FLIGHT TEST DATA SYSTEM FOR STRAIN MEASUREMENT

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Zachary David Wilson

December 2019

© 2019

Zachary David Wilson

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Flight Test Data System for Strain Measurement

AUTHOR: Zachary David Wilson

DATE SUBMITTED: December 2019

COMMITTEE CHAIR: Russell V. Westphal, Ph.D.
Professor of Mechanical Engineering

COMMITTEE MEMBER: William R. Murray, Ph.D.
Professor of Mechanical Engineering

COMMITTEE MEMBER: John R. Ridgely, Ph.D.
Professor of Mechanical Engineering

ABSTRACT

Flight Test Data System for Strain Measurement

Zachary David Wilson

This thesis describes the design and evaluation of two devices to be included in the next generation of the family of devices called the Boundary Layer Data System (BLDS). The first device, called the Quasi-Static Strain Data Acquisition System, is a continuation of the BLDS-M series of devices to be known as the Flight Test Data System (FTDS) that uses a modular approach to acquire non-flow, quasi-static mechanical strain measurements. Various breakout boards and development boards were used to synthesize the device, which were housed by a custom PCB board. The system is controlled by the Simblee™ System on a Chip (SOC), and strain measurements are acquired using the HX711 analog-to-digital converter (ADC), and acceleration measurements are acquired with the ADXL345 accelerometer. The Arduino IDE was used to program and troubleshoot the device. The second device, called the Dynamic Strain Data Acquisition System, is a laboratory proof-of-concept device that evaluates various methods of acquiring dynamic strain measurements that may be used in future FTDS designs. A custom PCB board was designed that houses the microcontroller and the various passive components and ICs used to acquire and store strain measurements. The system is controlled by the Atxmega128A4U microcontroller, and measurements are acquired using the AD7708 external ADC and the on-board ADC of the microcontroller. Atmel Studio™ was used to program the microcontroller in C/C++ and to troubleshoot the device. Both devices were tested extensively under room temperature and low temperature conditions to prove the reliability and survivability of each device. The quasi-static data acquisition system was validated to acquire and store measurements to a microSD card at 10 Hz, with a peak operating current under 60 mA. The dynamic data acquisition system was proven to acquire a thousand measurements at 1 kHz and store the data to a microSD card, with a peak operating current under 60 mA.

Keywords: Strain, Measure, BLDS, Modular, Static, Dynamic, Arduino, C, C++

ACKNOWLEDGMENTS

A special thanks to thank Dr. Russell Westphal for his endless support and guidance for the entirety of the project.

I would also like to thank those on the BLDS project team that I have worked with, specifically: Charlie Refvem, Nathan Hoyt, Neal Sharma, and Robert Taas. I would also like to thank Isabel Jellen, who contributed a great deal to the basis of this work.

Finally, I would like to thank my family who have unconditionally supported me through all of my educational career. I have been extremely fortunate and owe much of my success to the way my parents taught me to be disciplined and driven in everything I do.

TABLE OF CONTENTS

Page

LIST OF TABLES viii
LIST OF FIGURES ix
NOMENCLATURE xiii
CHAPTER
1. INTRODUCTION 1
1.1 Background 1
1.1.1 History of BLDS 1
1.1.2 Next Generation Modular BLDS (BLDS-M) 3
1.1.3 BLDS-M-RAKE 6
1.1.4 Purpose of Work 7
1.2 Requirements 8
2. QUASI-STATIC DATA ACQUISITION SYSTEM 10
2.1 Device Design 10
2.1.1 Component Selection and Specifications 11
2.1.2 System Design 17
2.1.3 Software Design 21
2.2 Device Testing and Validation 28
2.2.1 Current Draw 28
2.2.2 Data Acquisition Testing 29
2.2.3 Low Temperature Testing 31
2.3 Summary – Quasi-Static Data Acquisition System 36
3. DYNAMIC DATA ACQUISITION SYSTEM 38
3.1 Device Design 38
3.1.1 Component Selection and Specifications 39
3.1.2 System Configuration 42
3.1.3 Electronics Design and Analysis 47
3.1.4 Software Design 58
3.2 Device Testing and Validation 67
3.2.1 Current Draw 68
3.2.2 Static Data Acquisition Testing 69
3.2.3 Dynamic Data Acquisition Testing 72

3.2.4 Low Temperature Testing 78
3.3 Summary – Dynamic Data Acquisition System 80
4. CONCLUSIONS AND RECOMMENDATIONS 82
4.1 Conclusions 82
4.1.1 Quasi-Static Data Acquisition System 82
4.1.2 Dynamic Data Acquisition System 83
4.1.3 Software 83
4.2 Recommendations for Future Work 84
4.2.1 Programming IDE 84
4.2.2 Electronics and Circuitry 84
WORKS CITED 86
APPENDICES
A. QUASI-STATIC DATA ACQUISITION SYSTEM SCHEMATIC AND BOARD LAYOUT 89
B. HX711 LOAD CELL AMPLIFIER BREAKOUT SCHEMATIC AND BOARD LAYOUT 92
C. ADXL345 ACCELEROMETER BREAKOUT SCHEMATIC AND BOARD LAYOUT 94
D. HX711 AND ADXL345 ARDUINO DRIVERS 96
E. DYNAMIC DATA ACQUISITION SYSTEM SCHEMATIC AND BOARD LAYOUT 105
F. DERIVATIONS OF FILTER TRANSFER FUNCTIONS, CUTOFF FREQUENCIES, AND MATLAB SIMULATION 110
G. DYNAMIC DATA ACQUISITION SYSTEM C++ DRIVERS 119
H. DYNAMIC DATA ACQUISITION SYSTEM SCHEDULER, BASE TASK, SHARE, AND QUEUE C++ FILES 165
I. DYNAMIC DATA ACQUISITION SYSTEM DERIVED TASK C++ FILES 196
J. DERIVATION OF CANTILEVER BEAM EQUATION OF MOTION AND FIRST NATURAL FREQUENCY 223

LIST OF TABLES

Table Page

1-1. Initial BLDS Version 3 Requirements (Westphal et al. 2008) 2

1-2. Truncated list of Simblee™ features (RF Digital 2017) 5

1-3. Quasi-Static and Dynamic Device Requirements 8

2-1. Position of each device shown in Figure 2-1 11

2-2. Truncated list of HX711 features (AVIA Semiconductor, n.d.) 14

2-3. Timing values for Figure 2-5 (AVIA Semiconductor, n.d.) 15

2-4. Truncated list of ADXL345 features (Analog Devices, 2015) 16

2-5. Current draw and power consumption of device under various operating conditions (5V source) 28

2-6. Sample set of strain and acceleration data acquired at a frequency of 1 Hz 29

2-7. Comparison of the standard deviation in the acceleration data at room temperature and at -36°C 33

2-8. Tabulated strain deviation of device from 22 °C to -36 °C 36

3-1. Truncated list of AD7708 features (Analog Devices, 2001) 42

3-2. Current draw and power consumption of device under various operating conditions 69

3-3. Sample sets of data acquired by both methods 70

3-4. Natural frequencies observed in dynamic testing 78

LIST OF FIGURES

Figure Page

1-1. BLDS with cover removed to show internal circuitry (Westphal et al. 2008) 2

1-2. BLDS-M module functional diagram (Jellen et al. 2019) 4

1-3. BLDS-M Modules with quarter for scale (Jellen et al. 2019) 5

1-4. BLDS-M-RAKE Block Diagram (Hoyt 2019)..7

2-1. Top view of quasi-static device displaying each component 10

2-2. Simblee™ 29-GPIO breakout board pinout (RF Digital 2015).....................................12

2-3. RFduino™ MicroSD shield pinout illustration (RF Digital, 2013)................................13

2-4. HX711 block diagram and typical application (AVIA Semiconductor, n.d.) 14

2-5. Data output, input, and gain control timing (AVIA Semiconductor, n.d.) 15

2-6. ADXL345 functional block diagram (Analog Devices, 2015)......................................16

2-7. Quasi-Static data acquisition system device hardware diagram 18

2-8. Top and bottom views of the HX711 breakout provided by SparkFun (SparkFun, 2013) 19

2-9. Wheatstone bridge quarter-bridge configuration 20

2-10. Top and bottom views of the ADXL345 breakout provided by SparkFun (SparkFun, 2013) 21

2-11. Task diagram for quasi-static data acquisition system 24

2-12. Strain Reader (left) and Acceleration Reader (right) task state transition diagrams 25

2-13. MicroSD Driver task state transition diagram 26

2-14. User Interface task state transition diagram...27

2-15. Strain gauge testing setup. Overall testing setup (left), beam and dangling mass (middle), closeup of strain gauge location (right) 30

2-16. Cantilever beam schematic 30

2-17. Comparison of HX711 readings to theoretical strain values 31

2-18. Vacuum chamber test results for variation in acceleration as the chamber temperature decreases 32

2-19. Vacuum chamber test results for absolute variation in mechanical strain as the chamber temperature decreases 33

2-20. Results of freezer testing. Left: Strain readings with aluminum beam and strain gauge at -36 °C and electronics at 22 °C. Right: Strain readings with electronics at -36 °C and aluminum beam and strain gauge at 22 °C 35

3-1. Top view of assembled dynamic data acquisition system 39

3-2. Pinout/block diagram of the Atxmega architecture (Atmel, 2014) 40

3-3. AD7708/AD7718 pinout/block diagram (Analog Devices, 2001) 41

3-4. Dynamic data acquisition system device hardware diagram 43

3-5. Wheatstone half-bridge (left) and full-bridge (right) configurations 46

3-6. Dynamic data acquisition system PCB board outline 47

3-7. Power regulation and microcontroller schematic 48

3-8. FT232R and microUSB schematic 49

3-9. MicroSD socket schematic 50

3-10. AD7708 and quarter, half, and full-bridge schematic 51

3-11. AD780R voltage reference schematic 52

3-12. High-level ADC data acquisition block diagram 52

3-13. First-order passive low-pass filter schematic 53

3-14. Bode Diagram for first-order low-pass filter 54

3-15. First-order active low-pass filter schematic 54

3-16. Second-order active low-pass filter schematic 55

3-17. Bode Diagram for second-order low-pass filter 56

3-18. Differential amplifier + offset circuit schematic 56

3-19. Voltage divider circuit for providing a reference voltage of $Vcc/2$ 57

3-20. Task diagram for dynamic data acquisition system 63

3-21. AD7708 Driver task state transition diagrams ... 64

3-22. ADC Interface task state transition diagrams ... 64

3-23. MicroSD Driver task state transition diagram ... 65

3-24. User Interface task state transition diagram ... 66

3-25. Quarter-bridge (left), half-bridge (middle), and full-bridge (right) beam configurations ... 68

3-26. Calibration of the AD7708 (16-bit) ... 71

3-27. Comparison of on-board ADC (12-bit) readings to theoretical strain values ... 72

3-28. Dynamic cantilever beam model schematic ... 73

3-29. Assumed mode shapes of cantilever beam model for (top) first mode and (bottom) second mode ... 73

3-30. AD7708 (16-bit) dynamic beam testing results for the quarter-bridge configuration ... 75

3-31. AD7708 (16-bit) dynamic beam testing results for the half-bridge configuration ... 75

3-32. AD7708 (16-bit) dynamic beam testing results for the full-bridge configuration ... 76

3-33. On-board ADC (12-bit) dynamic testing results with passive 1st-order filtering ... 77

3-34. On-board ADC (12-bit) dynamic testing with active 1st-order filtering ... 77

3-35. On-board ADC (12-bit) dynamic beam testing with active 2nd-order filtering ... 77

3-36. Full-bridge low temperature test results acquired by the AD7708 (16-bit) ... 79

3-37. On-board ADC (12-bit) half-bridge low temperature test results ... 79

A-1. Quasi-static Data Acquisition System PCB top layer layout ... 90

A-2. Quasi-static Data Acquisition System PCB bottom layer layout ... 90

A-3. Quasi-static Data Acquisition System PCB top render ... 91

A-4. Quasi-static Data Acquisition System PCB bottom render ... 91

B-1. HX711 breakout board top layer layout ... 93

B-2. HX711 breakout board bottom layer layout ... 93

C-1. ADXL345 breakout board top layer layout ... 95

C-2. ADXL345 breakout board bottom layer layout 95

E-1. Dynamic Data Acquisition System PCB top layer layout 108

E-2. Dynamic Data Acquisition System PCB bottom layer layout 108

E-3. Dynamic Data Acquisition System PCB top render 109

E-4. Dynamic Data Acquisition System PCB bottom render 109

F-1. First-order active low-pass filter schematic 110

F-2. First-order low-pass filter linear graph (left) and normal tree (right) 111

F-3. Second-order active low-pass filter schematic 113

F-4. Second-order low-pass filter linear graph (left) and normal tree (right) 113

F-5. Simulink^{TM} block diagram of first-order filter simulation 115

F-6. Simulink^{TM} block diagram of second-order filter simulation 115

F-7. Bode Diagram for first-order low-pass filter 117

F-8. First-order low-pass filter response with noisy input 117

F-9. Bode Diagram for second-order low-pass filter 118

F-10. Second-order low-pass filter response with noisy input 118

J-1. Dynamic cantilever beam model schematic 223

J-2. Assumed mode shapes of cantilever beam model for (top) first mode and (bottom) second mode 224

NOMENCLATURE

A	=	cross-sectional area, m^2
b	=	width, m
c	=	mechanical damping, N-s/m
C	=	capacitance, F
E	=	elastic modulus, Pa
f_c	=	cutoff frequency, Hz
F	=	force, N
GF	=	strain gauge factor
h	=	height, m
i	=	electric current, A
I	=	second moment of inertia, m^4
L	=	length, m
L_{SG}	=	strain gauge location, m
M	=	moment, N-m
q	=	solution to equation of motion
R	=	electrical resistance, Ω
R_{ref}	=	nominal electrical resistance, Ω
T	=	temperature, °C
T_{ref}	=	reference temperature, °C
V	=	voltage, V
$V_{r_{strained}}$	=	voltage across a Wheatstone bridge with loaded strain gauge, V
$V_{r_{unstrained}}$	=	voltage across a Wheatstone bridge with unloaded strain gauge, V
α_T	=	temperature coefficient of resistance, Ω/Ω°C
α_l	=	thermal expansion coefficient, m/m°C
ε	=	mechanical strain
σ	=	mechanical stress, Pa
ρ	=	density, kg/m^3
υ	=	transverse deflection, m
ψ	=	assumed mode shape
ω_n	=	natural frequency, Hz

1. INTRODUCTION

This paper details the development of two prototype devices to support the development of the family of devices collectively called the Boundary Layer Data System (BLDS) (Westphal et al. 2006). A BLDS is a small, autonomous, self-contained instrument capable of acquiring measurements of the flow near the surface of an aircraft during flight. These new devices are the first BLDS devices to incorporate the ability to acquire timestamped mechanical strain data and save the data to an on-board micro-SD card. The BLDS devices were developed especially for boundary layer data acquisition with configurations available for various modes of operation (Westphal et al. 2008). The next generation of BLDS devices, known as BLDS-M, took a modular approach with separate devices responsible for various operations. The motivation and requirements for the new prototype BLDS devices will be discussed after presenting an overview of the development of the BLDS device family.

1.1 Background

1.1.1 History of BLDS

A BLDS device is an instrument used to measure various parameters in the boundary layer on the outside surface of an aircraft during flight. The basic requirements for the BLDS family are that each device must weigh less than a pound, operate autonomously and independent of any aircraft systems, and be mountable to the outside of an aircraft using adhesives (Westphal et al. 2006). The first generation of these devices flew in 2005. The base design of the BLDS system was solidified and tested in 2008 with the third generation of these devices. The third generation (Version 3) boasted improvements such as a dual microcontroller system, longer test times at low temperatures with better power management, and the addition of satellite devices for measuring skin friction as well as boundary layer profile measurements (Westphal et al. 2008). The

requirements of the base design for these devices are shown in Table 1-1 and a photo of the base device with the cover removed is shown in Figure 1-1.

Table 1-1: Initial BLDS Version 3 Requirements (Westphal et al. 2008).

Requirement	Quantity or Description
Operating Altitude	30,000+ ft (-55 °C)
Attachment design load criteria (2/3 in^2 tape/lb design load gives safety factor=3)	3 g dynamic loading, ±5 ° yaw, Vertical or horizontal surface
Dynamic pressure (local-at instrument mount location)	Up to 150 psf
Probes/sensors	Preston tube (fixed); freestream total pressure (fixed); temperature (fixed) total pressure (moving), rotatable single-hole probe (moving)
Battery life	> 2 hrs.
Weight	< 1 lb
Operation	Autonomous; no pilot involvement

Figure 1-1: BLDS with cover removed to show internal circuitry (Westphal et al. 2008).

The design of Version 3 of the BLDS device has served as the basis for all BLDS devices up until 2017. The device was flown on missions between 2008 and 2018, and developments were

made during that time to improve the functionality of the base device. Functionality was added to the device by implementing a Conrad probe, which added the ability to read and record information about the flow direction in addition to magnitude (Ulk 2010). To acquire unsteady boundary layer data, solutions such as the addition of a Kulite sensor and microphones were explored to capture time-varying pressure data (Karasawa 2011), (Lillywhite 2013). The ability to measure skin friction was also added by incorporating devices such as a Preston tube and a modified Stanton gauge (Kinkade 2014). Finally, Version 3 was altered to accommodate a 16-tube rake to instantaneously capture the entire boundary layer. Key specifications for the Version 3 system include 128KB of volatile random-access memory (RAM) program memory, 2 MB of non-volatile serial Flash memory, 11 analog inputs of 0-5 V range and 12-bit resolution, and a typical sample rate of 100 Hz (Westphal et al. 2017).

1.1.2 Next Generation Modular BLDS (BLDS-M)

Work on the next version of the BLDS, called the BLDS-M, began in 2017 as a proof-of-concept which incorporated newer components with lower power consumption and wireless communication. Rather than one device with multiple configurations, the next generation of BLDS devices took a modular approach which decreased power consumption and increased the functionality of the device family. The modular approach allows for lower power consumption and incorporation of only the necessary capabilities for a particular mission. Figure 1-2 displays the functional diagram for the first three modules that were designed and tested.

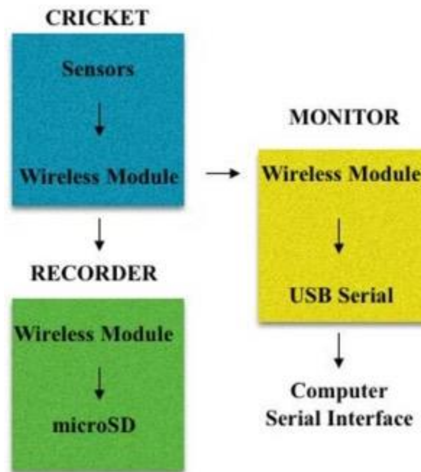


Figure 1-2: BLDS-M module functional diagram (Jellen et al. 2019).

The first three modules were named "Cricket", "Monitor", and "Recorder". The Cricket module houses and controls the sensors used for data acquisition and communicates to the Monitor and Recorder modules that provide live data monitoring and recording, respectively (Jellen et al. 2019). These devices utilize the Simblee™ development board, a microcontroller system with Bluetooth Low Energy (BLE) wireless capabilities. Simblee™ is an Internet of Things (IOT) System on a Chip (SOC) device that supports the Arduino IDE and includes an on-board radio antenna. The three modules were assembled using various breakout boards provided by Simblee™. The Cricket and Recorder modules incorporate the 7 GPIO Breakout and the 2x AAA shield provided by RF Digital™. The Recorder module includes a microSD shield for recording data. The Monitor module includes only the 29 GPIO Breakout and USB Shield for monitoring data over a computer serial port. Figure 1-3 shows an image of the three modules with a quarter for scale.

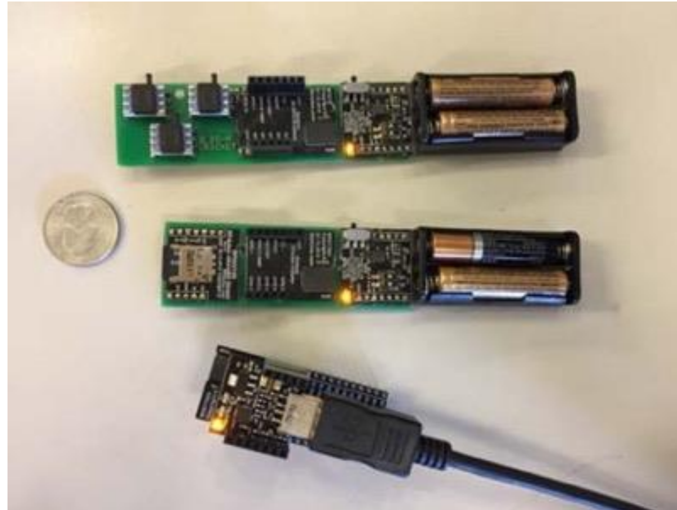


Figure 1-3: BLDS-M Modules with quarter for scale (Jellen et al. 2019).

The Simblee™ SOC boasts many characteristics that increase the functionality and effectiveness of the BLDS family, displayed in Table 1-2. The most notable features are the 3.3 V compatibility and 12mA current draw, the package dimensions, and the integrated antenna and shield which increases the efficiency, integrability, and functionality of the device, respectively.

Table 1-2: Truncated list of Simblee™ features (RF Digital 2017).

Features
7 x 10 x 2.2 mm package dimension
29 General Purpose I/O
12 mA typical operational current draw
600 uA sleep current draw
ARM Cortex M0 processor
AES encryption engine
1.8-3.6 V operating voltage
16 MHz clock (32 KHz precision clock)
Integrated Antenna and shield
-93 dBm receiver sensitivity
-55 dBm to +4 dBm TX power

The Simblee™ SOC includes most of the functionality of an Arduino with the addition of SimbleeCOM™, a BLE communication protocol which utilizes the on-board antenna. The only external component required for the chip to function reliably is a bypass capacitor on the power

input. These features make the Simblee^{TM} easy to integrate and effective for projects where wireless communication between devices is desirable. While none of the BLDS-M family of devices have been involved in a flight test, they have been proven to survive and function in a thermal vacuum chamber (Jellen et al. 2019). Unfortunately, the Simblee^{TM} and related products recently went end-of-life. However, the effort put forth by the BLDS project team into the BLDS-M family of devices successfully demonstrated that a clean-sheet design for the next generation of BLDS instruments was feasible and would provide many advantages. Mainly, the modular approach that is made possible by the wireless communication capabilities supported on modern SOC systems.

1.1.3 BLDS-M-RAKE

The newest addition to the family of BLDS devices that utilizes the Simblee^{TM} SOC is the BLDS-M-RAKE module. This device is a continuation of the BLDS-M series of devices with updated electronics and sensors intended to be mounted to an array of probes, or rake (Hoyt 2019). A printed circuit board (PCB) was designed to house the array of Honeywell pressure sensors, Simblee^{TM} SOC, microUSB, microSD socket, DC-DC boost regulator, and two AAA battery cells. The electronics followed the original three BLDS-M modules in that it operates on 3.3 V and is capable of communicating wirelessly with other modules. The Honeywell pressure sensors could be sampled and recorded to a microSD card at a frequency at 100 Hz with a peak current draw of 30 mA and a sleeping current draw of less than 3 mA (Hoyt 2019). A block diagram detailing the device design is shown in Figure 1-4.

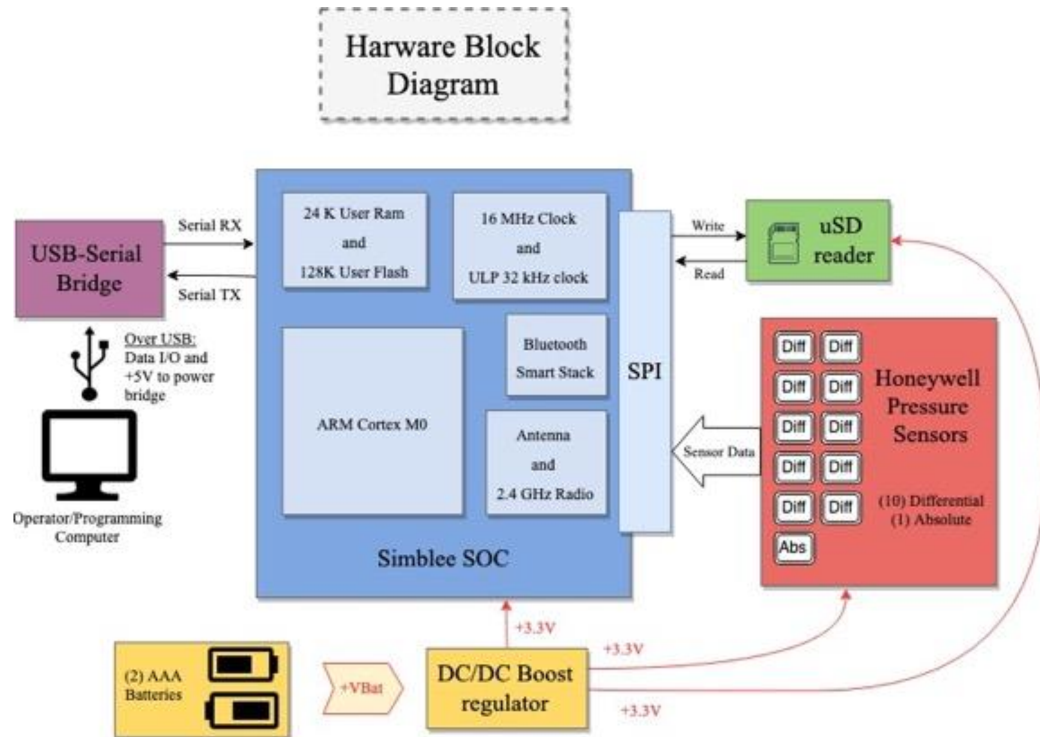


Figure 1-4: BLDS-M-RAKE Block Diagram (Hoyt 2019).

In addition to the design presented above, a flexible manifold to connect a reference pressure to an array of differential pressure sensors was designed and validated. This new manifold was designed to be cast from silicone, which improved on the old machined metal manifold design by providing an easier and much cheaper means for replacing damaged sensors or removing sensors for inspection (Hoyt 2019).

1.1.4 Purpose of Work

Previous BLDS devices were designed primarily for acquiring information about the boundary layer on the outer surface of an aircraft. Acquiring non-flow data that details the mechanical performance of the aircraft would be useful information as well. The motivation for the ability to acquire non-flow data such as in-flight mechanical strain data was the desire for empirical strain measurements that are present on the aircraft during flight. In addition, strain data acquired at a high frequency would detail the frequency and magnitudes of dynamic loading on the aircraft during flight. This project focuses primarily on acquiring mechanical strain measurements on the

outer surface of an aircraft. The design, testing, and evaluation of various methods of acquiring strain measurements is split into two parts: acquiring quasi-static strain measurements at low frequencies and acquiring dynamic strain measurements at high frequencies.

1.2 Requirements

The requirements for the mechanical strain data acquisition devices follow the basic design requirements laid out in the design of all previous BLDS-M devices. Because these devices are the first BLDS modules for acquiring strain data, they were designed as proof-of-concept devices. Thus, the mechanical design requirements typical of BLDS devices, excluding size and weight, were considered lower priority than the design requirements of the electronics and circuitry. In addition, wireless communication between devices was considered lower priority, as that ability was already demonstrated with the Cricket, Monitor, and Recorder modules. The requirements for each device are shown in Table 1-3. Strain gauges, a passive sensor whose resistance varies linearly with applied strain, will be used by both devices to acquire mechanical strain data.

Table 1-3: Quasi-Static and Dynamic Device Requirements.

<table>
<tr><th>Requirement</th><th>Quasi-Static</th><th>Dynamic</th></tr>
<tr><td>Operating Temperature</td><td colspan="2">> -40 °C</td></tr>
<tr><td>Sensors/Resolution</td><td>Strain Gauges, 24-bits
Accelerometer, 13-bits</td><td>Strain Gauges, 12-bits,
16-bits</td></tr>
<tr><td>Acquisition Frequency</td><td>10 Hz</td><td>> 1 kHz</td></tr>
<tr><td>Controller</td><td>Simblee™ SOC</td><td>Atxmega128A4U</td></tr>
<tr><td>Battery life</td><td colspan="2">> 2hrs.</td></tr>
<tr><td>Power Consumption</td><td colspan="2">< 60 mA (3.3 V)</td></tr>
<tr><td>Weight</td><td colspan="2">< 1lb</td></tr>
<tr><td>Size</td><td colspan="2">< 6 in x 6 in x 1 in</td></tr>
<tr><td>Operation</td><td colspan="2">Autonomous; no pilot involvement</td></tr>
</table>

Each device has the same requirements for the operating temperature, size, weight, and operation. To acquire data, the quasi-static strain device uses an external integrated circuit (IC) that controls an analog-to-digital converter (ADC) with a resolution of 24-bits and a selectable

acquisition frequency of 10 Hz or 80 Hz. The device also incorporates an external IC with an accelerometer for acquiring acceleration data. The quasi-static strain device is controlled by the Simblee^{TM} SOC for communicating wirelessly. The dynamic strain device has two interfaces for acquiring data at frequencies exceeding 1 kHz. One interface is an external IC that controls an ADC with a resolution of 16-bits and a maximum acquisition frequency of 1.3 kHz. The other interface is the on-board ADC interface of the Atxmega128A4U with a resolution of 12-bits and a maximum acquisition frequency of 2 MHz. The Atxmega128A4U is a low-power, peripheral-rich, 16-bit microcontroller that achieves throughputs approaching one million instructions per second per megahertz (Atmel, 2014).

The following chapters present the design and testing of each device. In Chapter 2, the design of the quasi-static data acquisition device is presented followed by a summary of the testing done to validate the requirements discussed in Table 1-3. Chapter 3, similar to Chapter 2, presents the design of the dynamic data acquisition device and then summarize the device testing and validation. Chapter 4 presents the conclusions found during the project and the recommendations for future work on BLDS instrumentation for acquiring strain measurements.

2. QUASI-STATIC DATA ACQUISITION SYSTEM

This chapter describes the design and development of the newest device belonging to the BLDS-M family of devices that acquires quasi-static mechanical strain and acceleration data. The device is intended to acquire strain and acceleration data on the outside surface of an aircraft during flight and save the timestamped data to a microSD card. The device is small, lightweight, and autonomous with an optional computer interface for setting device parameters and real-time data monitoring as an additional feature.

2.1 Device Design

Five main components encompass this prototype device. The components include the HX711 Load Cell Amplifier, the ADXL345 3-Axis Accelerometer, an RFduino™ microSD shield, an RFduino™ 2x AAA battery shield, and a Simblee™ SOC 29 GPIO breakout. The package dimensions of the device are approximately 2 in x 5 in x 1 in with a weight of approximately 0.15 lb. An image of the device can be seen in Figure 2-1 and the position of each component in the figure is detailed in Table 2-1.

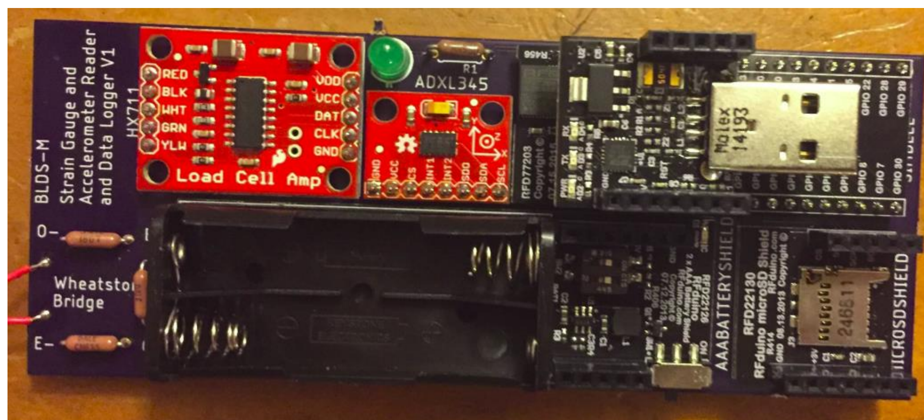


Figure 2-1: Top view of quasi-static device displaying each component.

Table 2-1: Position of each device shown in Figure 2-1.

Position	Component
Top Left	HX711 Load Cell Amplifier breakout
Top Center	ADXL345 3-Axis Accelerometer breakout
Top Right	Simblee™ SOC 29 GPIO breakout and RFduino™ microUSB shield
Bottom Left	Bridge resistors for strain gauge
Bottom Center	RFduino™ AAA Battery Shield
Bottom Right	RFduino™ MicroSD Shield

The selection and details of each component, excluding the Simblee™ SOC, are discussed below. Key details and design considerations of the Simblee™ SOC are provided in Section 1.1.2. The board schematic and layout were designed in Autodesk Eagle™ and are detailed in Appendix A.

2.1.1 Component Selection and Specifications

Breakout and development boards were used to develop the prototype device. The use of breakout/development boards allowed for proof-of-concept breadboarding and fast prototyping.

2.1.1.1 RFduino™ Breakout Boards

RFduino™ provides a development kit with a number of breakout boards to jumpstart any project involving RFduino™ products. The 29 GPIO breakout houses the Simblee™ SOC and provides access to 29 GPIO pins along with a 3.3 V input to power the SOC and a reset pin for resetting the chip (RF Digital 2015). The only other component mounted to the breakout board is a decoupling capacitor on the 3.3 V input. An illustration showing the pinout of the 29 GPIO breakout can be seen in Figure 2-2. The pinouts are made available with 0.1 in pitch male headers on the bottom side of the shield, as well as 0.1 in pitch female headers on the top side of the board for the ground, 3.3 V, reset, and GPIO pins 1-6.

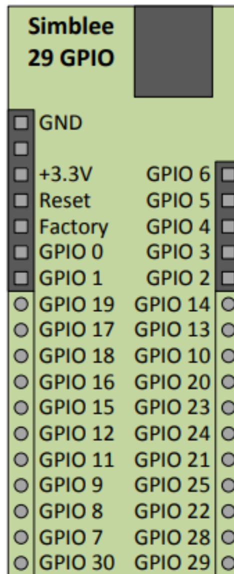


Figure 2-2: Simblee™ 29-GPIO breakout board pinout (RF Digital 2015).

The RFduino™ AAA Battery Shield provides a battery holder for two AAA batteries to power the device. To step up the 3.0 V provided by the batteries, a DC-DC converter is provided on the shield to boost the voltage to 3.3 V. The DC-DC converter maintains the 3.3 V level when the voltage of the AAA batteries begins to decrease due to power being consumed, which improves the battery life and reliability of the device. Pinouts for the 3.3 V rail and the device ground are available with 0.1 in pitch male and female headers on the bottom side and top side of the board, respectively.

To store the data acquired by the device, the RFduino™ MicroSD shield provides a microSD socket that provides pinouts for power, ground, and digital communication. The shield consists of a microSD socket and a decoupling capacitor on the 3.3 V rail. The shield makes it trivial to connect the digital communication pins to a 4-wire Serial Peripheral Interface (SPI) on a microcontroller. SPI is a commonly used communication protocol for exchanging two bytes at a time between two ICs. Pinouts for the 3.3 V rail, ground, and the communication pins are available with 0.1 in pitch male and female headers on the bottom side and top side of the board, respectively. An image displaying the pinout of the microSD shield is shown in Figure 2-3.

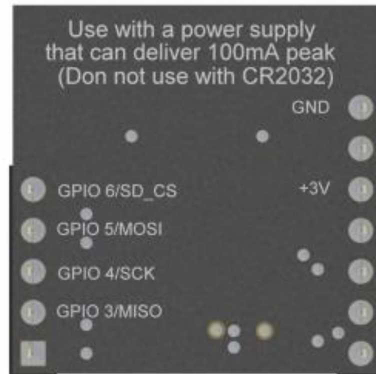


Figure 2-3: RFduino™ MicroSD shield pinout illustration (RF Digital, 2013).

2.1.1.2 HX711 Load Cell Amplifier

The HX711 is a precision 24-bit ADC designed for weight scales and industrial applications (AVIA Semiconductor, n.d.). While not intended for experimental flight-testing devices, it was chosen based on its 24-bit resolution, selectable conversion frequency of 10 Hz or 80 Hz, ease of integration, and simple digital interface and control. A truncated list of HX711 features are listed below in Table 2-2. The IC has an on-board 24-bit ADC for performing conversions for two differential input channels whose signal is amplified by an on-board programmable-gain amplifier (PGA). The ADC is driven by an internal oscillator and controlled by a digital serial interface that can be connected to any GPIO pins on a microcontroller. An on-board analog supply regulator and a bandgap reference set by the voltage level on a device pin abstracts the selection of a voltage reference. The HX711 allows for the use of any desired bridge configuration since there are no bridge elements included on the breakout board. A block diagram detailing each component of the device as well as a typical application is shown in Figure 2-4.

Table 2-2: Truncated list of HX711 features (AVIA Semiconductor, n.d.).

Features
6 x 9.9 x 1.4 mm package dimension
Two selectable differential input channels
Selectable gain of 32, 64, 128
On-chip oscillator
Simple digital control and serial interface
Selectable 10 Hz or 80 Hz conversion frequency
Operating current < 1.5 mA, power down < 1 uA
Operation supply voltage range: 2.6 ~ 5.5V
Operation temperature range: -40 ~ +85°C

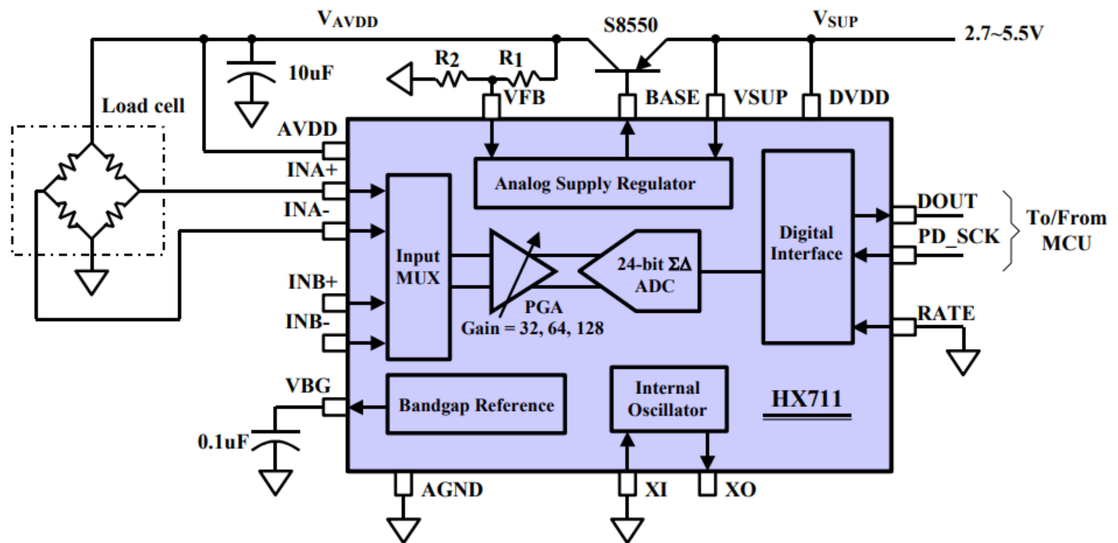


Figure 2-4: HX711 block diagram and typical application (AVIA Semiconductor, n.d.).

The device is controlled using two GPIO pins on a microcontroller. The PD_SCK pin is an input pin that is supplied a clock signal to control the timing of the communication. The DOUT pin is an output pin that sends data one bit at a time based on the timing determined by the clock frequency on the PD_SCK pin. Data from the HX711 is sent in twos complement form with the most-significant-bit (MSB) sent first, whose magnitude is determined by the analog input and the selected gain. The timing for the data output, input, and gain control is described in Figure 2-5, and the specific timing values are displayed in Table 2-3. To power down the device, the PD_SCK pin is raised for a period greater than 60 μs, and the pin is brought low to power the device back on.

The input gain is selected for the next conversion period by the number of the input PD_SCK pulses after the 24th bit is transferred.

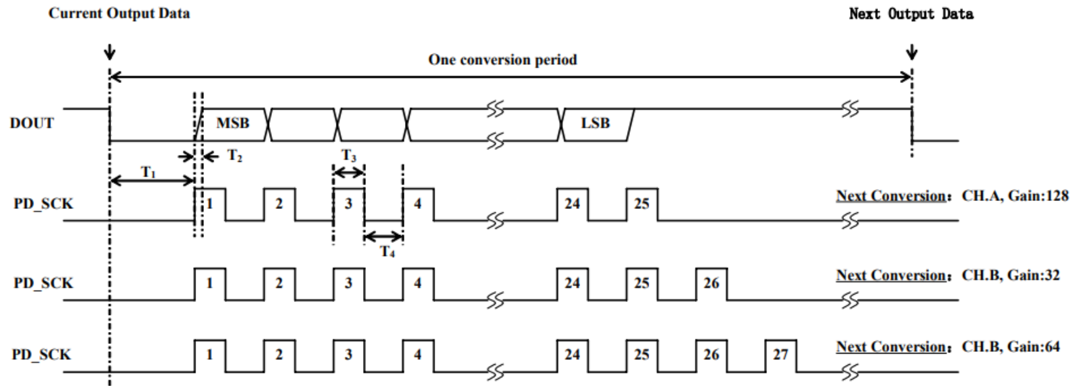


Figure 2-5: Data output, input, and gain control timing (AVIA Semiconductor, n.d.).

Table 2-3: Timing values for Figure 2-5 (AVIA Semiconductor, n.d.).

Symbol	Note	MIN	TYP	MAX	Unit
T_1	DOUT falling edge to PD_SCK rising edge	0.1			µs
T_2	PD_SCK rising edge to DOUT data ready			0.1	µs
T_3	PD_SCK high time	0.2	1	50	µs
T_4	PD_SCK low time	0.2	1		µs

Due to the simplicity of the digital interface, the device has some limitations. Control of the device through the digital interface is limited to initializing and placing the device in low power mode and transferring the most recent analog conversion. The conversion frequency is selected based on the voltage level of the RATE pin, and the PGA gain is selectable based on the input channel and the number of clock ticks, meaning that a dummy number must be transferred before selecting the gain. Gains of 128, 64, and 32 are selectable, and further limited by the input channel. Calibration of the HX711 must be performed in software.

2.1.1.3 ADXL345 3-Axis Accelerometer

The ADXL345 is an ultra-low power 3-axis accelerometer with 13-bit resolution and a selectable range up to ±16 *g* intended for a wide variety of applications (Analog Devices, 2015). It

is capable of measuring static acceleration due to gravity and dynamic acceleration resulting from vibrations or impulses in three directions, as well as detecting specific motions like single or double taps and free-falling. The output data rate of the device is selectable with a maximum rate of 3200 Hz and a resolution of 10 – 13 bits depending on the selected range. The internal registers used for controlling the device may be accessed through an SPI interface or a Two-Wire Interface (I^2C). The device features a 32-level first in, first out (FIFO) buffer to store data to minimize the host microcontroller processing activity and overall system power consumption (Analog Devices, 2015). A truncated list of ADXL345 features are listed below in Table 2-4, and the device functional block diagram is shown in Figure 2-6.

Table 2-4: Truncated list of ADXL345 features (Analog Devices, 2015).

Features
3 x 5 x 1 mm package dimensions
Controllable via SPI or I^2C
Selectable output frequency up to 3200 Hz
Selectable range: ±2 *g*, ±4 *g*, ±8 *g*, ±16 *g*,
Up to 13-bit resolution
Supply voltage range: 2.0 V to 3.6 V
Operating current < 140 µA, standby mode < 0.1 uA
Operation temperature range: -40 ~ +85 °C

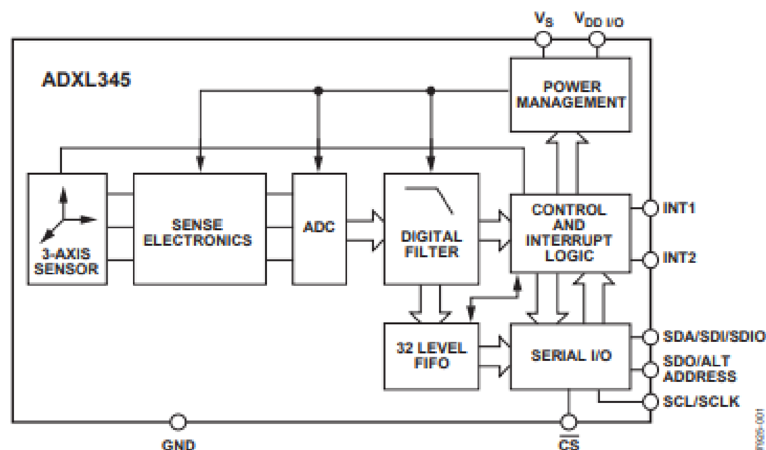


Figure 2-6: ADXL345 functional block diagram (Analog Devices, 2015).

The ADXL345 is controlled by writing to the internal registers of the device using standard communication protocols, either SPI or I^2C. In both cases, the device operates as a slave, and data is transferred in a 16-bit number in twos complement form. Calibration of the ADXL345 must be performed in software.

2.1.2 System Design

Using the design requirements and the selected components discussed in previous sections, a high-level functional block diagram of the quasi-static device was created, shown in Figure 2-7. The Simblee™ SOC runs software to control and communicate each component of the device. To acquire mechanical strain data, two GPIO pins on the Simblee™ are used to communicate with the HX711 through its serial interface. The ADXL345 acquires acceleration data and is controlled using the on-board SPI interface of the Simblee™. Data from each device is buffered, timestamped, and saved to a microSD card using the same SPI interface. Each component is powered by the RFduino™ Battery Shield which provides a step-up DC-DC converter to supply the device with a constant 3.3 V source. The RFduino™ USB Shield is only used when programming/debugging the device and for real-time data monitoring through a USB port on an external computer. The USB shield supplies 3.3 V as well, but both the USB shield and the Battery shield have reverse-voltage protection.

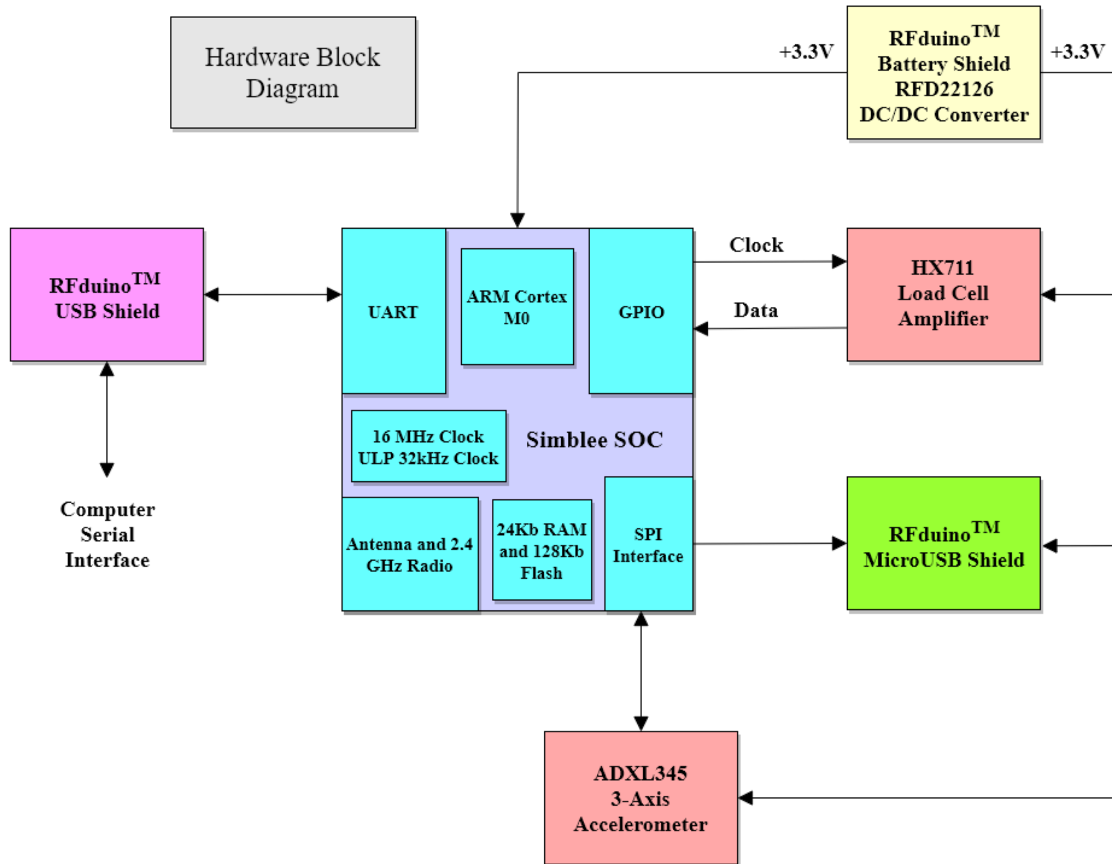


Figure 2-7: Quasi-Static data acquisition system device hardware diagram.

The Simblee™ SOC 29 GPIO breakout provides more than enough GPIO pins to control the device. The Simblee™ is programmed and data is monitored through the on-board Universal Synchronous/Asynchronous Receiver/Transmitter (USART) port, requiring two pins for the transmit (TX) and receive (RX) lines. The HX711 requires two GPIO pins to connect to the PD_SCK and DOUT lines to control the device and transfer data. The ADXL345 and the microUSB shield are both connected to the serial clock (SCK), master-in-slave-out (MISO), and master-out-slave-in (MOSI) SPI pins on the Simblee™. Both of these components require an additional chip-select (CS) pin. Finally, a status LED was added to the design driven by a GPIO pin for debugging purposes only, bringing the grand total use of 10 out of 29 GPIO pins on the Simblee™ SOC.

To speed up the development process, breakout boards for the HX711 and ADXL345 were purchased from SparkFun. SparkFun is an open-source hardware retailer that provides breakout boards for many common products as well as Arduino code for jumpstarting projects (Sparkfun, n.d.). The breakout boards greatly decreased the time required to successfully incorporate the HX711 and ADXL345 components.

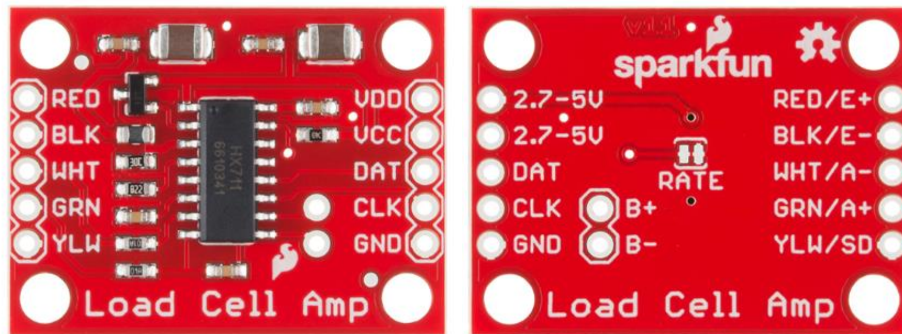


Figure 2-8: Top and bottom views of the HX711 breakout provided by SparkFun (SparkFun, 2013).

The HX711 breakout provides pinouts for power, ground, data, and clock lines. The breakout is designed very similarly to the application/block diagram suggested in the HX711 datasheet. The only difference is the addition of an inductor and capacitor that acts as a second order low pass filter upstream of the excitation pinout (E+). The schematic and PCB layout for the breakout is provided in Appendix B. A footprint for a jumper resistor is featured on the bottom side of the breakout for changing the conversion frequency of the device. By default, the breakout is configured for a frequency of 10 Hz. Soldering a jumper resistor to the footprint will short the RATE pin to ground and thus change the conversion frequency to 80 Hz.

The HX711 breakout is easily utilized for ADC conversions across the output of a Wheatstone bridge. A Wheatstone bridge is a simple circuit consisting of two voltage dividers in parallel. This circuit is commonly used with strain gauges to acquire data. An illustration of the Wheatstone bridge used in this design, commonly referred to as a quarter-bridge, is shown in Figure

2-9. The bridge consists of three 120 Ω precision resistors with a tolerance of 0.1% (Vishay, 2019) and a 120 Ω strain gauge between terminals A+ and E-.

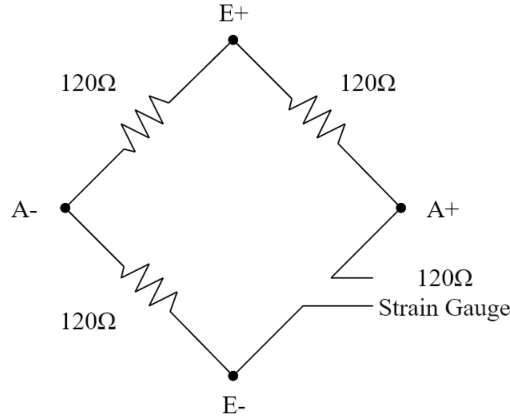


Figure 2-9: Wheatstone bridge quarter-bridge configuration.

When the gauge is unstrained, the bridge is balanced and the voltage across terminals A+ and A- is nominally zero. The differential voltage becomes nonzero when the gauge is strained and the resistance changes. Equations 2-1 and 2-2 can be used to convert the differential voltage across terminals A+ and A-, denoted as V_r, to strain (Omega Engineering, 1999).

$$V_{r_{net}} = V_{r_{strained}} - V_{r_{unstrained}} \quad (2\text{-}1)$$

$$\varepsilon = \frac{-4\, V_{r_{net}}}{GF\,(1 + 2\, V_{r_{net}})} \quad (2\text{-}2)$$

The breakout also features a bipolar-junction-transistor (BJT) whose state is controlled entirely by the HX711. When performing a conversion, the BJT provides the excitation voltage that drives electrical current to flow through the Wheatstone bridge and cause the differential voltage to be read by the HX711. This feature decreases the overall power consumed by the device, as the Wheatstone bridge configured with 120 Ω resistors has a steady current draw of 27.5 mA when excited by V_{E+} = 3.3 V.

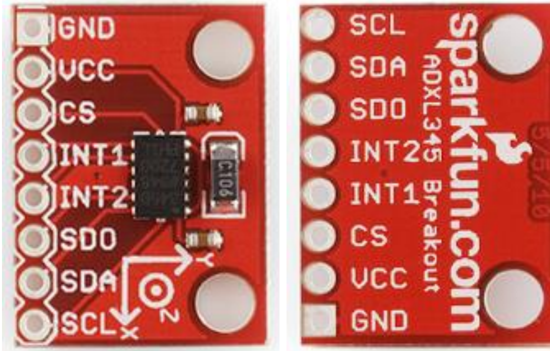


Figure 2-10: Top and bottom views of the ADXL345 breakout provided by SparkFun (SparkFun, 2013).

The ADXL345 breakout provides pinouts for power, ground, SPI/I^2C pins, and the two configurable interrupt pins which are unused in this device. The schematic and PCB layout for the breakout is provided in Appendix C. The only components included in this breakout, excluding the ADXL345 itself, are three decoupling capacitors for stabilizing the VCC rail.

2.1.3 Software Design

Software was written for the Simblee™ SOC using the Arduino IDE to control the various components of the device. The program running on the Simblee™ needed to accomplish the following: configure the internal clock to keep correct time for timestamping data, store the user-set parameters, acquire and record data to the microSD card using the selected components, convey the operational status of the device to the user through the serial interface, and report debugging information when prompted. To improve readability and reduce troubleshooting difficulty, the code was split into multiple files based on functionality. The concept of classes was heavily utilized to program the Simblee™. Driver files were written that contain functions to control various components of the device. In addition, the concepts of multi-tasking and finite-state machines were used to further simplify and improve functionality. The drivers, multi-tasking scheme, and tasks written for the embedded software are discussed in the following sections.

2.1.3.1 Drivers

An advantage of using a microcontroller that accepts the Arduino IDE is that Arduino provides many built-in libraries that are efficient and have been tested extensively. Arduino provides libraries for SD card, serial port, and SPI drivers that were utilized in this design (Arduino, 2019). The Serial library was used to communicate over the serial port, and the SPI library was used to write a driver for communicating with the ADXL345 over the SPI interface. The SD card library, which also utilizes the SPI library, was used for writing data to the microSD card. Two custom libraries were written for controlling the ADXL345 and the HX711. These libraries, or drivers coded in the Arduino IDE, add a level of abstraction between the device software and firmware to make the code easier to read and troubleshoot. The drivers written for the device are provided in Appendix D.

HX711 Driver:

The HX711 driver provides a class that, when instantiated, creates an object that contains variables and functions for controlling an HX711 IC on specified GPIO pins. Using the timing diagram in Figure 2-5, functions were written to acquire data from the IC. Data from the HX711 is output in 24-bit, twos complement form, and digitally acquired through the device serial interface. The driver constructs and returns the data as an unsigned 32-bit number. Other functions for controlling the device, such as *enable ()*, *disable ()*, and *getStatus ()*, are available for error checking and conserving power.

ADXL345 Driver:

The ADXL345 driver, similar to the HX711 driver, provides a class to create an object with variables and functions for controlling an ADXL345 IC. This class uses the on-board SPI interface to set parameters and acquire data from the IC. Functions were written to acquire data from the IC from an individual axis; and all three axes simultaneously. The ADXL345 outputs data in the form of a floating-point number between 0 and 1, scaled depending on the range setting of the device. The driver returns the data as a floating-point number. Like the HX711 driver, functions like *enable*

() and *disable ()* are provided, as well as functions for setting the range and conversion frequency for the device.

2.1.3.2 Multi-tasking

In embedded system software, a program is typically run in an infinite loop that will never exit so long as power is supplied to the device. Often times, a system must respond to one or multiple external stimuli at a time. The Simblee™ SOC ARM Cortex M0 is a single-core processor, meaning that it may only execute one instruction at a time. To combat this difficulty, finite-state machines may be used that execute certain lines of code given the state. Multiple finite-state machines were used in this design to control the device, along with a simple round-robin scheduler to determine which finite-state machine, or task, to run at any given time. While the Simblee™ still may only execute one instruction at a time, at a frequency approaching 16 MHz (Simblee™ Corp. 2016), this multi-tasking scheme ensures that each task gets appropriate attention from the processor in a timely manner.

The multi-tasking environment written for this device is cooperative, meaning that each task runs one state and then yields control back to the scheduler, which then determines which task is run next. This approach is contrary to preemptive, interrupt-driven environments, where the multi-tasking scheduler determines when the switch to another task occurs. Cooperative multitasking simplifies the programming difficulties, as the programmer does not need to consider things that could be problematic in a preemptive environment, like resource allocation. However, the programmer must also be wary of designing tasks that wait a comparatively long time without yielding control back to the scheduler in a cooperative environment.

2.1.3.3 Tasks

For this device, four finite-state machine style tasks were written that have variables associated with them to store the task state and the next task run time. The scheduler uses the next run time of the task to determine when to run the task again. Before a task is run, the task run

frequency is added to the corresponding next run time variable that the scheduler continuously compares to the program run time. Each task was built as a function with conditional statements that run code based on the current state of the task. Shares were used for inter-task communication. A share is a flag or variable stored in RAM that is only accessible and overwritable by certain tasks. A task diagram detailing the priority and timing of each task, and the shares between tasks is shown below in Figure 2-11. Dashed lines indicate that the share is a binary flag, and solid lines indicate that the share is a number.

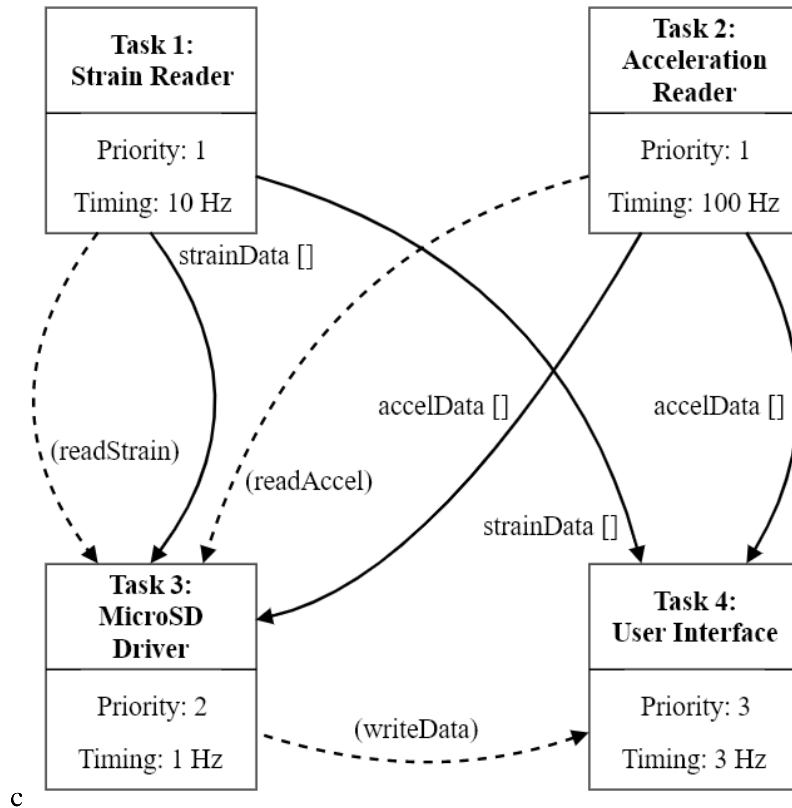


Figure 2-11: Task diagram for quasi-static data acquisition system.

Each task is given a run frequency and priority that the scheduler uses to determine which task to run next. The scheduler runs an infinite loop that compares the next run time of the task and the program time. A task runs when the program time is greater than or equal to the next run time.

Each task is also assigned a priority. In this environment, the priority of a task indicates the order in which the scheduler checks the next run time of each task.

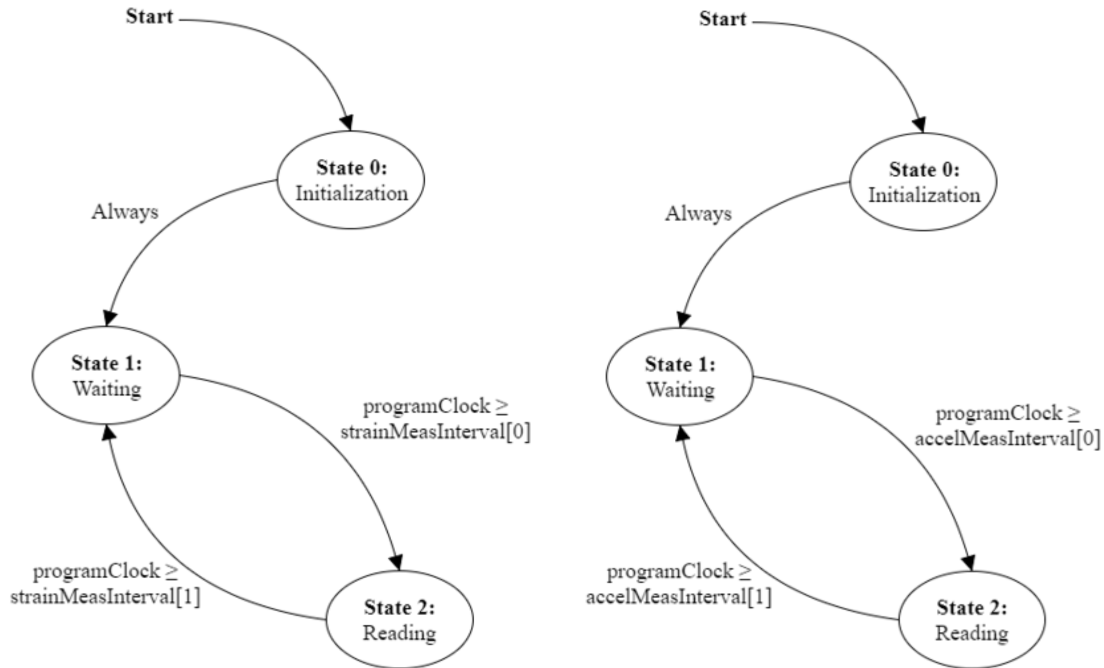


Figure 2-12: Strain Reader (left) and Acceleration Reader (right) task state transition diagrams.

The Strain Reader and Acceleration Reader tasks are responsible for controlling and acquiring data from the HX711 and ADXL345, respectively. The state machine designs for these tasks are almost identical, and are displayed in Figure 2-12. The initialization states for each task configure the GPIO pins and parameters for each IC and then transition to state 1. While in state 1, the task waits for the program clock to reach the user-specified time interval at which to take data. Once the interval is reached, the task transitions to state 2 and begins acquiring data, and stores the data in the corresponding share. In addition, while in state 2, a flag is raised and shared with Task 3 so that the MicroSD Driver Task knows to save either strain or acceleration data. When the data acquisition time interval is passed, the task transitions back to state 1. Both tasks are run at a frequency of 1 Hz so that data can be averaged and acquired every second.

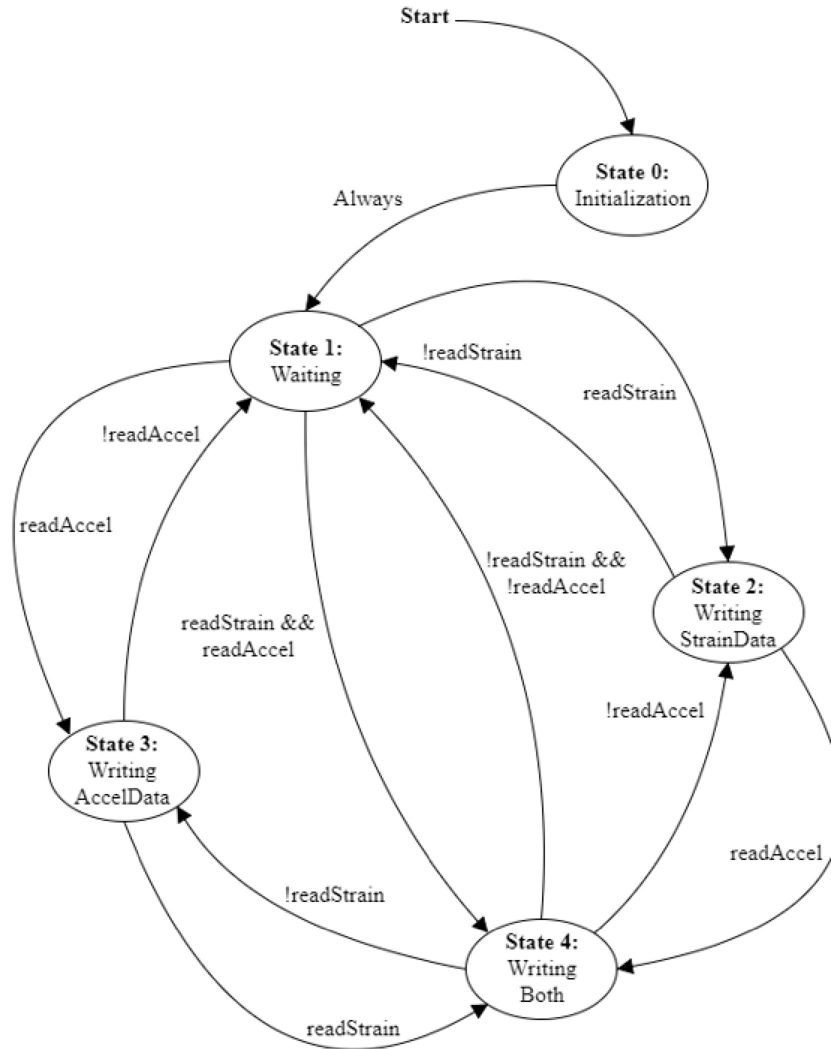


Figure 2-13: MicroSD Driver task state transition diagram.

The MicroSD Driver task, whose state machine is displayed in Figure 2-13, stores timestamped data to a microSD card. The initialization state configures the necessary GPIO pins, checks to see if a microSD card is inserted, and creates and saves a .txt file to the microSD where the data will be stored. In state 1, the task waits until the device begins acquiring data. The task may enter three different states depending on whether the device is acquiring strain data, acceleration data, or both. In addition, the task raises a flag to signal the user interface state to begin printing data to the serial monitor. When both tasks 1 and 2 are done acquiring data, the task transitions back to state 1.

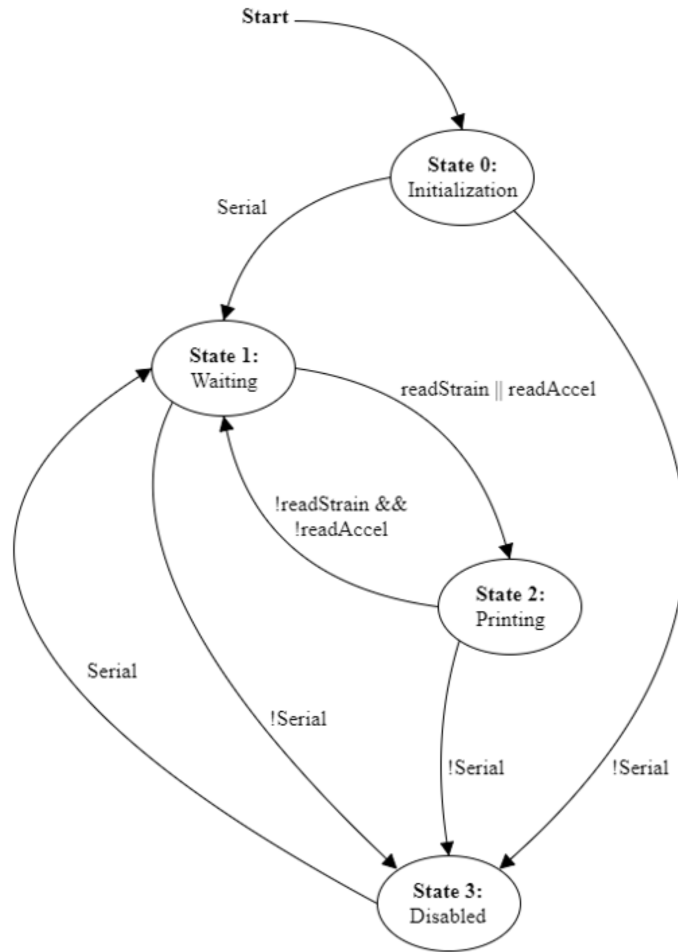


Figure 2-14: User Interface task state transition diagram.

The User Interface task is responsible for responding to commands and parameters sent over the serial port, and for printing data to the serial port while the device is acquiring data. The design of the state machine is shown in Figure 2-14. The task begins in the initialization state when it prompts the user to enter parameters like the time, date, and the time intervals at which to acquire strain and acceleration data. When the user enters all necessary parameters, the task transitions to state 1 where it waits until the device begins acquiring data. In state 2, the task will print data acquired by the device through the serial port. If the serial port becomes unavailable or is in an error state, the task transitions to state 3 where no code is executed, regardless of the current state.

2.2 Device Testing and Validation

The quasi-static data acquisition device was tested extensively to verify that it met the requirements discussed in Section 1.2. The current draw of the device during various operations was measured and analyzed to verify the device performance. Data was acquired by the device at room temperature and at lower temperatures in a thermal vacuum chamber to verify the performance and survivability at temperatures expected during flight testing. The strain data acquired by the device was verified for accuracy by comparing the results with stress/strain theory.

2.2.1 Current Draw

To determine the current draw of the device during various operating conditions, a USB cable was modified to allow a digital multimeter to be placed in series between the device and a USB port on a computer. The RFduino™ USB Shield was used instead of the AAA Battery Shield to power the device for this test. The current draw and power consumption of the device under various operating conditions are provided in Table 2-5.

Table 2-5: Current draw and power consumption of device under various operating conditions (5V source).

Device Condition	Current Draw (mA)	Power Consumed (W)
Waiting	20.0	0.10
Reading Strain	51.6	0.26
Reading Acceleration	22.5	0.11
Reading both	52.0	0.26

When the device is waiting, the only component that has a significant current draw is the Simblee™ microcontroller. The device required an additional 2.5 mA to obtain and record acceleration data using the ADXL345 accelerometer and microSD shield. Approximately 32 mA is added to the current draw when the device is reading and recording strain data using the HX711 amplifier and microSD shield. This large increase in current consumption is due to the current draw

of the Wheatstone Bridge. Strain gauges and bridge resistors with larger resistances could be used to minimize the current draw of the bridge and thus reduce the power consumption of the device.

2.2.2 Data Acquisition Testing

Strain and acceleration data were acquired at room temperature to test the device. The device stores a strain data point as a floating-point number in the form V_{out}/V_{in} x 10^3.The data are converted from volts to μ-strain after offloading the data from the microSD card using Equations 2-1 and 2-2. Acceleration data are stored in g as a floating-point number between 0 and 1, scaled by the range setting of the ADXL345. A sample set of strain and acceleration data acquired at a rate of 1 Hz is displayed in Table 2-6.

Table 2-6: Sample set of strain and acceleration data acquired at a frequency of 1 Hz.

Time	Strain Measurement	X-Axis	Y-Axis	Z-Axis
(ms)	V_{out}/V_{in} x 10^3 (-)	(-)	(-)	(-)
0	0.537	0	0.004	0.996
1000	0.5369	-0.004	0.012	1
2000	0.5368	0	0.008	0.996
3000	0.5367	0	0.008	0.996
4000	0.5366	0	0.004	0.992

A beam bending test was conducted to verify the data from the HX711 by comparing the readings to theoretical results. A 120 Ω constantan strain gauge was mounted on a 3 mm x 19 mm x 107 mm aluminum beam that was secured to the testing mount as shown in Figure 2-15. The strain gauge was mounted 35 mm from the secured end of the beam. A small hole at the end of the beam allowed for various masses to be suspended on the end of the beam while data is collected.

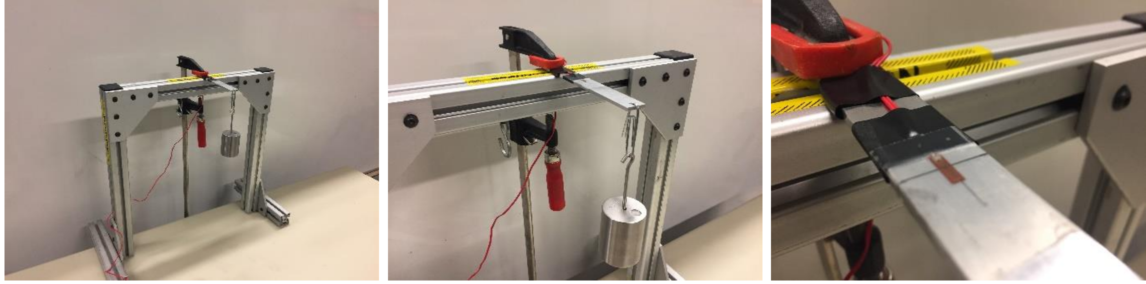


Figure 2-15: Strain gauge testing setup. Overall testing setup (left), beam and dangling mass (middle), closeup of strain gauge location (right).

The beam used to conduct this test was treated as a cantilever beam with a rectangular cross-section and a point mass at its end to produce the theoretical results. Assuming perfectly elastic deformation occurs, and the aluminum beam has an elastic modulus of 68.9 GPa, Equations 2-3, 2-4, and 2-5 were used to calculate the theoretical strain (ASM, 2018). The cantilever beam schematic used to derive Equation 2-5 is shown in Figure 2-16.

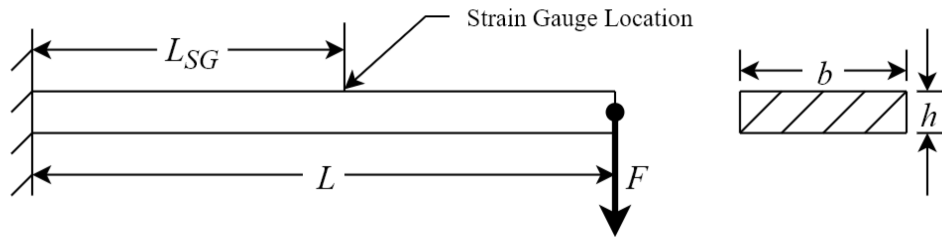


Figure 2-16: Cantilever beam schematic.

$$\sigma = \frac{Mh}{2I} \tag{2-3}$$

$$\varepsilon = \frac{\sigma}{E} \tag{2-4}$$

$$\varepsilon = \frac{6F(L - L_{SG})}{bh^2E} \tag{2-5}$$

The results of this test are provided in Figure 2-17. Masses of 0.1, 0.2, 0.5, 1.0, 2.0, and 5.0 kg were suspended at the end of the beam to produce the results. In addition, the beam was inverted and tested again to verify the readings while the gauge is in both tension and compression.

Equations 2-1 and 2-2 were used to convert the voltage ratio readings provided by the HX711 to mechanical strain.

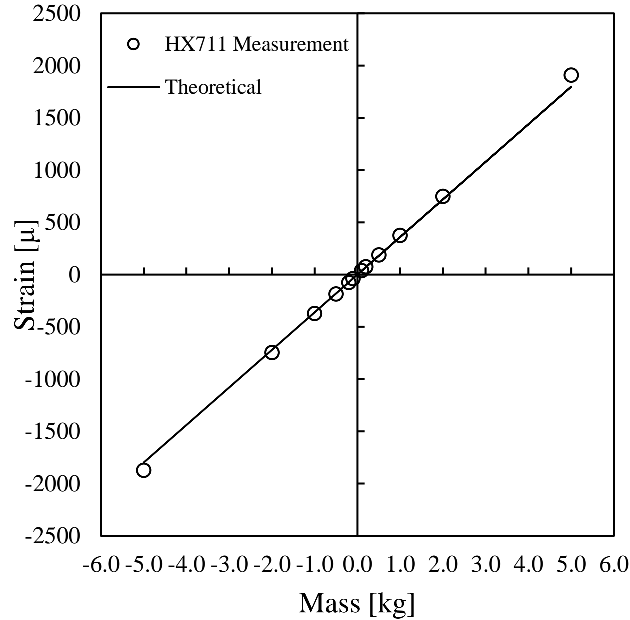


Figure 2-17: Comparison of HX711 readings to theoretical strain values.

The negative mass values shown on the x-axis in Figure 2-17 indicate that the beam was flipped over so that the strain gauge was in compression. The HX711 provided linear results that agreed with the theoretical values of strain, only deviating slightly. A typical value of tensile yield strength for aluminum is 276 MPa (ASM, 2018). Using the strain data from Figure 2-17, the maximum bending stress experienced by the beam was approximately 124 MPa, indicating that the beam was never strained beyond its elastic region during this test. A lack of precise dimensions of the beam and a slightly lower elastic modulus than the assumed 68.9 GPa used in the theoretical calculations are likely causes of this slight deviation.

2.2.3 Low Temperature Testing

Another test was conducted to study the survivability of the device at low temperatures and to determine the effect of temperature on the data acquired by each sensor. The prototype device and the beam with the mounted strain gauge used in the previous test were placed in a vacuum

chamber with dry ice to decrease the temperature to typical flight-test temperatures. A vacuum pump was used to lower the pressure in the chamber to avoid condensation on the device. The device and beam were placed such that the strain gauge was under no load and the accelerometer was reading approximately 0.0 g on its x-axis and y-axis and 1.0 g on its z-axis at room temperature. An absolute pressure gauge and thermocouple were used to acquire the pressure and temperature inside the chamber. Once the chamber temperature reached steady state at about -36 °C, heaters inside the chamber were activated to sublimate the dry ice and slowly bring the chamber back to room temperature. Pressure and temperature data were recorded every five minutes during the chamber cooling and heating processes to relate deviation of the device readings to temperature. Figure 2-18 shows the deviation in acceleration while Figure 2-19 shows the deviation in mechanical strain.

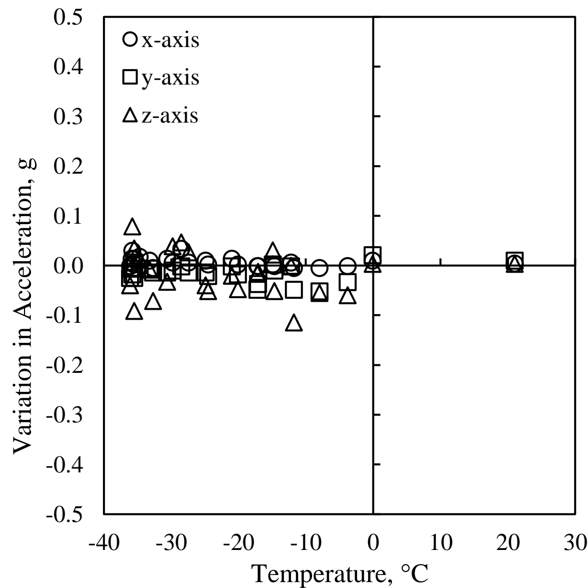


Figure 2-18: Vacuum chamber test results for variation in acceleration as the chamber temperature decreases.

The deviation in readings from the three axes of the ADXL345 accelerometer in the vacuum chamber test were used to create the plot in Figure 2-18. The effects of temperature on the acceleration readings were minimal compared to the strain readings of the HX711. There is no

trend in the data due to temperature, but there is an observable increase in noise for all three axes as the temperature decreases. A comparison of the standard deviation in the data at room temperature and at -36 °C is detailed in Table 2-7.

Table 2-7: Comparison of the standard deviation in the acceleration data at room temperature and at -36°C.

Axis	Average Reading	Standard Deviation	
		22°C	-36°C
	(-)	(-)	(-)
X-axis	0.00233	0.00333	0.00825
Y-axis	0.01792	0.00667	0.03563
Z-axis	0.99533	0.00850	0.05641

The HX711 was outputting negative values of strain during this test, indicating that the voltage across the Wheatstone Bridge corresponded to a gauge in compression. A data point at room temperature was included to show that the HX711 was reading no load initially. Temperature data were taken during the cooling process from -12 °C to -36 °C and during the heating process from -36 °C to 0 °C.

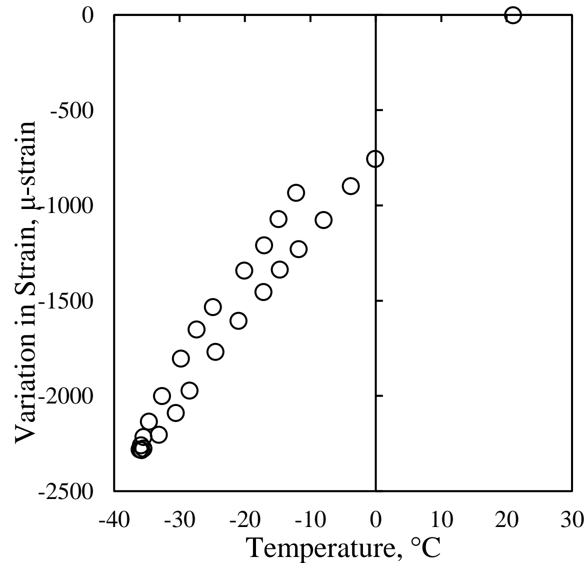


Figure 2-19: Vacuum chamber test results for absolute variation in mechanical strain as the chamber temperature decreases.

The maximum strain deviation occurred at -36 °C, with approximately 2276 μ from the reference value at room temperature. Additionally, there was approximately 300 μ of hysteresis between the readings taken during cooling and the readings taken during heating. There are many effects that could be responsible for the deviation in the data. Typically, constantan gauges have about a ±1.2 μ of error per ±1 °C of temperature variation, which could account for about 70 μ of the deviation seen in the data. The resistance of each precision resistor in the Wheatstone Bridge and the thermal shrinkage of the beam are both functions of temperature, which would be expected to cause further deviation in the data. These effects are captured by Equations 2-6 and 2-7, respectfully.

$$R = R_{ref}\left[1 + \alpha_T\left(T - T_{ref}\right)\right] \tag{2-6}$$

$$\varepsilon = \alpha_l\left(T - T_{ref}\right) \tag{2-7}$$

The published temperature coefficient of resistivity of the precision resistors utilized in this prototype is 0.05 ppm/°C (Vishay, 2019). This would account for approximately 14 μ of additional error in the measurement at -36 °C. Aluminum has a linear expansion coefficient of 24 ppm/°C, which could theoretically add 1344 μ of compressive strain at -36 °C (ASM, 2018). The remaining 848 μ of difference could be due to the temperature of the device causing the internal circuitry of the HX711 to behave differently than if it were at room temperature.

Additional testing was conducted to further investigate the large deviation in the strain measurement of the unloaded gauge observed when the device is subjected to low temperatures. The aluminum beam with the mounted strain gauge was placed in a freezer at -36 °C while the bridge resistors were at 22 °C along with all BLDS electronics recording strain data. Additionally, the electronics were placed in the freezer at -36 °C and recorded data while the aluminum beam was at 22 °C. The results of these tests are shown in Figure 2-20.

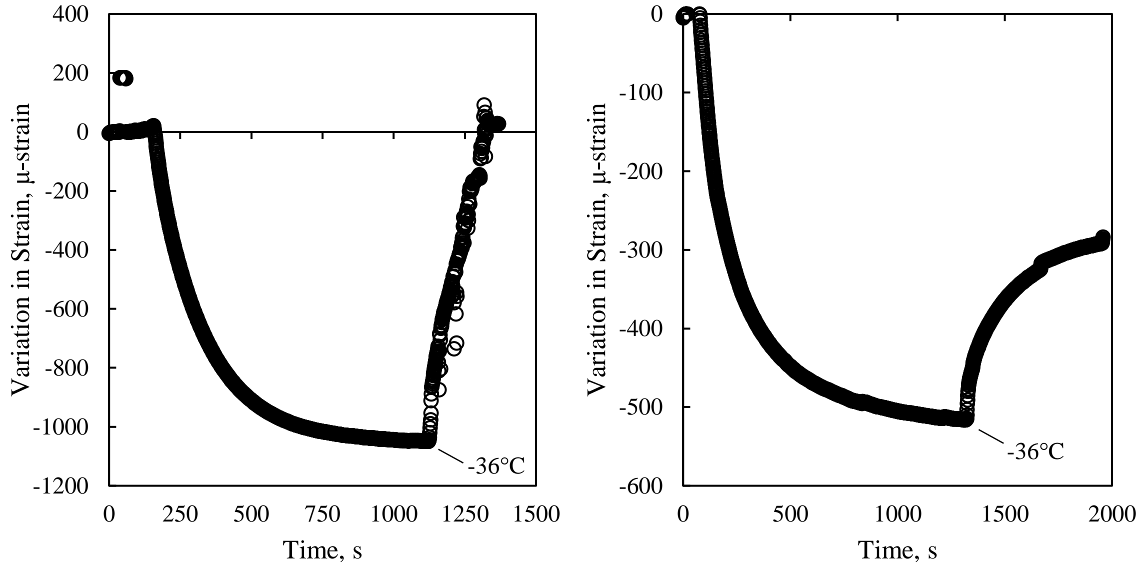


Figure 2-20: Results of freezer testing. Left: Strain readings with aluminum beam and strain gauge at -36 °C and electronics at 22 °C. Right: Strain readings with electronics at -36 °C and aluminum beam and strain gauge at 22 °C.

Before the aluminum beam was placed in the freezer, a 0.5 kg mass was suspended from the beam in the same way as previously described to verify that the HX711 provided the same result as shown in Figure 2-20. The unloaded beam was then placed in the freezer until the temperature reached -36 °C, when the strain reading settled at approximately -1050 μ, at which point the beam was removed and brought back to room temperature. The deviation seen in this test was due to the constantan gauge ±1.2 μ of error per ±1 °C of temperature variation and the thermal shrinkage of the beam. Theoretically, this deviation should be -1414 μ, although this was about 40% greater than what was observed.

The device was placed in the freezer and recorded strain data until the temperature settled at -36 °C, where the strain reading settled at approximately -514 μ, at which point the device was placed in a refrigerator to minimize condensation on the device while warming up. The unloaded beam was at room temperature during this time. Approximately 14 μ of the deviation seen in this test was due to the 120 Ω precision resistors changing resistance due to the reduced temperature. The other 500 μ of deviation was likely caused by the internal circuitry of the HX711 behaving differently while not at room temperature.

Table 2-8: Tabulated strain deviation of device from 22 °C to -36 °C.

<table>
<tr><th rowspan="2">Cause</th><th colspan="3">Deviation in Strain (μ)</th></tr>
<tr><th>Estimated</th><th>Vacuum Chamber Testing</th><th>Freezer Testing</th></tr>
<tr><td>Constantan gauge error due to temperature</td><td>-70</td><td rowspan="4">-2276</td><td rowspan="2">-1050</td></tr>
<tr><td>Thermal shrinkage of aluminum beam</td><td>-1344</td></tr>
<tr><td>Change in resistance due to temperature</td><td>-14</td><td rowspan="2">-584</td></tr>
<tr><td>HX711 error</td><td>-</td></tr>
<tr><td>Total</td><td>-1428</td><td>-2276</td><td>-1634</td></tr>
</table>

Table 2-8 shows the possible sources of error in the strain measurement along with the estimated and observed deviation associated with each source as seen in testing. The total deviation observed in the freezer tests was -1634 μ. Comparing this value to the -2276 μ deviation seen in the vacuum chamber tests leaves -642 μ of compressive strain that was not observed in the freezer tests.

A change in temperature has a clear effect on the offset in strain measurements. Whether this offset is due to changes in resistance of the bridge resistors or strain gauges, or the compression of the material at low temperatures, there will be an offset present while the device is acquiring data at -40 °C to -60 °C during a flight test. A possible solution to this problem is to run a routine in software that calibrates the device during the flight test before acquiring measurements. Calibrating the device when the device is at flight test temperature will ensure that the strain data will capture strain due to the mechanical stress on the aircraft surface of interest.

2.3 Summary – Quasi-Static Data Acquisition System

A prototype of the first BLDS-M device to incorporate non-flow measurement sensors has been developed to read mechanical strain and acceleration data and write the time stamped measurements to a microSD card for data logging. The system was designed by synthesizing various breakout boards for the selected components with a simple PCB. Breakouts provided by RFduino™ and Sparkfun were utilized to accelerate the development process. The HX711 and

ADXL345 were selected for acquiring strain and acceleration data, respectively. Multi-tasking software was written to control the device which runs multiple finite-state machine style processes. Custom drivers were written and libraries provided by Arduino were utilized to control the external ICs and peripherals.

The performance of the device has been tested through a power consumption test to determine the current draw of the device under various operating conditions. Additionally, the reliability and survivability of the device has been tested through a simple beam bending test and a thermal vacuum chamber test. The results of the beam bending test were compared with theory to verify the HX711 was providing accurate measurements. Data acquired during the low temperature testing was analyzed to determine the cause of the offset present in the data. The device is simple to use, small, and is entirely autonomous and battery-powered; it can be used stand-alone or, can communicate wirelessly with other BLDS modules or a computer.

3. DYNAMIC DATA ACQUISITION SYSTEM

This chapter describes the design and development of a benchtop proof-of-concept device intended for testing various methods of acquiring dynamic strain measurements which would be practical to use on an aircraft during flight. The device is powered by a USB port on a computer and is controlled by a serial interface through the same USB port. Data is saved to an on-board microSD card and may also be monitored in real-time through the serial interface. The design and components of this device are small, lightweight, and may be used in future BLDS designs for acquiring dynamic strain measurements.

3.1 Device Design

The proof-of-concept device incorporates six channels for acquiring data using various methods. Three of the channels sample data at a maximum rate of 1.3 kHz with 16-bit resolution. These channels are intended for testing three different Wheatstone bridge configurations to measure strain. The other three channels sample data at a maximum rate of 2 MHz with 12-bit resolution, and test various methods of filtering the measurements electronically.

The components utilized in this design were selected using the requirements outlined in Table 1-3. Two different approaches were taken for acquiring dynamic strain data, both of which involve reading voltage across a Wheatstone bridge. The first approach was to use an external chip with an ADC that amplified and filtered the bridge output digitally. The second approach was to filter and amplify the bridge output electronically, and convert the signal using the on-board ADC interface on the Atxmega128A4U.

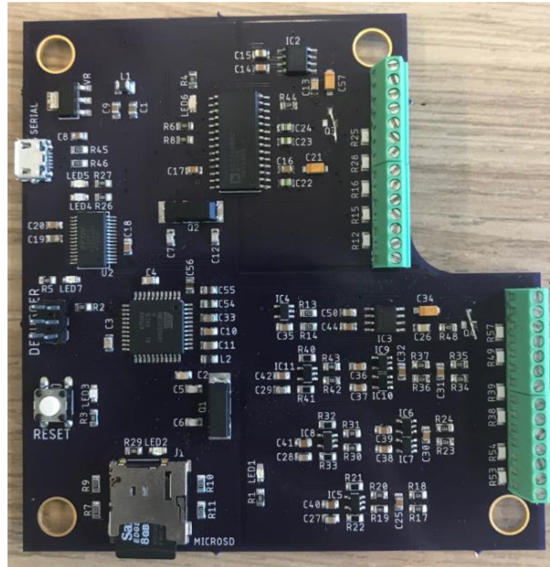


Figure 3-1: Top view of assembled dynamic data acquisition system.

A custom PCB was designed to house the electronics with microUSB and microSD sockets for supplying power and storing data, respectively. The PCB design includes screw terminals for connecting strain gauges to the on-board Wheatstone bridges. The overall PCB dimensions are 3.75 in x 3.8 in x 0.3 in. A top view of the assembled PCB is shown in Figure 3-1.

3.1.1 Component Selection and Specifications

3.1.1.1 Atxmega128A4U

The Atxmega128A4U is a 16-bit microcontroller and a member of a family of low power, high performance, and peripheral rich microcontrollers based on the AVR enhanced RISC architecture (Atmel, 2014). This microcontroller has the ability to execute powerful instructions in a single clock cycle, approaching a throughput of one million instructions per second per megahertz. The operating frequency of the device is set by the internal oscillators or by an external oscillator. The maximum operating frequency depends on the operating voltage: 0 – 12 MHz from 1.6 V, and 0 – 32 MHz from 2.7 – 3.6 V. The device includes 8 KB of RAM and 128 KB of programmable flash memory with a two-pin program and debug interface (PDI) for programming the device. The following relevant features are provided: eight-channel event system and

programmable multilevel interrupt controller, 34 GPIOs, 16-bit real-time counter (RTC), five 16-bit timer/counters, five USARTs, two two-wire serial interfaces (TWI), two SPI interfaces, and twelve 12-bit ADC interface channels (Atmel, 2014). A block diagram of the device is shown in Figure 3-2. The interrupt controller may be used to assign low, medium, or high priority for interrupts triggered by internal or external stimuli.

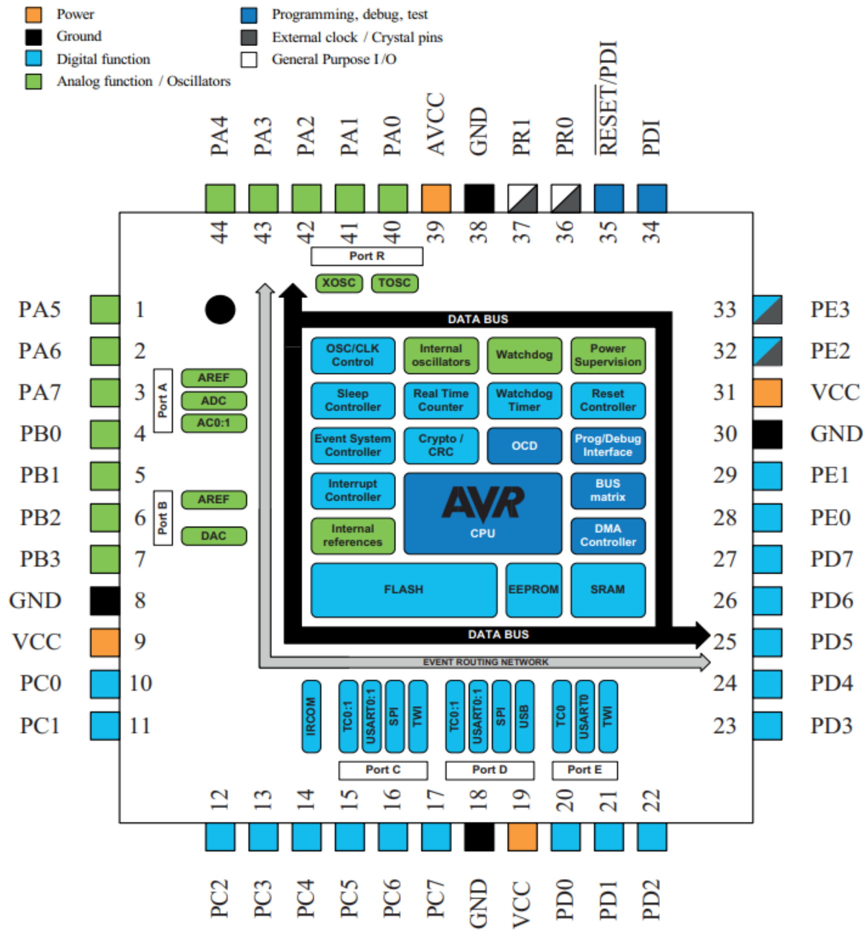


Figure 3-2: Pinout/block diagram of the Atxmega architecture (Atmel, 2014).

The Atxmega128A4U has 44 pins which are grouped into five ports. Ports A and B are for analog functions and peripherals, while ports C, D, and E are for digital functions and peripherals. Port R is reserved for the programming pins and external clock/crystal pins. The on-board ADC interface has 12-bit resolution, 2 MHz conversion frequency, and four of the twelve input channels

that may be sampled simultaneously. Each channel has individual input selection, result registers, and conversion start control (Atmel, 2014). Each channel may be tied to an interrupt that triggers when a conversion is complete. The ADC may be configured for 8 or 12-bit results, with conversion times of 2.5 µs or 3.5 µs, respectively.

3.1.1.2 AD7708

Although the Atxmega128A4U has more than enough ADC channels to satisfy the device requirements, it was desired to test a 16-bit ADC as well. The AD7708 is a 16-bit sigma-delta ADC with a programmable gain amplifier for low frequency measurement applications (Analog Devices, 2001). This ADC features 5 fully-differential input channels or 10 pseudo-differential input channels with individually configurable input signal ranges from 20 mV to 2.56 V. The device operates from a 32 kHz external crystal with an on-board Phase-locked-loop (PLL) for generating the required 4.194 MHz operating frequency (Analog Devices, 2001). The block diagram of the device is shown in Figure 3-3.

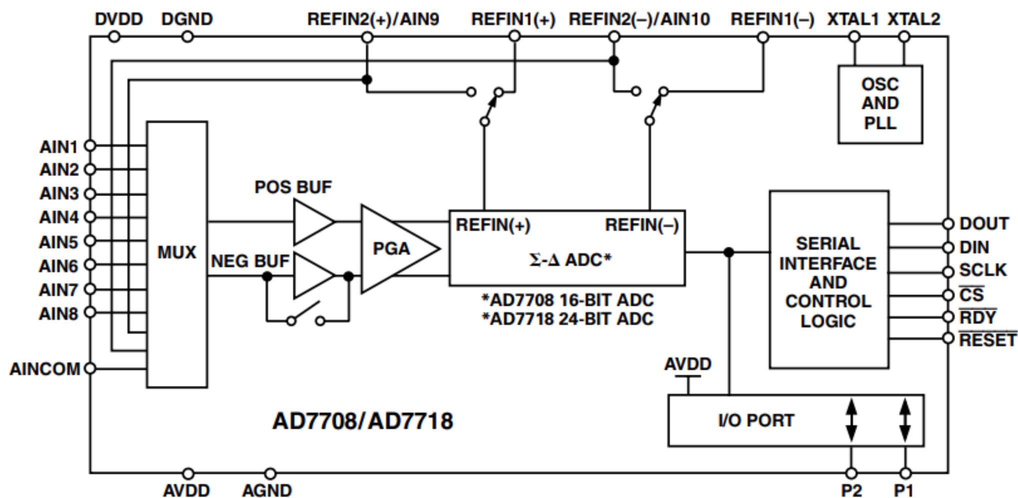


Figure 3-3: AD7708/AD7718 pinout/block diagram (Analog Devices, 2001).

The output data rate from the part is programmable, with a maximum conversion frequency of 1.3 kHz. The part is optimal for operating from a 5 V supply because when operating from a 3

V supply, the part dissipates about 3.84 mW. The device is controlled through a 4-wire SPI interface or a two-wire I/O port. A truncated list of device features is displayed in Table 3-1.

Table 3-1: Truncated list of AD7708 features (Analog Devices, 2001).

Features
8/10 channel sigma-delta ADCs
16-bit resolution
SPI compatible
Optimizable for analog performance or channel throughput
1.3kHz conversion frequency
Normal Operation: 1.28mA at 3.3V
Power-Down: 30 µA

3.1.1.3 FT232R

The FT232R is a USB to serial UART interface for converting and transferring UART data to a USB port on a computer. The chip handles the USB communication protocol on-board, thus no USB specific firmware programming is required. It is able to handle data transfer rates from 300 baud to 3 Mbaud with a 128 byte receive buffer and a 256 byte transmit buffer utilizing buffer smoothing technology to allow for high data throughput (FTDI, n.d.). The device includes an internal oscillator and accepts a logic level between 1.8 V and 5 V. In addition, the device has two LED driver pins, which indicate when the device is transmitting or receiving bytes. The device has a driver available for communicating through a COM port on a computer that may be monitored through a serial terminal (FTDI, n.d.).

3.1.2 System Configuration

The components discussed above were synthesized into a high-level system design for the dynamic data acquisition system, whose block diagram is shown in Figure 3-4. The Atxmega1284AU runs code written in C/C++ to control the device. Software is flashed onto the Atxmega128A4U using the on-board PDI interface and an Atmel ICE Debugger. The Atmel ICE

debugger is a powerful development tool for programming and debugging Atmel SAM™ and Atmel AVR™ microcontrollers (Atmel, 2016).

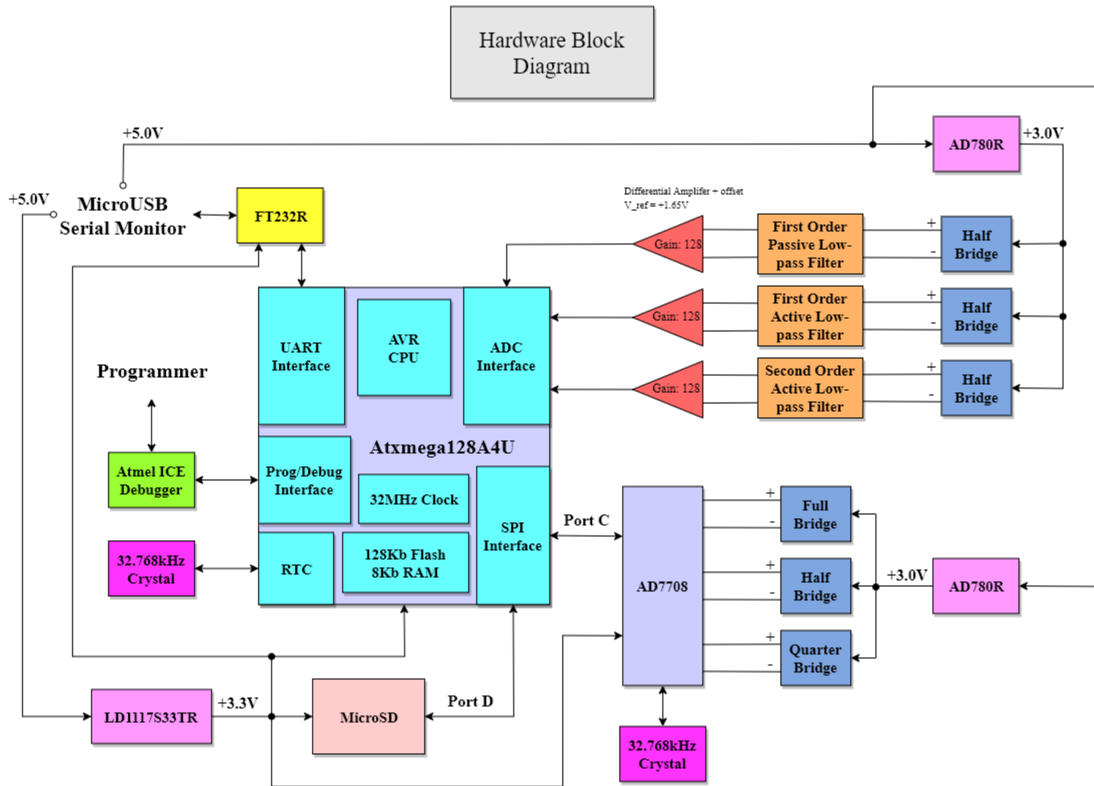


Figure 3-4: Dynamic data acquisition system device hardware diagram.

The on-board UART interface is used to transmit/receive data from the FT232R chip, which transmits/receives data from a serial terminal on a computer. The serial connection and 5 V power are supplied by a computer through a microUSB socket. The low-dropout regulator (LDO) LD1117S33TR was selected to regulate the supply voltage to 3.3 V. This device is the same LDO used in the RFduino™ USB Shield, and was proven to function at low temperatures during the quasi-static device testing. To provide a 32.768 kHz clock frequency for both the Atxmega128A4U and the AD7708, the MC-405 crystal was selected due to its high precision and stability at low temperatures (EPSON, n.d.). One crystal is used to drive the internal PLL of the AD7708, and another is used to drive the internal RTC of the Atxmega128A4U. To control the AD7708, the SPI interface on port C is used in 4-wire mode at a frequency of 4 MHz. The SPI interface on port D is

used in 4-wire mode at 8 MHz to write data to a microSD card. Separate SPI interfaces were used for interfacing with the AD7708 and a microSD card so that the device may communicate with both devices at once if necessary. The pinout for the Atxmega128A4U with pin descriptions and utilizations is displayed in Table 3-2.

Table 3-2: Atxmega128A4U chip pinout for dynamic data acquisition system.

Number	Pin Name	Description	Use
1	PA5	Port A GPIO 5	NC
2	PA6	Port A GPIO 6	NC
3	PA7	Port A GPIO 7	NC
4	PB0	Port B GPIO 0	Status LED
5	PB1	Port B GPIO 1	Status LED
6	PB2	Port B GPIO 2	Status LED
7	PB3	Port B GPIO 3	Status LED
8	GND	Ground	GND
9	VCC	Supply voltage	VCC
10	PC0	Port C GPIO 0	NC
11	PC1	Port C GPIO 1	Wheatstone bridge toggle
12	PC2	Port C GPIO 2	AD7708 reset
13	PC3	Port C GPIO 3	AD7708 status
14	PC4	Port C GPIO 4	AD7708 CS
15	PC5	Port C GPIO 5	SPI MOSI
16	PC6	Port C GPIO 6	SPI MISO
17	PC7	Port C GPIO 7	SPI SCK
18	GND	Ground	GND
19	VCC	Supply voltage	VCC
20	PD0	Port D GPIO 0	NC
21	PD1	Port D GPIO 1	NC
22	PD2	Port D GPIO 2	NC
23	PD3	Port D GPIO 3	SD card detect
24	PD4	Port D GPIO 4	SD card CS
25	PD5	Port D GPIO 5	SPI MOSI
26	PD6	Port D GPIO 6	SPI MISO
27	PD7	Port D GPIO 7	SPI SCK
28	PE0	Port E GPIO 0	NC
29	PE1	Port E GPIO 1	NC
30	GND	Ground	GND
31	VCC	Supply Voltage	VCC
32	PE2	Port E GPIO 2	RXD
33	PE3	Port E GPIO 3	TXD
34	PDI_DATA	PDI data pin	Programming/Debugging
35	PDI_CLK	Reset/PDI clock pin	Reset/Programming/Debugging
36	PR0	External clock/crystal pin 0	External crystal
37	PR1	External clock/crystal pin 1	External crystal
38	GND	Analog ground	GND
39	AVCC	Analog supply voltage	AVCC
40	PA0	Port A GPIO 0	3.0V Reference
41	PA1	Port A GPIO 1	ADC CH1
42	PA2	Port A GPIO 2	ADC CH2
43	PA3	Port A GPIO 3	ADC CH3
44	PA4	Port A GPIO 4	Wheatstone bridge toggle

Three channels on the AD7708 are wired to read voltage across three different Wheatstone bridges. To test different Wheatstone bridge configurations, a quarter-bridge, half-bridge, and full-bridge are connected to each of the three ADC channels. The quarter-bridge configuration is shown

in Section 2.1.2, and the half-bridge and full-bridge configurations are shown below in Figure 3-5. Equations 3-1 and 3-2 are used to calculate strain from the voltage output of the half-bridge and full-bridge, respectively (Omega Engineering, 1999). The 120 Ω resistors utilized in the quasi-static device were switched with 350 Ω precision resistors to reduce the power draw of the Wheatstone bridges (Vishay, 2015).

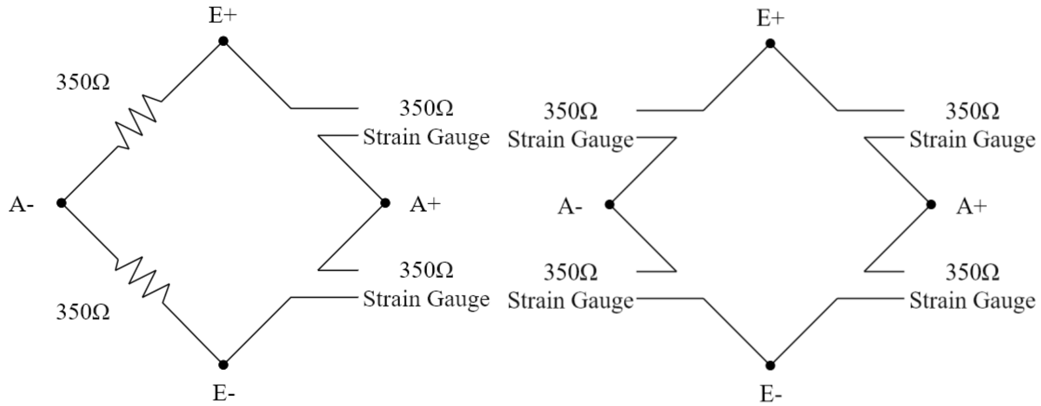


Figure 3-5: Wheatstone half-bridge (left) and full-bridge (right) configurations.

$$\varepsilon = \frac{-2V_{r_{net}}}{GF} \tag{3-1}$$

$$\varepsilon = \frac{-V_{r_{net}}}{GF} \tag{3-2}$$

The excitation voltage for the Wheatstone bridges was supplied using the AD780R, an ultrahigh precision band gap reference voltage IC that supplies 3.0 V from the 5 V device supply (Analog Devices, 2017). The measurement from each bridge is electronically filtered by a 0.1 µF bypass capacitor between the differential signals.

Additionally, three channels on the on-board ADC interface are used to measure voltage across three 350 Ω half-bridge configurations. The half-bridges are excited by a 3.0 V reference voltage supplied by an additional AD780R. Because the on-board ADC has no analog filter capabilities, differential first-order passive, first-order active, and second-order active low-pass filters are used to filter the measurements acquired by each of the three ADC channels. The

operational amplifier used in the first and second-order active filters is the LMV321ILT, a general-purpose rail-to-rail amplifier (STMicroelectronics, 2015). Downstream of the differential filters, a differential amplifier circuit is used to amplify and offset each signal, which is then converted using the on-board ADC. The differential filters and differential amplifier specifications will be presented in Section 3.1.3.2.

3.1.3 Electronics Design and Analysis

A custom PCB was designed to implement the high-level design discussed above into practice. The floorplan of the PCB was designed to minimize board area and measurement noise by separating the analog and digital sections of the board. The schematic and board were designed in Autodesk Eagle™, and are provided in Appendix E.

3.1.3.1 Board Design

The custom PCB was designed as a two-layer board, with each layer containing signal lines, power rails, and a ground plane. The top and bottom layer ground planes are stitched together with vias so they are of the same potential and there is a low impedance path for return current. The board outline is displayed in Figure 3-6.

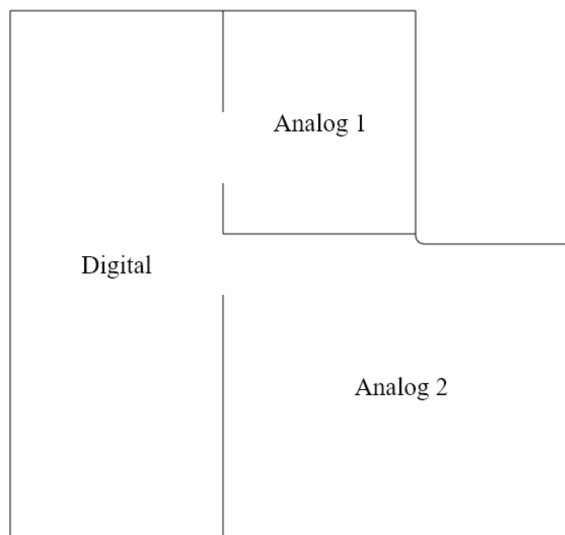


Figure 3-6: Dynamic data acquisition system PCB board outline.

To minimize the amount of digital noise being injected into the analog devices, the PCB ground planes were split into regions, with the analog regions solidly connected to the digital ground plane under each ADC. The microcontroller, LDO, microSD socket, FT232R, and the microUSB socket were segregated in the digital region so that digital noise is not present in the acquired measurements. The two analog regions separate the components used for each measurement method from the digital components.

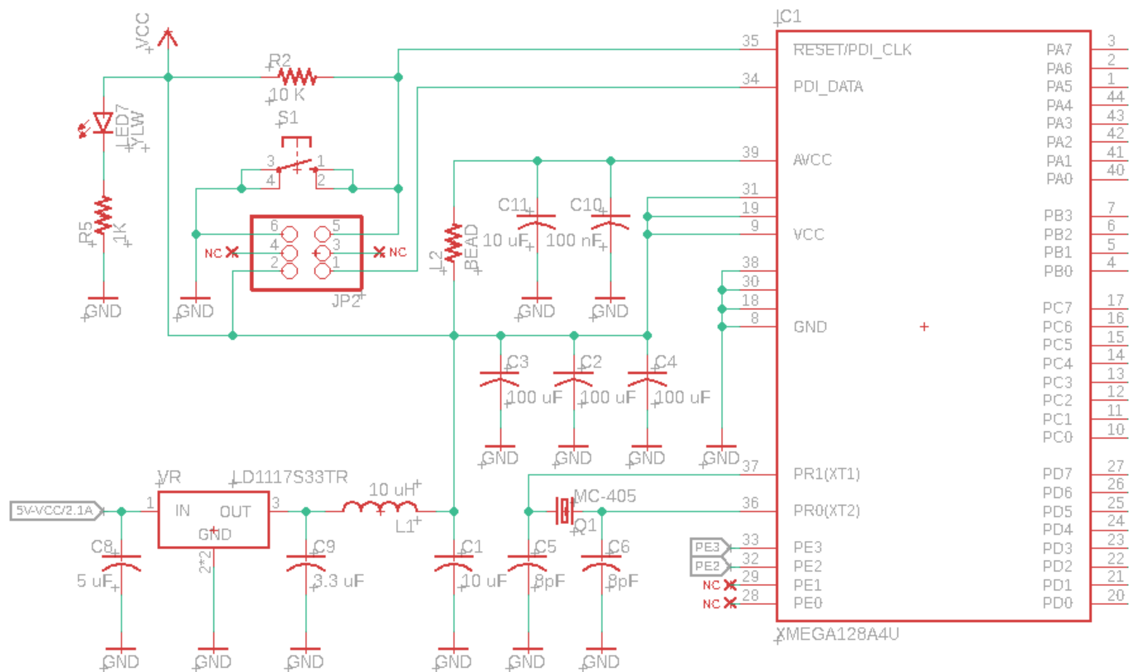


Figure 3-7: Power regulation and microcontroller schematic.

Surface-mount components were utilized throughout the design to minimize the board floor area. The schematic for the LDO and microcontroller circuitry is shown in Figure 3-7. The 5 V input and 3.3 V output of the LDO are filtered with ceramic decoupling capacitors, with an additional inductor-capacitor filter to stabilize the 3.3 V rail. The Atxmega128A4U has three digital supply pins, each decoupled with a large ceramic capacitor, and an analog supply pin, filtered with a ferrite bead and two decoupling capacitors. The reset pin is pulled high with a 10 kΩ resistor with a tactile switch that pulls the pin low when engaged to quickly reset the device. The microcontroller

PDI interface is connected to a programmer/debugger with 2x3 0.1 in pitch headers. The crystal signal lines are decoupled with small ceramic capacitors to stabilize the crystal frequency, as recommended in the datasheet (EPSON, n.d.). Additionally, a yellow LED was placed in series with a 1 kΩ resistor between the 3.3 V rail and ground to indicate that power is being supplied.

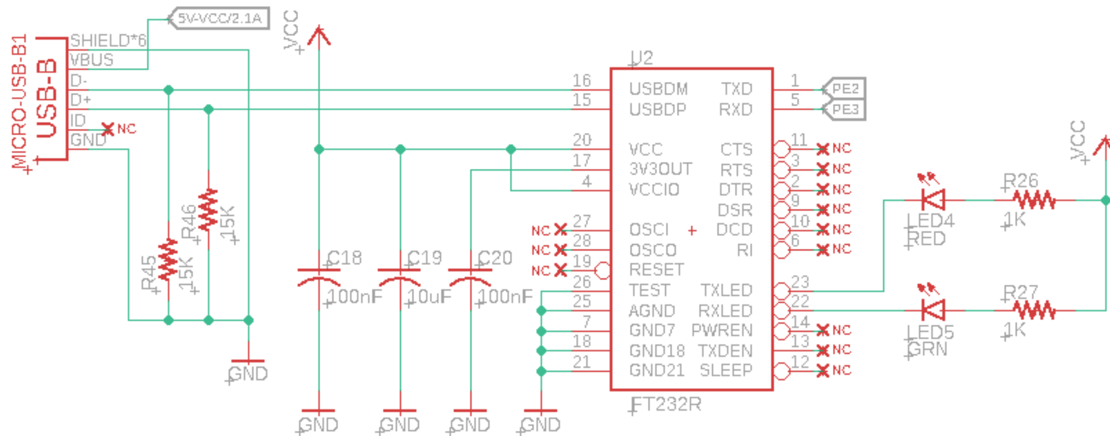


Figure 3-8: FT232R and microUSB schematic.

To communicate to an off-board serial monitor, the FT232R RXD and TXD lines are connected to the TX and RX lines, respectively, to the UART interface of the Atxmega128A4U. This schematic is displayed in Figure 3-8. Data is transferred through two lines, USBDM and USBDP, from the FT232R to the serial monitor which are pulled down with 15 kΩ resistors. The FT232R is powered by the 3.3 V rail and two ceramic decoupling capacitors. Two red and green LEDs are connected in series with two 1 kΩ resistors between 3.3 V and the TX and RX LED pins of the FT232R. The TX and RX LED lines are pulled low, lighting the LED, when data is being transferred and received, respectively. The microUSB socket is a surface-mount, five pin USB 2.0 connector that supplies 5 V to power the device from a computer USB port (FCI, n.d.).

The microSD socket is a push-in push-out, ultra-low-profile connector for storing data to a microSD card (Molex, 2019). Although various communication protocols may be used to write data to a microSD card, the device is designed to write data using SPI. The SPI communication pins are connected to the SPI pins on port C of the Atxmega128A4U, and the unused

communication pins are pulled high with 10 kΩ resistors. The microSD socket features a detect pin, connected to a GPIO on the microcontroller, which is pulled high when a microSD card is inserted and low when no microSD card is present. The CS pin on the microSD socket is pulled high by another 10 kΩ resistor, and pulled low while the device is storing data. The schematic for the microSD socket is displayed in Figure 3-9.

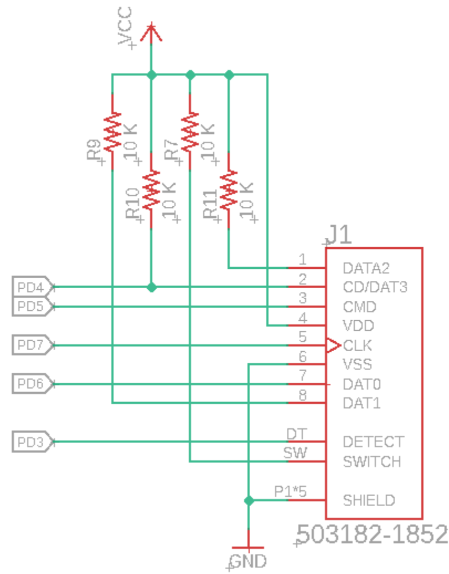


Figure 3-9: MicroSD socket schematic.

The schematic for the AD7708 was designed using the typical application suggested in the datasheet, and is shown in Figure 3-10 (Analog Devices, 2001). Power is supplied from the 3.3 V rail with two ceramic decoupling capacitors on the digital supply and a tantalum capacitor decoupling the analog supply. The crystal signal lines are decoupled with small ceramic capacitors like the implementation in the microcontroller schematic. The reset and CS signal lines are pulled high with 10 kΩ resistors, and the SPI communication pins are connected to the SPI interface on port C of the Atxmega128A4U. In addition, the ready pin that is pulled low when valid data is ready to transfer, is connected to a microcontroller GPIO.

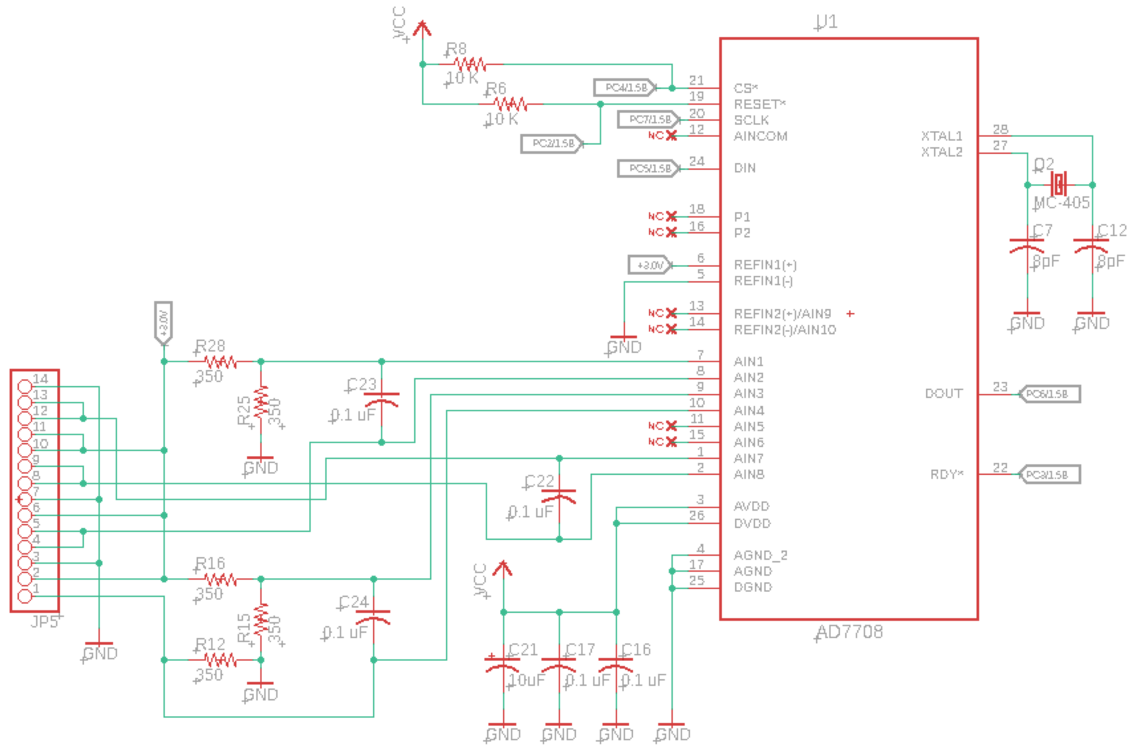


Figure 3-10: AD7708 and quarter, half, and full-bridge schematic.

The outputs of the quarter, half, and full-bridges are each connected to a differential ADC channel of the AD7708. Each differential channel has a 0.1 µF bypass capacitor between the positive and negative inputs to filter any high-frequency noise. The excitation voltage for the Wheatstone bridges is supplied by the 3.0 V output from the AD780R, which also supplies the positive reference voltage for the AD7708. A 14-pin screw terminal provides pinouts for completing each bridge with strain gauges.

Both methods of reading strain have a corresponding AD780R circuit for supplying the excitation voltage for each Wheatstone bridge. The AD780R has the ability to supply 2.5 V or 3.0 V depending on the circuit configuration. A schematic of the implementation of the AD780R in this design is displayed in Figure 3-11. For the device to supply 3.0 V, the O/P select pin of the device should be shorted to ground. As the device datasheet suggests, a 0.1 µF ceramic capacitor is placed between the TEMP pin and ground to reduce noise (Analog Devices, 2017). Decoupling capacitors are connected to the input and output of the device, as well as a polarized tantalum

capacitor to further reduce the noise in the readings from the Wheatstone bridges. In addition, a BSS123L metal-oxide semiconductor field-effect transistor (MOSFET) was placed in series between the 3.0 V reference voltages and Wheatstone bridges to toggle the excitation voltage, driven by a GPIO pin from the Atxmega128A4U (Fairchild, 2014).

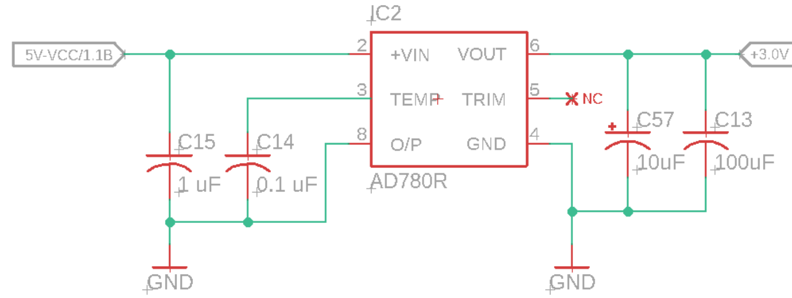


Figure 3-11: AD780R voltage reference schematic.

To acquire measurements using the Atxmega128A4U's ADC interface, the scheme detailed in Figure 3-12 was used to filter and amplify the differential signal from the Wheatstone bridge. The differential output of the bridge is filtered through a differential low-pass filter, which is downstream from a differential amplifier which amplifies and then offsets the signal by $V_{cc}/2$. The signal is connected to a GPIO pin on the Atxmega128A4U which is connected to the positive terminal of the ADC, while the negative terminal is connected to ground. The first-order passive, first-order active, and second-order active low-pass filter designs and the differential amplifier + offset circuit will be discussed below in detail.

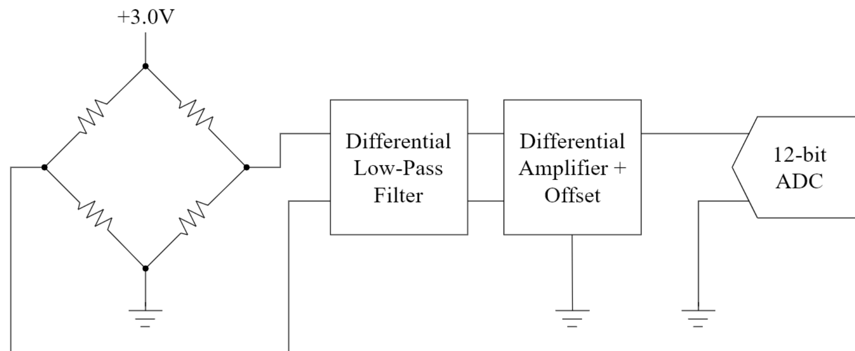


Figure 3-12: High-level ADC data acquisition block diagram.

3.1.3.2 Filter Design

Each electronic low-pass filter is designed for a cutoff frequency that attenuates signals not related to dynamic strain, whose bandwidth is expected to be limited to well below the acquisition frequency of 1 kHz. The filter will attenuate the signal by -3 dB at the cutoff frequency of the filter, so a higher cutoff frequency is desired so that the dynamics of interest are not filtered out. Each filter was modeled using circuit theory and MATLAB™ to help predict the filter performance. The derivations of each filter transfer function and cutoff frequency as well as the MATLAB™ simulation script are provided in Appendix F. A cutoff frequency of 5 kHz was chosen based on the predicted performance of each filter, which attenuates high-frequency electrical noise while leaving the frequency of interest unaffected.

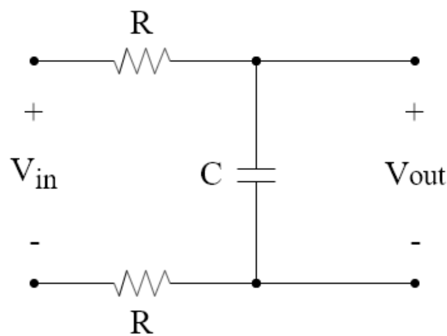


Figure 3-13: First-order passive low-pass filter schematic.

$$T(s) = \frac{1}{2RCs + 1} \tag{3-3}$$

$$f_c = \frac{1}{4\pi RC} \tag{3-4}$$

The schematic for a differential first-order passive low-pass filter is displayed in Figure 3-13. It is comprised of two resistors and a bypass ceramic capacitor between the differential signals. Equation 3-3 is the transfer function that models the filter and Equation 3-4 is an expression for finding the cutoff frequency of the filter. Two resistors of 5 kΩ and a capacitor of 3 nF were chosen to yield a cutoff frequency of 5.3 kHz.

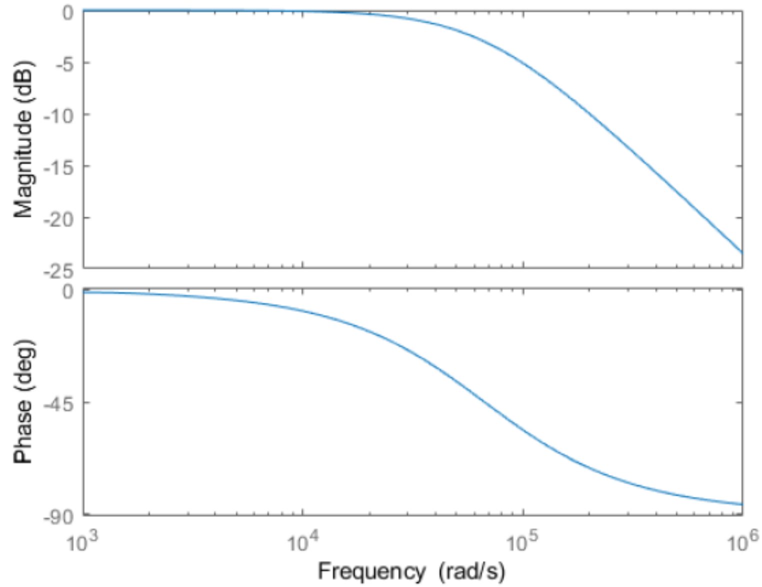


Figure 3-14: Bode Diagram for first-order low-pass filter.

The frequency response of the filter is shown in Figure 3-14. The Bode plot was created in MATLAB™ using the transfer function displayed in Equation 3-3. With the filter design presented above, the filter will leave signals of 1 kHz and lower unaffected and begin to attenuate frequencies above 1.59 kHz at a rate of -20 dB/decade.

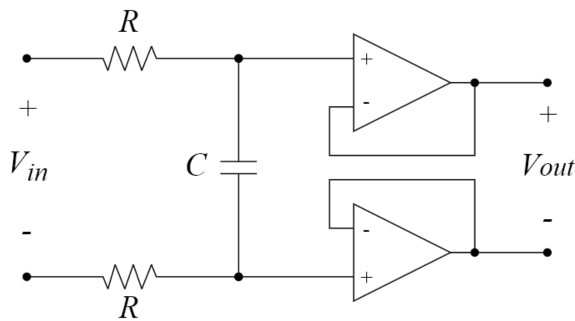


Figure 3-15: First-order active low-pass filter schematic.

Figure 3-15 displays the schematic of a differential first-order active low-pass filter. The only difference between the passive and active filters is the presence of voltage buffers on the outputs, using the LMV321 operation amplifier as a voltage buffer. The transfer function and cutoff frequency are the same for this filter, thus the same resistor and capacitor values may be used to

achieve a cutoff frequency of 5.3 kHz. Using an active filter in this application is useful because the buffers isolate the filter so that the circuit downstream does not affect the filter's actions. Operational amplifiers accomplish this effect due to their high-impedance inputs and low-impedance output.

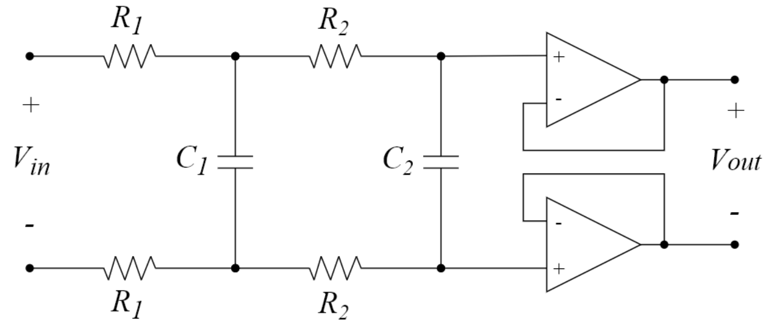


Figure 3-16: Second-order active low-pass filter schematic.

$$T(s) = \frac{\frac{1}{4R_1R_2C_1C_2}}{s^2 + \left(\frac{1}{2R_1C_1} + \frac{1}{2R_2C_2}\right)S + \frac{1}{4R_1R_2C_1C_2}} \tag{3-5}$$

$$f_c = \frac{1}{4\pi\sqrt{R_1R_2C_1C_2}} \tag{3-6}$$

The schematic of a differential second-order active low-pass filter is shown in Figure 3-5. Two first-order low-pass filters are cascaded to create the second-order filter and the voltage buffers provide the active filtering properties. Equation 3-5 is the transfer function that models the filter and Equation 3-6 is an expression for finding the cutoff frequency of the filter. Similar to the first-order filter designs, two resistors of 5 kΩ and two capacitors of 3 nF were utilized again to yield a cutoff frequency of 5.3 kHz for the filter. Using MATLAB™, the frequency response of the filter was generated and is shown in Figure 3-17. The second-order filter behaves similarly to the first-order filter, but with more a more aggressive attenuation rate of -40 dB/decade for frequencies above the cutoff frequency.

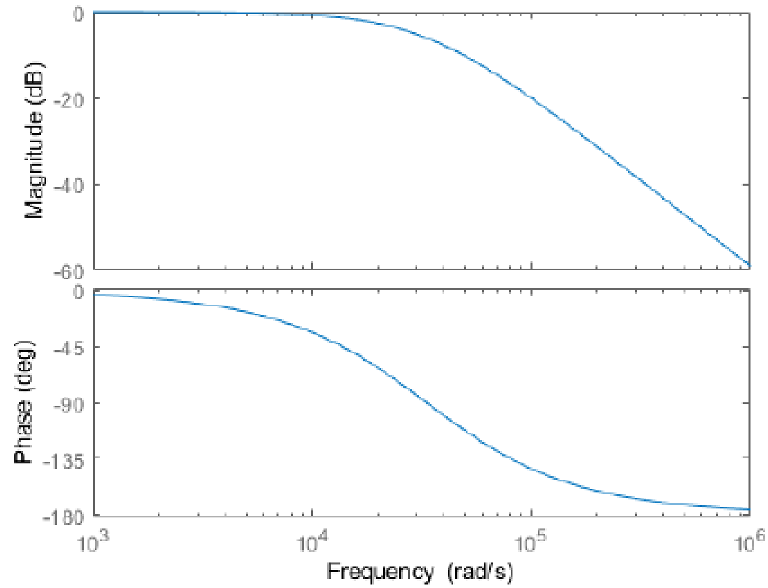


Figure 3-17: Bode Diagram for second-order low-pass filter.

3.1.3.3 Signal Amplification

To amplify the differential signal, a differential amplifier circuit was constructed using resistors and an operational amplifier. The selected design amplifies the differential voltage from the Wheatstone bridge and then offsets the difference by a reference voltage (Electronics Hub, 2019). The differential amplifier circuit is shown in Figure 3-18. Equation 3-7 was used to design the circuit for a gain of 128.

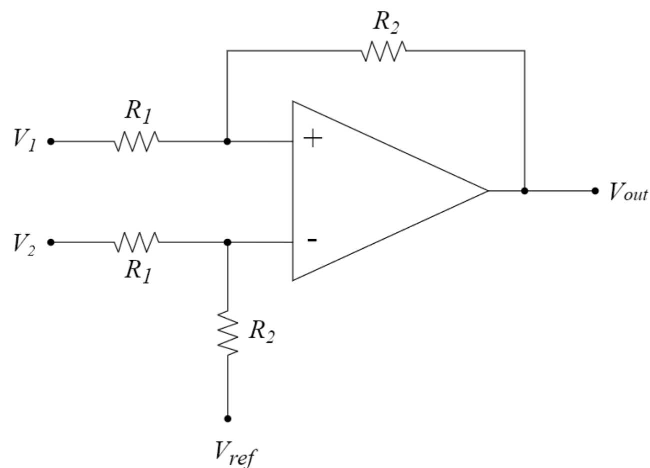


Figure 3-18: Differential amplifier + offset circuit schematic.

$$V_{out} = \frac{R_2}{R_1}(V_2 - V_1) + V_{ref} \tag{3-7}$$

Using Equation 3-7 to design for a gain of 128, R_1 and R_2 were selected to be 1 kΩ and 128 kΩ, respectively. V_{ref} was provided using the circuit displayed in Figure 3-19, a simple voltage divider circuit that halves the +3.0V reference voltage using two resistors and a voltage buffer. The output of the differential amplifier yields a single-ended signal that is offset by +1.5V, mid-range between ground and the +3.0V positive reference voltage of the ADC, and biased by the amplified differential signal. A gain of 128 for the differential signal and offset of +1.5V increases the precision of the measurement, as the ADC may see voltages that better span its range, rather than extremely low, unamplified signals.

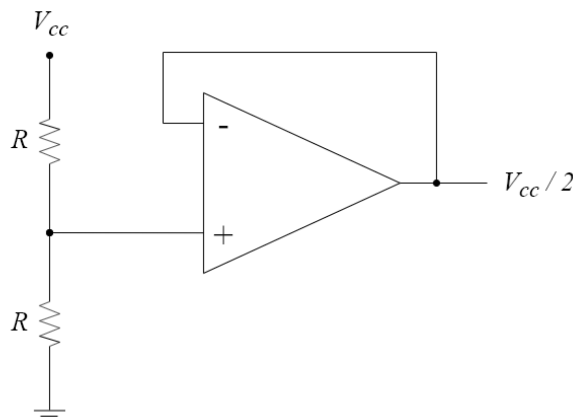


Figure 3-19: Voltage divider circuit for providing a reference voltage of $V_{cc}/2$.

If the voltage divider resistors are the same value, the output of the voltage divider will effectively output half of its input. In this design, resistors of 10 kΩ were selected. Theoretically, any value of resistance could achieve the desired effect, but a relatively high-value resistor was chosen to minimize the current consumption of the circuit.

3.1.4 Software Design

Software was written for the Atxmega128A4U in C/C++ for acquiring and storing strain measurements. Atmel Studio^{TM}, an integrated development platform (IDP) for developing and debugging AVR^{TM} and SAM^{TM} microcontroller applications, was used to develop the firmware and software for the device (MicroChip, 2019). The IDP connects seamlessly to the Atmel-ICE debugger for flashing code to the Atxmega128A4U and provides an interface for debugging the code in real-time (Atmel, 2016). The program running on the Atxmega128A4U needed to accomplish the following: configure the internal and external clocks to keep correct time for timestamping data, store the user-set parameters, acquire and record data to the microSD card through the on-board ADC and the AD7708, convey the operational status of the device to the user through the serial interface, and acquire and report debugging information through the serial port when prompted. Similar to the program written for the quasi-static data acquisition device, the code was split into multiple files based on functionality and the concept of classes and class inheritance were heavily utilized to make the code more readable. Drivers were written in C and C++ to control the on-board ADC, SPI, and UART interfaces, the AD7708, and storing data to a microSD card. Again, the concepts of multi-tasking and finite-state machines were used to further simplify and improve the functionality of the device. The drivers, multi-tasking scheme, and tasks written for the embedded software will be discussed below. A data structure known as a circular buffer is heavily utilized in the programming of this device. A circular buffer is a first-in-first-out (FIFO) data structure that stores data in a fixed-size array (Medium, 2019).

3.1.4.1 Drivers

The Atxmega128A4U provides various peripherals that may be assigned interrupt-service routines (ISR) that trigger when certain operations are complete. Drivers associated with these peripherals provide functions for normal peripheral operation as well as interrupt-driven operation

to improve the multi-tasking abilities of the device. The drivers written for the device are provided in Appendix G.

Serial Port Driver:

The serial port driver consists of two classes: a serial port driver class for interfacing with an on-board UART interface and a stream class which converts data to 8-bit characters and is architecture independent. The stream class, comprised of the files *stream.h* and *stream.cpp*, converts signed, unsigned integers, and floating-point number to a string of 8-bit characters that can then be sent through a peripheral. Functionality for the << operator is also defined in the stream class to improve code readability. The serial port class, comprised of the files *serial_port.h* and *serial_port.cpp*, inherits functions from the stream class and provides functions for sending and receiving data through one of the Atxmega128A4U's UART interfaces. The class constructor and *initialize ()* function assigns low-priority interrupts for transmitting and receiving multiple bytes of data at a time. When a character is received, the interrupt that triggers stores the received character in a circular buffer that may be retrieved using the *get_char ()* function. The *check_buf ()* function returns a boolean indicating if characters exist in the receive circular buffer. Using the << operator, individual characters, data of any individual type, and strings of characters may be sent through the UART port. When transmitting data, the first character is sent through the UART port, while the remaining characters are pushed to a circular buffer. The remaining characters in the buffer are transmitted in the interrupt that triggers when a character transfer is finished.

SPI Driver:

A driver consisting of the files *SPI.h* and *SPI.cpp* was written to abstract the use of the SPI interfaces on the Atxmega128A4U. This class provides functions for transmitting and receiving data with and without the use of interrupts. The class constructor, the *initialize ()* function, and the *set_mode* () function configures and enables the SPI port and associated high-priority interrupts. To transfer and receive a single byte without using interrupts, the *transfer ()* function may be used.

For transferring multiple bytes using interrupts, the *push_byte ()* and *transfer_ISR ()* functions are used for pushing bytes to a circular buffer and beginning SPI transmission. After calling *transfer_ISR ()*, each byte in the buffer is transferred in an ISR and all received bytes are stored in another buffer. The functions *data_ready ()* and *pull_byte ()* check if the transfer is complete and returns one received byte at a time.

MicroSD Driver:

To write data to a microSD card, a microSD driver was built on top of the FatFs filesystem module intended for use on small embedded systems (ChaN, 2019). The FatFs module provides functions for creating, reading, and writing files that is architecture independent and is comprised of the files *ff.h*, *ff.c*, *ffconf.h*, and *diskio.h*. Because the filesystem is architecture independent, the functions declared in *diskio.h* must be written based on the I/O capabilities of the microcontroller. A file named *mmc_avr.c* was written that defines the functions declared in *diskio.h* so the filesystem uses the SPI interface on port D and a timer on port C to interface with the microSD card. The file *mmc_avr.c* heavily borrows code from an example project on GitHub that writes data to an SD card using an Xmega device (sandcoffin, 2016). For further abstraction, a driver class comprised of the files *microSD.h* and *microSD.cpp* was created that uses the FatFs filesystem and inherits functions from the stream class discussed above to write data to a microSD card. The function *initialize ()* configures the hardware and the function *mount ()* mounts the filesystem to the workspace. *card_detect ()* returns a boolean indicating if a card has been inserted and the functions *create ()* and *open ()* creates and opens a file which can then be written. The << operator provided by the stream class may then be used to save data strings to the currently open file. The function *close ()* closes the file and saves all previously written data to the microSD card. Functions for reading data from a microSD card were not implemented in the microSD class because the acquired data is expected to be read using a computer.

ADC Driver:

The ADC interface driver provides functions for using an on-board ADC channel of the Atxmega128A4U and consists of the files *ADC.h* and *ADC.cpp*. Two classes encompass the driver, one for controlling an individual ADC channel and one for controlling the ADC interface as a whole. Similar to the SPI and serial port drivers, high-priority interrupts that trigger when a reading is complete are configured for each ADC channel. The ADC channel class contains functions for tying GPIO pins to the positive and negative sides of the ADC and taking readings. Functions such as *read ()* and *request ()* were written that initiate a conversion and wait until the conversion is complete, and request a reading where the result is stored in a circular buffer in an ISR. In addition, the function *acquire_data ()* uses interrupts for acquiring data at a specified frequency using a timer on port C to improve the multitasking capabilities of the device. Each channel has an associated circular buffer where the data acquired in an ISR is stored. The ADC class contains functions like *initialize ()*, *enable ()*, and *disable ()* for configuring the ADC interface. Four ADC channel objects are instantiated within the class for each channel, and the ADC class is declared a friend of the ADC channel class so that it has access to the private members of each ADC channel object.

AD7708 Driver:

The AD7708 driver uses the SPI driver discussed above to control and acquire data from the AD7708. This class is comprised of the files *AD7708.h* and *AD7708.*cpp. Again, functions are provided for acquiring data without the use of interrupts and functions that request readings which are acquired in an ISR, triggered by the SPI interface and stored in a circular buffer. Similar to the ADC channel class, the function *acquire_data ()* configures another timer ISR on port C that acquires data at a specified frequency. Functions like *initialize ()*, *enable ()*, and *disable ()* are available for configuring the device and *calibrate ()* is used to calibrate the device internally.

3.1.4.2 Multi-tasking

A more sophisticated cooperative multi-tasking scheme than the scheme utilized for the quasi-static device was implemented for this device. A cooperative scheduler class, comprised of the files *coscheduler.h* and *coscheduler.cpp*, was created to run background code that controls which tasks are run and collects debugging information. In addition, a task class, comprised of the files *task.h* and *task.cpp*, was created to serve as a base class that provides derived classes with basic task functionality. This base class stores variables such as the task state, next run time, and the task priority, and declares functions such as *initialize ()* and *run ()* to be defined in a derived class. The scheduler, when instantiated, saves pointers to the derived task classes so that it may call the *run ()* function when appropriate. The scheduler organizes each task pointer into a high-priority, medium-priority, or low-priority queue based on the task priority. In addition, an idle task may be instantiated that runs only when no other tasks require CPU time. Similar to the multi-tasking scheme used for the quasi-static device, the scheduler determines which tasks need to run based on the task run time variable. To keep time, files named *clock_timer_rtc.h* and *clock_timer_rtc.cpp* were written to keep time in milliseconds using a timer on port D and the on-board RTC.

For inter-task communication, classes for shares and queues were written that provide ways for sending data between tasks. These classes are comprised of the files *share.h* and *queue.h*. These classes were written as templates so that shares and queues of any individual data type may be instantiated. A queue is a FIFO data structure commonly used for sending a stream of data from one process to another. In this multi-tasking scheme, tasks may store pointers to the relevant shares and queues to communicate with other tasks. The files written for the scheduler, base task, shares, and queues are provided in Appendix H.

3.1.4.3 Tasks

The cooperative scheduler runs five finite-state machine tasks that control each major component of the device. Each task priority, timing, and the inter-task communication is detailed

in Figure 3-20. The solid lines indicate that the inter-task communication scheme is a queue, while the dashed lines indicate that the inter-task communication scheme is a share.

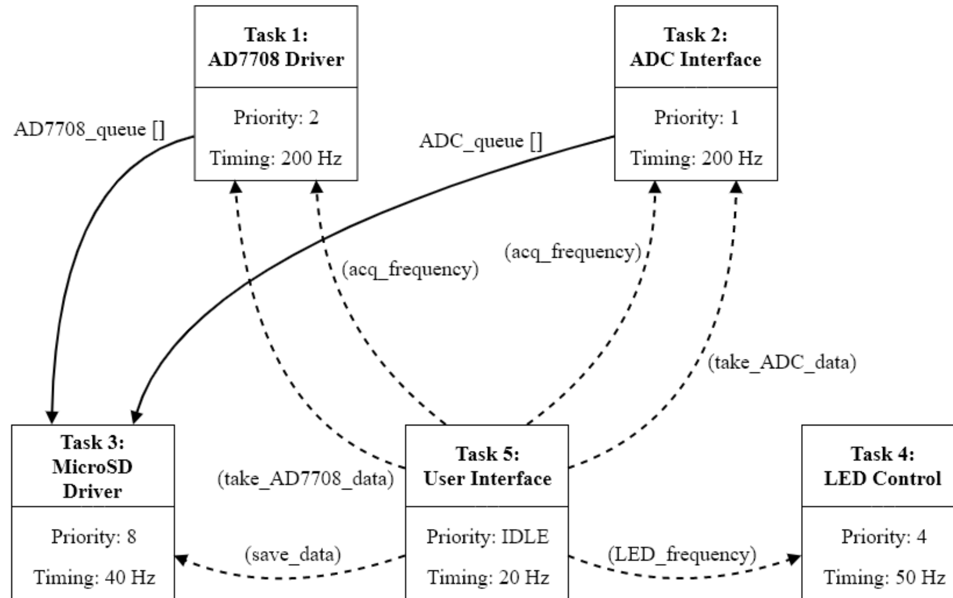


Figure 3-20: Task diagram for dynamic data acquisition system.

Tasks 1 and 2 control the AD7708 and the ADC interface of the Atxmega128A4U and send data to task 3 which controls and saves data to the microSD card. Task 5 uses the serial port to update the user on the status of the device and to interpret commands entered by the user. Task 4 is a simple task that only toggles an on-board LED every 500 ms to indicate that the program is running. The finite-state machine of tasks 1, 2, 3, and 5 will be discussed below. The derived task files are provided in Appendix I.

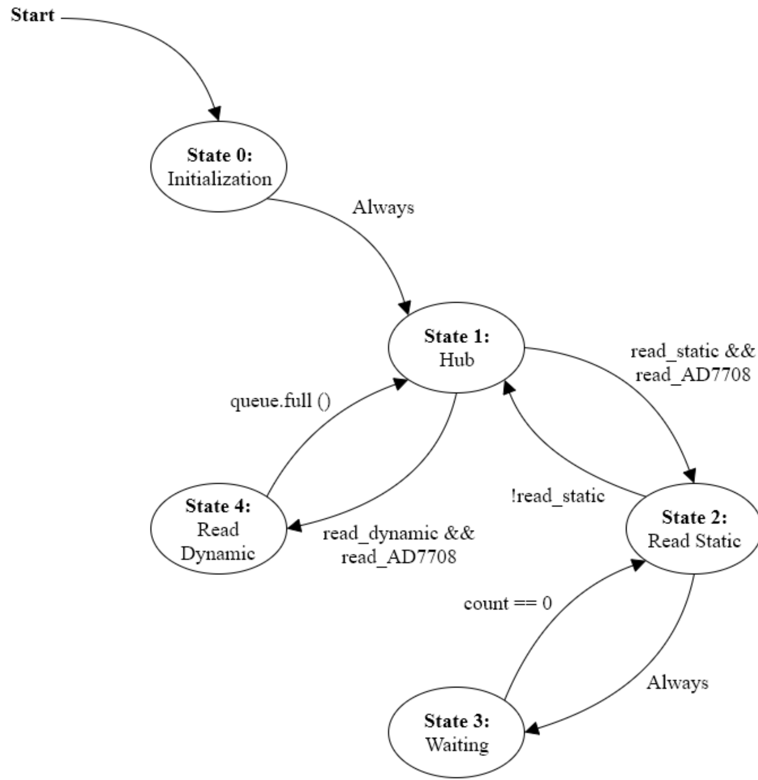


Figure 3-21: AD7708 Driver task state transition diagrams.

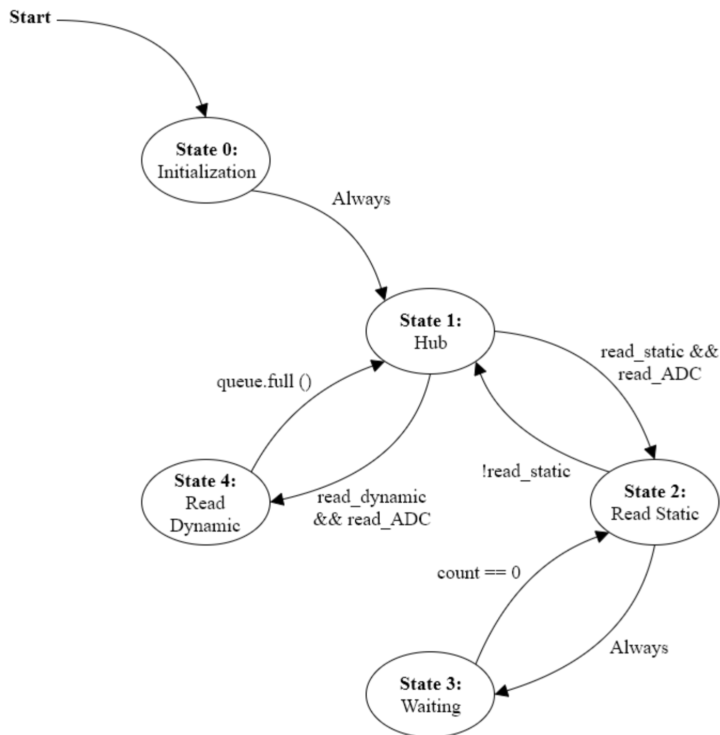


Figure 3-22: ADC Interface task state transition diagrams.

The AD7708 Driver and ADC Interface tasks have very similar state machine designs, displayed in Figures 3-21 and 3-22, respectively. In state 0, the corresponding interface is initialized and the task transitions to state 1, where it waits until a flag is raised by the User Interface task. The task either transitions to state 2 to take static data, or state 4 to take dynamic data. While in state 4, the data is collected using the interrupts associated with each device, and the task is only pulling the acquired data from the buffers and pushing data to the corresponding queue. When the queue is full, the task transitions back to state 1 to wait for another flag to be raised. In state 2, the task takes one reading before transitioning to state 3, where it waits until one second has elapsed, where it transitions back to state 2 to take another reading. When the user interface lowers the flag, the task transitions back to state 1 to wait for another flag to be raised.

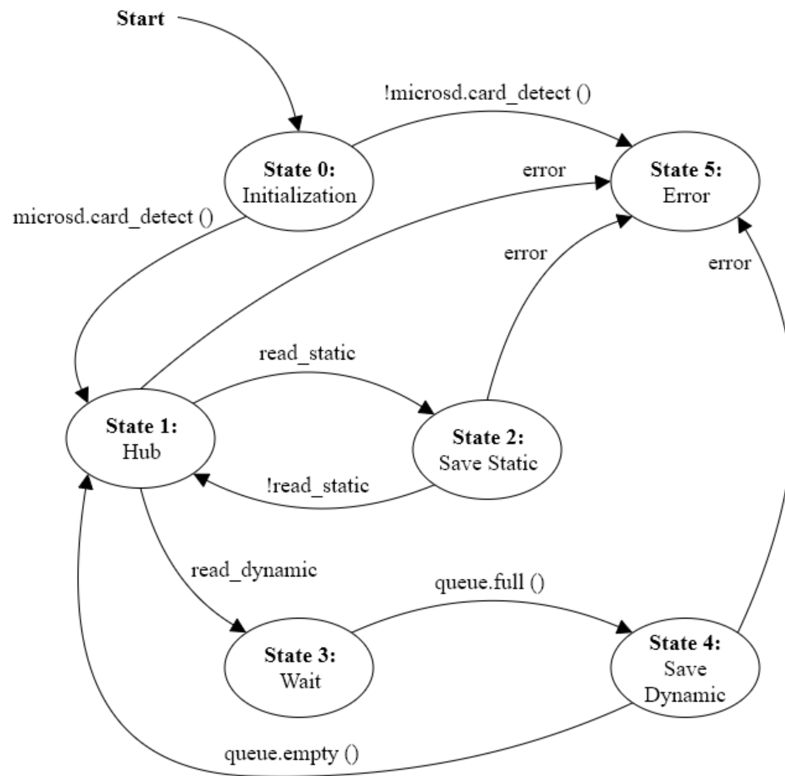


Figure 3-23: MicroSD Driver task state transition diagram.

The MicroSD Driver task, whose state machine is displayed in Figure 3-23, is responsible for storing any data received from the AD7708 Driver or ADC Interface tasks. The task first checks

if a microSD card is inserted. If a card is detected, the task configures and initializes the microSD card and transitions to state 1 where it waits for a flag to be raised. The task will enter state 2 if either task begins taking static data, in which the task stores each static reading to the microSD card and will continue until the flag is lowered. The task enters state 3 when the device begins taking dynamic data, where it waits until the queue is full. When the queue is full, the task transitions to state 4 and pulls and saves the entire queue of data before transitioning back to state 1. State 5 exists for handling any errors, mainly for if the device loses its connection with the microSD card.

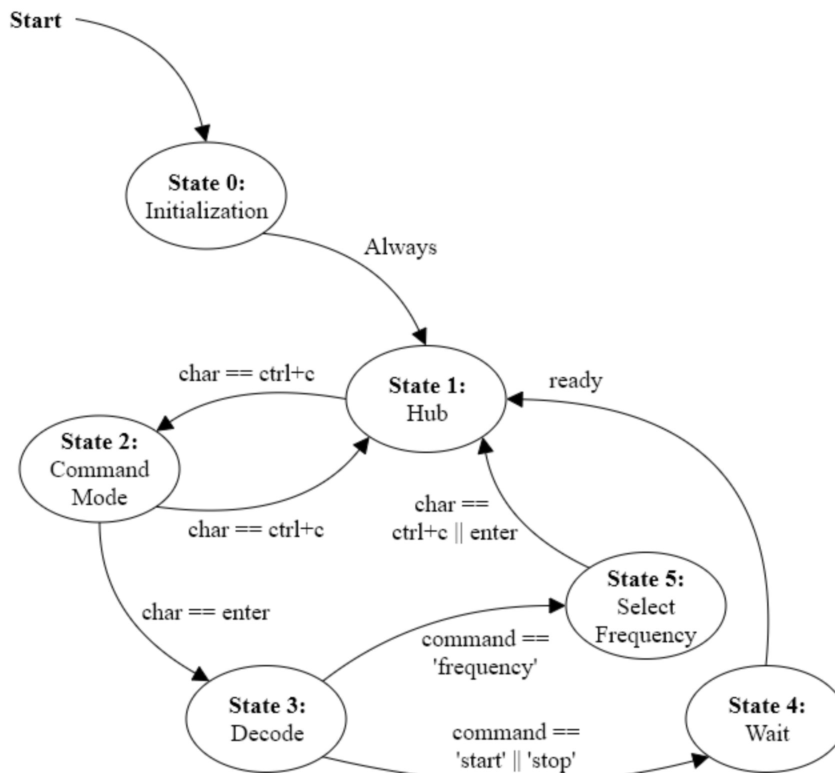


Figure 3-24: User Interface task state transition diagram.

The user interface task, whose state machine is displayed in Figure 3-24, is responsible for handling any commands from the user and for communicating the device status over the serial port. State 0 does nothing but ask the user to enter "ctrl+c" to enter command mode. Once the user does this, the task enters state 2 where the user is prompted to type a command through the serial port.

The task stores characters in a string and echoes each character through the serial port. Once the user presses "enter", the task enters state 3 where the command string is decoded and the command the user entered is determined. The task raises the appropriate flags to execute the command and transitions to state 4, where the task waits until the device is ready to accept another command. If the user enters the acquisition frequency command, the task enters state 5 where the user is prompted to enter a frequency at which to acquire dynamic data.

3.2 Device Testing and Validation

The dynamic data acquisition system was tested extensively with the procedures used to test the quasi-static device to verify that it met the requirements presented in Section 1.2. The current draw test was conducted to determine the power consumption of the device at various operating conditions. Static data was collected at room temperature to compare the results with theory and to calibrate each method of reading mechanical strain. Data was also taken in a low temperature chamber to confirm the survivability of the device and to observe the effect of temperature on each Wheatstone bridge configuration. To confirm the device is able to take dynamic measurements, data was acquired at a rate of 1 kHz using the two different ADCs and compared with theory.

Before conducting the tests discussed above, the device functionality was tested to verify that the overall performance of the device was as expected according to the design of the device. The only component that did not perform as expected was the MOSFET used to toggle the excitation voltage for the Wheatstone bridges. The drain-to-source voltage drop on the MOSFET was greater than expected, and altered the reference voltage such that using the component altered the offset in the data. To mitigate this problem, the BSS123L MOSFETs were removed from the board and the connection shorted so that the 3.0 V reference is directly connected to the Wheatstone

bridges. The downside of removing the MOSFETs is that each complete Wheatstone bridge draws about 8mA of current at all times, rather than only when the MOSFETs are active.

To test the quarter, half, and full-bridge configurations for acquiring static and dynamic data, three beams were designed and made out of aluminum bar stock with dimensions of 3.175 mm x 19.05 mm x 227 mm. The length of the beam was designed such that the natural frequency of each beam is 50 Hz in a cantilever configuration, which will be discussed below. Constantan strain gauges were mounted to each of the three beams for quarter, half and full-bridge configurations, shown in Figure 3-25. Additionally, small holes were drilled at the end of each beam for suspending masses during static strain testing.

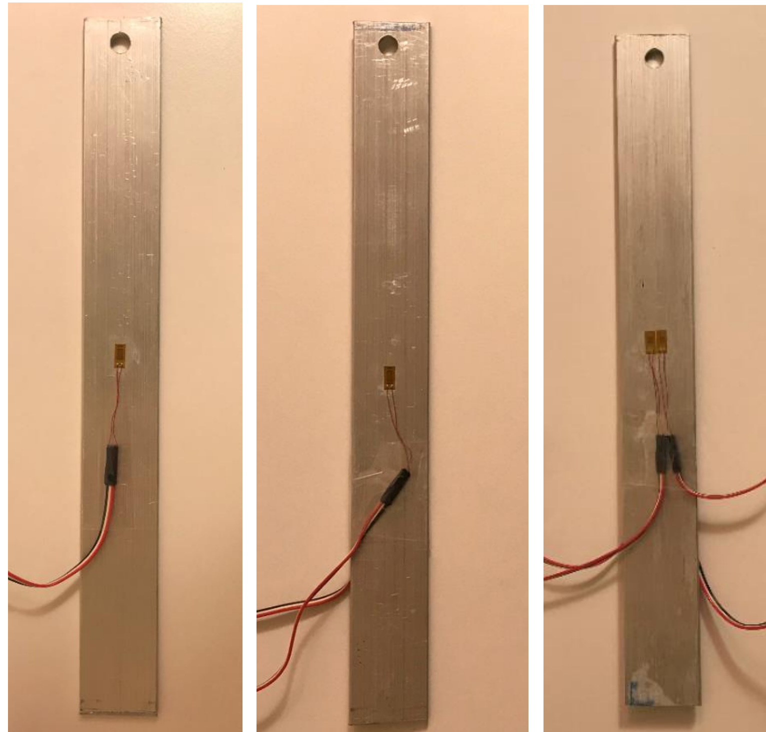


Figure 3-25: Quarter-bridge (left), half-bridge (middle), and full-bridge (right) beam configurations.

3.2.1 Current Draw

To determine the current draw of the device under various operating conditions, the same modified USB cable and digital multimeter setup was used in series with the device and a USB port

on a computer. When testing the current draw while the device is acquiring data, only the Wheatstone bridge being used to measure strain was connected. Both branches of all unused bridges were left incomplete during the current tests so that the device did not draw unnecessary current. The current draw and power consumption of the device under various operating conditions are provided in Table 3-2.

Table 3-2: Current draw and power consumption of device under various operating conditions.

Device Condition	Current Draw (mA)	Power Consumed (W)
Waiting	35.5	0.178
Acquiring Data (16-bit AD7708)	42.9	0.215
Acquiring Data (On-board 12-bit ADC)	47.1	0.236
Saving Data	51.3	0.257

While the device is not reading or recording data, the two components with the most significant current draw are the Atxmega128A4U and the FT232R, which draw approximately 9.5 mA and 15 mA, respectively (Atmel, 2014) (FTDI, n.d.). The current draw of the device could be reduced further by placing these components in a low-power state. The rest of the current draw is the natural current draw of the various components that are always active. The AD7708 pulls approximately 1.3 mA while active, and each Wheatstone bridge has a natural current draw of 8 mA when excited because of the 350 Ω resistors. Additionally, while only about 17 mA were observed during this test, microSD cards have non-continuous current draws of up to 100 mA (SanDisk, 2010).

3.2.2 Static Data Acquisition Testing

Both methods of reading strain were tested at room temperature to acquire static data. In both cases, the device stores acquired data in the form of an unsigned, 16-bit number. To process the data faster, the device stores the data to the microSD card as is, and may be converted to volts

and then strain after off-loading the data. Sample sets of data acquired at 1Hz by both methods are shown in Table 3-3.

Table 3-3: Sample sets of data acquired by both methods.

Runtime (ms)	Raw Data	
	On-board ADC (12-bit)	AD7708 (16-bit)
0	1004	31869
1000	1004	31894
2000	1004	31919
3000	1003	31869
4000	1006	31794
5000	1009	31819
6000	1007	31869
7000	1006	31869
8000	1009	31869
9000	1005	31794

The ADC data has a linear correlation between binary and volts, and may be converted by multiplying the number by the ratio of the reference voltage (3.0 V) and 4096. After subtracting the offset from the data, divide by 128 to account for the differential amplifier gain. The data is then converted from volts to μ-strain using Equations 2-2, 3-1, and 3-2. The AD7708 can acquire data at a maximum frequency of 1.3kHz with chopping disabled, but must be calibrated externally rather than using the internal calibration capability (Analog Devices, 2001). To acquire dynamic data with chopping disabled, each AD7708 channel must be calibrated before converting the data to strain.

A beam bending test was conducted to verify the data from the on-board ADC and to calibrate the AD7708 using the same methods and theory as discussed in Section 2.2.2. Each channel of the AD7708 was calibrated using the quarter, half, and full-bridge beams by taking readings and comparing them with theory. The quarter-bridge, half-bridge, and full-bridge Wheatstone bridge configurations correspond to channels 1, 2, and 3 on the AD7708, respectively. To calibrate each channel, readings were taken when the beams were unloaded to determine the offset. Then, various known masses were suspended at the end of the beam and data was acquired.

Theory was used to calculate the value of strain corresponding to each mass, and a linear correlation was made between the readings acquired by the AD7708 and the theoretical strain values. The half-bridge beam was used to acquire data using the three channels of the on-board ADC of the Atxmega128A4U. Channels 1, 2, and 3 on the Atxmega128A4U correspond to the data filtered by the first-order passive, first-order active, and second-order active low-pass electronic filters, respectively. The results of the static beam tests are displayed in Figure 3-26 and Figure 3-27 for each channel of the AD7708 and each channel of the on-board ADC, respectively.

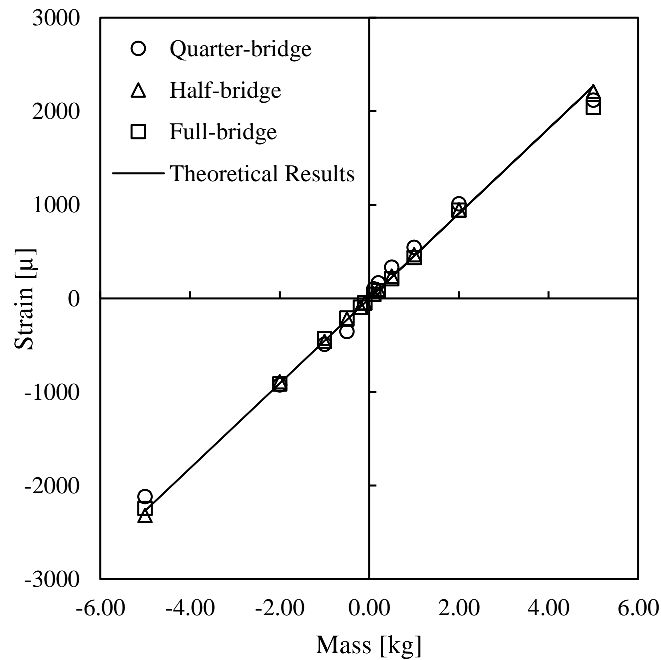


Figure 3-26: Calibration of the AD7708 (16-bit).

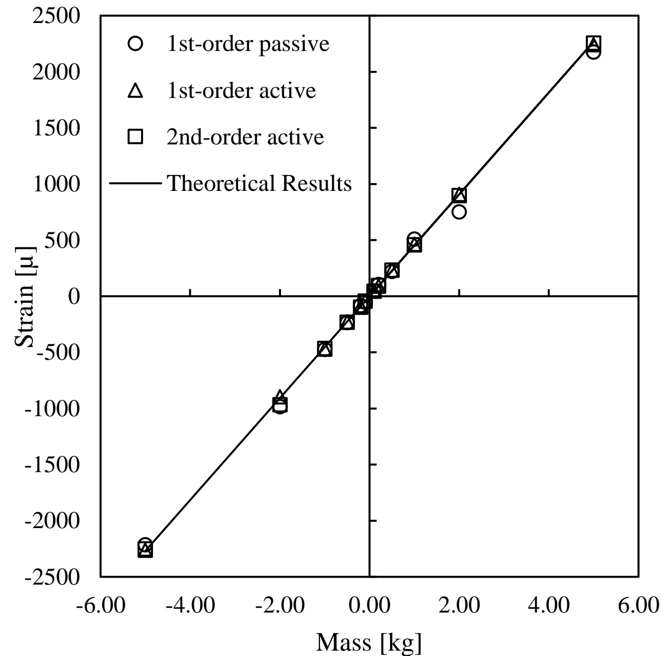


Figure 3-27: Comparison of on-board ADC (12-bit) readings to theoretical strain values.

Similar to the static beam tests of the previous device, masses of 0.1, 0.2, 0.5, 1.0, 2.0, and 5.0 kg were suspended at the end of the beam to produce the results. The negative masses in the plots refer to readings that were acquired when the beam was inverted (i.e., strain gauges in compression). Both methods of reading strain provided linear results which only deviated slightly for the larger masses. The maximum theoretical bending stress experienced by the beam was approximately 156 MPa, which indicates that the beam was never strained beyond the published elastic limit of 276 MPa for 6061 aluminum (ASM, 2018). Like the test conducted for the HX711, a lack of precise beam dimensions and elastic modulus are likely causes for the slight deviations in the measurements.

3.2.3 Dynamic Data Acquisition Testing

Dynamic testing was conducted in which the device acquired data at a frequency of 1 kHz on the same beams used for the static tests. The beams were mounted on the same testing setup used for the static tests, deflected, and released so they were vibrating freely. During this vibration, the device acquired and recorded a thousand data points for one second to verify that it could

capture the first mode of oscillation at the correct acquisition frequency. To verify the device recorded data at the correct frequency, the frequency of the oscillations in the acquired data was compared to the theoretical first natural frequency of the beams.

To determine the natural frequency of the beams, the Assumed Modes method was used to derive an equation of motion (EOM) for the transverse vibration and an expression for calculating the natural frequency of the cantilever beam (Craig et al. 2006). The schematic depicted in Figure 3-28 was used to derive the EOM and an expression for the natural frequency.

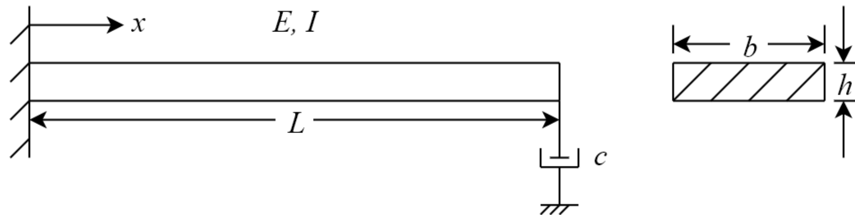


Figure 3-28: Dynamic cantilever beam model schematic.

To ensure accuracy and simplicity of the model, the Assumed Modes method was used to derive an EOM that described the first two modes of oscillation. Each mode was assumed to be of the shapes shown in Figure 3-29 and described by Equations 3-8 and 3-9. The mode shapes were chosen based on the boundary conditions of a cantilever beam.

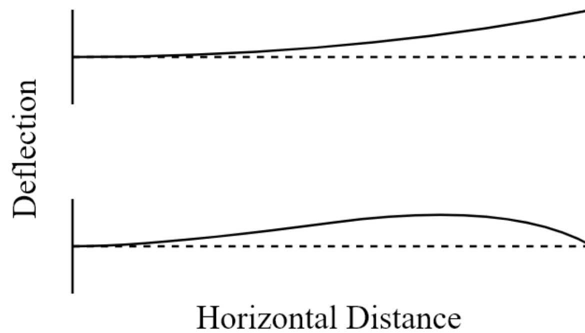


Figure 3-29: Assumed mode shapes of cantilever beam model for (top) first mode and (bottom) second mode.

$$\psi_1(x) = \left(\frac{x}{L}\right)^2 \tag{3-8}$$

$$\psi_2(x) = \left(\frac{x}{L}\right)^2 - \left(\frac{x}{L}\right)^3 \tag{3-9}$$

Given the mode shapes and the geometry of the beam, the EOM described in Equations 3-10 and 3-11 were derived using the Assumed Modes method. Equation 3-10 displays the coupled EOMs of each of the modes of oscillation varying with time while Equation 3-11 depicts the transverse displacement of the beam varying with time and the distance along the beam.

$$M\ddot{Q} + C\dot{Q} + KQ = 0 \quad Q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

$$M = \rho AL \begin{bmatrix} 1/5 & 1/30 \\ 1/30 & 1/105 \end{bmatrix}, \; C = c\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \; K = \frac{EI}{L^3}\begin{bmatrix} 4 & -2 \\ -2 & 4 \end{bmatrix} \tag{3-10}$$

$$v(x,t) = \psi_1(x)\, q_1(t) + \psi_2(x)\, q_2(t) \tag{3-11}$$

To derive the first natural frequency, the characteristic equation of the system, shown in Equation 3-12, was solved and yielded expressions for the first and second natural frequencies, the first of which is shown in Equation 3-13. The roots of the characteristic equation correspond to the eigenvalues of the system, the square roots of which yield the natural frequencies of the system.

$$|K - \omega^2 M| = 0 \tag{3-12}$$

$$\omega_{n_1} = 3.53\sqrt{\frac{EI}{\rho AL^4}} \tag{3-13}$$

Equation 3-13 was used to design the length of the beam, 227 mm, for a natural frequency of about 50 Hz. The derivation of the EOM and Equation 3-13 can be found in Appendix J.

Each method of reading strain was subject to three dynamic data acquisition tests. Each set of data corresponds to an ADC channel for one of the two methods. To induce oscillation at the natural frequency, the end of the beams used to conduct each test were deflected and released. Figure 3-30, Figure 3-31, and Figure 3-32 display the data acquired by the AD7708, and Figure 3-

33, Figure 3-34, and Figure 3-35 display the data acquired by the on-board ADC of the Atxmega128A4U. Each set of data is displayed in a plot showing the total dynamic response and a plot showing a snapshot of 50 ms of data. The magnitude of the oscillations observed in the data was not easily controllable with the setup used in the dynamic tests, and therefore varies between tests.

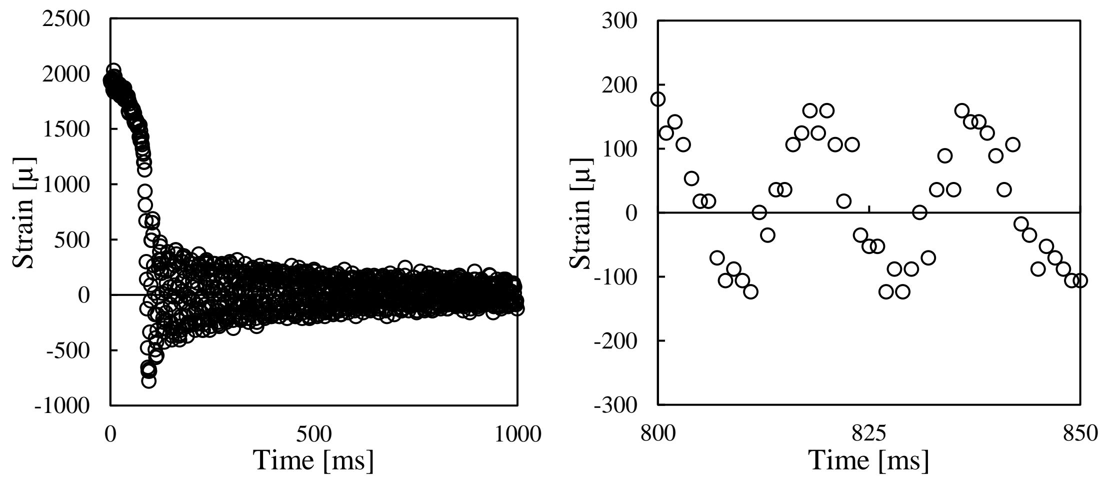


Figure 3-30: AD7708 (16-bit) dynamic beam testing results for the quarter-bridge configuration.

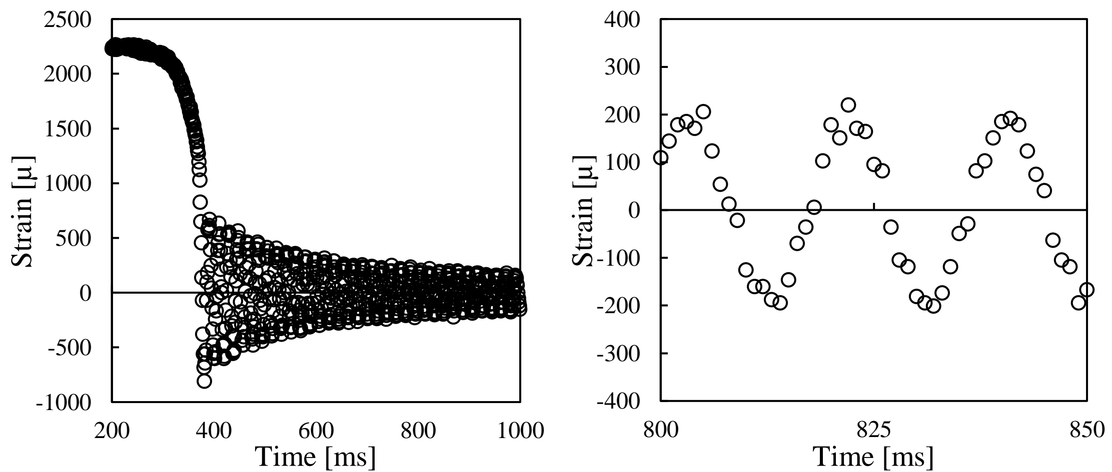


Figure 3-31: AD7708 (16-bit) dynamic beam testing results for the half-bridge configuration.

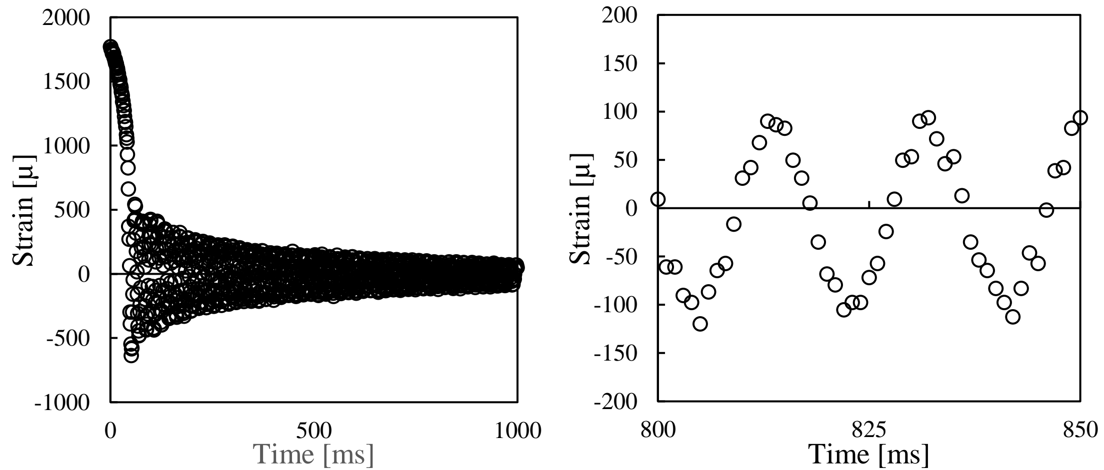


Figure 3-32: AD7708 (16-bit) dynamic beam testing results for the full-bridge configuration.

The results for the dynamic test using the quarter-bridge configuration have more noise than the results for the half-bridge and full-bridge configurations. This noise could be due to parasitic noise between the various components of the Wheatstone bridge and digital switching noise arising from the digital ICs in other areas of the PCB. One reason the half-bridge and full-bridge configurations performed better is that the noise injected into the excitation node of the Wheatstone bridge is subtracted between the differential output of the bridge. The noise in the data could also be caused by higher-frequency oscillations due to the higher modes of oscillation of the beams. As seen in the dynamic results, there are clear gaps in the data, which may be due to the quantization error of the AD7708. In a real ADC, small but finite gaps may exist between one digital number and a consecutive digital number depending on the smallest differential voltage that the ADC can resolve (Measurement Computing, n.d.). The AD7708 is no exception, and appears to have missing codes which may be due to non-linearities present in the device, based on the observed data. However, the missing codes could also be attributed to noise, and more testing would need to be performed to determine if the quantization errors are repeatable.

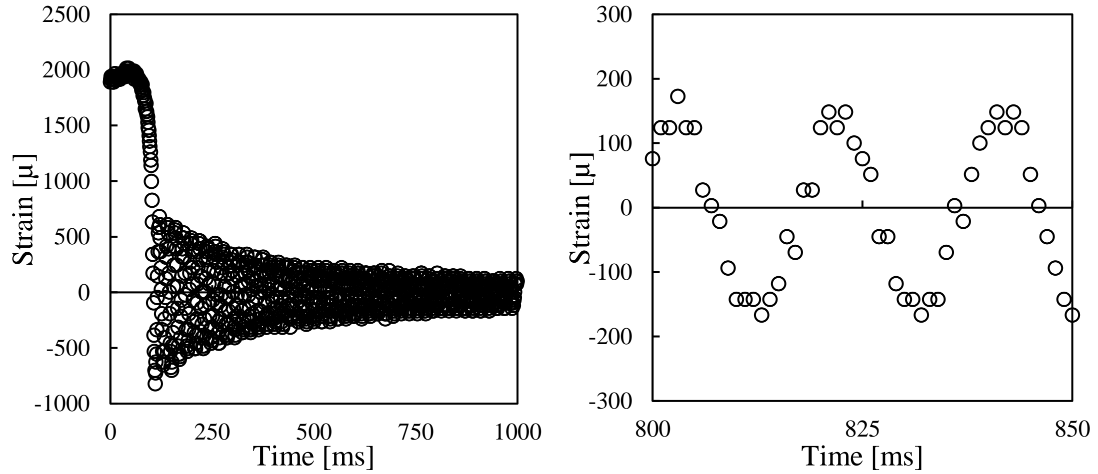


Figure 3-33: On-board ADC (12-bit) dynamic testing results with passive 1st-order filtering.

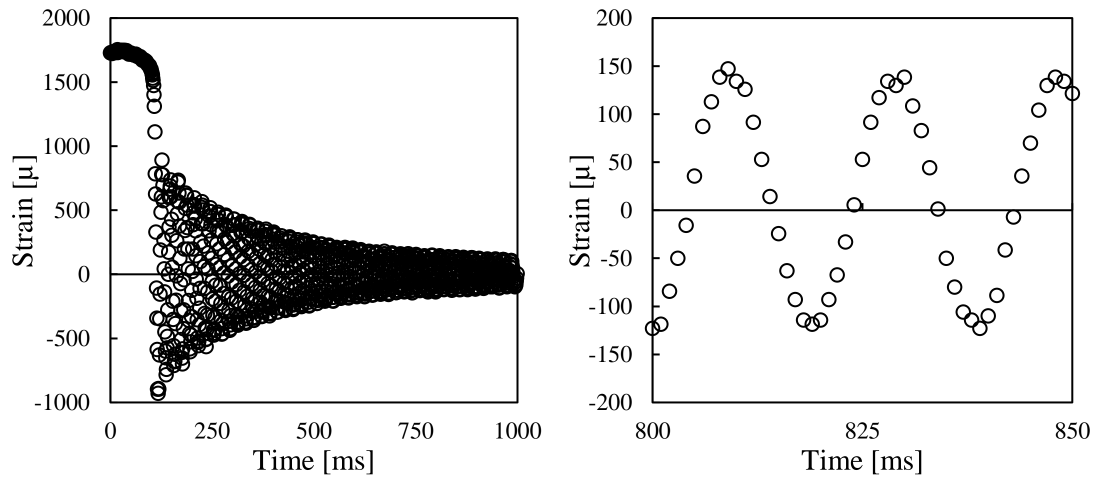


Figure 3-34: On-board ADC (12-bit) dynamic testing with active 1st-order filtering.

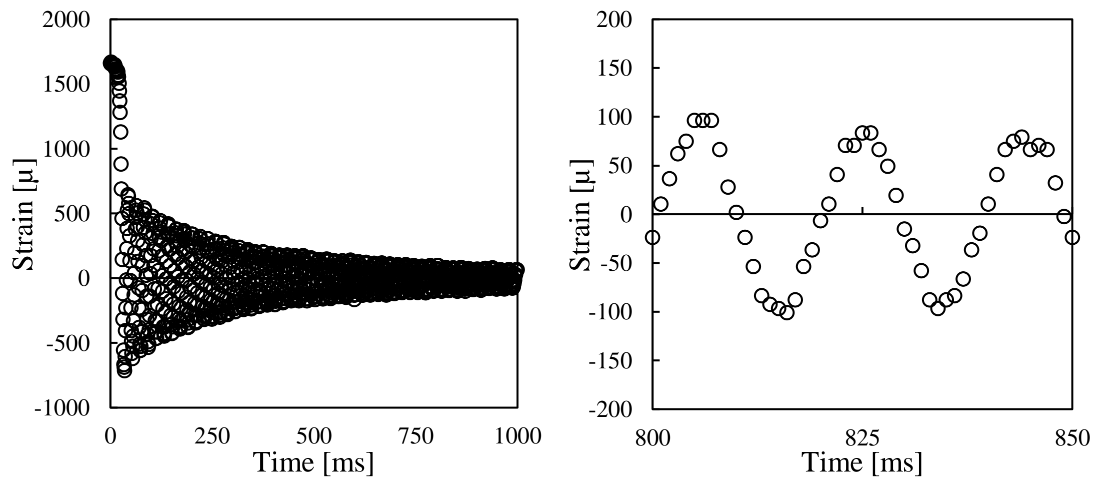


Figure 3-35: On-board ADC (12-bit) dynamic beam testing with active 2nd-order filtering.

Figures 3-33, 3-34, and 3-35 display the dynamic results acquired by the on-board ADC using a half-bridge configuration. It is clearly observable that the active filters attenuate noise much more effectively than the passive filter. Similar to the data acquired by the AD7708, the noise present in the data could be due to oscillations lower than 5 kHz caused by higher modes of oscillation of the beams that are not attenuated by the filters. Again, the on-board ADC is not ideal, and has errors that could be caused by non-linearities in the data although not as apparent as the results acquired by the AD7708.

Table 3-4: Natural frequencies observed in dynamic testing.

Method	Device Channel	Oscillation Frequency (Hz)
AD7708	1	52.6
	2	52.1
	3	51.0
Atxmega128A4U ADC	1	51.3
	2	51.5
	3	52.7

Table 3-4 details the average oscillation frequency observed in the dynamic testing results for both methods of reading strain. The beams were designed for a first natural frequency of 50 Hz, very similar to the frequencies observed during testing. The observed natural frequencies of the beams are slightly different than the theoretical natural frequency, and is most likely due to the actual parameters of the beam being slightly different than those used to calculate the theoretical natural frequency.

3.2.4 Low Temperature Testing

The device was also subjected to low temperature tests to study the survivability of the device at low temperatures and to observe the effect of temperature on the data acquired by each method of reading strain. To lower the temperature, the device was placed in a freezer along with two testing beams under no load where the temperature reached a minimum of -36 °C. Data was acquired with both ADCs using a half-bridge and full-bridge configuration with the on-board ADC

and the AD7708, respectively. Figure 3-36 and Figure 3-37 displays the low-temperature data acquired by the on-board ADC and the AD7708, respectively.

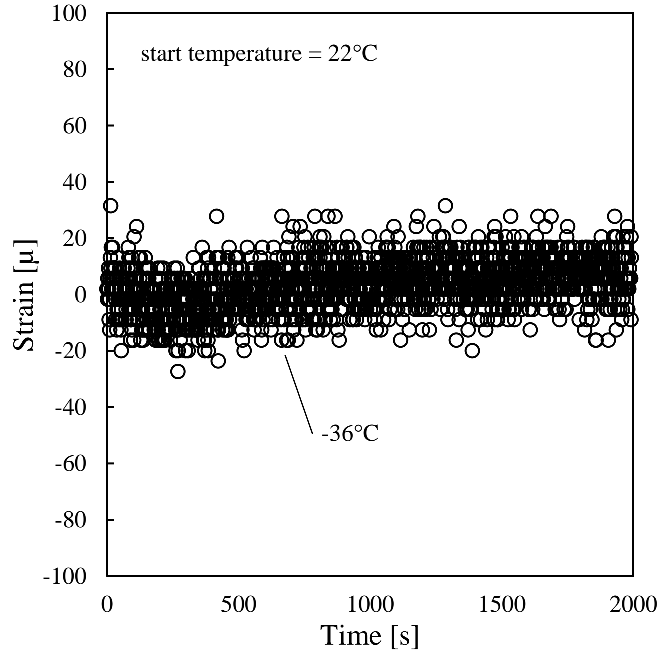


Figure 3-36: Full-bridge low temperature test results acquired by the AD7708 (16-bit).

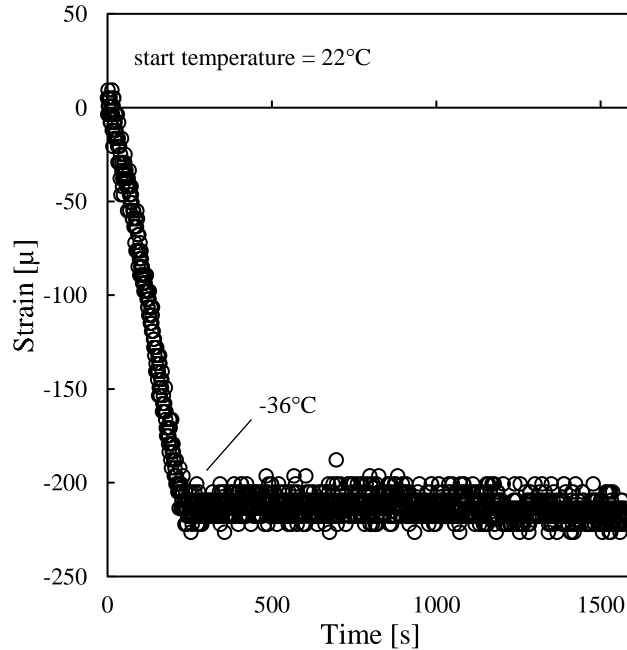


Figure 3-37: On-board ADC (12-bit) half-bridge low temperature test results.

The data acquired by the AD7708 with the full-bridge configuration, shown in Figure 3-36, shows about 7 μ of drift. This is due to the strain gauges all having the same nominal temperature coefficient of resistivity, described by Equation 2-7. The thermal shrinkage of the beam and change in gauge resistance due to temperature has no theoretical effect on the strain reading. Even though the bridge resistances change with the thermal shrinkage of the beam and temperature, each gauge resistance changes at the same rate, so the resistance values remain equal so the bridge voltage is balanced.

The data acquired by the on-board ADC with the half-bridge configuration has an observable drift of about -210 μ. The components comprising each branch of the half-bridge also have the same temperature coefficient of resistivity, and thus contributes nothing to the observed drift. The most likely cause of this drift is due to the LMV321 operational amplifiers used in the differential amplifiers to amplify the voltage have an input offset of about 2 μV/°C (STMicroelectronics, 2015). The device experienced a temperature change of 57 °C between room temperature and the minimum temperature of the freezer, making the input offset voltage drift about 0.114 mV. However, this drift would only account for about -110 μ of the drift seen in the test. Although the beam is in compression due to the low temperature, the strain due to the thermal shrinkage of the beam would not be observable in the measurements. Because the half-bridge configuration was used in this test, both strain gauges are in compression and theoretically should still have the same resistance.

3.3 Summary – Dynamic Data Acquisition System

The design and development of a benchtop proof-of-concept device intended for testing various methods of acquiring dynamic strain measurements was presented. The device is powered and controlled through a USB port and a serial interface on a computer. Data acquired by the device is stored on an on-board microSD card and may be monitored in real-time using an external serial

monitor. The components selected for this device were tested using a custom PCB with designs that may be used in future BLDS designs for acquiring dynamic strain measurements. Multi-tasking software was written in C/C++ to control the device which runs multiple finite-state machine style processes that utilize custom drivers to control the components of the device.

The device performance and power consumption were tested extensively. A current draw test was conducted to verify that the power consumption was within the requirements detailed in Section 1.2. The reliability of the device was tested extensively through multiple beam bending tests in which the device acquired static and dynamic data. The static testing proved that the AD7708 outputs linear data that was used to find a linear correlation between the output and the theoretical strain calculations. The static beam testing of the on-board ADC was compared with theory to validate the measurements. Both ADCs were used to acquire dynamic data on the beams to verify that the device was capable of capturing the first mode of oscillation of the beam. Additionally, the device was tested in a low temperature environment to determine the survivability of the device and the effect that temperature has on the measurements using the half-bridge and full-bridge configurations.

4. CONCLUSIONS AND RECOMMENDATIONS

This thesis details the design and development of two prototype devices to support the development of the family of devices collectively called the Boundary Layer Data System. The new devices are the first members of the BLDS family with the ability to acquire non-flow data – in this case, timestamped mechanical strain measurements – and save the data to an on-board microSD card. These new devices were designed with a modular approach to be the newest members of the next-generation BLDS family of devices to be known as the Flight Test Data System (FTDS). The conclusions found during the work outlined in this thesis for each device and recommendations for future work on strain data acquisition applications will be discussed below.

4.1 Conclusions

4.1.1 Quasi-Static Data Acquisition System

The quasi-static device met the requirements listed in Table 1-3. The device survived and acquired reliable data at room temperature and a temperature of -36 °C and successfully acquired data at a frequency of 10 Hz. During the low temperature tests, the device proved to have a battery life of more than two hours, with a current draw of less than 60 mA. Additionally, the device is small, autonomous and weighs less than a pound. The HX711 demonstrated the ability to output linear, 24-bit data and proved to be a reliable, accurate method for acquiring static strain measurements. To operate, the component requires few components with no external clock, making it a low-cost and easy solution for acquiring quasi-static strain data. The low temperature testing revealed that the strain readings drift as the temperature changes. Possible causes of the drift observed in the strain measurements include: thermal shrinkage of the beam, resistance changes in the bridge components, and offsets generated by the electronics due to operating at low temperatures. The ADXL345 accelerometer was shown to have a high degree of functionality and

provided reliable acceleration data in three directions. Decoupling capacitors are the only external components required, making the accelerometer easy and cheap to implement.

4.1.2 Dynamic Data Acquisition System

The dynamic data acquisition system provided a compact and robust solution for acquiring both static and dynamic strain measurements, while meeting the requirements displayed in Table 1-3. The device proved to survive and successfully acquire data at both room temperature and - 36 °C. Under all modes of operation, the current draw of the device was less than 60 mA. A PCB was designed with a small and practical layout with an overall volume of 3.75 in x 3.8 in x 0.3 in. The AD7708 was proven to provide reliable, 16-bit data at a frequency of 1 kHz. External components such as decoupling capacitors and a 32 kHz clock are required for the device to function. Additionally, a solution using various low-pass filters, differential amplifiers, and the on-board ADC of the microcontroller was demonstrated to acquire 12-bit data at a frequency of 1 kHz. Both ADCs were proven to reliably acquire data at 1 kHz and capture the first mode of oscillation of a beam designed for a first natural frequency of 50 Hz.

4.1.3 Software

Both data acquisition systems use a multi-tasking scheme to run multiple processes simultaneously. The static data acquisition system uses a cooperative, round-robin technique that runs finite-state machines to execute code. The dynamic data acquisition system uses a cooperative technique similar to the static data acquisition system, but with priority so that the higher-priority state machines are given more CPU time than the lower-priority state machines. Both programs use object-oriented programming and modular design to simplify the programming design and troubleshooting process. The static data acquisition system software was programmed using the Arduino IDE which doesn’t have a debug interface. Thus, printing data to the serial port was the only way to monitor and troubleshoot the program. The dynamic data acquisition system program

was programmed in C/C++ using Atmel Studio™ and the Atmel ICE Debugger to flash code and debug the device.

4.2 Recommendations for Future Work

Below are the recommendations for future work on BLDS-FTDS devices that acquire static and dynamic strain measurements.

4.2.1 Programming IDE

The Arduino IDE provided a simple method for programming the quasi-static device. Many open-source drivers and example code written using the Arduino IDE are available, making it a powerful tool that novice developers may use to jumpstart projects. However, the Arduino IDE does not have a debugger, and the serial port must be used to communicate the device status while troubleshooting. For future BLDS software that requires the use of a preemptive operating system or generous use of interrupts, a C/C++ IDE that includes a debug interface should be used. The Atmel ICE Debugger used with Atmel Studio™ provides an effective means of debugging complex, interrupt-driven software.

4.2.2 Electronics and Circuitry

To further reduce the power consumption of the device, each component should be placed in low power or sleep mode during idle operation. In a typical flight test, the device will be waiting for a user-specified time interval at which to acquire data, during which the device could be placed in a low-power state. Each component tested in this work has a low power mode of operation that may be configured in software.

While the methods used to acquire dynamic strain measurements were successful, there was a fair amount of noise present in the data. The power conditioning scheme used to regulate voltage from batteries injects more digital switching noise into the system. Additionally, the SOC chip used to communicate with other modules in the BLDS-FTDS family will inject more high-

frequency RF noise that could further contaminate the measurements. Disciplined PCB design techniques may be used to combat this. One such technique is to further separate the digital and analog sections of the board by implementing separate ground planes that meet near the power supply. An easy way to accomplish this is to use the “star” ground philosophy, where the board is designed such that the interaction between the digital and analog signal paths is minimized (Zumbahlen, 2012). This will also provide a very low impedance path for return current to the supply. Using the same principles, using large copper planes for the digital and analog supply lines will provide low impedance paths for powering each component and circuitry.

It is recommended that the full Wheatstone bridge configuration with strain gauges that have the same temperature coefficient of resistivity be used to acquire strain measurements during flight. The dynamic strain tests and low temperature tests shows that the full-bridge configuration reduces noise in the data and causes little to no drift. In situations where only one or two strain gauges are desired for acquiring data, the rest of the bridge should be constructed using the same gauge so that the data does not drift. The “dummy” gauges used to construct the rest of the Wheatstone bridge should be mounted such that they experience no strain and they are at the same temperature as the strain gauges of interest.

WORKS CITED

"Aluminum 6061-T6; 6061-T651." *ASM Material Data Sheet*, 2018, asm.matweb.com/search/SpecificMaterial.asp?bassnum=MA6061T6

"Analog to Digital Conversion," Measurement Computing, www.mccdaq.com/PDFs/specs/Analog-to-Digital.pdf.

"Atmel Studio 7." *Atmel Studio 7 | Microchip Technology*, 2019, www.microchip.com/mplab/avr-support/atmel-studio-7

ChaN., "FatFs - Generic FAT Filesystem Module," 2019, elm-chan.org/fsw/ff/00index_e.html

Craig, Roy R., and Andrew J. Kurdila. *Fundamentals of Structural Dynamics*. John Wiley & Sons, 2006.

"Crystal Unit," SEIKO EPSON CORPORATION, Product Number MC-405: Q1xMC4052xxxx00, https://support.epson.biz/td/api/doc_check.php?dl=brief_MC-405&lang=en

"FT232R USB UART IC Datasheet," Future Technology Devices International Ltd., https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf

Jellen, Z. Wilson, and R. Westphal, "A Next-Generation Modular Boundary Layer Data System," accepted for AIAA Aviation 2019, Dallas, Tx, June 2019.

Karasawa, "Unsteady Total Pressure Measurement for Laminar-to-Turbulent Transition Detection," MS Thesis, California Polytechnic State University, San Luis Obispo, 2011, http://digitalcommons.calpoly.edu/theses/613/

Kinkade, "Addition of a Stanton Gauge to the Boundary Layer Data System," MS Thesis, California Polytechnic State University, San Luis Obispo, 2014, https://digitalcommons.calpoly.edu/theses/1260/

"Libraries." *Arduino*, 2019, www.arduino.cc/en/reference/libraries.

"Low cost, low power, input/output rail-to-rail operational amplifiers Rev 8," STMicroelectronics, 2015, https://www.st.com/content/ccc/resource/technical/document/datasheet/b9/d8/3b/8f/a1/8f/46/01/CD00079372.pdf/files/CD00079372.pdf/jcr:content/translations/en.CD00079372.pdf

"Metal Film Resistors, Axial, Industrial, Precision," Vishay, September 2016, https://www.mouser.com/Passive-Components/Resistors/Film-Resistors/Metal-Film-Resistors-Through-Hole/Datasheets/_/N-7gz41?P=1z0x6xbZ1z0vpm5

"MICRO SD CONN. NORMAL ULTRA LOW-PROFILE ASSY," Molex, Drawing #SD-503182-011, September 2019, https://www.molex.com/pdm_docs/sd/5031821852_sd.pdf

"Micro USB 2.0 Connectors," FCI, https://cdn.amphenol-icc.com/media/wysiwyg/files/documentation/datasheet/inputoutput/io_usb_micro.pdf

N. Hoyt, "Compact Rake Boundary Layer Data System Module," MS Thesis, California Polytechnic State University, San Luis Obispo, 2013. No access URL available.

"N-Channel Logic Level Enhancement Mode Field Effect Transistor," Fairchild, 2014, https://www.onsemi.com/pub/Collateral/BSS123L-D.pdf

"Op Amp Differential Amplifier Circuit: Voltage Subtractor." *Electronics Hub*, 24 Feb. 2019, www.electronicshub.org/differential-amplifier/

Oswold, Connor. "Data Structures: Your Quick Intro to Circular Buffers." Medium, 7 May 2019, medium.com/better-programming/now-buffering-7a7d384faab5

"Practical Strain Gauge Measurements," Omega Engineering, 1999, https://www.omega.co.uk/techref/pdf/StrainGage_Measurement.pdf

R. Westphal, D. Frame, M. Bleazard, S. Jordan, A Wanner, B. Thompson, A. Bender, and A. Drake, "Design of a Third-Generation Boundary Layer Measurement System," AIAA paper 2008-7332, 2008.

R. Westphal, M. Bleazard, A. Drake, A. Bender, D. Frame and S. Jordan, "A Compact, Self-Containing System for Boundary Layer Measurement In-Flight", AIAA paper 2006-3828, AIAA Meeting Papers on Disc [CD-ROM], Vols. No. 10-13, 2006.

R. Westphal, R. Schelley, and D. Frame. "Instrument for In-Flight Boundary Layer Rake Measurements," AIAA paper 2017-0251, 2017.

"RFD22102 RFduino DIP," RFduino, December 2013, https://docs-emea.rs-online.com/webdocs/12c6/0900766b812c6383.pdf

"RFD77203 29-Pin Breakout Board," RF Digital Corporation, 2015, https://www.mouser.com/pdfdocs/SimbleeRFD77203Datasheet.pdf

S. Lillywhite, "Microphone-based Pressure Diagnostics for Boundary Layer Transition," MS Thesis, California Polytechnic State University, San Luis Obispo, 2013, https://digitalcommons.calpoly.edu/theses/1064/

sandcoffin, fatfs, GitHub repository, 2016, https://github.com/sandcoffin/fatfs

"SanDisk microSD OEM Product Manual," SanDisk, Document No. 80-36-03335, March 2010, https://www.alliedelec.com/m/d/04db416b291011446889dbd6129e2644.pdf

"Simblee™ Bluetooth Smart Module RFD77101," RF Digital, 2016, https://cdn.sparkfun.com/datasheets/IoT/Simblee%20RFD77101%20Datasheet%20v1.0.pdf

"SparkFun Load Cell Amplifier - HX711," SparkFun Electronics, 2013, https://www.sparkfun.com/products/13879

"The Atmel-ICE Debugger," Atmel, Atmel-42330C-Atmel-ICE_User Guide, 2016, http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-ICE_UserGuide.pdf

Ulk, "Implementation of a Conrad Probe on a Boundary Layer Measurement System," MS Thesis, California Polytechnic State University, San Luis Obispo, 2010, http://digitalcommons.calpoly.edu/theses/375/

"Ultra High-Precision Z-Foil Flip Chip Resistor," Vishay, October 2015, http://www.vishaypg.com/docs/63106/VFCP.pdf

"What Is SparkFun?," SparkFun Electronics, https://www.sparkfun.com/what_is_sparkfun.

Zumbahlen, Hank. "Staying Well Grounded.," Analog Devices, 2012, www.analog.com/en/analog-dialogue/articles/staying-well-grounded.html

"2.5 V/3.0 V High Precision Reference," Analog Devices, 2017, https://www.analog.com/media/en/technical-documentation/data-sheets/AD780.pdf

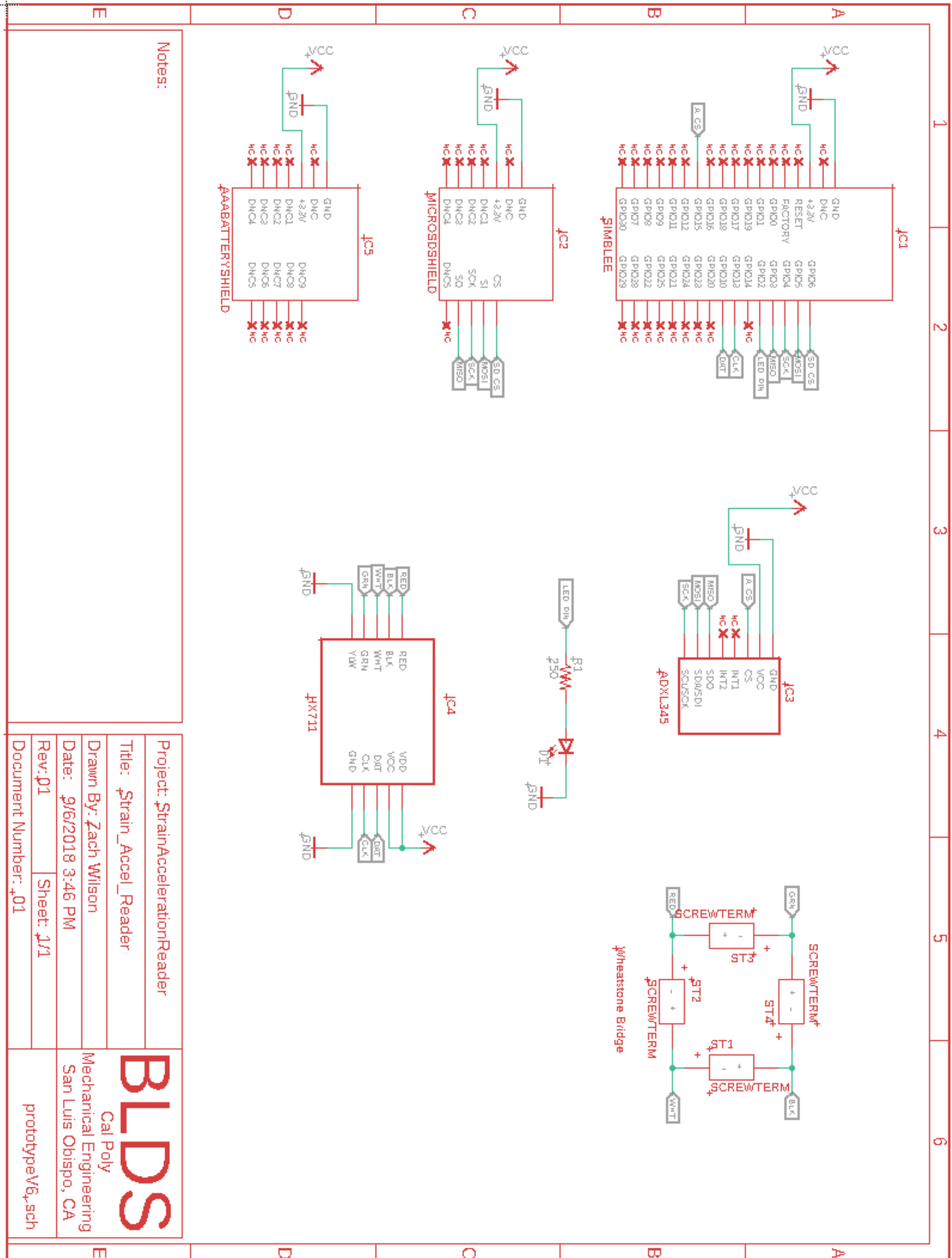
"24-bit Analog-to-Digital Converter (ADC) for Weigh Scales," AVIA Semiconductor, June 2018, https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/hx711_english.pdf

"3-Axis, ±2 g/ ±4 g/ ±8 g/ ±16 g Digital Accelerometer," Analog Devices, 2015, https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf

"8-/10-Channel, Low Voltage, Low Power, Σ-Δ ADCs," Analog Devices, 2001, https://www.analog.com/media/en/technical-documentation/data-sheets/AD7708_7718.pdf

"8/16-bit Atmel XMEGA Microcontroller," Atmel, 2014, http://ww1.microchip.com/downloads/en/devicedoc/atmel-8387-8-and16-bit-avr-microcontroller-xmega-a4u_datasheet.pdf

A. QUASI-STATIC DATA ACQUISITION SYSTEM SCHEMATIC AND BOARD LAYOUT



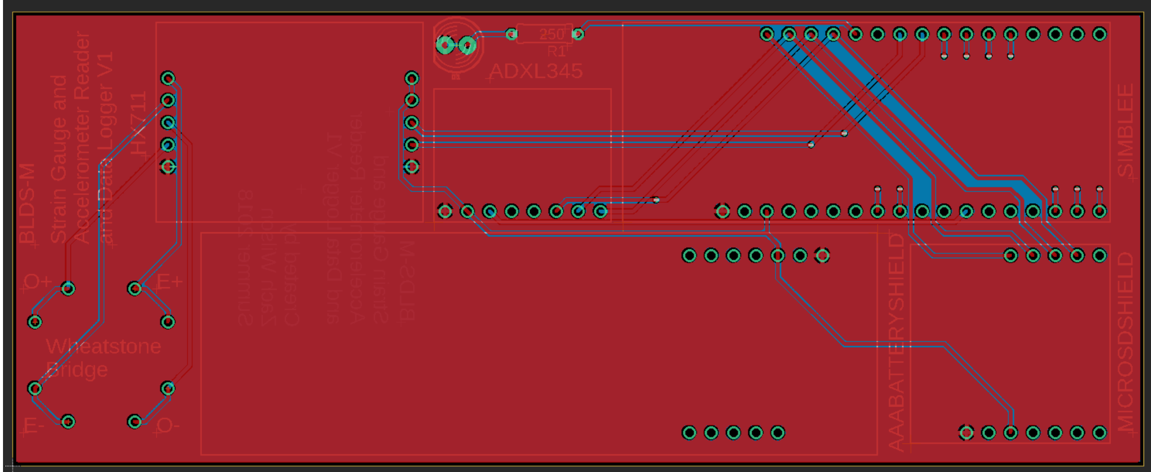


Figure A-1: Quasi-static Data Acquisition System PCB top layer layout.

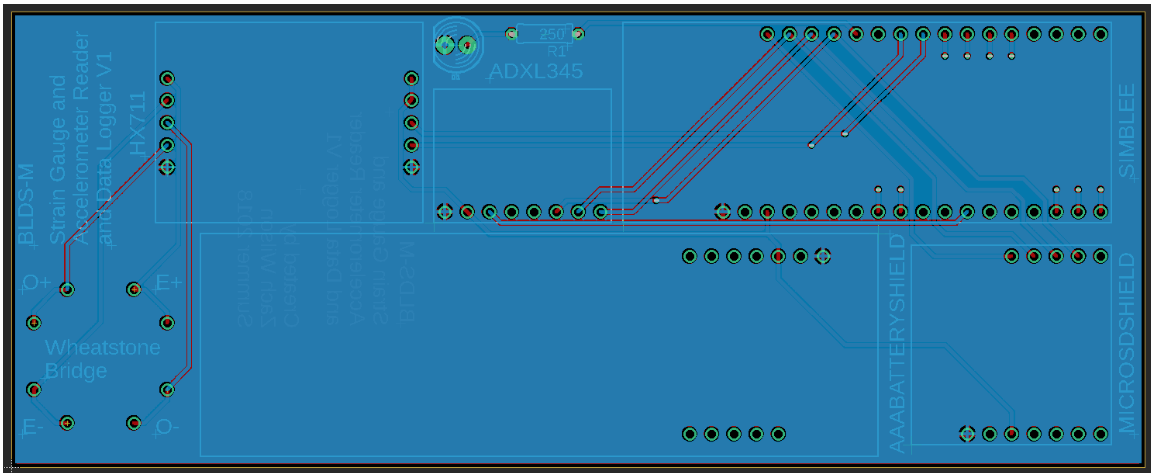


Figure A-2: Quasi-static Data Acquisition System PCB bottom layer layout.

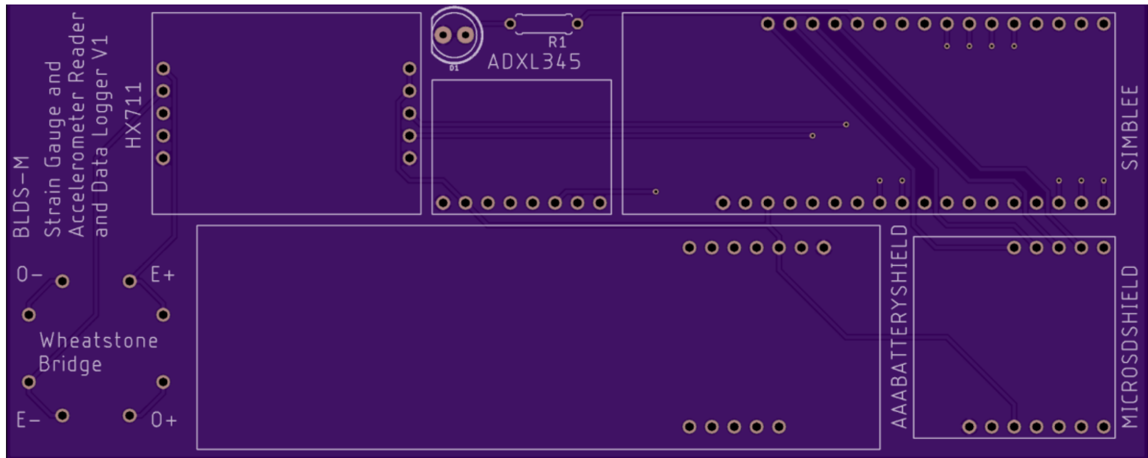


Figure A-3: Quasi-static Data Acquisition System PCB top render.

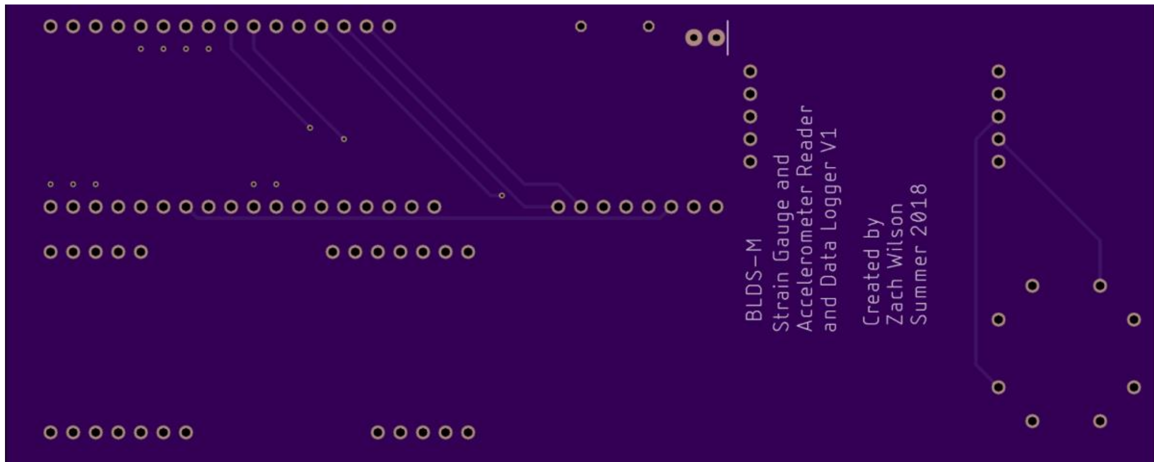
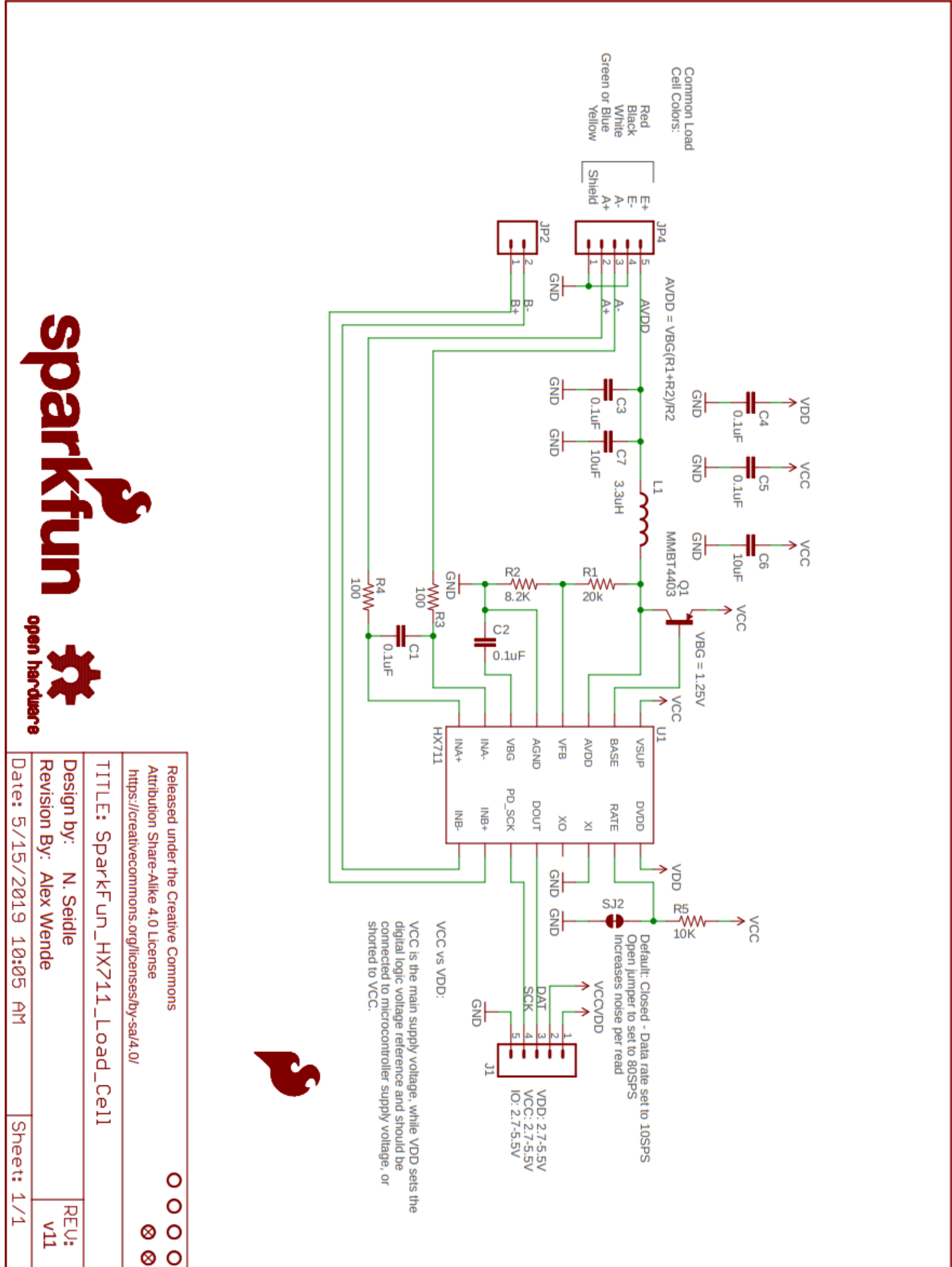


Figure A-4: Quasi-static Data Acquisition System PCB bottom render.

B. HX711 LOAD CELL AMPLIFIER BREAKOUT SCHEMATIC AND BOARD LAYOUT



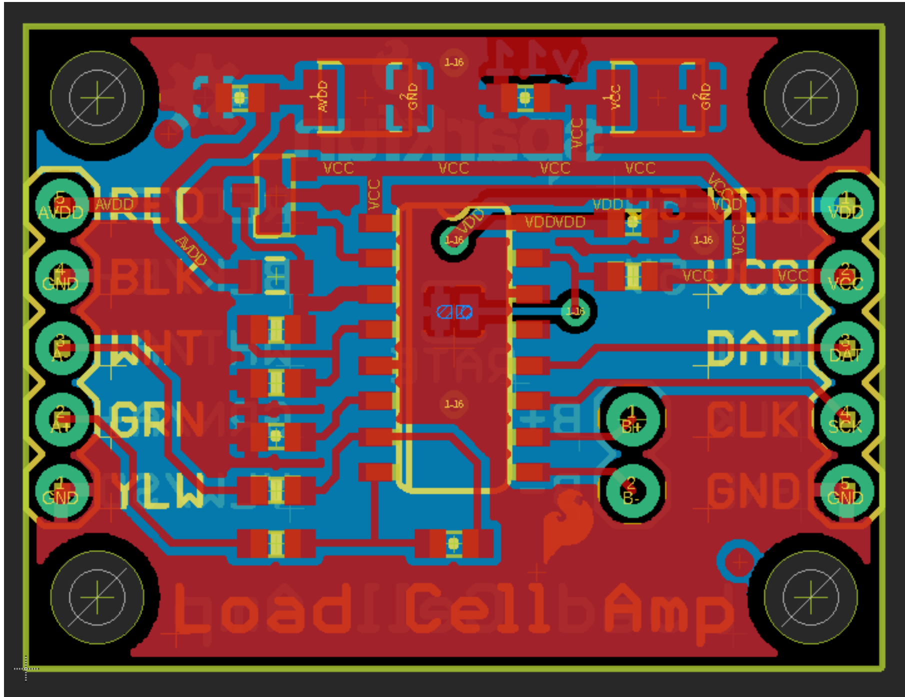


Figure B-1: HX711 breakout board top layer layout.

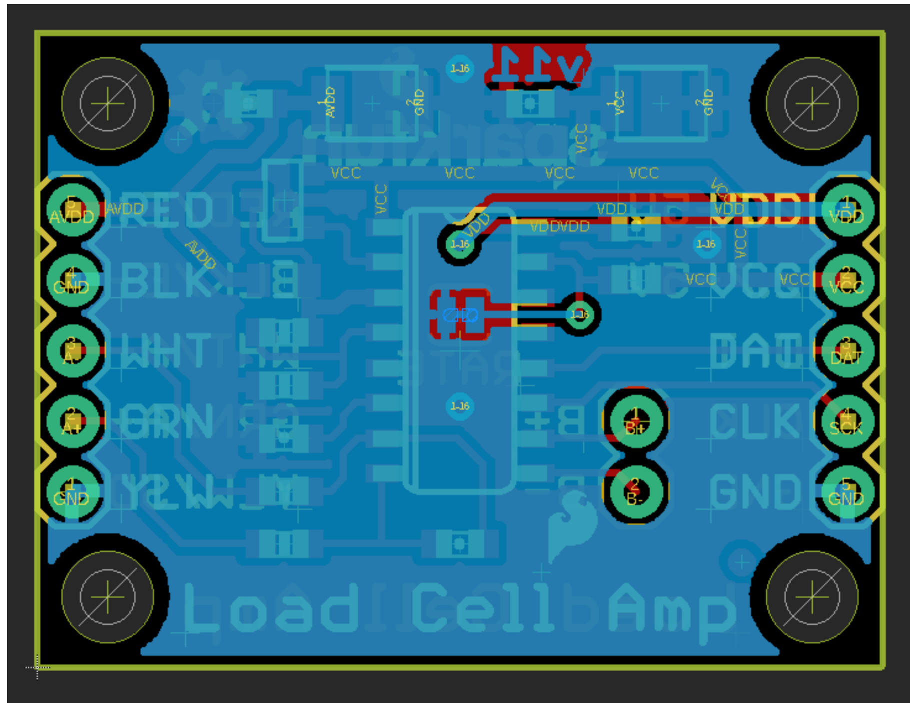
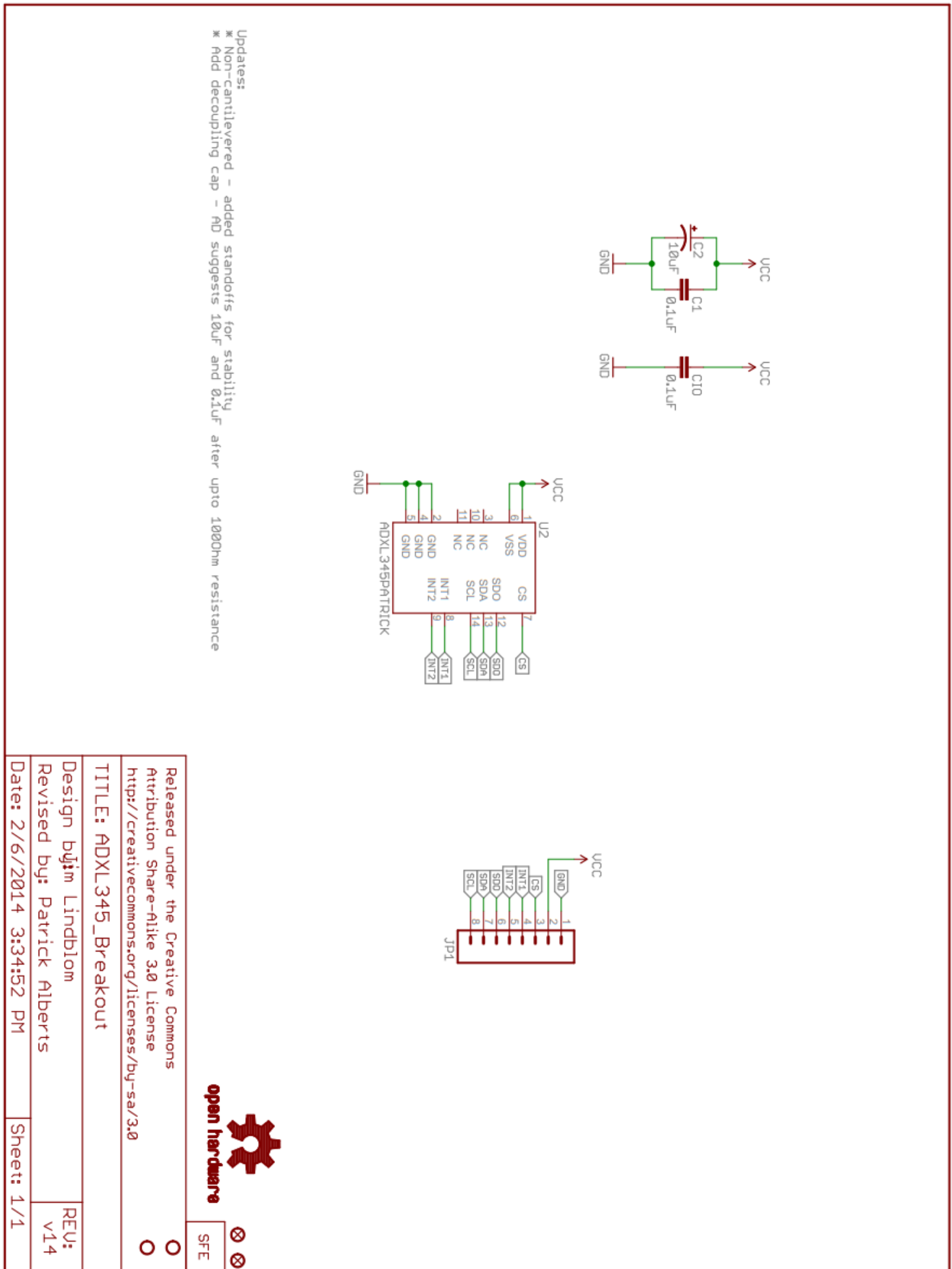


Figure B-2: HX711 breakout board bottom layer layout.

C. ADXL345 ACCELEROMETER BREAKOUT SCHEMATIC AND BOARD LAYOUT



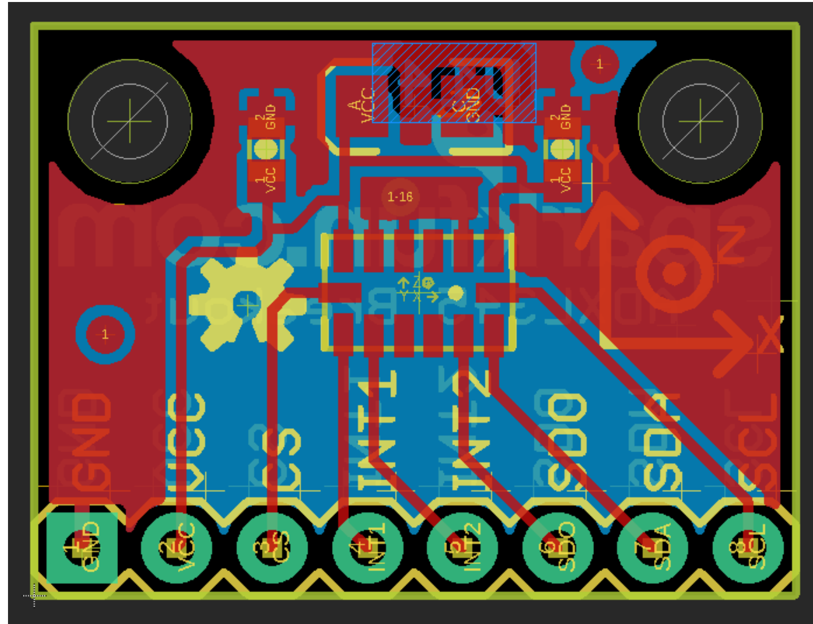


Figure C-1: ADXL345 breakout board top layer layout.

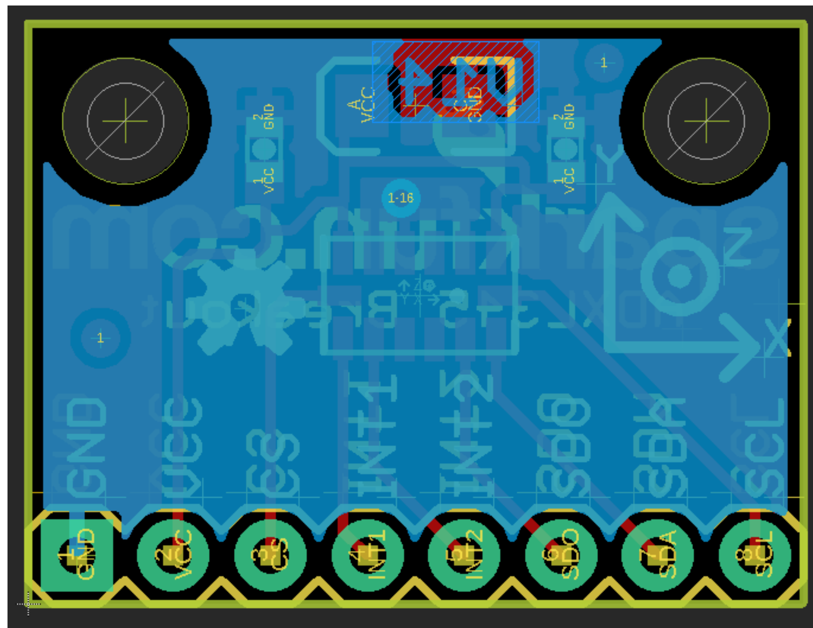


Figure C-2: ADXL345 breakout board bottom layer layout.

D. HX711 AND ADXL345 ARDUINO DRIVERS

```
/** @file HX711.h
 *  @brief HX711 Strain Gauge Amplifier Library - Class header file
 *
 *  Contributors: Zach Wilson 7/25/18
 *  References:
 *      HX711 Datasheet
 *  Libraries:
 *
 *  Notes:
 *      The methods provided by this class gives strain readings in voltage to minimize the calculations between
 *      readings. Use the equation below to calculate mechanical strain.
 *
 *      strainVoltage = output from readStrain () and readAverageStrain () methods.
 *      tareVoltage = output from readStrain () and readAverageStrain () methods while the gauge is under no load
(wind-off).
 *      voltageratio = (strainVoltage / 3.3) - (tareVoltage / 3.3)
 *      mechanicalStrain = (-4 * voltageratio / (GAUGE_FACTOR * (1 + (2 * voltageratio))))
 *
 */

#ifndef HX711_h
#define HX711_h

// Included Libraries
#include "Arduino.h"

/** @brief HX711 Strain Gauge Reader Class.
 *
 *  AVIA Semiconductor HX711 Load Cell Amplifier
 */
class HX711 {

  private:

    uint8_t _DAT;                      //!< Stores DAT pin
    uint8_t _CLK;                      //!< Stores CLK pin
    float _TARE = 0;                   //!< Tare for data offset
    uint8_t _GAIN = 0;                 //!< Value for selecting channel A or B
    float _GAUGE_FACTOR = 0;           //!< Strain Gauge conversion from change in resistance to mechanical strain

    int32_t readData ();

  public:

    float strainRatio = 0;             //!< Stores last reading from the HX711 in volts

    HX711 (uint8_t CLK, uint8_t DAT);
    void begin ();
    void reset ();
    void readStrain ();
    void readAverageStrain (uint8_t times);
    void chooseGain (uint8_t value);
    void setTare (uint8_t times);
    void setGaugeFactor (float value);
    void enable ();
    void disable ();
    bool getStatus ();

};

#endif
```

```
/** @file HX711.cpp
 *  @brief HX711 Strain Gauge Amplifier Library - Main class
 *
 *  Contributors: Zach Wilson 7/25/18
 *  References:
 *      HX711 Datasheet
 *  Libraries:
 *
 *  Notes:
 *
 *      The methods provided by this class give strain readings in voltage to minimize the calculations between
 *      readings. Use the equations below to calculate mechanical strain.
 *
 *      strainVoltageRatio = output from readStrain () and readAverageStrain () methods.
 *      tareVoltageRatio = output from readStrain () and readAverageStrain () methods while the gauge is under no load.
 *      voltageratio = strainVoltageRatio - tareVoltageRatio
 *      mechanicalStrain = (4 * voltageratio / (GAUGE_FACTOR * (1 + (2 * voltageratio))))
 *
 *
 */

// Included Libraries
#include "Arduino.h"
#include "HX711.h"

/** @brief Default Constructor. Creates an HX711 object.
 *  @details This constructor only stores the CLK and DAT pins for communication with the HX711.
 *  @param CLK Clock pin for communication with the HX711.
 *  @param DAT Data pin to retrieve reading from HX711 in bytes.
 */
HX711 :: HX711 (uint8_t CLK, uint8_t DAT) {

  _CLK = CLK;
  _DAT = DAT;

}

/** @brief Initializes the HX711 object.
 *  @details This method configures the CLK and DAT pins for output and input, respectively. It calls the powerUp ()
 *  method to wake up the device, then waits until the dataReady () method returns true. It then takes a few readings
 *  using the readAverageStrain () method and then powers down the device.
 */
void HX711 :: begin () {

  pinMode (_CLK, OUTPUT);
  pinMode (_DAT, INPUT);

  enable ();

  while (!getStatus ()) {}

  readAverageStrain (10);
  strainRatio = 0;

  disable ();

}

/** @brief Reads data from HX711.
 *  @details This method waits until the HX711 is ready to send data and uses the CLK pin to send 24 clock pulses to
 *  the HX711 to retrieve 3 bytes of data. It then converts the data into a 32-bit integer.
 *  @return Returns a 32-bit integer containing the value read by the HX711.
 */
int32_t HX711 :: readData () {

  int32_t strainData = 0;
  uint8_t rawData[3] = { 0 };

  while (!getStatus ()) {}                              // Waits until DAT and CLK pins are low, indicating HX711 chip
is ready to send data

  rawData[2] = shiftIn (_DAT, _CLK, MSBFIRST);
  rawData[1] = shiftIn (_DAT, _CLK, MSBFIRST);
  rawData[0] = shiftIn (_DAT, _CLK, MSBFIRST);

  for (int i = 0; i < _GAIN; i++) {                     // Sends a number of clock pulses to set HX711 chip to channel
A or B based on _GAIN
    digitalWrite (_CLK, HIGH);
    digitalWrite (_CLK, LOW);

  }

  if (rawData[2] & 0x80) {                              // Populates MSB if data is negative and leaves it alone if
positive
    strainData = 0xFF;

  }
  else {
    strainData = 0x00;

  }

  strainData = (strainData << 8) | rawData[2];          // Constructs a 32-bit number
  strainData = (strainData << 8) | rawData[1];
  strainData = (strainData << 8) | rawData[0];
```

```
  return strainData;

}

/** @brief Reads voltage from the HX711.
 *  @details Calls the readData () method and converts the 32-bit integer to a voltage reading. Stores reading
 *  in public variable strainVoltage
 */
void HX711 :: readStrain () {

  strainRatio = readData () / 8388608.0;

}

/** @brief Reads average value of voltage from the HX711.
 *  @details Calls the readStrain () method a number of times to get an average reading. Stores reading in public
 *  variable strainVoltage
 *  @param times Number of readings to average.
 */
void HX711 :: readAverageStrain (uint8_t times) {

  float summedValue = 0;

  for (uint8_t i = 0; i < times; i++) {
    readStrain ();
    summedValue += strainRatio;

  }

  strainRatio = summedValue / times;

}

/** @brief Adjusts for the tare.
 *  @details This method calls the readData () method a number of times to get an anverage reading. This method
 *  should be called when the strain gauge is under no load. Stores tare in the private variable _TARE.
 *  @param times Number of readings to average.
 */
void HX711 :: setTare (uint8_t times) {

  int32_t summedValue;
  int32_t buf;

  for (uint8_t i = 0; i < times; i++) {
    summedValue += readData ();

  }

  buf = summedValue / times;

  _TARE = (buf / 8388607.0) * (13.2 / 1000) / 3.3;

}

/** @brief Sets the gauge factor of the strain gauge.
 *  @details This method only stores the desired gauge factor in the private variable _GAUGE_FACTOR for converting
 *  from change in electrical resistance to mechanical strain.
 *  @param value The desired gague factor.
 */
void HX711 :: setGaugeFactor (float value) {

  _GAUGE_FACTOR = value;

}

/** @brief Sets the gain of the output data from the HX711.
 *  @details This method only stores the desired gain in the private variable _GAIN. Alters the number of clock
 *  pulses to send to the HX711 after a reading to set the channel and gain.
 *  @param value A number to set the _GAIN parameter.
 *      value = 1 : Gain = 128 on channel A
 *      value = 2 : Gain = 64 on channel B
 *      value = 3 : Gain = 32 on channel A
 */
void HX711 :: chooseGain (uint8_t value) {

  _GAIN = value;

}

/** @brief Powers on the HX711.
 *  @details Wakes up the HX711 by setting the CLK pin low.
 */
void HX711 :: enable () {

  digitalWrite (_CLK, LOW);

}

/** @brief Powers off the HX711.
 *  @details Puts the HX711 in low power mode by setting the CLK pin high for 60 microseconds.
 */
void HX711 :: disable () {

  digitalWrite (_CLK, LOW);
```

```
  digitalWrite (_CLK, HIGH);
  delayMicroseconds (60);

}

/** @brief Boolean indicating data is ready to be sent from the HX711.
 *  @details This method checks if both the DAT and CLK pins are low, indicating that data is ready to be sent and
received.
 *  @return Boolean indicating if data is ready to be sent.
 */
bool HX711 :: getStatus () {

  if ((!digitalRead (_DAT))&&(!digitalRead (_CLK))) {
    return true;

  }
  else {
    return false;

  }

}
```

```
/** @file ADXL345.h
 *  @brief ADXL345 Accelerometer Library - Class header file
 *
 *  Contributors: Zach Wilson 7/25/18
 *  References:
 *      ADXL345 Datasheet
 *  Libraries:
 *
 *  Notes:
 *      The methods provided by this class output acceleration data in units of g's. No calculations are
 *      necessary unless different unit of acceleration is desired.
 *
 */

#ifndef ADXL345_h
#define ADXL345_h

// Included Libraries
#include "Arduino.h"

// ADXL345 Data Registers
#define ADXL345_DATA_FORMAT        0x31    //!< Data format register
#define ADXL345_BW_RATE            0x2C    //!< Data rate control
#define  ADXL345_POWER_CTL         0x2D    //!< Power control register
#define ADXL345_DATAX0             0x32    //!< X-Axis Data LSB
#define ADXL345_DATAX1             0x33    //!< X-Axis Data MSB
#define ADXL345_DATAY0             0x34    //!< Y-Axis Data LSB
#define ADXL345_DATAY1             0x35    //!< Y-Axis Data MSB
#define ADXL345_DATAZ0             0x36    //!< Z-Axis Data LSB
#define ADXL345_DATAZ1             0x37    //!< Z-Axis Data MSB

/** @brief ADXL345 Accelerometer Reader Class.
 *
 *  Analog Devices Inc. ADXL345 Accelerometer
 */
class ADXL345 {

  private:

    uint8_t _CS;                       //!< Chip select pin
    union {                            //!< Union for assembling 6 raw data bytes into 3 signed 16-bit numbers
      uint8_t _accelData[6];           //!< Buffer for raw data
      int16_t _xyz_accelData[3];       //!< Buffer for assembled data
    };
    float _x_GAIN = 1.0;               //!< Gain for x-axis data
    float _y_GAIN = 1.0;               //!< Gain for y_axis data
    float _z_GAIN = 1.0;               //!< Gain for z-axis data

  public:

    float x_acceleration;          //!< Stores x-axis acceleration
    float y_acceleration;          //!< Stores y-axis acceleration
    float z_acceleration;          //!< Stores z-axis acceleration

    ADXL345 (uint8_t CS);
    void begin ();
    void reset ();
    void setGains (float x_GAIN, float y_GAIN, float z_GAIN);
    void readData ();
    void readXAccel ();
    void readYAccel ();
    void readZAccel ();
    void readXYZAccel ();
    void setRange (uint8_t value);
    void setRate (uint16_t value);
    void enable ();
    void disable ();

};

#endif
```

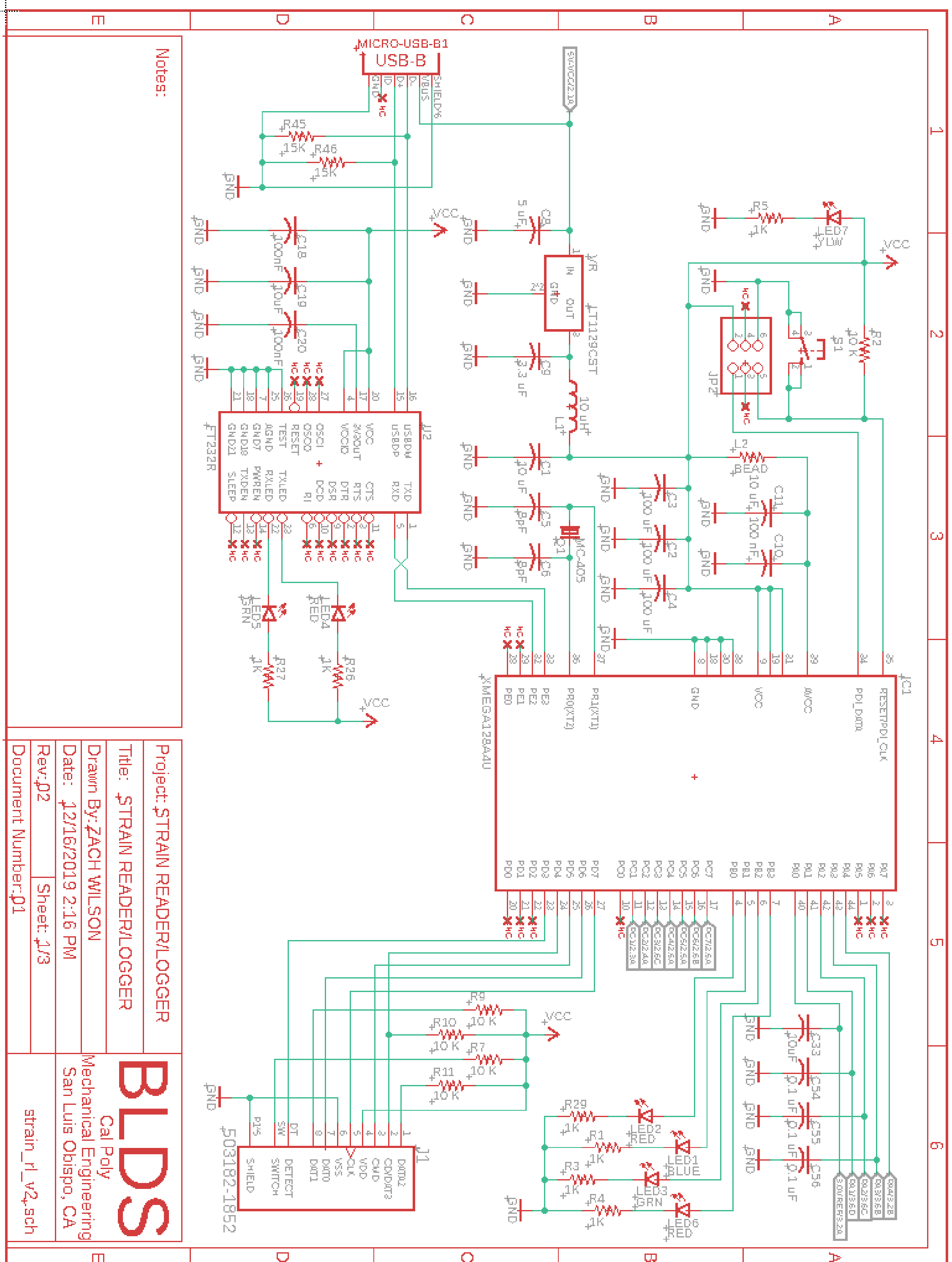
```
/** @file ADXL345.cpp
 *  @brief ADXL345 Accelerometer Library - Main class
 *
 *  Contributors: Zach Wilson 7/25/18
 *  References:
 *      ADXL345 Datasheet
 *  Libraries:
 *
 *  Notes:
 *      The methods provided by this class output acceleration data in units of g's. No calculations are
 *      necessary unless different unit of acceleration is desired.
 *
 */

// Included Libraries
#include "Arduino.h"
#include "ADXL345.h"
#include <SPI.h>

/** @brief Default constructor. Creates an ADXL345 object.
 *  @details This constructor only stores the chip select pin for the ADXL345 object.
 *  @param CS Chip select pin for the ADXL345 object.
 */
ADXL345 :: ADXL345 (uint8_t CS) {

  _CS = CS;

}

/** @brief Initializes the ADXL345 object.
 *  @details This method initializes the ADXL345 object by setting the chip select pin low and initializes the SPI
 *  library. Also places the ADXL345 object in standby mode using powerDown ().
 */
void ADXL345 :: begin () {

  pinMode (_CS, OUTPUT);
  digitalWrite (_CS, HIGH);
  SPI.begin ();

}

/** @brief Resets the ADXL345.
 *  @details
 */
void ADXL345 :: reset () {

}

/** @brief Sets gains of the ADXL345 object.
 *  @details This method stores the gains for the x, y, and z axis to convert the reading in bytes to g's.
 *  @param x_GAIN The gain to convert the x-axis reading into g's.
 *  @param y_GAIN The gain to convert the y-axis reading into g's.
 *  @param z_GAIN The gain to convert the z-axis reading into g's.
 */
void ADXL345 :: setGains (float x_GAIN, float y_GAIN, float z_GAIN) {

  _x_GAIN = x_GAIN;
  _y_GAIN = y_GAIN;
  _z_GAIN = z_GAIN;

}

/** @brief Reads data from the ADXL345 object.
 *  @details This method reads 6 bytes of data from the ADXL345 using the SPI library and stores the data in
 *  the private variable _accelData.
 */
void ADXL345 :: readData () {

  uint8_t address = 0x80 | ADXL345_DATAX0;                            // Address for LSB of x-axis data register
  address = address | 0x40;

  SPI.beginTransaction (SPISettings (5000000, MSBFIRST, SPI_MODE3));  // Begin SPI transaction and set CS pin low
  digitalWrite (_CS, LOW);
  SPI.transfer (address);
  for (uint8_t i = 0; i < 6; i++) {                                   // Transfers 6 bytes of data from ADXL345
    _accelData[i] = SPI.transfer (0x00);
  }
  SPI.endTransaction ();                                              // End transaction and set CS pin high

  digitalWrite (_CS, HIGH);

}

/** @brief Reads the x-axis acceleration.
 *  @details This method calls the readData () method and compiles the x-axis reading into a float. Stores reading
 *  in the public variable x_acceleration.
 */
void ADXL345 :: readXAccel () {

  int16_t x_data;

  readData ();                                                                   // Calls readData () method and
constructs x-axis data from 2 bytes
  x_data = (int16_t)((((uint8_t)_accelData[1]) << 8) | (uint8_t)_accelData[0]);
```

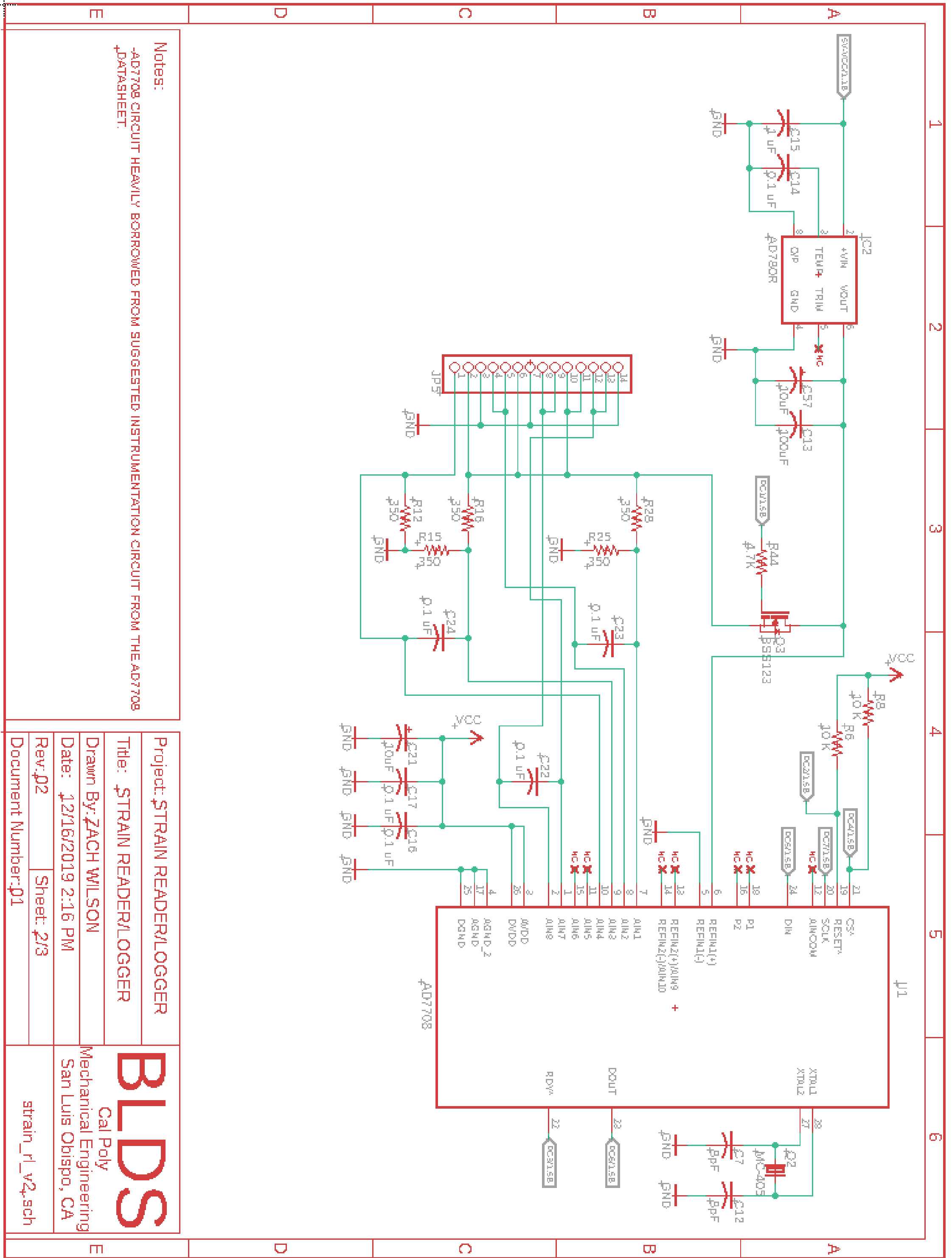
```
  x_acceleration = x_data * _x_GAIN;                                              // Stores reading in g's

}

/** @brief Reads the y-axis acceleration.
 *  @details This method calls the readData () method and compiles the y-axis reading into a float. Stores reading
 *  in the public variable y_acceleration.
 */
void ADXL345 :: readYAccel () {

  int16_t y_data;

  readData ();                                                                    // Calls readData () method and
constructs y-axis data from 2 bytes
  y_data = (int16_t)((((uint8_t)_accelData[3]) << 8) | (uint8_t)_accelData[2]);

  y_acceleration = y_data * _y_GAIN;                                              // Stores reading in g's

}

/** @brief Reads the z-axis acceleration.
 *  @details This method calls the readData () method and compiles the z-axis reading into a float. Stores reading
 *  in the public variable z_acceleration.
 */
void ADXL345 :: readZAccel () {

  int16_t z_data;

  readData ();                                                                    // Calls readData () method and
constructs z-axis data from 2 bytes
  z_data = (int16_t)((((uint8_t)_accelData[5]) << 8) | (uint8_t)_accelData[4]);

  z_acceleration = z_data * _z_GAIN;                                              // Stores reading in g's

}

/** @brief Reads the acceleration of all three axis.
 *  @details This method calls the readData () method and compiles the x, y, and z readings into floats
 *  and stores them in the public variables x_acceleration, y_acceleration, and z_acceleration.
 */
void ADXL345 :: readXYZAccel () {

  int16_t x_data;
  int16_t y_data;
  int16_t z_data;

  readData ();                                                                    // Calls readData () method and
constructs data for all axis
  x_data = (int16_t)((((uint8_t)_accelData[1]) << 8) | (uint8_t)_accelData[0]);
  y_data = (int16_t)((((uint8_t)_accelData[3]) << 8) | (uint8_t)_accelData[2]);
  z_data = (int16_t)((((uint8_t)_accelData[5]) << 8) | (uint8_t)_accelData[4]);

  x_acceleration = x_data * _x_GAIN;                                              // Stores 3 readings in 3 floats
  y_acceleration = y_data * _y_GAIN;
  z_acceleration = z_data * _z_GAIN;

}

/** @brief Sets the acceleration range of the ADXL345.
 *  @details Sends a byte to the ADXL345_DATA_FORMAT register that sets the desired g range.
 *  @param value Selects the acceleration range according to the integer value given.
 *      value = 2 : ± 2g
 *      value = 4 : ± 4g
 *      value = 8 : ± 8g
 *      value = 16 : ± 16g
 */
void ADXL345 :: setRange (uint8_t value) {

  uint8_t setting;

  switch (value) {
    case 2:
      setting = B00101000;             // Setting for ± 2g
      break;

    case 4:
      setting = B00101001;             // Setting for ± 4g
      break;

    case 8:
      setting = B00101010;             // Setting for ± 8g
      break;

    case 16:
      setting = B00101011;             // Setting for ± 16g
      break;

    default:
      setting = B00101010;             // Default is ± 8g
      break;

  }

  SPI.beginTransaction (SPISettings (5000000, MSBFIRST, SPI_MODE3));   // Begin SPI transaction and set CS pin low
  digitalWrite (_CS, LOW);
```

```
  SPI.transfer (ADXL345_DATA_FORMAT);                                   // Send one byte to ADXL345 data format register
  SPI.transfer (setting);
  SPI.endTransaction ();

  digitalWrite (_CS, HIGH);                                             // End transaction and set CS pin high

}

/** @brief Sets the rate of data output of the ADXL345.
 *  @details Sends a byte to the ADXL345_BW_RATE data register that sets the desired data output rate.
 *  @param value Selects a rate at which to output data.
 *      value = 6 : 6.25Hz
 *      value = 12 : 12.5Hz
 *      value = 25 : 25Hz
 *      value = 50 : 50Hz
 *      value = 100 : 100Hz
 *      value = 200 : 200Hz
 *      value = 400 : 400Hz
 *      value = 800 : 800Hz
 *      value = 1600 : 1600Hz
 *      value = 3200 : 3200Hz
 */
void ADXL345 :: setRate (uint16_t value) {

  uint8_t setting;

  switch (value) {
    case 6:
      setting = B00000110;      // Setting for 6.25Hz
      break;

    case 12:
      setting = B00000111;      // Setting for 12.5Hz
      break;

    case 25:
      setting = B00001000;      // Setting for 25Hz
      break;

    case 50:
      setting = B00001001;      // Setting for 50Hz
      break;

    case 100:
      setting = B00001010;      // Setting for 100Hz
      break;

    case 200:
      setting = B00001011;      // Setting for 200Hz
      break;

    case 400:
      setting = B00001100;      // Setting for 400Hz
      break;

    case 800:
      setting = B00001101;      // Setting for 800Hz
      break;

    case 1600:
      setting = B00001110;      // Setting for 1600Hz
      break;

    case 3200:
      setting = B00001111;      // Setting for 3200Hz
      break;

    default:
      setting = B00001010;      // Default is 100Hz
      break;

  }

  SPI.beginTransaction (SPISettings (5000000, MSBFIRST, SPI_MODE3));   // Begin SPI transaction and set CS pin low
  digitalWrite (_CS, LOW);

  SPI.transfer (ADXL345_BW_RATE);                                       // Send one byte to ADXL345 data rate register
  SPI.transfer (setting);
  SPI.endTransaction ();

  digitalWrite (_CS, HIGH);                                             // End transaction and set CS pin high

}

/** @brief Powers on the ADXL345.
 *  @details Sets the ADXL345 in measurement mode by sending a byte to the ADXL345_POWER_CTL data register.
 */
void ADXL345 :: enable () {

  uint8_t setting = B00001000;                                          // Setting for measurement mode

  SPI.beginTransaction (SPISettings (5000000, MSBFIRST, SPI_MODE3));   // Begin SPI transaction and set CS pin low
  digitalWrite (_CS, LOW);
```

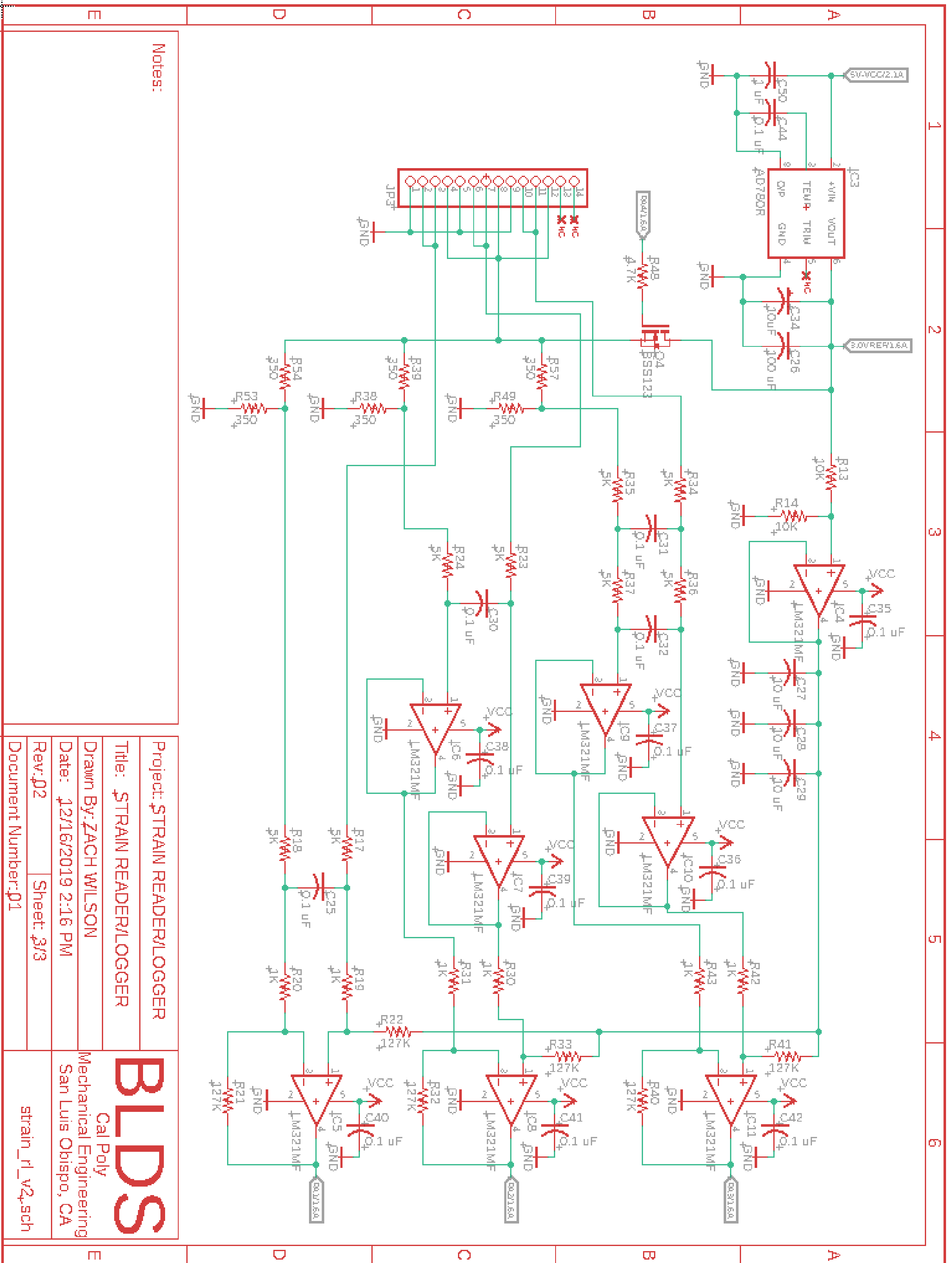
```
  SPI.transfer (ADXL345_POWER_CTL);                                    // Send one byte to ADXL345 power control
register
  SPI.transfer (setting);
  SPI.endTransaction ();

  digitalWrite (_CS, HIGH);                                            // End transaction and set CS pin high

}

/** Powers off the ADXL345.
 *  @details Sets the ADXL345 in standby mode with minimum power consumption by sending a byte to the
 *  ADXL345_POWER_CTL data register.
 */
void ADXL345 :: disable () {

  uint8_t setting = B00000000;                                         // Setting for standby mode

  SPI.beginTransaction (SPISettings (5000000, MSBFIRST, SPI_MODE3));   // Begin SPI transaction and set CS pin low
  digitalWrite (_CS, LOW);

  SPI.transfer (ADXL345_POWER_CTL);                                    // Send one byte to ADXL345 power control
register
  SPI.transfer (setting);
  SPI.endTransaction ();

  digitalWrite (_CS, HIGH);                                            // End transaction and set CS pin high

}
```

E. DYNAMIC DATA ACQUISITION SYSTEM SCHEMATIC AND BOARD LAYOUT





Notes:
-AD7708 CIRCUIT HEAVILY BORROWED FROM SUGGESTED INSTRUMENTATION CIRCUIT FROM THE AD7708 DATASHEET.
Project: STRAIN READER/LOGGER
Title: STRAIN READER/LOGGER
Drawn By: ZACH WILSON
Date: 12/16/2019 2:16 PM
Rev: 02
Sheet: 2/3
Document Number: 01
BLDS
Cal Poly
Mechanical Engineering
San Luis Obispo, CA
strain_rl_v2.sch
IC2
AD780R
+VIN
VOUT
TEMP
TRIM
O/P
GND
C15
C14
C57
C13
1 uF
0.1 uF
10uF
100uF
JP5
R12
R16
R28
R15
R25
350
R44
4.7K
Q3
BSS123
C24
C23
C22
C21
C17
C16
R8
R6
10 K
VCC
GND
NC
U1
AD7708
C7
C12
8pF
Q2
XTAL1
XTAL2
DOUT
RDY
DIN
AINCOM
SCLK
RESET
CS


Notes:
Project: STRAIN READER/LOGGER
Title: STRAIN READER/LOGGER
Drawn By: ZACH WILSON
Date: 12/16/2019 2:16 PM
Rev: 02
Sheet: 3/3
Document Number: 01
BLDS
Cal Poly
Mechanical Engineering
San Luis Obispo, CA
strain_rl_v2.sch

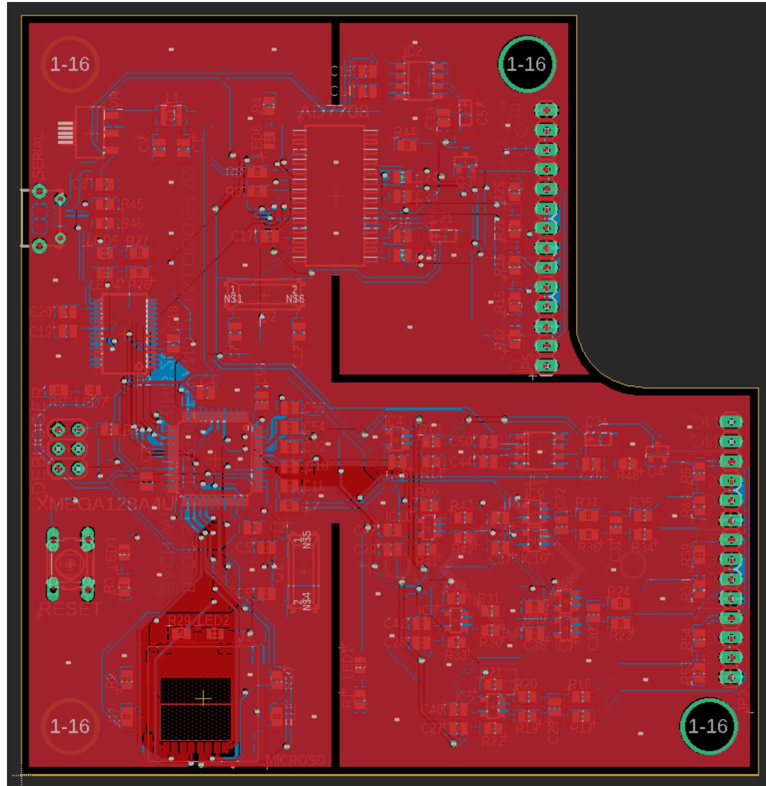


Figure E-1: Dynamic Data Acquisition System PCB top layer layout.

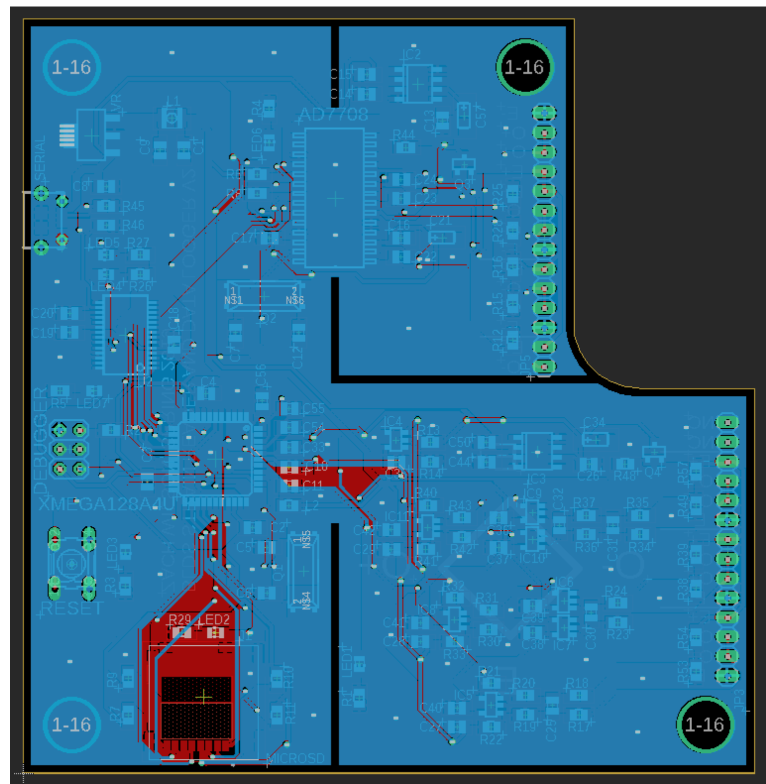


Figure E-2: Dynamic Data Acquisition System PCB bottom layer layout.

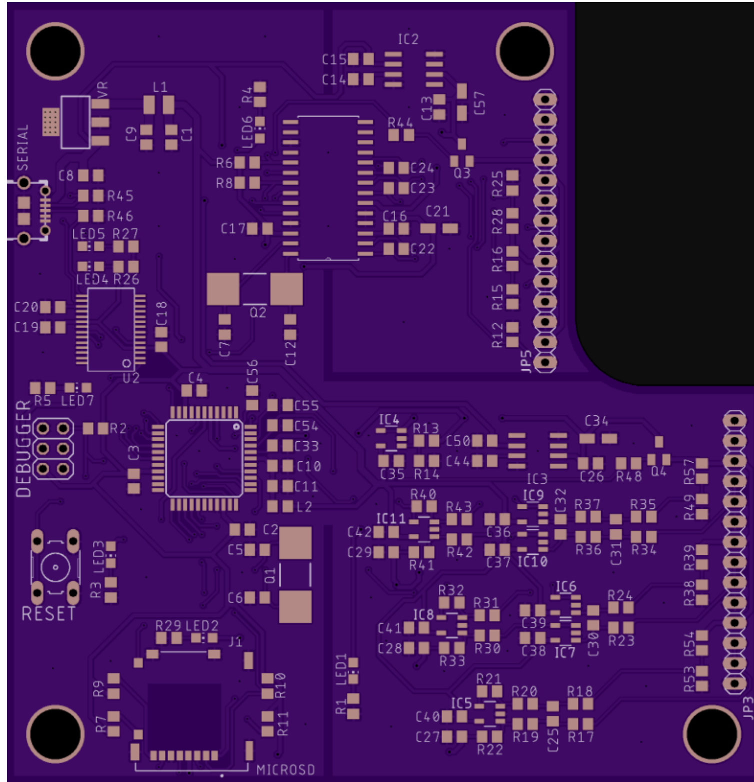


Figure E-3: Dynamic Data Acquisition System PCB top render.

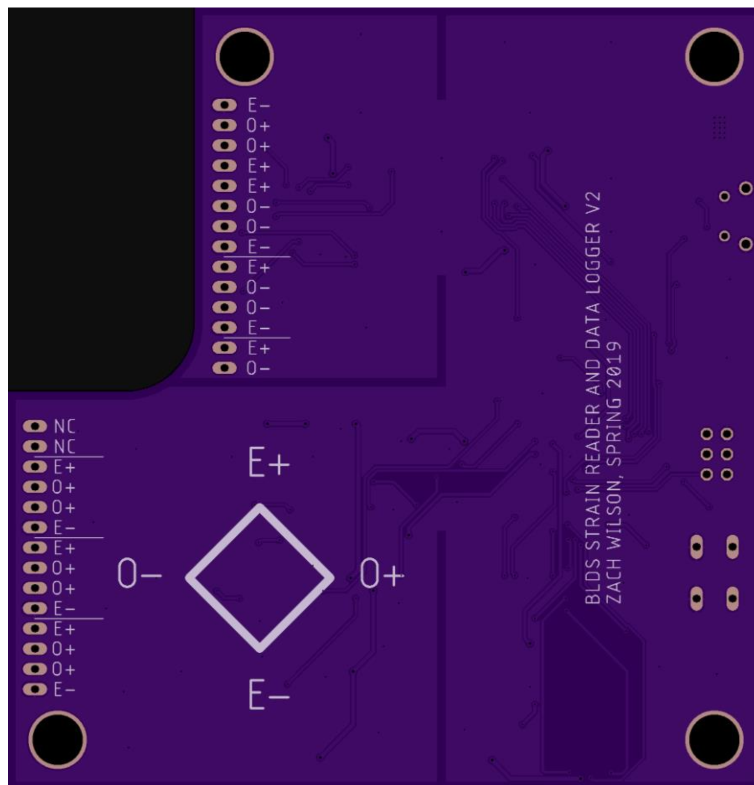


Figure E-4: Dynamic Data Acquisition System PCB bottom render.

F. DERIVATIONS OF FILTER TRANSFER FUNCTIONS, CUTOFF FREQUENCIES, AND MATLAB SIMULATION

Simulations of the first and second-order low-pass filters used to filter the strain data in the Dynamic Data Acquisition System were built to design and verify the performance of each filter. Using linear graph modeling, the state space model and output equation of each filter was derived in the form

$$\dot{x} = Ax + Bu \tag{F-1}$$

$$y = Cx + Du \tag{F-2}$$

where A, B, C, and D are matrices which describe the system and x and y are vectors containing the states and outputs of the system, respectively. From the state space model and output equation, the transfer function of each filter was calculated using the following equation

$$T(s) = C(sI - A)^{-1} + D \tag{F-3}$$

where I is the identity matrix and $T(s)$ is the resulting transfer function. Using MATLAB™, a Bode plot was generated and a simulation was run for each filter.

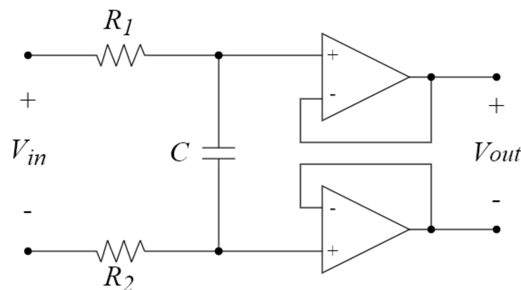


Figure F-1: First-order active low-pass filter schematic.

The schematic of the first-order low-pass filter is displayed in Figure F-1. Because of the voltage buffers on the differential output of the filter, the circuitry downstream of the filter does not affect the performance of the filter. Thus, the linear graph model does not need to include a

load across the filter output. A linear graph model and normal tree were generated from the first-order filter schematic, and are shown in Figure F-2.

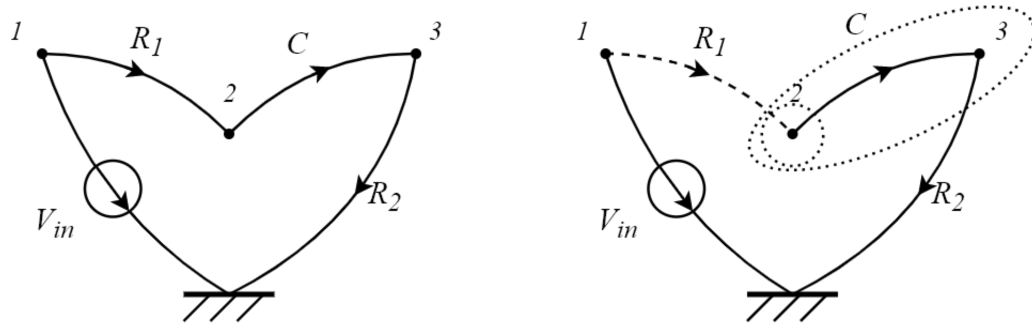


Figure F-2: First-order low-pass filter linear graph (left) and normal tree (right).

The first-order filter model contains two resistors, a capacitor, and a voltage input. The two resistors are modeled as D-type elements, which means the resistors are energy-dissipative elements rather than energy-storage elements. The capacitor is modeled as an A-type energy storage element, which means the capacitor stores energy by virtue of its across variable, voltage. The elemental equations describing the dynamics of each component are listed below.

$$\frac{dV_C}{dt} = \frac{1}{C} i_C \tag{F-4}$$

$$i_{R_1} = \frac{1}{R_1} V_{R_1} \tag{F-5}$$

$$V_{R_2} = R_2 i_{R_2} \tag{F-6}$$

The voltage across the capacitor V_C is the only state variable required to describe the dynamics of the system. The interconnections of the linear graph that relate each element are described by the continuity and constraint equations listed below.

$$i_c = i_{R_1} \tag{F-7}$$

$$i_{R_2} = i_{R_1} \tag{F-8}$$

$$V_{R_1} = V_{in} - V_{R_2} - V_C \tag{F-9}$$

Because resistors of the same nominal value were used in the real system, the following statement further simplifies the model:

$$R_1 = R_2 = R \tag{F-10}$$

The state space model and output equation were calculated by combining the equations listed above.

$$\frac{d}{dt}[V_C] = \left[-\frac{1}{2RC}\right]V_C + \left[\frac{1}{2RC}\right]V_{in} \tag{F-11}$$

$$V_C = [1]V_C + [0]V_C \tag{F-12}$$

Plugging the A, B, C, and D matrices into Equation F-3 results in the transfer function describing the filter. From the transfer function, it is simple to determine an expression of the filter cutoff frequency.

$$T(s) = \frac{1}{2RCs + 1} \tag{F-13}$$

$$f_c = \frac{1}{4\pi RC} \tag{F-14}$$

The filter was designed using Equation F-14 and simulated using the filter transfer function described in Equation F-14. The simulation and Bode plot used to evaluate the performance of the filter will be presented later.

The transfer function of the second-order filter was derived in a similar fashion, whose schematic, linear graph, and normal tree are displayed in Figures F-3 and F-4. Again, the model does not include a load across the output of the filter because of the voltage buffers on the differential output of the filter.

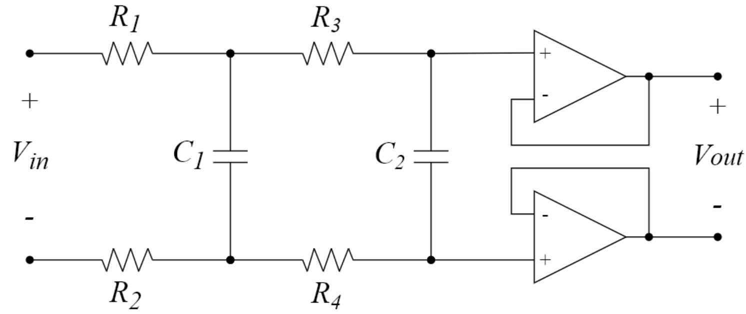


Figure F-3: Second-order active low-pass filter schematic.

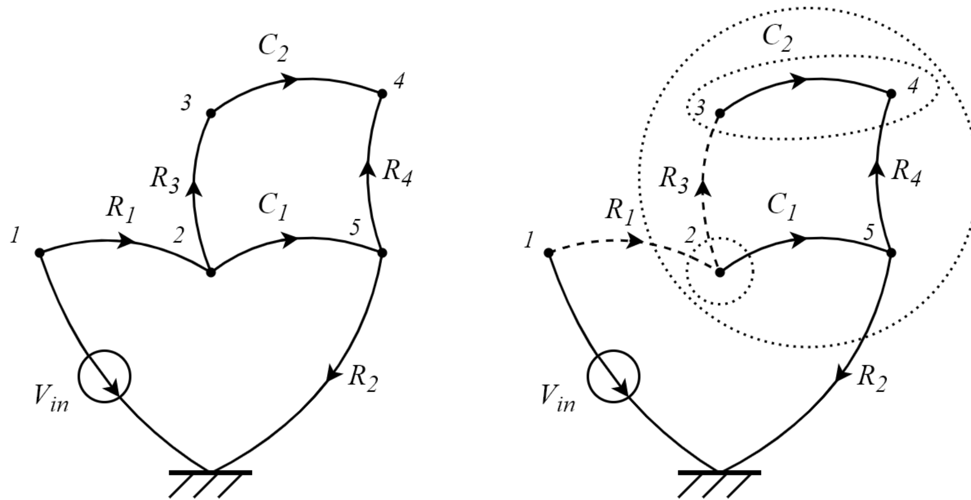


Figure F-4: Second-order low-pass filter linear graph (left) and normal tree (right).

The second-order filter model contains four resistors modeled as D-type elements, two capacitors modeled as A-type energy storage elements, and a voltage input. The elemental equations describing the dynamics of each component are listed below.

$$\frac{dV_{C_1}}{dt} = \frac{1}{C_1} i_{C_1} \tag{F-15}$$

$$\frac{dV_{C_2}}{dt} = \frac{1}{C_2} i_{C_2} \tag{F-16}$$

$$i_{R_1} = \frac{1}{R_1} V_{R_1} \tag{F-17}$$

$$i_{R_2} = \frac{1}{R_2} V_{R_2} \tag{F-18}$$

$$V_{R_3} = R_3 i_{R_3} \tag{F-19}$$

$$V_{R_4} = R_4 i_{R_4} \tag{F-20}$$

The voltages across the capacitors V_{C_1} and V_{C_2} are the state variables required to describe the dynamics of the system. The continuity and constraint equations of the system are listed below.

$$i_{C_1} = i_{R_1} - i_{R_2} \tag{F-21}$$

$$i_{C_2} = i_{R_2} \tag{F-22}$$

$$i_{R_4} = i_{R_2} \tag{F-23}$$

$$i_{R_3} = i_{R_1} \tag{F-24}$$

$$V_{R_1} = V_{in} - V_{C_1} - V_{R_3} \tag{F-25}$$

$$V_{R_2} = V_{C_1} - V_{C_2} - V_{R_4} \tag{F-26}$$

The following statements further simplify the model:

$$R_1 = R_2 \tag{F-27}$$

$$R_3 = R_4 \tag{F-28}$$

The state space model and output equation were calculated by combining the equations listed above.

$$\frac{d}{dt}\begin{bmatrix} V_{C_1} \\ V_{C_2} \end{bmatrix} = \begin{bmatrix} \frac{R_4}{C_1 R_2 (R_2 + R_4)} - \frac{1}{C_1 R_1 + C_1 R_3} - \frac{1}{C_1 R_2} & \frac{1}{C_1 R_2} - \frac{R_4}{C_1 R_2 (R_2 + R_4)} \\ \frac{1}{C_2 R_2 + C_2 R_4} & -\frac{1}{C_2 R_2 + C_2 R_4} \end{bmatrix} \begin{bmatrix} V_{C_1} \\ V_{C_2} \end{bmatrix} + \begin{bmatrix} \frac{1}{C_1 R_1 + C_1 R_3} \\ 0 \end{bmatrix} V_{in} \tag{F-29}$$

$$V_{C_2} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} V_{C_1} \\ V_{C_2} \end{bmatrix} + [0] V_{in} \tag{F-30}$$

Plugging the A, B, C, and D matrices into Equation F-3 results in the transfer function describing the filter. An expression for the filter cutoff frequency was determined by observation of the transfer function.

$$T(s) = \frac{\frac{1}{4R_1R_2C_1C_2}}{s^2 + \left(\frac{1}{2R_1C_1} + \frac{1}{2R_2C_2}\right)s + \frac{1}{4R_1R_2C_1C_2}} \quad \text{(F-31)}$$

$$f_c = \frac{1}{4\pi\sqrt{R_1R_2C_1C_2}} \quad \text{(F-32)}$$

A simulation was conducted in MATLAB™ with a cutoff frequency of f_c = 5.3 kHz to evaluate the performance of both filters. The following Simulink™ block diagrams and MATLAB™ script were used to run the simulation. Bode plots as well as plots that detail the attenuating capabilities of both filters were generated and are displayed below.

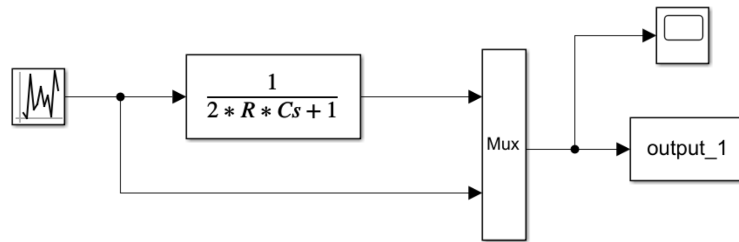


Figure F-5: Simulink™ block diagram of first-order filter simulation.

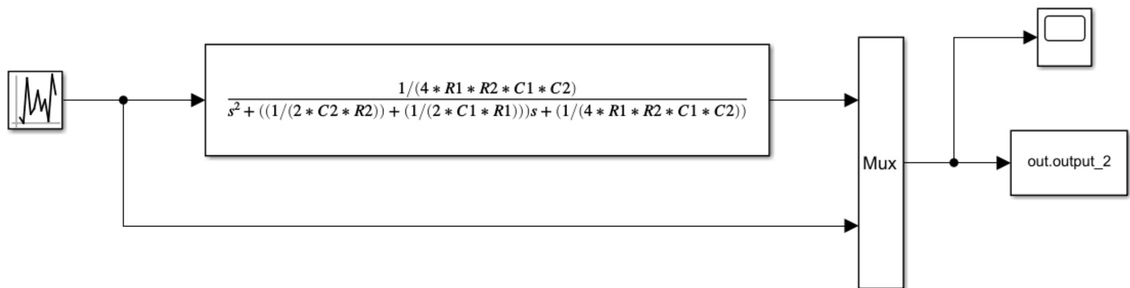


Figure F-6: Simulink™ block diagram of second-order filter simulation.

```
% ////////////////////////////////////////////////////////////////////////
%% Filter Simulations
% Zach Wilson, 4/6/19
%
% ////////////////////////////////////////////////////////////////////////
%%
clear all; clc; close all;

% ////////////////////////////////////////////////////////////////////////
%% First Order Low-pass Filter
% ////////////////////////////////////////////////////////////////////////

% Constants
R = 5000;                % Resistance
C = 0.003e-6;            % Capacitance
noise_amp = 0.02;        % Noise amplitude
noise_freq = 1/100000;   % Noise frequency

w_n_1 = (1/(2*R*C))/(2*pi);
display(w_n_1);

% Run the simulink file.
sim('first_order_RC_filter');

% Plot the output.
figure;
hold on;
plot(tout,output_1(:,2));
plot(tout,output_1(:,1));
title('RC Filter output');
ylim([-0.05, 0.05]);
xlabel('Time [s]');
ylabel('Output [V]');
legend('Noisy Data','Filtered Data');
hold off;

% Bode plot of transfer function
sys_1 = tf(1,[R*C 1]);
figure;
hold on;
bode(sys_1);
title(' ');
hold off;

% ////////////////////////////////////////////////////////////////////////
%% Second Order RC Filter
% ////////////////////////////////////////////////////////////////////////

% Constants
R1 = 5000;              % Resistance first stage
R2 = 5000;              % Resistance second stage
C1 = 0.003e-6;          % Capacitance first stage
C2 = 0.003e-6;          % Capacitance second stage
noise_amp = 0.02;       % Noise amplitude
noise_freq = 1/100000;  % Noise frequency

w_n_1 = (1/sqrt(4*R1*R2*C1*C2))/(2*pi);
display(w_n_1);

% Run the simulink file
sim ('second_order_RC_filter.slx');

% Plot the output.
figure;
hold on;
plot(ans.tout,ans.output_2(:,2));
plot(ans.tout,ans.output_2(:,1));
title('RC Filter output');
ylim([-0.05, 0.05]);
xlabel('Time [s]');
ylabel('Output [V]');
```

```
legend('Noisy Data','Filtered Data');
hold off;

% Bode plot of transfer function
sys_2 = tf(1/(4*R1*R2*C1*C2),[1 (1/(2*C2*R2))+(1/(2*C1*R1)) (1/(4*R1*R2*C1*C2))]);
figure;
hold on;
bode(sys_2);
title(' ');
hold off;
```

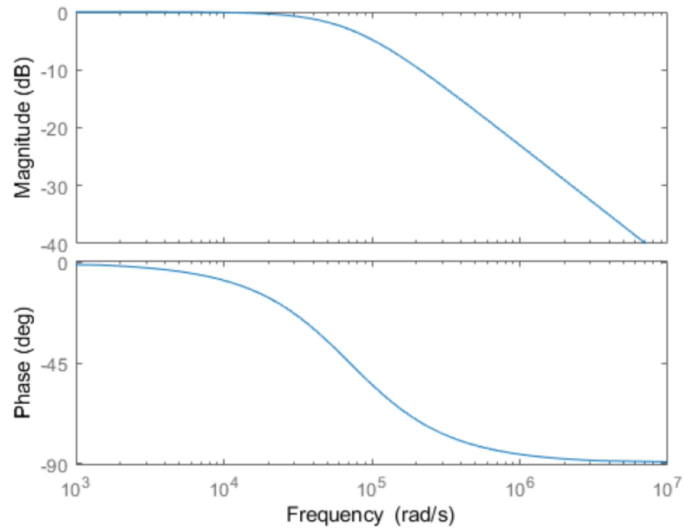


Figure F-7: Bode Diagram for first-order low-pass filter.

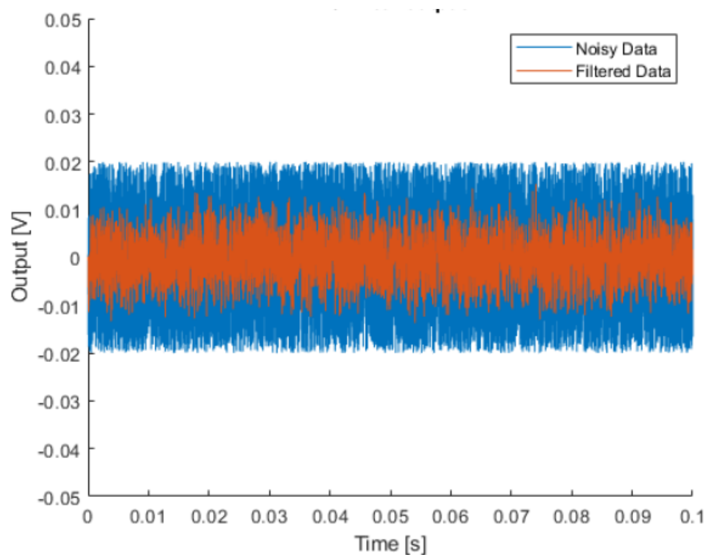


Figure F-8: First-order low-pass filter response with noisy input.

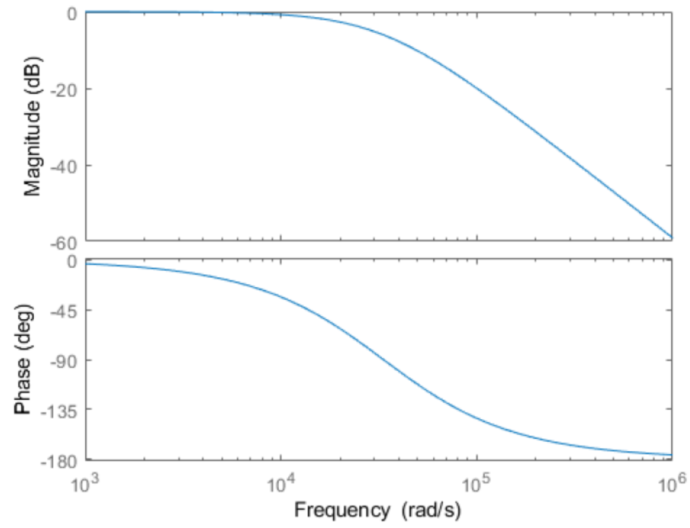


Figure F-9: Bode Diagram for second-order low-pass filter.

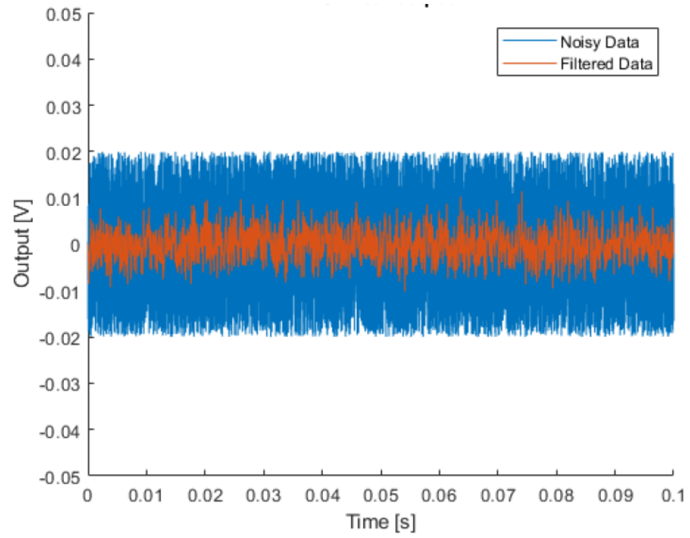


Figure F-10: Second-order low-pass filter response with noisy input.

G. DYNAMIC DATA ACQUISITION SYSTEM C++ DRIVERS

```
/*************************************************************************/
/** @file serial_port.cpp
 *  @brief Serial class header file.
 *
 *  @details This header file defines the serial_port class. It includes
 *  methods and operators for writing and reading characters to/from
 *  a chosen USART port.
 *
 *   Contributors:
 *      - Zach Wilson 12/7/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/7/2018 - initial release.
 *       - 2/2/2019 - implemented circular buffer class for transfer and
 *           receive buffers.
 *
 *  Created: 12/7/2018 10:54:46 PM
 */
/*************************************************************************/


#ifndef SERIAL_PORT_H_
#define SERIAL_PORT_H_

/* Included files. */
#include <avr/io.h>
#include <stdio.h>
#include <avr/interrupt.h>
#include "defines.h"
#include "stream_base.h"
#include "semaphore.h"
#include "circular_buffer.h"

//////////////////////////////////////////////////////////////////////////
// Check to see if the macro F_CPU is defined.
#ifndef  F_CPU
    #warning "CPU speed not defined."

    /** This serial interface requires the F_CPU macro is defined
     *  somewhere.
     */
    #define  F_CPU  32000000
#endif

//////////////////////////////////////////////////////////////////////////
/** Size of transfer buffer.
 */
#define  SERIAL_PORT_TRANSFER_SIZE  400

//////////////////////////////////////////////////////////////////////////
/** Size of receive buffer.
 */
#define  SERIAL_PORT_RECEIVE_SIZE  50

//////////////////////////////////////////////////////////////////////////
/** Declare the external store string function defined in string.h.
 */
extern void storeString (const char *, char *);

//////////////////////////////////////////////////////////////////////////
/** Declare the external integer to string conversion function defined in
 *  string.h.
 */
extern char * intToString (uint32_t, char *);

//////////////////////////////////////////////////////////////////////////
/** Declare the external float to string conversion function defined in
 *  string.h.
 */
extern void floatToString (float, char *, uint8_t);


//////////////////////////////////////////////////////////////////////////
/** @brief Serial Interface class.
 *  @details This class provides a USART serial interface through a user-
 *  specified port and baud rate. This class uses the receive complete and
 *  data register empty interrupts on USARTE0.
 */
class serial_port : public stream_base, public binary_semaphore
{
    private:
        /** 32-bit storage for desired baud rate.
         */
        uint32_t baud;

        /** 16-bit storage for the number to be put into the period
```

```
         *  register to set the timer period.
         */
        uint16_t BAUDRATE;

        /** Pointer to a USART port struct.
         */
        USART_t *p_USART;

        /** C-style string for serial port name.
         */
        char name [11];

        /** @brief Transmit data.
         *  @details This method places a string of data into the transfer
         *  buffer. To begin transmitting characters, this method places the
         *  first character of the string into the USART data transfer register.
         *  The rest of the string will be transmitted in interrupts.
         *  @param p_string Pointer to a string of characters to be transmitted.
         */
        void transmit_data (const char *);

        /** @brief Put character.
         *  @details This method puts a single character into the
         *  transfer buffer. The character that the head pointer of
         *  the buffer is pointing at is placed in the USART data
         *  register during the transmit complete interrupt. This
         *  method should only be called by the overloaded << operator
         *  because it doesn't raise the data register empty flag. If the
         *  transfer queue is full, then this function waits until the
         *  queue has room before inserting the character.
         *  @param character Character to be printed.
         */
        inline void put_char (char);

    public:
        /** Pointer to the serial port name string.
         */
        const char *p_name;

        /** Circular buffer for transferring characters through the serial port.
         *  This object is a static member of the class so it can be utilized in an
         *  interrupt service routine outside the class.
         */
        static circular_buffer <volatile char, SERIAL_PORT_TRANSFER_SIZE> transfer;

        /** Circular buffer for receiving characters through the serial port. This
         *  object is a static member of the class so it can be utilized in an
         *  interrupt service routine outside the class.
         */
        static circular_buffer <volatile char, SERIAL_PORT_RECEIVE_SIZE> receive;

        /** @brief Default constructor.
         *  @details This constructor instantiates an instance of the
         *  class and saves a pointer to a USART struct and calculates
         *  the baud rate setting to be put in the CTRLA register.
         *  @param baud Baud rate for serial communication.
         *  @param USART Pointer to a USART port struct.
         *  @param serial_name Pointer to the serial port name string.
         */
        serial_port (uint32_t, USART_t *, const char *);

        /** @brief Configure serial port.
         *  @details This function configures the USART port on the
         *  ATXMEGA128A4U microcontroller.
         */
        void configure (void);

        /** @brief Initialize serial port.
         *  @details This function enables the interrupts that drive the
         *  USART port and sends an indication through the port to confirm
         *  the object was created successfully.
         */
        void initialize (void);

        /** @brief Check buffer.
         *  @details This method checks to see if a character has
         *  been received and is ready to be read from the receive
         *  buffer.
         *  @return Boolean indicating if the receive buffer contains an
         *  unread character (true = yes).
         */
        inline bool check_buf (void);

        /** @brief Get character.
         *  @details This method reads a single character from the
         *  receive buffer. Call check_buf () before calling this
         *  function, otherwise a NULL character will be returned
         *  if the receive buffer doesn't currently hold data.
         *  @return Character read from the receive buffer.
         */
        inline char get_char (void);
};
```

```
//////////////////////////////////////////////////////////////////////////
/** @brief Put character.
 *  @details This method puts a single character into the transfer buffer.
 *  The character that the head pointer of the buffer is pointing at is
 *  placed in the USART data register during the transmit complete
 *  interrupt. This method should only be called by the overloaded <<
 *  operator because it doesn't raise the data register empty flag. If the
 *  transfer queue is full, then this function waits until the queue has
 *  room before inserting the character.
 *  @param character Character to be printed.
 */
inline void serial_port :: put_char (char character)
{
    // If the transfer buffer is full, wait until the buffer
    // has room before pushing another character. This could
    // cause an infinite loop if the data register empty
    // interrupts are disabled or have not been triggered
    // for some reason.
    while (transfer.full ()) {}

    // Push the desired character onto the transfer buffer.
    transfer.push_data (character);

    // Non-interrupt method of printing characters.
    /*
    // Wait while TX status flag is low indicating that
    // USART is not ready to send data.
    while ((p_USART->STATUS & USART_DREIF_bm) == 0) {}

    // Clear the TXC bit by writing it high to indicate that
    // the USART is currently transferring data and put the
    // desired character in the data register to print it.
    p_USART->STATUS |= USART_TXCIF_bm;
    p_USART->DATA = *p_char;
    */
}


//////////////////////////////////////////////////////////////////////////
/** @brief Check buffer.
 *  @details This method checks to see if a character has been received
 *  and is ready to be read from the receive buffer.
 *  @return Boolean indicating if the receive buffer contains an unread
 *  character (true = yes).
 */
inline bool serial_port :: check_buf (void)
{
    return !receive.empty ();
}


//////////////////////////////////////////////////////////////////////////
/** @brief Get character.
 *  @details This method reads a single character from the receive buffer.
 *  Call check_buf () before calling this function, otherwise a NULL
 *  character will be returned if the receive buffer doesn't currently
 *  hold data.
 *  @return Character read from the receive buffer.
 */
inline char serial_port :: get_char (void)
{
    // Only read from the buffer if the receive buffer contains a
    // character. Return NULL character if the receive buffer is
    // empty.
    return !receive.empty () ? receive.pull_data () : NULL;
}


#endif /* SERIAL_PORT_H_ */
```

```
/**********************************************************************/
/** @file serial_port.cpp
 *  @brief Serial main class file.
 *
 *   Contributors:
 *      - Zach Wilson 12/7/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/7/2018 - initial release.
 *
 *  Created: 12/7/2018 11:20:28 PM
 */
/**********************************************************************/


/* Included files. */
#include "serial_port.h"

////////////////////////////////////////////////////////////////////////
/** Circular buffer for transferring characters through the serial port.
 *  This object is a static public member of the class so it can be
 *  utilized in an interrupt service routine.
 */
circular_buffer <volatile char, SERIAL_PORT_TRANSFER_SIZE> serial_port :: transfer;

////////////////////////////////////////////////////////////////////////
/** Circular buffer for receiving characters through the serial port. This
 *  object is a static public member of the class so it can be utilized in
 *  an interrupt service routine.
 */
circular_buffer <volatile char, SERIAL_PORT_RECEIVE_SIZE> serial_port :: receive;


////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the class and
 *  saves a pointer to a USART struct and calculates the baud rate setting
 *  to be put in the CTRLA register.
 *  @param baud Baud rate for serial communication.
 *  @param USART Pointer to a USART port struct.
 *  @param serial_name Pointer to the serial port name string.
 */
serial_port :: serial_port (uint32_t _baud, USART_t *USART, const char *serial_name)
                      : stream_base (), binary_semaphore ()
{
    baud = _baud;
    p_USART = USART;
    p_name = &name [0];
    BAUDRATE = ((F_CPU / (baud * 16)) - 1);
    storeString (serial_name, &name [0]);
}


////////////////////////////////////////////////////////////////////////
/** @brief Configure serial port.
 *  @details This function configures the USART port on the ATXMEGA128A4U
 *  microcontroller.
 */
void serial_port :: configure (void)
{
    // Make sure the data register is empty before setup.
    p_USART->DATA = 0x00;

    // Set TX pin as output and high.
    PORTE.OUT |= (1 << PIN3_bp);
    PORTE.DIR |= (1 << PIN3_bp);

    // Set RX pin as input.
    PORTE.DIR &= ~(1 << PIN2_bp);

    // Lower XCK pin and set as output (asynchronous mode).
    PORTE.OUT &= ~(1 << PIN1_bp);
    PORTE.DIR |= (1 << PIN1_bp);

    // Select desired baud rate.
    if (baud == 0)
    {
        // Baudrate of 115200.
        p_USART->BAUDCTRLB = (-1 << USART_BSCALE0_bp);
        p_USART->BAUDCTRLA = (33 << USART_BSEL0_bp);
    }
    else
    {
        // Baudrate selected by user.
        p_USART->BAUDCTRLB = BAUDRATE;
        p_USART->BAUDCTRLA = (BAUDRATE >> 8);
    }

    // Enable TX pin and RX for transmission.
    p_USART->CTRLB = USART_TXEN_bm | USART_RXEN_bm;

    // Select 8-bit data size and set asynchronous mode.
```

```
    p_USART->CTRLC = USART_CHSIZE0_bm | USART_CHSIZE1_bm;

    // Enable low-level interrupts for receive and data register
    // empty so the program isn't stuck printing characters for
    // too long.
    p_USART->CTRLA = USART_DREINTLVL0_bm | USART_RXCINTLVL0_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Initialize serial port.
 *  @details This function enables the interrupts that drive the USART
 *  port and sends an indication through the port to confirm the object
 *  was created successfully. This method is not necessary for the serial
 *  port to function properly.
 */
void serial_port :: initialize (void)
{
    // Send indication through serial port.
    *this << clearscreen << "Serial port on " << p_name << " initialized." << endl << endl;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Transmit data.
 *  @details This method places a string of data into the transfer buffer.
 *  To begin transmitting characters, this method places the first
 *  character of the string into the USART data transfer register. The
 *  rest of the string will be transmitted in interrupts.
 *  @param p_string Pointer to a string of characters to be transmitted.
 */
void serial_port :: transmit_data (const char *p_string)
{
    // If the transfer queue is empty and a character is not currently
    // transmitting, place the first character in the DATA register.
    if (transfer.empty () && (p_USART->STATUS & USART_DREIF_bm))
    {
        p_USART->DATA = *p_string++;

        // If there are no more characters, exit.
        if (*p_string == NULL) return;
    }

    // Put the character in the transfer buffer and increment the pointer.
    // Repeat until it points to a NULL character, indicating the end of
    //the string.
    do
    {
        put_char (*p_string++);
    }
    while (*p_string != NULL);
}


//////////////////////////////////////////////////////////////////////////
/** @Brief Transmit register empty interrupt.
 *  @details This interrupt is triggered when the transmit register is
 *  empty after a character transmit has been completed and the USART is
 *  ready to transmit another character. Another character is transmitted
 *  in this interrupt if one is available in the transfer queue.
 */
ISR (USARTE0_DRE_vect)
{
    // If queue is empty no value is pulled and the data empty interrupt
    // flag is cleared to stop sending characters.
    if (serial_port :: transfer.empty ())
    {
        USARTE0.STATUS |= USART_DREIF_bm;
        return;
    }

    // Pull the next character from the transfer buffer into the transmit
    // register.
    USARTE0.DATA = serial_port :: transfer.pull_data ();
}

//////////////////////////////////////////////////////////////////////////
/** @brief Receive complete interrupt.
 *  @details This interrupt is triggered when a character is received over
 *  the USART. The character is placed in the receive buffer if there is
 *  room, and discarded if there isn't.
 */
ISR (USARTE0_RXC_vect)
{
    // If the receive queue is full no value is pushed to the receive
    // buffer.
    if (serial_port :: receive.full ()) return;

    // Push the character in the receive data register into the receive
    // buffer.
    serial_port :: receive.push_data (static_cast <char> (USARTE0.DATA));
}
```

```
/***********************************************************************/
/** @file SPI.h
 *  @brief Serial peripheral interface class header file.
 *
 *  @details This header file defines the serial peripheral interface
 *  class. It includes methods for acquiring data from external devices
 *  via the SPI interface on the ATXMEGA128A4U microcontroller. This class
 *  is interrupt driven and inherits methods/members from the buffer class.
 *  Currently, this class only supports configurations where the
 *  ATXMEGA128A4U is the master and the external device is the slave.
 *
 *   Contributors:
 *      - Zach Wilson 2/4/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 2/4/2019 - initial release.
 *
 *  Created: 2/4/2019 12:26:38 AM
 */
/***********************************************************************/


#ifndef SPI_H_
#define SPI_H_

/* Included files. */
#include <avr/io.h>
#include <avr/interrupt.h>
#include "semaphore.h"
#include "circular_buffer.h"
#include "serial_port.h"

////////////////////////////////////////////////////////////////////////
// Check to see if the macro F_CPU is defined.
#ifndef  F_CPU
    #warning "CPU frequency not defined."

    /** This class requires the F_CPU macro is defined somewhere.
     */
    #define F_CPU  32000000
#endif

////////////////////////////////////////////////////////////////////////
/** Defines the buffer size for the SPI class.
 */
#define  SPI_BUFFER_SIZE  50

////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the interface name.
 */
extern void storeString (const char *, char *);

////////////////////////////////////////////////////////////////////////
/** Enum defining the read and write macros indicating the direction of
 *  transmission.
 */
typedef enum  SPI_INTERFACE_COMMAND_ENUM
{
    /** Read macro for receiving a byte over SPI.
     */
    SPI_READ = 0,

    /** Write macro for sending a byte over SPI.
     */
    SPI_WRITE,
} SPI_INTERFACE_COMMAND_t;


////////////////////////////////////////////////////////////////////////
/** Enum defining the possible states of the SPI interface. These are used
 *  to determine if the interface is currently transmitting or receiving a
 *  byte of data.
 */
typedef enum  SPI_INTERFACE_STATES_ENUM
{
    /** State when the interface is disabled and not transmitting or
     *  receiving data.
     */
    SPI_DONE = 0,

    /** State where the interface is transceiving data.
     */
    SPI_TRANSCEIVING,

    /** State where the interface is writing data and not expecting a
     *  response.
     */
    SPI_WRITING,
} SPI_INTERFACE_STATES_t;
```

```
////////////////////////////////////////////////////////////////////////////
/** Enum defining the master and slave operating modes of the SPI
 *  interface.
 */
typedef enum  SPI_MASTER_SELECT_ENUM
{
    /** SPI master mode.
     */
    SPI_MASTER,

    /** SPI slave mode.
     */
    SPI_SLAVE,
} SPI_MASTER_SELECT_t;


////////////////////////////////////////////////////////////////////////////
/** Enum defining the different clock prescale values for SPI operation.
 */
typedef enum  SPI_PRESCALER_ENUM
{
    /** Clock ticks / 2.
     */
    SPI_DIV2 = 0,

    /** Clock ticks / 4.
     */
    SPI_DIV4,

    /** Clock ticks / 8.
     */
    SPI_DIV8,

    /** Clock ticks / 16.
     */
    SPI_DIV16,

    /** Clock ticks / 32.
     */
    SPI_DIV32,

    /** Clock ticks / 64.
     */
    SPI_DIV64,

    /** Clock ticks / 128.
     */
    SPI_DIV128,
} SPI_PRESCALER_ENUM_t;


////////////////////////////////////////////////////////////////////////////
/** Enum defining the different SPI transfer modes of operation.
 */
typedef enum  SPI_TRANSFER_MODES_ENUM
{
    /** SPI mode 0. Leading Edge: rising, sample. Trailing Edge: falling,
     *  setup.
     */
    SPI_MODE0 = 0,

    /** SPI mode 1. Leading Edge: rising, setup. Trailing Edge: falling,
     *  sample.
     */
    SPI_MODE1,

    /** SPI mode 2. Leading Edge: falling, sample. Trailing Edge: rising,
     *  setup.
     */
    SPI_MODE2,

    /** SPI mode 3. Leading Edge: falling, setup. Trailing Edge: rising,
     *  sample.
     */
    SPI_MODE3,
} SPI_TRANSFER_MODES_t;


////////////////////////////////////////////////////////////////////////////
/** Enum defining the different SPI data order modes of operation.
 */
typedef enum  SPI_DATA_ORDER
{
    /** Least significant byte first mode.
     */
    SPI_LSBFIRST = 0,

    /** Most significant byte first mode.
     */
    SPI_MSBFIRST,
} SPI_DATA_ORDER_t;
```

```
//////////////////////////////////////////////////////////////////////////
/** SPI data struct. Defines the data type used for the transfer queue in
 *  the SPI control struct.
 */
typedef struct
{
    /** Byte for storing data.
     */
    volatile uint8_t data;

    /** Boolean indicating the SPI operation associated with the byte.
     */
    volatile SPI_INTERFACE_COMMAND_t operation;
} SPI_DATA_STRUCT_t;


//////////////////////////////////////////////////////////////////////////
/** SPI control struct. Struct used for controlling and interfacing with
 *  an SPI interface through an interrupt service routine.
 */
typedef struct
{
    /** Pointer to the port of the chip select pin.
     */
    volatile PORT_t *p_cs_port;

    /** Chip select pin bitmask.
     */
    volatile uint8_t cs_bitmask;

    /** Circular buffer used for queuing data to be transmitted through
     *  the interface.
     */
    circular_buffer <SPI_DATA_STRUCT_t, SPI_BUFFER_SIZE> transfer_buffer;

    /** Circular buffer used as storage for incoming data from the
     *  interface.
     */
    circular_buffer <volatile uint8_t, SPI_BUFFER_SIZE> receive_buffer;

    /** Number indicating the state of the SPI interface.
     */
    volatile SPI_INTERFACE_STATES_t state;

    /** Temporary storage for reading/writing data.
     */
    SPI_DATA_STRUCT_t temp;
} SPI_CONTROL_STRUCT_t;


//////////////////////////////////////////////////////////////////////////
/** @brief Serial peripheral interface class.
 *  @details This class provides methods for interfacing with the SPI on
 *  ports C and D on the ATXMEGA128A4U microcontroller.  It includes
 *  methods for acquiring data from external devices 8-bits at a time.
 *  This SPI class is interrupt driven. Acquired data is placed in a
 *  buffer and remains there until fetched by another thread.
 */
class SPI : public binary_semaphore
{
    private:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** Pointer to an SPI interface.
         */
        SPI_t *p_SPI;

        /** Pointer to the port of the SPI interface.
         */
        PORT_t *p_PORT;

        /** Pointer to an SPI control struct.
         */
        SPI_CONTROL_STRUCT_t *p_control;

        /** C-style string for interface name.
         */
        char name [11];

    public:
        /** Pointer to the interface name.
         */
        const char *p_name;

        #ifdef  SPI_PORTC_FLAG
            /** SPI control struct for the interface on port C.
             */
            static SPI_CONTROL_STRUCT_t SPI_C_control;
        #endif
```

```
#ifdef  SPI_PORTD_FLAG
    /** SPI control struct for the interface on port D.
     */
    static SPI_CONTROL_STRUCT_t SPI_D_control;
#endif

/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the
 *  SPI class and saves a pointer to a SPI interface object and
 *  the port name.
 *  @param _p_SPI Pointer to a SPI interface object.
 *  @param port_name Pointer to the SPI port name.
 */
SPI (SPI_t *, const char *);

/** @brief Initialize SPI interface
 *  @details This method initializes the interface by saving a
 *  pointer to the SPI interface so the class has access to the
 *  registers. It also configures the interrupts that drive the
 *  interface and sets the desired SPI transmission frequency.
 *  @param _p_SPI Pointer to a serial port object.
 *  @param prescale Desired prescaler for SPI transmission
 *  frequency according to the xmega au manual.
 */
void initialize (stream_base *, SPI_MASTER_SELECT_t);

/** @brief Set SPI mode.
 *  @details This method sets the transmission frequency and the
 *  mode of transmission, as well as enable the SPI interface. It
 *  should be called right before calling the transmission () and
 *  request_bytes functions.
 *  @param prescale Desired prescaler for SPI transmission.
 *  @param mode Mode of SPI transmission.
 *  @param bit_order Order in which data is received.
 */
void set_mode (SPI_PRESCALER_ENUM_t, SPI_TRANSFER_MODES_t, SPI_DATA_ORDER_t);

/** @brief Transfer bytes in ISR.
 *  @details This method begins the transmission of all bytes in the
 *  SPI buffer. The data_ready () method will return true if all bytes
 *  have been transferred/received. This method should be used when
 *  SPI interrupts are desired for handling the data transfer.
 *  @param _p_cs_port Pointer to the port of the chip select pin.
 *  @param _cs_bitmask Chip select pin bitmask.
 */
void transfer_ISR (PORT_t *, uint8_t);

/** @brief Transfer data.
 *  @details This method sends and retrieves one byte over the SPI
 *  interface. It doesn't use interrupts, and the chip select pin
 *  must manually be lowered/raised before and after the transmission.
 *  @param byte Data to be transferred.
 *  @return Byte received from the slave device.
 */
inline uint8_t transfer (uint8_t);

/** @brief Push byte to transfer queue.
 *  @details This method pushes a single byte to the SPI transfer queue
 *  to be transferred to a slave device. The SPI communication will be
 *  initiated with a call to the transfer_ISR () method.
 *  @param byte Byte to be written.
 *  @param operation Boolean indicating if the operation associated with
 *  the byte is a read or write.
 */
inline void push_byte (uint8_t, SPI_INTERFACE_COMMAND_t);

/** @brief Data ready status.
 *  @details This method checks to see if all of the bytes have been
 *  transferred and all bytes have been received over interrupts.
 *  @return Boolean indicating if the transfer is complete.
 */
inline bool data_ready (void);

/** @brief Data available status.
 *  @details This method checks to see if any data is available in
 *  the SPI receive buffer.
 *  @return Number indicating how many bytes are available.
 */
inline uint16_t data_available (void);

/** @brief Pull byte from the receive queue.
 *  @details This method pulls a single byte off the receive queue.
 *  @return Byte pulled from the queue.
 */
inline uint8_t pull_byte (void);

/** @brief Flush receive buffer.
 *  @details This method flushes the receive buffer of the SPI
 *  interface.
 */
inline void flush (void);
```

```
};

//////////////////////////////////////////////////////////////////////////
/** @brief Transfer data.
 *  @details This method sends and retrieves one byte over the SPI
 *  interface. It doesn't use interrupts, and the chip select pin must
 *  manually be lowered/raised before and after calling the function.
 *  @param data Data to be transferred.
 */
inline uint8_t SPI :: transfer (uint8_t data)
{
    // Temporarily disable interrupts so the transmission complete bit
    // isn't disabled in an ISR.
    p_SPI->INTCTRL = 0x00;

    // Send the data and wait until the transmission is complete.
    p_SPI->DATA = data;
    while (!(p_SPI->STATUS & SPI_IF_bm)) {}

    // Read the data register and re-enable interrupts. Return the
    // received byte.
    data = p_SPI->DATA;
    p_SPI->INTCTRL = SPI_INTLVL0_bm | SPI_INTLVL1_bm;
    return data;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Push byte to transfer queue.
 *  @details This method pushes a single byte to the SPI transfer queue
 *  to be transferred to a slave device. The SPI communication will be
 *  initiated with a call to the transfer_ISR () method.
 *  @param byte Byte to be written.
 *  @param operation Boolean indicating if the operation associated with
 *  the byte is a read or write.
 */
inline void SPI :: push_byte (uint8_t data, SPI_INTERFACE_COMMAND_t operation)
{
    p_control->temp.data = data;
    p_control->temp.operation = operation;
    p_control->transfer_buffer.push_data (p_control->temp);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Data ready status.
 *  @details This method checks to see if all of the bytes have been
 *  transferred and all bytes have been received.
 *  @return Boolean indicating if the transfer is complete.
 */
inline bool SPI :: data_ready (void)
{
    return p_control->state == SPI_DONE;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Data available status.
 *  @details This method checks to see if any data is available in the SPI
 *  receive buffer.
 *  @return Number indicating how many bytes are available.
 */
inline uint16_t SPI :: data_available (void)
{
    return p_control->receive_buffer.buf_capacity ();
}


//////////////////////////////////////////////////////////////////////////
/** @brief Pull byte from the receive queue.
 *  @details This method pulls a single byte off the receive queue.
 *  @return Byte pulled from the queue.
 */
inline uint8_t SPI :: pull_byte (void)
{
    return p_control->receive_buffer.pull_data ();
}


//////////////////////////////////////////////////////////////////////////
/** @brief Flush receive buffer.
 *  @details This method flushes the receive buffer of the SPI interface.
 */
inline void SPI :: flush (void)
{
    p_control->receive_buffer.flush_buffer ();
}


//////////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTC_FLAG
```

```
    /** Instantiate an instance of the SPI class in static memory on port
     *  C. All files that include this header file will have access to
     *  this object.
     */
    static SPI SPI_C (&SPIC, "Port C");
#endif

//////////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTD_FLAG
    /** Instantiate an instance of the SPI class in static memory on port
     *  D. All files that include this header file will have access to
     *  this object.
     */
    //static SPI SPI_D (&SPID, "Port D");
#endif


#endif /* SPI_H_ */
```

```
/***********************************************************************/
/** @file SPI.cpp
 *  @brief Main class file.
 *
 *   Contributors:
 *      - Zach Wilson 2/4/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 2/4/2019 - initial release.
 *
 *  Created: 2/4/2019 12:45:48 AM
 */
/***********************************************************************/


/* Included files. */
#include "SPI.h"

//////////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTC_FLAG
    /** SPI control struct for the interface on port C.
     */
    SPI_CONTROL_STRUCT_t SPI :: SPI_C_control;
#endif

//////////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTD_FLAG
    /** SPI control struct for the interface on port D.
     */
    SPI_CONTROL_STRUCT_t SPI :: SPI_D_control;
#endif


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the SPI class
 *  and saves a pointer to a SPI interface object and the port name.
 *  @param _p_SPI Pointer to a SPI interface object.
 *  @param port_name Pointer to the SPI port name.
 */
SPI :: SPI (SPI_t *_p_SPI, const char *port_name) : binary_semaphore ()
{
    p_SPI = _p_SPI;
    p_name = &name [0];
    storeString (port_name, &name [0]);

    // Save a pointer to the port and control struct of the SPI interface.
    // There are only two choices, so if port C isn't chosen then the only
    // other option is port D.
    if (p_SPI == &SPIC)
    {
        p_PORT = &PORTC;
        #ifdef  SPI_PORTC_FLAG
            p_control = &SPI_C_control;
        #endif
    }
    else
    {
        p_PORT = &PORTD;
        #ifdef  SPI_PORTD_FLAG
            p_control = &SPI_D_control;
        #endif
    }
}


//////////////////////////////////////////////////////////////////////////
/** @brief Initialize SPI interface
 *  @details This method initializes the interface by saving a pointer to
 *  the SPI interface so the class has access to the registers. It
 *  configures the pins accordingly depending on the desired SPI mode and
 *  also configures the interrupts that drive the interface.
 *  @param serial Pointer to a serial port object.
 *  @param prescale Desired prescaler for SPI transmission frequency
 *  according to the xmega au manual.
 */
void SPI :: initialize (stream_base *serial, SPI_MASTER_SELECT_t spi_mode)
{
    p_serial = serial;

    // Initialize SPI control state.
    p_control->state = SPI_DONE;
    p_control->temp.data = 0;
    p_control->temp.operation = SPI_READ;

    // Set relevant SPI pins based on which mode we are in.
    if (spi_mode == SPI_MASTER)
    {
        // Pull default SS pin high so the SPI interface works correctly.
        p_PORT->OUT |= PIN4_bm;

        // Configure the MOSI, SCK, and SS pins as outputs and the MISO pin as input.
```

```
        p_PORT->DIR |= PIN7_bm | PIN5_bm | PIN4_bm;
        p_PORT->DIR &= ~PIN6_bm;

        // Configure master SPI mode and enable the interface.
        p_SPI->CTRL = SPI_MASTER_bm | SPI_ENABLE_bm;
    }
    else
    {
        // Configure the MISO pin as output and the MOSI, SCK, and SS pins as inputs.
        p_PORT->DIR &= ~(PIN7_bm | PIN5_bm | PIN4_bm);
        p_PORT->DIR |= PIN6_bm;

        // Configure slave SPI mode and enable the interface.
        p_SPI->CTRL = SPI_ENABLE_bm;
        p_SPI->DATA = 0;
    }

    // Enable high-priority interrupts unless we are in slave mode.
    spi_mode == SPI_MASTER ? p_SPI->INTCTRL = SPI_INTLVL0_bm | SPI_INTLVL1_bm
    : p_SPI->INTCTRL = 0x00;

          // Indicate through the serial port that the SPI interface initialized
    // successfully.
    #ifdef  SPI_DEBUG
        *p_serial << "SPI interface initialized on " << p_name << "." << endl;
    #endif
}


//////////////////////////////////////////////////////////////////////////
/** @brief Set SPI mode.
 *  @details This method sets the transmission frequency and the mode of
 *  transmission, as well as enable the SPI interface. It should be called
 *  right before calling the transmission () and request_bytes functions.
 *  @param prescale Desired prescaler for SPI transmission.
 *  @param mode Mode of SPI transmission.
 *  @param bit_order Order in which data is received.
 */
void SPI :: set_mode (SPI_PRESCALER_ENUM_t prescale, SPI_TRANSFER_MODES_t mode, SPI_DATA_ORDER_t bit_order)
{
    // Set the desired clock frequency. The SPI transmission frequency will be
    // (F_CPU / prescale).
    switch (prescale)
    {
        case SPI_DIV2:       p_SPI->CTRL |= SPI_CLK2X_bm;
                             p_SPI->CTRL &= ~(SPI_PRESCALER0_bm | SPI_PRESCALER1_bm);              break;
        case SPI_DIV4:       p_SPI->CTRL &= ~(SPI_CLK2X_bm | SPI_PRESCALER0_bm | SPI_PRESCALER1_bm);   break;
        case SPI_DIV8:       p_SPI->CTRL |= SPI_CLK2X_bm | SPI_PRESCALER0_bm;
                             p_SPI->CTRL &= ~SPI_PRESCALER1_bm;                                      break;
        case SPI_DIV16:      p_SPI->CTRL |= SPI_PRESCALER0_bm;
                             p_SPI->CTRL &= ~(SPI_CLK2X_bm | SPI_PRESCALER1_bm);                     break;
        case SPI_DIV32:      p_SPI->CTRL |= SPI_CLK2X_bm | SPI_PRESCALER1_bm;
                             p_SPI->CTRL &= ~SPI_PRESCALER0_bm;                                      break;
        case SPI_DIV64:      p_SPI->CTRL |= SPI_CLK2X_bm | SPI_PRESCALER0_bm | SPI_PRESCALER1_bm;    break;
        case SPI_DIV128:  p_SPI->CTRL |= SPI_PRESCALER0_bm | SPI_PRESCALER1_bm;
                             p_SPI->CTRL &= ~SPI_CLK2X_bm;                                           break;
        default:          p_SPI->CTRL &= ~(SPI_CLK2X_bm | SPI_PRESCALER0_bm | SPI_PRESCALER1_bm);   break;
    };

    // Set the bit order.
    switch (bit_order)
    {
        case SPI_MSBFIRST:    p_SPI->CTRL &= ~SPI_DORD_bm;   break;
        case SPI_LSBFIRST:    p_SPI->CTRL |= SPI_DORD_bm;        break;
        default:          p_SPI->CTRL &= ~SPI_DORD_bm;   break;
    };

    // Set the SPI mode.
    switch (mode)
    {
        case SPI_MODE0: p_SPI->CTRL &= ~(SPI_MODE0_bm | SPI_MODE1_bm);    break;
        case SPI_MODE1: p_SPI->CTRL |= SPI_MODE0_bm;
                          p_SPI->CTRL &= ~SPI_MODE1_bm;                   break;
        case SPI_MODE2: p_SPI->CTRL &= ~SPI_MODE0_bm;
                          p_SPI->CTRL |= SPI_MODE1_bm;                    break;
        case SPI_MODE3: p_SPI->CTRL |= SPI_MODE0_bm | SPI_MODE1_bm;       break;
        default:     p_SPI->CTRL &= ~(SPI_MODE0_bm | SPI_MODE1_bm);  break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Transfer bytes in ISR.
 *  @details This method begins the transmission of all bytes in the
 *  SPI buffer. The data_ready () method will return true if all bytes
 *  have been transferred/received. This method should be used when
 *  SPI interrupts are desired for handling the data transfer.
 */
void SPI :: transfer_ISR (PORT_t * _p_cs_port, uint8_t _cs_bitmask)
{
    // Save the port and bitmask of the chip select pin.
    p_control->p_cs_port = _p_cs_port;
```

```
    p_control->cs_bitmask = _cs_bitmask;

    // Pull the first byte to be transferred and check if we are reading
    // or writing data. Set the interface state accordingly and initiate
    // SPI communication by lowering the chip select pin and writing the
    // pulled byte to the SPI data register.
    p_control->temp = p_control->transfer_buffer.pull_data ();
    p_control->temp.operation == SPI_READ ? p_control->state = SPI_TRANSCEIVING
    : p_control->state = SPI_WRITING;
    p_control->p_cs_port->OUT &= ~p_control->cs_bitmask;
    p_SPI->DATA = p_control->temp.data;
}


//////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTC_FLAG
    /** @brief SPI port C interrupt.
     *  @details This interrupt service routine grabs the received data
     *  in the data register and places it in the SPI receive queue. It
     *  then places a single byte into the data register to start another
     *  SPI transmission if another address is in the queue.
     */
    ISR (SPIC_INT_vect)
    {
        // Determine the state of the interface.
        switch (SPI :: SPI_C_control.state)
        {
            // Transceiving data. Push the received byte to the receive buffer and
            // check to see if another byte is in the transfer buffer. If another
            // byte is available to transfer, check to see if the next operation is
            // a read or write.
            case SPI_TRANSCEIVING:
                SPI :: SPI_C_control.receive_buffer.push_data (SPIC.DATA);
                if (!SPI :: SPI_C_control.transfer_buffer.empty ())
                {
                    // Pull data from the transfer buffer to determine if the next
                    // operation is a read or write. Set the next state of the interface
                    // and send the data.
                    SPI :: SPI_C_control.temp = SPI :: SPI_C_control.transfer_buffer.pull_data ();
                    if (SPI :: SPI_C_control.temp.operation == SPI_WRITE) SPI :: SPI_C_control.state = SPI_WRITING;
                    SPIC.DATA = SPI :: SPI_C_control.temp.data;
                }
                else
                {
                    // If we reach this code then there are no more bytes to transceive,
                    // so we are done. Set the state and lower the chip select pin.
                    SPI :: SPI_C_control.state = SPI_DONE;
                    SPI :: SPI_C_control.p_cs_port->OUT |= SPI :: SPI_C_control.cs_bitmask;
                }
                break;
            // We are not expecting valid data to be in the data register, only check
            // if a byte is available to transfer.
            case SPI_WRITING:
                if (!SPI :: SPI_C_control.transfer_buffer.empty ())
                {
                    // Pull data from the transfer buffer to determine if the next
                    // operation is a read or write. Set the next state of the interface
                    // and send the data.
                    SPI :: SPI_C_control.temp = SPI :: SPI_C_control.transfer_buffer.pull_data ();
                    if (SPI :: SPI_C_control.temp.operation == SPI_READ) SPI :: SPI_C_control.state = SPI_TRANSCEIVING;
                    SPIC.DATA = SPI :: SPI_C_control.temp.data;
                }
                else
                {
                    // If we reach this code then there are no more bytes to transceive,
                    // so we are done. Set the state and lower the chip select pin.
                    SPI :: SPI_C_control.state = SPI_DONE;
                    SPI :: SPI_C_control.p_cs_port->OUT |= SPI :: SPI_C_control.cs_bitmask;
                }
                break;
            // We are done transceiving. This state exists for determining if the transmission
            // is complete or if one more ISR is triggered for some reason.
            case SPI_DONE:
                break;
            default:
                break;
        };
    }
#endif


//////////////////////////////////////////////////////////////////////
#ifdef  SPI_PORTD_FLAG
    /** @brief SPI port D interrupt.
     *  @details This interrupt service routine grabs the received data
     *  in the data register and places it in the SPI receive queue. It
     *  then places a single byte into the data register to start another
     *  SPI transmission if another address is in the SPI queue.
     */
    ISR (SPID_INT_vect)
    {
        // Determine the state of the interface.
```

```
        switch (SPI :: SPI_D_control.state)
        {
            // Transceiving data. Push the received byte to the receive buffer and
            // check to see if another byte is in the transfer buffer. If another
            // byte is available to transfer, check to see if the next operation is
            // a read or write.
            case SPI_TRANSCEIVING:
                SPI :: SPI_D_control.receive_buffer.push_data (SPID.DATA);
                if (!SPI :: SPI_D_control.transfer_buffer.empty ())
                {
                    // Pull data from the transfer buffer to determine if the next
                    // operation is a read or write. Set the next state of the interface
                    // and send the data.
                    SPI :: SPI_D_control.temp = SPI :: SPI_D_control.transfer_buffer.pull_data ();
                    if (SPI :: SPI_D_control.temp.operation == SPI_WRITE) SPI :: SPI_D_control.state = SPI_WRITING;
                    SPID.DATA = SPI :: SPI_D_control.temp.data;
                }
                else
                {
                    // If we reach this code then there are no more bytes to transceive,
                    // so we are done. Set the state and lower the chip select pin.
                    SPI :: SPI_D_control.state = SPI_DONE;
                    SPI :: SPI_D_control.p_cs_port->OUT |= SPI :: SPI_D_control.cs_bitmask;
                }
                break;
            // We are not expecting valid data to be in the data register, only check
            // if a byte is available to transfer.
            case SPI_WRITING:
                if (!SPI :: SPI_D_control.transfer_buffer.empty ())
                {
                    // Pull data from the transfer buffer to determine if the next
                    // operation is a read or write. Set the next state of the interface
                    // and send the data.
                    SPI :: SPI_D_control.temp = SPI :: SPI_D_control.transfer_buffer.pull_data ();
                    if (SPI :: SPI_D_control.temp.operation == SPI_READ) SPI :: SPI_D_control.state = SPI_TRANSCEIVING;
                    SPID.DATA = SPI :: SPI_D_control.temp.data;
                }
                else
                {
                    // If we reach this code then there are no more bytes to transceive,
                    // so we are done. Set the state and lower the chip select pin.
                    SPI :: SPI_D_control.state = SPI_DONE;
                    SPI :: SPI_D_control.p_cs_port->OUT |= SPI :: SPI_D_control.cs_bitmask;
                }
                break;
            // We are done transceiving. This state exists in case one more ISR is triggered
            // for some reason.
            case SPI_DONE:
                break;
            default:
                break;
        };
    }
#endif
```

```
/*********************************************************************/
/** @file microSD.h
 *  @brief MicroSD class header file.
 *
 *  @details This class provides an abstraction layer for interfacing
 *  with the ff.h microSD FATFS library for writing data to a microSD
 *  card. Reading from files on a microSD card is beyond the scope of
 *  the project, but could be easily implemented by adding methods to
 *  this class using the FATFS functions.
 *
 *  Link to FATFS webpage: http://elm-chan.org/fsw/ff/00index_e.html
 *  Link to custom sdcard interface for xmega devices:
 *       https://www.dolman-wim.nl/xmega/libraries/online/sdcard/html/index.html
 *
 *   Contributors:
 *      - Zach Wilson 2/21/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 2/21/2019 - initial release.
 *
 *  Created: 2/21/2019 4:23:39 PM
 */
/*********************************************************************/


#ifndef MICROSD_H_
#define MICROSD_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "ff.h"
#include "ffconf.h"
#include "diskio.h"
#include "stream_base.h"

//////////////////////////////////////////////////////////////////////
/** File control struct used for interfacing with an open file. It
 *  consists of the FIL data struct from the FATFS filesystem and booleans
 *  for determining the state of the FIL struct.
 */
typedef struct
{
    /** File struct used for interfacing with an open file. There should
     *  be one FIL object per open file.
     */
    FIL file;

    /** Boolean indicating if there is an open file in use.
     */
    bool file_open;
} FILE_CONTROL_t;


//////////////////////////////////////////////////////////////////////
/** @brief MicroSD card class.
 *  @details This class provides an abstraction layer for interfacing with
 *  the FATFS library for writing data to a microSD card.
 */
class microSD : public stream_base
{
    private:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** File struct for interfacing with an open file.
         */
        FILE_CONTROL_t data_file;

        /** Storage for the most recent disk error code.
         */
        DSTATUS disk_error;

        /** Storage for the most recent file error code.
         */
        FRESULT file_error;

        /** @brief Write data to file.
         *  @details This method writes a string of characters to the file
         *  using the f_write () function. For f_write () to work, the number
         *  of characters in the string must be input.
         *  @param p_string Pointer to the string of data to write.
         */
        void transmit_data (const char *);

    public:
        /** @brief Default constructor.
         *  @details This constructor instantiates an instance of the
         *  microSD class.
         *  @param serial Pointer to a serial port object for debugging.
         */
```

```
        microSD (stream_base *);

        /** @brief Initialize card.
         *  @details This method initializes the microSD card object by calling
         *  the disk_initialize () method and storing the result of the function
         *  call. All the work to configure the SPI interface on port D and timer
         *  TCC1 is done under the hood.
         *  @return Boolean indicating if the sd card was initialized
         *  successfully.
         */
        bool initialize (void);

        /** @brief Mount filesystem.
         *  @details This method attempts to mount the FATFS filesystem existing
         *  in static memory by calling the f_mount function. The result of this
         *  function is stored and a message is sent to the serial port based on
         *  the result.
         *  @return Boolean indicating if the filesystem was mounted successfully.
         */
        bool mount (void);

        /** @brief Detect microSD card.
         *  @details This method polls a pin to determine if a microSD card is
         *  inserted.
         *  @return Boolean indicating if an SD card is detected.
         */
        inline bool card_detect (void);

        /** @brief Create file.
         *  @details This method creates and opens a new file that can be written.
         *  The close () method should be called after writing a batch of data to
         *  this file.
         *  @param p_name Pointer to a string of the filename.
         *  @return Boolean indicating if the file was created successfully.
         */
        bool create (const char *);

        /** @brief Open file.
         *  @details This method opens an existing file on the microSD card so that
         *  data may be written to it.
         *  @param p_name Pointer to a string of the filename.
         *  @return Boolean indicating if the file was opened successfully.
         */
        bool open (const char *);

        /** @brief Get character.
         *  This method returns one character from the open file.
         *  @return Character retrieved from the open file.
         */
        char get_char (void);

        /** @brief Close file.
         *  @details This method closes and saves the file that is currently open.
         *  @return Boolean indicating if the file was closed and saved
         *  successfully.
         */
        bool close (void);
};


//////////////////////////////////////////////////////////////////////////
/** @brief Detect microSD card.
 *  @details This method polls a pin to determine if a microSD card is
 *  inserted.
 *  @return Boolean indicating if an SD card is detected.
 */
inline bool microSD :: card_detect (void)
{
    if (PORTD.IN & PIN3_bm) return true;
    else return false;
}


#endif /* MICROSD_H_ */
```

```
/*********************************************************************/
/** @file microSD.h
 *  @brief Main class file.
 *
 *  @details This class provides an abstraction layer for interfacing
 *  with the ff.h microSD FAT library for writing data to a microSD card.
 *
 *   Contributors:
 *      - Zach Wilson 5/18/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 5/18/2019 - initial release.
 *
 *  Created: 5/18/2019 10:29:50 PM
 */
/*********************************************************************/


/* Included files. */
#include "microSD.h"

//////////////////////////////////////////////////////////////////////
/** Static file system struct for interfacing with an SD card.
 */
static FATFS fsys;


//////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the microSD
 *  class.
 *  @param serial Pointer to a serial port object for debugging.
 */
microSD :: microSD (stream_base *serial) : stream_base ()
{
    p_serial = serial;
    data_file.file_open = false;
}


//////////////////////////////////////////////////////////////////////
/** @brief Initialize card.
 *  @details This method initializes the microSD card object by calling
 *  the disk_initialize () method and storing the result of the function
 *  call. All the work to configure the SPI interface on port D and timer
 *  TCC1 is done under the hood.
 *  @return Boolean indicating if the sd card was initialized
 *  successfully.
 */
bool microSD :: initialize (void)
{
    disk_error = disk_initialize (0);
    switch (disk_error)
    {
        case RES_OK:
            #ifdef  MICROSD_DEBUG
                *p_serial << "MicroSD card initialized successfully." << endl;
            #endif
            return true;
            break;
        default:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Error. Could not initialize MicroSD card." << endl;
            #endif
            return false;
            break;
    };
}


//////////////////////////////////////////////////////////////////////
/** @brief Mount filesystem
 *  @details This method attempts to mount the FATFS filesystem existing
 *  in static memory by calling the f_mount function. The result of this
 *  function is stored and a message is sent to the serial port based on
 *  the result.
 *  @return Boolean indicating if the filesystem was mounted successfully.
 */
bool microSD :: mount (void)
{
    file_error = f_mount (&fsys, "", 0);
    switch (file_error)
    {
        case FR_OK:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Filesystem mounted successfully." << endl;
            #endif
            return true;
            break;
        default:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Error. Could not mount filesystem." << endl;
```

```
            #endif
            return false;
            break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Create file.
 *  @details This method creates and opens a new file that can be written.
 *  The close () method should be called after writing a batch of data to
 *  this file.
 *  @param p_name Pointer to a string of the filename.
 *  @return Boolean indicating if the file was created successfully.
 */
bool microSD :: create (const char *p_name)
{
    file_error = f_open (&data_file.file, p_name, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
    switch (file_error)
    {
        case FR_OK:
            #ifdef  MICROSD_DEBUG
                *p_serial << "File created successfully." << endl;
            #endif
            return true;
            break;
        default:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Error. Could not create new file." << endl;
            #endif
            return false;
            break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Open file.
 *  @details This method opens an existing file on the microSD card so
 *  that data may be written to it.
 *  @param p_name Pointer to a string of the filename.
 *  @return Boolean indicating if the file was opened successfully.
 */
bool microSD :: open (const char *p_name)
{
    file_error = f_open (&data_file.file, p_name, FA_OPEN_APPEND | FA_WRITE | FA_READ);
    switch (file_error)
    {
        case FR_OK:
            #ifdef  MICROSD_DEBUG
                *p_serial << "File opened successfully." << endl;
            #endif
            data_file.file_open = true;
            return true;
            break;
        default:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Error. Could not open file." << endl;
            #endif
            return false;
            break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Write data to file.
 *  @details This method writes a string of characters to the file using
 *  the f_write () function. For f_write () to work, the number of
 *  characters in the string must be an input.
 *  @param p_string Pointer to the string of data to write.
 */
void microSD :: transmit_data (const char *p_string)
{
    UINT count = 0;
    const char *p_temp = p_string;
    while (*p_temp++ != NULL) count++;
    f_write (&data_file.file, p_string, count, &count);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Get character.
 *  This method returns one character from the open file.
 *  @return Character retrieved from the open file.
 */
char microSD :: get_char (void)
{
    UINT count = 1;
    char *p_char = 0x00;
    f_read (&data_file.file, p_char, count, &count);
    return *p_char;
```

```
}


///////////////////////////////////////////////////////////////////////////
/** @brief Close file.
 *  @details This method closes and saves the file that is currently open.
 *  @return Boolean indicating if the file was closed and saved
 *  successfully.
 */
bool microSD :: close (void)
{
    file_error = f_close (&data_file.file);
    switch (file_error)
    {
        case FR_OK:
            #ifdef  MICROSD_DEBUG
                *p_serial << "File closed successfully." << endl;
            #endif
            data_file.file_open = false;
            return true;
            break;
        default:
            #ifdef  MICROSD_DEBUG
                *p_serial << "Error. Could not close file." << endl;
            #endif
            return false;
            break;
    };
}
```

```c
/*********************************************************************/
/** @file mmc_avr.c
 *  @brief Customized version of mmc_avr.c
 *
 *  @details This file was modified for the ATXMEGA128A4U
 *  microcontroller for saving data to an SD card.
 *
 *   Contributors:
 *      - Zach Wilson 5/28/19 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 5/28/19 - Modified to use the SPI interface on PORTD and timer
 *           TCC1 on the ATXMEGA128A4U microcontroller.
 */
/*********************************************************************/
/*!
 *  \file      mmc_avr.c
 *  \brief     This is customized version of mmc_avr.c
 *             The original version is from sample \e avr release R0.11a
 *  \date      30 march 2016
 *
 *  \details   It is modified by Wim Dolman (<a href="mailto:w.e.dolman@hva.nl">w.e.dolman@hva.nl</a>)
 *             for the HvA-Xmegaboard version 2
 *             The most important global changes are:
 *             - It is suitable for the ATxmega256a3u.
 *             - It uses the Xmega style (DIRSET, OUTSET, ...).
 *             - It uses SPID with a remap of (SCK and MOSI) in stead of bit banging.
 *             .
 *
 *             The most important detailed changes are:
 *             1.  A static function sdspi_init() is added that initialize the SPI
 *             2.  A static function sdspi_write_multi_byte() is added to write multiple bytes to the SPI
 *             3.  A static function sdspi_read_multi_byte() is added to read multiple bytes from the SPI
 *             4.  The static function power_on() is changed. It initislizes the SPI of port C in the Xmega-style.
 *                 The power dectection is removed.
 *             5.  The static function power_off() is changed. It disables the SPI of port C in the Xmega-style.
 *                 The power detcetion is removed.
 *             6.  The static function xchg_spi() is now an alias for sdspi_transfer_byte()
 *             7.  The static function xmit_spi_multi() is now an alias for sdspi_write_multi_byte()
 *             8.  The static function rcvr_spi_multi() is now an alias for sdspi_read_multi_byte()
 *
 */

/*-----------------------------------------------------------------------*/
/* MMCv3/SDv1/SDv2 (in SPI mode) control module                          */
/*-----------------------------------------------------------------------*/
/*
/  Copyright (C) 2014, ChaN, all right reserved.
/
/ * This software is a free software and there is NO WARRANTY.
/ * No restriction on use. You can use, modify and redistribute it for
/   personal, non-profit or commercial products UNDER YOUR RESPONSIBILITY.
/ * Redistributions of source code must retain the above copyright notice.
/
/  Changed 30 march 2016
/  Wim Dolman (<a href="email:w.e.dolman@hva.nl">w.e.dolman@hva.nl</a>)
/  Made it suitable for the HvA-Xmegaboard. The most important chenges are:
/    -  Xmega256au3
/    -  Xmega-style (DIRSET, OUTSET, ...)
/    -  Using SPID with remap of (SCK and MOSI) in stead of SPDR
/    -  Socket power control removed. HvA-Xmegaboard has no power control.
/
/  This implentation uses a SPI and a timeout. For the timeout a
/  100Hz decrement timer is necessary. For example timer D0:
/    void init_timer(void)
/    {
/      TCD0.CTRLB     = TC_WGMODE_NORMAL_gc;
/      TCD0.CTRLA     = TC_CLKSEL_DIV8_gc;
/      TCD0.INTCTRLA  = TC_OVFINTLVL_LO_gc;
/      TCD0.PER       = 39999;  // f=fcpu/(N*(PER+1))=32M/8*40000=100 Hz
/    }
/
/    ISR(TCD0_OVF_vect)
/    {
/       disk_timerproc();  // Drive timer procedure of low level disk I/O
/    }
/------------------------------------------------------------------------*/

#include <avr/io.h>
#include "diskio.h"

/*------------------------------------------------------------------------
   Module Private Functions
-------------------------------------------------------------------------*/
//@cond
// MMC/SD command (SPI mode)
#define CMD0   (0)        /* GO_IDLE_STATE */
#define CMD1   (1)        /* SEND_OP_COND */
#define ACMD41 (0x80+41)  /* SEND_OP_COND (SDC) */
#define CMD8   (8)        /* SEND_IF_COND */
#define CMD9   (9)        /* SEND_CSD */
#define CMD10  (10)       /* SEND_CID */
```

```c
#define CMD12  (12)       /* STOP_TRANSMISSION */
#define CMD13  (13)       /* SEND_STATUS */
#define ACMD13 (0x80+13)  /* SD_STATUS (SDC) */
#define CMD16  (16)       /* SET_BLOCKLEN */
#define CMD17  (17)       /* READ_SINGLE_BLOCK */
#define CMD18  (18)       /* READ_MULTIPLE_BLOCK */
#define CMD23  (23)       /* SET_BLOCK_COUNT */
#define ACMD23 (0x80+23)  /* SET_WR_BLK_ERASE_COUNT (SDC) */
#define CMD24  (24)       /* WRITE_BLOCK */
#define CMD25  (25)       /* WRITE_MULTIPLE_BLOCK */
#define CMD32  (32)       /* ERASE_ER_BLK_START */
#define CMD33  (33)       /* ERASE_ER_BLK_END */
#define CMD38  (38)       /* ERASE */
#define CMD55  (55)       /* APP_CMD */
#define CMD58  (58)       /* READ_OCR */

static DSTATUS Stat = STA_NOINIT;     // Disk status
static BYTE CardType;                 // b0:MMC, b1:SDv1, b2:SDv2,
                                      //         b3:Block addressing
static volatile BYTE Timer1, Timer2;  // 100Hz decrement timer
//@endcond

/*-----------------------------------------------------------------------*/
/* SPI Chip Select for SD-card HvA-Xmegaboard version 2                  */
/*-----------------------------------------------------------------------*/
#define sdspi_CSlow()   PORTD.OUTCLR = PIN4_bm           //!< added:  define for Chip Select low
#define sdspi_CShigh()  PORTD.OUTSET = PIN4_bm           //!< added:  define for Chip Select high
#define FCLK_SLOW()     SPID.CTRL = (SPI_CLK2X_bm|SPI_ENABLE_bm|SPI_MASTER_bm|SPI_MODE_0_gc|SPI_PRESCALER_DIV64_gc)  //!< added:
set slow clock
#define FCLK_FAST()     SPID.CTRL = (SPI_CLK2X_bm|SPI_ENABLE_bm|SPI_MASTER_bm|SPI_MODE_0_gc|SPI_PRESCALER_DIV4_gc)   //!< added:
set fast clock
#define CS_LOW()        PORTD.OUTCLR = PIN4_bm           //!< added:  define for Chip Select low
#define CS_HIGH()       PORTD.OUTSET = PIN4_bm           //!< added:  define for Chip Select high
#define MMC_CD          ( bit_is_clear(PORTD, 3) )       //!< Card detected. yes:non zero, no:zero
#define MMC_WP          0                                //!< Write protected not implemented with HvA-Xmegaboard

/*-----------------------------------------------------------------------*/
/* Transmit/Receive data from/to MMC via SPI  (Platform dependent)       */
/*-----------------------------------------------------------------------*/

/* Exchange a byte */
static unsigned char sdspi_transfer_byte(unsigned char databyte)
{
  /* Added: Changed SPIC and PORTC to SPID and PORTD. */
  SPID.DATA = databyte;
  while ( !(SPID.STATUS & (SPI_IF_bm))) ;
  return SPID.DATA;
}

/* Send a data block fast */
static void sdspi_write_multi_byte(const unsigned char *p, unsigned int cnt)
{
  /* Added: Changed SPIC and PORTC to SPID and PORTD. */
  do {
    SPID.DATA = *p++;
    while ( !(SPID.STATUS & (SPI_IF_bm))) ;
    SPID.DATA = *p++;
    while ( !(SPID.STATUS & (SPI_IF_bm))) ;
  } while (cnt -= 2);
}

/* Receive a data block fast */
static void sdspi_read_multi_byte(unsigned char *p, unsigned int cnt)
{
  /* Added: Changed SPIC and PORTC to SPID and PORTD. */
  do {
    SPID.DATA = 0xFF;
    while ( !(SPID.STATUS & (SPI_IF_bm))) ;
    *p++ = SPID.DATA;
    SPID.DATA = 0xFF;
    while ( !(SPID.STATUS & (SPI_IF_bm))) ;
    *p++ = SPID.DATA;
  } while (cnt -= 2);
}

/*-----------------------------------------------------------------------*/
/* Replacements for spi-functions                                       */
/*-----------------------------------------------------------------------*/
#define xchg_spi        sdspi_transfer_byte                //!< alias for sdspi_transfer_byte()
#define xmit_spi_multi  sdspi_write_multi_byte             //!< alias for sdspi_write_multi_byte()
#define rcvr_spi_multi  sdspi_read_multi_byte              //!< alias for sdspi_read_multi_byte()

/*-----------------------------------------------------------------------*/
/* Power Control  (Platform dependent)                                   */
/*-----------------------------------------------------------------------*/
/* When the target system does not support socket power control, there   */
/* is nothing to do in these functions and chk_power always returns 1.   */

static void power_on (void)
{
  // {  /* Remove this block if no socket power control */
    //PORTE &= ~_BV(7);  /* Socket power on (PE7=low) */
```

```
    //DDRE |= _BV(7);
    //for (Timer1 = 2; Timer1; );  /* Wait for 20ms */
  // }
  /* Added: Changed SPIC and PORTC to SPID and PORTD. */
  PORTD.DIRCLR   = PIN6_bm;             // MISO        input
  PORTD.DIRSET   = PIN5_bm;             // MOSI        output
  PORTD.DIRSET   = PIN7_bm;             // SCK         output
  PORTD.DIRSET   = PIN4_bm;             // chip select output
  PORTD.PIN6CTRL = PORT_OPC_PULLUP_gc;  // MISO        pullup

  PORTD.DIRCLR   = PIN3_bm;             // card detect input

// was getest  SPID.CTRL = (SPI_CLK2X_bm|SPI_ENABLE_bm|SPI_MASTER_bm|SPI_MODE_0_gc|SPI_PRESCALER_DIV64_gc);
// 8 MHz misschien kan 16 ook
  SPID.CTRL = SPI_ENABLE_bm|SPI_MASTER_bm|SPI_MODE_0_gc|SPI_PRESCALER_DIV4_gc;
}

static void power_off (void)
{
  /* Added: Changed SPIC and PORTC to SPID and PORTD. */
  SPID.CTRL = 0;       // Disable SPI function

  PORTD.DIRCLR   = PIN6_bm;             // MISO        input
  PORTD.DIRSET   = PIN5_bm;             // MOSI        output
  PORTD.DIRSET   = PIN7_bm;             // SCK         output
  PORTD.DIRSET   = PIN4_bm;             // chip select output
  PORTD.PIN6CTRL = PORT_OPC_PULLUP_gc;  // MISO        pullup

  PORTD.DIRCLR   = PIN3_bm;             // card detect input

  {  /* Remove this block if no socket power control */
    // PORTE |= _BV(7);    /* Socket power off (PE7=high) */
    // for (Timer1 = 20; Timer1; );  /* Wait for 20ms */
  }
}

//@cond
/*
 * FROM HERE NOTHING CHANGED!
 * From it is completely equal to avr-mmc.c of example avr/mmc-avr.c
 */


/*-----------------------------------------------------------------------*/
/* Wait for card ready                                                   */
/*-----------------------------------------------------------------------*/

static
int wait_ready (  /* 1:Ready, 0:Timeout */
  UINT wt      /* Timeout [ms] */
)
{
  BYTE d;

  Timer2 = wt / 10;
  do
    d = xchg_spi(0xFF);
  while (d != 0xFF && Timer2);

  return (d == 0xFF) ? 1 : 0;
}



/*-----------------------------------------------------------------------*/
/* Deselect the card and release SPI bus                                 */
/*-----------------------------------------------------------------------*/

static
void deselect (void)
{
  CS_HIGH();    /* Set CS# high */
  xchg_spi(0xFF);  /* Dummy clock (force DO hi-z for multiple slave SPI) */
}



/*-----------------------------------------------------------------------*/
/* Select the card and wait for ready                                    */
/*-----------------------------------------------------------------------*/

static
int select (void)  /* 1:Successful, 0:Timeout */
{
  CS_LOW();    /* Set CS# low */
  xchg_spi(0xFF);  /* Dummy clock (force DO enabled) */
  if (wait_ready(500)) return 1;  /* Wait for card ready */

  deselect();
  return 0;  /* Timeout */
}
```

```
/*-----------------------------------------------------------------------*/
/* Receive a data packet from MMC                                        */
/*-----------------------------------------------------------------------*/

static
int rcvr_datablock (
  BYTE *buff,      /* Data buffer to store received data */
  UINT btr      /* Byte count (must be multiple of 4) */
)
{
  BYTE token;


  Timer1 = 20;
  do {              /* Wait for data packet in timeout of 200ms */
    token = xchg_spi(0xFF);
  } while ((token == 0xFF) && Timer1);
  if (token != 0xFE) return 0;  /* If not valid data token, retutn with error */

  rcvr_spi_multi(buff, btr);    /* Receive the data block into buffer */
  xchg_spi(0xFF);          /* Discard CRC */
  xchg_spi(0xFF);

  return 1;            /* Return with success */
}



/*-----------------------------------------------------------------------*/
/* Send a data packet to MMC                                             */
/*-----------------------------------------------------------------------*/

#if  _USE_WRITE
static
int xmit_datablock (
  const BYTE *buff,  /* 512 byte data block to be transmitted */
  BYTE token      /* Data/Stop token */
)
{
  BYTE resp;


  if (!wait_ready(500)) return 0;

  xchg_spi(token);          /* Xmit data token */
  if (token != 0xFD) {  /* Is data token */
    xmit_spi_multi(buff, 512);    /* Xmit the data block to the MMC */
    xchg_spi(0xFF);          /* CRC (Dummy) */
    xchg_spi(0xFF);
    resp = xchg_spi(0xFF);      /* Reveive data response */
    if ((resp & 0x1F) != 0x05)    /* If not accepted, return with error */
      return 0;
  }

  return 1;
}
#endif



/*-----------------------------------------------------------------------*/
/* Send a command packet to MMC                                          */
/*-----------------------------------------------------------------------*/

static
BYTE send_cmd (    /* Returns R1 resp (bit7==1:Send failed) */
  BYTE cmd,    /* Command index */
  DWORD arg    /* Argument */
)
{
  BYTE n, res;


  if (cmd & 0x80) {  /* ACMD<n> is the command sequense of CMD55-CMD<n> */
    cmd &= 0x7F;
    res = send_cmd(CMD55, 0);
    if (res > 1) return res;
  }

  /* Select the card and wait for ready except to stop multiple block read */
  if (cmd != CMD12) {
    deselect();
    if (!select()) return 0xFF;
  }

  /* Send command packet */
  xchg_spi(0x40 | cmd);        /* Start + Command index */
  xchg_spi((BYTE)(arg >> 24));    /* Argument[31..24] */
  xchg_spi((BYTE)(arg >> 16));    /* Argument[23..16] */
  xchg_spi((BYTE)(arg >> 8));      /* Argument[15..8] */
```

```
  xchg_spi((BYTE)arg);        /* Argument[7..0] */
  n = 0x01;               /* Dummy CRC + Stop */
  if (cmd == CMD0) n = 0x95;      /* Valid CRC for CMD0(0) + Stop */
  if (cmd == CMD8) n = 0x87;      /* Valid CRC for CMD8(0x1AA) Stop */
  xchg_spi(n);

  /* Receive command response */
  if (cmd == CMD12) xchg_spi(0xFF);    /* Skip a stuff byte when stop reading */
  n = 10;                 /* Wait for a valid response in timeout of 10 attempts */
  do
    res = xchg_spi(0xFF);
  while ((res & 0x80) && --n);

  return res;      /* Return with the response value */
}



/*-----------------------------------------------------------------------

   Public Functions

-----------------------------------------------------------------------*/


/*-----------------------------------------------------------------------*/
/* Initialize Disk Drive                                                 */
/*-----------------------------------------------------------------------*/

DSTATUS disk_initialize (
  BYTE pdrv    /* Physical drive nmuber (0) */
)
{
  BYTE n, cmd, ty, ocr[4];

  /* Added: Initialization for timer interrupts on TCC1. */
  TCC1.CTRLA     = TC_CLKSEL_DIV8_gc;
  TCC1.INTCTRLA  = TC_OVFINTLVL_LO_gc;
  TCC1.PER       = 39999;  // f=fcpu/(N*(PER+1))=32M/8*40000=100 Hz

  if (pdrv) return STA_NOINIT;    /* Supports only single drive */
  power_off();            /* Turn off the socket power to reset the card */
  if (Stat & STA_NODISK) return Stat;  /* No card in the socket */
  power_on();              /* Turn on the socket power */
  FCLK_SLOW();

  for (n = 10; n; n--) xchg_spi(0xFF);  /* 80 dummy clocks */

  ty = 0;
  if (send_cmd(CMD0, 0) == 1) {      /* Enter Idle state */
    Timer1 = 100;            /* Initialization timeout of 1000 msec */
    if (send_cmd(CMD8, 0x1AA) == 1) {  /* SDv2? */
      for (n = 0; n < 4; n++) ocr[n] = xchg_spi(0xFF);    /* Get trailing return value of R7 resp */
      if (ocr[2] == 0x01 && ocr[3] == 0xAA) {        /* The card can work at vdd range of 2.7-3.6V */
        while (Timer1 && send_cmd(ACMD41, 1UL << 30));  /* Wait for leaving idle state (ACMD41 with HCS bit) */
        if (Timer1 && send_cmd(CMD58, 0) == 0) {    /* Check CCS bit in the OCR */
          for (n = 0; n < 4; n++) ocr[n] = xchg_spi(0xFF);
          ty = (ocr[0] & 0x40) ? CT_SD2 | CT_BLOCK : CT_SD2;  /* SDv2 */
        }
      }
    } else {              /* SDv1 or MMCv3 */
      if (send_cmd(ACMD41, 0) <= 1)   {
        ty = CT_SD1; cmd = ACMD41;  /* SDv1 */
      } else {
        ty = CT_MMC; cmd = CMD1;  /* MMCv3 */
      }
      while (Timer1 && send_cmd(cmd, 0));      /* Wait for leaving idle state */
      if (!Timer1 || send_cmd(CMD16, 512) != 0)  /* Set R/W block length to 512 */
        ty = 0;
    }
  }
  CardType = ty;
  deselect();

  if (ty) {      /* Initialization succeded */
    Stat &= ~STA_NOINIT;    /* Clear STA_NOINIT */
    FCLK_FAST();
  } else {      /* Initialization failed */
    power_off();
  }

  return Stat;
}



/*-----------------------------------------------------------------------*/
/* Get Disk Status                                                       */
/*-----------------------------------------------------------------------*/

DSTATUS disk_status (
  BYTE pdrv    /* Physical drive nmuber (0) */
```

```
)
{
  if (pdrv) return STA_NOINIT;  /* Supports only single drive */
  return Stat;
}



/*-----------------------------------------------------------------------*/
/* Read Sector(s)                                                        */
/*-----------------------------------------------------------------------*/

DRESULT disk_read (
  BYTE pdrv,      /* Physical drive nmuber (0) */
  BYTE *buff,      /* Pointer to the data buffer to store read data */
  DWORD sector,    /* Start sector number (LBA) */
  UINT count      /* Sector count (1..128) */
)
{
  BYTE cmd;


  if (pdrv || !count) return RES_PARERR;
  if (Stat & STA_NOINIT) return RES_NOTRDY;

  if (!(CardType & CT_BLOCK)) sector *= 512;  /* Convert to byte address if needed */

  cmd = count > 1 ? CMD18 : CMD17;      /*  READ_MULTIPLE_BLOCK : READ_SINGLE_BLOCK */
  if (send_cmd(cmd, sector) == 0) {
    do {
      if (!rcvr_datablock(buff, 512)) break;
      buff += 512;
    } while (--count);
    if (cmd == CMD18) send_cmd(CMD12, 0);  /* STOP_TRANSMISSION */
  }
  deselect();

  return count ? RES_ERROR : RES_OK;
}



/*-----------------------------------------------------------------------*/
/* Write Sector(s)                                                       */
/*-----------------------------------------------------------------------*/

#if _USE_WRITE
DRESULT disk_write (
  BYTE pdrv,      /* Physical drive nmuber (0) */
  const BYTE *buff,  /* Pointer to the data to be written */
  DWORD sector,    /* Start sector number (LBA) */
  UINT count      /* Sector count (1..128) */
)
{
  if (pdrv || !count) return RES_PARERR;
  if (Stat & STA_NOINIT) return RES_NOTRDY;
  if (Stat & STA_PROTECT) return RES_WRPRT;

  if (!(CardType & CT_BLOCK)) sector *= 512;  /* Convert to byte address if needed */

  if (count == 1) {  /* Single block write */
    if ((send_cmd(CMD24, sector) == 0)  /* WRITE_BLOCK */
      && xmit_datablock(buff, 0xFE))
      count = 0;
  }
  else {        /* Multiple block write */
    if (CardType & CT_SDC) send_cmd(ACMD23, count);
    if (send_cmd(CMD25, sector) == 0) {  /* WRITE_MULTIPLE_BLOCK */
      do {
        if (!xmit_datablock(buff, 0xFC)) break;
        buff += 512;
      } while (--count);
      if (!xmit_datablock(0, 0xFD))  /* STOP_TRAN token */
        count = 1;
    }
  }
  deselect();

  return count ? RES_ERROR : RES_OK;
}
#endif


/*-----------------------------------------------------------------------*/
/* Miscellaneous Functions                                               */
/*-----------------------------------------------------------------------*/

#if _USE_IOCTL
DRESULT disk_ioctl (
  BYTE pdrv,    /* Physical drive nmuber (0) */
  BYTE cmd,    /* Control code */
  void *buff    /* Buffer to send/receive control data */
```

```
)
{
  DRESULT res;
  BYTE n, csd[16], *ptr = buff;
  DWORD csize;


  if (pdrv) return RES_PARERR;

  res = RES_ERROR;

  if (Stat & STA_NOINIT) return RES_NOTRDY;

  switch (cmd) {
  case CTRL_SYNC :    /* Make sure that no pending write process. Do not remove this or written sector might not left updated. */
    if (select()) res = RES_OK;
    break;

  case GET_SECTOR_COUNT :  /* Get number of sectors on the disk (DWORD) */
    if ((send_cmd(CMD9, 0) == 0) && rcvr_datablock(csd, 16)) {
      if ((csd[0] >> 6) == 1) {  /* SDC ver 2.00 */
        csize = csd[9] + ((WORD)csd[8] << 8) + ((DWORD)(csd[7] & 63) << 16) + 1;
        *(DWORD*)buff = csize << 10;
      } else {          /* SDC ver 1.XX or MMC*/
        n = (csd[5] & 15) + ((csd[10] & 128) >> 7) + ((csd[9] & 3) << 1) + 2;
        csize = (csd[8] >> 6) + ((WORD)csd[7] << 2) + ((WORD)(csd[6] & 3) << 10) + 1;
        *(DWORD*)buff = csize << (n - 9);
      }
      res = RES_OK;
    }
    break;

  case GET_BLOCK_SIZE :  /* Get erase block size in unit of sector (DWORD) */
    if (CardType & CT_SD2) {  /* SDv2? */
      if (send_cmd(ACMD13, 0) == 0) {  /* Read SD status */
        xchg_spi(0xFF);
        if (rcvr_datablock(csd, 16)) {        /* Read partial block */
          for (n = 64 - 16; n; n--) xchg_spi(0xFF);  /* Purge trailing data */
          *(DWORD*)buff = 16UL << (csd[10] >> 4);
          res = RES_OK;
        }
      }
    } else {          /* SDv1 or MMCv3 */
      if ((send_cmd(CMD9, 0) == 0) && rcvr_datablock(csd, 16)) {  /* Read CSD */
        if (CardType & CT_SD1) {  /* SDv1 */
          *(DWORD*)buff = (((csd[10] & 63) << 1) + ((WORD)(csd[11] & 128) >> 7) + 1) << ((csd[13] >> 6) - 1);
        } else {          /* MMCv3 */
          *(DWORD*)buff = ((WORD)((csd[10] & 124) >> 2) + 1) * (((csd[11] & 3) << 3) + ((csd[11] & 224) >> 5) + 1);
        }
        res = RES_OK;
      }
    }
    break;

  /* Following commands are never used by FatFs module */

  case MMC_GET_TYPE :    /* Get card type flags (1 byte) */
    *ptr = CardType;
    res = RES_OK;
    break;

  case MMC_GET_CSD :    /* Receive CSD as a data block (16 bytes) */
    if (send_cmd(CMD9, 0) == 0    /* READ_CSD */
      && rcvr_datablock(ptr, 16))
      res = RES_OK;
    break;

  case MMC_GET_CID :    /* Receive CID as a data block (16 bytes) */
    if (send_cmd(CMD10, 0) == 0    /* READ_CID */
      && rcvr_datablock(ptr, 16))
      res = RES_OK;
    break;

  case MMC_GET_OCR :    /* Receive OCR as an R3 resp (4 bytes) */
    if (send_cmd(CMD58, 0) == 0) {  /* READ_OCR */
      for (n = 4; n; n--) *ptr++ = xchg_spi(0xFF);
      res = RES_OK;
    }
    break;

  case MMC_GET_SDSTAT :  /* Receive SD statsu as a data block (64 bytes) */
    if (send_cmd(ACMD13, 0) == 0) {  /* SD_STATUS */
      xchg_spi(0xFF);
      if (rcvr_datablock(ptr, 64))
        res = RES_OK;
    }
    break;

  case CTRL_POWER_OFF :  /* Power off */
    power_off();
    Stat |= STA_NOINIT;
    res = RES_OK;
```

```
    break;

  default:
    res = RES_PARERR;
  }

  deselect();

  return res;
}
#endif


/*-----------------------------------------------------------------------*/
/* Device Timer Interrupt Procedure                                      */
/*-----------------------------------------------------------------------*/
/* This function must be called in period of 10ms                        */

void disk_timerproc (void)
{
  BYTE n, s;


  n = Timer1;        /* 100Hz decrement timer */
  if (n) Timer1 = --n;
  n = Timer2;
  if (n) Timer2 = --n;
  s = Stat;

  if (MMC_WP)        /* Write protected */
    s |= STA_PROTECT;
  else          /* Write enabled */
    s &= ~STA_PROTECT;

  if (MMC_CD)        /* Card inserted */
    s &= ~STA_NODISK;
  else          /* Socket empty */
    s |= (STA_NODISK | STA_NOINIT);

  Stat = s;        /* Update MMC status */
}
//@endcond


/* Added: Timer overflow interrupt for handling the timeout of the FATFS
   functions. */
///////////////////////////////////////////////////////////////////////////
/** Timer overflow interrupt.
 */
ISR (TCC1_OVF_vect)
{
    disk_timerproc ();  // Drive timer procedure of low level disk I/O
}
```

```
/*********************************************************************/
/** @file ADC.h
 *  @brief ADC converter class header file.
 *
 *  @details This class provides an analog-to-digital converter interface
 *  to be used on port A. It provides methods for requesting A/D
 *  conversions on user-specified port pins. Converted data is stored in
 *  a queue to wait to be retrieved by the available retrieve methods.
 *  This class also acts as a container class for objects of the ADC
 *  channel class, which includes methods for interfacing with a specific
 *  channel of the ADC interface.
 *
 *   Contributors:
 *       - Zach Wilson 2/17/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *       - 2/17/2019 - initial release.
 *
 *  Created: 2/17/2019 1:44:02 PM
 */
/*********************************************************************/


#ifndef ADC_H_
#define ADC_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "circular_buffer.h"
#include "serial_port.h"

//////////////////////////////////////////////////////////////////////////
/** Defines the buffer size for the ADC converter class.
 */
#define  ADC_BUFFER_SIZE  40

//////////////////////////////////////////////////////////////////////////
/** Define for selecting the chip internal ground as an input to the
 *  analog comparator.
 */
#define  ADC_INTERNAL_GND  0x07

//////////////////////////////////////////////////////////////////////////
/** Define for selecting the port internal ground as an input to the
*  analog comparator.
*/
#define  ADC_PAD_GND  0x0F

//////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the interface name.
 */
extern void storeString (const char *, char *);


//////////////////////////////////////////////////////////////////////////
/** Channel enum. Defines the macros used for selecting which ADC channel
 *  to interact with.
 */
typedef enum  ADC_CHANNEL_ENUM
{
    /** Channel 0.
     */
    ADC_CHANNEL_0 = ADC_CH0START_bm,

    /** Channel 1.
     */
    ADC_CHANNEL_1 = ADC_CH1START_bm,

    /** Channel 2.
     */
    ADC_CHANNEL_2 = ADC_CH2START_bm,

    /** Channel 0.
     */
    ADC_CHANNEL_3 = ADC_CH3START_bm,
} ADC_CHANNEL_t;


//////////////////////////////////////////////////////////////////////////
/** ADC control struct. Contains the data buffers for each ADC channel
 *  where the conversion results are stored in an ISR.
 */
typedef struct
{
    /** Circular buffer for queuing read data from ADC channel 0.
     */
    circular_buffer <volatile uint16_t, ADC_BUFFER_SIZE> ch0_buffer;

    /** Circular buffer for queuing read data from ADC channel 1.
     */
```

```
    circular_buffer <volatile uint16_t, ADC_BUFFER_SIZE> ch1_buffer;

    /** Circular buffer for queuing read data from ADC channel 2.
     */
    circular_buffer <volatile uint16_t, ADC_BUFFER_SIZE> ch2_buffer;

    /** Circular buffer for queuing read data from ADC channel 3.
     */
    circular_buffer <volatile uint16_t, ADC_BUFFER_SIZE> ch3_buffer;

    /** Frequency at which to take dynamic data.
     */
    volatile uint16_t timer_freq;

    /** Active channel bitmask to determine which channels are taking
     *  data.
     */
    volatile uint8_t active_channels;
} ADC_CONTROL_STRUCT_t;


//////////////////////////////////////////////////////////////////////
/** @brief ADC channel class.
 *  @details This class provides methods for interfacing with a channel
 *  of the ADC interface. Objects of this class are instantiated within
 *  the ADC class for each of the 4 channels of the ADC interface.
 */
class ADC_channel
{
    /** Declare the ADC class a friend so it may alter some private
     *  members of this class.
     */
    friend class ADC;

    private:
        /** Pointer to the port of the ADC interface.
         */
        static PORT_t *p_PORT;

        /** Pointer to an ADC channel interface.
         */
        ADC_CH_t *p_channel;

        /** Pointer to the channel circular buffer.
         */
        circular_buffer <volatile uint16_t, ADC_BUFFER_SIZE> *p_buffer;

        /** Pin to connect to the positive side of the analog comparator.
         */
        uint8_t pos_pin;

        /** Pin to connect to the negative side of the analog comparator.
         */
        uint8_t neg_pin;

    public:
        /** @brief Default constructor.
         *  @details This constructor instantiates an instance of the ADC
         *  channel class.
         */
        ADC_channel (void);

        /** @brief Select pin.
         *  @details This method ties a port pin to the positive end of the ADC
         *  comparator. It also configures the ADC channel and for single-ended
         *  conversion and should be called before performing a conversion on the
         *  channel.
         *  @param pos_pin Bitmask of the pin tied to the positive side of the
         *  ADC comparator.
         */
        void tie_pin (uint8_t);

        /** @brief Set ADC gain.
         *  @details This method sets the gain of the ADC channel being used.
         *  @param gain Number corresponding to the gain setting.
         */
        void set_gain (uint8_t);

        /** @brief Read data.
         *  @details This method reads an A/D conversion from the specified
         *  port pin. It calls the request method and then waits until the
         *  the conversion is complete before returning the data, so there is
         *  some overhead consequences for calling this function.
         *  @return Converted A/D data from the specified pin.
         */
        uint16_t read (void);

        /** @brief Check buffer.
         *  @details This method checks to see if data is available in the buffer.
         *  @param channel Number corresponding to an ADC channel.
         */
        inline bool check_buf (void);
```

```
        /** @brief Request conversion.
         *  @details This method requests one A/D conversion on the specified pin.
         *  The converted data will be stored in the buffer until the retrieve method
         *  is called. This method does not have as much overhead as the read method
         *  because it does not wait for the conversion to be complete before returning.
         */
        inline void request (void);

        /** @brief Retrieve converted data.
         *  @details This method retrieves one 16-bit block of data from the converted
         *  data buffer.
         *  @return Converted A/D data from the buffer.
         */
        inline uint16_t retrieve (void);
};


////////////////////////////////////////////////////////////////////////////
/** @brief Check buffer.
 *  @details This method checks to see if data is available in the buffer.
 *  @return Boolean indicating if data is in the buffer
 *  (true == available).
 */
inline bool ADC_channel :: check_buf (void)
{
    return !p_buffer->empty ();
}


////////////////////////////////////////////////////////////////////////////
/** @brief Request conversion.
 *  @details This method requests one A/D conversion on the specified pin.
 *  The converted data will be stored in the buffer until the retrieve
 *  method is called. This method does not have as much overhead as the
 *  read method because it does not wait for the conversion to be complete
 *  before returning.
 */
inline void ADC_channel :: request (void)
{
    p_channel->CTRL |= ADC_CH_START_bm;
}


////////////////////////////////////////////////////////////////////////////
/** @brief Retrieve converted data.
 *  @details This method retrieves one 16-bit block of data from the
 *  converted data buffer. If the buffer is empty, a NULL value will be
 *  returned.
 *  @return Converted A/D data from the buffer.
 */
inline uint16_t ADC_channel :: retrieve (void)
{
    return p_buffer->pull_data ();
}


////////////////////////////////////////////////////////////////////////////
/** @brief Analog-to-digital converter class.
 *  @details This class provides methods for initializing and controlling
 *  an ADC interface on port A for single-ended analog-to-digital
 *  conversion.
 */
class ADC
{
    private:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** Pointer to an ADC interface.
         */
        ADC_t *p_ADC;

        /** Pointer to the port of the ADC interface.
         */
        PORT_t *p_PORT;

        /** C-style string for interface name.
         */
        char name [11];

    public:
        /** Pointer to the interface name.
         */
        const char *p_name;

        /** ADC control struct for use in an ISR.
         */
        static ADC_CONTROL_STRUCT_t ADC_control;

        /** ADC channel object for channel 0.
```

```
         */
        ADC_channel channel0;

        /** ADC channel object for channel 1.
         */
        ADC_channel channel1;

        /** ADC channel object for channel 2.
         */
        ADC_channel channel2;

        /** ADC channel object for channel 3.
         */
        ADC_channel channel3;

        /** @brief Default constructor.
         *  @details This constructor instantiates an instance of the ADC
         *  converter interface on a specified port.
         *  @param ADC_name Pointer to a string for the interface name.
         *  @param ADC Pointer to the ADC interface.
         *  @param PORT Pointer to the port containing the ADC interface.
         */
        ADC (const char *, ADC_t *, PORT_t *);

        /** @brief Initialize ADC interface.
         *  @details This method initializes the ADC interface. To actually
         *  read data, the desired pin must be set as input before attempting
         *  to read the data.
         *  @param serial Pointer to a serial port object for debugging.
         */
        void initialize (stream_base *);

        /** @brief Enable ADC interface.
         *  @details This method enables the ADC interface. It should be called
         *  before attempting to read data from any ADC channel.
         */
        inline void enable (void);

        /** @brief Disable ADC interface.
         *  @details This method disables the ADC interface. It should be called
         *  when the device is no longer reading data.
         */
        inline void disable (void);

        /** @brief Acquire dynamic data.
         *  @details This method acquires data from a selected channel at a user-
         *  specified frequency using output compare interrupts on a timer. The
         *  data is acquired in an ISR and stored in the ADC data queue until
         *  the retrieve () method is called.
         *  @param channels Bitmask of active channels.
         *  @param frequency Frequency at which to take data (in clock ticks).
         */
        inline void acquire_data (uint8_t, uint16_t);

        /** @brief Stop taking dynamic data.
         *  @details This method disables the timer and resets the count so that
         *  output compare interrupts are no longer triggered.
         */
        inline void stop_data (void);
};


//////////////////////////////////////////////////////////////////////////
/** @brief Enable ADC interface.
 *  @details This method enables the ADC interface. It should be called
 *  before attempting to read data from any ADC channel.
 */
inline void ADC :: enable (void)
{
    p_ADC->CTRLA |= ADC_ENABLE_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Disable ADC interface.
 *  @details This method disables the ADC interface. It should be called
 *  when the device is no longer reading data.
 */
inline void ADC :: disable (void)
{
    p_ADC->CTRLA &= ~ADC_ENABLE_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Acquire dynamic data.
 *  @details This method acquires data from a selected channel at a user-
 *  specified frequency using output compare interrupts on a timer. The
 *  data is acquired in an ISR and stored in the ADC data queue until the
 *  retrieve () method is called.
 *  @param channels Bitmask of active channels.
 *  @param frequency Frequency at which to take data (in clock ticks).
```

```
 */
inline void ADC :: acquire_data (uint8_t channels, uint16_t frequency)
{
    // Update the active channel bitmask and timer frequency. Initialize
    // the timer and output compare interrupts that drive the data
    // acquisition.
    ADC_control.active_channels = channels;
    ADC_control.timer_freq = frequency * 125; // FREQ = frequency * (32,000,000 / (256 * 1000))
    TCD0.CNT = 0x0000;
    TCD0.CCB = ADC_control.timer_freq;
    TCD0.INTCTRLB |= TC0_CCBINTLVL0_bm | TC0_CCBINTLVL1_bm;
    TCD0.CTRLA = TC0_CLKSEL1_bm | TC0_CLKSEL2_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Stop taking dynamic data.
 *  @details This method disables the timer and resets the count so that
 *  output compare interrupts are no longer triggered.
 */
inline void ADC :: stop_data (void)
{
    // Disable the timer and the output compare interrupts.
    TCD0.INTCTRLB &= ~(TC0_CCBINTLVL0_bm | TC0_CCBINTLVL1_bm);
    TCD0.CTRLA = 0x00;
    TCD0.CNT = 0x0000;
}


//////////////////////////////////////////////////////////////////////////
/** Instantiate an instance of the ADC converter class in static memory.
 *  All files that include this header file will have access to this
 *  object.
 */
static ADC ADC ("ADC", &ADCA, &PORTA);


#endif /* ADC_H_ */
```

```
/*********************************************************************/
/** @file ADC.cpp
 *  @brief ADC converter main class file.
 *
 *
 *   Contributors:
 *       - Zach Wilson 2/17/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *       - 2/17/2019 - initial release.
 *
 *  Created: 2/17/2019 1:44:02 PM
 */
/*********************************************************************/


/* Included files. */
#include "ADC.h"

//////////////////////////////////////////////////////////////////////////
/** Pointer to the port of the ADC interface.
 */
PORT_t * ADC_channel :: p_PORT;


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the ADC
 *  channel class.
 */
ADC_channel :: ADC_channel (void) {}


//////////////////////////////////////////////////////////////////////////
/** @brief Select pin.
 *  @details This method ties a port pin to the positive end of the ADC
 *  comparator. It also configures the ADC channel and for single-ended
 *  conversion and should be called before performing a conversion on the
 *  channel.
 *  @param _pos_pin Bitmask of the pin tied to the positive side of the
 *  ADC comparator.
 */
void ADC_channel :: tie_pin (uint8_t _pos_pin)
{
    // Configure pin as input and set low.
    p_PORT->DIR &= ~_pos_pin;
    pos_pin = _pos_pin;

    // Select a gain of 1 and single-ended input mode.
    p_channel->CTRL = ADC_CH_INPUTMODE0_bm;

    // Connect the selected pins to the analog comparator.
    switch (pos_pin)
    {
        case PIN1_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS0_bm;                                              break;
        case PIN2_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS1_bm;                                              break;
        case PIN3_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS0_bm | ADC_CH_MUXPOS1_bm;                      break;
        case PIN4_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS2_bm;                                              break;
        case PIN5_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS2_bm | ADC_CH_MUXPOS0_bm;                      break;
        case PIN6_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS2_bm | ADC_CH_MUXPOS1_bm;                      break;
        case PIN7_bm:  p_channel->MUXCTRL = ADC_CH_MUXPOS2_bm | ADC_CH_MUXPOS1_bm | ADC_CH_MUXPOS1_bm;  break;
        default:    p_channel->MUXCTRL = ADC_CH_MUXPOS0_bm;                                              break;
    };

    // Enable high-priority interrupts on the ADC channel.
    p_channel->INTCTRL = ADC_CH_INTLVL0_bm | ADC_CH_INTLVL1_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Set ADC gain.
 *  @details This method sets the gain of the ADC channel being used.
 *  @param gain Number corresponding to the gain setting.
 */
void ADC_channel :: set_gain (uint8_t gain)
{
    // Set the gain of the channel according to the xmega au manual.
    switch (gain)
    {
        case 1:  p_channel->CTRL &= ~(ADC_CH_GAIN0_bm | ADC_CH_GAIN1_bm | ADC_CH_GAIN2_bm); break;
        case 2:  p_channel->CTRL |= ADC_CH_GAIN0_bm;
                  p_channel->CTRL &= ~(ADC_CH_GAIN1_bm | ADC_CH_GAIN2_bm);                    break;
        case 4:  p_channel->CTRL |= ADC_CH_GAIN1_bm;
                  p_channel->CTRL &= ~(ADC_CH_GAIN0_bm | ADC_CH_GAIN2_bm);                    break;
        case 8:  p_channel->CTRL |= ADC_CH_GAIN0_bm | ADC_CH_GAIN1_bm;
                  p_channel->CTRL &= ~ADC_CH_GAIN2_bm;                                         break;
        case 16: p_channel->CTRL |= ADC_CH_GAIN2_bm;
                  p_channel->CTRL &= ~(ADC_CH_GAIN0_bm | ADC_CH_GAIN1_bm);                    break;
        case 32: p_channel->CTRL |= ADC_CH_GAIN0_bm | ADC_CH_GAIN2_bm;
                  p_channel->CTRL &= ~ADC_CH_GAIN1_bm;                                         break;
        case 64: p_channel->CTRL |= ADC_CH_GAIN1_bm | ADC_CH_GAIN2_bm;
                  p_channel->CTRL &= ~ADC_CH_GAIN0_bm;                                         break;
```

```
        default: p_channel->CTRL &= ~(ADC_CH_GAIN0_bm | ADC_CH_GAIN1_bm | ADC_CH_GAIN2_bm); break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Read data.
 *  @details This method reads an A/D conversion from the specified port
 *  pin. It calls the request method and then waits until the the
 *  conversion is complete before returning the data, so there is some
 *  overhead consequences for calling this function.
 *  @return Converted A/D data from the specified pin.
 */
uint16_t ADC_channel :: read (void)
{
    // Wait until the channel is inactive, then initialize a ADC
    // conversion. Wait until the data is converted and stored in the
    // queue and then pull and return the data.
    while (p_channel->CTRL & ADC_CH_START_bm) {}
    p_channel->CTRL |= ADC_CH_START_bm;
    while (p_buffer->empty ()) {}
    return p_buffer->pull_data ();
}

//////////////////////////////////////////////////////////////////////////
/** ADC control struct for use in an ISR.
 */
ADC_CONTROL_STRUCT_t ADC :: ADC_control;


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the ADC
 *  converter interface on a specified port.
 *  @param ADC_name Pointer to a string for the interface name.
 *  @param ADC Pointer to the ADC interface.
 *  @param PORT Pointer to the port containing the ADC interface.
 */
ADC :: ADC (const char *ADC_name, ADC_t *ADC, PORT_t *PORT)
{
    p_ADC = ADC;
    p_PORT = PORT;
    ADC_channel :: p_PORT = PORT;
    p_name = &name [0];
    storeString (ADC_name, &name [0]);

    // Save pointers to the correct channels and buffers.
    channel0.p_channel = &p_ADC->CH0;
    channel1.p_channel = &p_ADC->CH1;
    channel2.p_channel = &p_ADC->CH2;
    channel3.p_channel = &p_ADC->CH3;
    channel0.p_buffer = &ADC_control.ch0_buffer;
    channel1.p_buffer = &ADC_control.ch1_buffer;
    channel2.p_buffer = &ADC_control.ch2_buffer;
    channel3.p_buffer = &ADC_control.ch3_buffer;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Initialize ADC interface.
 *  @details This method initializes the ADC interface. To actually read
 *  data, the desired pin must be set as input before attempting to read
 *  the data.
 *  @param serial Pointer to a serial port object for debugging.
 */
void ADC :: initialize (stream_base *serial)
{
    p_serial = serial;

    // Configure ADC interface for signed, 12-bit mode with a voltage
    // reference on pin 0 on port A.
    p_PORT->DIR &= ~PIN0_bm;
    p_ADC->CTRLA = ADC_FLUSH_bm;
    p_ADC->CTRLB = ADC_CONMODE_bm;
    p_ADC->REFCTRL = ADC_REFSEL1_bm;
    p_ADC->PRESCALER = ADC_PRESCALER1_bm;

    // Configure TCD0 for high-priority CCB compare interrupts. The timer
    // starts when the clock prescaler is selected.
    TCD0.CTRLA = 0x00;
    TCD0.CTRLB |= TC0_CCBEN_bm;
    TCD0.CTRLC |= TC0_CMPB_bm;
    TCD0.CNT = 0x00;
    TCD0.PER = 0xFFFF;

    // Send an indication that the ADC interface was initialized successfully.
    *p_serial << p_name << " interface initialized." << endl;
}


//////////////////////////////////////////////////////////////////////////
```

```
/** @brief Channel 0 ADC complete interrupt.
 *  @details This interrupt service routine places the result of the
 *  A/D conversion in the channel 0 buffer.
 */
ISR (ADCA_CH0_vect)
{
    ADC :: ADC_control.ch0_buffer.push_data (static_cast <uint16_t> (ADCA.CH0.RES));
}


//////////////////////////////////////////////////////////////////////////
/** @brief Channel 1 ADC complete interrupt.
 *  @details This interrupt service routine places the result of the
 *  A/D conversion in the channel 1 buffer.
 */
ISR (ADCA_CH1_vect)
{
    ADC :: ADC_control.ch1_buffer.push_data (static_cast <uint16_t> (ADCA.CH1.RES));
}


//////////////////////////////////////////////////////////////////////////
/** @brief Channel 2 ADC complete interrupt.
 *  @details This interrupt service routine places the result of the
 *  A/D conversion in the channel 2 buffer.
 */
ISR (ADCA_CH2_vect)
{
    ADC :: ADC_control.ch2_buffer.push_data (static_cast <uint16_t> (ADCA.CH2.RES));
}


//////////////////////////////////////////////////////////////////////////
/** @brief Channel 3 ADC complete interrupt.
 *  @details This interrupt service routine places the result of the
 *  A/D conversion in the channel 3 buffer.
 */
ISR (ADCA_CH3_vect)
{
    ADC :: ADC_control.ch3_buffer.push_data (static_cast <uint16_t> (ADCA.CH3.RES));
}


//////////////////////////////////////////////////////////////////////////
/** @brief TCD0 Channel B output compare ISR.
 *  @details This ISR is triggered on every output compare of channel B on
 *  TCD0. The data acquisition frequency is added to the compare register
 *  of channel A so that an interrupt occurs at a precise frequency. A
 *  conversion is requested on each active channel each time this ISR is
 *  triggered.
 */
ISR (TCD0_CCB_vect)
{
    TCD0.CCB += ADC :: ADC_control.timer_freq;
    ADCA.CTRLA |= ADC :: ADC_control.active_channels;
}
```

```
/*************************************************************************/
/** @file AD7708.h
 *  @brief AD7708 class header file.
 *
 *  @details This class provides methods for controlling the AD7708
 *  analog-to-digital converter device. The device is controlled and
 *  data is retrieved through an SPI interface.
 *
 *   Contributors:
 *      - Zach Wilson 3/16/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/16/2019 - initial release.
 *
 *  Created: 3/16/2019 11:05:45 AM
 */
/*************************************************************************/


#ifndef AD7708_H_
#define AD7708_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "SPI.h"
#include "serial_port.h"

//////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the device name.
 */
extern void storeString (const char *, char *);


//////////////////////////////////////////////////////////////////////////
/** Enum defining the addresses of the registers of the AD7708.
 */
typedef enum  AD7708_REGISTERS
{
    /** Communications register. All operations to other registers are
     *  initiated through the Communications Register. This controls
     *  whether subsequent operations are read or write operations and
     *  also selects the register for that subsequent operation.
     */
    COMM_REG = 0x00,

    /** Status register. Provides status information on conversions,
     *  calibrations and error conditions. This register is accessed by
     *  requesting a read from the communications register.
     */
    STATUS_REG = 0x00,

    /** Mode register. Controls functions such as mode of operation,
     *  channel configuration, oscillator operation in power-down.
     */
    MODE_REG = 0x01,

    /** Control register. This register is used to select the active
     *  channel input, configure the operating input range, and select
     *  unipolar or bipolar operation.
     */
    ADC_CONTROL_REG = 0x02,

    /** Filter register.  This register determines the amount of averaging
     *  performed by the sinc filter and consequently determines the data
     *  update rate of the AD7708. The filter register determines the
     *  update rate for operation with CHOP enabled and CHOP disabled.
     */
    FILTER_REG = 0x03,

    /** Data result register.  Provides the most up-to-date conversion
     *  result for the selected channel on the AD7708.
     */
    DATA_RESULT_REG = 0x04,

    /** Callibration register.  Contains a 16-bit word which is the offset
     *  calibration coefficient for the part. The contents of this register
     *  are used to provide offset correction on the output from the digital
     *  filter. There are five offset Registers on the part and these are
     *  associated with input channels as outlined in the ADCCON register.
     */
    CALLIBRATION_REG = 0x05,

    /** Gain register. Contains a 16-bit word which is the gain
     *  calibration coefficient for the part. The contents of this
     *  register are used to provide gain correction on the output
     *  from the digital filter. There are five Gain Registers on the part
     *  and these are associated with input channels as outlined in the
     *  ADCCON register.
     */
    GAIN_REG = 0x06,
```

```c
    /** Input/output control register. This register is used to control
     *  and configure the I/O port.
     */
    IO_CONTROL_REG = 0x07,

    /** Device ID register. Contains an 8-bit byte which is the identifier
     *  for the part.
     */
    DEV_ID_REG = 0x0F,
} AD7708_REGISTERS_t;


////////////////////////////////////////////////////////////////////////////
/** Enum defining the different modes of operation for the AD7708.
 */
typedef enum AD7708_MODE_REGISTER_ENUM
{
    /** Power down mode.
     */
    POWER_DOWN_MODE = 0x00,

    /** Idle mode. In Idle Mode the ADC filter and modulator are held in a
     *  reset state although the modulator clocks are still provided.
     */
    IDLE_MODE = 0x01,

    /** Single conversion mode. In Single Conversion Mode, a single
     *  conversion is performed on the enabled channels. On completion of
     *  the conversion the ADC data registers are updated, the relevant
     *  flags in the STATUS register are written, and idle mode is
     *  reentered with the MD2-MD0 being written accordingly to 001.
     */
    SINGLE_CONVERSION_MODE = 0x02,

    /** Continuous conversion mode. In continuous conversion mode, the ADC
     *  data registers are regularly updated at the selected update rate.
     */
    CONTINUOUS_CONVERSION_MODE = 0x03,

    /** Internal zero scale calibration mode. Internal short automatically
     *  connected to the enabled channel(s).
     */
    INTERNAL_ZERO_SCALE_CALIBRATION_MODE = 0x04,

    /** Internal full scale calibration mode. External VREF is connected
     *  automatically to the ADC input for this calibration.
     */
    INTERNAL_FULL_SCALE_CALIBRATION_MODE = 0x05,

    /** System zero scale calibration.
     */
    SYSTEM_ZERO_SCALE_CALIBRATION = 0x06,

    /** System full scale calibration.
     */
    SYSTEM_FULL_SCALE_CALIBRATION = 0x07,
} AD7708_MODE_REGISTER_t;


////////////////////////////////////////////////////////////////////////////
/** AD7708 control struct.
 */
typedef struct
{
    /** Boolean indicating if a reading has been requested.
     */
    volatile bool request_reading;

    /** Timer interrupt frequency.
     */
    volatile uint16_t timer_freq;

    /** Pointer to an SPI object for controlling an SPI interface.
     */
    SPI *p_SPI;

    /** Pointer to an I/O port.
     */
    PORT_t *p_PORT;

    /** Chip select pin bitmask.
     */
    uint8_t cs_pin;

    /** Device ready pin bitmask.
     */
    uint8_t ready_pin;
} AD7708_CONTROL_STRUCT_t;


////////////////////////////////////////////////////////////////////////////
```

```
/** @brief AD7708 class.
 *  @details This class provides methods for controlling the AD7708
 *  analog-to-digital converter device. The device is controlled and
 *  data is retrieved through an SPI interface.
 */
class AD7708
{
    private:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** Pointer to an SPI object.
         */
        SPI *p_SPI;

        /** Pointer to an I/O port.
         */
        PORT_t *p_PORT;

        /** Chip select pin bitmask.
         */
        uint8_t cs_pin;

        /** Device ready pin bitmask.
         */
        uint8_t ready_pin;

        /** Device ready pin number.
         */
        uint8_t ready_number;

        /** Device reset pin bitmask.
         */
        uint8_t reset_pin;

        /** Pin bitmask for controlling the excitation voltage of the
         *  wheatstone bridges.
         */
        uint8_t excite_pin;

        /** C-style string for device name.
         */
        char name [11];

    public:
        /** Pointer to the device name.
         */
        const char *p_name;

        /** Control struct for use in an ISR.
         */
        static AD7708_CONTROL_STRUCT_t ADC_control;

        /** 16-bit gain of the previously calibrated channel of the device.
         */
        uint16_t gain;

        /** @brief Default constructor.
         *  @details This constructor instantiates an instance of the object
         *  and saves a pointer to a serial port object, a pointer to an SPI
         *  interface, and an I/O port and pins for controlling the device.
         *  @param SPI Pointer to an SPI interface.
         *  @param PORT Pointer to a I/O port.
         *  @param _cs_pin Chip select pin bitmask.
         *  @param _ready_pin Device ready pin.
         *  @param _reset_pin Reset pin bitmask.
         *  @param _excite_pin Pin bitmask for controlling the wheatstone
         *  bridge excitation voltage.
         *  @param dev_name Pointer to a string of the device name.
         */
        AD7708 (SPI *, PORT_t *, uint8_t, uint8_t, uint8_t, uint8_t, const char *);

        /** @brief Initialize device.
         *  @details This method configures and initializes the SPI interface and
         *  GPIO pins for communicating to the device and configures the settings
         *  of the device. The device is reset which starts it in a low power
         *  mode.
         *  @param serial Pointer to a serial port.
         */
        void initialize (stream_base *);

        /** @brief Enable device.
         *  @details This method powers on and enables the device. It should be
         *  called before attempting to read data from the device.
         *  @return Status of external oscillator (true == oscillator locked).
         */
        bool enable (void);

        /** @brief Disable Device.
         *  @details This method disables and places the device in a low power
         *  state. It should be called when we are done reading data from the
```

```
		 *  device.
		 */
		void disable (void);

		/** @brief Select channel.
		 *  @details This method configures the device so that data from the
		 *  specified channel will be sent to the data result register to be
		 *  retrieved with calls to the request (), data_ready (), and
		 *  retrieve () methods.
		 *  @param channel Selected channel to read data from. Channel codes
		 *  provided in datasheet.
		 */
		void select_channel (uint8_t);

		/** @brief Calibrate device.
		 *  @details This method performs a zero-mode calibration and a full-scale
		 *  calibration of the selected channel of the device. It is not essential
		 *  that the system is in a zero-differential voltage state during the
		 *  calibration because the channel is shorted to ground internally during
		 *  the zero-scale calibration and the reference voltage during the full-
		 *  scale calibration.
		 */
		void calibrate (void);

		/** @brief Retrieve channel gain.
		 *  @details This method retrieves the 16-bit word in the gain register
		 *  for the selected channel of the device. This value is used to calculate
		 *  the analog signal level read by the ADC channel.
		 *  @return 16-bit number that is the calibrated gain for the selected c
		 *  channel.
		 */
		uint16_t retrieve_gain (void);

		/** @brief Device status.
		 *  @details This method returns a boolean indicating if the device has
		 *  valid data available in the current data register. This is checked by
		 *  checking if the device ready pin is low.
		 *  @return Boolean indicating if the device is ready to send valid data.
		 */
		inline bool device_ready (void);

		/** @brief Read data from channel.
		 *  @details This method reads one sample of data from the currently
		 *  selected channel of the device.
		 *  @return Assembled 16-bit data sample.
		 */
		uint16_t read (void);

		/** @brief Request reading from channel.
		 *  @details This method requests one sample of data from the selected
		 *  channel. The data is acquired in the SPI interrupt service routine.
		 *  Call the retrieve () function after the data has been transmitted to
		 *  acquire the data.
		 */
		void request (void);

		/** @brief Acquire dynamic data.
		 *  @details This method acquires data from the selected channel at a user-
		 *  specified frequency using output compare interrupts on a timer. The
		 *  data is acquired in an ISR and stored in the SPI receive queue until
		 *  the retrieve () method is called.
		 *  @param frequency Frequency at which to take data (in clock ticks).
		 */
		void acquire_data (uint16_t);

		/** @brief Stop taking dynamic data.
		 *  @details This method disables the timer and resets the count so that
		 *  output compare interrupts are no longer triggered.
		 */
		void stop_data (void);

		/** @brief Check if data has been transmitted.
		 *  @details This method returns a boolean indicating if data is ready to
		 *  be retrieved from the SPI interface.
		 *  @return Boolean indicating if data is available to retrieve.
		 */
		inline bool data_available (void);

		/** @brief Retrieve data.
		 *  @details This method returns one sample of data stored in the SPI
		 *  interface. The transfer_complete () method should be called before
		 *  calling this method to ensure that valid data is retrieved.
		 *  @return Assembled 16-bit data sample.
		 */
		uint16_t retrieve (void);
};


//////////////////////////////////////////////////////////////////////////
/** @brief Device status.
 *  @details This method returns a boolean indicating if the device has
 *  valid data available in the current data register. This is checked by
```

```
 *  checking if the device ready pin is low.
 *  @return Boolean indicating if the device is ready to send valid data.
 */
inline bool AD7708 :: device_ready (void)
{
    return !(p_PORT->IN & ready_pin);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Check if data has been transmitted.
 *  @details This method returns a boolean indicating if data is ready to
 *  be retrieved from the SPI interface. If there is at least two bytes
 *  in the SPI receive buffer, then there is at least one reading
 *  available.
 *  @return Boolean indicating if data is available to retrieve.
 */
inline bool AD7708 :: data_available (void)
{
    return p_SPI->data_available () > 1;
}


//////////////////////////////////////////////////////////////////////////
/** AD7708 amplifier object. All files that include this header will have
 *  access to this object.
 */
static AD7708 adc_amp (&SPI_C, &PORTC, PIN4_bm, PIN3_bm, PIN2_bm, PIN1_bm, "AD7708");


#endif /* AD7708_H_ */
```

```
/***********************************************************************/
/** @file AD7708.cpp
 *  @brief AD7708 main class file.
 *
 *
 *   Contributors:
 *      - Zach Wilson 3/16/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/16/2019 - initial release.
 *
 *  Created: 3/16/2019 11:52:25 AM
 */
/***********************************************************************/


/* Included files. */
#include "AD7708.h"

////////////////////////////////////////////////////////////////////////
/** Control struct for use in an ISR.
 */
AD7708_CONTROL_STRUCT_t AD7708 :: ADC_control;


////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor instantiates an instance of the object and
 *  saves a pointer to a serial port object, a pointer to an SPI
 *  interface, and an I/O port and pins for controlling the device.
 *  @param SPI Pointer to an SPI interface.
 *  @param PORT Pointer to a I/O port.
 *  @param _cs_pin Chip select pin bitmask.
 *  @param _ready_pin Device ready pin.
 *  @param _reset_pin Reset pin bitmask.
 *  @param _excite_pin Pin bitmask for controlling the wheatstone bridge
 *  excitation voltage.
 *  @param dev_name Pointer to a string of the device name.
 */
AD7708 :: AD7708 (SPI *_SPI, PORT_t *PORT, uint8_t _cs_pin, uint8_t _ready_pin, uint8_t _reset_pin, uint8_t _excite_pin, const
char *dev_name)
{
    p_SPI = _SPI;
    ADC_control.p_SPI = _SPI;
    ADC_control.p_PORT = PORT;
    ADC_control.cs_pin = _cs_pin;
    ADC_control.ready_pin = _ready_pin;
    ADC_control.request_reading = false;
    p_PORT = PORT;
    cs_pin = _cs_pin;
    ready_pin = _ready_pin;
    switch (_ready_pin)
    {
        case PIN0_bm:  ready_number = 0;
        case PIN1_bm:  ready_number = 1;
        case PIN2_bm:  ready_number = 2;
        case PIN3_bm:  ready_number = 3;
        case PIN4_bm:  ready_number = 4;
        case PIN5_bm:  ready_number = 5;
        case PIN6_bm:  ready_number = 6;
        case PIN7_bm:  ready_number = 7;
    };
    reset_pin = _reset_pin;
    excite_pin = _excite_pin;
    p_name = &name [0];
    storeString (dev_name, &name [0]);
}


////////////////////////////////////////////////////////////////////////
/** @brief Initialize device.
 *  @details This method configures and initializes the SPI interface and
 *  GPIO pins for communicating to the device and configures the settings
 *  of the device. The device is reset which starts it in a low power
 *  mode.
 *  @param serial Pointer to a serial port.
 */
void AD7708 :: initialize (stream_base *serial)
{
    p_serial = serial;

    // Configure the chip select, reset, and excitation control pins as
    // outputs and the ready pin as an input. Initialize the chip select
    // and reset pins high and the excitation control pin low.
    p_PORT->DIR |= cs_pin | reset_pin | excite_pin;
    p_PORT->DIR &= ~ready_pin;
    p_PORT->OUT |= cs_pin | reset_pin;
    p_PORT->OUT &= ~excite_pin;

    // Configure the high-priority interrupts for the ready pin on port C
    // for falling-edge sensing.
    p_PORT->INT0MASK = ready_pin;
```

```
    *(&p_PORT->PIN0CTRL + ready_number) = PORT_ISC1_bm;
    p_PORT->INTCTRL = PORT_INT0LVL0_bm | PORT_INT0LVL1_bm;

    // Configure TCD0 for high-priority CCA compare interrupts. The timer
    // starts when the clock prescaler is selected.
    TCD0.CTRLA = 0x00;
    TCD0.CTRLB |= TC0_CCAEN_bm;
    TCD0.CTRLC |= TC0_CMPA_bm;
    TCD0.CNT = 0x00;
    TCD0.PER = 0xFFFF;

    // Configure the SPI interface settings.
    p_SPI->initialize (p_serial, SPI_MASTER);
    p_SPI->set_mode (SPI_DIV8, SPI_MODE0, SPI_MSBFIRST);

    // Perform a hard reset by toggling the reset pin. This will start the
    // device in a low power state.
    p_PORT->OUT &= ~reset_pin;
    p_PORT->OUT |= reset_pin;

    // Confirm initialization through serial port.
    #ifdef  AD7708_DEBUG
        *p_serial << "AD7708 initialized." << endl;
    #endif
}


//////////////////////////////////////////////////////////////////////////
/** @brief Enable device.
 *  @details This method powers on and enables the device. The device is
 *  configured for continuous conversion mode, where each channel is
 *  updated at 1.3kHz, and an input range of +/- 40mV.
 *  @return Status of external oscillator (true == oscillator locked).
 */
bool AD7708 :: enable (void)
{
    // Push a boolean to the stack to store the status of the oscillator.
    bool status;

    // Initialize the ADC following steps laid out in datasheet.
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (FILTER_REG);
    p_SPI->transfer (0x03);            // Set ADC update frequency to 1.3kHz.
    p_SPI->transfer (ADC_CONTROL_REG);
    p_SPI->transfer (0b00000000);      // Set max range of 20mV.
    p_SPI->transfer (MODE_REG);
    p_SPI->transfer (0b10010000 | CONTINUOUS_CONVERSION_MODE);  // Start ADC in continuous conversion mode, chopping disabled.

    // Wait until the PLL has locked on to external oscillator.
    for (uint16_t index = 0;; index++)
    {
        p_SPI->transfer (0x40 | STATUS_REG);
        if (p_SPI->transfer (0x00) & 0x01)
        {
            #ifdef  AD7708_DEBUG
                *p_serial << "AD7708 PLL locked onto external oscillator." << endl << "AD7708 enabled." << endl;
            #endif
            status = true;
            break;
        }
        else if (index > 20000)
        {
            #ifdef  AD7708_DEBUG
                *p_serial << "AD7708 PLL failed to lock onto external oscillator." << endl;
            #endif
            disable ();
            status = false;
            break;
        }
    }

    // Assert the chip select pin and return the status of the oscillator.
    // Also assert the excitation control pin to turn on the wheatstone
    // bridges.
    p_PORT->OUT |= cs_pin | excite_pin;
    return status;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Disable Device.
 *  @details This method disables the wheatstone bridges and places the
 *  device in a low power state while not altering any other settings.
 *  Call the enable () method to power on the device again.
 */
void AD7708 :: disable (void)
{
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (MODE_REG);
    p_SPI->transfer (0b10010000 | POWER_DOWN_MODE);
    p_PORT->OUT |= cs_pin;
    p_PORT->OUT &= ~excite_pin;
```

```
    // Confirm the device was disabled.
    #ifdef  AD7708_DEBUG
        *p_serial << "AD7708 disabled." << endl;
    #endif
}


//////////////////////////////////////////////////////////////////////
/** @brief Select channel.
 *  @details This method configures the device so that data from the
 *  specified channel will be sent to the data result register to be
 *  retrieved with calls to the request (), data_ready (), and retrieve ()
 *  methods.
 *  @param channel Selected channel to read data from. Channel codes
 *  provided in datasheet.
 */
void AD7708 :: select_channel (uint8_t channel)
{
    // Transfer necessary bytes to the device so that the proper channel
    // is selected.
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (ADC_CONTROL_REG);
    p_SPI->transfer (channel | 0b00000000);
    p_PORT->OUT |= cs_pin;
}


//////////////////////////////////////////////////////////////////////
/** @brief Calibrate device.
 *  @details This method performs a zero-mode calibration and a full-scale
 *  calibration of the selected channel of the device. It is not essential
 *  that the system is in a zero-differential voltage state during the
 *  calibration because the channel is shorted to ground internally during
 *  the zero-scale calibration and the reference voltage during the full-
 *  scale calibration.
 */
void AD7708 :: calibrate (void)
{
    // Push a byte to the stack to store the status bits of the device.
    uint8_t status;

    // Write to the mode register and select zero-scale calibration.
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (MODE_REG);
    p_SPI->transfer (0b10010000 | INTERNAL_ZERO_SCALE_CALIBRATION_MODE);

    // Wait until the device is in idle mode which indicates the calibration
    // is complete.
    for (;;)
    {
        p_SPI->transfer (0x40 | MODE_REG);
        status = p_SPI->transfer (0x00) & 0x07;
        if (status == 0x01) break;
    }
    #ifdef  AD7708_DEBUG
        *p_serial << "Zero-scale calibration complete." << endl;
    #endif

    // Write to the mode register and select full-scale calibration.
    p_SPI->transfer (MODE_REG);
    p_SPI->transfer (0b10010000 | INTERNAL_FULL_SCALE_CALIBRATION_MODE);

    // Wait until the device is in idle mode which indicates the calibration
    // is complete.
    for (;;)
    {
        p_SPI->transfer (0x40 | MODE_REG);
        status = p_SPI->transfer (0x00) & 0x07;
        if (status == 0x01) break;
    }
    #ifdef  AD7708_DEBUG
        *p_serial << "Full-scale calibration complete." << endl;
    #endif

    // Reconfigure the device to continuous conversion mode.
    p_SPI->transfer (MODE_REG);
    p_SPI->transfer (0b10010000 | CONTINUOUS_CONVERSION_MODE);

    // Store the gain of the selected channel.
    gain = retrieve_gain ();
    p_PORT->OUT |= cs_pin;
}

//////////////////////////////////////////////////////////////////////
/** @brief Retrieve channel gain.
 *  @details This method retrieves the 16-bit word in the gain register
 *  for the selected channel of the device. This value is used to calculate
 *  the analog signal level read by the ADC channel.
 *  @return 16-bit number that is the calibrated gain for the selected
 *  channel.
 */
```

```
uint16_t AD7708 :: retrieve_gain (void)
{
    // Push 16 bits onto the stack for assembling the number.
    uint8_t msb, lsb;

    // Request a read from the device from the gain register and store
    // the result.
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (0x40 | GAIN_REG);
    msb = p_SPI->transfer (0xFF);
    lsb = p_SPI->transfer (0xFF);
    p_PORT->OUT |= cs_pin;

    // Assemble and return the value.
    return static_cast <uint16_t> (msb << 8 | lsb);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Read data from channel.
 *  @details This method reads one sample of data from the currently
 *  selected channel of the device.
 *  @return Assembled 16-bit data sample.
 */
uint16_t AD7708 :: read (void)
{
    // Push 16 bits onto the stack for assembling the data.
    uint8_t msb, lsb;

    // Wait until valid data exists in the register then retrieve the data.
    while (p_PORT->IN & ready_pin) {}
    p_PORT->OUT &= ~cs_pin;
    p_SPI->transfer (0x40 | DATA_RESULT_REG);
    msb = p_SPI->transfer (0xFF);
    lsb = p_SPI->transfer (0xFF);
    p_PORT->OUT |= cs_pin;

    // Return the assembled data.
    return static_cast <uint16_t> (msb << 8 | lsb);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Request reading from channel.
 *  @details This method requests one sample of data from the selected
 *  channel. The data is acquired in the SPI interrupt service routine.
 *  Call the retrieve () function after the data has been transmitted to
 *  acquire the data.
 *  @param channel Channel of the device to read the data from.
 */
void AD7708 :: request (void)
{
    // Check to see if valid data exists in the data register. If valid
    // data is in the register (ready pin low), then initiate an SPI
    // transfer. Otherwise, raise the request_reading boolean in the ADC
    // control struct so that a transfer will be initiated in an ISR.
    p_SPI->flush ();
    if (p_PORT->IN & ready_pin)
    {
        // Push the bytes necessary to request a channel read.
        p_SPI->push_byte (0x40 | DATA_RESULT_REG, SPI_WRITE);
        p_SPI->push_byte (0x00, SPI_READ);
        p_SPI->push_byte (0x00, SPI_READ);
        p_SPI->transfer_ISR (p_PORT, cs_pin);
    }
    else ADC_control.request_reading = true;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Acquire dynamic data.
 *  @details This method acquires data from the selected channel at a user-
 *  specified frequency using output compare interrupts on a timer. The
 *  data is acquired in an ISR and stored in the SPI receive queue until
 *  the retrieve () method is called.
 *  @param frequency Frequency at which to take data (in clock ticks).
 */
void AD7708 :: acquire_data (uint16_t frequency)
{
    // Save the user specified frequency and set the first output compare
    // value. Select the timer clock prescaler to start the timer.
    ADC_control.timer_freq = frequency * 125; // FREQ = frequency * (32,000,000 / (256 * 1000))
    TCD0.CNT = 0x0000;
    TCD0.CCA = ADC_control.timer_freq;
    TCD0.INTCTRLB |= TC0_CCAINTLVL0_bm | TC0_CCAINTLVL1_bm;
    TCD0.CTRLA = TC0_CLKSEL1_bm | TC0_CLKSEL2_bm;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Stop taking dynamic data.
 *  @details This method disables the timer and resets the count so that
```

```
 *  output compare interrupts are no longer triggered.
 */
void AD7708 :: stop_data (void)
{
    TCD0.INTCTRLB &= ~(TC0_CCAINTLVL0_bm | TC0_CCAINTLVL1_bm);
    TCD0.CTRLA = 0x00;
    TCD0.CNT = 0x0000;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Retrieve data.
 *  @details This method returns one sample of data stored in the SPI
 *  interface. The transfer_complete () method should be called before
 *  calling this method to ensure that valid data is retrieved.
 *  @return Assembled 16-bit data sample.
 */
uint16_t AD7708 :: retrieve (void)
{
    // Push 16 bits onto the stack for assembling the data.
    uint8_t msb, lsb;
    msb = p_SPI->pull_byte ();
    lsb = p_SPI->pull_byte ();
    return static_cast <uint16_t> (msb << 8 | lsb);
}


//////////////////////////////////////////////////////////////////////////
/** @brief Ready pin ISR.
 *  @details This ISR is triggered when the device ready pin is lowered,
 *  which indicates that the device has valid data ready to transfer. An
 *  SPI transmission is initiated if the request_reading boolean has been
 *  raised in either the timer output compare ISR or by a call to the
 *  request () method.
 */
ISR (PORTC_INT0_vect)
{
    if (AD7708 :: ADC_control.request_reading)
    {
        AD7708 :: ADC_control.p_SPI->push_byte (0x40 | DATA_RESULT_REG, SPI_WRITE);
        AD7708 :: ADC_control.p_SPI->push_byte (0x00, SPI_READ);
        AD7708 :: ADC_control.p_SPI->push_byte (0x00, SPI_READ);
        AD7708 :: ADC_control.p_SPI->transfer_ISR (AD7708 :: ADC_control.p_PORT, AD7708 :: ADC_control.cs_pin);
        AD7708 :: ADC_control.request_reading = false;
    }
}


//////////////////////////////////////////////////////////////////////////
/** @brief TCD0 Channel A output compare ISR.
 *  @details This ISR is triggered on every output compare of channel A on
 *  TCD0. The data acquisition frequency is added to the compare register
 *  of channel A so that an interrupt occurs at a precise frequency. The
 *  ready pin is polled to determine if the AD7708 has valid data in the
 *  data result register. If there is (ready pin low), an SPI transmission
 *  is initiated to retrieve the data. Otherwise, the request_reading
 *  boolean is raised and the transmission will be initiated in the ISR
 *  triggered by the falling edge of the ready pin.
 */
ISR (TCD0_CCA_vect)
{
    TCD0.CCA += AD7708 :: ADC_control.timer_freq;
    if (!(AD7708 :: ADC_control.p_PORT->IN & AD7708 :: ADC_control.ready_pin))
    {
        AD7708 :: ADC_control.p_SPI->push_byte (0x40 | DATA_RESULT_REG, SPI_WRITE);
        AD7708 :: ADC_control.p_SPI->push_byte (0x00, SPI_READ);
        AD7708 :: ADC_control.p_SPI->push_byte (0x00, SPI_READ);
        AD7708 :: ADC_control.p_SPI->transfer_ISR (AD7708 :: ADC_control.p_PORT, AD7708 :: ADC_control.cs_pin);
        AD7708 :: ADC_control.request_reading = false;
    }
    else AD7708 :: ADC_control.request_reading = true;
}
```

H. DYNAMIC DATA ACQUISITION SYSTEM SCHEDULER, BASE TASK, SHARE, AND QUEUE C++ FILES

```
/*********************************************************************/
/** @file coscheduler.cpp
 *  @brief Cooperative scheduler class header file.
 *
 *  @details This class provides a cooperative multitasking scheduler with
 *  methods for initializing the cooperative multitasking environment and
 *  running the scheduler. All members of this class are static because
 *  there should only be one of these objects ever existing at a time. The
 *  base task, share, and queue classes are friends of this class and
 *  automatically add derived objects to the appropriate scheduler list
 *  in their respective constructors.
 *
 *
 *   Contributors:
 *      - Zach Wilson 12/9/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/9/2018 - initial release.
 *
 *  Created: 12/9/2018 12:33:47 AM
 */
/*********************************************************************/


#ifndef COSCHEDULER_H_
#define COSCHEDULER_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "clock_timer_rtc.h"
#include "list.h"
#include "serial_port.h"
#include "queue.h"
#include "share.h"

//////////////////////////////////////////////////////////////////////////
/** Maximum tasks allowed in queue. Increase this number if more tasks are
 *  required.
 */
#define  COSCHEDULER_TASK_QUEUE  10

//////////////////////////////////////////////////////////////////////////
/** Define the task stack queue for use in context switching.
 */
#define  TASK_CONTROL_BLOCK_STACK_SIZE  50

//////////////////////////////////////////////////////////////////////////
/** This defines the IDLE keyword so that an idle task can be added at the
 *  end of the scheduler run list.
 */
#define  IDLE  0

//////////////////////////////////////////////////////////////////////////
/** Forward declaration of the task base class so the scheduler class can
 *  use its datatype.
 */
class task_base;


//////////////////////////////////////////////////////////////////////////
/** Enum defining the flags of the task queues, the idle task, and debug
 *  reports. The scheduler uses these flags to decide which task or debug
 *  report to attempt to run.
 */
typedef enum  TASK_QUEUE_PRIORITY_ENUM
{
    /** Debug report flag.
     */
    COSCHEDULER_DEBUG_REPORT_FLAG = 0,

    /** Idle task flag.
     */
    COSCHEDULER_IDLE_TASK_FLAG,

    /** Low priority task queue flag.
     */
    COSCHEDULER_LOW_PRIORITY_QUEUE_FLAG,

    /** Medium priority task queue flag.
     */
    COSCHEDULER_MED_PRIORITY_QUEUE_FLAG,
```

```
    /** High priority task queue flag.
     */
    COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG,
} TASK_QUEUE_PRIORITY_t;


//////////////////////////////////////////////////////////////////////
/** Enum defining the status of a task.
 */
typedef enum  TASK_STATUS_ENUM
{
    /** Flag indicating that the task is free and not queued.
     */
    FREE = 0,

    /** Flag indicating that the task is ready to run.
     */
    READY,

    /** Flag indicating that the task is inserted into the queue and
     *  awaiting a chance to run.
     */
    QUEUED,

    /** Flag indicating that the task yielded to the kernel cooperatively.
     */
    YIELD,
} TASK_STATUS_t;


//////////////////////////////////////////////////////////////////////
/** Definition of simple list struct to be used as the kernel for choosing
 *  which queue to scan while scheduling tasks.
 */
typedef struct
{
    /** Buffer to serve as storage for the list.
     */
    TASK_QUEUE_PRIORITY_t buffer [20];

    /** Pointer to the first value in the list.
     */
    TASK_QUEUE_PRIORITY_t *p_first;

    /** Pointer to the last value in the list.
     */
    TASK_QUEUE_PRIORITY_t *p_last;
} KERNEL_LIST_t;


//////////////////////////////////////////////////////////////////////
/** Task control block struct. The kernel uses this struct to control and
 *  schedule each task accordingly.
 */
struct  TASK_CONTROL_BLOCK_t
{
    /** Pointer to the current task.
     */
    task_base *p_task;

    /** Pointer to the next task control block.
     */
    TASK_CONTROL_BLOCK_t *p_next_block;

    /** Basic stack to be used for context switching. The actual task will
     *  not have access to this stack, it is to be used only by the kernel
     *  during a context switch.
     */
    volatile uint8_t task_stack [TASK_CONTROL_BLOCK_STACK_SIZE];

    /** Pointer to the task stack.
     */
    volatile uint8_t *volatile p_stack;

    /** Status of the task (FREE/READY/QUEUED).
     */
    TASK_STATUS_t status;

    /** Dynamic priority of the task. Not implemented yet.
     */
    uint8_t d_priority;
};


//////////////////////////////////////////////////////////////////////
/** @brief Cooperative scheduler class.
 *  @details This class provides a cooperative task scheduler to be
 *  implemented in a cooperative multitasking environment with multiple
 *  finite-state-machine style programs running pseudo-simultaneously. The
 *  scheduler object should be instantiated before share, queue, and task
 *  objects so the scheduler can save pointers to each object.
 */
```

```
class coscheduler
{
    /** Declare the base task class a friend so the base task constructor can
     *  add task pointers to the scheduler task lists.
     */
    friend class task_base;

    /** Declare the base share class a friend so the base share constructor
     *  can add share pointers to the share list.
     */
    friend class share_base;

    /** Declare the base queue class a friend so the base queue constructor
     *  can add queue pointers to the share list.
     */
    friend class queue_base;

    private:
        /** Pointer to a serial port object.
         */
        static stream_base *p_serial;

        /** Scheduler list to be used as the kernel for controlling which
         *  tasks are run. This list holds flags in the order in which the
         *  scheduler pulls tasks from each queue.
         */
        static KERNEL_LIST_t kernel_list;

        /** High priority task queue. Tasks with high priority (1-3) are
         *  added to this queue when the task is instantiated. The scheduler
         *  will attempt to run these tasks before any lower priority tasks
         *  or the idle task.
         */
        static circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> high_priority_queue;

        /** Medium priority task queue. Tasks with medium priority (4-7) are
         *  added to this queue when the task is instantiated. The scheduler
         *  will attempt to run these tasks if none of the high priority tasks
         *  are ready to run.
         */
        static circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> med_priority_queue;

        /** Low priority task queue. Tasks with low priority (8-10) are added
         *  to this queue when the task is instantiated. The scheduler will
         *  attempt to run these tasks if none of the high or medium priority
         *  tasks are ready to run.
         */
        static circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> low_priority_queue;

        /** Linked list of task control blocks. Task base pointers are added to
         *  this linked list when a task is instantiated. The scheduler uses this
         *  list to determine which task should run next. Tasks are added to this
         *  list in the order in which they are instantiated.
         */
        static TASK_CONTROL_BLOCK_t task_control [COSCHEDULER_FREE_TASK_LIST];

        /** Idle task storage. A task pointer is saved here when a task is
         *  instantiated and assigned IDLE priority. This is a pointer to an
         *  idle task to run only when no other tasks need to run.
         */
        static TASK_CONTROL_BLOCK_t idle_task_control;

        /** Task debug list. A task pointer is saved here when a task is
         *  instantiated. The scheduler uses this list to print debug
         *  information from each task during debug reports. Tasks are added to
         *  this list in order in which they are added to the scheduler.
         */
        static list <task_base *, COSCHEDULER_FREE_TASK_LIST> task_debug_list;

        /** Share list. Share base pointers are added to this list when a share
         *  object is instantiated. The scheduler uses this list to print debug
         *  information from each share during debug reports. Shares are added to
         *  this list in the order in which they are instantiated.
         */
        static list <share_base *, COSCHEDULER_SHARE_LIST> share_debug_list;

        /** Queue list. Queue base pointers are added to this list when a share
         *  object is instantiated. The scheduler uses this list to print debug
         *  information from each share during debug reports. Shares are added to
         *  this list in the order in which they are instantiated.
         */
        static list <queue_base *, COSCHEDULER_QUEUE_LIST> queue_debug_list;

        /** Number of scheduler iterations.
         */
        static uint32_t iteration_number;

        /** Scheduler debug struct for running task reports.
         */
        static debug_t scheduler_debug;

    public:
```

```
        /** Number of tasks in scheduler.
         */
        static uint8_t task_number;

        /** @brief Default constructor of Scheduler class.
         *  @details This constructor creates an instance of the scheduler class.
         *  @param serial Pointer to a serial port object.
         */
        coscheduler (stream_base *);

        /** @brief Initialize scheduler.
         *  @details This method initializes all components of the scheduler. It
         *  initializes the serial port, each task, share, and queue object
         *  added to each respective list and enables the clock.
         */
        static void initialize (void);

        /** @brief Run scheduler.
         *  @details This method runs the scheduler and should never exit so long as
         *  power is supplied to the device.
         */
        static void run (void);

        /** @brief Context switch.
         *  @details This method performs a context switch from one task to another in
         *  the preemptive multitasking environment. Not implemented yet.
         *  @param prev_task Task to switch from.
         *  @param next_task Task to switch to.
         */
        static void context_switch (task_base *, task_base *);

        /** Define COSCHEDULER_DEBUG in RTOS_defines.h to compile the methods below.
         */
        #ifdef  TASK_DEBUG

            /** @brief Task debug report.
             *  @details This method prints a debug report of all the tasks through the
             *  serial port. Information like number of transitions, number of task runs,
             *  average timing, and average frequency are reported.
             */
            static void taskDebugReport (void);

            /** @brief Task state report.
             *  @details This method prints the current state of each task queued in the
             *  scheduler for debugging.
             */
            static void taskStateReport (void);

            /** @brief Share debug report.
             *  @details This method prints a debug report of all the shares through the
             *  serial port.
             */
            static void shareDebugReport (void);

            /** @brief Queue debug report.
             *  @details This method prints a debug report of all the queues through the
             *  serial port.
             */
            static void queueDebugReport (void);

        #endif  /* COSCHEDULER_DEBUG */
};


#endif /* COSCHEDULER_H_ */
```

```
/**********************************************************************/
/** @file coscheduler.cpp
 *  @brief Cooperative scheduler main class file.
 *
 *   Contributors:
 *      - Zach Wilson 12/9/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/9/2018 - initial release.
 *
 *  Created: 12/9/2018 12:33:47 AM
 */
/**********************************************************************/


/* Included files. */
#include "coscheduler.h"

////////////////////////////////////////////////////////////////////////
/** Pointer to a serial port object.
 */
stream_base *coscheduler :: p_serial;

////////////////////////////////////////////////////////////////////////
/** Scheduler list to be used as the kernel for controlling which tasks
 *  are run. This list holds flags in the order in which the scheduler
 *  pulls tasks from each queue.
 */
KERNEL_LIST_t coscheduler :: kernel_list;

////////////////////////////////////////////////////////////////////////
/** High priority task queue. Tasks with high priority (1-3) are added to
 *  this queue when the task is instantiated. The scheduler will attempt
 *  to run these tasks before any lower priority tasks or the idle task.
 */
circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> coscheduler :: high_priority_queue;

////////////////////////////////////////////////////////////////////////
/** Medium priority task queue. Tasks with medium priority (4-7) are added
 *  to this queue when the task is instantiated. The scheduler will attempt
 *  to run these tasks if none of the high priority tasks are ready to run.
 */
circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> coscheduler :: med_priority_queue;

////////////////////////////////////////////////////////////////////////
/** Low priority task queue. Tasks with low priority (8-10) are added to
 *  this queue when the task is instantiated. The scheduler will attempt to
 *  run these tasks if none of the high or medium priority tasks are ready
 *  to run.
 */
circular_buffer <TASK_CONTROL_BLOCK_t *, COSCHEDULER_TASK_QUEUE> coscheduler :: low_priority_queue;

////////////////////////////////////////////////////////////////////////
/** Linked list of task control blocks. Task base pointers are added to
 *  this linked list when a task is instantiated. The scheduler uses this
 *  list to determine which task should run next. Tasks are added to this
 *  list in order of the user specified priority.
 */
TASK_CONTROL_BLOCK_t coscheduler :: task_control [COSCHEDULER_FREE_TASK_LIST];

////////////////////////////////////////////////////////////////////////
/** Idle task storage. This is a pointer to an idle task to run only when
 *  no other tasks need to run.
 */
TASK_CONTROL_BLOCK_t coscheduler :: idle_task_control;

////////////////////////////////////////////////////////////////////////
/** Task debug list. A task pointer is saved here when a task is
 *  instantiated. The scheduler uses this list to print debug information
 *  from each task during debug reports. Tasks are added to this list in
 *  order in which they are added to the scheduler.
 */
list <task_base *, COSCHEDULER_FREE_TASK_LIST> coscheduler :: task_debug_list;

////////////////////////////////////////////////////////////////////////
/** Share list. Share base pointers are added to this list when a share
 *  object is instantiated. The scheduler uses this list to print debug
 *  information from each share during debug reports. Shares are added to
 *  this list in the order in which they are instantiated.
 */
list <share_base *, COSCHEDULER_SHARE_LIST> coscheduler :: share_debug_list;

////////////////////////////////////////////////////////////////////////
/** Queue list. Queue base pointers are added to this list when a share
 *  object is instantiated. The scheduler uses this list to print debug
 *  information from each share during debug reports. Shares are added to
 *  this list in the order in which they are instantiated.
 */
list <queue_base *, COSCHEDULER_QUEUE_LIST> coscheduler :: queue_debug_list;

////////////////////////////////////////////////////////////////////////
/** Number of tasks in scheduler.
```

```
 */
uint8_t coscheduler :: task_number;

//////////////////////////////////////////////////////////////////////////
/** Number of scheduler iterations.
 */
uint32_t coscheduler :: iteration_number;

//////////////////////////////////////////////////////////////////////////
/** Scheduler debug struct for running task reports.
 */
debug_t coscheduler :: scheduler_debug;


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor of Scheduler class.
 *  @details This constructor creates an instance of the scheduler class.
 *  @param serial Pointer to a serial port object.
 */
coscheduler :: coscheduler (stream_base *serial)
{
    p_serial = serial;
    task_number = 0;
    iteration_number = 0;

    // Flush all lists, queues and the idle task for initialization. This is done
    // in the list constructor but is done here just to be safe.
    kernel_list.p_first = &kernel_list.buffer [0];
    kernel_list.p_last = &kernel_list.buffer [0];
    high_priority_queue.flush_buffer ();
    med_priority_queue.flush_buffer ();
    low_priority_queue.flush_buffer ();
    task_debug_list.flush ();
    share_debug_list.flush ();
    queue_debug_list.flush ();
    idle_task_control.p_task = NULL;

    // Initialize the task control block linked list by linking the next block
    // in the list to the previous. The last link in the list will point to a
    // NULL value, indicating the end of the list.
    uint8_t task_index = 0;
    while (task_index < (COSCHEDULER_FREE_TASK_LIST - 1))
    {
        task_control [task_index].p_task = NULL;
        task_control [task_index].p_stack = &task_control [task_index].task_stack [TASK_CONTROL_BLOCK_STACK_SIZE - 1];
        task_control [task_index].p_next_block = &task_control [task_index + 1];
        task_index++;
    }
    task_index++;
    task_control [task_index].p_next_block = NULL;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Initialize scheduler.
 *  @details This method initializes all components of the scheduler. It
 *  initializes the serial port, each task, share, and queue object added
 *  to each respective list and enables the clock.
 */
void coscheduler :: initialize (void)
{
    // Initialize the scheduler list.
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_MED_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_LOW_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_MED_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_IDLE_TASK_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_MED_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_LOW_PRIORITY_QUEUE_FLAG;
    *kernel_list.p_last++ = COSCHEDULER_DEBUG_REPORT_FLAG;
    *kernel_list.p_last = COSCHEDULER_MED_PRIORITY_QUEUE_FLAG;

    // Save a task control block pointer that points to the first value
    // in the linked list and an index for bookkeeping.
    TASK_CONTROL_BLOCK_t * p_task_control = &task_control [0];
    uint8_t index = 0;

    // Cycle through the run list until all tasks have been initialized.
    while (p_task_control)
    {
        // Run a task and increment the task count if a task exists at the
        // current position in the list.
        if (p_task_control->p_task)
        {
            p_task_control->p_task->initialize ();
        }
```

```
        // Set the pointer to point to the next task control block.
        p_task_control = p_task_control->p_next_block;
    }

    // Initialize the idle task if one is in the scheduler.
    if (idle_task_control.p_task)
    {
        idle_task_control.p_task->initialize ();
    }
    *p_serial << endl;

    // Cycle through the share list until all shares have been initialized.
    for (index = 0; index < COSCHEDULER_SHARE_LIST; index++)
    {
        // Run a task and increment the task count if a task exists at the
        // current position in the list.
        if (share_debug_list.read (index))
        {
            share_debug_list.read (index)->initialize ();
        }
    }
    *p_serial << endl;

    // Cycle through the queue list until all queues have been initialized.
    for (uint8_t index = 0; index < COSCHEDULER_QUEUE_LIST; index++)
    {
        // Run a task and increment the task count if a task exists at the
        // current position in the list.
        if (queue_debug_list.read (index))
        {
            queue_debug_list.read (index)->initialize ();
        }
    }
    *p_serial << endl;

    // Indicate that the scheduler was initialized successfully.
    *p_serial << "Cooperative multitasking environment initialized." << endl << endl;
}


//////////////////////////////////////////////////////////////////////
/** @brief Run scheduler.
 *  @details This method runs the scheduler and never exits. It determines
 *  which tasks to run based on priority and timing parameters. Debug
 *  reports requested from the tasks are run here as well.
 */
void coscheduler :: run (void)
{
    // Start the task control pointer at the start of the linked list.
    TASK_CONTROL_BLOCK_t *p_task_control = &task_control [0];
    uint8_t index = 0;

    // Add all of the tasks in the scheduler to the appropriate task queue.
    for (; index < task_number; index++)
    {
        // Determine which queue to add the task to.
        if (p_task_control->p_task->priority < 4)
        {
            // Add task to high priority queue if the priority is (1-3)
            high_priority_queue.push_data (p_task_control);
            p_task_control->status = QUEUED;
            #ifdef  TASK_DEBUG
                *p_serial << p_task_control->p_task->p_name << " task pushed to high priority queue." << endl;
            #endif
        }
        else if (p_task_control->p_task->priority < 8)
        {
            // Add task to medium priority queue if the priority is (4-7)
            med_priority_queue.push_data (p_task_control);
            p_task_control->status = QUEUED;
            #ifdef  TASK_DEBUG
                *p_serial << p_task_control->p_task->p_name << " task pushed to medium priority queue." << endl;
            #endif
        }
        else
        {
            // Add task to low priority queue if the priority is (8-10)
            low_priority_queue.push_data (p_task_control);
            p_task_control->status = QUEUED;
            #ifdef  TASK_DEBUG
                *p_serial << p_task_control->p_task->p_name << " task pushed to low priority queue." << endl;
            #endif
        }

        // Move to the next location in the linked list.
        p_task_control = p_task_control->p_next_block;
    }

    // Save each queue capacity on the stack to avoid excessive calls of
    // buf_capacity ().
    uint8_t high_queue_capacity = high_priority_queue.buf_capacity ();
    uint8_t med_queue_capacity = med_priority_queue.buf_capacity ();
```

```
uint8_t low_queue_capacity = low_priority_queue.buf_capacity ();

// Configure and enable the clock on timer TCD1 that counts milliseconds.
// This will enable the clock that the scheduler uses for task scheduling.
clock.enable ();

// Indicate that we have started cooperative multitasking.
*p_serial << endl << "Starting cooperative multitasking." << endl << endl;

// Infinite loop that runs the scheduler.
for (;;)
{
    // RTOS kernel. Runs a group of tasks according to what flag the
    // scheduler list pointer is pointing to.
    switch (*kernel_list.p_first)
    {
        case  COSCHEDULER_HIGH_PRIORITY_QUEUE_FLAG:
            // Reset the index variable.
            index = 0;

            // Attempt to run the high priority tasks.
            for (;index < high_queue_capacity;)
            {
                // Pull a task from the high priority queue and see if it is ready
                // to run. Run the task and store debug information.
                p_task_control = high_priority_queue.pull_data ();
                p_task_control->status = FREE;
                if (p_task_control->p_task->ready ())
                {
                    #ifdef  TASK_DEBUG
                        p_task_control->p_task->begin ();
                    #endif
                    p_task_control->p_task->run ();
                    #ifdef  COSCHEDULER_DEBUG
                        *p_serial << p_task_control->p_task->p_name << " task ran." << endl;
                    #endif
                    #ifdef  TASK_DEBUG
                        p_task_control->p_task->end ();
                    #endif
                }

                // Reinsert the task into the queue and increment the index.
                high_priority_queue.push_data (p_task_control);
                p_task_control->status = QUEUED;
                index++;
            }
            break;
        case  COSCHEDULER_MED_PRIORITY_QUEUE_FLAG:
            // Reset the index variable.
            index = 0;

            // Attempt to run the medium priority tasks.
            for (;index < med_queue_capacity;)
            {
                // Pull a task from the medium priority queue and see if it is ready
                // to run. Run the task and store debug information.
                p_task_control = med_priority_queue.pull_data ();
                p_task_control->status = FREE;
                if (p_task_control->p_task->ready ())
                {
                    #ifdef  TASK_DEBUG
                        p_task_control->p_task->begin ();
                    #endif
                    p_task_control->p_task->run ();
                    #ifdef  COSCHEDULER_DEBUG
                        *p_serial << p_task_control->p_task->p_name << " task ran." << endl;
                    #endif
                    #ifdef  TASK_DEBUG
                        p_task_control->p_task->end ();
                    #endif
                }

                // Reinsert the task into the queue and increment the index.
                med_priority_queue.push_data (p_task_control);
                p_task_control->status = QUEUED;
                index++;
            }
            break;
        case  COSCHEDULER_LOW_PRIORITY_QUEUE_FLAG:
            // Reset the index variable.
            index = 0;

            // Attempt to run the low priority tasks.
            for (;index < low_queue_capacity;)
            {
                // Pull a task from the medium priority queue and see if it is ready
                // to run. Run the task and store debug information.
                p_task_control = low_priority_queue.pull_data ();
                p_task_control->status = FREE;
                if (p_task_control->p_task->ready ())
                {
                    #ifdef  TASK_DEBUG
```

```
                            p_task_control->p_task->begin ();
                        #endif
                        p_task_control->p_task->run ();
                        #ifdef  COSCHEDULER_DEBUG
                            *p_serial << p_task_control->p_task->p_name << " task ran." << endl;
                        #endif
                        #ifdef  TASK_DEBUG
                            p_task_control->p_task->end ();
                        #endif
                    }

                    // Reinsert the task into the queue and increment the index.
                    low_priority_queue.push_data (p_task_control);
                    p_task_control->status = QUEUED;
                    index++;
                }
                break;
            case  COSCHEDULER_IDLE_TASK_FLAG:
                // Run the idle task if it has reached the next interval.
                if (idle_task_control.p_task && idle_task_control.p_task->ready ())
                {
                    // Call the begin function to store debug information if debug mode is
                    // enabled.
                    #ifdef  TASK_DEBUG
                        idle_task_control.p_task->begin ();
                    #endif
                    idle_task_control.p_task->run ();
                    #ifdef  COSCHEDULER_DEBUG
                        *p_serial << "Idle task ran." << endl;
                    #endif
                    #ifdef  TASK_DEBUG
                        idle_task_control.p_task->end ();
                    #endif

                    // Set the status of the task to FREE and indicate that a task run.
                    idle_task_control.status = FREE;
                }
                break;
            case  COSCHEDULER_DEBUG_REPORT_FLAG:
                // Check to see if any the debug flags have been raised to run a debug report.
                #ifdef  TASK_DEBUG
                    if (scheduler_debug.task_debug)        taskDebugReport ();
                    else if (scheduler_debug.task_state)   taskStateReport ();
                    else if (scheduler_debug.share_debug)  shareDebugReport ();
                    else if (scheduler_debug.queue_debug)  queueDebugReport ();
                #endif
                break;
            default:
                break;
        };

        // Increment the scheduler list pointer and roll over to the beginning
        // of the list if the pointer went out of bounds.
        kernel_list.p_first != kernel_list.p_last ? kernel_list.p_first++
        : kernel_list.p_first = &kernel_list.buffer [0];

        // Save the  iteration number of the scheduler.
        #ifdef  TASK_DEBUG
            iteration_number++;
        #endif
    }
}


//////////////////////////////////////////////////////////////////////////
/** @brief Context switch.
 *  @details This method performs a context switch from one task to another
 *  in the preemptive multitasking environment. Not implemented yet.
 *  @param prev_task Task to switch from.
 *  @param next_task Task to switch to.
 */
void coscheduler :: context_switch (task_base * prev_task, task_base * next_task)
{
    __asm__ __volatile__
    (
        "push  r31             \n\t"
        "push  r30             \n\t"
        "ldi   r30, prev_task  \n\t"
        "st    -Z, r0          \n\t"
        "st    -Z, r1          \n\t"
        "st    -Z, r2          \n\t"
        "st    -Z, r3          \n\t"
        "st    -Z, r4          \n\t"
        "st    -Z, r5          \n\t"
        "st    -Z, r6          \n\t"
        "st    -Z, r7          \n\t"
        "st    -Z, r8          \n\t"
        "st    -Z, r9          \n\t"
        "st    -Z, r10         \n\t"
        "st    -Z, r11         \n\t"
        "st    -Z, r12         \n\t"
        "st    -Z, r13         \n\t"
```

```
        "st    -Z, r14         \n\t"
        "st    -Z, r15         \n\t"
        "st    -Z, r16         \n\t"
        "st    -Z, r17         \n\t"
        "st    -Z, r18         \n\t"
        "st    -Z, r19         \n\t"
        "st    -Z, r20         \n\t"
        "st    -Z, r21         \n\t"
        "st    -Z, r22         \n\t"
        "st    -Z, r23         \n\t"
        "st    -Z, r24         \n\t"
        "st    -Z, r25         \n\t"
        "st    -Z, r26         \n\t"
        "st    -Z, r27         \n\t"
        "st    -Z, r28         \n\t"
        "st    -Z, r29         \n\t"
        "pop   r28             \n\t"
        "pop   r29             \n\t"
        "st    -Z, r28         \n\t"
        "st    -Z, r29         \n\t"
        "pop   r28             \n\t"
        "pop   r29             \n\t"
        "st    -Z, r28         \n\t"
        "st    -Z, r29         \n\t"
        :
        :
        :
    );
}


/* Define COSCHEDULER_DEBUG in RTOS_defines.h to compile the methods below.
 */
#ifdef  TASK_DEBUG
    ////////////////////////////////////////////////////////////////////////////
    /** @brief Task debug report.
     *  @details This method prints a debug report of all the tasks through
     *  the serial port. Information like number of transitions, number of task
     *  runs, average timing, and average frequency are reported.
     */
    void coscheduler :: taskDebugReport (void)
    {
        // Declare a task pointer to avoid multiple calls of debug_list.read ().
        task_base *task_pointer;

        // First create a header for the debug report. Print scheduler debug information.
        *p_serial << endl << "              TASK DEBUG REPORT" << endl << endl
                          << "Program time:   " << clock.millis_32 () << " ms" << endl
                          << "Iteration number:   " << iteration_number << endl << endl
                          << " TASK  |     NAME    | RUNS | AVG T | AVG F" << endl << endl;

        // Loop through the task list and display their debug information.
        for (uint8_t index = 0; index < COSCHEDULER_FREE_TASK_LIST; index++)
        {
            // Save a pointer to the current task.
            task_pointer = task_debug_list.read (index);

            // Print the task debugging information if a task exists at the buffer
            // location and increment the task count.
            if (task_pointer)
            {
                *p_serial << "   " << (index + 1) << "        "
                          << task_pointer->p_name << "       "
                          << task_pointer->run_number << "      "
                          << task_pointer->avg_time  <<   "     "
                          << task_pointer->avg_freq << endl;
            }
        }

        // Check if an idle task exists.
        if (idle_task_control.p_task)
        {
            *p_serial << " Idle      "
                      << idle_task_control.p_task->p_name <<      "       "
                      << idle_task_control.p_task->run_number <<  "      "
                      << idle_task_control.p_task->avg_time <<    "      "
                      << idle_task_control.p_task->avg_freq << endl;
        }

        // Reset the debug flag and send one more endline character.
        scheduler_debug.task_debug = false;
        *p_serial << endl;
    }


    ////////////////////////////////////////////////////////////////////////////
    /** @brief Task state report.
     *  @details This method prints the current state of each task queued in
     *  the scheduler for debugging.
     */
    void coscheduler :: taskStateReport (void)
    {
```

```
    // Declare a task pointer to avoid multiple calls of debug_list.read ().
    task_base *task_pointer;

    // First create a header for the state report.
    *p_serial << endl << "              TASK STATE REPORT" << endl << endl;
    *p_serial << " TASK  |     NAME    |  STATE  |  TRANSITIONS" << endl << endl;

    // Loop through the task list and display their state information.
    for (uint8_t index = 0; index < COSCHEDULER_FREE_TASK_LIST; index++)
    {
        // Save a pointer to the current task.
        task_pointer = task_debug_list.read (index);

        // Print the task debugging information if a task exists at the buffer
        // location and increment the task count.
        if (task_pointer)
        {
            *p_serial << "   " << (index + 1) << "    "
                        << task_pointer->p_name << "           "
                        << task_pointer->state << "          "
                        << task_pointer->transition_number << endl;
        }
    }

    // Check if an idle task exists at the end of the queue.
    if (idle_task_control.p_task)
    {
        *p_serial << " Idle    "
                    << idle_task_control.p_task->p_name << "           "
                    << idle_task_control.p_task->state << "          "
                    << idle_task_control.p_task->transition_number << endl;
    }

    // Reset the state flag and send one more endline character.
    scheduler_debug.task_state = false;
    *p_serial << endl;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Share debug report.
 *  @details This method prints a debug report of all the shares through
 *  the serial port.
 */
void coscheduler :: shareDebugReport (void)
{
    // Declare a queue pointer to avoid multiple calls of queue_list.read ().
    share_base *share_pointer;

    // First create a header for the state report.
    *p_serial << endl << "              SHARE DEBUG REPORT" << endl << endl;
    *p_serial << " SHARE  |     NAME    |  SENDS  |  RETRIEVES" << endl << endl;

    // Loop through the queue list and display their debug information.
    for (uint8_t index = 0; index < COSCHEDULER_SHARE_LIST; index++)
    {
        // Save a pointer to the current task.
        share_pointer = share_debug_list.read (index);

        // Print the task debugging information if a task exists at the buffer
        // location and increment the task count.
        if (share_pointer)
        {
            *p_serial << "   " << (index + 1) << "         "
            << share_pointer->p_name <<           "        "
            << share_pointer->send_number <<      "       "
            << share_pointer->retrieve_number << endl;
        }
    }

    // Reset the state flag and send one more endline character.
    scheduler_debug.share_debug = false;
    *p_serial << endl;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Queue debug report.
 *  @details This method prints a debug report of all the queues through
 *  the serial port.
 */
void coscheduler :: queueDebugReport (void)
{
    // Declare a queue pointer to avoid multiple calls of queue_list.read ().
    queue_base *queue_pointer;

    // First create a header for the state report.
    *p_serial << endl << "              QUEUE DEBUG REPORT" << endl << endl;
    *p_serial << " QUEUE  |     NAME    |  CAP  |  PUSHES  |  PULLS" << endl << endl;

    // Loop through the queue list and display their debug information.
    for (uint8_t index = 0; index < COSCHEDULER_QUEUE_LIST; index++)
```

```
        {
            // Save a pointer to the current task.
            queue_pointer = queue_debug_list.read (index);

            // Print the task debugging information if a task exists at the buffer
            // location and increment the task count.
            if (queue_pointer)
            {
                *p_serial << "   " << (index + 1) << "        "
                << queue_pointer->p_name <<           "        "
                << queue_pointer->capacity () <<     "          "
                << queue_pointer->push_number <<     "       "
                << queue_pointer->pull_number << endl;
            }
        }

        // Reset the state flag and send one more endline character.
        scheduler_debug.queue_debug = false;
        *p_serial << endl;
    }
#endif  /* TASK_DEBUG */
```

```
/***********************************************************************/
/** @file task_base.h
 *  @brief Base task class header file.
 *
 *  @details This class is used as a base class to higher-level user-
 *  defined task classes so that each task can use the variables and
 *  methods defined here. It is also used so that a cooperative or
 *  preemptive scheduler can save pointers to each task, have access to
 *  the timing and debugging members defined here, and call the virtual
 *  methods initialize () and run () that will be defined in the derived
 *  task classes.
 *
 *   Contributors:
 *      - Zach Wilson 12/9/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/9/2018 - initial release.
 *
 *  Created: 12/9/2018 12:33:47 AM
 */
/************************************************************************/


#ifndef TASK_BASE_H_
#define TASK_BASE_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "serial_port.h"
#include "stack.h"

////////////////////////////////////////////////////////////////////////
/** Forward declaration of coscheduler class so the task base class can
 *  declare the coscheduler a friend.
 */
class coscheduler;

////////////////////////////////////////////////////////////////////////
/** Forward declaration of share base class so the task base class can
 *  declare the share base class a friend.
 */
class share_base;

////////////////////////////////////////////////////////////////////////
/** Forward declaration of queue base class so the task base class can
 *  declare the task base class a friend.
 */
class queue_base;

////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the desired task name.
 */
extern void storeString (const char *, char *);


////////////////////////////////////////////////////////////////////////
/** Definition of the global flag struct. This struct may be used in lieu
 *  of declaring a share for a simple global boolean.
 */
typedef struct
{
    /** Read HX711 flag. Setting this flag will cause the HX711 task to
     *  begin reading data from the HX711.
     */
    bool read_HX711;

    /** Read static data flag. Setting this flag will cause the AD7708 task
     *  to begin reading static data at a specified frequency.
     */
    bool read_static_AD7708;

    /** Read dynamic data flag. Setting this flag will cause the AD7708 task to
     *  begin reading data from the AD7708 at a specified frequency.
     */
    bool read_dynamic_AD7708;

    /** Read static data flag. Setting this flag will cause the ADC task to begin
     *  taking static data.
     */
    bool read_static_ADC;

    /** Read dynamic data flag. Setting this flag will cause the ADC task to begin
     *  taking readings from the ADC interface.
     */
    bool read_dynamic_ADC;

    /** Saving data flag. This flag is set when the microSD task is saving dynamic
     *  data so the user interface task doesn't prematurely ask the user to enter
     *  another command.
     */
```

```
    bool saving_dynamic_data;

    /** Data acquisition mode. This flag indicates if the device is in static or
     *  dynamic data acquisition mode (static == false, dynamic == true).
     */
    bool daq_mode;

    /** Data acquisition method. This flag indicates if the device is collecting
     *  data from the AD7708 or the on-board ADC interface (AD7708 == false,
     *  ADC == true).
     */
    bool daq_method;
} GLOBAL_FLAG_STRUCT_t;


//////////////////////////////////////////////////////////////////////////
/** @brief Base task class.
 *  @details This class defines a base class for a task to be implemented
 *  in a finite-state-machine style program in a cooperative multitasking
 *  environment.
 */
class task_base
{
    /** Declare the scheduler as a friend so the scheduler can access
     *  the task state.
     */
    friend class coscheduler;

    /** Declare the share base class as a friend so the share base
     *  constructor can add shares to the share list.
     */
    friend class share_base;

    /** Declare the queue base class as a friend so the queue base
     *  constructor can add queues to the queue list.
     */
    friend class queue_base;

    private:
        /** 32-bit storage for program time at which to run next iteration.
         */
        uint32_t next_time;

        /** 8-bit storage for program time at which to run next iteration.
         *  This is used for debugging only.
         */
        uint8_t next_time_8bit;

        /** 8-bit storage for storing the most recent task start/end time.
         *  This is used for debugging only.
         */
        uint8_t task_time;

    protected:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** C-style string for task name.
         */
        char name [11];

        /** Task state variable that defines the current state of the
         *  finite-state-machine.
         */
        uint8_t state;

        /** Share list. Share base pointers are added to this list when
         *  a share object is instantiated. Tasks can use this list to
         *  save pointers to the shares they need access to. Shares are
         *  added to this list in the order in which they are
         *  instantiated.
         */
        static list <share_base *, COSCHEDULER_SHARE_LIST> share_list;

        /** Queue list. Queue base pointers are added to this list when
         *  a queue object is instantiated. Tasks can use this list to save
         *  pointers to the queues they need access to. Queues are added to
         *  this list in the order in which they are instantiated.
         */
        static list <queue_base *, COSCHEDULER_QUEUE_LIST> queue_list;

        /** This struct contains global flags that can be read/altered in
         *  any task if there is a need for one task to cause another to
         *  transition based on events such as user commands.
         */
        static GLOBAL_FLAG_STRUCT_t global_flags;

        /** @brief Default constructor of Task class.
         *  @details This constructor creates an instance of the task
         *  class. It should only be called by the constructor of a
         *  derived class and nothing else. Priority and timing for each
```

```
     *  task is assigned here. The IDLE keyword should be used to
     *  assign the idle task.
     *  @param serial Pointer to a serial port object.
     *  @param task_name Pointer to the task name string.
     *  @param _priority Priority at which to schedule the task.
     *  @param _timing Timing at which to run the task.
     */
    task_base (serial_port *, const char *, uint8_t, uint8_t);

    /** @brief State transition method.
     *  @details This method causes the task to transition to a
     *  specified state on the next iteration and increments the
     *  number of transitions of the task.
     *  @param new_state Specified state to transition to at next
     *  iteration.
     */
    inline void transition_to (uint8_t);

    /** @brief Delay task.
     *  @details This method causes the task to sleep until its next
     *  run time. This should only be used in a preemptive environment.
     */
    void delay_task (void);

    /** @brief Raise task debug flag.
     *  @details This function raises the task debug report flag to
     *  run a task debug report from the scheduler. It should be
     *  called from a user- interface style-task in which commands are
     *  received through a serial port.
     */
    void runDebugReport (void);

    /** @brief Raise task state flag.
     *  @details This function raises the task state report flag to
     *  run a task state report from the scheduler. It should be
     *  called from a user-interface style-task in which commands are
     *  received through a serial port.
     */
    void runStateReport (void);

    /** @brief Raise share debug flag.
     *  @details This function raises the share debug report flag to
     *  run a share debug report from the scheduler. It should be
     *  called from a user-interface style-task in which commands are
     *  received through a serial port.
     */
    void runShareReport (void);

    /** @brief Raise queue debug flag.
     *  @details This function raises the queue debug report flag to
     *  run a queue debug report from the scheduler. It should be
     *  called from a user-interface style-task in which commands are
     *  received through a serial port.
     */
    void runQueueReport (void);

public:
    /** Pointer to a string for the task name.
     */
    const char *p_name;

    /** 8-bit storage for task frequency.
     */
    uint8_t timing;

    /** 8-bit storage for task priority.
     */
    uint8_t priority;

    /** 8-bit storage for average task time.
     */
    uint8_t avg_time;

    /** 8-bit storage for average task frequency.
     */
    uint8_t avg_freq;

    /** 32-bit storage for number of task runs.
     */
    uint32_t run_number;

    /** 32-bit storage for number of transitions between states.
     */
    uint16_t transition_number;

    /** 8-bit storage for the total number of existing tasks in
     *  memory.
     */
    static uint8_t task_count;

    /** @brief Ready status.
     *  @details This method returns a boolean indicating if the
```

```
         *  task is ready to run based on the current program time.
         */
        inline bool ready (void);

        /** @brief Begin task.
         *  @details This method stores the task start time. It also
         *  stores the run time of the task, average task timing, and
         *  average task run frequency. This method should only be used
         *  if debugging information is desired.
         */
        inline void begin (void);

        /** @brief Set next task time.
         *  @details This method should be the first thing called in the
         *  task to set the next time at which to run the task.
         */
        inline void set_next_time (void);

        /** @brief End task.
         *  @details This method stores the task end time and sets
         *  the next run time for the task. It also stores the run time
         *  of the task, average task timing, and average task run
         *  frequency. Also increments the task run number by one. This
         *  method should only be implemented if debugging information
         *  is desired.
         */
        inline void end (void);

        /** Virtual method to be implemented in a derived class.
         */
        virtual void initialize (void);

        /** Virtual method to be implemented in a derived class.
         */
        virtual void run (void);
};


/* Co-scheduler, share and queue files included here so the compiler
   knows what members exist in these classes. */
#include "coscheduler.h"
#include "share_base.h"
#include "queue_base.h"


//////////////////////////////////////////////////////////////////////////
/** @brief Ready status.
 *  @details This method returns a boolean indicating if the task is ready
 *  to run based on the current program time.
 */
inline bool task_base :: ready (void)
{
    return clock.millis_32 () >= next_time;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Begin task.
 *  @details This method stores the task start time. It also stores the
 *  run time of the task, average task timing, and average task run
 *  frequency. This method should only be used if debugging information is
 *  desired.
 */
inline void task_base :: begin (void)
{
    // Store task time and average run frequency for debugging.
    task_time = clock.millis_8 ();

    // Increment task run number by one.
    run_number++;

    // Calculate average task run frequency.
    task_time >= next_time_8bit ? avg_freq += task_time - next_time_8bit + timing
    : avg_freq = next_time_8bit - (255 - task_time) + timing;
    avg_freq /= 2;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Set next task time.
 *  @details This method should be the first thing called in the task to
 *  set the next time at which to run the task.
 */
inline void task_base :: set_next_time (void)
{
    // Add desired time interval to next time.
    next_time += timing;
    next_time_8bit += timing;
}


//////////////////////////////////////////////////////////////////////////
```

```
/** @brief End task.
 *  @details This method stores the task end time and stores the run time
 *  of the task, average task timing, and average task run frequency.
 *  Also increments the task run number by one. This method should only be
 *  implemented if debugging information is desired.
 */
inline void task_base :: end (void)
{
    // Store task end time for debugging.
    clock.millis_8 () >= task_time ? task_time = clock.millis_8 () - task_time
    : task_time = task_time - (255 - clock.millis_8 ());

    // Calculate average task time.
    avg_time += task_time;
    avg_time /= 2;
}


//////////////////////////////////////////////////////////////////////////
/** @brief State transition method.
 *  @details This method causes the task to transition to a specified
 *  state on the next iteration and increments the number of transitions
 *  of the task.
 *  @param new_state Specified state to transition to at next iteration.
 */
inline void task_base :: transition_to (uint8_t new_state)
{
    state = new_state;
    transition_number++;
}


#endif /* TASK_BASE_H_ */
```

```
/**********************************************************************/
/** @file task.cpp
 *  @brief Main class file.
 *
 *   Contributors:
 *      - Zach Wilson 12/9/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/9/2018 - initial release.
 *
 *  Created: 12/9/2018 12:33:47 AM
 */
/**********************************************************************/


/* Included files. */
#include "task_base.h"

//////////////////////////////////////////////////////////////////////////
/** 8-bit storage for the total number of existing tasks in memory.
 */
uint8_t task_base :: task_count;

//////////////////////////////////////////////////////////////////////////
/** Share list. Share base pointers are added to this list when a share
 *  object is instantiated. Tasks can use this list to save pointers to
 *  the shares they need access to. Shares are added to this list in the
 *  order in which they are instantiated.
 */
list <share_base *, COSCHEDULER_SHARE_LIST> task_base :: share_list;

//////////////////////////////////////////////////////////////////////////
/** Queue list. Queue base pointers are added to this list when a queue
 *  object is instantiated. Tasks can use this list to save pointers to
 *  the queues they need access to. Queues are added to this list in the
 *  order in which they are instantiated.
 */
list <queue_base *, COSCHEDULER_QUEUE_LIST> task_base :: queue_list;

//////////////////////////////////////////////////////////////////////////
/** This struct contains global flags that can be read/altered in any task
 *  if there is a need for one task to cause another to transition based
 *  on events such as user commands.
 */
GLOBAL_FLAG_STRUCT_t task_base :: global_flags;


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor of Task class.
 *  @details This constructor creates an instance of the task class. It
 *  should only be called by the constructor of a derived class and
 *  nothing else. Priority and timing for each task is assigned here. The
 *  IDLE keyword should be used to assign the idle task.
 *  @param serial Pointer to a serial port object.
 *  @param task_name Pointer to the task name string.
 *  @param _priority Priority at which to schedule the task.
 *  @param _timing Timing at which to run the task.
 */
task_base :: task_base (serial_port *serial, const char *task_name, uint8_t _priority, uint8_t _timing)
{
    p_serial = serial;
    p_name = &name [0];
    priority = _priority;
    timing = _timing;
    avg_freq = _timing;
    next_time = 0;
    next_time_8bit = 0;
    storeString (task_name, &name [0]);

    // Add the task to the scheduler.
    if (priority == IDLE)
    {
        // Store as the idle task.
        coscheduler :: idle_task_control.p_task = this;
        coscheduler :: idle_task_control.p_next_block = NULL;
        coscheduler :: idle_task_control.status = FREE;
    }
    else
    {
        // Store as a regular task.
        coscheduler :: task_control [coscheduler :: task_number].p_task = this;
        coscheduler :: task_control [coscheduler :: task_number].status = FREE;
        coscheduler :: task_debug_list.add (this);
        coscheduler :: task_number++;
    }

    // Keep count of how many tasks have been instantiated.
    task_count++;
}


//////////////////////////////////////////////////////////////////////////
```

```
/** Virtual method to be implemented in a derived class.
 */
void task_base :: initialize (void) {}


//////////////////////////////////////////////////////////////////////////
/** Virtual method to be implemented in a derived class.
 */
void task_base :: run (void) {}


//////////////////////////////////////////////////////////////////////////
/** @brief Delay task.
 *  @details This method causes the task to sleep until its next run time.
 *  This should only be used in a preemptive environment.
 */
void task_base :: delay_task (void)
{
    for (;;)
    {
        if (clock.millis_32 () >= next_time) return;
    }
}


//////////////////////////////////////////////////////////////////////////
/** @brief Raise task debug flag.
 *  @details This function raises the task debug report flag to run a
 *  task debug report from the scheduler. It should be called from a user-
 *  interface style-task in which commands are received through a serial
 *  port.
 */
void task_base :: runDebugReport (void)
{
    coscheduler :: scheduler_debug.task_debug = true;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Raise task state flag.
 *  @details This function raises the task state report flag to run a
 *  task state report from the scheduler. It should be called from a user-
 *  interface style-task in which commands are received through a serial
 *  port.
 */
void task_base :: runStateReport (void)
{
    coscheduler :: scheduler_debug.task_state = true;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Raise share debug flag.
 *  @details This function raises the share debug report flag to run a
 *  share debug report from the scheduler. It should be called from a user-
 *  interface style-task in which commands are received through a serial
 *  port.
 */
void task_base :: runShareReport (void)
{
    coscheduler :: scheduler_debug.share_debug = true;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Raise queue debug flag.
 *  @details This function raises the queue debug report flag to run a
 *  queue debug report from the scheduler. It should be called from a user-
 *  interface style-task in which commands are received through a serial
 *  port.
 */
void task_base :: runQueueReport (void)
{
    coscheduler :: scheduler_debug.queue_debug = true;
}
```

```
/***********************************************************************/
/** @file share.h
 *  @brief Share class header file.
 *
 *  @details This class provides an abstract share data structure
 *  template for data transfer between tasks. It adds an abstraction
 *  layer by inheriting members from the share base class.
 *
 *
 *   Contributors:
 *      - Zach Wilson 1/28/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 1/28/2019 - initial release.
 *
 *  Created: 1/28/2019 5:18:07 PM
 */
/***********************************************************************/


#ifndef SHARE_H_
#define SHARE_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "semaphore.h"
#include "share_base.h"


////////////////////////////////////////////////////////////////////////
/** @brief Abstract share class.
 *  @details This class provides an abstract share data structure template
 *  for data transfer between tasks. It adds an abstraction layer by
 *  inheriting members from the share base class. Shares should be
 *  instantiated before tasks so the tasks can save pointers to each
 *  share.
 */
template <class Type>
class share : public share_base
{
    private:
        /** Storage for the share.
         */
        Type storage;

    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass a pointer
         *  to a serial port and a pointer to the share name to the
         *  base constructor.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to the share name.
         */
        share (stream_base *, const char *);

        /** @brief Initialize the share.
         *  @details This method indicates over the serial port that the
         *  object was instantiated successfully. It should only be called
         *  by the scheduler to confirm that the share pointer is saved in
         *  the scheduler.
         */
        void initialize (void);

        /** @brief Send to share.
         *  @details This method stores a value in the share and increments
         *  the send number variable. If the share class is in debug mode,
         *  an indication is sent to the serial port.
         *  @param value Value to be stored in the share.
         */
        inline void send (Type);

        /** @brief Retrieve share.
         *  @details This method retrieves the stored value in the share and
         *  increments the retrieve number variable. If the share class is in
         *  debug mode, an indication is sent to the serial port.
         *  @return Value that is stored in the share.
         */
        inline Type retrieve (void);
};


////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass a pointer to a serial
 *  port and a pointer to the share name to the base constructor.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to the share name.
 */
template <class Type>
share <Type> :: share (stream_base *serial, const char *name)
    : share_base (serial, name) {}
```

```
//////////////////////////////////////////////////////////////////////////
/** @brief Initialize the share.
 *  @details This method indicates over the serial port that the object
 *  was instantiated successfully. It should only be called by the
 *  scheduler to confirm that the share pointer is saved in the scheduler.
 */
template <class Type>
void share <Type> :: initialize (void)
{
    *p_serial << p_name << " share initialized." << endl;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Send to share.
 *  @details This method stores a value in the share and increments the
 *  send number variable. If the share class is in debug mode, an
 *  indication is sent to the serial port.
 *  @param value Value to be stored in the share.
 */
template <class Type>
inline void share <Type> :: send (Type value)
{
    storage = value;
    send_number++;

    // Print an indication to the serial port.
    #ifdef  SHARE_DEBUG
        *p_serial << "Sent " << value << " to the " << p_name << " share." << endl;
    #endif
}


//////////////////////////////////////////////////////////////////////////
/** @brief Retrieve share.
 *  @details This method retrieves the stored value in the share and
 *  increments the retrieve number variable. If the share class is in
 *  debug mode, an indication is sent to the serial port.
 *  @return Value that is stored in the share.
 */
template <class Type>
inline Type share <Type> :: retrieve (void)
{
    // Print an indication to the serial port.
    #ifdef  SHARE_DEBUG
        *p_serial << "Retrieved " << storage << " from the " << p_name << " share." << endl;
    #endif

    retrieve_number++;
    return storage;
}


#endif /* SHARE_H_ */
```

```
/**********************************************************************/
/** @file share_base.h
 *  @brief Base share class header file.
 *
 *  @details This class provides a base class for a generic share class.
 *  This class will make it possible for the scheduler class to save
 *  pointers to the share even though each share could store data of a
 *  different type.
 *
 *
 *   Contributors:
 *      - Zach Wilson 1/28/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 1/28/2019 - initial release.
 *
 *  Created: 1/28/2019 5:01:44 PM
 */
/**********************************************************************/


#ifndef SHARE_BASE_H_
#define SHARE_BASE_H_

/* Included files. */
#include <avr/io.h>
#include "serial_port.h"

//////////////////////////////////////////////////////////////////////////
/** Forward declaration of coscheduler class so the share base constructor
 *  can add shares to the scheduler list.
 */
class coscheduler;

//////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the desired task name.
 */
extern void storeString (const char *, char *);


//////////////////////////////////////////////////////////////////////////
/** @brief Base share class.
 *  @details This class provides a base class for a generic share class.
 *  This class will make it possible for the scheduler class to save
 *  pointers to the share even though each share could store data of a
 *  different type.
 */
class share_base
{
    protected:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** C-style string for the share name.
         */
        char name [11];

    public:
        /** Pointer to a string for storing the share name.
         */
        const char *p_name;

        /** Count storing how many share objects have been instantiated.
         */
        static uint8_t share_count;

        /** Number of sends to the share.
         */
        uint32_t send_number;

        /** Number of retrieves from the share.
         */
        uint32_t retrieve_number;

        /** @brief Default constructor.
         *  @details This constructor stores the share name and a pointer
         *  to a serial port for debugging.
         *  @param Pointer to a serial port object.
         *  @param Pointer to the share name.
         */
        share_base (stream_base *, const char *);

        /** Virtual method to be implemented in a derived class.
         */
        virtual void initialize (void);
};


/* Co-scheduler file included here so the compiler
   knows what members exist in the coscheduler class. */
```

```
#include "coscheduler.h"

#endif /* SHARE_BASE_H_ */
```

```
/***********************************************************************/
/** @file share_base.h
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *      - Zach Wilson 1/28/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 1/28/2019 - initial release.
 *
 *  Created: 1/28/2019 5:01:44 PM
 */
/***********************************************************************/


/* Included files. */
#include "share_base.h"

//////////////////////////////////////////////////////////////////////////
/** Count storing how many share objects have been instantiated.
 */
uint8_t share_base :: share_count;


//////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor stores the share name and a pointer to a
 *  serial port for debugging. It also places the share pointer in the
 *  share list belonging to the scheduler.
 *  @param Pointer to a serial port object.
 *  @param Pointer to the share name.
 */
share_base :: share_base (stream_base *serial, const char *share_name)
{
    p_serial = serial;
    p_name = &name [0];
    share_count++;
    storeString (share_name, &name [0]);

    // Add the object's pointer to all task and scheduler lists.
    coscheduler :: share_debug_list.add (this);
    task_base :: share_list.add (this);
}


//////////////////////////////////////////////////////////////////////////
/** Virtual method to be implemented in a derived class.
 */
void share_base :: initialize (void) {}
```

```
/************************************************************************/
/** @file queue.h
 *  @brief Queue class header file.
 *
 *  @details This class provides an abstract queue data structure
 *  template for data transfer between tasks. It adds an abstraction
 *  layer by inheriting members from the circular buffer class and the
 *  queue base class.
 *
 *
 *   Contributors:
 *      - Zach Wilson 1/27/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 1/27/2019 - initial release.
 *
 *  Created: 1/27/2019 9:32:22 PM
 */
/************************************************************************/


#ifndef QUEUE_H_
#define QUEUE_H_

/* Included files. */
#include <avr/io.h>
#include "defines.h"
#include "semaphore.h"
#include "circular_buffer.h"
#include "queue_base.h"


////////////////////////////////////////////////////////////////////////////
/** @brief Abstract queue class.
 *  @details This class provides an abstract queue data structure template
 *  for data transfer between tasks. It adds an abstraction layer by
 *  inheriting members from the circular buffer class and the queue base
 *  class. Queues must be instantiated before tasks so the tasks can save
 *  pointers to the queues.
 */
template <class Type, uint16_t q_size>
class queue : public queue_base, public circular_buffer <Type, q_size>
{
    public:
        /** @brief Default Constructor.
         *  @details This constructor does nothing but pass a pointer to
         *  a serial port, a pointer to the queue name, and the desired
         *  queue size to the base constructors.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to the queue name.
         */
        queue (stream_base *, const char *);

        /** @brief Initialize the queue.
         *  @details This method indicates over the serial port that the
         *  object was instantiated successfully. It should only be called
         *  by the scheduler to confirm that the queue pointer is saved in
         *  the scheduler.
         */
        void initialize (void);

        /** @brief Queue capacity.
         *  @details This method reports the current number of values stored
         *  in the queue.
         *  @return Number of values stored in the queue.
         */
        inline uint16_t capacity (void);

        /** @brief Push to queue.
         *  @details This method pushes a value onto the queue and increments
         *  the push number variable. If the queue class is in debug mode, an
         *  indication is sent to the serial port.
         *  @param value Value to be pushed to the queue.
         *  @return Boolean indicating if the push was successful (true == success).
         */
        bool push (Type);

        /** @brief Pull from queue.
         *  @details This method pulls a value from the queue and increments
         *  the pull number variable. If the queue class is in debug mode, an
         *  indication is sent to the serial port.
         *  @param pointer Pointer to the memory location to place the pulled value.
         *  @return Boolean indicating if the pull was successful (true == success).
         */
        bool pull (Type *);

        /** @brief Pull from queue.
         *  @details This method pulls a value from the queue and increments
         *  the pull number variable. If the queue class is in debug mode, an
         *  indication is sent to the serial port.
         *  @param value Reference to the variable to place the pulled value.
         *  @return Boolean indicating if the pull was successful (true == success).
```

```
         */
        bool pull (Type &);

        /** @brief Pull from queue.
         *  @details This method pulls a value from the queue and increments
         *  the pull number variable. If the queue class is in debug mode, an
         *  indication is sent to the serial port.
         *  @return Value pulled from the queue.
         */
        Type pull (void);
};


//////////////////////////////////////////////////////////////////////////////
/** @brief Default Constructor.
 *  @details This constructor does nothing but instantiate the object and
 *  pass a pointer to a serial port, a pointer to the queue name, and the
 *  desired queue size to the base constructors.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to the queue name.
 */
template <class Type, uint16_t q_size>
queue <Type, q_size> :: queue (stream_base *serial, const char *name)
    : queue_base (serial, name), circular_buffer <Type, q_size> () {}


//////////////////////////////////////////////////////////////////////////////
/** @brief Initialize the queue.
 *  @details This method indicates over the serial port that the object
 *  was instantiated successfully. It should only be called by the
 *  scheduler to confirm that the queue pointer is saved in the scheduler.
 */
template <class Type, uint16_t q_size>
void queue <Type, q_size> :: initialize (void)
{
    *p_serial << p_name << " queue initialized." << endl;
}


//////////////////////////////////////////////////////////////////////////////
/** @brief Queue capacity.
 *  @details This method reports the current number of values stored in
 *  the queue.
 *  @return Number of values stored in the queue.
 */
template <class Type, uint16_t q_size>
inline uint16_t queue <Type, q_size> :: capacity (void)
{
    return this->buf_capacity ();
}


//////////////////////////////////////////////////////////////////////////////
/** @brief Push to queue.
 *  @details This method pushes a value onto the queue and increments the
 *  push number variable. If the queue is in debug mode, an indication is
 *  sent to the serial port.
 *  @param value Value to be pushed to the queue.
 */
template <class Type, uint16_t q_size>
bool queue <Type, q_size> :: push (Type value)
{
    // Do not push the value if the queue is full.
    if (this->full ())
    {
        #ifdef  QUEUE_DEBUG
            *p_serial << p_name << " queue overloaded." << endl;
        #endif
        return false;
    }

    // Push the value onto the circular buffer and increment the push number
    // variable.
    this->push_data (value);
    push_number++;

    // Print the pushed value to the serial port.
    #ifdef  QUEUE_DEBUG
        *p_serial << "Pushed " << value << " to " << p_name << " queue." << endl;
    #endif

    // If we reach this code then the push was successful.
    return true;
}


//////////////////////////////////////////////////////////////////////////////
/** @brief Pull from queue.
 *  @details This method pulls a value from the queue and increments the
 *  pull number variable. If the queue is in debug mode, an indication is
 *  sent to the serial port.
 *  @param pointer Pointer to the memory location to place the pulled value.
```

```
 */
template <class Type, uint16_t q_size>
bool queue <Type, q_size> :: pull (Type *pointer)
{
    // Do not pull a value if the queue is empty.
    if (this->empty ())
    {
        #ifdef  QUEUE_DEBUG
            *p_serial << p_name << " queue empty." << endl;
        #endif
        return false;
    }

    // Pull a value from the circular buffer to the pointer location and
    // increment the pull number variable.
    *pointer = this->pull_data ();
    pull_number++;

    // Print the pulled value to the serial port.
    #ifdef  QUEUE_DEBUG
        *p_serial << "Pulled " << *pointer << " from " << p_name << " queue." << endl;
    #endif

    // If we reach this code then the pull was successful.
    return true;
}


//////////////////////////////////////////////////////////////////////////
/** @brief Pull from queue.
 *  @details This method pulls a value from the queue and increments the
 *  pull number variable. If the queue is in debug mode, an indication is
 *  sent to the serial port.
 *  @param value Reference to the variable to place the pulled value.
 */
template <class Type, uint16_t q_size>
bool queue <Type, q_size> :: pull (Type &value)
{
    // Do not pull a value if the queue is empty.
    if (this->empty ())
    {
        #ifdef  QUEUE_DEBUG
            *p_serial << p_name << " queue empty." << endl;
        #endif
        return false;
    }

    // Pull a value from the circular buffer and increment the pull number
    // variable.
    value = this->pull_data ();
    pull_number++;

    // Print the pulled value to the serial port.
    #ifdef  QUEUE_DEBUG
        *p_serial << "Pulled " << value << " from " << p_name << " queue." << endl;
    #endif

    // If we reach this code then the pull was successful.
    return true;
}


/** @brief Pull from queue.
 *  @details This method pulls a value from the queue and increments
 *  the pull number variable. If the queue class is in debug mode, an
 *  indication is sent to the serial port.
 *  @return Value pulled from the queue.
 */
template <class Type, uint16_t q_size>
Type queue <Type, q_size> :: pull (void)
{
    // Do not pull a value if the queue is empty.
    if (this->empty ())
    {
        #ifdef  QUEUE_DEBUG
            *p_serial << p_name << " queue empty." << endl;
        #endif
        return NULL;
    }

    // Pull a value from the circular buffer and increment the pull number
    // variable.
    Type value = this->pull_data ();
    pull_number++;

    // Print the pulled value to the serial port.
    #ifdef  QUEUE_DEBUG
        *p_serial << "Pulled " << value << " from " << p_name << " queue." << endl;
    #endif

    // Return the pulled value.
    return value;
```

```
}

#endif /* QUEUE_H_ */
```

```
/***********************************************************************/
/** @file queue_base.h
 *  @brief Base queue class header file.
 *
 *  @details This class provides a base class for a generic queue class.
 *  This class will make it possible for the scheduler class to save
 *  pointers to the queues even though each queue could store data of a
 *  different type.
 *
 *
 *   Contributors:
 *       - Zach Wilson 1/27/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *       - 1/27/2019 - initial release.
 *
 *  Created: 1/27/2019 4:29:42 PM
 */
/***********************************************************************/


#ifndef QUEUE_BASE_H_
#define QUEUE_BASE_H_

/* Included files. */
#include <avr/io.h>
#include "serial_port.h"

//////////////////////////////////////////////////////////////////////////
/** Forward declaration of coscheduler class so the queue base constructor
 *  can add queues to the scheduler lists.
 */
class coscheduler;

//////////////////////////////////////////////////////////////////////////
/** Declare the external store string method defined in string.h for
 *  storing the desired task name.
 */
extern void storeString (const char *, char *);


//////////////////////////////////////////////////////////////////////////
/** @brief Base queue class.
 *  @details This base class allows the scheduler to store pointers to
 *  multiple types of queues and to call methods for printing debugging
 *  information during debug reports.
 */
class queue_base
{
    protected:
        /** Pointer to a serial port object.
         */
        stream_base *p_serial;

        /** C-style string for queue name.
         */
        char name [11];

    public:
        /** Pointer to a string for storing the queue name.
         */
        const char *p_name;

        /** Count storing how many queue objects have been instantiated.
         */
        static uint8_t queue_count;

        /** Number of pushes to the queue.
         */
        uint32_t push_number;

        /** Number of pulls from the queue.
         */
        uint32_t pull_number;

        /** @brief Default constructor.
         *  @details This constructor stores the queue name and a pointer
         *  to a serial port for debugging.
         *  @param serial Pointer to a serial port object.
         *  @param queue_name Pointer to the queue name.
         */
        queue_base (stream_base *, const char *);

        /** Virtual method to be implemented in a derived class.
         */
        virtual void initialize (void);

        /** Virtual method to be implemented in a derived class.
         */
        virtual uint16_t capacity (void);
};
```

```
/* Co-scheduler file included here so the compiler
   knows what members exist in the coscheduler class. */
#include "coscheduler.h"


#endif /* QUEUE_BASE_H_ */
```

```cpp
/***********************************************************************/
/** @file queue_base.cpp
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *      - Zach Wilson 1/27/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 1/27/2019 - initial release.
 *
 *  Created: 1/27/2019 4:49:10 PM
 */
/***********************************************************************/


/* Included files. */
#include "queue_base.h"

/////////////////////////////////////////////////////////////////////////
/** Count storing how many queue objects have been instantiated.
 */
uint8_t queue_base :: queue_count;


/////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor stores the queue name and a pointer to a
 *  serial port for debugging and initializes all variables.
 *  @param serial Pointer to a serial port object.
 *  @param queue_name Pointer to the queue name.
 */
queue_base :: queue_base (stream_base *serial, const char *queue_name)
{
    p_serial = serial;
    p_name = &name [0];
    push_number = 0;
    pull_number = 0;
    queue_count++;
    storeString (queue_name, &name [0]);

    // Add the object's pointer to all task and scheduler lists.
    coscheduler :: queue_debug_list.add (this);
    task_base :: queue_list.add (this);
}


/////////////////////////////////////////////////////////////////////////
/** Virtual method to be implemented in a derived class.
 */
void queue_base :: initialize (void) {}


/////////////////////////////////////////////////////////////////////////
/** Virtual method to be implemented in a derived class.
 */
uint16_t queue_base :: capacity (void)
{
    return 0;
}
```

I. DYNAMIC DATA ACQUISITION SYSTEM DERIVED TASK C++ FILES

```
/***********************************************************************/
/** @file task_AD7708.h
 *  @brief Class header file.
 *
 *  @details This class defines a task that requests and retrieves data
 *  from the AD7708 analog-to-digital converter and stores the data in a
 *  queue.
 *
 *   Contributors:
 *      - Zach Wilson 3/16/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/16/2019 - initial release.
 *
 *  Created: 3/16/2019 10:33:53 AM
 */
/***********************************************************************/


#ifndef TASK_AD7708_H_
#define TASK_AD7708_H_

/* Included files. */
#include "serial_port.h"
#include "task_base.h"
#include "share.h"
#include "queue.h"
#include "AD7708.h"

////////////////////////////////////////////////////////////////////////
/** Active channel on the AD7708 device based on the values layed out in
 *  the datasheet.
 */
#define  ACTIVE_CHANNEL  0b10110000


////////////////////////////////////////////////////////////////////////
/** @brief AD7708 control task.
 *  @details This class defines a task that requests and retrieves data
 *  from the AD7708 analog-to-digital converter and stores the data in a
 *  queue.
 */
class task_AD7708 : public task_base
{
    private:
        /** Enumeration defining the task states.
         */
        typedef enum  TASK_AD7708_STATES
        {
            /** State 0. Make sure the AD7708 is initialized and
             *  disabled and transition to state 1.
             */
            INIT = 0,

            /** State 1. Wait until either the static or dynamic flag is
             *  raised and transition to state 2 or state 3.
             */
            HUB,

            /** State 2. Request a reading from the device and then transition
             *  to state 3. If the static flag is lowered from the user interface
             *  task, transition back to state 1.
             */
            READ_STATIC,

            /** State 3. Increment the count variable until it is time to
             *  transition back to state 2 to request another reading.
             */
            WAIT,

            /** State 4. Retrieve the data acquired in the ISRs and send it to
             *  the dynamic queue. Stop and transition to state 1 when the queue
             *  is full.
             */
            READING_DYNAMIC,

            /** Dead state if something goes wrong during task execution.
             */
            DEAD,
        } TASK_AD7708_STATES_t;

        /** Count for controlling when the task reads static data.
         */
        uint16_t count;

        /** Pointer to a share for the data acquisition frequency.
```

```
         */
        share <uint16_t> *acq_freq_share;

        /** Pointer to the static strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 10> *p_queue_static;

        /** Pointer to the dynamic strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 1000> *p_queue_dynamic;

    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass the pointer to
         *  the serial port to the base task constructor and saves share and
         *  queue pointers by reading them from the share and queue lists and
         *  statically casting them into the desired type.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to a string of the task name.
         *  @param priority Priority at which to schedule the task.
         *  @param timing Timing at which to run the task.
         */
        task_AD7708 (serial_port *, const char *, uint8_t, uint16_t);

        /** @brief Initialize task.
         *  @details This method initializes the task by initializing all
         *  task variables and starting the task in state 0.
         */
        void initialize (void);

        /** @brief Run task.
         *  @details This method runs one iteration of the finite-state-machine
         *   of the task.
         *       - state 0: INIT - Make sure the AD7708 is initialized and
         *           disabled and transition to state 1.
         *       - state 1: HUB - Wait until either the static or dynamic flag is
         *           raised and transition to state 2 or state 3.
         *       - state 2: READ_STATIC - Request a reading from the device and then
         *           transition to state 3. If the static flag is lowered from the user
         *           interface task, transition back to state 1.
         *       - state 3: WAIT - Increment the count variable until it is time to
         *           transition back to state 2 to request another reading.
         *       - state 4: READING_DYNAMIC -Retrieve the data acquired in the ISRs and
         *           send it to the dynamic queue. Stop and transition to state 1 when
         *           the queue is full.
         */
        void run (void);
};


#endif /* TASK_AD7708_H_ */
```

```
/*********************************************************************/
/** @file task_AD7708.cpp
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *      - Zach Wilson 3/16/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/16/2019 - initial release.
 *
 *  Created: 3/16/2019 10:33:53 AM
 */
/**********************************************************************/


/* Included files. */
#include "task_AD7708.h"


//////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass the pointer to the
 *  serial port to the base task constructor and saves share and queue
 *  pointers by reading them from the share and queue lists and statically
 *  casting them into the desired type.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to a string of the task name.
 *  @param priority Priority at which to schedule the task.
 *  @param timing Timing at which to run the task.
 */
task_AD7708 :: task_AD7708 (serial_port *serial, const char *name, uint8_t priority, uint16_t timing)
                           : task_base (serial, name, priority, timing)
{
    // Save and statically cast share and queue pointers from the
    // scheduler lists.
    acq_freq_share = static_cast <share <uint16_t> *> (share_list.read (1));
    p_queue_static = static_cast <queue <uint16_t, 10> *> (queue_list.read (0));
    p_queue_dynamic = static_cast <queue <uint16_t, 1000> *> (queue_list.read (1));
}


//////////////////////////////////////////////////////////////////////
/** @brief Initialize task.
 *  @details This method initializes the task by initializing all task
 *  variables and starting the task in state 0.
 */
void task_AD7708 :: initialize (void)
{
    // Initialize the task variables.
    count = 0;

    // Indicate through serial port that the task initialized
    // successfully.
    #ifdef  TASK_DEBUG
        *p_serial << p_name << " task initialized." << endl;
    #endif

    // Set the state variable to state 0.
    transition_to (INIT);
}


//////////////////////////////////////////////////////////////////////
/** @brief Run task.
 *  @details This method runs one iteration of the finite-state-machine
 *   of the task.
 *      - state 0: INIT - Make sure the AD7708 is initialized and
 *          disabled and transition to state 1.
 *      - state 1: HUB - Wait until either the static or dynamic flag is
 *          raised and transition to state 2 or state 3.
 *      - state 2: READ_STATIC - Request a reading from the device and
 *          then transition to state 3. If the static flag is lowered from
 *          the user interface task, transition back to state 1.
 *      - state 3: WAIT - Increment the count variable until it is time to
 *          transition back to state 2 to request another reading.
 *      - state 4: READING_DYNAMIC -Retrieve the data acquired in the ISRs
 *          and send it to the dynamic queue. Stop and transition to state
 *          1 when the queue is full.
 */
void task_AD7708 :: run (void)
{
    // Set the next time so the task will run again.
    set_next_time ();

    // Finite-state-machine. This switch statement determines which state
    // to run based on the current value of the state variable.
    switch (state)
    {
        /* State 0. */
        case INIT:
            // Initialize the device and call the enable method so the device
```

```
        // locks on to the 32kHz crystal. Then disable the device.
        adc_amp.initialize (p_serial);
        adc_amp.enable () ? *p_serial << "AD7708 locked on to external oscillator and enabled." << endl
        : *p_serial << "AD7708 could not lock on to external oscillator." << endl;
        adc_amp.disable ();

        // Transition to state 1.
        transition_to (HUB);
        break;
    /* State 1. */
    case HUB:
        // Check if the user interface task has raised the read static
        // flag.
        if (global_flags.read_static_AD7708)
        {
            // Enable and calibrate the proper channel of the device and
            // begin data acquisition and transition to state 2. Also make
            // sure the queue is empty and turn on the LED.
            p_queue_static->flush_buffer ();
            adc_amp.enable ();
            adc_amp.select_channel (ACTIVE_CHANNEL);
            adc_amp.calibrate ();

            // Delay for a few hundred milliseconds to give the device time
            // to warm up.
            clock.delay_ms (300);
            *p_serial << "Acquiring data." << endl << endl;
            PORTB.OUT |= AD7708_LED;
            transition_to (READ_STATIC);
            break;
        }
        // Check if the user interface task has raised the read dynamic
        // flag.
        else if (global_flags.read_dynamic_AD7708)
        {
            // Enable and calibrate the proper channel of the device and
            // begin data acquisition and transition to state 4. Also make
            // sure the queue is empty and turn on the LED.
            p_queue_dynamic->flush_buffer ();
            adc_amp.enable ();
            adc_amp.select_channel (ACTIVE_CHANNEL);
            adc_amp.calibrate ();

            // Delay for a few hundred milliseconds to give the device time
            // to warm up.
            clock.delay_ms (300);
            adc_amp.acquire_data (1);
            *p_serial << "Acquiring data." << endl;
            PORTB.OUT |= AD7708_LED;
            transition_to (READING_DYNAMIC);
            break;
        }
        break;
    /* State 2. */
    case READ_STATIC:
        // Check to see if the static flag is lowered. If it is, transition
        // back to state 1.
        if (!global_flags.read_static_AD7708)
        {
            adc_amp.disable ();
            PORTB.OUT &= ~AD7708_LED;
            transition_to (HUB);
            break;
        }

        // Request a reading from the device and transition to state 3.
        adc_amp.request ();
        transition_to (WAIT);
        break;
    /* State 3. */
    case WAIT:
        // If data is available from the device, push it to the static queue.
        while (adc_amp.data_available ()) p_queue_static->push (adc_amp.retrieve ());

        // Poll the static flag to see if we are still reading data. If the
        // flag is low, disable the device and transition to state 1.
        if (!global_flags.read_static_AD7708)
        {
            adc_amp.disable ();
            PORTB.OUT &= ~AD7708_LED;
            transition_to (HUB);
            break;
        }

        // Increment the count variable until it is time to transition back to
        // state 2 to request another reading.
        count++;
        if (count >= (acq_freq_share->retrieve () / timing))
        {
            count = 0;
            transition_to (READ_STATIC);
            break;
```

```
            }
            break;
        /* State 4. */
        case READING_DYNAMIC:
            // Check to see if the read flag has been lowered or if the queue
            // is full.
            if (p_queue_dynamic->full ())
            {
                // Disable the device and transition to state 1.
                adc_amp.stop_data ();
                adc_amp.disable ();
                global_flags.read_dynamic_AD7708 = false;
                *p_serial << "Data acquired." << endl;
                PORTB.OUT &= ~AD7708_LED;
                transition_to (HUB);
                break;
            }
            // Push all readings from the device to the queue.
            while (adc_amp.data_available ()) p_queue_dynamic->push (adc_amp.retrieve ());
            break;
        /* Dead state. */
        case DEAD:
            break;
        /* Undefined State. */
        default:
            // Default state just in case the task enters an undefined state.
            #ifdef  TASK_DEBUG
                *p_serial << p_name << " task entered undefined state!" << endl;
            #endif

            // Send task to a dead state where no code is executed.
            transition_to (DEAD);
            break;
    };
}
```

```
/**********************************************************************/
/** @file task_ADC.h
 *  @brief Class header file.
 *
 *  @details This class defines a task that requests and retrieves data
 *  from the ADC interface and pushes it to a data queue.
 *
 *   Contributors:
 *      - Zach Wilson 3/15/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/15/2019 - initial release.
 *
 *  Created: 3/15/2019 9:55:40 PM
 */
/**********************************************************************/


#ifndef TASK_ADC_H_
#define TASK_ADC_H_

/* Included files. */
#include "serial_port.h"
#include "task_base.h"
#include "share.h"
#include "queue.h"
#include "ADC.h"


//////////////////////////////////////////////////////////////////////
/** @brief ADC Interface task.
 *  @details This class defines a task that requests and retrieves data
 *  from the ADC interface and pushes it to a data queue. The data is
 *  acquired via an ISR at a precise frequency using a timer.
 */
class task_ADC : public task_base
{
    private:
        /** Enumeration defining the task states.
         */
        typedef enum  TASK_ADC_STATES
        {
            /** State 0. Make sure the ADC interface is initialized and
             *  disabled and transition to state 1.
             */
            INIT = 0,

            /** State 1. Wait until either the static or dynamic flag is
             *  raised and transition to state 2 or state 3.
             */
            HUB,

            /** State 2. Request a reading from the interface and then
             *  transition to state 3. If the static flag is lowered from
             *  the user interface task, transition back to state 1.
             */
            READ_STATIC,

            /** State 3. Increment the count variable until it is time to
             *  transition back to state 2 to request another reading.
             */
            WAIT,

            /** State 4. Retrieve the data acquired in the ISRs and push it
             *  to the dynamic queue. Stop and transition to state 1 when the
             *  queue is full.
             */
            READING_DYNAMIC,

            /** Dead state if something goes wrong during task execution.
             */
            DEAD,
        } TASK_ADC_STATES_t;

        /** Count for controlling when the task reads static data.
         */
        uint16_t count;

        /** Pointer to a share for the data acquisition frequency.
         */
        share <uint16_t> *acq_freq_share;

        /** Pointer to the static strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 10> *p_queue_static;

        /** Pointer to the dynamic strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 1000> *p_queue_dynamic;
```

```
    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass the pointer to
         *  the serial port to the base task constructor and saves share and
         *  queue pointers by reading them from the share and queue lists and
         *  statically casting them into the desired type.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to a string of the task name.
         *  @param priority Priority at which to schedule the task.
         *  @param timing Timing at which to run the task.
         */
        task_ADC (serial_port *, const char *, uint8_t, uint16_t);

        /** @brief Initialize task.
         *  @details This method initializes the task by initializing all
         *  task variables and starting the task in state 0.
         */
        void initialize (void);

        /** @brief Run task.
         *  @details This method runs one iteration of the finite-state-machine
         *  of the task.
         *       - state 0: INIT - Make sure the ADC interface is initialized and
         *           disabled and transition to state 1.
         *       - state 1: HUB - Wait until either the static or dynamic flag is
         *           raised and transition to state 2 or state 3.
         *       - state 2: READ_STATIC - Request a reading from the interface and
         *           then transition to state 3. If the static flag is lowered from
         *           the user interface task, transition back to state 1.
         *       - state 3: WAIT - Increment the count variable until it is time to
         *           transition back to state 2 to request another reading.
         *       - state 4: READING_DYNAMIC - Retrieve the data acquired in the ISRs
         *           and push it to the dynamic queue. Stop and transition to state
         *           1 when the queue is full.
         */
        void run (void);
};


#endif /* TASK_ADC_H_ */
```

```
/*********************************************************************/
/** @file task_ADC.cpp
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *      - Zach Wilson 3/15/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 3/15/2019 - initial release.
 *
 *  Created: 3/15/2019 10:06:13 PM
 */
/**********************************************************************/


/* Included files. */
#include "task_ADC.h"


//////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass the pointer to the
 *  serial port to the base task constructor and saves share and queue
 *  pointers by reading them from the share and queue lists and statically
 *  casting them into the desired type.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to a string of the task name.
 *  @param priority Priority at which to schedule the task.
 *  @param timing Timing at which to run the task.
 */
task_ADC :: task_ADC (serial_port *serial, const char *name, uint8_t priority, uint16_t timing)
                          : task_base (serial, name, priority, timing)
{
    // Save and statically cast share and queue pointers from the
    // scheduler lists.
    acq_freq_share = static_cast <share <uint16_t> *> (share_list.read (1));
    p_queue_static = static_cast <queue <uint16_t, 10> *> (queue_list.read (0));
    p_queue_dynamic = static_cast <queue <uint16_t, 1000> *> (queue_list.read (1));
}


//////////////////////////////////////////////////////////////////////
/** @brief Initialize task.
 *  @details This method initializes the task by initializing all task
 *  variables and starting the task in state 0.
 */
void task_ADC :: initialize (void)
{
    // Initialize the task variables.
    count = 0;

    // Indicate through serial port that the task initialized
    // successfully.
    #ifdef  TASK_DEBUG
        *p_serial << p_name << " task initialized." << endl;
    #endif

    // Set the state variable to state 0.
    transition_to (INIT);
}


//////////////////////////////////////////////////////////////////////
/** @brief Run task.
 *  @details This method runs one iteration of the finite-state-machine
 *  of the task.
 *       - state 0: INIT - Make sure the ADC interface is initialized and
 *           disabled and transition to state 1.
 *       - state 1: HUB - Wait until either the static or dynamic flag is
 *           raised and transition to state 2 or state 3.
 *       - state 2: READ_STATIC - Request a reading from the interface and
 *           then transition to state 3. If the static flag is lowered from
 *           the user interface task, transition back to state 1.
 *       - state 3: WAIT - Increment the count variable until it is time to
 *           transition back to state 2 to request another reading.
 *       - state 4: READING_DYNAMIC - Retrieve the data acquired in the ISRs
 *           and push it to the dynamic queue. Stop and transition to state
 *           1 when the queue is full.
 */
void task_ADC :: run (void)
{
    // Set the next time so the task will run again.
    set_next_time ();

    // Finite-state-machine. This switch statement determines which state
    // to run based on the current value of the state variable.
    switch (state)
    {
        /* State 0. */
        case INIT:
            // Initialize and tie all ADC channels to the proper pins on
```

```
        // PORTA and make sure it is disabled.
        ADC.initialize (p_serial);
        ADC.channel0.tie_pin (PIN3_bm);
        ADC.channel1.tie_pin (PIN5_bm);
        ADC.channel2.tie_pin (PIN6_bm);
        ADC.channel3.tie_pin (PIN7_bm);
        ADC.disable ();

        // Initialize the pin that controls the wheatstone bridges low
        // so they don't consume any power while the ADC is disabled.
        PORTA.OUT &= ~EXCITE_PIN;
        PORTA.DIR |= EXCITE_PIN;

        // Transition to state 1.
        transition_to (HUB);
        break;
    /* State 1. */
    case HUB:
        // Check if the user interface task has raised the read static
        // flag.
        if (global_flags.read_static_ADC)
        {
            // Enable and calibrate the proper channel of the device and
            // begin data acquisition and transition to state 2. Also make
            // sure the queue is empty and turn on the LED.
            p_queue_static->flush_buffer ();
            ADC.enable ();
            *p_serial << "Acquiring data." << endl << endl;
            PORTB.OUT |= ADC_LED;
            transition_to (READ_STATIC);
            break;
        }
        // Check if the user interface task has raised the read dynamic data
        // flag.
        else if (global_flags.read_dynamic_ADC)
        {
            // Enable the ADC interface, assert the pin that controls the
            // wheatstone bridges, and begin data acquisition at 1kHz.
            // Transition to state 2.
            *p_serial << "Acquiring data." << endl;
            PORTA.OUT |= EXCITE_PIN;
            p_queue_dynamic->flush_buffer ();
            ADC.enable ();
            ADC.acquire_data (ADC_CHANNEL_0, 1);
            PORTB.OUT |= ADC_LED;
            transition_to (READING_DYNAMIC);
            break;
        }
        break;
    /* State 2. */
    case READ_STATIC:
        // Check to see if the static flag is lowered. If it is, transition
        // back to state 1.
        if (!global_flags.read_static_ADC)
        {
            ADC.disable ();
            PORTB.OUT &= ~ADC_LED;
            transition_to (HUB);
            break;
        }

        // Request a reading from the interface and transition to state 3.
        ADC.channel0.request ();
        transition_to (WAIT);
        break;
    /* State 3. */
    case WAIT:
        // If data is available from the interface, push it to the static queue.
        while (ADC.channel0.check_buf ()) p_queue_static->push (ADC.channel0.retrieve ());

        // Poll the static flag to see if we are still reading data. If the
        // flag is low, disable the device and transition to state 1.
        if (!global_flags.read_static_ADC)
        {
            ADC.disable ();
            PORTB.OUT &= ~ADC_LED;
            transition_to (HUB);
            break;
        }

        // Increment the count variable until it is time to transition back to
        // state 2 to request another reading.
        count++;
        if (count >= (acq_freq_share->retrieve () / timing))
        {
            count = 0;
            transition_to (READ_STATIC);
            break;
        }
        break;
    /* State 4. */
    case READING_DYNAMIC:
```

```
            // Check to see if the read flag has been lowered or if the queue
            // is full.
            if (p_queue_dynamic->full ())
            {
                // Disable the interface and wheatstone bridges and transition
                // to state 1.
                ADC.stop_data ();
                PORTA.OUT &= ~EXCITE_PIN;
                ADC.disable ();
                *p_serial << "Data acquired." << endl;
                global_flags.read_dynamic_ADC = false;
                PORTB.OUT &= ~ADC_LED;
                transition_to (HUB);
                break;
            }
            // Push all readings from the ADC to the queue.
            while (ADC.channel0.check_buf ()) p_queue_dynamic->push (ADC.channel0.retrieve ());
            break;
        /* Dead state. */
        case DEAD:
            break;
        /* Undefined State. */
        default:
            // Default state just in case the task enters an undefined state.
            #ifdef  TASK_DEBUG
                *p_serial << p_name << " task entered undefined state!" << endl;
            #endif

            // Send task to a dead state where no code is executed.
            transition_to (DEAD);
            break;
    };
}
```

```
/**********************************************************************/
/** @file task_microSD.h
 *  @brief Class header file.
 *
 *  @details This class defines a task that stores strain data read from
 *  the AD7708 analog-to-digital converter or the ADC interface on the
 *  Atxmega128a4u microcontroller.
 *
 *   Contributors:
 *      - Zach Wilson 2/24/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 2/24/2019 - initial release.
 *
 *  Created: 2/24/2019 1:51:03 PM
 */
/***********************************************************************/


#ifndef TASK_MICROSD_H_
#define TASK_MICROSD_H_

/* Included files. */
#include "serial_port.h"
#include "task_base.h"
#include "share.h"
#include "queue.h"
#include "microSD.h"

////////////////////////////////////////////////////////////////////////
/** Define for the filename the data will be saved in on the microSD card.
 */
#define  FILE_NAME  "test.txt"


////////////////////////////////////////////////////////////////////////
/** @brief MicroSD task.
 *  @details This class defines a task that stores strain data read from
 *  the HX711 amplifier or ADC conversion pins on the Atxmega128a4u
 *  microcontroller.
 */
class task_microSD : public task_base
{
    private:
        /** Enumeration defining the task states.
         */
        typedef enum  TASK_MICROSD_STATES
        {
            /** State 0. Initialize the driver and create a file. Close
             *  the file and transition to state 1. If no SD card is
             *  inserted, transition to state 5.
             */
            INIT = 0,

            /** State 1. Wait until the user inputs the operation they wish
             *  to perform. Transition to state 2 or 3 depending on which
             *  flag is raised.
             */
            HUB,

            /** State 2. Save the static data acquired by the AD7708. Once the
             *  static flag is lowered, transition back to state 1.
             */
            SAVE_STATIC,

            /** State 3. Wait until the dynamic queue is full and then
             *  transition to state 4 to begin saving the data in the queue.
             */
            WAIT,

            /** State 4. Save ten blocks of data for each task run until the
             *  queue is empty. Then transition back to state 1.
             */
            SAVE_DYNAMIC,

            /** State 5. Wait until an SD card is detected and then transition
             *  back to state 0 to initialize the card.
             */
            NO_SD,

            /** Dead state if something goes wrong during task execution.
             */
            DEAD,
        } TASK_MICROSD_STATES_t;

        /** Pointer to a microSD object for saving data.
         */
        microSD *p_microSD;

        /** Count for timestamping data.
         */
        uint16_t count;
```

```
        /** Pointer to a share for the data acquisition frequency.
         */
        share <uint16_t> *acq_freq_share;

        /** Pointer to the static strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 10> *p_queue_static;

        /** Pointer to the dynamic strain queue for sending data from the
         *  ADC and AD7708 tasks to the microSD task.
         */
        queue <uint16_t, 1000> *p_queue_dynamic;

    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass the pointer to
         *  the serial port to the base task constructor and saves share and
         *  queue pointers by reading them from the share and queue lists and
         *  statically casting them into the desired type.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to a string of the task name.
         *  @param priority Priority at which to schedule the task.
         *  @param timing Timing at which to run the task.
         *  @param _microSD Pointer to a microSD object for saving data.
         */
        task_microSD (serial_port *, const char *, uint8_t, uint8_t, microSD *);

        /** @brief Initialize task.
         *  @details This method initializes the task by initializing all
         *  task variables and starting the task in state 0.
         */
        void initialize (void);

        /** @brief Run task.
         *  @details This method runs one iteration of the finite-state-
         *  machine of the task.
         *       - state 0: INIT - Initialize the driver and create a file. Close
         *           the file and transition to state 1. If no SD card is
         *           inserted, transition to state 5.
         *       - state 1: HUB - Wait until the user inputs the operation they wish
         *           to perform. Transition to state 2 or 3 depending on which
         *           flag is raised.
         *       - state 2: SAVE_STATIC - Open and save the static data acquired by
         *           the AD7708. Once the flag is lowered, transition back to state 1.
         *       - state 3: WAIT - Wait until the dynamic queue is full and then
         *           transition to state 4 to begin saving the data in the queue.
         *       - state 4: SAVE_DYNAMIC - Save ten blocks of data for each task run
         *           until the queue is empty. Then transition back to state 1.
         *       - state 5: NO_SD - Wait until an SD card is detected and then transition
         *           back to state 0 to initialize the card.
         */
        void run (void);
};


#endif /* TASK_MICROSD_H_ */
```

```
/***********************************************************************/
/** @file task_microSD.cpp
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *       - Zach Wilson 2/24/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *       - 2/24/2019 - initial release.
 *
 *  Created: 2/24/2019 2:34:35 PM
 */
/************************************************************************/


/* Included files. */
#include "task_microSD.h"


////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass the pointer to the
 *  serial port to the base task constructor and saves share and queue
 *  pointers by reading them from the share and queue lists and statically
 *  casting them into the desired type.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to a string of the task name.
 *  @param priority Priority at which to schedule the task.
 *  @param timing Timing at which to run the task.
 *  @param _microSD Pointer to a microSD object for saving data.
 */
task_microSD :: task_microSD (serial_port *serial, const char *name, uint8_t priority, uint8_t timing, microSD *_microSD)
                           : task_base (serial, name, priority, timing)
{
     p_microSD = _microSD;

     // Save and statically cast share and queue pointers from the
     // scheduler lists.
     acq_freq_share = static_cast <share <uint16_t> *> (share_list.read (1));
     p_queue_static = static_cast <queue <uint16_t, 10> *> (queue_list.read (0));
     p_queue_dynamic = static_cast <queue <uint16_t, 1000> *> (queue_list.read (1));
}


////////////////////////////////////////////////////////////////////////
/** @brief Initialize task.
 *  @details This method initializes the task by initializing all task
 *  variables and starting the task in state 0.
 */
void task_microSD :: initialize (void)
{
     // Initialize the task variables.


     // Indicate through serial port that the task initialized
     // successfully.
     #ifdef  TASK_DEBUG
          *p_serial << p_name << " task initialized." << endl;
     #endif

     // Set the state variable to state 0.
     transition_to (INIT);
}


////////////////////////////////////////////////////////////////////////
/** @brief Run task.
 *  @details This method runs one iteration of the finite-state-
 *  machine of the task.
 *        - state 0: INIT - Initialize the driver and create a file. Close
 *            the file and transition to state 1. If no SD card is
 *            inserted, transition to state 5.
 *        - state 1: HUB - Wait until the user inputs the operation they
 *            wish to perform. Transition to state 2 or 3 depending on which
 *            flag is raised.
 *        - state 2: SAVE_STATIC - Open and save the static data acquired by
 *            the AD7708. Once the flag is lowered, transition back to state
 *            1.
 *        - state 3: WAIT - Wait until the dynamic queue is full and then
 *            transition to state 4 to begin saving the data in the queue.
 *        - state 4: SAVE_DYNAMIC - Save ten blocks of data for each task run
 *            until the queue is empty. Then transition back to state 1.
 *        - state 5: NO_SD - Wait until an SD card is detected and then
 *            transition back to state 0 to initialize the card.
 */
void task_microSD :: run (void)
{
     // Set the next time so the task will run again.
     set_next_time ();

     // Finite-state-machine. This switch statement determines which state
```

```
// to run based on the current value of the state variable.
switch (state)
{
    /* State 0. */
    case INIT:
        // If a card is not inserted, transition to state 5.
        if (!p_microSD->card_detect ())
        {
            *p_serial << "No SD card inserted." << endl;
            transition_to (NO_SD);
            break;
        }
        *p_serial << "SD card detected." << endl;

        // Initialize the SD card and filesystem and transition
        // to state 1.
        p_microSD->initialize ();
        p_microSD->mount ();
        if (p_microSD->create (FILE_NAME)) *p_serial << "Data file created." << endl;
        p_microSD->close ();
        transition_to (HUB);
        break;
    /* State 1. */
    case HUB:
        // Check to see if any flags are raised by the user interface
        // task and transition to the appropriate state and turn on
        // the LED.
        if (global_flags.read_static_AD7708 || global_flags.read_static_ADC)
        {
            count = 0;
            PORTB.OUT |= MICROSD_LED;
            p_microSD->open (FILE_NAME);
            *p_microSD << "Strain Reader/Logger V2." << endl << "Device: ";

            // Edit the header in the file to include the selected device.
            global_flags.daq_method ? *p_microSD << "ADC Interface" << endl
            : *p_microSD << "AD7708" << endl;

            *p_microSD << "Timestep (" << acq_freq_share->retrieve () << "ms),Voltage (bits)" << endl;
            p_microSD->close ();
            transition_to (SAVE_STATIC);
            break;
        }
        else if (global_flags.read_dynamic_AD7708 || global_flags.read_dynamic_ADC)
        {
            count = 0;
            transition_to (WAIT);
            break;
        }
        break;
    /* State 2. */
    case SAVE_STATIC:
        // If data exists in the static data queue, pull the data and save it to the file.
        if (!p_queue_static->empty ())
        {
            // Open the file and save every block of data in the queue.
            p_microSD->open (FILE_NAME);
            while (!p_queue_static->empty ())
            {
                *p_microSD << count++ << "," << p_queue_static->pull () << endl;
            }
            p_microSD->close ();
            break;
        }
        // Stop saving data and transition to state 1 if both flags are lowered.
        if (!global_flags.read_static_AD7708 && !global_flags.read_static_ADC)
        {
            PORTB.OUT &= ~MICROSD_LED;
            transition_to (HUB);
            break;
        }
        break;
    /* State 3. */
    case WAIT:
        // If the dynamic queue is full, transition to state 4 and open
        // the file to start saving the data.
        if (p_queue_dynamic->full () &&
                !global_flags.read_dynamic_AD7708 &&
                !global_flags.read_dynamic_ADC)
        {
            *p_serial << "Saving data." << endl;
            p_microSD->open (FILE_NAME);
            *p_microSD << "Strain Reader/Logger V2." << endl << "Device: ";

            // Edit the header in the file to include the selected device.
            global_flags.daq_method ? *p_microSD << "ADC Interface" << endl
            : *p_microSD << "AD7708" << endl;

            *p_microSD << "Timestep (" << acq_freq_share->retrieve () << "ms),Voltage (bits)" << endl;
            count = 0;
            global_flags.saving_dynamic_data = true;
            PORTB.OUT |= MICROSD_LED;
```

```
                    transition_to (SAVE_DYNAMIC);
                    break;
                }
                break;
            /* State 4. */
            case SAVE_DYNAMIC:
                // Check if data is available to save.
                if (!p_queue_dynamic->empty ())
                {
                    // Try to save 10 blocks of data from the queue. If all values are pulled,
                    // break the loop and transition to state 1.
                    for (uint8_t index = 0; index < 10; index++)
                    {
                        *p_microSD << count++ << "," << p_queue_dynamic->pull () << endl;
                        if (p_queue_dynamic->empty ())
                        {
                            p_microSD->close ();
                            *p_serial << "Data saved." << endl << endl;
                            global_flags.saving_dynamic_data = false;
                            PORTB.OUT &= ~MICROSD_LED;
                            transition_to (HUB);
                            break;
                        }
                    }
                    break;
                }
                else
                {
                    // If we reach this code, then there are no more values to pull.
                    // Close the file and transition to state 1.
                    p_microSD->close ();
                    *p_serial << "Data saved." << endl;
                    global_flags.saving_dynamic_data = false;
                    PORTB.OUT &= ~MICROSD_LED;
                    transition_to (HUB);
                    break;
                }
                break;
            /* State 5. */
            case NO_SD:
                // Check to see if an SD card has been inserted. If one has, transition
                // to state 0 to initialize the SD card.
                if (p_microSD->card_detect ()) transition_to (INIT);
                break;
            /* Dead state. */
            case DEAD:
                break;
            /* Undefined State. */
            default:
                // Default state just in case the task enters an undefined state.
                #ifdef  TASK_DEBUG
                    *p_serial << p_name << " task entered undefined state!" << endl;
                #endif

                // Send task to a dead state where no code is executed.
                transition_to (DEAD);
                break;
        };
}
```

```
/**********************************************************************/
/** @file task_led.h
 *  @brief Class header file.
 *
 *  @details This is a simple task class that just blinks an LED every
 *  second.
 *
 *   Contributors:
 *      - Zach Wilson 2/5/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 2/5/2019 - initial release.
 *
 *  Created: 2/5/2019 5:28:04 PM
 */
/**********************************************************************/


#ifndef TASK_LED_H_
#define TASK_LED_H_

/* Included files. */
#include "serial_port.h"
#include "task_base.h"
#include "share.h"
#include "queue.h"


//////////////////////////////////////////////////////////////////////////
/** @brief LED task.
 *  @details This task blinks an LED every second.
 */
class task_led : public task_base
{
    private:
        /** Enumeration defining the task states.
         */
        typedef enum  TASK_LED_STATES
        {
            /** State 0. This state configures the LED pin to an output
             *  and lowers the pin for initialization.
             */
            INIT = 0,

            /** State 1. This state increments a count variable until one
             *  frequency goes by and then raises the LED pin.
             */
            LED_OFF,

            /** State 2. This state increments a count variable until one
             *  frequency go by and then lowers the LED pin.
             */
            LED_ON,

            /** Dead state if something goes wrong during task execution.
             */
            DEAD,
        } TASK_LED_STATES_t;

        /** Pointer to a share for the LED blinking frequency.
         */
        share <uint16_t> *freq_share;

        /** Count variable to keep track of when to transition between the LED_ON and
         *  LED_OFF states.
         */
         uint16_t count;

    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass the pointer to
         *  the serial port to the base task constructor and saves share and
         *  queue pointers by reading them from the share and queue lists and
         *  statically casting them into the desired type.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to a string of the task name.
         *  @param priority Priority at which to schedule the task.
         *  @param timing Timing at which to run the task.
         */
        task_led (serial_port *, const char *, uint8_t, uint8_t);

        /** @brief Initialize task.
         *  @details This method initializes the task by initializing all
         *  task variables and starting the task in state 0.
         */
        void initialize (void);

        /** @brief Run task.
         *  @details This method runs one iteration of the finite-state-
         *  machine of the task.
         *       - state 0: INIT - This state configures the LED pin to an
         *           output and lowers the pin for initialization.
```

```
         *       - state 1: LED_OFF - This state increments a count variable
         *           until one frequency goes by and then raises the LED pin.
         *       - state 2: LED_ON - This state increments a count variable
         *           until one frequency go by and then lowers the LED pin.
         */
        void run (void);
};

#endif /* TASK_LED_H_ */
```

```
/**********************************************************************/
/** @file task_led.cpp
 *  @brief Main class file.
 *
 *
 *   Contributors:
 *       - Zach Wilson 1/8/2019 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *       - 1/8/2019 - initial release.
 *
 *  Created: 1/8/2019 5:32:08 PM
 */
/***********************************************************************/


/* Included files. */
#include "task_led.h"


///////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass the pointer to the
 *  serial port to the base task constructor and saves share and queue
 *  pointers by reading them from the share and queue lists and statically
 *  casting them into the desired type.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to a string of the task name.
 *  @param priority Priority at which to schedule the task.
 *  @param timing Timing at which to run the task.
 */
task_led :: task_led (serial_port *serial, const char *name, uint8_t priority, uint8_t timing)
                          : task_base (serial, name, priority, timing)
{
    // Save and statically cast share and queue pointers from the
    // scheduler lists.
    freq_share = static_cast <share <uint16_t> *> (share_list.read (0));
}


///////////////////////////////////////////////////////////////////////
/** @brief Initialize task.
 *  @details This method initializes the task by initializing all task
 *  variables and starting the task in state 0.
 */
void task_led :: initialize (void)
{
    // Initialize the task variables.
    count = 0;

    // Indicate through serial port that the task initialized
    // successfully.
    #ifdef  TASK_DEBUG
        *p_serial << p_name << " task initialized." << endl;
    #endif

    // Set the state variable to state 0.
    transition_to (INIT);
}


///////////////////////////////////////////////////////////////////////
/** @brief Run task.
 *  @details This method runs one iteration of the finite-state-
 *  machine of the task.
 *       - state 0: INIT - This state configures the LED pin to an output
 *           and lowers the pin for initialization.
 *       - state 1: LED_OFF - This state increments a count variable until
 *           0.9 seconds goes by and then raises the LED pin.
 *       - state 2: LED_ON - This state increments a count variable until
 *           0.1 seconds go by and then lowers the LED pin.
 */
void task_led :: run (void)
{
    // Set the next time so the task will run again.
    set_next_time ();

    // Finite-state-machine. This switch statement determines which state
    // to run based on the current value of the state variable.
    switch (state)
    {
        /* State 0. */
        case INIT:
            // Configure LED pins and initialize low.
            PORTB.DIR |= MICROSD_LED | AD7708_LED | ADC_LED | OTHER_LED;
            PORTB.OUT &= ~(MICROSD_LED | AD7708_LED | ADC_LED | OTHER_LED);

            // Transition to state 1.
            transition_to (LED_OFF);
            break;
        /* State 1. */
        case LED_OFF:
```

```
            count++;
            if (count >= (freq_share->retrieve () / timing))
            {
                PORTB.OUT |= OTHER_LED;
                count = 0;
                transition_to (LED_ON);
                break;
            }
            break;
        /* State 2. */
        case LED_ON:
            count++;
            if (count >= freq_share->retrieve () / timing)
            {
                PORTB.OUT &= ~OTHER_LED;
                count = 0;
                transition_to (LED_OFF);
                break;
            }
            break;
        /* Dead state. */
        case DEAD:
            break;
        /* Undefined State. */
        default:
            // Default state just in case the task enters an undefined state.
            #ifdef  TASK_DEBUG
                *p_serial << p_name << " task entered undefined state!" << endl;
            #endif

            // Send task to a dead state where no code is executed.
            transition_to (DEAD);
            break;
    };
}
```

```
/***********************************************************************/
/** @file task_user.h
 *  @brief User interface task header file.
 *
 *  @details This is a task that serves as a user interface that stores
 *  and decodes commands input by the user over a serial port.
 *
 *   Contributors:
 *      - Zach Wilson 12/16/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/16/2018 - initial release.
 *
 *  Created: 12/16/2018 9:51:37 PM
 */
/************************************************************************/


#ifndef TASK_USER_H_
#define TASK_USER_H_

/* Included files. */
#include "serial_port.h"
#include "task_base.h"
#include "coscheduler.h"
#include "share.h"
#include "queue.h"

////////////////////////////////////////////////////////////////////////////
/** Defines the maximum command string length.
 */
#define  USER_COMMAND_LENGTH  15

////////////////////////////////////////////////////////////////////////////
/** Declare the external string to integer conversion function defined
 *  in string.h for use in converting inputs from the user.
 */
extern uint16_t stringToInt (char *);


////////////////////////////////////////////////////////////////////////////
/** @brief User interface task.
 *  @details This is a task that provides an interface to the user through
 *  the serial port for controlling the device.
 */
class task_user : public task_base
{
    private:
        /** Enumeration defining the task states.
         */
        typedef enum  TASK_USER_STATES
        {
            /** State 0. Print to the serial port and transition to state
             *  1.
             */
            INIT = 0,

            /** State 1. Check if a character was received over the serial
             *  port. Transition to state 2 if the character was ctrl+c.
             */
            HUB,

            /** State 2. Store characters received over the serial port
             *  into a string for a command. Transition to state 3 when
             *  the user hits 'enter' or to state 1 when the user hits
             *  ctrl+c.
             */
            COMMAND_MODE,

            /** State 3. Decode the command string and carry out the
             *  necessary action. Always transition to state 4.
             */
            DECODE,

            /** State 4. Waits until the scheduler is done printing debug
             *  reports. Always transitions back to state 1.
             */
            WAIT,

            /** State 5. Prompt the user to enter a frequency. Convert the
             *  entered number to a decimal and send it to the frequency
             *  share. Transition back to state 1.
             */
            SELECT_FREQ,

            /** Dead state if something goes wrong during task execution.
             */
            DEAD
        } TASK_USER_STATES_t;


        /** Enumeration defining the possible the user can send through the
```

```
 *  chosen serial port. These integers are the results of the
 *  decode () command that decodes the string entered by the user.
 */
typedef enum  TASK_USER_COMMANDS
{
    /** 'debug' command. This command causes a task debug report to
     *  print to the serial port.
     */
    DEBUG_COMMAND = 1163,

    /** 'state' command. This command causes a task state report to
     *  print to the serial port.
     */
    STATE_COMMAND = 1181,

    /** 'share' command. This command causes a share debug report to
     *  print to the serial port.
     */
    SHARE_COMMAND = 1161,

    /** 'share' command. This command causes a queue debug report to
     *  print to the serial port.
     */
    QUEUE_COMMAND = 1189,

    /** 'frequency' command. This command will send a prompt to the user
     *  that asks for a new data acquisition frequency. The next
     *  16-bit number entered by the user will be the new frequency
     *  of data acquisition in milliseconds.
     */
    FREQ_COMMAND = 4060,

    /** 'static' command. This command will cause the device to
     *  switch to static data acquisition mode.
     */
    STATIC_COMMAND = 1696,

    /** 'dynamic' command. This command will cause the device to
     *  switch to dynamic data acquisition mode.
     */
    DYNAMIC_COMMAND = 2292,

    /** 'start' command. This command will cause the device to start
     *  taking data based on the current settings of the device.
     */
    START_COMMAND = 1232,

    /** 'stop' command. This command will cause the device to stop
     *  taking static data.
     */
    STOP_COMMAND = 791,

    /** 'ADC' command. This command causes the device to select the
     *  on-board ADC interface for reading strain.
     */
    ADC_COMMAND = 268,

    /** 'AD7708' command. This command causes the device to select the
     *  AD7708 device for reading strain.
     */
    AD7708_COMMAND = 883,

    /** 'help' command. This command will print the available commands and
     *  their descriptions to the serial port.
     */
    HELP_COMMAND = 884,
} TASK_USER_COMMANDS_t;

/** Pointer to a share for the LED blinking frequency.
 */
share <uint16_t> *freq_share;

/** Pointer to a share for the data acquisition frequency.
 */
share <uint16_t> *acq_freq_share;

/** Storage for the command string.
 */
char command [USER_COMMAND_LENGTH];

/** Number for storing the result of the decode function.
 */
//uint16_t command_result;

/** Pointer to the command string.
 */
char *p_command;

/** Pointer to the command string for decoding the command.
 */
char *p_decode;
```

```
        /** Count for the number of iterations of the WAIT state.
         */
        uint8_t count;

        /** @brief Decode command.
         *  @details This method decodes the command in the command string by
         *  adding up the decimal values of each character and returning the
         *  resulting number.
         *  @return A unique 16-bit number corresponding to the command that
         *  was entered by the user.
         */
        uint16_t decode (void);

    public:
        /** @brief Default constructor.
         *  @details This constructor does nothing but pass the pointer to
         *  the serial port to the base task constructor and saves share and
         *  queue pointers by reading them from the share and queue lists and
         *  statically casting them into the desired type.
         *  @param serial Pointer to a serial port.
         *  @param name Pointer to a string of the task name.
         *  @param priority Priority at which to schedule the task.
         *  @param timing Timing at which to run the task.
         */
        task_user (serial_port *, const char *, uint8_t, uint8_t);

        /** @brief Initialize task.
         *  @details This method initializes the task by initializing all
         *  task variables and starting the task in state 0.
         */
        void initialize (void);

        /** @brief Run task.
         *  @details This method runs one iteration of the finite-state-
         *  machine of the task.
         *       - state 0: INIT - Print to the serial port and transition.
         *       - state 1: HUB - Check if a character was received over the
         *           serial port. Transition to state 2 if the character was
         *           ctrl+c.
         *       - state 2: COMMAND_MODE - Store characters received over the
         *           serial port into a string for a command. Transition to
         *           state 3 when the user hits 'enter' or to state 1 when the
         *           user hits ctrl+c.
         *       - state 3: DECODE - Decode the command string and carry out
         *           the necessary action. Always transition to state 4.
         *       - state 4: WAIT - Waits until the scheduler is done printing
         *           debug reports. Always transitions back to state 1.
         *       - state 5: SELECT_FREQ - Prompt the user to enter a frequency.
         *           Convert the entered number to a decimal and send it to the
         *           frequency share. Transition back to state 1.
         */
        void run (void);
};


#endif /* TASK_USER_H_ */
```

```
/***********************************************************************/
/** @file task_user.cpp
 *  @brief User interface task main class file.
 *
 *   Contributors:
 *      - Zach Wilson 12/16/2018 <zawilson@calpoly.edu>
 *
 *  ChangeLog:
 *      - 12/16/2018 - initial release.
 *
 *  Created: 12/16/2018 10:38:10 PM
 */
/************************************************************************/


/* Included files. */
#include "task_user.h"


////////////////////////////////////////////////////////////////////////
/** @brief Default constructor.
 *  @details This constructor does nothing but pass the pointer to
 *  the serial port to the base task constructor and saves share and
 *  queue pointers by reading them from the share and queue lists and
 *  statically casting them into the desired type.
 *  @param serial Pointer to a serial port.
 *  @param name Pointer to a string of the task name.
 *  @param priority Priority at which to schedule the task.
 *  @param timing Timing at which to run the task.
 */
task_user :: task_user (serial_port *serial, const char *name, uint8_t priority, uint8_t timing)
                          : task_base (serial, name, priority, timing)
{
    // Save and statically cast share and queue pointers from the
    // scheduler lists.
    freq_share = static_cast <share <uint16_t> *> (share_list.read (0));
    acq_freq_share = static_cast <share <uint16_t> *> (share_list.read (1));
}


////////////////////////////////////////////////////////////////////////
/** @brief Initialize task.
 *  @details This method initializes the task by initializing all
 *  task variables and starting the task in state 0.
 */
void task_user :: initialize (void)
{
    // Initialize the global flag struct.
    global_flags.read_HX711 = false;
    global_flags.read_static_AD7708 = false;
    global_flags.read_dynamic_AD7708 = false;
    global_flags.read_static_ADC = false;
    global_flags.read_dynamic_ADC = false;
    global_flags.saving_dynamic_data = false;
    global_flags.daq_mode = false;
    global_flags.daq_method = false;

    // Initialize the task variables.
    p_command = &command [0];
    p_decode = &command [0];
    //command_result = 0;

    // Indicate through serial port that the task initialized
    // successfully.
    #ifdef  TASK_DEBUG
        *p_serial << p_name << " task initialized." << endl;
    #endif

    // Set the state variable to state 0.
    transition_to (INIT);
}


////////////////////////////////////////////////////////////////////////
/** @brief Run task.
 *  @details This method runs one iteration of the finite-state-
 *  machine of the task.
 *       - state 0: INIT - Print to the serial port and transition.
 *       - state 1: HUB - Check if a character was received over the
 *           serial port. Transition to state 2 if the character was
 *           ctrl+c.
 *       - state 2: COMMAND_MODE - Store characters received over the
 *           serial port into a string for a command. Transition to
 *           state 3 when the user hits 'enter' or to state 1 when the
 *           user hits ctrl+c.
 *       - state 3: DECODE - Decode the command string and carry out
 *           the necessary action. Always transition to state 4.
 *       - state 4: WAIT - Waits 5 runs to make sure the scheduler is
 *           done printing debug reports. Always transitions back to
 *           state 1.
 *       - state 5: SELECT_FREQ - Prompt the user to enter a frequency.
 *           Convert the entered number to a decimal and send it to the
```

```
 *           frequency share. Transition back to state 1.
 */
void task_user :: run (void)
{
    // Set the next time so the task will run again.
    set_next_time ();

    // Finite-state-machine. This switch statement determines which state
    // to run based on the current value of the state variable.
    switch (state)
    {
        /* State 0. */
        case INIT:
            // Send intro for user interface and send a frequency of 500ms and 1000ms
            // to the frequency shares.
            *p_serial << endl << "Press ctrl+c to enter command mode." << endl << endl;
            freq_share->send (500);
            acq_freq_share->send (1000);

            // Transition to state 1.
            transition_to (HUB);
            break;
        /* State 1. */
        case HUB:
            // Check if a character has been received over the serial port.
            if (p_serial->check_buf ())
            {
                // If ctrl+c is received, transition to state 2 and indicate
                // that the program is in command mode.
                if (p_serial->get_char () == 0x03)
                {
                    *p_serial << "Enter a command: ";
                    transition_to (COMMAND_MODE);
                    break;
                }
            }
            break;
        /* State 2. */
        case COMMAND_MODE:
            // Check if a character has been received over the serial port.
            if (p_serial->check_buf ())
            {
                // Pull the character that was received from the receive buffer and
                // store it on the stack.
                char character = p_serial->get_char ();

                // If ctrl+c was pressed, return to hub state and reset all command
                // variables.
                if (character == 0x03)
                {
                    *p_serial << endl << endl << "Returned to hub state." << endl << endl
                                              << "Press ctrl+c to enter command mode." << endl << endl;
                    p_command = &command [0];
                    p_decode = &command [0];
                    transition_to (HUB);
                    break;
                }
                // Transition to state 3 if the user hits 'enter', indicating that the
                // command string is finished.
                else if (character == 0x0D)
                {
                    *p_serial << endl;
                    transition_to (DECODE);
                    break;
                }
                // Ignore unwanted characters. Pick up another character at the next
                // task run.
                else if (character <= 0x1F) break;

                // Decrement the command pointer if the character was a delete.
                else if (character == 0x7F)
                {
                    // Only decrement and print the character if there are characters
                    // to delete from the command string.
                    if (p_command <= &command [0]) break;
                    p_command--;

                    // Echo the character.
                    *p_serial << character;
                    break;
                }
                // Store the character in the command string.
                else if (p_command <= &command [USER_COMMAND_LENGTH - 1])
                {
                    *p_command = character;
                    p_command++;

                    // Echo the character.
                    *p_serial << character;
                    break;
                }
            }
```

```
    break;
/* State 3. */
case DECODE:
    // This bracket resolves the scope of the command_result stack variable.
    {
        // Decode the command and store the result on the stack.
        uint16_t command_result = decode ();
        //*p_serial << command_result; // Print the result of the decode for debugging.

        // Determine which command was entered and take the appropriate action.
        switch (command_result)
        {
            case START_COMMAND: // 'start' command.
                // Raise the appropriate flag to start acquiring data based on
                // the current settings.
                if (global_flags.daq_method)
                {
                    global_flags.daq_mode ? global_flags.read_dynamic_ADC = true
                    : global_flags.read_static_ADC = true;
                }
                else
                {
                    global_flags.daq_mode ? global_flags.read_dynamic_AD7708 = true
                    : global_flags.read_static_AD7708 = true;
                }

                // Transition to state 4.
                transition_to (WAIT);
                break;
            case STATIC_COMMAND: // 'static' command.
                // Raise the global read static strain flag to signal to the
                // AD7708 task to begin gathering data.
                global_flags.daq_mode = false;
                *p_serial << "Set data acquisition mode to static." << endl << endl;

                // Transition to state 1.
                transition_to (HUB);
                break;
            case DYNAMIC_COMMAND: // 'dynamic' command.
                // Raise the global read dynamic strain flag to signal to the
                // ADC task to begin gathering data.
                global_flags.daq_mode = true;
                *p_serial << "Set data acquisition mode to dynamic." << endl << endl;

                // Transition to state 1.
                transition_to (HUB);
                break;
            case STOP_COMMAND: // 'stop' command.
                // Lower both read static flags to signal to the tasks to
                // stop acquiring data.
                global_flags.read_static_AD7708 = false;
                global_flags.read_static_ADC = false;
                *p_serial << "Stopped acquiring data." << endl << endl;
                transition_to (WAIT);
                break;
            case FREQ_COMMAND: // 'frequency' command.
                // Prompt the user to enter the desired data acquisition frequency.
                *p_serial << endl << "Enter a frequency between 0 - 65,535 (ms): ";

                // Reset the character pointers and transition to state 5.
                p_command = &command [0];
                p_decode = &command [0];
                transition_to (SELECT_FREQ);
                break;
            case ADC_COMMAND: // 'ADC' command.
                // Raise the global device selection flag to select the ADC interface
                // for reading data.
                global_flags.daq_method = true;
                *p_serial << "Selected ADC interface for acquiring data." << endl << endl;

                // Transition to state 1.
                transition_to (HUB);
                break;
            case AD7708_COMMAND: // 'AD7708' command.
                // Lower the global device selection flag to select the AD7708 device
                // for reading data.
                global_flags.daq_method = false;
                *p_serial << "Selected AD7708 for acquiring data." << endl << endl;

                // Transition to state 1.
                transition_to (HUB);
                break;
            case DEBUG_COMMAND: // 'debug' command.
                // Raise the task debug report flag to run a task debug
                // report from the scheduler.
                runDebugReport ();
                transition_to (WAIT);
                break;
            case STATE_COMMAND: // 'state' command.
                // Raise the task state report flag to run a task state
                // report from the scheduler.
                runStateReport ();
```

```
                    transition_to (WAIT);
                    break;
                case SHARE_COMMAND: // 'share' command.
                    // Raise the share debug report flag to run a share debug
                    // report from the scheduler.
                    runShareReport ();
                    transition_to (WAIT);
                    break;
                case QUEUE_COMMAND: // 'queue' command.
                    // Raise the queue debug report flag to run a queue debug
                    // report from the scheduler.
                    runQueueReport ();
                    transition_to (WAIT);
                    break;
                case HELP_COMMAND: // 'help' command.
                    // Print the names and descriptions of all available commands.
                    /*
                    *p_serial << "Command List:" << endl
                            << " 'start':     Start data acquisition based on current device settings." << endl
                            << " 'static':        Selects static mode of data acquisition. (1Hz)." << endl
                            << " 'dynamic':       Selects Dynamic mode of data acquisition. (100Hz - 1000Hz)." << endl
                            << " 'stop':          Stops static data acquisition." << endl
                            << " 'frequency': Prompts user to enter a frequency between 0 and 65,535 ms." << endl
                            << " 'ADC':           Selects the on-board ADC as the mode of data acquisition." << endl
                            << " 'AD7708':        Selects the external AD7708 device for data acquisition." << endl
                            << " 'debug':     Runs a task debug report." << endl
                            << " 'state':     Runs a task state report." << endl
                            << " 'share':     Runs a share debug report." << endl
                            << " 'queue':     Runs a queue debug report." << endl
                            << " 'help':          Prints all available commands to the serial port." << endl << endl;
                    */
                    transition_to (WAIT);
                    break;
                default:
                    // Send an indication if no match was found. Reset the
                    // character pointers.
                    *p_serial << "Invalid command." << endl << endl << "Enter a command: ";

                    // Transition to state 2.
                    transition_to (COMMAND_MODE);
                    break;
            };
            // Reset character pointers.
            p_command = &command [0];
            p_decode = &command [0];
            break;
        }
        break;
    /* State 4. */
    case WAIT:
        count++;

        // Transition to state 1 after 5 iterations.
        if (count >= 5)
        {
            // Transition back to state 1 once all flags are lowered.
            if (!global_flags.read_dynamic_AD7708 &&
                    !global_flags.read_dynamic_ADC &&
                    !global_flags.saving_dynamic_data)
            {
                p_command = &command [0];
                p_decode = &command [0];
                count = 0;
                transition_to (HUB);
                break;
            }
            break;
        }
        break;
    /* State 5. */
    case SELECT_FREQ:
        // Check if a character has been received over the serial port.
        if (p_serial->check_buf ())
        {
            // Pull the character that was received from the receive buffer and
            // store it on the stack.
            char character = p_serial->get_char ();

            // If ctrl+c was pressed, return to hub state and reset all command
            // pointers.
            if (character == 0x03)
            {
                *p_serial << endl << endl << "Returned to hub state." << endl << endl
                << "Press ctrl+c to enter command mode." << endl << endl;
                p_command = &command [0];
                p_decode = &command [0];
                transition_to (HUB);
                break;
            }
            // If enter was pressed, append the string with a NULL character and reset
            // the pointer. Convert the string to an integer and send it to the frequency
            // share and transition to state 1.
```

```
                else if (character == 0x0D)
                {
                    *p_command = NULL;
                    p_command = &command [0];
                    p_decode = &command [0];
                    acq_freq_share->send (stringToInt (p_command));
                    *p_serial << endl << "Data acquisition frequency set at "
                                << acq_freq_share->retrieve () << "ms." << endl << endl;
                    transition_to (HUB);
                    break;
                }
                // Decrement the command pointer if the character was a delete.
                else if (character == 0x7F)
                {
                    // Only decrement and print the character if there are characters
                    // to delete from the command string.
                    if (p_command <= &command [0]) break;
                    p_command--;

                    // Echo the character.
                    *p_serial << character;
                    break;
                }
                // Ignore unwanted characters. Pick up another character at the next
                // task run.
                else if (character <= 0x2F || character >= 0x3A) break;

                // Store the character in the command string.
                else if (p_command <= &command [USER_COMMAND_LENGTH - 6])
                {
                    *p_command = character;
                    p_command++;

                    // Echo the character.
                    *p_serial << character;
                    break;
                }
            }
            break;
        /* Dead state. */
        case DEAD:
            break;
        /* Undefined State. */
        default:
            // Default state just in case the task enters an undefined state.
            #ifdef  TASK_DEBUG
                *p_serial << p_name << " task entered undefined state!" << endl;
            #endif

            // Send task to a dead state where no code is executed.
            transition_to (DEAD);
            break;
    };
}


//////////////////////////////////////////////////////////////////////////
/** @brief Decode command.
 *  @details This method decodes the command in the command string by
 *  adding up the decimal values of each character and returning the
 *  resulting number.
 *  @return A unique 16-bit number corresponding to the command that
 *  was entered by the user.
 */
uint16_t task_user :: decode (void)
{
    uint16_t command_result = 0;
    for (uint8_t index = 0; p_decode < p_command; index++)
    {
        command_result += (*p_decode % index + *p_decode * index);
        p_decode++;
    }
    return command_result;
}
```

J. DERIVATION OF CANTILEVER BEAM EQUATION OF MOTION AND FIRST NATURAL FREQUENCY

The equation of motion of the transverse deflection of a cantilever beam was derived to design the beams used in the dynamic testing of the Dynamic Data Acquisition System. From the equation of motion, an expression for the first natural frequency was derived which was used to design the length of the beams. The Assumed Modes method was used to derive the coupled equations of motion for the first two modes of oscillation. The schematic of the system is shown in Figure J-1.

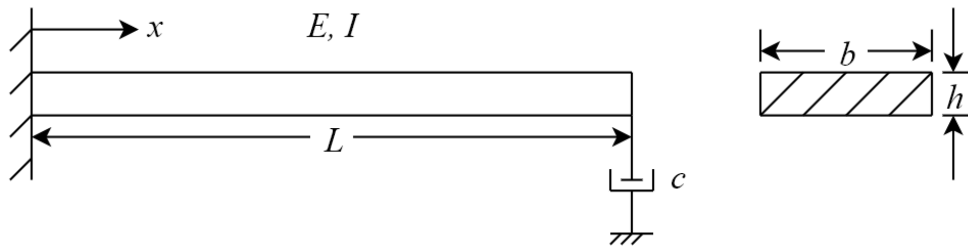


Figure J-1: Dynamic cantilever beam model schematic.

The schematic assumes that the beam is non-rigid, and the mass is evenly distributed throughout the beam. A point-damper was included at the end of the beam as well to add damping to the response. The Assumed Modes method results in an equation of motion of the form seen in Equation J-1, where M, C, and K are matrices and Q is a vector containing the time-varying transverse deflections of the first two modes of oscillation.

$$M\ddot{Q} + C\dot{Q} + KQ = 0 \quad Q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \tag{J-1}$$

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \tag{J-2}$$

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \tag{J-3}$$

$$K = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \tag{J-4}$$

When utilizing the Assumed Modes method, it is necessary to assume the mode shapes of the system. The chosen mode shapes are displayed in Figure J-2 and detailed in Equation J-5 and Equation J-6.

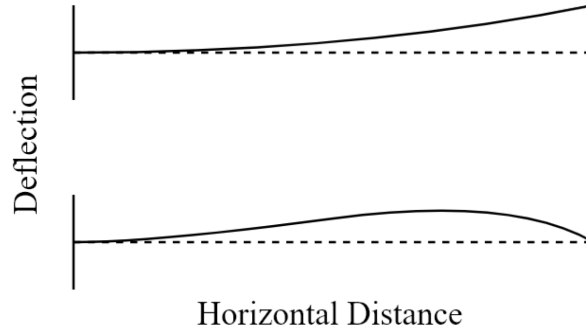


Figure J-2: Assumed mode shapes of cantilever beam model for (top) first mode and (bottom) second mode.

$$\psi_1(x) = \left(\frac{x}{L}\right)^2 \quad \text{(J-5)}$$

$$\psi_2(x) = \left(\frac{x}{L}\right)^2 - \left(\frac{x}{L}\right)^3 \quad \text{(J-6)}$$

Then, to find the total dynamic response of the system, Equation J-7 is used.

$$v(x,t) = \psi_1(x)\, q_1(t) + \psi_2(x)\, q_2(t) \quad \text{(J-7)}$$

Before employing the Assumed Modes method, it is necessary to calculate the first and second derivatives of both mode shape functions.

$$\psi_1{}'(x) = \frac{2}{L^2}x \quad \text{(J-8)}$$

$$\psi_1{}''(x) = \frac{2}{L^2} \quad \text{(J-9)}$$

$$\psi_2{}'(x) = \frac{2}{L^2}x - \frac{3}{L^3}x^2 \quad \text{(J-10)}$$

$$\psi_2{}''(x) = \frac{2}{L^2} - \frac{6}{L^3}x \quad \text{(J-11)}$$

Then, using the continuous Principle of Virtual Displacements, the stiffness matrix was populated.

$$k_{11} = \int_0^L EI\, \psi_1''(x)\ \psi_1''(x)\, dx \tag{J-12}$$

$$k_{11} = \int_0^L EI \left(\frac{2}{L^2}\right)\left(\frac{2}{L^2}\right) dx \tag{J-13}$$

$$k_{11} = \frac{4EI}{L^3} \tag{J-14}$$

$$k_{12} = k_{21} = \int_0^L EI\, \psi_1''(x)\ \psi_2''(x)\, dx \tag{J-15}$$

$$k_{12} = k_{21} = \int_0^L EI \left(\frac{2}{L^2}\right)\left(\frac{2}{L^2} - \frac{6}{L^3}x\right) dx \tag{J-16}$$

$$k_{12} = k_{21} = -\frac{2EI}{L^3} \tag{J-17}$$

$$k_{22} = \int_0^L EI\, \psi_2''(x)\ \psi_2''(x)\, dx \tag{J-15}$$

$$k_{22} = \int_0^L EI \left(\frac{2}{L^2} - \frac{6}{L^3}x\right)\left(\frac{2}{L^2} - \frac{6}{L^3}x\right) dx \tag{J-16}$$

$$k_{22} = \frac{4EI}{L^3} \tag{J-17}$$

Similarly, the mass matrix was also populated using the continuous Principle of Virtual Displacements.

$$m_{11} = \int_0^L \rho A\, \psi_1(x)\ \psi_1(x)\, dx \tag{J-18}$$

$$m_{11} = \int_0^L \rho A \left(\left(\frac{x}{L}\right)^2\right)\left(\left(\frac{x}{L}\right)^2\right) dx \tag{J-19}$$

$$m_{11} = \frac{\rho A L}{5} \tag{J-20}$$

$$m_{12} = m_{21} = \int_0^L \rho A\, \psi_1(x)\ \psi_2(x)\, dx \tag{J-18}$$

$$m_{12} = m_{21} = \int_0^L \rho A \left(\left(\frac{x}{L}\right)^2\right)\left(\left(\frac{x}{L}\right)^2 - \left(\frac{x}{L}\right)^3\right) dx \tag{J-19}$$

$$m_{12} = m_{21} = \frac{\rho A L}{30} \tag{J-20}$$

$$m_{22} = \int_0^L \rho A\, \psi_2(x)\ \psi_2(x)\, dx \tag{J-21}$$

$$m_{22} = \int_0^L \rho A \left(\left(\frac{x}{L}\right)^2 - \left(\frac{x}{L}\right)^3\right)\left(\left(\frac{x}{L}\right)^2 - \left(\frac{x}{L}\right)^3\right) dx \quad \text{(J-22)}$$

$$m_{22} = \frac{\rho A L}{105} \quad \text{(J-23)}$$

The only damping that exists in the model is a point-damper, thus the discrete Principle of Virtual displacements was utilized to populate the damping matrix.

$$c_{11} = c\, \psi_1(L)\, \psi_1(L) \quad \text{(J-24)}$$

$$c_{11} = c \quad \text{(J-25)}$$

$$c_{12} = c_{21} = c\, \psi_1(L)\, \psi_2(L) \quad \text{(J-26)}$$

$$c_{12} = c_{21} = 0 \quad \text{(J-27)}$$

$$c_{22} = c\, \psi_2(L)\, \psi_2(L) \quad \text{(J-28)}$$

$$c_{22} = 0 \quad \text{(J-29)}$$

In summary, the above calculations result in the following mass, damping, and stiffness matrices for the system.

$$M = \rho A L \begin{bmatrix} 1/5 & 1/30 \\ 1/30 & 1/105 \end{bmatrix}, \; C = c \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \; K = \frac{EI}{L^3}\begin{bmatrix} 4 & -2 \\ -2 & 4 \end{bmatrix} \quad \text{(J-30)}$$

To determine the natural frequencies of the system, the characteristic equation of the system was used, displayed in Equation J-31.

$$|K - \omega^2 M| = 0 \quad \text{(J-31)}$$

$$\begin{vmatrix} \frac{4EI}{L^3} - \frac{\rho A L}{5}\omega^2 & \frac{-2EI}{L^3} - \frac{\rho A L}{30}\omega^2 \\ \frac{-2EI}{L^3} - \frac{\rho A L}{30}\omega^2 & \frac{4EI}{L^3} - \frac{\rho A L}{105}\omega^2 \end{vmatrix} = 0 \quad \text{(J-32)}$$

$$\omega^4 - 1224\frac{EI}{\rho A L^4}\omega^2 + 15120\left(\frac{EI}{\rho A L^4}\right)^2 = 0 \quad \text{(J-33)}$$

Using the quadratic equation, the above equation was solved. The two solutions obtained from solving the characteristic equation are expressions for the first and second natural frequencies

of the system. The expression for the first natural frequency was used to design the length of the cantilever beams.

$$\omega_{n_1} = 3.53\sqrt{\frac{EI}{\rho A L^4}} \tag{J-34}$$

$$\omega_{n_2} = 34.8\sqrt{\frac{EI}{\rho A L^4}} \tag{J-35}$$