A Distributed Data Acquisition Network for Vibration Measurement

by

Jeremy Thomas Braun

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2003

Author ....
Department of Electrical Engineering and Computer Science
May 19, 2003

Certified by ....
J. Kim Vandiver
Professor of Ocean Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Distributed Data Acquisition Network for Vibration Measurement

by

Jeremy Thomas Braun

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering

Abstract

Current data acquisition technology for studying vortex-induced vibration (VIV) in underwater pipes is insufficient for studying mode shape and high mode number vibrations. This work presents a prototype data acquisition system capable of acquiring high mode number data suitable for mode shape analysis. A suitable VIV experimental setup is presented, and the instrumentation design hurdles are analyzed. Using modern surface mount technology electronics, a prototype printed circuit board was designed and built. The prototype node is capable of being networked with other devices on an RS-485 serial bus to provide many simultaneous data capture points along the length of a pipe. Bi-axial acceleration data are sampled using a sigma-delta analog to digital converter, and stored in a local EEPROM by a PIC microcontroller. A networked serial communications protocol is developed, and the software for both the node microcontroller and a host computer that controls the sampling process is described. Sampling parameters and firmware updates can be carried out remotely over the serial bus, and captured data can be uploaded back to the host computer at the surface. The prototype's analog and communications performance is analyzed for robustness and scalability. Recommendations for future work to improve the performance of the device are given.

Thesis Supervisor: J. Kim Vandiver
Title: Professor of Ocean Engineering

Acknowledgments

Many people have helped me through my time at MIT. First and foremost, my parents put me through my first four years of school and always have offered their support freely. Kim Vandiver, John Kassakian, Jeffrey Lang, Paul Gray, Ron Roscoe and Kent Lundberg all had an enormous impact on my academic life throughout college. My roomates Rita and Emily deserve thanks for putting up with me while I wrote my thesis, and for valuable proof reading comments. Sherri, for being a friend and companion. Anne Hunter, for being available and clueful at all hours of the day. The theater staff at Rinaldi Tile Company for teaching me to sew, build and relax. TEΦ Xi chapter for many fond memories and a rich undergraduate life. Rachel, for always having a shoulder free.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern offshore oil and gas production requires floating drilling and processing facilities in water depths exceeding 5000 feet in the Gulf of Mexico. Exploration drilling and oil and gas production require long cylindrical drilling risers and production tubing and pipelines. These cylinders reach 55 inches in diameter for vertical drilling risers and 28 inches for catenary production risers. They are most often made of steel. In the case of catenary risers they may exceed 8000 feet in length. Ocean currents can cause these large steel cylinders to experience vortex-induced vibration (VIV). Vortices shed off the pipes as water flows past them. Lift and drag forces from the vortices cause the pipes to vibrate, which may result in fatigue of the steel and possible failure.

Low mode number vibration has been studied on short pipe lengths in controlled laboratory environments. Experiments with longer pipes have been conducted in limited numbers under uniform flow conditions. Real offshore risers experience non-uniform, or sheared, currents in the ocean. These currents can excite multiple modes and higher mode numbers in long sections of pipe. Few high quality data sets are available that permit the evaluation of high mode number vibration and mode shape identification. Data samples must be taken at many locations simultaneously along the length of the pipe which is not feasible in a long pipe using current instrumentation schemes.

The distributed sensor system presented in this thesis is intended to permit measurement at a large number of locations for a reasonable cost. The system will enable response amplitude and frequency analysis over the entire riser length. The design solves many deficiencies of currently available instrumentation schemes.

Chapter 2

Background and Motivation

Some background information about VIV is necessary to understand some of the requirements and challenges present in the design of this data acquisition system. The limitations and shortcomings of current data collection methods are also discussed.

2.1 Vortex Induced Vibration

A cylindrical body positioned perpendicular to a fluid flow will shed vortices in its wake. For a particular cylindrical body in a flow, the Reynolds number is defined as $\mathrm{Re} = \frac{Du}{\nu}$ where D is the diameter of the pipe, u is the flow velocity, and ν is the kinematic viscosity of the fluid. As the Reynolds number becomes larger than about 100, vortices shed from alternate sides of the cylinder, as shown in Figure 2-1.

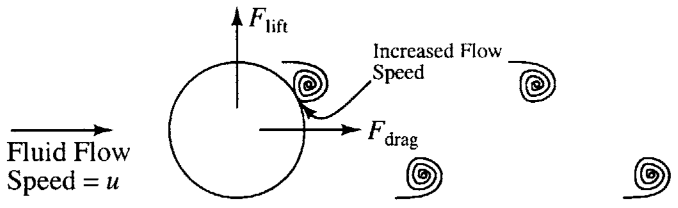


Figure 2-1: A cylinder shedding vortices

Increased local flow speed where the vortex is formed leads to reduced pressure, which applies a lift force transverse to the flow to the cylinder. When a vortex is shed from the opposite side of the pipe a force is experienced in the opposite direction.

The frequency at which this shedding occurs is called the Strouhal frequency, and can be found as

$$f_s = \frac{s_t u}{D}$$

The constant s_t is the Strouhal number (experimentally found to be ≈ 0.17 for flexible moving cylinders). Each vortex also has a drag component associated with it inline with the flow direction. This force is applied with each vortex so it occurs at twice the Strouhal frequency. The lift and drag forces are both proportional to the square of the flow speed.

If f_s is close to a natural frequency of the pipe the vortices will positively reinforce its vibration. This is a behavior called lock-in, and typically exhibits vibrations with the largest amplitude, and most strain.

When the Strouhal frequency and the pipe's natural frequencies do not coincide the behavior of the pipe is hard to predict. Several different modes can be excited in the pipe, and the amplitude fluctuates drastically. In some cases the higher frequency inline vibrations, caused by the drag force of every vortex, can dominate the pipe's response if they occur near a natural mode of the pipe. When the fluid flow is sheared (varies with depth) multiple vibrational modes of the pipe can be excited at the same time.

2.2 Limitations of Current VIV Data

The mode shape of long pipes experiencing high mode number VIV is of immediate interest in this research area. There is a paucity of data that is of sufficient spatial density to analyze this type of behavior. In order to look at mode shapes, the acceleration of the vibrating pipe must be sampled at many locations simultaneously along the length of the pipe. VIV acceleration data are not always collected in a manner which allows easy mode-shape analysis.

There are two main sources for VIV data: smaller laboratory research experiments and real data from offshore risers. Laboratory experiments often have many accelerometers wired back to a central data collection computer, which does collect data samples simultaneously along the length of the pipe. Since these experiments are usually conducted in a laboratory setting, they are often on too small a scale to excite higher mode numbers in the test pipes.

Data collection on offshore risers presents some other interesting challenges. It is prohibitively expensive to wire a significant portion of the riser with accelerometers each of which is connected directly back to the surface. Large portions of the risers can be instrumented using small, autonomous battery-powered units which are attached to the riser as it is deployed and collected weeks or months later. Unfortunately, because each node's sampling clock is free-running, the samples are all taken at different times with respect to one another. Thus the data are often useless for mode-shape analysis.

Clearly another instrumentation solution is needed. This paper details the design of a data acquisition system that would be suitable for acquiring the synchronous data sets needed to study high mode number VIV.

Fairing research is another area where this system would be useful. The industry standard way to prevent VIV is to add pivoting fairings to the risers, which prevent the formation of vortices in ocean currents. Few data are available on how long sections of pipe perform with different fairing coverage patterns. A suitably instrumented pipe could be used to determine whether 25%, 50%, or 100% of the pipe's surface requires fairings to effectively suppress VIV.

Chapter 3

Proposed Experiment

In order to study high-mode-number VIV, a very long (relative to its diameter) pipe is required. The proposed experimental apparatus consists of a 400 foot length of fiber reinforced composite tubing with a 1" inner diameter, and 1.38" outer diameter. Inside the pipe will be a number of nodes, each equipped with a bi-axial accelerometer with its sensitive axes perpendicular to the axis of the pipe. Figure 3-1 contains a

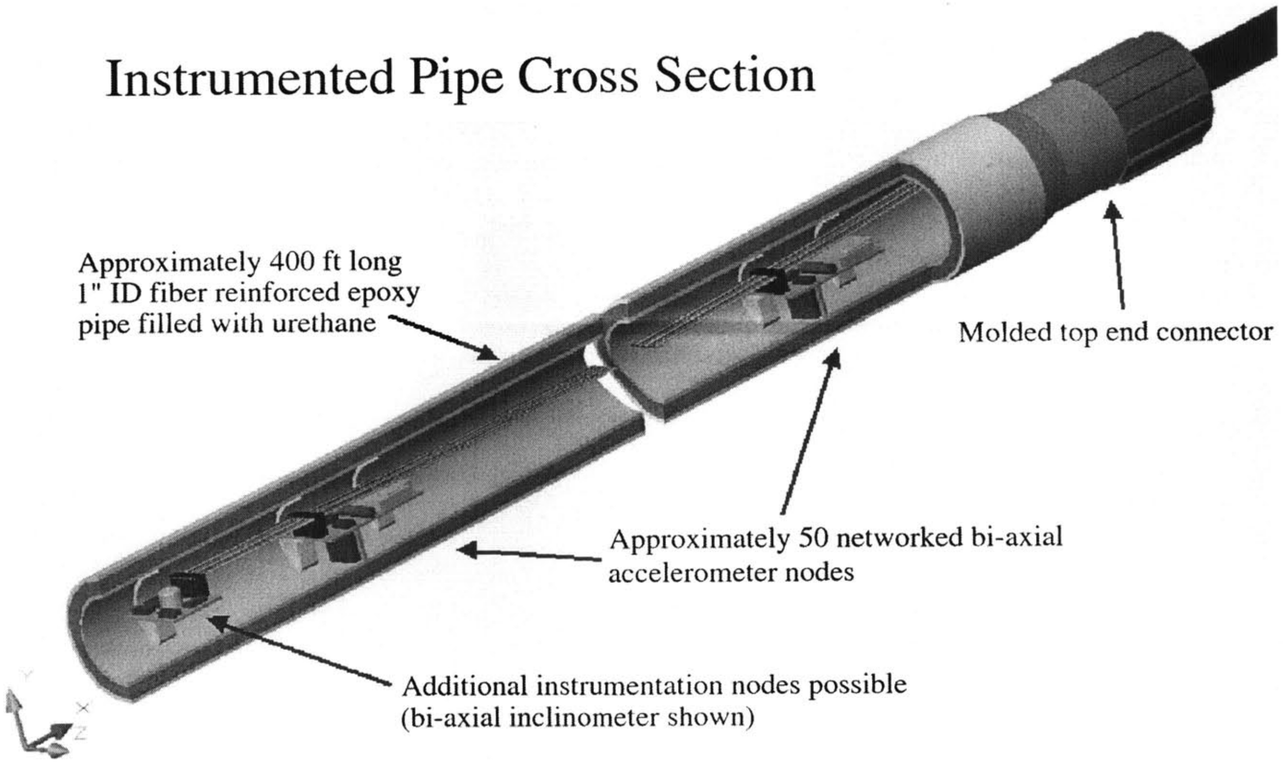

Instrumented Pipe Cross Section
Approximately 400 ft long
1" ID fiber reinforced epoxy
pipe filled with urethane
Molded top end connector
Approximately 50 networked bi-axial
accelerometer nodes
Additional instrumentation nodes possible
(bi-axial inclinometer shown)
Y
X
Z

Figure 3-1: Cut-away view of a section of pipe

conceptual cut-away view of a section of the pipe. Each individual node will contain analog-to-digital converters for each accelerometer axis to digitize the acceleration data, and a storage element to hold that data. All of the nodes will be linked together on a communications bus, so they can be controlled remotely from the surface, and so that the experimental data can be transmitted back to the surface.

The number of required nodes is driven by a spatial Nyquist criterion. Mode shapes are approximately sinusoidal. At least two measurement points per wavelength are required. For the planned experiments the mode number may be as high as 30. The number of wavelengths is $n/2$, so the required minimum number of sensors is approximately 30. The number of required sensors increases in proportion to peak flow velocity.

3.1 Physical System Requirements

The biggest design challenge is making the printed circuit board (PCB) small enough to fit inside the pipe. In order to store and transport the pipe, it must be coiled into a spool about eight - ten feet in diameter. Therefore, the board must be made narrower to keep it from binding inside the coiled pipe and breaking. Figure 3-2 shows a diagram of a coiled section of pipe containing an accelerometer node. From

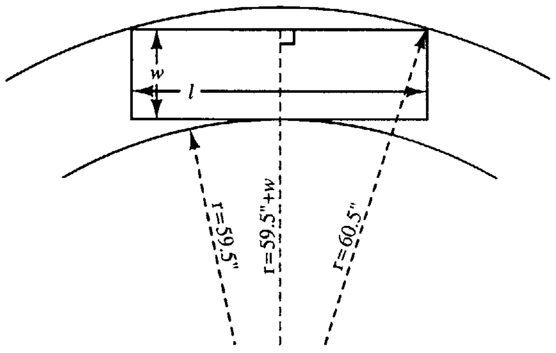


Figure 3-2: Accelerometer node in a coiled section of pipe

the diagram, the required width of the board in inches can be found as

$$W = \sqrt{60.5^2 - \left(\frac{L}{2}\right)^2} - 59.5$$

A a six inch long board would require a width of not more than .91".

Another physical restriction on the design of the PCB is that components which stand a significant distance off of the board can not be used, as they will not clear the wall of the pipe – especially near the edges of the board. For these reasons, surface mount technology (SMT) components should be used wherever possible as they save 50% or more of the space of normal through-hole components.

To insert the nodes into the final 400 foot long pipe, they will have to be pulled along on a string or other strength member. It is a poor idea to use the communications or power wires for this as it is too easy to break a solder connection, and render the entire system unusable. One way to prevent this is to run a long kevlar string along the underside of all the boards. Each board is anchored to the string, leaving some slack in the connecting wires to protect them from damage. When laying out the circuit board, space must be left for this strength member to run so that it does not rub against any components and damage them.

Once inside the pipe, the nodes have to be physically coupled to the pipe to accurately measure its acceleration. If they are left free, the circuit boards will rattle inside of the pipe, corrupting the quality of the data. The usual solution to this problem is to fill the pipe with a potting compound upon final assembly. Potting compounds in general are liquid mixes which can be poured into hollow spaces and which harden later. Care must be taken to select a potting material that is not too stiff, as using too hard a potting compound may cause the solder joints or PCB traces to break when the pipe is vibrating. The potting compound must have enough give to prevent it from ripping components from the board during operation and storage. It must also be non-conductive to prevent it from causing electrical shorts on the boards.

Filling the pipe with potting compound is an irreversible procedure. Once the compound hardens, physical access to the nodes is no longer possible, short of cutting the pipe open. Therefore, during the design process it is important to keep in mind that the device should be electronically configurable from the surface through the communications bus.

3.2 Accelerometer and ADC

Once the nodes are potted inside of the pipe, the orientation of the accelerometer's sensitive axes needs to be determined relative to each other. The easiest way to do this is to place the pipe flat on the ground and roll it over slowly while collecting data. The earth's 1g gravitational field will register on a DC-sensitive accelerometer and the orientation of each accelerometer can be determined from these data. Accurate DC sensitivity will heavily influence the design choices made in the analog signal path in section 4.

The fundamental frequencies of interest range up to about 25 Hz. Ideally the third harmonic would be preserved as much as possible as this is often a significant component of vortex-induced vibration. The Nyquist Theorem requires that a sampling rate of at least 150 Hz be used. Since building a perfect low-pass filter is impossible, increasing the target sample rate to 250 Hz gives about an octave in which to design an appropriate anti-aliasing filter.

As discussed in section 2.2, every ADC needs to sample at the same time in order to study mode shapes of the pipe. Therefore, the converter chosen should be externally triggerable. In addition, some method of distributing a triggering signal to all of the ADCs in the pipe needs to be developed.

Other considerations include ADC resolution and accelerometer range. In order to provide the needed dynamic range for slower speed (lower amplitude) experiments, an ADC with at least 16 bits of resolution is desired. Recall from section 2.1 that the forces exerted by the vortices are proportional to the square of the flow speed. For the faster planned experiments the peak acceleration experienced will be on the order of $20g$ - 30g. The chosen accelerometers must be able to measure signals of this magnitude. A programmable gain element in the analog signal path could also be useful in increasing the dynamic range for lower-amplitude signals.

3.3 Communications Bus

A serial communication bus is the ideal choice for data transfer in this system. The number of wires that can fit inside the pipe is small, so the less wires, the better. RS-485 is an asynchronous serial communications protocol which allows for speeds up to 10 Mbps, and up to several hundred nodes to be connected together on a shared bus up to 4000 feet in length. The data signal is transmitted along a pair of differentially-driven wires, which provides high noise immunity and good data retention over long distances. RS-485 transceivers are available in tiny packages from several manufacturers, and inexpensive 100 Ω unshielded twisted pair (UTP) Ethernet cable is perfect for wiring the bus.

3.4 Data Storage

To extract useful information from an experiment's data set, at least 20 cycles of steady-state VIV need to be monitored. With two 16-bit channels sampling a 1.0 Hz signal at 250 Hz the storage required for each node is

$$32\frac{\text{bits}}{\text{sample}} \cdot 250\frac{\text{samples}}{\text{sec}} \cdot 20\frac{\text{cycles}}{1\text{ Hz}} = 160\text{ kbits}$$

A few EEPROM products are available in small SMT packages with this capacity, and many FLASH products are available in much larger capacities, if desired. The prototype uses 512 kbit 24LC515 EEPROM from Microchip. It provides plenty of storage space and is available at reasonable costs from Digikey and other distributors.

3.5 Control

Each node needs to have a microcontroller or microprocessor to control the synchronous sampling procedure, data storage and retrieval, serial communications, etc. The chosen microcontroller should be small, easy to program, and should have peripheral units that facilitate these functions. Because physical access to the nodes will be impossible once installed in the pipe, it is extremely useful to chose a microcontroller with the ability to re-program itself remotely from the surface.

Chapter 4

System Implementation

The first part of this section discusses the actual node hardware components. The second section details the serial communication protocol used. The third and fourth sections discuss the software for the individual nodes, and the host computer, respectively.

4.1 Accelerometer Node

A system block diagram is shown in Figure 4-1. A detail of the acceleration signal

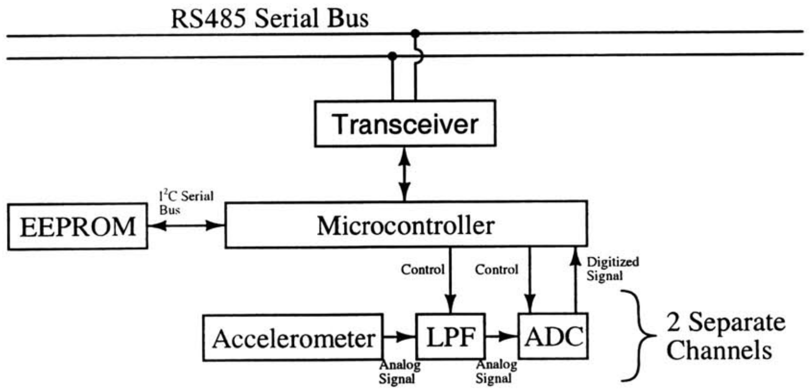


Figure 4-1: System block diagram

path is shown in Figure 4-2. The acceleration signals from the accelerometers pass through some buffering op-amps, a low pass filter, and a compensation network into the analog to digital converters (ADCs). A microcontroller interfaces with the

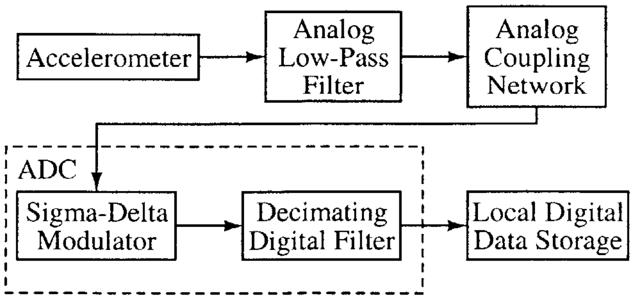


Figure 4-2: Acceleration signal path detail

ADCs, triggering each sample and retrieving the last conversion performed. The microcontroller stores the data in an external EEPROM and provides various services over the RS-485 bus, described in sections 4.3.1 and 4.3.2.

4.1.1 Accelerometer

The accelerometer chosen for the prototype node is the ADXL250 from Analog Devices. The ADXL250 contains two micro-electromechanical (MEM) accelerometers, oriented orthogonal to each other in the plane of the chip. It comes in a 14-lead surface mount cerpac package, small enough to fit inside of the composite pipe. Both channels are capable of producing a $\pm 50g$ signal, at 38 mV/g.

Both channel outputs are differential signals referred to a reference voltage generated on-chip which is nominally one half the power supply voltage. The reference voltages for the x and y channels are available at the x_{off} and y_{off} terminals of the ADXL250 respectively. It also offers decent noise performance of about $1\text{mg}/\sqrt{\text{Hz}}$, and can operate from a single 5 V supply.

4.1.2 Buffer and Filter

The output resistance of the accelerometer reference channel is 30 kΩ, and the output amplifier of the device is capable of supplying up to 100μA. In order to drive the ADCs, these signals must be buffered by a unity-gain op-amp. In addition, the accelerometer signals need to be low-pass filtered to avoid aliasing problems. The LTC2440s sample at 1.8 MHz. The low-pass filter consists of a second-order low-pass filter implemented with a Sallen-Key circuit. See appendix A for a treatment of the Sallen-Key architecture. The 3 dB point is located at 880 Hz. The 40 dB/decade rolloff provided by the low pass filter provides 120 dB of attenuation at half the sampling frequency (900 kHz). The phase shift of the filter is about −4° at 25 Hz, the highest principle frequency of interest, and the magnitude is only attenuated by 7 mdB. The frequency response of the low-pass filter can be found in figure 4-3. .

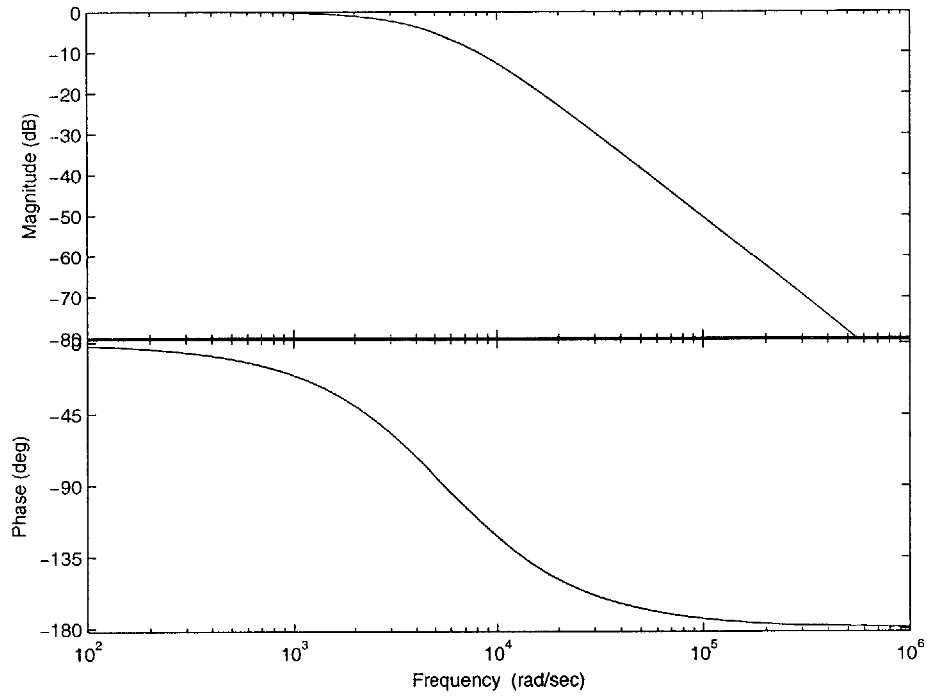


Figure 4-3: Low pass filter frequency response

Accurate DC performance is also a requirement of the sampling system. Most op-amps have a small (several mV) DC offset voltage associated with them. At sensitivities of 38 mV/g, several offset voltages added together will become a significant source of error at low amplitudes of experienced acceleration. In order to preserve performance with little DC offset error, expensive chopper-stabilized op-amps can be used. The LTC2052 was selected for its low noise, rail-to-rail output capability, and single 5 V supply operation. The op-amps periodically disconnect the op-amp (internally) from the input and output pins, and measures the offset voltage. The measured offset voltage is applied back to the inputs of the amplifier to correct the error. In this manner, offset voltages of several microvolts per op-amp, rather than several millivolts, can be obtained. To prevent the input voltage from exceeding the input range of the LTC2052, a resistive divider is used to half the accelerometer output with respect to the reference voltage.

4.1.3 Analog to Digital Converters and Coupling Network

Many analog-to-digital converters were considered for use. Most ADCs currently in production are free-running devices. An internal or external clock is divided down by the ADC and is used to determine the sampling rate. It is impractical to share a high-frequency ADC clock between all the nodes in the pipe. The majority of suitable externally triggerable ADCs are manufactured by Linear Technologies.

Both the LTC1864 and the LTC2440 were considered as potential ADC choices. The LTC1864 is a 16-bit switched-capacitor successive approximation device, while the LTC2440 is a 24-bit sigma-delta modulator ADC. Both also offer differential inputs, useful for the bipolar signals provided by the accelerometers. For the prototype the LTC2440 was chosen because its sigma-delta architecture offers a unique feature not provided by successive approximation ADCs.

Sigma-delta modulator ADCs work by purposefully oversampling the input signal at many times the desired data output rate. The ratio between the frequency at which the input is sampled and the frequency at which digital output codes are produced is called the oversampling ratio (OSR) of the ADC. By sampling at a higher rate than necessary, and using digital on-chip filtering to reduce the data rate by a factor equal to the OSR, anti-aliasing filters can be made simpler for a particular application. In addition, larger OSRs mean more samples are averaged together to produce an output code. Larger averages mean higher resolutions are possible as the OSR increases. The OSR of the LTC2440 is software configurable. The internal clock of the LTC2440 allows for the modes and oversampling ratios listed in Table 4.1. The 220 Hz mode of the LTC2440 is close to the target 250 Hz sampling rate. At this data output rate, the converter has an effective resolution of 16 bits. Slower experiments with lower frequencies of interest also exhibit smaller pipe deflections, which lead to lower signal levels. By increasing the OSR, the extra bits of resolution can be used to provide

Table 4.1: LTC2440 oversampling ratios and data output rates

OSR	Output Rate
64	3.52 kHz
128	1.76 kHz
256	880 Hz
512	440 Hz
1024	220 Hz
2048	110 Hz
4096	55 Hz
8192	27.5 Hz
16384	13.75 Hz
32768	6.875 Hz

gains in powers of two in software by discarding the unused (zero) most-significant bits. By providing a programmable gain in software, valuable circuit board space and component cost are saved.

Both ADC converters considered have switched-capacitor analog inputs. A MOSFET acts as a sampling switch which connects and disconnects the input to a sampling capacitor with a value between 1 pF and 30 pF. Figure 4-4 illustrates a typical

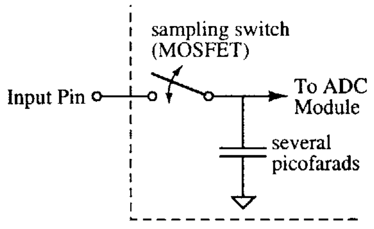


Figure 4-4: Typical switched capacitor input

switched capacitor input. The current spikes caused by the sampling capacitor being connected to the input can not be corrected by an op-amp alone. Glitches caused by the sampling capacitor in the input voltage will cause erroneous conversions and corrupt data from the ADC.

The ideal solution is to create a charge reservoir at the input by connecting a large (typically 0.1 μF) capacitor to it, close to the chip. The current required to charge the sampling capacitor will not change the voltage across the larger filter capacitor.

The input to the ADC will remain stable, and an accurate conversion will result. Unfortunately, most op-amps, especially rail-to-rail output op-amps, cannot directly drive large capacitive loads. Voltage gain in the output stage and the negative phase shift caused by the filter capacitor cause the op-amp to oscillate. The solution is the coupling network shown in Figure 4-5. The 100 Ω resistor and 330 pF capacitor

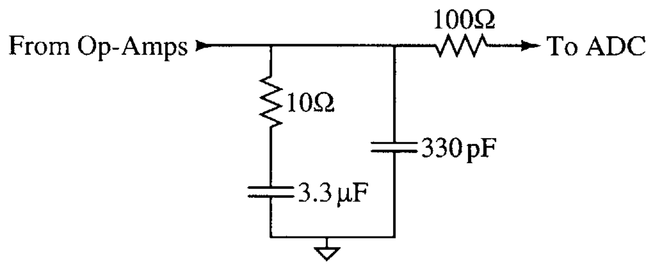


Figure 4-5: ADC input coupling network

attenuate frequencies above 5 MHz, keeping glitches caused by the sampling capacitor from being fed back to the driving op-amps. The 3.3 μF capacitor provides the charge reservoir necessary to stabilize the input voltage to the ADC. The 10 Ω series resistor keeps the op-amp from becoming unstable with such a large capacitive load.

4.1.4 Microcontroller

The microcontroller is the heart of the sampling node. It controls the data acquisition process, the serial communications with the host computer, and the local storage of acceleration data. Choosing a microcontroller is difficult as there are many different manufacturers and models, each with different features.

The PIC16F876 was deemed the most appropriate choice for both its price and features. The entire PIC16 model line offers a simple, easy to program 35-instruction instruction set architecture. The PIC16F87X series of microcontrollers are able to write to their own program memory, allowing for the remote update of the firmware. This series also includes a built-in asynchronous serial port module, as well as an I^2C serial interface, discussed in the sections below.

Microchip's microcontrollers are also relatively inexpensive, and are kept in stock at several major component distributors. The 8051 architecture was considered, but many 8051 clones do not allow for remote program updates, and most come in packages too large for practical use in this application. The author has also worked extensively with similar microcontrollers in the PIC16 product line in the past.

4.1.5 External Data EEPROM

The external data EEPROM is used to store the samples taken during an experiment. Ideally, the interface would be serial, not parallel, as wires and space on the circuit board are scarce resources. An ideal interface choice is the I^2C serial bus. I^2C is a two-wire, short-range serial bus standard invented by Philips for communication between integrated circuits. Microchip makes I^2C EEPROMs in a variety of sizes. The 24LC515 is a 512 kilobit device that meets the storage requirements detailed in section 3.4. It comes in an 8-pin wide small outline package, which fits easily on the circuit board.

4.1.6 RS-485 Serial Bus Transceiver

The RS-485 transceiver is responsible for converting the 5 V logic levels used by the PIC's on-board serial port to the differential RS-485 signal. It is also responsible for tristating the bus drivers when in receive mode so that another transmitter can use the bus. The normal input impedance of an RS-485 transceiver is 12 kΩ, allowing for a maximum of 32 nodes to be connected to the bus. By using a hi-impedance RS-485 driver, that number can be more than doubled. A slew rate-limited device to reduce electromagnetic interference from sharp transitions on the bus is also required. The LTC1487 from Linear Technologies is one of many parts available that fit these requirements, and is readily available from a number of electronics distributors.

4.2 Serial Communication Protocol

The RS-485 serial bus is a shared resource. Only one transceiver can actively drive the bus wires at a time, otherwise data corruption results. A command/response serial protocol is used to ensure that only one node or the host computer is trying to transmit on the bus at any one time.

The protocol is packet-based and reserves three special control characters listed in Table 4.2. One is for delimiting the start of packets, one is for delimiting the end of packets, and one is used as an escape character.

Table 4.2: Serial control characters

Name	Function	Decimal	Hexidecimal
STX	Start of packet	15	0x0F
ETX	End of packet	3	0x03
DLE	Escape character	16	0x10

Figure 4-6 is a diagram of a data packet. The ADDRL, ADDRH, ADDRU, and DATA sections are optional, however the rest of the bytes shown are necessary in a well-formed packet.

The contents of a full packet are illustrated in Figure 4-6, and are discussed in detail below.

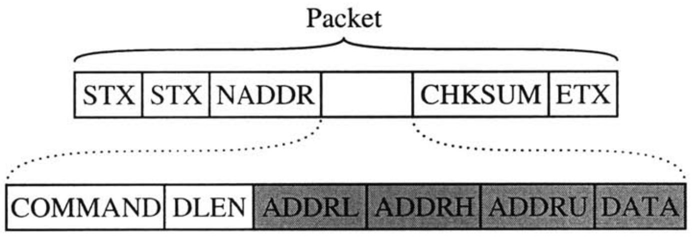


Figure 4-6: Packet contents

4.2.1 Start of Packet and Baud-Detection

The first two bytes of any packet to be transmitted are two STX characters which mark the start of a packet. Two STX characters are sent in order to allow the baud rate to be auto-detected. When a node receives a byte at the beginning of its receive loop, it counts the number of clock cycles between the first and last rising edges. Using this count, it can automatically compute the baud rate being used on the serial bus and configure the baud rate generator module appropriately. Figure 4-7 is a timing diagram detailing this process.

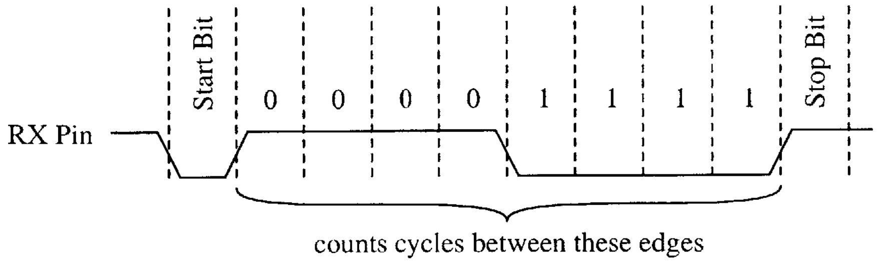


Figure 4-7: Autobaud timing diagram

After the first STX character, a second is sent to signal the start of a packet. A one-byte node address is also sent. Each hardware node stores a unique address in its on-board non-volatile EEPROM storage, which is loaded during the power-on sequence. If a packet's address matches the node's address after the packet is received the node continues to process it. If the address does not match it waits for the next packet. The COMMAND byte identifies a command or action to perform. Available commands are listed in Table 4.3. The DLEN byte is used as a counter for some of the commands, detailed in the section below. DLEN must be non-zero unless a reset command is desired. The ADDRL, ADDRH and ADDRU bytes store addresses for the data transfer commands. A variable length data field comes next. The length is determined by DLEN and the particular COMMAND requested. The CHKSUM byte is the two's complement of the 8-bit sum of all bytes from the address up to the last data byte, inclusive. The ETX byte marks the end of the packet. If the ETX, STX, or DLE bytes appear between the STX or ETX bytes that delineate a packet, they are preceded by an extra DLE to escape them, to avoid a framing error.

Once a node correctly receives a packet addressed to it, it carries out the requested command and responds with a similar packet to either return the requested data or to indicate the reception of the command packet. After the response packet is sent, the node frees the RS-485 bus and the host computer is free to transmit another packet.

4.3 Node Firmware

The firmware installed on every node is split into two different parts, the bootloader and the user program. Both programs implement the serial communication protocol described in Section 4.2, and a subset of the commands listed in Table 4.3. Having separate program code spaces allows the sampling firmware on each node to be updated remotely from the surface.

4.3.1 Bootloader

One reason the PIC16F876 microcontroller was chosen as the development platform is its ability to rewrite its program memory. This allows self-updating code to be written for the device. As discussed in section 3.5, remote updates are a necessity for this system because the nodes will be embedded in potting compound in a pipe section, and will be physically unavailable for manual repair and updates.

A small, well-tested piece of software that runs at startup called the bootloader is responsible for providing command functionality to allow for remote updates. The bootloader implements the first six commands in Table 4.3, as well as a special reset command, described below. The assembly code for the bootloader can be found in Appendix D.

Command: Reset

The Reset command, implemented in both the user firmware and the bootloader causes the microcontroller to reset itself, and begin execution from the beginning of the bootloader. To select this command, a packet with `DLEN` set to zero should be sent to the node to be reset. For this reason, any valid packet which is not a reset command should have a `DLEN` field not equal to zero.

Command: ReadVersion

The ReadVersion command returns two bytes in the data segment of the response packet. This command is implemented in both the bootloader and the user firmware.

If the most significant bit is zero, the device is currently in bootloader mode. If it is set, the node is in user mode. The remaining fifteen bits make up major and minor version numbers which can be used to keep track of which software revision is running on the node.

Command: WriteProgMem

WriteProgMem is a command used to write to the program memory on the microcontroller to update the user program on the sampling node. It is given `DLEN` words in little-endian order to write to the program memory address specified in the `ADDR` bytes of the packet. Little-endian order means placing the least significant byte before the most significant bytes of a multi-byte number. This byte order is used on Intel machines, on which the host computer software was developed. Writes to program memory must be aligned on 4-word (8-byte) boundaries. This allows newer PIC devices that only support writing the program memory in 4-word blocks to be supported easily in the future.

Command: ReadProgMem

The ReadProgMem command is the companion to the WriteProgMem command. It reads `DLEN` words from the microcontroller's program storage EEPROM, and returns it in the response packet. This command's intended use is to verify that a user program code update is successful.

Command: WriteEE

This command is used to write `DLEN` bytes of data from the packet to the data EEPROM on the microcontroller. This is in a separate memory space from the program memory. Writes to the internal EEPROM of the microcontroller can be used to effect a change of address for a node. Care must be taken that the new address does not exist on the serial bus already, otherwise potentially damaging bus collisions could occur. This can also be used to write persistent configuration data to each device, such as calibration data.

Command: ReadEE

This command reads `DLEN` bytes from the data EEPROM memory on board the microcontroller. This can be used to verify that an address change was successful, or to read other persistent configuration parameters that may be stored in the node's EEPROM.

Command: DoRVReset

This command is implemented in both the bootloader and the user firmware. It causes the node to restart at the user firmware entry point. This is useful for getting address, or other configuration changes, to take effect. A simple response packet is sent to let the host computer know the reset was successful.

When the user code starts, it becomes responsible for handling all serial communication. Commands that are implemented by the bootloader are no longer available, unless a real reset is performed. The most significant bit in the version allows the host computer to query a node to find out what state, bootloader or user, it is in. Thus, an appropriate reset command can be carried out before attempting to use a function unimplemented in the current mode.

4.3.2 User Firmware

The bootloader exists primarily to provide services that allow the host software to update the user firmware and perform other maintenance tasks. The user code is responsible for providing the ability to sample and store data, and to transmit it back to the surface.

As section 4.3.1 mentioned, the user code implements the DoRVReset and ReadVersion commands, but none of the read/write commands that modify the microcontroller's internal memory stores. This is an attempt at program and data memory security, to keep the user program from writing to the program or internal data memory, and corrupting it. The assembly code for the user firmware can be found in Appendix E.

Command: SetSampleCookie

An experiment is defined as the collection of a single data set at a particular sample rate, and the transmission of those data back to the surface. During an experiment, every node on the bus must take a sample at the same time in order for accurate studies of mode shapes to be performed. The sampling clock for the entire bus must be derived from a single node. The node which generates the sampling clock is said to hold the sampling cookie. Section 4.5 covers the sampling process in more detail.

The SetSampleCookie command is used to indicate to a node if it does or does not have the sample cookie. If the first byte in the data section of the packet is 0xAA, then the node assumes it is to generate the sampling clock. Otherwise, the node expects to be a slave node during sampling, and listens for the sample trigger byte on the serial bus.

Command:GetSampleCookie

This command is used to ensure that only one node holds the sampling cookie before an experiment begins. If more than one node has its cookie bit set at the time of an experiment, bus contention will occur when sampling begins. This can cause asynchronous sampling or missed samples, and can damage the RS-485 drivers on the boards.

Command: Set/GetNumSamples

During an experiment each node maintains a count of how many samples it has taken. Once it reaches the 24-bit number set by SetNumSamples, the node exits sampling mode, and returns to the serial packet receive/response routine. GetNumSamples is used to verify that every node has the correct number of samples set before an experiment begins.

Command: Set/GetSampleRate

The node which holds the sampling cookie generates the sampling clock using the onboard 16-bit TMR1 on the microcontroller. The value used to reset the timer, which determines the sampling period, is set/read by SetSampleRate and GetSampleRate, respectively.

Command: EnterSampleMode

Once an experiment is configured, each node is sent this command. Once a node enters sampling mode, it must complete an experiment. After sending a response packet to the EnterSampleMode command, the node waits to receive five `STX` bytes in a row. In this way other packet-based transfers can occur on the bus without causing a node to start sampling data (because only 2 `STX` bytes every appear in a row in any packet transmission). Once every node has received and responded to the the EnterSampleMode command, the host computer sends five `STX` bytes, and the experiment begins. See section 4.5 for information on sampling process during an experiment.

Command: Start/GetMemoryPage

In order to increase the utilization of the serial bus transactions when uploading experiment data to the host computer the data retrieval functions have been pipelined. It takes a significant amount of time to read data from the external EEPROM. Reading 128 bytes with a 384.6 kHz clock takes at least 2.6 ms, time enough to send another 33 bytes on the serial bus. If the page read size is larger, this wasted time grows as well. Consequently, reading a memory page is split into two operations. StartMemoryPage sets the length and address of the block to be read. The node responds immediately with an acknowledge packet, and then reads the requested memory page into internal memory, while the host computer is triggering page reads on other nodes.

After the node has acquired the page from the external EEPROM, the host computer sends a GetMemoryPage request, and the node sends the page back from its fast internal RAM, increasing the utilization of the serial bus during data upload.

Command: TakeSendSample

This is primarily a debugging command which causes the node to take one sample from both accelerometer channels and send it back to the host computer. Note that the ADCs used will output the result from the last conversion, not the one triggered by reading a sample from it. To get a current result, two TakeSendSample commands must be executed in a row. The value returned by the second command issued is the result from the conversion triggered by the first command.

Command: Get/SetOSR

The LTC2440 ADCs used can have their over-sampling ratio programmed via the serial bus. This command allows the host computer to select the particular OSR to be used for the current experiment, or to determine the current OSR being used by a particular node.

Command: GetEOC

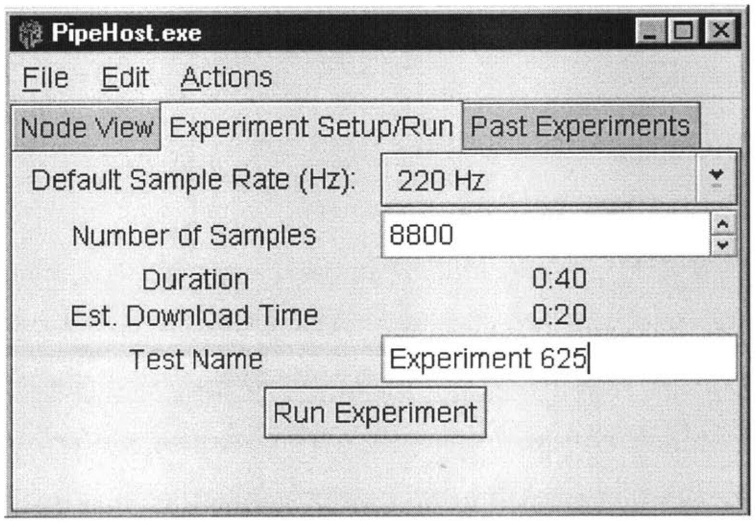
The LTC2440 continuously samples the input over a conversion period. Since the microcontroller reads data from the converter very close to the output rate, it is possible that a conversion could still be in progress when the microcontroller tries to read a sample. The end of conversion (EOC) output of the ADC is monitored during an experiment. After an experiment is finished, the host computer can use GetEOC to determine if this error occurred. If so, the timer reset value that controls the sampling rate can be modified using the SetSampleRate command to increase the sampling trigger period.

4.4 Host Computer Software

The host computer software is a program used to communicate with the sampling nodes over the serial bus. It is written in C, and uses the Gimp Tool Kit (GTK) widget set to provide an easy to use graphical user interface. The software allows the user to update the user firmware on the sampling nodes on the bus. It also allows the user to easily set-up and run experiments, through a simple point-and-click interface. The sample rate and the number of samples is selected. The user clicks 'OK', and the software sets up the serial bus to take data, has the nodes collect the samples, and downloads the data to files in the host computer. It also stores data about each experiment and can export the data as comma separated value or Matlab files for import into popular signal processing programs. A planned enhancement is an integrated experiment browser, that provides an Explorer-like interface to the experiments stored on disk.

4.5 Sampling Process Overview

The first step in initiating an experiment is to set the sampling parameters for all of the nodes. Figure 4-8 shows the window used to set these values. The first option



Default Sample Rate (Hz): 220 Hz
Number of Samples 8800
Duration 0:40
Est. Download Time 0:20
Test Name Experiment 625

Figure 4-8: Experiment setup window

is the sampling rate. The LTC2440 ADC is capable of several different oversampling ratios and data output rates. This dialog box allows the user to select the particular OSR/data rate that is best for the particular experiment being run. The user must also select the number of samples to be taken. For data output rates lower than

220 Hz, a software-configurable gain of powers of two could be added, because the LTC2440's resolution exceeds 16 bits for these lower data rates. This is not currently implemented in the prototype node software. The comments in the TakeSample procedure in the user assembly code found in Appendix E describe how this feature may be added to the prototype. The number of samples together with the sample rate determines the length of the experiment. Once these two parameters are set, 'Run Experiment' can be selected.

Figure 4-9 is a flow chart that outlines the sampling process. Before an experiment starts, the host computer sets the number of samples on each node and ensures that only one node on the bus has the sampling cookie. By default the node with the lowest address on the bus is given the sampling cookie. The user may select a different node to hold the cookie from the serial bus view screen, shown in Figure 4-10. The timer reset value is computed and set on the node with the cookie. Every node is sent the EnterSampleMode command, and five `STX` bytes are written to the bus. At this point the sampling process is taken over by the node that holds the sampling cookie.

During the sampling process, the cookie node waits for its timer to overflow, indicating that a sample should be taken. It takes control of the serial bus, and sends a `DLE` byte. Every node monitors the serial bus for a falling edge caused by the start bit of the `DLE`. On this falling edge each node takes a sample and stores it into the external data EEPROM. The `DLE` byte is removed from the serial receive buffer, the number of samples counter is decremented, and every node continues to wait for the next falling edge on the serial bus. When all the samples have been taken, the nodes return to the packet reception routine. As mentioned in section 4.3.2, the first sample taken from the ADCs is old, and should be discarded. Each node throws away the first sample it takes, and does not store it in the data EEPROM.

Using the serial bus as the sampling trigger line offers several advantages. No extra wires besides the power and the two RS-485 lines need to be run in the pipe. Because the RS-485 hardware is designed to accurately carry digital signals long distances, no extra hardware is needed to prevent erroneous samples from being taken. The

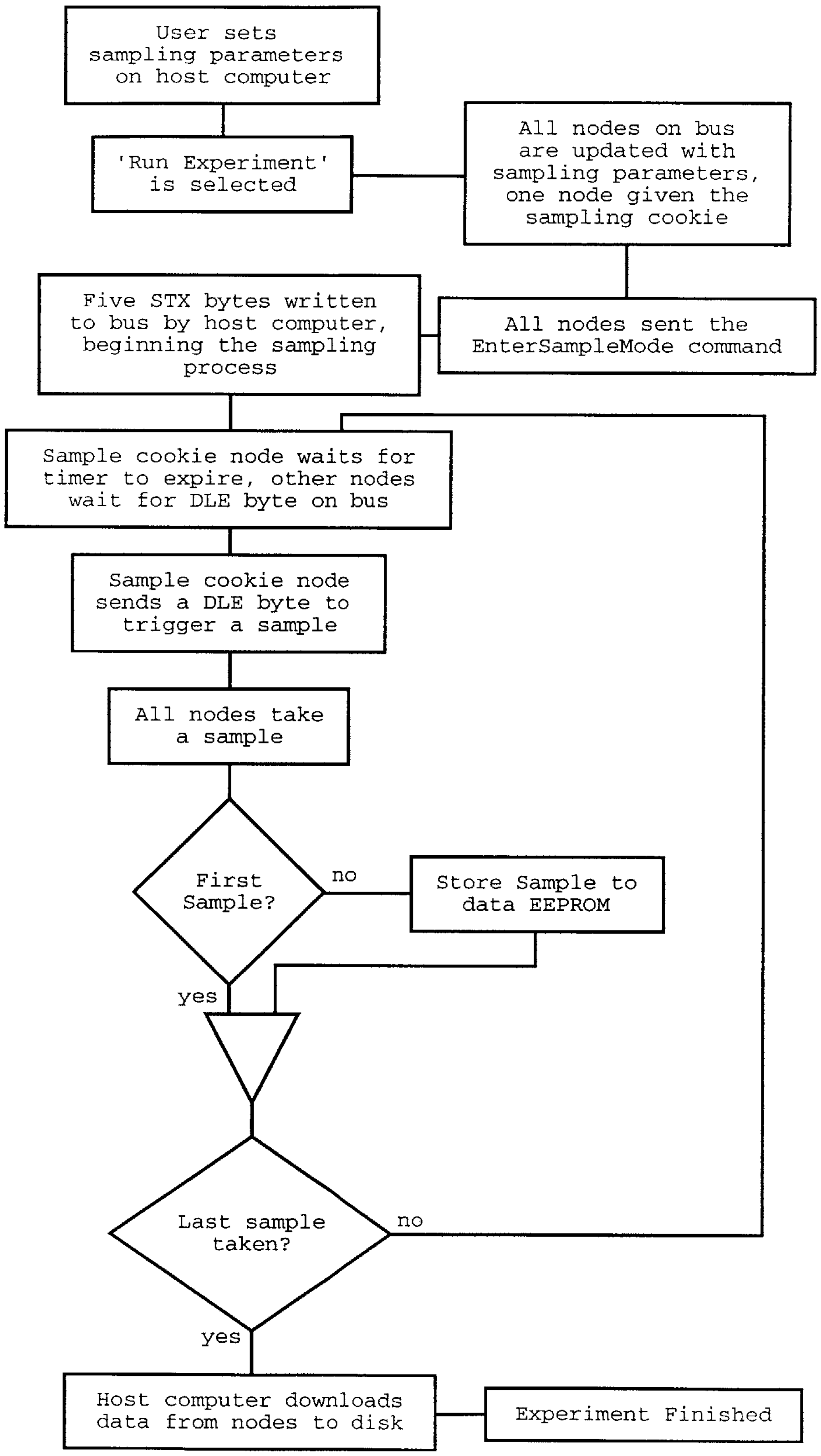
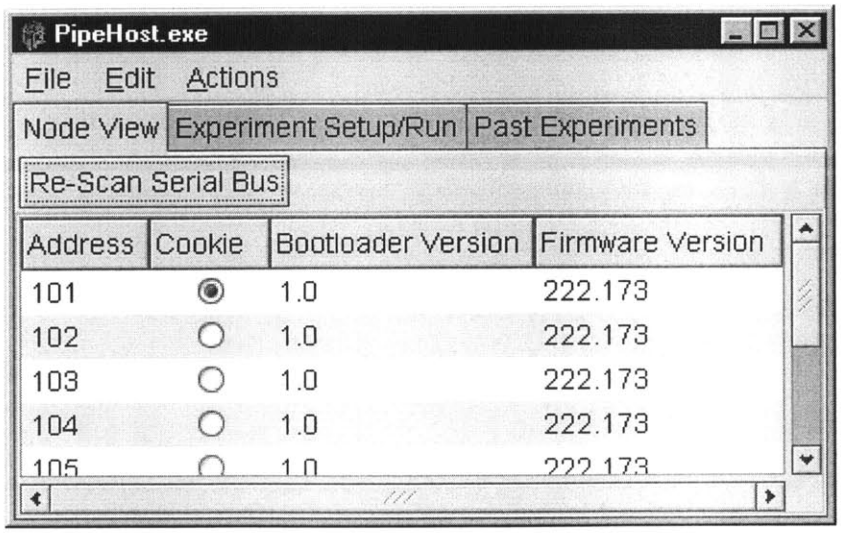

User sets sampling parameters on host computer
'Run Experiment' is selected
All nodes on bus are updated with sampling parameters, one node given the sampling cookie
All nodes sent the EnterSampleMode command
Five STX bytes written to bus by host computer, beginning the sampling process
Sample cookie node waits for timer to expire, other nodes wait for DLE byte on bus
Sample cookie node sends a DLE byte to trigger a sample
All nodes take a sample
First Sample?
no
Store Sample to data EEPROM
yes
Last sample taken?
no
yes
Host computer downloads data from nodes to disk
Experiment Finished

Figure 4-9: Flowchart of the sampling process

Table 4.3: Serial command definitions

Name	Value	Description
ReadVersion	0	get firmware version number
ReadProgMem	1	read a block of program memory
WriteProgMem	2	write a block of program memory
ReadEE	3	read a block of EEPROM data memory
WriteEE	4	write a block of EEPROM data memory
DoRVReset	5	execute a soft reset to the user code
SetNumSamples	6	set the number of samples for an experiment
GetNumSamples	7	get the number of samples for an experiment
SetSampleRate	8	set the sample rate for an experiment
GetSampleRate	9	get the sample rate for an experiment
SetSampleCookie	10	mark this node as the sample clock
GetSampleCookie	11	check if this node is set as the sample clock
EnterSampleMode	12	enter the sampling loop
StartMemoryPage	13	read an external EEPROM block to RAM
GetMemoryPage	14	send the last page fetched using StartMemoryPage
TakeSendSample	15	take a single sample and transmit it
GetOSR	16	get the LTC2440 oversampling ratio
SetOSR	17	set the LTC2440 oversampling ratio
GetEOC	18	see if an End of Conversion error occurred

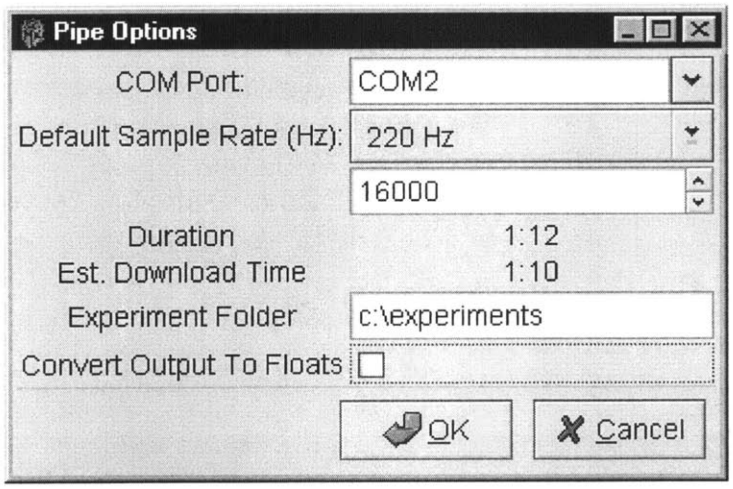


Address	Cookie	Bootloader Version	Firmware Version
101	●	1.0	222.173
102	○	1.0	222.173
103	○	1.0	222.173
104	○	1.0	222.173
105	○	1.0	222.173

Figure 4-10: Sample cookie node selection dialog

host computer can also count the DLE bytes that appear on the bus to provide an experiment progress bar for the user.

After all the samples have been collected, the host computer begins issuing Start-MemoryPage and GetMemoryPage requests to nodes on the bus. Data are written in comma separated value (CSV) format to files in the experiment directory, specified in the entry box shown in Figure 4-8. Each node's data are placed in a separate file, named after the node address. The resulting files can be imported into Matlab or Excel for processing later. Data can be written as either raw digital output codes (numbers ranging from -2^{15} to $2^{15}-1$), or as float values representing voltages. The selected output format is chosen in the default configuration dialog, shown in Figure 4-11.


Pipe Options

COM Port: COM2

Default Sample Rate (Hz): 220 Hz

16000

Duration 1:12

Est. Download Time 1:10

Experiment Folder c:\experiments

Convert Output To Floats ☐

Figure 4-11: Default experiment configuration dialog

Chapter 5

Results

5.1 Printed Circuit Board

The Eagle automated printed circuit board (PCB) design CAD package was used to design a circuit board that implements the system described in section 4. A complete schematic of the board can be found in Figure B-1. Figure 5-1 is a photograph of a completed prototype. The final circuit board is 0.95" wide and $3\frac{7}{8}$" long.

The PCB is composed of four layers. The top and bottom layers are both for routing signals. The two inner layers form power and ground planes which help reduce noise and aid routing signal traces in such a small space. Almost all components used are surface mount technology in order to conserve space. Most of the components were laid out on the top side of the board in order to leave a clear path for the kevlar strength member to run along the bottom.

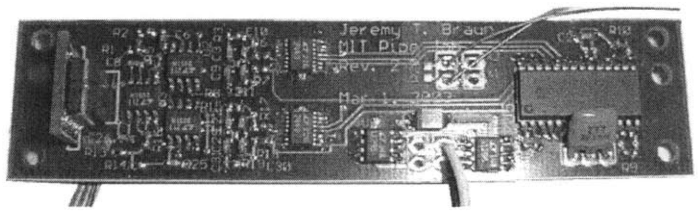


Figure 5-1: Prototype printed circuit board

5.1.1 Layout Considerations

Accelerometer Mount

As the sensitive axes of the accelerometer are in the plane of the chip, it must be mounted in the plane of the cross section of the pipe in order to measure inline and crossflow vibration. Figure 5-2 illustrates how the accelerometer is attached

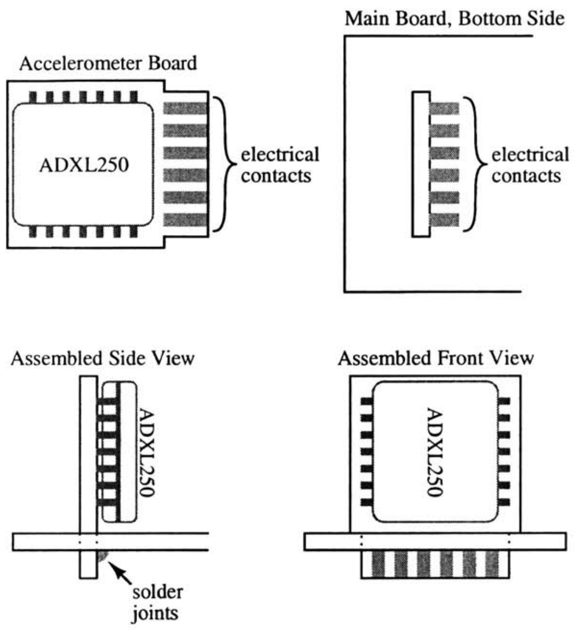


Figure 5-2: Accelerometer mounting

to the main circuit board. It is soldered onto it's own PCB which has a tab with solder contacts on one side. This is inserted through a slot on the main board, and the contacts are soldered to each other on the reverse side. Note the accelerometer mounted on the left in Figure 5-1.

Power Planes

The 5 V power plane is split into two separate sections, one for the analog circuitry, and one for the digital components. Each has its own voltage regulator and filter capacitors. This is an attempt to keep the high-frequency digital noise present on the microcontroller's 5 V supply from contaminating the analog supply. The ground plane for both supplies is joined at only one point on the board, to keep large ground currents on the digital side of the board from affecting the analog ground plane. Figures B-3 and B-4 of the board's copper layers illustrates this separation.

5.2 Software Test

The host computer software was successfully compiled and installed on a Windows 98 computer. Communication tests were successful, and multiple prototype nodes were successfully connected together on the RS-485 bus. An RS-232 to RS-485 converter was used to connect the host computer to the RS-485 bus. The 485SD9TB from B&B Electronics was selected for its low cost and ease of use. Example experiments were run using a Hewlett-Packard signal generator as input. Figure 5-3 shows a few example waveforms captured with the sampling hardware. The ability to remotely update

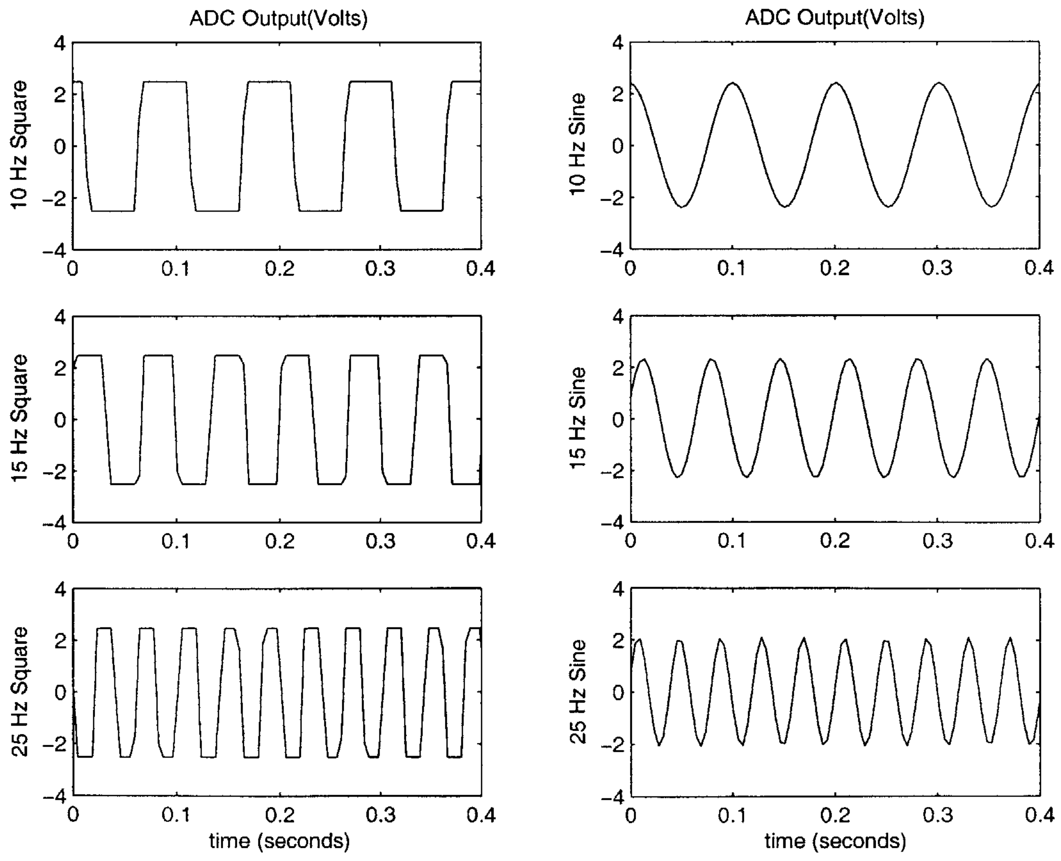


Figure 5-3: Captured test traces

the user program was also successfully implemented and tested on the prototype.

Chapter 6

Future Work

Several problems exist in the final design of this prototype. Data transmission, aliasing and ADC selection, and board size/layout are all issues that should be addressed in future versions of this data acquisition system.

6.1 Data Transmission

The RS-485 serial bus was chosen because it allows for a large number of devices to be connected together on long wire lengths using only two signal wires and a common ground connection. The specification allows for speeds up to 10 Mbps. However, the maximum speed of most RS-232 serial ports on computers today is only 115 kbps or, at most, 230 kbps.

For example, assume that a 60 second long experiment has just been run, sampling at 250Hz. Each node has

$$250\frac{\text{samples}}{\text{sec}} \cdot 60\text{ sec} \cdot 32\frac{\text{bits}}{\text{sample}} = 480000\text{ bits}$$

to upload to the host computer. With 50 nodes on a 115 kbps connection, this can take up to four and a half minutes, not including the serial protocol overhead. Waiting five minutes or more for the data from a one minute experiment is unacceptable, especially when experiment time on a boat is being billed.

One solution is to use a dedicated RS-485 card capable of higher bus speeds. The largest disadvantage of this is that most RS-485 interface boards capable of speeds greater than a normal serial port are PCI cards for desktop computers, not PCMCIA cards for laptops. Taking a desktop computer, monitor, keyboard and mouse to run experiments is much more inconvenient than using a laptop. One option to consider in the final implementation of the system is splitting the serial bus into two separate busses, with adjacent nodes on alternate busses. This is illustrated in Figure 6-1. Two separate serial ports on a laptop can be used to transmit data from both busses

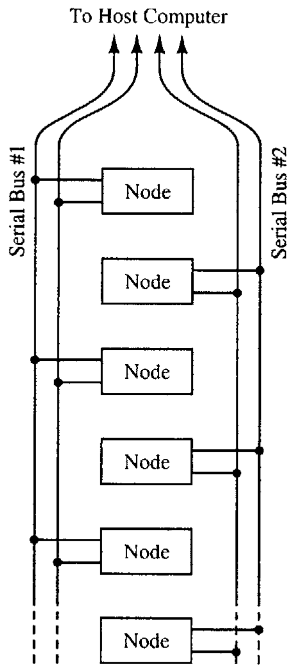


Figure 6-1: Dual serial bus implementation

in parallel, effectively halving the upload time. This has an added advantage. If one bus breaks, the other might remain functional, and the entire pipe may not be lost. A special node could sit at the surface, connected to both serial busses, to provide a simultaneous sample trigger.

Other communication protocols could also be considered. Unshielded twisted pair Ethernet offers speeds of 10 or 100 Mbps. The wire pairs in 10Base-T and 100Base-T wiring are not shared. All connections are dedicated links between two transceivers. Packets from the far end of the bus would have to be relayed up the bus from node to node to reach the host computer. This chaining would also require that another sample triggering method be used. 10Base-2 Ethernet is a shared two-wire bus that would probably serve best. Unfortunately, 10Base-2 uses 50 Ω coaxial cable, which is physically large and unwieldy. Such a cable could probably be used in place of the kevlar strength member, but finding room for such a thick wire in the pipe, and the taps to provide electrical connections to the nodes will be problematic.

6.2 Aliasing

Another reason the LTC2440 was chosen as the ADC is because the datasheet claims that "Combined with a large oversampling ratio, the LTC2440 significantly simplifies anti-aliasing filter requirements" [10]. Other sigma-delta modulator ADCs such as the one included on Burr-Brown's PCM3501 claim that because the over-sampling ratio is 64 times the sampling frequency, only a single-pole filter is required for 16-bit resolution [3]. After construction of the prototype it was determined that the anti-aliasing filter described in section 4.1.2 is radically insufficient for this application. Figure 6-2 shows a few example captures that exhibit significant aliasing problems.

To prevent aliasing, a better low-pass filter must be inserted into the signal path before the ADC. There are many different filter choices, and implementation options, outlined below.

6.2.1 Filter Types

There are several different low-pass filter designs that offer distinct advantages and disadvantages over each other. They are Butterworth, Chebychev, elliptical, and Bessel filters.

Butterworth Filters

Butterworth filters offer the steepest rolloff of any filter whose magnitude response is maximally flat in the pass-band. Additionally, the pole-pairs that make up the the transfer function of a Butterworth filter have low Q, or quality factors. This

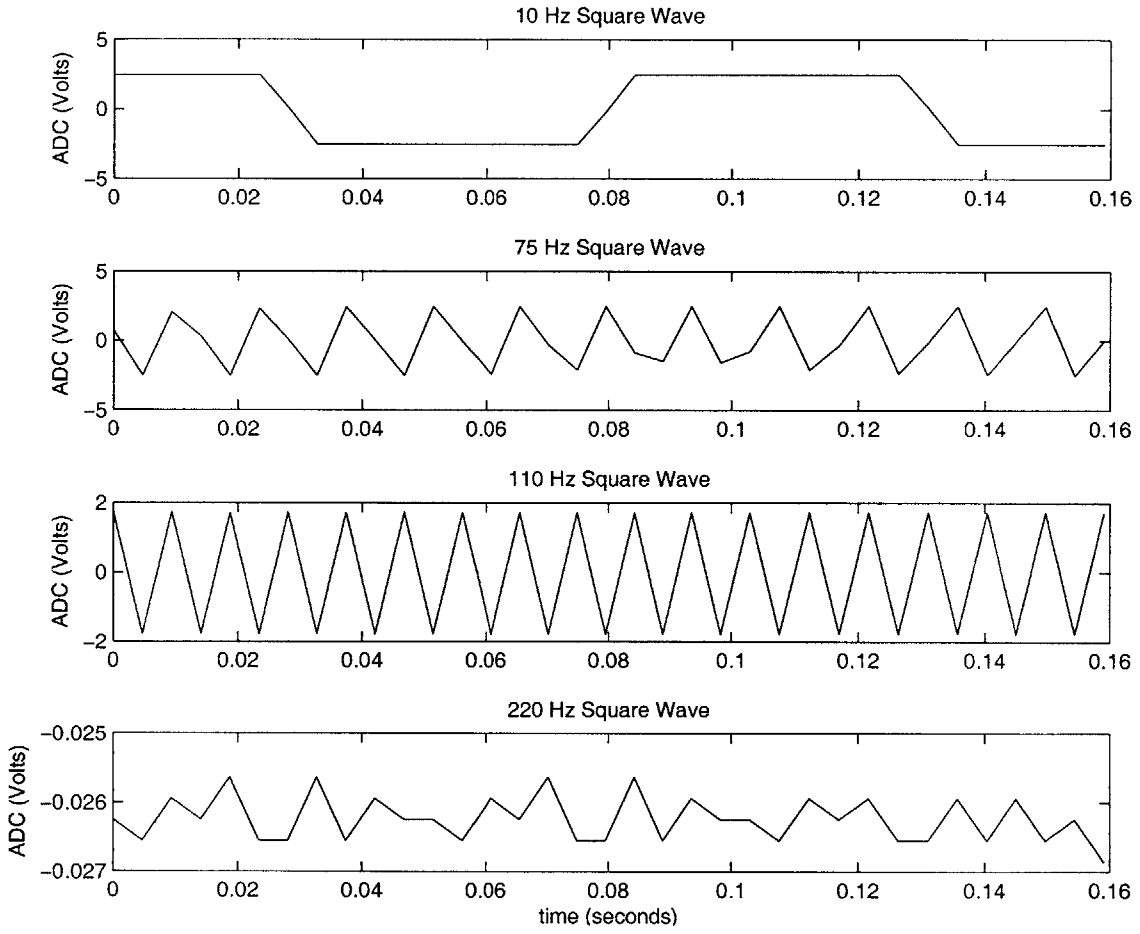


Figure 6-2: Sample 5 V_{PP} square wave captures demonstrating aliasing

means that the poles lie a fair distance away from the real axis in the s-plane, and the circuits used to build these filters are easily tunable. The phase response of a Butterworth filter can cause significant peak overshoot to occur, however, which can reduce its usefulness.

Chebychev and Elliptical

Chebychev filters allow steeper cutoff slopes than the Butterworth filter of a particular order, at the expense of some ripple in the pass-band magnitude response. If more ripple is tolerated, a steeper rolloff is attainable. Unfortunately, this also can translate into worse overshoot and phase-distortion problems.

Like Chebychev filters, elliptical filters have some amount of ripple in the pass-band. Elliptical filters also have ripple in the stop-band. They use a series of notch filters to increase the cutoff slope dramatically. However, the transient response of an elliptical filter is the worst of any of the filters discussed thus far.

In addition the Q values of Chebychev pole-pairs tend to be higher than Butterworth filters, and the Q values of an elliptical filter are higher still. High Q values mean that the circuit implementations of these filters are highly sensitive to component variations, and hence difficult to build. Most filters (especially elliptical) built out of discrete components make use of trimmer capacitors or resistors to tune the extremely high Q circuits. This is impractical in a design that must be as physically small as possible.

Bessel Filters

Bessel filters exhibit the worst cutoff slope of any of the filters described thus far. However, their phase response is almost linear with frequency. Since the time delay of a sinusoid at a particular frequency is equal to the phase shift divided by the frequency, a Bessel filter's transient response only looks like a time delay, with some attenuation at higher frequencies. That is

$$\Delta\phi = \omega\Delta t \rightarrow \Delta t = \frac{\Delta\phi}{\omega}$$

In addition, because the cutoff slope is so shallow, the Q's of the pole pairs tend to be very low, which means the filter can be implemented using fixed 1% resistors and capacitors.

Figure 6-3 shows example Butterworth, Chebychev, elliptical, and Bessel filter frequency responses. Each filter exhibits 96 dB of attenuation by 110 Hz, sufficient

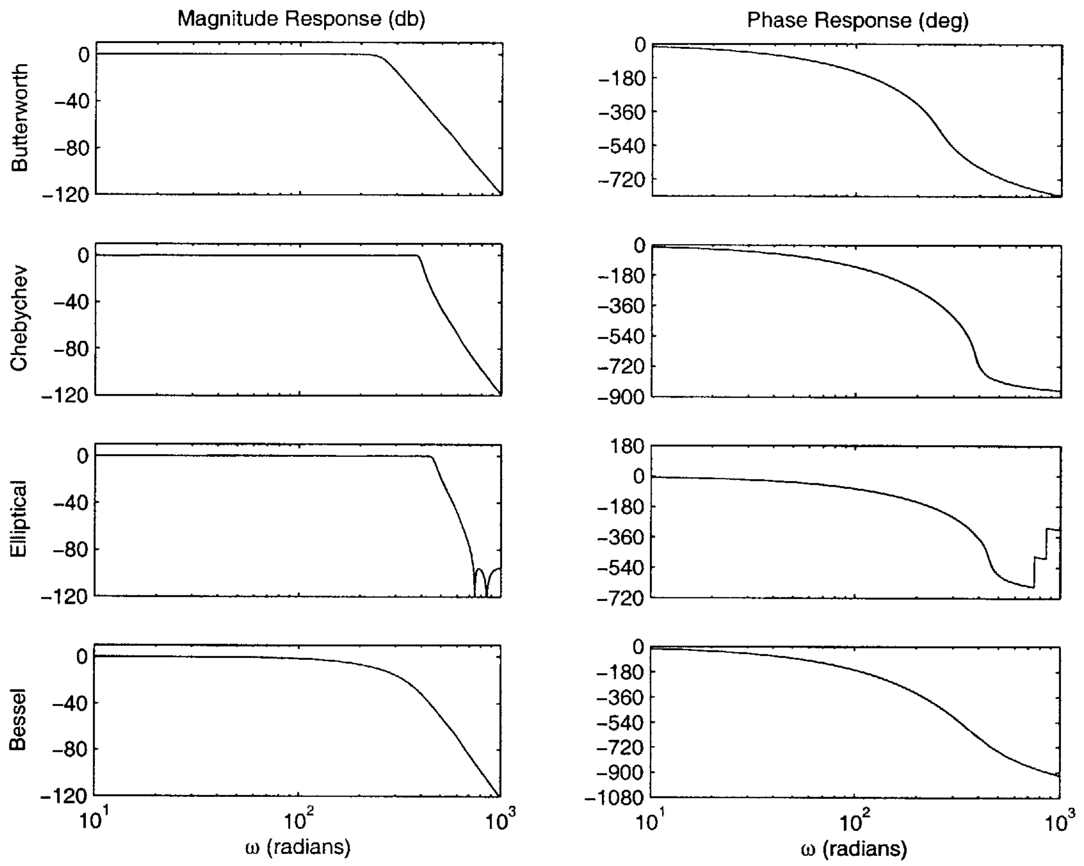


Figure 6-3: Sample anti-aliasing filter frequency responses

for 16-bit accurate conversions. Table 6.1 summarizes the characteristics of each filter. Figure 6-4 shows example transient responses for each of the filters to a 10 Hz input square wave. Notice the severe distortion that the first three filters produce. The Bessel filter's output, by comparison, is only smoothed and shifted in time. In a system where the mode shape of the pipe is of particular interest, the Bessel filter, even with its poor cutoff slope, is the best choice for an anti-aliasing filter.

Table 6.1: Summary of filter characteristics

Filter Type	# Poles	−3 dB (Hz)	−96 dB (Hz)	Passband Ripple (dB)
Butterworth	10	40.0	121.0	*n/a*
Chebychev	10	62.0	125.0	.1
Elliptical	8	72.9	116.7	.1
Bessel	12	21.8	124.7	*n/a*

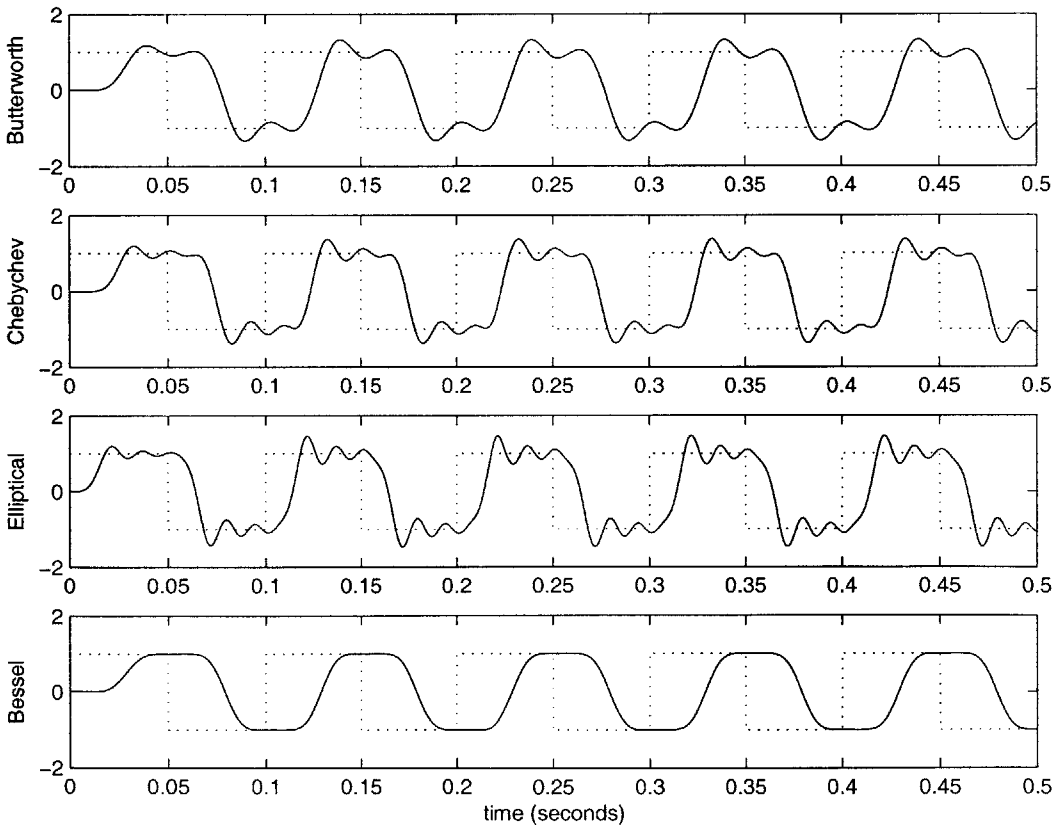


Figure 6-4: Filter transient responses to a 10 Hz square wave

6.2.2 Filter Implementation

There are several ways to implement electronic filters. A passive filter can be built out of inductors, resistors, and capacitors. Another option is using an integrated circuit that utilizes switched-capacitors to give a desired frequency response. Finally, an active filter can be built using op-amps and discrete resistors and capacitors. Each method has advantages and disadvantages.

A strictly passive filter suffers from many problems that make it unsuitable, especially for the higher-order filters required in this system. They require inductors, which are large and difficult to manufacture. They are expensive, and exhibit gains much less than unity.

Switched capacitor filters offer many advantages over passive elements. They offer high-order transfer functions, up to 12 poles, in extremely small packages. Most filters also have a tunable cutoff frequency which is controlled by an external clock running at some multiple of the cutoff frequency. Their major disadvantage is that, like most op amps, they exhibit DC offset voltages of several to tens of millivolts. As described in section 3.2, DC performance is too valuable to this instrumentation system to sacrifice.

Active filter implementations using chopper-stabilized op-amps can provide extremely low DC offset voltages and high-order transfer functions. The major disadvantage is that the number of components required is much larger for a given filter order than in a switched-capacitor design. A popular circuit implementation for a complex pole-pair is the Sallen-Key circuit. Appendix A describes the Sallen-Key architecture and Appendix C is a Matlab script written to help design Sallen-Key circuit implementations of Bessel filters. Figure C-1 shows a 12th order Bessel filter designed using this Matlab script, and implemented on a protoboard.

In future versions of this data acquisition system, it would be beneficial to layout such a filter for both accelerometer channels before the signal reaches the ADCs.

6.3 ADC Selection and Analog Front End

The primary reason for choosing the LTC2440 over the LTC1864 was that the sigma-delta converter had "reduced anti-aliasing filter requirements." The apparent need for an anti-aliasing filter in front of the device suggests that easier to operate LTC1864 should be used instead.

It would be worthwhile to add a programmable analog gain block after the low pass filter, if space allows. This would allow slower experiments to increase their dynamic range, reducing the quantization noise caused by the analog to digital conversion process.

6.4 Board Size and Layout

The current prototype is approximately 0.95" by 3 $\frac{7}{8}$". It has been successfully fit inside of a test section of the composite pipe. It is still a little too large for reasons described in section 3.1.

If a low-pass filter such as the one described in Appendix C is added, layout issues will become even more of a problem. It may be advantageous to find a microcontroller that comes in a smaller package than the 28-SOIC used for the PIC16F876. Additionally, putting major components on both sides of the board could help alleviate many space and width issues. An alternate means of attaching the kevlar strength member must be found so that it does not interfere with any of the components, if both sides of the board are heavily populated.

Chapter 7

Conclusions

The initial system design and testing was successful. There is still a great deal of room for improvement, as detailed in section 6. The VIV research project which motivated this thesis will work with a professional instrument contractor to design of the data acquisition system for this experiment. This document, along with the prototype device, provides a good starting point for further design work on similar VIV data acquisition systems.

Appendix A

Sallen-Key Circuit Analysis

The generalized Sallen-Key circuit is shown in Figure A-1. Z_i are impedance el-

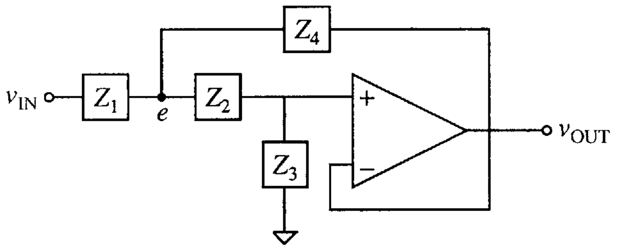


Figure A-1: Generalized Sallen-Key op-amp circuit

ements, with corresponding admittances Y_i. Writing node equations at the node marked e and at the + input of the op-amp yields

$$\begin{aligned}\frac{v_{OUT}-e}{Z_2}+\frac{v_{OUT}}{Z_3} &= 0\\ \frac{v_{IN}-e}{Z_1}+\frac{v_{OUT}-e}{Z_2}+\frac{v_{OUT}-e}{Z_4} &= 0\end{aligned}$$

Collecting like terms gives

$$\begin{aligned}e &= v_{OUT}\left(1+\frac{Y_3}{Y_2}\right)\\ e(Y_1+Y_2+Y_4) &= v_{IN}Y_1+v_{OUT}(Y_2+Y_4)\end{aligned}$$

Substituting the first equation above into the second for e, and solving for $\frac{v_{\mathrm{OUT}}}{v_{\mathrm{IN}}}$ gives

$$\frac{v_{\mathrm{OUT}}}{v_{\mathrm{IN}}} = \frac{1}{1 + \frac{Y_3(Y_1+Y_2+Y_4)}{Y_1 Y_2}}$$

$$\frac{v_{\mathrm{OUT}}}{v_{\mathrm{IN}}} = \frac{1}{1 + \frac{Z_1}{Z_3} + \frac{Z_2}{Z_3} + \frac{Z_1 Z_2}{Z_3 Z_4}}$$

If Z_1 and Z_2 are replaced with resistors R_1 and R_2, and Z_3 and Z_4 are replaced with capacitors C_1 and C_2, the transfer function becomes

$$\frac{v_{\mathrm{OUT}}}{v_{\mathrm{IN}}} = \frac{1}{1 + (R_1 + R_2)C_1 s + R_1 R_2 C_1 C_2 s^2}$$

This is the transfer function of a second order low-pass filter. For small values of s the transfer function has a value of 1. As s increases, the magnitude response decreases, and the phase response moves toward -180°. Through careful choice of component values, the poles of this circuit can be placed at any arbitrary locations. By chaining together multiple Sallen-Key circuits, transfer functions of any arbitrary order can be realized.

Appendix B

Prototype PCB CAD Drawings

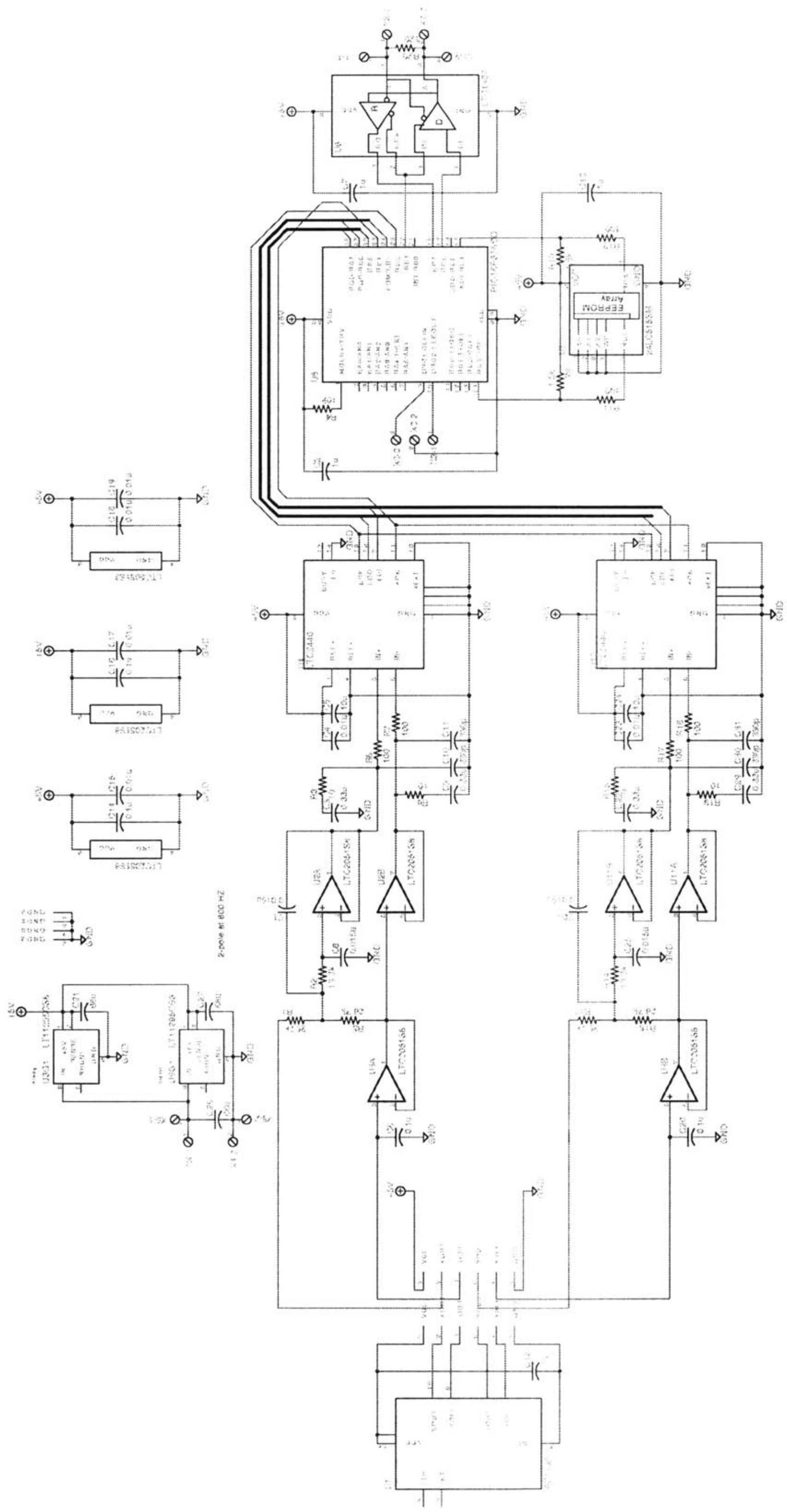


Figure B-1: Prototype schematic

Layer1

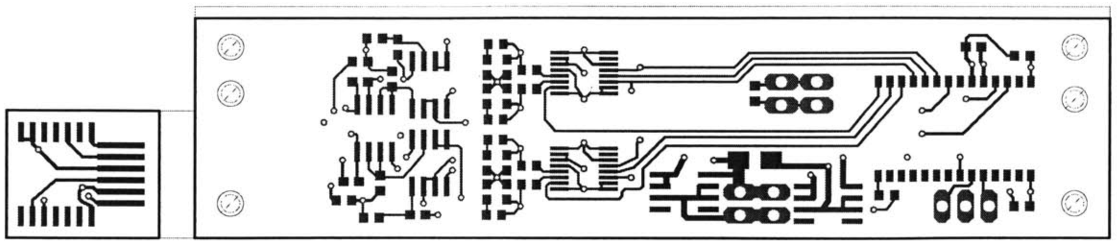


Figure B-2: Top copper layer

Layer 2

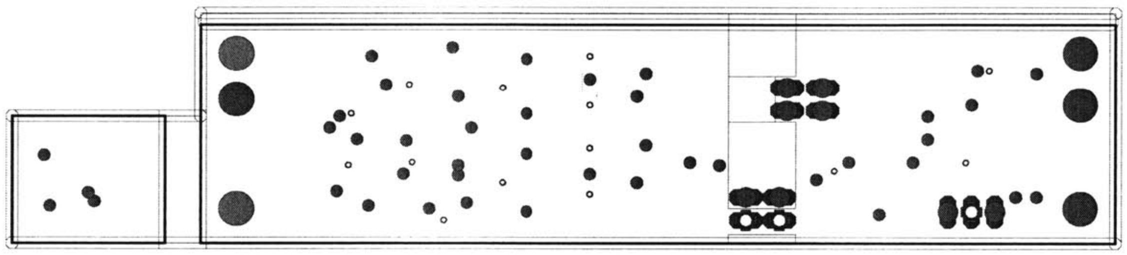


Figure B-3: Ground copper layer

Layer 3

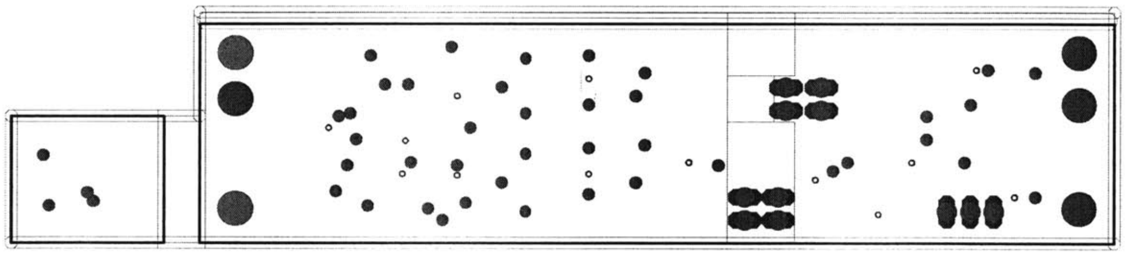


Figure B-4: V_{CC} copper layer

Layer 4

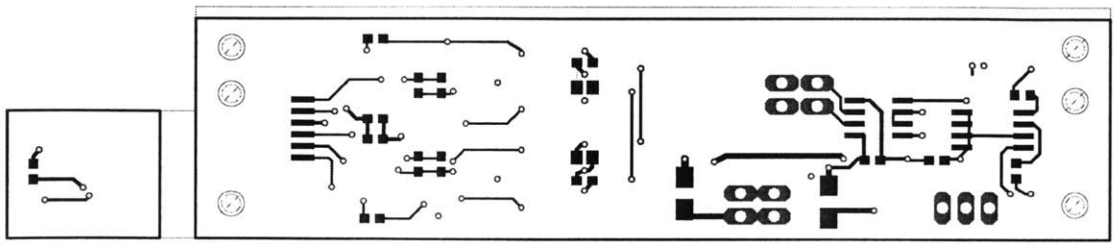


Figure B-5: Bottom copper layer

Appendix C

Bessel Filter Design

Computing the pole locations for a Bessel filter by hand is tedious. Matlab contains a function called `besself` which will calculate the pole locations of an n^{th} order Bessel filter at a given cutoff frequency. The Matlab scripts below find the pole locations for a specified filter and computes the required resistor values for a Sallen-Key implementation. The user can iteratively chose capacitor values until the computed resistor values are acceptable.

C.1 sallenkey.m

```
function [r,Q] = sallenkey(c1,c2,p)
%takes in two capacitor values, and the coeffs for a order 2 poly,
%determins the r1, r2 values for a sallen-key implementation of
%unity gain

%note: p(1) = r1*r2*c1*c2
%note: p(2) = (r1+r2)*c1
Q = sqrt(p(1))/p(2);
prod = p(1)/c1/c2;
sqrt(prod);
sum = p(2)/c1;

%gives 2 values for R1
r = roots([1 -sum prod]);
```

C.2 besselvalues.m

```
% Uses sallenkey.m and the user-defined capacitor values below to
% determine resistor values for a Sallen-Key implementation of a
% bessel filter
format short g
[zeros,poles,gain] = besself(12,50*2*pi); %compute pole locations

%user-defined capacitor values
%pairs for each Sallen-Key circuit
caps = [.33e-6 .82e-6 ...
      .68e-6 .82e-6 ...
      .33e-6 .82e-6 ...
      .33e-6 .82e-6 ...
      .22e-6 .82e-6 ...
      .068e-6 .82e-6 ];

for i=1:2:11,
  r = abs(real(poles(i))); %real part of first pole pair
  im = abs(imag(poles(i))); %imag part of first pole pair
  %compute the characteristic equations for each circuit
  polys((i+1)/2,:) = [1/(r^2+im^2) 2*r/(r^2+im^2) 1];
end

for i=1:6,
  % compute resistor and Q values for each pole pair
  [r,q] = sallenkey(caps(i*2-1),caps(i*2),polys(i,:));
  values(i,:) = [q r(1) r(2) caps(i*2-1) caps(i*2)];
end
```

C.3 Implementation

The example filter, whose capacitor values and cutoff frequency are given in the scripts above, was implemented on a protoboard and tested as a proof-of-concept for later work. The ω_n argument passed to `besself` was $2\pi * 50$, resulting in a filter with a -3 dB point at 21.8 Hz. Figure C-1 is a schematic of the resulting filter, and Figure C-2 is a photograph of the protoboard implementation.

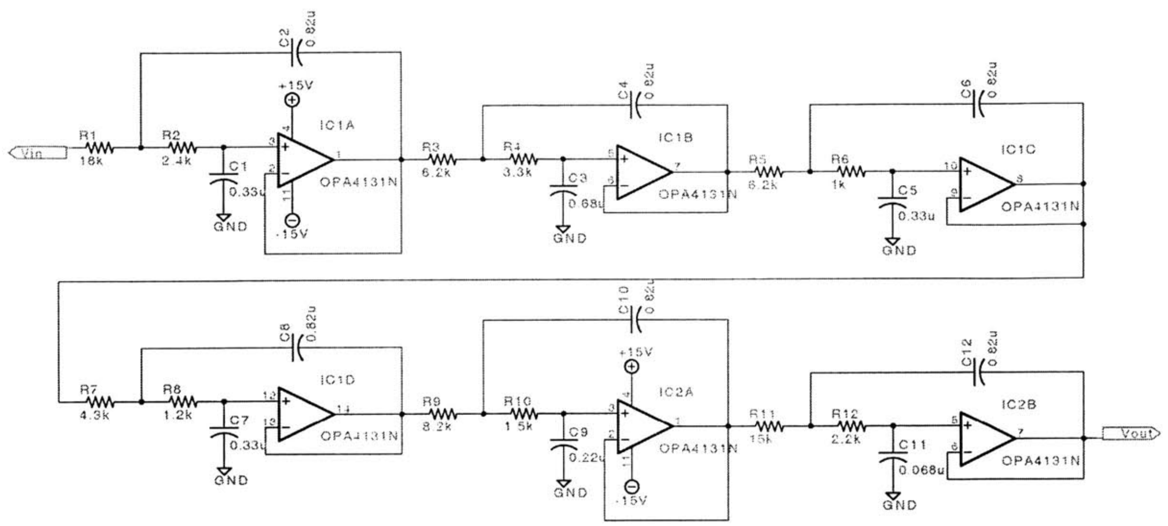


Figure C-1: Schematic: 12th order Bessel low-pass filter

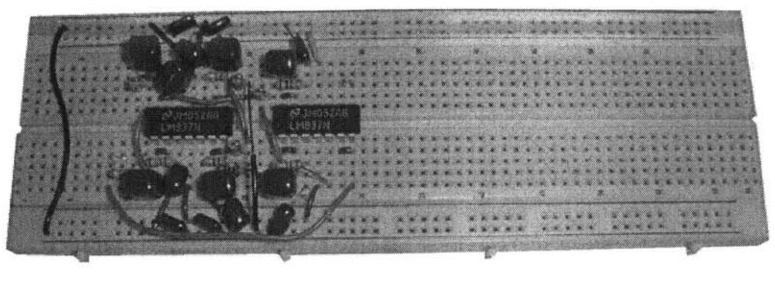


Figure C-2: Prototype 12th order Bessel low-pass filter

Appendix D

PIC Bootloader Assembly Code

```
;;; register definitions
CHKSUM equ 0x71   ; Checksum accumulator
COUNTER equ 0x72 ; General counter
ABTIME equ 0x73
MyAddr equ  0x77
RcvAddr equ  0x78

PCLATH_TEMP equ 0x7D
STATUS_TEMP equ 0x7E
W_TEMP      equ    0x7F

DATA_BUFF  equ     0x10    ; Start of receive buffer
;;; Data mapped in receive buffer
COMMAND     equ    0x10
DATA_COUNT equ  0x11
ADDRESS_L   equ    0x12
ADDRESS_H   equ    0x13
ADDRESS_U   equ    0x14
PACKET_DATA equ 0x15

; *****************************************************************
        ORG    0x0000                    ; Re-map Reset vector
_VRese
        bcf     STATUS,RP0
        bsf     STATUS,RP1
        clrf    PCLATH
        goto    _Setup

        ORG    0x0004
```

```
_VIn
        movwf W_TEMP
        swapf  STATUS,W
        movwf STATUS_TEMP
        clrf     STATUS
        movf    PCLATH,W
        movwf PCLATH_TEMP
        clrf     PCLATH
        goto    _RVInt                 ; Re-map Interrupt vector

; ********************************************************



; ********************************************************
; Setup the appropriate registers.
_Setup
        movlw 0xFF
        movwf EEADR               ; Point to last location
        bsf      STATUS,RP0
        clrf     EECON1
        bsf      EECON1,RD          ; Read the control code
        bcf      STATUS,RP0
        movfw EEDATA
        movwf MyAddr

_UartSetup:
        bcf      STATUS,RP1
        movlw b'10000000'          ; Setup rx and tx, CREN disable
        movwf RCSTA
        bsf      STATUS,RP0
        bcf      TRISB,1                ; Setup T/R pin
        bcf      TRISC,6                ; Setup tx pin
        movlw b'00100110'
        movwf TXSTA
        bsf      STATUS,IRP
; ********************************************************

_Autobaud
;       p = The number of instructions between the first and last
;           rising edge of the RS232 control sequence 0x0F. Other
;           possible control sequences are 0x01, 0x03, 0x07, 0x1F,
;           0x3F, 0x7F.
;
;       SPBRG = (p / 32) - 1 BRGH = 1
```

```
        bcf     STATUS,RP1
        bsf     STATUS,RP0
        movlw   b'00000011'
        movwf   OPTION_REG
        bcf     STATUS,RP0
        bcf     RCSTA,CREN

        call    _WaitForRise

        clrf    TMR0                ; Start counting

        call    _WaitForRise

        movf    TMR0,W              ; Read the timer
        movwf   ABTIME

        bcf     STATUS,C
        rrf     ABTIME,W
        btfss   STATUS,C            ; Rounding
        addlw   0xFF

        bsf     STATUS,RP0
        movwf   SPBRG
        bcf     STATUS,RP0
        bsf     RCSTA,CREN          ; Enable receive

        movf    RCREG,W
        movf    RCREG,W

        bsf     STATUS,RP0
        movlw   b'11111111'
        movwf   OPTION_REG
; ***********************************************************

                ;;      movfw   MyAddr
                ;;      call    WrRS485

; ***********************************************************
; Read and parse the data.
_StartOfLine
        bcf     STATUS,RP0
        bcf     STATUS,RP1
        call    _RdRS485                    ; Look for a start of line
        xorlw   STX                 ; <STX><STX>
        btfss  STATUS,Z
        goto   _Autobaud           ;was StartOfline
        movlw  DATA_BUFF           ; Point to the buffer
        movwf  FSR
        call   _RdRS485                     ; get address
        movwf  RcvAddr
        movwf  CHKSUM              ; include in chksum calc
_GetNextDat
        call   _RdRS485                     ; Get the data
        xorlw  STX                 ; Check for a STX
        btfsc  STATUS,Z
        goto   _StartOfLine        ; Yes, start over

_NoSTX movf RXDATA,W
        xorlw  ETX                 ; Check for a ETX
        btfsc  STATUS,Z
        goto   _CheckSum           ; Yes, examine checksum

_NoETX movf RXDATA,W
        xorlw  DLE                 ; Check for a DLE
        btfss  STATUS,Z
        goto   _NoDLE              ; Check for a DLE
        call   _RdRS485                     ; Yes, Get the next byte

_NoDLE movf RXDATA,W
        movwf  INDF                ; Store the data
        addwf  CHKSUM,F            ; Get sum
        incf   FSR,F

        goto   _GetNextDat

_CheckSum
        movf   CHKSUM,F            ; Checksum test
        btfss  STATUS,Z
        goto   _Autobaud
; ************************************************************

; ************************************************************
;;; Pre-setup, common to all commands
        movfw  MyAddr
        xorwf  RcvAddr,W
        btfss  STATUS,Z      ; set?  address match
        goto   _Autobaud     ; wait for next packet
        bsf    STATUS,RP1
        movf   ADDRESS_L,W ; Setup pointers for addresses
        movwf EEADR
        movf  ADDRESS_H,W
        movwf EEADRH
        movlw PACKET_DATA
        movwf FSR
        movf  DATA_COUNT,W ; setup COUNTER
        movwf COUNTER
        btfsc STATUS,Z
        goto  _VReset         ; count is zero, error (Special Command)
; ****************************************************************

; ****************************************************************
_CheckCommand
        movf  COMMAND,W         ; Test for a valid command
        sublw d'5'
        btfss STATUS,C
        goto  _Autobaud

        movf  COMMAND,W         ; Perform calculated jump
        addwf PCL,F

        goto  _ReadVersion      ; 0
        goto  _ReadProgMem      ; 1
        goto  _WriteProgMem     ; 2
        goto  _ReadEE           ; 3
        goto  _WriteEE                  ; 4
        goto  _DoRVReset                ; 5
; ****************************************************************

_DoRVReset:
        clrf  STATUS                                      ;B2->B0
        goto  _RVRese


; ****************************************************************
;;; Commands
;;;
_ReadVersion
        movlw BOOT_MINOR_VERSION
        movwf DATA_BUFF + 2
        movlw BOOT_MAJOR_VERSION
        movwf DATA_BUFF + 3

        movlw 0x04
        goto  _WritePacket
```

```
_ReadProgMem
_RPM1 bsf    STATUS,RP0
      bsf    EECON1,EEPGD
      bsf    EECON1,RD
      nop
      nop
      bcf    STATUS,RP0
      movf   EEDATA,W
      movwf  INDF
      incf   FSR,F
      movf   EEDATH,W
      movwf  INDF
      incf   FSR,F

      incf   EEADR,F
      btfsc  STATUS,Z
      incf   EEADRH,F

      decfsz COUNTER,F
      goto   _RPM1             ; Not finished then repeat

      rlf    DATA_COUNT,W      ; Setup packet length
      addlw  0x05

      goto   _WritePacket

_WriteProgMem
      bsf    STATUS,RP0
      movlw  b'10000100'       ; Setup writes
      movwf  EECON1
      bcf    STATUS,RP0
      movlw  b'11111100'       ; Force a boundry
      andwf  EEADR,F

      movlw  0x04
      movwf  TEMP

_Lp1
      movf   INDF,W
      movwf  EEDATA
      incf   FSR,F
      movf   INDF,W
      movwf  EEDATH
      incf   FSR,F
```

```
        call    _StartWrite

        incf    EEADR,F
        btfsc   STATUS,Z
        incf    EEADRH,F

        decfsz  TEMP,F
        goto    _Lp1

        decfsz  COUNTER,F
        goto    _WriteProgMem          ; Not finished then repeat

        goto    _SendAcknowledge                ; Send acknowledge


_ReadEE
        bsf     STATUS,RP0
        clrf    EECON1

        bsf     EECON1,RD              ; Read the data
        bcf     STATUS,RP0
        movf    EEDATA,W
        movwf   INDF
        incf    FSR,F

        incf    EEADR,F                ; Adjust EEDATA pointer

        decfsz  COUNTER,F
        goto    _ReadEE                ; Not finished then repeat

        movf    DATA_COUNT,W           ; Setup packet length
        addlw   0x05

        goto    _WritePacket

_WriteEE
        movf    INDF,W
        movwf   EEDATA
        incf    FSR,F
        call    _WriteWaitEEData                ; Write data

        incf    EEADR,F                ; Adjust EEDATA pointer

        decfsz  COUNTER,F
        goto    _WriteEE                        ; Not finished then repeat
        goto    _SendAcknowledge             ; Send acknowledge
; ********************************************************


; ********************************************************
; Send the data buffer back.
;
; <STX><STX>[<DATA>...]<CHKSUM><ETX>

_SendAcknowledge
        movlw   0x01                    ; Send acknowledge

_WritePacke
        movwf   COUNTER

        bsf     PORTB,1
        call    _RS485Delay
        movlw   STX                     ; Send start condition
        call    _WrRS485
        call    _WrRS485
        movfw   MyAddr
        call    _WrRS485                        ; master address == 0
        clrf    CHKSUM                  ; Reset checksum

        movlw   DATA_BUFF               ; Setup pointer to buffer area
        movwf   FSR

_SendNext                               ; Send DATA
        movf    INDF,W
        addwf   CHKSUM,F
        incf    FSR,F
        call    _WrData
        decfsz  COUNTER,F
        goto    _SendNext

        comf    CHKSUM,W                ; Send checksum
        addlw   0x01
        call    _WrData

        movlw   ETX                     ; Send stop condition
        call    _WrRS485
        bcf     PORTB,1
        goto    _Autobaud
; ********************************************************
; ***********************************************************
; Write a byte to the serial port.

_WrData
        movwf TXDATA            ; Save the data

        xorlw  STX              ; Check for a STX
        btfsc  STATUS,Z
        goto   _WrDLE           ; No, continue WrNext

        movf   TXDATA,W
        xorlw  ETX              ; Check for a ETX
        btfsc  STATUS,Z
        goto   _WrDLE           ; No, continue WrNext

        movf   TXDATA,W
        xorlw  DLE              ; Check for a DLE
        btfss  STATUS,Z
        goto   _WrNext          ; No, continue WrNext

_WrDLE
        movlw  DLE              ; Yes, send DLE first
        call   _WrRS485

_WrNex
        movf   TXDATA,W         ; Then send STX

; ***********************************************************
_WrRS485:
        bcf    STATUS,RP1
        movwf  TXREG
        bsf            STATUS,RP0
        btfss  TXSTA,TRMT ; 1 if byte transmitted
        goto   $ - 1
        bcf    STATUS,RP0
        return

_RS485Delay:
        ;; 1 bit long delay, to allow for change-over
        ;; time between transmit/receive (43 cycles)
        movlw 0xD        ;1
        movwf TEMP ;1
_RS485DelayLoop:
        decfsz TEMP,F ; 1, 2
        goto    _RS485DelayLoop ; 2 cycles ;2+3*(x-1)+2+4 plenty
        return

; *******************************************************

; *******************************************************
_RdRS485:
        btfsc   RCSTA,OERR ; Reset on overrun
        goto    _VRese
        btfss   PIR1,RCIF     ; Wait for RS485 Data
        goto    $ - 1
        movf    RCREG,W
        movwf   RXDATA
        return
; *******************************************************


; *******************************************************
_WaitForRise
        btfsc   PORTC,7              ; Wait for a falling edge
        goto    _WaitForRise
_WtSR   btfss   PORTC,7              ; Wait for starting edge
        goto    _WtSR
        return
; *******************************************************


; *******************************************************
; Unlock and start the write or erase sequence.

_StartWrite
        bsf     STATUS,RP0
        movlw   0x55                 ; Unlock
        movwf   EECON2
        movlw   0xAA
        movwf   EECON2
        bsf     EECON1,WR            ; Start the write
        nop
        nop
        bcf     STATUS,RP0
        return
; *******************************************************
; ***********************************************************
_WriteWaitEEData
        bsf     STATUS,RP0
        movlw   b'00000100'        ; Setup for EEData
        movwf   EECON1
        call    _StartWrite
        bsf             STATUS,RP0
        btfsc   EECON1,WR          ; Write and wait
        goto    $ - 1
        bcf     STATUS,RP0
        return
; ***********************************************************


; ***********************************************************
        ORG     0x200
_RVReset

        ORG     0x204
_RVIn

; ***********************************************************
```

Appendix E

PIC User Firmware Assembly Code

```
        LIST   P=pic16f876
        include p16f876.inc

;OUT
#define RS485TXEN   PORTB,1
; IN
#define P_SDI0      PORTB,3
#define P_SDI1      PORTB,6
;OUT
#define P_SDO0      PORTB,7
#define P_SDO1      PORTB,5

; OUTS
#define P_SCLK      PORTB,2
#define P_ADCS      PORTB,4

#define NUMCOMMANDS 18

#define MAJOR_VERSION 0xDE
#define MINOR_VERSION 0xAD


        include bootloaderdef.h

        cblock  0x20
;;; NOTE:      These are in register bank 0
;;; DoubleNote: Register banks 2 and 3 are reserved for incoming/outgoing
;;;             packets
MemConfig
MemAddrH
MemAddrL
Sample0_L
Sample0_H
Sample1_L
Sample1_H

OSR

NumSamp_L
NumSamp_H
NumSamp_U

SampRate_L
SampRate_H
FLAGS

MemLen
MemCounter
I2CState
MemPtr

MEMORY_BUFF
            endc

CHKSUM equu 0x71  ; Checksum accumulator
COUNTER equ 0x72 ; General counter
MyAddr equ  0x77
RcvAddr equ  0x78

;;; The following start in bank 2
DATA_BUFF  equ    0x10   ; Start of receive buffer
;;; Data mapped in receive buffer
COMMAND    equ    0x10
DATA_COUNT equ  0x11
ADDRESS_L  equ    0x12
ADDRESS_H  equ    0x13
ADDRESS_U  equ    0x14
PACKET_DATA equ 0x15

PCLATH_TEMP equ 0x7D
STATUS_TEMP equ 0x7E
```

```
W_TEMP      equ     0x7F

#define COOKIE_BIT FLAGS,0
#define CALL_WRITE_PACKET FLAGS,1
#define DISCARD_SAMPLE FLAGS,2
#define EOC_ERROR FLAGS,3

        include bootloaderstub.h


RVReset:
        org     0x200   ; Remapped Reset Vector
        goto    start
        org     0x204   ; Remapped Interrupt Vector
int:
        bsf     STATUS,RP0
        btfss   PIE1,TMR1IE
        goto    int_return
        bcf     STATUS,RP0
        btfss   PIR1,TMR1IF
        goto    int_return
tmr1int:
        bcf     T1CON,TMR1ON
        bcf     PIR1,TMR1IF
        movfw   SampRate_H
        movwf   TMR1H
        movfw   SampRate_L
        movwf   TMR1L
        bsf     T1CON,TMR1ON        ; enable timer
int_return:
        clrf    STATUS
        movfw   PCLATH_TEMP
        movwf   PCLATH
        movfw   STATUS_TEMP
        movwf   STATUS
        swapf   W_TEMP,F
        swapf   W_TEMP,W
        retfie


; *****************************************************
;;; Takes the next bit of the OSR, presents it on P_SDO0/1,
;;; saves it in OSRSAVE
PRESENT_OSR MACRO
```

```
        bcf     P_SDO0
        bcf     P_SDO1
        rlf     OSR,F
        btfsc   STATUS,C
        bsf     P_SDO0
        btfsc   STATUS,C
        bsf     P_SDO1
        ENDM

INC_BUF_POINTER MACRO
        incf    FSR,F
        movlw   0x70
        xorwf   FSR,W          ; Check if reached end of bank
        movlw   0xA0           ; bank 3 start
        btfsc   STATUS,Z
        movwf   FSR            ; end bank 2, move to bank 3
        ENDM
;;; No Wait Needed
I2CStartBit     MACRO XLabel          ; MUST BE CALLED FROM B0 or B1
        bsf     STATUS,R
        bsf     SSPCON2,SEN ; do the start
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,SEN
        goto    $ - 7
        bcf     STATUS,R
        ENDM


;;; No Wait Needed — check I2C docs
I2CReStartBit MACRO XLabel
        BSF     STATUS,RP0
        BSF     SSPCON2,RSEN
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,RSEN
        goto    $ - 7
        bcf     STATUS,RP0
        ENDM

;;; No Wait Needed -- check I2C docs
I2CStopBit      MACRO XLabel
        BSF     STATUS,RP0
        BSF     SSPCON2,PEN
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,PEN
        goto    $ - 7
        BCF     STATUS,RP0
        ENDM


I2CWriteW       MACRO XLabel
        MOVWF SSPBUF        ; initiate write cycle
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,RP0
        btfsc   SSPSTAT,R_W
        goto    $ - 7
        bcf     STATUS,RP0
        ENDM

I2CNoAck        MACRO XLabel
        BSF     STATUS,RP0
        BSF     SSPCON2,ACKD
        BSF     SSPCON2,ACKEN
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,ACKEN
        goto    $ - 7
        BCF     STATUS,RP0
        ENDM

;;; No Wait Needed — check I2C docs
I2CAck          MACRO XLabel
        BSF     STATUS,RP0
        BCF     SSPCON2,ACKD
        BSF     SSPCON2,ACKEN
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,ACKEN
        goto    $ - 7
        BCF     STATUS,RP0
        ENDM

;;; No Wait Needed — check I2C docs
I2CStartRead MACRO XLabel
        BSF     STATUS,RP0
        BSF     SSPCON2,RCEN
        bcf     STATUS,RP0
        btfsc   SSPCON,WCOL
        goto    XLabel
        btfsc   PIR2,BCLIF
        goto    XLabel
        bsf     STATUS,R
        btfsc   SSPCON2,RCEN
        goto    $ - 7
        BCF     STATUS,RP0
        ENDM

I2CTestAck      MACRO XLabel
        bsf     STATUS,RP0
        btfsc   SSPCON2,ACKSTA
        goto    XLabel
        bcf     STATUS,RP0
        ENDM

TransmitEnable MACRO
        bsf     RS485TXEN
        call    RS485Delay
        ENDM
```

```
TransmitDisable MACRO
        bsf     STATUS,RP0
        btfss   TXSTA,TRMT ; 1 if byte transmitted
        goto    $ - 1
        bcf     STATUS,RP0
        bcf     RS485TXEN
        ENDM

start:
        bsf     STATUS,RP0
        movlw   b'01001001'
        movwf   TRISB           ; set outputs
        movlw   d'12'
        movwf   SSPADD          ; 384kHz I2C bus
        clrf    SSPSTAT         ; slew rate control, I2C levels
        clrf    SSPCON2         ; reset all flags
        bcf     STATUS,RP0
        movlw   b'00101000'     ; enable port, enable master clock bgen
        movwf   SSPCON
        bcf     P_SCLK          ; idle state
        bsf     P_ADCS          ; idle state
;;; The following defaults produce a 220Hz sampling rate
        movlw   b'00101000'
        movwf   OSR             ; OSR = 220Hz
        clrf    T1CON
        movlw   0xA7
        movwf   SampRate_H
        movlw   0x22
        movwf   SampRate_L
        clrf    FLAGS
        goto    SendAcknowledge

recv_packet:
        bcf     STATUS,RP0
        bcf     STATUS,RP1
        call    RdRS485         ; Look for a start of line
        xorlw   STX             ; <STX><STX>
        btfss   STATUS,Z
        goto    recv_packet     ; was StartOfline
        call    RdRS485         ; Look for a start of line
        xorlw   STX             ; <STX><STX>
        btfss   STATUS,Z
        goto    recv_packet     ; was StartOfline
        bsf     STATUS,IRP      ; INDF is bank 2/3 now
        movlw   DATA_BUFF       ; Point to the buffer
        movwf FSR
        call    RdRS485         ; get address
        movwf RcvAddr
        movwf CHKSUM           ; include in chksum calc
GetNextDat
        call    RdRS485         ; Get the data
        xorlw   STX             ; Check for a STX
        btfsc   STATUS,Z
        goto    recv_packet     ; Yes, start over

NoSTX movf  RXDATA,W
        xorlw   ETX             ; Check for a ETX
        btfsc   STATUS,Z
        goto    CheckSum        ; Yes, examine checksum

NoETX movf  RXDATA,W
        xorlw   DLE             ; Check for a DLE
        btfss   STATUS,Z
        goto    NoDLE           ; Check for a DLE
        call    RdRS485         ; Yes, Get the next byte

NoDLE movf  RXDATA,W
        movwf INDF             ; Store the data
        addwf CHKSUM,F         ; Get sum
        INC_BUF_POINTER ; trashes w
        goto    GetNextDat

CheckSum
        movf    CHKSUM,F        ; Checksum test
        btfss   STATUS,Z
        goto    recv_packet
; *******************************************************

; *******************************************************
;;; Pre-setup, common to all commands
        movfw MyAddr
        xorwf RcvAddr,W
        btfss   STATUS,Z        ; set?  address match
        goto    recv_packet     ; wait for next packet
        bsf     STATUS,RP1
        movf    ADDRESS_L,W ; Setup pointers for addresses
        movwf EEADR             ;
        movf    ADDRESS_H,W
        movwf EEADRH            ;
        movlw PACKET_DATA ;
        movwf FSR
        movf  DATA_COUNT,W ; setup COUNTER
        movwf COUNTER
        btfsc STATUS,Z
        goto  _VReset ; count is zero, error (Special Command)
; **********************************************************

; **********************************************************
;;; ALWAYS check that the jump table below doesn't cross a 256-byte page

CheckCommand
        movlw HIGH $
        movwf PCLATH       ; Preserve memory moundary...
        movf  COMMAND,W         ; Test for a valid command
        sublw NUMCOMMANDS
        btfss STATUS,C
        goto  recv_packet

        movf  COMMAND,W         ; Perform calculated jump
        addwf PCL,F

        goto  GetVersion     ; 0
        ;; To overwrite the bootloader, modify this file with new
        ;; code that contains ReadProgMem/WriteProgMem
        ;; load that user code, write the bootloader area
        ;; then re-load old usercode
        ;; hopefully that's pretty fail-safe
;       goto  ReadProgMem       ; 1
        goto  UnImplemented ; 1
;       goto  WriteProgMem      ; 2
        goto  UnImplemented ; 2
;       goto  ReadEE            ; 3
        goto  UnImplemented ; 3
;       goto  WriteEE           ; 4
        goto  UnImplemented ; 4
;       goto  DoRVReset         ; 5
        goto  RVReset        ; 5

        goto  SetNumSamples ; 6
        goto  GetNumSamples ; 7
        goto  SetSampleRate ; 8
        goto  GetSampleRate ; 9
        goto  SetSampleCookie ; 10 0xA
        goto  GetSampleCookie ; 11 0xB
        goto  EnterSampleMode ; 12 0xC
        goto    StartMemoryPage ; 13 0xD
        goto    GetMemoryPage ; 14
        goto    TakeSendSample ; 15
        goto    GetOSR          ; 16
        goto    SetOSR          ; 17
        goto    GetEOC          ; 18



;;; Commands
        ;; Set/Get Number Of Samples To Take (3 bytes) (sample = 4 bytes)
        ;; Set/Get Sample Rate (2 bytes in form of timer 1 reset value)
        ;; Set/Get Sample Cookie
        ;; Enter Sample Mode (5 STX bytes)
        ;;
        ;; Start Read of Memory Page
        ;; Send Memory Page
        ;; Take a sample



; ******************************************************
;;; Commands
;;;
GetVersion
        movlw   MINOR_VERSION
        movwf   DATA_BUFF + 2
        movlw   MAJOR_VERSION
        movwf   DATA_BUFF + 3

        movlw   0x04
        goto    WritePacket

SetNumSamples:
        bcf     STATUS,RP1

        movfw   INDF            ; starts at PACKET_DATA
        movwf   NumSamp_L

        incf    FSR,F
        movfw   INDF
        movwf   NumSamp_H

        incf    FSR,F
        movfw   INDF
        movwf NumSamp_U
        goto  SendAcknowledge

GetNumSamples:
        bcf   STATUS,RP1

        movfw NumSamp_L
        movwf INDF          ; starts at PACKET_DATA

        incf  FSR,F
        movfw NumSamp_H
        movwf INDF

        incf  FSR,F
        movfw NumSamp_U
        movwf INDF
        movlw 0x8
        goto  WritePacket

SetOSR:
        bcf   STATUS,RP1
        movfw INDF          ; starts at PACKET_DATA
        movwf OSR
        goto  SendAcknowledge

GetOSR:
        bcf   STATUS,RP1
        movfw OSR           ; starts at PACKET_DATA
        movwf INDF
        movwf 0x6
        goto  WritePacket

GetOSR:
        bcf   STATUS,RP1
        movlw 0x0
        btfsc EOC_ERROR
        movlw 0xAA

        movwf INDF
        movwf 0x6
        goto  WritePacket


SetSampleRate:
        bcf   STATUS,RP1
        movfw INDF          ; starts at PACKET_DATA
        movwf SampRate_L

        incf  FSR,F
        movfw INDF
        movwf SampRate_H

        goto  SendAcknowledge


GetSampleRate:
        bcf   STATUS,RP1

        movfw SampRate_L
        movwf INDF          ; starts at PACKET_DATA

        incf  FSR,F
        movfw SampRate_H
        movwf INDF

        movlw 0x7
        goto  WritePacket


SetSampleCookie:
        bcf   STATUS,RP1
        bcf   COOKIE_BIT
        movlw 0xAA
        xorwf INDF,W
        btfsc STATUS,Z
        bsf   COOKIE_BIT
        goto  SendAcknowledge


GetSampleCookie:
        bcf   STATUS,RP1
        movlw 0
        btfsc COOKIE_BIT
        movlw 0x1
        movwf INDF          ; starts at PACKET_DATA
        movlw 0x6
        goto  WritePacket

StartMemoryPage:
        bcf     STATUS,RP1
        ;; get the address bytes and len out first
        ;; note: address is 16 bytes long.  bit 16 is used as bit 2
        ;; (of 3) in the configuration word chip select
        movlw DATA_COUNT ; Point to DLEN field

        movwf FSR
        movfw INDF
        movwf MemLen

        incf    FSR,F
        movfw INDF          ; AddrL
        movwf MemAddrL

        incf    FSR,F
        movfw INDF          ; AddrH
        movwf MemAddrH
        ;; send acknowledge so bus not held
        bsf     CALL_WRITE_PACKET
        call    SendAcknowledge
        ;; Make the configureation word
        movlw b'00001010'   ; see below
        movwf MemConfig
        bcf     STATUS,C
        rlf     MemAddrH,F  ; addr<15> now in carry
        rlf     MemConfig,F ; C=0,MemConfig=0001010a<15>
        rrf     MemAddrH,F  ; reset, C=0
        rlf     MemConfig,F ; C=0,MemConfig=001010a<15>0
        rlf     MemConfig,F ; C=0,MemConfig=01010a<15>00
        rlf     MemConfig,F ; C=0,MemConfig=1010a<15>000

InitI2CReadPage:
        bcf     STATUS,RP0
        bcf     PIR2,BCLIF
        bcf     SSPCON,WCOL
        movfw MemLen
        movwf MemCounter
        ;; MemLen field will be length in bytes to read
        ;; addrh/addrl is address (addru unused)
        ;; places data starting at MEMORY_BUFF in banks 0 and 1
        bcf     STATUS,IRP
        movlw MEMORY_BUFF ; beginning address of data
        movwf FSR
        ;; StartCond
        I2CStartBit           InitI2CReadPage
        ;; Write MemConfig w/ack
        movfw  MemConfig
        I2CWriteW           InitI2CReadPage
        I2CTestAck          InitI2CReadPage
        ;; Write MemAddrH w/ack
        movfw  MemAddrH
        I2CWriteW           InitI2CReadPage
        I2CTestAck          InitI2CReadPage
        ;; Write MemAddrL w/ack
        movfw  MemAddrL
        I2CWriteW           InitI2CReadPage
        I2CTestAck          InitI2CReadPage
        ;; RepeatedStartCond
        I2CReStartBit       InitI2CReadPage
        ;; Read MemLen-1 bytes w/ack
        bsf    MemConfig,0
        movfw  MemConfig
        I2CWriteW           InitI2CReadPage
        I2CTestAck          InitI2CReadPage
        goto   MemReadLoopStar
MemReadLoop:
        I2CAck        InitI2CReadPage
MemReadLoopStart:
        I2CStartRead        InitI2CReadPage
        movfw  SSPBUF
        movwf  INDF
        INC_BUF_POINTER
        decfsz MemCounter,F
        goto   MemReadLoop
        ;; Read 1 byte w/ noack
        I2CNoAck            InitI2CReadPage
        I2CStopBit          InitI2CReadPage
        goto   recv_packe


GetMemoryPage:
        bcf    STATUS,RP1
        TransmitEnable
        movlw  STX
        call   WrRS485
        call   WrRS485
        movfw  MyAddr
        call   WrRS485

        movlw  0xD          ; Command
        movwf CHKSUM
        call   WrData

        movfw MemLen         ; DataLen
        movwf COUNTER
        addwf  CHKSUM,F
        call   WrData

        movfw MemAddrL
        addwf  CHKSUM,F
        call   WrData

        movfw MemAddrH
        addwf  CHKSUM,F
        call   WrData

        movlw 0              ; AddrU
        call   WrData

        bcf    STATUS,IRP
        movlw MEMORY_BUFF
        movwf FSR
        goto   SendNext

EnterSampleMode:
        ;; First send acknowledge
        bcf    STATUS,RP1
        bsf    CALL_WRITE_PACKET
        call   SendAcknowledge
        ;; Wait for 5 STXs in a row
SampleModeInitVars:
        clrf   MemAddrH
        clrf   MemAddrL
        movlw b'10100000'
        movwf MemConfig
        bsf    DISCARD_SAMPLE
        bcf    EOC_ERROR
EnterSampleModeWaitSTXinit:
        movlw 0x5
        movwf COUNTER
EnterSampleModeWaitSTX:
        call   RdRS485
        xorlw  STX
        btfss  STATUS,Z      ; is STX?
        goto   EnterSampleModeWaitSTXinit
        decfsz  COUNTER,F
        goto    EnterSampleModeWaitSTX
        ;; received 5 STXs in a row
        btfss   COOKIE_BIT
        goto    SampleModeSlaveLoop
SampleModeMasterLoopInit:
        TransmitEnable          ; Enable Transmit
        bcf     T1CON,TMR1ON ; Reset the timer first time
        bcf     PIR1,TMR1IF
        movfw SampRate_H
        movwf TMR1H
        movfw SampRate_L
        movwf TMR1L
        bsf     T1CON,TMR1ON ; Start timer
SampleModeMasterLoop:
        btfss   PIR1,TMR1IF ; timer expire wait
        goto    SampleModeMasterLoop
        bcf     T1CON,TMR1ON ; reset timer
        bcf     PIR1,TMR1IF
        movfw SampRate_H
        movwf TMR1H
        movfw SampRate_L
        movwf TMR1L
        bsf     T1CON,TMR1ON
        movlw DLE               ; sample trigger byte
        ;; don't need to wait for TX to be free, as the timer
        ;; delay ensures this for reasonable serial speeds
        call    WrRS485
        call    TakeSample      ; take the sample
        btfsc   DISCARD_SAMPLE
        goto    SampleModeMasterLoopClearDiscard
        call    StoreSample     ; Store the sample
        decf    NumSamp_L,F ; dec low
        comf    NumSamp_L,W ; see if rollover
        btfss   STATUS,Z
        goto    SampleModeMasterLoop ; no rollover, continue
        decf    NumSamp_H,F ; dec high
        comf    NumSamp_H,W
        btfss   STATUS,Z
        goto    SampleModeMasterLoop ; no rollover, continue
        decf    NumSamp_U,F ; dec upper
        comf    NumSamp_U,W
        btfss   STATUS,Z
        goto    SampleModeMasterLoop ; no rollover, continue
        TransmitDisable         ; stop
        goto    recv_packet     ; return
SampleModeMasterLoopClearDiscard:
        bcf     DISCARD_SAMPLE
        goto    SampleModeMasterLoop

SampleModeSlaveLoop:
        ;; Wait for the falling edge of the start bit
        bcf     STATUS,RP0
        bcf     STATUS,RP1
        btfss   PORTC,7                 ; Wait for a positive value
        goto    $ - 1
_WtSR btfsc     PORTC,7                 ; Wait for falling edge
        goto    $ - 1
        ;; sample byte began transmitting, continue
        call    TakeSample      ; take the sample
        btfsc   DISCARD_SAMPLE
        goto    SampleModeSlaveLoopClearDiscard
        call    StoreSample     ; Store the sample
        decf    NumSamp_L,F ; dec low
        comf    NumSamp_L,W ; see if rollover
        btfss   STATUS,Z
        goto    SampleModeSlaveLoop ; no rollover, continue
        decf    NumSamp_H,F ; dec high
        comf    NumSamp_H,W
        btfss   STATUS,Z
        goto    SampleModeSlaveLoop ; no rollover, continue
        decf    NumSamp_U,F ; dec upper
        comf    NumSamp_U,W
        btfss   STATUS,Z
        goto    SampleModeSlaveLoop ; no rollover, continue
        goto    recv_packet     ; return
SampleModeSlaveLoopClearDiscard:
        bcf     DISCARD_SAMPLE
        goto    SampleModeSlaveLoop

UnImplemented:
        movlw   0xFF
        movwf   DATA_BUFF
        movlw   0xC
        movwf   DATA_BUFF + 1
        movlw   0x0
        movwf   DATA_BUFF + 2
        movlw   0x0
        movwf   DATA_BUFF + 3
        movlw   0x0
        movwf DATA_BUFF + 4
        movlw 'N'
        movwf DATA_BUFF + 5
        movlw 'O'
        movwf DATA_BUFF + 6
        movlw '_'
        movwf DATA_BUFF + 7
        movlw 'I'
        movwf DATA_BUFF + 8
        movlw 'M'
        movwf DATA_BUFF + 9
        movlw 'P'
        movwf DATA_BUFF + 0xA
        movlw 0x0
        movwf DATA_BUFF + 0xB
        movlw 0x0C
        goto  WritePacket

TakeSendSample:
        ;; Takes a single sample, sends it back
        bcf   STATUS,RP1
        call  TakeSample
        bsf   STATUS,IRP
        movlw DATA_BUFF+2
        ;;; Little Endian Format
        movwf FSR
        movfw Sample0_L
        movwf INDF

        incf  FSR,F
        movfw Sample0_H
        movwf INDF

        incf  FSR,F
        movfw Sample1_L
        movwf INDF

        incf  FSR,F
        movfw Sample1_H
        movwf INDF

        movlw 0x6
        goto  WritePacket

StoreSample:
        bcf     STATUS,RP0
        bcf     PIR2,BCLIF
        bcf     SSPCON,WCOL
        bcf     STATUS,IRP
        movlw   MemConfig
        movwf   FSR
        movlw   0x7
        movwf   COUNTER
        I2CStartBit     StoreSample
StoreSampleLoop:
        movfw   INDF
        I2CWriteW       StoreSample
        I2CTestAck      StoreSample
        incf    FSR,F
        decfsz  COUNTER,F
        goto    StoreSampleLoop
        I2CStopBit      StoreSample
        movlw   0x4
        addwf   MemAddrL,F
        btfsc   STATUS,C
        incf    MemAddrH,F
        return



TakeSample:
        ;; To implement a software gain,
        ;; shift out extra (more than 16 bits) here, save the sign bit
        ;; and discard some of the higher order bits (depending on an
        ;; extra variable set by an additional Set/Get command)
        ;; Then, store this sample. It is also probably worthwhile
        ;; to ensure that these bits are always the same as the sign
        ;; bit (to ensure that the dynamic range of the device is being
        ;; exceeded).
        movlw   0x10
        movwf   COUNTER
        bcf     P_SCLK
        bcf     P_ADCS          ; Start ADC output

        ;; save OSR for restore later
        movf    OSR,W
        movwf   TEMP

        ;; present OSR<4>, EOC appears
        PRESENT_OSR
        btfsc   P_SDI0
        bsf     EOC_ERROR
        btfsc   P_SDI1
        bsf     EOC_ERROR
        ;skip three bits
        bsf     P_SCLK          ; EOC, OSR<4> sampled
        bcf     P_SCLK

        ;; present OSR <3>, '0' appears
        PRESENT_OSR
        bsf     P_SCLK
        ; next bit is the sign bit, then data bits
samplebitloop:
        bcf     P_SCLK          ; clocks in next bit

        ;; present next OSR bi
        PRESENT_OSR             ; We don't care that this gets
                                ; repeated/trashed as the remaining
                                ; bits don't matter to the 2440

        bcf     STATUS,C
        btfsc   P_SDI0
        bsf     STATUS,C
        rlf     Sample0_L,F
        rlf     Sample0_H,F

        bsf     P_SCLK          ; Put this here to make clock symmetrical

        bcf     STATUS,C
        btfsc   P_SDI1
        bsf     STATUS,C
        rlf     Sample1_L,F
        rlf     Sample1_H,F

        decfsz  COUNTER,F
        goto    samplebitloop
        bsf     P_ADCS          ; start new conversion
        bcf     P_SCLK      ; return to idle state
        ;; restore OSR
        movf    TEMP,W
        movwf   OSR
        movlw   b'10000000'
        xorwf   Sample0_H,F
        xorwf   Sample1_H,F
        return                  ; return value
```

```
; *******************************************************
; Send the data buffer back.
;
; <STX><STX>[<DATA>...]<CHKSUM><ETX>

SendAcknowledge
        movlw  0x01           ; Send acknowledge

WritePacket:
        movwf  COUNTER

        TransmitEnable
        movlw  STX            ; Send start condition
        call   WrRS485
        call   WrRS485
        movfw  MyAddr
        call   WrRS485        ; master address == 0
        clrf   CHKSUM         ; Reset checksum

        movlw  DATA_BUFF      ; Setup pointer to buffer area
        movwf  FSR

SendNext                      ; Send DATA
        movf   INDF,W
        addwf  CHKSUM,F
        call   WrData
        INC_BUF_POINTER       ; trashes w
        decfsz COUNTER,F
        goto   SendNext

        comf   CHKSUM,W       ; Send checksum
        addlw  0x01
        call   WrData

        movlw  ETX            ; Send stop condition
        call   WrRS485
        TransmitDisable
        btfss  CALL_WRITE_PACKET ; we did a call, need to retur
        goto   recv_packet
        bcf    CALL_WRITE_PACKET
        return                ; return
; *******************************************************

; *******************************************************
; Write a byte to the serial port.

WrData
        movwf TXDATA            ; Save the data

        xorlw  STX              ; Check for a STX
        btfsc  STATUS,Z
        goto   WrDLE            ; No, continue WrNext

        movf   TXDATA,W
        xorlw  ETX              ; Check for a ETX
        btfsc  STATUS,Z
        goto   WrDLE            ; No, continue WrNext

        movf   TXDATA,W
        xorlw  DLE              ; Check for a DLE
        btfss  STATUS,Z
        goto   WrNext           ; No, continue WrNext

WrDLE
        movlw DLE               ; Yes, send DLE first
        call   WrRS485

WrNex
        movf   TXDATA,W         ; Then send STX


; *********************************************************
WrRS485:
        bcf    STATUS,RP1
        btfss  PIR1,TXIF     ; not needed, as we always transmit a byte
        goto   $ − 1         ; full
        movwf TXREG
        return

RS485Delay:                  ; 1 bit long delay, to allow for change-over
                             ; time between transmit/receive (43 cycles)
        movlw 0xD            ;1
        movwf TEMP ;1
RS485DelayLoop:
        decfsz TEMP,F ; 1, 2
        goto   RS485DelayLoop ; 2 cycles ;2+3*(x-1)+2+4
        return

; *********************************************************
; ************************************************************
RdRS485:
        btfss   RCSTA,OERR ; Reset on overrun
        goto    RdRS485Con
        bcf     RCSTA,CREN
        bsf     RCSTA,CREN
RdRS485Cont:
        btfss   PIR1,RCIF      ; Wait for RS485 Data
        goto    $ - 1
        movf    RCREG,W
        movwf   RXDATA
        return
; ************************************************************

        org     0x21FF
;;; Node Address follows. Note! de, not db (EEPROM storage)
        de      0x65

        END
```

Bibliography

[1] Analog Devices, Norwood, MA. *ADXL150/ADXL250 Datasheet,* Rev. 0, 1998.

[2] Axelson, Jan. *Serial Port Complete:* Programming and Circuits for RS-232 and RS-485 Links and Networks. Lakeview Research, Madison, WI, 2000.

[3] Burr-Brown Corporation, U.S.A. *PCM3501 Datasheet*, 1999.

[4] The GTK+ Development Team. *GTK+ 2.2 API Reference*, 2003.

[5] Lenk, John D. *Simplified Design of Filter Circuits.* Newnes, Boston, MA, 1999.

[6] Linear Technology Corporation, Milpitas, CA. *LT1129-5 Datasheet*, 1994.

[7] Linear Technology Corporation, Milpitas, CA. *LTC1387 Datasheet*, 1997.

[8] Linear Technology Corporation, Milpitas, CA. *LTC2052 Datasheet*, 2000.

[9] Linear Technology Corporation, Milpitas, CA. *LTC1864/LTC1865 Datasheet*, 2001.

[10] Linear Technology Corporation, Milpitas, CA. *LTC2440 Datasheet*, 2002.

[11] Meiksin, Z. H. *Complete Guide to Active Filter Design, OP AMPS, & Passive Components.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[12] Microchip Technology, Chandler, AZ. *PIC16F87X Datasheet*, 2001.

[13] Microchip Technology, Chandler, AZ. *AN851: A FLASH Bootloader for PIC16 and PIC18 Devices*, 2002.

[14] Microchip Technology Inc., Chandler, AZ. *24LC515 Datasheet*, 2002.

[15] Middlehurst, Jack. *Practical Filter Design.* Prentice Hall, New York, NY, 1993.

[16] Vandiver, J. Kim. Dimensionless Parameters Important to the Prediction of Vortex-Induced Vibration of Long, Flexible Cylinders in Ocean Currents. *Journal of Fluids and Structures*, 7:423–455, 1993.

[17] Vandiver, J. Kim. *Research Challenges in the Vortex-Induced Vibration Prediction of Marine Risers*, number OTC 8698 in Offshore Technology Conference, Houston, TX, May 1998.

[18] Vandiver, J. Kim and Cornut, Stéphane F. A. *Offshore VIV Monitoring at Schiehallion - Analysis of Riser VIV Response*, number OMAE 005022 in ETCE/OMAE 2000 Joint Converence, New Orleans, LA, February 2000.

[19] Vandiver, J. Kim and Mazel, Charles H. *A Field Study of Vortex-Excited Vibrations of Marine Cables*, number OTC 2491 in Offshore Technology Conference, Houston, May 1976.

[illegible]