ON THE USE OF MICROCONTROLLERS FOR DATA ACQUISITION IN AN
INTRODUCTORY MEASUREMENTS COURSE

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in Aerospace Engineering

by

Thomas R. Szarek, B.S.A.E.

Dr. Patrick F. Dunn, Director

Graduate Program in Aerospace and Mechanical Engineering
Notre Dame, Indiana
December 2003
A microcontroller-based, miniature data acquisition system was developed for use in a sophomore-level Introduction to Measurement course. This board demonstrates that principles of proper data acquisition are applicable not only in the laboratory, but also in “real-world” applications where traditional data-acquisition systems are not easily implemented. Two experiments made use of the board’s capability – one to determine the altitude achieved by a model rocket and one to measure the surface temperature distribution of a lake. A custom-made rocket carried the required electronics, including an accelerometer to measure the rocket’s acceleration and a pressure transducer to measure the rocket’s velocity. A custom-made electric-powered boat carried a thermocouple to measure the water temperature and a Global Positioning System (GPS) to determine the boat’s location. The details of the electronic board, rocket, and boat design, along with the results of the experiments carried out by the students, will be presented.
DEDICATION

To my parents, who always told me to shoot for the stars.
FIGURES

2.1 Estes Altitrak system [2] ........................................... 5
2.2 Original prototype board developed by Joel Preston for the under-
graduate research experiment ............................................ 6
2.3 Block diagram of the Intel 8051 Microcontroller (recreated from [5]) . 8
2.4 8051 RAM memory map (recreated from [5]) .......................... 12
2.5 Block diagram of the four modes of operation for Timer 0 (recreated
from [5]) ................................................................... 17
2.6 The main components of a model rocket [7] ........................... 21
2.7 Cross-section of a typical model rocket engine [8] ................... 22
3.1 Components of the first board (actual size) .......................... 26
3.2 Differential amplifier circuit ............................................. 27
3.3 Front panel of the LabView interface program ........................ 30
3.4 Components of the second board (actual size) ....................... 39
4.1 Custom-made rocket for carrying electronics package ......... 45
4.2 Location of accelerometer inside the wooden coupler ............ 46
4.3 Location of pressure transducer and pressure taps inside the nose cone. 47
4.4 Launch of rocket showing the launch pad and stake where the external
interrupt wires are attached (courtesy P. F. Dunn) .................... 47
4.5 Centrifuge system ...................................................... 48
4.6 User interface and keypad for centrifuge ............................... 52
4.7 Strain-gage beam balance with rocket motor and collar (courtesy P.
F. Dunn) ......................................................................... 53
4.8 Thrust curve for Estes D12 series of engines (from estesrockets.com) . 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9</td>
<td>Results of digital notch filtering from 20 to 30 Hz on the measured</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>thrust data.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>Thrust data in various stages the of transform to obtain the average</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>thrust profile.</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Results of the accelerometer calibration.</td>
<td>62</td>
</tr>
<tr>
<td>4.12</td>
<td>Results of the pressure transducer calibration.</td>
<td>62</td>
</tr>
<tr>
<td>4.13</td>
<td>Acceleration and Velocity data for a typical launch.</td>
<td>63</td>
</tr>
<tr>
<td>4.14</td>
<td>Comparison of integrated acceleration with measured velocity.</td>
<td>64</td>
</tr>
<tr>
<td>4.15</td>
<td>Flight profile of typical model rocket launch.</td>
<td>65</td>
</tr>
<tr>
<td>4.16</td>
<td>Comparison of typical acceleration data with acceleration recorded</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>on the crash flight.</td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>Comparison measured and predicted acceleration and velocity.</td>
<td>66</td>
</tr>
<tr>
<td>4.18</td>
<td>Schematic of arrangement used to determine the rocket’s height.</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Custom-made Boat (courtesy P. F. Dunn).</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Thermocouple dynamic calibration.</td>
<td>77</td>
</tr>
<tr>
<td>5.3</td>
<td>Plot of lake surface temperature distribution.</td>
<td>78</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

In this world, there are not many things I can say for certain. This I can – without the help, guidance, and motivation of countless people, I would have never been able to accomplish this endeavor. From family, to teachers, to friends, to those persons that I do not know by name, but who have changed the course of my life, there are not enough pages in this book to express my gratitude.

First, I owe so much of who I am as an engineer to my advisor Dr. Patrick Dunn. Without your constant guidance and well-timed prodding, I would have never completed this project, and worse yet, would never have gained all the knowledge and experience that have shared with me.

To my readers Drs. Thomas Corke and Flint Thomas, thank you for your questions and your feedback.

To Dr. Stephen Batill for serving as chair for my thesis defense.

To the GE Company whose grant to develop modules for the Engineering Learning Center made this research possible, which I hope will make it possible for the next generation of engineers to learn from what I have done.

To electronics specialist Joel Preston, from whom I can say for certain I learned everything I know about electronics.

To machinist Mike Swadener who was invaluable in helping to construct all hardware needed to conduct the experiments.

To my lab partner Jason Miller who first started me down this path of research.

To my roommate Dave Tyler who served as an excellent proffreeder. To all my
professors and colleagues for teaching me so much, whether is be in the classroom, in the laboratory, or in the computer room late at night with an assignment due the next day.

To all my friends, from high school, undergrad, and graduate school, from those who are far away to those I hold near to my heart, you are a constant reminder that we work to live, not live to work. I am blessed to enjoy what I do for a living, but I am more blest to enjoy the company you provide.

Finally, to my family, my parents Diane and Bob, my brother Mark, my grandparents, aunt and uncles, cousins, and close family friends, simply put, without your constant support and love, I would not be the person I am today.

Thank you all.
CHAPTER 1

INTRODUCTION

1.1 Motivation

Today’s engineering graduates can no longer be constrained to specific fields of study such as aerospace, mechanical, electrical, or chemical. While specialization in such fields are necessary, every modern engineering project is a multidisciplinary effort requiring engineers who are familiar with tools and techniques from outside of their discipline.

In the past four years, there has been an effort in the Department of Aerospace and Mechanical Engineering at the University of Notre Dame to introduce undergraduates to the most basic building block of all mechatronic systems – the microcontroller.

This is of particular importance in modern engineering applications as digital processors find their ways into more and more, and smaller and smaller, technologies. Microcontrollers are used in applications as diverse as toaster ovens, automobiles, and unpiloted aerial vehicles.

1.2 Objectives

This project focused on introducing engineering students to the idea of using microcontrollers as tools for data acquisition. After first defining the scope of the experiments the students would be asked to conduct, there were three specific development objectives.
First, an electronics board consisting of a microcontroller and all required auxiliary circuits necessary for data acquisition (i.e. signal conditioning op-amps, analog-to-digital converters, solid-state memory units) needed to be designed and manufactured. The major constraints on the board were size and weight given the board’s application in a model rocket. Further, it was sought to make the board as adaptable as possible for future applications. Two generations of this electronics board were designed as part of this project.

Secondly, a software interface tool needed to be designed to allow the students to easily change acquisition parameters such as sampling rate and number of samples without needing to modify the actual software running on the microcontroller. A LabView program was written to give students the ability to simply set the acquisition parameters and retrieve data from the board.

Finally, all the hardware required for the experiments needed to be manufactured. This included a custom-made model rocket large enough to carry the electronics and powerful enough to lift the weight of the system. It was fitted with an accelerometer and a pressure transducer that connected to a total and static pressure tap placed in the rockets nose cone. A closed-loop, speed controlled centrifuge was designed along with a user interface circuit so the students could easily calibrate the accelerometer. A custom-made electric powered boat was designed to carry a thermocouple and GPS unit. All the electronics were housed in a water resistant box for their operation.

1.3 Application

The experiments and hardware developed in this project were designed specifically for AME250: Introduction to Measurement and Data Analysis. This course is offered in the sophomore year and consists of both lecture and laboratory compo-
ments. The course objectives are stated as:

This course is designed to introduce the sophomore aerospace engineering student to experimental methods used in the acquisition of data, statistical techniques used in the interpretation of data, and to uncertainty analysis. This information is presented in a lecture format and then reinforced by having the student perform four data acquisition system-based exercises in the laboratory.

Given the goals of the course, teaching aspects of the microcontroller internal functionality and the software required to operate the system were not of primary concern. Instead the students used the system as tools in much the same manner as an oscilloscope would be used.

The experiments were implemented on a trial basis for the 2001 and 2002 school years. A group of approximately twenty students per year, twenty-five percent of the total course enrollment, was selected to conduct either the rocket or boat experiments in lieu of the normal final laboratory experiment.

Students were asked to not only prepare a technical report on their results, but also to make a presentation to the entire class to share some of their experiences and knowledge gained with the other students.
CHAPTER 2

BACKGROUND

2.1 Project History

The history of this project dates back to 1998. At this time, the final project for the AME250 class was to predict the height obtained by a model rocket. Thrust measurements of the rocket engines and a drag analysis on the rocket shape were performed. Numerical simulations were found to be in close agreement with the measured heights; however, the means by which the rockets height could be measured was fraught experimental error.

The system used to measure the rocket was the Estes AltiTrak. It is a “gun” with a free-to-rotate weight, a trigger to lock the weight in place, and angle markings. A picture of the device is shown in Figure 2.1. By pointing the gun at the rocket and releasing the trigger, the device would lock the weight at the angle at which the gun was held with reference to the ground. If the ground distance from the user to the rocket was known, then a tangent relation would give the rockets height.

The sources for error were numerous. First, the ground distance used was that of the user to the launch pad, which assumes that the rocket would travel straight up. If the rocket deviated from the purely vertical path, which it almost always does in the presence of wind, the ground distance assumption would be incorrect. Secondly, if the user were to look through the sight incorrectly, the recorded angle would also be incorrect. Further, the weight is subject to inertial effects, such that
if the user were moving the gun when the trigger was released, the weight might be set at an angle not perpendicular to the ground.

Given these uncertainties, the author and his lab partner Jason Miller decided to further investigate the rocket’s trajectory as an undergraduate research project. The eventual approach employed was to measure the rocket’s flight characteristics with an on-board microcontroller and sensors.

A prototype miniature data acquisition system was designed and constructed by the department’s electronic specialist Joel Preston. The design consisted of a 5V voltage regulator, interface buttons and light-emitting diodes (LEDs), analog filters, and the Analog Devices ADuC812 Microconverter that served as the microcontroller, A/D converter, and memory unit. A picture of the unit is shown in Figure 2.2. A more detailed explanation of the board can be found in the paper by Miller and Szarek [1]. The results from the first prototype proved so successful that the project was introduced into the AME250 curriculum the next year.
2.2 Microcontrollers

2.2.1 History

Microprocessors can be defined simply as computers on a single chip. While the idea of placing all the required transistors for a computing on a single piece of silicon has been around since 1969, the first truly successful chip to do so was Intel’s 8080. The 8080 was introduced in 1974 and became the heart of the world’s first PC, the Altair. The chip, while state-of-the art at the time, is almost laughable by today’s computing standards. The chip ran at 2 MHz, with an 8-bit architecture, and had 6,000 transistors. For comparison, an Intel Pentium 4, introduced in 2000, runs at 1.5 GHz, with a 32-bit architecture and has 42 million transistors [3].

This humble chip gave birth to all microcomputers. As manufacturing technology improved, the additional silicon space was utilized for one of two purposes. The microprocessor, which sits at the heart of every personal computer, followed the path
of using the extra silicon for more speed and computational power. Mathematical
operations in the 8080 could be performed 8-bits at a time and usually took one
clock cycle. By 1978, the Intel 8088 has moved onto a 16-bit architecture and 5
MHz clock speed, meaning not only could the additions be performed on numbers
twice as large, but they could be carried out more than twice as fast. So went the
progression that lead to the microprocessors we have today [4].

On the other hand, in 1976 Intel introduced the 8748, what could be considered
the first microcontroller. This processor maintained the 8-bit architecture but added
internal memory both in the form of random access memory (RAM) for temporarily
storing information and erasable programmable read-only memory (EPROM) for
storing code, which survived even when power to the chip was turned off. Even
today, microprocessors are dependent on external RAM and ROM (hard drives),
while almost all modern microcontrollers have both types of memory internally. The
designers also added additional input-output (I/O) lines so the chip could interface
with other devices and an 8-bit timer, which counted the clock cycles so time-
dependent events could programmed.

Chips like the 8748 became known as microcontrollers because they saw their
biggest market as control systems, such as the ones you would find in washing
machines and traffic lights. In 1980, Intel’s 8051 became the standard for all mi-
crocontrollers. It contained over 60,000 transistors with 4 kB ROM, 128 bytes of
RAM, 32 I/O lines, a serial port, and two 16-bit timers.

To this day, most microcontrollers maintain the 8051 processor at their core.
They have added more memory, serial ports, I/O lines, and other features such as
analog-to-digital (A/D) conversion, but everything is backwards compatible to the
original 8051.
2.2.2 8051 Features

A microcontroller, like any digital circuit, is simply a collection of transistors etched into silicon. These transistors, when arranged properly, allow for simple logical operations such as AND, OR, etc.; arithmetic operations such as addition, subtraction, etc.; and memory storage capabilities. To fully understand their functionality requires covering topics broad enough to fill an entire circuit design curriculum and so for brevity will not be discussed in this thesis. It is important to have some understanding of the inner working of a microcontroller, though, to understand the possibilities and limitations of their application. The following sections\(^1\) discuss the operation of some of the major parts the 8051 microcontroller. A block diagram of these features is shown in Figure 2.3.

\(^1\)Most of the information provided in the following sections comes from the author’s experience in working with microcontrollers; however, the organization along with some of the material presented is drawn from MacKenzie’s *The 8051 Microcontroller* [5].

---

Figure 2.3. Block diagram of the Intel 8051 Microcontroller (recreated from [5]).
2.2.2.1 Central Processing Unit

The central processing unit (CPU) of a microcontroller, like the CPU of a computer, is the part of the chip that carries out the code provided by software by performing the required computations. The CPU in very simplified form, consist of five parts.

The first is the program counter (PC). This is a memory array that holds the address of the next instruction in the code to be executed. On a chip reset, this array is populated with zeros so as to start the code at the first location.

When the code begins, it will go to the first memory location, as indicated by the PC, and load the instruction in to the instruction register (IR). This simply holds the 8-bit instruction during its operations.

The IR itself is rather mundane. The meaning of the code is determined by the instruction decode and control unit. This unit processes the instruction and ensures that its purpose is carried out.

Any arithmetic or logical operations that need to be carried out are done in the aptly named arithmetic and logic unit (ALU). This array of transistors is where the term 8-bit logic is applied. If the ALU is 8-bits wide, then the microcontroller is an 8-bit architecture since operations will be carried out 8-bits at a time. The two bytes to be operated on are loaded into the “top” of the ALU. As they descend “down” the ALU they are acted upon (ANDed, ORed, carried over, etc.) as dictated by the instruction decode and control unit. The output from the ALU appears at the bottom.

Finally, several registers, simply reserved memory locations, provide temporary space to house the bytes to be operated on and the results of the calculations.

The CPU is connected to all other parts of the chip via data buses. In this way data from the memory (either internal or external), A/D converter, or serial port
decoder can be used by the CPU and passed between the different chip peripherals.

As an example, if an instruction in code is calling on the CPU to obtain a byte of memory from RAM, the instruction decoder will send the address for the desired byte along with a read instruction to memory and then process the returned byte sent into the CPU by memory. This byte would temporarily be stored in an unused register. The ALU would then act upon this byte as specified by the next instruction, again placing the result in a register. The result might then be passed back into memory, or used by a succeeding instruction that might perform a logical operation on the result, such as that called for by an “IF” statement in the higher-level code. The result of the logical operation could then be used to set the PC, leading to the code to jump to one of two subroutine locations, and so on.

It is important to note that most programmers will never actually involve themselves directly in writing the instruction set carried out by the code. This machine language comes as the result of a compiler, a program that transforms higher level code such as assembly language or C into machine code.

2.2.2.2 Memory Structure

The 8051 consist of two types of memory, ROM and RAM, both of which can be either internal or external. The ROM is used for code storage. In the original 8051, programmers would send their code to the factory to have it permanently etched into the internal ROM. Using internal ROM allowed for faster operation, but meant that code could not be changed once it was etched into the silicon. Later use of EPROM, which allowed memory to be changeable and yet retainable during power loses, meant almost all microcontrollers began using internal EPROM for code memory.

The RAM is used for variable storage and functionality control. It is volatile,
meaning, its content is lost on power loss. There are four sections of RAM as shown in the memory diagram in Figure 2.4.

The first type of RAM is the general purpose RAM. This is what is used to store variables in the code. This can be accessed by the CPU as described in the example in the previous section.

The second type of RAM is the bit-addressable RAM. This memory section allows the programmer to use single bit variables. This is useful because it is faster to read and operate on a single bit than operating on an entire byte, not to mention much more compact since in the other seven bits in a byte operation would go unused.

The third group is the memory locations reserved to function as registers. There are four register banks reserved at the beginning of the memory stack. While these location can be used simply as general purpose RAM, they have the additional capability of serving as the registers for the CPU described in the previous section. Their biggest advantage is that the CPU can be programmed to see the address of these banks as a single byte rather than the usual two bytes required to access other memory locations. This allows for faster operation, an important consideration when moving data into and out of the registers with nearly every operation.

The final class of RAM is the special function registers (SFRs). The SFRs are used to control the operation of all the attached peripheral devices. The SFR location in memory was purposely sparsely populated so as to allow for the addition of future functionality. The exact operation of these SFRs will be discussed when their associated peripherals are discussed in more detail below.
Figure 2.4. 8051 RAM memory map (recreated from [5]).
2.2.2.3 Ports

The original 8051 had four, eight-pin, general-purpose input/output (IO) ports. These pins could be set either high or low by writing to the Port n (Pn) SRF, where “n” refers to the port number. When a one is set, the pin is pulled high internally. These ports can also be used as inputs, where an external device grounds the pin to make the read value of the Pn SFR a zero.

The Pn SFR can either be written to or read from as a byte or one bit at a time. SFRs with this capability are referred to as bit-addressable and easily allow the user to change a single bit without concern for the values of the other bits in the register.

The ports also served as pins for the other peripherals on the chip. Ports 0 and 2 serve as both the address and data lines for any external memory. Port 3 serves as the pins for the timers, external interrupts, and serial ports described below. Port 1 is always reserved for general I/O. If a pin is being used with a peripheral device, it can not be used as an I/O pin.

The original 8051 was not capable of sourcing or sinking much current. Later derivatives added direct sourcing and sinking capability meaning external devices, such as an LED, could be powered directly from the microcontroller pin rather than through a transistor. For this functionality, the user must specify whether the pin is to be a push-pull output or act as an open-drain input. This requires an additional SFR for each port.

2.2.2.4 Interrupts

Interrupts are event-driven interruptions of the main program. This allows the microcontroller to respond to important occurrences when they happen, rather than having to continuously check for them.
Interrupts can be generated by any number of sources, primarily from the on-chip peripherals. For example, the serial port system can generate an interrupt saying a byte was just received. The A/D system can generate an interrupt saying its conversion is complete. A user can also generate an interrupt by pushing a button tied to one of the external interrupt pins. Note that these can occur at any time in the code's operation and their exact timing cannot be predicted.

When an interrupt is received, a flag is set in one of several SFRs corresponding to the interrupt source. Setting this flag initializes the interrupt service routine (ISR). The ISR will store the current memory pointer for the code, load the PC with the memory location for the routine to be executed based on the interrupt source, and at the completion of that routine, point the main program back to its original place. In this manner, the CPU can give the appearance of performing several operations simultaneously, although in reality it is simply hopping from one subroutine to another.

Interrupts must be either enabled or disabled by setting the Interrupt Enable (IE) SFR. This SFR acts as the on-off switch for all the interrupts (later 8051-derivatives with more peripherals have added additional SFRs). Each bit in the register corresponds to a particular interrupt. Setting a one in a position will enable that interrupt. Setting a zero disables it. In silicon, the memory location acts as an input to an AND gate along with the interrupt flag bit. Only when a one is present in both, will the ISR be activated.

Bit seven of the IE register is the global interrupt enable. All interrupts, regardless of the individual states are only enabled when this bit is set to one. This allows the programmer to easily suspend all interrupt occurrences without having to disable each individual interrupt source.

To avoid conflicts of multiple interrupts, or receiving one interrupt while execut-
ing the routine of another, a very specific interrupt polling sequence is enforced. The programmer can modify this sequence slightly by setting each interrupt to one of two priority levels using the Interrupt Priority (IP) SFR. The higher level interrupts are first executed and then the lower level ones. The order within the level is still fixed by the polling sequence. When multiple interrupts do occur, the lower level interrupt is held and executed after the higher level one is finished.

It is worth mentioning that the reset pin can be considered to function as an interrupt that is always enabled. There are no means to disable the reset pin in software. On a reset, the chip is set back to its initial state and the PC set to memory location zero.

2.2.2.5 Timers

Working hand-in-hand with the interrupt system are the timers. Timers are simply counters of a specific length that increment with each incoming clocking source transition. The original 8051 had two 16-bit timers.

The clocking source can come either from the main system clock (in which case it is pre-divided by 12 so as to allow for reasonable rates) or from an external source such as a specially reserved timer pin or an external interrupt pin. In the later case, the timer is used mainly for event counting.

When the system clock is used, the timer is primarily being used for generating interrupts at specific intervals. When the counter, which is of a specific length such as 8, 13, or 16 bits, overflows an interrupt is generated. A counter works in much the same way a human adds long numbers. When the sum of two numbers is greater than ten, a one is carried to the next column and the remainder placed in the original column. When adding in binary, any number greater than one results in a carry. With only a fixed number of bits with which to work, an overflow occurs
when the most-significant bit (MSb) generates a carry. The counter will go from having all ones to all zeros with a carry bit being passed out. The MSb’s carry bit is used to set the timer interrupt flag. In this way interrupts can be generated at regular intervals.

If the length of the timer were the only parameter that could be set, the possible frequencies that could be used would be quite restrictive. To obtain the full spectrum of possible frequencies, the user can initialize the timer to some value, thus generating the interrupt sooner based on the number of clock cycles left until overflow.

There are four modes of operation for the timers as illustrated in Figure 2.5. Mode 0 operates the timer in 13-bit mode, meaning the counter is 13-bits long. This mode is for backwards compatibility with the 8051 predecessor the 8048. Mode 1 operates the timer in 16-bit mode, meaning the full 16-bits are used for counting. The disadvantages of Modes 0 and 1 are that if the user specifies the time to overflow by setting the initial value of the timer, it becomes necessary to do so after every overflow to ensure regularly timed interrupts.

Mode 2 makes use of a feature called auto-reload. The timer functions in an 8-bit mode with the lower byte serving as the counter and the upper byte holding the initialization value. On the 8-bit overflow, the lower byte is set to the value stored in the upper byte. Mode 3, which is only available for Timer 0, allows the timer to function as two independent 8-bit timers.

All of the parameters mentioned above are set by the Timer Control (TCON) and Timer Mode (TMOD) SFRs. Modes, clocking sources, interrupt flags, and run enables for both timers are contained in these two SFRs. The values of the timers themselves can be accessed using the TH0, TL0, TH1, and TL1 SFRs where “H” and “L” refer to the high and low bytes and “0” and “1” to the timer number.
Figure 2.5. Block diagram of the four modes of operation for Timer 0 (recreated from [5]).
Later 8051-derivatives have added additional timers and many have given those additional timers full 16-bit auto-reload capability. This is particularly useful for baud rate generation for serial port applications.

2.2.2.6 Serial Port

The original 8051 came with a single serial port system. The serial port consist of two pins, TXD and RXD for transmit and receive, respectively. The serial port can be operated in one of four modes. Mode 0 allows for 8-bit synchronous communication. Mode 1 allows for 8-bit asynchronous communication. Modes 2 and 3 are the same as 0 and 1 except with 9-bits transmitted. The serial port modes, enables, and interrupt flag are set in the Serial Port Control (SCON) SFR.

The two modes of interest are Modes 0 and 1. When in Mode 0, the serial port acts as a simple shift register. Data written in the Serial Port Buffer (SBUF) SFR is clocked out over the RXD line one bit at a time at a frequency set by the system clock divided by twelve. The TXD line is used to pass out the clock pulses. The receiving party uses the rising edge of the clock to sample the data line. Incoming data also comes over the RXD line with the microcontroller continuing to generate the clock pulses over the TXD. Received bytes are also placed in the SBUF register.

In Mode 1, the serial port operates as a universal asynchronous receiver transmitter (UART). This is the mode most computer users are familiar with from dealing with their computer’s serial port. Here the RXD and TXD lines are used in their traditional receive and transmit roles with the timing on the lines coming from one of the timers. The original 8051 used Timer 1 to generate the baud rate, although later derivatives allowed additional timers to be used, as well as allowing different baud rates to be used for receive and transmit. The timer overflows are further divided by either 32 or 16 depending on the desired baud rate.
For every byte to be transmitted, ten clocking cycles are used. First a start bit that is always a zero is sent, then the eight data bytes starting with the least-significant bit (LSb) are transmitted, and finally a stop bit that is always a one is output.

Interrupts are generated either after a transmit or after a successful receive. For a successful receive, a proper start bit must be detected (as opposed to spurious noise on the line), and any previously received bytes have had to been read by software. Upon a successful transmit or receive, the TI or RI interrupt flags are set in the SCON register, and, if enabled, the serial port service routine activated. Since the same routine is called for either a transmit or receive, it is necessary for the programmer is check the status of the interrupt flags to see which occurrence caused the interrupt. For this reason, the interrupt flags are not cleared by hardware and must be cleared in software. Failing to clear these flags will result in missed receive bytes since received bytes will not be loaded into the SBUF register if the RI flag is not zero.

2.2.2.7 A/D Conversion

While not an original feature of the 8051, A/D conversion, because of its importance to this project, will be briefly explanation below. The conversion specifics for the Cygnal C8051F020, the microcontroller used for this project, will be described.

The Cygnal C8051F020 has an eight-channel, 12-bit A/D system. The eight channels can function as either eight single-ended inputs, four differential inputs, or any combination of these two states (i.e. four single-ended and two differential inputs). The channel functionality is set by the Analog Multiplexer Configuration (AMUX0CF) SFR.

The A/D converter is not capable of simultaneous sampling. Rather is uses
only a single A/D converter with an analog multiplexer to select which of the eight lines to sample. The multiplexer is controlled by the Analog Multiplexer Channel Selection (AMUX0SL) SFR.

An internal operational amplifier provides the capability of gaining the incoming signal. By setting the appropriate bits in the A/D Conversion Configuration (ADC0CF) SFR, gains of 0.5, 1, 2, 4, 8, or 16 can be set for the selected line.

Conversions start based on the mode of operation of the A/D converter as set by the A/D Converter Control (ADC0CN) SFR. The four possible conversion start events are writing a one to AD0BUSY bit of ADC0CN, a Timer 2 overflow, a Timer 3 overflow, and a rising edge detected on the external Conversion Start (CNVSTR) pin. In all cases the AD0BUSY bit is set high. At the completion of a conversion, the AD0BUSY bit is set low, the conversion complete flag AD0INT is set high, and if enabled, the A/D Conversion Complete interrupt initialized. The result of the conversion is stored in the ADC0H and ADC0L SFRs.

2.3 Model Rocketry

2.3.1 General Construction

A model rocket is a rocket weighing less than 1.5 kg propelled by a solid rocket engine generating no more than 320 N-s of thrust [6]. As with all rockets, model rockets are built to be lightweight. Typical construction material includes cardboard, balsa, rubber, and plastic. Rockets are sold to hobbyists either as complete kits or by the part for custom-made projects. Figure 2.6 shows the main components of a model rocket.

The main body of the rocket is typically made out of a cardboard tube with no internal structure. The rigidity of the tube provides all the required structure. Any payload, as well as the required recovery system (often a parachute), are carried
inside the body tube.

The engine is attached to the rocket body by the engine mount. The mount provides a secure housing for the rocket engine and transfers the thrust to the rocket body through two cardboard disks that serve as spacers between the engine mount and body tube.

Attached to the outside, at the base of the body tube are the fins. The fins act to stabilize the rocket in flight. They are made either from balsa or plastic and are arranged either in sets of three or four. These are often aligned with the flow, but can be angled so as to induce a rotation of the rocket.

Finally, a nose cone made of either plastic or balsa is fitted inside the top of the body tube. The most common nose cone shape is the ogive, although the specific proportions vary largely from rocket to rocket.

2.3.2 Engine

The rocket engine is a solid propellant encased in a cardboard shell. There are three main parts to the rocket engine: the propellant, the non-thrust delay, and the ejection charge, as shown in Figure 2.7.

The propellant is the first section to be lit. Its initial concave shape creates a period of large thrust boasting the rocket off of the launch pad. The engine then
enters a period of constant thrust before all the propellant is spent. This phase can last from as little as 0.25 s to as much as 3 s, but typically lasts about 1 s [9].

The delay phase, as the name implies, produces no-thrust and is intended to serve as a delay. During this period a white smoke is emitted to make the rocket easier to follow. This phase allows the rocket to coast and reach its maximum altitude. A well-timed ejection charge will go off right at the rocket’s maximum height when its vertical velocity is zero. To achieve this timing, engine are sold with a variety of delay times, usually on the order of 4 s.

Finally, after the delay period, an ejection charge is fired. This propels gases up into the body tube and forces out the recovery system. Engines are sold in varying sizes. They are coded based on their total impulse, maximum thrust, and delay time.
2.3.3 Recovery System

The recovery system is comprised of two main parts – the parachute or streamer which slows the rocket’s decent, and the recovery wading, which protects the parachute from the hot gases of the ejection charge. The recovery wading is a flame-resistant tissue paper that is crumbled up and placed just above the engine.

In order for the recovery system to work properly, two events must occur. First, the rocket’s nose cone must be loose enough to allow the ejection charge to force the parachute or streamer out through the top of the body tube. Second, the parachute or streamer must properly deploy. This requires that the recovery system be properly stored before placing it in the body tube. Simply jamming the parachute or streamer into the tube will cause the supporting lines to become tangled and make the system ineffective.
3.1 Overview

The crux of this project was the development of a microcontroller-based, miniature data acquisition system. In the end, two systems were developed.

The first board was designed by electronics specialist Joel Preston in the fall of 2000. Its requirements were specified by the author. The software and a LabView interface program were written by the author.

The second board was designed entirely by the author. The design reflects changes both in technology and in the board’s expected application based on lessons learned during the two years of conducting experiments.

3.2 First Board

3.2.1 Requirements

The first board was designed to be an all-in-one data acquisition system. The prevailing thought at the time was that the board would be used for various experiments, and so a robustness of design was necessary.

The requirements for the A/D capabilities called for eight channels of 12-bit resolution. To handle a wide variety of instruments, the sampled voltage range was chosen to go from -5 to +5 V. The desire to potentially make use of unconditioned instruments, such as a Wheatstone bridge output from a strain-gage array, dictated
that gain-producing operational amplifiers be used on each line. The op-amps could
gain the incoming signal by a factor of 1, 10, 100, or 1000.

The microcontroller needed to have at least one UART channel to interface with
a computer for retrieving the sampled data. The microcontroller also needed to
interface with a memory chip so a serial peripheral (SPI) bus was needed. It was
also decided that the microcontroller should have a real-time clock (RTC). An RTC
is a counter that is driven by a separate crystal whose frequency allows to counter
to conveniently overflow on whole seconds, thus making keeping time easier.

The memory chip needed to have enough capacity to store a meaningful amount
of data, while at the same time not using too much board space or I/O lines from the
microcontroller. For this reason, a serial chip was used instead of a parallel. In this
configuration, it took only four lines to communicate with the memory chip (clock,
read, write, and chip select) instead of at least nine with a parallel chip (eight data
lines and ready line).

3.2.2 Layout

The layout of the board is shown in Figure 3.1. The easiest way to describe the
board’s operation is to follow the steps taken during a typical A/D conversion.

First the analog voltage is passed to the board by plugging the instrument into
one of the eight plugs. Each plug has four pins: +5 V, two input pins for differential
inputs, and ground. Should the instrument require a voltage other than +5 V, two
additional plugs allow the user access to adjustable positive and negative voltage
regulators. These allow the user to set any voltage within the range of the supply
voltage from the batteries by adjusting a variable resistor.

The analog voltage is then passed into the National Semiconductor LM324 quad
op-amps. Each chip has four op-amps, so two chips are used for the eight lines. The
Figure 3.1. Components of the first board (actual size).
op-amps are in a differential amplifier configuration as shown in Figure 3.2. The difference of the two incoming signals is gained by that ratio of $R_2/R_1$. $R_1$ is fixed and $R_2$ is selected by a jumper array. To set the gain, one must set two jumpers to the appropriate resistors. The op-amps were supplied by the +5 and -5V buses.

The conditioned analog voltage is then passed into an 8-channel Maxim MAX328 analog multiplexer. This chip takes eight inputs and passes out one depending on the channel selected. The channel is selected by three lines coming from the microcontroller.

The selected analog voltage is passed into a Linear Technologies LTC1276 12-bit A/D converter. Here the analog voltage is converted into a 12-bit digital number. This number is passed out of the chip in parallel, 8-bits at a time. One of the microcontroller’s ports is tied to the A/D converter to receive the incoming data. Additionally, three lines are used for communicating. One is pulled low to begin the conversion, one is used by the A/D chip to indicated that the conversion is completed, and the last is used to select the byte to be output by the A/D chip.
The microcontroller used is an Atmel AT90S8353 AVR. This chip is not an 8051-derivative, however, much of its layout and operation is identical to an 8051 chip. It has 4 I/O ports, a UART system, and SPI bus, and a RTC. The microcontroller interfaces with all the peripheral chips – the analog multiplexer, the A/D converter, the memory chip, and the RS-232 converter. In addition a status indicator LED was tied to one of the unused I/O lines, and a momentary push-button was connected to the External Interrupt 0 pin.

An Atmel 803 memory chip could store 32 kB of data and was programmed over the SPI bus. Because the microcontroller’s code was loaded over the SPI pins, a Fairchild 74HC4053 digital multiplexer was required to keep the microcontroller programming instructions separate from the memory instructions. The three lines (clock, read, and write) were simultaneously switched as indicated by a pin from the programming circuitry. (This circuitry came with the prototyping kit for the AVR chip and was not developed in-house.)

Finally, a Maxim MAX203 RS-232 converter was used to convert the UART lines from the microcontroller from transistor-transistor logic (TTL) levels (0 to 5 V) to RS-232 standard levels (-5 to +5 V) to allow communication with standard RS-232 devices such as a computer serial port.

Two 5V voltage regulators were required to drive the circuitry. Since the positive 5V drew a significant amount of current, a 9V battery had to be used to provide the required power. The -5V bus drew very little current, and therefore permitted the use of a type N 12V battery which to minimized size and weight.

The board itself measured 57×127 cm (2.25×5 in.) and weighed 57 g (2 oz) without batteries, 110 g (3.88 oz) with batteries.
3.2.3 Software
3.2.3.1 LabView

A LabView interface program was developed to allow the students to easily interface with and program the DAQ system. Figure 3.3 shows the front panel of the program. The wire diagram for the program is not presented due to its complexity and difficulty in presenting LabView “code.” The code’s complexity comes mainly from the error checking used to validate the data received from the board. The user interface and board programming routines are straightforward.

The students can control five parameters of the boards operation – the channels to be sampled, the sampling frequency, the number of samples per channel to be recorded, the initial time delay, and the sampling mode.

The channels are selected by the eight push buttons. Any combination of channels is available; however, the order of sampling is always fixed from channel 0 to channel 7. The sampling frequency is the frequency for each channel, not for each conversion. The number of samples to be taken for each channel is limited only by the available memory. The initial time delay is the number of seconds the program will wait from the interrupt button push until sampling commences. Finally, the sampling mode determines the method in which multiple channel acquisitions are conducted. “Normal” mode means all samples are equally spaced. “Burst” mode means the channels are sampled sequentially as quickly as the A/D converter is able.

Once the settings are selected, the user can write these setting to the board using the “Write New Board Settings” button in the “Actions” section of the front panel. From here, the user can also “Read Current Board Settings”, “Download Data in Memory”, and “Clear Data in Memory”. Downloaded data is displayed in the “Main Output” window and can be saved using the “Save to File” button. The output file begins with a summary of the board settings for the acquisition. Then each
Figure 3.3. Front panel of the LabView interface program.
“data set,” corresponding to each button push, is formatted into columns. The first column is the time of the sample; the remaining columns are the conversions from each channel as an integer number from -2048 to 2047, the possible combination for the 12 bit converter.

A more detailed description of the communication protocol between the board and the LabView program along with the method for setting the conversion properties will be discussed in the following section.

3.2.3.2 On Board

The software for the electronics board served two simple purposes – convert analog data to digital form and store it in memory, and also export that data to a computer. Rather than discuss in detail, how the board achieved this functionality, a general overview of the program flow will be discussed. The actual C code is available in Appendix B for the curious reader.

The program begins with the initializations for the various chip components. First the port pins are set up as either outputs or inputs as required. Port A has the pins for the A/D converter output, so they are programmed as inputs. Several pins on Port C (0, 4, and 5) are used to control the A/D chip (i.e. begin conversion, output results, select high or low byte) and are set as outputs. Pins 1 through 3 on Port C are used to set the channel on the analog multiplexer and are set as outputs. Pin 6 on Port D is attached to the status LED and is set as an output.

Next the interrupts are initialized. External Interrupt 0 is wired to a push button that users can push. The push button grounds the pin and so is set to trigger on the falling edge. External Interrupt 1 is wired to the A/D converter. The converter indicates that it has completed its conversion by pulling this pin high. For this reason, the interrupt is set to trigger on a rising edge. Interrupt 1 is disabled unless
a conversion is in progress. Interrupt 0 is enabled.

The UART is configured to operate at 9600 baud and the receive and transmit functionality is enabled.

The SPI bus is set up to operate with the memory chip. Bits are output from MSb to LSb. The clock line is set to be low when idle and operate at a frequency one-quarter the system clock. The microcontroller is set to operate as the master in communications, meaning it will always generate the clock pulses, even on a receive. Pin 4 on Port B is set as the memory chip select. The SPI interrupt is disabled.

Finally, the RTC is initialized by using Timer 2 as the counter. The Timer 2 Interrupt is enabled.

After the initialization phase, the program enters a continuous loop. In this loop the chip can operate in one of three modes depending on its current task.

The standard mode is Mode 0. In Mode 0, the microcontroller is waiting for some user input, either from an external interrupt or from a serial receive. The program uses the RTC to blink the status LED every second with the LED on for one quarter-second and off for three quarter-seconds. A capital “P” is written whenever the LED is turned on, and a lower-case “p” written whenever the LED is turned off. This allows the LabView program to detect when the board is connected to the computer.

On an External Interrupt button push, the program will jump to the external interrupt routine where the UART interrupt is disabled, the external interrupt is disabled, Timer 2 is reset to zero, and the program is set to operate in Mode 1. Pushing the External Interrupt button begins the conversion process.

Mode 1 is essentially a delay before beginning the data acquisition. This delay is programmable, so the microcontroller accesses the delay parameter in location 6 of the on-chip EEPROM memory. The number stored indicates the number of
quarter-seconds the program should wait before beginning conversions. This delay can range from 0 to 64 seconds. To indicate to the user that the interrupt has been detected and that the A/D conversions will begin shortly, the LED blink pattern is changed so the LED is on for three quarter-seconds and off for one quarter-second. After the appropriate delay, the chip is prepared for data taking. Timer 2 is disabled and Timer 1 enabled. Timer 1 is used to set the sampling rate for the conversions. The timer clock divider and the timer auto-reload value are stored in the on-chip EEPROM locations 2 through 4. The timer is enabled and the program set to enter Mode 2.

This mode is where the conversions actually take place. First, the last external memory address written to is retrieved from the on-chip EEPROM in locations 9 and 10. The memory pointer is stored in EEPROM so it is retained in case of power loss. Each new data set must begin at the beginning of a memory page (each page is 64 bytes long). This is because if the user is taking data in burst mode, the write to memory is done as an entire page and must begin at the page start. The memory write begins with a header to indicate the current data set. The data set number is stored in on-chip EEPROM location 11. The hexadecimal byte “F1” is the code used to indicate the start of a new data set. The code and the data set number are written into memory and the memory pointer updated.

The channels to be sampled are stored in location 1 of the on-chip EEPROM. A “FOR” loop cycles through the eight channels from 0 to 7. If a one is present in the nth bit of the location 1 byte, that channel is sampled, else it is skipped. The sampling begins by first setting the analog multiplexer to the correct channel. If the sampling is to be done in normal mode, that is, time between multiple channel sampling is evenly spaced, the program is halted to wait for the Timer 1 interrupt. This is done by setting variable “contin” to one, and keeping a “WHILE” loop active
until “contin” is set to zero. This occurs in the Timer 1 Compare Match Interrupt Routine. This ensures that the conversions take place based on the frequency set by Timer 1. The code then disables the Timer 1 interrupt and enables the External Interrupt 1. The A/D convert line is pulled low to start the conversion and the code again waits for the A/D converter to trigger Interrupt 1. When this happens, External Interrupt 1 is disabled, the LSB read from the A/D converter and written to memory. The A/D converter is then told to load the MSB and this is read. Since the conversion is only 12 bits in length, the highest 4 bits of the MSB are unused. To add a level of error checking to the data, the channel number is written to these four bits, thus indicating definitively the channel from which the data is taken. This byte is then written to memory. If operating in normal mode, the code loops through each channel in this manner. If operating in burst mode, the code operates in a similar fashion, with two exceptions. First, the code is only halted for the Timer 1 interrupt on the first channel. Conversions for the remaining channels occur as soon as the previous channel is completed. Second, to allow the multiple channels to be sampled as close to one another as possible, the writes to memory are only done at the completion of all the conversions. The previous conversions are stored temporarily in an array.

This process continues until the specified number of samples, stored in locations 7 and 8 of the on-chip EEPROM, are taken. The code then finalizes the data set by sending the code “F2” and then data set number. The interrupts are reset to their starting conditions and the program returned to Mode 0.

Communication with the LabView program can only occur when the code is in Mode 0 and is driven by a UART receive interrupt. When a character is received, the code first looks to see if it is one of the four recognized letters – “R” for Read settings, “W” for Write settings, “O” for Output data, or “X” for delete data. If
the received byte is not one of these four, it is ignored and the program returns to Mode 0. If a valid letter is detected, the code will check to see if the next byte is a repeat of the first. This ensures random characters do not accidentally prompt undue action. If a second letter is received, the code will write an “S” and an echo of the letter to the LabView code to indicated a Successful handshake. The LabView will then begin to write or standby to receive data.

If it requests a read of the settings, the microcontroller will output the first nine memory locations of the on-chip EEPROM. This included the channels to be sampled, sampling frequency, sampling method, initial delay, and number of samples settings. Following these nine bytes a check sum, the eight bit sum of these bytes, is sent.

If a write to settings is requested, the code will wait to receive the first data byte, and then echo it back. This continues for all nine bytes. After that, if all bytes were correctly echoed, the LabView will send a “Y” indicating that the microcontroller should write these values to memory. After a successful write, the board will respond with a “Y”.

To output the data, the board will first transmit the current memory pointer and data set count along with a check sum. This allows the LabView code to know how many bytes it will be receiving. The data is then output sixteen bytes at a time followed by a checksum. LabView uses the checksum to confirm the validity of the sixteen bytes it just received. Should a checksum not match, the data is considered to have been corrupted during transmission, and another attempt is made.

Finally, if the code to clear memory is received, the board first sends the current memory pointer to the LabView program so the user can see how many bytes will be cleared. This is to ensure that the user does not delete data that has yet to be downloaded. If the user confirms the delete, a “Y” is sent by LabView and the
microcontroller will program the memory with hexadecimal “FF”. A “C” is sent as each byte is Cleared so the LabView program can update the user on the progress of the clear. Once the clear is complete, the memory pointer and data set number are reset to zero.

3.3 Second Board
3.3.1 Requirements

The success of the experiments using microcontroller-based data acquisition systems led to a grant to develop a multi-disciplinary experiment for the Engineering Learning Center at the University of Notre Dame. As part of this experiment, a new DAQ system was to be developed. This board was developed not only for the grant project, but also to be of use in several undergraduate courses in the aerospace curriculum, specifically AME250 and AME441 Senior Design.

Drawing from the experience gained with the first board, several changes were made to the requirements for the system. The most important change was the addition of a second UART. This would allow the board to both communicate with a serial peripheral, such as a GPS, and a computer at the same time. Secondly, it was decided that the signal-conditioning portion of the board was too large and was not needed. Instruments requiring conditioning would need to do so before reaching the board. A third change was to operate the board from a single-supply. This means only positive analog voltages could be sampled by the A/D converter. While at first this seems restrictive, it should be noted that most of the devices with which the board would interface (e.g. accelerometer, pressure transducer, thermocouple) are conditioned to return a positive voltage from 0 to 5 V. It was also desired to find a microcontroller capable of generating pulse-width modulated (PWM) signals that could be used to control servos. Finally, all the unused I/O lines would be passed
out to connectors. This would allow students to easily create add on circuits that could expand the board’s capability.

3.3.2 Hardware

The ideal microcontroller was found in the Cygnal C8051F020. The chip is an 8051 derivative and has an on-chip eight-channel 12-bit A/D converter with programmable gain up to sixteen, two digital-to-analog converters (DACs), two UARTs, the Philips two-wire (I²C) protocol, five channel PWM, five 16-bit timers, 64 general I/O lines, 64 kB of EEPROM shared between code and data, and operates at clock speeds of up to 25 MHz. As a additional bonus, a later release of the chip (C8051F120) is pin-compatible with the 020, but has 128 kB of EEPROM and operates at 100 MHz. The chip’s only drawback was that it operated on 3.3 V rather than 5 V; however, all the I/O lines are 5 V tolerant. In hindsight, this turned out to be a blessing. By operating the board on a 3.3 V level, a lighter three-cell nickel-cadmium (NiCD) or nickel-metal-hydride (NiMH) battery could be used to supply power rather than the 9 V cells that had been used. Also, very lightweight lithium-ion batteries operate at 3.6 volts, meaning they could be used once they become commercially available.

Only four chips were needed to compliment the Cygnal microcontroller – A Maxim MAX604 3.3 V regulator, two Texas Instrument TPS60110 5 V charge pumps, and a Maxim MAX3224 RS-232 two-channel converter. The 3.3V regulator is low-dropout, meaning that it can maintain a regulated 3.3 V with an input voltage as low as 3.6 V. It is capable of supplying 200 mA. The charge pumps take the raw battery voltage and increase the voltage to a regulated 5 V. Each is capable of supplying 300 mA. One is used to supply the analog circuitry and the other the digital circuitry. A jumper is available to join the two together so a total of 600 mA
can be supplied if one component is particularly demanding. The RS-232 converter comes with an AutoShutdown feature that will automatically place the chip in a low power mode, thus conserving power. The Cygnal chip can also be placed in a low power mode, making it possible to “turn off” the board without unplugging the battery. In reality, the board is drawing microamps, meaning it would take years to completely drain the battery. The surface mount versions of all the chips were used in order to minimize the board’s size.

Figure 3.4 shows the location of the components as well as the general layout of the board. The board can be broken into four sections. To the left is the analog section. Three four-pin connectors line the side of the board, and a fourth is perpendicular to the board’s surface. These are where the analog instruments are attached. Each instrument is designed to be placed on a card no more than 19 mm (0.75 in.) wide. An accelerometer plug-in card is shown. Each connector has +5V from the charge pump, ground, and two A/D lines. If an instrument only outputs to one of these lines, the other should be passed out through a connector at the other end of the card. In this way, instruments can be stacked so no analog line goes unused. A 2 Ω resistor in series with the battery is located near the 3.3 V regulator. The resistor is required by the battery charging circuitry. One end of the resistor is the ground point. These are placed in the analog section so the analog ground traces are as short as possible to minimize noise on the lines. On either edge of the board are the I/O pins made available in the analog section. These include the comparator lines, four lines to the 8-bit A/D converter, the DAC lines, eight general-purpose I/O lines from a dedicated port (Port 5), the I²C lines, and access to 3.3 V, 5 V, raw battery voltage, and ground. The connectors used are on the 0.1-inch standard allowing plug in circuits to be made on easily available prototyping boards. Finally, at the bottom of the analog section is a series of eight jumpers. These jumpers pass
the incoming analog voltages to the A/D converter on the Cygnal chip. If a plug-in is designed to condition the incoming analog lines (i.e. a filter board) the designer can access the incoming voltages from the first pin, and then pass into the A/D converter the conditioned voltages through the second pins. For normal operation, jumpers must be on all the lines.

The second section of the board is the main section. This section houses the microcontroller, crystal, status LED, and the two charge pumps. The status LED is amber and is designed to be used to be as a simple status indicator (i.e. power on). The two charge pumps are each activated by a control line from the microcontroller. All unoccupied space in the main section is clad with copper to serve as a heat sink for the charge pumps. Again, I/O pins are made available on connectors along the board’s edge. The available pins are the interrupts, two indicator LED (to
be discussed in more detail below), a dedicated bit addressable port (Port 3), the
PCA pins, the CNVSTR pin to begin conversion A/D conversion, the Reset pin, two
dedicated ports (Ports 4 and 6), the two UARTs at TTL levels, the I²C lines, and the
various voltage levels. Also, a memory expansion card can be plugged into connectors
located in this section. The memory chips used are the Microchip 24FC525 I²C Serial
EEPROM chips. Each chip has a memory capacity of 64 kB. The advantage of using
the I²C lines for communicating with these chips is that no chip select lines are
required. The I²C protocol dictates that each chip on the bus has a unique address.
When communication with a particular chip is required, its address is sent over the
data line. If present, the chip will respond and communication commence. The I²C
lines are passed to all sections of the board because of the ease of communication on
the bus. With the memory chips, up to four can be individually addressed giving a
total memory expansion of 256 kB.

The third section of the board is where the user-interface plug-in card attaches
to the board. This card contains the push-buttons for External Interrupts 0 and
1, the Reset button, and two indicator LEDs, one red and one green. Each of the
push-buttons can be disabled by removing a jumper. The LEDs can be operated in
either indoor or outdoor mode by another jumper. Removing the jumper places an
additional resistor in series with the LEDs, thus decreasing the current through them
and making them dimmer. This conserves battery power, but may make the LEDs
difficult to see in bright environments, such as daylight. By placing the jumper, the
resistor is bypassed and more current is passed through the LEDs. The four jumpers
make a ring around the Reset button to protect it from accidental resets. The plug
in has one male and one female connector for mating with the board. This allows a
cable to be attached from the male connector on the board to the male connector
on the plug in so the user interface can be mounted separately from the board.
The final section is the communication section. This is where the RS-232 converter is located. Serial peripherals can be plugged into one of two rugged, locking receptacles on the bottom of the board. Each plug has +5V, the transmit and receive lines for the particular UART, and ground. A set of jumpers allows the user to select whether the TTL or RS-232 levels are output. The connectors in the communication section allow access to the I^2C lines, the UART lines at RS-232 levels, a dedicated port (Port 7), and the various voltages.

On the right side of the board, a twelve-pin connector is used to dock the board with a charging station. The twelve pins pass off the board the JTAG lines used for programming, the I^2C lines and UART0 lines for communication with a computer, and the lines associated with the battery charging.

The board measures 4.1 cm (1.6 in.) by 10.2 cm (4 in.) in size and weighs 33 g (1.16 oz). This is a fifty-percent savings in size and weight with increased functionality. With a three-cell 720 mAh NiMH battery, the package weighs 72 g (2.54 oz). A larger 1100 mAh capacity battery can be used with the system then weighting 109 g (3.84 oz).

3.3.3 Applications

There is yet no single software code available for the board since it is designed to be used for varying applications. A library of subroutines is being developed to give the students an easy way of programming the unit for their own particular application.

For the simple data acquisition application, the code would be similar to that described above for the first board. The more interesting applications apply to the area of automation, specifically in the field of unpiloted aerial vehicles (UAVs). The board’s capability to easily interface with multiple serial devices and to generate
PWM for servo control make this board especially applicable to UAVs. This will be discussed further in the Future Work section.

The ability for designers to easily access the unused I/O lines also makes this board ideally suited for educational applications. Students could easily and cheaply design their own plug-in cards on standard 0.1-inch prototyping boards, while at the same time taking advantage of the power of the Cygnal chip. This board would be an excellent tool for introducing students to microcontrollers and their applications.
4.1 Objective

The rocket project was introduced in the spring of 2001; however, additional parts were added in the spring of 2002. There were several objectives to the project. The primary focus was to demonstrate the use of microcontroller-based data acquisition systems, and these systems relevance to proper data acquisition techniques. As part of the project, students also had reinforced concepts of error estimation, calibration, and digital filtering.

Students were divided into three groups, each with four to eight members. The first group was tasked with predicting the rocket’s height. This group measured the rocket motors thrust profile using a four stain-gage beam balance. They also made a drag estimation using an Estes technical report. This prediction served as a baseline for the groups measuring the rocket’s height.

The second group was tasked with using the DAQ system to determine the rocket’s height. They used two instruments in the rocket. An accelerometer was used to measure the rocket’s acceleration along the lengthwise axis. A pressure transducer was also attached to two pressure taps located on the nose cone of the rocket to give the rocket’s airspeed. The students had to calibrate these instruments using a custom-made speed controlled centrifuge and a wind tunnel, respectively. They then had to determine the rocket’s maximum height from the flight data.
The third group used the AltiTrak system mentioned earlier to measure the rocket’s height. To gain a better understanding of the error associated with the system, they performed a series of experiments to determine the instrument’s systematic (bias), random, and repeatability errors. Further, they were asked to improve the instruments accuracy by using multiple AltiTrak’s to triangulate not only the rocket’s maximum height, but also its location, thus eliminating the straight flight assumption.

4.2 Equipment

4.2.1 Rocket

The rocket used for the two years of experiments was the same used in the undergraduate research project described above. While the rocket was not developed as part of this project, its critical nature to the experiment dictates that several aspects of its design be mentioned.

The most important feature of the design is the electronics compartment. This section of the rocket is where the all the electronics and batteries were carried. This section had to be separate from the rest of the rocket so the ejection charge did not damage the electronics when it was fired. For this reason, a wooden firewall served as a coupler between the cargo bay and the rest of the rocket. The coupler was milled by hand to join the two different sized sections together snugly. This can be seen in Figure 4.1. The entire electronics section is 5 cm (2 in.) in diameter and has a usable length of 20 cm (8 in.).

The wooden coupler was also designed to be the housing for the accelerometer. There were two requirements for mounting the accelerometer. First, it had to be in a secure location to minimize vibrations. Second, it was desired to place the accelerometer at the rocket’s center of mass, so any pitching motion of the
rocket would not be measured by the accelerometer as a centripetal acceleration. By mounting the accelerometer inside the couple, both of these goals were achieved. The accelerometer’s placement can be seen in Figure 4.2. The plastic covering was removable so as to be able to remove the accelerometer for calibration. The eye hook was used to attach the parachute recovery system. It should be noted that technically the rocket’s center of mass does vary with time as the rocket fuel is expelled, but relative to the rest of the rocket’s mass, this does not lead to a large enough shift in the center of mass to be of concern.

Another feature of the rocket is the two pressure ports in the nose cone. The nose cone is a plastic ogive that can be separated into two halves. The first pressure port was drilled and placed at the nose cone’s point to serve as a total pressure tap. A support piece in the nose cone’s design ensured that the tap remained straight during flight. The second tap was placed near the nose cone’s bottom where it would be normal to the rocket body. It should be noted that this is not technically
a static port since the flow is accelerated over the nose cone such that the velocity at this location is greater than the free-stream. This is not an issue since the rocket itself was used in calibration in the wind tunnel. The location of the two pressure taps and the pressure transducer is shown in Figure 4.3.

Finally, a pair of wires ran outside of the rocket through the wooden coupler. These wires were attached to two contacts mounted to a wooden stake that sat next to the rocket on the launch pad. These wires were connected to a push-button on the launcher that would trigger the external interrupt so as to start the data acquisition routine. An LED mounted in the wooden coupler served as a status indicator so the launch team could confirm from a safe distance that the DAQ had begun sampling. These can be seen in Figure 4.4.

In all, the rocket is 57 cm (22.5 in.) in length and weighs 213 g (7.5 oz) including the motor. With the electronics payload, the rocket weighs 323 g (11.4 oz).
Figure 4.3. Location of pressure transducer and pressure taps inside the nose cone.

Figure 4.4. Launch of rocket showing the launch pad and stake where the external interrupt wires are attached (courtesy P. F. Dunn).
4.2.2 Centrifuge

4.2.2.1 Hardware

A centrifuge needed to be constructed to calibrate the accelerometers over their full range of operation. The centrifuge system consisted of four parts – a motor and power supply, the Kerr PIC-SERVO motor-control board to control the motor, a user interface LCD display and keypad, and a microcontroller to operate the unit as a whole. The unit can be seen in Figure 4.5.

A Matsushita MPX-40C4WA motor used was. This motor features an optical encoder, meaning that the position and speed of the motor can be monitored by a controller. The motor was tested to a speed of 1200 RPM loaded, greater than any speed required for the calibration.

The Kerr PIC-SERVO motor-control board is a PIC microcontroller-based motor control system. The chipset allows the user to simply program, using the RS-232 protocol, the motor’s motion in terms of a velocity and an acceleration (or
deceleration) to that velocity. The chip then takes encoder feedback from the motor and varies the pulse width of the voltage sent to the motor to control the motor’s speed. The board features several safety features that will not be discussed for sake of brevity. The curious reader can find more information on the board’s operation in the Kerr data sheet [10].

The LCD and keypad were part of a single package solution, the Scott Edwards Electronics TRM425. The LCD screen is 4×20 character display. The display is controlled using the RS-232 protocol. The keypad is also decoded by the unit and the button push transmitted over the RS-232 lines after being preceded by a ready request. This means that before the keypad result is transmitted, a line from the unit is set high. The listener needs to detect this line and pull it low to instruct the unit to send the keypad character.

The microcontroller used was the ATMEL AT90S8535 AVR. The chip’s two functions were to take the user-inputted speed and acceleration profile and send this to the Kerr motor-control board in the proper format, and then constantly receive the motor’s position from the Kerr board so the motor speed could be calculated and displayed to the user.

The motor was encased in a wooden box, 51×51×30 cm (20×20×12 in.). The motor shaft reached into the top half of the box where the centrifuge arm attached. The back of the bottom half of the box housed the motor, power supply, power switch, and power cable. The front half served as a shelf in which to store the box that contained all the electronics. The electronics were connected to the power supply and motor by a pair of cables. The motor would be safely enclosed in the box for any operation to protect against any objects that might come loose while spinning.
4.2.2.2 Software

The largest challenge in programming the AVR was dividing time between the two peripherals (the motor-controller and the keypad interface). Because the AVR has only one UART port, the two devices had to be multiplexed. This means that when one device is communicating, the other cannot be heard. The motor-controller only responded when prompted, so receives from it could be easily predicted. The keypad, however, could be pressed at any time. Fortunately, the transmit was preceded by the ready request. When the AVR received a ready request, it would complete any communication with the motor-controller, and then switch to hear the keypad transmit.

The program implemented three basic routines. First it set up the motor-control board with the appropriate address, control gains, and safety limits. Second, it sets up the display for the LCD (this will be discussed further in the User Interface section). Finally, it continually requested the motor’s position from the motor-controller to calculate and display the motor speed, while at the same time always being alert to a key press. On a key press, the code would interpret the key press (e.g. an entered number, backspace key, enter key, etc.) and generate the proper display on the LCD. It uses the input from the user to set the desired speed and start and terminate the motor’s operation.

The motor’s speed is calculated four times a second by looking at the difference in the motor’s position from the previous read to the current read. The time is not presumed to be exactly 0.25 s since writes from the keypad may delay the request for the motor’s position. The count on the timer used to generate the request is used for the time difference.

The actual C-code can be found in Appendix A.
4.2.2.3 User Interface

The user interface was designed to be simple and relatively redundant. There are three key features to the display, as shown in Figure 4.6. The first line provides a constant display of the motors speed and features a spinning line icon when the motor is in motion. The bottom two lines are where the user can see the programmed velocity and acceleration, as well as serving as the place where these values are entered.

The user can specify two parameters. One sets the motor’s speed in revolutions per minute (RPM) and the other the acceleration in RPM per second. For example, a velocity of 200 RPM with an acceleration of 100 RPM/s would be achieved in two seconds with the motor starting from rest.

An arrow to the left of either the velocity or acceleration indicated which parameter the user is set to input. To change parameters, the user simply presses the enter key. When a number key is pressed, a cursor appears on the right side of the screen and the number being entered appears. The user can enter up to five number, although this value may be coerced by software. The backspace key removes numbers. When enter is pressed, the value is entered and the arrow moves to the next parameter. If the user has cleared all the entered numbers and then presses enter, the old value is retained.

After a number has been entered, it will blink indicating that it has not yet been loaded to the motor-controller. Pressing the “LOAD” button will program the motor controller with the new trajectory, and the numbers will stop blinking. Pressing the “START” button will instruct the motor to enforce the trajectory, changing the motor’s speed at the set acceleration rate. Finally, pressing the “STOP” button will bring the motor to a stop at the set acceleration rate.
Figure 4.6. User interface and keypad for centrifuge.
4.3 Procedure

4.3.1 Altitude Prediction

The rocket prediction group performed two analyses as part of their experiment. The first was to measure the rocket motor’s thrust profile. The second was to estimate the drag coefficient for the rocket.

4.3.1.1 Thrust Measurements

The thrust profile was measured using a strain-gage beam balance. The balance, shown in Figure 4.7, consisted of four stain-gages mounted to a steel beam with a metal can at its end. A metal collar could be placed in the can to hold the rocket motor in place during firing. Vent holes in the bottom of the collar allowed the ejection charge gases to escape to prevent the engine from shooting too high into the air at ejection.

The four stain-gages are used as the individual legs in a Wheatstone bridge circuit. They are arranged so as to cancel the effect of any twist on the bridge. Only normal deflection of the beam is recorded. The output of the bridge is conditioned

Figure 4.7. Strain-gage beam balance with rocket motor and collar (courtesy P. F. Dunn).
Figure 4.8. Thrust curve for Estes D12 series of engines (from estesrockets.com)

and made available to be passed into an acquisition system, for this experiment an oscilloscope.

The motor used for the rocket is the Estes D12-5. This classification means the rocket will produce 20 N·s total impulse, have an average thrust of 12 N, and an ejection delay of 5 s. Estes lists the motor’s maximum thrust as 33 N and thrust duration of 1.6 s. The typical thrust curve provided by Estes is shown in Figure 4.8.

Students calibrated the strain-gage bridge by hanging weights from the end of the beam. Because the beam had previously been used in numerous experiments, the linearity of the bridge output had been well established. For this reason, only two data points were needed for calibration. The conditioning circuit allowed to both zero the bridge and to vary the output slope. The output was zeroed for zero-thrust. The output was set so 1 V was equal to 10 N of thrust.

The output was recorded on a Fluke PM3380B digital oscilloscope. The oscilloscope was set to automatically trigger at a non-zero voltage level (i.e. when the
rocket was fired). A negative half-second delay was used to record the time before firing as a zero-reference in case the bridge had drifted. Four rocket motors were tested. The first two were recorded with a temporal resolution such that the entire rocket firing, from ignition to ejection charge, was recorded. For the final two, the resolution was increased and the recording window reduced so only the thrust producing portion of the firing was recorded.

4.3.1.2 Drag Estimation

The second part of the experiment required the students to estimate the drag on the rocket. The Estes TR-11 Model Rocket Technical Report [11] provided all the required information for making the estimation. For brevity’s sake, the full explanation of the drag calculations will not be presented. The analysis assumes that each element of the rocket has an associated drag and that these can be summed along with an interference drag to give the total drag on the rocket. The drag data is based on both theoretical and experimental results. The sources for drag on the rocket are the nose cone, body tube, fins, launch lug, and separation at the rocket’s base.

4.3.2 On-board Measurement

A second group was tasked with determining the rocket’s maximum altitude by using on-board sensors and the microcontroller-based DAQ system. Two instruments were used, an Analog Devices ADX150 accelerometer and a Motorola MPX5010 differential pressure transducer.

4.3.2.1 Accelerometer Calibration

The accelerometer has an advertised range of ±5 g’s (earth’s gravitational acceleration) with an output voltage of 0 to 5 V. The zero-offset, slope, and range
were verified using the centrifuge. The board was programed to take 20 samples over 1 second for each sampling set. A ten-second delay was used from button press to sampling start. The students would load the desired motor speed based on the target calibration acceleration, begin the acquisition routine, start the motor, wait for acquisition to end, then stop the motor and begin the process over for the next data point.

4.3.2.2 Pressure Transducer Calibration

The pressure transducer has an output range of 0 to 10 kPa, or for standard atmosphere conditions, a pitot-static differential pressure corresponding to a airspeed of 127 m/s (neglecting compressibility effects). This is well above the expected maximum velocity of the rocket, however, for simplicity and weight considerations, the output was not rescaled.

The front half of the rocket was mounted in-line with the flow in the undergraduate, 16×16 inch, in-draft wind tunnel at the University of Notre Dame. A pitot-static probe attached to a previously calibrated pressure transducer was used to measure the free-stream. The voltage output from the nose-cone transducer was recorded for various free-streams.

4.3.2.3 Launch Procedure

The launches were carried out on Riehle Fields on the Notre Dame campus. The fields are used for various intramural sports and provide a large launch area with few obstructions.

The electronics were placed inside the rocket, and the rocket with motor placed on the launch pad. The two wires for the external interrupt were attached to contacts mounted on the stake as described above. The launch control box operator would ensure that the area immediately surrounding the rocket was clear and then begin
a count down from fifteen. At T-13 seconds, the external interrupt was pressed, beginning the data acquisition routine. The routine called for a ten-second initial time delay. The LED would flash once a second, letting the ground team know that the acquisition routine had begun. At T-3 seconds, the data acquisition would begin and the LED would light solid. Once the operator verified the start of data acquisitions, the launch would commence at T-0.

The board samples data for a total of fifteen seconds, three on the pad and twelve in flight. The accelerometer and pressure transducer are each sampled twenty times a second.

When the rocket returns to the ground, the DAQ board is attached to a computer and the data downloaded using the LabView interface program.

4.3.3 AltiTrak Error

A third group was tasked with finding the error associated with the Estes Alti-Trak system to see if the on-board measurements were an improved way of measuring the rocket’s altitude. Several experiments were devised to determine the total error in the system.

4.3.3.1 Static Error

The first source of error presumed to exist in the AltiTrak system was a bias error from the user looking through the sight incorrectly. To determine this error, each member of the group made several measurements of an object of a known height from a known distance. The top of Hesburgh Library on the Notre Dame campus was taken to be the reference point. Blueprints of the building and surrounding landscaping were obtained from University Engineering to obtain precise measurements of the library’s height and distance from it at various locations.
4.3.3.2 Dynamic Error

A second source of error was believed to come as a result of the weight swinging while it is locked in place. To test this, several smaller rockets were launched and simultaneous measurements were taken on four AltiTraks. This experiment also demonstrated another source of error – that not every user would measure the same point in the rocket’s trajectory as being the maximum location.

4.3.3.3 Total Error

The error found from these two experiments was combined with the resolution error of the instrument (1 degree) to give the total error. The summed-squares method was used to combine these errors.

4.4 Results

4.4.1 Altitude Predication

The beam-balance is subject to one detrimental property that can distort the results. The beam will tend to oscillate at its natural frequency, especially when it is subjected to a large instantaneous pulse, similar to the initial thrust peak of the rocket motor. To eliminate these oscillations, the students used a digital notch filter.

The filter simply transformed the temporal signal into a discrete Fourier space. The Fourier components associated with the beam vibration, which take the form of a spike in the Fourier intensity plot, can then be removed, and the signal recreated. The results of this filtering is shown in Figure 4.9.

Before a numerical integration could be performed, a typical thrust curve needed to be defined from the four sets for motor thrust data. While the same rocket motor is used, each individual engine demonstrates different properties, most noticeably,
maximum thrust, time to maximum thrust, and total burn time.

In order to average these engines, two transforms were performed to align the data. First, the maximum thrust times were all aligned. This ensured that the thrust peak would not be lost or rounded off. The second transform cast the thrust curve after the maximum thrust peak into non-dimensional time. The time scales were set so the burn out time was one time unit. It should be noted that a similar transform was not performed for the time before the maximum thrust peak because defining an exact ignition time proved difficult. Also, the thrust data has already appeared to have collapsed into a common form in that region.

The thrust data could now be averaged. The later half of the thrust curve was restored to a real time scale by multiplying it by the average of the four burn times. Figure 4.10 shows the various thrust curves as they are transformed and averaged.

The thrust profile measured by the students agrees in form with that provided
Figure 4.10. Thrust data in various stages the of transform to obtain the average thrust profile.

by Estes, but showed a lower maximum thrust, average thrust, and total impulse. Table 4.4 compares the key parameters of the rocket engines with Estes’ predictions.

Results of the drag estimate gave a drag coefficient of 0.029 referenced to the rockets projected area to the flow.

Using a simple trapezoidal integration scheme, the predicted maximum height of the rocket given by the students was 100 m.

4.4.2 On-board Measurement
4.4.2.1 Accelerometer Calibration

Data was taken with the accelerometer in both a positive and negative orientation for accelerations from 0 to 10 g’s in 1 g intervals. The data recorded is shown in Figure 4.11. The accelerometer clearly demonstrates linear behavior from -9 to +6 g’s, better than the advertised range of the accelerometer. Beyond that, the output
is pegged at the maximum output voltage.

To make use of the larger negative range, the accelerometer was positioned in the rocket so accelerations in the upward direction would be recorded as negative accelerations. It should also be noted that the accelerometer is subject to gravitational acceleration, such that when it is placed vertically, it measures a 1 g acceleration. This needs to be subtracted from the measured accelerations.

### 4.4.2.2 Pressure Transducer Calibration

As expected, the pressure transducer voltage output varied linearly with pressure difference, meaning it varies as the square of the velocity. This can be seen in Figure 4.12. The transducer demonstrated linearity up to the maximum tunnel velocity of 40 m/s, which is near the maximum velocity expected for the rocket.

### 4.4.2.3 Flight Data

Over the two years of conducting the experiment, there were numerous sets of flight data recorded. Two sets of data, one typical and one interesting in its

---

**TABLE 4.1**

ROCKET MOTOR PROPERTIES

FOR MEASURED, AVERAGED, AND SPECIFIED DATA

OF THE ESTES D12-5 ENGINE

<table>
<thead>
<tr>
<th>Motor</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Average</th>
<th>Estes Specs [9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Thrust (N)</td>
<td>22.32</td>
<td>22.53</td>
<td>24.70</td>
<td>23.32</td>
<td>23.22</td>
<td>32.90</td>
</tr>
<tr>
<td>Average Thrust (N)</td>
<td>9.34</td>
<td>9.56</td>
<td>9.19</td>
<td>9.36</td>
<td>9.36</td>
<td>12.00</td>
</tr>
<tr>
<td>Burn Time (s)</td>
<td>1.94</td>
<td>1.73</td>
<td>1.73</td>
<td>1.82</td>
<td>1.80</td>
<td>1.60</td>
</tr>
<tr>
<td>Total Impulse (N-s)</td>
<td>18.21</td>
<td>16.63</td>
<td>15.95</td>
<td>17.03</td>
<td>16.96</td>
<td>20.00</td>
</tr>
</tbody>
</table>
Figure 4.11. Results of the accelerometer calibration.

Figure 4.12. Results of the pressure transducer calibration.
probative value, are presented here, along with a comparison of the predicted and measured data.

Figure 4.13 shows the typical acceleration and velocity measurements taken onboard the rocket. There are four distinct phases to the flight data. First is when the rocket is sitting on the pad. Then the thrust phase is evident from the positive acceleration and increasing velocity. Next the coasting phase can be seen from the negative acceleration (as the result of drag and gravity) and decreasing velocity. Finally, the ejection charge is seen as the large acceleration pulse. This comes about from the hot gases propelling the electronics section forward so the parachute can be deployed. During the recovery phase the electronics bay is upside down, making the measured accelerations shown meaningless.

When recorded acceleration data is integrated, one finds excellent agreement with the recorded velocity data as shown in Figure 4.14. Integrating either of these
Figure 4.14. Comparison of integrated acceleration with measured velocity.

time series again will give the height as a function of time (assuming the rocket remains vertical). The altitude profile is shown in Figure 4.15.

The measured altitude is typically 100 m, demonstrating excellent agreement with the predicted height.

4.4.2.4 Crash Investigation

During the second launch of the spring 2001 experiments, the rocket became separated from the parachute and the electronics package free-fell from 100 m to the ground. Surprisingly, the electronics remained intact and the data recorded during the flight was recovered.

The parachute showed signs of fire with the shroud lines singed and the plastic chute showing several areas where it was melted. The reason for the fire was not known since recovery wadding had been placed in the rocket just as it had been on all pervious flights.
The recorded data provided two hints as to the cause of the crash. It showed that the ejection charge had gone off after only a second and a half of coast rather than the usual five seconds and that it had been much more violent than the typical ejection charge. This is shown in Figure 4.16. It was speculated that if the ejection charge had been set off early, that not only would hot gasses be propelled into the ejection wadding, but also the unused delay propellant. The propellant would be hot enough to burn through the shroud lines and parachute. While this investigation was rather trivial in nature, it did demonstrate to the students another application of on-board data acquisition.

4.4.2.5 Comparision to Predictions

A comparision of the measured and predicted acceleration and velocity are shown in Figure 4.17. There is excellent agreement between the two.

The prediction was based on the averaged thrust profile, the rocket weight, and
Figure 4.16. Comparison of typical acceleration data with acceleration recorded on the crash flight.

Figure 4.17. Comparison measured and predicted acceleration and velocity.
the estimated drag coefficient. Because the drag force on the rocket is a function of rocket’s velocity, a numerical integration was necessary. A simple trapezoidal scheme was employed.

Despite the excellent agreement between the data, it is clear the drag is under-predicted as the rocket decelerates faster during the coast phase than is predicted by the drag estimation. The effect of the predicted drag is negligible when compared to that of gravity. The most likely cause for this error is that the drag estimation assumes the rocket is traveling straight at a zero angle of attack to the flow. In reality, the rocket is wobbling, being stabilized by a lift force from the fins. This results in two additional sources of drag. The first is the lift-induced drag from the fins. The second is the flow separation behind the body tube at an angle of attack.

4.4.3 AltiTrak Error
4.4.3.1 Static Error

The measurements taken of the library showed two types of systematic (bias) errors in the device. The first was the expected bias error of each individual user. Measurements taken at various distances from the building demonstrated a measurement scatter of two degrees. The standard deviation of the measurements was approximately one degree.

A second, unexpected error was also noted. When comparing the measured angles to the expected angles, the measurements were consistently lower. Careful observation of the weight during the measurement process revealed that the locking mechanism was moving the weight, thus creating a bias error with the instrument itself, an average of two degrees. If corrected for, this error could be eliminated, but since the magnitude of this shift varies with each trial, no correction was made and the effect simply included as another error source.
4.4.3.2 Dynamic Error

The results of the dynamic calibration demonstrated the expected random error. Each user measured the rocket’s height differently, again with a standard deviation of approximately one degree. It was found that the error could be minimized if each user agreed upon a set event to be considered the maximum altitude (e.g. the ejection charge firing), and if each user allowed the AltiTrak to settle before releasing the locking mechanism so as to reduce inertial effects.

4.4.3.3 Total Error

Combining the four sources of error (user bias, instrument bias, random, and resolution) the total error of the AltiTrak was found to be approximately two degrees.

Since the error in the rocket’s height is desired, it is necessary to develop an arrangement where the rocket’s height can be determined with a minimum amount of error. The system employed would be to use two AltiTraks a known distance and angle apart. By measuring not only the rocket’s elevation angle, but also its azimuth angle relative to each station, the rockets altitude can be shown to be

\[
h = \frac{\ell \tan \alpha_1}{\cos \theta_1 + \sin \theta_1 \cot \theta_2}.
\]

(4.1)

Figure 4.18 shows the arrangement used to derive this equation.

To find the error in \(h\), the error in \(\alpha_1\), \(\theta_1\), and \(\theta_2\) must be known. Assuming the error in the \(\theta\)s is the same as in \(\alpha\) (the error of the AltiTrak), the height error can be found to be
Clearly, the error is dependent on the actual values of $\alpha_1$, $\theta_1$, and $\theta_2$. Taking the value for all of these to be 45 degrees and selecting $\ell$ such that $h$ will be 100 m (the altitude achieved by our rocket), the error in $h$ is found to be 7.8 m, or 7.8%. This is a best case scenario. If the rocket, again at 100 m, were measured by the Station 1 with an elevation of 30 degrees, the error becomes 12.6 m. This shows that proper planning of the measurement locations can reduce error, but not sufficiently to give
reliable results.

In the end, the AltiTrak system is a good measurement system for its cost and intended application. For more precise measurements though, the system is not reliable.

4.5 Student Feedback

All the students who participated in the rocket project were enthusiastic and motivated to perform the experiment. The students were anxious to delve further into the project and offered several suggestions on how to improve the project.

The first concern was with the assumption that the rocket travels straight up. The assumption only appears when subtracting the gravitational acceleration for the measured acceleration. Because the component is the cosine of the angle, even for moderate angles, the error is not significant. Several methods were discussed on how the rocket’s pitching motion could be measured.

The first method was to use gyros; however, solid-state gyros are the only ones small enough to fit inside the rocket, and these are often unreliable. A new gyro released by Analog Devices with low-drift may make this an option for future experiments.

The second method would be to use an array of accelerometers. Since accelerometers not located at the center of gravity would detect centrifugal accelerations, by using multiple accelerometers one could resolve the linear and angular accelerations. Again, error becomes the greatest obstacle to making such an arrangement practical. Precise measurement of the accelerometers relative position and minimizing noise on the accelerometer outputs would all be necessary to obtain any confidence in the calculations.

The groups investigating the AltiTrak error presented alternate methods for
ground-based tracking. One group proposed using radio signal intensity to triangulate the rocket’s positions, but error again proved to make this solution not practical. The one option for rocket tracking that might prove successful is the use of a video camera to follow the rocket. The rocket’s size makes it impossible to see on a screen in a wide field of view. In order to follow the rocket in a tight field of view, the camera’s position relative to the ground must be known.

Two students from the 2002 class are currently developing a system where a miniature camera is controlled by a pair of servos and used to follow an object. Currently, the system is set to track a GPS unit, but could be modified to track an object on a screen if more powerful vision recognition software were used.
CHAPTER 5

BOAT PROJECT

5.1 Objective

The boat project was introduced in the spring of 2002. The objective of the project was to demonstrate an application for microcontroller-based data acquisition other than the aerospace-related rocket project, since both aerospace and mechanical engineers were enrolled in the course.

The students were asked to investigate the surface temperature distribution of an on-campus lake. The temperature profile of this lake was thought to be particularly interesting because warm effluent from the campus power plant is dumped into the lake. The effects can clearly be seen in the winter time when the other campus lake freezes over while the lake investigated remains unfrozen.

To accomplish this, students operated a custom-made boat carrying a thermocouple and a Global Positioning System (GPS) unit. The DAQ system was used to record both the temperature readings from the thermocouple and the position from the GPS unit.

5.2 Equipment

5.2.1 Boat

A remote control boat was needed to carry all the required electronics on to the lake. After examining the required payload capacity, as well as to minimize cost, it was decided to build a custom boat rather than buy a kit boat. Rather
than manufacturing a boat hull, model airplane floats were used in a twin pontoon configuration. The engine and payload were carried on a raised 0.24 cm (.093 in.) phenylic sheet. The electronics were housed in a water-resistant box. A geared, electric Astro Challenger Cobalt motor was used to power the boat. A drive shaft connected to motor to the propeller. Figure 5.1 shows the finished boat.

It should be noted that this design was by no means meant to be a final, polished design. It was meant to be a low-cost means for carrying out the experiment on a trial basis. If successful, a more streamlined design would be employed.

5.2.2 Control

A standard Futaba 4-channel transmitter was used to control the boat. Only two channels were used – throttle and rudder. A five-channel receiver was carried on the boat and connected to a Tekin speed controller and a Futaba S3004 servo.
The speed controller regulates the flow of current from the 12V battery pack to the motor. To insure maximum torque, the full 12V is always supplied to the motor, but is pulsed to control the motor speed. The servo was connected to two rudders, one on each pontoon, through a series of push rods.

5.2.3 Electronics

There were three main components to the electronics used in this experiment: the GPS, the thermocouple, and the DAQ system. They will be described below.

5.2.3.1 GPS

The GPS unit used was a Garmin GPS35-LVS. It is approximately 5 × 10 × 2.5 cm (2 × 4 × 1 in.) in size and weighs 113 g (4 oz) The unit operates on regulated 5 V and draws approximately 100 mA. The location in latitude and longitude, height above sea level, time, and number of visible satellites are output once a second at RS-232 levels at a user-selected baud rate of up to 9600. The GPS gives latitude and longitude with accuracy of one-tenth of a minute, and one-tenth a meter in altitude.

Experiments conducted by Gariel Torres show that the unit has a absolute error of approximately 5 m. This was found by observing the drift in the GPS location over a long period of time while the unit is held still. Over short periods of time, the unit is not subject to much drift. This means from second-to-second, the unit can be trusted to give an accurate relative reading.

5.2.3.2 Thermocouple

An Omega type-K (chromel-alumel) grounded thermocouple was used with an in-house circuit designed by Joel Preston to make the water temperature measurements. The thermocouple circuit used the Analog Devices AD595 chip, which inter-
faces with type K thermocouple and conditions the output so each degree Centigrade corresponds to 10 mV.

The students wanted to expand the experiment to see if there was any correlation between the air temperature and the water temperature. For this reason, a second thermocouple was carried on top of the boat to record the air temperature.

5.2.3.3 DAQ System

The DAQ system used for the experiment was the first board as described above. Only slight modifications needed to be made to the board itself. First, connectors for the thermocouple and a connector for the GPS needed to be made. Secondly, the gain for the thermocouple inputs was set to 10 to give to 100 mV/°C resolution.

The more significant modification came in the code. The first modification involved adding a subroutine to interpret the data coming from the GPS. For this application, only the latitude, longitude, and number of satellites (to check the validity of the received signal) were needed. Based on the location of the lake, only the fractional components of the minutes needed to be stored since the degrees and minutes would not change anywhere on the surface of the lake. This meant that the latitude and longitude could be stored using only four bytes combined. The number of satellites was stored in one byte.

Rather than starting the conversion routine with an external interrupt as had previously been done, the conversion routine began when the first byte from the GPS was received. In this manner, the temperature data and GPS coordinates were synchronized. Sixteen samples were taken with each coordinate and they were all stored so the mean and standard deviation could be calculated later. Since both the air and water temperature were recorded, with each sample taking up 2 bytes, another 64 bytes of data were recorded.
In all, 69 bytes of data were recorded per second. Based on the available memory, data could be recorded for just under eight minutes. This was approximately the same time one battery pack would last at full throttle.

Because there was only one serial port available, data had to be outputted over the same port that was receiving data from the GPS. For this reason, the same baud rate had to be used. Two modes of operation were available. They were selected by pressing the interrupt button. In mode 0, memory could be read and cleared just as in the standard program. In mode 1, the incoming GPS data was interpreted and stored along with the temperature measurements. Once the data taking was completed, the board was placed back into mode 0 and the data retrieved.

5.3 Procedure

Two groups, each with four members, were tasked with the experiment. The first step was to confirm the Analog Device’s chip calibration. This was done by measuring the voltage output after submerging the thermocouple in water of two different temperatures, one ice water and one room temperature, both of which had mercury thermometers in them giving the actual temperature. The calibration matched.

Next, students were asked to determine the dynamic response to the thermocouple to see what time lag, if any, was associated with the device. This was found by subjecting the thermocouple to a sudden change in temperature (dunking in ice water) and examining the response. Defining a time constant as

$$\ln \left[ \frac{V(t) - V_f}{V_i - V_f} \right] = \frac{1}{\tau} t.$$  \hspace{1cm} (5.1)

As Figure 5.2 shows, the thermocouple’s time constant is 0.4 s, meaning that for a sudden temperature change it will reach ninety-nine percent of the temperature
change in two seconds. Since steep temperature changes were not anticipated in the lake, the dynamic response of the thermocouple is, at worst, on the same order of accuracy as the GPS.

Originally the students were asked to map the entire lake, but due to time constraints, only two sections of the lake could be mapped, the south-east and north-east corners. This was sufficient, since the two areas provided a broad range of depth, sun-lighting conditions, and distance from the plant return pipes (located in the south-east corner). The students drove the boat around the areas of interest.

5.4 Results

Figure 5.3 shows the temperature recorded along the boat paths. White indicates the warmest temperatures of approximately 20 °C (68 °F) and black the coldest temperatures of approximately 14 °C (58 °F). The air temperature was nearly con-
stant at 21 °C (70 °F). The standard deviation in the thermocouple measurement was never more than 0.16 °C (0.3 °F).

The students concluded that there were two major factors affecting the temperature distribution. First was water depth. The areas near shore are always warmer than the deeper water. The second factor was the effect of direct sunlight. The south part of the lake was shaded by trees on the shore, whereas the north part of the lake was in direct sunlight on the day measurements were taken. For this reason, the north part of the lake was warmer than the south part.

Of interest is that the return water from the power plant seemed to have no effect on the temperature. In fact, some of the coldest temperatures were measured over the plume. It was later discovered that the return from the plant was not a long chimney-like structure as was presumed, but rather only a short pipe at the very bottom of lake. This would give the warm-return water plenty of time to diffuse, and thus make its effects not directly visible.
5.5 Student Feedback

Overall, the students gave very favorable comments on the experiment and their experiences. They offered numerous suggestions for future classes.

First, they thought the students should be more involved in the construction of the boat and electronic circuits. While this is not practical for the curriculum at present, it might be possible in other courses to allow the students to have a more hands-on experience with the design and manufacturing of measurement systems.

Other suggestions focused on streamlining the experimental apparatuses. Using larger batteries, repositioning the motor to maximize thrust, and creating a rake of thermocouples to cover more of the lake with each pass will all be employed when this experiment is conducted in the future.

Finally, the students made suggestions to increase the breadth of the experiment. They expressed a desire to sample not just the surface temperature distribution, but also the temperature profile with depth. Additions of ultrasonic depth sensors and current measuring devices were suggested.

In all cases, it was clear that the students were willing to probe deeper and expend more effort on understanding the very simple problem posed to them. Such personal involvement with the experiment was the exact goal of using microcontroller-based data-acquisition systems to move experiments outside of the laboratory.
6.1 Conclusions

The project to introduce mechanical and aerospace engineering students to using microcontrollers for data acquisition proved exceedingly successful. Two classes of sophomore engineers used the systems to conduct research in a real-world environment and gained insight into the principles of data acquisition and the potential applications of microcontrollers. Several students went on to conduct undergraduate research projects to further explore the system’s potential.

The work has also impacted other fields of study in the department. Because of the success of the rocket project, the senior design course in the aerospace curriculum made use of microcontrollers to perform flight testing on the model aircraft designed by the seniors. The system recorded parameters such as airspeed, altitude, and GPS location. The system also explored the area of flight automation by using the measured parameters as inputs to control a control algorithm.

Thanks to a grant for the General Electric Corporation, students will have the opportunity to learn about microcontroller based data acquisition even sooner in their academic careers. A module based around the second board will be available to students to explore in the Engineering Learning Center. The module will allow the students to calibrate an accelerometer using a centrifuge and then to use the accelerometer to study the motion of a pendulum.
6.2 Future Work

The results of this project will continue to influence the undergraduate curriculum for years to come. The system designed will be used in AME250 again as well as in AME441: Senior Design. The board is robust enough that it is the author’s hope that other courses will make use of its capabilities as well.

The AME250 curriculum is going to be organized so that each experiment conducted during the semester will build upon the final goal of measuring the rocket’s height. To start the year, students will build their own kit rockets, try to guess how high they will go, and then measure the heights using the AltiTrak. Then throughout the semester, students will conduct experiments to allow them to better predict and to better measure the rocket’s height. They will calibrate the accelerometers, measure the motors’ thrust profiles, and even measure the rocket’s drag directly in a wind tunnel. At the end of the semester, they will launch the rockets again, this time instrumented, and compare their original predictions and measurements with their final, more educated, ones.

The AME441 class will continue to use the system as a “black box” recording flight characteristics. In the near future though, the goal will be to use the system for automated flight. In this manner, much more precise flight testing can be conducted by removing the human pilot from the loop.

The board itself will continue to be perfected, with the new version including features such as a battery current monitor. The main development work in the future will be to develop instrument plug-in cards as the need arises. Eventually, the author would like to see the system made available as a commercial product so other institutions may add similar experiments to their curriculums.
APPENDIX A

CENTRIFUGE CONTROLLER CODE

#include <io8515.h>
#include <macros.h>
#include <eeprom.h>

#include <io8515.h>
#include <macros.h>
#include <eeprom.h>

#pragma interrupt_handler ext_zero:2
#pragma interrupt_handler uart_rec:12

unsigned char contin, mode;
unsigned char chan_sel, time_delay, samp_freqh, samp_freql, pre_scale, stage;
unsigned char data_mode, mem_pointh, mem_pointl, set_count, count_down;
unsigned char num_samph, num_sampl, samp_count;
unsigned char l, stat_byte, stat_byte2, checksum;
unsigned char pos_byte, pos_byte2, pos_byte3, pos_byte4;

void display_screen(void);
char display[80];
char arrow_pos = 0;
char mult_sel = 0;
signed char curs_pos = -1;
char curs_on = 0;
char vel_valid = 0;
char acc_valid = 0;
char data_in[5];
int target_rpm = 0;
int accel = 0;
unsigned char i = 0;
unsigned char k = 45;
char run = 0;
char write_ok = 0;
char count = 3;
char read_error = 0;

void TransmitByte( unsigned char data ) {
    while ( !(USR & (1<<UDRE)) );
    /* wait for empty transmit buffer */
    UDR = data; /* start transmission */
}
unsigned char ReceiveByte( void ) {
    char rec_udr;
    TCNT0 = 0x00;
    TIFR &= ~0x02;
    TCCR0 = 0x03;
    while (!(USR&(1<<RXC)))&&((0x02 & TIFR))); // wait for incoming data
    TCCR0 = 0x00;
    if (TIFR & 0x02) {
        rec_udr = 0x00;
        read_error = 1;
        PORTB &= ~0x20;
    } else {
        rec_udr = UDR;
    }
    return rec_udr;
}

void delay( unsigned int x, unsigned int y ) {
    unsigned int a, b;
    for(a=0;a<x;a++)
        for(b=0;b<y;b++);
}

void BinaryOut( unsigned char data ) {
    unsigned int n, p;
    p = 128;
    for(n=0;n<8;n++)
        if (p&data) {
            TransmitByte(49);
        } else {
            TransmitByte(48);
        }
        p = p/2;
}

void ext_zero() {
    GIMSK &= ~0x40; // Disable Interrupt 0
    write_ok = 1;
    PORTB &= ~0x40;
}

void SendString(char string2[])
{
    unsigned char w;
    for(w=0;w<(strlen(string2));w++)
    {
        TransmitByte(string2[w]);
    }
}
void SendNumber(long num, char length) {
    unsigned char w, z, send;
    unsigned long p;
    p = 1;
    z = 1;
    for (w = 2; w <= length; w++) {
        p = p*10;
    }
    for (w = 1; w <= length; w++) {
        send = num/p;
        if (((send) == 0) && (z == 1) && (w < length)) {
            TransmitByte(32);
        } else {
            TransmitByte(send + 48);
            z = 0;
        }
        num = num%p;
        p = p/10;
    }
}

void uart_rec() {
    unsigned char rec_byte, trans_byte, check_sum, check_int, byte_out;
    unsigned int count, count2, count_limit;
    unsigned int n, p, data_out;
    char str_out[5];
    long vel_long, acc_long;
    UCR &= ~0x80; //Disable UART Receive Complete Interrupt
    rec_byte = UDR; //UDR = UART receive byte register
    if (rec_byte == 35) {
        curs_on = 0;
        TransmitByte(4);
        if (curs_pos > -1) {
            data_out = 0;
            p = 1;
            for (n = 1; n <= curs_pos; n++) {
                data_out = data_out + p*data_in[curs_pos-n];
                data_in[curs_pos-n] = 0;
                p = p*10;
            }
            curs_pos = -1;
            if (arrow_pos == 0) {
                target_rpm = data_out;
                vel_valid = 1;
            } else {
                accel = data_out;
                acc_valid = 1;
            }
        } else {
            target_rpm = data_out;
            vel_valid = 1;
        }
    } else {
        target_rpm = 0;
        vel_valid = 0;
    }
}
if (arrow_pos == 0) {
    TransmitByte(16);
    TransmitByte(104);
    TransmitByte(32);
    TransmitByte(13);
    TransmitByte(126);
    arrow_pos = 1;
} else {
    TransmitByte(16);
    TransmitByte(104);
    TransmitByte(126);
    TransmitByte(13);
    TransmitByte(32);
    arrow_pos = 0;
}
else if ((rec_byte > 47) && (rec_byte < 58)) {
    curs_on = 1;
    if (curs_pos == -1) {
        TransmitByte(16); // Go to Vel num position
        TransmitByte(64+40+14);
        SendNumber(target_rpm,5); // Write Velocity Target
        TransmitByte(16); // Go to Accel num position
        TransmitByte(64+60+14);
        SendNumber(accel,5); // Write Accel
        TransmitByte(16);
        TransmitByte(118+arrow_pos*20);
        SendString(" "); // five spaces
        curs_pos = 0;
    }
    if (curs_pos == 5) {
    } else {
        data_in[curs_pos] = rec_byte - 48;
        TransmitByte(16);
        TransmitByte(122+arrow_pos*20-curs_pos);
        for (n=0;n<=curs_pos;n++) {
            TransmitByte(data_in[n]+48);
        }
        if ((rec_byte == 48) && (curs_pos == 0)) {
        } else {
            curs_pos = curs_pos + 1;
        }
    }
    TransmitByte(16);
    TransmitByte(122+arrow_pos*20);
} else if (rec_byte==42) {
    if (curs_pos > 0) {
        curs_pos = curs_pos - 1;
        data_in[curs_pos] = 0;
        TransmitByte(16);
        TransmitByte(122+arrow_pos*20-curs_pos);
        TransmitByte(32);
        for (n=0;n<curs_pos;n++) {
            TransmitByte(data_in[n]+48);
        }
        TransmitByte(16);
        TransmitByte(122+arrow_pos*20);
    }
    if (curs_pos == 0) {
        TransmitByte(48);
        TransmitByte(16);
        TransmitByte(123+arrow_pos*20);
        TransmitByte(8);
        curs_pos = -1;
    }
} else if (rec_byte == 65) {
    PORTD &= ~0x10;
    delay(20,500);

    l = 1;
    while(l || read_error) {
        l = 1;
        read_error = 0;
        vel_long = target_rpm;
        vel_long = vel_long*1118 + vel_long*55/100;
        acc_long = accel;
        acc_long = acc_long*5727/10000;

        TransmitByte(0xAA); // Header Byte
        TransmitByte(0x01); // Address Byte
        TransmitByte(0x94); // Command Byte
        TransmitByte(0x36);
        check_sum = 0xCB;
        for (n=1;n<=4;n++) {
            byte_out = vel_long%256;
            check_sum = check_sum + byte_out;
            TransmitByte(byte_out);
            vel_long = vel_long/256;
        }
        for (n=1;n<=4;n++) {
            byte_out = acc_long%256;
            check_sum = check_sum + byte_out;
            TransmitByte(byte_out);
            acc_long = acc_long/256;
        }
    }
}
PORTD |= 0x10;
delay(20,500);
vel_valid = 0;
acc_valid = 0;

} else if (rec_byte == 66) {
PORDD &= ~0x10;
delay(20,500);
if (run) {
    l = 1;
    while (l || read_error) {
        l = 1;
        read_error = 0;
        TransmitByte(0xAA); // Header Byte
        TransmitByte(0x01); // Address Byte
        TransmitByte(0x0B); // Command String (Clear Sticky Bytes)
        TransmitByte(0x0C); // Checksum
        stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
    }
}
}

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0x05); // Command Byte (Start Motor)
    TransmitByte(0x06); // Check Sum
    stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
PORTD |= 0x10;
delay(20,500);
}
else if (rec_byte == 67) {
PORTD &= ~0x10;
delay(20,500);
}

l = 1;
while(l || read_error) {
l = 1;
read_error = 0;
TransmitByte(0xAA); // Header Byte
TransmitByte(0x01); // Address Byte
TransmitByte(0x17); // Command Byte (Stop Motor)
TransmitByte(0x09); //
TransmitByte(0x21); // Check Sum
stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
PORTD |= 0x10;
delay(20,500);
}

void main() {
unsigned char count, chan_shift, n;
unsigned int curr_add, samp_set, mem_loc, page_loc, time, time_old;
unsigned long pos, pos_old, act_vel, d_pos;

/* PORT SETUP */
DDRA |= 0xFF; // Enable Port A 0-7 as output
DDRB |= 0xFF; // Enable Port B 0-7 as output
DDRC |= 0xFF; // Enable Port C 0-7 as output
DDRD |= 0xF0; // Enable Port D 4-7 as output

/* INTERRUPT SETUP */
SREG |= 0x80; // Enable Global Interrupt
GIMSK &= ~0x40; // Disable Interrupt 0
GIMSK &= ~0x80; // Disable Interrupt 1
MCUCR |= 0x0A; // (tied with next line)
MCUCR &= ~0x05; // Sets falling edge to trigger Interrupt 0
// Sets falling edge to trigger Interrupt 1

/* UART SETUP */
UART_TRANSMIT_ON(); // Enable UART Transmitting
UART_RECEIVE_ON(); // Enable UART Receiving
UBRR = 25; // Set Baud Rate to 9600
UCR &= ~0x80; // Disable UART receive cpld interrupt

delay(500,500);
delay(500,500);

PORTD |= 0x10; // Set PD4 High (select LCD Display)
delay(20,500);

TransmitByte(27);
TransmitByte(101);
TransmitByte(191);
TransmitByte(12);
TransmitByte(1);
TransmitByte(2);
TransmitByte(87);
TransmitByte(65);
TransmitByte(73);
TransmitByte(84);
TransmitByte(3);
TransmitByte(27);
TransmitByte(101);
TransmitByte(0);
delay(20,500);

UBRR = 12; // Set Baud Rate to 19200
PORTD &= ~0x10; // Set PD4 Low (select Motor Controller)
delay(20,500);

for (n=1;n<=16;n++) {
    TransmitByte(0x00); // Clears Buffer
}

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0x00); // Address Byte
    TransmitByte(0x21); // Command String (Set Address)
    TransmitByte(0x01); // Set Individual Address
    TransmitByte(0xFF); // Set Group Address
    TransmitByte(0x21); // Checksum
    stat_byte = ReceiveByte();
    stat_byte2 = ReceiveByte();
    l = stat_byte&0x02;
}

l = 1;
while(l) {
    l = 1;
    read_error = 0;
    TransmitByte(0x00); // Address Byte (Group Address)
    TransmitByte(0x1A); // Command String (Change Baud)
    TransmitByte(0x81); // New Baud Rate (9600)
    TransmitByte(0x9A); // Checksum
    // No Status Byte Returned
    delay(20,500);
    UBRR = 25;
    delay(20,500);
    TransmitByte(0x00); // Address Byte
    TransmitByte(0x0E); // Command Byte (Null Command)
    TransmitByte(0x0F); // Checksum
    stat_byte = ReceiveByte();
    stat_byte2 = ReceiveByte();
    l = stat_byte&0x02;
}

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0x0B); // Command String (Clear Sticky Bits)
    TransmitByte(0x0C); // Checksum

    stat_byte = ReceiveByte();
    stat_byte2 = ReceiveByte();
    l = stat_byte&0x02;
}

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    PORTD |= 0x10;
    delay(20,500);
    TransmitByte(12);
    TransmitByte(1);
    SendString(" GENERAL WARNINGS ");
    SendString(" ");
    SendString(" Over Current: ");
    if (stat_byte&0x04) {
        SendString("Yes");
    } else {
        SendString(" No");
    }
    SendString(" Motor Power On: ");
    if (stat_byte&0x08) {
        SendString("Yes");
    } else {
        SendString(" No");
    }
    l = (stat_byte&0x04)|(~(stat_byte&0xF7));
    delay(20,500);

    if(l || read_error) {
        PORTD &=~0x10;
        delay(20,500);

        TransmitByte(0xAA); // Header Byte
        TransmitByte(0x01); // Address Byte
        TransmitByte(0x0B); // Command String (Clear Sticky Bits)
        TransmitByte(0x0C); // Checksum

        stat_byte = ReceiveByte();
    }
stat_bytes2 = ReceiveByte();

TransmitByte(0xAA); // Header Byte
TransmitByte(0x01); // Address Byte
TransmitByte(0x0E); // Command Byte (Null Command)
TransmitByte(0x0F); // Checksum

stat_bytes = ReceiveByte();
stat_bytes2 = ReceiveByte();
delay(100,500);
}
}

PORTD &= ~0x10;
delay(20,500);

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0xE6); // Command String (Set Gains)
    TransmitByte(0x64); //
    TransmitByte(0x00); //
    TransmitByte(0x08); //
    TransmitByte(0x03); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0xFF); //
    TransmitByte(0x00); //
    TransmitByte(0xA0); //
    TransmitByte(0x0F); //
    TransmitByte(0x01); //
    TransmitByte(0x00); //
    TransmitByte(0xE5); // Checksum

    stat_bytes = ReceiveByte();
    stat_bytes2 = ReceiveByte();
    l = stat_bytes&0x02;
}

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
TransmitByte(0x01); // Address Byte
TransmitByte(0x12); // Command String (Set Return Bytes)
TransmitByte(0x01); //
TransmitByte(0x14); // Checksum

stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0x94); // Command String (Load Trajectory)
    TransmitByte(0x36); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0x00); //
    TransmitByte(0xCB); // Checksum

    stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0x00); // Command String (Reset Position)
    TransmitByte(0x01); // Checksum

93
stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
pos = 0;

l = 1;
while(l || read_error) {
    l = 1;
    read_error = 0;
    TransmitByte(0xAA); // Header Byte
    TransmitByte(0x01); // Address Byte
    TransmitByte(0x17); // Command String (Stop Motor, Enable Move)
    TransmitByte(0x09); //
    TransmitByte(0x21); // Checksum

    stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte = ReceiveByte();
stat_byte2 = ReceiveByte();
l = stat_byte&0x02;
}
delay(500,500);
delay(500,500);
delay(500,500);

PORTD |= 0x10; // Set PD4 High (LCD Display)
PORTB |= 0xFF; // set LED high = on
TransmitByte(12); //clear screen,hide cursor,to position 1
TransmitByte(4);
TransmitByte(1);
SendString(" CURRENT RPM: 0 "); //send a 4
SendString("-----------"); //send a 4
SendString(" TARGET RPM : 0 "); //send a 4
SendString(" ACCEL RPM/s: 0 "); //send a 4

while(1) {
    PORTD |= 0x10;
    TransmitByte(7);
    UCR &= ~0x80;
    TransmitByte(4); // Cursor Off

    PORTD |= 0x10;
if (curs\_on == 0) {
    TransmitByte(16); // Go to Vel num position
    TransmitByte(64+40+14);
    SendNumber(target\_rpm,5); // Write Velocity Target
    TransmitByte(16); // Go to Accel num position
    TransmitByte(64+60+14);
    SendNumber(accel,5); // Write Accel
}

TransmitByte(1);
if (run) {
    TransmitByte(124);
    PORTB &= 0xFE;
    PORTB |= 0x08;
} else {
    TransmitByte(32);
    PORTB |= 0x0F;
}

UCR |= 0x80;
delay(333, 500);
UCR &= ~0x80;
if (run) {
    TransmitByte(1);
    TransmitByte(47);
    PORTB &= 0xFD;
    PORTB |= 0x01;
}

UCR |= 0x80;
delay(333,500);
UCR &= ~0x80;
if (curs\_on) { // Turn on cursor if necessary
    TransmitByte(16);
    TransmitByte(122+arrow\_pos*20);
    TransmitByte(5);
} else {
    if (vel\_valid) {
        TransmitByte(16);
        TransmitByte(64+40+14);
        SendString(" "); // five spaces
    }
    if (acc\_valid) {
        TransmitByte(16);
        TransmitByte(64+60+14);
        SendString(" "); // five spaces
    }
}
if (run) {
    TransmitByte(1);
    TransmitByte(45);
PORTB &= 0xFB;
PORTB |= 0x02;
}
UCR |= 0x80;
delay(333, 500);
UCR &= ~0x80;
if (run) {
  TransmitByte(1);
  TransmitByte(128);
  PORTB &= 0xF7;
  PORTB |= 0x04;
}
UCR |= 0x80;
delay(292, 500);
PORTB &= ~0x20;
while (write_ok);
PORTB |= 0x20;
UCR &= ~0x80;
PORTB &= ~0x80;
pos_old = pos;
UCR &= ~0x80;
PORTD &= ~0x10;
delay(20,500);
TransmitByte(0xAA); // Header Byte
TransmitByte(0x01); // Address Byte
TransmitByte(0x0E); // Command Byte (Null Command)
TransmitByte(0x0F); // Checksum
checksum = 0;
read_error = 0;
stat_byte = ReceiveByte();
pos_byte = ReceiveByte();
pos_byte2 = ReceiveByte();
pos_byte3 = ReceiveByte();
pos_byte4 = ReceiveByte();
stat_byte2 = ReceiveByte();

time = TCNT1L;
time = time + TCNT1H*256;
TCCR1B = 0x04;
TCNT1H = 0x00;
TCNT1L = 0x00;

checksum = stat_byte + pos_byte + pos_byte2 + pos_byte3 + pos_byte4;
pos = pos_byte4;
pos = 256*pos + pos_byte3;
pos = 256*pos + pos_byte2;
pos = 256*pos + pos;
PORTD |= 0x10;
delay(20,500);
if ((checksum == stat_byte2) && (read_error == 0)) {
    time = time + time_old;
    if (pos == pos_old) {
        run = 0;
    } else {
        run = 1;
    }
}

d_pos = pos-pos_old;
act_vel = (d_pos*468 + d_pos*75/100)/time;
TransmitByte(16);
TransmitByte(64+14);
SendNumber(act_vel,5);
time_old = 0;
} else {
    pos = pos_old;
    time_old = time + time_old;
    read_error = 0;
}
PORTD |= 0x10;
UCR |= 0x80;
PORTB |= 0x80;
}
APPENDIX B

FIRST BOARD CODE

```c
#include <io8535.h>
#include <macros.h>
#include <eeprom.h>

#pragma interrupt_handler ext_0:2
#pragma interrupt_handler ext_1:3
#pragma interrupt_handler timer_2:4
#pragma interrupt_handler timer_1:7
#pragma interrupt_handler uart_rec:12

unsigned char contin, mode;
unsigned char chansel, timedelay, sampfreqh, sampfreql, pre_scale, stage;
unsigned char data_mode, mem_pointh, mem_pointl, set_count, count_down;
unsigned char num_samph, num_sampl, samp_count;

static unsigned char data[16];

unsigned char i = 0;
unsigned char k = 45;

void TransmitByte( unsigned char data ) {
    while ( !(USR & (1<<UDRE)) );
    /* wait for empty transmit buffer */
    UDR = data; /* start transmission */
}

unsigned char ReceiveByte( void ) {
    while ( !(USR & (1<<RXC)) ); /* wait for incoming data */
    /* return the data */
    return UDR;
}

void timer_1() {
    contin = 0;
}

void timer_2() {
```

98
contin = 0;
}

void Delay( unsigned int x, unsigned int y ) {
    unsigned int a, b;
    for(a=0; a<x; a++) {
        for(b=0; b<y; b++);
    }
}

unsigned char SPIConfirm() {
    PORTB &= ~0x10; // Make Chip Select low
    SPDR = 0x05; // Send RDSR instruction (0000 X101)
    while (~SPSR); // Wait for previous write to be cpld
    SPDR = 0x00; // Generate clock pulses via blank send
    while (~SPSR); // Wait for previous write to be cpld
    PORTB |= 0x10; // Make Chip Select high
    return (SPDR&0x01); // Return 0 if write complete, else return 1
}

void SPIWriteInit( unsigned int add ) {
    PORTB &= ~0x10; // Make Chip Select low
    SPDR = 0x06; // Send WREN instruction (0000 X110)
    while (~SPSR); // Wait for previous write to be cpld
    PORTB |= 0x10; // Make Chip Select high
    PORTB &= ~0x10; // Make Chip Select low
    SPDR = 0x02; // Send WRITE instruction (0000 0010)
    while (~SPSR); // Wait for previous write to be cpld
    SPDR = (add/256); // Send first half of address
    while (~SPSR); // Wait for previous write to be cpld
    SPDR = (add%256); // Send second half of address
    while (~SPSR); // Wait for previous write to be cpld
}

void BinaryOut( unsigned char data ) {
    unsigned int n, p;
    p = 128;
    for(n=0; n<8; n++) {
        if (p&data) {
            TransmitByte(49);
        } else {
            TransmitByte(48);
        }
        p = p/2;
    }
}

void SPIWrite( unsigned char data ) {
    SPDR = data; // Send byte to be written
while (~(SPSR&0x7f)); //Wait for previous write to be cpld

void SPIWriteClose( ) {
    PORTB |= 0x10; //Make Chip Select high
    while(SPIConfirm()); //Confirm Write Cycle is cpld
}

void SPIReadInit( unsigned int add ) {
    PORTB &= ~0x10; //Make Chip Select low
    SPDR = 0x03; //Send READ instruction (0000 0011)
    while (~(SPSR&0x7f)); //Wait for previous write to be cpld
    SPDR = (add/256); //Send first half of address
    while (~(SPSR&0x7f)); //Wait for previous write to be cpld
    SPDR = (add%256); //Send second half of address
    while (~(SPSR&0x7f)); //Wait for previous write to be cpld
}

unsigned char SPIRead( ) {
    SPDR = 0x00; //Generate clock pulses via blank send
    while (~SPSR&0x80); //Wait for previous write to be cpld
    return SPDR; //Return output byte
}

void SPIReadClose( ) {
    PORTB |= 0x10; //Make Chip Select high
}

void ext_0() {
    GIMSK &= ~0x40; //Disable Interrupt 0
    UCR &= ~0x80; //Disable UART Receive Interrupt
    TIMSK &= ~0x80; //Disable Timer 2 Interrupt
    TCCR2 &= ~0x03; //Stop Timer 2
    TCNT2 = 0x00; //Reset Timer 2 to zero
    PORTC &= 0xEF; //Set CS low
    PORTC &= 0xFE; //set HBEN low
    mode = 1;
    contin = 0;
    TCCR2 |= 0x03; //Start Timer 2 in PCK2/32 Mode
    TIMSK |= 0x80; //Enable Timer 2 Interrupt
    while (ASSR&0x07); //Ensure above writes are cpld
}

void ext_1() {
    contin = 0;
}

void uart_rec() {
    unsigned char rec_byte, trans_byte, check_sum, check_int;
unsigned int count, count2, count, limit;

UCR &= ~0x80;  // Disable UART Receive Complete Interrupt
GIMSK &= ~0x40;  // Disable Interrupt 0
TIMSK &= ~0x80;  // Disable Timer 2 Interrupt
TCCR2 &= ~0x03;  // Stop Timer 2
TCNT2 = 0x00;   // Reset Timer 2 to zero
PORTD &= ~0x40; // Turn LED on

check_sum = 0;
rec_byte = UDR;  // UDR = UART receive byte register
if (rec_byte == 82) {  // If send command is "R"
    rec_byte = ReceiveByte();  // Wait for Second "R"
    TransmitByte(83);  // Transmit "S"
    TransmitByte(rec_byte);  // Transmit "R"
    for (count=1; count<9; count++) {
        trans_byte = EEPROMread(count);  // transmit settings
        TransmitByte(trans_byte);
        check_sum = trans_byte + check_sum;
    }
    TransmitByte(check_sum);
} else if (rec_byte == 87) {  // If send command is "W"
    rec_byte = ReceiveByte();  // Wait for Second "W"
    TransmitByte(83);  // Transmit "S"
    TransmitByte(rec_byte);  // Transmit "W"
    chan_sel = ReceiveByte();
    TransmitByte(chan_sel);
    pre_scale = ReceiveByte();
    TransmitByte(pre_scale);
    samp_freqh = ReceiveByte();
    TransmitByte(samp_freqh);
    samp_freql = ReceiveByte();
    TransmitByte(samp_freql);
    time_delay = ReceiveByte();
    TransmitByte(time_delay);
    data_mode = ReceiveByte();
    TransmitByte(data_mode);
    num_samph = ReceiveByte();
    TransmitByte(num_samph);
    num_sampl = ReceiveByte();
    TransmitByte(num_sampl);
    if (ReceiveByte() == 89) {  // 89 = "Y", if y received
        while(EEPROMwrite(1,chan_sel));  // then write to memory
        while(EEPROMwrite(2,pre_scale));
        while(EEPROMwrite(3,samp_freqh));
        while(EEPROMwrite(4,samp_freql));
        while(EEPROMwrite(5,time_delay));
        while(EEPROMwrite(6,data_mode));
        while(EEPROMwrite(7,num_samph));
        while(EEPROMwrite(8,num_sampl));
    }
TransmitByte(89);  //and send a "Y"
}
} else if (rec_byte == 79) {// 79 = "O"
    rec_byte = ReceiveByte();  // Wait for Second "O"
    TransmitByte(83);  // Transmit "S"
    TransmitByte(rec_byte);  // Transmit "O"
    check_sum = 0;
    for (count=9; count<12 ; count++) {
        trans_byte = EEPROMread(count);
        TransmitByte(trans_byte);
        check_sum = trans_byte + check_sum;
    }
    TransmitByte(check_sum);
    count_limit = (EEPROMread(9)) * 256 + EEPROMread(10);
    check_sum = 0;
    SPIReadInit(0);
    for (count=0; count<count_limit; count++) {
        trans_byte = SPIRead();
        TransmitByte(trans_byte);
        check_sum = trans_byte + check_sum;
        if (((count+1)%16)==0) {
            TransmitByte(check_sum);
            check_sum = 0;
        }
    }
    SPIReadClose();
    TransmitByte(check_sum);
} else if (rec_byte == 88) {// 88 = "X"
    rec_byte = ReceiveByte();  // Wait for Second "X"
    TransmitByte(83);  // Transmit "S"
    TransmitByte(rec_byte);  // Transmit "X"
    TransmitByte(EEPROMread(9));
    TransmitByte(EEPROMread(10));
    TransmitByte(83+rec_byte+EEPROMread(9)+EEPROMread(10));
    if (ReceiveByte() == 89) {// Clear only if confirm "Y" rcvd.
        count_limit = EEPROMread(9) * 256 + EEPROMread(10);
        for (count=0; count<count_limit; count++) {
            SPIWriteInit(count);
            SPIWrite(0xFF);
            SPIWriteClose();
            TransmitByte(67);
        }
    } while(EEPROMwrite(9,0x00));  // Reset Memory Pointer MSB
    while(EEPROMwrite(10,0x00));  // Reset Memory Pointer LSB
    while(EEPROMwrite(11,0x00));  // Reset Data Sets
    TransmitByte(89);  // Write "Y" to confirm erase
} else {
PORTD |= 0x40; // Turn LED off
GIMSK |= 0x40; // Enable Interrupt 0
TCCR2 |= 0x03; // Start Timer 2 in PCK2/32 Mode
TIMSK |= 0x80; // Enable Timer 2 Interrupt
while (ASSR&0x07); //Ensure above writes are cpld
UCR |= 0x80; // Enable UART Receive cpld Interrupt

void main() {
    unsigned char count, chan_shift, l, m;
    unsigned int curr_add, samp_set, mem_loc, page_loc;

    /* PORT SETUP */
    DDRD |= 0x40; // Enable LED as output
    PORTD |= 0x40; // Turn LED off
    DDRA &= ~0xFF; // Enable Port A 0-7 as input
    DDRC |= 0x3F; // Enable Port C 0-5 as output
    PORTC |= 0x20; //Set RD high (set low to begin conversions)
    PORTC &= ~0x01; // Set HBEN low
    PORTC &= ~0x10; // Set CS low
    PORTC &= ~0x0E; // Set Channel Select to 0 (000)

    /* INTERRUPT SETUP */
    SREG |= 0x80; // Enable Global Interrupt
    MCUCR |= 0x0E; // (tied with next line)
    MCUCR &= ~0x01; //Sets falling edge to trigger Interrupt 0
                   //Sets rising edge to trigger Interrupt 1
    GIMSK |= 0x40; // Enable Interrupt 0
    GIMSK &= ~0x80; // Disable Interrupt 1

    /* UART SETUP */
    UART_TRANSMIT_ON(); // Enable UART Transmitting
    UART_RECEIVE_ON(); // Enable UART Receiving
    UBRR = BAUD9600; // Set Baud Rate to 9600

    /* SPI SETUP */
    DDRB |= 0xB0; // Set SCK, MOSI, and SS as Outputs
    SPCR = 0x50; // Disable: SPI Interrupt (Bit 7)
                // Enable: SPI Functionality (Bit 6)
                // Set: Data Order MSB->LSB (Bit 5)
                // Chip as Master (Bit 4)
                // Clock Polarity low when idle(Bit 3)
                // Clock Phase Option 0 (Bit 2)
                // Clock Rate Select fclk/4(Bits 1&0)
    PORTB |= 0x10; // Chip Select set high
    PORTB &= ~0x10; // set Chip Select low
    SPDR = 0x06; // send WREN instruction (0000 X110)
while (~(SPSR|0x7f)); // wait for previous write to be cpld
PORTB |= 0x10; // set Chip Select high

/** RTC SETUP */
TIMSK &= ~0xC0; // Disable OCIE2 and TOIE2 interrupts
ASSR |= 0x08; // Set Timer 2 for asynchronous operation
TCNT2 = 0x00; // Reset Timer 2 to zero
OCR2 = 0xFF; // Set Timer 2 Compare Register to FF
TCCR2 = 0x03; // Disable Pulse Width Modilator (Bit 6)
// Disconnects Timer 2 from output pin
// OC2 (Bits 5 & 4)
// Sets Timer 2 to reset on Compare
// Match (Bit 3)
// Start Timer 2 in PCK2/32 Mode
// (Bits 2, 1 & 0)
TIMSK |= 0x80; // Enable Timer 2 Interrupt
while (ASSR&0x07); // Ensure above writes are complete

/** TIMER 1 SETUP */
TCCR1A = 0x00;
TCNT1H = 0x00;
TCNT1L = 0x00;
OCR1AH = 0x00;
OCR1AL = 0x00;
TCCR1B = 0x00;
TCCR2 = 0x03; // Start Timer 2 in PCK2/32 Mode
TIMSK |= 0x80; // Enable Timer 2 Interrupt
while (ASSR&0x07); // Ensure above writes are cpld
UCR |= 0x80; // Enable UART Receive Interrupt
GIMSK |= 0x40; // Enable Interrupt 0
GIMSK &= ~0x80; // Disable Interrupt 1

mode = 0;

/** MAIN LOOP */
while(1) {
    contin = 1;

    switch (mode) {
    case 0 :
        count = 4;
        UCR |= 0x80; // Enable UART Receive Interrupt
        GIMSK |= 0x40; // Enable Interrupt 0
        while (count) {
            if (mode) {
                count = 0;
            } else {
        
104
while(contin);
contin = 1;
count = count - 1;
if (count) {
    PORTD |= 0x40; // Turn LED off
    TransmitByte(112); // Transmit "p" on UART line
} else {
    PORTD &= ~0x40; // Turn LED on
    TransmitByte(80); // Transmit "P" on UART line
}
break;
case 1 :
    PORTD |= 0x40;
count = EEPROMread(6);
while (count) {
    while(contin);
    contin = 1;
count = count - 1;
    if (count%4) {
        PORTD &= ~0x40;
        TransmitByte(80);
    } else {
        PORTD |= 0x40;
        TransmitByte(112);
    }
}
TIMSK &= ~0x80; // Disable Timer 2 Interrupt
TCCR2 &= ~0x03; // Stop Timer 2
TCNT2 = 0x00; // Reset Timer 2 to zero
GIMSK |= 0x80; // Enable Interrupt 1
TCCR1A = 0x00;
TCNT1H = 0x00;
TCNT1L = 0x00;
OCR1AH = EEPROMread(3);
OCR1AL = EEPROMread(4);
TCCR1B = (EEPROMread(2)|0x08);
TIMSK |= 0x10; // Enable Timer 1A Compare Match Interrupt
mode = 2;
break;
case 2 :
    if (EEPROMread(10)%64) {
        curr_add = EEPROMread(9)*256 + ((EEPROMread(10)/64)+1)*64;
    } else {
        curr_add = EEPROMread(9)*256 + EEPROMread(10);
    }
EEPROMwrite(11,(EEPROMread(11)+1));
contin = 0;
page_loc = 0;
SPIWriteInit(curr_add);
if (EEPROMread(5)) {
    SPIWrite(EEPROMread(11));
curr_add = curr_add + 1;
SPIWriteClose();
SPIWriteInit(curr_add);
SPIWrite(0xF1);
curr_add = curr_add + 1;
SPIWriteClose();
} else {
    SPIWrite(EEPROMread(11));
SPIWrite(0xF1);
curr_add = curr_add + 64;
page_loc = 2;
}
samp_set = EEPROMread(7)*256 + EEPROMread(8);
while (samp_set>0) {
    chan_shift = 0x01;
    mem_loc = 0;
    for (count=0;count<8;count++) {
        if (EEPROMread(1)&chan_shift) {
            if (EEPROMread(5)) {
                PORTC &= ~0x0E;
PORTC |= (count*2);
GIMSK &= ~0x80; // Disable Interrupt 1
while(contin);
GIMSK |= 0x80; // Enable Interrupt 1
contin = 1;
PORTC &= 0xDF; // Set RD low, begins conversion
TIMSK &= ~0x18; // Disable Timer 1 Interrupt
while(contin);
TIMSK |= 0x18; // Enable Timer 1 Interrupt
contin = 1;
SPIWriteInit(curr_add);
l = PINA;
SPIWrite(l); // Read least significant bits
curr_add = curr_add + 1;
PORTC |= 0x20; // set RD high
PORTC |= 0x10; // set CS high
PORTC |= 0x01; // set HBEN high
SPIWriteClose();
PORTC &= 0xEF; // Set CS low
PORTC &= 0xCF; // Set RD low
SPIWriteInit(curr_add);
m = PINA;
m = (count*16)||(m&0x0F);
SPIWrite(m); // Read most significant bits
curr_add = curr_add + 1;
PORTC |= 0x20; // set RD high
PORTC |= 0x10; // set CS high
PORTC &= 0xEF; // Set CS low
PORTC &= 0xFE; // set HBEN low
SPIWriteClose();

} else {
    PORTC &= ~0x0E;
    PORTC |= (count*2);
    if (count==0) {
        GIMSK &= ~0x80; // Disable Interrupt 1
        while(contin);
        contin = 1;
        GIMSK |= 0x80; // Enable Interrupt 1
    }
    PORTC |= 0xDF; // Set RD low, begin conversion
    TIMSK &= ~0x18; // Disable Timer 1 Interrupt
    while(contin);
    TIMSK |= 0x18; // Enable Timer 1 Interrupt
    mem_loc = mem_loc + 1;
    contin = 1;
    l = PINA;
    PORTC |= 0x20; // set RD high
    PORTC = 0x10; // set CS high
    PORTC = 0x01; // set HBEN high
    data[mem_loc] = l; // Read least significant bits
    PORTC &= 0xEF; // Set CS low
    PORTC &= 0xCF; // Set RD low
    mem_loc = mem_loc + 1;
    m = PINA;
    m = (count*16) | (m&0x0F);
    PORTC |= 0x20; // set RD high
    PORTC = 0x10; // set CS high
    data[mem_loc] = m; // Read most significant bits
    PORTC &= 0xFE; // set HBEN low
    PORTC &= 0xFE; // Set CS low
}

} else {

}

chan_shift <<= 1;

for (count=0;count<mem_loc;count++) {
    if (page_loc==0) {
        SPIWriteInit(curr_add);
        curr_add = curr_add + 64;
    }
    SPIWrite(data[count+1]);
page_loc = page_loc + 1;
if (page_loc==0) {
    SPIWriteClose();
    page_loc = 0;
}
}  
samp_set = samp_set - 1;

GIMSK &= ~0x80;  // Disable Interrupt 1
TIMSK &= ~0x10;  // Disable Timer 1 Interrupt
if (page_loc==0) {
    SPIWriteInit(curr_add);
    SPIWrite(EEPROMread(11));
    curr_add = curr_add + 1;
    SPIWriteClose();
    SPIWriteInit(curr_add);
    SPIWrite(0xF2);
    curr_add = curr_add + 1;
} else {
    SPIWrite(EEPROMread(11));
    SPIWrite(0xF2);
    page_loc = page_loc + 2;
    for (count=0;count<(64-page_loc);count++) {
        SPIWrite(0xFF);
    }
}
SPIWriteClose();
EEPROMwrite(9,(curr_add/256));
EEPROMwrite(10,(curr_add%256));

TCCR1B = 0x00;
TCNT1H = 0x00;
TCNT1L = 0x00;
OCR1AH = 0x00;
OCR1AL = 0x00;
TCCR2 |= 0x03;  // Start Timer 2 in PCK2/32 Mode
TIMSK |= 0x80;  // Enable Timer 2 Interrupt
while (ASSR&0x07);  //Ensure above writes are cpld
UCR |= 0x80;  // Enable UART Receive Interrupt
GIMSK |= 0x40;  // Enable Interrupt 0
mode = 0;
break;
default :
    TCCR2 |= 0x03;  // Start Timer 2 in PCK2/32 Mode
    TIMSK |= 0x80;  // Enable Timer 2 Interrupt
    while (ASSR&0x07);  //Ensure above writes are cpld
    UCR |= 0x80;  // Enable UART Receive Interrupt
    GIMSK |= 0x40;  // Enable Interrupt 0
GIMSK &= ~0x80; // Disable Interrupt 1
mode = 0;
BIBLIOGRAPHY


