGP1
DGND
RG1
CGPS1
1uF
SMT Tant
BATTERY
USB-UART Subcircuit
TPUTX1
TPURX1
JUSB1
USBTX_OP
USBRX_OP
TX
RX
SUSPEND
_SUSPEND
_RESET
5V
3.3V
GND
VUSB
VUSB_OP
DUSB1
5.1V
CUSB1
0.1uF
SMT Cer
CP2102-microUSB
1SMB5913BT3
SPI Flash Memory Subcircuit
TPMH1
UMEM1
TPMC1
MEMH_OP
MEMC_OP
MEMD_OP
RH1
47k
TPMD1
TPMS1
_HOLD
VCC
NC0
NC1
NC2
NC3
S
Q
C
D
NC7
NC6
NC5
NC4
VSS
_W/VPP
MEMS_OP
MEMQ_OP
MEMW_OP
CMEM1
1uF
SMT Tant
TPMQ1
M25P64
TPMW1
Accelerometer Subcircuit
Tant SMT caps are on the Accelerometer boards and used to set the bandwidth and take out HF noise from Vcc.
ADXL330-SIP
JPAXL1
VDD
GND
XOUT
YOUT
ZOUT
ST
AREF
AGND
COA1
0.1uF
COA2
6.8uF
SMT Cer
SMT Tant
OP1A
OP1B
OP1C
OP1D
TLV2404
TPX1
TPY1
TPZ1
AXLX_OP
AXLY_OP
AXLZ_OP
DREF_OP
DGND_OP
AREF_OP
AGND_OP
Designer: Carlton C. Stedman II
Title
Subcircuits
Size
A
Document Number
3
Rev
5
Date:
Thursday, May 10, 2007
Sheet
1
of
1

APPENDIX B: MATLAB CODE TO CREATE .KML FILE

This appendix contains the MATLAB code from which the .kml file is created for Google Earth.

```
% Input file name to which .kml file will be saved, test name, and test
% description
file_input_name = input('Enter Textfile Name to Examine (no extention): ',
's');
file_name = input('Enter .kml File Name to Save (no extention): ', 's');
test_name = input('Enter Test Name: ', 's');
desc_name = input('Enter Test Description: ', 's');

% Import data into MATLAB and parse for ','
fid = fopen([file_input_name '.txt'], 'r');
A = fread(fid, 'uint8=>char');
fclose(fid);

% Variables for parsing GPS strings:
% time = time of GPS reading
% lat = latitude of GPS reading
% n_s = hemisphere of latitude reading (North or South)
% long = longitude of GPS reading
% e_w = hemisphere of longitude reading (East or West)
% xmin = minimum x axis output from accelerometer
% xmax = maximum x axis output from accelerometer
% ymin = minimum y axis output from accelerometer
% ymax = maximum y axis output from accelerometer
% zmin = minimum z axis output from accelerometer
% zmax = maximum z axis output from accelerometer
time = char(zeros(1,length(A)));
lat = char(zeros(1,length(A)));
n_s = char(zeros(1,length(A)));
long = char(zeros(1,length(A)));
e_w = char(zeros(1,length(A)));
xmax = char(zeros(1,length(A)));
xmin = char(zeros(1,length(A)));
ymax = char(zeros(1,length(A)));
ymin = char(zeros(1,length(A)));
zmax = char(zeros(1,length(A)));
zmin = char(zeros(1,length(A)));
f1 = zeros(1,length(A));
f2 = zeros(1,length(A));
f3 = zeros(1,length(A));
f4 = zeros(1,length(A));
f5 = zeros(1,length(A));
f6 = zeros(1,length(A));
f7 = zeros(1,length(A));
f8 = zeros(1,length(A));
f9 = zeros(1,length(A));
f10 = zeros(1,length(A));
f11 = zeros(1,length(A));
b = 1;
c = 1;
```

```
d = 1;
e = 1;
f = 1;
g = 1;
h = 1;
i = 1;
j = 1;
k = 1;
l = 1;
m = 0;

% Parsing the GPS string
% ------------------ BEGIN PARSING----------------------
remain = A;
while true
   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f1 = sprintf('%s', str);
   time(b:b + length(f1) - 1) = f1;
   time(b + length(f1)) = ' ';
   b = b + length(f1) + 1;
   m = 1;

   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f2 = sprintf('%s', str);
   lat(c:c + length(f2) - 1) = f2;
   lat(c + length(f2)) = ' ';
   c = c + length(f2) + 1;

   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f3 = sprintf('%s', str);
   n_s(d:d + length(f3) - 1) = f3;
   n_s(d + length(f3)) = ' ';
   d = d + length(f3) + 1;

   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f4 = sprintf('%s', str);
   long(e:e + length(f4) - 1) = f4;
   long(e + length(f4)) = ' ';
   e = e + length(f4) + 1;

   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f5 = sprintf('%s', str);
   e_w(f:f + length(f5) - 1) = f5;
   e_w(f + length(f5)) = ' ';
   f = f + length(f5) + 1;

   [str, remain] = strtok(remain, ',');
   if isempty(str),  break;  end
   f6 = sprintf('%s', str);
   xmax(g:g + length(f6) - 1) = f6;
   xmax(g + length(f6)) = ' ';
```

```
    g = g + length(f6) + 1;

    [str, remain] = strtok(remain, ',');
    if isempty(str),  break;  end
    f7 = sprintf('%s', str);
    xmin(h:h + length(f7) - 1) = f7;
    xmin(h + length(f7)) = ' ';
    h = h + length(f7) + 1;

    [str, remain] = strtok(remain, ',');
    if isempty(str),  break;  end
    f8 = sprintf('%s', str);
    ymax(k:k + length(f8) - 1) = f8;
    ymax(k + length(f8)) = ' ';
    k = k + length(f8) + 1;

    [str, remain] = strtok(remain, ',');
    if isempty(str),  break;  end
    f9 = sprintf('%s', str);
    ymin(i:i + length(f9) - 1) = f9;
    ymin(i + length(f9)) = ' ';
    i = i + length(f9) + 1;

    [str, remain] = strtok(remain, ',');
    if isempty(str),  break;  end
    f10 = sprintf('%s', str);
    zmax(j:j + length(f10) - 1) = f10;
    zmax(j + length(f10)) = ' ';
    j = j + length(f10) + 1;

    [str, remain] = strtok(remain, ';');
    if isempty(str),  break;  end
    f11 = sprintf('%s', str);
    zmin(l:l + length(f11) - 2) = f11(2:length(f11));
    zmin(l + length(f11) - 1) = ' ';
    l = l + length(f11);
end
% ------------------ END PARSING----------------------

% Calculating input variables:
% mag_test = magnitude voltage from accelerometer [V]
% long_test = longitude values from GPS
% lat_test = latitude values from GPS
% time = time from GPS
% ----------------BEGIN CALCULATIONS--------------------
% mag_test
% Find maximum magnitude from maximum and minimum outputs (minimum output
% rearranged to compare with maximum)
x1min = zeros(1,length(xmin));
y1min = zeros(1,length(ymin));
z1min = zeros(1,length(zmin));

x1max = zeros(1,length(xmax));
y1max = zeros(1,length(ymax));
z1max = zeros(1,length(zmax));
```

```
x1 = zeros(1,length(xmax));
y1 = zeros(1,length(ymax));
z1 = zeros(1,length(zmax));

xmin1 = str2num(xmin) .* (3.3 / 4095);
ymin1 = str2num(ymin) .* (3.3 / 4095);
zmin1 = str2num(zmin) .* (3.3 / 4095);

x1min = (3.3/2) + ((3.3/2) - xmin1);
y1min = (3.3/2) + ((3.3/2) - ymin1);
z1min = (3.3/2) + ((3.3/2) - zmin1);

x1max = str2num(xmax) .* (3.3 / 4095);
y1max = str2num(ymax) .* (3.3 / 4095);
z1max = str2num(zmax) .* (3.3 / 4095);

x1 = max(x1min, x1max) - ((3.3)/2);
y1 = max(y1min, y1max) - ((3.3)/2);
z1 = max(z1min, z1max) - ((3.3)/2);
mag_test = sqrt((x1.^2)+(y1.^2)+(z1.^2));

% Calculate number of g's from magnitude
g_s = zeros(1,length(mag_test));
for i = 1:length(mag_test)
    if mag_test(i) < 0.33
        g_s(i) = (0.33 + (0.33 - mag_test(i))) / 0.33;
    else
        g_s(i) = mag_test(i) / 0.33;
    end
end

% long_test and lat_test
format long g
long_new = zeros(1,length(str2num(long)));
lat_new = zeros(1,length(str2num(lat)));
long_new = str2num(long);
lat_new = str2num(lat);

% Convert MNEA longitude and latitude style (degrees [D] and minutes [M])
into
% degress
% longitude (NMEA): DDDMM.MMMM
% latitude (NMEA): DDMM.MMMM
% NOTE: DDD (or DD) = whole degrees
%       MM = whole minutes
%       .MMMM = partial minutes
long1 = long_new.*(1e-2);
long2 = long1 - floor(long1);
long3 = long2.*(1e2);
long4 = long3./60;
long5 = floor(long1)+long4;

% Take into account north or south hemisphere for Google Earth
if e_w(1) == 'W'
```

```
    long_test = long5.*(-1);
elseif e_w(1) == 'E'
    long_test = long5;
end

lat1 = lat_new.*(1e-2);
lat2 = lat1 - floor(lat1);
lat3 = lat2.*(1e2);
lat4 = lat3./60;
lat5 = floor(lat1)+lat4;

% Take into account east or west hemisphere for Google Earth
if n_s(1) == 'S'
    lat_test = lat5.*(-1);
elseif n_s(1) == 'N'
    lat_test = lat5;
end

% Convert MNEA time style into standard style
% date (NMEA): DDMMYY
% time (NMEA): HHMMSS.SS (UTC)
% NOTE: DD = day
%       MM = month
%       YY = year
%       HH = hours
%       MM = minutes
%       SS = whole seconds
%       .SS = partial seconds

% time
t1 = str2num(time).*(1e-4);
t2 = floor(t1);
t3 = str2num(time).*(1e-2);
t4 = t3 - floor(t3);
t5 = round(t4.*(1e2));
t6 = str2num(time) - t5;
t7 = str2num(time).*(1e-4);
t8 = t7 - floor(t7);
t9 = round(t8.*(1e2));
hour = t2 - 5;
minute = t9;
second = t5;
% ----------------END CALCULATIONS--------------------

% Start saving .kml file
diary ([file_name '.kml'])
diary on

% Start creating .kml file
% Adds the appriopriate heading, test name, test description, and style of
% pin for roughness classification
disp('<?xml version="1.0" encoding="UTF-8"?>')
disp('<kml xmlns="http://earth.google.com/kml/2.1">')
disp('<Document>')
disp('   <Style id="style1">')
disp('      <Icon>')
```

```
disp('         <href>http://maps.google.com/mapfiles/kml/pushpin/grn-
pushpin.png</href>')
disp('      </Icon>')
disp('   </Style>')
disp('   <Style id="style2">')
disp('      <Icon>')
disp('         <href>http://maps.google.com/mapfiles/kml/pushpin/ltblu-
pushpin.png</href>')
disp('      </Icon>')
disp('   </Style>')
disp('   <Style id="style3">')
disp('      <Icon>')
disp('         <href>http://maps.google.com/mapfiles/kml/pushpin/ylw-
pushpin.png</href>')
disp('      </Icon>')
disp('   </Style>')
disp('   <Style id="style4">')
disp('      <Icon>')
disp('         <href>http://maps.google.com/mapfiles/kml/pushpin/pink-
pushpin.png</href>')
disp('      </Icon>')
disp('   </Style>')
disp('   <Style id="style5">')
disp('      <Icon>')
disp('         <href>http://maps.google.com/mapfiles/kml/pushpin/red-
pushpin.png</href>')
disp('      </Icon>')
disp('   </Style>')
disp('<Folder>')
disp(['   <name>' test_name '</name>'])
disp('   <open>1</open>')
disp(['   <description>' desc_name '</description>'])

% Adds placement pins to GPS locations with accelerometer data
% -----------------------BEGIN LOOP-----------------------------
% Loop to determine level of road roughness, point number, and description
% Level 1 (Very Smooth) = Magnitudes less than 0.34 (green) <= CHANGE
% Level 2 (Smooth) = Magnitudes between 0.34 and 0.375 (light blue)
% Level 3 (Average) = Magnitudes between 0.375 and 0.41 (yellow)
% Level 4 (Rough) = Magnitudes between 0.41 and 0.445 (pink)
% Level 5 (Very Rough) = Magnitudes above 0.445 (red)
for i=1:length(mag_test)
disp('   <Placemark>')
if (mag_test(i) < 0.34)
if minute(i) < 10
    if second(i) < 10
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 1</b><br/>Location is very smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    else
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 1</b><br/>Location is very smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    end
elseif second(i) < 10
```

```
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 1</b><br/>Location is very smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
else
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 1</b><br/>Location is very smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
end
disp('      <styleUrl>#style1</styleUrl>')
end

if (mag_test(i) >= 0.34) && (mag_test(i) < 0.375)
if minute(i) < 10
    if second(i) < 10
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 2</b><br/>Location is smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    else
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 2</b><br/>Location is smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    end
elseif second(i) < 10
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 2</b><br/>Location is smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
else
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 2</b><br/>Location is smooth.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
end
disp('      <styleUrl>#style2</styleUrl>')
end

if (mag_test(i) >= 0.375) && (mag_test(i) < 0.41)
if minute(i) < 10
    if second(i) < 10
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 3</b><br/>Location is average.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    else
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 3</b><br/>Location is average.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    end
elseif second(i) < 10
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
```

```
/><b>Roughness Level 3</b><br/>Location is average.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
else
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 3</b><br/>Location is average.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
end
disp('      <styleUrl>#style3</styleUrl>')
end

if (mag_test(i) >= 0.41) && (mag_test(i) < 0.445)
if minute(i) < 10
    if second(i) < 10
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 4</b><br/>Location is rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    else
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 4</b><br/>Location is rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    end
elseif second(i) < 10
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 4</b><br/>Location is rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
else
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 4</b><br/>Location is rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
end
disp('      <styleUrl>#style4</styleUrl>')
end

if (mag_test(i) >= 0.445)
if minute(i) < 10
    if second(i) < 10
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 5</b><br/>Location is very rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    else
        disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':0' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 5</b><br/>Location is very rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
    end
elseif second(i) < 10
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':0' num2str(second(i)) '.<hr
/><b>Roughness Level 5</b><br/>Location is very rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
else
```

```
    disp(['      <description><b>Point ' num2str(i) '</b><br/>Time: '
num2str(hour(i)) ':' num2str(minute(i)) ':' num2str(second(i)) '.<hr
/><b>Roughness Level 5</b><br/>Location is very rough.<br/> Magnitude is '
num2str(mag_test(i)) ' (' num2str(g_s(i)) ' g`s).</description>'])
end
disp('      <styleUrl>#style5</styleUrl>')
end

% Adds GPS location to point
disp('      <Point>')
disp(['         <coordinates>' num2str(long_test(i)) num2str(',')
num2str(lat_test(i)) '</coordinates>'])
disp('      </Point>')
disp('   </Placemark>')
end
% ------------------------END LOOP----------------------------

% Finishes .kml file
disp('</Folder>')
disp('</Document>')
disp('</kml>')
diary off
disp('The file is complete. It is located in the current directory.')
```

APPENDIX C: "C" CODE

This appendix contains the C code.

```
/*************** STFL-I based Serial Flash Memory Driver *********************

   Filename:    c2082.c
   Description: Library routines for the M25P05A, M25P10A, M25P20, M25P40, M25P80
                M25P16, M25P32, M25P64 Serial Flash Memories

   Version:     V2.0
   Date:        20/07/2005
   Authors:     Tan Zhi, STMicroelectronics, Shanghai (China)
   Copyright (c) 2004 STMicroelectronics.

   THE PRESENT SOFTWARE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH
   CODING INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A
   RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR
   CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH
   SOFTWARE AND/OR THE USE MADE BY CUSTOMERS OF THE CODING INFORMATION CONTAINED HEREIN
   IN CONNECTION WITH THEIR PRODUCTS.
*******************************************************************************

   Version History.
   Ver.   Date      Comments

   1.0   16/11/2004   Initial release
   2.0   20/07/2005   Add support for M25P32, M25P64
                      Add JEDEC ID support for M25P05A, M25P10A

*******************************************************************************

   This source file provides library C code for M25P05A, M25P10A, M25P20,
   M25P40, M25P80, M25P16, M25P32, M25P64 serial flash devices.

   The following functions are available in this library(some memories may only support
   a subset of the list, refer to the specific product datasheet for details):

      Flash(WriteEnable, 0)                        to disable Write Protect in the Flash memory
      Flash(WriteDisable, 0)                       to enable Write Protect in the Flash memory
      Flash(ReadDeviceIdentification, ParameterType)  to get the Device Identification from the
device
      Flash(ReadManufacturerIdentification, ParameterType)(if available in the memory)    to get
the manufacturer Identification from the device
      Flash(ReadStatusRegister, ParameterType)      to get the value in the Status Register from
the device
      Flash(WriteStatusRegister, ParameterType)      to set the value in the Status Register from
the device
      Flash(Read, ParameterType)                   to read from the Flash device
      Flash(FastRead, ParameterType)               to read from the Flash device in a faster way
      Flash(PageProgram, ParameterType)            to write an array of elements within one page
      Flash(SectorErase, ParameterType)            to erase a whole sector
      Flash(BulkErase, ParameterType)              to erase the whole memory
      Flash(DeepPowerDown, 0)                      to set the memory into the low power
consumption mode
      Flash(ReleaseFromDeepPowerDown, 0)           to wake up the memory from the low power
consumption mode
      Flash(Program, ParameterType)                to program an array of elements
      FlashErrorStr()                              to return an error description (define
VERBOSE)

   Note that data Bytes will be referred to as elements throughout the document unless otherwise
specified.

   For further information consult the related Datasheets and Application Note.
   The Application Note gives information about how to modify this code for
   a specific application.
```

```
   The hardware specific functions which may need to be modified by the user are:

      FlashWrite() used to write an element (uCPUBusType) to the Flash memory
      FlashRead()  used to read an element (uCPUBusType) from the Flash memory
      FlashTimeOut() to return after the function has timed out

   A list of the error conditions can be found at the end of the header file.

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

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

#include "c2082.h" /* Header file with global prototypes */
#include "Serialize.h" /* Header file with SPI master abstract prototypes */

#ifdef TIME_H_EXISTS
  #include <time.h>
#endif

#ifdef SYNCHRONOUS_IO
#define WAIT_TILL_Instruction_EXECUTION_COMPLETE(x) FlashTimeOut(0); while(IsFlashBusy()) \
    { \
        if(Flash_OperationTimeOut == FlashTimeOut(x)) return  Flash_OperationTimeOut; \
    };

#else
    // do nothing
#endif


/*****************************************************************************
    Global variables: none
*****************************************************************************/

/*****************************************************************************
Function:     ReturnType Flash( InstructionType insInstruction, ParameterType *fp )
Arguments:    insInstruction is an enum which contains all the available Instructions
    of the SW driver.
              fp is a (union) parameter struct for all Flash Instruction parameters
Return Value: The function returns the following conditions:

   Flash_AddressInvalid,
   Flash_MemoryOverflow,
   Flash_PageEraseFailed,
   Flash_PageNrInvalid,
   Flash_SectorNrInvalid,
   Flash_FunctionNotSupported,
   Flash_NoInformationAvailable,
   Flash_OperationOngoing,
   Flash_OperationTimeOut,
   Flash_ProgramFailed,
   Flash_SpecificError,
   Flash_Success,
   Flash_WrongType

Description:  This function is used to access all functions provided with the
   current Flash device.

Pseudo Code:
   Step 1: Select the right action using the insInstruction parameter
   Step 2: Execute the Flash memory Function
   Step 3: Return the Error Code
*****************************************************************************/
ReturnType Flash( InstructionType insInstruction, ParameterType *fp ) {
    ReturnType  rRetVal;
        ST_uint8  ucStatusRegister;
    ST_uint16 ucDeviceIdentification;
#ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
    ST_uint8  ucManufacturerIdentification;
```

```
#endif
   switch (insInstruction) {
      case WriteEnable:
           rRetVal = FlashWriteEnable( );
           break;

      case WriteDisable:
           rRetVal = FlashWriteDisable( );
           break;

      case ReadDeviceIdentification:
           rRetVal = FlashReadDeviceIdentification(&ucDeviceIdentification);
           (*fp).ReadDeviceIdentification.ucDeviceIdentification = ucDeviceIdentification;
           break;

#ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
      case ReadManufacturerIdentification:
           rRetVal = FlashReadManufacturerIdentification(&ucManufacturerIdentification);
           (*fp).ReadManufacturerIdentification.ucManufacturerIdentification =
ucManufacturerIdentification;
           break;
#endif

      case ReadStatusRegister:
            rRetVal = FlashReadStatusRegister(&ucStatusRegister);
           (*fp).ReadStatusRegister.ucStatusRegister = ucStatusRegister;
           break;

      case WriteStatusRegister:
           ucStatusRegister = (*fp).WriteStatusRegister.ucStatusRegister;
            rRetVal = FlashWriteStatusRegister(ucStatusRegister);
           break;

      case Read:
           rRetVal = FlashRead( (*fp).Read.udAddr,
                                (*fp).Read.pArray,
                                (*fp).Read.udNrOfElementsToRead
                              );
           break;

      case FastRead:
           rRetVal = FlashFastRead( (*fp).Read.udAddr,
                                    (*fp).Read.pArray,
                                    (*fp).Read.udNrOfElementsToRead
                                  );
           break;

      case PageProgram:
           rRetVal = FlashProgram( (*fp).PageProgram.udAddr,
                                   (*fp).PageProgram.pArray,
                                   (*fp).PageProgram.udNrOfElementsInArray
                                 );
           break;

      case SectorErase:
           rRetVal = FlashSectorErase((*fp).SectorErase.ustSectorNr );
           break;

      case BulkErase:
           rRetVal = FlashBulkErase( );
           break;


 #ifndef NO_DEEP_POWER_DOWN_SUPPORT
      case DeepPowerDown:
           rRetVal = FlashDeepPowerDown( );
           break;

      case ReleaseFromDeepPowerDown:
           rRetVal = FlashReleaseFromDeepPowerDown( );
           break;
```

```
#endif

      case Program:
           rRetVal = FlashProgram( (*fp).Program.udAddr,
                                   (*fp).Program.pArray,
                                   (*fp).Program.udNrOfElementsInArray);
           break;

       default:
           rRetVal = Flash_FunctionNotSupported;
           break;

   } /* EndSwitch */
   return rRetVal;
} /* EndFunction Flash */

/*****************************************************************************
Function:     FlashWriteEnable( void )
Arguments:    void

Return Value:
   Flash_Success

Description:  This function sets the Write Enable Latch(WEL)
              by sending a WREN Instruction.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially
*****************************************************************************/
ReturnType  FlashWriteEnable( void )
{
    CharStream char_stream_send;
    ST_uint8  cWREN = SPI_FLASH_INS_WREN;

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length = 1;
    char_stream_send.pChar  = &cWREN;

    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTransOnly_DeSelectSlave
              );
    return Flash_Success;
}

/*****************************************************************************
Function:     FlashWriteDisable( void )
Arguments:    void

Return Value:
   Flash_Success

Description:  This function resets the Write Enable Latch(WEL)
              by sending a WRDI Instruction.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially
*****************************************************************************/
ReturnType  FlashWriteDisable( void )
{
    CharStream char_stream_send;
    ST_uint8  cWRDI = SPI_FLASH_INS_WRDI;

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length = 1;
    char_stream_send.pChar  = &cWRDI;
```

```
    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTransOnly_DeSelectSlave
              );
    return Flash_Success;
}

/*****************************************************************************
Function:     FlashReadDeviceIdentification( ST_uint16 *uwpDeviceIdentification)
Arguments:    uwpDeviceIdentificaiton, 16-bit buffer to hold the DeviceIdentification read from
the
              memory, with memory type residing in the higher 8 bits, and
              memory capacity in the lower ones.

Return Value:
   Flash_Success
   Flash_WrongType(if USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE defined)

Description:  This function returns the Device Identification (memory type + memory capacity)
              by sending an SPI_FLASH_INS_RDID Instruction.
              After retrieving the Device Identificaiton, the routine checks if the device is
              an expected device(defined by EXPECTED_DEVICE). If not,
              Flash_WrongType is returned.

              If USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE is defined, the returned 16-bit
              word comprises memory type(higher 8 bits) and memory capacity
              (lower 8 bits).
              If USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE is NOT defined, only the lower
              8-bit byte of the returned 16-bit word is valid information,i.e. the
              Device Identification.
              For memories that have a capacity of more than 16Mb(inclusive),
              USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE is defined by default.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially
   Step 3: Device Identification is returned
*****************************************************************************/
ReturnType  FlashReadDeviceIdentification( ST_uint16 *uwpDeviceIdentification)
{
#ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  cRDID = SPI_FLASH_INS_RDID;
    ST_uint8  pIdentification[3];

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length  = 1;
    char_stream_send.pChar   = &cRDID;

    char_stream_recv.length  = 3;
    char_stream_recv.pChar   = &pIdentification[0];

    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    // Step 3: Device Identification is returned ( memory type + memory capacity )
    *uwpDeviceIdentification = char_stream_recv.pChar[1];
    *uwpDeviceIdentification <<= 8;
    *uwpDeviceIdentification |= char_stream_recv.pChar[2];

    if(EXPECTED_DEVICE == *uwpDeviceIdentification)
        return Flash_Success;
    else
        return Flash_WrongType;
```

```
#else   // USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE not defined
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  pIns[4];
    ST_uint8  cRDID = SPI_FLASH_INS_RES;
    ST_uint8  pIdentification;

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length  = 4;
    char_stream_send.pChar   = &pIns[0];
    pIns[0] = SPI_FLASH_INS_RES;
    pIns[1] = SPI_FLASH_INS_DUMMY;
    pIns[2] = SPI_FLASH_INS_DUMMY;
    pIns[3] = SPI_FLASH_INS_DUMMY;

    char_stream_recv.length  = 1;
    char_stream_recv.pChar   = &pIdentification;

    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    // Step 3: Get the returned device Identification
    *uwpDeviceIdentification = *char_stream_recv.pChar;

    if(EXPECTED_DEVICE == *uwpDeviceIdentification)
        return Flash_Success;
    else
        return Flash_WrongType;
#endif
}

#ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
/*****************************************************************************
Function:     FlashReadManufacturerIdentification( ST_uint8 *ucpManufactureIdentification)
Arguments:    ucpManufacturerIdentification, 8-bit buffer to hold the manufacturer identification
              being read from the memory

Return Value:
   Flash_WrongType: if any value other than MANUFACTURER_ST(0x20) is returned
   Flash_Success : if MANUFACTURER_ST(0x20) is correctly returned

Description:  This function returns the Manufacturer Identification(0x20) by sending an
              SPI_FLASH_INS_RDID Instruction.
              After retrieving the Manufacturer Identification, the routine checks if the device
is
              an ST memory product. If not, Flash_WrongType is returned.

Note: The availability of this function should be checked in the appropriate datasheet
       for each memory.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially
   Step 3: get the Manufacturer Identification
*****************************************************************************/
ReturnType  FlashReadManufacturerIdentification( ST_uint8 *ucpManufacturerIdentification)
{
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  cRDID = SPI_FLASH_INS_RDID;
    ST_uint8  pIdentification[3];

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length  = 1;
    char_stream_send.pChar   = &cRDID;
    char_stream_recv.length  = 3;
```

```
    char_stream_recv.pChar   = &pIdentification[0];

    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    // Step 3: get the Manufacturer Identification
    *ucpManufacturerIdentification = pIdentification[0];
    if(MANUFACTURER_ST == *ucpManufacturerIdentification)
    {
        return Flash_Success;
    }
    else
    {
        return Flash_WrongType;
    }
}
#endif // end of #ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE

/*****************************************************************************
Function:     FlashReadStatusRegister( ST_uint8 *ucpStatusRegister)
Arguments:    ucpStatusRegister, 8-bit buffer to hold the Status Register value read
              from the memory

Return Value:
   Flash_Success

Description:  This function reads the Status Register by sending an
               SPI_FLASH_INS_RDSR Instruction.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially, get the Status Register content

*****************************************************************************/
ReturnType  FlashReadStatusRegister( ST_uint8 *ucpStatusRegister)
{
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  cRDSR = SPI_FLASH_INS_RDSR;


    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length  = 1;
    char_stream_send.pChar   = &cRDSR;
    char_stream_recv.length  = 1;
    char_stream_recv.pChar   = ucpStatusRegister;

    // Step 2: Send the packet serially, get the Status Register content
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    return Flash_Success;
}


/*****************************************************************************
Function:     FlashWriteStatusRegister( ST_uint8 ucStatusRegister)
Arguments:    ucStatusRegister, an 8-bit new value to be written to the Status Register

Return Value:
   Flash_Success

Description:  This function modifies the Status Register by sending an
              SPI_FLASH_INS_WRSR Instruction.
```

```
              The Write Status Register (WRSR) Instruction has no effect
              on b6, b5, b1(WEL) and b0(WIP) of the Status Register.b6 and b5 are
              always read as 0.

Pseudo Code:
   Step 1: Disable Write protection
   Step 2: Initialize the data (i.e. Instruction & value) packet to be sent serially
   Step 3: Send the packet serially
   Step 4: Wait until the operation completes or a timeout occurs.
*****************************************************************************/
ReturnType  FlashWriteStatusRegister( ST_uint8 ucStatusRegister)
{
    CharStream char_stream_send;
    ST_uint8  pIns_Val[2];

    // Step 1: Disable Write protection
    FlashWriteEnable();

    // Step 2: Initialize the data (i.e. Instruction & value) packet to be sent serially
    char_stream_send.length = 2;
    char_stream_send.pChar  = pIns_Val;
    pIns_Val[0] = SPI_FLASH_INS_WRSR;
    pIns_Val[1] = ucStatusRegister;

    // Step 3: Send the packet serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTransOnly_DeSelectSlave
              );
    // Step 4: Wait until the operation completes or a timeout occurs.
    WAIT_TILL_Instruction_EXECUTION_COMPLETE(1)
    return Flash_Success;
}

/*****************************************************************************
Function:     FlashRead( ST_uint32 udAddr, ST_uint8 *ucpElements, ST_uint32 udNrOfElementsToRead)
Arguments:    udAddr, start address to read from
              ucpElements, buffer to hold the elements to be returned
              udNrOfElementsToRead, number of elements to be returned, counted in bytes.

Return Value:
   Flash_AddressInvalid
   Flash_Success

Description:  This function reads the Flash memory by sending an
              SPI_FLASH_INS_READ Instruction.
              by design, the whole Flash memory space can be read with one READ Instruction
              by incrementing the start address and rolling to 0x0 automatically,
              that is, this function is across pages and sectors.

Pseudo Code:
   Step 1: Validate address input
   Step 2: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 3: Send the packet serially, and fill the buffer with the data being returned
*****************************************************************************/
ReturnType  FlashRead( uAddrType udAddr, ST_uint8 *ucpElements, ST_uint32 udNrOfElementsToRead)
{
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  pIns_Addr[4];

    // Step 1: Validate address input
    if(!(udAddr <  FLASH_SIZE)) return Flash_AddressInvalid;

    // Step 2: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length   = 4;
    char_stream_send.pChar    = pIns_Addr;
    pIns_Addr[0]              = SPI_FLASH_INS_READ;
    pIns_Addr[1]              = udAddr>>16;
    pIns_Addr[2]              = udAddr>>8;
```

```
    pIns_Addr[3]              = udAddr;

    char_stream_recv.length   = udNrOfElementsToRead;
    char_stream_recv.pChar    = ucpElements;

    // Step 3: Send the packet serially, and fill the buffer with the data being returned
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    return Flash_Success;
}

/*****************************************************************************
Function:     FlashFastRead( ST_uint32 udAddr, ST_uint8 *ucpElements, ST_uint32
udNrOfElementsToRead)
Arguments:    udAddr, start address to read from
              ucpElements, buffer to hold the elements to be returned
              udNrOfElementsToRead, number of elements to be returned, counted in bytes.

Return Value:
   Flash_AddressInvalid
   Flash_Success

Description:  This function reads the Flash memory by sending an
              SPI_FLASH_INS_FAST_READ Instruction.
              by design, the whole Flash memory space can be read with one FAST_READ Instruction
              by incrementing the start address and rolling to 0x0 automatically,
              that is, this function is across pages and sectors.

Pseudo Code:
   Step 1: Validate address input
   Step 2: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 3: Send the packet serially, and fill the buffer with the data being returned
*****************************************************************************/
ReturnType  FlashFastRead( uAddrType udAddr, ST_uint8 *ucpElements, ST_uint32
udNrOfElementsToRead)
{
    CharStream char_stream_send;
    CharStream char_stream_recv;
    ST_uint8  pIns_Addr[5];

    // Step 1: Validate address input
    if(!(udAddr <  FLASH_SIZE)) return Flash_AddressInvalid;

    // Step 2: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length   = 5;
    char_stream_send.pChar    = pIns_Addr;
    pIns_Addr[0]              = SPI_FLASH_INS_FAST_READ;
    pIns_Addr[1]              = udAddr>>16;
    pIns_Addr[2]              = udAddr>>8;
    pIns_Addr[3]              = udAddr;
    pIns_Addr[4]              = SPI_FLASH_INS_DUMMY;

    char_stream_recv.length   = udNrOfElementsToRead;
    char_stream_recv.pChar    = ucpElements;

    // Step 3: Send the packet serially, and fill the buffer with the data being returned
    Serialize(&char_stream_send,
              &char_stream_recv,
              enumEnableTansRecv_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    return Flash_Success;
}

/*****************************************************************************
Function:     FlashPageProgram( ST_uint32 udAddr, ST_uint8 *pArray, ST_uint32
```

```
udNrOfElementsInArray)
Arguments:    udAddr, start address to write to
              pArray, buffer to hold the elements to be programmed
              udNrOfElementsInArray, number of elements to be programmed, counted in bytes

Return Value:
   Flash_AddressInvalid
   Flash_OperationOngoing
   Flash_OperationTimeOut
   Flash_Success

Description:  This function writes a maximum of 256 bytes of data into the memory by sending an
              SPI_FLASH_INS_PP Instruction.
              by design, the PP Instruction is effective WITHIN ONE page,i.e. 0xXX00 - 0xXXff.
              when 0xXXff is reached, the address rolls over to 0xXX00 automatically.
Note:
              This function does not check whether the target memory area is in a Software
              Protection Mode(SPM) or Hardware Protection Mode(HPM), in which case the PP
              Instruction will be ignored.
              The function assumes that the target memory area has previously been unprotected at
both
              the hardware and software levels.
              To unprotect the memory, please call FlashWriteStatusRegister(ST_uint8
ucStatusRegister),
              and refer to the datasheet for the setup of a proper ucStatusRegister value.
Pseudo Code:
   Step 1: Validate address input
   Step 2: Check whether any previous Write, Program or Erase cycle is on going
   Step 3: Disable Write protection (the Flash memory will automatically enable it again after
           the execution of the Instruction)
   Step 4: Initialize the data (Instruction & address only) packet to be sent serially
   Step 5: Send the packet (Instruction & address only) serially
   Step 6: Initialize the data (data to be programmed) packet to be sent serially
   Step 7: Send the packet (data to be programmed) serially
   Step 8: Wait until the operation completes or a timeout occurs.
*****************************************************************************/
ReturnType  FlashPageProgram( uAddrType udAddr, ST_uint8 *pArray , ST_uint16
udNrOfElementsInArray)
{
    CharStream char_stream_send;
    ST_uint8  pIns_Addr[4];

    // Step 1: Validate address input
    if(!(udAddr <  FLASH_SIZE)) return Flash_AddressInvalid;

    // Step 2: Check whether any previous Write, Program or Erase cycle is on-going
    if(IsFlashBusy()) return Flash_OperationOngoing;

    // Step 3: Disable Write protection
    FlashWriteEnable();

    // Step 4: Initialize the data (Instruction & address only) packet to be sent serially
    char_stream_send.length   = 4;
    char_stream_send.pChar    = pIns_Addr;
    pIns_Addr[0]              = SPI_FLASH_INS_PP;
    pIns_Addr[1]              = udAddr>>16;
    pIns_Addr[2]              = udAddr>>8;
    pIns_Addr[3]              = udAddr;

    // Step 5: Send the packet (Instruction & address only) serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumNull
              );

    // Step 6: Initialize the data (data to be programmed) packet to be sent serially
    char_stream_send.length   = udNrOfElementsInArray;
    char_stream_send.pChar    = pArray;

    // Step 7: Send the packet (data to be programmed) serially
```

```
    Serialize(&char_stream_send,
              ptrNull,
              enumNull,
              enumDisableTransOnly_DeSelectSlave
              );

    // Step 8: Wait until the operation completes or a timeout occurs.
    WAIT_TILL_Instruction_EXECUTION_COMPLETE(1)

    return Flash_Success;
}

/*****************************************************************************
Function:     ReturnType FlashSectorErase( uSectorType uscSectorNr )
Arguments:    uSectorType is the number of the Sector to be erased.

Return Values:
   Flash_SectorNrInvalid
   Flash_OperationOngoing
   Flash_OperationTimeOut
   Flash_Success

Description:  This function erases the Sector specified in uscSectorNr by sending an
              SPI_FLASH_INS_SE Instruction.
              The function checks that the sector number is within the valid range
              before issuing the erase Instruction. Once erase has completed the status
              Flash_Success is returned.
Note:
              This function does not check whether the target memory area is in a Software
              Protection Mode(SPM) or Hardware Protection Mode(HPM), in which case the PP
              Instruction will be ignored.
              The function assumes that the target memory area has previously been unprotected at
both
              the hardware and software levels.
              To unprotect the memory, please call FlashWriteStatusRegister(ST_uint8
ucStatusRegister),
              and refer to the datasheet to set a proper ucStatusRegister value.

Pseudo Code:
   Step 1: Validate the sector number input
   Step 2: Check whether any previous Write, Program or Erase cycle is on going
   Step 3: Disable Write protection (the Flash memory will automatically enable it
           again after the execution of the Instruction)
   Step 4: Initialize the data (Instruction & address) packet to be sent serially
   Step 5: Send the packet (Instruction & address) serially
   Step 6: Wait until the operation completes or a timeout occurs.
*****************************************************************************/
ReturnType  FlashSectorErase( uSectorType uscSectorNr )
{
    CharStream char_stream_send;
    ST_uint8  pIns_Addr[4];

    // Step 1: Validate the sector number input
    if(!(uscSectorNr < FLASH_SECTOR_COUNT)) return Flash_SectorNrInvalid;

    // Step 2: Check whether any previous Write, Program or Erase cycle is on going
    if(IsFlashBusy()) return Flash_OperationOngoing;

    // Step 3: Disable Write protection
    FlashWriteEnable();

    // Step 4: Initialize the data (Instruction & address) packet to be sent serially
    char_stream_send.length   = 4;
    char_stream_send.pChar    = &pIns_Addr[0];
    pIns_Addr[0]              = SPI_FLASH_INS_SE;
    #ifdef FLASH_SMALLER_SECTOR_SIZE
    pIns_Addr[1]              = uscSectorNr>>1;
    pIns_Addr[2]              = uscSectorNr<<7;
    #else
    pIns_Addr[1]              = uscSectorNr;
    pIns_Addr[2]              = 0;
```

```
    #endif
    pIns_Addr[3]              = 0;

    // Step 5: Send the packet (Instruction & address) serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    // Step 6: Wait until the operation completes or a timeout occurs.
    WAIT_TILL_Instruction_EXECUTION_COMPLETE(3)

    return Flash_Success;
}
/*******************************************************************************
Function:     ReturnType FlashBulkErase( void )
Arguments:    none

Return Values:
   Flash_OperationOngoing
   Flash_OperationTimeOut
   Flash_Success

Description:  This function erases the whole Flash memory by sending an
              SPI_FLASH_INS_BE Instruction.
Note:
              This function does not check whether the target memory area (or part of it)
                        is in a Software Protection Mode(SPM) or Hardware Protection Mode(HPM),
                        in which case the PP Instruction will be ignored.
              The function assumes that the target memory area has previously been unprotected at
both
              the hardware and software levels.
              To unprotect the memory, please call FlashWriteStatusRegister(ST_uint8
ucStatusRegister),
              and refer to the datasheet to set a proper ucStatusRegister value.

Pseudo Code:
   Step 1: Check whether any previous Write, Program or Erase cycle is on going
   Step 2: Disable the Write protection (the Flash memory will automatically enable it
           again after the execution of the Instruction)
   Step 3: Initialize the data (Instruction & address) packet to be sent serially
   Step 4: Send the packet (Instruction & address) serially
   Step 5: Wait until the operation completes or a timeout occurs.
*******************************************************************************/
ReturnType  FlashBulkErase( void )
{
    CharStream char_stream_send;
    ST_uint8  cBE = SPI_FLASH_INS_BE;

    // Step 1: Check whether any previous Write, Program or Erase cycle is on going
    if(IsFlashBusy()) return Flash_OperationOngoing;

    // Step 2: Disable Write protection
    FlashWriteEnable();

    // Step 3: Initialize the data(Instruction & address) packet to be sent serially
    char_stream_send.length   = 1;
    char_stream_send.pChar    = &cBE;

    // Step 4: Send the packet(Instruction & address) serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTansRecv_DeSelectSlave
              );

    // Step 5: Wait until the operation completes or a timeout occurs.
    WAIT_TILL_Instruction_EXECUTION_COMPLETE(BE_TIMEOUT)

    return Flash_Success;
```

```
}

#ifndef NO_DEEP_POWER_DOWN_SUPPORT
/*****************************************************************************
Function:     FlashDeepPowerDown( void )
Arguments:    void

Return Value:
   Flash_OperationOngoing
   Flash_Success

Description:  This function puts the device in the lowest consumption
              mode (the Deep Power-down mode) by sending an SPI_FLASH_INS_DP.
              After calling this routine, the Flash memory will not respond to any
              subsequent Instruction except for the RDP Instruction.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Check whether any previous Write, Program or Erase cycle is on going
   Step 3: Send the packet serially
*****************************************************************************/
ReturnType  FlashDeepPowerDown( void )
{
    CharStream char_stream_send;
    ST_uint8  cDP = SPI_FLASH_INS_DP;

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length = 1;
    char_stream_send.pChar  = &cDP;

    // Step 2: Check whether any previous Write, Program or Erase cycle is on going
    if(IsFlashBusy()) return Flash_OperationOngoing;

    // Step 3: Send the packet serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTransOnly_DeSelectSlave
              );

    return Flash_Success;
}

/*****************************************************************************
Function:     FlashReleaseFromDeepPowerDown( void )
Arguments:    void

Return Value:
   Flash_Success

Description:  This function takes the device out of the Deep Power-down
              mode by sending an SPI_FLASH_INS_RES.

Pseudo Code:
   Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
   Step 2: Send the packet serially
*****************************************************************************/
ReturnType  FlashReleaseFromDeepPowerDown( void )
{
    CharStream char_stream_send;
    ST_uint8  cRES = SPI_FLASH_INS_RES;

    // Step 1: Initialize the data (i.e. Instruction) packet to be sent serially
    char_stream_send.length = 1;
    char_stream_send.pChar  = &cRES;

    // Step 2: Send the packet serially
    Serialize(&char_stream_send,
              ptrNull,
              enumEnableTransOnly_SelectSlave,
              enumDisableTransOnly_DeSelectSlave
```

```
            );
    return Flash_Success;
}
#endif //   end of #ifndef NO_DEEP_POWER_DOWN_SUPPORT

/*****************************************************************************
Function:     FlashProgram( ST_uint32 udAddr, ST_uint8 *pArray, ST_uint32 udNrOfElementsInArray )
Arguments:    udAddr, start address to program
              pArray, address of the  buffer that holds the elements to be programmed
              udNrOfElementsInArray, number of elements to be programmed, counted in bytes

Return Value:
   Flash_AddressInvalid
   Flash_MemoryOverflow
   Flash_OperationTimeOut
   Flash_Success

Description:  This function programs a chunk of data into the memory at one go.
              If the start address and the available space are checked successfully,
              this function programs data from the buffer(pArray) to the memory sequentially by
              invoking FlashPageProgram(). This function automatically handles page boundary
              crossing, if any.
              Like FlashPageProgram(), this function assumes that the memory to be programmed
              has been previously erased or that bits are only changed from 1 to 0.
Note:
              This function does not check whether the target memory area is in a Software
              Protection Mode(SPM) or Hardware Protection Mode(HPM), in which case the PP
              Instruction will be ignored.
              The function assumes that the target memory area has previously been unprotected at
both
              the hardware and software levels.
              To unprotect the memory, please call FlashWriteStatusRegister(ST_uint8
ucStatusRegister),
              and refer to the datasheet for a proper ucStatusRegister value.

Pseudo Code:
   Step 1: Validate address input
   Step 2: Check memory space available on the whole memory
   Step 3: calculte memory space available within the page containing the start address(udAddr)
   Step 3-1: if the page boundary is crossed, invoke FlashPageProgram() repeatedly
   Step 3-2: if the page boundary is not crossed, invoke FlashPageProgram() once only
*****************************************************************************/
ReturnType  FlashProgram( ST_uint32 udAddr, ST_uint8 *pArray, ST_uint32 udNrOfElementsInArray )
{
    ST_uint16 ucMargin;
    ST_uint16 ucPageCount, ucRemainder;
    ReturnType typeReturn;

    // Step 1: Validate address input
    if(!(udAddr <  FLASH_SIZE)) return Flash_AddressInvalid;

    // Step 2: Check memory space available on the whole memory
    if(udAddr + udNrOfElementsInArray > FLASH_SIZE) return Flash_MemoryOverflow;

    // Step 3: calculte memory space available within the page containing the start
address(udAddr)
    ucMargin = (ST_uint8)(~udAddr) + 1;

    // Step 3-1: if the page boundary is crossed, invoke FlashPageWrite() repeatedly
    if(udNrOfElementsInArray > ucMargin)
    {
        typeReturn = FlashPageProgram(udAddr, pArray, ucMargin);
        if(Flash_Success != typeReturn) return typeReturn;             // return immediately if
Not successful

        udNrOfElementsInArray -= ucMargin;                             // re-calculate the
number of elements
        pArray += ucMargin;                                            // modify the pointer to
the buffer
        udAddr += ucMargin;                                            // modify the start
address in the memory
```

```
        ucPageCount = udNrOfElementsInArray / FLASH_WRITE_BUFFER_SIZE;  // calculate the number
of pages to be programmed
        ucRemainder = udNrOfElementsInArray % FLASH_WRITE_BUFFER_SIZE;   // calculate the
remainder after filling up one or more whole pages
        while(ucPageCount--)
        {
            typeReturn = FlashPageProgram(udAddr, pArray, FLASH_WRITE_BUFFER_SIZE);
            if(Flash_Success != typeReturn) return typeReturn;          // return immediately if
Not successful
            pArray += FLASH_WRITE_BUFFER_SIZE;
            udAddr += FLASH_WRITE_BUFFER_SIZE;
        };
        return FlashPageProgram(udAddr, pArray, ucRemainder);
    }
    // Step 3-2: if the page boundary is not crossed, invoke FlashPageWrite() once only
    else
    {
        return FlashPageProgram(udAddr, pArray, udNrOfElementsInArray);
    }
}

/*****************************************************************************
Function:     IsFlashBusy( )
Arguments:    none

Return Value:
   TRUE
   FALSE

Description:  This function checks the Write In Progress (WIP) bit to determine whether
              the Flash memory is busy with a Write, Program or Erase cycle.

Pseudo Code:
   Step 1: Read the Status Register.
   Step 2: Check the WIP bit.
*****************************************************************************/
BOOL IsFlashBusy()
{
    ST_uint8 ucSR;

    // Step 1: Read the Status Register.
    FlashReadStatusRegister(&ucSR);

    // Step 2: Check the WIP bit.
    if(ucSR & SPI_FLASH_WIP)
        return TRUE;
    else
        return FALSE;
}

#ifdef VERBOSE
/*****************************************************************************
Function:     FlashErrorStr( ReturnType rErrNum );
Arguments:    rErrNum is the error number returned from other Flash memory Routines

Return Value: A pointer to a string with the error message

Description:  This function is used to generate a text string describing the
   error from the Flash memory. Call with the return value from other Flash memory routines.

Pseudo Code:
   Step 1: Return the correct string.
*****************************************************************************/
ST_sint8 *FlashErrorStr( ReturnType rErrNum )
{
   switch(rErrNum)
   {
   case Flash_AddressInvalid:
      return "Flash - Address is out of Range";
   case Flash_MemoryOverflow:
      return "Flash - Memory Overflows";
```

```
   case Flash_PageEraseFailed:
      return "Flash - Page Erase failed";
   case Flash_PageNrInvalid:
      return "Flash - Page Number is out of Range";
   case Flash_SectorNrInvalid:
      return "Flash - Sector Number is out of Range";
  case Flash_FunctionNotSupported:
      return "Flash - Function not supported";
   case Flash_NoInformationAvailable:
      return "Flash - No Additional Information Available";
   case Flash_OperationOngoing:
      return "Flash - Operation ongoing";
   case Flash_OperationTimeOut:
      return "Flash - Operation TimeOut";
   case Flash_ProgramFailed:
      return "Flash - Program failed";
   case Flash_Success:
      return "Flash - Success";
   case Flash_WrongType:
      return "Flash - Wrong Type";
   default:
      return "Flash - Undefined Error Value";
   } /* EndSwitch */
} /* EndFunction FlashErrorString */
#endif /* VERBOSE Definition */


/*****************************************************************************
Function:     FlashTimeOut(ST_uint32 udSeconds)
Arguments:    udSeconds holds the number of seconds before TimeOut occurs

Return Value:
   Flash_OperationTimeOut
   Flash_OperationOngoing

Example:   FlashTimeOut(0)  // Initializes the Timer

           While(1) {
              ...
              If (FlashTimeOut(5) == Flash_OperationTimeOut) break;
              // The loop is executed for 5 Seconds before the operation is aborted
           } EndWhile

*****************************************************************************/
#ifdef TIME_H_EXISTS
/*---------------------------------------------------------------------------
Description:   This function provides a timeout for Flash polling actions or
   other operations which would otherwise never return.
   The Routine uses the function clock() inside ANSI C library "time.h".
---------------------------------------------------------------------------*/
ReturnType FlashTimeOut(ST_uint32 udSeconds){
   static clock_t clkReset,clkCount;

   if (udSeconds == 0) { /* Set Timeout to 0 */
      clkReset=clock();
   } /* EndIf */

   clkCount = clock() - clkReset;

   if (clkCount<(CLOCKS_PER_SEC*(clock_t)udSeconds))
      return Flash_OperationOngoing;
   else
         return Flash_OperationTimeOut;
}/* EndFunction FlashTimeOut */

#else
/*---------------------------------------------------------------------------
Description:   This function provides a timeout for Flash polling actions or
   other operations which would otherwise never return.
   The Routine uses COUNT_FOR_A_SECOND which is considered to be a loop that
   counts for one second. It needs to be adapted to the target Hardware.
```

```
------------------------------------------------------------------------------*/
ReturnType FlashTimeOut(ST_uint32 udSeconds) {

   static ST_uint32 udCounter = 0;
   if (udSeconds == 0) { /* Set Timeout to 0 */
     udCounter = 0;
   } /* EndIf */

   if (udCounter == (udSeconds * COUNT_FOR_A_SECOND)) {
      udCounter = 0;
      return Flash_OperationTimeOut;
   } else {
      udCounter++;
      return Flash_OperationOngoing;
   } /* Endif */

} /* EndFunction FlashTimeOut */
#endif /* TIME_H_EXISTS */

/*****************************************************************************
 End of c2082.c
*****************************************************************************/

/* SPI MEMORY TEST PROGRAM */

#include "msp430x16x.h"      // Definitions, constants, etc for msp430F169

#define DEBUG
#include "mqp.h"
#include "mqpdipsw.h"

#define LCD4BIT
#include "mqplcd.h"

/****************** FUNCTION DECLARATIONS ***************************/
/* Modes */
#include "mqpdatalog.h"
#include "mqpdownload.h"
#include "mqpdebug.h"

/******************** MAIN FUNCTION ********************/
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;   /* stop the watchdog timer */
  lcdInit();                 /* initialize the LCD */
  dipswInit();               /* initialize the DIP switch */
  taInit();
  _EINT();

  /* print welcome message */
  lcdPrints( "GPS Road\nMapping" );
  hwDelay( &timerA, 1, 0x7FFF );  /* set up timer A to wait for .5 second */
  while( timerA.status != DONE );
  lcdClear();

  lcdPrints( "v0.9" );
  hwDelay( &timerA, 1, 0x7FFF );  /* set up timer A to wait for .5 second */
  while( timerA.status != DONE );
  lcdClear();

  /* main loop */
  while(1){
    /* run specified mode based on DIP switch */
    switch( dipswRead() ){
    case DATALOGMODE:
      lcdPrints( "Datalog\nMode" );
      hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
      while( timerA.status != DONE );
      datalogMode();
      break;
    case DOWNLOADMODE:
```

```
      lcdPrints( "Download\nMode" );
      hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
      while( timerA.status != DONE );
      downloadMode();
      break;
    case DEBUGMODE:
      lcdPrints( "Debug\nMode" );
      hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
      while( timerA.status != DONE );
      debugMode();
      break;
    case DELETEMODE:
       lcdPrints( "Delete\nMode" );
       hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
       while( timerA.status != DONE );

        /* set up USART1 for SPI for SFM */
       _DINT();
       sfmInit1();
       if( sfmPost() == Flash_Success ){
        sfmErase();
       } else {
         lcdClear();
         lcdPrints( "Bad Mem!" );
       }
       break;
    default:
      lcdPrints( "No Mode?\n" );
      hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
      while( timerA.status != DONE );
    }

    /* sleep a bit */
    hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
    while( timerA.status != DONE );
    lcdClear();
  }
}

/****** implementation File for support of STFL-I based Serial Flash Memory Driver *****

   Filename:    Serialize.c
   Description:  Support to c2076.c. This files is aimed at giving a basic
    example of the SPI serial interface used to communicate with STMicroelectronics
    serial Flash devices. The functions below are used in an environment where the
    master has an embedded SPI port (STMicroelectronics µPSD).

   Version:     1.0
   Date:        08-11-2004
   Authors:    Tan Zhi, STMicroelectronics, Shanghai (China)
   Copyright (c) 2004 STMicroelectronics.

   THE PRESENT SOFTWARE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH
   CODING INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A
   RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR
   CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH
   SOFTWARE AND/OR THE USE MADE BY CUSTOMERS OF THE CODING INFORMATION CONTAINED HEREIN
   IN CONNECTION WITH THEIR PRODUCTS.
*******************************************************************************

   Version History.
   Ver.   Date      Comments

   1.0   08/11/2004   Initial release

*******************************************************************************/
#include "Serialize.h"

/* added by Carlton */
void SelectSlave(void);
void DeSelectSlave(void);
```

```
void EnableTrans(void);
void DisableTrans(void);
void EnableRcv(void);
void DisableRcv(void);
/*******************/

/** uPSD specific includes
#include ".\uPSD\uPSD3300.h"
#include ".\uPSD\TurboLite_hardware.h"
#include ".\uPSD\Turbo_timer.h"
#include ".\uPSD\Turbo_SPI.h"
**/
/*****************************************************************************
Function:     InitSPIMaster(void)
Arguments:
Return Values:There is no return value for this function.
Description:  This function is a one-time configuration for the CPU to set some
              ports to work in SPI mode (when they have multiple functions. For
              example, in some CPUs, the ports can be GPIO pins or SPI pins if
              properly configured).
              please refer to the specific CPU datasheet for proper
              configurations.
*****************************************************************************/

void InitSPIMaster(void)
{
/*
       P4SFS0 |= 0x70;
       P4SFS1 |= 0x70;       // Setup P4[4..6] Port as SPI
                                    // P4.7 works in GPIO mode as the Slave Select signal

       SPICON1=0x00;  //bit3:TEIE=0. SPI transmission end interrupt disable
                                    //bit2:RORIE=0.       SPI receive overrun interrupt disable
                                    //bit1:TIE=0.  SPI transmission interrupt disable
                                    //bit0:RIE=0   SPI reception interrupt disable

       SPICLKD=0x2C;  //select frequency divider=0x2C

       SPICON0=0x72;  //bit6:TE=1.   SPI Transmitter enable
                                    //bit5:RE=1.   SPI Receiver enable
                                    //bit4:SPIEN=1. SPI enable
                                    //bit3:SSEL=0.        SPI Slave select output is
disabled,use P4.7 as the Select Slave signal
                                    //bit2:FLSB=0.        SPI Transfer the MSB first
                                    //bit1:SPO=1.  SPI Sample data on the rising edge of the
clock
*/
}

/*****************************************************************************
Function:     ConfigureSpiMaster(SpiMasterConfigOptions opt)
Arguments:    opt configuration options, all acceptable values are enumerated in
              SpiMasterConfigOptions, which is a typedefed enum.
Return Values:There is no return value for this function.
Description:  This function can be used to properly configure the SPI master
              before and after the transfer/receive operation
Pseudo Code:
   Step 1  : perform or skip select/deselect slave
   Step 2  : perform or skip enable/disable transfer
   Step 3  : perform or skip enable/disable receive
*****************************************************************************/

void ConfigureSpiMaster(SpiMasterConfigOptions opt)
{

    if(enumNull == opt) return;

    if(opt & MaskBit_SelectSlave_Relevant)   (opt & MaskBit_SlaveSelect) ? SelectSlave() :
DeSelectSlave();
    if(opt & MaskBit_Trans_Relevant)(opt & MaskBit_Trans) ? EnableTrans():DisableTrans();
    if(opt & MaskBit_Recv_Relevant) (opt & MaskBit_Recv)  ? EnableRcv():DisableRcv();
```

```
}

/********* added by Carlton *************/
void SelectSlave(void)
{
  P2OUT &= ~(SPIS);    /* P2.5 = S = Low (selected) */
}

void DeSelectSlave(void)
{
  P2OUT |= SPIS;      /* P2.5 = S = High (deselected) */
}

void EnableTrans(void)
{
  P2OUT |= SPIHOLD;    /* P2.4 = HOLD = High */
}

void DisableTrans(void)
{
  P2OUT &= ~(SPIHOLD);   /* P2.4 = HOLD = Low */
}

void EnableRcv(void)
{
  P2OUT &= ~(SPIW);      /* P2.6 = W = Low (enabled) */
}

void DisableRcv(void)
{
  P2OUT |= SPIW;        /* P2.6 = W = High (disabled) */
}

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

/*****************************************************************************
Function:     Serialize(const CharStream* char_stream_send,
              CharStream* char_stream_recv,
              SpiMasterConfigOptions optBefore,
              SpiMasterConfigOptions optAfter
              )
Arguments:    char_stream_send, the char stream to be sent from the SPI master to
              the Flash memory, usually contains instruction, address, and data to be
              programmed.
              char_stream_recv, the char stream to be received from the Flash memory
              to the SPI master, usually contains data to be read from the memory.
              optBefore, configurations of the SPI master before any transfer/receive
              optAfter, configurations of the SPI after any transfer/receive
Return Values:TRUE
Description:  This function can be used to encapsulate a complete transfer/receive
              operation
Pseudo Code:
   Step 1  : perform pre-transfer configuration
   Step 2  : perform transfer/ receive
    Step 2-1: transfer ...
        (a typical process, it may vary with the specific CPU)
        Step 2-1-1:  check until the SPI master is available
        Step 2-1-2:  send the byte stream cycle after cycle. it usually involves:
                     a) checking until the transfer-data-register is ready
                     b) filling the register with a new byte
    Step 2-2: receive ...
        (a typical process, it may vary with the specific CPU)
        Step 2-2-1:  Execute ONE pre-read cycle to clear the receive-data-register.
        Step 2-2-2:  receive the byte stream cycle after cycle. it usually involves:
                     a) triggering a dummy cycle
                     b) checking until the transfer-data-register is ready(full)
                     c) reading the transfer-data-register
   Step 3  : perform post-transfer configuration
*****************************************************************************/
Bool Serialize(const CharStream* char_stream_send,
               CharStream* char_stream_recv,
```

```
              SpiMasterConfigOptions optBefore,
              SpiMasterConfigOptions optAfter
              )
{

    ST_uint32 i;
    ST_uint32 length;
    unsigned char* pChar;

    // Step 1  : perform pre-transfer configuration
    ConfigureSpiMaster(optBefore);

//    swDelay( 5, DMSEC );

    // Step 2  : perform transfer / receive
    // Step 2-1: transfer ...
    length = char_stream_send->length;
    pChar  = char_stream_send->pChar;

    // 2-1-1 Wait until SPI is available

//    swDelay( 5, DMSEC );

    // 2-1-2 send the byte stream cycle after cycle
    for(i = 0; i < length; ++i)
    {
      //dusart0.tx.status=FREE;
//      while (!(IFG1 & UTXIFG0 ));          // check until the transfer-data-register is
ready(not full)

      (*hsfm).rx.length = 0;
      (*hsfm).tx.length = 1;
      (*hsfm).tx.status = WORK;
      if( (*hsfm).port == SPI0 ){
        TXBUF0 = *(pChar++);          // fill the register with a new byte
      } else if( (*hsfm).port == SPI1 ){
        TXBUF1 = *(pChar++);          // fill the register with a new byte
      }

      while( (*hsfm).tx.status  != FREE );
    }

     swDelay( 5, DMSEC );

     // Step 2-2: receive ...
     // Step 2-2-1:  execute ONE pre-read cycle to clear the receive-data-register.
     (*hsfm).rx.length = 0;
     (*hsfm).rx.status = WORK;
     if( (*hsfm).port == SPI0 ){
        foo = RXBUF0;                        // read the transfer-data-register
     } else if( (*hsfm).port == SPI1 ){
        foo = RXBUF1;                        // read the transfer-data-register
     }

     // Step 2-2-2:  send the byte stream cycle after cycle.
     if(ptrNull != (int)char_stream_recv)                  // skip if no reception needed
     {
         length = char_stream_recv->length;
         pChar  = char_stream_recv->pChar;
         for(i = 0; i < length; ++i)
         {
             //dusart0.status = WRITE;
             //dusart0.status = READ;

            (*hsfm).tx.length = 1;
            (*hsfm).tx.status = WORK;
            (*hsfm).rx.length = 0;
            (*hsfm).rx.status = WORK;
            if( (*hsfm).port == SPI0 ){
              TXBUF0 = SPI_FLASH_INS_DUMMY;                    // triggering a dummy cycle
            } else if( (*hsfm).port == SPI1 ){
```

```
              TXBUF1 = SPI_FLASH_INS_DUMMY;                      // triggering a dummy cycle
            }

            while( (*hsfm).tx.status != FREE );
            while( (*hsfm).rx.status != FREE );
//             while (!(IFG1 & UTXIFG0 ));       // checking until the transfer-data-register is
ready
//             while (!(IFG1 & URXIFG0 ));

           *(pChar++) = (*hsfm).rx.buffer[ (*hsfm).rx.length-1 ];                          // read
the transfer-data-register
            (*hsfm).rx.length = 0;
         }
     }

    // Step 3  : perform post-transfer configuration
    ConfigureSpiMaster(optAfter);

    return TRUE;
}

/***************** Header File for STFL-I based Serial Flash Memory Driver *****

   Filename:    c2082.h
   Description: Header file for c2082.c
                Also consult the C file for more details.

                Please note that some necessary changes are made in favor of the
                SPI-specific communication property which slightly differs from
                the STFL-I Specification designed for parallel NOR Flash memories. The major
                differences from the SPECIFICATION OF THE STFL-I SOFTWARE DRIVER INTERFACE
               (Specification-STFL-I-V2-1a) are the following:
                - Flash Configuration Selection is not used.
                - BASE_ADDR is not used.
                - InstructionType enumuerations are re-formulated to use SPI Flash instructions.
                - CONFIGURATION CONSTANTS are fixed, with #define ins(A) not used.
                ...

   Version:     2.0
   Date:        20-07-2005
   Authors:    Tan Zhi, STMicroelectronics, Shanghai (China)
   Copyright (c) 2004-2005 STMicroelectronics.

   THE PRESENT SOFTWARE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH
   CODING INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A
   RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR
   CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH
   SOFTWARE AND/OR THE USE MADE BY CUSTOMERS OF THE CODING INFORMATION CONTAINED HEREIN
   IN CONNECTION WITH THEIR PRODUCTS.
*******************************************************************************

   Version History.
   Ver.   Date      Comments

   1.0   16/11/2004   Initial release
   2.0   20/07/2005   Add support for M25P32, M25P64
                      Add JEDEC ID support for M25P05A, M25P10A

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


/*************** User Change Area ******************************************

   The purpose of this section is to show how the SW Drivers can be customized
   according to the requirements of the hardware and Flash memory configurations.
   It is possible to choose the Flash memory start address, the CPU Bit depth, the number of
Flash
   chips, the hardware configuration and performance data (TimeOut Info).

   The options are listed and explained below:
```

********* Data Types *********
The source code defines hardware independent datatypes assuming that the
compiler implements the numerical types as

```
unsigned char    8 bits (defined as ST_uint8)
char             8 bits (defined as ST_sint8)
unsigned int     16 bits (defined as ST_uint16)
int              16 bits (defined as ST_sint16)
unsigned long    32 bits (defined as ST_uint32)
long             32 bits (defined as ST_sint32)
```

In case the compiler does not support the currently used numerical types,
they can be easily changed just once here in the user area of the headerfile.
The data types are consequently referenced in the source code as (u)ST_sint8,
(u)ST_sint16 and (u)ST_sint32. No other data types like 'CHAR','SHORT','INT','LONG'
are directly used in the code.

********* Flash Type *********
This driver supports the following Serial Flash memory Types

```
M25P05A       512Kb Serial Flash Memory                #define USE_M25P05A
M25P10A       1Mb Serial Flash Memory                  #define USE_M25P10A
M25P20        2Mb Serial Flash Memory                  #define USE_M25P20
M25P40        4Mb Serial Flash Memory                  #define USE_M25P40
M25P80        8Mb Serial Flash Memory                  #define USE_M25P80
M25P16        16Mb Serial Flash Memory                 #define USE_M25P16
M25P32        32Mb Serial Flash Memory                 #define USE_M25P32
M25P64        64Mb Serial Flash Memory                 #define USE_M25P64
```

********* Flash and Board Configuration *********
The driver also supports different configurations of the Flash chips
on the board. In each configuration a new data Type called
'uCPUBusType' is defined to match the current CPU data bus width.
This data type is then used for all accesses to the memory.

Because SPI interface communications are controlled by the
SPI master, which, in turn, is accessed by the CPU as an 8-bit data
buffer, the configuration is fixed for all cases.

********* TimeOut *********
There are timeouts implemented in the loops of the code, in order
to enable a timeout detection for operations that would otherwise never terminate.
There are two possibilities:

1) The ANSI Library functions declared in 'time.h' exist

   If the current compiler supports 'time.h' the define statement
   TIME_H_EXISTS should be activated. This makes sure that
   the performance of the current evaluation HW does not change
   the timeout settings.

2) or they are not available (COUNT_FOR_A_SECOND)

   If the current compiler does not support 'time.h', the define
   statement cannot be used. In this case the COUNT_FOR_A_SECOND
   value has to be defined so as to create a one-second delay.
   For example, if 100000 repetitions of a loop are
   needed to give a time delay of one second, then
   COUNT_FOR_A_SECOND should have the value 100000.

   Note: This delay is HW (Performance) dependent and therefore needs
   to be updated with every new HW.

   This driver has been tested with a certain configuration and other
   target platforms may have other performance data, therefore, the
   value may have to be changed.

   It is up to the user to implement this value to prevent the code
   from timing out too early and allow correct completion of the device

```
      operations.


   ********* Additional Routines *********
   The drivers also provide a subroutine which displays the full
   error message instead of just an error number.

   The define statement VERBOSE activates additional Routines.
   Currently it activates the FlashErrorStr() function

   No further changes should be necessary.

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

#ifndef __c2082__H__
#define __c2082__H__

#define DRIVER_VERSION_MAJOR 2
#define DRIVER_VERSION_MINOR 0


typedef unsigned char  ST_uint8; /* All HW dependent Basic Data Types */
typedef          char  ST_sint8;
typedef unsigned int   ST_uint16;
typedef          int   ST_sint16;
typedef unsigned long  ST_uint32;
typedef          long  ST_sint32;

/* With SYNCHRONOUS_IO defined, each function that sends an Instruction(e.g. PE)
   shall not return until the Flash memory finishes executing the Instruction
   or a pre-set timeout limit is reached. the pre-set timeout value is in
   accordance with the datasheet of each memory.

   To achieve Send-n-Forget feature, comment out this #define*/
#define SYNCHRONOUS_IO

#define USE_M25P64
/* Possible Values: USE_M25P05A
                                    USE_M25P10A
                                    USE_M25P20
                                    USE_M25P40
                                    USE_M25P80
                                    USE_M25P16
                                    USE_M25P32
                                    USE_M25P64
                    */

/*#define TIME_H_EXISTS*/  /* set this macro if C-library "time.h" is supported */
/* Possible Values: TIME_H_EXISTS
                    - no define - TIME_H_EXISTS */

#ifndef TIME_H_EXISTS
  #define COUNT_FOR_A_SECOND 0xFFFFFF   /* Timer Usage */
#endif

#define VERBOSE /* Activates additional Routines */
/* Currently the Error String Definition */

/********************** End of User Change Area ***************************/

/*****************************************************************************
Device Constants
*****************************************************************************/

#define MANUFACTURER_ST (0x20)    /* ST Manufacturer Identification is 0x20 */
#define MEMORYTYPE_M25Pxx (0x20)  /* JEDEC Memory Type for M25Pxx Identification is 0x20 */
#define ANY_ADDR (0x0)            /* Any address offset within the Flash Memory will do */

#ifdef USE_M25P05A                      /* The M25P05A device */
   #define USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE /* undefine this macro to read One-Byte
Signature*/
```

```
   #ifdef  USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        #define EXPECTED_DEVICE (0x2010) /* Preferred Device Identification: please refer to the
datasheet */
   #else
        #define EXPECTED_DEVICE (0x05)   /* Device Identification: please refer to the datasheet
*/
   #endif
   #define FLASH_SIZE (0x010000)         /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x0100)     /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x02)     /* Total device size in Sectors */
   #define FLASH_SMALLER_SECTOR_SIZE     /* Sector Size  = 256Kb*/
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x6)              /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P05A */

#ifdef USE_M25P10A                       /* The M25P10A device */
   #define USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE /* undefine this macro to read One-Byte
Signature*/
   #ifdef  USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        #define EXPECTED_DEVICE (0x2011) /* Preferred Device Identification: please refer to the
datasheet */
   #else
        #define EXPECTED_DEVICE (0x10)   /* Device Identification: please refer to the datasheet
*/
   #endif
   #define FLASH_SIZE (0x020000)         /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x0200)     /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x04)     /* Total device size in Sectors */
   #define FLASH_SMALLER_SECTOR_SIZE     /* Sector Size  = 256Kb*/
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x6)              /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P10A */

#ifdef USE_M25P20                        /* The M25P20 device */
   #define EXPECTED_DEVICE (0x11)        /* Device Identification for the M25P20 */
   #define FLASH_SIZE (0x040000)         /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x0400)     /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x04)     /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x6)              /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P20 */

#ifdef USE_M25P40                        /* The M25P40 device */
   #define EXPECTED_DEVICE (0x12)        /* Device Identification for the M25P40 */
   #define FLASH_SIZE (0x080000)         /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x0800)     /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x08)     /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x03)             /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P40 */

#ifdef USE_M25P80                        /* The M25P80 device */
   #define EXPECTED_DEVICE (0x13)        /* Device Identification for the M25P80 */
   #define FLASH_SIZE (0x0100000)        /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x01000)    /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x10)     /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x14)             /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P80 */

#ifdef USE_M25P16                        /* The M25P16 device */
```

```
   #define USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE /* undefine this macro to read One-Byte
Signature*/
   #ifdef  USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        #define EXPECTED_DEVICE (0x2015) /* Device Identification for the USE_M25P16 */
   #else
        #define EXPECTED_DEVICE (0x14)   /* Device Identification for the USE_M25P16 */
   #endif
   #define FLASH_SIZE (0x0200000)        /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x02000)    /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x20)     /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100 /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                   /* Minimum Write Access */
   #define BE_TIMEOUT (0x46)             /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P16 */

#ifdef USE_M25P32                        /* The USE_M25P32 device */
   #define USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE /* undefine this macro to read One-Byte
Signature*/
   #ifdef  USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        #define EXPECTED_DEVICE (0x2016)    /* Device Identification for the USE_M25P32 */
   #else
        #define EXPECTED_DEVICE (0x15)      /* Device Identification for the USE_M25P32 */
   #endif
   #define FLASH_SIZE (0x0400000)           /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x04000)       /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x40)        /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100    /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                      /* Minimum Write Access */
   #define BE_TIMEOUT (0x80)                /* Timeout in seconds suggested for Bulk Erase
Operation*/
#endif /* USE_M25P32 */


#ifdef USE_M25P64                        /* The USE_M25P64 device */
   #define USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE /* undefine this macro to read One-Byte
Signature*/
   #ifdef  USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        #define EXPECTED_DEVICE (0x2017)    /* Device Identification for the USE_M25P64 */
   #else
        #define EXPECTED_DEVICE (0x16)      /* Device Identification for the USE_M25P64 */
   #endif
   #define FLASH_SIZE (0x0800000)           /* Total device size in Bytes */
   #define FLASH_PAGE_COUNT (0x08000)       /* Total device size in Pages */
   #define FLASH_SECTOR_COUNT (0x80)        /* Total device size in Sectors */
   #define FLASH_WRITE_BUFFER_SIZE 0x100    /* Write Buffer = 256 bytes */
   #define FLASH_MWA 1                      /* Minimum Write Access */
   #define BE_TIMEOUT (0x160)               /* Timeout in seconds suggested for Bulk Erase
Operation*/
   #define NO_DEEP_POWER_DOWN_SUPPORT       /* No support for Deep Power-down feature*/
#endif /* USE_M25P64 */
/*****************************************************************************
     DERIVED DATATYPES
*****************************************************************************/
/******** InstructionsCode ********/
#define SPI_FLASH_INS_DUMMY 0xAA            // dummy byte
enum
{
       //Instruction set
       SPI_FLASH_INS_WREN        = 0x06,           // write enable
       SPI_FLASH_INS_WRDI        = 0x04,           // write disable
       SPI_FLASH_INS_RDSR        = 0x05,           // read status register
       SPI_FLASH_INS_WRSR        = 0x01,           // write status register
       SPI_FLASH_INS_READ        = 0x03,           // read data bytes
       SPI_FLASH_INS_FAST_READ   = 0x0B,           // read data bytes at higher speed
       SPI_FLASH_INS_PP          = 0x02,           // page program
       SPI_FLASH_INS_SE          = 0xD8,           // sector erase

       #ifndef NO_DEEP_POWER_DOWN_SUPPORT
       SPI_FLASH_INS_RES         = 0xAB,           // release from deep power-down
       SPI_FLASH_INS_DP          = 0xB9,           // deep power-down
```

```
        #endif

    #ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
        SPI_FLASH_INS_RDID        = 0x9F,            // read identification
    #endif

        SPI_FLASH_INS_BE          = 0xC7             // bulk erase
};


/******** InstructionsType ********/

typedef enum {
   WriteEnable,
   WriteDisable,
   ReadDeviceIdentification,
   #ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
   ReadManufacturerIdentification,
   #endif
   ReadStatusRegister,
   WriteStatusRegister,
   Read,
   FastRead,
   PageProgram,
   SectorErase,
   BulkErase,

   #ifndef NO_DEEP_POWER_DOWN_SUPPORT
   DeepPowerDown,
   ReleaseFromDeepPowerDown,
   #endif

   Program
} InstructionType;

/******** ReturnType ********/

typedef enum {
   Flash_AddressInvalid,
   Flash_MemoryOverflow,
   Flash_PageEraseFailed,
   Flash_PageNrInvalid,
   Flash_SectorNrInvalid,
   Flash_FunctionNotSupported,
   Flash_NoInformationAvailable,
   Flash_OperationOngoing,
   Flash_OperationTimeOut,
   Flash_ProgramFailed,
   Flash_WrongType,
   Flash_Success
} ReturnType;

/******** SectorType ********/

typedef ST_uint8 uSectorType;

/******** PageType ********/

typedef ST_uint16 uPageType;

/******** AddrType ********/

typedef ST_uint32 uAddrType;

/******** ParameterType ********/

typedef union {

    /**** WriteEnable has no parameters ****/

    /**** WriteDisable has no parameters ****/
```

```
/**** ReadDeviceIdentification Parameters ****/
struct {
  ST_uint16 ucDeviceIdentification;
} ReadDeviceIdentification;

/**** ReadManufacturerIdentification Parameters ****/
struct {
  ST_uint8 ucManufacturerIdentification;
} ReadManufacturerIdentification;

/**** ReadStatusRegister Parameters ****/
struct {
  ST_uint8 ucStatusRegister;
} ReadStatusRegister;

/**** WriteStatusRegister Parameters ****/
struct {
  ST_uint8 ucStatusRegister;
} WriteStatusRegister;

/**** Read Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsToRead;
  void *pArray;
} Read;

/**** FastRead Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsToRead;
  void *pArray;
} FastRead;

/**** PageWrite Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsInArray;
  void *pArray;
} PageWrite;

/**** PageProgram Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsInArray;
  void *pArray;
} PageProgram;

/**** PageErase Parameters ****/
struct {
  uPageType upgPageNr;
} PageErase;

/**** SectorErase Parameters ****/
struct {
  uSectorType ustSectorNr;
} SectorErase;

/**** Write Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsInArray;
  void *pArray;
} Write;

/**** Program Parameters ****/
struct {
  uAddrType udAddr;
  ST_uint32 udNrOfElementsInArray;
  void *pArray;
```

```
    } Program;

} ParameterType;


/*****************************************************************************
    Standard functions
*****************************************************************************/
  ReturnType  Flash( InstructionType insInstruction, ParameterType *fp );
  ReturnType  FlashWriteEnable( void );
  ReturnType  FlashWriteDisable( void );
  ReturnType  FlashReadDeviceIdentification( ST_uint16 *uwpDeviceIdentification);
#ifdef USE_JEDEC_STANDARD_TWO_BYTE_SIGNATURE
  ReturnType  FlashReadManufacturerIdentification( ST_uint8 *ucpManufacturerIdentification);
#endif
  ReturnType  FlashReadStatusRegister( ST_uint8 *ucpStatusRegister);
  ReturnType  FlashWriteStatusRegister( ST_uint8 ucStatusRegister);
  ReturnType  FlashRead( uAddrType udAddr, ST_uint8 *ucpElements, ST_uint32
udNrOfElementsToRead);
  ReturnType  FlashFastRead( uAddrType udAddr, ST_uint8 *ucpElements, ST_uint32
udNrOfElementsToRead);
  ReturnType  FlashPageProgram( uAddrType udAddr, ST_uint8 *pArray, ST_uint16
udNrOfElementsInArray );
  ReturnType  FlashSectorErase( uSectorType uscSectorNr );
  ReturnType  FlashBulkErase( void );
#ifndef NO_DEEP_POWER_DOWN_SUPPORT
  ReturnType  FlashDeepPowerDown( void );
  ReturnType  FlashReleaseFromDeepPowerDown( void );
#endif
  ReturnType  FlashProgram( ST_uint32 udAddr, ST_uint8 *pArray , ST_uint32
udNrOfElementsInArray);

/*****************************************************************************
    Utility functions
*****************************************************************************/
#ifdef VERBOSE
   ST_sint8 *FlashErrorStr( ReturnType rErrNum );
#endif

  ReturnType  FlashTimeOut( ST_uint32 udSeconds );

/*****************************************************************************
List of Errors and Return values, Explanations and Help.
*****************************************************************************

Error Name:   Flash_AddressInvalid
Description:  The address given is out of the range of the Flash device.
Solution:     Check whether the address is in the valid range of the
              Flash device.
*****************************************************************************

Error Name:   Flash_PageEraseFailed
Description:  The Page erase Instruction did not complete successfully.
Solution:     Try to erase the Page again. If this fails once more, the device
              may be faulty and need to be replaced.
*****************************************************************************

Error Name:   Flash_PageNrInvalid
Note:         The Flash memory is not at fault.
Description:  A Page has been selected (Parameter), which is not
              within the valid range. Valid Page numbers are from 0 to
              FLASH_PAGE_COUNT - 1.
Solution:     Check that the Page number given is in the valid range.
*****************************************************************************

Error Name:   Flash_SectorNrInvalid
Note:         The Flash memory is not at fault.
Description:  A Sector has been selected (Parameter), which is not
              within the valid range. Valid Page numbers are from 0 to
              FLASH_SECTOR_COUNT - 1.
Solution:     Check that the Sector number given is in the valid range.
```

```
*****************************************************************************

Return Name:  Flash_FunctionNotSupported
Description:  The user has attempted to make use of a functionality not
              available on this Fash device (and thus not provided by the
              software drivers).
Solution:     This may happen after changing Flash SW Drivers in existing
              environments. For example an application tries to use a
              functionality which is no longer provided with the new device.
*****************************************************************************

Return Name:  Flash_NoInformationAvailable
Description:  The system cannot give any additional information about the error.
Solution:     None
*****************************************************************************

Error Name:   Flash_OperationOngoing
Description:  This message is one of two messages that are given by the TimeOut
              subroutine. It means that the ongoing Flash operation is still within
              the defined time frame.
*****************************************************************************

Error Name:   Flash_OperationTimeOut
Description:  The Program/Erase Controller algorithm could not finish an
              operation successfully. It should have set bit 7 of the Status
              Register from 0 to 1, but that did not happen within a predetermined
              time. The program execution was therefore cancelled by a
              timeout. This may be because the device is damaged.
Solution:     Try the previous Instruction again. If it fails a second time then it
              is likely that the device will need to be replaced.
*****************************************************************************

Error Name:   Flash_ProgramFailed
Description:  The value that should be programmed has not been written correctly
              to the Flash memory.
Solutions:    Make sure that the Page which is supposed to receive the value
              was erased successfully before programming. Try to erase the Page and
              to program the value again. If it fails again then the device may
              be faulty.
*****************************************************************************

Error Name:   Flash_WrongType
Description:  This message appears if the Manufacture and Device Identifications read from
              the current Flash device do not match the expected identifier
              codes. This means that the source code is not explicitely written for
              the currently used Flash chip. It may work, but the operation cannot be
              guaranteed.
Solutions:    Use a different Flash chip with the target hardware or contact
              STMicroelectronics for a different source code library.
*****************************************************************************

Return Name:  Flash_Success
Description:  This value indicates that the Flash memory Instruction was executed
              correctly.
*****************************************************************************/

/*****************************************************************************
    External variable declaration
*****************************************************************************/

// none in this version of the release

/*****************************************************************************
Flash Status Register Definitions (see Datasheet)
*****************************************************************************/
enum
{
       SPI_FLASH_SRWD = 0x80,                    // Status Register Write Protect
       SPI_FLASH_BP2  = 0x10,                    // Block Protect Bit2
       SPI_FLASH_BP1  = 0x08,                    // Block Protect Bit1
       SPI_FLASH_BP0  = 0x04,                    // Block Protect Bit0
```

```
       SPI_FLASH_WEL  = 0x02,                      // write enable latch
       SPI_FLASH_WIP  = 0x01                      // write/program/erase in progress indicator
};

/*****************************************************************************
Specific Function Prototypes
*****************************************************************************/
typedef unsigned char BOOL;

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

BOOL IsFlashBusy();

/*****************************************************************************
List of Specific Errors and Return values, Explanations and Help.
*****************************************************************************

// none in this version of the release
*****************************************************************************/

#endif /* __c2082__H__  */
/* In order to avoid a repeated usage of the header file */

/*****************************************************************************
     End of c2082.h
*****************************************************************************/

/* MQP header */
/* some general constants and functions */

#ifndef MQP_H
#define MQP_H

char char256[256];  /* sfmbuffer */
char char180[180];  /* GPS RX buffer, USB TX buffer */
char char125[125];  /* SFM RX buffer */
char char32[32];    /* SFM TX buffer */
char char8[8];      /* scratch value */
char char1[1];      /* USB RX buffer */

int sfmreadpos;

#endif

/* mqpadc.h */
/***************** header file for ADC interface ********************/
#ifndef MQPADC_H
#define MQPADC_H

#define INTMAX 0xFFFF

#define XAXIS ADC12MEM0
#define YAXIS ADC12MEM1
#define ZAXIS ADC12MEM2

typedef unsigned int AXLint;

/* accelerometer axis */
typedef struct _AXLaxis{
  AXLint value;  /* the latest value read from the ADC */
  AXLint max;    /* the maximum value read from the ADC */
  AXLint min;    /* the minimum value read from the ADC */
}AXLaxis;

/* triple-axis accelerometer */
```

```
typedef struct _AXL3{
  AXLaxis x;    /* X axis */
  AXLaxis y;    /* Y axis */
  AXLaxis z;    /* Z axis */
}AXL3;

void axlInit(void);
void axlReset(void);
void axlConvert(void);
void axlRead(void);

AXL3 axl;   /* the accelerometer */

/******************** init_adc() ************************
 * function to initialize the ADC channels
 * channel setup:
 *  0: Accelerometer X Axis
 *  1: Accelerometer Y Axis
 *  2: Accelerometer Z Axis
 */
void axlInit(void)
{
       /* set up conversion clocks, turn adc on, use multiple samples */
       ADC12CTL0 = SHT0_6 + ADC12ON + MSC;
       /* set up for samping signal select and single sequence sample */
       ADC12CTL1 = SHP + CONSEQ_1;

       /* set up channel zero for input, use AVcc with respect to Avss */
       ADC12MCTL0 = INCH_0 + SREF_0;
       /* set up channel one for input, use AVcc with respect to Avss */
       ADC12MCTL1 = INCH_1 + SREF_0;
       /* set up channel two for input, use AVcc with respect to Avss */
       ADC12MCTL2 = INCH_2 + SREF_0 + EOS;

       /* enable conversion */
       ADC12CTL0 |= ENC;

        /* clear accelerometer structure */
        axlReset();
}

/******************** axlReset() ***************************/
/* function to clear accelerometer structure */
void axlReset(void)
{
  /* x axis */
  axl.x.value = 0;
  axl.x.max   = 0;
  axl.x.min   = INTMAX;

  /* y axis */
  axl.y.value = 0;
  axl.y.max   = 0;
  axl.y.min   = INTMAX;

  /* z axis */
  axl.z.value = 0;
  axl.z.max   = 0;
  axl.z.min   = INTMAX;
}

/******************** axlConvert() ***************************/
/* function to read a conversion from the ADC into ADC memory */
/* post: after being called, the ADC12MEM# has the new conversion in it */
void axlConvert(void)
{
   ADC12CTL0 |= ADC12SC;             /* start up a conversion */
   while( ADC12CTL0 & ADC12SC );     /* hang until conversion is done */
}

/******************** axlRead() ***************************/
```

```
/* function to get current ADC values into accelerometer structure */
void axlRead(void)
{
  /* do a conversion */
  axlConvert();

  /* store in structure */
  axl.x.value = XAXIS;
  axl.y.value = YAXIS;
  axl.z.value = ZAXIS;

  /* check for new max */
  if( XAXIS > axl.x.max ){ axl.x.max = XAXIS; }
  if( YAXIS > axl.y.max ){ axl.y.max = YAXIS; }
  if( ZAXIS > axl.z.max ){ axl.z.max = ZAXIS; }

  /* check for new min */
  if( XAXIS < axl.x.min ){ axl.x.min = XAXIS; }
  if( YAXIS < axl.y.min ){ axl.y.min = YAXIS; }
  if( ZAXIS < axl.z.min ){ axl.z.min = ZAXIS; }
}
#endif

/* mqpdatalog.h */
/***************** header file for datalogging mode *********************/

#ifndef MQPDATALOG_H
#define MQPDATALOG_H

#include <stdio.h>

#include "mqptimer.h"
#include "mqpgps.h"
#include "mqpsfm.h"
#include "mqpaxl.h"

void datalogMode(void);
void datalogInit(void);
void datalogLoop(void);
void datalogKill(void);

void datalogMode(void)
{
  /* initialize */
  datalogInit();

  /* run loop */
  while( dipswRead() == DATALOGMODE ){
    datalogLoop();
  }

  /* kill */
  datalogKill();
}

void datalogInit(void)
{
  _DINT();

/* by default, each interrupt is disabled */
  /* set up USART0 for UART for GPS */
  gpsInit();
  gpsPost();

  /* set up USART1 for SPI for SFM */
  sfmInit1();
  if( sfmPost() != Flash_Success ){
    lcdClear();
    lcdPrints( "Bad Mem!\nReset..." );

    _EINT();
```

```
    hwDelay( &timerA, 1, 0x7FFF );  /* set up timer A to wait for .5 second */
    while( timerA.status != DONE );
    exit(1);                        /* die */
  }

  /* locate first available spot in SFM */
  sfmpos = 0;
  _EINT();
  sfmEnable();
  for( i=0; i<FLASH_SIZE; i++ ){
    fp.Read.udAddr = sfmpos;
    fp.Read.udNrOfElementsToRead = 1;
    fp.Read.pArray = (void*)char1;
    rRetVal=Flash( Read, &fp );

    /* check - is it 0xFF? */
    if( char1[0] == 0xFF ){
      break;
    }

    sfmpos++;
  }
  sfmDisable();
  _DINT();

  lcdClear();
  lcdPrints( "Start:\n" );
  sprintf( char8, "%i", sfmpos );
  lcdPrints( char8 );
   _EINT();
  hwDelay( &timerA, 3, 0x7FFF );  /* set up timer A to wait for 1.5 second */
  while( timerA.status != DONE );\
  _DINT();

  /* set up ADC0-3 for AXL */
  axlInit();

  /* clear LCD */
  lcdClear();
}

char buf8[9];
char *token;
char axls[30];

char utc[7];
char latitude[9];
char ns[2];
char longitude[10];
char ew[2];

int inumSats;

void datalogLoop(void)
{
  /* set up timer A to wait for 2.5 seconds */
  hwDelay( &timerA, 3, 0x7FFF );
  _EINT();

  /* enable GPS interrupts so we can read from it */
  gpsEnable();

  /* reset accelerometer values to default */
  axlReset();

  /* loop until timer is up or GPS data received (do at least once) */
  do{
    axlRead();    /* sample ADCs */
  }while( timerA.status != DONE );

  /* disable GPS interrupts */
```

```
  gpsDisable();

  token      = NULL;

  /* has data been received? */
  if( (*hgps).rx.length > 0 ){
    token = (*hgps).rx.buffer;
//    strcpy( token, "$GPGGA,151530.05,4216.308,N,71482.544,W,1,05,v.v,w.w,M,x.x,M,y.y,zzzz*hh"
);

    /* find type of string */
    token = strpbrk( token, "$" );
    strncpy( stringType, token+1, 5 );
    stringType[5] = '\0';             /* null terminate */

    /* hhmmss,llll.lll,a,nnnnn.nnn,b,XMIN,XMAX,YMIN,YMAX,ZMIN,ZMAX; */

    /* GPGGA string? */
    if( strncmp( stringType, "GPGGA", 5 ) == 0 ){
      /* get number of satelites */
      token = strpbrk( token+1, "," );  /* remove GPGGA part */

      token++;                          /* skip past comma */
      strncpy( utc, token, 6 );         /* utc (minus the ".ss" part) */
      utc[6] = '\0';                    /* null terminate */

      token+=10;                        /* skip past ".ss" and comma */
      strncpy( latitude, token, 8 );    /* latitude */
      latitude[8] = '\0';               /* null terminate */

      token += 9;                       /* skip past comma */
      strncpy( ns, token, 1 );          /* N/S */
      ns[1] = '\0';                     /* null terminate */

      token += 2;                       /* skip past comma */
      strncpy( longitude, token, 9 );   /* longitude */
      longitude[9] = '\0';              /* null terminate */

      token += 10;                      /* skip past comma */
      strncpy( ew, token, 1 );        /* E/W */
      ew[1] = '\0';                     /* null terminate */

      token += 4;                       /* skip past "quality" and comma */
      strncpy( numSats, token, 2 );     /* number of satelites */
      numSats[2] = '\0';                /* null terminate */
    }

    lcdLine1();
    lcdPrints( "S:" );
    lcdPrints( numSats );

    /* only record if number of active satelites is > 3 */
    if( atoi( numSats) > 3 ){
      lcdPrints( "\nR" );

      /* construct string */
      sprintf( (*hgps).rx.buffer, "%s,%s,%s,%s,%s,%d,%d,%d,%d,%d,%d;", \
        utc, latitude, ns, longitude, ew, \
        axl.x.max, axl.x.min, axl.y.max, axl.y.min, axl.z.max, axl.z.min );

      (*hgps).rx.length = strlen( (*hgps).rx.buffer );

      sfmEnable();

      /* write some data */
      fp.Program.udAddr = sfmpos;
      fp.Program.udNrOfElementsInArray = (*hgps).rx.length;
      fp.Program.pArray = (void*)(*hgps).rx.buffer;
      rRetVal = Flash( Program, &fp );

      sfmpos += (*hgps).rx.length;
```

```
      sfmDisable();
    } else {  /* otherwise, re-initialize */
      _DINT();
      gpsInit();
    }
  }
}

void datalogKill(void)
{
  /* flush SFM buffer to SFM */
  /** sfmFlush() **/

  /* free up buffers, nullify pointers */
  dusart0.tx.buffer = NULL;
  dusart0.rx.buffer = NULL;
  dusart1.tx.buffer = NULL;
  dusart1.rx.buffer = NULL;
}
#endif

/* mqpdebug.h */
/***************** header file for debug mode ********************/

#ifndef MQPDEBUG_H
#define MQPDEBUG_H

void debugMode(void);
void debugInit(void);
void debugLoop(void);
void debugKill(void);

void debugMode(void)
{
  /* initialize */
  debugInit();

  /* run loop */
  while( dipswRead() == DEBUGMODE ){
    debugLoop();
  }

  debugKill();
}

void debugInit(void)
{
  _DINT();

}

void debugLoop(void)
{
}

void debugKill(void)
{
  /* free up buffers, nullify pointers */
  dusart0.tx.buffer = NULL;
  dusart0.rx.buffer = NULL;
  dusart1.tx.buffer = NULL;
  dusart1.rx.buffer = NULL;
}
#endif

/* mqpdipsw.h */
/***************** header file for DIPSW interface ********************/

#ifndef MQPDIPSW_H
#define MQPDIPSW_H
```

```
void dipswInit(void);
char dipswRead(void);

#define DIPSWDIR  P1DIR   /* dipswitch directional port */
#define DIPSWSEL  P1SEL   /* dipswitch control port */
#define DIPSWIN   P1IN    /* dipswitch input port */
#define DIPSWBITS (BIT3|BIT2|BIT1|BIT0) /* dipswitch bits used */
#define DIPSWSHR  0   /* amount to shift read in bits to the right  */

/* modes of operation */
#ifdef MQPPROTO
  #define DATALOGMODE   0x00  /* 1111 */
  #define DOWNLOADMODE  0x01  /* 1110 */
  #define DEBUGMODE     0x03  /* 0111 */
  #define DELETEMODE    0x0C
#else
  #define DATALOGMODE  0x0F  /* 1111 */
  #define DOWNLOADMODE 0x0E  /* 1110 */
  #define DEBUGMODE    0x0C  /* 1100 */
  #define DELETEMODE   0x0D  /* 1101 */
#endif

/******************** init_dipsw() ************************
 * function to initialize the dipswitch
 */
void dipswInit(void)
{
  DIPSWDIR &= ~(DIPSWBITS);   /* set dipswitch port to input direction */
  DIPSWSEL &= ~(DIPSWBITS);   /* set dipswitch port I/O option */
}

/******************** dipswRead() ************************
 * function to read from the dipswitch
 */
char dipswRead(void)
{
  char input;

  /* read in from dipswitch, mask off unneeded bits */
  input = DIPSWIN & DIPSWBITS;

  /* shift bits to the right to ignore unused bits */
  input >>= DIPSWSHR;

  return( input );  /* return what was read in */
}

#endif

/* mqpdownload.h */
/***************** header file for download mode ********************/

#ifndef MQPDOWNLOAD_H
#define MQPDOWNLOAD_H

#include "mqptimer.h"
#include "mqpusb.h"
#include "mqpsfm.h"

void downloadMode(void);
void downloadInit(void);
void downloadLoop(void);
void downloadKill(void);

void downloadMode(void)
{
  /* initialize */
  downloadInit();

  /* run loop */
  while( dipswRead() == DOWNLOADMODE ){
```

```
    downloadLoop();
  }

  downloadKill();
}


void downloadInit(void)
{
  _DINT();

  /* set up USB for UART1 */
  usbInit();
  usbPost();

  /* print menu to USB */
  usbPrintMenu();

  /* set up SFM for SPI0 */
  sfmInit0();
  if( sfmPost() != Flash_Success ){
    lcdClear();
    lcdPrints( "Bad Mem!\nReset..." );

    _EINT();
    hwDelay( &timerA, 1, 0x7FFF );  /* set up timer A to wait for .5 second */
    while( timerA.status != DONE );
    exit(1);
  }

    sfmreadpos = 0;
}

void downloadLoop(void)
{
  /* set up timer A to wait for 2.5 seconds */
  hwDelay( &timerA, 3, 0x7FFF );
  usbEnable();
  (*husb).rx.status=WORK;
  _EINT();

  /* wait for USB command or time out */
  while( timerA.status != DONE && (*husb).rx.status != FREE );

  /* if given command to download memory */
  if( (*husb).rx.status == FREE )
  {
    /* look at latest character */
    switch( (*husb).rx.buffer[ (*husb).rx.length ] )
    {
    case 'C':
    case 'c': /* clear memory */
      usbPrint( "Clearing memory.\n\r" );
      sfmErase();
      usbPrint( "Memory cleared.\n\r" );
      break;
    case 'D':
    case 'd': /* download memory */
      usbPrint( "Downloading memory.\n\r" );
      usbDump();
      usbPrint( "Memory downloaded.\n\r" );
      break;
    case 'H':
    case 'h': /* help - print menu again */
      usbPrintMenu();
      break;
    default:
      usbPrint( "Not a valid command.\n\r" );
    }
  }
```

```
  usbDisable();
  _DINT();
}

void downloadKill(void)
{
  /* free up buffers, nullify pointers */
  dusart0.tx.buffer = NULL;
  dusart0.rx.buffer = NULL;
  dusart1.tx.buffer = NULL;
  dusart1.rx.buffer = NULL;
}
#endif

/* mqpgps.h */
/***************** header file for GPS ********************/

#ifndef MQPGPS_H
#define MQPGPS_H

void gpsInit(void);
void gpsEnable(void);
void gpsDisable(void);
void gpsRecv(void);
void gpsSend(void);
void gpsPost(void);

char stringType[6];
char numSats[3];

#include "mqpusart.h"
#include <stdlib.h>

/******************** gpsInit() *************************
 * function to initialize USART 0
 * post: USART 0 set up for 4800 baud 8-none-1 connection
 *    with no flow control and interrupts are enabled
 */
void gpsInit(void){
  P3SEL |= (BIT4|BIT5);   /* P3.4,5 = USART0 TXD/RXD */
  UCTL0 |= CHAR;          /* 8-bit character, SWRST=1 */
  UTCTL0 |= SSEL0;        /* UCLK = ACLK */

  /* 4800 baud */
  UBR00 = 0x06;
  UBR10 = 0x00;
  UMCTL0 = 0x77;

  UCTL0 &= ~SWRST;          /* initialize USART0 state machine */
  IE1 |= URXIE0 + UTXIE0;   /* enable USART0 RX/TX interrupts */
  IFG1 &= ~UTXIFG0;         /* clear inital flag on POR */

  dusart0.tx.buffer = NULL;
  dusart0.tx.index  = 0;
  dusart0.tx.length = 0;
  dusart0.tx.size   = 0;
  dusart0.tx.status = FREE;

  dusart0.rx.buffer = char180;
  dusart0.rx.length = 0;
  dusart0.rx.size   = 180;
  dusart0.rx.status = FREE;
  dusart0.rx.stopbyte=0x0A; /* end of sentence */

  dusart0.port      = UART0;
  dusart0.recv      = gpsRecv;
  dusart0.send      = gpsSend;

  hgps = &dusart0;

//  numSats = NULL;
```

```
}

/******************** gpsEnable() ************************
 * function to enable interrupts for GPS
 */
void gpsEnable(void)
{
    ME1 |= URXE0;
    swDelay( 500, DMSEC );
}

/******************** gpsDisable() ************************
 * function to disable interrupts for GPS
 */
void gpsDisable(void)
{
  ME1 &= ~(URXE0);
}

/******************** gpsRecv() ************************
 * function to recieve characters from UART and buffer them
 * from PORT 2 of GPS using NMEA interface
 */
void gpsRecv(void)
{
  if( (*hgps).rx.length<(*hgps).rx.size ){
    (*hgps).rx.buffer[ (*hgps).rx.length ] = RXBUF0;
    (*hgps).rx.length++;
  } else {
    (*hgps).rx.status=FREE;   /* an overflow has occured */
  }
  if( RXBUF0 == (*hgps).rx.stopbyte ){ /* stop byte reached? */
    (*hgps).rx.status=FREE;
  }
}

/******************** gpsSend() ************************
 * function to transmit next character in buffer via UART
 * to PORT 1 of GPS using TSIP interface
 */
void gpsSend(void)
{
}

void gpsPost(void)
{
}

#endif

/* mqpio.h */
/***************** header file for IO structures ********************/

#ifndef MQPIOBUFFER_H
#define MQPIOBUFFER_H

enum IOstate{
  FREE, WORK
};

typedef struct _IObuffer{
  char* buffer;         /* the buffer */
  int length;           /* length of used buffer */
  int size;             /* max size of buffer */
  enum IOstate status;  /* current status */
  union{
    int index;      /* position in buffer for transmitting */
    char stopbyte;  /* stop byte for recieving */
  };
} IObuffer;
```

```
enum IOport {
    UART0, UART1, SPI0, SPI1
};

typedef struct _IOdevice{
  IObuffer rx;            /* reception buffer */
  IObuffer tx;            /* transmission buffer */
  void (*recv)(void);     /* receive function pointer */
  void (*send)(void);     /* send function pointer */
  enum IOport port;       /* port this device is attached to */
} IOdevice, *IOhandle;

#endif

/* mqplcd.h */
/***************** header file for LCD interface ******************/
/*
  Author: Carlton C. Stedman II
  Last update: 2/19/2007
*/

#ifndef MQPLCD_H
#define MQPLCD_H

void lcdInit(void);
void lcdClock(void);
void lcdWrite(unsigned char);
void lcdCmd(unsigned char);
void lcdPrint(char);
void lcdPrints(char* );

#include "mqptimer.h"
#include <string.h>

/* LCD Control Port defines */
#define LCDCTRLDIR  P4DIR
#define LCDCTRLSEL  P4SEL
#define LCDCTRLOUT  P4OUT
#define LCDEN       BIT1    /* LCD enable */
#define LCDRW       BIT2    /* LCD R/W */
#define LCDRS       BIT3    /* LCD RS */

/* LCD Data Port defines */
#define LCDDIR      P4DIR
#define LCDSEL      P4SEL
#define LCDOUT      P4OUT

#define LCDCOLS 16
#define LCDROWS 2

/*********************************************************************/
/********************** LCD using 8-Bit Interface ********************/
/*********************************************************************/
#ifdef LCD8BIT
#define lcdLine1() { lcdCmd( 0x80 ); }
#define lcdLine2() { lcdCmd( 0xC0 ); }
#define lcdClear() { lcdCmd( 0x01 ); lcdLine1(); }

/******************** init_lcd() *************************/
void lcdInit(void)
{
  /* wait for 20ms */
  swDelay( 20, DMSEC );

  /* function set: 8-bit interface */
  lcdCmd( 0x30 );

  /* wait for 5ms */
  swDelay( 5, DMSEC );

  /* function set: 8-bit interface */
```

```
  lcdCmd( 0x30 );

  /* wait for 200us */
  swDelay( 2, DHUSEC );

  /* function set: 8-bit interface */
  lcdCmd( 0x30 );

  /* function set: 8-bit interface, two-line display with 5x8 char font */
  lcdCmd( 0x38 );

  /* turn display off */
  lcdCmd( 0x08 );

  /* turn display clear */
  lcdClear();

  /* set for entry mode */
  lcdCmd( 0x06 );

  /* turn on display, cursor off */
  lcdCmd( 0x0C );
}

/****************** lcdClock() ************************/
/* function to flash the enable on the LCD */
void lcdClock(void)
{
  LCDCTRLDIR |= (LCDEN);
  LCDCTRLSEL &= ~(LCDEN);
  LCDCTRLOUT &= ~(LCDEN);       /* clear enable */

  swDelay( 4, DMSEC );          /* wait 4 ms */
  LCDCTRLOUT |= (LCDEN);        /* set enable */

  swDelay( 2, DMSEC );          /* wait 2 ms */
  LCDCTRLOUT &= ~(LCDEN);       /* clear enable */
}

/******************* lcdWrite() ************************/
/* function to write data onto the data lines of the LCD */
void lcdWrite( unsigned char databus )
{
  /* set P4.0-4.7 to output direction */
  LCDDIR |= (BIT0|BIT1|BIT2|BIT3|BIT4|BIT5|BIT6|BIT7);
  /* set P4.0-4.7 I/O option */
  LCDSEL &= ~(BIT0|BIT1|BIT2|BIT3|BIT4|BIT5|BIT6|BIT7);
  /* P4.0-4.7 output = databus */
  LCDOUT = databus;

  lcdClock();
}

/******************* lcdPrint() ************************/
/* function to print a character at the cursor on the LCD */
void lcdPrint( char databus )
{
  LCDCTRLDIR |= (LCDRW|LCDRS);              /* set P2.2-2.3 to output direction */
  LCDCTRLSEL &= ~(LCDRW|LCDRS);             /* P2.2-2.3 I/O option */
  LCDCTRLOUT &= ~(LCDRW);                   /* clear R/W flag on P2.2 */
  LCDCTRLOUT |= (LCDRS);                    /* set RS flag on P2.3 */

  lcdWrite( databus );
}

#endif

/*************************************************************************/
/*********************** LCD using 4-Bit Interface *******************/
/*************************************************************************/
#ifdef LCD4BIT
```

```
#define lcdLine1() { lcdCmd( 0x80 ); lcdCmd( 0x00 ); }
#define lcdLine2() { lcdCmd( 0xC0 ); lcdCmd( 0x00 ); }
#define lcdClear() { lcdCmd( 0x00 ); lcdCmd( 0x10 ); lcdLine1(); }

void lcdInit(void)
{
  /* wait for 20ms */
  swDelay( 20, DMSEC );

  /* function set: 8-bit interface */
  lcdCmd( 0x30 );

  /* wait for 5ms */
  swDelay( 5, DMSEC );

  /* function set: 8-bit interface */
  lcdCmd( 0x30 );

  /* wait for 200us */
  swDelay( 2, DHUSEC );

  /* function set: 8-bit interface */
  lcdCmd( 0x30 );

  /* function set: 4-bit interface */
  lcdCmd( 0x20 );

  /* function set: 4-bit interface, two-line display with 5x8 char font */
  lcdCmd( 0x20 );
  lcdCmd( 0x80 );

  /* turn display off */
  lcdCmd( 0x00 );
  lcdCmd( 0x80 );

  /* turn display on */
  lcdCmd( 0x00 );
  lcdCmd( 0x10 );

  /* set for entry mode */
  lcdCmd( 0x00 );
  lcdCmd( 0x60 );

  /* turn on display, cursor off */
  lcdCmd( 0x00 );
  lcdCmd( 0xC0 );
}

/****************** lcdClock() *************************/
/* function to flash the enable on the LCD */
void lcdClock(void)
{
  LCDCTRLDIR |= (LCDEN);
  LCDCTRLSEL &= ~(LCDEN);
  LCDCTRLOUT &= ~(LCDEN);       /* clear enable */

  swDelay( 8, DMSEC );          /* wait 8 ms */
  LCDCTRLOUT |= (LCDEN);        /* set enable */

  swDelay( 4, DMSEC );           /* wait 4 ms */
  LCDCTRLOUT &= ~(LCDEN);       /* clear enable */
}

/******************* lcdWrite() *************************/
/* function to write data onto the data lines of the LCD */
void lcdWrite( unsigned char databus )
{
  /* set P4.4-4.7 to output direction */
  LCDDIR |= (BIT4|BIT5|BIT6|BIT7);
  /* set P4.4-4.7 I/O option */
  LCDSEL &= ~(BIT4|BIT5|BIT6|BIT7);
```

```
  /* P4.4-4.7 output = databus */
  LCDOUT &= 0x0F;
  LCDOUT += databus;

  lcdClock();
}

/******************* lcdPrint() *************************/
/* function to print a character at the cursor on the LCD */
void lcdPrint( char databus )
{
  LCDCTRLDIR |= (LCDRW|LCDRS);             /* set P4.2-4.3 to output direction */
  LCDCTRLSEL &= ~(LCDRW|LCDRS);            /* P4.2-4.3 I/O option */
  LCDCTRLOUT &= ~(LCDRW);                  /* clear R/W flag on P4.2 */
  LCDCTRLOUT |= (LCDRS);                   /* set RS flag on P4.3 */

  lcdWrite( databus & 0xF0 );              /* send high nybble of data */
  lcdWrite( databus << 4 );                /* send low nybble of data */
}

#endif

/******************* lcdCmd() *************************/
/* function to send a command to the LCD */
void lcdCmd( unsigned char databus )
{
  LCDCTRLDIR |= (LCDRW|LCDRS);              /* set P4.2-4.3 to output direction */
  LCDCTRLSEL &= ~(LCDRW|LCDRS);             /* P4.2-4.3 I/O option */
  LCDCTRLOUT &= ~(LCDRW|LCDRS);             /* P4.2-4.3 output = 00 (off) */

  lcdWrite( databus );
}

/******************* lcdPrints() *************************/
/* function to print a string at the cursor on the LCD */
void lcdPrints( char* buffer )
{
  char i;

  for( i=0; i<strlen(buffer); i++ ){
    /* check to see if we should skip down a line */
    if( buffer[i]=='\n' ){
      lcdLine2();        /* set DDRAM address to line 2, position 1 */
    } else{
      lcdPrint( buffer[i] );
    }
  }
}

#endif

/* mqpsfm.h */
/***************** header file for SFM ********************/

#ifndef MQPSFM_H
#define MQPSFM_H

void sfmInit0(void);
void sfmInit1(void);
void sfmEnable(void);
void sfmDisable(void);
void sfmRecv(void);
void sfmSend(void);
int  sfmPost(void);
void sfmErase(void);

int sfmBuffer(char*, int);
int sfmFlush();

#define SFMFULL     0x0A
#define SFMNFULL    0xA0
```

```
#define SFMFLUSH    0x0C
#define SFMNFLUSH   0xC0

#include "mqpusart.h"
#include <stdlib.h>

#define SPIHOLD BIT4
#define SPIS    BIT5
#define SPIW    BIT6

/* SPI memory library */
#include "c2082.h"
#include "c2082.c"
#include "Serialize.h"
#include "Serialize.c"

/* GLOBALS */
ParameterType fp;     /* contains all flash memory parameters */
ReturnType rRetVal;   /* return type enum for flash memory */
char foobarbaz;

IObuffer sfmbuffer;
int sfmpos;

/******************** init_spi0() *************************
 * function to initialize USART 0 for SPI
 * post: USART 0 set up for 3-wire SPI connection, 8-bit master
 */
void sfmInit0(void){
  P3SEL |= (BIT1|BIT2|BIT3);    /* setup P3 for SPI mode */
  P3OUT = 0x20;
  P3DIR |= 0x30;

  U0ME  |= UTXE0 + URXE0;       /* enable USART0 transmit and receive modules */
  UCTL0 = CHAR + SYNC + MM;     /* 8-bit, SPI, Master */
  UTCTL0 = CKPL + SSEL1 - STC;  /* Polarity, SMCLK, 3-wire */
  U0BR0 = 0x02;                 /* SPICLK = SMCLK/2 */
  U0BR1 = 0x00;
  UMCTL0 = 0x00;

  IE1 |= URXIE0 + UTXIE0;       /* RX and TX interrupt enable */
  UCTL0 &= ~SWRST;              /* enable SPI */
  IFG1 &= ~UTXIFG0;             /* clear initial flag on POR */

  P2DIR |= (SPIHOLD|SPIS|SPIW);         /* setup for P2.4-6 for HOLD, S and W/VPP */
  P2SEL &= ~(SPIHOLD|SPIS|SPIW);        /* set I/O option for output */
  P2OUT |= (SPIS);                      /* falling edge of select after power-up, turn on */
  swDelay( 2, DHUSEC );                 /* wait 200 us */
  P2OUT &= ~(SPIHOLD|SPIS|SPIW);        /* clear hold, select and disable write on slave (all
active low)*/

  dusart0.tx.buffer = char32;
  dusart0.tx.length = 0;
  dusart0.tx.index  = 0;
  dusart0.tx.size   = 32;
  dusart0.tx.status = FREE;

  dusart0.rx.buffer = char125;
  dusart0.rx.length = 0;
  dusart0.rx.size   = 125;
  dusart0.rx.status = FREE;

  dusart0.port      = SPI0;
  dusart0.recv      = sfmRecv;
  dusart0.send      = sfmSend;

  hsfm = &dusart0;

  sfmbuffer.buffer = char256;
  sfmbuffer.index  = 0;
  sfmbuffer.length = 0;
```

```
  sfmbuffer.size   = 256;
  sfmbuffer.status = FREE;

  sfmpos = 0;
}

/******************** init_spi1() ************************
 * function to initialize USART 1 for SPI
 * post: USART 1 set up for 3-wire SPI connection, 8-bit master
 */
void sfmInit1(void){
  P5SEL |= (BIT1|BIT2|BIT3);    /* setup P5 for SPI mode */
  P5OUT = 0x20;
  P5DIR |= 0x30;

  U1ME  |= UTXE1 + URXE1;       /*  enable USART1 transmit and receive modules */
  UCTL1 = CHAR + SYNC + MM;     /* 8-bit, SPI, Master */
  UTCTL1 = CKPL + SSEL1 - STC;  /* Polarity, SMCLK, 3-wire */
  U1BR0 = 0x02;                 /* SPICLK = SMCLK/2 */
  U1BR1 = 0x00;
  UMCTL1 = 0x00;

  IE2 |= URXIE1 + UTXIE1;       /* RX and TX interrupt enable */
  UCTL1 &= ~SWRST;              /* SPI enable */
  IFG2 &= ~UTXIFG1;             /* clear initial flag on POR */

  P2DIR |= (SPIHOLD|SPIS|SPIW);         /* setup for P2.4-6 for HOLD, S and W/VPP */
  P2SEL &= ~(SPIHOLD|SPIS|SPIW);        /* set I/O option for output */
  P2OUT |= (SPIS);                      /* falling edge of select after power-up, turn on */
  swDelay( 2, DHUSEC );                 /* wait 200 us */
  P2OUT &= ~(SPIHOLD|SPIS|SPIW);        /* clear hold, select and disable write on slave (all
active low)*/

  dusart1.tx.buffer = char32;
  dusart1.tx.index  = 0;
  dusart1.tx.length = 0;
  dusart1.tx.size   = 32;
  dusart1.tx.status = FREE;

  dusart1.rx.buffer = char125;
  dusart1.rx.length = 0;
  dusart1.rx.size   = 125;
  dusart1.rx.status = FREE;

  dusart1.port      = SPI1;
  dusart1.recv      = sfmRecv;
  dusart1.send      = sfmSend;

  hsfm = &dusart1;

  sfmbuffer.buffer = char256;
  sfmbuffer.index  = 0;
  sfmbuffer.length = 0;
  sfmbuffer.size   = 256;
  sfmbuffer.status = FREE;

  sfmpos = 0;
}

/******************** sfmEnable() ************************
 * function to enable interrupts for SFM
 */
void sfmEnable(void)
{
  if( (*hsfm).port == SPI0 ){
    ME1 |= USPIE0;                  /* enable the SPI module */
  } else if( (*hsfm).port == SPI1 ){
    ME2 |= USPIE1;                  /* enable the SPI module */
  }
}
```

```
/******************** sfmDisable() ************************
 * function to disable interrupts for SFM
 */
void sfmDisable(void)
{
  if( (*hsfm).port == SPI0 ){
    ME1 &= ~(USPIE0);                 /* enable the SPI module */
  } else if( (*hsfm).port == SPI1 ){
    ME2 &= ~(USPIE1);                 /* enable the SPI module */
  }
}

/******************** sfmRecv() ************************
 * function to
 */
void sfmRecv(void){
  (*hsfm).rx.status=FREE;

  if( (*hsfm).rx.length<(*hsfm).rx.size ){
    if( (*hsfm).port == SPI0 ){
      (*hsfm).rx.buffer[ (*hsfm).rx.length ] = RXBUF0;
    } else if( (*hsfm).port == SPI1 ){
      (*hsfm).rx.buffer[ (*hsfm).rx.length ] = RXBUF1;
    }

    (*hsfm).rx.length++;
  }
}

/******************** sfmSend() ************************
 * function to
 */
void sfmSend(void){
  (*hsfm).tx.status=WORK;

  (*hsfm).tx.index++;
  if( (*hsfm).tx.index < (*hsfm).tx.length ){
    if( (*hsfm).port == SPI0 ){
      TXBUF0 = (*hsfm).tx.buffer[(*hsfm).tx.index];
    } else if( (*hsfm).port == SPI1 ){
      TXBUF1 = (*hsfm).tx.buffer[(*hsfm).tx.index];
    }
  }
  else{
    (*hsfm).tx.status = FREE;
    (*hsfm).tx.index  = 0;
    (*hsfm).tx.length = 0;
  }
}

int i;

/******************** sfmPost() ************************
 * function to do a Power-on Self Test for the SFM
 */
int sfmPost(void){
  sfmEnable();
  _EINT();

  for(i=0; i<10 && rRetVal != Flash_Success; i++ ){
    /* read manufacturer identification */
    rRetVal=Flash(ReadManufacturerIdentification, &fp );
    foo = fp.ReadManufacturerIdentification.ucManufacturerIdentification;

#ifdef DEBUG
    lcdClear();
    lcdPrints( FlashErrorStr( rRetVal ) );
    lcdPrints( "\n" );
    lcdPrints( FlashErrorStr( rRetVal ) + 8 );
    swDelay( 1, DHSEC );
#endif
```

```
  }

  sfmDisable();
  _DINT();

  return rRetVal;
}

void sfmErase(void)
{
  lcdClear();
  lcdPrint( '5' );
  swDelay( 3, DHSEC );
  lcdPrint( '4' );
  swDelay( 3, DHSEC );
  lcdPrint( '3' );
  swDelay( 3, DHSEC );
  lcdPrint( '2' );
  swDelay( 3, DHSEC );
  lcdPrint( '1' );
  swDelay( 3, DHSEC );
  lcdPrint( '0' );
  swDelay( 3, DHSEC );
  lcdPrints( "\n" );
  lcdPrints( "Deleting" );

  sfmEnable();
  _EINT();

  /* erase all memory */
  rRetVal = Flash( BulkErase, &fp);

  sfmDisable();
  _DINT();

  lcdClear();
  lcdPrints( FlashErrorStr( rRetVal ) );
  lcdPrints( "\n" );
  lcdPrints( FlashErrorStr( rRetVal ) + 8 );
  swDelay( 3, DHSEC );
}

/******************** sfmBuffer() *************************
 * function to buffer more data to send for SFM or return saying it's full
 */
int sfmBuffer( char* buffer, int length )
{
  /* is it full or has too much been stored to buffer this? */
  if( sfmbuffer.length + length < sfmbuffer.size ){
    strncpy( sfmbuffer.buffer + sfmbuffer.length, buffer, length );
    sfmbuffer.length += length;
  } else{ /* yep, say it's full */
    return SFMFULL;
  }

  return SFMNFULL;  /* otherwise, say it's not full */
}

/******************** sfmFlush() *************************
 * function to flush data from the SFM buffer to the SFM
 * pre: assumes interrupts enabled
 */
int sfmFlush(void)
{
    sfmEnable();
    _EINT();

    /* write some data */
    fp.Program.udAddr = sfmpos;
    fp.Program.udNrOfElementsInArray = sfmbuffer.length;
    fp.Program.pArray = (void*)sfmbuffer.buffer;
```

```
    rRetVal = Flash( Program, &fp );

    lcdClear();
    lcdPrints( FlashErrorStr( rRetVal ) );
    lcdPrints( "\n" );
    lcdPrints( FlashErrorStr( rRetVal ) + 8 );
    swDelay( 3, DHSEC );

    sfmpos += sfmbuffer.length;   /* update position */

    /* say it's empty */
    sfmbuffer.length = 0;

    sfmDisable();
    _DINT();

    return SFMFLUSH;
}


#endif

/* mqptimer.h */
/**************header file for timer functions and structures ****************/

#ifndef MQPTIMER_H
#define MQPTIMER_H

enum TimerStatus {
   DONE, RUNNING
};

typedef struct _Timer{
  void (*start)(void);  /* function pointer to start() function */
  void (*stop)(void);   /* function pointer to stop() function */
  int count;            /* number of iterations to do */
  int counter;          /* current number of iterations run */
  int offset;           /* offset to start TAR at */
  enum TimerStatus status;  /* status of timer */
} Timer;

void swDelay(unsigned int, unsigned int);   // simple SW delay loop
void hwDelay(Timer*, unsigned int, unsigned int);

void taInit(void);    /* initialize Timer A */
void taStart(void);   /* start Timer A */
void taStop(void);    /* stop Timer A */

Timer timerA;

void taInit(void)
{
  timerA.count = 1;
  timerA.offset = 0;
  timerA.status = DONE;
  timerA.start = taStart;
  timerA.stop = taStop;
}

void taStart(void)
{
  timerA.counter = 0;
  timerA.status = RUNNING;
  TAR = timerA.offset;      /* set TAR at offset value */
  TACTL = TASSEL_1 + TAIE + MC_2; /* use ACLK, interrupt-driven, cont counter */
}

void taStop(void)
{
  TACTL = MC_0;          /* set to stop counting */
  timerA.status = DONE;
```

```
}

#pragma vector=TIMERA1_VECTOR
__interrupt void timerA_interrupt(void)
{
  timerA.counter++;   /* increment counter */

  /* check if done counting */
  if( timerA.counter >= timerA.count ){
      taStop();         /* stop the clock */
  } else{
      TACTL &= ~(TAIFG);  /* otherwise, clear TAIFG flag */
  }
}

/****************** hwDelay() ************************/
void hwDelay(Timer *ptimer, unsigned int count, unsigned int offset )
{
  (*ptimer).count   = count;
  (*ptimer).offset  = offset;
  (*(*ptimer).start)();
}

#define DHSEC  65535          /* half-second delay multiplier */
#define DMSEC  33             /* millisecond delay multiplier */
#define DHUSEC 3              /* 100 us delay multipler */

/****************** swDelay() ************************/
void swDelay(unsigned int max_cnt, unsigned int multiplier)
{
   unsigned int  cnt1=0, cnt2;

   while (cnt1 < max_cnt)
   {
     cnt2 = 0;
     while (cnt2 < multiplier)
       cnt2++;
     cnt1++;
   }
}
#endif

/* mqpuart.h */
/***************** header file for UART interface *********************/

#ifndef MQPUSART_H
#define MQPUSART_H

#include "mqpio.h"
IOdevice dusart0, dusart1;      /* IOdevices on USART ports */
IOhandle husb, hsfm, hgps;      /* IO handles for USB, SFM and GPS */

/****************** usart0_tx() ************************
 * interrupt service routine for USART 0 tranmission
 * pre: to be entered, GIE must be set and a new value has been
 *    written to TXBUF0
 */
#pragma vector=UART0TX_VECTOR
__interrupt void usart0_tx (void)
{
  dusart0.tx.status=WORK;
  (*dusart0.send)();
}

/****************** usart0_rx() ************************
 * interrupt service routine for USART 0 reception
 * pre: to be entered, GIE must be set and a new value has been
 *    recieved in RXBUF0
 * post: latest received character added to buffer, unless buffer overflow
 */
char foo;
```

```
#pragma vector=UART0RX_VECTOR
__interrupt void usart0_rx (void)
{
   dusart0.rx.status=WORK;
   (*dusart0.recv)();
   foo=RXBUF0;              /* pull out of buffer */
}

/****************** usart1_tx() ************************
 * interrupt service routine for USART 1 tranmission
 * pre: to be entered, GIE must be set and a new value has been
 *    written to TXBUF1
 */
#pragma vector=UART1TX_VECTOR
__interrupt void usart1_tx (void)
{
  dusart1.tx.status=WORK;
  (*dusart1.send)();
}

/****************** usart1_rx() ************************
 * interrupt service routine for USART 1 reception
 * pre: to be entered, GIE must be set and a new value has been
 *    recieved in RXBUF1
 * post: latest received character added to buffer, unless buffer overflow
 */
#pragma vector=UART1RX_VECTOR
__interrupt void usart1_rx (void)
{
  dusart1.rx.status=WORK;
  (*dusart1.recv)();
  foo=RXBUF1;              /* pull out of buffer */
}

#endif

/* mqpusb.h */
/***************** header file for USB *********************/

#ifndef MQPUSB_H
#define MQPUSB_H

void usbInit(void);
void usbEnable(void);
void usbDisable(void);
void usbRecv(void);
void usbSend(void);
void usbPost(void);
void usbPrint(char*);
void usbPrintMenu(void);
void usbDump(void);

#include "mqpusart.h"
#include "mqpsfm.h"
#include <string.h>
#include <stdlib.h>

/******************** usbInit() *************************
 * function to initialize USART 1
 * post: USART 1 set up for 921600 baud 8-none-1 connection
 *    using a 8 MHz crystal placed in XT2
 *    with no flow control and interrupts are enabled
 */
void usbInit(void){
  P3SEL |= (BIT6|BIT7);                     // P3.6,7 = USART1 TXD/RXD

  BCSCTL1 &= ~XT2OFF;                       // XT2on
  BCSCTL2 |= SELM_2 + SELS;                 // MCLK= SMCLK= XT2 (safe)

  UCTL1 |= CHAR;                            // 8-bit character
  UTCTL1 |= SSEL1;                          // UCLK = SMCLK
```

```
  /* 921600 baud */
  UBR01 = 0x08;
  UBR11 = 0x00;
  UMCTL1 = 0x5B;                            // modulation

  UCTL1 &= ~SWRST;          /* initialize USART1 state machine */
  IE2 |= URXIE1 + UTXIE1;   /* enable USART1 RX/TX interrupts */
  IFG2 &= ~UTXIFG1;         /* clear inital flag on POR */

  dusart1.tx.buffer = char180;
  dusart1.tx.index  = 0;
  dusart1.tx.length = 0;
  dusart1.tx.size   = 180;
  dusart1.tx.status = FREE;

  dusart1.rx.buffer = char1;
  dusart1.rx.length = 0;
  dusart1.rx.size   = 1;
  dusart1.rx.status = FREE;
  dusart1.rx.stopbyte='\n'; /* end of input */

  dusart1.port    = UART1;
  dusart1.recv    = usbRecv;
  dusart1.send    = usbSend;

  husb = &dusart1;
}

/******************** usbEnable() ************************
 * function to enable interrupts for USB
 */
void usbEnable(void)
{
  ME2 |= UTXE1 + URXE1;                     /* enable USART1 TXD/RXD */
}

/******************** usbDisable() ************************
 * function to disable interrupts for USB
 */
void usbDisable(void)
{
    ME2 &= ~(UTXE1);
    ME2 &= ~(URXE1);
}

/******************** usbRecv() ************************
 * function to receive data over USB
 */
void usbRecv(void)
{
  (dusart1).rx.status=FREE;

  if( (dusart1).rx.length<(dusart1).rx.size ){
    (dusart1).rx.buffer[ (dusart1).rx.length ] = RXBUF1;
    (dusart1).rx.length++;
  } else{
    (dusart1).rx.buffer[0] = RXBUF1;
    (dusart1).rx.length = 0;
  }
}

/******************** usbSend() ************************
 * function to send data over USB
 */
void usbSend(void)
{
  (dusart1).tx.status=WORK;

  (dusart1).tx.index++;
  if( (dusart1).tx.index < (dusart1).tx.length ){
```

```
    TXBUF1 = (dusart1).tx.buffer[(dusart1).tx.index];
  }
  else{
    (dusart1).tx.length = 0;
    (dusart1).tx.index  = 0;
    (dusart1).tx.status = FREE;
  }
}

/******************** usbPost() *************************
 * function to do Power On Self-Test of USB
 */
void usbPost(void)
{
  usbEnable();
  _EINT();

  usbPrint( "GPS Road Roughness\n\rv0.9\n\r" );

  usbDisable();
  _DINT();
}

/******************** usbPrint() *************************
 * function to print string to USB
 * pre: assumes USB interrupts enabled
 */
void usbPrint( char* buffer )
{
//  swDelay( 5, DMSEC );  /* wait just a tiny bit */

  /* copy passed-in string into transmit buffer */
  strncpy( (dusart1).tx.buffer, buffer, strlen(buffer) );
  (dusart1).tx.index=0;
  (dusart1).tx.length=strlen( buffer );

  /* send the first char to initialize transfer */
  TXBUF1 = (dusart1).tx.buffer[(dusart1).tx.index];
}

/******************** usbPrintMenu() *************************
 * function to print menu of commands to USB
 * pre: assumes USB interrupts enabled
 */
void usbPrintMenu(void)
{
  usbPrint( "\n\r USB Menu\n\r---------\n\r   c clear memory\n\r\
   d download from memory\n\r   h print this help menu again\n\r" );
}

/******************** usbPrint() *************************
 * function to print entire memory out to USB
 * pre: assumes interrupts enabled
 */
void usbDump(void)
{
  sfmEnable();

  sfmreadpos = 0;

  while( sfmreadpos < FLASH_SIZE ){
    fp.Read.udAddr = sfmreadpos;
    fp.Read.udNrOfElementsToRead = (dusart1).tx.size;
    fp.Read.pArray = (void*)(dusart1).tx.buffer;
    rRetVal=Flash(Read, &fp );

    sfmreadpos += (dusart1).tx.size;

    /* send along first char to initiate transfer */
    (dusart1).tx.index=0;
    (dusart1).tx.length=(dusart1).tx.size;
```

```
    TXBUF1 = (dusart1).tx.buffer[(dusart1).tx.index];
  }

  sfmDisable();
}

#endif

/****** Header File for support of STFL-I based Serial Flash Memory Driver *****

   Filename:    Serialize.h
   Description: Header file for Serialize.c
                Consult also the C file for more details.

   Version:     1.0
   Date:        08-11-2004
   Authors:     Tan Zhi, STMicroelectronics, Shanghai (China)
   Copyright (c) 2004 STMicroelectronics.

   THE PRESENT SOFTWARE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH
   CODING INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A
   RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR
   CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH
   SOFTWARE AND/OR THE USE MADE BY CUSTOMERS OF THE CODING INFORMATION CONTAINED HEREIN
   IN CONNECTION WITH THEIR PRODUCTS.
*******************************************************************************

   Version History.
   Ver.   Date      Comments

   1.0   08/11/2004   Initial release

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

#ifndef _SERIALIZE_H_
#define _SERIALIZE_H_

#include "c2082.h"

#define ptrNull 0   // a null pointer

#define True  1
#define False 0
typedef unsigned char Bool;


// mask bit definitions for SPI master side configuration
enum
{
    MaskBit_Trans                       = 0x01,  // mask bit for Transfer enable/disable
    MaskBit_Recv                        = 0x02,  // mask bit for Receive enable/disable
    MaskBit_Trans_Relevant              = 0x04,  // check whether MaskBit_Trans is necessary
    MaskBit_Recv_Relevant               = 0x08,  // check whether MaskBit_Recv is necessary

    MaskBit_SlaveSelect                 = 0x10,  // mask bit for Slave Select/Deselect
    MaskBit_SelectSlave_Relevant        = 0x20,  // check whether MaskBit_SelectSlave is
necessary

};

// Acceptable values for SPI master side configuration
typedef enum _SpiMasterConfigOptions
{
    enumNull                            = 0,     // do nothing
    enumEnableTransOnly                 = 0x05,  // enable transfer
    enumEnableRecvOnly                  = 0x0A,  // enable receive
    enumEnableTansRecv                  = 0x0F,  // enable transfer & receive

    enumEnableTransOnly_SelectSlave     = 0x35,  // enable transfer and select slave
    enumEnableRecvOnly_SelectSlave      = 0x3A,  // enable receive and select slave
    enumEnableTansRecv_SelectSlave      = 0x3F,  // enable transfer & receive and select slave
```

```
    enumDisableTransOnly                  = 0x04,  // disable transfer and deselect slave
    enumDisableRecvOnly                   = 0x08,  // disable receive
    enumDisableTransRecv                  = 0x0C,  // disable transfer & receive

    enumDisableTransOnly_DeSelectSlave    = 0x24,  // disable transfer and deselect slave
    enumDisableRecvOnly_DeSelectSlave     = 0x28,  // disable receive and deselect slave
    enumDisableTansRecv_DeSelectSlave     = 0x2C   // disable transfer & receive and deselect
slave

}SpiMasterConfigOptions;

// char stream definition for
typedef struct _structCharStream
{
    ST_uint8* pChar;                              // buffer address that holds the streams
    ST_uint32 length;                             // length of the stream in bytes
}CharStream;

void InitSPIMaster();                             // one-time setting to work in SPI mode
void ConfigureSpiMaster(
                      SpiMasterConfigOptions opt  // configuration options
                      );
Bool Serialize(
              const CharStream* char_stream_send,   // char stream to be sent to the
memory(incl. instruction, address etc)
              CharStream* char_stream_recv,         // char stream to be received from the
memory
              SpiMasterConfigOptions optBefore,     // Pre-Configurations on the SPI master side
              SpiMasterConfigOptions optAfter       // Post-Configurations on the SPI master
side
             );

#endif //end of file
```