AN ABSTRACT OF THE THESIS OF

Justin B. Goins for the degree of Honors Baccalaureate of Science in Electrical and Computer Engineering presented on May 28, 2010. Title: Design and Development of a Low-Cost, High Resolution, High Mobility Acceleration Data Logger.

Abstract approved: ____

Roger Traylor

To understand the forces exerted on structures during tsunamis, model cities are subjected to human-generated tsunamis. The study of vertical evacuation, wherein individuals could take refuge in the upper levels of strong buildings, has resulted in a growing need for a low cost wireless acceleration data logger. Such a device could be placed inside the walls of scale buildings to measure the forces encountered. This document describes the development and implementation of a suitable module. The major implication of this project is that researchers can more accurately record the forces that are exerted on structures and better understand the types of construction that best withstand tsunamis.

Key Words: acceleration, wireless, network, logger, IEEE 802.15.4

Corresponding e-mail address: justin.goins@lifetime.oregonstate.edu

Design and Development of a Low-Cost, High Resolution,

High Mobility Acceleration Data Logger

by

Justin B. Goins

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in
Electrical and Computer Engineering (Honors Scholar)

Presented May 28, 2010
Commencement June 2010

ACKNOWLEDGEMENTS

I would like to acknowledge my senior design partners Andrew Christensen and Joseph Gross. Our senior design project is the result of a collaborative effort and we have all worked hard to produce the final result. This is a project in which we have all invested countless hours and this thesis would not be possible without their help over the past nine months.

I would also like to thank each of the people on my thesis committee. It has been my honor to know Dr. Kartikeya Mayaram, Roger Traylor, and Don Heer. I am deeply indebted for the inspiration and knowledge that they have provided me. It is my hope that I can someday help others by sharing the knowledge that I have gained from them at Oregon State University.

GROUP CONTRIBUTIONS

The design presented in this paper is the culmination of nine months of work in senior design class. Collectively, the three team members have invested nearly a thousand hours into developing the final product. In order to divide the work equally, each member was assigned a particular task. The following chart shows the initial division of labor.

Project Block	Assignee
Battery Charger	Andrew Christensen
Voltage Regulators	Joseph Gross
16-bit ADC	Justin Goins
Visual Indicators	Joseph Gross
Transceiver	Andrew Christensen
Microcontroller	Justin Goins
Microcontroller Firmware	Joseph Gross
Accelerometers	Andrew Christensen

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX EQUATIONS

LIST OF APPENDIX FIGURES

DEDICATION

This thesis is dedicated to my parents. They have provided constant support and encouragement throughout the past four years. My mom has always placed a strong emphasis on the importance of acquiring a good education and it is primarily her influence that has helped me progress to where I am today. Without her help I would never be graduating with an Honors Degree from Oregon State University. Thank you.

INTRODUCTION

Researchers at the O. H. Hinsdale Wave Research Laboratory are currently investigating the feasibility of vertical tsunami evacuation, in which members of the public would be evacuated upwards in buildings, rather than being relocated horizontally to higher ground. The use of such methods could potentially save lives by reducing the time needed for individuals to reach safety.

In order to better understand which building designs are best able to withstand a tsunami, researchers are subjecting scale buildings to model tsunami waves. If small data loggers could be embedded inside model tsunami debris and structures, the data loggers could record the acceleration caused by the tsunami waves and by the subsequent collisions of debris. This data could then be transmitted back to a computer system where researchers and engineers could analyze the recorded values.

In the fall of 2007, the laboratory commissioned a group of students to design a suitable low-cost wireless acceleration data logger. The data logger met most of the system requirements but it exhibited several flaws that prevented it from being used in professional research. In particular, the poor wireless transmission range (≤20 ft) and a faulty power supply limited its usefulness.

In September of 2010, the project was assigned to the author's senior design group. The team was tasked with developing an improved version of the module. A list of customer requirements was provided along with the stipulation that the device must utilize the existing computer graphical user interface. The senior design team consisted of the author and two additional undergraduate engineering students. The design presented in this paper is the result of our work.

CUSTOMER REQUIREMENTS

The O. H. Hinsdale Wave Research Laboratory presented the senior design team with a list of project requirements and potential improvements over the existing design. The customer requirements are as follows:

- The final product must be able to operate under the following water and air temperatures (1.7 - 35º C and 7.2 - 23.9º C, respectively).
- The datalogger must be waterproof to a depth of 10 feet.
- The final product should exhibit a neutral buoyancy.
- The module must be able to survive accelerations of 98.1 m/s^2.
- Acceleration data must be recorded at a rate of at least 50 times per second.
- The device must begin saving acceleration measurements within 2 ms of a 5 volt pulse issued by the Wave Lab's equipment.

Table 1: Customer Requirements

Requirement	Specific Details
Air Temperature	1.7 - 35 º Celsius
Water Temperature	7.2 - 23.9 º Celsius
Buoyancy	≥ 0 N
Waterproof	≤ 10 ft below the water's surface
Impact	≤ 98.1 m/s^2
Volume	≤ 12x9x5 cm (540 cm^3)
Trigger Time	≤ 2 ms
Sampling Frequency	≥ 50 Hz

TOP LEVEL BLOCK DIAGRAM

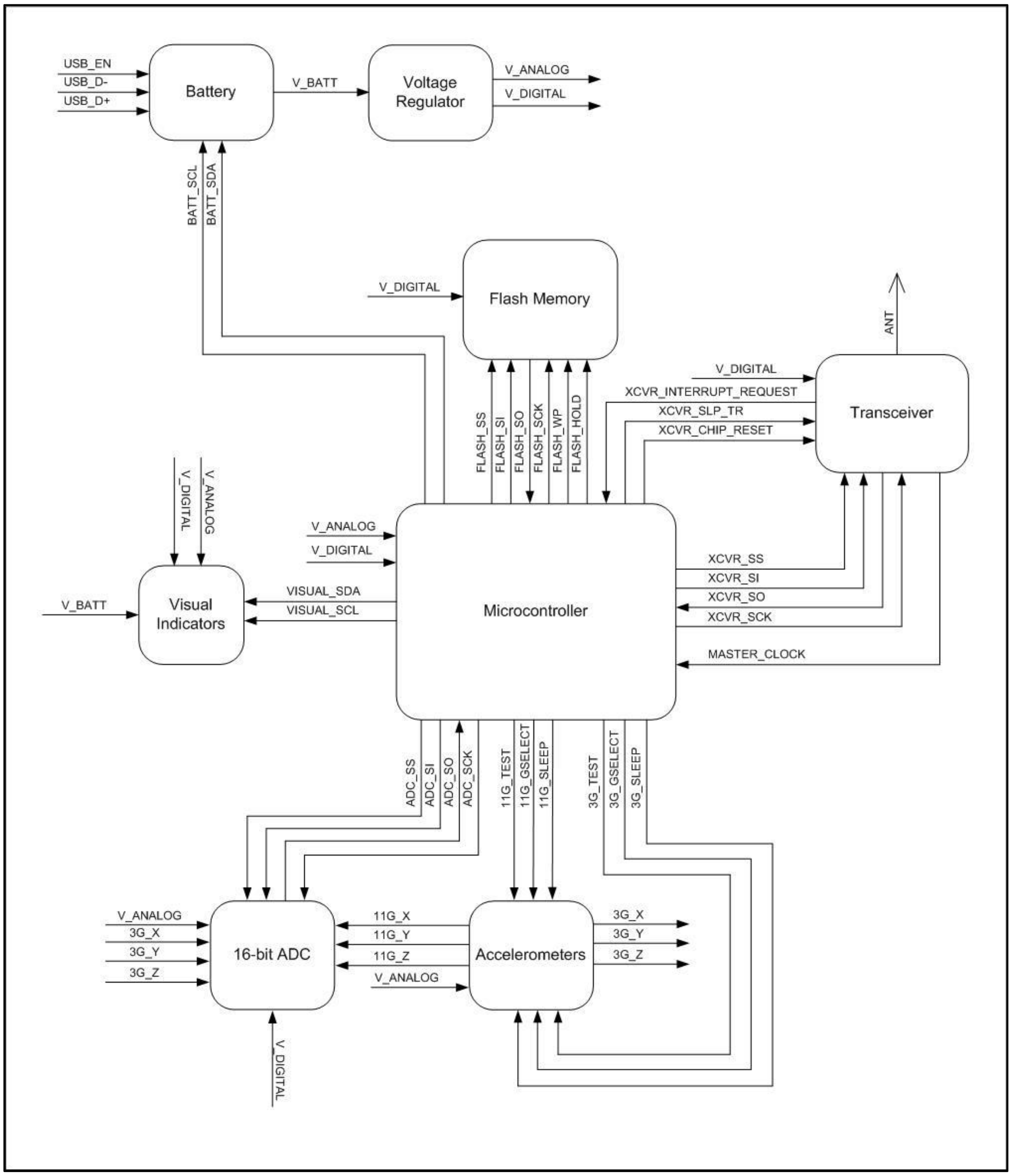

Figure 1: Top Level Block Diagram

DESIGN INFORMATION

Battery Block

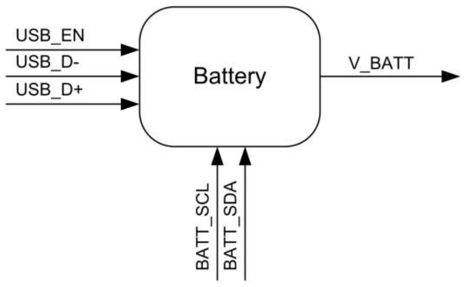

Figure 2: Battery Interface Diagram

Table 2: Battery Interface Definition

Signal Name	Properties
V_BATT	3.7 VDC, 900mA maximum, unregulated
USB_D-	3.3 VDC USB data protocol
USB_D+	3.3 VDC USB data protocol
USB_EN	3.3 VDC IO enable pin
BATT_SDA	I^2C Data
BATT_SCL	I^2C Clock

Design

The UBI-5093 was chosen as the system's battery due to its small size (36mm x 54mm x 6.2mm) and exceptional power density (900mAH capacity at a nominal voltage of 3.7V) [28]. This battery is only 6.2 mm thick which allows it to fit into the case underneath the PCB (therefore allowing a larger PCB in the same enclosure).

Since the battery is permanently soldered to the PCB, a charging circuit had to be included in the design. In order to allow the battery to be charged from a personal computer, a Mini-USB connector was used to power the charging circuit. The LP3947 IC was then chosen because it was designed to operate from a USB port and included the ability to monitor the battery's current voltage and temperature. Additionally, if the IC detects any abnormalities in temperature or charge current, the charging process will halt and the user will be notified of the error [17].

A fuel gauge circuit is included to monitor the battery's current state of charge. The fuel gauge also allows the device to predict its RTE (run-time to empty) which can then be transmitted back to the computer and displayed to the user. If the battery's voltage falls below 3.1V, the low battery indicators will be illuminated so that the user can be alerted to charge the battery. The circuit was based on the data sheet available from Texas Instruments [25].

Implementation

The design for the LP3947 charging circuit was derived from the datasheet provided by National Semiconductor [28]. The ISEL pin selects the charge current and is pulled high by an internal pull-up circuit (selecting the 100 mA charge mode). Grounding the MODE pin sets the chip to receive its charging power from a USB input. Additional pin states and modes can be found on page 2 of the data sheet [17].

The BQ27541 is a fuel gauge IC made by Texas Instruments to monitor the charge status of a Lithium-Ion battery. The reference schematic can be found on page 33 of the datasheet [25]. The IC can be programmed via I^2C to provide the state of charge, remaining capacity, RTE, and the current battery voltage. The microcontroller firmware can then use this information to implement power saving features as necessary to conserve the battery life.

Voltage Regulator Block

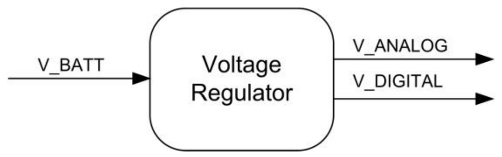

Figure 3: Voltage Regulator Interface Diagram

Table 3: Voltage Regulator Interface Definition

Signal Name	Properties
V_BATT	3.7 VDC, 900 mA maximum, unregulated
V_ANALOG	3.3 VDC, 300 mA maximum, regulated
V_DIGITAL	3.3 VDC, 300 mA maximum, regulated

Design

The voltage regulator block consists of two switching voltage regulators with one additional low dropout linear voltage regulator that is optimized for low noise performance (30 μV_{RMS} typical) [26]. The additional LDO regulator was added at the advice of an HP engineer who was present at our initial design review session. The output voltages are divided into two supplies, one intended for analog circuitry and the other for digital circuitry. This approach helps prevent high frequency noise generated by the digital circuitry from corrupting the analog circuitry power supply.

Implementation

The TPS63031 chip was selected for its buck-boost capability and its acceptable input voltage range of 1.8 V to 5.5 V [27] (page 1). When the battery's voltage is greater than 3.3 V, the chip is in buck mode, stepping the voltage down to 3.3 V and delivering up to 800 mA of current [27]. When the battery voltage falls below 3.3 V, the switching regulator will transition to boost mode, stepping the voltage up to 3.3 V and delivering up to 500 mA of current [27].

The TPS63031 datasheet provides two equations to determine a suitable inductor value for the switching portion of the circuit (see Equation B-1). The datasheet recommends inductance values between 1.5 µH and 4.7 µH [27].

Texas Instruments also provides two equations to calculate the peak inductor current (see Equation B-3). The datasheet suggests the use of the larger value between I_1 and I_2 [27]. Using a worst case current draw of 300 mA (calculated by allocating 50mA towards the microcontroller, 120 mA to the LEDS, 25 mA to the transceiver, 30 mA to the Flash, and 75 mA to the remaining circuitry) and a minimum switching frequency of 2200 KHz, the predicted peak current draw is approximately 500 mA. Based on the datasheet's typical application circuit, a 1.5 uH SMD inductor with a 10% I_{SAT} current of 1.8 A was chosen to fill this role. The current rating of this inductor is quite sufficient for this application and leaves an adequate safety margin. This current draw is also well within the battery's recommendation of 1 C (900 mA) for the maximum discharge rate [28] (page 1).

The PS/SYNC pin controls the chip's power conservation mode and will be driven to logic one on both voltage channels [27] (page 4). While in low battery mode, the microcontroller will send a low (0 V) signal to the EN pin, thereby disabling V_ANALOG [27] (page 4).

In order to provide adequate overhead voltage for the 3.3V LDO regulator, a TPS63030 is used with its output voltage set to 3.6 V. This voltage is determined by the voltage divider formed by R101 and R102. The datasheet provides a formula that can be used to determine appropriate values (see Equation B-6) [27] (page 14). Choosing R_2 as 100K ohms, R_1 is calculated as 620K ohms. The closest resistor value in the E96 (1% tolerance) series is 619k.

The output of the TPS63030 is decoupled with 0.1uF and 10uF capacitors to help prevent high frequency noise from entering the TPS73633 3.3V LDO regulator. The capacitors placed on the output and noise rejection pins (C104 and C105, respectively) are specified in the data sheet [26] (page 7).

Visual Indicator Block

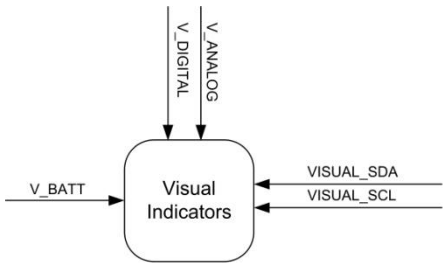

Figure 4: Visual Indicator Interface Diagram

Table 4: Visual Indicator Interface Definition

Signal Name	Properties
V_BATT	3.7 VDC, 900mA maximum, unregulated
V_ANALOG	3.3 VDC, 300 mA maximum, regulated
VISUAL_SDA	I^2C Data
VISUAL_CLK	I^2C Clock

Design

The visual indicator system is included to alert lab employees whenever a module needs human intervention. The LEDs will always be either off, on, or blinking.

Once the battery capacity drops to a software configurable threshold (i.e. 5% remaining) the LEDs will be disabled and the entire device will enter a low power mode. Lithium Ion batteries are extremely sensitive to over discharge and can be permanently damaged if the voltage drops below 2.4V [9]. Consequently, the low battery LEDs will be programmed to blink only until this critical voltage is approached.

Implementation

In order to ensure that the indicators are clearly visible, high brightness LEDs were selected for the project. The SMD LEDs have a 120 degree viewing angle and a brightness of up to 1800 millicandela [19][20].

The TLC59108 LED driver is capable of driving eight LEDs with configurable duty cycle and current. The IC interfaces with the microcontroller via I^2C. One resistor is used to select the device's I^2C address and a second resistor is chosen to create the device's reference current. The TI datasheet provides a simple formula to determine the necessary resistor value (see Equation B-6) [24]. A reference resistor value of 619 ohms was chosen to set the default LED current at 30.2 mA. The LED controller also includes the ability to vary LED currents for each channel and detect LED open circuit malfunctions [24].

Flash Memory Block

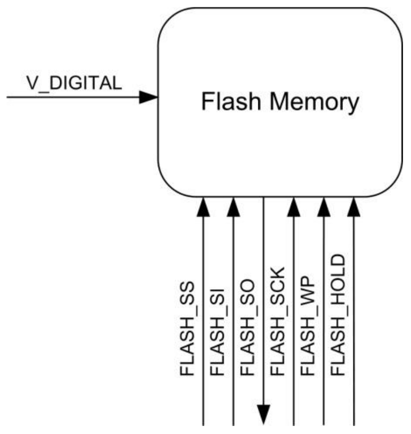

Figure 5: Flash Memory Interface Diagram

Table 5: Flash Memory Interface Definition

Signal Name	Properties
V_DIGITAL	3.3 VDC, 300 mA maximum, regulated
FLASH_SS	SPI Slave Select
FLASH _SI	SPI Slave Data Input
FLASH _SO	SPI Slave Data Output
FLASH _SCK	SPI Clock
FLASH _WP	Flash Write Protect (Active Low)
FLASH _HOLD	SPI Slave Select (Active Low)

Design

Based on the project requirements, digital acceleration values will be recorded to the flash memory at a minimum of 50 Hz. Since the module contains dual 3-axis accelerometers, six channels of 16 bit data will be recorded at every 20 ms iteration. In order to record a 30 minute test, the minimum capacity is as follows.

$$MEM_{REQ} = (6\ channels)x(16\ bits\ /\ channel)x(50\ samples\ /\ second\ )x(60\ seconds\ /\ minute)x(30\ minutes\ /\ test) = 1.08MB$$

A 16 Megabit (2MB) SPI flash IC fulfills this space requirement with plenty of room to spare.

Implementation

The flash memory IC draws its supply voltage from V_DIGITAL in order to minimize the high frequency switching noise present on V_ANALOG. Decoupling capacitors have also been added on the V_{CC} lines to help reduce transient voltages.

All I/O lines are connected to the microcontroller, consisting of four serial peripheral interface lines and two additional wires for the write protect and hold lines [23]. The flash IC is effectively self contained and does not require any additional external components.

Transceiver Block

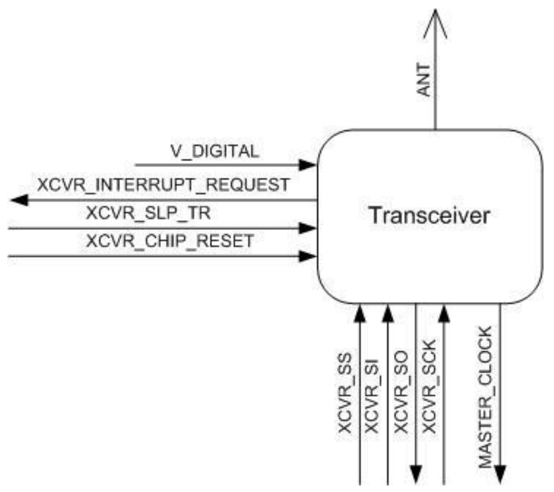

Figure 6: Transceiver Interface Diagram

Table 6: Transceiver Interface Definition

Signal Name	Properties
V_DIGITAL	3.3 VDC, 300 mA maximum, regulated
XCVR_SS	SPI Slave Select
XCVR _SI	SPI Slave Data Input
XCVR _SO	SPI Slave Data Output
XCVR _SCK	SPI Clock
MASTER_CLOCK	3.3 VDC Clock Signal 16 MHz square wave 50% duty cycle Output driver is software selectable between 2 mA, 4 mA, 6 ma and 8 mA
XCVR_INTERRUPT_REQUEST	3.3 VDC Interrupt signal Active high signal
XCVR_SLP_TR	3.3 VDC Initiates transceiver power-down mode Also triggers the start of a packet transmission Active high signal
XCVR_CHIP_RESET	3.3 VDC Active high signal resets the transceiver and restores the default startup configuration
ANT	IEEE 802.15.4 RF Signal Frequency: 2.402 GHz - 2.480 GHz Modulation: DSSS Power: -10 dBm to 3 dBm (software selectable)

Design

The transceiver block is designed around an Atmel RF230 IEEE 802.15.4 integrated circuit. The RF circuitry is designed to operate using Offset-QPSK with half-sine pulse shaping [6]. The transceiver is driven from a high precision (10 PPM frequency stability) 16.0 MHz crystal [1]. An internal clock divider in the RF230 is programmed to output an 8.0 MHz clock to the microcontroller.

Most of the necessary RF components are already included in the transceiver IC. A complete RF solution is formed with the RF230, a 16MHz crystal, a 100Ω differential to 50Ω single ended balun, and an external antenna [6].

Implementation

The design for the AT86RF230 transceiver came from an Atmel application circuit [6]. Resistor R202 and capacitor C203 form a low pass filter that is designed to reduce high frequency switching noise from the output clock. It is important that this filter be placed as close to the CLKM pin as possible to work most effectively.

The crystal specifies a load capacitance of 10 pF [1]. The optimum values of CX201 and CX202 were then found by solving Equation B-7 and assuming C_{STRAY} to be 5 pF [15].

A 1/4-wave single ended antenna is used with a voltage standing wave ratio (VSWR) of approximately 1.31 and a bandwidth of 50 MHz [4]. A VSWR of ≤ 2.0 ensures that at least 88.9% of the energy sent to the antenna will be radiated into space [29]. In order to convert the differential antenna output from the transceiver into a single ended output, a 100 ohm to 50 balun is used. The balun configuration was adapted from the example on page 8 of the datasheet [6].

Microcontroller Block

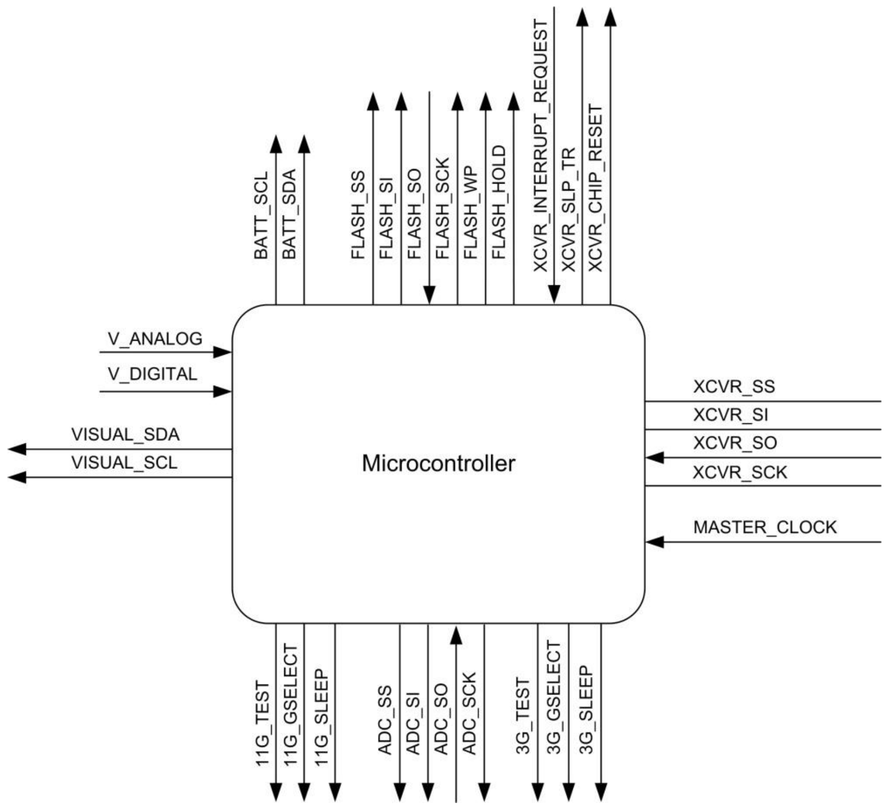

Figure 7: Microcontroller Interface Diagram

Table 7: Microcontroller Interface Definition

Signal Name	Properties
V_BATT	3.7VDC, 900 mA maximum, unregulated
V_ANALOG	3.3VDC, 300 mA maximum, regulated
V_DIGITAL	3.3VDC, 300 mA maximum, regulated
MASTER_CLOCK	3.3VDC Clock Signal 16 MHz square wave 50% duty cycle Output driver is software selectable between 2

	mA, 4 mA, 6 ma and 8 mA
BATT_SDA	I^{2}C Data
BATT_SCL	I^{2}C Clock
XCVR_SS	SPI Slave Select
XCVR _SI	SPI Slave Data Input
XCVR _SO	SPI Slave Data Output
XCVR _SCK	SPI Clock
XCVR_INTERRUPT_REQUEST	3.3 VDC Interrupt signal Active high signal
XCVR_SLP_TR	3.3 VDC Initiates transceiver power-down mode Also triggers the start of a packet transmission Active high signal
XCVR_CHIP_RESET	3.3 VDC Active high signal resets the transceiver and restores the default startup configuration
3G_TEST	Used to test accelerometer functionality
3G_GSELECT	Selects between 3G and 11G mode
3G_SLEEP	Enables low power
11G_TEST	Used to test accelerometer functionality
11G_GSELECT	Selects between 3G and 11G mode
11G_SLEEP	Enables low power
VISUAL_SDA	I^{2}C Data
VISUAL_CLK	I^{2}C Clock
FLASH_SS	SPI Slave Select
FLASH _SI	SPI Slave Data Input
FLASH _SO	SPI Slave Data Output
FLASH _SCK	SPI Clock
FLASH _WP	Flash Write Protect (Active Low)
FLASH _HOLD	SPI Slave Select (Active Low)
ADC_SS	SPI Slave Select

ADC _SI	SPI Slave Data Input
ADC _SO	SPI Slave Data Output
ADC _SCK	SPI Clock

Design

The Atmel XMEGA 64A4 was initially selected as the microcontroller of choice. The XMEGA series of microcontrollers can consume as little as 1.7 uA in power-down mode while running at 3.3 V [7]. The chip consumes only 7.5 mA while clocked at 16 MHz. The 64A4 also offers an attractive array of features including dual SPI ports, multiple 16 bit timers, 8 event channels, 34 I/O lines, and a maximum operating frequency of 32 MHz [7].

Unfortunately, the 64A4 was very difficult to obtain at design time. After contacting multiple distributors it was discovered that Atmel's production of the 64A4 was several months behind schedule and no one seemed to know when the IC would become available. Consequently, another microcontroller was chosen instead.

The XMEGA 128A3 is a similar model that is currently available for purchasing. The 128A3 has a slightly higher power consumption (9.5 mA in active mode while clocked at 16 MHz) [8] but it is still suitable for this application.

Implementation

The microcontroller external clock is provided by the transceiver block as described on page 70 of the RF230 datasheet [6]. The system clock section (pg 18) of the microcontroller datasheet confirms that an external clock signal can be used with the XMEGA A3 [8]. A brief description of the various clock configurations is provided on page 20 of the datasheet.

16-bit ADC Block

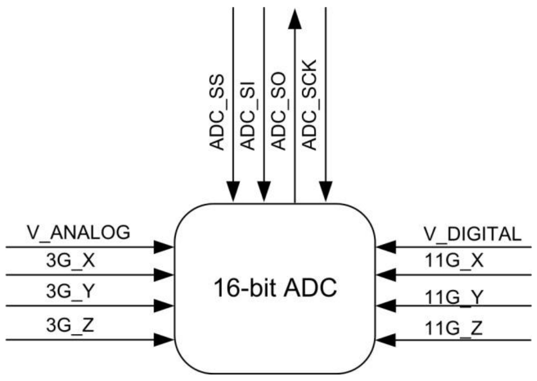

Figure 8: 16-bit ADC Interface Diagram

Table 8: 16-bit ADC Interface Definition

Signal Name	Properties
V_ANALOG	3.3 VDC, 300 mA maximum, regulated
V_DIGITAL	3.3 VDC, 300 mA maximum, regulated
ADC_SS	SPI Slave Select
ADC _SI	SPI Slave Data Input
ADC _SO	SPI Slave Data Output
ADC _SCK	SPI Clock
3G_X	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
3G_Y	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
3G_Z	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_X	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_Y	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_Z	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC

Design

The project utilizes a self-contained 16-bit, 8 channel ADC (separate from the microcontroller). The only additional components needed are the decoupling capacitors placed on each voltage reference pin. Since the module uses dual 3-axis accelerometers, only six channels are necessary. In order to avoid any floating I/O lines, the remaining two channels are connected to V_ANALOG and GND, respectively.

Implementation

In order to minimize the ADC's exposure to high frequency switching noise, the IC is powered by V_ANALOG. Table 7 of the AD7689 datasheet explains that the REF pin must have a reference voltage applied where 0.5 V <= Vref <= V_ANALOG [3] (pg 9). Vref was chosen as V_ANALOG because the two accelerometers can potentially output voltages up to 3.2VDC (see page 3 of the MMA7340L datasheet) [12]. C502 exists to decouple the REF pin as specified on page 9 of the datasheet. C500 and C501 are included to minimize noise from the voltage regulators (see page 6 of Intersil Application Note 1325, as well as page 30 of the AD7689 datasheet) [14][3].

The SPI bus connections are taken from the sample schematic on page 29 of the datasheet [3]. R500 is a pull-up resistor that exists to maintain the Slave Output line at 3.3 VDC until the ADC chip pulls the line low. R500 is also shown on page 229 of the datasheet [3].

Accelerometer Block

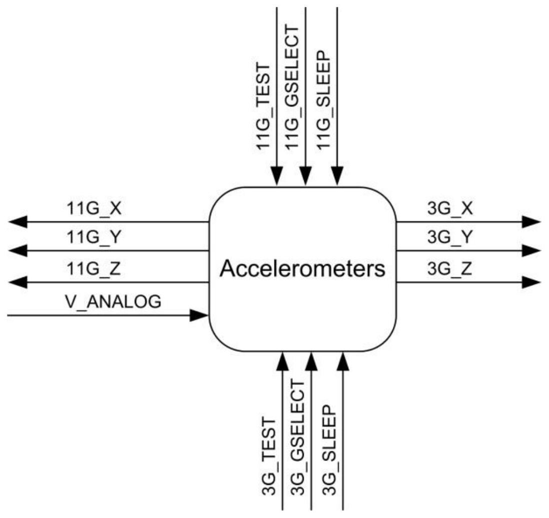

Figure 9: Accelerometer Interface Diagram

Table 9: Accelerometer Interface Definition

Signal Name	Properties
V_ANALOG	3.3 VDC, 300 mA maximum, regulated
3G_X	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
3G_Y	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
3G_Z	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_X	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_Y	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
11G_Z	Analog voltage (0 V - 3.2 V) 0g -> 1.65 VDC @ 25ºC
3G_TEST	Used to test accelerometer functionality
3G_GSELECT	Selects between 3G and 11G mode

3G_SLEEP	Enables low power
11G_TEST	Used to test accelerometer functionality
11G_GSELECT	Selects between 3G and 11G mode
11G_SLEEP	Enables low power

Design

The project utilizes dual accelerometers in order to provide greater sensitivity at low accelerations while still maintaining the ability to record large acceleration values. One accelerometer will be set to a sensitivity level of ±3g while the other will be set at ±11g. In the ±3g setting the accelerometer is able to represent acceleration more precisely (440mV per 9.8 m/s^2) at the cost of decreased range [12]. Unfortunately, it is possible that the module may experience acceleration magnitudes in excess of 3g's. In this situation, the ±11g accelerometer plays an important role by providing data when the high sensitivity accelerometer has reached the edge of its dynamic range. It spreads each g of acceleration over 117 mV [12] which gives it more range but less accuracy. Each chip is capable of collecting at both sensitivity settings (but not simultaneously), hence the reason that both accelerometers are utilized simultaneously. Both sets of data are saved into the external flash as well.

Implementation

The accelerometer's nominal voltage reading is 1.65 volts at 0 g’s [12]. When the accelerometer is orientated positively in earth's gravitational field the voltage reading will increase to represent 9.8 m/s^2. If the accelerometer is rotated upside down, the output voltage will decrease from the effects of gravity.

The 11g accelerometer's G-SELECT pin will need to be pulled high by the microcontroller in order to select the proper sensitivity range [12] (Page 4). Capacitors C601, C602, C603, C701, C702, and C703 are included to filter high frequency noise from the accelerometer output lines. The value of 3.3 nF was specified on page 5 of the datasheet [12].

MICROCONTROLLER FIRMWARE

Design

The following flow chart illustrates the firmware's operation.

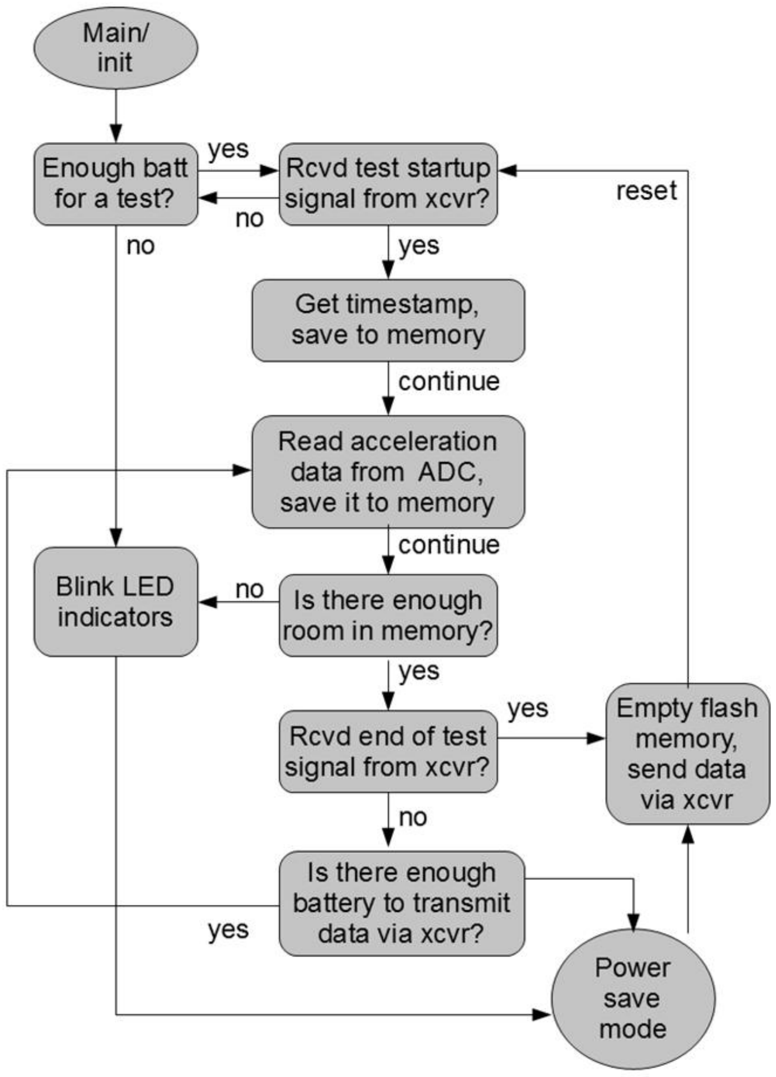
Main/ init
Enough batt for a test?
yes
no
Rcvd test startup signal from xcvr?
no
reset
yes
Get timestamp, save to memory
continue
Read acceleration data from ADC, save it to memory
continue
Blink LED indicators
no
Is there enough room in memory?
yes
yes
Rcvd end of test signal from xcvr?
Empty flash memory, send data via xcvr
no
Is there enough battery to transmit data via xcvr?
yes
Power save mode

Figure 10: Microcontroller Firmware Flowchart

Testing

The firmware will begin by initializing the visual indicators, the transceiver, the external flash, the ADC, the V_ANALOG voltage regulator, and the accelerometers. After initialization, the software will then proceed to follow the flowchart shown in Figure 10. The bullet points listed below explain each test in the firmware flowchart.

- "Enough batt for a test?":

  If there is insufficient battery for a test, the visual indicators will display the low battery warning and the module will be disabled until the battery is charged.
- "Rcvd test startup signal from xcvr?":

  The microcontroller will remain in an idle state in order to ensure minimal power consumption. An interrupt routine will monitor the XCVR_INTERRUPT_REQUEST line for the start of test signal. Additionally, the battery voltage will be periodically checked to ensure that the battery's charge is sufficient.
- "Get timestamp, save to memory":

  When the transceiver receives a start test signal, a time stamp is immediately saved to flash.
- Data collection loop
  - "Read acceleration data from ADC, save it to memory":

    An interrupt service routine (triggered by a timer) retrieves a data sample from the ADC every 20ms and saves it to the external flash.
  - "Is there enough room in memory?":

Before the sample can be saved, the remaining flash capacity is checked to ensure that there is sufficient memory. If there is no space for the sample, the external memory is considered to be full and a signal goes to the LED indicators to signify the error condition.

- "Rcvd end of test signal from xcvr?":

  An interrupt service routine monitor's the transceiver to see if it has received an end of test signal. When the signal is received, any partially filled buffers are flushed to the external flash and the internal EEPROM records are updated with the completed test information.

- "Is there enough battery to transmit data via xcvr":

  The Fuel Gauge IC is consulted via I^2C to determine the remaining battery capacity. If the charge level is becoming critical, the module will shutdown to conserve power and extend the battery life.

PCB DESIGN

The PCB (printed circuit board) design was implemented in a four layer arrangement. The stack-up consists of a top copper layer containing signal lines and controlled impedance traces. The second layer consists of a ground plane. The third layer is primarily for routing V_DIGITAL and V_ANALOG, in addition to a limited number of signal traces. The bottom (fourth) layer contains additional signal traces as shown in Figure 11 [2].

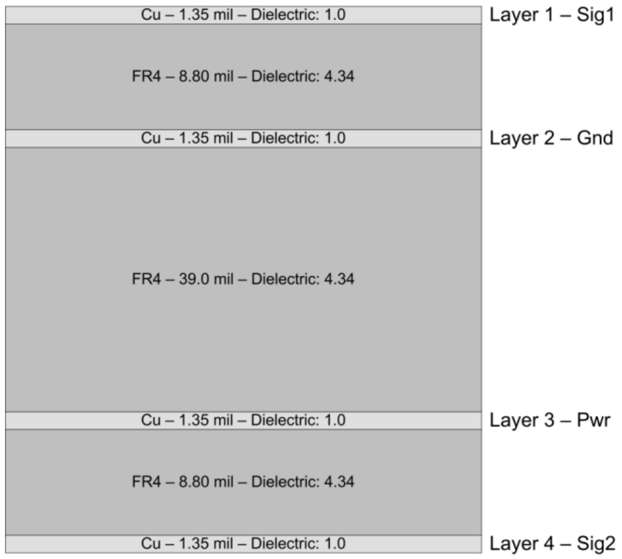
Cu – 1.35 mil – Dielectric: 1.0
Layer 1 – Sig1
FR4 – 8.80 mil – Dielectric: 4.34
Cu – 1.35 mil – Dielectric: 1.0
Layer 2 – Gnd
FR4 – 39.0 mil – Dielectric: 4.34
Cu – 1.35 mil – Dielectric: 1.0
Layer 3 – Pwr
FR4 – 8.80 mil – Dielectric: 4.34
Cu – 1.35 mil – Dielectric: 1.0
Layer 4 – Sig2

Figure 11: Four Layer PCB Stackup - Cross Sectional View

In general, the layout of the PCB can be divided into physical regions, each corresponding to a particular system block. The voltage regulators and battery monitoring circuit are grouped together in one corner on the top of the PCB. The ADC and accelerometers are located on the bottom of PCB, along with the microcontroller circuitry.

The most sensitive portion of the PCB's design is the RF transceiver. The module's RF circuitry operates in the 2.4 GHz band and controlled impedance lines are necessary for proper operation. The previous data logger design used faulty data while computing the necessary trace widths (incorrectly assuming that the substrate between PCB layers one and two was 18.8 mils in thickness). The manufacturer's website indicates that the substrate thickness is actually 8.8 mils [2]. Consequently, the previous RF design suffered from mismatched impedances which significantly reduced the system's maximum wireless range.

The RF230 datasheet explains that pins RFP and RFN form a 100Ω differential port. In order to properly match the impedances, special care was taken while selecting the trace widths.

$$Z_{DIFF} = 2 * Z_0[1 - 0.48 * e^{\left(-0.96 * \frac{S}{H}\right)}]$$

$$Z_0 = \frac{87}{\sqrt{\varepsilon_r * 1.41}} * LN\left(\frac{5.98 * H}{0.8 * W + T}\right)$$

The equation above represents the differential impedance between two PCB traces on layers one and two (where S and H are given in inches and ε_r represents the dielectric constant) [10][11][16]. The following diagram shows the physical layout from a side perspective [10].

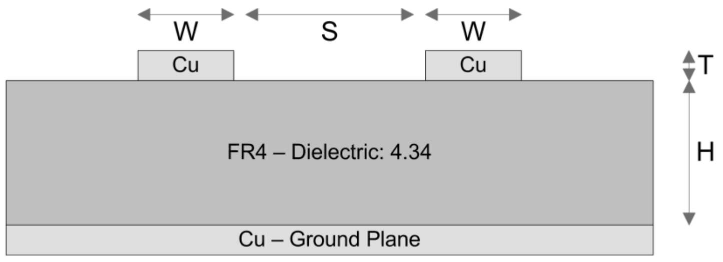

Figure 12: Microstrip Side Portrait

In practice, the differential impedance was calculated with a more direct approach. Mentor Graphics' PADS program (the PCB layout software used for this design) includes the capability to estimate the impedance of a PCB trace. Consequently, Z_{DIFF} was calculated by using the Z_0 value provided by PADS and simply plugging in the number to determine the differential impedance value.

In order to help prevent noise, the second PCB layer contains no signal traces and is reserved entirely for use as a ground plane. There is purposely no ground plane beneath the antenna due to the manufacturer's recommendation [4].

DATA STRUCTURE IMPLEMENTATION

In order to extend the microcontroller's internal EEPROM life, a form of wear leveling was implemented. In most cases the EEPROM is only guaranteed to operate properly for 100,000 write cycles [8]. However, if the user does not need to store data in 100% of the available memory, the spare memory can be utilized to extend the expected EEPROM life.

Atmel Application Note AVR101 explains one method of implementing "High Endurance EEPROM Storage" [5]. The scheme uses multiple circular buffers in EEPROM as shown in the figure below.

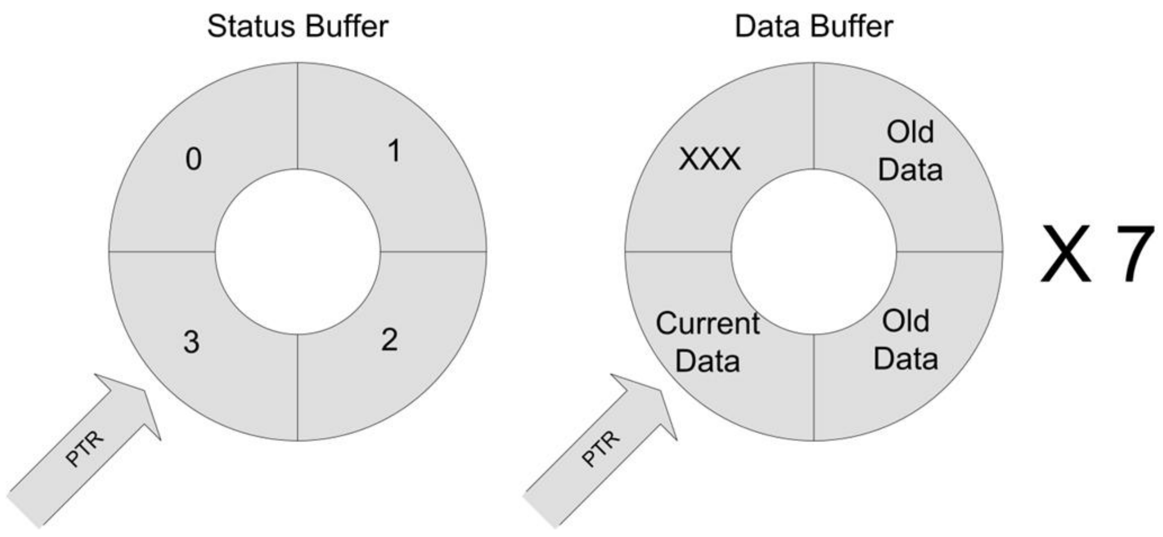

Figure 13: EEPROM Wear Leveling Implementation

One buffer (the data buffer) contains the actual user data while the other buffer (the status buffer) holds only integers. When data is written to EEPROM, a pointer in RAM is incremented and the user data is written to the next slot in the data buffer. At the same time, the status buffer's next available location is written with an incremented number.

When the device is powered off, the volatile pointer is lost. When the power returns, the pointer's previous location can be determined by cycling through the status buffer's contents and detecting the first slot that meets the condition $Contents[n+1] \neq Contents[n] + 1$ [5]. This implementation works even if the integer value overflows to zero.

CONCLUSION

The team successfully designed a module that meets all but one of the original project requirements. The customer requirements originally specified that the device must begin saving acceleration measurements within 2 ms of a 5 volt pulse issued by the Wave Lab's equipment. Unfortunately, it is impossible to guarantee that this requirement is met while using the current equipment.

In order to detect the 5 V pulse, the Wave Lab provided a USB compatible data acquisition unit that could interface with a personal computer and trigger the acceleration recording process. Unfortunately, the acquisition unit does not sample faster than 200 Hz meaning that in a worst case scenario, 5 ms will elapse before the unit detects the 5 V signal's presence. Consequently, the trigger time is unpredictable and cannot be guaranteed to activate the module in less than 2 ms. The project mentor later explained that the automated trigger had not worked on the previous model and it was not an issue if the new design did not automatically trigger.

However, all remaining requirements were satisfied. The hardware has worked flawlessly in all of the tests to date and the O. H. Hinsdale Wave Research Laboratory is intending to use the device beginning in June, 2010.

The senior design team was able to improve on several features of the previous model. The wireless range (≥ 80 feet in testing) is far superior and the visual indication system is easily visible, even while submerged 10 feet below the water's surface. The inclusion of a 16 bit ADC and ±3g accelerometers allows the module to record acceleration values with higher precision.

Also, the battery charger circuit is now built into the device so there is never a need to remove the battery, barring any physical damage.

The module was tested in the Wave Lab and it performed very well. The final product interacts with the existing graphical user interface and the battery life is more than sufficient to operate for an hour. The case is small, lightweight and has been proven waterproof to 10 feet. A short demonstration was given to the project's mentors and they were very happy to see the results. They are enthusiastic about using the module for research purposes within the coming months.

This project has provided an excellent learning experience and the real world practice will inevitably prove useful in the years to come. The assignment has provided the author with an extensive review on embedded development including everything from protocols such as SPI and I^2C to four layer PCB design and controlled impedance traces. The nine month project required nearly 1000 hours of student labor but the experience has been priceless and the long term benefits will inevitably payoff.

BIBLIOGRAPHY

[1] Abracon Corporation, “ABM 9 Technical Datasheet,” Abracon Corporation. [Online] Available: http://www.abracon.com/Resonators/abm9.pdf. [Accessed 15 November 2009].

[2] Advanced Circuits, ".062 Finished Thickness on PCB's," Advanced Circuits. [Online] Available: http://www.4pcb.com/stackups-controlled-dielectric/. [Accessed 22 May 2010]

[3] Analog Devices, "AD7682/AD7689," Analog Devices, Inc. [Online]. Available: http://www.analog.com/static/imported-files/data_sheets/AD7682_7689.pdf. [Accessed 30 November 2009].

[4] Antenna Factor, “ANT-2.4-JJB-RA,”Antenna Factor. [Online] Available: http://www.antennafactor.com/documents/ANT-2.4-JJB-xx_Data_Sheet.pdf. [Accessed 2 February 2009].

[5] Atmel Corporation, Appl. Note AVR101.

[6] Atmel, “AT86RF230 Technical Datasheet,” Atmel Corporation. [Online] Available: http://www.atmel.com/dyn/resources/prod_documents/doc5131.pdf. [Accessed 15 November 2009].

[7] Atmel, "ATxmega16A4/32A4/64A4/128A4 Preliminary," Atmel Corporation. [Online]. Available: http://atmel.com/dyn/resources/prod_documents/doc8069.pdf. [Accessed 14 November 2009].

[8] Atmel, "ATxmega64A3/128A3/192A3/256A3 Preliminary," Atmel Corporation. [Online]. Available: http://atmel.com/dyn/resources/prod_documents/doc8068.pdf. [Accessed 14 November 2009].

[9] Chester Simpson, "Characteristics Of Rechargeable Batteries," National Semiconductor Corporation. [Online] Available: http://www.national.com/appinfo/power/files/f19.pdf [Accessed 21 May 2010].

[10] Douglas Brooks, "Differential Impedance What's the Difference?," UltraCAD Design, Inc. [Online] Available: http://www.ultracad.com/articles/diff_z.pdf. [Accessed 22 May 2010].

[11] Douglas Brooks, "PCB Impedance Control: Formulas and Resources," UltraCAD Design, Inc. [Online] Available: http://www.ultracad.com/articles/formula.pdf. [Accessed 22 May 2010].

[12] Freescale Semiconductor, “MMA7340L Technical Datasheet,” Freescale Semiconductor. [Online] Available: http://www.freescale.com/files/sensors/doc/data_sheet/MMA7340L.pdf. [Accessed 5 December 2009].

[13] Henry W. Ott, "PCB Stack-Up," Henry Ott Consultants. [Online] Available: http://www.hottconsultants.com/techtips/pcb-stack-up-2.html. [Accessed 22 May 2010].

[14] Intersil, "Choosing and Using Bypass Capacitors," Intersil Americas Inc. [Online]. Available: http://www.intersil.com/data/an/an1325.pdf. [Accessed 5 December 2009].

[15] James B. Northcutt. (1998, February). "Specifying Crystals for Use in VCXOs and TCXOs for Wireless Design." *Wireless Design & Development*. [Online]. Available: http://www.foxonline.com/tech3031.htm [Accessed 22 May, 2010].

[16] Logic Systems, "Typical FR4 Laminate Properties, 0.059" [1.5mm]," Logic Systems Corporation. [Online] Available: http://www.lscpcb.com/fr4specs.htm. [Accessed 22 May 2010].

[17] National Semiconductor, "LP3947," National Semiconductor. [Online] Available: http://www.national.com/mpf/LP/LP3947.html#Overview. [Accessed 15 November 2009].

[18] NXP, "The I2C-Bus Specification," NXP Semiconductors. [Online]. Available: http://www.nxp.com/acrobat_download2/literature/9398/39340011.pdf. [Accessed 5 December 2009].

[19] OSRAM, "PointLED Enhanced Thinfilm LED -- LR P47F-U2AB-1-1-Z," OSRAM Opto Semiconductors. [Online] Available: http://catalog.osram-os.com/catalogue/catalogue.do;jsessionid=74B0AF55271023A752A5A2E1F061E872?act=downloadFile&favOid=020000000000bc28000200b6 [Accessed 14 January 2010].

[20] OSRAM, "TOPLED Enhanced Thinfilm LED -- LA T67F-U2AB-24-Z," OSRAM Opto Semiconductors. [Online] Available: http://catalog.osram-os.com/catalogue/catalogue.do;jsessionid=2E00060302E798ACA185A88D03AF7201?act=downloadFile&favOid=020000030001d516000100b6 [Accessed 14 January 2010].

[21] Otter, "Otter Box" [Online]. Available: http://www.otterbox.com/waterproof-cases/otterbox-1000/. [Accessed 15 November 2009].

[22] Paul Horowitz and Winfield Hill, *The Art of Electronics*, 2nd ed. New York: Cambridge University Press, 2008, pp. 490.

[23] SST, "16 Mbit, serial Flash memory, 75 MHz SPI bus interface" Numonyx, B. V. [Online]. Available: http://www.numonyx.com/Documents/Datasheets/M25P16.pdf [Accessed 9 March, 2010].

[24] TI, "8-Bit Fast Mode Plus (FM+) I2C Bus Constant-Current LED Sink Driver," Texas Instruments. [Online] Available: http://focus.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=tlc59108&fileType=pdf [Accessed 4 December 2009].

[25] TI, "BQ27541 Datasheet," Texas Instruments. [Online] Available: http://focus.ti.com/lit/ds/symlink/bq27541.pdf. [Accessed 15 November 2009].

[26] TI, "Cap-Free, NMOS, 400mA Low-Dropout Regulator with Reverse Current Protection," Texas Instruments. [Online] Available: http://focus.ti.com/lit/ds/symlink/tps73633.pdf [Accessed 19 January 2010].

[27] TI, "High Efficient Single Inductor Buck-Boost Converter w/1-A Switches (Rev. A)," Texas Instruments. [Online] Available: http://focus.ti.com/lit/ds/symlink/tps63031.pdf [Accessed 27 November 2009].

[28] ULTRALIFE Batteries, “UBP563450/PCM Technical Datasheet,” ULTRALIFE Batteries. [Online]. Available: http://www.ultralifebatteries.com/documents/techsheets/UBI-5093_UBP563450.pdf. [Accessed 15 November 2009

[29] Linx Technologies, Appl. Note AN-00501.

APPENDICES

APPENDIX A: SCHEMATICS

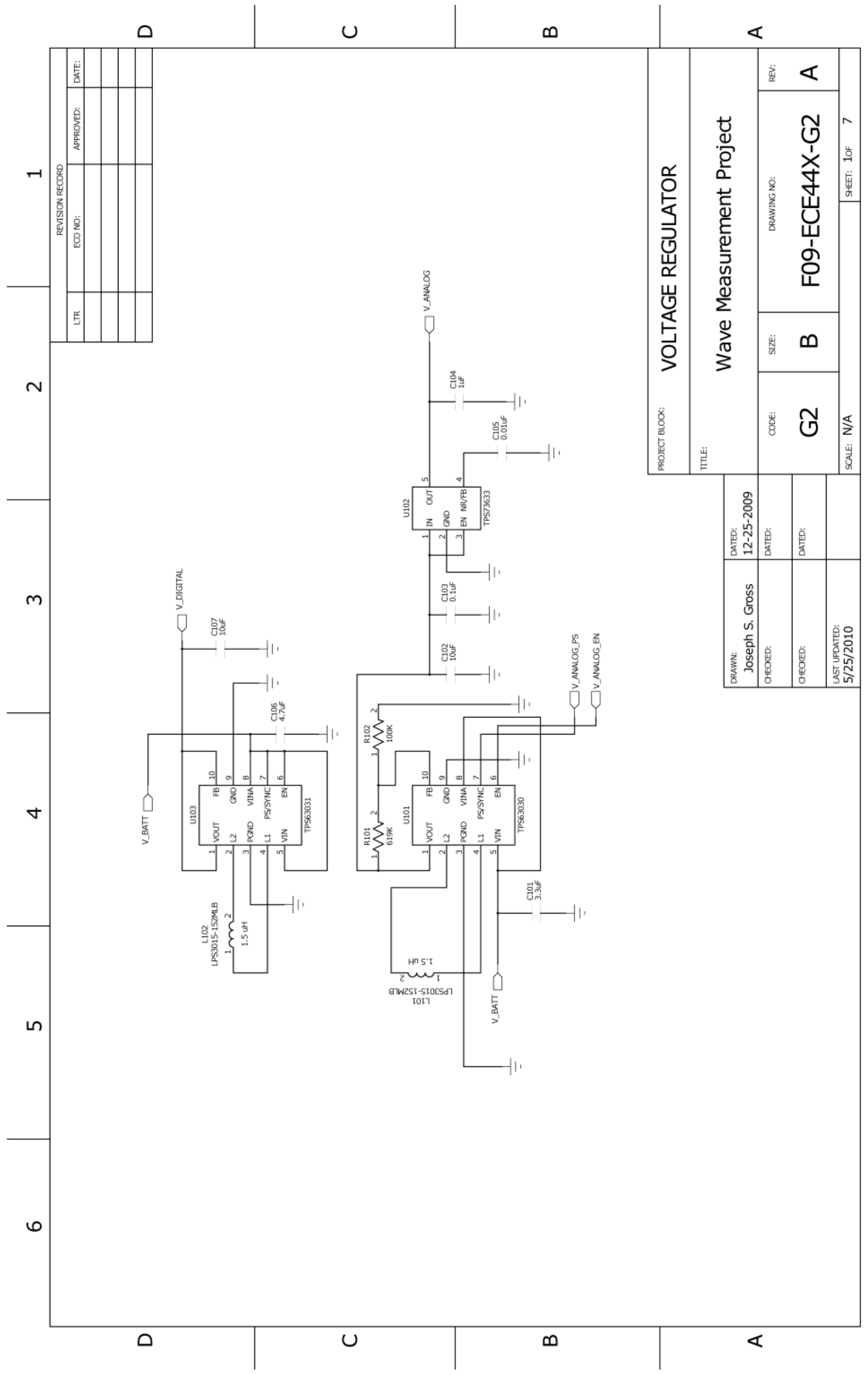

Figure A-1: Voltage Regulator Schematic

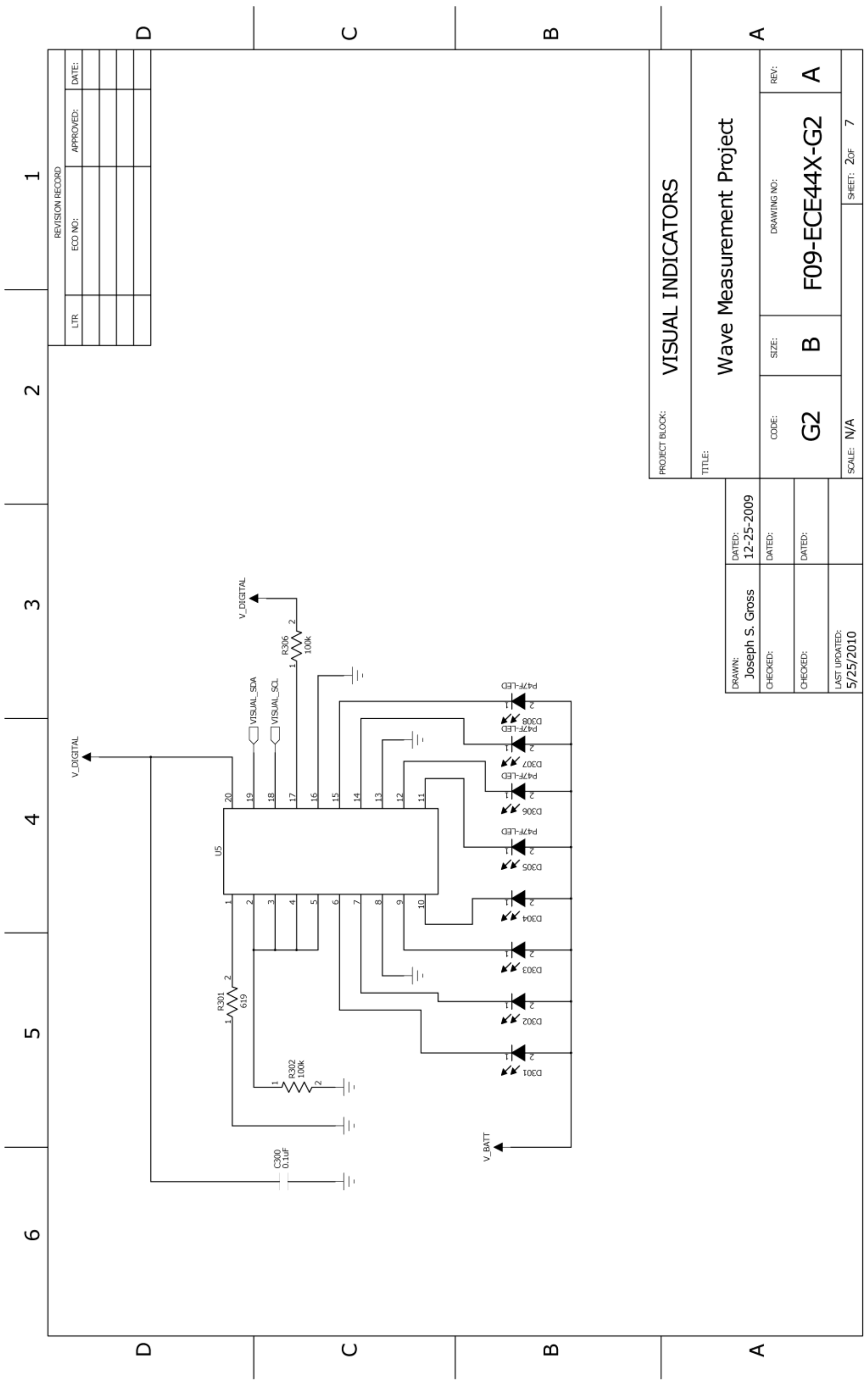

Figure A-2: Visual Indicator Schematic

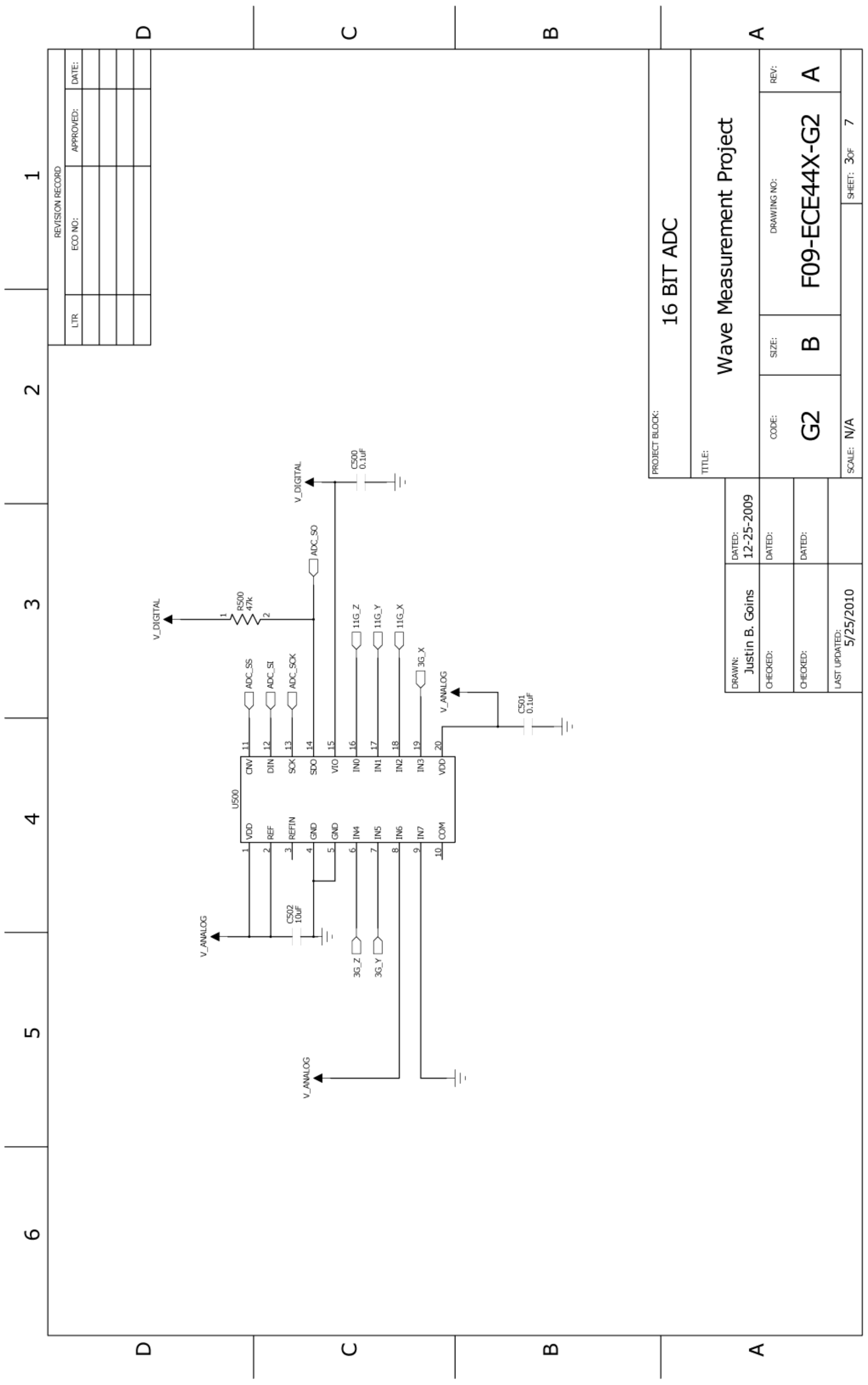

Figure A-3: 16-bit ADC Schematic

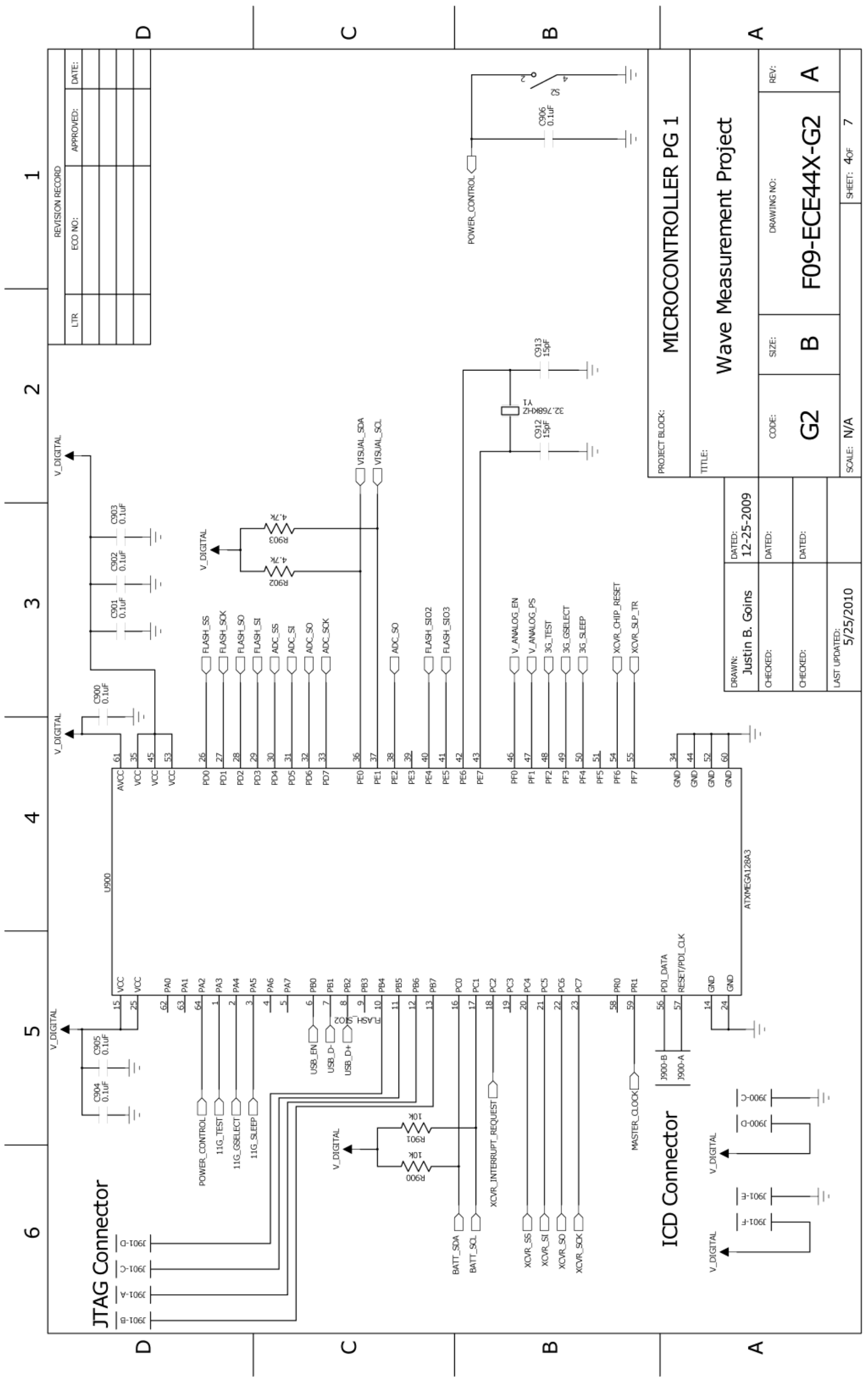

Figure A-4: Microcontroller Schematic

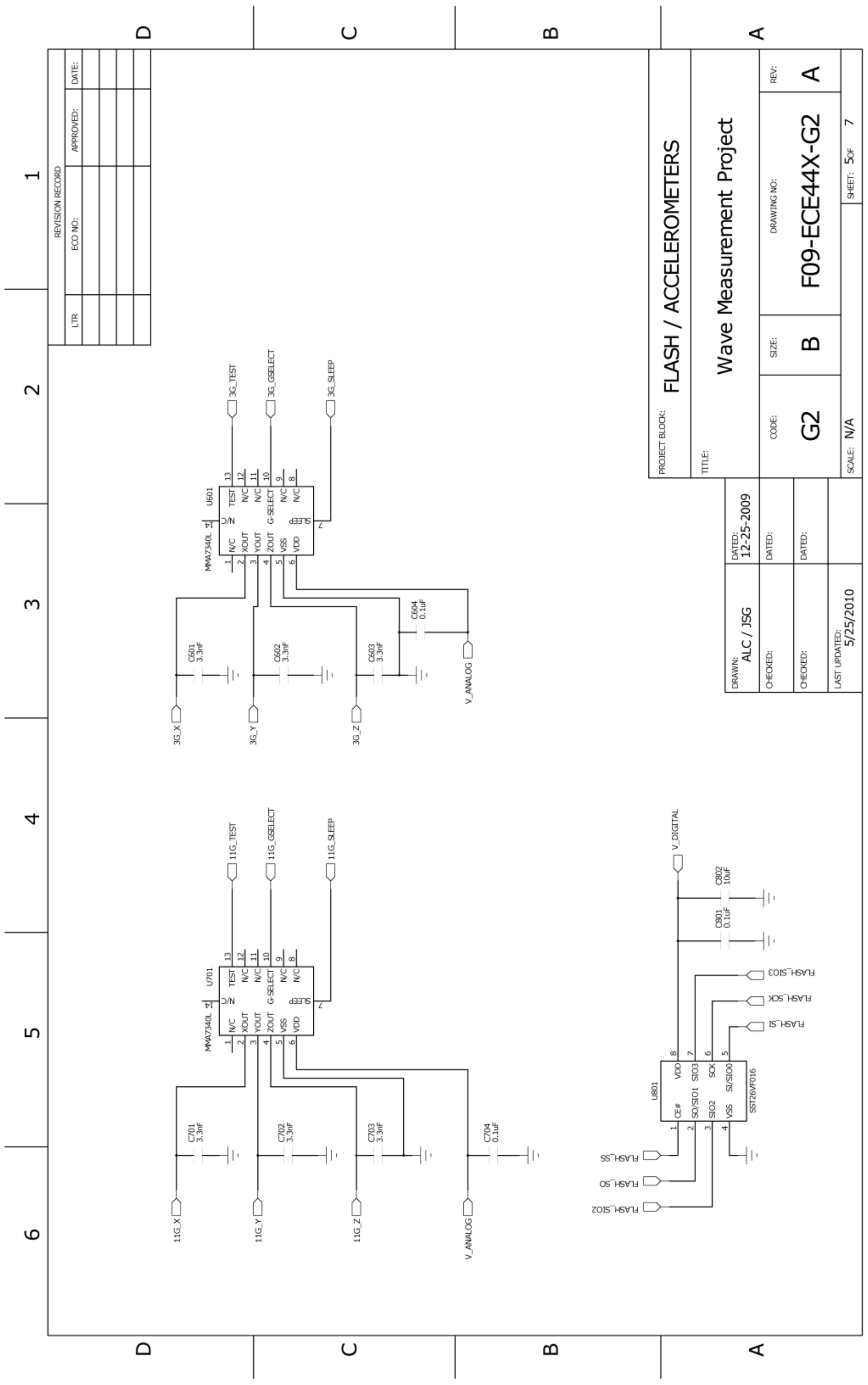

Figure A-5: Flash & Accelerometer Schematics

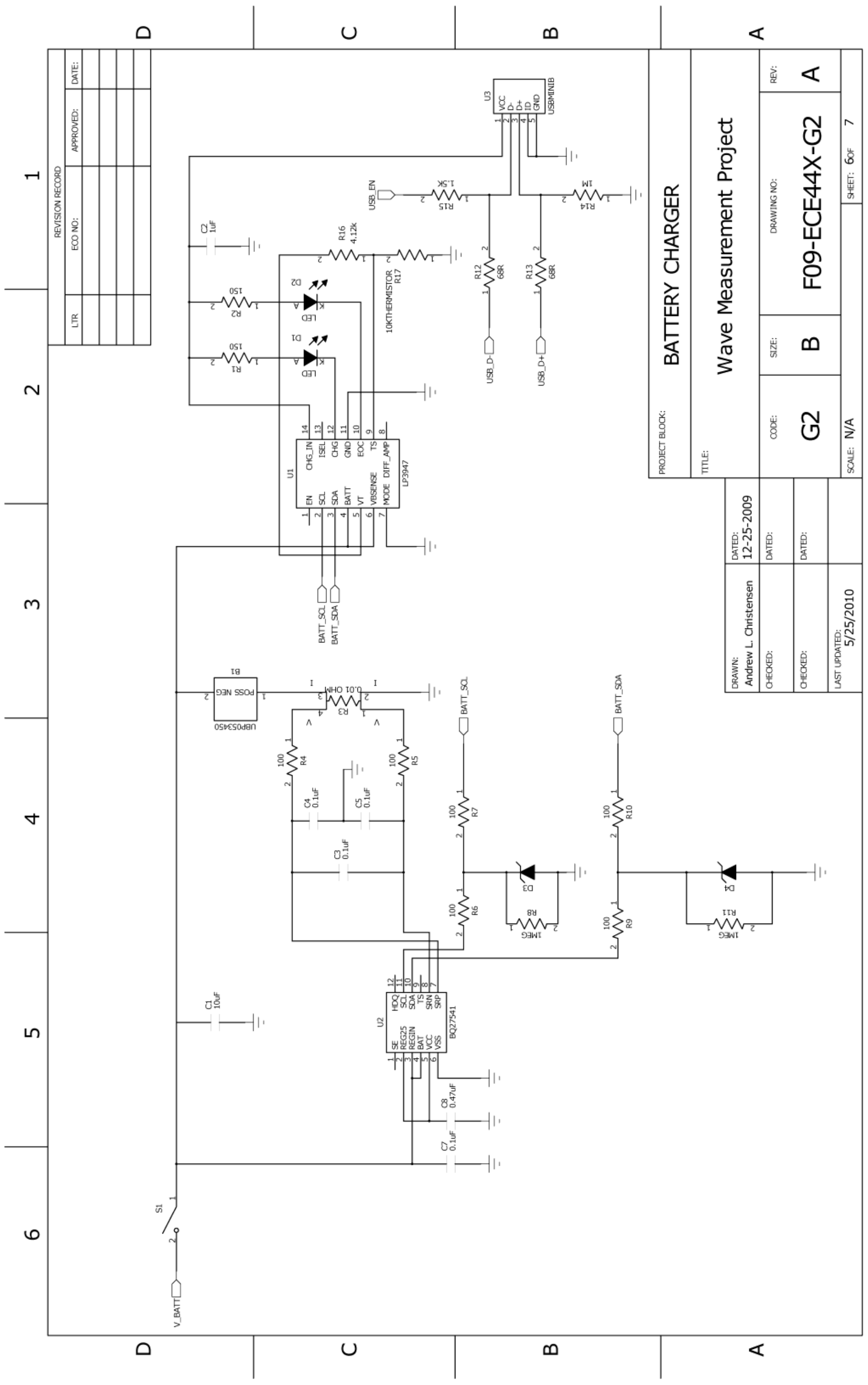

Figure A-6: Battery Charging & Monitoring Schematic

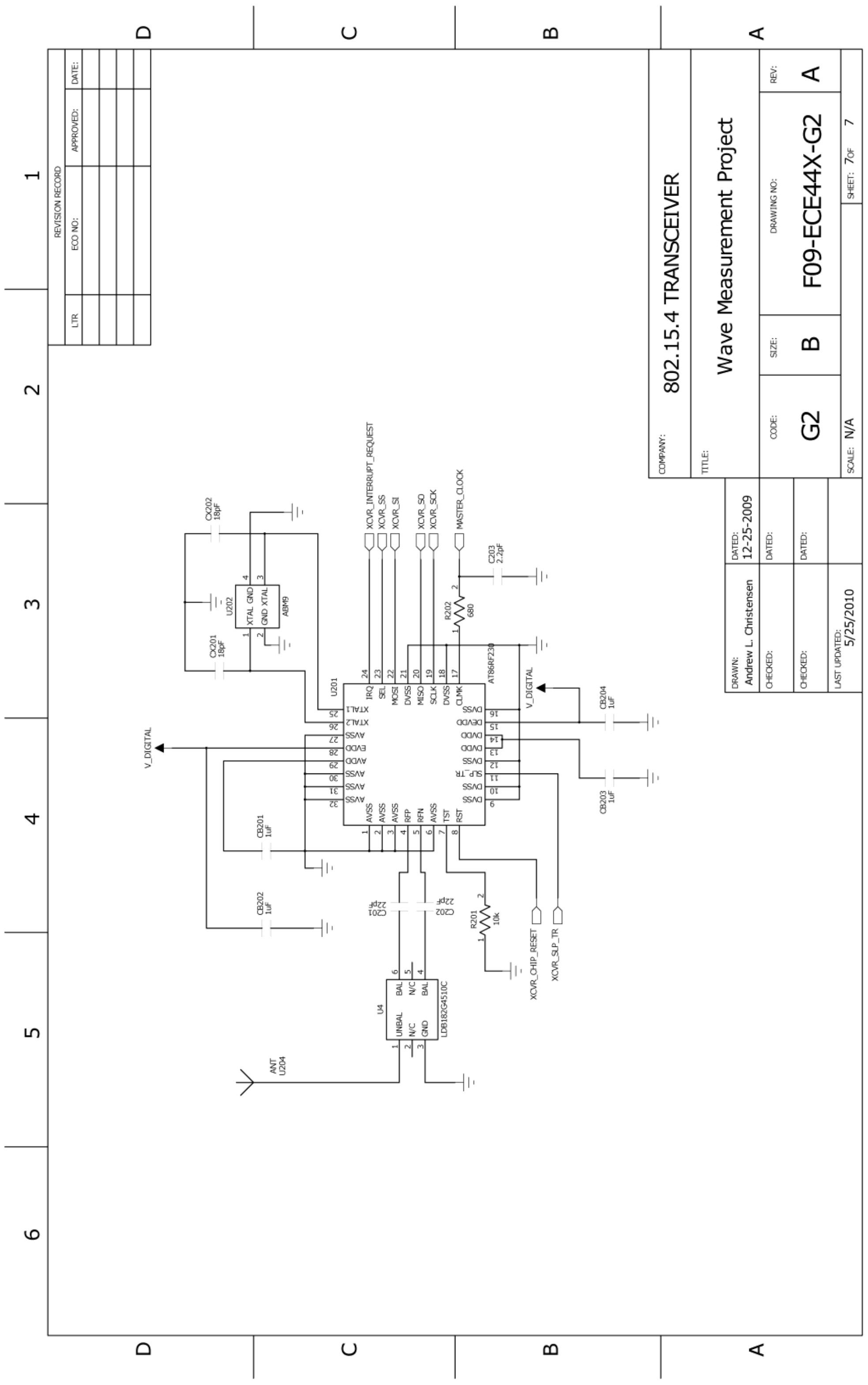

Figure A-7: Transceiver Schematic

APPENDIX B: EQUATIONS

Power Supply Equations

L_1 represents the minimum inductance to operate in step down mode and L_2 represents the minimum inductance needed to operate in boost mode [27].

$$L_1 = (V_{IN1} - V_{OUT}) * 0.5 * \frac{\square s}{A}$$

Equation B-1: Minimum VREG Inductance During Buck Mode

$$L_2 = V_{OUT} * 0.5 * \frac{us}{A}$$

Equation B-2: Minimum VREG Inductance During Boost Mode

$$I_1 = \frac{I_{OUT}}{0.8} + \frac{V_{OUT}\,(V_{IN1} - V_{out\,)}}{2 * V_{IN1} * f * L}$$

Equation B-3: Peak Inductor Current During Buck Mode

$$I_2 = \frac{V_{OUT} * I_{OUT}}{0.8 * V_{IN2}} + \frac{V_{IN2} * (V_{OUT} - V_{IN2})}{2 * V_{OUT} * f * L}$$

Equation B-4: Peak Inductor Current During Boost Mode

$$R_1 = R_2 * (\frac{V_{OUT}}{V_{FB}} - 1)$$

Equation B-5: Resistor Values Needed To Program Output Voltage

Visual Indicator Equations

$$I_{OUT\ (target\ )} = \frac{1.25}{R_{EXT}} * 15 * A$$

Equation B-6: Estimated LED Current (Per Channel)

Transceiver Equations

$$C_{LOAD} = C_{STRAY} + \frac{CX201 * CX202}{CX201 + CX202}$$

Equation B-7: Calculation Of Crystal Load Capacitance

APPENDIX C: PCB LAYOUT

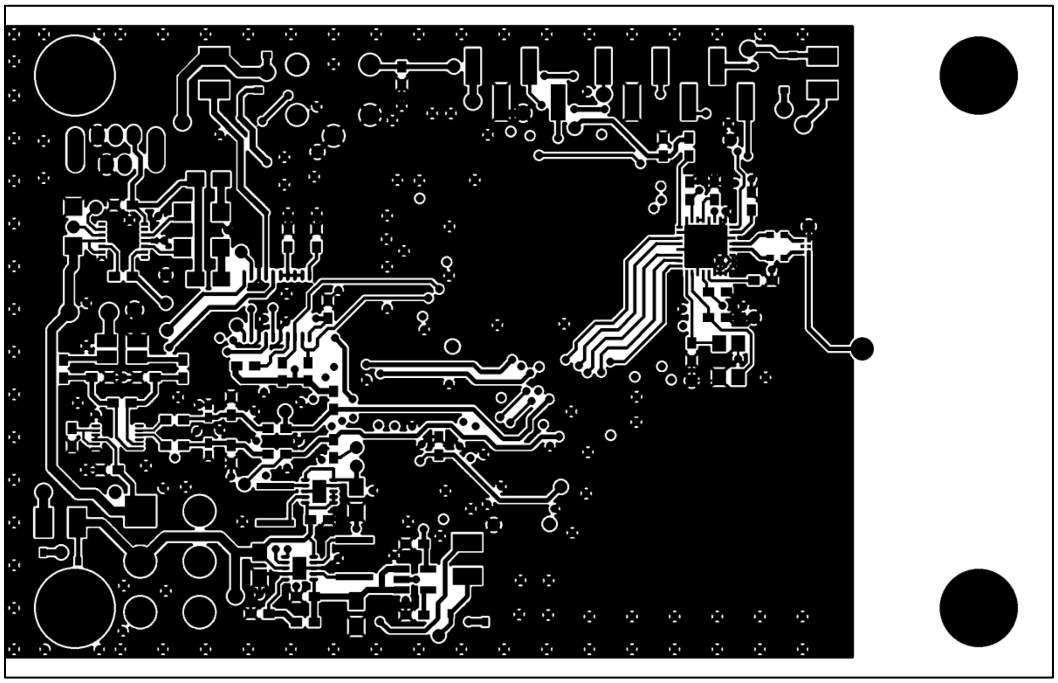

Figure C-1: Layer 1 Copper

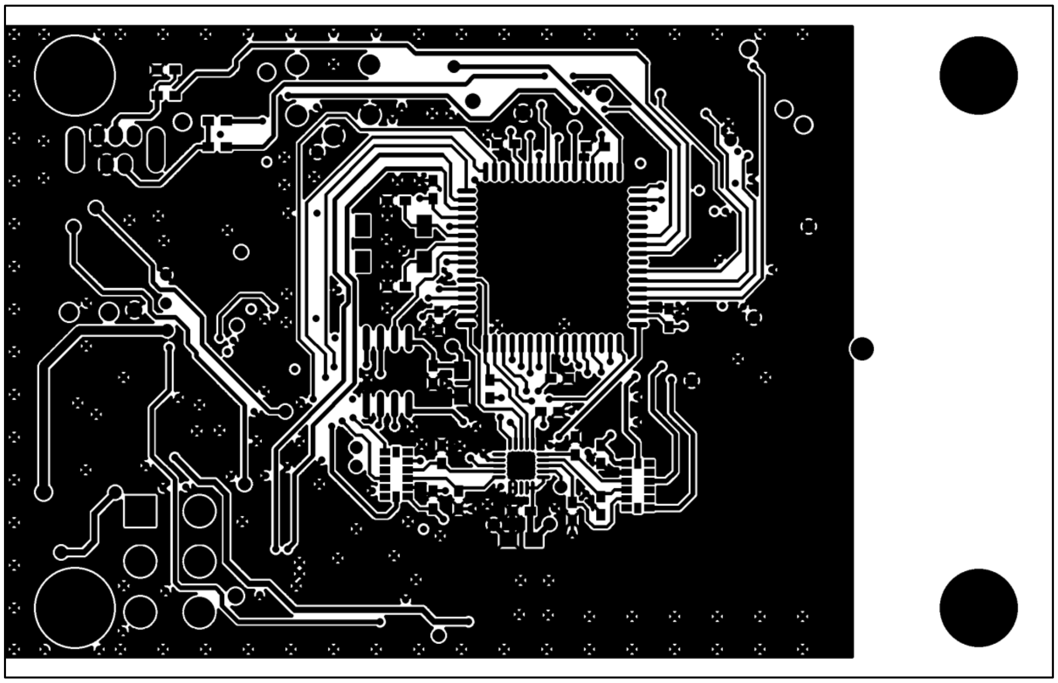

Figure C-2: Layer 4 Copper

APPENDIX D: G2W10.C SOURCE FILE

```
/*
  The following code was written by Justin Goins.
  Additional contributions are included from Joseph Gross.

  Senior Design Group 2 Firmware
  Last Updated May 22, 2010
*/

/* Default CPU frequency */
#define F_CPU 2000000UL

#include <util/delay.h>
#include <avr/io.h>
#include <stdlib.h>
#include "flash.h"
#include "adc.h"

#include "visual.h"

#include "xcvr.h"
#include "at86rf230_registermap.h"

#include <avr/interrupt.h>

#include "extra.h"
#include "eeprom_driver.h"
#include "eeprom_leveling.h"


// Globals
volatile uint8_t flash_buffer[2][256], flash_buffer_index, flash_current_buffer, flash_buffer_is_full;
volatile uint8_t extFlashWriteInProgress = 0x00;
volatile uint8_t opcode = 0x00; // Intialize to zero
hal_rx_frame_t myframe; // This frame will hold data from the TX


// External EEPROM TX Complete Flag
ISR(USARTD0_TXC_vect)
{
  extFlashWriteInProgress = ZERO;
  return;
}

// SPM wakeup interrupt
ISR(NVM_SPM_vect)
{
  // Do nothing
  return;
}

// EEPROM wakeup interrupt
ISR(NVM_EE_vect)
{
  // We need to disable the IRQ
  // It will get re-enabled the next time it's needed
  NVM_INTCTRL = 0x00;
  return;
}

// External button IRQ wakeup
ISR(PORTA_INT0_vect)
{

  // Enable the yellow LEDs after the debug button is pressed
  yellow_leds_on();
  _delay_ms(3000);
  yellow_leds_off();
  return;
}

// TMR F0 Compare ISR
ISR(TCF0_OVF_vect)
{
  uint8_t spi, isloaded;

  uint8_t temp_buffer_index = 0; // Use to keep track of which sample we are on
  uint8_t temp_buffer[16]; // Create an array to hold the current sample

  // Two channels need to be dumped
  // Grab the actual samples
  for (int i = 0; i < 8; i++) { // TODO: Fix hardcoded number of channels

    PORTD_OUT &= ~( 1 << 4 );
    // Send a dummy byte
    SPID_DATA = 0x00;
    while(!(SPID_STATUS & (1<<7))) { } // Wait for the SPI transaction to finish
    spi = SPID_DATA; // This is is the top 8 msb of our sample
    SPID_DATA = 0x00; // Start the background transaction again
    isloaded = 0x00;
    do { // Store the first byte while waiting for the second
      if ( isloaded == 0x00 ) {
        isloaded = 0x01;
          temp_buffer[temp_buffer_index] = spi;
          temp_buffer_index++; // Increment the index
      }
    } while(!(SPID_STATUS & (1<<7))); // While waiting for the 8 lsb to show up

    spi = SPID_DATA; // This is now the 8 lsb of our sample
    // Store the second half of our sample
    temp_buffer[temp_buffer_index] = spi;
    temp_buffer_index++; // Increment the index
    PORTD_OUT |= ( 1 << 4 ); // Pull SS High

    // Now we need to delay long enough to fulfill the adc's tconv specification (3.2 uS)
    TCC0_CNT = 0x0000; // Clear any existing value on the timer
    TCC0_CTRLA = 0x01; // Start the timer with no prescaler

    // Now just wait for the time to elapse by polling the overflow interrupt
    // The delay is absolutely necessary!! If it is too short you will get giberish
    while (!(TCC0_INTFLAGS & 1<<0)); // Wait for the interrupt flag to get set
    // Reset the interrupt flag
    TCC0_INTFLAGS |= ( 1 << 0 );
  }

  // We now have 2 junk samples and 6 valid samples stored in temp_buffer
  // (The two samples (four bytes) stored in temp_buffer[0] to temp_buffer[3] are junk)
  for (int i = 0; i < 12; i++) {
    // Load a sample into the buffer
    flash_buffer[flash_current_buffer][flash_buffer_index] = temp_buffer[i+4]; // Note the 4 byte
offset
    flash_buffer_index++; // Increment the index
    i++; // We actually increment the counter ourselves to compensate for the fact that were writing
two bytes per iteration
    flash_buffer[flash_current_buffer][flash_buffer_index] = temp_buffer[i+4]; // Note the 4 byte
offset
    flash_buffer_index++; // Increment the index

    // Now we need to see if our buffer is full
    // The flash_buffer_index will loop around to 0 if the buffer is full
    if (flash_buffer_index == 0) { // Time to swap buffers
      // We're basically assuming that the other buffer has been emptied by now
      // The buffer emptying is taken care of by the main function
      if (flash_current_buffer == 1) {
        flash_current_buffer = 0;
      } else {
        flash_current_buffer = 1;
      }
      flash_buffer_index = 0; // Reset the index
      flash_buffer_is_full = 1; // This will tell the main function to empty the buffer
    }
  }

  return;
}

// RTC Compare ISR
ISR(RTC_COMP_vect)
{

  // This is no longer being used
  return;
}

// XCVR IRQ wakeup
ISR(PORTC_INT0_vect)
{
    // Read the incoming packet
    hal_frame_read( &myframe );
    hal_register_read( 0x0F ); // Read interrupts so that the TX resets the IRQ flag

    opcode = myframe.data[9]; // Update the opcode


  return;
}

void timer_configuration() {

  /*
    Attempt to configure timer F0 for our use
  */
  TCF0_CTRLA = 0x00; // Disable the timer
  TCF0_CTRLD = 0x00; // Disable events
  TCF0_CTRLD = 0x00; // Use in 16 bit mode
  TCF0_INTCTRLA = 0x00; // For now we will ignore overflows/underflows and other timer errors
  // Set the desired trigger time...
  //TCF0_PER = 0b1111010000100100; // Set the timer period for 1/2 second (if you're using an 8 MHz
source)
  //TCF0_PER = 0b0000100111000100; // Set the timer period for 1/50 second
  //TCF0_PER = 0b10011100010; // Set the timer period for 100 Hz
  TCF0_PER = 0b1001110001; // Set the timer period for 200 Hz
  TCF0_CCA = 0x00; // This is unused in the current mode

  TCF0_CTRLB = 0x00; // Disable the CCs
  TCF0_CTRLFSET = 0x08; // Force timer restart
  //TCF0_INTCTRLA = 0x03; // Enable overflow as a high level interrupt
  TCF0_INTCTRLA = 0x02; // Enable the overflow interrupt as medium level
  TCF0_INTCTRLB = 0x00; // Disable all CC interrupts
  TCF0_CNT = 0x0000; // Clear any existing value
  //TCF0_CTRLA = 0x05; // Start the timer with a 64 prescaler

  /*
    Attempt to configure timer C0 for our use
  */
  TCC0_CTRLA = 0x00; // Disable the timer
  TCC0_CTRLD = 0x00; // Disable events
  TCC0_CTRLD = 0x00; // Use in 16 bit mode
  // Set the desired trigger time...
  TCC0_PER = 0b0000001000011010; // Set the timer period for 3.25 uS
  //TCC0_PER = 0b0000100111000100; // Set the timer period for 20 mS
  TCC0_CCA = 0x00; // This is unused in the current mode

  TCC0_CTRLB = 0x00; // Disable the CCs
  TCC0_CTRLFSET = 0x08; // Force timer restart
  TCC0_INTCTRLA = 0x00; // We do not want the interrupts
  TCC0_INTCTRLB = 0x00; // Disable CC interrupts
  TCC0_CNT = 0x0000; // Clear any existing value
  //TCC0_CTRLA = 0x01; // Start the timer with no prescaler

  /*
    RTC Configuration
  */
  RTC_CTRL = 0x00; // Shut off the RTC so it doesn't immediately start counting
  CLK_RTCCTRL = 0x0B; // Select the external 32KHz crystal as the RTC source and enable the crystal
  RTC_INTFLAGS = 0x00; // Clear the RTC interrupts
  while (RTC_STATUS & (1 << 0)) {} // Wait until the SYNCBUSY bit is clear
  // Set the TMR period. The low byte must be written first if you're doing it manually!
  RTC_PER = 0b0011000000000000; // Set to 12288
  RTC_CNT = 0x0000; // Clear any existing value
  //RTC_INTCTRL = 0x08; // Set the RTC compare IRQ as a medium level interrupt
  //RTC_INTCTRL = 0x0C; // Set the RTC compare IRQ as a high level interrupt
  //RTC_CTRL = 0x03; // Start the RTC with prescaler of 8
  RTC_INTCTRL = 0x00; // Disable RTC interrupts
}


int main() {


/* Set I/O port directions and values */


  /* INITIALIZATION */

  PORTA_DIR = 0b11111011; // 1 = output, 0 = input
  PORTA_OUT = 0b00110000; //not sleep 11g accel and set to 11g mode
  PORTB_DIR = 0b10001001;
  PORTB_OUT = 0b00000000;

  PORTC_DIR = 0b10111011;
  PORTC_OUT = 0b00010011;

  PORTD_DIR = 0b10111011;
  PORTD_OUT = 0b00010001;

  PORTE_DIR = 0b00111011;
  PORTE_OUT = 0b00110011;

  PORTF_DIR = 0xFF;
  PORTF_OUT = 0b01110011; //xcvr not sleep, 3g mode selected, 3g not sleep,

  /* CLOCK SETUP */
  CCP = 0xD8,              // IO Register Protection
  //CLK_PSCTRL = 0b00000011; // Set prescalers... Currently FCPU = FPER = 16MHz / 2 / 2 = 4 Mhz
  CLK_PSCTRL = 0b00000100; // Set prescalers... In theory this should be 16 MHz / 2 = 8 Mhz
  //CLK_PSCTRL = 0b00000000; // Set prescalers... In theory this should be 16 MHz

  /* INTERRUPT CONFIGURATION */
  PMIC_CTRL = 0x07;
  // Configure external button interrupt
  //PORTA_INTCTRL = 0x03; // Set INT0 to high level
  PORTA_INTCTRL |= 0x02; // Set INT0 to medium level
  PORTA_INT0MASK |= 0x04; // Enable bit 2 on the INT0 IRQ mask
  PORTA_PIN2CTRL = 0b00011010; // Set A2 to be pulled up and trigger on falling edge

  // Configure SLEEP mode to idle
  SLEEP_CTRL = 0x01;

  /*
    Useful variable declarations
  */

  test_container current_test; // Create a structure to hold the test details

  // The following vars hold info from EEPROM
  uint8_t nextTestNum, nextAvailableSector, nextAvailablePage, nextAvailableByte;

  // Variable for SPI data
  uint8_t spi;

  // Variables used while recording samples
  uint8_t buffertoempty = 0, junk;
  flash_buffer_is_full = 0;

  /*
    Initializations
  */

  for (int i = 0; i < 3; i++) {
    current_test.start[ i ] = 0;
    current_test.stop[ i ] = 0;
  }
  for (int i = 0; i < 4; i++) {
    current_test.timestamp_low[ i ] = 0;
    current_test.timestamp_high[ i ] = 0;
  }
  current_test.duration[0] = DEFAULT_CAPTURE_TIME & 0xFF; // Grab the bottom bytes
  current_test.duration[1] = (uint8_t)((DEFAULT_CAPTURE_TIME >> 8) & 0xFF);

  current_test.adc_channels = DEFAULT_ADC_CHANNELS; // Enable all six channels by default
  current_test.period[0] = DEFAULT_SAMPLE_PERIOD & 0xFF; // Grab the bottom bytes
  current_test.period[1] = (uint8_t)((DEFAULT_SAMPLE_PERIOD >> 8) & 0xFF);

  current_test.errata[0] = ZERO;
  current_test.errata[1] = ZERO;

  // The following line shows how to enable the EEPROM IRQ but you can't enable it until after you
have started the EEPROM write operation
  //NVM.INTCTRL = 0x03; //Enable the EEPROM ready IRQ

  _delay_ms(10);
  visual_twi_init();
  led_on_mode();


  red_leds_off();
  yellow_leds_off();
  // New Code Here
  _delay_us(10);
  red_leds_on();
  _delay_ms(1000);
  red_leds_off();

  // xcvr, flash, adc setup
  xcvr_spi_init();
  adc_spi_init();
  flash_spi_init();

  INTEN(); // Enable interrupts

  /*
    IMPORTANT: Interrupts must be enabled before trying to access the EEPROM!
    Otherwise the microcontroller has no way to wake up from sleep mode.
  */

  /*
    Retrieve some EEPROM values
    I've implemented EEPROM wear leveling using circular buffers.
    The process should be transparent to any functions requesting to read or write EEPROM values.
    With the current implementation, EEPROM bytes should be good to 400K write cycles (rather than the
standard 100K).
  */

  // EEPROM General Initialization
  EEPROM_FlushBuffer();
  EEPROM_DisableMapping();

  // The following example demonstrates EEPROM usage
  eeprom_pageinquestion = EEPROM_CONFIGURATION_PAGE;
  EEPROM_FlushBuffer();
  findCurrentEepromAddr( &EeBufPtr );
  nextTestNum = EeReadValue( EeBufPtr, NEXT_TEST_NUM );
  nextAvailableSector = EeReadValue( EeBufPtr, NEXT_AVAILABLE_SECTOR );
  nextAvailablePage = EeReadValue( EeBufPtr, NEXT_AVAILABLE_PAGE );
  nextAvailableByte = EeReadValue( EeBufPtr, NEXT_AVAILABLE_BYTE );

  // Update the current_test structure
  current_test.start[0] = nextAvailableSector;
  current_test.start[1] = nextAvailablePage;
  current_test.start[2] = nextAvailableByte;
  for (int i = 0; i < 3; i++) {
    current_test.stop[i] = current_test.start[i];
  }

  // XCVR Testing
  uint8_t incoming;

  // Reset the chip
  tat_reset_trx();

  // Ask the chip to go into TRX_OFF
  HAL_SS_LOW();
  hal_subregister_write( SR_TRX_CMD, CMD_FORCE_TRX_OFF );
  _delay_us( TIME_P_ON_TO_TRX_OFF ); //Wait for the transition to be complete.
  HAL_SS_HIGH();


  // Tell the chip to implement clock changes immediately
  HAL_SS_LOW();
  hal_subregister_write( SR_CLKM_SHA_SEL, 0x00 );
  HAL_SS_HIGH();

  // Change the clock output to 16MHz
  HAL_SS_LOW();
  hal_subregister_write( SR_CLKM_CTRL, 0x05 );
  HAL_SS_HIGH();

  // Swap to 16MHz clock
  _delay_us(10);

  PORTCFG.CLKEVOUT = 0x00; // Don't output the clock on any external pins

  OSC_XOSCCTRL = 0xC0;     // Select 16 MHz external clk
  OSC.CTRL = 0x09;         // Enables the external oscillator and keeps the 2MHz clock on as well
  do {_delay_ms(2);} while ((OSC.STATUS & 0x08) == 0x00);   //wait for stability
  OSC.CTRL = 0x08;         // Disables the 2MHz internal clock
  CCP = 0xD8;              // IO Register Protection
  CLK.CTRL   = 0x03;       // Sets external clock out
  HAL_SS_LOW();
  hal_subregister_write( SR_CHANNEL, 0x0F); // Set to channel 15 -- Should be 0x0F
  HAL_SS_HIGH();

  tat_set_short_address(SHORT_ADDRESS); //set address to 0x1001
  tat_set_pan_id(PAN_ID); //set to to 0xBEEF lol dont know what this does yet

  tat_set_device_role(0x00);
  tat_configure_csma(234, 0xE2);
  tat_use_auto_tx_crc(0x01);

  // Set IRQ Mask
  hal_register_write( 0x0E, 0x08 ); // Only enable the TX/RX Complete IRQ

  //tat_set_trx_state(RX_ON);
  tat_set_trx_state(RX_AACK_ON);

  //Build IEEE 802.15.4 frame
  txFrame[0] = 0x61;
  txFrame[1] = 0x88;
  txFrame[2] = 0;
  txFrame[3] = PAN_ID & 0xFF;
  txFrame[4] = (PAN_ID >> 8) & 0xFF;
  txFrame[5] = DEST_ADDRESS & 0xFF;
  txFrame[6] = (DEST_ADDRESS >> 8) & 0xFF;
  txFrame[7] = SHORT_ADDRESS & 0xFF;
  txFrame[8] = (SHORT_ADDRESS >> 8) & 0xFF;

  clearTXErrorFlag();
  rxFlag = 1;

  _delay_us(200);
  hal_register_read( 0x01 ); //Read Status
  hal_register_read( 0x06 ); // Read RSSI
  hal_register_read( 0x0E ); // Read IRQ Mask
  hal_register_read( 0x0F ); // Read IRQ Status

  incoming = hal_register_read( 0x0F ); // Read IRQ Status

  hal_subregister_read( SR_TX_PWR );
  // Clear any existing interrupts from the XCVR
  hal_register_read( 0x0F );

  // Configure XCVR IRQ
  //PORTC_INTCTRL = 0x02; // Set INT0 to medium level
  PORTC_INTCTRL = 0x03; // Set INT0 to high level
  PORTC_INT0MASK = 0x04; // Enable bit 2 on the INT0 IRQ mask
  PORTC_PIN2CTRL = 0x01; // Set C2 to trigger IRQ on rising edge

  // Configure the timers with default values
  timer_configuration();

  // More variables
  uint8_t spibuffer, spibuffer2;
  uint8_t isloaded;
  uint8_t realtime_buffer_index;


  while (1) {

    switch( opcode ) {

      // Start capturing data samples
      case 0x20:
        opcode = 0x00;
        red_leds_off();
        yellow_leds_off();
        // Start the timers

        TCF0_CTRLA = 0x05; // Start the timer with a 64 prescaler to sample at 50 Hz

        // We need to adjust the TMR start value to trip immediately and trigger an interrupt
        // TODO

        //Store timestamp data... should be myframe.data[10-17]
        //Frame Data Format: [0x20] [byte 7 (msb)] [byte 6] [byte 5] [byte 4] [byte 3] [byte 2] [byte
1] [byte 0 (msb)]
        //timestamp[] Format: [byte 7 (msb)] [byte 6] [byte 5] [0x00] [byte 4] [byte 3] [byte 2]
[0x00] [byte 1] [byte 0] [0x00] [0x00]
```

```
        // Note that we only receive 8 bytes for the timestamp but we end up sending 12 bytes back
since come zeros are implied
        current_test.timestamp_high[0] = myframe.data[10];
        current_test.timestamp_high[1] = myframe.data[11];
        current_test.timestamp_high[2] = myframe.data[12];
        current_test.timestamp_high[3] = myframe.data[13];
        current_test.timestamp_low[0] = myframe.data[14];
        current_test.timestamp_low[1] = myframe.data[15];
        current_test.timestamp_low[2] = myframe.data[16];
        current_test.timestamp_low[3] = myframe.data[17];

        // Variable initializations
        flash_buffer_is_full = 0; // Reset the variable for good measure (but we should never have to)
        flash_buffer_index = 0; // Start back at index 0
        flash_current_buffer = 0; // Go back to using buffer 0

        yellow_leds_on();

        while (opcode != 0x30) {

          // Flush the flash buffer when necessary
          // I initially set the buffer empty
          if (flash_buffer_is_full == 1) {

            // Determine which buffer to flush
            if (flash_current_buffer == 0) { // We must need to flush buffer 1
              buffertoempty = 1;
            } else {
              buffertoempty = 0;
            }
            // We need to tell the flash chip that we'd like to write the next page in line
            // The current sector, page and byte is stored in current_test.stop[i]
            // Request write permission
            PORTD_OUT &= ~(0x01); // SS low
            USARTD0_DATA = 0x06; // Request write permission
            while(!(USARTD0_STATUS & 1<<6)); // Wait until the TX is finished
            USARTD0_STATUS = 1<<6; // Reset the TX complete IRQ
            spibuffer = USARTD0_DATA;
            spibuffer2 = USARTD0_DATA; // Immediately read the next byte to flush the buffer
            PORTD_OUT |= 0x01; // Release SS line

            // Now that we've been granted write permission we just need to program the page
            // This part of the code is expected to be randomly interrupted by timers (and even the
transceiver)
            PORTD_OUT &= ~(0x01); // SS low
            USARTD0_DATA = 0x02; // Issue write page command
            while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
            USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            USARTD0_DATA = current_test.stop[0]; // Load up the sector address
            while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
            USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            USARTD0_DATA = current_test.stop[1]; // Load up the page address
            while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
            USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            USARTD0_DATA = 0x00; // Byte address (we always begin programming at byte 0)
            while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
            USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            for (int i = 0; i < FLASH_BYTES_IN_PAGE; i++) {
              USARTD0_DATA = flash_buffer[buffertoempty][i]; // Load in more data
              while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
              USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            }
            while(!(USARTD0_STATUS & 1<<6)); // Wait until the upload is finished
            USARTD0_STATUS = 1<<6; // Reset the transmission complete flag
            PORTD_OUT |= 0x01; // Release SS line
            spibuffer = USARTD0_DATA;
            spibuffer2 = USARTD0_DATA; // Read the status byte
            // NOTE: After we finish writing to the page we will lose our write permission and have to
request it again!
            flash_buffer_is_full = 0; // Reset the variable

            // Empty the SPI buffer
            do {
              junk = USARTD0_DATA;
            } while (USARTD0_STATUS & 1<<7);


            // Increment the page count
            current_test.stop[1]++;

            // Note that the page count will automatically roll around to 0
            // If it is zero that means we need to increment the sector number
            if (current_test.stop[1] == 0) {
              current_test.stop[0]++; // This will not ever overflow because the device has less than
256 sectors

              // We also need to be careful to keep an eye on our current sector number
              if (current_test.stop[0] == FLASH_NO_SECTORS) {
                while (1) {
                  // TODO
                  red_leds_on();
                  _delay_ms(1500);
                  red_leds_off();
                  _delay_ms(1500);
                }

                // TODO: Check to see if there is more memory available
                // If there is, change to sector 0. If there isn't trigger an out of memory error.
              }
            }
            // Read Status Reg
            PORTD_OUT &= ~(0x01); // SS low
            USARTD0_DATA = 0x05; // Read status register
            while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
            USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
            USARTD0_DATA = 0x00; // Load up the new packet
            while(!(USARTD0_STATUS & 1<<6)); // Wait until the TX is finished
            USARTD0_STATUS = 1<<6; // Reset the TX complete IRQ
            spibuffer = USARTD0_DATA;
            spibuffer2 = USARTD0_DATA; // Read the status byte
            PORTD_OUT |= 0x01; // Release SS line

          } // end if loop


          // Else

          // Do nothing for now

        } // If we made it here then we need to stop recording

        // For now we will just turn off the timers and reset the count
        // TODO: Set the timers to immediately interrupt
        TCF0_CTRLA = 0x00;
        //RTC_CTRL = 0x00;
        TCF0_CNT = 0x0000; // Clear any existing value
        //RTC_CNT = 0x0000; // Clear any existing value

        // Now we need to actually flush out the partial buffer to flash
        // The current sector, page and byte is stored in current_test.stop[i]
        // Request write permission
        PORTD_OUT &= ~(0x01); // SS low
        USARTD0_DATA = 0x06; // Request write permission
        while(!(USARTD0_STATUS & 1<<6)); // Wait until the TX is finished
        USARTD0_STATUS = 1<<6; // Reset the TX complete IRQ
        spibuffer = USARTD0_DATA;
        spibuffer2 = USARTD0_DATA; // Immediately read the next byte to flush the buffer
        PORTD_OUT |= 0x01; // Release SS line

        // Now that we've been granted write permission we just need to program the page
        PORTD_OUT &= ~(0x01); // SS low
        USARTD0_DATA = 0x02; // Issue write page command
        while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
        USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
        USARTD0_DATA = current_test.stop[0]; // Load up the sector address
        while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
        USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
        USARTD0_DATA = current_test.stop[1]; // Load up the page address
        while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
        USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
        USARTD0_DATA = 0x00; // Byte address (we always begin programming at byte 0)
        while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
        USARTD0_STATUS = 1<<5; // Reset the DRE IRQ

        for (int i = 0; i < flash_buffer_index; i++) {
          USARTD0_DATA = flash_buffer[flash_current_buffer][i]; // Load in more data
          while(!(USARTD0_STATUS & 1<<5)); // Wait until the data register is empty
          USARTD0_STATUS = 1<<5; // Reset the DRE IRQ
        }

        while(!(USARTD0_STATUS & 1<<6)); // Wait until the upload is finished
        USARTD0_STATUS = 1<<6; // Reset the transmission complete flag
        PORTD_OUT |= 0x01; // Release SS line
        spibuffer = USARTD0_DATA;
        spibuffer2 = USARTD0_DATA; // Read the status byte
        // NOTE: After we finish writing to the page we will lose our write permission and have to
request it again!

        // Empty the SPI buffer
        do {
          junk = USARTD0_DATA;
        } while (USARTD0_STATUS & 1<<7);

        // Update everything

        /*
          The current test information needs to be updated
        */
        // Update the starting information in case we have been capturing multiple tests in one
session
        EEPROM_FlushBuffer();
        eeprom_pageinquestion = ((nextTestNum * 3) + 0); // Select the info page containing starting
info for the current test
        findCurrentEepromAddr( &EeBufPtr );

        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[0] = current_test.start[0]; // The start page
        eeprom_buffer[1] = current_test.start[1]; // The start sector
        eeprom_buffer[2] = current_test.start[2]; // The start byte
        eeprom_buffer[3] = current_test.timestamp_high[0]; // t_high
        eeprom_buffer[4] = current_test.timestamp_high[1]; // t_high
        eeprom_buffer[5] = current_test.timestamp_high[2]; // t_high
        eeprom_buffer[6] = current_test.timestamp_high[3]; // t_high
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer

        EEPROM_FlushBuffer();
        eeprom_pageinquestion = ((nextTestNum * 3) + 1); // Select the info page containing the low
timestamp bytes
        findCurrentEepromAddr( &EeBufPtr );

        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[0] = current_test.timestamp_low[0]; // The start page
        eeprom_buffer[1] = current_test.timestamp_low[1]; // The start sector
        eeprom_buffer[2] = current_test.timestamp_low[2]; // The start byte
        eeprom_buffer[3] = current_test.timestamp_low[3]; // t_high
        eeprom_buffer[4] = 0x00; // duration // These numbers are unused right now
        eeprom_buffer[5] = 0x00; // duration
        eeprom_buffer[6] = 0x00; // adc channels
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer

        // Update the ending information for the test
        EEPROM_FlushBuffer();
        eeprom_pageinquestion = ((nextTestNum * 3) + 2); // Select the info page containing ending
info for the current test
        findCurrentEepromAddr( &EeBufPtr );
        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[0] = 0x00; // The period (unused right now)
        eeprom_buffer[1] = 0x00; // The period (unused right now)
        eeprom_buffer[2] = current_test.stop[0]; // The ending sector
        eeprom_buffer[3] = current_test.stop[1]; // The ending page
        eeprom_buffer[4] = (flash_buffer_index - 1); // The last byte that was written
        eeprom_buffer[5] = 0x00; // Errata (unused right now)
        eeprom_buffer[6] = 0x00; // Errata (unused right now)
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer


        // The system info also needs to be updated with the next available empty page

        // Now we need to let the microcontroller know where it should put future tests
        // Check to see if the next page is on another sector
        current_test.stop[1]++; // Increment current page
        if (current_test.stop[1] == 0 ) { // The next page must go on the next sector
          // Increment the sector number
          current_test.stop[0]++; // This will not ever overflow because the external flash device has
less than 256 sectors

          // We also need to be careful to keep an eye on our current sector number
          if (current_test.stop[0] == FLASH_NO_SECTORS) {
            while (1) {
              red_leds_on();
              _delay_ms(1500);
              red_leds_off();
              _delay_ms(1500);
            }

            // TODO: Check to see if there is more memory available
            // If there is, change to sector 0. If there isn't trigger an out of memory error.
          }
        }

        // At this point we should now be at the start of a fresh page (All tests must start at byte 0
of a fresh page)

        // Update the EEPROM Config Page
        eeprom_pageinquestion = EEPROM_CONFIGURATION_PAGE;
        EEPROM_FlushBuffer();
        findCurrentEepromAddr( &EeBufPtr );
        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[NEXT_TEST_NUM] = nextTestNum + 1; // Increment the test ID number
        nextAvailableSector = current_test.stop[0];
        eeprom_buffer[NEXT_AVAILABLE_SECTOR] = nextAvailableSector;
        nextAvailablePage = current_test.stop[1];
        eeprom_buffer[NEXT_AVAILABLE_PAGE] = nextAvailablePage;
        nextAvailableByte = 0x00; // Always start at byte 0 on the page
        eeprom_buffer[NEXT_AVAILABLE_BYTE] = nextAvailableByte;
        eeprom_buffer[4] = 0x00; // These are currently unused
        eeprom_buffer[5] = 0x00;
        eeprom_buffer[6] = 0x00;
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer


        // Reset some variables
        flash_buffer_is_full = 0; // Reset the variable for good measure (but we should never have to)
        flash_buffer_index = 0;
        flash_current_buffer = 0;
        nextTestNum++; // Get ready to write to the next test

        // Update the current_test structure

        current_test.start[0] = nextAvailableSector;
        current_test.start[1] = nextAvailablePage;
        current_test.start[2] = nextAvailableByte;
        for (int i = 0; i < 3; i++) {
          current_test.stop[i] = current_test.start[i];
        }

        yellow_leds_off();


        break;

      // Send a chunk of data back
      case 0x41:
        opcode = 0x00;

        // TODO:
        volatile uint8_t txbuffer[96];
        volatile uint8_t txindex = 0;
        uint8_t numberofsectorstoread = 0;

        uint8_t current_sector;
        uint8_t current_page;
        uint8_t current_byte;
        uint8_t end_sector;
        uint8_t end_page;
        uint8_t end_byte;


        current_sector = 0;
        current_page = 0;
        current_byte = 0;

        /***** Start new code *****/

        // Start our buffer index at zero
        txindex = 0;

        // TODO: This code is not capable of handling tests that loop around the end of the flash
        for (int i = 0; i < nextTestNum; i++) {
          /* Start by sending timestamp info for the current test */
          eeprom_pageinquestion = (i * 3); // Get info for the current test
          EEPROM_FlushBuffer();
          findCurrentEepromAddr( &EeBufPtr );
          // get the beginning of flash for test pointers
```

```
          current_sector = EeReadValue( EeBufPtr, 0 );
          current_page = EeReadValue( EeBufPtr, 1 );
          current_byte = EeReadValue( EeBufPtr, 2 );

          // grab the high bytes of the time stamp
          txbuffer[txindex] = EeReadValue( EeBufPtr, 3 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          txbuffer[txindex] = EeReadValue( EeBufPtr, 4 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          txbuffer[txindex] = EeReadValue( EeBufPtr, 5 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          txbuffer[txindex] = 0x00;
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          txbuffer[txindex] = EeReadValue( EeBufPtr, 6 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }


          // Retrieve the rest of the timestamp info
          eeprom_pageinquestion = ((i * 3) + 1);
          EEPROM_FlushBuffer();
          findCurrentEepromAddr( &EeBufPtr );
          // Get the low bytes of the time stamp
          txbuffer[txindex] = EeReadValue( EeBufPtr, 0 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          txbuffer[txindex] = EeReadValue( EeBufPtr, 1 );
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }

            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }
          txbuffer[txindex] = 0x00;
          txindex++;
          // Check to see if we need to empty the transmitter buffer
          if (txindex == 96) {
            opcode = 0x88; // This opcode means that the module is waiting on the GUI
            // Time to send the data
            transceiver_send_data(txbuffer,96);
            while (opcode != 0x41) {
              if (opcode == 0x42) { // We have received a request to retransmit the data
                opcode = 0x00;
                transceiver_send_data(txbuffer,96);
              }
```

```
    }; // Wait
    // Reset the opcode
    opcode = 0x00;
    // Reset the txindex
    txindex = 0;
  }

  txbuffer[txindex] = EeReadValue( EeBufPtr, 2 );
  txindex++;
  // Check to see if we need to empty the transmitter buffer
  if (txindex == 96) {
    opcode = 0x88; // This opcode means that the module is waiting on the GUI
    // Time to send the data
    transceiver_send_data(txbuffer,96);
    while (opcode != 0x41) {
      if (opcode == 0x42) { // We have received a request to retransmit the data
        opcode = 0x00;
        transceiver_send_data(txbuffer,96);
      }

    }; // Wait
    // Reset the opcode
    opcode = 0x00;
    // Reset the txindex
    txindex = 0;
  }

  txbuffer[txindex] = EeReadValue( EeBufPtr, 3 );
  txindex++;
  // Check to see if we need to empty the transmitter buffer
  if (txindex == 96) {
    opcode = 0x88; // This opcode means that the module is waiting on the GUI
    // Time to send the data
    transceiver_send_data(txbuffer,96);
    while (opcode != 0x41) {
      if (opcode == 0x42) { // We have received a request to retransmit the data
        opcode = 0x00;
        transceiver_send_data(txbuffer,96);
      }

    }; // Wait
    // Reset the opcode
    opcode = 0x00;
    // Reset the txindex
    txindex = 0;
  }

  txbuffer[txindex] = 0x00;
  txindex++;
  // Check to see if we need to empty the transmitter buffer
  if (txindex == 96) {
    opcode = 0x88; // This opcode means that the module is waiting on the GUI
    // Time to send the data
    transceiver_send_data(txbuffer,96);
    while (opcode != 0x41) {
      if (opcode == 0x42) { // We have received a request to retransmit the data
        opcode = 0x00;
        transceiver_send_data(txbuffer,96);
      }

    }; // Wait
    // Reset the opcode
    opcode = 0x00;
    // Reset the txindex
    txindex = 0;
  }

  txbuffer[txindex] = 0x00;
  txindex++;
  // Check to see if we need to empty the transmitter buffer
  if (txindex == 96) {
    opcode = 0x88; // This opcode means that the module is waiting on the GUI
    // Time to send the data
    transceiver_send_data(txbuffer,96);
    while (opcode != 0x41) {
      if (opcode == 0x42) { // We have received a request to retransmit the data
        opcode = 0x00;
        transceiver_send_data(txbuffer,96);
      }
```

```
            }; // Wait
            // Reset the opcode
            opcode = 0x00;
            // Reset the txindex
            txindex = 0;
          }

          // Move to final page of info for the current test
          eeprom_pageinquestion = ((i * 3) + 2);
          EEPROM_FlushBuffer();
          findCurrentEepromAddr( &EeBufPtr );
          // Retrieve the ending sector, page, and byte for the current test
          end_sector = EeReadValue( EeBufPtr, 2 );
          end_page = EeReadValue( EeBufPtr, 3 );
          end_byte = EeReadValue( EeBufPtr, 4 );

          // Initialize our connection to the flash
          PORTD_OUT &= ~(0x01); // SS low

          USARTD0_DATA = 0x03; // Request read access to the chip
          while(!(USARTD0_STATUS & 1<<5)); // Wait until there is more room in the transmit buffer
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Read the returned byte

          USARTD0_DATA = current_sector; // Specify the sector
          while(!(USARTD0_STATUS & 1<<5)); // Wait until there is more room in the transmit buffer
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Read the returned byte
          USARTD0_DATA = current_page; // Specify the page
          while(!(USARTD0_STATUS & 1<<5)); // Wait until there is more room in the transmit buffer
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Read the returned byte
          USARTD0_DATA = current_byte; // Specify the byte
          while(!(USARTD0_STATUS & 1<<5)); // Wait until there is more room in the transmit buffer
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Read the returned byte

          // We now have an open connection to the flash chip
          // All we have to do is send it a dummy byte and it will return the next byte of flash in
line

          // Start by sending two junk bytes to preload our receive buffer
          USARTD0_DATA = 0x00; // Send a dummy byte
          while(!(USARTD0_STATUS & 1<<5)); // Wait until the DRE is finished
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Junk
          USARTD0_DATA = 0x00; // Send a dummy byte
          while(!(USARTD0_STATUS & 1<<5)); // Wait until the DRE is finished
          USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
          spi = USARTD0_DATA; // Junk

          // Determine the number of sectors to read
          if (end_sector >= current_sector) {
            numberofsectorstoread = (end_sector - current_sector) + 1; // We always have to read from
at least one sector
          } else {
            // In this case, we must have looped around the end of flash
            numberofsectorstoread = (FLASH_NO_SECTORS - current_sector) + end_sector + 1; // We still
need to read from at least one sector
          }

          // Read all full sectors in the test
          while (numberofsectorstoread > 1) {
            // In this case, we know that we need to read the entire sector

            do {

              // Prepare to read the current page
              current_byte = 0; // Reset the byte index
              // We need to start from the current page and continue to read the rest of the sector
              do {
                // We need to grab the test data
                USARTD0_DATA = 0x00; // Send a dummy byte
                while(!(USARTD0_STATUS & 1<<5)); // Wait until the DRE is finished
                USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
                txbuffer[txindex] = USARTD0_DATA; // Place the byte in the buffer

                // Increment our counters
                txindex++;

                // Check to see if we need to empty the transmitter buffer
                if (txindex == TX_BUFFER_SIZE) {
                  opcode = 0x88; // This opcode means that the module is waiting on the GUI
                  // Time to send the data
                  transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
                  while (opcode != 0x41) {
                    if (opcode == 0x42) { // We have received a request to retransmit the data
                      opcode = 0x00;
                      transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
                    }

                  }; // Wait
                  // Reset the opcode
                  opcode = 0x00;
                  // Reset the txindex
                  txindex = 0;
                }
                current_byte++; // Move to the next byte in the page
              } while (current_byte != 0); // This will become zero on the 256th iteration (i.e. when
we are done with the current page)

              // Increment the current page
              current_page++;

            } while (current_page != 0); // The current page will wrap around to 0 when the sector has
been completely read


            // Decrement the remaining number of sectors
            numberofsectorstoread--;

          }

          // When the numberofsectorstoread reaches 1, we know that we are on a partial sector
          // Now we need to read all the remaining pages that are 100% full
          while (current_page < end_page) {
            current_byte = 0; // Reset the byte index
            // We need to read the entire page of flash
            do {
              // We need to grab the test data
              USARTD0_DATA = 0x00; // Send a dummy byte
              while(!(USARTD0_STATUS & 1<<5)); // Wait until the DRE is finished
              USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
              txbuffer[txindex] = USARTD0_DATA; // Place the byte in the buffer

              // Increment our counters
              txindex++;

              // Check to see if we need to empty the transmitter buffer
              if (txindex == TX_BUFFER_SIZE) {
                opcode = 0x88; // This opcode means that the module is waiting on the GUI
                // Time to send the data
                transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
                while (opcode != 0x41) {
                  if (opcode == 0x42) { // We have received a request to retransmit the data
                    opcode = 0x00;
                    transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
                  }

                }; // Wait
                // Reset the opcode
                opcode = 0x00;
                // Reset the txindex
                txindex = 0;
              }
              current_byte++; // Move to the next byte in the page
            } while (current_byte != 0); // This will become zero on the 256th iteration (i.e. when we
are done with the current page)

            // Increment the current page
            current_page++;
          }

          // Lastly, we just need to finish reading the final page
          do {
              USARTD0_DATA = 0x00; // Send a dummy byte
              while(!(USARTD0_STATUS & 1<<5)); // Wait until the DRE is finished
              USARTD0_STATUS = 1<<5; // Reset the DRE complete IRQ
              txbuffer[txindex] = USARTD0_DATA; // Place the byte in the buffer

              // Increment our counters
              txindex++;
```

```
        // Check to see if we need to empty the transmitter buffer
        if (txindex == TX_BUFFER_SIZE) {
          opcode = 0x88; // This opcode means that the module is waiting on the GUI
          // Time to send the data
          transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
          while (opcode != 0x41) {
            if (opcode == 0x42) { // We have received a request to retransmit the data
              opcode = 0x00;
              transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
            }

          }; // Wait
          // Reset the opcode
          opcode = 0x00;
          // Reset the txindex
          txindex = 0;
        }
        current_byte++; // Move to the next byte in the page
    } while (current_byte <= end_byte);

    // If we reached here then we are done sending the contents of the current test
    // Tack on an additional 4 bytes of FF
    for (int k = 0; k < 4; k++){
      txbuffer[txindex] = 0xFF; // Put a byte of FF into the buffer
      txindex++;

      // Check to see if we need to empty the transmitter buffer
      if (txindex == TX_BUFFER_SIZE) {
        opcode = 0x88; // This opcode means that the module is waiting on the GUI
        // Time to send the data
        transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
        while (opcode != 0x41) {
          if (opcode == 0x42) { // We have received a request to retransmit the data
            opcode = 0x00;
            transceiver_send_data(txbuffer,TX_BUFFER_SIZE);
          }

        }; // Wait
        // Reset the opcode
        opcode = 0x00;
        // Reset the txindex
        txindex = 0;
      }
    }

  } // end for loop through each test

  PORTD_OUT |= 0x01; // Release SS line

  // Tack on an additional 4 bytes of FFs (this way we send 8 FFs at the end)
  for (int k = 0; k < 4; k++){
    txbuffer[txindex] = 0xFF; // Put a byte of FF into the buffer
    txindex++;
    // Check to see if we need to empty the transmitter buffer
      if (txindex == 96) {
        opcode = 0x88; // This opcode means that the module is waiting on the GUI
        // Time to send the data
        transceiver_send_data(txbuffer,96);
        while (opcode != 0x41) {
          if (opcode == 0x42) { // We have received a request to retransmit the data
            opcode = 0x00;
            transceiver_send_data(txbuffer,96);
          }

        }; // Wait
        // Reset the opcode
        opcode = 0x00;
        // Reset the txindex
        txindex = 0;
      }
  }

  // Send any remaining buffer contents
  if (txindex != 0 ) {
    transceiver_send_data(txbuffer, 96 );
  }

  // Fill the transceiver buffer with FFs
  for (int i = 0; i < 96; i++) {
    txbuffer[i] = 0xFF;
```

```
        }

        // If the computer requests additional data, we feed it FFs.
        // From what I can see, the computer seems to always request one more packet than it needs.
        while (opcode != 0x41) { }; // Wait until we get the request
        // Send the computer a packet of FFs
        transceiver_send_data(txbuffer,96);

        break;

      // Check for errors
      case 0x50:

        opcode = 0x00;
        uint8_t error_status[7] = {0, 'e', 'r', 'r', 'o', 'r', '\0'};

        // Transmit the info back to base
        transceiver_send_data(error_status, 7);

        break;

      // If the opcode is already 0x00 there's no need to continue
      case 0x00:
        break;


      // Erase Memory
      case 0x10: // TODO: check to make sure this works just like button
        opcode = 0x00;
        red_leds_on();

        EEPROM_FlushBuffer();
        for (int i = 0; i < 64; i++) {
          for (int j = 0; j < 4; j++) {
            EEPROM_WriteByte( i, (unsigned char)(j & EEPROM_BYTE_ADDRESS_MASK), 0x00);
          }
        }

        eeprom_pageinquestion = EEPROM_CONFIGURATION_PAGE;
        findCurrentEepromAddr( &EeBufPtr );
        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[NEXT_TEST_NUM] = 0x00;
        eeprom_buffer[NEXT_AVAILABLE_SECTOR] = 0x00;
        eeprom_buffer[NEXT_AVAILABLE_PAGE] = 0x00;
        eeprom_buffer[NEXT_AVAILABLE_BYTE] = 0x00;
        eeprom_buffer[4] = 0x00;
        eeprom_buffer[5] = 0x00;
        eeprom_buffer[6] = 0x00;
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer

        eeprom_pageinquestion = ZERO; // Preload information for the first test
        findCurrentEepromAddr( &EeBufPtr );
        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[0] = 0x00; // The starting sector
        eeprom_buffer[1] = 0x00; // The starting page
        eeprom_buffer[2] = 0x00; // The starting byte
        eeprom_buffer[3] = 0x00; // Timestamp data... this doesn't have to be initialized to anything
in particular
        eeprom_buffer[4] = 0x00; // Timestamp data... this doesn't have to be initialized to anything
in particular
        eeprom_buffer[5] = 0x00; // Timestamp data... this doesn't have to be initialized to anything
in particular
        eeprom_buffer[6] = 0x00; // Timestamp data... this doesn't have to be initialized to anything
in particular
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer

        eeprom_pageinquestion = 2; // Preload ending information for the first test
        findCurrentEepromAddr( &EeBufPtr );
        // Load up the 7 values before writing them out to EEPROM
        eeprom_buffer[0] = 0x00; // The period
        eeprom_buffer[1] = 0x00; // The period
        eeprom_buffer[2] = 0x00; // Ending sector
        eeprom_buffer[3] = 0x00; // Ending page
        eeprom_buffer[4] = 0x00; // Ending byte
        eeprom_buffer[5] = 0x00; // Eratta
        eeprom_buffer[6] = 0x00; // Eratta
        EeWriteBuffer(&EeBufPtr); // Load and write the buffer


        // Request write permission on the external flash
        PORTD_OUT &= ~(0x01); // SS low
        USARTD0_DATA = 0x06; // Request write permission
        while(!(USARTD0_STATUS & 1<<6)); // Wait until the TX is finished
        USARTD0_STATUS = 1<<6; // Reset the TX complete IRQ
        spi = USARTD0_DATA;
        spi = USARTD0_DATA; // Immediately read the next byte
        PORTD_OUT |= 0x01; // Release SS line

        // Issue bulk erase command
        // I hope you didn't want anything on the chip!
        PORTD_OUT &= ~(0x01); // SS low
        USARTD0_DATA = 0xC7; // Bulk erase command
        while(!(USARTD0_STATUS & 1<<6)); // Wait until the TX is finished
        USARTD0_STATUS = 1<<6; // Reset the TX complete IRQ
        spi = USARTD0_DATA;
        spi = USARTD0_DATA; // Immediately read the next byte
        PORTD_OUT |= 0x01; // Release SS line

        // Reset the volatile pointers
        // (EEPROM has already been updated)
        eeprom_pageinquestion = EEPROM_CONFIGURATION_PAGE;
        EEPROM_FlushBuffer();
        findCurrentEepromAddr( &EeBufPtr );
        nextTestNum = EeReadValue( EeBufPtr, NEXT_TEST_NUM );
        nextAvailableSector = EeReadValue( EeBufPtr, NEXT_AVAILABLE_SECTOR );
        nextAvailablePage = EeReadValue( EeBufPtr, NEXT_AVAILABLE_PAGE );
        nextAvailableByte = EeReadValue( EeBufPtr, NEXT_AVAILABLE_BYTE );

        // Update the current_test structure
        current_test.start[0] = nextAvailableSector;
        current_test.start[1] = nextAvailablePage;
        current_test.start[2] = nextAvailableByte;
        for (int i = 0; i < 3; i++) {
          current_test.stop[i] = current_test.start[i];
        }

        _delay_ms(8000);
        red_leds_off();
        yellow_leds_off();
        break;

      /*
        This case is used to provide the computer with information about the module's current memory
usage.
        We include information about the next available byte, where the memory is first utilized
        (this location could change depending on whether or not wear-leveling is implemented),
        and the total device capacity (including already used bytes).
      */
      case 0x70:
        opcode = 0x00;

        uint8_t mem_info[9];

        // We can check how much memory we've used by looking at the last written sector/page/byte of
the current test
        // On initialization, this is set from internal EEPROM and subsequent tests update this number
so it should always be valid
        mem_info[0] = current_test.stop[0];
        mem_info[1] = current_test.stop[1];
        mem_info[2] = current_test.stop[2];

        // Retrieve information about where the first test begins
        eeprom_pageinquestion = 0; // Information about the start Sector, Page, and Byte for test 0
will always be on page 0
        EEPROM_FlushBuffer();
        findCurrentEepromAddr( &EeBufPtr ); // You should never have to change this line. It's only
here to utilize the circular buffer
        mem_info[3] = EeReadValue( EeBufPtr, 0 ); // Contains the starting sector of the first test
        mem_info[4] = EeReadValue( EeBufPtr, 1 ); // Contains the starting page of the first test
        mem_info[5] = EeReadValue( EeBufPtr, 2 ); // Contains the starting byte of the first test

        // Include information about our device
        // We actually send the maximum index of each sector, page, and byte. (NOT the actual number
of sectors, pages, and bytes)
        // i.e. The sectors go from 0 to 31
        mem_info[6] = (FLASH_NO_SECTORS - 1);
        mem_info[7] = (FLASH_NO_PAGES - 1); // Prevent overflow
        mem_info[8] = (FLASH_BYTES_IN_PAGE - 1); // Prevent overflow

        // Transmit the info back to base
        transceiver_send_data(mem_info, 9);
        break;

      // Return a single sample from each channel
      case 0x60:
        realtime_buffer_index = 0; // Use to keep track of which sample we are on
        uint8_t realtime_buffer[16]; // Create an array to hold the current sample

        // Two channels need to be dumped
        // Grab the actual samples
        for (int i = 0; i < 8; i++) { // TODO: Fix hardcoded number of channels

          PORTD_OUT &= ~( 1 << 4 );

          // Send a dummy byte
          SPID_DATA = 0x00;
          while(!(SPID_STATUS & (1<<7))) { } // Wait for the SPI transaction to finish
          spi = SPID_DATA; // This is is the top 8 msb of our sample
          SPID_DATA = 0x00; // Start the background transaction again
          isloaded = 0x00;
          do { // Store the first byte while waiting for the second
            if ( isloaded == 0x00 ) {
              isloaded = 0x01;
                realtime_buffer[realtime_buffer_index] = spi;
                realtime_buffer_index++; // Increment the index
            }
          } while(!(SPID_STATUS & (1<<7))); // While waiting for the 8 lsb to show up

          spi = SPID_DATA; // This is now the 8 lsb of our sample
          // Store the second half of our sample
          realtime_buffer[realtime_buffer_index] = spi;
          realtime_buffer_index++; // Increment the index
          PORTD_OUT |= ( 1 << 4 ); // Pull SS High
          _delay_us(3); // Small delay (this is actually shorter than 1us due to the clock change from
2MHz to 8MHz)
          // The delay is absolutely necessary!! If it is too short you will get giberish
        }     // Transmit the info back to base
        transceiver_send_data(&realtime_buffer[4], 12); // Start sending data from index location 2

        break;

      //Ping
      case 0xF0:
        opcode = 0x00;
        _delay_ms(1); // Delay needed because the dongle can't send/receive too fast

        uint8_t pong[5] = {'p', 'o', 'n', 'g', '\0'};
        transceiver_send_data(pong, 5);
        break;

      default:
        break;
    }

    SLEEPCPU(); // Since the transceiver is interrupt driven, any transceiver activity will wake the
CPU from sleep

  }

return 0;
}
```

APPENDIX E: EXTRA.H HEADER FILE

```
/*
  This file was written by Justin Goins
*/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>

#ifndef MYSTRUCTURES
  #define MYSTRUCTURES 0

  #define SLEEPCPU() asm("sleep"    "\n\t")
  #define INTEN() asm("sei"   "\n\t")
  #define INTDIS() asm("cli"    "\n\t")

  /* Transceiver definitions */
  #define OPERATING_CHANNEL 0xF
  #define PAN_ID 0xBEEF
  #define SHORT_ADDRESS 0x1001 // Originally 0x1001
  #define DEST_ADDRESS 0x0001
  #define TX_BUFFER_SIZE 96

  /* EEPROM definitions */
  #define EEPROM_NO_PAGES     64
  #define EEPROM_BYTES_IN_PAGE 32
  // Masking out the byte address in a page
  #define EEPROM_BYTE_ADDRESS_MASK    0x1f
  #define EEPROM_CONFIGURATION_PAGE 60 // This is the EEPROM page that contains system defaults, etc

  // EEPROM Byte Locations
  // (these numbers represent where the values are located within the EEPROM page)
  #define NEXT_TEST_NUM 0
  #define NEXT_AVAILABLE_SECTOR 1
  #define NEXT_AVAILABLE_PAGE 2
  #define NEXT_AVAILABLE_BYTE 3

  /* FLASH definitions */
  #define FLASH_NO_SECTORS 32
  #define FLASH_NO_PAGES 256
  #define FLASH_BYTES_IN_PAGE 256

  /* Definitions for device recognition */
  #define PARTCODE      0xFA
  #define SIGNATURE_BYTE_1  0x1E
  #define SIGNATURE_BYTE_2  0x97
  #define SIGNATURE_BYTE_3  0x44

  // Pin Definitions
  #define ADC_PORT PORTD
  #define ADC_PORT_OUTCLR PORTD_OUTCLR
  #define ADC_PORT_OUTSET PORTD_OUTSET
  #define ADC_SS_PIN (1 << 4)

  // Random Constants
  #define ZERO 0x00
  #define ONE 0x01
  #define DEFAULT_CAPTURE_TIME 10 // In seconds (must be four bytes or less)
  #define OP_CODE_OFFET 9 // Where the op code is in the packet
  #define DEFAULT_ADC_CHANNELS 0b00111111
  #define DEFAULT_SAMPLE_PERIOD 50 // In Hz (must be four bytes or less)

  /*
    Structures
  */
  typedef struct{
      volatile uint8_t start[ 3 ]; // This will have a sector, page, and byte number
      uint8_t timestamp_high[ 4 ];
    uint8_t timestamp_low[ 4 ];
      uint8_t duration[ 2 ];
      uint8_t adc_channels;
    uint8_t period[ 2 ];
    volatile uint8_t stop[ 3 ]; // This will have a sector, page, and byte number
    uint8_t errata[ 2 ]; // This is for future use
  } test_container;

#endif
```

APPENDIX F: EEPROM_LEVELING.C SOURCE FILE

```c
/*******************************************************/
/* AVR101 "High endurance EEPROM storage"              */
/*                                                     */
/* Filename: High_Endurance_EEPROM.c                   */
/* Date: 2002.08.15                                    */
/* Author: jllassen                                    */
/*                                                     */
/* Modified: 2003.10.06 (raapeland)                    */
/* Modified: 2006.07.18 (raapeland)                    */
/* Modified: 2010.03.16                                */
/*                                                     */
/* Compiler: IAR EWAVR 2.26C, IAR EWAVR 4.12A          */
/*******************************************************/

/*
  This file was heavily modified by Justin Goins in order to optimize EEPROM usage and ensure
compatibility with the XMEGA series.
  This file has also been changed to compile on AVR Studio 4.18.
*/

#include <stdlib.h>
#include "eeprom_leveling.h"
#include "eeprom_driver.h"
#include "extra.h"
#include "visual.h"

void findCurrentEepromAddr( unsigned int *EeBufPtr )
{
  unsigned char temp, prevstatus;
  unsigned int EeBufEnd;

  *EeBufPtr = 0; // Point to the status buffer slot 0 by default

  temp = 0; // start by pointing to status buffer slot 0
  prevstatus = 0; // This will hold whatever the old status value was

  EeBufEnd = *EeBufPtr + EE_STATUS_BUFFER_SIZE;   // The first address outside the buffer

  // Identify the last written element of the status buffer
  do {
    prevstatus = EeReadBuffer( temp );
  *EeBufPtr = temp;
    temp++; // Increment to the next index
    if (temp == EeBufEnd) { // Break if end of buffer, so we don't compare out-of-bounds.
    break;
  }
  } while (EeReadBuffer(temp) == prevstatus + 1);


  *EeBufPtr = *EeBufPtr + EE_PARAM_BUFFER_SIZE; // Point to the last used element of the first
parameter buffer
  // If the status buffer is full of zeros, *EeBufPtr will return as 4 (which is the index of the
first byte of data)
}

// Return the specified byte from EEPROM
char EeReadBuffer( unsigned int address )
{
  return (char)EEPROM_ReadByte( (unsigned char)eeprom_pageinquestion, (unsigned char)(address) );
}

// Specialized function for use with the WHAM internal EEPROM format
char EeReadValue( unsigned int address, uint8_t specified_byte )
{
  return (char)EEPROM_ReadByte( (unsigned char)eeprom_pageinquestion, (unsigned char)(address +
(specified_byte * EE_PARAM_BUFFER_SIZE)) );
}

void EeWriteBuffer( unsigned int *address)
{
  unsigned char EeOldStatusValue;


  EeOldStatusValue = EeReadBuffer( *address - EE_PARAM_BUFFER_SIZE );
  (*address)++;
  if( *address == (EE_START + EE_PARAM_BUFFER_SIZE + EE_STATUS_BUFFER_SIZE) )
  {
    // Wrap around if necessary.
    *address = EE_START + EE_STATUS_BUFFER_SIZE;
  }


  // Update the status buffer
  EEPROM_WriteByte( (unsigned char)eeprom_pageinquestion, (unsigned char)(((*address) -
EE_PARAM_BUFFER_SIZE) & EEPROM_BYTE_ADDRESS_MASK), (EeOldStatusValue + 1));

    // Update the parameters in the EEPROM buffer
    for (int k = 0; k < EE_BYTES_PER_PAGE; k++ ) {
    //EEPROM_LoadByte( (unsigned char)(((*address) + (EE_PARAM_BUFFER_SIZE * k)) &
EEPROM_BYTE_ADDRESS_MASK), k+1 );
    EEPROM_WriteByte( (unsigned char)eeprom_pageinquestion, (unsigned char)(((*address) +
(EE_PARAM_BUFFER_SIZE * k)) & EEPROM_BYTE_ADDRESS_MASK), eeprom_buffer[k]);
  }
}
```

APPENDIX G: EEPROM LEVELING.H HEADER FILE

```
/*
  This file was written by Justin Goins
  Based on the AVR101 Application Note from Atmel.
*/

#include <stdlib.h>
#include <avr/io.h>

/*******************************************************/
/* Define the number of levels in the buffer, */
/* - four levels will guarantee 400k writing of the parameter */
/*******************************************************/
#define EE_PARAM_BUFFER_SIZE 4
#define EE_STATUS_BUFFER_SIZE EE_PARAM_BUFFER_SIZE
#define EE_BYTES_PER_PAGE 7
#define EE_START 0 // Where to start utilizing the EEPROM Page

/*******************************************************/
/* Global variables:                                  */
/* Initialize the parameter buffer pointers to be able to resume at the right location.*/
/*******************************************************/
unsigned int EeBufPtr;
uint8_t eeprom_buffer[ EE_BYTES_PER_PAGE ]; // Using EEPROM wear leveling we can put 7 bytes per 32
byte page
uint8_t eeprom_pageinquestion; // This will be used to keep track of which EEPROM page to write

/*******************************************************/
/* Prototyping of functions used */
/*******************************************************/
void findCurrentEepromAddr( unsigned int *EeBufPtr );
char EeReadValue( unsigned int address, uint8_t specified_byte );
char EeReadBuffer( unsigned int address );
void EeWriteBuffer( unsigned int *address );
```

APPENDIX H: DESIGN PHOTOGRAPHS