PEDIATRIC DYNAMOMETER

NSF Summer Undergraduate Fellowship in Sensor Technologies
Armand O'Donnell (Electrical Engineering) – University of Pennsylvania
Advisor: Dr. Jay Zemel

ABSTRACT

Weight-bearing activity has been demonstrated to benefit childhood bone development [1]. To further investigate this connection, the Growth and Development Laboratory at the Children's Hospital of Philadelphia wishes to study the effects of strenuous activity among school-age children. Acquiring quantitative information about the magnitude and duration of force on children's feet has posed a challenge, as surveys serve only to collect qualitative information about exercise. While the equipment necessary to perform human kinetic analyses exists, devices used for directly monitoring the activities of a child can be large, awkward, and obtrusive. A small, inconspicuous, self-contained mobile device to collect and store data about the physical activity of the individual wearing it would facilitate medical research in pediatric bone health studies.

The data acquisition device proposed by Dr. Jay Zemel of the Moore School of Electrical engineering is a small, battery-powered, in-shoe physical activity dynamometer (Foot-PAD). It uses a strip of piezoelectric plastic as a force sensor along with circuitry capable of taking measurements and storing data over a period of three to fourteen days. After this period, information can be uploaded from the dynamometer to a computer, at which point the data can be analyzed or stored in an archive for later analysis.

Earlier projects established the feasibility of using a piezoelectric polyvinylidene fluoride (PVDF) sensor for the dynamometer and presented a small but non-operational design. As a result of this summer's work, the pediatric dynamometer has been programmed to sample the piezoelectric sensor, make simple calculations based on these readings, store relevant data on a small flash memory chip, and extract this information, uploading it to a personal computer after the acquisition has finished. The hardware design has also been optimized to take up as little space as possible and consume minimal power.

Table of Contents

1. INTRODUCTION

1.1. Background

The correlation of a child's exercise habits to bone density[1] has been a topic of recent interest. The Children's Hospital of Philadelphia (CHOP) is attempting to quantitatively document the kinetic activities of a child over the course of days or weeks, specifically the average and peak force on a child's foot. Measurements of force shall be taken continuously over a period of time. Surveys and questionnaires are common sources of data used medical research, but they are susceptible to sampling and reporting biases and are inherently inaccurate, particularly if data is collected from children or adolescents[2]. Due to these issues, CHOP cannot use surveys for this childhood bone health study; more reliable means are necessary.

If instrumentation is used to record childhood activities, it must be sensitive and designed for the specific application. According to an Iowa State University study[3]:

> To accurately assess children's activity patterns, an instrument must be sensitive enough to detect, code, or record sporadic and intermittent activity. Care also must be used to select criterion measures that reflect appropriate physical activity guidelines for children.

The study also concluded that in studies involving children, there is an "accuracy/ practicality" tradeoff such that more accurate devices are accordingly expensive and obtrusive. A device is needed that can accurately measure the force on a child's feet over the course of a day or longer without affecting the child's daily routine.

Portable devices have been patented for use in more general kinetic studies, mainly for calculating energy expenditure over the course of a day. These patens describe detailed methods for sensing force on the feet of an injured patient[4], the use of a piezoelectric sensor to analyze human locomotion[5] and calculation of the metabolic expenditure of energy by placing a force-sensitive resistor, or strain gage, inside of a shoe[6].

The Electrical Engineering department of the University of Pennsylvania is collaborating with CHOP to find a solution. Embedding a portable dynamometer inside the shoe has been proposed and is under development (see Preceding Work). Dr. B Zemel of CHOP suggests that it should not be noticeable or else attention will be drawn to the device, and the participant, a child, may not want to wear it. It is also necessary to store the data in a central location, preferably a personal computer, after the sampling has taken place. The device should be relatively inexpensive to fabricate so that multiple instances of the device can be used by different participants simultaneously. Once adequate data has been received by the hospital, it can be interpreted by specialists in the field of childhood bone development.

1.2 Preceding Work

The pediatric dynamometer under development at Penn is fundamentally different from what has been done in that it is a completely self-contained unit. The embedded microprocessor combined with the sensor and memory allows data to be processed and recorded autonomously and locally, not relying on an external communications link to a storage device. The nature and sheer quantity of information this device is capable of recording may enable medical researchers to address other research issues beyond monitoring children's activity and calculating their energy expenditure.

Work on this project was started in the summer of 2004 by Sunfest fellow Olivia Tsai[7]. Since then, it has progressed as a senior-year design project during the 2004-05[8] and 2005-06[9] academic years. The dynamometer has evolved from a conceptual plan to a functioning wireless data acquisition device worn attached to the leg to a two-square-inch small circuit that fits inside the heel of a young child's shoe with room to spare.

1.3 Goals of This Project

As of the start of the Sunfest project in May 2006, a hardware design for the pediatric dynamometer had been established and most of the details for its design and operation were close to implementation. The following conditions needed to be met before this device could be utilized by volunteer children:

1. The microcontroller used by this device requires a program that will oversee sensor sampling, signal processing (if necessary), documentation of results to an onboard memory chip, and serial communications with a personal computer for data retrieval and archival.
2. The existing hardware layout of the pediatric dynamometer was a generic design that would have worked, but could be streamlined to include fewer components. Furthermore, the hardware layout must be consistent with the software algorithms used in (1).
3. The pediatric dynamometer must be tested under real-world conditions inside a shoe, which should include exposing the sensor and circuits to moisture, mechanical stress, and a range of temperatures.
4. A simple software front-end was needed to easily interface the device to the communications port of a PC in a format readily accessible to the researchers' existing software.

Programming the device and finalizing its layout were the highest priorities for the 2006 Sunfest project. With a solid software foundation and a reliable hardware backbone, the pediatric dynamometer could be reconfigured and reprogrammed to record the desired data in flash memory. The optimal sampling rate could then be decided upon. An important goal was that the microcontroller's software could accommodate modified algorithms for extracting useful information from the sensor's output.

2. Building the Pediatric Dynamometer

2.1 Device Overview

In order to meet the pediatric dynamometer's size, storage capacity, and power consumption constraints, a number of design decisions have been made over the course of this project's development. The heart of the pediatric dynamometer is the PIC16F88 flash microcontroller. This device oversees the interaction of the other components, makes calculations, and performs some simple data interpretation. The polyvinylidene diflouride (PVDF) thin film sensor's output is proportional to the force on the sole of a shoe. The sensor's output is conditioned by a charge amplifier circuit connected to the analog-to-digital converter inside of the PIC. After processing, the resulting information is recorded on a 16 megabit serial EEPROM that was chosen for its extremely small size and relatively high storage capacity.

Each of the device modules and associated software are described in detail with explanations as to how each stage interacts with the others. An overview of the operation of the completed device is then presented. A number of issues arose during the project that may require alternative solutions as discussed in section 5, Recommendations for Future Work.

2.2 PIC16F88 Flash Microcontroller

Microcontrollers, the backbones of "embedded systems," have become ubiquitous over the past few decades. As processors they provide appreciable computing power and can execute millions of instructions per second. As fabrication technology has improved, the microcontroller's voltage and power requirements have decreased so that they can run for extended periods of time off of 3.3V button-cell battery power. The PIC and all of the other devices in the pediatric dynamometer are powered by a 3.3V button cell battery. Most importantly, microcontrollers offer a wealth of features, including general digital inputs and outputs, analog-to-digital converters, serial interfaces, and timers.

The PIC series of microcontrollers offers a line of units which best meet the needs of the pediatric dynamometer. The PIC16F88 was chosen because of its low power consumption, small size, and availability of features: it provides an asynchronous serial port for interfacing with a PC, and a synchronous serial port used to read from and write to a small serial EEPROM (electrically-erasable read-only memory) chip. There is a multiplexed analog-to-digital converter on this PIC, essential for sampling information from the sensor. In addition, the PIC16F88 has an internal clock that removes the need for a timing crystal, saving space on our circuit board.

PIC microcontrollers typically contain EEPROM (electrically erasable programmable read-only memory) or flash memory in which the program instructions are stored. The PIC is compatible with in-circuit serial programming, which allows users to program and debug the PIC in a prototype dynamometer. In-circuit programming simplifies the prototyping phase by eliminating the external programming of the PIC, which requires removing the PIC from the circuit.

Finally, and of significant importance, is the PIC's low cost. Since the childhood activity study would involve dozens or even hundreds of volunteers, it is desirable to create the device from inexpensive parts. If purchased directly from Microchip™, one PIC16F88 costs between two and three dollars, depending on the quantity ordered.

One of the surprises encountered when working with the PIC over the course of the design process was the fact that some of the microcontroller's features frequently caused unexpected results. The watchdog timer, which automatically resets the microcontroller after a specified number of instruction cycles, is enabled by default and must be explicitly disabled if its operation is not desired. The 16F88 employs a low-voltage brown-out detect that automatically switches the device off when the supply voltage drops below 4V, which naturally presented a problem when working with the 3.3V lithium battery voltage.

2.3 PVDF Sensor

Measuring the vertical force of a foot inside a shoe with a PVDF sensor had been established by previous groups[8, 9]. Numerous specific experiments were carried out on these sensors under a variety of circumstances.

The principle behind these sensors is piezoelectricity. PVDF is a piezoelectric thin-film polymer. When the sensor is bent, charges accumulate on the surfaces, inducing current in the external circuit. Figure 1 shows the basic operation of the PVDF sensor in response to a force in the horizontal direction.

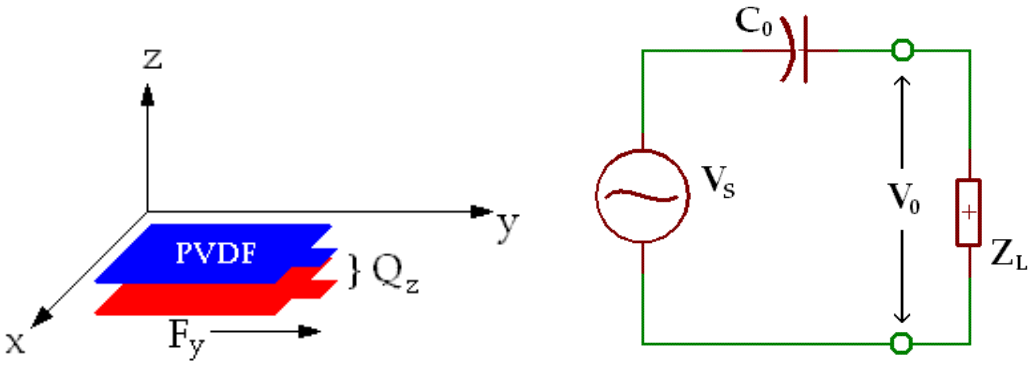


Figure 1a (left): PVDF Sensor Force-Charge Relation
1b (right) PVDF Equivalent Circuit [11]

The linear relationship governing this charge displacement is: $Q_z = d_{zy}F_y$ where Q_z is the charge displaced across the planar surface, d_{zy} is the longitudinal piezo strain coefficient, and F_y is the force along the y-axis. The equivalent circuit[11] shows a voltage source V_S in series with the sensor's capacitance C_0. The external circuit is the load, Z_L. For further reference, see [10].

In order to be used with other circuitry, the PVDF is coated with a thin layer of conducting material, typically silver, and is then laminated with a protective plastic coating. A pair of terminals is attached for connection to the external circuit. Charge induced in the external circuit during an instant of time is fundamentally expressed as the definition of current: $dQ/dt = I$. The behavior of the PVDF sensor in a circuit analysis context is as follows: any change in the deflection of the device results in an immediate current flow through the load Z_L. At any period of time when the sensor is stationary the net current through Z_L is zero.

Earlier projects done on the pediatric dynamometer investigated the PVDF sensor's response to mechanical stress. It was shown that the most satisfactory mechanical arrangement for the PVDF sensor inside a shoe was to mount it on a cantilever of spring steel[8a]. A series of calibration experiments were made by placing standard weights on top of the sensor one after the other in set increments and viewing the output of the PVDF sensor in each case. For deflection within the bounds of normal human locomotion, it was found that the amount of charge displaced was proportional to the vertical force on the sensor[8b].

The pediatric dynamometer's primary goal is to record *force*. The piezoelectric sensors' response to a change in applied force produces a temporary current so long as the force is changing, and not over the entire duration of the force. The force is obtained from the charge displaced as a function of time, not just at a single instant. Mathematically, the force is proportional to the time integration of the signal.

Integration can be done in two ways: using the microcontroller's ADC to sample the current and store digital signals in an accumulator, and analog integration with a charge amplifier. Software in the PIC provides successive sampling of the current flow in and out of the sensor. The accumulator value is then a digital representation of the total

charge displaced by the sensor. This method produced promising results, as the sample data presented both the original PVDF output alongside the integration of the data. Software integration has two drawbacks: analog device offset and sampling rate. Signal offset is added to the sensor output causing the accumulator to introduce an error each sampling cycle. Over time, the offset accumulation introduces a drift, potentially rendering the data useless. The sampling rate must also be sufficiently high or else data may be lost if the force changes suddenly.

Analog integration of the PVDF sensor signal is an alternate and widely used approach. One form of analog integrator, called a charge amplifier, was investigated this summer and was chosen for use in the pediatric dynamometer. The next section provides an in-depth explanation of the charge amplifier.

The PVDF sensor also responds to a change in temperature, the pyroelectric effect. In the dynamometer application, the low-frequency signal introduced by the pyroelectric effect is unwanted. When a test subject first puts on a shoe, the inside of the shoe (particularly the sensor inside it) will warm up from room temperature to near body temperature. This introduces a slow drift in the input signal that can be reduced by adding a charge-leakage resistor to the charge amplifier as discussed in the next section.

2.4 Charge Amplifier

The signal current from the PVDF sensor can be integrated continuously with a charge amplifier (CA) to provide a voltage proportional to the charge displaced in the PVDF sensor. The CA initially examined is composed of three stages, each controlled by an operational amplifier. The Electronics Workbench schematic for the CA is shown in Figure 2.

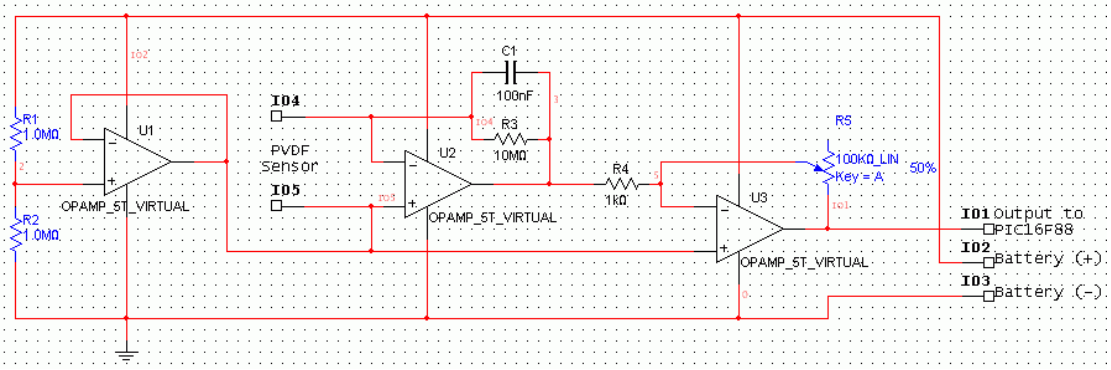


Figure 2: Charge Amplifier Prototype—Electronics Workbench™

The first stage provides a voltage reference of half the source voltage for the rest of the circuit. The PVDF sensor moves current in both directions, resulting in positive and negative voltages across its terminals. Since the entire system is powered from a single source, a button-cell battery, the op-amp's input and output range is from 0V to the source voltage. An offset is needed for negative voltages to be represented. A logical choice is the half-source reference so that an equal amplitude signal can be represented above and below the reference without saturating the amplifier (reaching the max or min voltage of the amplifier).

The second stage is the charge amplifier, which integrates the PVDF output signal connected between IO4 and IO5. Referring to the voltage at IO5 as V_{ref}, the voltage

across C1 as V_C, and the output voltage of the amplifier U2 as V_{CA}, and applying the negative feedback condition of the op-amp ($V_{IO4} = V_{IO5}$) results in the following relationship: $V_{CA} = V_{ref} + V_C$. The capacitor voltage V_C depends on the charge on it: $V_C = Q_C/C$. Differentiating with respect to time yields:

$$\frac{dV_C}{dt} = \frac{1}{C}\frac{dQ_C}{dt} = \frac{1}{C}i_C(t)$$

Integrating with respect to time yields:

$$\int \frac{dV_C}{dt}dt = \int \frac{1}{C}i_C(t)dt \Rightarrow V_C = \frac{1}{C}\int_0^{t_0} i_C(t)dt$$

where the CA is powered at time 0 (such that $V_C = 0$), and t_0 is the time of measurement. The current induced by the PVDF sensor is balanced by the op-amp, which outputs a voltage that cancels out the capacitor voltage. **The output voltage of the charge amplifier with respect to the reference voltage is proportional to the charge on the feedback capacitor.** A resistor is connected in parallel with the feedback capacitor to slowly discharge it. This attenuates the integration of very low-frequency signals, but preserves the integration of desired signals within the 1-5 Hz range.

The third stage is a simple amplifier with variable gain. The third stage was not used in the final design but helped during prototyping because it allowed the gain to be adjusted as needed.

A photograph of the prototype built to test the CA's operation is shown in Figure 3. Note that C1 and R3 can be replaced easily, as the prototype contains a 4-pin socket so we can experiment with different R-C pairs. The potentiometer, shown on the right of Figure 2 and the bottom of Figure 3, is used to adjust the gain of the output stage.

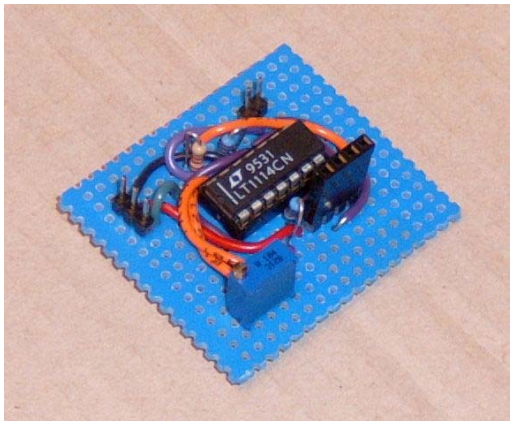
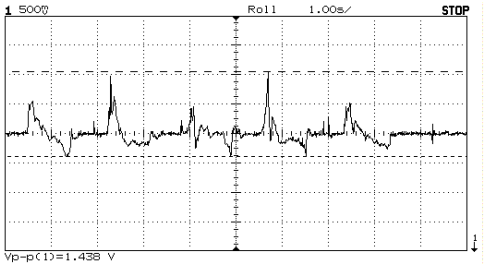


Figure 3: Charge Amplifier Prototype

The charge amplifier was tested with a PVDF sensor placed inside of the heel of a shoe. The CA does not integrate the input signal from the PVDF element appreciably for low values of R (below 10MΩ) or C (below 10nF). When R or C are low, the charge is removed from the capacitor quickly, and the integration is attenuated. Controlled, even steps were taken to test the effectiveness of the CA. The CA cannot integrate without a feedback capacitor. Even with a feedback capacitor, charge leaks off very quickly when a 1MΩ resistor is used, canceling out the integration.



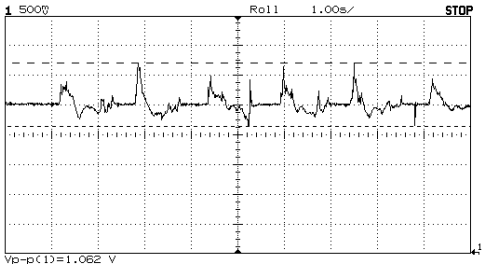


Figure 4 (left): CA using a 1MΩ resistor and no capacitor
Figure 5 (right): CA using a 1MΩ resistor and 100pF capacitor

A larger capacitor is able to retain charge for a longer amount of time. However, the amplitude of the integration is lower with large capacitors because more charge needs to be displaced to attain the same voltage increase (Q=CV).

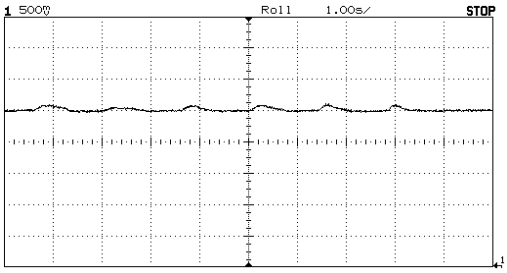
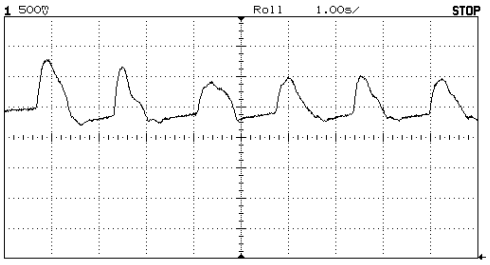


Figure 6: CA using 1MΩ resistor and 1μF capacitor

Using a 10MΩ resistor reduces the charge leak from the capacitor, but not completely. Taking short steps with the PVDF introduces and removes charge quickly enough so that the 10MΩ doesn't draw substantial charge from the capacitor. For long steps, however, there is noticeable charge leakage, and on the up-step, the counter-charge results in an offset that takes a while to restore.



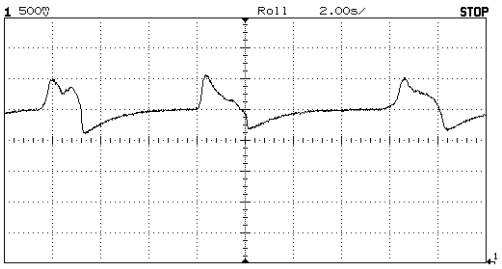
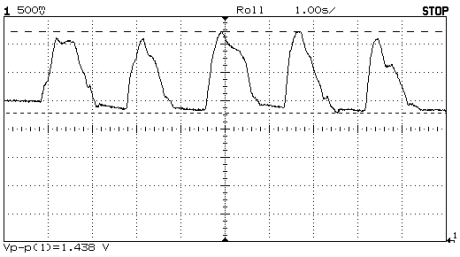


Figure 7 (left): CA using 10MΩ resistor and 100nF capacitor—short steps
Figure 8 (right): Same CA feedback network—longer steps (note 2s/div)

Omitting the resistor altogether yields surprisingly clean integration.



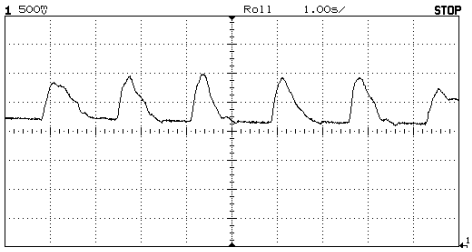
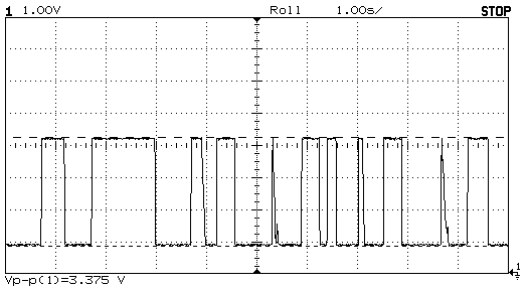


Figure 9 (left): CA using a 47nF capacitor, and without a resistor.
Figure 10 (right): CA using a 100nF capacitor, and without a resistor.

Also, care must be taken not to choose too small a capacitance. If this happens, the op-amp may drive the CA into saturation, since for a small capacitor, little charge exchange is needed for a large voltage swing.



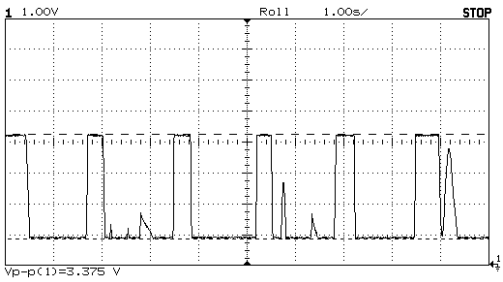


Figure 11 (left): CA using a 100pF capacitor.
Figure 12 (right): CA using a 1nF capacitor.

Another problem lies with omitting the shunt resistor across the capacitor. Although no drift due to mechanical operation of the PVDF sensor was detected, pyroelectric effects caused the CA to drift considerably from the half-source voltage, and at times even drove the CA into saturation. This effect is particularly noticeable when the user first puts on his/her shoe or takes it off, thereby introducing thermal variation as the sensor changes from room temperature to body temperature or vice versa.

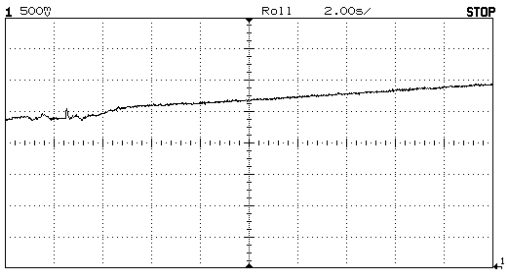


Figure 13: Removing sensor from shoe:
CA output, using no resistor and 100nF capacitor.

This effect is less problematic when a shunt resistor is connected across the capacitor. Since the pyroelectric effect is a low-frequency problem, a large RC time constant can be used to remove very low frequencies on the order of tenths of Hz. Using a 10MΩ resistor eliminates the CA's response to the pyroelectric effect.

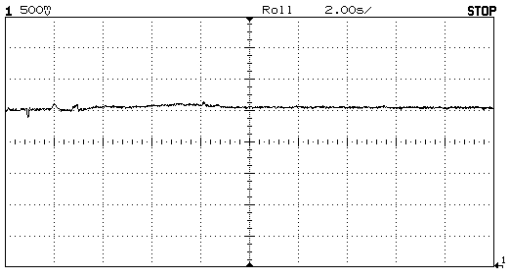


Figure 14: With a 10MΩ resistor in the feedback network, pyroelectric drift is attenuated.

Despite these low-frequency concerns, the charge amplifier was used for the final design because it simplified PIC measurements. The PIC sampled the output of the charge amplifier to obtain the force. Using a suitable high-impedance resistor, as much as 50MΩ -200MΩ, is an effective compromise. Such a high resistance only attenuates extremely low frequencies. It is also possible to use a smaller RC feedback circuit on the CA. Though this causes distortion on the signal, extracting the integration can be done in software by multiplying each previous sample by:

$$e^{-Tsamp/RC}$$

and subtracting the difference from the next sample. This particular solution has been proven in MATLAB, and will be discussed more in-depth in section 5.

Of particular interest for childhood bone density studies are measurements of the peak force, the average force over the duration of a step, and the amount of time each step takes. In the case of a low-frequency offset, as long as the CA has not been driven into saturation, the peak-to-peak values are all that is needed to extract useful information about the step. This means data resembling Figures 7 and 8 may be useful after all.

2.5 AUSART and Voltage Level Shifter (Serial Module)

The PIC microcontroller can communicate with a personal computer using compatible signals, requiring only a voltage-level shifter circuit in between the PIC and computer. The AUSART, or Addressable Universal Synchronous/Asynchronous Receiver and Transmitter, is set up in software and configures two ports on the PIC, one as a receiver (Rx) and another as a transmitter (Tx). As long as the computer and PIC are configured to receive and transmit data at the same number of bits per second, the "baud rate," no common clock or handshaking is necessary for normal two-way communications when the module is operating in asynchronous mode.

The standard for this asynchronous protocol is to hold the line at a high level until a byte is transmitted. This is often referred to as "idle high". Each byte that is sent in either direction begins with a start bit, logic "0", to notify the target device that a byte is coming. Then, at the pre-determined frequency, the bits to be transmitted appear on the line, least-significant bit first. A number of test programs were written to utilize the serial port along with other features of the device. The following example in Figure 15 demonstrates a pair of bytes, the second generated by the PIC's analog-to-digital converter (ADC), being sent out of the serial port to the computer. A variable resistor connected to provide a range of voltages was connected to one of the PIC's analog inputs. The ADC, which converts a voltage signal to a 10-bit binary number, was set to run continuously. After the analog-to-digital conversion completed, the most significant eight

bits were sent over the serial line. Having this example program readily available facilitated later use of both the serial port and ADC.

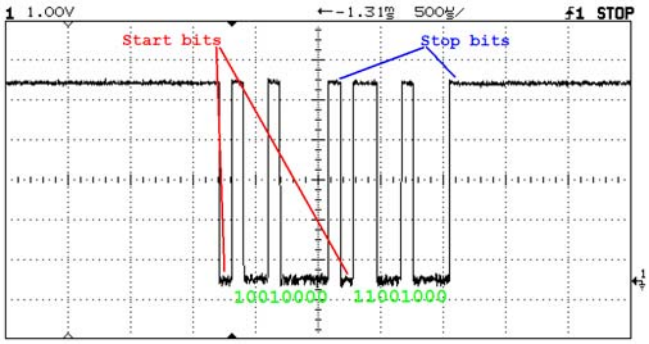


Figure 15: Sending signals 0x09, 0x13 over the serial line

Beginning with the LSB, the signal can be reversed to obtain: 00001001 00010011. In hexadecimal, these correspond to 0x09, 0x13. The analog-to-digital converter, which will be discussed in subsequent sections, is producing the value: (1 * 2 ^ 4) + (3 * 2 ^ 0) = 19 in decimal. The first bit, 0x13, is simply a byte that was used to signal the receiving program that a series of bytes has ended. For the sake of this example, only one byte was sent, the eight most significant bits of the analog-to-digital converter.

An additional piece of hardware that is required for these communications is a level shifter, which changes the signal levels from the rs232 (serial communications port) to a level the PIC uses. Since the PIC is powered by a small 3.3V lithium button-cell battery, it only has a 3.3V source to work with. As a result, the PIC represents a “0” digitally by setting a pin to 0V, its low voltage, and a “1” by with a signal around 3.3V, its highest available voltage. The rs232 communications port, however, represents a ‘1” with -12V and a “0” with +12V. These voltages are outside the ranges which the PIC can provide from its power source alone.

A small serial module was built that performed these voltage level conversions. The MAX231 integrated circuit performs a standard 0-5V to ±12V conversion with minimal external hardware. It contains a built-in inverting charge pump, so applying a +12V supply voltage to this chip allows a -12V signal to be sent from the MAX231, as well. Our PIC circuit, however, uses a 0-3.3V configuration. Therefore, a separate conversion must be made with an up-shifter, which converts signals transmitted by the PIC from 3.3V to 5V, and a down-shifter, which performs the opposite conversion for signals to be received by the PIC.

The up-shifter consists of a pair of NPN bipolar transistors each arranged in an inverting configuration. A voltage divider at the base of the first transistor halves the input voltage. As these transistors generally have a base-emitter junction voltage of about .7V at room temperature, the first transistor begins inverting when the input signal reaches around 1.4V, about half of the PIC’s 0-3.3V range. Since these are digital signals, the temperature dependence of the bipolar transistor’s gain or the saturation current of the base-emitter junction are not important. Whether the inverter switches at 1V or 2V or anywhere in-between is immaterial; it is only essential that the device shift somewhere reasonably in between 0V and 3.3V. The second transistor simply inverts the inverted signal presented by the first, preserving the binary value of the original signal. This two-

inverter configuration reproduces and amplifies these digital signals very reliably over a range of frequencies.

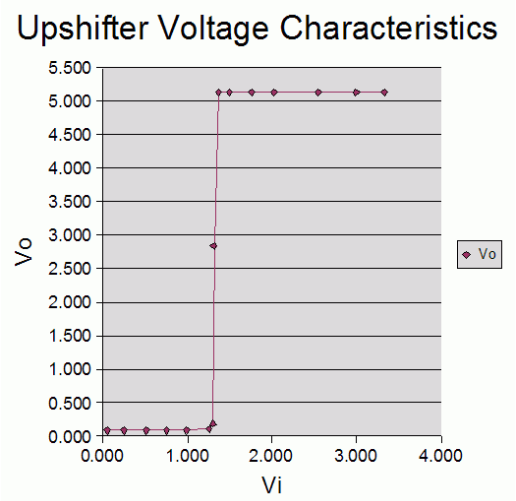


Figure 16: DC Transfer Function of Digital Level Up-shifter

Testing has shown that the PIC configuration used for the pediatric dynamometer communicates most efficiently at 38.4kBaud, or 38,400 bits per second. Start/stop bits must also be transmitted, so each byte send and received is essentially 10 bits. Using a function generator, 38.4kBaud transmission with a 3V peak is simulated by a square wave signal. Figure 17 shows the input signal on channel 2 and the output of the level shifter on channel 1.

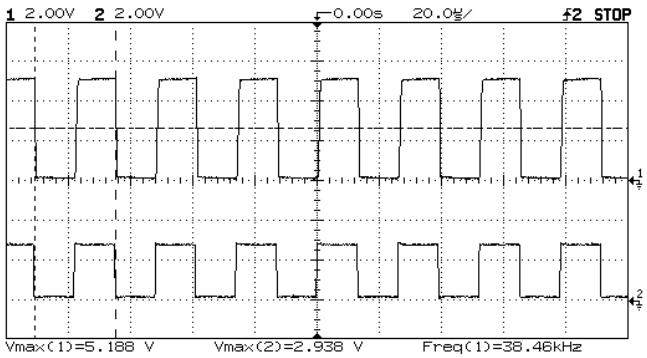


Figure 17: Digital Level Up-shifter operating at 38.4kHz

The down-shifter is even simpler, consisting of a zener diode with a 3V drop and series resistor. This fundamental design exhibits minimal distortion below 500kHz, which is considerably higher than the transmission rates the dynamometer employs.

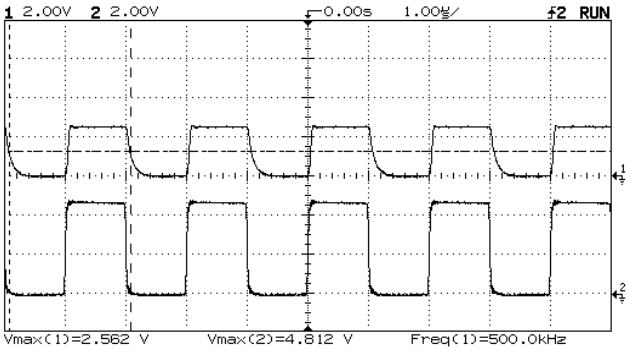


Figure 18: Digital Level Down-Shifter operating at 500kHz

Creating a permanent design for this device was facilitated by the use of Eagle™ Layout Editor software. A schematic of the level shifter's final design is documented in Figure 19.

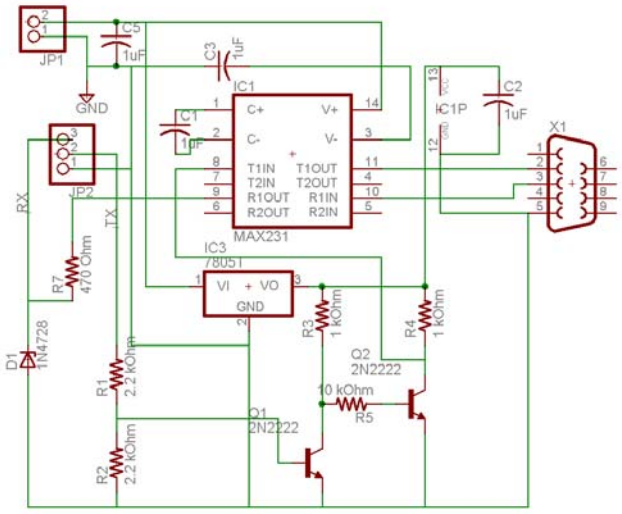


Figure 19: Schematic of Digital Level Shifter (Serial Module)

The hardware was then arranged using the same software suite. Figure 20 shows the final component layout for the serial module. It was milled on a T-Tech numerically controlled micro-mill and cut from standard copper-surfaced PCB.

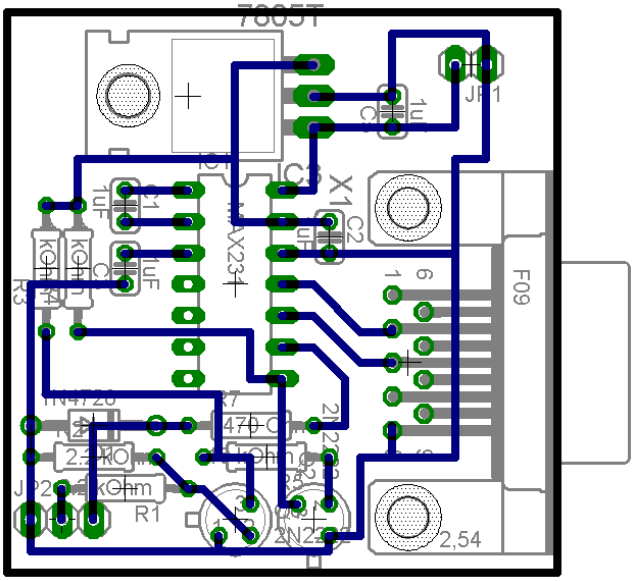


Figure 20: Layout of Digital Level Shifter (Serial Module)

This board was populated with the necessary components and its operation verified by testing the communication link between the PIC and a PC in the lab. A photo of the completed Serial Module is shown in Figure 21. The circuit board measures less than 1.8" x 1.8".

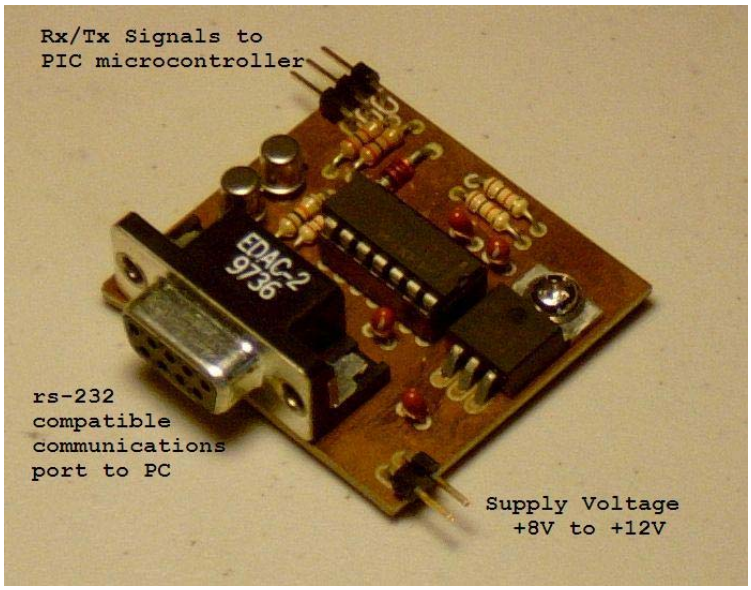


Figure 21: Completed Digital Level Shifter (Serial Module)

When the pediatric dynamometer is ready to be used for clinical research, it will be necessary to produce a number of these Serial Modules, one for each location where information will be downloaded from the dynamometer. This digital level shifter can operate with a supply ranging from 8 to 12 volts, so a standard 9V battery would be a likely candidate due to its ubiquity and low cost. Another alternative is to use a 9V or 12V DC wall adapter to provide a more reliable and maintenance-free source of power.

2.6 Synchronous Serial Communications with EEPROM/Flash

Microcontrollers frequently communicate with peripheral devices via a protocol called Serial Peripheral Interface, or SPI. SPI is similar to the serial protocol used for rs232 communications described above, except that SPI is a synchronous serial protocol (SSP). It requires a special clock signal (SCK) provided by the PIC in addition to serial data in (SDI) and serial data out (SDO). There is one additional signal called chip select (CS) which enables the particular device with which the PIC intends to communicate. This way, multiple devices can be connected to the SPI bus, and the PIC can specify which one it is communicating to by setting the appropriate chip select signals.

As an example, the 8-bit character 'W' represented by “01010111” in binary is written to an SPI device. Figure 22 shows the serial clock SCK on channel 1 of the oscilloscope and SDO on channel 2. A common standard dictates that the data is latched to the receiving device on the rising edge of the clock, when SCK switches from low to high. An important distinction must be made in the order of bits transmitted in synchronous and asynchronous modes: SSP bytes are transferred with the most significant bit (MSB) first, while AUSART communications begin with the LSB as discussed in section 2.4.

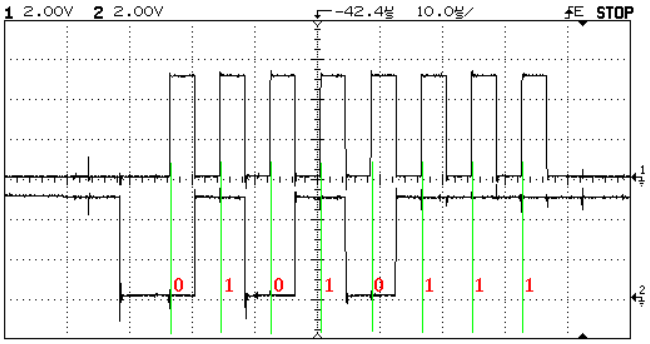


Figure 22: Transmission of 'W' via SSP

Each device connected to a microcontroller has its own dedicated chip select signal. By convention, CS is normally high when the microcontroller is not communicating with the device. Once the microcontroller sets CS low, instructions and data can be exchanged with the peripheral. Figure 23 shows the chip select signal (CS, channel 2) going low while the clock signal (SCK, channel 1) oscillates 8 times, indicating transmission a byte to a peripheral device.

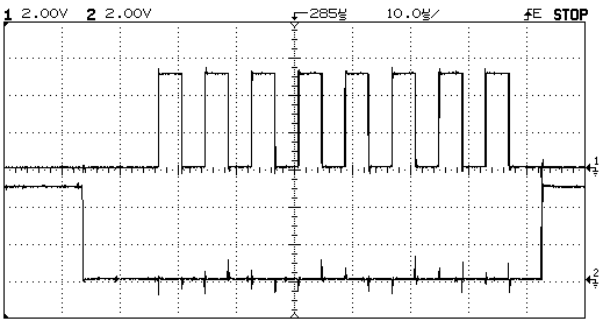


Figure 23: CS Low as SCK oscillates 8 times

The SPI protocol allowed the PIC to read and write a peripheral memory chip serially. Two devices were used for testing the SSP module in software: first, an Atmel 25256 EEPROM was used because of its higher supply voltage specification and larger package, which made it easier to connect to our prototype. The ATMEL has a 256kbit capacity. The Spansion S25FL016A serial flash chip has a 16Mbit capacity and is much smaller in size than the Atmel model, making it an attractive solution for the data storage needs of the pediatric dynamometer. Because of its small size and use of surface mount technology, though, the Spansion flash was used only in later prototypes.

The Atmel and Spansion memories first receive instructions from the microcontroller. These instructions are eight bit codes that initiate a certain operation, such as read, write, erase, and write-enable/disable. The instruction may be followed by data for appropriate instructions. Read and write instructions require a 24-bit address indicating where the data will be read from or written to. Erase and write-enable instructions do not require a specific address.

Another subtle difference between the Atmel 25256 and Spansion S25FL016A exists in their fabrication technology: the Atmel memory is EEPROM, electrically-erasable programmable read-only memory, while the Spansion device is flash memory. The Atmel device can be re-written to freely, because the internal circuitry automatically erases a block of memory when a write instruction is received. The Spansion device can only write 0's to memory blocks, not 1's. In order to re-write 1's over existing 0's, sectors of memory must first be cleared. This required writing a software routine that cleared the Spansion flash memory upon receiving instruction.

2.7 Integrating AUSART/SSP Components in Software

It is necessary for the dynamometer to be able to transfer data from the serial flash memory to the serial port of a computer. General-purpose functions were written to communicate with a device connected to the SSP bus and a computer connected to the AUSART. Additional functions enable and disable the SSP and AUSART interfaces as needed. The PIC16F88 cannot use the SSP and AUSART features simultaneously, as pin 8 of the microcontroller can act exclusively as an input Rx from the AUSART module or an output SDO to the SSP module.

Fortunately, disabling one protocol frees up the resources to enable the other. Continuously switching between SSP and AUSART is not as intuitive as it may seem, and requires the PIC to carefully manage the states of the pins and features of the protocols so that no conflicts occur. The following four functions oversee this operation: `enableAUS()`, `disableAUS()`, `enableSSP()`, and `disableSSP()`.

```
// Set up reception/transmission from serial port
void enableAUS(){
        // Set up PORTB
        PORTB.2 = 1;    // Remove artifact of SSP
        TRISB.5 = 0;    // PortB5 output for Tx
        TRISB.2 = 1;    // PortB2 input for Rx
        // Set up AUSART
        TXEN = 1;       // Enable Transmission
        CREN = 1;       // Enable Continuous Receive
        SPEN = 1;       // Enable Port
        RCIF = 0;       // Clear Serial Receive Interrupt Flag
        RCIE = 1;       // Enable Serial Receive Interrupt
}
```

Table 1: `enableAUS()` method

Before the AUSART can be used, it is necessary to configure the pins on the PIC. Since asynchronous serial communications are idle high, it is important that the PIC begins AUSART mode with the Tx line on `PORTB.2` high; otherwise, the PIC will interpret a low signal as a "start bit" as explained in 2.5. Furthermore, it is necessary to set `PORTB.5` as an output Tx to the serial module and `PORTB.2` as an input Rx from the serial module.

The last five lines enable specific features of the AUSART. Transmission and reception must be enabled before use; theoretically the PIC can use the asynchronous serial interface as only a receiver or transmitter, but the dynamometer uses two-way communication with the PC. When in communications mode, the dynamometer uses an interrupt to receive data from the PC: whatever the PIC is doing is interrupted when a byte is received on the serial line. The two modes of the pediatric dynamometer will be discussed in greater detail in the software results section, 3.2.

```
// Stop receiving/transmitting from serial port
void disableAUS(){
        while(!TRMT);  // Ensures that all
                       // transmission is complete
        PORTB.5 = 1;   // Trick rs232 into thinking
                       // there's a signal
        RCIE = 0;      // Disable Serial Receive Interrupt
        TXEN = 0;      // Disable Transmission
        CREN = 0;      // Disable Continuous Receive
        SPEN = 0;      // Disable Port
}
```

Table 2: `disableAUS()` method

Disabling the AUSART interface requires freeing up the resources needed by the SSP interface. Before this can begin, it is salutary to ensure that the serial port has finished transmitting its last byte of data. This is done by repeatedly polling the bit `TRMT` until it is set (`while(!TRMT);`). If the serial port is turned off before transmission has completed, important data may be lost.

Even after disabling the AUSART, the serial module is still connected to the PIC. The output line (Tx, `PORTB.5`) to the computer must be held high, or else the computer will interpret the low state as a start bit and receive spurious data (usually 0x00, a zero byte). This is done by setting the pin high (`PORTB.5 = 1;`) before disabling the AUSART.

In a manner similar to enabling the port, each of the port's features must be disabled or else unwanted operation may occur. Particularly, disabling serial receive interrupt (`RCIE = 0;`) will prevent the AUSART from receiving data on `PORTB.2`. Since `PORTB.2` is also used as SDO, data sent from the PIC to the serial flash memory will be received by the PIC's AUSART unless serial receive is disabled. If serial receive is not disabled, the next time AUSART is enabled, the SDO data will have been read into the AUSART receive flag, causing unpredictable results.

```
// Synchronous serial port for off-chip EEPROM
void enableSSP(){
        // Set up PORTB
        TRISB.1 = 1;    // PortB1 input for SDI
        TRISB.2 = 0;    // PortB2 output for SDO
        TRISB.4 = 0;    // PortB4 output for SCK
        // Set up SSP
        SSPEN = 1;      // Enable SSP
//      SSPIE = 1;      // Enable SSP Interrupt
}
```

Table 3: `enableSSP()` method

Enabling the SSP interface requires fewer instructions than enabling the AUSART. Again, the appropriate `PORTB` pins of the PIC need to be set to input/output for proper SPI operation. One line (`SSPEN = 1;`) enables the SSP interface. Another optional line (`SSPIE = 1;`) can be used to enable interrupts whenever an SPI byte transmission is complete. The pediatric dynamometer uses polling with the SPI interface because it is less of a priority than sampling the ADC and reading from the serial port, which must be done in real-time.

```
void disableSSP(){
        PORTB.4 = 0;    // Clock at a low level
        SSPEN = 0;              // Disable SSP
//      SSPIE = 0;              // Disable SSP Interrupt
}
```

Table 4: `disableSSP()` method

Finally, before disabling SSP interface, it is necessary to ensure that the clock SCK remains low while SSP is not in use. Disabling SSP while `PORTB.4` is set high will result in a rising edge, which initiates data reception on the peripheral device. After clearing bit `PORTB.4`, the SPI can then be disabled by clearing (`SSPEN = 0;`).

Armed with these four functions, the PIC can interface with the serial port of the computer and the serial flash memory by switching modes as needed. Table 5 shows an example of how the pediatric dynamometer sends the contents of the flash memory to the computer one byte at a time.

```
        enableSSP();
        // Begin reading at address 0
        begin_InstSSP
        sendByteSSP(READ);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        // Read each byte and write to terminal
        // until we receive a blank byte
        do {
                readBuf = receiveByteSSP();
                disableSSP();
                enableAUS();
                serialOut(readBuf);
                disableAUS();
                enableSSP();
        } while(readBuf != 0xFF);
        disableSSP();
        end_InstSSP
```

Table 5: Reading from the serial flash memory

SSP must first be enabled. The PIC then sets chip select low in the instruction `begin_InstSSP`. Once CS is low, the serial flash memory begins receiving instructions. The next four lines send four bytes of data to the memory: first, a read instruction is sent, followed by the twenty-two bit address 0x000000, the very beginning of the flash memory. After these four bytes are sent, the memory sends bytes as long as the clock SCK keeps oscillating and CS remains low. The line (`readBuf = receiveByteSSP();`) stores a byte sent by the memory into the buffer `readBuf`. Then, SSP is disabled and AUSART is enabled. The contents of the buffer is sent out to the serial port in the statement (`serialOut(readBuf);`). Once transmission of this byte is complete, the AUSART is disabled, SSP is enabled, and the cycle can continue. Before it does, however, the PIC checks the buffer `readBuf` to see if it is 0xFF, which in binary is 11111111. All 1's is characteristic of flash memory that has either not been written to or has been cleared. The PIC interprets this condition as the end of data stored in memory and stops reading. SSP is disabled and the CS signal is set high to indicate the end of the read instruction.

The read/write cycle was demonstrated using the HyperTerminal© program. A sequence similar to the one in Table 5 was used to write a character to the flash memory every time a character was typed to the terminal. Pressing Ctrl + R initiates the memory dump shown in Table 5.

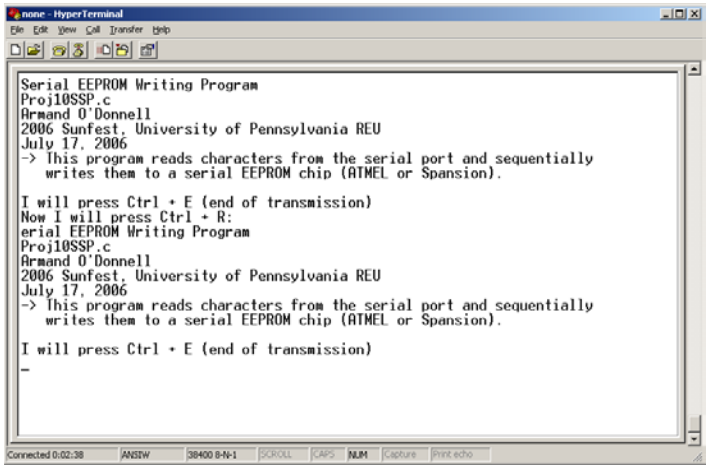


Figure 24: Screen capture of serial flash memory I/O

3. Final Results

3.1 Hardware Design Results

This summer's work resulted in a fully functional pediatric dynamometer. The final design consumed less space than any previous design. Figure 25 shows the schematic of the dynamometer in Eagle Layout Editor©.

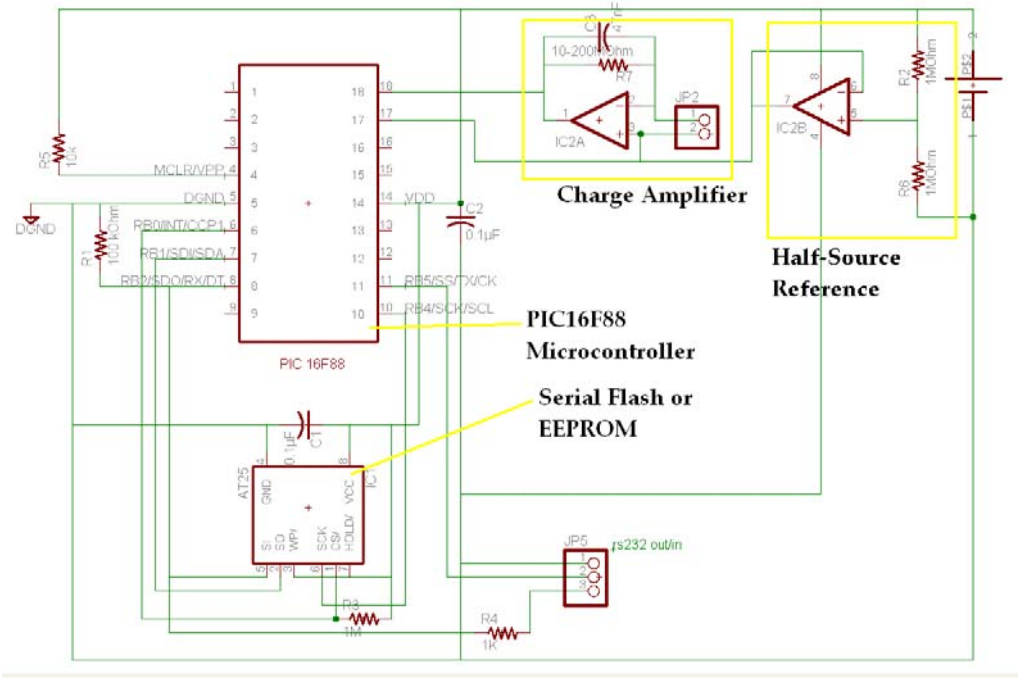


Figure 25: Schematic of Pediatric Dynamometer

A 3.3V lithium cell provides power to this circuit, shown on the upper right. The charge amplifier, shown in the top right corner of Figure 25, does not have a gain stage like the one discussed in Section 2.4. Using a 47nF capacitor in the feedback loop of the charge amplifier provided sufficient voltage swing for mechanical stress on the PVDF sensor without saturating the amplifier. A 200MΩ resistor is used in the feedback loop to minimize drift while attenuating very low frequency signals. Note that both the half-source reference and the output of the charge amplifier are connected to the PIC16F88. These are analog inputs. The PIC's ADC samples both the half-source reference and the output of the charge amplifier. A comparison is done so that if the reference voltage is not exactly 3.3V/2 = 1.65V, the PIC can still calculate the CA's output relative to the reference. How this is done in software will be discussed in more detail in section 3.2.

There are two inputs to this system; the PVDF sensor is connected to the two pins on the right side of the charge amplifier. The bottom of Figure 25 shows a three-pin connector marked “rs232 out/in” to which the serial module is connected. The 1kΩ resistor (R4) between pin 3 of this connector and pin 8 (Rx) of the PIC serves to prevent the signal from the serial module from interfering with the SDO signal from the PIC in SPI mode. When the AUSART is enabled, pin 8 of the PIC becomes a high-impedance input and the 1kΩ resistor does not attenuate the signal from the serial module. When the PIC is in SPI mode, pin 8 becomes a low-impedance output providing SDO data to the serial flash memory. The serial module still provides an idle-high signal, but the 1kΩ resistor prevents the signals from loading each other. Figure 26 shows the Rx/SDO pin 8 of the PIC on channel 1. Note how both AUSART and SSP signals are at full strength. Channel 2 shows the signal at pin 3 of the “rs232 out/in” connector. In this case, only the AUSART signal is preserved and the SPI signal is attenuated.

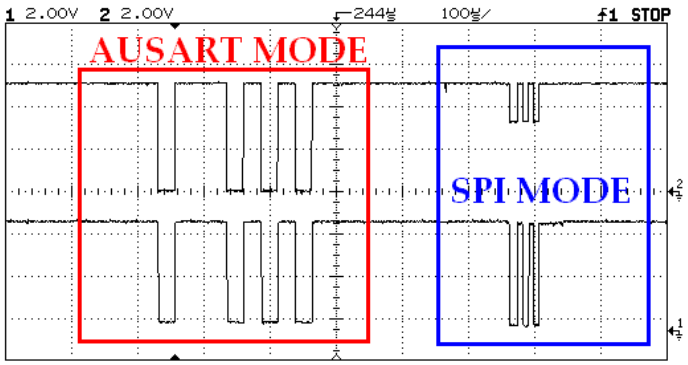
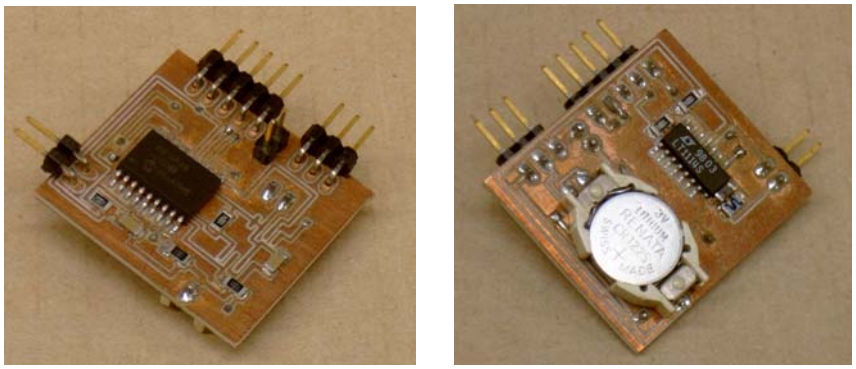


Figure 26: Comparison of SDO, Rx signals

The 100kΩ resistor between ground and pin 8 allows the PIC to detect a serial connection. When SSP is disabled and PORTB.2 is an input, the 100kΩ resistor to ground pulls this input low. When the serial module is connected, the idle-high signal from the serial module pulls high, directing the PIC to stop sampling the ADC and enter communications mode. A discussion of the two modes is presented in section 3.2.

To test the pediatric dynamometer's design, a prototype was milled on the T-Tech 5000 CNC milling machine and populated with surface-mount technology (SMT) components.


Figures 27, 28: Top and bottom of pediatric dynamometer prototype

The prototype shown in Figures 27 and 28 has some extra pins for programming and debugging the PIC and for an external power source. These photos were taken while testing the AUSART interface, when the serial flash memory had not yet been soldered on. The prototype measures 1.215” x 1.335” (3.086cm x 3.391). A photo comparing the 2004-2005 and 2005-2006 Senior Design projects with this summer's prototype is shown in Figure 29. Space was saved by avoiding use of a timing crystal, since the PIC contains its own internal RC oscillator that is satisfactory for the dynamometer.

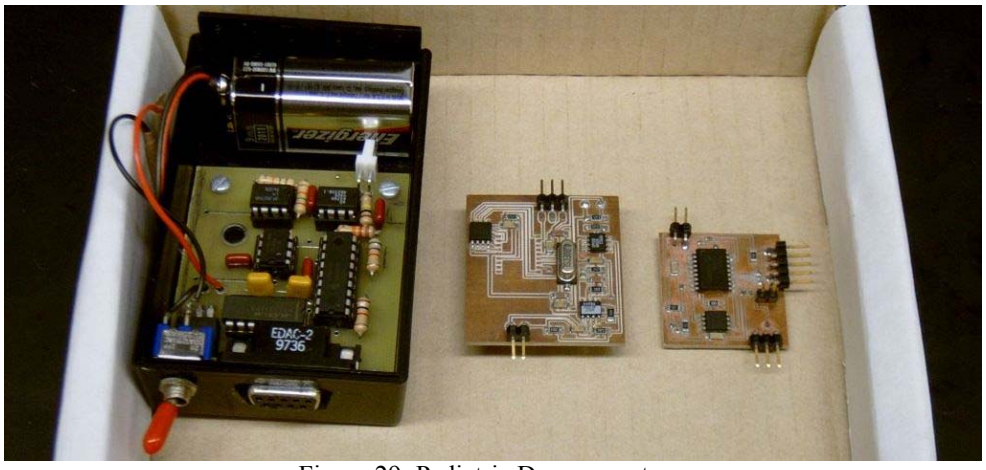
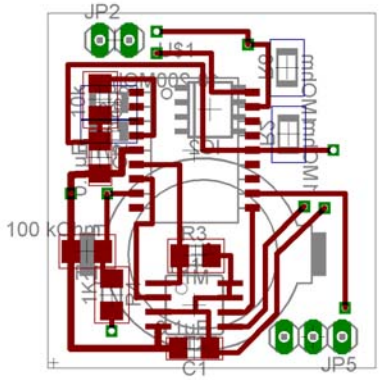


Figure 29: Pediatric Dynamometers:
(left) Senior Design 2004-2005
(center) Senior Design 2005-2006
(right) Sunfest 2006—Prototype

The digital components of the prototype function as expected. The PIC is able to read and write the Spansion serial flash memory and communicate with a personal computer via the serial module. The analog components are more sensitive to slight imperfections in fabrication and sometimes did not perform as desired in the prototype. The half-source reference circuit provided a 1.63V reference, which is satisfactory. The charge amplifier worked but some connection in the circuit introduced a parasitic resistance much lower than the desired 200MΩ. As a result, the charge amplifier suffered from decay signals similar to what is shown in Figure 8. Future options for preventing this from occurring include placing a small guard ring around the charge amplifier feedback loop and using "rub out" to remove all of the copper from the area surrounding the charge amplifier, minimizing the risk of parasitic impedances.

Removing the test and power pins from the prototype reduces the size of the board. Since only two operational amplifiers are needed, the LT1112 is more space-efficient than the LT1114 quad op-amp used by the prototype. The resulting pediatric dynamometer design could be used in the bone health studies outlined in the background. Figures 30 and 31 show the layout of the final design of the pediatric dynamometer. The board measures 1.135" x 1.150".



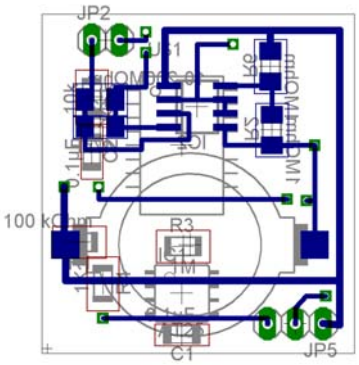


Figure 30 (left): Top layer of pediatric dynamometer
Figure 31 (right): Bottom layer. Final 2006 SUNFEST design.

The battery used in this design cannot provide sufficient power to the pediatric dynamometer for more than two days. Using a larger lithium battery like the one used by the 2005-2006 Senior Design team should allow this year's design to run for a week or more. Further work can be done to improve this year's design by incorporating a larger battery. Care must be taken to keep the design small in size.

3.2 Software Design Results

The pediatric dynamometer operates in two modes: sampling mode and communications mode. In sampling mode, the PIC consumes less power by operating at a lower frequency, 250kHz. It samples the ADC connected to the charge amplifier 200 times per second and the ADC connected to the half-source reference every 256 cycles (roughly once every 1.28s). Every sample, the PIC compares the current measurement with previous measurements and updates the appropriate data as necessary.

The PIC's goal is to determine the beginning of a step, record important data for that step, and determine the end of the step, writing important data to the flash memory. This includes: a timestamp of when the step started, the duration of time the step lasted, the average force on the PVDF sensor, and the max force over the period of the step. Table 6 is a flow chart that outlines this operation.

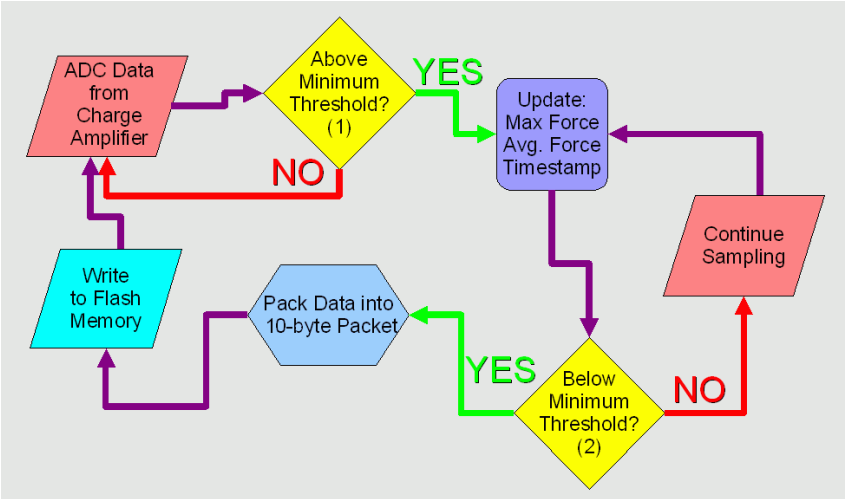


Table 6: Sampling mode algorithm

The PIC determines the beginning of a step by comparing the ADC of the CA with the ADC of the half-source reference. If the CA's value is above the reference by a set threshold, the PIC interprets it as the beginning of a step and records the timestamp. Processing of the max and accumulated force continues every sample. Once the CA's measurement is below a different threshold, the PIC detects the end of a step, and calculates the length of the step by subtracting the timestamp from the current time. The threshold for detecting the beginning of a step is higher than the threshold for the end of a step. Hysteresis prevents the PIC from detecting a small variation above the threshold as a complete step.

After every step, the PIC writes a 10-byte packet of information to the flash memory. The first three bytes encompass the timestamp, represented in 5ms increments

from the time the PIC begins sampling. This counter overflows once every 23.3 hours, almost once a day, but upon inspection of the data it should be evident where one day ends and another begins. The next two bytes contains the duration of the step in 5ms increments. The max step length is therefore 327.7 seconds. This is followed by a three-byte number that represents the accumulated sum of each ADC measurement made during the step. This number can be divided by the two-byte step length to obtain the average force. The PIC does not support division in hardware, and software division takes hundreds of cycles. Therefore, it is preferred to carry out the average force calculations on the computer once the data has been uploaded. Finally, the last two bytes contain the maximum ADC measurement over the course of the step, corresponding to the maximum force.

Once a serial connection is detected, the PIC immediately stops sampling the ADC, and switches to 8MHz operation. At this speed, the PIC can communicate with a PC's serial port at 34.8kbaud with perfect data fidelity. In communications mode the PIC can receive the following instructions: dump the raw contents of the flash memory (Ctrl + R), begin a hex dump of the flash memory (Ctrl + X), clear the flash memory (Ctrl + E), and begin sampling (Ctrl + S). Using HyperTerminal©, each of these operations was tested.

If the user enters Ctrl + S, the PIC slows down to 250kHz, and waits for the user to detach the serial module. Once this happens, sampling proceeds until the cable is connected again. If the user wishes to erase the flash memory and enters Ctrl + E, the PIC first prompts the user for confirmation, and only proceeds if the user presses 'y'.

By the end of the summer, the software interface between the PIC and the computer worked perfectly. The PIC provides a raw data or hex dump of the contents of the flash memory upon request. A calibration mode was added to test the operation of the algorithm in Table 6. Pressing Ctrl + C at the terminal begins real-time sampling of the ADC. Measurements of the half-source reference and charge amplifier are continually updated on the screen along with the accumulated force for the current step and max force. An 'S' appears at the end of the data whenever the PIC detects the beginning of a step, and disappears at the end of the step. Connecting a small PVDF sensor to the board in Figures 27 and 28 and gently deflecting it allowed testing of both the hardware and the algorithm when in calibration mode.

One aspect of the software that needs more work is writing to the flash memory in sampling mode. The flash memory can be written to using other programs that run in 8MHz mode, but for some unknown reason, the flash memory does not receive data when the PIC is in sampling mode. Inspection of the CS, SCK, and SDO signals with an oscilloscope verify that the PIC does operate correctly by writing to the flash memory at the end of a step; however, the flash memory is empty or contains old data whenever the PIC tries to read it after exiting sampling mode. Future work will require isolating this software problem and fixing it so that data acquired in sampling mode can be recorded on the flash memory.

4. Conclusions

As a result of this summer's work, a functioning pediatric dynamometer was developed. The digital end of the hardware prototype functioned as desired. The basis for communicating among the PIC, a personal computer, and flash memory has been established. The current configuration has enough storage capacity to record 200,000

steps, enough for weeks or even months of measurements. Furthermore, the current digital subsystem can be used for other types of studies, since the hardware for recording data, storing it, and retrieving it does not depend on the type of data being recorded.

The analog circuitry has been reconsidered this summer. Adding the analog charge amplifier allowed us to remove the signal conditioning circuits used by the two previous senior design teams—saving the space of an 8-pin instrumentation amplifier. The CA also served as an alternative to software integration. It made prototyping easier, since connecting an oscilloscope probe to the output of the CA displayed a signal proportional to the force on a scope; software integration required the integral to be sent over the serial port into a computer and displayed in software, a more time consuming proposition. The CA introduced its own set of problems, not the least of which was trying to attenuate low-frequency signals like the pyroelectric effect while keeping the RC time constant high enough not to leak too much charge off of the capacitor. The CA built on our final prototype operated with unexpectedly high parasitic charge loss, which was not a problem in the large CA built on a separate board discussed in Section 2.4. Ideally, a controlled experiment should compare the results of software integration and the charge amplifier.

Another issue to take into consideration is that when the charge amplifier contains a resistor in the feedback loop, charge will be drawn off of all signals, including higher frequency signals. This effect is less immediately evident with higher frequency signals. As time goes on, the charge amplifier will settle to a state where the average signal is the half-source reference. This means that after minutes of walking, steps will no longer introduce a signal beginning at the half-source reference; the signal will settle to a level below the half-source reference such that if the signal is plotted in time, the area above the half-source reference will equal the area below it. This isn't necessarily a problem, except for the fact that the current algorithm measures all signals with respect to the half-source reference. Further work should include sampling the minimum force and taking the average with respect to *it*.

Much was learned over the course of this summer, as the project required reading numerous datasheets for specifications and instructions, milling circuit boards for prototypes, and programming the PIC microcontroller in both C and assembly. The PIC and Spansion flash memory often acted unexpectedly, but consulting the datasheet frequently clarified the issue. Using the T-Tech 5000 CNC milling machine became easier with practice; fortunately, since even slight changes in the design required building new prototypes, there were plenty of opportunities. Circuit layout in Eagle was another skill that was developed over time, as the software takes a bit of getting used to.

The pediatric dynamometer is ready to enter the final stages of design and testing. Once the last few software bugs are ironed out, it will be ready for testing in the field. Controlled experiments can be done to acquire calibration data as needed. A variety of batteries can be tested with the dynamometer, allowing designers to make an informed decision on the best overall cell in terms of cost, size, and capacity. Once these hurdles are overcome, the pediatric dynamometer will be ready for mass production and use in pediatric bone health studies.

5. Acknowledgments

This pediatric dynamometer would not have been possible if it were not for the perseverance of Dr. Jay Zemel in continuing the project. Many thanks for your advice,

not only about electronics and design, but about engineering and education in general. Thank you Dr. Van der Spiegel, for organizing Sunfest and our summer accommodations. Sid Deliwala, for being available to dispense the necessary parts—on such short notice—and for your help with lab equipment and PIC references. Finally, Dr. Haim Bau and Dr. Howard Hu deserve a special thank you for allowing us to use our noisy T-Tech 5000 in their lab.

6. References

[1] Mackelvie KJ, McKay HA, Khan KM, Crocker PR. *A school-based exercise intervention augments bone mineral accrual in early pubertal girls.* J Pediatr. 2001 Oct; pp. 139(4):501-8.

[2] Elgar F., Roberts C., Tudor-Smith C., Moore L.; *Validity of self-reported height and weight and predictors of bias in adolescents*. J Adolesc Health. 37 (5): pp 371-5. Nov 2005.

[3] Welk G., Corbin C., Dale D.; *Measurement issues in the assessment of physical activity in children*. Res Q Exer Sport, 71 (2nd Suppl): S 59-73. Jun 2000.

[4] United States Patent #5,269,081: Gray, *Force Monitoring Shoe*; 1993.

[5] United States Patent #4,814,661: Ratzlaff et al., *Systems for measurement and analysis of forces exerted during human locomotion*; 1989.

[6] United States Patent #5,925,001: Hoyt et al., *Foot Contact Sensor System*; 1999.

[7] Tsai O., *Dynamometer – The New Activity Monitor*, Report no. TR-ST01NOV04, Center for Sensor Technologies, Univ. of Pennsylvania, pp. 199-221, 2004.

[8] Unpublished Senior Design Report: Lamptey L., Sin J., Wong M.; *Pediatric Step Monitor for Bone Health Studies*. Univ. of Pennsylvania, (a) 7-8, (b) 39-43. 2005.

[9] Unpublished Senior Design Report: Kam M., Tran D.; *Flexible Pediatric Dynamometer for Bone Health Studies*. Univ. of Pennsylvania, pp. 6-7, 2006.

[10] Bauer S., BauerGogonea S., Dansachmüller M., Dennler G., Graz I. Kaltenbrunner M., Keplinger C., Reiss H., Sariciftci N.S., Singh T.B., Schwödiauer R.; *Piezoelectric Polymers,* Mater. Res. Soc. Symp. Proc. Vol. 889, pp. 1-4. 2006.

[11] Measurement Specialties, Norristown, PA. *Piezo Film Sensors: Technical Manual*. April 1999.

Appendix A. Recommendations for Future Work
A.1 Digital Signal Regression

After using a "charge amplifier" to determine the amount of deflection done on the PVDF sensor, it became clear that two conflicting goals arose from the RC feedback loop. A small time constant ensured that the system would not drift away from the 1.65V reference, yet a substantial amount of charge bled off of the capacitor very quickly, canceling out the integration. On the other hand, a large time constant led to fewer losses during integration, but whenever the PVDF sensor returned to its original position, the signal took a very long time to return to the 1.65V reference.

This situation can be modeled in Matlab, fabricating a few simple mathematical examples that are theoretically accurate. Figure A1 shows a pair of unit impulses, one with amplitude +1 and the other with amplitude -1, 50 samples apart. In this case, the PIC will take 200 samples/second, so 50 samples corresponds to a quarter-second. The second graph shows the signal after integration.

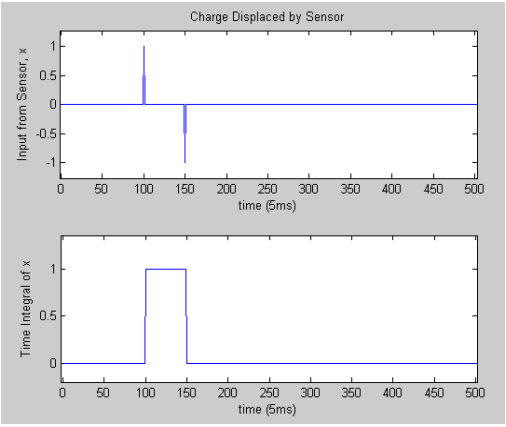


Figure A1: Integral of a "unit impulse"

Taking into account the exponential decay of the charge stored in the capacitor, the exponential function $e^{-t/RC}$ is loaded into an array, where the time constant is .796. Matlab's convolution function simulates the integration with this exponential decay taken into account. Integrating with respect to time in the continuous domain is essentially the same as taking the convolution of the "unit step" function in discrete domain with respect to samples.

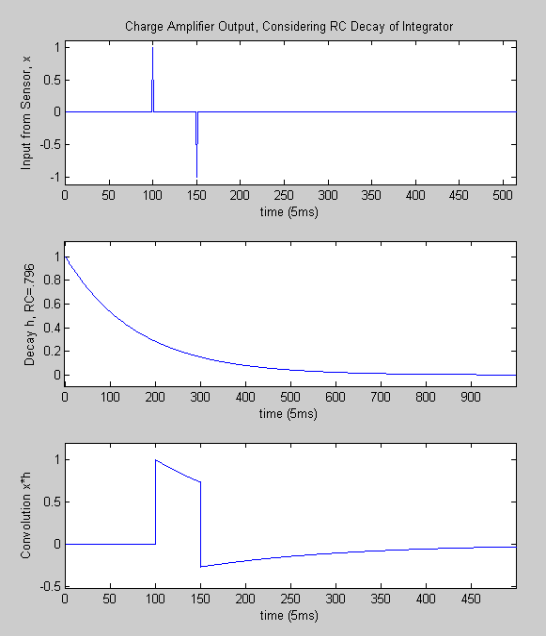


Figure A2: The RC time constant degrades charge

The RC parasitic effect degrades the integration from what it should be, as shown in Figure A1. In particular, right after the first impulse, charge is lost from the capacitor and after the second pulse; it takes very long for the signal to return to zero.

Two unit impulses are hardly a general case, so a more continuous series of inputs is considered in Figure A3. The next example shows how the system responds to a pair of triangular current pulses close together.

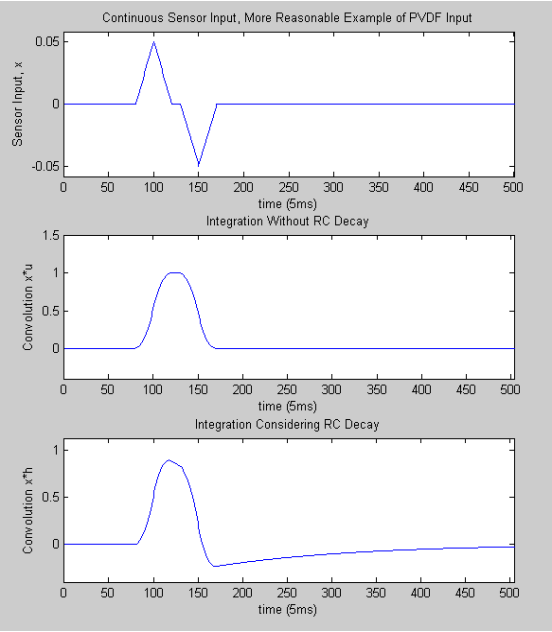


Figure A3: Triangular impulse

The analog signal is not that distorted in this case. However, the rising and falling pulses occur very closely together. This might be characteristic of a quick step, but there is no guarantee that all steps will be like this. Longer steps will have the rising and falling impulses further apart, and as a result the feedback loop will have more time to lose charge.

In this case, the degradation of the data is more severe. It will be impossible to directly apply this sampled data to the pediatric dynamometer.

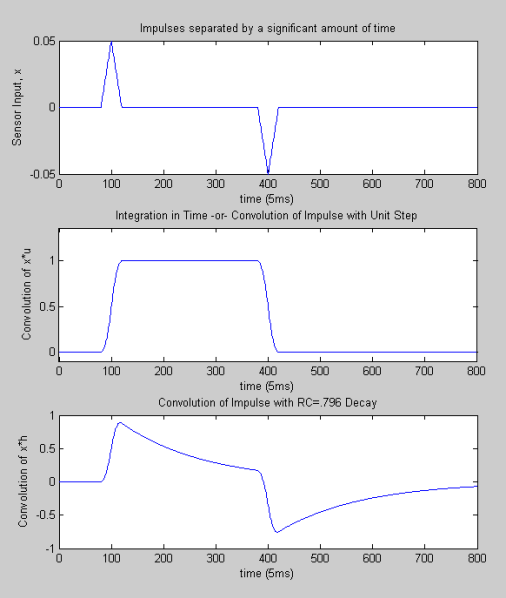


Figure A4: Separated impulses

The effects of convolution can be reversed using a function called deconvolution, in the Matlab function *deconv*. It is possible to reconstruct the original function from the convoluted result if the response of the system is known:

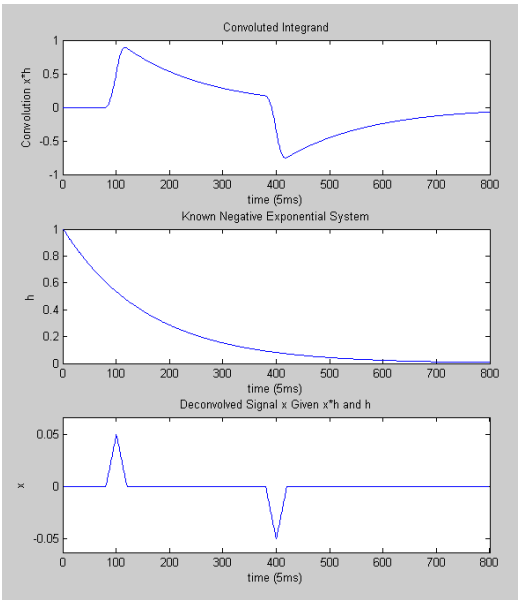


Figure A5: Reconstruction of original data

Unfortunately, there are two problems with this solution: deconvolution requires a fair amount of processing power, and would require the PIC to do a lot of computation, if real-time deconvolution were even possible. The other problem, more importantly, is that the deconvolution of the analog integration is completely useless to us, as it contains nothing more than the data from the sensor; we might as well just read that in directly.

Fortunately, a small algorithm written for Matlab takes the distorted integrand and performs a regression on it, expecting the decay due to the RC time constant.

```
%   Armand O'Donnell
%   June 2, 2006
%   SUNFEST, Pediatric Dynamometer
%   DeconvCA, an algorithm to perform data regression on a signal
%   whose integral has undergone exponential decay.

%   RC Time constant, dependent on amplifier feedback loop
RC = .796; % for Rf = 700kOhm, Cf = 1uF

reconInt(1) = 0;

for iCnt = 2:1000;
expNext = y(iCnt-1)*exp(-.005/RC);
if y(iCnt) < (expNext - .001) | y(iCnt) > (expNext + .001)
reconInt(iCnt) = reconInt(iCnt - 1) + (y(iCnt)-expNext);
else
reconInt(iCnt) = reconInt(iCnt - 1);
end
end

clear expNext;
```

Table A1: DeconvCA.m

A few advantages of this code are as follows: first of all, although this example calculates the next expected input from the value of $e^{-.005/RC}$, one could easily arrange the code so that the program never needs to calculate the exponent. For each iteration, a constant $e^{-.005/RC}$ will be multiplied to the value calculated in the former iteration. If there is a significant discrepancy, it must be due to a signal from the PVDF sensor; so the integral is adjusted accordingly. One can easily change the RC time constant by altering the first line of code. Although the program uses intervals of .005 seconds to correlate with the 200 samples/second, the rate could also be changed with minimal code alteration.

This program was run on the distorted integral on the bottom of Figure A4 and is compared to the theoretical (fully digital) integration. As far as I am concerned, the regression produces a result that is strikingly similar to what we are looking for:

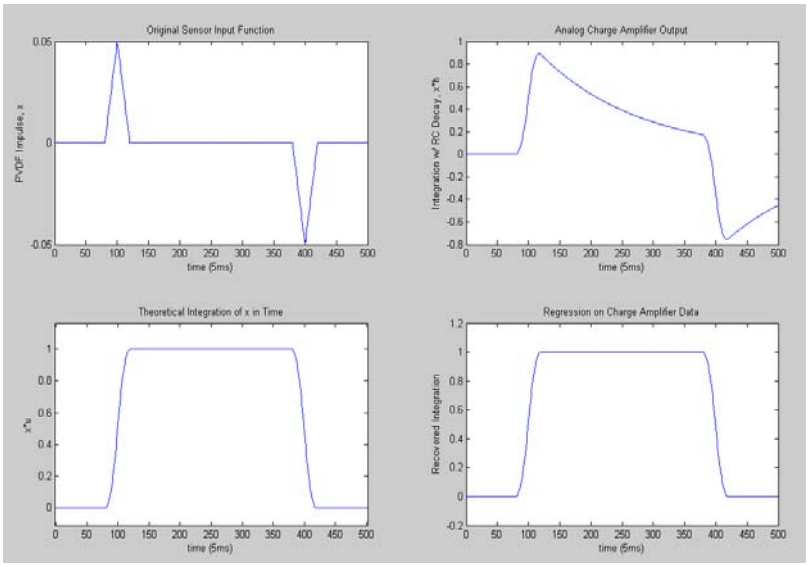


Figure A6: Regression from distorted integral to reconstructed integral.

The next step is to figure out the effect of an incorrect RC time constant. Since each iteration multiplies the former by the exponent, the algorithm is somewhat sensitive to errors in the exponent. This produces an offset so that the error "corrects" itself by compensating either above or below center in the steady state.

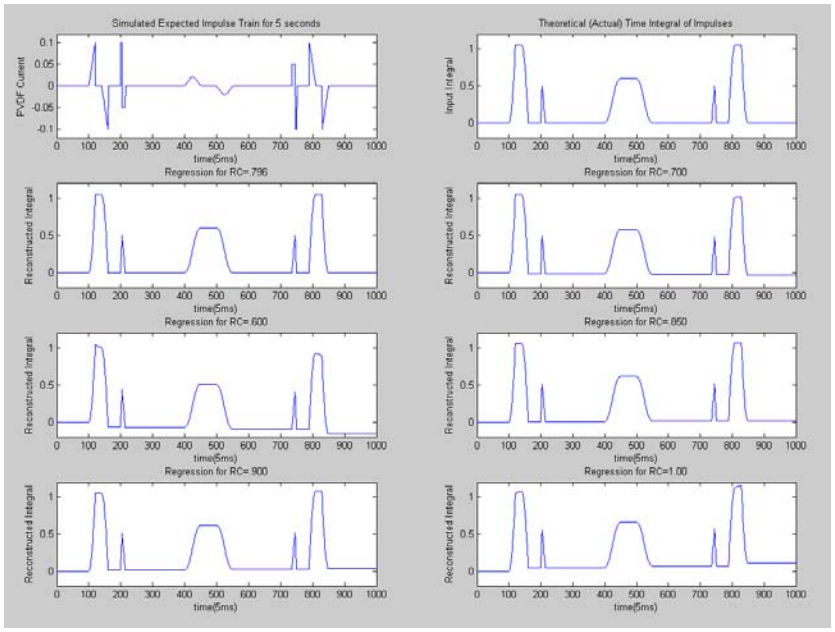


Figure A7: System response with incorrect RC time constants

Figure A7 shows that the signal only produces significant offset for RC time constants more than 10% outside of the true value. Otherwise, the small offset produced is negligible. Components within 5% of the rated value are readily available, in which case the RC constant can be measured experimentally and assumed to be the same for all devices made. Furthermore, it should be possible to write a module in the software that calibrates the time constant to match that of the analog charge amplifier device.

A.2 PC Software Component

The PIC's communication with the PC has been proven with a common serial terminal program. This is fine in the lab, but medical researchers will desire more user-friendly software. Furthermore, the terminal program is incapable of storing data in a file for processing or later use. An appropriate software interface must be written to accomplish these tasks.

Perhaps a Java graphical program would be a practical solution. Java is preferred because it is compatible across platforms. A graphical user interface can feature function buttons such as “read flash”, “erase flash”, and “begin sampling”. Pressing one of these buttons would send a signal to the PIC and initiate the corresponding function.

The Java program would also be able to capture the raw data dump from the PIC, process the data into decimal values, and store the measurements in a standard format. For example, the comma-separated values (*.csv) format can be used by a number of spreadsheet programs, and a little formatting would allow scientific software such as Matlab to do statistical analysis on the data presented.

Appendix B. Source Code listing

```
/*
        Armand O'Donnell
        July 25, 2006
        Sunfest-University of Pennylvania
        Proj12ADCStore.c

        This software can be used on a fully functional pediatric
        dynamometer. The PIC samples the ADC 200 times per second,
        and can figure out when a step begins and ends. When a
        step begins, a timestamp is set. A summation of the force
        begins, so that the average force can be calculated. The
        max force is also documented here.

        At the end of a step, the statistics for that particular
        step are saved to the EEPROM.

        When a serial module is connected to the unit, the clock
        frequency is increased to 8MHz and the device enters
        serial communications mode. In this setting, the PIC can
        communicate with a personal computer. The following
        control codes bring about the corresponding functions:

        Ctrl + R: Raw dump of EEPROM data. This is the fastest
        method of reading the pediatric dynamometer, but the
        data is unreadable on a terminal; instead, it should be
        read by a special program used for interfacing with the
        dynamometer. Ideally, it could create a list of
        measurements stored in a comma-separated values file (.csv).
        This can be    read into an Excel spreadsheet or Matlab
        program for proper data analysis.

        Ctrl + X: Hex dump of EEPROM data. The read information
        will be displayed on a terminal screen, one line of data
        corresponding to a step in the following format:
        BBBBBB LLLL AAAAAA MMMM
        B - Beginning timestamp in 5ms units
        L - Length of step in 5ms increments
        A - accumulated force over duration of step
        M - max force measurement during step.

        Ctrl + S: Sample mode. PIC will switch to 250kHz mode, wait
        for serial module to be removed, then it will start
        sampling ADC continuously, in data acquisition mode

        Ctrl + E: Erases entire EEPROM. PIC will poll EEPROM's
        status register once a second until EEPROM signals that
        erasure is complete. This key combination also clears a
        number of variables, like the master timer, so that the
        data acquisition can begin from scratch.

        Uses a PIC16F88, a Spansion S25FL016A EEPROM or Atmel
        25256 Serial EEPROM, and a properly assembled charge
        amplifier.
*/

// Interrupt handler library
#include "int16CXX.h"

// Spansion S25FL016A Serial Memory Programming Codes
#define WREN    0x06            // Write enable
#define WRDI    0x04            // Write disable
#define READ    0x03            // Read data bytes
#define SE      0xD8            // Sector erase
#define BE      0xC7            // Bulk (chip) erase
#define RDSR    0x05            // Read Status Register
#define WRSR    0x01            // Write to Status Register
#define PP      0x02            // Page program

// Chip select pin, drive low to use memory
```

```
#pragma bit     CS @ PORTB.0

// Chip select low to initiate instruction to EEPROM
#define begin_InstSSP  CS = 0;\
                                                nop();\
                                                nop();\
                                                nop();

// Notifies EEPROM of end of instruction
#define end_InstSSP           nop();\
                                                nop();\
                                                nop();\
                                                CS = 1;

bit     sample_mode_flag;      // 0 for serial comms mode, 1 for sampling mode
bit     write_ready;   // Signals that a step is ready to be written to EEPROM
bit     insideStepFlag;        // Currently sampling a step or not?
long    ADCData;       // 16-bit variable containing 10-bit result of last ADC
uns24   masterTimer;   // Will oversee timestamp implementation, incremented every 5ms
uns24   stepStartTime; // Timestamp contents at beginning of current step
long    calData;       // Digital representation of half-source reference
long    relSample;     // ADC measurement of Charge Amplifier relative to calData
long    forceMax;      // Max relative ADC measurement for current step
uns24   forceAccum;    // Summation of CA measurements, used to calculate average on PC

// Temporary variables for writing to EEPROM
uns16   wr_stepLength;         // Number of 5ms cycles for previous step
uns24   wr_stepStartTime;
uns24   wr_forceAccum;
long    wr_forceMax;

/*
        ADC units (about 3mv) of high/low threshold for signal to be considered a step
        Uses hysterisis: if ADC signal is STEPHIGHTH above calData, begin measuring inside
        of a step. Continue until signal is STEPLOWTH above calData.
*/
#define STEPLOWTH      10
#define STEPHIGHTH     20


// Pre-define serial I/O functions
char serialIn;
void serialOut(uns8 txByte);
void enableAUS();
void disableAUS();
void enableSSP();
void disableSSP();
void sendByteSSP(char wrData);
char receiveByteSSP();

// Hexadecimal formatting functions for hex dump
char hiHex(char binData);
char lowHex(char binData);

#pragma origin 4
interrupt serverX(){
        int_save_registers;

        // Serial Port Receive Interrupt
        if(RCIF){
                // Receive input byte
                serialIn=RCREG;
                RCIF = 0;       // Clear USART Receive Flag
        }

        // Analog-to-Digital Conversion Completion Interrupt
        if(ADIF && ADIE){

                ADIF = 0;       // Clear ADC flag

                // Read from ADC
```

```
        ADCData.low8 = ADRESL;
        ADCData.high8 = ADRESH;

        // ADCON0.3 -> AN1 selected, sample is from Charge amplifier
        if(ADCON0.3){
                // Calculate ADC relative to calibration
                relSample = ADCData - calData;

                // If not yet inside of a step and threshold is reached
                if(!insideStepFlag && (relSample > STEPHIGHTH)){
                        // Initialize accumulators for new step
                        insideStepFlag = 1;
                        stepStartTime = masterTimer;
                        forceAccum = 0;
                        forceMax = 0;
                }

                // If currently inside of a step:
                if(insideStepFlag){
                        // update accumulator with current force
                        forceAccum += relSample;
                        // test upper bound
                        if(relSample > forceMax)
                                forceMax = relSample;
                        // end of step?
                        if(relSample < STEPLOWTH){
                                // Bail if still writing to EEPROM
                                if(!write_ready){
                                        insideStepFlag = 0;
                                        // Store useful info into temp variables
                                        wr_stepStartTime = stepStartTime;
                                        wr_stepLength = masterTimer - stepStartTime;
                                        wr_forceAccum = forceAccum;
                                        wr_forceMax = forceMax;
                                        // Signal EEPROM write
                                        write_ready = 1;
                                }
                        }
                }

                // Every 256 cycles, sample half-source reference
                if(masterTimer.low8 == 0){
                        ADCON0.3 = 0;  // Switch to AN0 for ADC
                        GO = 1; // Begin ADC for AN0
                }
        }

        // !ADCON0.3 -> AN0 selected, sample is half-source voltage
        else{
                // Update calibration data
                calData = ADCData;
                ADCON0.3 = 1;  // Next sample from AN1, Charge Amplifier
        }
}

// Timer1 overflow interrupt
if(TMR1IF && TMR1IE){

        // Next int in 5ms (312 cycles for 250 kHz)
        TMR1L = 0xC8;
        TMR1H = 0xFE;

        // Increment timestamp
        masterTimer++;

        // Begin an A/D conversion
        GO = 1;

        TMR1IF = 0;
}
```

```
        int_restore_registers;
}


// Send to rs232 chip (MAX231)
void serialOut(uns8 txByte){
        while(!TXIF);           // Check to see if buffer is Empty
        TXREG = txByte;         // Transmit data from ADC to AUSART
}

// Set up reception/transmission from serial port
void enableAUS(){
        // Set up PORTB
        PORTB.2 = 1;            // Remove artifact of SSP
        TRISB.5 = 0;            // PortB5 output for Tx
        TRISB.2 = 1;            // PortB2 input for Rx
        // Set up AUSART
        TXEN = 1;               // Enable Transmission
        CREN = 1;               // Enable Continuous Receive
        SPEN = 1;               // Enable Port
        RCIF = 0;               // Clear Serial Receive Interrupt Flag
        RCIE = 1;               // Enable Serial Receive Interrupt
}

// Stop receiving/transmitting from serial port
void disableAUS(){
        while(!TRMT);           // Ensures that all transmission is complete
        PORTB.5 = 1;            // Trick rs232 into thinking there's no signal
        RCIE = 0;               // Disable Serial Receive Interrupt
        TXEN = 0;               // Disable Transmission
        CREN = 0;               // Disable Continuous Receive
        SPEN = 0;               // Disable Port
}

// Synchronous serial port for off-chip EEPROM
void enableSSP(){
        // Set up PORTB
        TRISB.1 = 1;            // PortB1 input for SDI
        TRISB.2 = 0;            // PortB2 output for SDO
        TRISB.4 = 0;            // PortB4 output for SCK
        // Set up SSP
        SSPEN = 1;              // Enable SSP
//      SSPIE = 1;              // Enable SSP Interrupt
}

void disableSSP(){
        PORTB.4 = 0;            // Clock at a low level
        SSPEN = 0;              // Disable SSP
//      SSPIE = 0;              // Disable SSP Interrupt
}

void sendByteSSP(char wrData){
        SSPBUF = wrData;
        while(!BF);
}

char receiveByteSSP(){
        SSPBUF = 0x00; // Dummy value
        while(!BF);
        return SSPBUF;
}

// High 4 bits to hex character
char hiHex(char binData){

        binData >>= 4;
        binData &= 0x0F;

        if(binData < 10)
                return ('0' + binData);
        else
```

```
                return ('A' + binData - 10);
}

// Low 4 bits to hex character
char lowHex(char binData){

        binData &= 0x0F;

        if(binData < 10)
                return ('0' + binData);
        else
                return ('A' + binData - 10);
}


void main(){

        // Set up internal Oscillator
        OPTION_REG = 0x87;      // Disable PORTB Pull-ups, TRM0 256 Prescalar
//      OSCCON = 0x78;          // 8MHz Internal Clock, internal RC for system clock
        OSCCON = 0x28;          // 250kHz, internal RC for system clock

        TRISB.0 = 0;            // Chip select of serial EEPROM
        TRISB.2 = 1;            // Ground RB2 through 100kOhm resistor

        // General AUSART Settings
        BRGH = 1;                       // High baud rate setting
        CREN = 1;                       // Enable Port, enable continuous receive
//      SPBRG = 51;                     // 9.615 Kbaud for 8MHz Clock, BRGH = 1
//      SPBRG = 25;                     // 19.231 Kbaud for 8MHz Clock, BRGH = 1
//      SPBRG = 16;                     // 29.412 Kbaud for 8MHz Clock, BRGH = 1
        SPBRG = 12;                     // 38.462 Kbaud for 8MHz Clock, BRGH = 1
//      SPBRG = 8;                      // 55.556 Kbaud for 8MHz Clock, BRGH = 1

        // Set up Timer1
        T1CON = 0x0D;           // No Prescaler, enable Timer1, internal clock

        // SSP Settings
        CKE = 1;                // Clocking mode compatible with the serial EEPROM
        SSPCON = 0x00;          // Data latched on rising edge,
                                // SPI Master mode, clock = OSC/4
        TRISB.0 = 0;            // Output for Chip select

        // Set up A/D
        TRISA |= 0x03;          // RA0, RA1 are inputs
        ANSEL = 0x03;           // Enable AN0, AN1
        ADCON0 = 0x00;          // Use system clock/2, Turn A/D on, sel AN0
        ADCON1 = 0x80;          // Right-justified output
        ADON = 1;               // Prepare first A/D conversion

        // Enable Peripheral Interrupts (like serial int), Global Interrupt Enable
        INTCON = 0xC0;

        // Instantiate local variables
        uns24   addrEEPROM;     // Pointer to next byte in memory
        char readBuf = 0;       // Read buffer
        char deadStream = 0;    // Counter to detect end of EEPROM

        // Initialize global variables
        write_ready = 0;        // Do not write yet
        insideStepFlag = 0;     // Not yet inside of a step
        serialIn = '\0';        // No serial data to interpret
        sample_mode_flag = 1;

        // First int in 250ms
        TMR1L = 0xF7;
        TMR1H = 0xC2;

        TMR1IE = 1;             // Enable Timer1 overflow interrupt
        ADIE = 1;               // Enable ADC Conversion Completion Interrupt
        PEIE = 1;               // Enable peripheral interrupts
```

```
GIE = 1;              // Global interrupt enable

// Drive chip select high
end_InstSSP

// Main program loop
while(1){

        // Low-frequency sample mode
        if(sample_mode_flag){

                // Check to see if serial module is connected
                if(PORTB.2){

                        sample_mode_flag = 0;
                        // Disable Timer1 and Interrupts
                        TMR1IE = 0;
                        TMR1IF = 0;
                        T1OSCEN = 0;
                        // Disable ADC
                        ADIE = 0;
                        GO = 0;
                        ADON = 0;
                        // Clear sample mode flags
                        insideStepFlag = 0;
                        write_ready = 0;

                        // Wait for one second (250kHz clock)
                        TMR1L = 0xDC;
                        TMR1H = 0x0B;
                        TMR1IF = 0;
                        while(!TMR1IF);

                        // 8MHz Internal Clock, internal RC for system clock
                        OSCCON = 0x78;
                        // Wait for stable frequency
                        while(!OSCCON.2);

                        enableAUS();
                }

                // Write step data to EEPROM
                if(write_ready){

                        enableSSP();

                        // Write enable instruction--obligatory
                        begin_InstSSP
                        sendByteSSP(WREN);
                        end_InstSSP

                        // 64us Delay in between instructions
                        nop();
                        nop();
                        nop();
                        nop();

                        // Page program instruction
                        begin_InstSSP
                        sendByteSSP(PP);
                        // Address bytes (2 or 3)
                        // Disable next line for ATMEL, only 16-bit addresses!
                        sendByteSSP(addrEEPROM.high8);
                        sendByteSSP(addrEEPROM.mid8);
                        sendByteSSP(addrEEPROM.low8);
                        // Write timestamp to EEPROM first (3 bytes)
                        sendByteSSP(wr_stepStartTime.high8);
                        sendByteSSP(wr_stepStartTime.mid8);
                        sendByteSSP(wr_stepStartTime.low8);
                        // Write step length to EEPROM (2 bytes)
                        sendByteSSP(wr_stepLength.high8);
```

```
                sendByteSSP(wr_stepLength.low8);
                // Write force summation to EEPROM (3 bytes)
                sendByteSSP(wr_forceAccum.high8);
                sendByteSSP(wr_forceAccum.mid8);
                sendByteSSP(wr_forceAccum.low8);
                // Write max force to EEPROM last (2 bytes)
                sendByteSSP(wr_forceMax.high8);
                sendByteSSP(wr_forceMax.low8);
                end_InstSSP

                disableSSP();

                addrEEPROM += 10;      // Move forward 10 bytes in EEPROM

                PORTB.2 = 0; // Low so that serial signal can be recognized
                TRISB.2 = 1;   // Return RB2 to input for sampling
                               // for serial module connection

                write_ready = 0;       // Clear write flag
        }

}

// Serial Communication mode
else {

        // No serial input -> do nothing
        if(serialIn == '\0')
                continue;

        // Ctrl + C: Calibration mode
        if(serialIn == 3){

                serialIn = '\0';

                ADON = 1;
                ADIE = 1;

                while(!serialIn){
                        // ADC TEST PROCEDURE
                        GO = 1;
                        while(GO);

                        masterTimer++;
                        write_ready = 0;

                        // Continuously displays the following quantities:
                        // Current sample relative to calibration
                        serialOut(hiHex(relSample.high8));
                        serialOut(lowHex(relSample.high8));
                        serialOut(hiHex(relSample.low8));
                        serialOut(lowHex(relSample.low8));

                        serialOut(' ');

                        // Calibration data (should be near 0x0200)
                        serialOut(hiHex(calData.high8));
                        serialOut(lowHex(calData.high8));
                        serialOut(hiHex(calData.low8));
                        serialOut(lowHex(calData.low8));

                        serialOut(' ');

                        // Maximum force for current/last step
                        serialOut(hiHex(forceMax.high8));
                        serialOut(lowHex(forceMax.high8));
                        serialOut(hiHex(forceMax.low8));
                        serialOut(lowHex(forceMax.low8));

                        serialOut(' ');
```

```
                // Accumulator value for last step
                serialOut(hiHex(forceAccum.high8));
                serialOut(lowHex(forceAccum.high8));
                serialOut(hiHex(forceAccum.low8));
                serialOut(lowHex(forceAccum.low8));

                serialOut(' ');

                //The letter S will appear if inside a step
                if(insideStepFlag)
                        serialOut('S');

                serialOut(0x0D);
                serialOut(0x0A);

                // END ADC TEST PROCEDURE

        }
        serialIn = '\0';
}

// Ctrl + S: sample mode
if(serialIn == 19){

        serialOut(0x0D);
        serialOut(0x0A);
        serialOut('R');
        serialOut('e');
        serialOut('m');
        serialOut('o');
        serialOut('v');
        serialOut('e');
        serialOut(' ');
        serialOut('C');
        serialOut('a');
        serialOut('b');
        serialOut('l');
        serialOut('e');
        serialOut('.');
        serialOut(0x0D);
        serialOut(0x0A);

        disableAUS();

        // Turn RB2 into an input
        PORTB.2 = 0;
        TRISB.2 = 1;

        // 250kHz, internal RC for system clock
        OSCCON = 0x28;

        // Enable Timer1
        T1OSCEN = 1;

        // Wait for user to remove Serial Module
        // for one second (250kHz clock)
        while(PORTB.2);
        TMR1L = 0xDC;
        TMR1H = 0x0B;
        TMR1IF = 0;
        while(!TMR1IF);

        // Next int in 5ms (312 cycles for 250 kHz)
        TMR1L = 0xC8;
        TMR1H = 0xFE;

        // Enable Timer1 Overflow interrupt
        TMR1IF = 0;
        TMR1IE = 1;

        // Enable ADC
```

```
        ADON = 1; // ADC on
        ADIE = 1; // Enable ADC Conversion Completion interrupt

        sample_mode_flag = 1;

        serialIn = '\0';
}

// Ctrl + E? Erase entire EEPROM (Spansion only)
else if(serialIn == 5){

        // Ask for confirmation
        serialOut('S');
        serialOut('u');
        serialOut('r');
        serialOut('e');
        serialOut('?');

        while(  serialIn != 'y' &&
                serialIn != 'Y' &&
                serialIn != 'n' &&
                serialIn != 'N' ) ;

        if(     serialIn == 'n' ||
                serialIn == 'N' ) {
                serialOut(' ');
                serialOut('C');
                serialOut('a');
                serialOut('n');
                serialOut('c');
                serialOut('e');
                serialOut('l');
                serialOut('e');
                serialOut('d');
                serialOut('.');
                serialOut(0x0D);
                serialOut(0x0A);
                serialIn = '\0';
                continue; }

        GIE = 0;        // Disable all interrupts

        disableAUS();
        enableSSP();

        // Write enable instruction--obligatory
        begin_InstSSP
        sendByteSSP(WREN);
        end_InstSSP

        // 2us Delay in between instructions
        nop();
        nop();
        nop();
        nop();

        // Write status register, to write-enable all sectors
        begin_InstSSP
        sendByteSSP(WRSR);
        sendByteSSP(0x00);      // Free all sectors of flash
        end_InstSSP

        // 2us Delay in between instructions
        nop();
        nop();
        nop();
        nop();

        // Read status register
        begin_InstSSP
        sendByteSSP(RDSR);
```

```
        // Wait for status register to be written to
        while(receiveByteSSP() & 0x01);
        end_InstSSP

        // 2us Delay in between instructions
        nop();
        nop();
        nop();
        nop();

        // Write enable instruction--obligatory
        begin_InstSSP
        sendByteSSP(WREN);
        end_InstSSP

        // 2us Delay in between instructions
        nop();
        nop();
        nop();
        nop();

        // Bulk Erase instruction
        begin_InstSSP
        sendByteSSP(BE);
        sendByteSSP(0x00); // 24-bit fill value (not important)
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        end_InstSSP

        T1OSCEN = 1;   // Enable Timer1
        readBuf = 0x01;
        while((readBuf & 0x01)){
                // Wait for 33ms (8MHz clock)
                TMR0IF = 0;
                TMR1L = 0x00;
                TMR1H = 0x00;
                while(!TMR1IF);

                // Read status register of EEPROM
                begin_InstSSP
                sendByteSSP(RDSR);
                readBuf = receiveByteSSP();
                end_InstSSP

                // Display periods while waiting.....
                disableSSP();
                enableAUS();
                serialOut('.');
                disableAUS();
                enableSSP();
        }

        T1OSCEN = 0;   // Disable Timer1 and Interrupts

        // Confirm memory deletion to user
        disableSSP();
        enableAUS();
        serialOut('[');
        serialOut('O');
        serialOut('K');
        serialOut(']');
        serialOut(0x0D);
        serialOut(0x0A);

        serialIn = '\0';
        addrEEPROM = 0;        // Reset EEPROM pointer
        masterTimer = 0;       // Reset timestamp

        GIE = 1;
}
```

```
// Ctrl + R? Dump raw contents of EEPROM
else if(serialIn == 18){
        GIE = 0;                // Disable all interrupts
        addrEEPROM = 0;
        deadStream = 0;

        disableAUS();
        enableSSP();
        // Begin reading at address 0
        begin_InstSSP
        sendByteSSP(READ);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        // Read each byte and write to terminal until
        // we receive 10 blank bytes (a blank step)
        do {
                readBuf = receiveByteSSP();
                disableSSP();
                enableAUS();
                serialOut(readBuf);
                disableAUS();
                enableSSP();
                addrEEPROM++;
                if(readBuf == 0xFF)
                        deadStream++;
                else
                        deadStream = 0;
        } while(deadStream < 10);
        disableSSP();
        end_InstSSP
        enableAUS();

        // Go back to first blank byte
        addrEEPROM -= deadStream;
        // Clear pseudo-flag, wait for next keypress
        serialIn = '\0';
        GIE = 1;        // Enable all interrupts
}

// Ctrl + X? Formatted Hex Dump (useful for Terminals)
else if(serialIn == 24){
        GIE = 0;                // Disable all interrupts
        addrEEPROM = 0;
        deadStream = 0;

        disableAUS();
        enableSSP();
        // Begin reading at address 0
        begin_InstSSP
        sendByteSSP(READ);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        sendByteSSP(0x00);
        // Read each byte and write to terminal until
        // we receive 10 blank bytes (a blank step)
        do {

                deadStream = 0;

                // Read timestamp from EEPROM (3 bytes)
                wr_stepStartTime.high8 = receiveByteSSP();
                wr_stepStartTime.mid8 = receiveByteSSP();
                wr_stepStartTime.low8 = receiveByteSSP();
                // Read step length from EEPROM (2 bytes)
                wr_stepLength.high8 = receiveByteSSP();
                wr_stepLength.low8 = receiveByteSSP();
                // Read force summation from EEPROM (3 bytes)
                wr_forceAccum.high8 = receiveByteSSP();
                wr_forceAccum.mid8 = receiveByteSSP();
                wr_forceAccum.low8 = receiveByteSSP();
```

```
                        // Read max force from EEPROM last (2 bytes)
                        wr_forceMax.high8 = receiveByteSSP();
                        wr_forceMax.low8 = receiveByteSSP();

                        disableSSP();
                        enableAUS();

                        // Write timestamp to terminal (6 hex chars)
                        serialOut(hiHex(wr_stepStartTime.high8));
                        serialOut(lowHex(wr_stepStartTime.high8));
                        serialOut(hiHex(wr_stepStartTime.mid8));
                        serialOut(lowHex(wr_stepStartTime.mid8));
                        serialOut(hiHex(wr_stepStartTime.low8));
                        serialOut(lowHex(wr_stepStartTime.low8));
                        serialOut(' '); // Insert a space
                        // Write step length to terminal (4 hex chars)
                        serialOut(hiHex(wr_stepLength.high8));
                        serialOut(lowHex(wr_stepLength.high8));
                        serialOut(hiHex(wr_stepLength.low8));
                        serialOut(lowHex(wr_stepLength.low8));
                        serialOut(' '); // Insert a space
                        // Write force summation to terminal (6 hex chars)
                        serialOut(hiHex(wr_forceAccum.high8));
                        serialOut(lowHex(wr_forceAccum.high8));
                        serialOut(hiHex(wr_forceAccum.mid8));
                        serialOut(lowHex(wr_forceAccum.mid8));
                        serialOut(hiHex(wr_forceAccum.low8));
                        serialOut(lowHex(wr_forceAccum.low8));
                        serialOut(' '); // Insert a space
                        // Write max force to terminal (4 hex chars)
                        serialOut(hiHex(wr_forceMax.high8));
                        serialOut(lowHex(wr_forceMax.high8));
                        serialOut(hiHex(wr_forceMax.low8));
                        serialOut(lowHex(wr_forceMax.low8));
                        // Insert Newline at end of each step
                        serialOut(0x0D);
                        serialOut(0x0A);

                        disableAUS();
                        enableSSP();
                        addrEEPROM+=10;

                        // Check to see if we have reached the end of data
                        if(wr_stepStartTime == -1)
                                deadStream++;
                        if(wr_stepLength == -1)
                                deadStream++;
                        if(wr_forceAccum == -1)
                                deadStream++;
                        if(wr_forceMax == -1)
                                deadStream++;

                } while(deadStream < 4);

                disableSSP();
                end_InstSSP
                enableAUS();

                // Go back to first blank byte
                addrEEPROM -= 10;
                // Clear pseudo-flag, wait for next keypress
                serialIn = '\0';
                GIE = 1;        // Enable all interrupts
        }

        // Echo character to terminal
        else {
                serialOut(serialIn);
                // Enter? Proper newline
                if(serialIn==0x0D)
                        serialOut(0x0A);
```

```
                                // Backspace? Make last char disappear
                                else if(serialIn==0x08){
                                        serialOut(' ');
                                        serialOut(0x08);
                                }
                                serialIn = '\0';
                        }

                }


        }
}
```