UNIVERSITY OF CALIFORNIA

Los Angeles

SmartCast - Novel Textile Sensors for Embedded Pressure Sensing of Orthopedic Casts

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Electrical Engineering

by

Andrew Danilovic

2013

ABSTRACT OF THE THESIS

SmartCast - Novel Textile Sensors for Embedded Pressure Sensing of Orthopedic Casts

by

Andrew Danilovic

Master of Science in Electrical Engineering

University of California, Los Angeles, 2013

Professor William J. Kaiser, Chair

An orthopedic plaster or fiberglass cast is often applied after a bone fracture to hold the broken bone in place and allow the fracture to heal and the bone to set. If this cast becomes loose, due to a reduction in swelling for example, the bone may not heal properly and a misalignment of the bone may occur. If this problem goes unnoticed, surgery may need to be performed to re-break and reset the bone. This is particularly a problem with patients who are unable to accurately express what they are feeling, e.g. children, and who may not be able to identify that their cast is loose. The SmartCast system has been designed to avoid this complication and to identify early if a cast is not fitting well. SmartCast consists of a sleeve with novel textile sensors capable of measuring the pressure the cast is applying to an appendage as well as electronics to sample and log the sensor data. Several types of textile piezoresistive sensor sleeves have been designed and tested to determine which type would best be able to measure the cast pressure. A full prototype system, including firmware, sensor sleeve, and printed circuit board, has been designed, implemented and tested. The SmartCast system will allow doctors and orthopedic professionals to obtain a better understanding of how well a cast fits over time and to avoid the painful procedure of resetting a bone.

The thesis of Andrew Danilovic is approved.

Majid Sarrafzadeh

Robert Candler

William J. Kaiser, Committee Chair

University of California, Los Angeles

2013

I would like to thank Professor William Kaiser for his unfailing positivity and for his dedication to his students, and Professor Majid Sarrafzadeh and Professor Robert Candler for their help in reviewing this thesis.

I would also like to thank Henrik Borgstrom, Mahdi Ashktorab, and Bijan Mapar for their help and support during the course of this project. They provided invaluable guidance and advice during system development. I would like to thank Minta Manning, Senior Wardrobe Technician in the UCLA Theatre Department, for sewing and constructing two of the SmartCast prototype textile sensor matrices. I would like to thank my SmartCast team members, Maryam Shahbazi and Carson Umsted, for helping to coordinate our work on SmartCast.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Background

A plaster or fiberglass cast placed over a broken bone may become loose over time if swelling decreases. This is particularly a problem with patients who are unable to accurately express what they are feeling, e.g. children, and who may not be able to identify that their cast is loose. The SmartCast system uses novel textile pressure sensors to measure the pressure the cast is placing on the arm to give medical professionals an understanding of how well a cast fits over time.

There are a number of types of sensors that are able to measure applied force or pressure, such as[8]:

1. capacitive sensor with compressible, elastic dielectric
2. electroactive polymer or conductive elastomer based sensors, e.g. Polyvinylidene fluoride (PVDF), polypyrrole (PPY)
3. magnetic pressure sensors, e.g. Variable Reluctance Pressure sensors (VRP)
4. optical sensors, e.g. Fabry–Perot interferometer
5. silicon based piezoelectric and piezoresistive sensors for industrial applications

While these sensors have been well validated and used extensively, the SmartCast system instead uses textile pressure sensors due to their ease of construction and integration into a wearable sleeve.

Wearable computing and textile sensors have become increasingly popular research topics. The International Symposium on Wearable Computers (ISWC) has been published annually since

1996, and is currently in its 16th edition. Capacitive and piezoresistive textile sensors capable of measuring strain, bend, and applied force have recently been developed[14]. Two main types of textile based sensors are textiles that are coated with electroactive polymers which act as the sensing element, and those that use the textile itself as the sensing element. The SmartCast sensor sleeve users the latter. Recent applications of textile sensors include noninvasive continuous health and biomechanical monitoring, sitting posture classification[9], daily cardiovascular disease prevention and respiration monitoring[7, 10, 13], and biosignal monitoring[19]. [20] uses the elasticity of the conductive thread itself as a pressure transducer, but yields a small dynamic range. [10] describes a capacitive sensor that relies on the changing capacitance between two parallel plates as they move laterally with applied strain. [19] describes a mattress sensing system using polymer sensors to monitor biosignals such as heart rate and respiration during sleep, either at home or in a healthcare facility. The MIT Media Lab has also released open source e-textile kits that can form the basis of modular designs using textile sensors[4]. [15] uses a matrix of polymer based pressure sensors to act as a kind of skin. This thesis explores a similar matrix of sensors but applied to orthopedic casts and consisting of textile piezoresistive sensors. The SmartCast sensor sleeve is a new application of textile sensors to Orthopedic cast monitoring.

1.2 Objectives and Contributions

The goal of the SmartCast system is to help medical professionals and orthopedists understand how well a plaster or fiberglass cast fits over time with a future goal of early detection of loose fitting casts. With this knowledge, medical professionals could then better understand the lifecycle of the cast and make informed decision on when to replace casts before bones set improperly. To this end, the SmartCast system consists of a mobile, wearable platform capable of measuring and logging data from an array of piezoresistive sensors worn under a plaster or fiberglass cast for a period of 2-3 months.

Section 3.3.3 demonstrates that the resolution of the SmartaCast sensors to applied force varies with the force being applied. The resolution ranges from approximately 0.003 Newtons (N) with 0.01 N force being applied to 0.2 N with 1.4 N force being applied. **Section 3.5** shows that the forces under a cast can range from 0.1 N to 0.5 N, which is right in line with what the SmartCast sensors can detect.

The work described in this thesis consists of the following contributions:

1. Design, analysis and validation of a mobile, wearable electronics platform, including all system code and firmware, which measures and records data from an array of piezroresistive sensors
2. Design, testing, and validation of several piezoresistive textile sensors and sensor sleeves
3. SmartCast sensor network fix to nullify the effects of multiple paths through the sensor network which confound sensor measurements

Chapter 2 - System & Sensor Design

2.1 Overview of SmartCast Textile Sensors and Sensor Matrix

The basic textile pressure sensor tested with the SmartCast system is shown in **Figure 1**. It consists of several layers of material:

Layer 1.	non-conductive fabric, e.g. cotton, polyester, etc.
Layer 2.	elastic conductive fabric, e.g. MedTex180
Layer 3.	piezoresistive material, e.g. polyester w/ woven carbon fibers or Velostat
Layer 4.	elastic conductive fabric
Layer 5.	non-conductive fabric

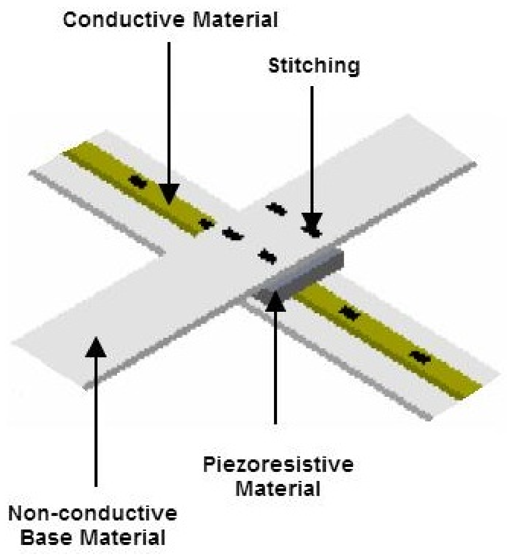


Figure 1: Basic Textile Pressure Sensor Design

Layer 3 acts like a pressure sensitive resistor, and Layers 2 and 4 act as wires attached to either ends of the resistor. As the applied force increases, the resistance decreases. The resistance vs. applied force curve generally follows a 1/x shape. In the next section, several sensor prototypes are shown that deviate slightly from the design in **Figure 1**, but all are based off of and are very similar to it.

Figure 2 shows a 4x4 sensor matrix consisting of 16 of the sensors shown in **Figure 1**. Note the 4 pieces of material which are stitched to a row, and then folded over the other rows. The purpose of these pieces is to bring all the rows to the top of the matrix; as this will be placed under a cast, we will only have access to those connections at the very top, near the edge of the cast. They are folded in such a way to avoid shorts between the rows. A sensor can be measured by attaching leads to the corresponding row and column for that sensor, though a complication with this method is discussed in Chapter 4 - Sensor Network Fix.

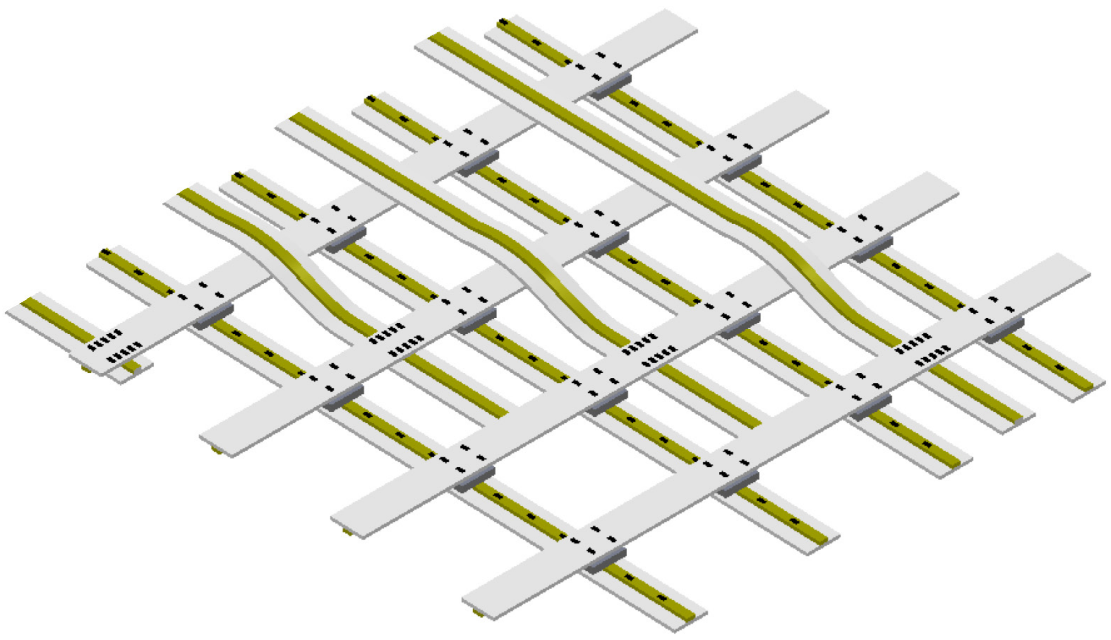


Figure 2: Sleeveless 4x4 Sensor Matrix Design

2.2 Evolution of the SmartCast Sensor Network

The first SmartCast sensor sleeve designed is shown in **Figure 3**. Comparing this design to **Figure 1**, here Layer 1 is a spandex arm sleeve, Layer 3 consists of three layers of polyester with woven carbon fibers, and Layer 5 has been left out. This was an effective design for measuring the pressure on the arm, but improvements were made in subsequent prototypes to decrease the size of the sensors and connection elements, and to add waterproofing. Waterproofing will be necessary as the sleeve will live in an environment under the cast which may include blood, sweat, etc.

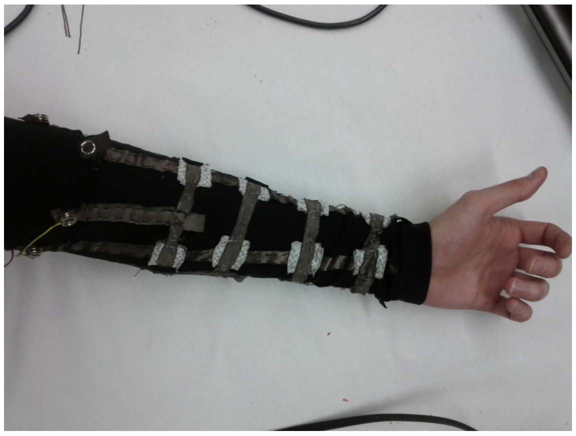


Figure 3: Original Sleeve Design

Figure 4 shows the most recent prototype constructed by Minta Manning based off of the original design in **Figure 3**. Improvements in this design are that here, each sensor is made separately, and then stitched on to the sleeve, allowing for easier manufacturability. Also, each sensor has increased waterproofness due to the top and bottom layer of nylon material, though the conductive material connections are still exposed to the environment, which may cause shorts to occur. Each sensor uses Velostat as the piezoresistive element, which was much more reliable than polyester with carbon fibers. Conductive thread was used to bring the columns and rows to the top of the sleeve in an effort to decrease the size of connecting elements. Some issues with this sleeve are that it is necessary to use conductive epoxy (i.e. cold solder) at the interface between each sensor and between the conductive thread and sensors, as stitching two pieces of conductive material together is not a reliable connection. Copper wires were attached to the conductive thread at the top of the sleeve by using conductive epoxy and heatshrink.

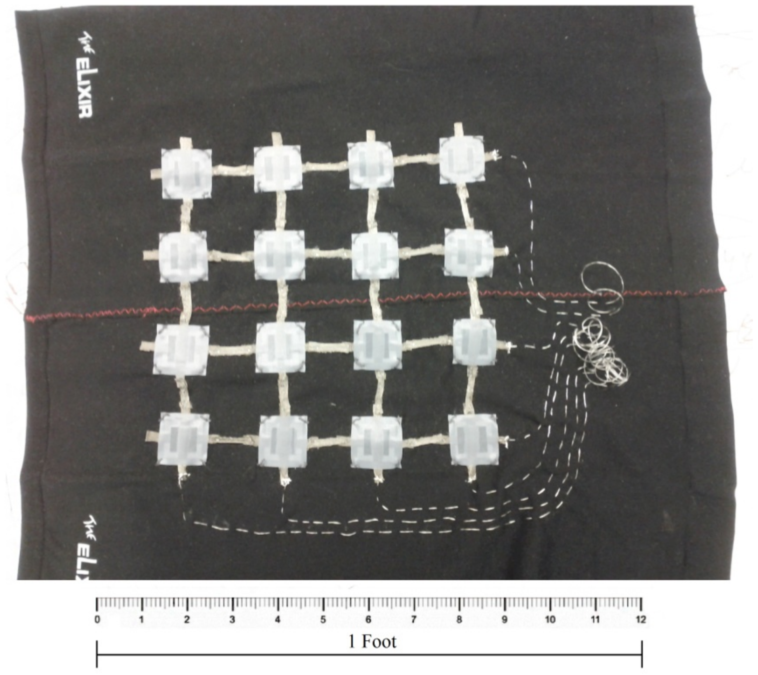


Figure 4: First Sensor Sleeve Prototype

Figure 5 and **Figure 6** show two sleeveless prototypes covered completely in nylon to make the entire sensor matrix waterproof. While these sensors exhibited a strong waterproof property, they were not elastic. Elasticity is an important property for the SmartCast sensor sleeve as a subject's arm may experience swelling and the sleeve must not be constricting, but must expand and contract with the subject's arm. **Figure 7** shows a full 4x4 sleeveless sensor matrix similar to the matrix in **Figure 5** but without the top and bottom layer of nylon applied. As we will see in section 3.3 Platform Test with Masses, this prototype was not successful, as polyester with carbon fibers is not reliable as the piezoresistive element.

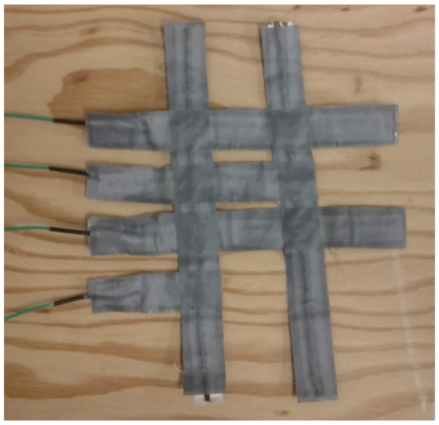


Figure 5: Grey Nylon Textile Sensor

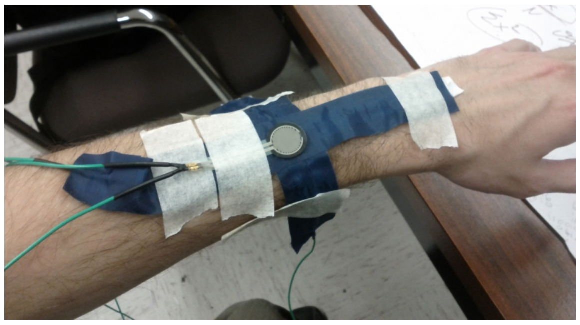


Figure 6: Blue Nylon Textile Sensor Being Worn for Testing

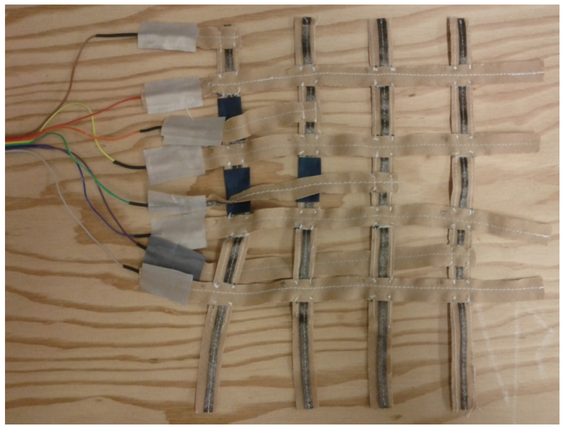


Figure 7: 1st Sleeveless Sensor Matrix Prototype

2.3 System Software Architecture

The SmartCast system has two major software modules:

1. SmartCast Application
2. Memory Management

The SmartCast application is responsible for sampling the sensor matrix, calculating on-line statistics, and executing necessary sleep and wake up routines. The Memory Management module is responsible for logging data and utilizing the internal EEPROM and Flash memory as well as the external SD Card. **Figure 8** is a diagram of these two modules and the main routines in each.

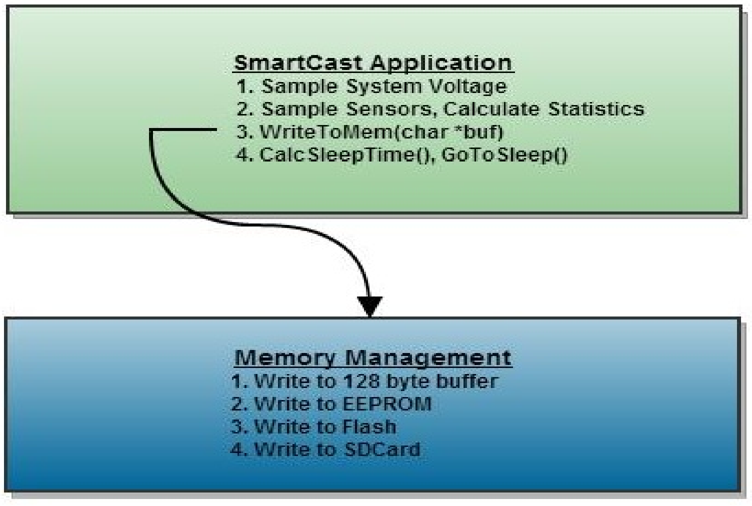


Figure 8: Software Architecture

As is described in section 3.1 Power Consumption & System Lifetime, the microcontroller internal memory is used as a buffer for the SD Card, thus saving energy. In order to free the

SmartCast application from the responsibility of handling multiple types of memory, the memory hierarchy is abstracted away into the Memory Management module. The SmartCast application converts the calculated sensor means and variances to a string representation and then calls the WriteToMem(char *buf) function, in which the string buffer is passed. The Memory Management module then handles how this data is written to the three types of memory. The data is always written first to a 128 byte buffer; this simplifies the code as Flash and EEPROM are organized in pages of 128 bytes. Once this buffer is full, the buffer's contents are copied to EEPROM. Then, when EEPROM becomes full, the buffer's contents are instead copied to Flash memory. Then, after approximately 45-50 wake events, Flash becomes full and the entire contents of EEPROM and Flash are written to the SD Card, as well as any extra data still contained in the 128 byte buffer. In this way, the SD Card is only written to every 45-50 wake events, thus allowing the SD Card to be turned off most of the time.

It is important to carefully handle the Flash memory writes as the internal Flash memory of the ATMega328p also contains the program code; it would be disastrous to blindly write to Flash disregarding the application section. **Figure 9** shows the memory map for the ATMega328p Flash memory. The bottom 4 pages of Flash Memory is a special section which contains the code that is able to write to internal memory. The application code resides in the top section of Flash, from address 0x0000 until *__data_load_end*, a symbol in the Linker .map file which must be read during run time in order to determine the last Flash address of the program code. Thus, the section of Flash that can be written to by the SmartCast application resides in the middle of the memory map, between the application and bootloader sections.

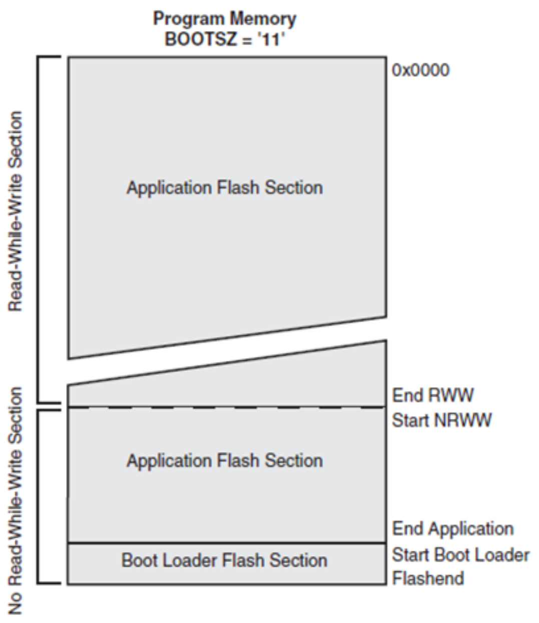


Figure 9: ATmega328p Program Memory Map[1]

The following is an example calculation showing how to find the free addresses of Flash memory which can store SmartCast application data:

- BOOTLOADER_PAGES = 4 (the bootloader code takes up 4 pages)
- SPM_PAGESIZE = 128 bytes (i.e. the size of a Flash page)
- FLASHEND = 0x7FFF (i.e. the last Flash address)
- __data_load_end = 24265_{10} (i.e. the program code is 24265_{10} bytes for example)
- Address_of_1st_Free_Page = __data_load_end rounded up to nearest multiple of 128 = 24320_{10}
- Address_of_Last_Free_Page = (FLASHEND + 1 – ((BOOTLOADER_PAGES + 1) * SPM_PAGESIZE)) = 32128_{10}

(add 1 to FLASHEND and BOOTLOADER_PAGES to get the address of the next free byte and page respectively)

- Available Flash Memory = (Address_of_Last_Free_Page +128) – Address_of_1st_Free_Page = 7936 bytes

2.4 Case & Form Factor

The SmartCast sensor sleeve is designed to be worn under the fiberglass or plaster cast. A PCB and case have also been designed to house the necessary electronics to sample and store the data from the sensors. This case is also intended to be worn on the arm during the months the cast is on the arm. **Figure 10** shows the case and PCB form factor designed in Solid Works 2010. Instead of having this custom case produced using injection molding, a similar off-the-shelf case was used.

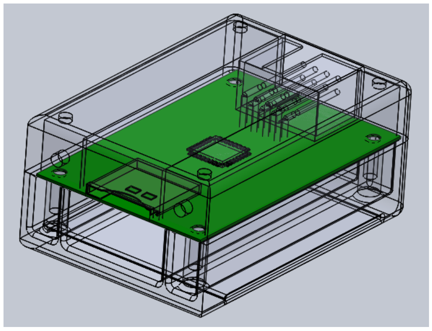


Figure 10: SmartCast Initial Case Design and PCB Form Factor

Figure 11 shows the prototype PCB with the SD Card and attached connections.

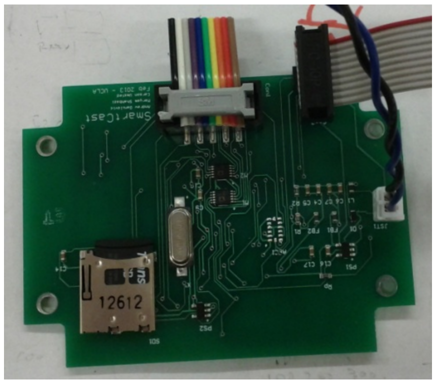


Figure 11: SmartCast Prototype PCB
(Designed by SmartCast team members Maryam Shahbazi and Carson Umsted and tested by Andrew Danilovic)

Chapter 3 - System & Sensor Validation

3.1 Power Consumption & System Lifetime

The manner in which memory is utilized directly affects system energy consumption because the three types of memory all have different write times and current draws. As can be seen from the data in **Table 1** (p. 19) the following shows the memory hierarchy in terms of energy cost:

Lowest Energy Cost - Internal Flash

Mid Energy Cost - Internal EEPROM

Highest Energy Cost - SD Card

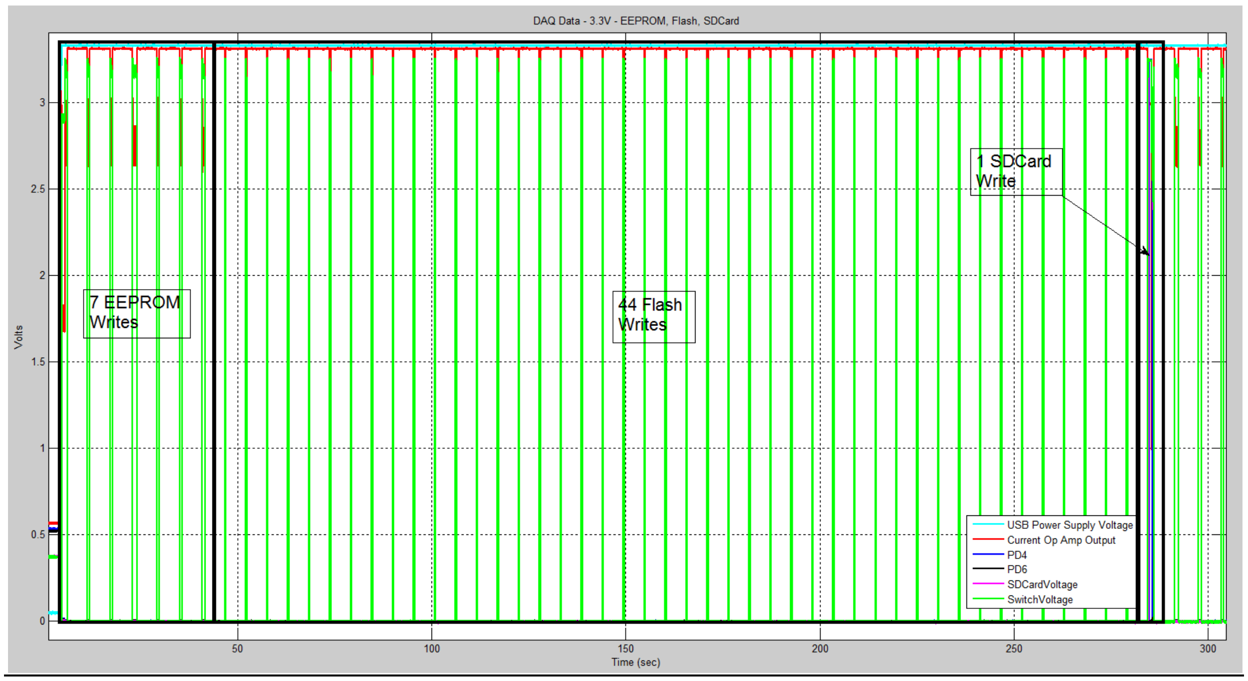


Figure 12: SmartCast Voltage Signals captured by NI DAQ

Figure 12 shows several SmartCast voltage signals captured by an NI DAQ, and shows multiple wake/sample/store events. **Figure 13** shows the system current draw for the same period of time. From this data, we can clearly see the differences in write time and current draw between the three types of memory. This data was obtained by using an op-amp current meter circuit[12] and a 1 Ω resistor. The equation to calculate current was then:

$$I_{dd} = \frac{V_{dd} - V_{out}}{10\frac{V}{V} * 1\Omega} \qquad \text{Eq. 1}$$

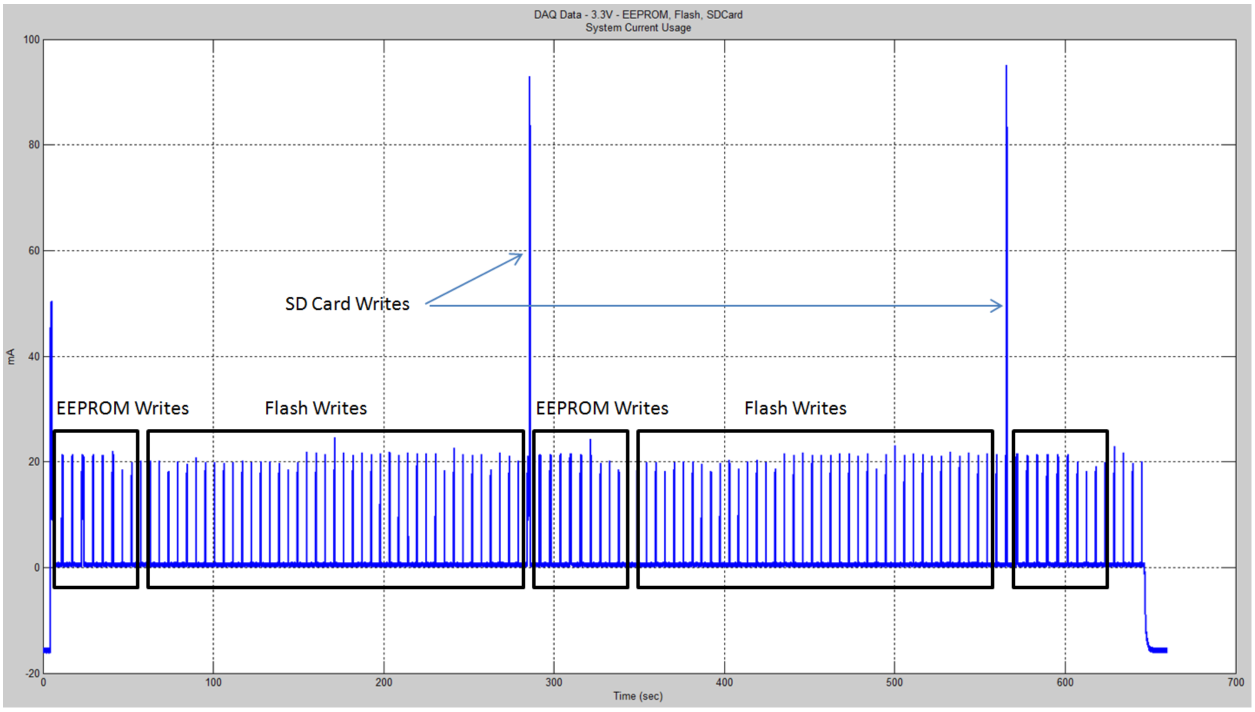


Figure 13: SmartCast System Current

From **Figure 12** and **Figure 13**, we can obtain the data in **Table 1**, which contains the current draw and write time for the three types of memory. In order to estimate system lifetime based on this data, Eq. 2 has been plotted in **Figure 14**.

$$Lifetime = num_{cycles} * \ duration_{cycle} \qquad \text{Eq. 2}$$

Eq. 2 assumes that the battery capacity is drained linearly with time. Real batteries have a more complicated discharge pattern, but the Ultimate Lithium brand used to power the SmartCast system are much more linear than regular Alkaline batteries.

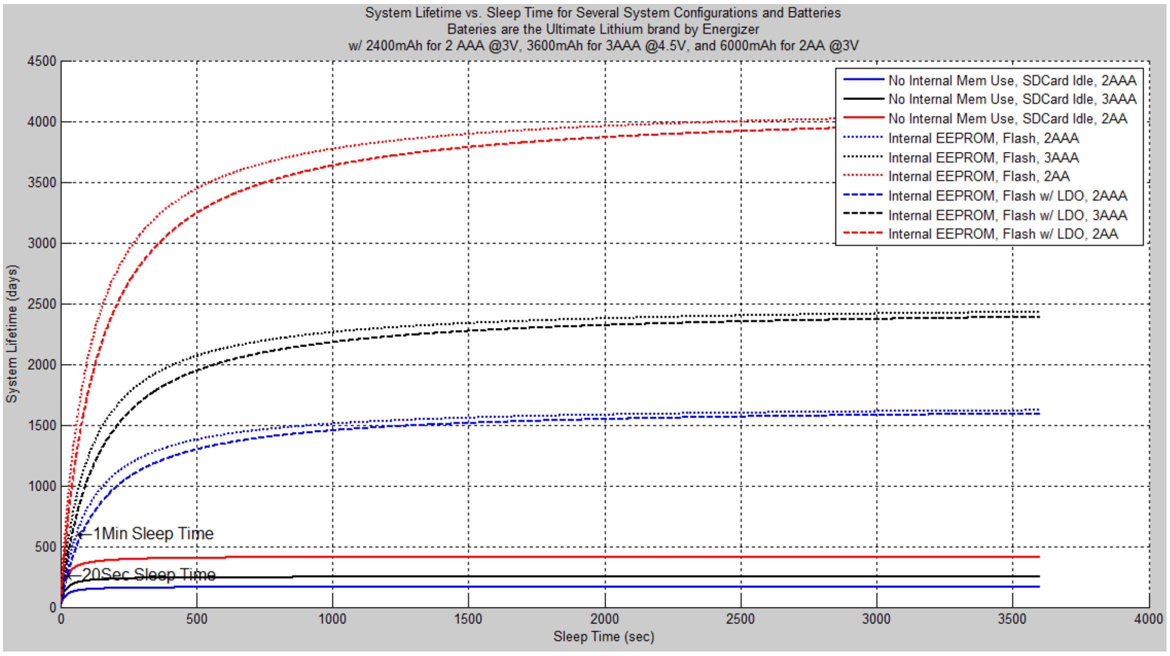


Figure 14: System Lifetime, Zoomed Out

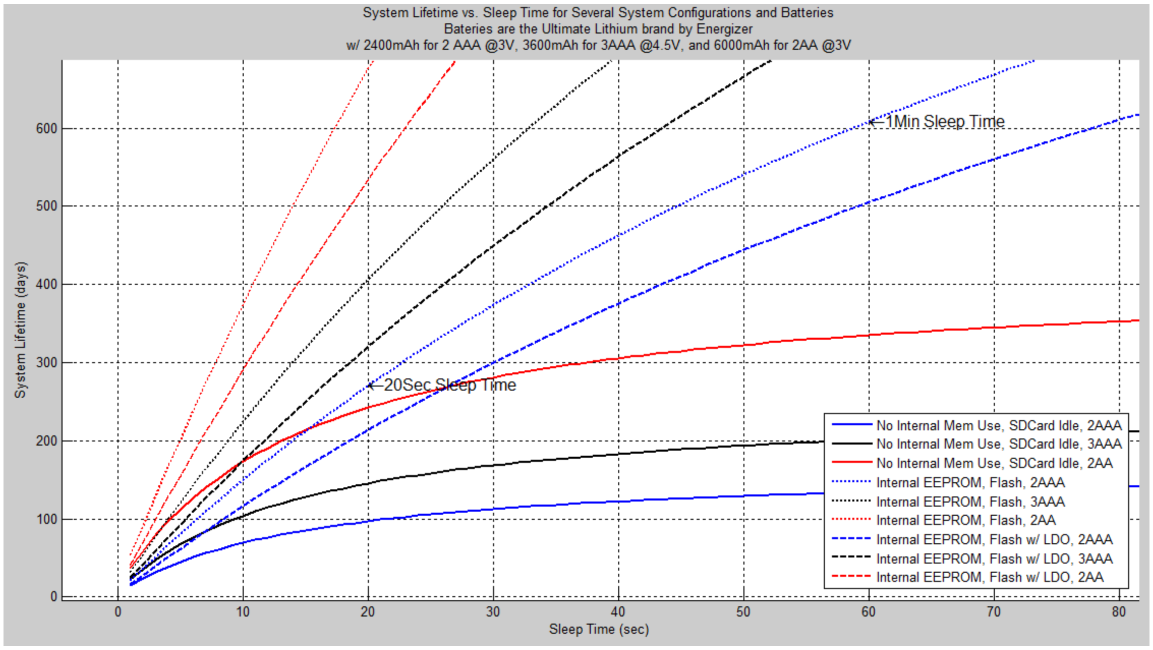


Figure 15: System Lifetime, Zoomed In

Eq. 2 shows the System Lifetime calculation. Here, the term "cycle" refers to the number of wake/sample/store/sleep events that occur before writing to the SD Card. Because of the size limitations of the internal memory on the ATMega328p, 1 cycle includes 7 wake/sample/store/sleep events where the internal EEPROM is written to, 44 wake/sample/store/sleep events where the internal Flash is written to, and 1 wake/sample/store/sleep event where the SD Card is written to. num_{cycles} is the number of such cycles, each one including the 7+44+1 wake/sample/store/sleep events, that occur before the battery capacity is depleted. The units of battery capacity in this discussion are mAh. $duration_{cycle}$ is the time duration of 1 cycle, i.e. the duration of the 7+44+1 wake/sample/store/sleep events.

$$num_{cycles} = \frac{Capacity_{battery}}{C_{Consumed_{EEPROM}} + C_{Consumed_{Flash}} + C_{Consumed_{SDCard}}} \quad \text{Eq. 3}$$

Eq. 3 shows the calculation of num_{cycles}. $C_{Consumed_{EEPROM}}$ is the battery capacity that is consumed for the 7 wake/sample/store/sleep events in which EEPROM is written to. Similarly, $C_{Consumed_{Flash}}$is the battery capacity that is consumed for the 44 wake/sample/store/sleep events in which Flash is written to. $C_{Consumed_{SDCard}}$ is the battery capacity consumed during the wake/sample/store/sleep event in which the SD Card is written to. Eq. 4 , 5, 6, and 7 show how these quantities are calculated. $num_{EEPROMCycles}$ and $num_{FlashCycles}$ are 7 and 44 respectively. I_{EEPROM} is the average current during the wake/sample/store event in which EEPROM is written to. Similarly, I_{Flash} is the average current during the wake/sample/store event in which Flash is written to. $I_{SDCardInit}$ and $I_{SDCardWrite}$ are the average currents during the wake/sample/store event in which the SD Card is written to.

$$C_{Consumed_{EEPROM}} = \left((I_{EEPROM} * t_{EEPROM}) + C_{Consumed_{Sleep}}\right) * num_{EEPROMCycles} \qquad \text{Eq. 4}$$

$$C_{Consumed_{Flash}} = \left((I_{Flash} * t_{Flash}) + C_{Consumed_{Sleep}}\right) * num_{FlashCycles} \qquad \text{Eq. 5}$$

$$\begin{aligned} C_{Consumed_{SDCard}} &= (I_{SDCardInit} * t_{SDCard}) + (I_{SDCardWrite} * t_{SDCard}) \\ &\quad + C_{Consumed_{Sleep}} \end{aligned} \qquad \text{Eq. 6}$$

$$C_{Consumed_{Sleep}} = I_{Sleep} * t_{Sleep} \qquad \text{Eq. 7}$$

Eq. 8 shows the calculation of $duration_{cycle}$.

$$\begin{aligned} duration_{cycle} = &\left((t_{EEPROM} + t_{Sleep}) * num_{EEPROMCycles}\right) \\ &+ \left((t_{Flash} + t_{Sleep}) * num_{FlashCycles}\right) \\ &+ (t_{SDCardInit} + t_{SDCardWrite} + t_{Sleep}) \end{aligned} \qquad \text{Eq. 8}$$

Eq. 2 is plotted in the figures on page 16, where t_{Sleep} has been varied from 1 second to 3600 seconds (1 hour).

From **Figure 14** we can see that system lifetime is a non-linear function of the sleep time. This is expected, because as you increase the sleep time, you achieve diminishing returns on the system lifetime because the system consumes a small amount of current while sleeping. Therefore, with very long sleep times, the system lifetime begins to level off. Furthermore, **Figure 15** shows the crossover point where it becomes better to turn the SD Card off instead of idle it. Turning the SD Card off decreases the memory current draw to 0 while it is off, but forces a longer initialization sequence to occur on wake up. The longer the system is asleep for, the better it becomes to turn the SD Card completely off as opposed to idling it, which draws a small amount of current.

<u>Data Derived from Figure 12 and Figure 13 and used to create Figure 14 and Figure 15</u>

All times are in seconds and all currents are in mA.

No Internal Mem Use, SDCard Idle						
I_sample	I_write	I_sleep	t_sample	t_write		
10	100	0.6	0.41	0.05		
Internal EEPROM, Flash						
I_EEPROM	I_Flash	I_SDCard_Init	I_SDCard_Write	I_Sleep		
15	12	10	100	0.06		
t_EEPROM	t_Flash	t_SDCard_Init	t_SDCard_Write			
0.7	0.3	0.7	0.9			
Internal EEPROM, Flash w/ LDO						
I_EEPROM	I_Flash	I_SDCard_Init	I_SDCard_Write	I_Sleep	I_LDO	I_LDO_Sleep
15	12	10	100	0.06	0.008	0.0005
t_EEPROM	t_Flash	t_SDCard_Init	t_SDCard_Write			
0.7	0.3	0.7	0.9			

Table 1: System Memory Write Current & Time Data

3.2 Cyclic Mass Test

3.2.1 Experimental Procedure

In order to put the textile sensors through rigorous testing, Carson Umsted designed and built an apparatus which can apply a sine wave force to the sensors for an extended period of time. This apparatus is shown in the right side of **Figure 16**.

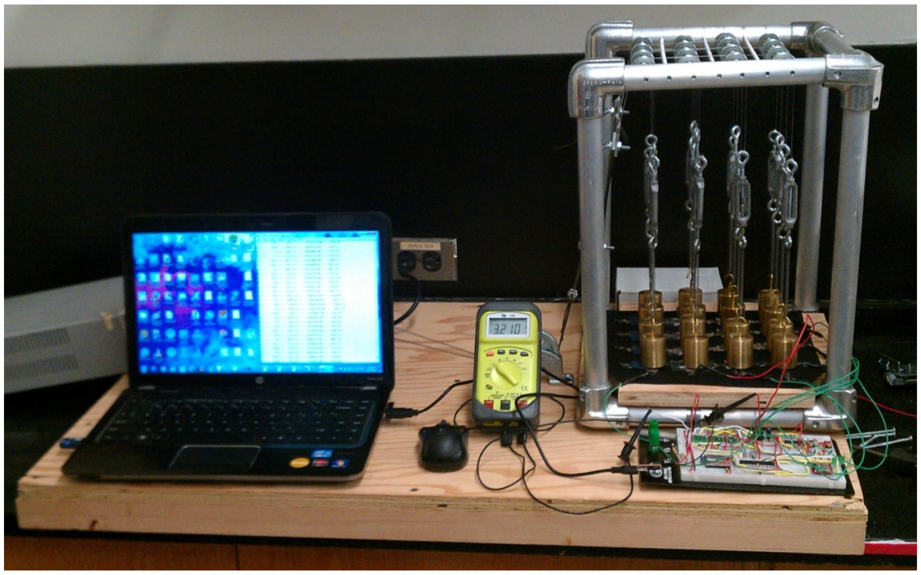


Figure 16: Cyclic Test Experimental Set-Up

This apparatus contains a DC motor which pulls 16 200g masses as it rotates a shaft. This causes the masses to move up and down with sine wave behavior. This test does not utilize the full dynamic range of the sensors as the masses were not lifted completely off the sensors each cycle, but does provide a useful testing platform for both the system and sensor matrix.

3.2.2 Results

Figure 17 shows the results from the cyclic mass test performed on the most recent sensor sleeve prototype with 2 layers of Velostat as the piezoresistive sensing element and shows the resistance of all 16 sensors vs. time. This test ran for approximately 7 days continuously.

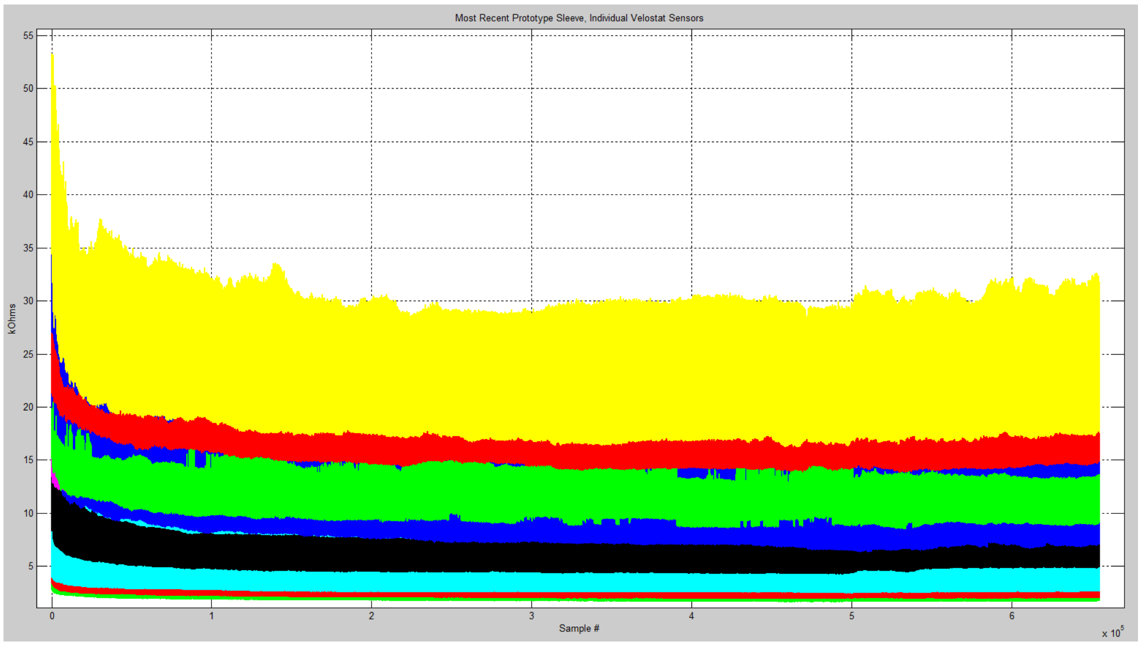


Figure 17: Cyclic Mass Test Results for Prototype 1 Sleeve, Approx. 7 days

Figure 18 shows a zoomed in view of **Figure 17** and clearly shows the cyclic behavior of the masses as the motor shaft rotates. We can see from this figure that there is a difference between the output range of each sensor, but all are in the same order of magnitude. We therefore cannot compare the output from different sensors directly, as each sensor gives relative, not absolute, information on the force being applied.

In **Figure 19**, the peaks and troughs of the individual sine waves have been plotted in black using a peak detector, clearly showing the envelope of each sensor for this 7 day test. All 16 sensors exhibit a decrease in their output for the first several hours. I conclude from observation of the test apparatus that this is due to the relaxation of the springs holding the masses, i.e. the test apparatus itself affects the results.

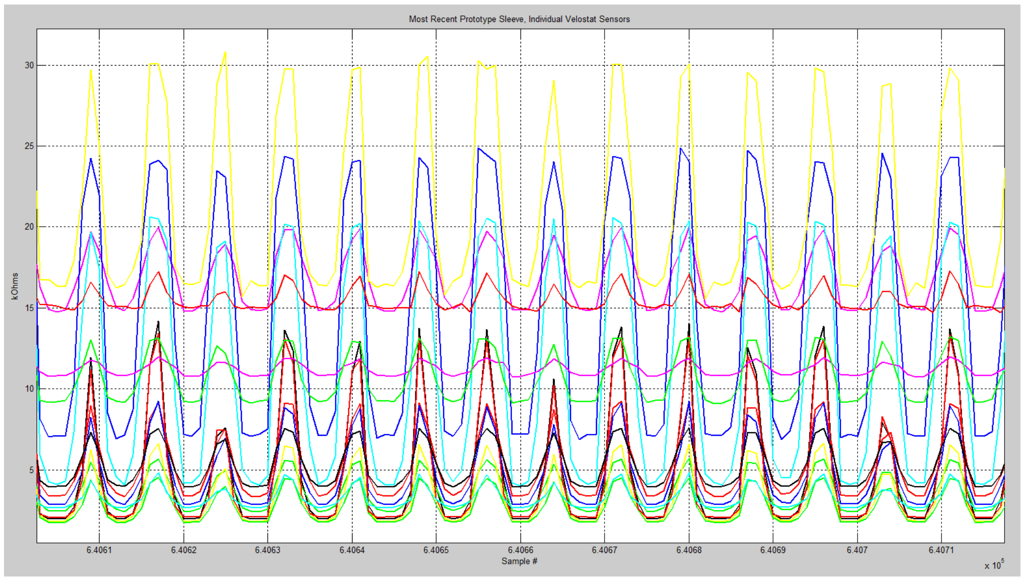


Figure 18: Zoomed in View of **Figure 17**

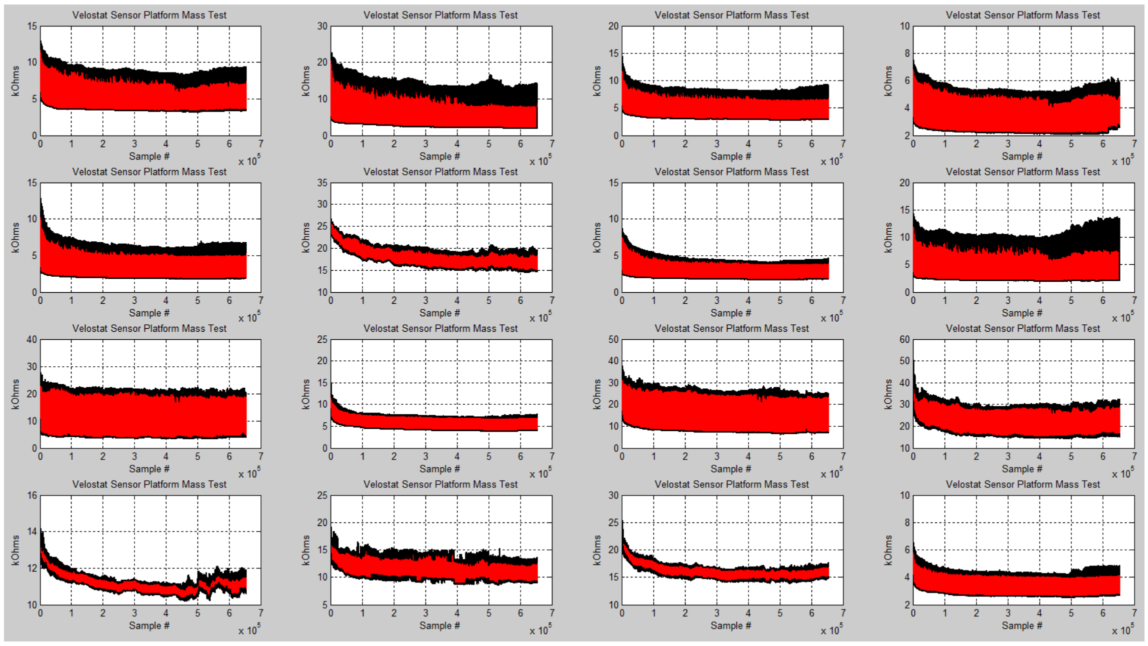


Figure 19: Data from **Figure 17** w/ Each Sensor Plotted Independently

3.3 Platform Test with Masses

3.3.1 Experimental Procedure

In order to select the appropriate material and number of layers for the piezoresistive element, this test was carried out to determine the resistance vs. force applied curves for each type of sensor. This test used the platforms shown in **Figure 20**, which consist of a plastic base covered by a soft cotton layer, to mimic the environment under a cast. The different piezoresistive materials were then placed between these two platforms, and known masses were placed on top while measuring the sensor output using the SmartCast system. In this way, the resistance vs. mass curves shown in the next section were obtained. The X-axis can easily be changed to Force by multiplying by the acceleration due to gravity.

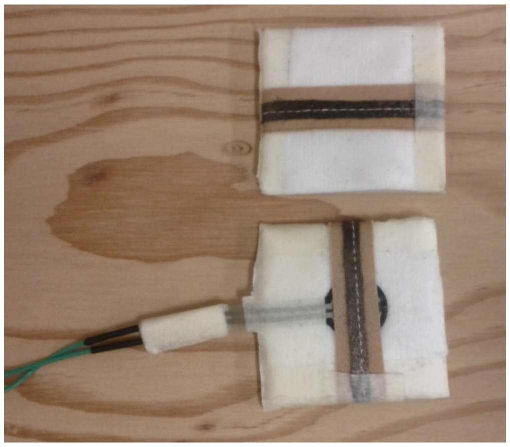


Figure 20: Mass Test Platforms

3.3.2 Results

The following types of sensors were tested in this experiment:

Sensor 1. 2 & 3 layers of polyester w/ carbon fibers, **Figure 21** & **Figure 22**

Sensor 2. 2 & 3 layers of polyester w/ carbon fibers covered w/ nylon, **Figure 23** & **Figure 24**

Sensor 3. 1, 2, & 3 layers of Velostat w/ and w/o nylon, **Figure 25**

Sensor 1 was the first type of sensor constructed. As can be seen in **Figure 21**, 3 layers has a larger range in resistance vs. mass than 2 layers, but 3 layers of material exhibits large variability between sensors. **Figure 21** clearly shows the 1/x behavior of the sensors with respect to resistance, and **Figure 22** shows that the sensors are linear with respect to conductance. This was shown to be true for all the types of sensors tested in this experiment.

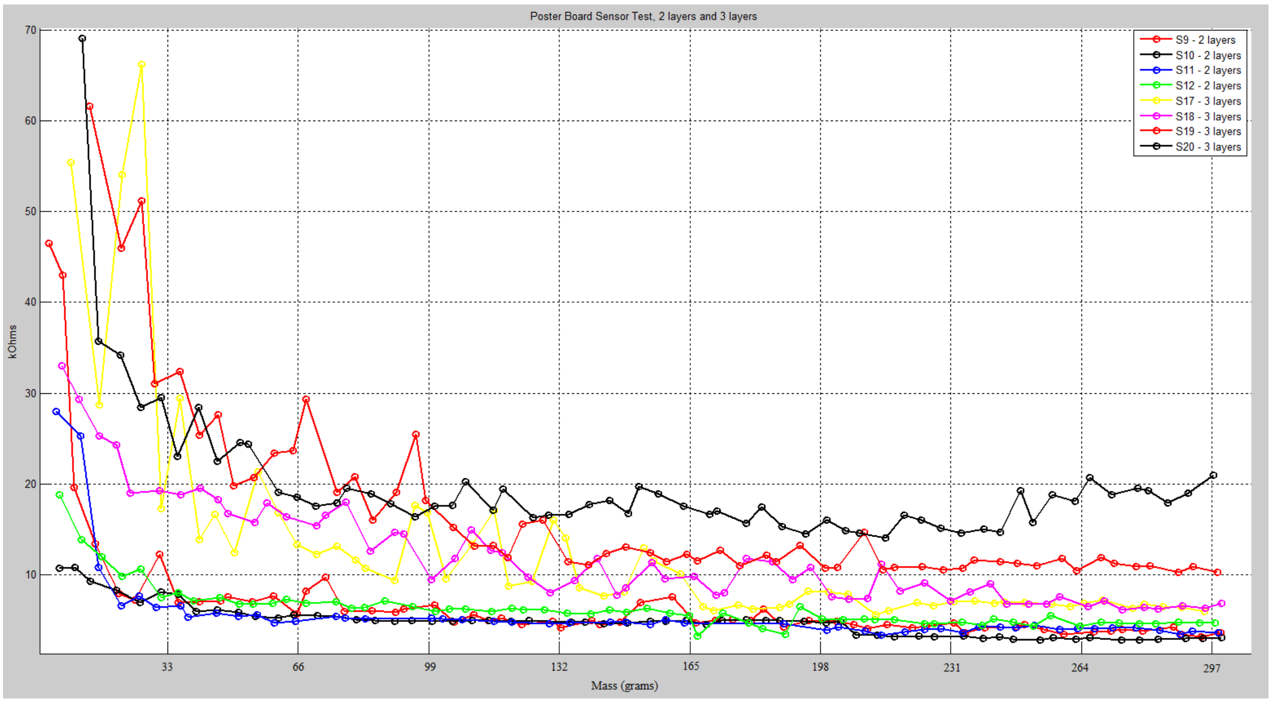


Figure 21: Polyester-Carbon, 2 & 3 Layers, Mass Test, Resistance

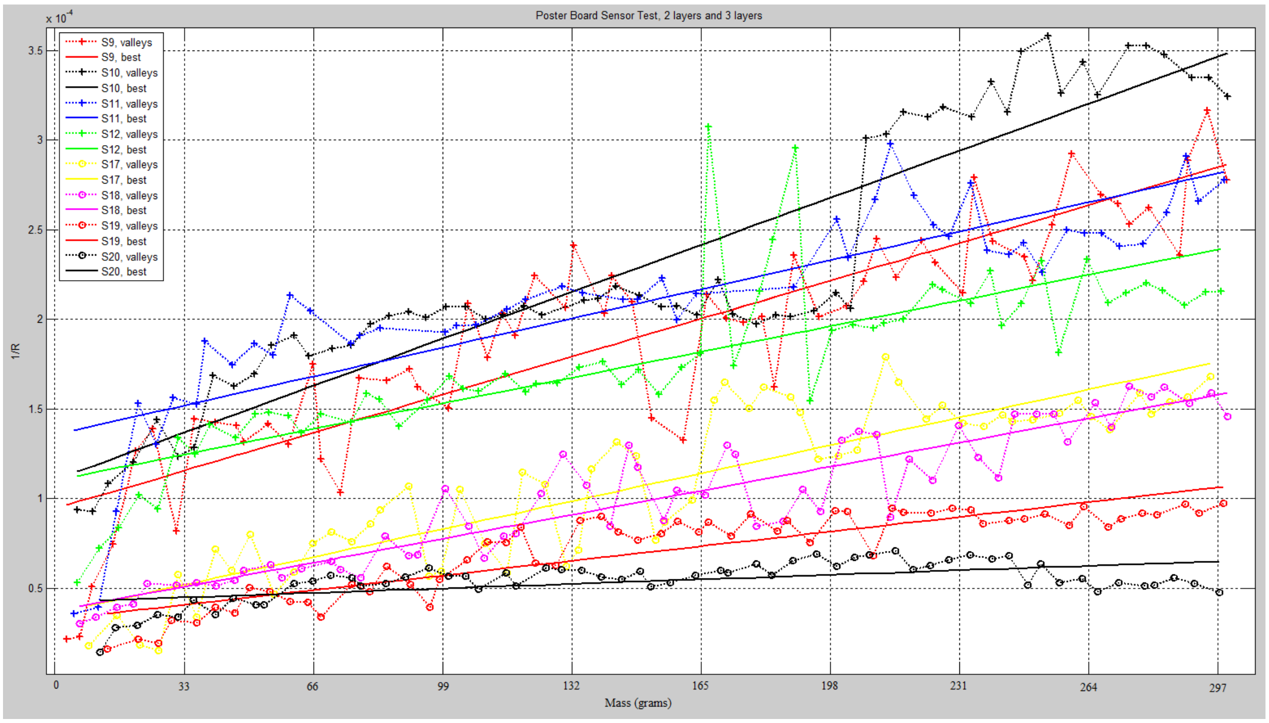


Figure 22: Polyester-Carbon, 2 & 3 Layers, Mass Test, Conductance

Figure 23 and **Figure 24** show the data for the same sensors, but with a top and bottom layer of nylon. It was thought that adding the nylon layers would decrease the variability between sensors by keeping the sandwich of materials tightly held together. The data shows however that this did not reduce the variability between sensors as expected.

Due to the large variability between individual sensors constructed out of polyester with carbon fibers, another piezoresistive material, Velostat, was chosen. From the data in **Figure 25**, we can see that 2 layers of Velostat with a top and bottom layer of nylon had a suitable resistance range and had less variability between sensors then the other combinations tested so far.

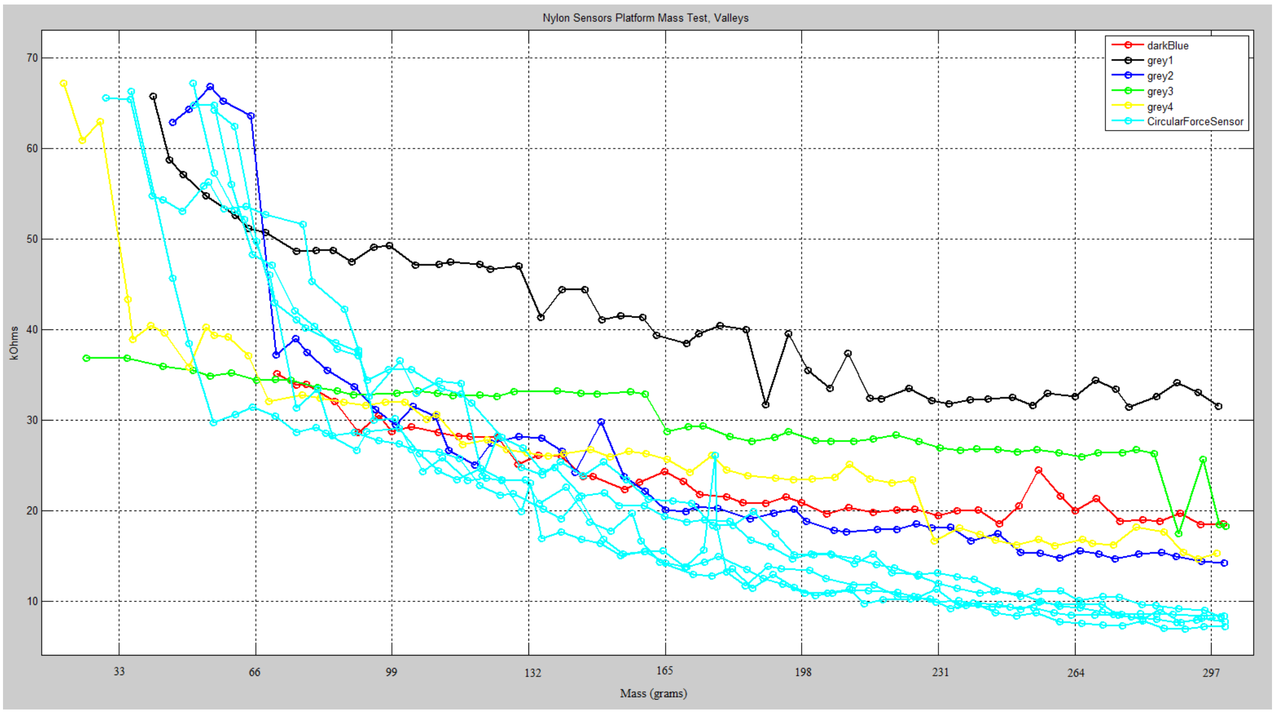


Figure 23: Polyester-Carbon-Nylon, 2 Layers, Mass Test, Resistance

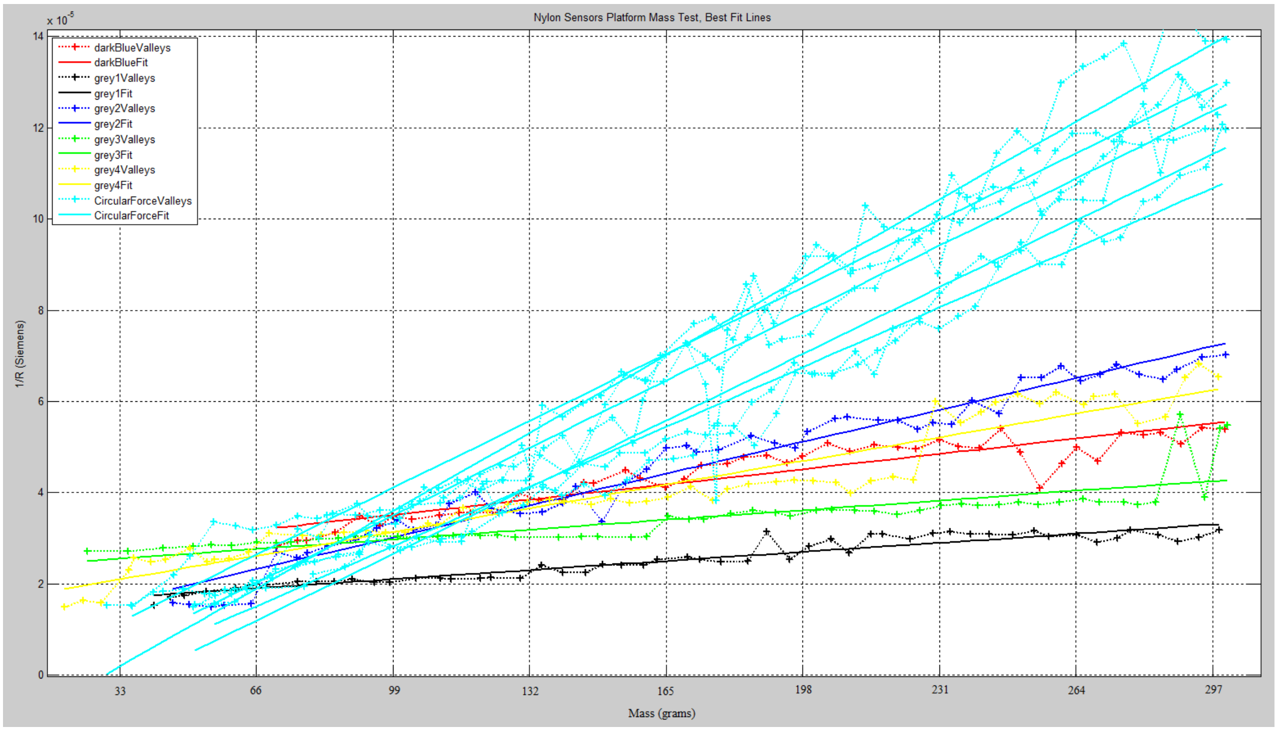


Figure 24: Polyester-Carbon-Nylon, 2 Layers, Mass Test, Conductance

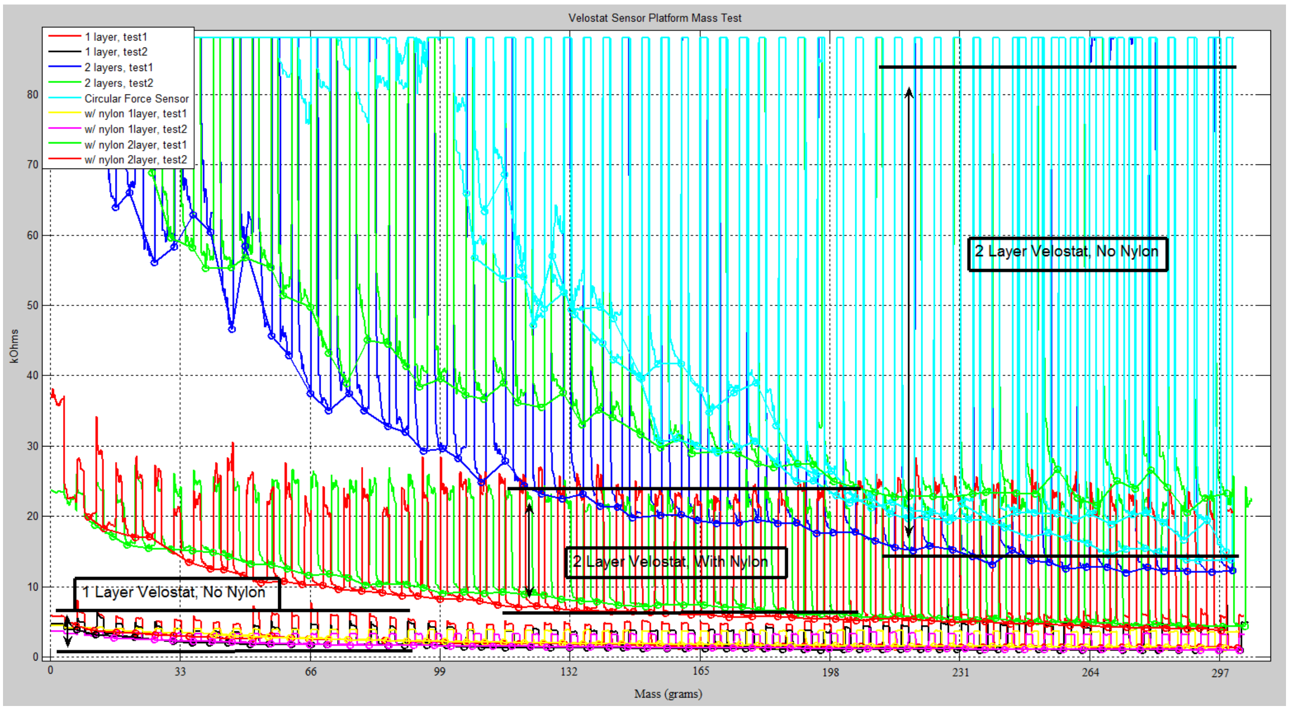


Figure 25: Velostat, w/ and w/out Nylon, 1, 2, & 3 Layers, Mass Test

3.3.3 Sensitivity and Resolution

The sensitivity of a sensor can be defined as the amount of change in the output per unit of change in the input. The sensitivity of the SmartCast fabric sensors can therefore be obtained as the slopes of the curves in **Figure 21**, **Figure 23**, and **Figure 25**, as the Y-axis in these plots represents the output (resistance) and the X-axis represents the input (applied mass or force). Since these slopes are clearly not constant, the sensitivity of these sensors to applied force varies with the force being applied, i.e. the sensors are more sensitive when less force is applied and are less sensitive when more force is applied. It is difficult to obtain the sensitivities from the plots of conductance vs. applied mass, as these plots mask the fact that the sensitivities change with applied force.

The force resolution of the SmartCast system plus textile sensor can be obtained from the sensor sentivitiy and from the resolution of the analog interface. The SmartCast analog interface, comprised of the transimpedance amplifier and 10-bit ADC, has a resolution of 49 Ohms per LSB, as the following calculation shows:

$$R_{sensor} = \left(\frac{1V - V_{out}}{0.25}\right) * 12.53\ kOhms \qquad \text{Eq. 9}$$

$$analog\ interface\ resolution = \left(\frac{1V - (1023/1024)}{0.25}\right) * 12.53\ kOhms$$
$$= 48.95\ Ohms \qquad \text{Eq. 10}$$

Eq. 9 is the equation describing the transimpedance amplifier in the analog interface (see Appendix A), and Eq. 10 shows the calculation to obtain the resolution of the analog interface. The applied force resolution can then be obtained as:

$$applied\ force\ resolution\ [Newtons] = \frac{analog\ interface\ resolution\ [Ohms]}{sensitivity(applied\ force)\ \left[\frac{Ohms}{Newtons}\right]} \qquad \text{Eq. 11}$$

Eq. 11 can be used to obtain how well the SmartCast system can resolve applied force. Function notation parentheses have been used in the denominator of Eq. 11 to illustrate that the sensitivity is a function of applied force and brackets have been used to show the units of each term.

In order to model the sensitivity of the sensors, the Matlab command polyfit was used to fit polynomial curves to the curves in **Figure 25**, which shows the output vs. input for the velostat fabric sensors. The command polyfit finds the coefficients of a polynomial that fits the data in a least squares sense. **Figure 26** shows the best fit polynomial curves of order 3.

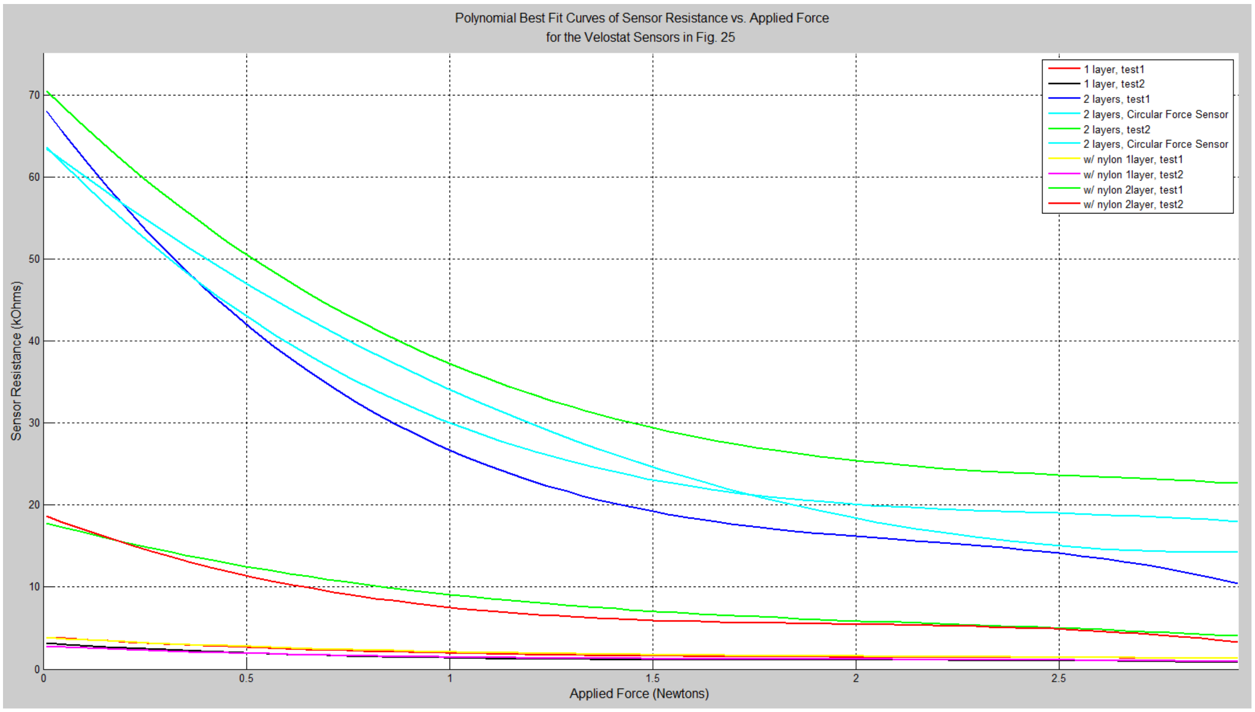


Figure 26: Polynomial Best Fit Curves of Rensor Resistance vs. Applied Force for the Velostat Sensors in **Figure 25**

This is essentially the same plot as **Figure 25** but with best fit polynomial curves and with the x-axis rescaled to be in units of Newtons. Now, to obtain the sensitivity of the velostat sensors, the derivatives of the best fit curves in **Figure 26** were taken to obtain the corresponding slopes as they change with applied force. These slopes are plotted in **Figure 27**.

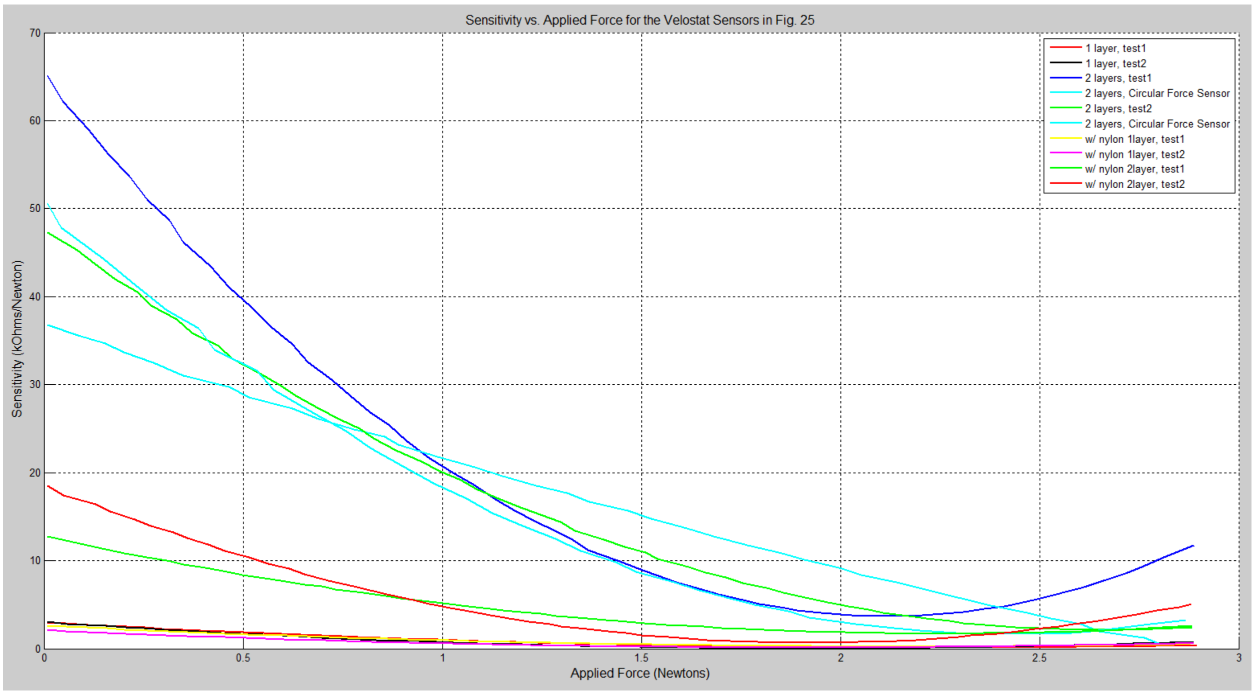


Figure 27: Sensitivity vs. Applied Force for the Velostat Sensors in **Figure 25**

Figure 27 illustrates that the sensitivity of the velostat sensors decreases quickly when more force is applied. The sensor resolutions were obtained from the Eq. 11 by dividing the analog interface resolution, 49 Ohms, by the sensitivity, and are shown in **Figure 28**. Here, the y-axis is the resolution of a sensor, i.e. the smallest change in applied force that the sensor can resolve. Lower values of resolution are better, i.e. lower values of resolution indiciate that the sensor is better able to resolve applied force. The poles centered around 2 Newtons in **Figure 28** are due to the error in estimating the curves in **Figure 25** using polynomials of order 3. It is perhaps more useful to look at **Figure 29**, which is just a zoomed in view of **Figure 28**, showing when the resolution starts to worsen as more force is applied.

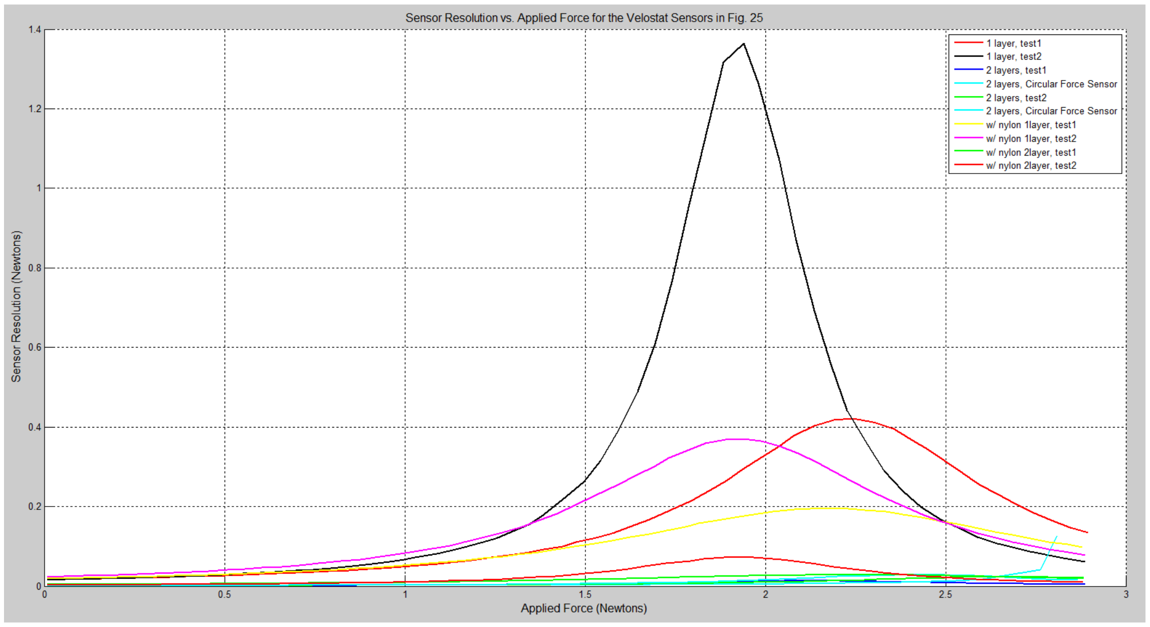


Figure 28: Sensor Resolution vs. Applied Force for the Velostat Sensors in **Figure 25**

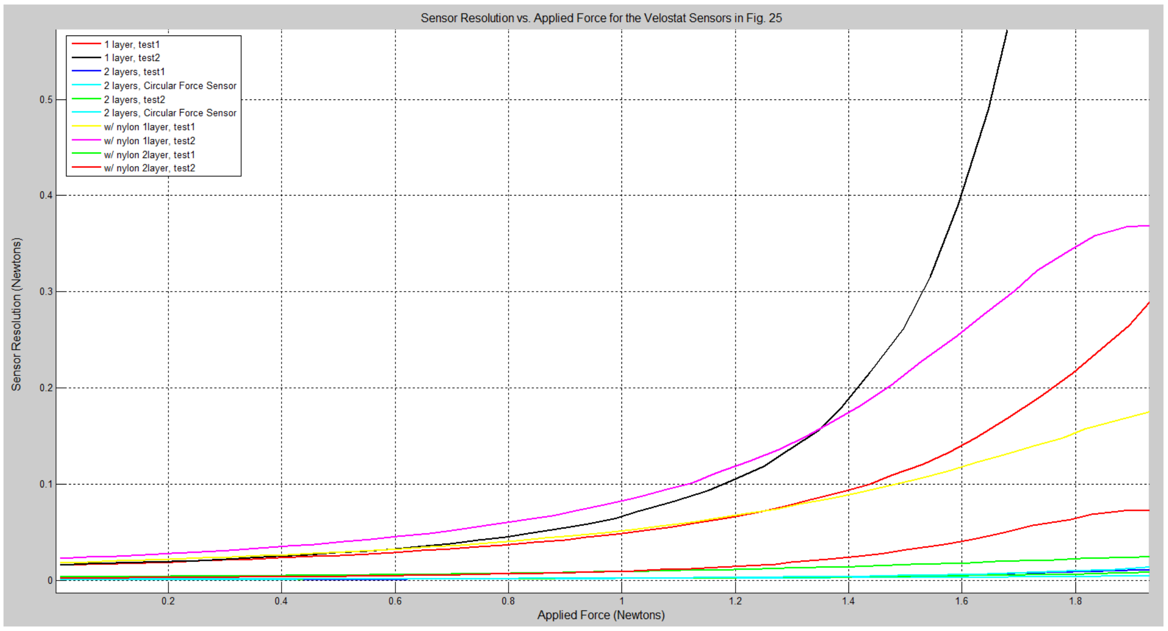
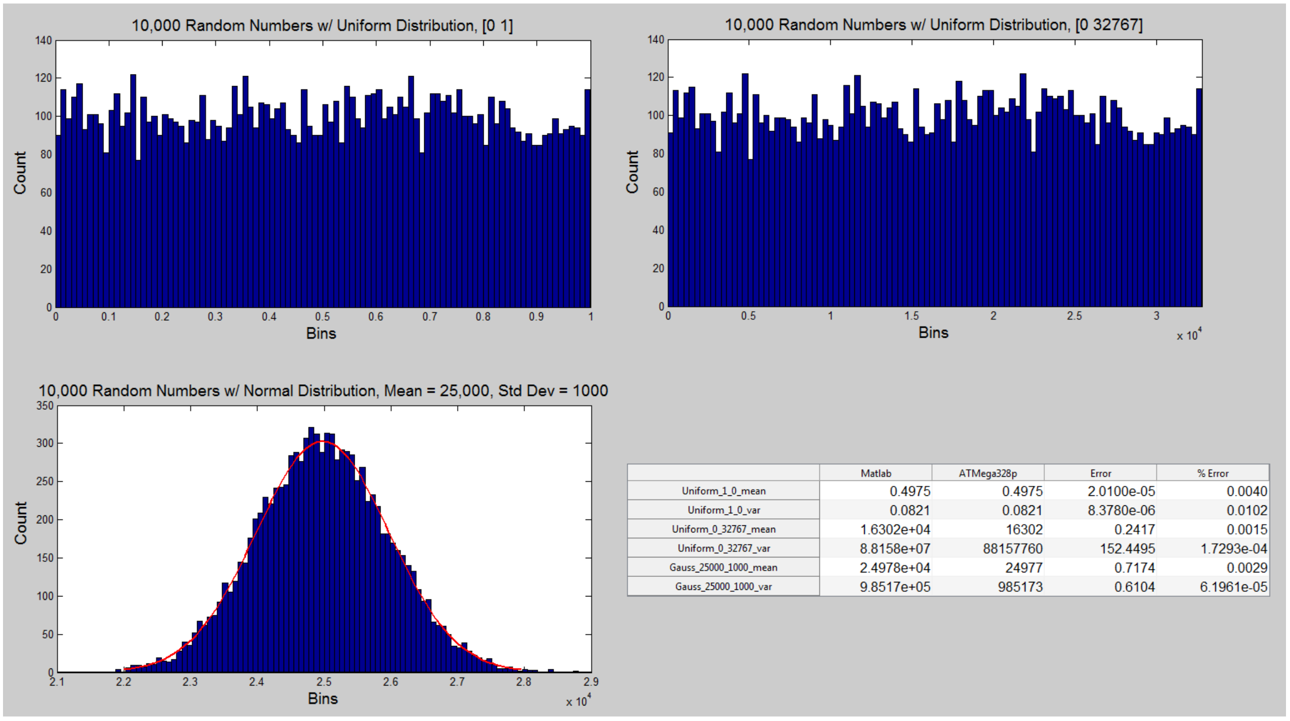


Figure 29: Zoomed in View of **Figure 28**

3.4 Online Statistics Calculations

On-line calculation of the variance of each set of samples was done in order to record a measure of the validity of a set of samples. If the variance is high for a particular sensor reading, this can mean either there is a problem with the sensor or the subject is currently moving. Throwing out samples that contain high variance is also a way to analyze samples that were taken when the subject was at rest; samples taken while the subject is resting should give a better indication of the state of the cast.



	Matlab	ATMega328p	Error	% Error
Uniform_1_0_mean	0.4975	0.4975	2.0100e-05	0.0040
Uniform_1_0_var	0.0821	0.0821	8.3780e-06	0.0102
Uniform_0_32767_mean	1.6302e+04	16302	0.2417	0.0015
Uniform_0_32767_var	8.8158e+07	88157760	152.4495	1.7293e-04
Gauss_25000_1000_mean	2.4978e+04	24977	0.7174	0.0029
Gauss_25000_1000_var	9.8517e+05	985173	0.6104	6.1961e-05

Figure 30: Online Statistic Calculation Verification

In order to verify the online calculations done by the SmartCast system, three sets of random numbers were generated and then run through the online algorithm on the SmartCast platform. **Figure 30** shows the three sets of random numbers generated. **Figure 30** also contains a table of data which compares the SmartCast online statistic algorithm with Matlab routines that perform the same calculations. As can be seen in **Figure 30**, there is very little error between the Matlab routines and the SmartCast online algorithm. Furthermore, as the mean and variance of only 10 samples is calculated at one time and then the algorithm reinitialized, the variables cannot grow unbounded and therefore overflow is not possible.

3.5 Forces Under a Cast

In order to determine the range of forces that must be detected under a loose fitting cast, a pre-made plaster cast was put over a subject's arm for a short duration of time while wearing a SmartCast sensor. 7 trials were performed, each one with a different SmartCast sensor being worn on the top of the arm as well as an Interlink Force Sensing Resistor (FSR) as a reference. The pre-made cast was cut open, placed over the subject's arm, and secured using adhesive tape. As the cast was pre-made, it was not snug against the subject's arm, and so this experiment should simulate the conditions of an ill-fitting, or loose, cast.

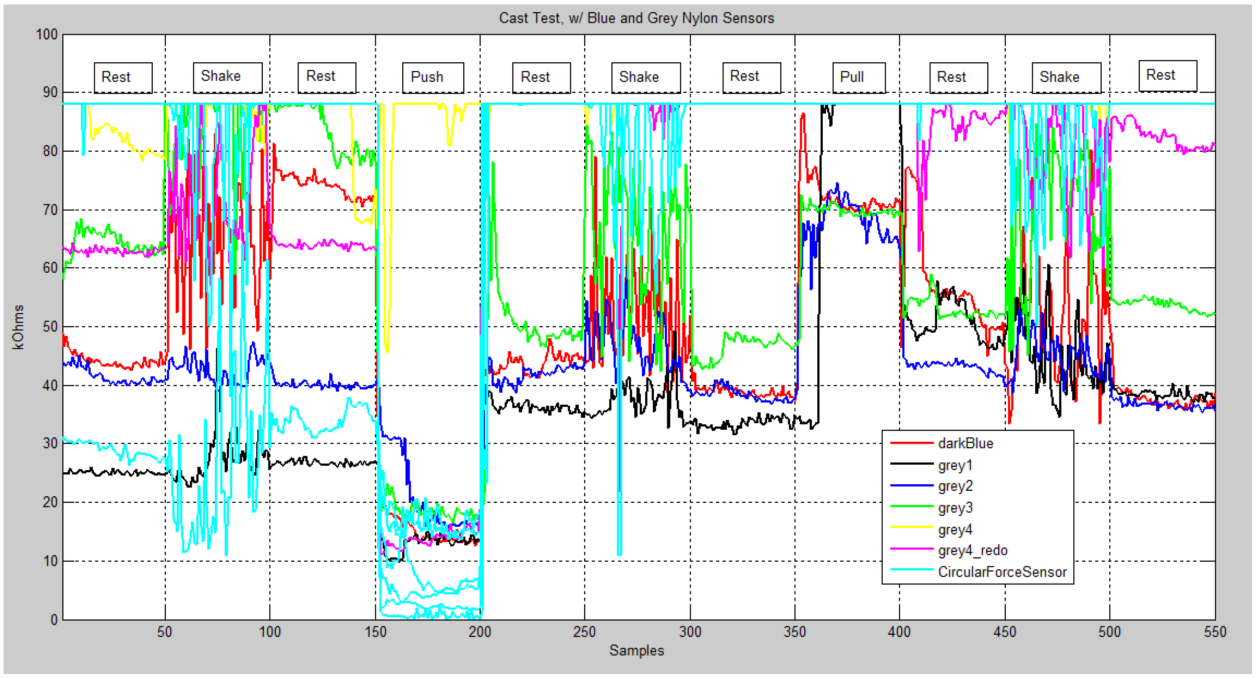


Figure 31: Loose-Fitting Cast Test

Figure 31 shows the data from this experiment. The y-axis is the output of the sensors worn during the test and the x-axis is the sample number, with 1 sample taken approximately every second. The light blue lines correspond to the the FSR sensor by Interlink, used as a reference for the SmartCast sensors, and the colored lines correspond to the different SmartCast sensors worn as well.

For the 1st 50 seconds, the subject held his arm still, i.e. a rest period. For the 2nd 50 seconds, the subject shook his arm to see the effects movement and vibration would have on the sensor output. This was followd by another rest period for 50 seconds. Then, the experiment administrator physically pushed down on the spot directly over where the sensor was located to simulate a very tight fitting cast. The process continued for the duration of the experiment.

Figure 32 shows the Resistance vs. Applied Mass or Force curve for Interlink's FSR sensor. This curve will be used to obtain the forces under the cast in the experiment explained above.

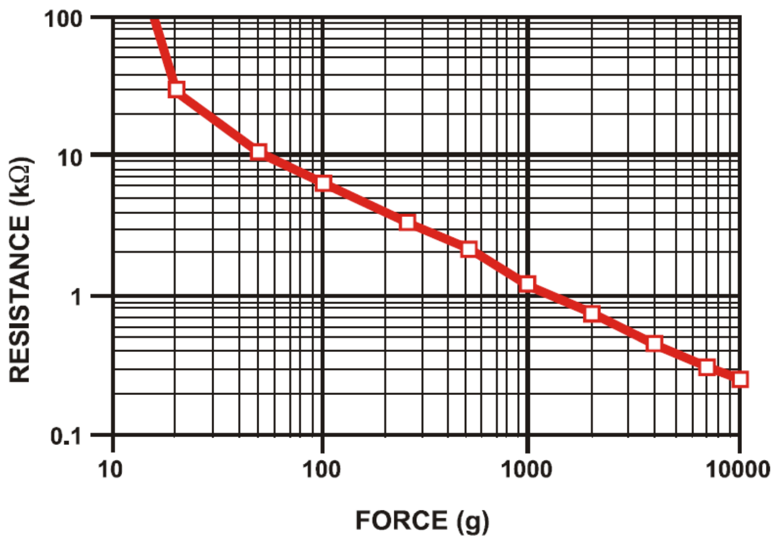


Figure 32: Resistance vs. Applied Mass for Interlink Force Sensing Resistor® (FSR®)[21]
(The Interlink datasheet uses the term "Force" but the x-axis in the plots are in units of grams.)

Before the 1st shake period, when the cast was fitting reasonably well, at least the best it did during this experiment with a pre-made cast, the output of the FSR during one of the trials was approximately 30kOhms. From **Figure 32**, this would correspond to approximately

$0.02 \text{ kgrams } * \ 9.8\frac{\text{m}}{\text{s}^2} = \ .196$ Newtons of applied force. During the push period, which should simulate a very tight fitting cast, the output of the FSR was approximately 10 kOhms, which from **Figure 32** corresponds to approximately $0.05 \text{ kgrams } * \ 9.8\frac{\text{m}}{\text{s}^2} \ = \ 0.49$ Newtons of applied force. After the 1st shake and push period, the output of the FSR sensor, illustraed by the light blue lines in **Figure 31**, rises above 90 kOhms, the limit of detection of the SmartCast analog interface. Looking at **Figure 32**, this would correspond to very little applied force, approximately $0.01\text{kgrams } * \ 9.8\frac{\text{m}}{\text{s}^2} \ = \ 0.098$ Newtons. This experiment therefore demonstrates that the forces under a cast can range from approximately 0.098 N when very loose and ill-fitting to 0.49 N when worn very tightly.

Chapter 4 - Sensor Network Fix

4.1 Problem Specification

Figure 33 shows the SmartCast resistive sensor network equivalent circuit. Each resistor corresponds to a textile sensor on the sleeve.

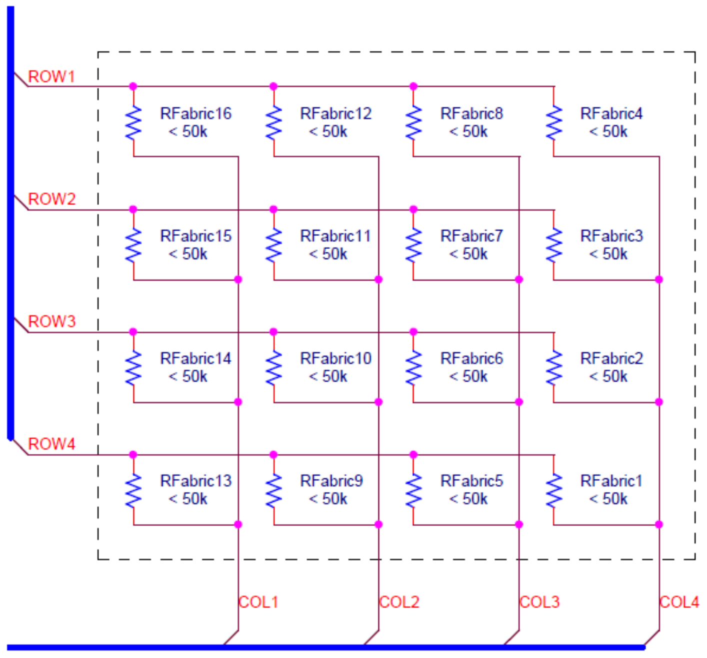


Figure 33: The SmartCast 4x4 Resistive Sensor Network Equivalent Circuit

Figure 34 shows the intended path through the network when the Row1 and Column1 switches are turned on and all other switches are open.

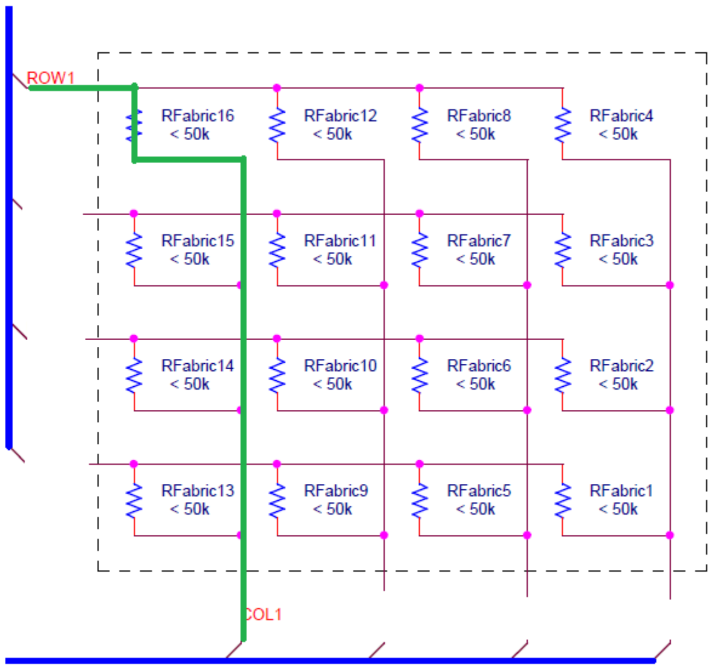


Figure 34: How to Select Sensor RFabric16

Figure 35 shows some of the other paths through the network. These paths exist even though the switches for the other rows and columns are open. The additional paths mean that the resistance measured is not the actual resistance of a single sensor, but of several sensors in parallel.

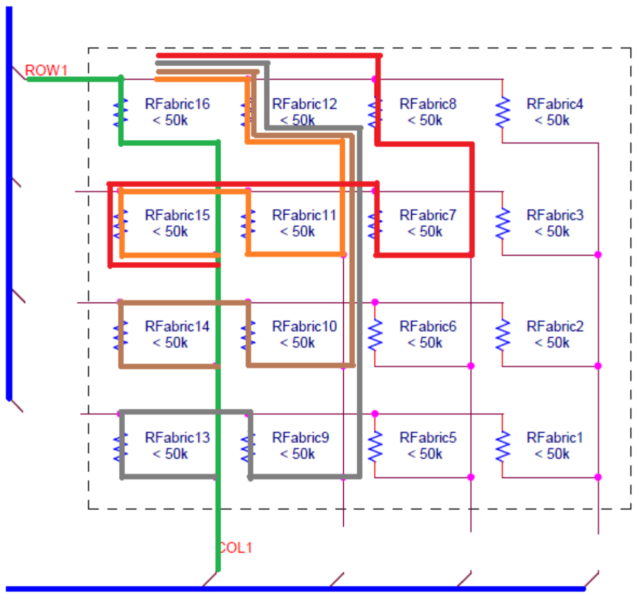


Figure 35: Additional Paths in Sensor Network

There are actually nine paths in **Figure 35** in addition to the intended path through R16. **Figure 36** shows the equivalent circuit for all 10 paths. There are 16 of these equivalent circuits, 1 for each of the possible switch combinations. V_{adc} is the voltage that is measured using the ADC of the microcontroller.

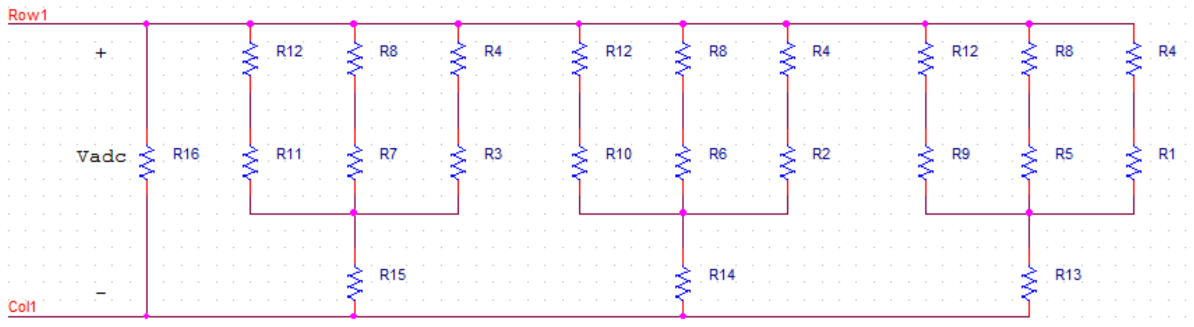


Figure 36: Equivalent Circuit for a Single Path

Figure 37 shows the equivalent circuit for a 2x2 network. The following derives the equations for the equivalent resistance with the goal of calculating the actual sensor resistances despite the additional confounding paths.

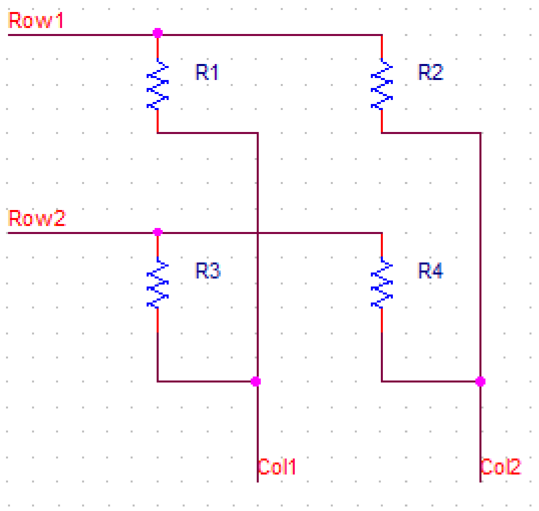


Figure 37: Equivalent Circuit for a 2x2 Sensor Network

Figure 38 shows all the intended paths through the 2x2 network shown in **Figure 37**, i.e. the paths correspond to the correct measurements of each sensor resistance.

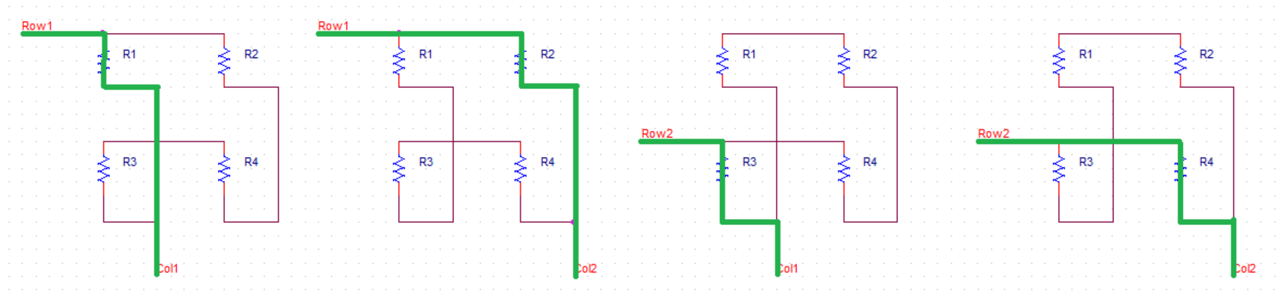


Figure 38: All 4 Intended Paths in 2x2 Network

Figure 39 shows that here, there is only 1 additional path for each of the 4 combinations of switches, i.e. when a particular sensor is selected, there is only 1 additional path.

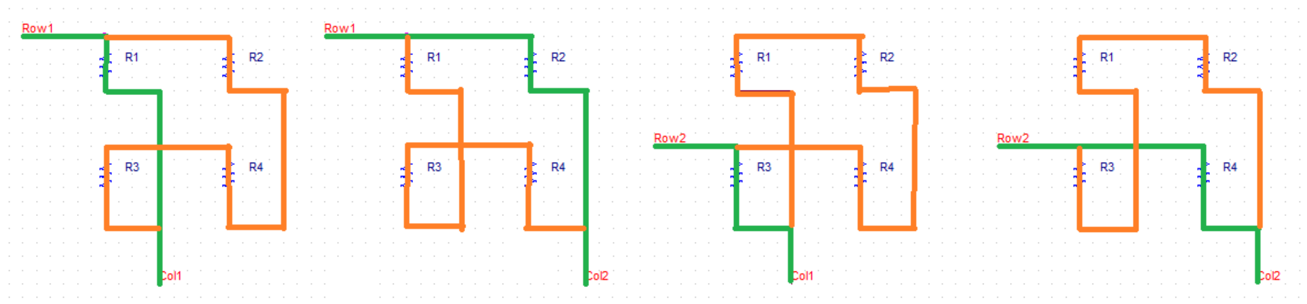


Figure 39: All Additional Paths in 2x2 Network

Figure 40 shows the equivalent circuits for the 2x2 network.

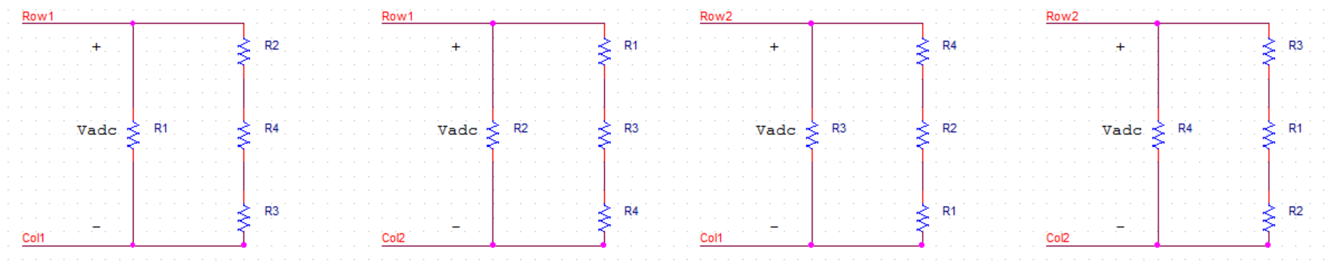


Figure 40: The Equivalent Circuits for 2x2 Network

Eq. 12 through Eq. 15 show the equivalent resistances for the circuits in **Figure 40**.

$$\frac{1}{R_{eq1}} = \frac{1}{R_1} + \frac{1}{R_2 + R_3 + R_4} \qquad \text{Eq. 12}$$

$$\frac{1}{R_{eq2}} = \frac{1}{R_2} + \frac{1}{R_1 + R_3 + R_4} \qquad \text{Eq. 13}$$

$$\frac{1}{R_{eq3}} = \frac{1}{R_3} + \frac{1}{R_1 + R_2 + R_4} \qquad \text{Eq. 14}$$

$$\frac{1}{R_{eq4}} = \frac{1}{R_4} + \frac{1}{R_1 + R_2 + R_3} \qquad \text{Eq. 15}$$

Eq. 16 through Eq. 19 are rearranged to try and solve for the equivalent resistances. At first glance, it seems that with 4 equations and 4 unknowns it should be possible to solve for each R_{eq}, but in practice this was not possible as these equations are non-linear.

$$R_{eq1} = \frac{R_1(R_2 + R_3 + R_4)}{R_1 + R_2 + R_3 + R_4} \qquad \text{Eq. 16}$$

$$R_{eq2} = \frac{R_2(R_1 + R_3 + R_4)}{R_1 + R_2 + R_3 + R_4} \qquad \text{Eq. 17}$$

$$R_{eq3} = \frac{R_3(R_1 + R_2 + R_3)}{R_1 + R_2 + R_3 + R_4} \qquad \text{Eq. 18}$$

$$R_{eq4} = \frac{R_4(R_1 + R_2 + R_3)}{R_1 + R_2 + R_3 + R_4} \qquad \text{Eq. 19}$$

Eq. 20

Figure 41 shows a Matlab script which tries to solve for the equivalent resistances for a 2x2 network using the symbolic toolbox. The output however is "Warning: Explicit solution could not be found." Thus, a change must be made to the hardware circuitry in order to account for the multiple paths in the sensor network.

```
syms R1 R2 R3 R4 Req1 Req2 Req3 Req4
[solve_R1, solve_R2, solve_R3, solve_R4] = solve(...
 'Req1 = (((R1*R2) + (R1*R3) + (R1*R4)) / (R1 + R2 + R3 + R4))', ...
'Req2 = (((R2*R1) + (R2*R3) + (R2*R4)) / (R1 + R2 + R3 + R4))', ...
 'Req3 = (((R3*R1) + (R3*R2) + (R3*R4)) / (R1 + R2 + R3 + R4))', ...
'Req4 = (((R4*R1) + (R4*R2) + (R4*R3)) / (R1 + R2 + R3 + R4))')
```

Figure 41: Matlab Symbolic Toolbox Script (Version 2009b)

4.2 Comparison to Literature

[15] contains a very similar circuit for measuring pressure sensors, shown in **Figure 42**, as that used in SmartCast. In order to decrease the current through additional paths, drain resistors are used to leak the current to ground. This circuit was simulated in NI Multisim to better understand the measurement error vs Rdrain resistor value, the current draw from the drain resistors, and to help design the SmartCast interface. Adding drain resistors increases the system power consumption and it was important to analyze the impact of this as SmartCast intends to be a mobile system.

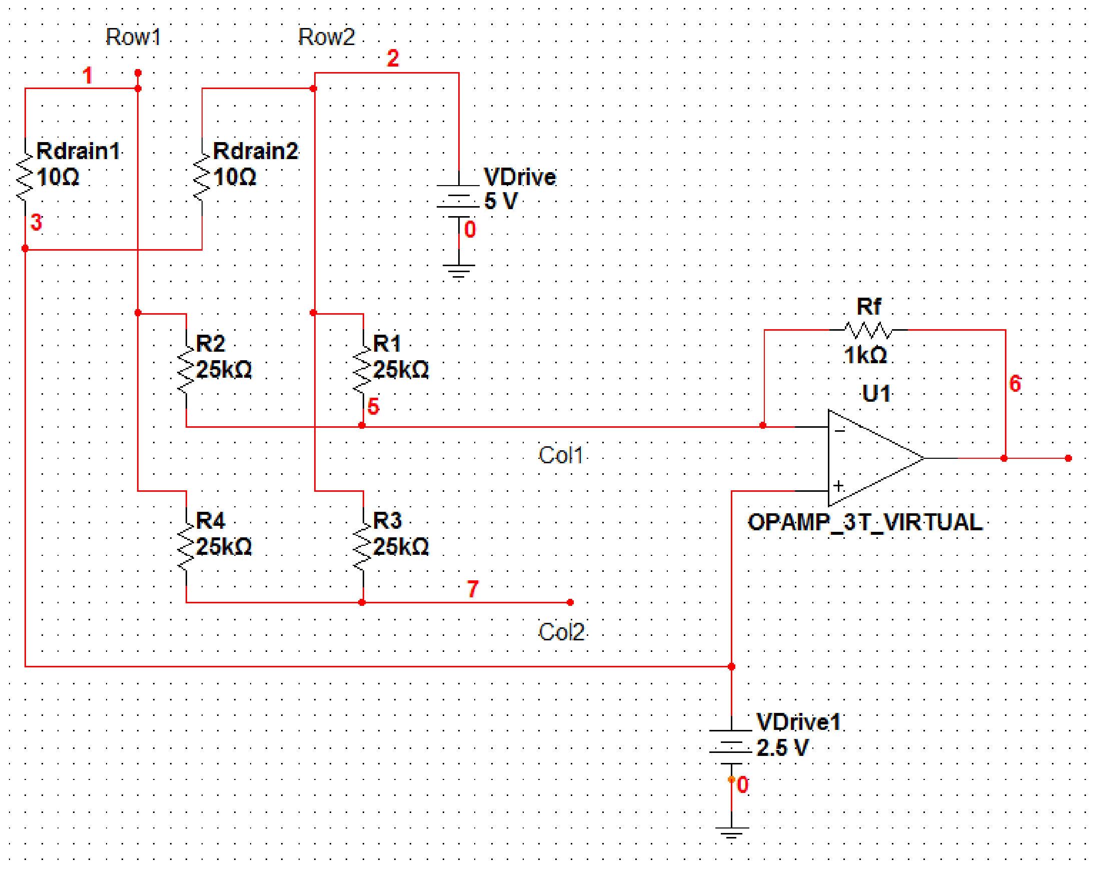


Figure 42: SmartSkin Analog Interface[15]

Figure 43 shows the error in measured resistance value vs. drain resistance values. The Y-axis on the left is the measured value of R1 (i.e. the sensor). The X-axis is the actual R1 value which is known at simulation time. The Y-axis on the right is the percent error between the measured and actual R1 value. The red line is a perfect x=y line; any deviation from this line with respect to the Y-axis on the left indicates error in the R1 measurement. The solid lines depict the measured R1 value and correspond to the left Y-axis. The dotted lines depict the percent error. The blue lines, both solid and dotted, correspond to the case when Rdrain1 is infinity (i.e. an open circuit) and Rdrain2 is 0 ohms (i.e. a short). This is the best case (i.e. the case with the least error) as non-zero Rdrain values introduce measurement error. The black lines correspond to the case when then additional paths are disconnected but the Rdrain resistors are still in place, showing that the drain resistors do introduce some error. The green lines correspond to the case when the additional paths are connected and there are no drain resistors to correct the measurement error, i.e. the worst case. The large percent error shown by the dotted green line indicates that measurement error introduced by the additional paths needs to be addressed.

Figure 44 shows the current draw and percent error for three different sensor resistance values and for different values of Rdrain. For this simulation the drain resistor values was varied from 0 to 1kOhms; these values correspond to the X-axis. The left Y-axis is the current draw and the right Y-axis is the percent error. As can be seen from the solid lines, the current draw increases for small drain resistances, i.e. the drain resistors provide a path for current from Vdrive to ground. This implies that to decrease energy consumption, it is better to have larger drain resistors, as the current draw increases when drain resistors are small. What is also apparent from **Figure 44** is that when all sensor resistances are small, more error is introduced, as can be seen from the dotted red line. This can be explained by examining the circuit because sensors R2, R3, and R4 act as though they are in parallel with R1, due to the additional path through the network. Thus, when R2, R3 and R4 have lower resistance, this decreases the measured resistance because it is like a small resistance (R2+R3+R4) in parallel with another small resistance (R1), i.e. a smaller resistance in parallel will affect the equivalent resistance more than a large resistance. This implies that to decrease measurement error, we must have a large minimum to the sensor resistances.

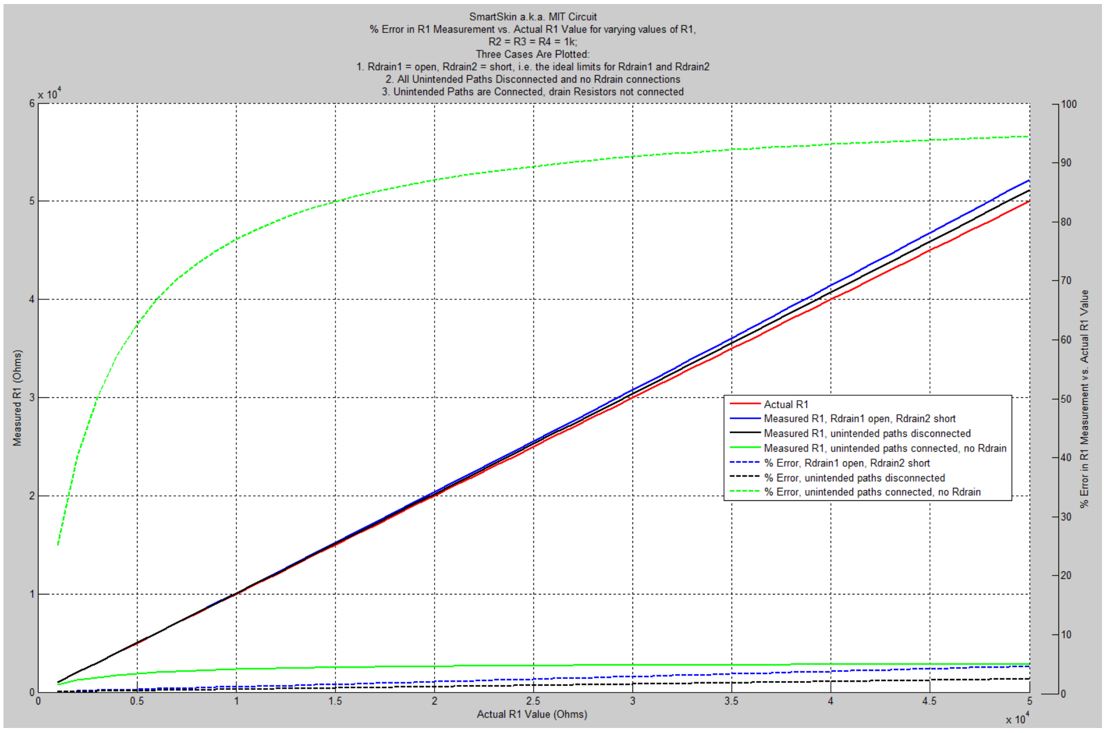


Figure 43: % Error in SmartSkin Resistance Measurement

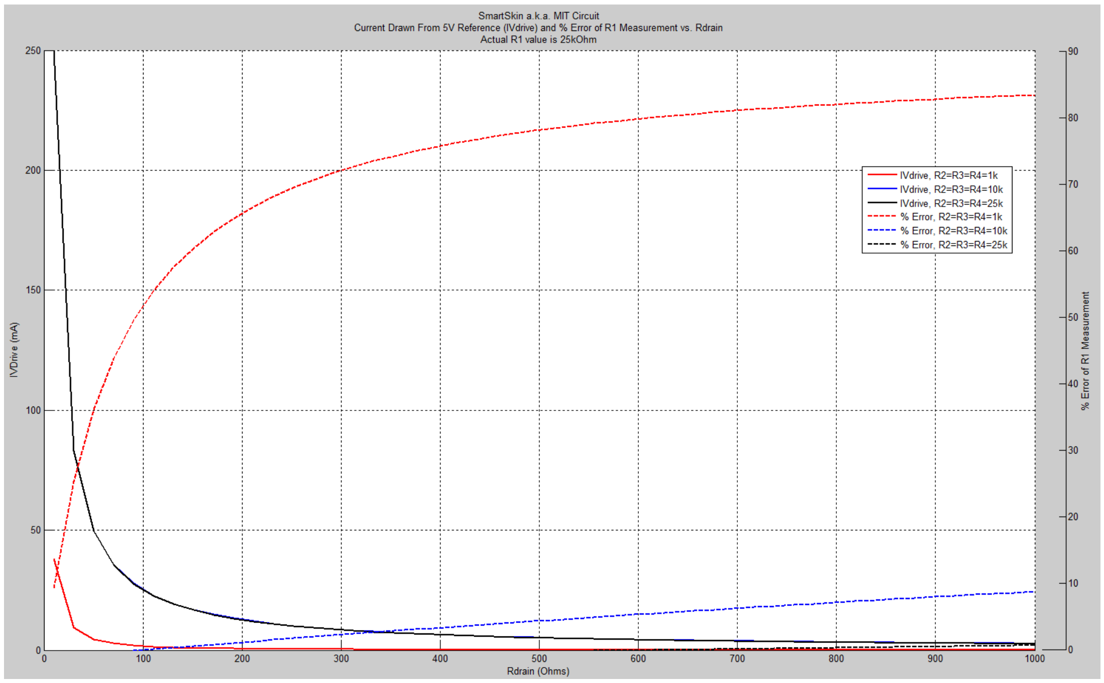


Figure 44: SmartSkin Analog Interface Current Draw

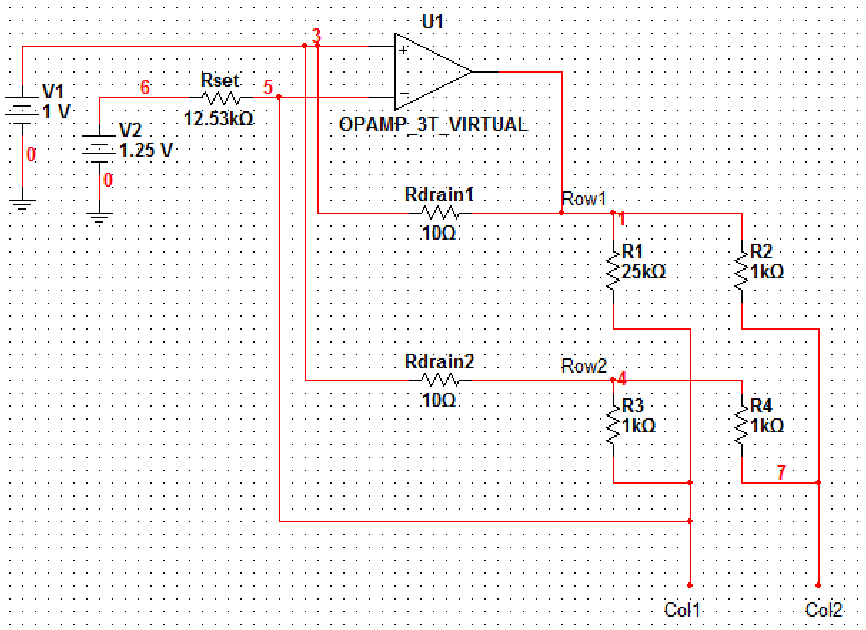


Figure 45: SmartCast Analog Interface w/ Drain Resistors

A similar analysis was carried out for the SmartCast analog interface, shown in **Figure 45**. Here, the circuit is a transimpedance amplifier, i.e. the current through the sensor is converted to a voltage, which is read by the microcontroller ADC (the sensors are in fact in the feedback path, as opposed to the circuit in **Figure 42**). Similar conclusions can be drawn as above, namely that to decrease current draw and power consumption, large drain resistors are needed, and to decrease measurement error, sensors with a large minimum resistance need to be used. In fact, to achieve the minimum current draw, the drain resistors connected to the unused rows (i.e. the rows not connected to the sensor currently being measured) need to be shorts, and to achieve minimum error, the drain resistor connected to the used row (i.e. the row connected to the sensor currently being measured) needs to be open. In other words, the unused rows need to be connected to the 1V reference directly, and the used row needs to completely disconnected from it. Therefore, in the SmartCast implementation of this fix, no drain resistors are used, and switches controlled by digital logic are used to connect/disconnect the rows to/from the 1V reference depending on which sensor is being measured. This implementation negates the additional paths by connecting the unused rows to the 1V reference, thereby creating a 0V drop across a resistor in each additional path, which cause no current to flow, thereby killing the additional path.

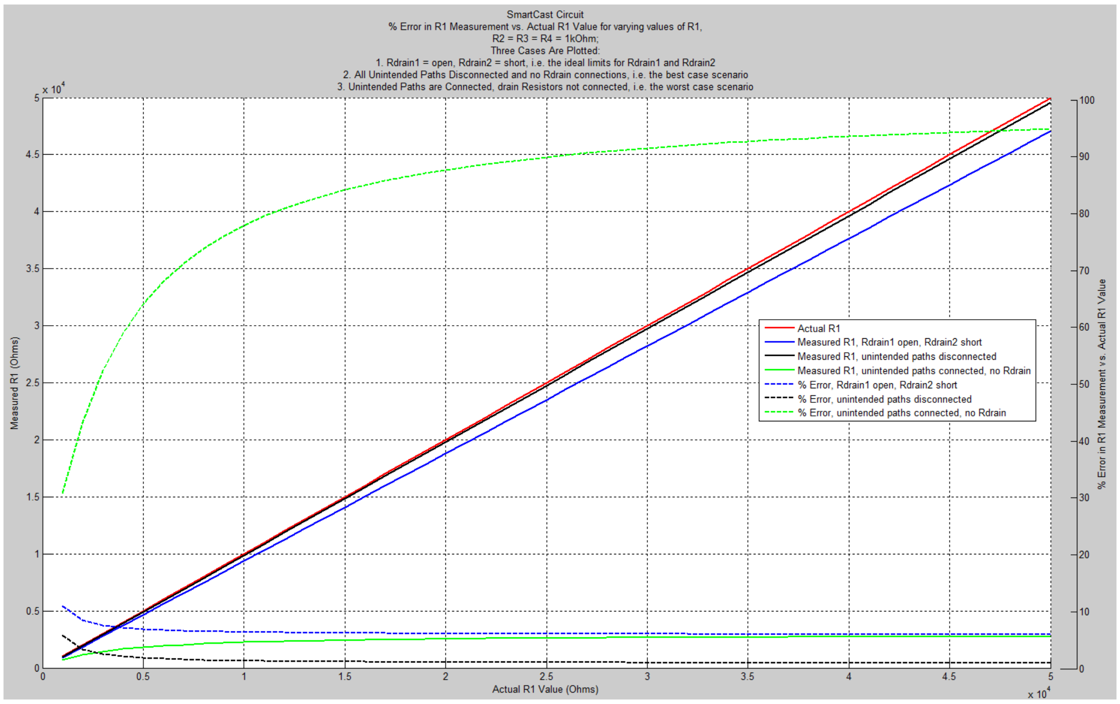


Figure 46: % Error in SmartCast Resistance Measurement

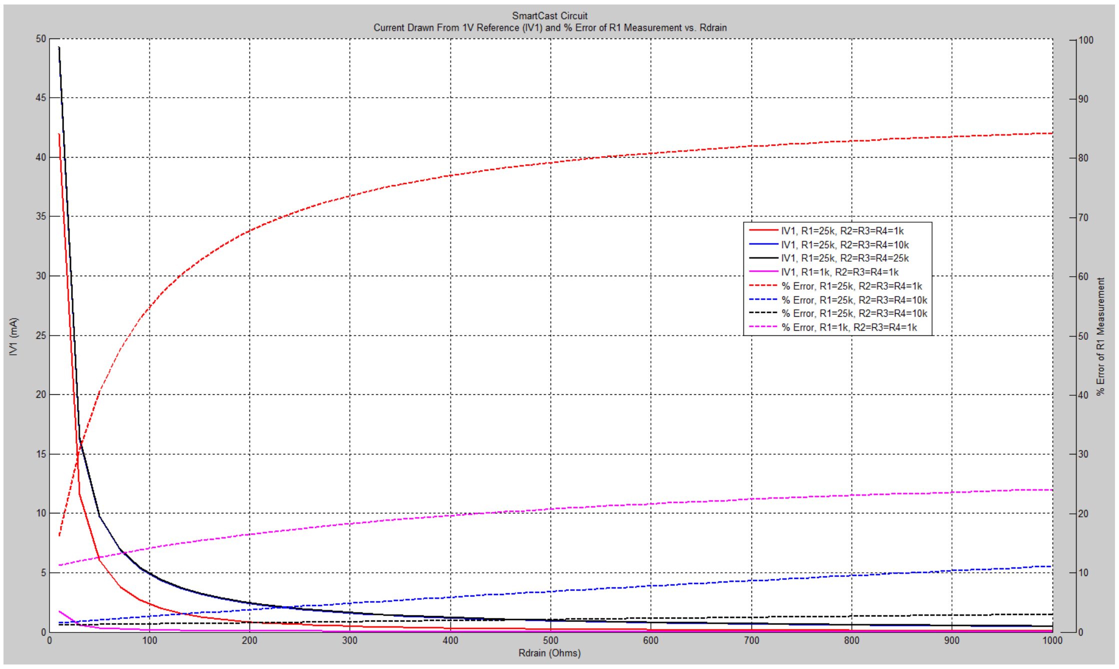


Figure 47: SmartCast Analog Interface Current Draw

4.3 Solution Implementation and Validation

Figure 48 shows a functional diagram of the additional circuitry used in the SmartCast system to implement the network fix. The light blue lines and blocks are the additional components comprising the fix. In order to save on microcontroller pins, the same control lines for AnlogMux1 were sent to a 2-4 decoder, thus no changes to the software are needed in this hardware implementation. The 2-4 decoder decodes the 2 control lines into a 4 bit binary number, each bit of which controls a single-pole single-throw switch. Each switch shares a common input, which is the 1V reference shown in the diagram. These are active high switches, and have the correct logic to connect the unused rows to the 1V reference, thereby killing the current through the extra paths.

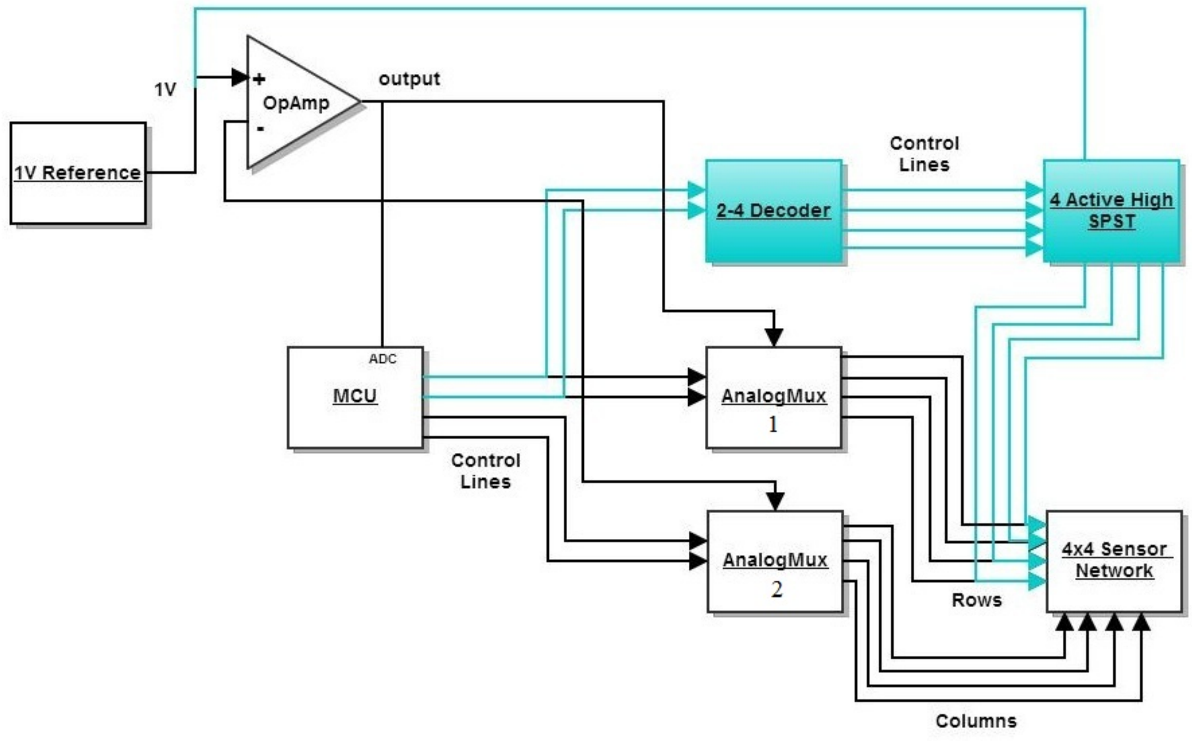


Figure 48: Functional Diagram of Analog Interface w/ Network Fix

In order to test the implementation of the network fix, the SmartCast system was connected to a bank of 16 ceramic resistors. **Figure 49** shows the data when the bank was sampled without the network fix and **Figure 50** shows data with the fix applied. Since the resistor values are known, we can calculate the error between the SmartCast measurements and the actual resistance values, thereby validating the network fix.

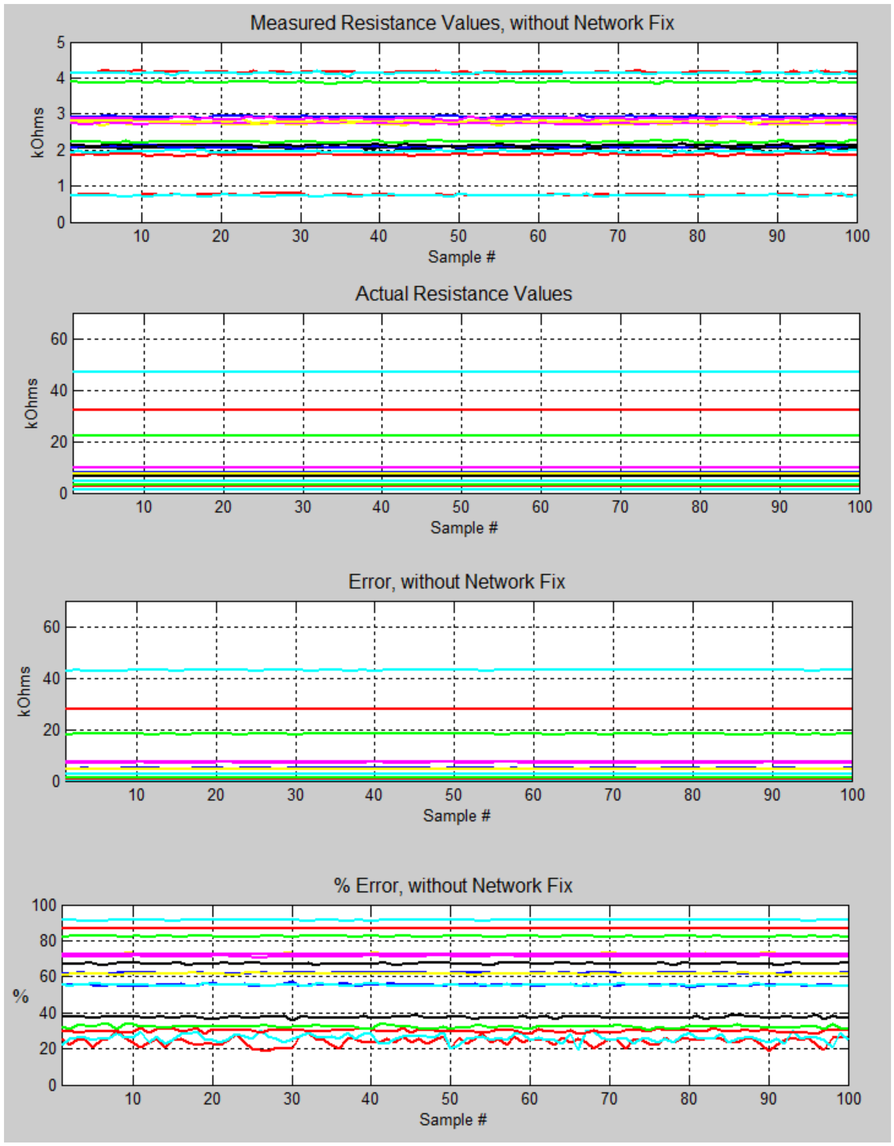


Figure 49: Resistance Measurements without Network Fix

The top plot of **Figure 49** shows the SmartCast measurements for the 16 resistors for 100 wake events. The next lower plot shows the actual resistance values. The bottom two plots show the error and error percentage, respectively; likewise for **Figure 50**. It can be clearly seen in **Figure 50** that the error drops considerably when applying the hardware network fix, thus validating this solution.

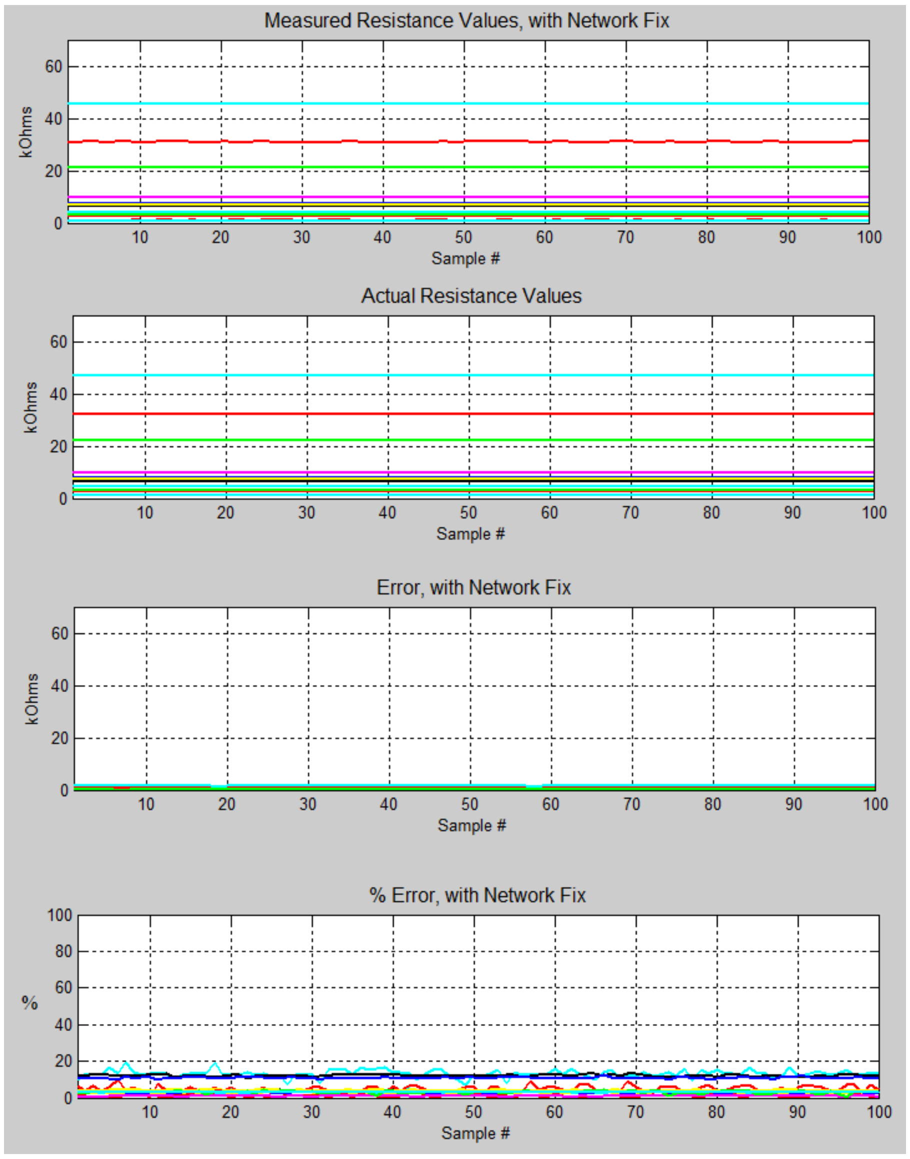


Figure 50: Resistance Measurements with Network Fix

Chapter 6 - Conclusion

6.1 Conclusion

A sensor sleeve has been designed that allows the pressure applied by an orthopedic cast to be measured. A number of experiments were carried out to determine the most appropriate textile sensor for this application. Out of all the types of sensors tested, two layers of Velostat with a top and bottom layer of nylon had the least variability between sensors with a suitable resistance range. Individual sensors can be constructed separately and then sewn onto a spandex-nylon sleeve to ease the manufacturing process. The output of these sensors cannot be compared directly as they give an indication of the relative, not absolute, force being applied.

The original SmartCast sensor matrix contained a number of additional paths through the network which confounded the resistance measurements. In order to develop an appropriate solution, the sensor network in [15] was examined and compared to the sensor network used in SmartCast. While the analog interface used in [15] is different than that used in the SmartCast system, a similar solution was used in SmartCast to weaken the effect of the multiple paths, namely tying the unused rows to the 1V reference.

The SmartCast system firmware has been developed and verified and is shown in Appendix D. This code was used to obtain all the sensor measurement data presented in this thesis. Furthermore, the firmware has been designed to extend the system lifetime, namely by using the sleep mode of the microcontroller, cutting power to the external SDCard and by using the internal Flash and EEPROM memory of the microcontroller.

6.2 Future Work

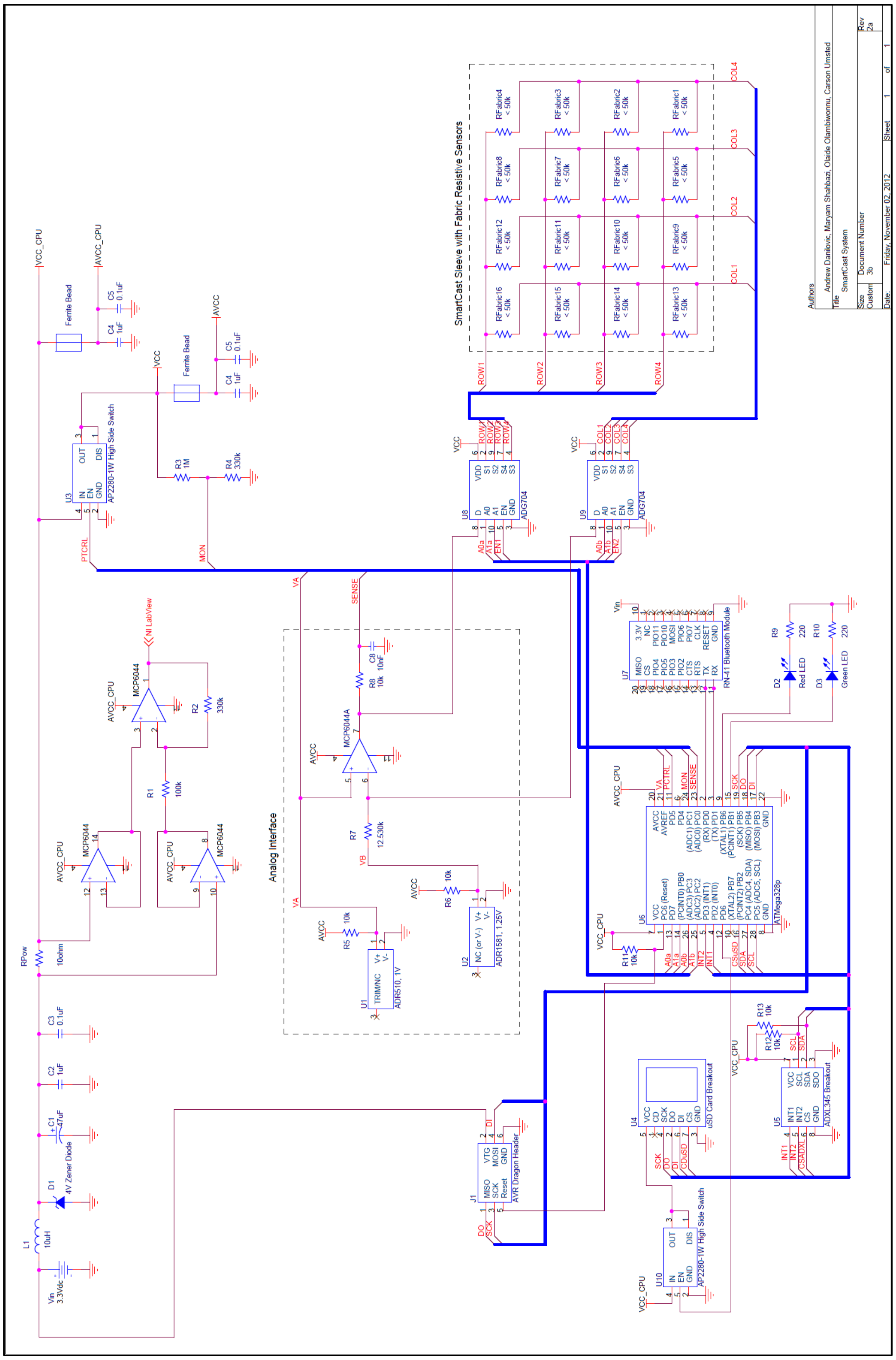
A number of improvements can be made to the SmartCast system to increase its usefulness, durability and ease of use. Waterproof testing should be done to determine the durability of the SmartCast system sensor sleeve in a real-life environment. While the individual textile sensors have been made watertight by the addition of a top and bottom layer of nylon, their connections are exposed and a short may occur if liquid with a high salt content, e.g. sweat, were to touch the connections.

Work should be done to reduce the size of the PCB and case to allow the SmartCast system to be easily worn by small children. Currently, the main component adding to the size of the SmartCast system is the power supply, which consists of 2-3 AAA batteries. A large power supply is necessitated by the use of the external SD Card, which is not low power storage. A different class of low power, embedded storage would need to be integrated into the SmartCast system to reduce the size of the power supply, which would then allow the PCB size to be reduced as well.

Patient trials should be conducted to obtain data from real subjects. This data can then be analyzed to determine if algorithms can be developed which could alert the patient or doctor that the cast is fitting improperly.

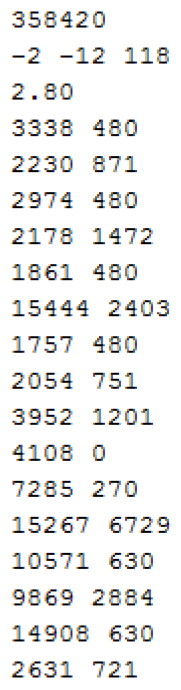
Appendices

A. SmartCast Circuit Schematic



B. Sample SmartCast SD Card Data Format

Figure 51 shows an example of the SmartCast SD Card data format. The data from 1 wake-sample-store event is eventually stored in a text file on the SD Card. One set of data contains 19 lines. The first line is the sample number; in this example the sample number is 358420, which means this set of data is from the 358420'th wake event. The next line contains the raw output of the accelerometer, in the format: X-axis, Y-axis, Z-axis. The third line is the system voltage as measured by the ATMega328p. The remaining 16 lines are the sensor means and variances. For example, on the 4th line, 3338 is the mean resistance value (in Ohms) of Sensor 1 over 10 samples of the onboard ADC for this particular wake event. 480 is the sensor variance over those same 10 samples.



```
358420
-2 -12 118
2.80
3338 480
2230 871
2974 480
2178 1472
1861 480
15444 2403
1757 480
2054 751
3952 1201
4108 0
7285 270
15267 6729
10571 630
9869 2884
14908 630
2631 721
```

Figure 51: SmartCast SD Card Data Format

Additionally, every time data is written to the SD Card, an additional line is inserted into the data with debug information. This line has the following format:

RXXXX

where the X's are hexadecimal numbers corresponding to debug codes defined in the software, such as an interrupt occurred or the hard limit to the number of Flash writes defined in software has been exceeded. In order to remove these lines from the data.txt file, regular expressions can be used. For example, in a text editor, a search-and-replace can be done for the regex: \nR\d+\n. This will find all the debug lines and they can then be automatically removed be replacing those lines with nothing, allowing a Matlab script to then easily parse the data file for plotting.

C. System Lifetime Calculation Matlab Code

```
clc;
clear;
set(0,'defaultlinelinewidth', 2);

t_Sleep = (1:1:3600) / 3600;

%These capacities are for the Energizer
%line of Ultimate Lithium batteries
%   2AAA  3AA  2AA
C = [2400 3600 6000];

%Original System Configuration, i.e. put the SD card into idle mode
%but write to it at each wake event
I_sample = 10; %mA, during sample period
I_write = 100; %mA, during SD Card write
I_sleep = 0.6; %mA, during sleep period

sample_duration = 0.41 / 3600; %0.41 sec * 1 hr/3600sec, from DAQ data
write_duration = 0.05 / 3600;
cycle_duration = sample_duration + write_duration + t_Sleep;

C_sample = I_sample * sample_duration;
C_write = I_write * write_duration;
C_sleep = I_sleep * t_Sleep;

C_wake_store_sleep = C_sample + C_write + C_sleep;

 for x = 1:length(C)
 num_cycles(x, :) = C(x) ./ C_wake_store_sleep;
 end

 for x = 1:length(C)
 Lifetime(x, :) = (num_cycles(x, :) .* cycle_duration) / 24; %days
 end

%--------------------------------------------------
%Sys Config with EEPROM and FLASH writes
%Average current during a wake event in which we write to internal memory
I_EEPROM = 15;          %mA
I_Flash = 12;           %mA

I_SDCard_Init = 10;     %mA
I_SDCard_Write = 100;   %mA
I_Sleep = 0.060;         %mA

%These times include the sampling
t_EEPROM = 0.7 / 3600;          %hours
t_Flash = 0.3 / 3600;           %hours
t_SDCard_Init = 0.7 / 3600;     %hours
t_SDCard_Write = 0.9 / 3600;    %hours
```

```
%num of EEPROM writes in a EEPROM-Flash-SDcard sequence
num_EEPROM_Writes = 7;
%num of Flash writes in a EEPROM-Flash-SDcard sequence
num_Flash_Writes = 44;
num_SDCard_Writes = 1;

cycle_duration_Config2 = ((t_EEPROM + t_Sleep) * num_EEPROM_Writes) + ...
    ((t_Flash + t_Sleep) * num_Flash_Writes)  + t_SDCard_Init + ...
    t_SDCard_Write + t_Sleep;

C_EEPROM = I_EEPROM * t_EEPROM;
C_Flash = I_Flash * t_Flash;
C_SDCard_Init = I_SDCard_Init * t_SDCard_Init;
C_SDCard_Write = I_SDCard_Write * t_SDCard_Write;
C_Sleep = I_Sleep * t_Sleep;

C_EEPROM_Flash_SDCard = ((C_EEPROM + C_Sleep) * num_EEPROM_Writes) + ...
    ((C_Flash + C_Sleep) * num_Flash_Writes) + C_SDCard_Init + ...
    C_SDCard_Write + C_Sleep;

 for x = 1:length(C)
 num_cycles_Config2(x, :) = C(x) ./ C_EEPROM_Flash_SDCard;
 end

 for x = 1:length(C)
 Lifetime_Config2(x, :) = (num_cycles_Config2(x, :) .* ...
     cycle_duration_Config2) / 24; %days
 end

%--------------------------------------------------
%3rd Configuration - With LDO
I_EEPROM_L = 20;          %mA
I_Flash_L = 18;           %mA
I_SDCard_Init_L = 10;     %mA
I_SDCard_Write_L = 100;   %mA
I_Sleep_L = 0.06;        %mA
I_LDO = 0.008;            %mA
I_LDO_Sleep = 0.0005;     %mA

%These times include the sampling
t_EEPROM_L = 0.7 / 3600;          %hours
t_Flash_L = 0.3 / 3600;           %hours
t_SDCard_Init_L = 0.7 / 3600;     %hours
t_SDCard_Write_L = 0.9 / 3600;    %hours

num_EEPROM_Writes_L = 7;          %num of EEPROM writes in a EEPROM-Flash-
SDcard sequence
num_Flash_Writes_L = 44;
num_SDCard_Writes_L = 1;

cycle_duration_Config2_L = ((t_EEPROM_L + t_Sleep) * num_EEPROM_Writes_L) +
((t_Flash_L + t_Sleep) * num_Flash_Writes_L)  + t_SDCard_Init_L +
t_SDCard_Write_L + t_Sleep;
```

```
C_EEPROM_L = (I_EEPROM_L + I_LDO) * t_EEPROM_L;
C_Flash_L = (I_Flash_L + I_LDO) * t_Flash_L;
C_SDCard_Init_L = (I_SDCard_Init_L + I_LDO) * t_SDCard_Init_L;
C_SDCard_Write_L = (I_SDCard_Write_L + I_LDO) * t_SDCard_Write_L;
C_Sleep_L = (I_Sleep_L + I_LDO_Sleep) * t_Sleep;

C_EEPROM_Flash_SDCard_L = ((C_EEPROM_L + C_Sleep_L) * num_EEPROM_Writes_L) +
((C_Flash_L + C_Sleep_L) * num_Flash_Writes_L) + C_SDCard_Init_L +
C_SDCard_Write_L + C_Sleep_L;

 for x = 1:length(C)
 num_cycles_Config2_L(x, :) = C(x) ./ C_EEPROM_Flash_SDCard_L;
 end

 for x = 1:length(C)
 Lifetime_Config2_L(x, :) = (num_cycles_Config2_L(x, :) .*
cycle_duration_Config2_L) / 24; %days
 end

figure;
hold on;
grid on;
hOrig_b1 = plot((t_Sleep*3600), Lifetime(1,:), '-b');
hOrig_b2 = plot((t_Sleep*3600), Lifetime(2,:), '-k');
hOrig_b3 = plot((t_Sleep*3600), Lifetime(3,:), '-r');
hInt_b1 = plot((t_Sleep*3600), Lifetime_Config2(1,:), ':b');
hInt_b2 = plot((t_Sleep*3600), Lifetime_Config2(2,:), ':k');
hInt_b3 = plot((t_Sleep*3600), Lifetime_Config2(3,:), ':r');
plot((t_Sleep*3600), Lifetime_Config2_L(1,:), '--b');
plot((t_Sleep*3600), Lifetime_Config2_L(2,:), '--k');
plot((t_Sleep*3600), Lifetime_Config2_L(3,:), '--r');


ylabel('System Lifetime (days)');
xlabel('Sleep Time (sec)');
title({'System Lifetime vs. Sleep Time for Several System Configurations and
Batteries','Bateries are the Ultimate Lithium brand by Energizer', ...
    'w/ 2400mAh for 2 AAA @3V, 3600mAh for 3AAA @4.5V, and 6000mAh for 2AA
@3V'});
legend('No Internal Mem Use, SDCard Idle, 2AAA', ...
    'No Internal Mem Use, SDCard Idle, 3AAA', ...
    'No Internal Mem Use, SDCard Idle, 2AA', ...
    'Internal EEPROM, Flash, 2AAA', ...
    'Internal EEPROM, Flash, 3AAA', ...
    'Internal EEPROM, Flash, 2AA', ...
    'Internal EEPROM, Flash w/ LDO, 2AAA', ...
    'Internal EEPROM, Flash w/ LDO, 3AAA', ...
    'Internal EEPROM, Flash w/ LDO, 2AA');

text(20, Lifetime_Config2(1,20),...
     '\leftarrow20Sec Sleep Time',...
     'FontSize',12);

 text(60, Lifetime_Config2(1,60),...
     '\leftarrow1Min Sleep Time',...
     'FontSize',12);

set(TheAxis, 'yticklabel', ...
   reshape(num2str(get(TheAxis, 'YTick'),[],1), '%.6f'))
```

D. Matlab Code to Calculate Sensor Resolution & Sensitivity

```
%------------------------------------------------------------------------
%--------------------------------ReadText---------------------------------
%------------------------------------------------------------------------
close all;
clear all;
clc;

set(0,'defaultlinelinewidth', 2);

colors = ['r', 'k', 'b', 'g', 'y', 'm', 'g', 'r', 'c', 'k', 'b', 'y', 'm',
'g', 'r', 'c', ...
    'c', 'b'];

disp 'Begin Text Read'

NUM_SENSORS = 16;
NUM_SENSOR_PER_LINE = 2;
NUM_ACC = 1;
NUM_ACC_PER_LINE = 3;
NUM_OTHER_LINES = 2;
NUM_LINES_IN_GROUP = NUM_ACC + NUM_OTHER_LINES + NUM_SENSORS;
                                        %1 Sample # line
                                        %1 acc line
                                        %1 voltage line
                                        %16 Sensor lines

index_offset=((NUM_SENSORS*2)+(NUM_ACC*NUM_ACC_PER_LINE)+NUM_OTHER_LINES);

sampleBuf = '%lu\n';
bufferAcceleration = '%d %d %d\n';
bufferVoltage = '%e\n';
resistanceBuf1 = '%lu %lu\n';
resistanceBuf2 = '%lu %lu\n';
resistanceBuf3 = '%lu %lu\n';
resistanceBuf4 = '%lu %lu\n';
resistanceBuf5 = '%lu %lu\n';
resistanceBuf6 = '%lu %lu\n';
resistanceBuf7 = '%lu %lu\n';
resistanceBuf8 = '%lu %lu\n';
resistanceBuf9 = '%lu %lu\n';
resistanceBuf10 = '%lu %lu\n';
resistanceBuf11 = '%lu %lu\n';
resistanceBuf12 = '%lu %lu\n';
resistanceBuf13 = '%lu %lu\n';
resistanceBuf14 = '%lu %lu\n';
resistanceBuf15 = '%lu %lu\n';
resistanceBuf16 = '%lu %lu\n';

combinedBuf = strcat(sampleBuf, bufferAcceleration, bufferVoltage, ...
    resistanceBuf1, resistanceBuf2, resistanceBuf3, resistanceBuf4, ...
    resistanceBuf5, resistanceBuf6, resistanceBuf7, resistanceBuf8, ...
    resistanceBuf9, resistanceBuf10, resistanceBuf11, resistanceBuf12, ...
    resistanceBuf13, resistanceBuf14, resistanceBuf15, resistanceBuf16);

%-------------------
%THIS USER MUST INPUT THIS!!
numLines = 34923; %you need an extra blank line at the end
NUM_GROUPS_v1layer1 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v1layer1 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast Winter
2013\' ...
    'PlatformTest\try3\velostat_1layer1.txt'];
fid = fopen(file_v1layer1);
s_v1layer1 = fscanf(fid, combinedBuf, [index_offset, NUM_GROUPS_v1layer1]);
fclose(fid);
% %-------------------
%THIS USER MUST INPUT THIS!!
numLines = 35037; %you need an extra blank line at the end
NUM_GROUPS_v1layer2 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v1layer2 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast Winter
2013\' ...
    'PlatformTest\try3\velostat_1layer2.txt'];
fid = fopen(file_v1layer2);
s_v1layer2 = fscanf(fid, combinedBuf, [index_offset, NUM_GROUPS_v1layer2]);
fclose(fid);
% %-------------------
%THIS USER MUST INPUT THIS!!
numLines = 34619; %you need an extra blank line at the end
NUM_GROUPS_v2layer1 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v2layer1 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast Winter
2013\' ...
    'PlatformTest\try3\velostat_2layer1.txt'];
fid = fopen(file_v2layer1);
s_v2layer1 = fscanf(fid, combinedBuf, [index_offset, NUM_GROUPS_v2layer1]);
fclose(fid);
% %-------------------
%THIS USER MUST INPUT THIS!!
numLines = 34562; %you need an extra blank line at the end
NUM_GROUPS_v2layer2 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v2layer2 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast Winter
2013\' ...
    'PlatformTest\try3\velostat_2layer2.txt'];
fid = fopen(file_v2layer2);
s_v2layer2 = fscanf(fid, combinedBuf, [index_offset, NUM_GROUPS_v2layer2]);
fclose(fid);
% %-------------------
%---------------try4 data-----------------------------
%---------------try4 data-----------------------------
%---------------try4 data-----------------------------
% %-------------------
%THIS USER MUST INPUT THIS!!
numLines = 34771; %you need an extra blank line at the end
NUM_GROUPS_v1layer_Blue1 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
```

```
file_v1layer_Blue1 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast
Winter 2013\' ...
    'PlatformTest\try3\velostat_1layer_nylon_darkblue_1.txt'];
fid = fopen(file_v1layer_Blue1);
s_v1layer_Blue1 = fscanf(fid, combinedBuf, [index_offset,
NUM_GROUPS_v1layer_Blue1]);
fclose(fid);
% %------------------
% %------------------
%THIS USER MUST INPUT THIS!!
numLines = 34961; %you need an extra blank line at the end
NUM_GROUPS_v1layer_Blue2 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v1layer_Blue2 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast
Winter 2013\' ...
    'PlatformTest\try3\velostat_1layer_nylon_darkblue_2.txt'];
fid = fopen(file_v1layer_Blue2);
s_v1layer_Blue2 = fscanf(fid, combinedBuf, [index_offset,
NUM_GROUPS_v1layer_Blue2]);
fclose(fid);
% %------------------
% %------------------
%THIS USER MUST INPUT THIS!!
numLines = 35132; %you need an extra blank line at the end
NUM_GROUPS_v2layer_Green1 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v2layer_Green1 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast
Winter 2013\' ...
    'PlatformTest\try3\velostat_2layer_nylon_green_1.txt'];
fid = fopen(file_v2layer_Green1);
s_v2layer_Green1 = fscanf(fid, combinedBuf, [index_offset,
NUM_GROUPS_v2layer_Green1]);
fclose(fid);
% %------------------
% %------------------
%THIS USER MUST INPUT THIS!!
numLines = 34581; %you need an extra blank line at the end
NUM_GROUPS_v2layer_Green2 = floor(numLines / NUM_LINES_IN_GROUP); %how many
resistance measurements we have per sensor
file_v2layer_Green2 = ['C:\Users\D\Documents\Word Documents\EE202C\SmartCast
Winter 2013\' ...
    'PlatformTest\try3\velostat_2layer_nylon_green_2.txt'];
fid = fopen(file_v2layer_Green2);
s_v2layer_Green2 = fscanf(fid, combinedBuf, [index_offset,
NUM_GROUPS_v2layer_Green2]);
fclose(fid);
% %------------------


%--------------------------------------------------------------------------
%--------------------------Plot Best Fit Curves----------------------------
%--------------------------------------------------------------------------
disp 'Begin Text Plot'

Resistance_rows = 6:2:36;

poly_order = 3;
```

```
grams_to_Newtons = 9.8/1000;  %to account for Newtons being kg * m . s^2
Newtons_to_grams = 1000 / 9.8;

figure;
hold on;
    %the following should be rolled into a loop
    %no nylon, 1 layers
    [minLoc minMag] =
peakfinder_min(s_v1layer1(Resistance_rows(1), :)./1000);  %minMag has units
of kOhms
    minLoc_size(1) = size(minLoc, 2);   %for indexing
    minLoc_v(1,1:minLoc_size(1)) = rescale(minLoc, 1, 300); %rescale from
grams to Newtons
    [p, ErrorEst] = polyfit(minLoc_v(1,1:minLoc_size(1)), minMag,
poly_order);  %curve fit
    pop_fit_v(1,1:minLoc_size(1)) = polyval(p, minLoc_v(1,1:minLoc_size(1)),
ErrorEst); %pop_fit has units of kOhms
    plot(minLoc_v(1,1:minLoc_size(1)) * grams_to_Newtons,
pop_fit_v(1,1:minLoc_size(1)), '-r');

    [minLoc minMag] = peakfinder_min(s_v1layer2(Resistance_rows(1), :)./1000);
    minLoc_size(2) = size(minLoc, 2);
    minLoc_v(2,1:minLoc_size(2)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(2,1:minLoc_size(2)), minMag, poly_order);
    pop_fit_v(2,1:minLoc_size(2)) = polyval(p, minLoc_v(2,1:minLoc_size(2)),
ErrorEst);
    plot(minLoc_v(2,1:minLoc_size(2)) .* grams_to_Newtons,
pop_fit_v(2,1:minLoc_size(2)), '-k');

    %no nylon, 2 layers
    [minLoc minMag_v2layer1] =
peakfinder_min(s_v2layer1(Resistance_rows(1), :)./1000);
    minLoc_size(3) = size(minLoc, 2);
    minLoc_v(3,1:minLoc_size(3)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(3,1:minLoc_size(3)), minMag_v2layer1,
poly_order);
    pop_fit_v(3,1:minLoc_size(3)) = polyval(p, minLoc_v(3,1:minLoc_size(3)),
ErrorEst);
    plot(minLoc_v(3,1:minLoc_size(3)) * grams_to_Newtons,
pop_fit_v(3,1:minLoc_size(3)), '-b');


    %FSR1
    [minLoc minMag] =
peakfinder_min(s_v2layer1(Resistance_rows(16), :)./1000);
    minLoc_size(4) = size(minLoc, 2);
    minLoc_v(4,1:minLoc_size(4)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(4,1:minLoc_size(4)), minMag, poly_order);
    pop_fit_v(4,1:minLoc_size(4)) = polyval(p, minLoc_v(4,1:minLoc_size(4)),
ErrorEst);
    plot(minLoc_v(4,1:minLoc_size(4)) * grams_to_Newtons,
pop_fit_v(4,1:minLoc_size(4)), '-c');

    [minLoc minMag_v2layer2] =
peakfinder_min(s_v2layer2(Resistance_rows(1), :)./1000, 15);
    minLoc_size(5) = size(minLoc, 2);
    minLoc_v(5,1:minLoc_size(5)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(5,1:minLoc_size(5)), minMag_v2layer2,
poly_order);
    pop_fit_v(5,1:minLoc_size(5)) = polyval(p, minLoc_v(5,1:minLoc_size(5)),
ErrorEst);
    plot(minLoc_v(5,1:minLoc_size(5)) * grams_to_Newtons,
pop_fit_v(5,1:minLoc_size(5)), '-g');

    %FSR2
    [minLoc minMag] =
peakfinder_min(s_v2layer2(Resistance_rows(16), :)./1000);
    minLoc_size(6) = size(minLoc, 2);
    minLoc_v(6,1:minLoc_size(6)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(6,1:minLoc_size(6)), minMag, poly_order);
    pop_fit_v(6,1:minLoc_size(6)) = polyval(p, minLoc_v(6,1:minLoc_size(6)),
ErrorEst);
    plot(minLoc_v(6,1:minLoc_size(6)) * grams_to_Newtons,
pop_fit_v(6,1:minLoc_size(6)), '-c');

    %I had to tune the peaks vector to calculate the correct difference
    %one layer with nylon
    [minLoc minMag_Blue1] =
peakfinder_min(s_v1layer_Blue1(Resistance_rows(1), :)./1000, 0.3);
    minLoc_size(7) = size(minLoc, 2);
    minLoc_v(7,1:minLoc_size(7)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(7,1:minLoc_size(7)), minMag_Blue1,
poly_order);
    pop_fit_v(7,1:minLoc_size(7)) = polyval(p, minLoc_v(7,1:minLoc_size(7)),
ErrorEst);
    plot(minLoc_v(7,1:minLoc_size(7)) * grams_to_Newtons,
pop_fit_v(7,1:minLoc_size(7)), '-y');

    [minLoc minMag_Blue2] =
peakfinder_min(s_v1layer_Blue2(Resistance_rows(1), :)./1000, 0.6);
    minLoc_size(8) = size(minLoc, 2);
    minLoc_v(8,1:minLoc_size(8)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(8,1:minLoc_size(8)), minMag_Blue2,
poly_order);
    pop_fit_v(8,1:minLoc_size(8)) = polyval(p, minLoc_v(8,1:minLoc_size(8)),
ErrorEst);
    plot(minLoc_v(8,1:minLoc_size(8)) * grams_to_Newtons,
pop_fit_v(8,1:minLoc_size(8)), '-m');

    %two layers with nylon
    [minLoc minMag_Green1] =
peakfinder_min(s_v2layer_Green1(Resistance_rows(1), :)./1000);
    minLoc_size(9) = size(minLoc, 2);
    minLoc_v(9,1:minLoc_size(9)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(9,1:minLoc_size(9)), minMag_Green1,
poly_order);
    pop_fit_v(9,1:minLoc_size(9)) = polyval(p, minLoc_v(9,1:minLoc_size(9)),
ErrorEst);
    plot(minLoc_v(9,1:minLoc_size(9)) * grams_to_Newtons,
pop_fit_v(9,1:minLoc_size(9)), '-g');
```

```
    [minLoc minMag_Green2] =
peakfinder_min(s_v2layer_Green2(Resistance_rows(1), :)./1000, 6.5);
    minLoc_size(10) = size(minLoc, 2);
    minLoc_v(10,1:minLoc_size(10)) = rescale(minLoc, 1, 300);
    [p, ErrorEst] = polyfit(minLoc_v(10,1:minLoc_size(10)), minMag_Green2,
poly_order);
    pop_fit_v(10,1:minLoc_size(10)) = polyval(p,
minLoc_v(10,1:minLoc_size(10)), ErrorEst);
    plot(minLoc_v(10,1:minLoc_size(10)) * grams_to_Newtons,
pop_fit_v(10,1:minLoc_size(10)), '-r');


    legend('1 layer, test1', '1 layer, test2', '2 layers, test1', ...
        '2 layers, Circular Force Sensor', '2 layers, test2', ...
        '2 layers, Circular Force Sensor', ...
        'w/ nylon 1layer, test1', 'w/ nylon 1layer, test2', ...
        'w/ nylon 2layer, test1', 'w/ nylon 2layer, test2');


    grid on;
    title_string = sprintf(['Polynomial Best Fit Curves of Sensor Resistance
vs. Applied Force\n' ...
    'for the Velostat Sensors in Fig. 25']);
    title(title_string);
    xlabel('Applied Force (Newtons)');
    ylabel('Sensor Resistance (kOhms)');
    axis([0 (300 * grams_to_Newtons) 0 75]);


%------------------
ADC_resolution = 49;    %49 Ohms per LSB of the 10 bit ADC

%---------------------------------------------------------------------------
%------------------------Calculate and Plot Sensitivity--------------------
%---------------------------------------------------------------------------
figure;
grid on;
hold on;
colors = ['r', 'k', 'b', 'c', 'g', 'c', 'y', 'm', 'g', 'r'];
for j=1:10
    %estimate slope
    size1(j) = size(pop_fit_v(j,1:minLoc_size(j)), 2) - 1;
    y_diff(j,1:size1(j)) = diff(pop_fit_v(j,1:minLoc_size(j))); %y_diff has
units of kOhms
    x_diff(j,1:size1(j)) = diff(minLoc_v(j,1:minLoc_size(j)));  %x_diff has
units of grams
    Sensitivity(j,1:size1(j)) = (y_diff(j,1:size1(j)) ./ (grams_to_Newtons *
x_diff(j,1:size1(j))));   %sensitivity, (kOhms / Newtons)
    plot(minLoc_v(j,1:length(Sensitivity(j,1:size1(j)))) * grams_to_Newtons,
abs(Sensitivity(j,1:size1(j))), colors(j));
end
    title('Sensitivity vs. Applied Force for the Velostat Sensors in Fig.
25');
    xlabel('Applied Force (Newtons)');
    ylabel('Sensitivity (kOhms/Newton)');
    %axis([0 (300 * x_axis_scaling) 0 700]);
        legend('1 layer, test1', '1 layer, test2', '2 layers, test1', ...
        '2 layers, Circular Force Sensor', '2 layers, test2', ...
        '2 layers, Circular Force Sensor', ...
        'w/ nylon 1layer, test1', 'w/ nylon 1layer, test2', ...
        'w/ nylon 2layer, test1', 'w/ nylon 2layer, test2');


%---------------------------------------------------------------------------
%--------------------%Calculate and Plot Sensor Resolution-----------------
%---------------------------------------------------------------------------
figure;
grid on;
hold on;
for j=1:10
    Sensor_Resolution(j,1:size1(j)) = (ADC_resolution./1000) ./
Sensitivity(j,1:size1(j)); %units of Newtons
    plot(minLoc_v(j,1:length(Sensitivity(j,1:size1(j)))) * grams_to_Newtons,
abs(Sensor_Resolution(j,1:size1(j))), colors(j));
end;
    title('Sensor Resolution vs. Applied Force for the Velostat Sensors in
Fig. 25');
    xlabel('Applied Force (Newtons)');
    ylabel('Sensor Resolution (Newtons)');
    %axis([0 (300 * x_axis_scaling) 0 (150 * x_axis_scaling)]);
        legend('1 layer, test1', '1 layer, test2', '2 layers, test1', ...
        '2 layers, Circular Force Sensor', '2 layers, test2', ...
        '2 layers, Circular Force Sensor', ...
        'w/ nylon 1layer, test1', 'w/ nylon 1layer, test2', ...
        'w/ nylon 2layer, test1', 'w/ nylon 2layer, test2');
```

E. SmartCast System Code

```c
/*
 * ConfigFile.h
 *
 * Created: 11/29/2012 8:40:35 PM
 *  Author: Andrew Danilovic
 */

#ifndef CONFIGFILE_H_
#define CONFIGFILE_H_

#include <stdint.h>

#define NUM_LINES_CONFIG_FILE 7

void ParseConfigFile(uint8_t acc_data[]);
void BlinkRedLED();

#endif /* CONFIGFILE_H_ */
```

```
/*
 * ConfigFile.cpp
 *
 * Created: 11/29/2012 8:40:42 PM
 *  Author: Andrew Danilovic
 */

#include "ConfigFile.h"
#include "../SmartCastApp/SmartCast.h"
#include "SmartCardUser.h"

int8_t ReadSignedByte(File &configFile);
int8_t ReadDec_CR_LF_EOF_HEX(File &configFile);
int8_t ReadDec_CR_LF_EOF(File &configFile);
uint16_t ConfirmSafe_int(uint16_t max);
uint8_t ConfirmSafe_hex(uint8_t min, uint8_t max);
void ParseLine1(File &configFile);
void ParseLine2_end(File &configFile, uint8_t acc_data[]);
void PrintParsedData(uint8_t acc_data[]);
void BlinkRedLED();

char *config_init = "SCC\r\n";     //1st line of config file

/*******************************************************************
* File: ConfigFile.cpp
* NAME : void BlinkRedLED()
* Description: Blink LEDs
* Inputs: None
* Outputs: None
*******************************************************************/
void BlinkRedLED() {
       while(1) {
              PORTB |= (uint8_t)(1<<CONFIG_LED);
              //Serial.println(where);
              delay(50);
              PORTB &= (uint8_t)~(1<<CONFIG_LED);
              delay(50);
       }
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : int8_t ReadSignedByte(File &configFile)
* Description: Check if byte read from file is a signed char
* Inputs: configFile, ptr to a file
* Outputs: signed byte representing an ASCII char
*******************************************************************/
int8_t ReadSignedByte(File &configFile) {
       int16_t unsafe = configFile.read();
       if(unsafe >= -128 && unsafe <= 127) {return unsafe;}
       else {BlinkRedLED();}
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : int8_t ReadDec_CR_LF_EOF_HEX(File &configFile)
* Description: check if byte is /r, /n, eof, or a valid hexadecimal
* number
* Inputs: configFile, ptr to a file
* Outputs: signed byte representing an ASCII char
*******************************************************************/
int8_t ReadDec_CR_LF_EOF_HEX(File &configFile) {
       int8_t unsafe = ReadSignedByte(configFile);
       if(unsafe >= 48 && unsafe <= 57) {return unsafe;}      //return if ASCII decimal
       else if(unsafe >= 65 && unsafe <= 70) {return unsafe;} //return if hex char, i.e.
A,B,C,D,E,F
       else { //else check if CR or LF or eof (which is -1)
              if(unsafe == '\r') {return unsafe;}
              else if(unsafe == '\n') {return unsafe;}
              else if(unsafe == -1) {return unsafe;}
              else {BlinkRedLED();}
       }
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : int8_t ReadDec_CR_LF_EOF(File &configFile)
* Description: check if byte is /r, /n, or end of file
* Inputs: configFile, ptr to a file
* Outputs: signed byte representing an ASCII char
*******************************************************************/
int8_t ReadDec_CR_LF_EOF(File &configFile) {
       int8_t unsafe = ReadSignedByte(configFile);
       if(unsafe >= 48 && unsafe <= 57) {return unsafe;}      //return if ASCII decimal
       else { //else check if CR or LF or eof (which is -1)
              if(unsafe == '\r') {return unsafe;}
              else if(unsafe == '\n') {return unsafe;}
              else if(unsafe == -1) {return unsafe;}
              else {BlinkRedLED();}
       }
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : uint16_t ConfirmSafe_int(uint16_t max)
* Description: check if unsigned 16 bit number is positive, equal to
* or below the max input value
* Inputs: max, the maximum the value the number can be, temp_buf
* Outputs: valid unsigned 16 bit number
*******************************************************************/
uint16_t ConfirmSafe_int(uint16_t max) {
       int32_t unsafe = atoi(temp_buf);
       if(unsafe >= 0 && unsafe <= max) {return unsafe; /*safe*/}
       else {BlinkRedLED();}
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : uint8_t ConfirmSafe_hex(uint8_t min, uint8_t max)
* Description: check if a number is between min and max
* Inputs: min, max, temp_buf
* Outputs: valid 8 bit number
*******************************************************************/
uint8_t ConfirmSafe_hex(uint8_t min, uint8_t max) {
       long unsafe = strtol(temp_buf, NULL, 16);
       if(unsafe >= min && unsafe <= max) {return unsafe; /*safe*/}
```

```
	else {BlinkRedLED();}
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : void ParseLine1(File &configFile)
* Description: parse the 1st line of the config file, which must
* conform to a specific format defined by config_init
* Inputs: configFile, config_init
* Outputs:
*******************************************************************/
void ParseLine1(File &configFile) {
	for(uint8_t i = 0; i < 5; i++) {
		if(ReadSignedByte(configFile) == config_init[i]) {/*do nothing, you're
good*/}
			else {BlinkRedLED();}
	}
}

/*******************************************************************
* File: ConfigFile.cpp
* NAME : void ParseLine2_end(File &configFile, uint8_t acc_data[])
* Description: continue parsing the rest of the config file, if any
* errors are found in the format, blink red LED in an infinite loop
* Inputs: configFile, acc_data[], which stores the accelerometer
* parameters
* Outputs:
*******************************************************************/
void ParseLine2_end(File &configFile, uint8_t acc_data[]) {
	uint8_t i = 0;
	uint8_t j = 0;
	bool eof = 0;

	//the i index here is the line number, the data starts on line 2
	for(i = 2; i <= NUM_LINES_CONFIG_FILE; i++) {	//max NUM_LINES_CONFIG_FILE lines
of data
		for(j = 0; j < 10; j++) {//here, j is the column num, assume max 10 columns
in file
			if(i < 3) {	//for the decimal lines
				temp_buf[j] = ReadDec_CR_LF_EOF(configFile);
				if(temp_buf[j] == '\n') {break; /*end of line*/}
				else if(temp_buf[j] == -1) {eof = 1; break; /*end of file*/}
				else {/*keep going*/}
			}
			else { //for the HEX lines
				temp_buf[j] = ReadDec_CR_LF_EOF_HEX(configFile);
				if(temp_buf[j] == '\n') {break; /*end of line*/}
				else if(temp_buf[j] == -1) {eof = 1; break; /*end of file*/}
				else {/*keep going*/}
			}
		}
		if(j == 10) {BlinkRedLED();}		//i.e. didn't find \n
		if(!eof) {
			//at this point, we have read in a full line, but must null
terminate.
			//we've read up to the \n, so insert a \0.
			//j is the index of the last element
			if(j >= 1 && j <= 25) {temp_buf[j] = '\0';}
```

```
                        else {BlinkRedLED();}       /*i.e. the number was too small*/

                        //convert to integer and assign to variable
                        int unsafe = 0;
                        switch (i) {
                                case 2:
                                        sleep_time = ConfirmSafe_int(3600);
                                        //sleep_time = 5000;
                                        break;
                                /*
                                case 3:
                                        collect_stat_loop = ConfirmSafe_int(100);
                                        break;
                                */
                                case 3:
                                        acc_data[0] = ConfirmSafe_hex(23, 28);
                                        break;
                                case 4:
                                        acc_data[1] = ConfirmSafe_hex(0, 0xFF);
                                        break;
                                case 5:
                                        acc_data[2] = ConfirmSafe_hex(0, 0xFF);
                                        break;
                                case 6:
                                        acc_data[3] = ConfirmSafe_hex(0, 0xFF);
                                        break;
                                case 7:
                                        acc_data[4] = ConfirmSafe_hex(0, 0xFF);
                                        break;
                                default:
                                        BlinkRedLED();
                        }
                }
                else if(i < NUM_LINES_CONFIG_FILE) {BlinkRedLED();}    //error reading
bytes from file
        }
}
/*
void PrintParsedData(uint8_t acc_data[]) {
        Serial.println(sleep_time);
        Serial.println(collect_stat_loop);
        Serial.println(acc_data[0]);
        Serial.println(acc_data[1]);
        Serial.println(acc_data[2]);
        Serial.println(acc_data[3]);
        Serial.println(acc_data[4]);
}
*/

/*******************************************************************
* File: ConfigFile.cpp
* NAME : void ParseConfigFile(uint8_t acc_data[])
* Description: top level parse function, called by init
* Inputs: acc_data[], which stores the accelerometer
* parameters
* Outputs:
*******************************************************************/
void ParseConfigFile(uint8_t acc_data[]) {
	char *config_file = "config.txt";
	SDPowerCtrl(ON);
	if(InitializeSDCard() == 0) {
		if(error_flags & SMARTCAST_ERROR) {BlinkRedLED();}
		File configFile = SD.open(config_file, FILE_READ);
		if (configFile) {
			ParseLine1(configFile);
			ParseLine2_end(configFile, acc_data);
		}
		else {BlinkRedLED();}		//error opening config file
		//PrintParsedData(acc_data);
		SDPowerCtrl(OFF);
	}
	else {BlinkRedLED();}		//i.e. no SD card present
}
```

```
/*
 * SmartCardUser.h
 *
 * Created: 7/22/2012 6:35:36 PM
 *  Author: Andrew Danilovic
 */


#ifndef SMARTCARDUSER_H_
#define SMARTCARDUSER_H_

#include "../SD.h"
#include "../SmartCastApp/SmartCast.h"

//#define INCLUDECHECKSD
//#define INCLUDEDUMPFILE
//#define USE_TIMERS
//#define USE_SD_CARD

extern const uint8_t CardDetect;   //This is PD2, also known as INT0
extern const uint8_t hardwareSS;   //PB2

uint8_t InitializeSDCard();
void WriteToSDCard(int myVal);
extern boolean WriteToSDCardbuf(uint16_t start_page, uint16_t end_page);
void checkSDCard();
void DumpFile(char* myFile);
void rmFile(char* myFile);
void rmAllFiles();

#endif /* SMARTCARDUSER_H_ */
```

```
/*
 * SmartCardUser.cpp
 *
 * Created: 7/22/2012 6:35:26 PM
 *  Author: Andrew Danilovic
 */

#include "Arduino.h"
#include "SmartCardUser.h"
#include "../utility/Sd2Card.h"
#include "../SmartCastApp/RunningStat.h"
#include "../SmartCastApp/SmartCast.h"
#include "../utility/memdebug.h"
#include "../utility/StackPaint.h"
#include "../utility/myFlash.h"
#include "../SmartCastApp/Interrupts.h"
#include "../utility/myEEPROM.h"

const uint8_t hardwareSS = 10;    //PB2

char flag_buf[8];          //create temp buf to put error flags in, must be 8 long to
include null term

/*******************************************************************
* File: SmartCardUser.cpp
* NAME : uint8_t InitializeSDCard()
* Description: Initialize the SD Card by calling the Arduino
* SD Card library functions for initialization, blink LED,
* set error flags based on return code from Arduino library
* Inputs:
* Outputs: returns 0 on success, 1 on failure
*******************************************************************/
uint8_t InitializeSDCard() {
       pinMode(hardwareSS, OUTPUT);      //just to make sure

       /*returns an 8 bit number,
       << 1 card.init status
       << 2 volume.init status
       << 3 root.openRoot status
       if all 3 are 1, success
       if any are 0, failure
       */

       LED_ON(&LED3);

       uint8_t ret = SD.begin(hardwareSS);

       //a ret value of 7 or 6 is OK, i.e. card.init can fail if we've called initialize
       //again without taking the card out, because it's already been initialized
       //but if the ret is anything other than 7 or 6, total fail
       switch (ret) {
              case 0x00:
                     error_flags |= SMARTCAST_ERROR;
                     error_flags |= SDCARDTIMEOUT_ERROR;
                     ret = 1;
              break;

              case 0x01:
```

```
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;

                case 0x02:
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;

                case 0x03:
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;

                case 0x04:
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;

                case 0x05:
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;

                case 0x06:
                        ret = 0;
                break;

                case 0x07:
                        ret = 0;
                break;

                default:
                        error_flags |= SMARTCAST_ERROR;
                        error_flags |= SDCARDTIMEOUT_ERROR;
                        ret = 1;
                break;
        }

        LED_OFF(&LED3);
        return ret;
}

/*******************************************************************
* File: SmartCardUser.cpp
* NAME : boolean WriteToSDCardbuf(uint16_t start_page, uint16_t end_page)
* Description: this function is misnamed, write to 3 different kinds of
* memory, 1st EEPROM, then Flash, then SD Card, rigourously
* check for errors
* Inputs: start_page, 1st page of memory that can be written to,
* end_page, last page of memory that can be written to
* Outputs: return 0 on success, 1 on fail
*******************************************************************/
```

```
boolean WriteToSDCardbuf(uint16_t start_page, uint16_t end_page) {

	uint8_t ret = 0;

	//Serial.println("WRSD");
	//Serial.println(start_page);
	//Serial.println(end_page);

	char* myFile = "data.txt";

	//------------------------------------------------
	#ifdef USE_SD_CARD
		detachInterrupt(0);
		PORTB &= (uint8_t)~(1<<6);
		SDPowerCtrl(ON);

		#ifdef USE_TIMERS
			StartTimer();
		#endif
		if(InitializeSDCard() == 0) {
		#ifdef USE_TIMERS
			//sprintf(Func_name, "ISD");
			StopTimer();
		#endif

			File dataFile = SD.open(myFile, FILE_WRITE);

			#ifdef USE_TIMERS
				StartTimer();
			#endif

			if (dataFile) {
				//save the data in the order we took it
				memset(FlashBuf, 0, FLASH_BUF_SIZE);	//clear buffer, as
its global and we use it in the following

				//save EEPROM first
				if(!(error_flags & EEPROM_LIFE_EXCEEDED)) {
					for(uint8_t i = 0; i < NUM_CHUNKS; i++) {
						PORTB ^= (1<<6);
						ReadFromEEPROM(FlashBuf, (i<<7));
						ret = dataFile.print(FlashBuf);
						if(ret <= 0) {error_flags |= SMARTCAST_ERROR;
error_flags |= SDCARDWRITE_ERROR; TurnOff();} //write error
					}
				}

				//then save Flash
				if(!(error_flags & FLASH_LIFE_EXCEEDED)) {
					for(uint16_t i = start_page; i > end_page; i-
=SPM_PAGESIZE) {
						for(uint16_t j = 0; j < FLASH_BUF_SIZE; j++) {
							PORTB ^= (1<<6);
							FlashBuf[j] = ReadFlashByte(i, j);
						}
						ret = dataFile.print(FlashBuf);
						if(ret <= 0) {error_flags |= SMARTCAST_ERROR;
error_flags |= SDCARDWRITE_ERROR; TurnOff();} //write error
```

```
                                        }
                                }

                                //write error flags to SDCard
                                ret = sprintf(FlashBuf, "\nR%X\n", error_flags);
                                if(ret <= 0) {                  //if failed to write to temp buf
                                                error_flags |= SMARTCAST_ERROR;
                                                error_flags |= BUFF_ERROR;
                                } else {                        //write to temp buf succeeded
                                        ret = dataFile.print(FlashBuf);
                                        if(ret <= 0) {error_flags |= SMARTCAST_ERROR;
error_flags |= SDCARDWRITE_ERROR; TurnOff();} //write error
                                        else {ClearErrorFlags();}   //only clear the flags if
they were written to the SDCard successfully
                                }
                        } else {error_flags |= SMARTCAST_ERROR; error_flags |=
SDCARDFILE_ERROR; TurnOff();} //failed to open SDCard file

                        #ifdef USE_TIMERS
                                //sprintf(Func_name, "1stWrite");
                                StopTimer();
                        #endif //USE_TIMERS

                #endif //USE_SD_CARD
                //-------------------------------------------------

                //-------------------------------------------------
                #ifdef USE_SD_CARD
                        #ifdef USE_TIMERS
                                StartTimer();
                        #endif
                        dataFile.close();
                        #ifdef USE_TIMERS
                                //sprintf(Func_name, "CloseFile");
                                StopTimer();
                        #endif
                #endif //USE_SD_CARD


                #ifdef USE_SD_CARD
                /*#ifdef USE_TIMERS  //don't need GoToIdle now because we're turning the SD
card on and off each time
                                StartTimer();
                        #endif
                        if(!SD.card.GoToIdle(SPI_HALF_SPEED, hardwareSS)) {
                                error_flags |= SMARTCAST_ERROR;
                                error_flags |= SDCARDTIMEOUT_ERROR;
                        }
                        #ifdef USE_TIMERS
                                sprintf(Func_name, "GoToIdle");
                                StopTimer();
                        #endif
                */

                SDPowerCtrl(OFF);
                PORTB &= (uint8_t)~(1<<6);
                attachInterrupt(0, myINT0_Func, RISING);
                return 0;
```

```
        }
        else {
                SDPowerCtrl(OFF);
                PORTB &= (uint8_t)~(1<<6);
                attachInterrupt(0, myINT0_Func, RISING);
                //Serial.println("SDInitFail");
                return 1;
        }

        #endif //USE_SD_CARD
        //-------------------------------------------------
}
```

```
/*


 * Interrupts.h
 *
 * Created: 10/18/2012 3:31:44 PM
 *  Author: Andrew Danilovic
 */


#ifndef INTERRUPTS_H_
#define INTERRUPTS_H_

//for ADC measurements
extern volatile uint8_t hi,lo;

extern void myINT0_Func();
extern void myINT1_Func();

#endif /* INTERRUPTS_H_ */
```

```
/*
 * Interrupts.cpp
 *
 * Created: 10/18/2012 3:31:20 PM
 *  Author: Andrew Danilovic
 */

#include <avr/interrupt.h>
#include "Interrupts.h"
#include "Arduino.h"
#include "Sleep.h"
#include "SmartCast.h"

//for ADC measurements
volatile uint8_t hi,lo;

/*******************************************************************
* File: Interrupts.cpp
* NAME : ISR(BADISR_vect)
* Description: default interrupt vector, i.e. if no vector is
* specified for a particular interrupt but an interrupt occurs,
* this function is executed, shouldn't happen in normal execution
* Inputs:
* Outputs:
*******************************************************************/
ISR(BADISR_vect) {
       error_flags |= SMARTCAST_ERROR;
       error_flags |= INTERRUPT_ERROR;
}

/*******************************************************************
* File: Interrupts.cpp
* NAME : ISR(ADC_vect)
* Description: interrupt vector called when reading the ADC
* Inputs:
* Outputs: lo, hi, the bits of the ADC
*******************************************************************/
ISR(ADC_vect) {
       sleep_disable();
       lo = ADCL;
       hi = ADCH;
}

/*******************************************************************
* File: Interrupts.cpp
* NAME : void myINT0_Func()
* Description: ADXL345 interrupt 0, set flag for background execution
* of code in main loop
* Inputs:
* Outputs:
*******************************************************************/
//INT1 on the ADX345 goes to INT0
void myINT0_Func() {
       detachInterrupt(0);
       //read int source on the ADXL to clear the interrupt on the ADXL
       //uint8_t ret = myADXL345.getInterruptSource();
       //blink LEDs
       error_flags |= SMARTCAST_ERROR;
	error_flags |= INT0_TRIGGERED;
}

/*******************************************************************
* File: Interrupts.cpp
* NAME :myINT1_Func()
* Description: ADXL345 interrupt 1, set flag for background execution
* of code in main loop
* Inputs:
* Outputs:
*******************************************************************/
void myINT1_Func() {
	//EIMSK &= ~(1<<INT1);
	error_flags |= SMARTCAST_ERROR;
	error_flags |= INT1_TRIGGERED;
	//service_int = 1;
}
```

```c
/*
 * RunningStat.h
 *
 * Created: 8/26/2012 6:19:21 PM
 *  Original Code by: John Cook http://www.johndcook.com/standard_deviation.html
 *  Adapted by: Andrew Danilovic
 */


#ifndef RUNNINGSTAT_H_
#define RUNNINGSTAT_H_

#include <stdint.h>  //for Atmel data types, such as uint32_t

//#define TEST_STATS
//----------------------------
//----------------------------
struct RunningStat {
	int8_t m_n;
	float m_oldM, m_newM, m_oldS, m_newS;
};

void Clear(struct RunningStat *s);
void Push(struct RunningStat *s, uint32_t x);
uint32_t Mean(struct RunningStat *s);
uint32_t Variance(struct RunningStat *s);

#ifdef TEST_STATS
	int random(int range, int shift);
	float uniform_1_0();
	int uniform_a_b(int a, int b);
	uint32_t BoxMuller_custom(float mean, float std_dev);
	void TestStats();
#endif
//----------------------------
//----------------------------

#endif /* RUNNINGSTAT_H_ */
```

```
/*
 * RunningStat.cpp
 *
 * Created: 8/26/2012 7:08:29 PM
 *  Original Code by: John Cook http://www.johndcook.com/standard_deviation.html
 *  Adapted by: Andrew Danilovic
 */

#include "RunningStat.h"
#include <math.h>    //the Atmel math.h file
#include "Arduino.h"

/*******************************************************************
* File: RunningStat.cpp
* NAME : void Clear(struct RunningStat *s)
* Description: set m_n to 0 which in effect re-initializes s
* Inputs: s, a ptr to a RunningStat struct
* Outputs:
*******************************************************************/
void Clear(struct RunningStat *s) {
       s->m_n = 0;
}

/*******************************************************************
* File: RunningStat.cpp
* NAME : void Push(struct RunningStat *s, uint32_t x)
* Description: update statistics with next data point
* Inputs: s, x, a 32 bit unsigned number, i.e. the next data point
* Outputs:
*******************************************************************/
void Push(struct RunningStat *s, uint32_t x) {
       s->m_n++;

       if(s->m_n == 1)
       {
              s->m_oldM = s->m_newM = x;
              s->m_oldS = 0.0;
       }
       else
       {
              s->m_newM = s->m_oldM + (x - s->m_oldM)/s->m_n;
              s->m_newS = s->m_oldS + (x - s->m_oldM)*(x - s->m_newM);

              // set up for next iteration
              s->m_oldM = s->m_newM;
              s->m_oldS = s->m_newS;
       }
}

/*******************************************************************
* File: RunningStat.cpp
* NAME : uint32_t Mean(struct RunningStat *s)
* Description: calculate and return mean value of data
* Inputs: s
* Outputs: mean value of data so far
*******************************************************************/
uint32_t Mean(struct RunningStat *s) {
       return (s->m_n > 0) ? s->m_newM : 0.0;
```

```
}

/*****************************************************************
* File: RunningStat.cpp
* NAME : uint32_t Variance(struct RunningStat *s)
* Description: calculate and return variance of data
* Inputs: s
* Outputs: variance of data so far
*****************************************************************/
uint32_t Variance(struct RunningStat *s) {
	return ( (s->m_n > 1) ? s->m_newS/(s->m_n - 1) : 0 );
}

#ifdef TEST_STATS
/*****************************************************************
* File: RunningStat.cpp
* NAME : int random(int range, int shift)
* Description: generate random number from 0 to range, shifted by
* shift
* Inputs: range, shift
* Outputs: random number
*****************************************************************/
int random(int range, int shift) {
	return (rand() % range) + shift;
}

/*****************************************************************
* File: RunningStat.cpp
* NAME : float uniform_1_0()
* Description: generate uniform random float from 0 to 1
* Inputs:
* Outputs: uniform random float from 0 to 1
*****************************************************************/
float uniform_1_0() {
	return float(random(RAND_MAX, 0)) / RAND_MAX;
}

/*****************************************************************
* File: RunningStat.cpp
* NAME : int uniform_a_b(int a, int b)
* Description: generate uniform random number between a and b
* Inputs: a, b, the range of the uniform random number
* Outputs: uniform random float from a to b
*****************************************************************/
int uniform_a_b(int a, int b) {
	return random( (b-a), a);
}

/*****************************************************************
* File: RunningStat.cpp
* NAME : uint32_t BoxMuller_custom(float mean, float std_dev)
* Description: generate a normally distributed random number
* Inputs: desired mean, std_dev
* Outputs: normlly distributed random number
*****************************************************************/
uint32_t BoxMuller_custom(float mean, float std_dev) {

	float U = 0;
```

```
        float V = uniform_1_0();

        do {
                U = uniform_1_0();
        } while (U == 0);

        float X = sqrt(-2*log(U)) * cos(2*M_PI*V);
        float Y = sqrt(-2*log(U)) * sin(2*M_PI*V);

        return (X*std_dev) + mean;
}

/*********************************************************************
* File: RunningStat.cpp
* NAME : void TestStats()
* Description: top level function called to test the on-line statistics
* calculations. print to serial terminal so as to save to a log file
* on a PC
* Inputs:
* Outputs:
*********************************************************************/
//if you're using this function, remember to change the return values of Mean
//Variance to floats!
void TestStats() {

        RunningStat myStatTest;
        Clear(&myStatTest);

        randomSeed(43);

        /*
        Serial.println("uniform_1_0()");
        for(int i = 0; i < 10000; i++) {
                float ran = uniform_1_0();
                Push(&myStatTest, ran);
                Serial.println(ran, 4);
        }
        Serial.println("uniform_1_0(), m");
        Serial.println(Mean(&myStatTest), 4);
        Serial.println("uniform_1_0(), v");
        Serial.println(Variance(&myStatTest), 4);
        */

        /*
        Serial.println("uniform_0_32767()");
        for(int i = 0; i < 10000; i++) {
                uint32_t ran = uniform_a_b(0, RAND_MAX);
                Push(&myStatTest, ran);
                Serial.println(ran);
        }
        Serial.println("uniform_0_32767(), m");
        Serial.println(Mean(&myStatTest));
        Serial.println("uniform_0_32767(), v");
        Serial.println(Variance(&myStatTest));
        */

        /*
        Serial.println("BM_25000_1000()");
        for(int i = 0; i < 10000; i++) {
                uint32_t ran = BoxMuller_custom(25000, 1000);
                Push(&myStatTest, ran);
                Serial.println(ran);
        }
        Serial.println("BM_25000_1000(), m");
        Serial.println(Mean(&myStatTest));
        Serial.println("BM_25000_1000(), v");
        Serial.println(Variance(&myStatTest));

        while(1) {}
        */
}
#endif
```

```
/*
 * Sleep.h
 *
 * Created: 9/2/2012 7:54:42 PM
 *  Original code: Arduino Narcoleptic library, https://code.google.com/p/narcoleptic/
 *  Adapted by: Andrew Danilovic
 */


#ifndef SLEEP_H_
#define SLEEP_H_

#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <avr/sleep.h>

extern void __GoToSleep(uint8_t wdt_period);
extern void GoToSleep(uint32_t milliseconds);


#endif /* SLEEP_H_ */
```

```
/*
 * Sleep.cpp
 *
 * Created: 9/2/2012 7:54:48 PM
 *  Original code: Arduino Narcoleptic library, https://code.google.com/p/narcoleptic/
 *  Adapted by: Andrew Danilovic
 */

#include <Arduino.h>
#include "Sleep.h"
#include "SmartCast.h"

/*******************************************************************
* File: Sleep.cpp
* NAME : SIGNAL(WDT_vect)
* Description: watchdog timer interrupt, just reset it as
* it's used as a sleep timer
* Inputs:
* Outputs:
*******************************************************************/
SIGNAL(WDT_vect) {

       wdt_disable();
       wdt_reset();
       WDTCSR &= ~_BV(WDIE);
}

/*******************************************************************
* File: Sleep.cpp
* NAME : void __GoToSleep(uint8_t wdt_period)
* Description: put the MCU to sleep for wdt_period
* Inputs: wdt_period, amount of time to sleep for
* Outputs:
*******************************************************************/
void __GoToSleep(uint8_t wdt_period) {

       wdt_enable(wdt_period);                              //enable the Watchdog timer
to trigger after wdt_period milliseconds
       wdt_reset();                                         //reset the Watchdog timer
to begin counting from 0
       WDTCSR |= _BV(WDIE);                          //put watchdog timer in Interrupt
Mode (and possibly System Reset Mode depending on the value of WDE)
       set_sleep_mode(SLEEP_MODE_PWR_DOWN);     //enter Power-down Mode
       sleep_mode();                                        //go to sleep
       wdt_disable();                                              //disable the
Watchdog timer if possible
       WDTCSR &= ~_BV(WDIE);                                //disable the Watchdog
timer Interrupt Mode
}

/*******************************************************************
* File: Sleep.cpp
* NAME : void GoToSleep(uint32_t milliseconds)
* Description: put the MCU to sleep for milliseconds, calls internal
* function to implement the sleep, if greater than 8 seconds, the MCU
* will wake up and immediately go back to sleep, the watchdog timer
* can only be put to sleep for 8 seconds max
* Inputs: milliseconds, sleep time
* Outputs:
********************************************************************/
void GoToSleep(uint32_t milliseconds) {

       //sei();

       while (milliseconds >= 8000 && (!(error_flags & INT0_TRIGGERED))) {
              __GoToSleep(WDTO_8S); milliseconds -= 8000;
              if(error_flags & INT0_TRIGGERED) {
                     break;
              }
       }

       if (milliseconds >= 4000 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_4S); milliseconds -= 4000; }
       if (milliseconds >= 2000 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_2S); milliseconds -= 2000; }
       if (milliseconds >= 1000 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_1S); milliseconds -= 1000; }
       if (milliseconds >= 500 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_500MS); milliseconds -= 500; }
       if (milliseconds >= 250 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_250MS); milliseconds -= 250; }
       if (milliseconds >= 125 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_120MS); milliseconds -= 120; }
       if (milliseconds >= 64 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_60MS); milliseconds -= 60; }
       if (milliseconds >= 32 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_30MS); milliseconds -= 30; }
       if (milliseconds >= 16 && (!(error_flags & INT0_TRIGGERED)))
{ __GoToSleep(WDTO_15MS); milliseconds -= 15; }

       //cli();
}
```

```
/*
 * SmartCast.h
 *
 * Created: 7/22/2012 6:06:20 PM
 *  Author: Andrew Danilovic
 */

#ifndef SMARTCAST_H_
#define SMARTCAST_H_

#include <stdint.h>
#include "RunningStat.h"
#include "ADXL345.h"

#define MEASUREMENTDELAY_US 30 //us delay between resistance measurements
#define NUM_SENSORS 16
#define COLLECT_STAT_LOOP 10

//-------------------
//config variables
extern uint16_t sleep_time;
//extern uint8_t collect_stat_loop;
//-------------------

#define OFF 0
#define ON 1

#define TEMP_BUF_SIZE 25
//----------------------------
//--------Pin Defines---------
extern const uint8_t ADC_0;       //This is pin 23, i.e. ADC0
extern const uint8_t ADC_1;       //This is pin 24, i.e. ADC1
extern const uint8_t SEL4_PC2;    //A1 for analogMux2
extern const uint8_t SEL3_PC3;    //A0 for analogMux2

#define CONFIG_LED 7
#define SDCARD_LED 4
#define SEL2_PB0 8
#define SEL1_PD7 7
#define PWR_CTRL_PIN PD5
#define PWR_CTRL_SD_CARD 6
//----------------------------
//----------------------------

//SmartCast bitFields for error conditions
#define SMARTCAST_ERROR                 (1<<0)       //Active high
#define ACC_ERROR                       (1<<1)       //Active high
#define SDCARDTIMEOUT_ERROR       (1<<2)       //Active high
#define SDCARDWRITE_ERROR         (1<<3)       //Active high
#define SDCARDFILE_ERROR          (1<<4)       //Active high
#define BUFF_ERROR                      (1<<5)       //Active high
#define      INTERRUPT_ERROR                  (1<<6)       //BADISR_vect has been
called
#define      INT0_TRIGGERED                   (1<<7)       //
#define      INT1_TRIGGERED                   (1<<8)       //
#define      EEPROM_1_PAGE_LEFT         (1<<9)       //
#define      EEPROM_FULL                      (1<<10)             //
#define      EEPROM_LIFE_EXCEEDED (1<<11)             //
```

```
#define       FLASH_FULL                         (1<<12)             //
#define       FLASH_LIFE_EXCEEDED        (1<<13)             //
extern uint16_t error_flags;
//to clear a bit: error_flags &= ~SMARTCAST_ERROR;
//to set a bit: error_flags |= SMARTCAST_ERROR;
//to check a bit: if(error_flags & SMARTCAST_ERROR) {}
//to print error codes as a hex value: printf("%X\n", error_flags);
//or Serial.println(error_flags, HEX);  // print as an ASCII-encoded hexadecimal

extern uint16_t myT0;
extern uint16_t myT1;
extern char Func_name[15];

enum LED_state {LON, LOFF};
typedef struct LED {
       LED_state state;     //on or off
       uint8_t pin;
};
extern LED LED1, LED3;

//----------------------------
//----------------------------
extern ADXL345 myADXL345;

typedef struct myVoltage {
       uint8_t ones;
       uint8_t tenths;
};

extern uint32_t num_coll_samles;

extern char temp_buf[TEMP_BUF_SIZE];
//----------------------------
//----------------------------

extern void LED_OFF(struct LED *myLED);
extern void LED_ON(struct LED *myLED);
extern void LED_TOGGLE(struct LED *myLED);

extern void MeasureRMatrix(struct RunningStat Sensors[]);
extern void MeasureResistance_loop(struct RunningStat Sensors[]);
extern void MeasureVoltage();
extern void MeasureAcceleration();

extern void TakeMeasurements();
extern void RecordData();

extern void SCDumpFile(char* myFile);
extern void PowerCtrl(uint8_t on_off);
extern void SDPowerCtrl(uint8_t on_off);
extern void ClearErrorFlags();
extern void BlinkLEDs();
extern void SleepManagement();

extern void StartTimer();
extern void StopTimer();

extern void TurnOff();
```

```
#endif /* SMARTCAST_H_ */
```

```
/*
 * SmartCast.cpp
 *
 * Created: 8/26/2012 7:08:05 PM
 *  Author: Andrew Danilovic
 */

#include <avr/power.h>
#include "SmartCast.h"
#include "Arduino.h"
#include "RunningStat.h"
#include "../SmartCardApp/SmartCardUser.h"
#include "Sleep.h"
#include "Interrupts.h"
#include "../utility/myFlash.h"

void PrepareSleep();
void BlinkLED2();

#define RESOLUTION 86.0215053763   //Rf = 22k

#ifdef USE_TIMERS
       uint16_t myT0 = 0;
       uint16_t myT1 = 0;
       char Func_name[15] = {0};
#endif

//-------------------
//config variables
uint16_t sleep_time = 250;  //default
//uint8_t collect_stat_loop;
//-------------------

struct LED LED1, LED3;

uint16_t error_flags;

ADXL345 myADXL345;

char temp_buf[TEMP_BUF_SIZE];

struct RunningStat Sensors[NUM_SENSORS];

int accval[3];
myVoltage sysVoltage;
uint32_t num_coll_samles;

//----------------------------
//--------Pin Defines---------
//TODO: Make these consts
//These are here because you can't initialize variables to non-constant values in a C
file
const uint8_t ADC_0 = A0;          //This is pin 23, i.e. ADC0
const uint8_t ADC_1 = A1;          //This is pin 24, i.e. ADC1 //the MON channel
const uint8_t SEL4_PC2 = A2;       //A1 for analogMux2
const uint8_t SEL3_PC3 = A3;       //A0 for analogMux2
//----------------------------
//----------------------------
```

```
/*******************************************************************
* File: SmartCast.cpp
* NAME : uint32_t __MeasureResistance_loop(uint8_t counter)
* Description: set analog mux control signals to select a particular
* sensor based on counter. then, read adc value and convert to
* resistance in ohms
* Inputs: counter, 8 bit number corresponding to a particular
* sensor
* Outputs: ADC_val_int, resistance of a particular sensor
*******************************************************************/
uint32_t __MeasureResistance_loop(uint8_t counter)
{
       uint32_t ADC_val_int = 0;
       float ADC_val_float = 0;

       //create a 4 bit binary number to set the analog muxes appropriately
       uint8_t A0b = counter & 0x01;
       uint8_t A1b = (counter & 0x02) >> 1;
       uint8_t A0a = (counter & 0x04) >> 2;
       uint8_t A1a = (counter & 0x08) >> 3;

       digitalWrite(SEL1_PD7, A0a);      //A0
       digitalWrite(SEL2_PB0, A1a);      //A1
       digitalWrite(SEL3_PC3, A0b);      //A0
       digitalWrite(SEL4_PC2, A1b);      //A1

       //THIS DELAY IS VERY IMPORTANT!------------
       //you don't want to take a sample when the pins SEL1_PD7 etc. are switching,
       //so wait a bit until the voltages settle
       delayMicroseconds(MEASUREMENTDELAY_US);
       //THIS DELAY IS VERY IMPORTANT!------------

       ADC_val_int = analogRead(ADC_0);
       //the maximum analogRead can return is 1024
       //1 - AnalogRead/1024, this is from our equation of our circuit
       //Rg / (Vb-Va) = 22000 / (0.25) = 88000, this is the max R value
       //88000 / 1024 = 88000 >> 10 = 85.9375
       /*ADC_val_int = 1024 - ADC_val_int;
       ADC_val_float = (float)ADC_val_int * 85.9375;
       ADC_val_int = (uint32_t)ADC_val_float;
       return ADC_val_int;*/

       ADC_val_int = 1024 - ADC_val_int;
       ADC_val_float = (float)ADC_val_int * RESOLUTION;
       ADC_val_int = (uint32_t)ADC_val_float;
       return ADC_val_int;

       //Alternatively we could do:
       /*
       ADC_val_int = analogRead(ADC_0);
       ADC_val_int = 1024 - ADC_val_int;
       ADC_val_int = ADC_val_int * 50120;
       return ADC_val_int >> 10;
       */

       /*
       ADC_val_float = (float)ADC_val_int / 1024.0;
```

```
	ADC_val_float = 1.0 - ADC_val_float;
	ADC_val_int = ADC_val_float * 50120;	//
	return ADC_val_int;
	*/
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void __ClearStats(struct RunningStat Sensors[])
* Description: clear the on-line statistics, this is done
* for every wake event, i.e. the statistics are only calculated
* for a single wake event
* Inputs: ptr to RunningStat struct array
* Outputs:
*******************************************************************/
void __ClearStats(struct RunningStat Sensors[]) {
	for(uint8_t i = 0; i < (NUM_SENSORS); i++) {
		Clear(&Sensors[i]);
	}
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void MeasureResistance_loop(struct RunningStat Sensors[])
* Description: read sensor, and update online statistics for that
* sensor with the new value
* Inputs: ptr to runningstat array
* Outputs:
*******************************************************************/
void MeasureResistance_loop(struct RunningStat Sensors[]) {
	for(uint8_t i = 0; i < (NUM_SENSORS); i++) {
		Push(&Sensors[i], __MeasureResistance_loop(i));
	}
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void MeasureRMatrix(struct RunningStat Sensors[])
* Description: 1st clear statistics, then sample the matrix of
* sensors in a loop, toggle LED's for visible indication of sampling
* Inputs: ptr to runningstat array
* Outputs:
*******************************************************************/
void MeasureRMatrix(struct RunningStat Sensors[]) {
	LED_OFF(&LED1);
	PORTB &= (uint8_t)~(1<<6);

	__ClearStats(Sensors);		//clear statistics

	pinMode(ADC_0, INPUT);
	//ADCSRA |= (uint8_t)(1 << ADEN);	//turn ADC back on
	//ADCSRA |= (1<<ADIE);		//enable the ADC interrupt
	//ADMUX = (0<<REFS1) | (0<<REFS0) | (0<<ADLAR) | (0<<MUX3) | (0<<MUX2) | (0<<MUX1)
| (0<<MUX0);
	//set_sleep_mode(SLEEP_MODE_ADC);	//configure sleep mode for ADC measurements

	for(uint8_t i = 0; i < COLLECT_STAT_LOOP; i++) {
		LED_TOGGLE(&LED1);
```

```
			MeasureResistance_loop(Sensors);
			//Serial.println();
		}

		//ADCSRA &= ~(1<<ADIE);		//disable the ADC interrupt

		LED_OFF(&LED1);
		PORTB &= (uint8_t)~(1<<6);
}

/********************************************************************
* File: SmartCast.cpp
* NAME : void MeasureVoltage()
* Description: measure system voltage
* Inputs:
* Outputs:
********************************************************************/
void MeasureVoltage() {
		uint32_t ADC_val_int = 0;

		pinMode(ADC_1, INPUT);
		//e.g. analogRead = 838
		//val = 838 * 4 = 3352; 0.248 ~ 0.25 = 1/4, so just use 4 so we can bit shift
		//ones = 3352 / 1024 = 3.27 rounded down, so just 3
		//tenths = ((3352 * 100) / 1024)) - (ones * 100)

		//delay(100);

		ADC_val_int = analogRead(ADC_1);				//units = ADC counts
		sysVoltage.ones = ADC_val_int >> 8;
		ADC_val_int = ADC_val_int * 100;
		ADC_val_int = (ADC_val_int >> 8) - (sysVoltage.ones * 100);
		sysVoltage.tenths = ADC_val_int;

		//we have a voltage divider, with 1M and 330k
		//vout = vin * 330k / (1M + 330k) = 0.2481203008
		//we want to calculate Vin = Vout / 0.248
		//sysVoltage = ADC_val_float / 0.248;
		//return sysVoltage;
}

/********************************************************************
* File: SmartCast.cpp
* NAME : void MeasureAcceleration()
* Description: measure system acceleration
* Inputs:
* Outputs:
********************************************************************/
void MeasureAcceleration() {
		myADXL345.readAccel(&accval[0], &accval[1], &accval[2]);
}

/********************************************************************
* File: SmartCast.cpp
* NAME : void TakeMeasurements()
* Description: perform measurements, optionally time each measurement
* for debugging
* Inputs:
```

```
* Outputs:
*******************************************************************/
void TakeMeasurements() {

       num_coll_samles++;

       #ifdef USE_TIMERS
              StartTimer();
       #endif
                     MeasureVoltage();
       #ifdef USE_TIMERS
              //sprintf(Func_name, "MV");
              StopTimer();
       #endif

       #ifdef USE_TIMERS
              StartTimer();
       #endif
                     MeasureRMatrix(Sensors);
       #ifdef USE_TIMERS
              //sprintf(Func_name, "MR");
              StopTimer();
       #endif

       #ifdef USE_TIMERS
              StartTimer();
       #endif
                     //MeasureAcceleration();
       #ifdef USE_TIMERS
              //sprintf(Func_name, "MA");
              StopTimer();
       #endif
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void RecordData()
* Description: convert measurement data to strings and log
* measurement data to memory with a particular format
* Inputs:
* Outputs:
*******************************************************************/
void RecordData() {

       int16_t ret = 0;

       //fill num_samples buffer
       ret = sprintf_P(temp_buf, PSTR("%lu\n"), num_coll_samles);
       if(ret <= 0) {
              error_flags |= SMARTCAST_ERROR;
              error_flags |= BUFF_ERROR;
       } else {Serial.print(temp_buf); /*WriteToMemory(temp_buf);*/}

       //fill accelerometer buffer
       ret = sprintf_P(temp_buf, PSTR("%d %d %d\n"), accval[0], accval[1], accval[2]);
       if(ret <= 0) {
              error_flags |= SMARTCAST_ERROR;
              error_flags |= BUFF_ERROR;
```

```
	} else {Serial.print(temp_buf); /*WriteToMemory(temp_buf);*/}

	//fill voltage buffer
	if(sysVoltage.tenths < 10) {
		ret = sprintf_P(temp_buf, PSTR("%u.0%u\n"), sysVoltage.ones, 
sysVoltage.tenths);
	}
	else {
		ret = sprintf_P(temp_buf, PSTR("%u.%u\n"), sysVoltage.ones, 
sysVoltage.tenths);
	}
	if(ret <= 0) {
		error_flags |= SMARTCAST_ERROR;
		error_flags |= BUFF_ERROR;
	} else {Serial.print(temp_buf); /*WriteToMemory(temp_buf);*/}

	for(uint8_t i = 0; i < NUM_SENSORS; i++) {
		ret = sprintf_P(temp_buf, PSTR("%lu %lu\n"), Mean(&Sensors[i]), 
Variance(&Sensors[i]));
		if(ret <= 0) {
			error_flags |= SMARTCAST_ERROR;
			error_flags |= BUFF_ERROR;
		} else {Serial.print(temp_buf); /*WriteToMemory(temp_buf);*/}
	}

	//----------------------
	//ret = sprintf(temp_buf, "R%X\n", error_flags);
	//Serial.print(temp_buf);
	//Serial.println();
	//----------------------
}

void SCDumpFile(char* myFile) {
	#ifdef INCLUDECHECKSD
		checkSDCard();
	#endif

	#ifdef INCLUDEDUMPFILE
		DumpFile(myFile);
	#endif
}

void LED_OFF(struct LED *myLED) {
	digitalWrite(myLED->pin, OFF);
	myLED->state = LOFF;
}
void LED_ON(struct LED *myLED) {
	digitalWrite(myLED->pin, ON);
	myLED->state = LON;
}

void LED_TOGGLE(struct LED *myLED) {
	if(myLED->state == LOFF) LED_ON(myLED);
	else LED_OFF(myLED);
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void PowerCtrl(uint8_t on_off)
* Description: turn on or off power to analog sub-systems
* Inputs:
* Outputs:
*******************************************************************/
void PowerCtrl(uint8_t on_off) {
       digitalWrite(PWR_CTRL_PIN, on_off);
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void SDPowerCtrl(uint8_t on_off)
* Description: turn on or off power to SD Card, delay when turning
* on to allow SD Card to initialize properly
* Inputs:
* Outputs:
*******************************************************************/
void SDPowerCtrl(uint8_t on_off) {
       digitalWrite(PWR_CTRL_SD_CARD, on_off);
       if(on_off == ON) {delay(100);}
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void ClearErrorFlags()
* Description: clear error flags, this function is only called if
* the SDCard write was successful
* Inputs:
* Outputs:
*******************************************************************/
void ClearErrorFlags() {

       //Serial.print("f");
       //Serial.println(error_flags, HEX);

       //reset only some of the error flags, i.e. don't reset the interrupt flags, do
that in the background process
       error_flags &=
~((SMARTCAST_ERROR)|(ACC_ERROR)|(SDCARDTIMEOUT_ERROR)|(SDCARDWRITE_ERROR)|
              (SDCARDFILE_ERROR)|(BUFF_ERROR)|(INTERRUPT_ERROR)|(INT1_TRIGGERED));
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void TurnOff()
* Description: turn off power to all sub systems and sleep indefinitely,
* called in case an error occurs to
* Inputs:
* Outputs:
*******************************************************************/
void TurnOff() {
       detachInterrupt(0);
       Serial.println("TurnOff()");
       while(1) {
              SleepManagement();
              BlinkLED2();
       }
}
```

```
/*******************************************************************
* File: SmartCast.cpp
* NAME : void BlinkLEDs()
* Description: tblink system LEDs as visual indication of activity
* Inputs:
* Outputs:
*******************************************************************/
void BlinkLEDs() {
       PORTB |= (uint8_t)(1<<6);
       PORTB |= (uint8_t)(1<<CONFIG_LED);
       LED_ON(&LED1);
       LED_ON(&LED3);
       delay(100);
       PORTB &= (uint8_t)~(1<<6);
       LED_OFF(&LED1);
       LED_OFF(&LED3);
       PORTB &= (uint8_t)~(1<<CONFIG_LED);
}

void BlinkLED2() {
       PORTB |= (uint8_t)(1<<6);
       delay(100);
       PORTB &= (uint8_t)~(1<<6);
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void PrepareSleep()
* Description: turn off all the pins we don't need while sleeping,
* and prepare system for sleep
* Inputs:
* Outputs:
*******************************************************************/
void PrepareSleep() {

       digitalWrite(SEL1_PD7, OFF);      //A0
       digitalWrite(SEL2_PB0, OFF);      //A1
       digitalWrite(SEL3_PC3, OFF);      //A0
       digitalWrite(SEL4_PC2, OFF);      //A1

       ADCSRA &= (uint8_t)~(1 << ADEN);  //This saves an extra 0.1mA!
       //power_adc_disable();
       //ACSR |= (uint8_t)(1 << ACD);

       //----------------------------------------
       //turn off SD card, i.e. turn off SPI interface
       SPCR &= (uint8_t)~(1 << SPE);     //This turns off the SPI interface

       //pinMode(SPI_MISO_PIN, OUTPUT);  //DO NOT DO THIS, THIS CAUSES CURRENT TO LEAK TO
GROUND
       //digitalWrite(SPI_MISO_PIN, LOW);

       pinMode(hardwareSS, OUTPUT);      //make sure there is no voltage on the SPI pins
       digitalWrite(hardwareSS, LOW);
       pinMode(SPI_MOSI_PIN, OUTPUT);
       pinMode(SPI_SCK_PIN, OUTPUT);
       digitalWrite(SPI_MOSI_PIN, LOW);
```

```
        digitalWrite(SPI_SCK_PIN, LOW);
        //----------------------------------------

        //myADXL345.powerOff();

        LED_OFF(&LED1);
        LED_OFF(&LED3);
        PORTB &= (uint8_t)~(1<<6);

        SDPowerCtrl(OFF);
        PowerCtrl(OFF);
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void PrepareWake()
* Description: prepare system for wake up
* Inputs:
* Outputs:
*******************************************************************/
void PrepareWake() {

        //power_adc_enable();
        ADCSRA |= (uint8_t)(1 << ADEN);    //turn ADC back on

        PowerCtrl(ON);

        //Serial.println("turnon1");
        //MeasureAcceleration();
        //myADXL345.powerOn();
        //Serial.println("turnon2");
        //MeasureAcceleration();

        //THIS DELAY IS VERY IMPORTANT!------------
        //we need to wait for the analog reference voltage to settle
        delayMicroseconds(500);
        //THIS DELAY IS VERY IMPORTANT!------------

        //myADXL345.powerOn();
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : uint32_t __CalcSleepTime()
* Description: currently, just return global variable, but in the
* future, we could calculate sleep time based of system voltage etc.
* Inputs:
* Outputs:
*******************************************************************/
uint32_t __CalcSleepTime() {

        return sleep_time;

        //calc sleep time as a function of the system voltage
        //uint16_t temp_volt = sysVoltage.ones + sysVoltage.tenths/10;
        /*if(sysVoltage.ones >= 2) {
                if(sysVoltage.tenths >= 70) {      //i.e. 2.70
                        return 10;
```

```
              }
              else {TurnOff();}
       } else {TurnOff();}
       */
}

/*******************************************************************
* File: SmartCast.cpp
* NAME : void SleepManagement()
* Description: top level sleep function, prepares system for sleep
* and wake
* Inputs:
* Outputs:
*******************************************************************/
void SleepManagement() {
       PrepareSleep();
       GoToSleep(__CalcSleepTime());
       PrepareWake();
}

#ifdef USE_TIMERS
void StartTimer() {
       PORTB |= (uint8_t)(1<<7);
       myT0 = (uint16_t)millis();

}
void StopTimer() {
       myT1 = (uint16_t)millis();
       PORTB &= (uint8_t)~(1<<7);
       Serial.print(Func_name);
       Serial.print(" ");
       Serial.print(myT1 - myT0);
       Serial.println("ms");
}
#endif
```

```
/*
 * myFlash.h
 *
 * Created: 10/31/2012 5:55:03 PM
 *  Author: Andrew Danilovic
 */


#ifndef MYFLASH_H_
#define MYFLASH_H_

#include <avr/pgmspace.h>
#include <avr/boot.h>
#include "../SmartCastApp/SmartCast.h"

//Use SPM_PAGESIZE, which is also defined to be 128
//#define FLASH_PAGE_SIZE 128      //64 words = 128 bytes, see p. 302 in datasheet

//need to add 1 to line up with the page boundaries, i.e. FLASHEND is 0x7FFF, not 0x8000
//#define LAST_ACCESSIBLE_PAGE (((FLASHEND)-640)+1)    //currently the bootloader is
taking up 4 pages, so 1st page is 128*5=640 less
#define LAST_ACCESSIBLE_PAGE 32128 //hardcoded value, when the bootloader is only 4 pages
#define MAX_FLASH_WRITES 9500      //Flash has a 10,000 write/erase cycle lifetime

#define FLASH_BUF_SIZE SPM_PAGESIZE
//#define FLASH_BUF_SIZE 64

extern char FlashBuf[FLASH_BUF_SIZE];

BOOTLOADER_SECTION extern void __WriteToFlash(uint16_t page, char *buf);
extern void WriteToMemory(char *buf);

BOOTLOADER_SECTION extern char ReadFlashByte(uint16_t page, uint8_t offset);

extern uint32_t RoundPow(uint32_t numToRound, uint8_t multiple);
extern void initFlashPageIndices();

#endif /* MYFLASH_H_ */
```

```
/*
 * myFlash.cpp
 *
 * Created: 10/31/2012 5:54:53 PM
 *  Author: Andrew Danilovic
 */

#include "myFlash.h"
#include "../SmartCardApp/SmartCardUser.h"
#include <avr/eeprom.h>
#include "myEEPROM.h"
#include "../SmartCastApp/SmartCast.h"

void DecreaseFlashIndex();
void ResetFlashPtr();

char FlashBuf[FLASH_BUF_SIZE];

uint16_t flash_page_index;                          //point to the current flash page
extern int __data_load_end;                         //use this to avoid overwriting program
memory
uint16_t first_accessible_page;                     //this tells us which is the first page
available
int16_t flash_pages_avail;                          //keep track of how many pages are
available
uint16_t num_flash_writes;                          //use this to avoid going past the
lifetime of the flash

/*********************************************************************
* File: myFlash.cpp
* NAME : void __WriteToFlash(uint16_t page, char *buf)
* Description: writed a buffer of data to flash memory.
* adapted from the non-gnu Atmel online manual, see:
* http://www.nongnu.org/avr-libc/user-manual/group__avr__boot.html
* Inputs: page to write to, and buffer to write to memory
* Outputs:
*********************************************************************/
void __WriteToFlash(uint16_t page, char *buf) {

       uint16_t i;
       uint8_t sreg;

       // Disable interrupts.
       sreg = SREG;
       cli();

       eeprom_busy_wait();

       boot_page_erase(page);
       boot_spm_busy_wait();      // Wait until the memory is erased.

       for (i=0; i<SPM_PAGESIZE; i+=2)
       {
              // Set up little-endian word.
              uint16_t w = *buf++;
              w += (*buf++) << 8;

              boot_page_fill(page + i, w);
	}

	boot_page_write(page);     // Store buffer in flash page.
	boot_spm_busy_wait();       // Wait until the memory is written.

	// Reenable RWW-section again. We need this if we want to jump back
	// to the application after bootloading.
	boot_rww_enable();

	// Re-enable interrupts (if they were ever enabled).
	SREG = sreg;
}

/*********************************************************************
* File: myFlash.cpp
* NAME : uint8_t TrySDCardWrite()
* Description: tries to write to the SD Card, if successful,
* reset flash and eeprom memory pointers and indices as we
* successfully wrote the data in those memories to the SD Card
* Inputs:
* Outputs: return 0 if successful, 1 otherwise
*********************************************************************/
uint8_t TrySDCardWrite() {
	//----------------write to SDCard memory----------------
	if(WriteToSDCardbuf(LAST_ACCESSIBLE_PAGE, first_accessible_page) == 0) {
		ResetFlashPtr();     //ONLY RESET FLASH PTR IF SDCARD write was successful
		ResetEEPROMPtr();
		return 0;
	}
	else {return 1;}     //no SDCard present
}

/*********************************************************************
* File: myFlash.cpp
* NAME : void WriteToFlash()
* Description: this function is misnamed and needs refactoring.
* this function contains the memory management logic, i.e.
* write to EEPROM until, then write to Flash, then when both are full,
* write all that data to the SD Card. This code could also be cleaned up,
* but this has been tested to work
* Inputs:
* Outputs:
*********************************************************************/
void WriteToFlash() {
	if(error_flags & FLASH_LIFE_EXCEEDED) {                              //FLASH has
exceeded
		if( (error_flags & EEPROM_LIFE_EXCEEDED) == 0) {//EEPROM has not exceeded
			if( (error_flags & EEPROM_1_PAGE_LEFT) == 0) {
				WriteToEEPROM(FlashBuf);
				memset(FlashBuf, 0, FLASH_BUF_SIZE);     //clear buffer for
next go around
			}
			else { //there's just 1 EEPROM page left
				WriteToEEPROM(FlashBuf);
				memset(FlashBuf, 0, FLASH_BUF_SIZE);     //clear buffer for
next go around
				//************************
```

```
                        while(TrySDCardWrite() == 1) {     //i.e. while SDCard write
not successful
                                SleepManagement();
                        }
                        //************************
                    }
            }
            else { /*There is no internal memory left at this point*/
                    //this will just write the error flags to the SDCard
                    WriteToSDCardbuf(LAST_ACCESSIBLE_PAGE, first_accessible_page);
                    //just turn off really
            }
    }
    else { //Flash is still working
            if( (error_flags & EEPROM_LIFE_EXCEEDED) == 0) {//EEPROM has not exceeded
                    if( (error_flags & EEPROM_FULL) == 0) {   //EEPROM is not full
                            WriteToEEPROM(FlashBuf);
                            memset(FlashBuf, 0, FLASH_BUF_SIZE);      //clear buffer for
next go around
                    }
                    else { //EEPROM is full, go to Flash
                            if(flash_pages_avail > 1) {
                                    __WriteToFlash(flash_page_index, FlashBuf);
                                    DecreaseFlashIndex();
                                    memset(FlashBuf, 0, FLASH_BUF_SIZE);      //clear
buffer for next go around
                            }
                            else {
                                    if(flash_pages_avail > 0) { //there's just 1 flash page
left, so fill it and write to the SDCard
                                            __WriteToFlash(flash_page_index, FlashBuf);
                                            DecreaseFlashIndex();
                                            memset(FlashBuf, 0, FLASH_BUF_SIZE);
        //clear buffer for next go around
                                    }
                                    //************************
                                    while(TrySDCardWrite() == 1) {     //i.e. while SDCard
write not successful
                                            SleepManagement();   //go to sleep until we can
write to the SD Card
                                    }
                                    //************************
                            }
                    }
            }
            else {/*EEPROM has exceeded before Flash, which should never happen*/}
    }
}

/*******************************************************************
* File: myFlash.cpp
* NAME : WriteToMemory(char *buf)
* Description: top level function called by the SmartCast app to save
* data to memory
* Inputs: buffer of data
* Outputs:
*******************************************************************/
void WriteToMemory(char *buf) {
	int16_t lenTempBuf = strlen(buf);
	int16_t lenFlashBuf = strlen(FlashBuf);
	uint8_t room_left = FLASH_BUF_SIZE - lenFlashBuf;

	if(lenTempBuf <= room_left) {
		strncat(FlashBuf, buf, lenTempBuf);
	}
	else {
		strncat(FlashBuf, buf, room_left); //fill up the buffer before writing to
flash
		WriteToFlash();
		memset(FlashBuf, 0, FLASH_BUF_SIZE);	//clear buffer for next go around
		lenFlashBuf = strlen(FlashBuf);	//don't forget to copy the !REMAINING!
part of the buffer that put us over the limit!
		if((lenTempBuf + lenFlashBuf - room_left) <= FLASH_BUF_SIZE) {
			strncat(FlashBuf, buf+room_left, lenTempBuf-room_left);
		}
		else {}		//Flash buffer overflow error, shouldn't happen, i.e. we don't
send that much at once
	}
}

/*********************************************************************
* File: myFlash.cpp
* NAME : char ReadFlashByte(uint16_t page, uint8_t offset)
* Description: read from Flash
* Inputs: page and offset to read from
* Outputs: 8 bit data from Flash
*********************************************************************/
char ReadFlashByte(uint16_t page, uint8_t offset) {
	return pgm_read_byte((const uint16_t)page+offset);
}

/*********************************************************************
* File: myFlash.cpp
* NAME : extern void initFlashPageIndices()
* Description: init indices to Flash memory, these indices
* allow the program to know which addresses of Flash can be written to
* Inputs:
* Outputs:
*********************************************************************/
extern void initFlashPageIndices() {
	first_accessible_page = RoundPow((int)&__data_load_end, SPM_PAGESIZE);
	flash_page_index = LAST_ACCESSIBLE_PAGE;
	flash_pages_avail = (LAST_ACCESSIBLE_PAGE - first_accessible_page)>>7;
}

/*********************************************************************
* File: myFlash.cpp
* NAME : extern void DecreaseFlashIndex()
* Description: decrement index with error checking
* Inputs:
* Outputs:
*********************************************************************/
extern void DecreaseFlashIndex() {
	if(flash_page_index >= SPM_PAGESIZE) {	//don't decrement flash_page_index unless
it is larger than SPM_PAGESIZE
		flash_page_index -= SPM_PAGESIZE;
```

```
        }
        flash_pages_avail = (flash_page_index - first_accessible_page)>>7;
        if(flash_pages_avail <= 0) {
                error_flags |= SMARTCAST_ERROR;
                error_flags |= FLASH_FULL;
        }
}

/******************************************************************
* File: myFlash.cpp
* NAME : extern void ResetFlashPtr()
* Description: reset flash indices and increment write counter,
* as Flash has a write lifetime, don't want to exceed that
* Inputs:
* Outputs:
******************************************************************/
extern void ResetFlashPtr() {
        if(num_flash_writes < MAX_FLASH_WRITES) {
                num_flash_writes++;  //increment counter here because we will be writing to
flash in a round robin fashion
                flash_page_index = LAST_ACCESSIBLE_PAGE;
                flash_pages_avail = (LAST_ACCESSIBLE_PAGE - first_accessible_page)>>7;
                error_flags &= ~FLASH_FULL;
        }
        else {
                /*stop writing to flash*/
                flash_page_index = LAST_ACCESSIBLE_PAGE;
                flash_pages_avail = 0;
                error_flags |= SMARTCAST_ERROR;
                error_flags |= FLASH_LIFE_EXCEEDED;
        }
}

/******************************************************************
* File: myFlash.cpp
* NAME : uint32_t RoundPow(uint32_t numToRound, uint8_t multiple)
* Description: round a number up to a specific multiple
* Inputs: number to round and multiple
* Outputs: rounded number
******************************************************************/
uint32_t RoundPow(uint32_t numToRound, uint8_t multiple) {

        if(numToRound > 32767) {    //32*1024 = 32768, subtract to account for the -1 sign
bit
                //this page does not exist in the ATMega328p, so make this equal to
FLASHEND-512, i.e. the last available page
                //this way, when we later check if first_accessible_page <
LAST_ACCESSIBLE_PAGE, we'll
                //fail there if first_accessible_page is larger
                return FLASHEND-512; //i.e. subtract the size of the bootloader, in this
case 4 pages, 4*128 = 512
        }

        if(multiple == 0) {
                return numToRound;
        }

        uint32_t remainder = numToRound % multiple;
```

```
	if(remainder == 0) {
		return numToRound;
	}

	if(remainder > ( numToRound + multiple ) ) {
		return FLASHEND>>7;		//don't want to return a negative number
	}

	return numToRound + multiple - remainder;
}
```

```c
/*
 * myEEPROM.h
 *
 * Created: 10/31/2012 5:53:50 PM
 *  Author: Andrew Danilovic
 */


#ifndef MYEEPROM_H_
#define MYEEPROM_H_

#include <avr/eeprom.h>

#define EEPROMSizeATmega328p   1024
#define NUM_CHUNKS 8
#define MAX_NUM_WRITES 99500

extern void InitEEPROMPtrs();
extern void WriteToEEPROM(char *buf);
extern void ResetEEPROMPtr();
extern void ReadFromEEPROM(char *buf, uint16_t address);

#endif /* MYEEPROM_H_ */
```

```
/*
 * myEEPROM.cpp
 *
 * Created: 10/31/2012 5:53:37 PM
 *  Author: Andrew Danilovic
 */

#include "myEEPROM.h"
#include <avr/interrupt.h>
#include "myFlash.h"
#include "../SmartCastApp/SmartCast.h"

uint16_t __address;
uint16_t num_EEPROM_writes;

/*******************************************************************
* File: myEEPROM.cpp
* NAME : void InitEEPROMPtrs()
* Description: initialize pointers to EEPROM memory and write counter
* Inputs:
* Outputs:
*******************************************************************/
void InitEEPROMPtrs() {
       __address = 0;
       num_EEPROM_writes = 0;
}

/*******************************************************************
* File: myEEPROM.cpp
* NAME : void WriteToEEPROM(char *buf)
* Description: write buffer to eeprom, with error checking to ensure
* the lifetime is not exceeded before write and valid address
* Inputs: buffer of data
* Outputs:
*******************************************************************/
void WriteToEEPROM(char *buf) {

       if((error_flags & EEPROM_LIFE_EXCEEDED)|(error_flags & EEPROM_FULL)) {return;}
//don't write anymore

       uint8_t sreg = SREG;
       cli();
       eeprom_busy_wait();
       eeprom_write_block(buf, (uint16_t *)__address, FLASH_BUF_SIZE);
       SREG = sreg;

       __address += FLASH_BUF_SIZE;
       if(__address == 896) {      //this is the 2nd to last free page of EEPROM
              error_flags |= EEPROM_1_PAGE_LEFT;
       } else {error_flags &= ~EEPROM_1_PAGE_LEFT;}

       if(__address >= EEPROMSizeATmega328p) {
              error_flags |= EEPROM_FULL;
       }
}

/*******************************************************************
* File: myEEPROM.cpp
* NAME : void ResetEEPROMPtr()
* Description: reset eeprom indices, and increment write counter
* Inputs:
* Outputs:
*******************************************************************/
void ResetEEPROMPtr() {
	if(num_EEPROM_writes < MAX_NUM_WRITES) {
		num_EEPROM_writes++;
		__address = 0;
		error_flags &= ~EEPROM_FULL;
		error_flags &= ~EEPROM_1_PAGE_LEFT;
	}
	else {
		error_flags |= EEPROM_LIFE_EXCEEDED;  //don't write anymore
	}
}

/*******************************************************************
* File: myEEPROM.cpp
* NAME : void ReadFromEEPROM(char *buf, uint16_t address)
* Description: read into buf from address
* Inputs: buf, address
* Outputs:
*******************************************************************/
void ReadFromEEPROM(char *buf, uint16_t address) {
	if(__address > EEPROMSizeATmega328p) {return;}
	uint8_t sreg = SREG;
	cli();
	eeprom_busy_wait();
	eeprom_read_block(buf, (uint16_t *)address, FLASH_BUF_SIZE);
	SREG = sreg;
}
```

```
/*
 * SDCardCORETes1a.cpp
 *
 * Created: 5/10/2012 10:46:24 PM
 *  Author: Andrew Danilovic
 */

/*
Notes:

The project is now linking to libm.a, which is the Atmel
math.h library. This has been hand optimized for the Atmel
processors, and is better then the GCC library, according to Google.
I believe you can also add just 'm' in the project properties
under Linker->Libraries as per this site:
http://support.atmel.com/bin/customer.exe?=&action=viewKbEntry&id=339

*/

#include "Arduino.h"
#include "SmartCastApp/SmartCast.h"
#include "SmartCastApp/ADXL345.h"
#include "SmartCardApp/SmartCardUser.h"
#include "SmartCastApp/Interrupts.h"
#include "SmartCastApp/Sleep.h"
#include "utility/memdebug.h"
#include "utility/StackPaint.h"
#include "utility/myFlash.h"
#include "utility/myEEPROM.h"
#include "SmartCastApp/RunningStat.h"
#include "SmartCardApp/ConfigFile.h"

/*******************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void InitializePins()
* Description: initialize all required pins for the SmartCast app
* Inputs:
* Outputs:
*******************************************************************/
void InitializePins() {

       //-------------------------------
       //--------Configure Pins---------
       pinMode(SEL1_PD7, OUTPUT);
       pinMode(SEL2_PB0, OUTPUT);
       pinMode(SEL3_PC3, OUTPUT);
       pinMode(SEL4_PC2, OUTPUT);
       pinMode(PWR_CTRL_PIN, OUTPUT);
       pinMode(PWR_CTRL_SD_CARD, OUTPUT);
       pinMode(hardwareSS, OUTPUT);    //This is the hardware SS pin, must be set as
output for the SD library to work

       pinMode(ADC_0, OUTPUT);     //just set these here, not really needed
       pinMode(ADC_1, OUTPUT);

       pinMode(ADXL345_INT1_PIN, INPUT);
       pinMode(ADXL345_INT2_PIN, INPUT);
```

```
        digitalWrite(SEL1_PD7, LOW);
        digitalWrite(SEL2_PB0, LOW);
        digitalWrite(SEL3_PC3, LOW);
        digitalWrite(SEL4_PC2, LOW);
        PowerCtrl(OFF);
        SDPowerCtrl(OFF);
        //-------------------------------
        //-------------------------------

        //-------------------------------
        //---------DEBUG Pins------------
        DDRB |= (uint8_t)(1<<7);    //set PB7 as an output, for Debug
        PORTB &= (uint8_t)~(1<<7);
        //-------------------------------
        //-------------------------------

        //-------------------------------
        //-------Initialize LEDs---------
        LED1.pin = 9;
        pinMode(LED1.pin, OUTPUT);
        LED_OFF(&LED1);

        LED3.pin = SDCARD_LED;
        pinMode(LED3.pin, OUTPUT);
        LED_OFF(&LED3);

        DDRB |= (uint8_t)(1<<6);                  //set PB6 as an output, one of the LEDs
        PORTB &= (uint8_t)~(1<<6);

        DDRB |= (uint8_t)(1<<CONFIG_LED);         //set PB7 as an output, one of the LEDs
        PORTB &= (uint8_t)~(1<<CONFIG_LED);
        //-------------------------------
        //-------------------------------
}

/*******************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void InitializeSerial()
* Description: initialize the serial port
* Inputs:
* Outputs:
*******************************************************************/
void InitializeSerial() {

        Serial.begin(9600);

        //Calibrate the serial connection
        /*
        while(1) {
               Serial.print("(abc, ");
               Serial.print(8);
               Serial.print(") ");
               //delay(500);
        }
        */
        /*

        uint32_t test12 = 0;
```

```
	while(1) {
		UBRR0L = test12;
		Serial.print("(abc, ");
		Serial.print(test12);
		Serial.print(") ");
		test12 += 1;
	}
	*/

	UBRR0L = 8;   //needed to calibrate the serial comm, because we're using the
onboard clock
}

/*******************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void InitializeMisc()
* Description: initialize other necessary variables and turn power
* on to analog sub system
* Inputs:
* Outputs:
*******************************************************************/
void InitializeMisc() {

	analogReference(EXTERNAL);

	error_flags = 0;
	num_coll_samles = 0;
	PowerCtrl(ON);
}

/*******************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void InitializeInternalMem()
* Description: initialize memory indices
* Inputs:
* Outputs:
*******************************************************************/
void InitializeInternalMem() {
	InitEEPROMPtrs();
	initFlashPageIndices();
}

/*******************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void setup(void)
* Description: top level initialize function, only called once at boot
* Inputs:
* Outputs:
*******************************************************************/
void setup(void) {

	uint8_t acc_data[5] = {0x19, 0x15, 0x15, 0x50, 0xA0};  //default values

	InitializePins();
	InitializeSerial();
	InitializeMisc();
	BlinkLEDs();
```

```
        //ParseConfigFile(acc_data);
        InitializeInternalMem();

        //-------------ADXL345 Initialize--------------
        //myADXL345.init(ADXL345_ADDR_ALT_LOW, acc_data);
        //----------------------------------------------
        //if(no errors) {
               BlinkLEDs();  //shows we're done initializing and ready to begin sampling
        //}
        //else {
        //     Blink in a certain pattern to indicate error
        //}
}

/********************************************************************
* File: SDCardCORETes1a.cpp
* NAME : void loop()
* Description: main infinite loop, measure sensors, log data,
* go to sleep, handle background processing of interrupts on wake,
* interrupts will wake up the microcontroller
* Inputs:
* Outputs:
********************************************************************/
void loop() {

        TakeMeasurements();
        RecordData();
        SleepManagement();

        if(error_flags & INT0_TRIGGERED) {
               error_flags &= ~INT0_TRIGGERED;
               uint8_t ret = myADXL345.getInterruptSource();
               attachInterrupt(0, myINT0_Func, RISING);
        }
}
```

References

[1] Atmel. "ATmega328p Datasheet." Atmel.com, Feb. 2013. Web. <http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet.pdf>.

[2] "AVR Libc." *AVR Libc*. Savannah.nongnu.org, 3 Jan. 2012. Web. <http://www.nongnu.org/avr-libc/user-manual/index.html>.

[3] Brady, S. et al. "The Development and Characterisation of Conducting Polymeric-based Sensing Devices." *Synthetic Metals* 154.1-3 (2005): 25-28. *ScienceDirect*. Web. <http://www.sciencedirect.com/science/article/pii/S0379677905004613#>.

[4] Buechley, Leah. "LilyPad Arduino: How an Open Source Hardware Kit Is Sparking New Engineering and Design Communities." Web. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.168.5521>.

[5] Carpi, F., and D. De Rossi. "Electroactive Polymer-Based Devices for E-Textiles in Biomedicine." *Information Technology in Biomedicine, IEEE Transactions on*9.3 (2005): 295-318. *IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1504800>.

[6] Cook, John D. "Accurately Computing Running Variance." John D. Cook, n.d. Web. 1 May 2013. <www.johndcook.com>.

[7] Dunne, L., S. Brady, B. Smyth, and D. Diamond. "Initial Development and Testing of A Novel Foam-Based Pressure Sensor for Wearable Sensing." *Journal of Neuroengineering and Rehabilitation* (2005): *NCBI*. Web. <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC554000/>.

[8] Fraden, Jacob. *Handbook of Modern Sensors: Physics, Designs, and Applications*. 3rd ed. New York: Springer, 2004. Print

[9] Mattmann, C., Amft, O., Harms, H., Troster, G., Clemens, F. "Recognizing Upper Body Postures using Textile Strain Sensors," *Wearable Computers, 2007 11th IEEE International Symposium on* , vol., no., pp.29,36, 11-13 Oct. 2007 <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4373773>.

[10] Merritt, C., T. Nagle, and E. Grant. "Textile-Based Capacitive Sensors for Respiration Monitoring." *IEEE Sensors Journal* 9.1 (2009): 71-78. *IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4711330&tag=1>.

[11] Meyer, J., P. Lukowicz, and G. Troster. "Textile Pressure Sensor for Muscle Activity and Motion Detection." *Wearable Computers, 2006 10th IEEE International*

Symposium on (2006): 69-72. *IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4067729>.

[12] Microchip. "MCP6041/2/3/4, 600 NA, Rail-to-Rail Input/Output Op Amps."*MCP6041 Datasheet*. Microchip, 3 Apr. 2013. Web. <http://ww1.microchip.com/downloads/en/DeviceDoc/21669c.pdf>.

[13] Mitchell, E. et al. "Breathing Feedback System with Wearable Textile Sensors." *Body Sensor Networks (BSN), 2010 International Conference on* (2010): 56-61.*IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5504719&tag=1>.

[14] Pacelli, M., L. Caldani, and R. Paradiso. "Textile Piezoresistive Sensors for Biomechanical Variables Monitoring." *35th Annual Internation Conference of the IEEE Engineering in Medicine and Biology Society* 1 (2006): 5358-361. *NCBI*. Web. <http://www.ncbi.nlm.nih.gov/pubmed/17946696>.

[15] Papakostas, T.V., J. Lima, and M. Lowe. "A Large Area Force Sensor For Smart Skin Applications." *Sensors* 2 (2002): 1620-624. *IEEEXPLORE*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1037366>.

[16] Plusea. "Neoprene Bend Sensor IMPROVED." *Instructables.com*. Instructables.com, 23 Apr. 2009. Web. <http://www.instructables.com/id/Neoprene-Bend-Sensor-IMPROVED/>.

[17] Rekimoto, J. "SmartSkin: An Infrastructure for Freehand Manipulation On Interactive Surfaces." *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2002): 113-20. *ACM*. Web. <http://dl.acm.org/citation.cfm?id=503397>.

[18] Sergio, M. et al. "A Dynamically Reconfigurable Monolithic CMOS Pressure Sensor for Smart Fabric." *Solid-State Circuits, IEEE Journal of* 38.6 (2003): 966-75.*IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1201999>.

[19] Wang, Mingjiang. et al. "A Matress System for Human Biosignals Monitoring." *Prognostics and System Health Management (PHM), 2012 IEEE Conference on* (2012): 23-25. *IEEEXplore*. Web. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6228967>.

[20] Zhang, Hui, and Xiao-Ming Tao. "A Single-Layer Stitched Electrotextile As Flexible Pressure Mapping Sensor." *Journal of The Textile Institute* 103.11 (2012): 1151-159. Web. <http://www.tandfonline.com/doi/abs/10.1080/00405000.2012.664868>.

[21] "Force Sensing Resistor Integration Guide and Evaluation Parts Catalog." N.p., 26 Feb. 2008. Web.