A DESIGN AND IMPLEMENTATION OF A LOW-POWER EMBEDDED SYSTEM FOR DATA COLLECTION IN AN AIRBORNE SEA ICE THICKNESS OBSERVING SYSTEM

By

Thimira Sanuka Thilakarathna Asurapmudalige, B.S.

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

In

Electrical Engineering

University of Alaska Fairbanks

December 2022

APPROVED:

Dr. Dejan Raskovic, Committee Chair
Dr. Micheal Hatfield, Committee Member
Dr. Denise Thorsen, Committee Member
Dr. Denise Thorsen, Chair
Department of Electrical Engineering
Dr. William Schnabel, Dean
College of Engineering and Mines
Dr. Richard Collins, Director
Graduate School

Abstract

The Long-Range Airborne Snow and Sea Ice Thickness Observing System (LASSITOS) is an airborne electromagnetic (AEM) system, currently under development, which uses a custom-designed instrument mounted on an Unmanned Aerial System (UAS) to measure Arctic sea ice and snow thickness. This project requires specialized instruments that are both low-power and lightweight.

This thesis describes a design and implementation of a prototype data logging system based on an ultra-low power microcontroller, for the LASSITOS instrument. Three 32-bit Analog-to-Digital Converter (ADC) integrated circuits (IC) are used to sample and convert the receiving EM signal at a rate of 19200 SPS. The system is capable of writing the sampled data and diagnostic data to the SD card at a combined rate of up to 307200 B/s. A 30 KB circular buffer is used to avoid data loss during SD card busy periods. Three DMA channels are used to optimize the communication between the ADCs and the SD card over SPI to achieve these data rates.

Table of Contents

Page

List of Figures

List of Tables

Acknowledgments

I would like to thank those who made this thesis possible.

Specifically, I would like to thank Dr. Dejan Raskovic for taking me as a research student and including me to this project and giving guidance. My appreciation also goes to my committee members Dr. Denise Thorsen and Dr. Michal Hatfield for all the advice and support. I would like to thank Dr. Achille Capelli for helping me with lab tests in this project. I would like to thank all the faculty and staff in the Department of Electrical Engineering.

I would like to give special thanks my best friends Dan Reichardt and Amber Reichardt for supporting me to write, proofread and format this thesis and all their encouragement. I also would like to thank Prof. Paul Reichardt and Terry Reichardt for all the encouragement and advice. Last, but not least, I would like to thank my wife Hashini Jayakody for her support and patience and my parents Thilak and Sandya who have guided and encouraged me since my childhood.

Chapter 1 Introduction

Accurate knowledge of sea ice distribution is crucial for the understanding of Arctic ice cover and future predictions of Arctic marine environments. With the Arctic ice pack transitioning from perennial to seasonal, ice thickness is a key variable in understanding the ice-ocean system [1].

Airborne electromagnetic (AEM) methods allow measurement of the upper and lower limits of the ice thickness within 10% accuracy of total ice thickness [1]. The MAiSIE [2] instrument and EM-Bird [3] are examples of AEM sea ice measurement devices that are mounted to a helicopter. However, current AEM systems, including MAiSIE and EM-Bird, do not account for snow depth, which can be a significant fraction of ice thickness. Additionally, helicopter mounted systems are limited both from the perspective of total range and the high cost of helicopter flight time.

The purpose of developing the Long-Range Airborne Snow and Sea Ice Thickness Observing System (LASSITOS) is to measure sea ice thickness on a large scale. The sea ice data collection, enabled by developing and sharing this instrument, will be used by the scientific community, federal agencies, industry, and Arctic residents.

The British Antarctic Survey (BAS) developed a sled equipped with a lightweight and rugged ice-thickness electromagnetic (EM) sensor, which was used to measure sea ice thickness in localized areas. It used a Single-Board Computer (SBC), rather than a Personal Computer, which represented a significant improvement in miniaturization among EM-based ice thickness measuring systems. [1]. The LASSITOS team was given possession of the BAS sled and is

building upon this open-source design with further miniaturizing using low-power and lightweight components to be deployed with an Unmanned Aerial System (UAS).

1.1 LASSITOS

The LASSITOS consists of a UAS and an instrument that is currently under development. The LASSITOS instrument consists of an EM subsystem, snow radar, laser altimeter, Global Navigation Satellite System (GNSS) and inertial navigation unit (INU) that will be carried by the UAS in Arctic sea ice regions and will collect data related to snow and sea ice thickness for post-processing and analysis. Minimizing the weight and miniaturization of the instrument are key design objectives of this project because it directly affects the performance of the UAS.

1.2 Design Concept of LASSITOS Instrument

The LASSITOS will collect data related to three distance measurements (H_s, H_i, and H_w) as shown in Figure 1.1. To measure the distance to the upper surface of snow or ice (H_s), the laser altimeter will be used. The snow radar will be used to measure the distance to the upper ice surface (H_i), while the EM subsystem will be used to measure the distance to the ice-water interface (H_w). A GNSS and INU are required to monitor the position, altitude, and attitude of the instrument during the flight.

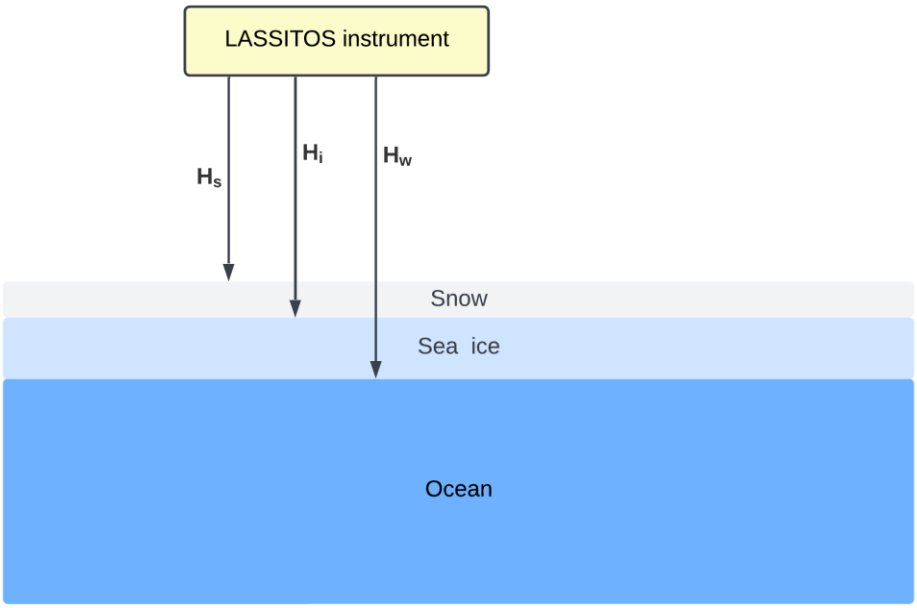

Figure 1.1: LASSITOS instrument distance measurements.

The EM subsystem consists of an EM transmitter (TX) and a receiver (RX) system. The TX system transmits a low-frequency (~0.5-10 kHz) EM field, which is the primary field through a TX coil. This transmitted field induces a secondary field in the conductive sea water under the sea ice. The RX system measures the sum of these two fields. At these low frequencies, the conductivity of the sea ice is similar to that of the free space. Analyzing the received signal's field strength and phase information in the frequency domain, the H_w can be determined. With these distance measurements, both the snow depth (H_{snow}) and the ice thickness (H_{ice}) can be calculated according to Equations 1.1 and 1.2.

$$H_{snow} = H_i - H_s \quad (1.1)$$

$$H_{ice} = H_w - H_i \quad (1.2)$$

A specialized data logging system is required for signal conditioning, sampling, converting, and saving the received signal. This thesis describes the prototype design of the data logger for the receiver system of the EM subsystem for H_w measurement.

1.3 Receiver System of the EM Subsystem

The LASSITOS research group received the EM-sled developed by the BAS, which was examined to determine the approach they used and limitations of their EM system. The LASSITOS team developed a prototype embedded system, which is a modification of the BAS EM system, focusing on design improvements to reduce ground noise and increase resolution of Analog-to-Digital Converter (ADC) sampled data.

There are three RX coils in the RX system and the developed system consists of three RX channel boards and a microcontroller development board, which is the development board designed and widely used in the University of Alaska Fairbanks (UAF) Space Systems Engineering Program (SSEP). These three RX channel boards are used for axial measurements X, Y, and Z of receiving signals from the three coils. Each RX channel board consists of three stages; the first two stages filter and amplify the receiving signal that comes from the receiver coil and the third stage converts the amplified signal into a digital signal using an ADC. The ADS1263 [4] integrated circuit (IC) is used as the ADC which provides 32-bit digitized samples at a rate of 19200 samples per second (SPS).

The microcontroller development board uses an MSP430F6779A (MSP) [5] microcontroller with a Secure Data (SD) card slot. The MSP controls the ADCs, collects sampled data, and arranges and writes data to the SD card as 512-byte-long data frames.

One of the design objectives for the microcontroller software is to collect all sampled data from all the RX channel boards without data loss. The Direct Memory Access (DMA) controller of the MSP is used to maximize the data flow rate between the ADCs and the SD card. A 30 KB-circular data buffer is implemented to hold data while the SD card is in a busy state, thereby avoiding data loss. Using this developed prototype, the system was able to write sampled data to high speed SD cards at the rate of 307200 Bytes per second through Serial Peripheral Interface (SPI).

1.4 Prior Work Review

One of the primary design challenges of this data logger was efficient collection of incoming data, which then needed to be organized and written to an SD card using an ultra-low power microcontroller, such as the MSP430 family microcontrollers [6]. Such a design would allow this system to be used in other long-term, battery operated data collection systems.

According to the SD specifications [7], the SD card busy states occur writing data . The frequency of busy states can be reduced by increasing the number of blocks per write transaction [8] and it can help to increase the throughput of the SD card communication.

During the development of the LASSITOS system, a variety of other research projects were reviewed, some of which were instructive. EM receivers developed for tunnel exploration [9] have used an FPGA and ARM-based data logging system to log data to an SD card. The iNODE-5 instrument [10] used an MSP430F5659-based system to collect multiple sensor data and save it to an SD card. Both designs used the double-buffering technique, wherein one buffer receives data from sensors while the other buffer is used to write to the SD card. The Underwater Acoustic Recorder [11] used multiple MSP430 microcontrollers and multiple SD cards to

achieve higher data rates. The approaches in each of these designs were considered in developing this system.

In this thesis, Chapter 2 explains the hardware design of the prototype embedded system, designed for receiver of the EM subsystem of LASSITOS. Chapter 3 describes the software that runs on the microcontroller. It also discusses the techniques used to optimize data communication between the ADCs and the SD card. In order to verify the designed system functioning as expected, a series of lab tests have been conducted, and the Chapter 4 presents and discusses these test results. Chapter 5 contains an overall conclusion and recommendations about the future work related to this project. The source code of the software that runs on the MSP430F6779A can be found in the Appendix.

Chapter 2 Receiver System Hardware Design

The EM subsystem of the LASSITOS instrument consists of a TX system and an RX system. The TX system generates a primary EM field that is transmitted from the TX coil. This EM field induces a secondary field on the surface of the sea water and the RX coils of the RX system pick up both primary and secondary fields. The primary and secondary fields generate a voltage signal on the RX coil that is then sampled and converted into digital format and stored for later use. The block diagram in Figure 2.1 shows the RX coils on the left and the data storage (SD card) on the right. The Sample/Convert and process system requires low-power microcontroller-based hardware-software design and the main goal of this thesis work is to implement that system.

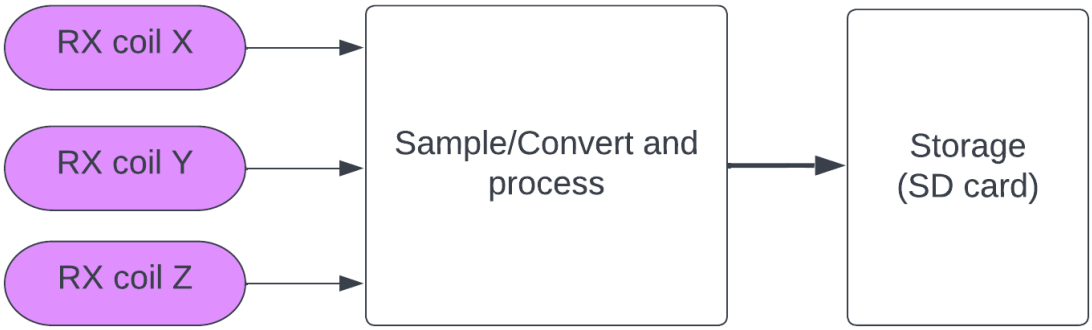

Figure 2.1: Block diagram showing the signal receiving coils on the left, and Data storage on the right.

For sampling and converting the analog signals from the three coils, two main techniques were considered:

1) sampling all three channels using one ADC IC, which has simultaneous sampling capability, or

2) sampling using three separate ADC ICs.

One of the specific requirements of the LASSITOS team was to collect higher resolution data using a 32-bit ADC IC. There were no available 32-bit simultaneous sampling ADC ICs during 2021 due to a global chip shortage. Thus, the second method was used and ADS1263 ADC IC was selected for this project. This ADC IC has a 32-bit resolution and has a maximum sampling rate of 38 kSPS.

Before sampling and converting the analog signal, it has to be filtered and amplified. For this purpose, a separate RX channel board, with two amplifier ICs and an ADS1263, was designed and built. In order to filter out the noise between channels, three separate RX channel boards were used rather than putting the boards on a single printed circuit board (PCB).

Figure 2.2 shows the block diagram of signal and data flow direction and the hardware organization. From RX coils to RX channel boards, the signal is analog. From the ADC the analog signal is then converted to a digital signal and sent to the microcontroller. The microcontroller organizes and formats sample data as required, which is sent to the SD card for storage. An SD card is selected as the data storage device due to its availability, cost-effectiveness, and compatibility with the microcontroller.

To control the ADCs and to store the data coming from the ADCs to the SD card, a microcontroller-based system is required. The ADS1263 uses SPI to communicate with a microcontroller. In this application, the microcontroller should have at least six communication channels, including:

- three SPI channels for receiving data from ADCs,
- one SPI channel for communicating with the SD card,

- one Universal Asynchronous Receiver-Transmitter (UART) channel to communicate with a serial terminal for debugging purposes, and
- at least one spare channel for potentially including additional devices.

The SSEP lab of the UAF uses a custom microcontroller development board with an MSP430F6779A microcontroller and an SD card slot that they have made available for our use on this project. After investigating the above requirements and features of this microcontroller, this development board was selected for initial development purposes.

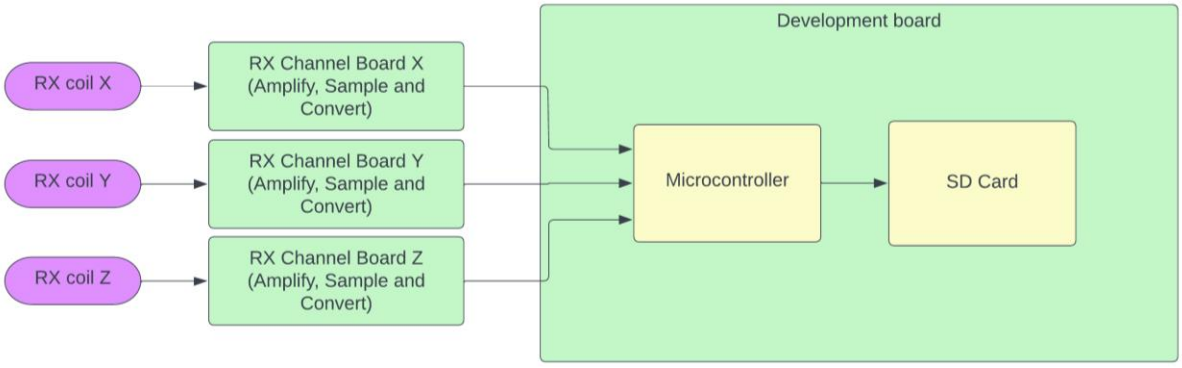

Figure 2.2: Initial system development block diagram. The green color blocks represent individual PCBs.

2.1 RX Channel Board

The RX Channel Board consists of three sections. On the signal receiving end of the board, a low-noise op-amp IC, LTC6244 [12] is used as the first stage amplifier that sends the amplified signals to a differential op-amp IC, LTC6362 [13] for second stage amplification. After amplification, the signal is provided to the ADC to sample and convert the analog signal to a digital format.

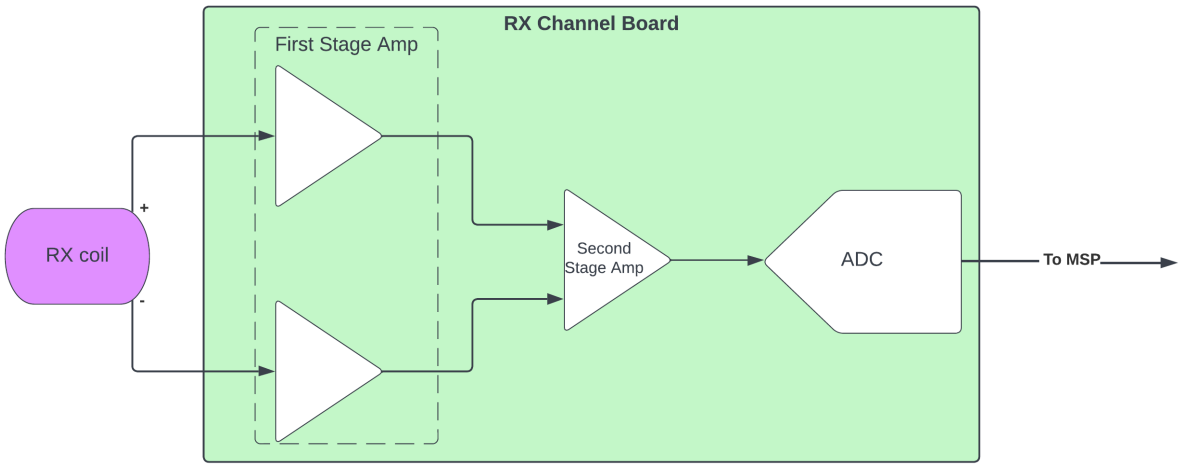

Figure 2.3: Functional block diagram and the signal path of the RX channel board.

Figure 2.3 shows the functional block diagram of the RX channel board and the signal flow direction. The electromagnetic coil has two terminals. The signals coming from these terminals are amplified individually by the LTC6244. The output of these two signals is then amplified using the differential amplifier LTC6362 in the second stage. Figure 2.4 shows the completed PCB of the RX channel board, first and second stage amplification, and the ADC section of the PCB.

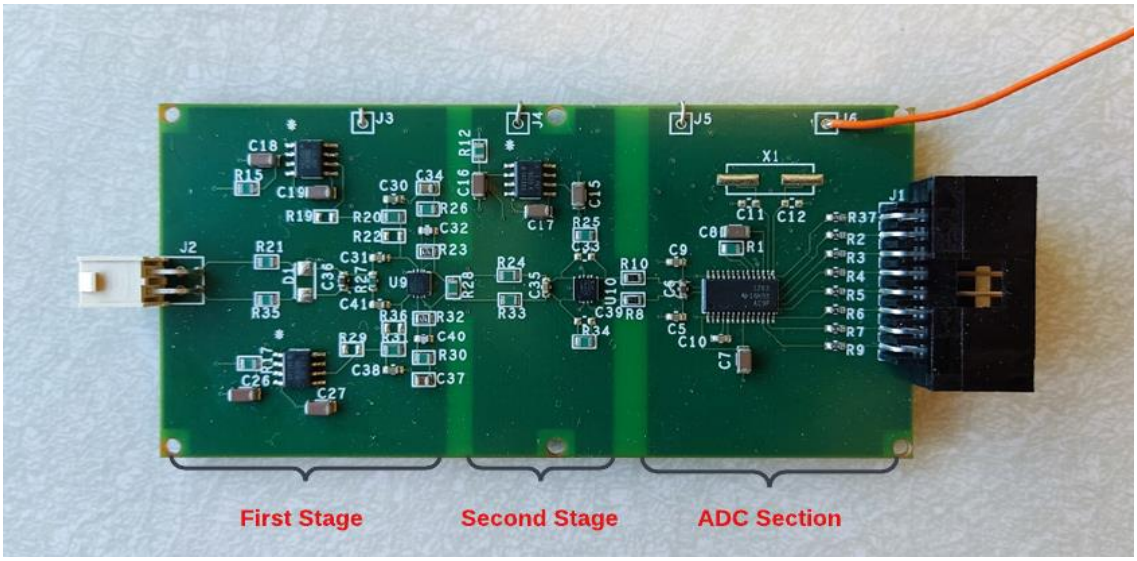

Figure 2.4: Completed PCB of the RX channel board.

Figure 2.5 shows the functional block diagram of the ADS1263. This ADC IC has space for eleven analog inputs; however, only two signals, received by pins $AIN0$ and $AIN1$, are produced by the second-stage amplifier. The input multiplexer (MUX) selects these two signals, and then provides them to the programmable-gain amplifier (PGA).

The PGA amplified signal is then converted to a 32-bit digital signal using the Delta-Sigma ADC. After the conversion, the signal is sent to a digital filter for decimation and held until the microcontroller is ready to capture the data by communicating with the serial interface of the ADC IC.

The ADS1263 requires two voltage sources: 5 V for $AVDD$ for the analog section of the ADC and 3.3 V for the digital functionality of the ADC IC. The ADS1263 requires a 7.3728 MHz clock signal for conversion, which can be provided by three different methods:

- Internal oscillator,
- External crystal oscillator, or
- External clock.

If nothing is connected to the $XTAL1$ and $XTAL2$ pins of the IC, the internal oscillator can be used as the clock source. Initially, the RX channel board was designed using an external crystal oscillator. This was modified to accommodate a need for the synchronization of all three ADCs using a common clock, so in the latest design, an external clock is provided by a benchtop waveform generator for testing purposes. For field deployment in a UAS, an embedded clock source will need to be added to the instrument. This will be done in the next revision of the board.

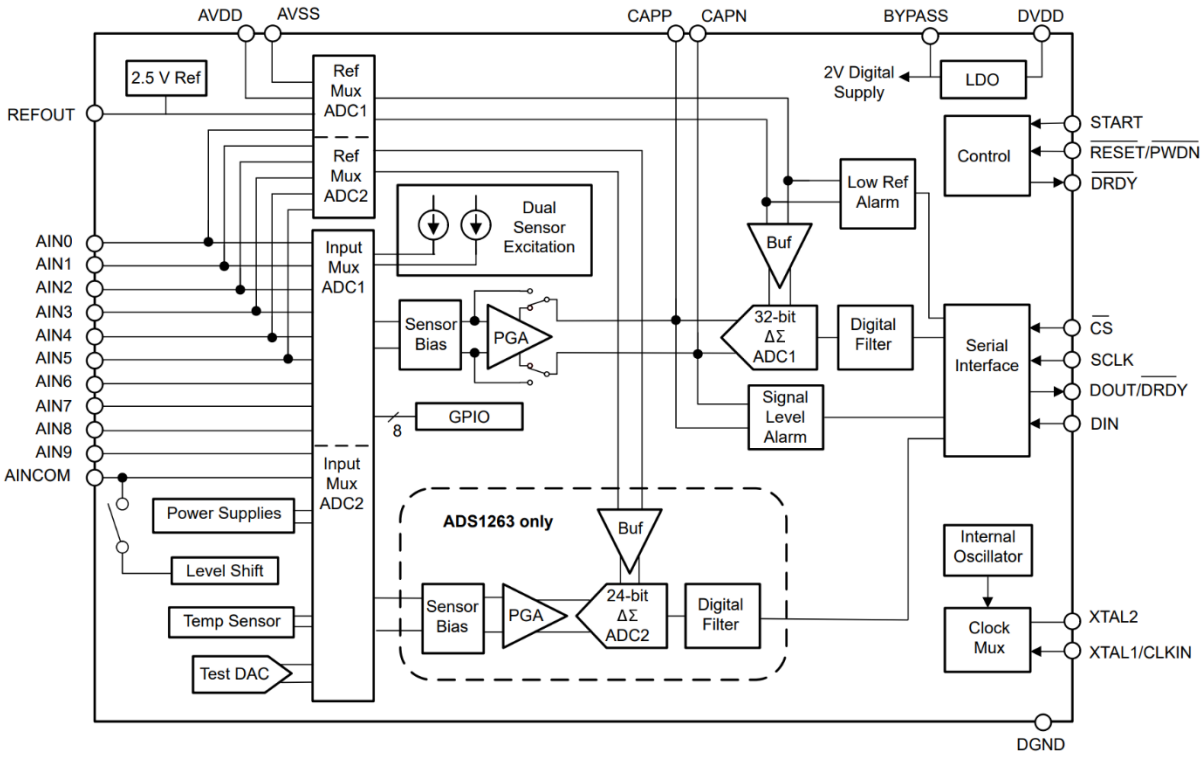

Figure 2.5: Functional block diagram of the ADS1263 [4] (Courtesy of Texas Instruments).

The 32-bit ADC Modulator is a pipelined Delta-Sigma modulator. It samples the analog input signal at a high sample rate (921.6 kHz), which is then filtered and decimated to achieve the final sample rate.

The serial interface is compatible with the SPI bus and consists of four signals: $\overline{CS}$, $SCLK$, $DOUT/\overline{DRDY}$, and DIN. This is used to read the converted data and control and program the ADC.

ADC1 conversions can be controlled by the $START$ pin or by sending commands through a serial bus. *ADC1* operates as a continuous conversion until stopped by pulling the $START$ signal down or using the STOP command. The pulse conversion mode only performs one conversion when it receives the start signal or command. In this project, continuous sampling mode is used.

The 32-bit ADC of the ADS1263 supports sample rates of 2.5, 5, 10, 16.6, 20, 50, 60, 100, 400, 1200, 2400, 4800, 7200, 14400, 19200 and 38400 SPS. The intended EM frequency of the EM subsystem is 1-9 kHz. According to Nyquist Sampling Theorem, the sampling frequency needs to be at least twice the frequency of the EM subsystem, resulting in a minimum sample rate of 18 kHz. As the sampling rate rises, the noise free bits of the ADC decreases, so 19200 SPS is selected as the minimal feasible sampling rate for this project.

The $\overline{DRDY}$ pin indicates when the *ADC1* conversion data is ready for retrieval by transitioning to low. The software polling is also possible to detect new conversion data, which is done by reading the status register. In this project, only the $\overline{DRDY}$ pin (hardware polling) is used to detect the availability of new conversion data.

The reading of conversion data can be done using one of two methods: *read data direct* or *read by command*. After the conversion, *ADC1* data is written to an output shift register and an internal data holding register.

When using the *read data direct* method, the sample data can be read from the $DOUT/\overline{\text{DRDY}}$ pin. The *ADC1* data field consists of an optional status-byte, four bytes of conversion data, and an optional checksum byte. These optional bytes can be enabled or disabled by configuring the registers of the ADS1263.

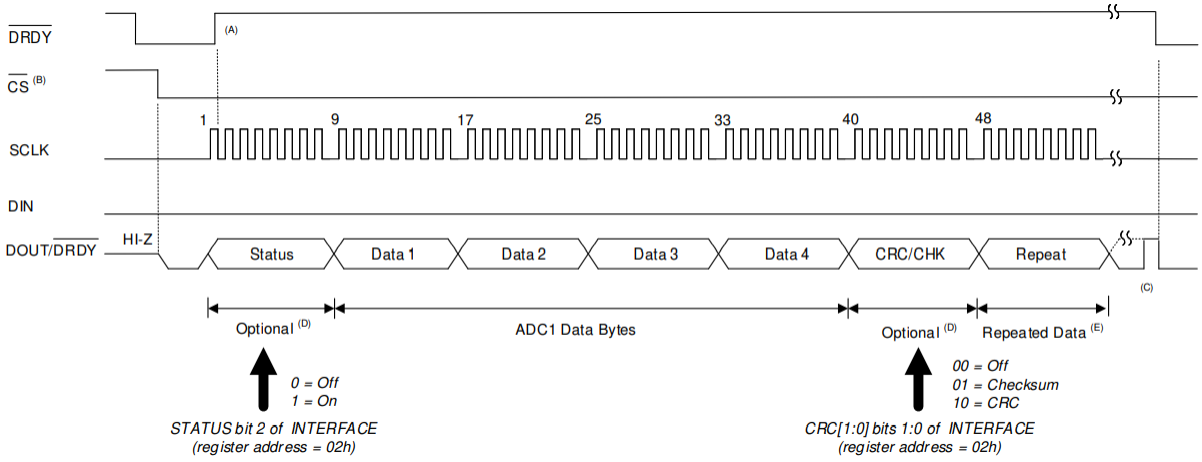

Figure 2.6: Read data direct method data sequence [4] (Courtesy of Texas Instruments).

Figure 2.6 shows the *read data direct* method data sequence. The $\overline{DRDY}$ pin can be used to detect whether data is available to read. In this project, the Status byte and Checksum byte are disabled to avoid the additional time required in reading the data. As $SCLK$, DIN, and $DOUT/\overline{DRDY}$ pins are SPI pins, by sending four bytes with all bits zero this data can be read. This method only required 32 clock signals for $SCLK$.

If *read data by command* is used, the opcode would need to be sent and the *ADC1* data would be read from the data holding register. Figure 2.7 shows the *read data by command* sequence.

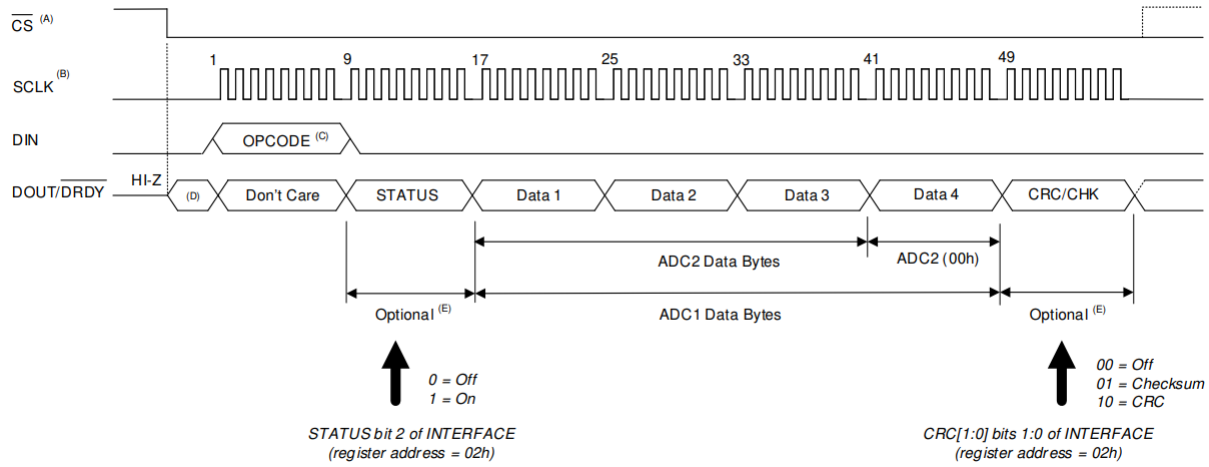

Figure 2.7: Read data by command sequence [4] (Courtesy of Texas Instruments).

This method would add an extra byte for data reception operation. Therefore, the *read data direct* method is used in this project to minimize the time required for capturing data.

The *ADC1* data is in 2's-complement format and represents positive and negative values. *ADC1* outputs the most significant bit (MSB) first. Table 2.1 shows the input signal and the corresponding output 32-bit code of *ADC1*.

Table 2.1: Input signal and corresponding output hexadecimal code [4].

Input signal (V)	***ADC1* OUTPUT Code (32 Bits)**
$\geq \frac{V_{REF}}{Gain} \times \frac{(2^{n-1}-1)}{2^{n-1}}$	0x7FFFFFFF
$\frac{V_{REF}}{Gain \times 2^{n-1}}$	0x00000001
0	0x00000000
$\frac{-V_{REF}}{Gain \times 2^{n-1}}$	0xFFFFFFFF
$\leq \frac{-V_{REF}}{Gain}$	0x80000000

The calibration of all three ADCs is essential in sampling data. Calibration can be done either by sending calibration commands or by direct user calibration. In user calibration, the user must calculate the proper values and write them to the calibration registers. The ADC can perform offset calibration itself. This is the only calibration method used in this project during the test/development phase.

When performing the offset self-calibration, the *ADC1* offset error is corrected. Before performing this calibration, all analog input connections need to be forced open. When the ADC receives the self-calibration command, it shorts the inputs of the internal PGA and averages sixteen readings to reduce the conversion noise for accurate calibration. After the calibration is done, the result is then written to offset the calibration registers and the input multiplexer needs to be reconfigured to receive proper input signals. This offset value is subtracted from each sample when it performs the sample conversion. The complete self-offset calibration process consists of the following steps.

1) Enable continuous-conversion mode.
2) Select desired gain and reference voltage.
3) Program the *ADC1* input multiplexer register to 0xFF to open all inputs before sending the calibration command.
4) Start conversion by setting the start pin high.
5) Send the self-offset calibration command.

After the calibration has been completed, the $\overline{DRDY}$ pin is driven high, and this can be used to detect the completion of the calibration. For the sample rate 19200 SPS, the calibration time is 1.490 ms.

One SPI channel is used for all three ADS1263s with two DMA channels as shown in Figure 2.8. Chip select signals are used to activate each ADC's SPI. The main reason to use one SPI channel was to reduce the overall data receiving time. As all three ADCs share the same SPI channel, each ADCs uses the chip-select signal to communicate with the microcontroller.

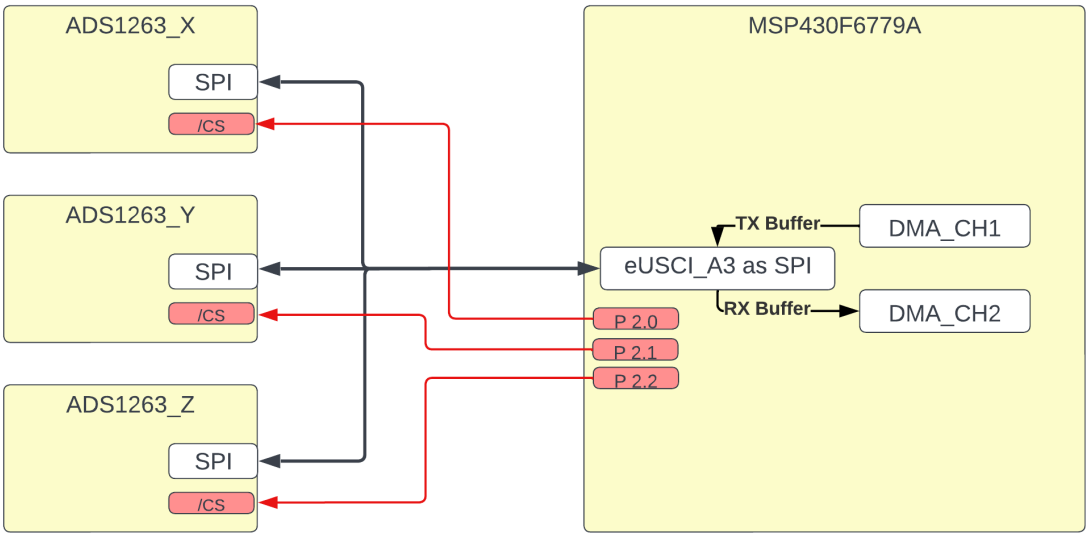

Figure 2.8: Using one SPI channel for all three ADCs.

2.2 Microcontroller and Development Board

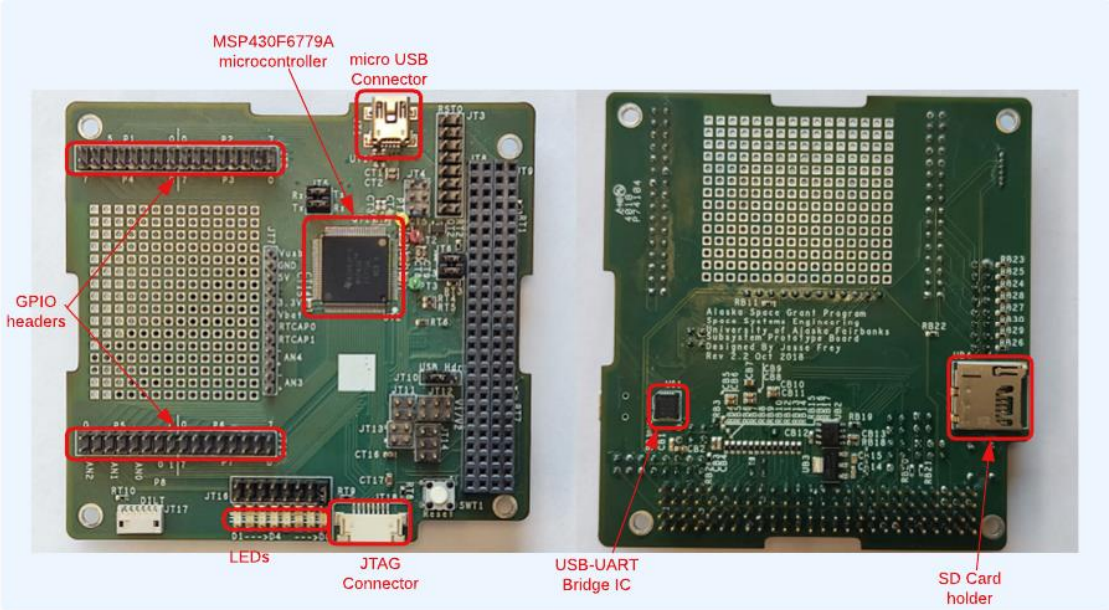

Figure 2.9: The front (left) and back (right) sides of the development board with named components.

The specific microcontroller used in this project is a Texas Instrument's MSP430F6779A microcontroller, which is used on the SSEP development board, as shown in Figure 2.9. It consists of a microcontroller, mini-Universal Serial Bus (USB) connector, USB-UART bridge IC, SD card holder, Joint Test Action Group (JTAG) connector, 8 Light Emitting Diodes, and header connectors with General Purpose Input-Output (GPIO) and power. Figure 2.10 shows the functional block diagram of this microcontroller.

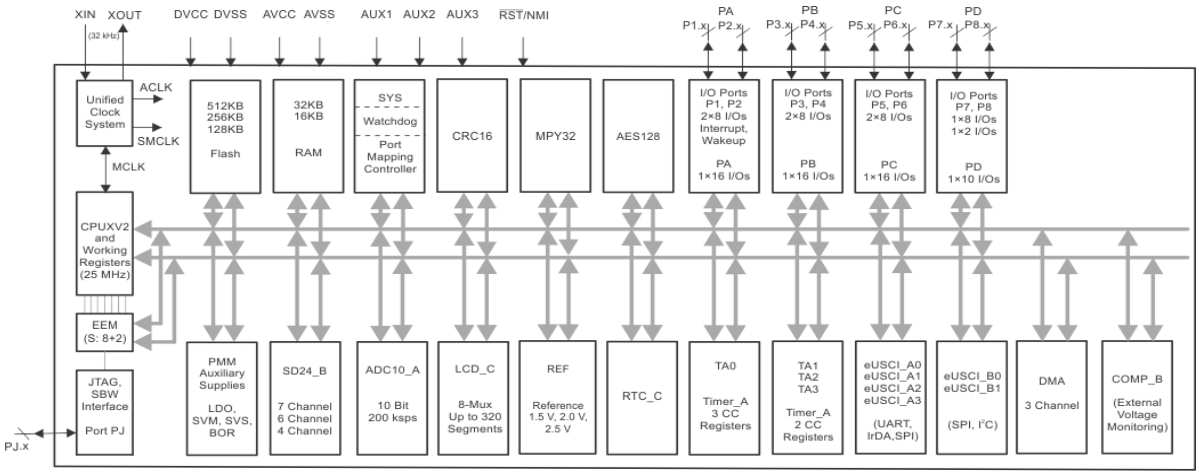

Figure 2.10: Functional block diagram of the MSP430F67799A microcontroller [5] (Courtesy of Texas Instruments).

The key features of this microcontroller include and used in this project:

- 32 KB Random Access Memory (RAM) - Most of the RAM of this microcontroller is used to buffer the data sent to the SD card. The SD cards have a random resting time. During this resting period, The RAM was used to buffer the data coming from the ADCs;
- 3 DMA channels - All three DMA channels are used in this project. Two of them are used to capture sample data from the ADCs and the other DMA channel is used to send organized sample data to the SD card;
- 6 Universal Serial Communication Interface (USCI) modules - USCI modules of the MSP430F6779A are mainly divided into two sections A and B. There are four USCI A modules and all four are capable of SPI, IrDA, and UART. There are two USCI B modules and those two modules are capable of only SPI and Inter-Integrated Circuit (I^2C). One USCI A SPI is used for all three ADCs. One USCI module is used as UART, and it is used mainly for debugging and issuing control commands to the microcontroller;

- 1 Unified Clock System (UCS) module - The UCS module is used to generate different clock speeds for testing the system;
- Input/Output (I/O) ports with port-mapping - The port mapping feature is used to map each module's signals to relevant I/O pins of the microcontroller; and
- 4 Timer/Counters - One of the timer/counters used to generate the required external clock for the ADCs.

The development board has a micro-SD card slot connected to the microcontroller through one chip select pin and three SPI bus pins. All the microcontroller GPIO pins have male-header pin connectors on the development board, enabling the connection of other external devices. Other connectors include:

- a JTAG connector, for programming the microcontroller and debugging purposes, and
- a mini-USB connector to connect the microcontroller to the computer. A USB-UART bridge IC works as a translator between the microcontroller and the USB connector.

2.3 Final Hardware Configuration

The final hardware configuration used in this thesis is shown in Figure 2.11. Each green block represents a PCB, and each light-yellow color block represents an IC. Power for the development board can be supplied either using a USB connector or a JTAG connector. In the development phase, both of these connections are used.

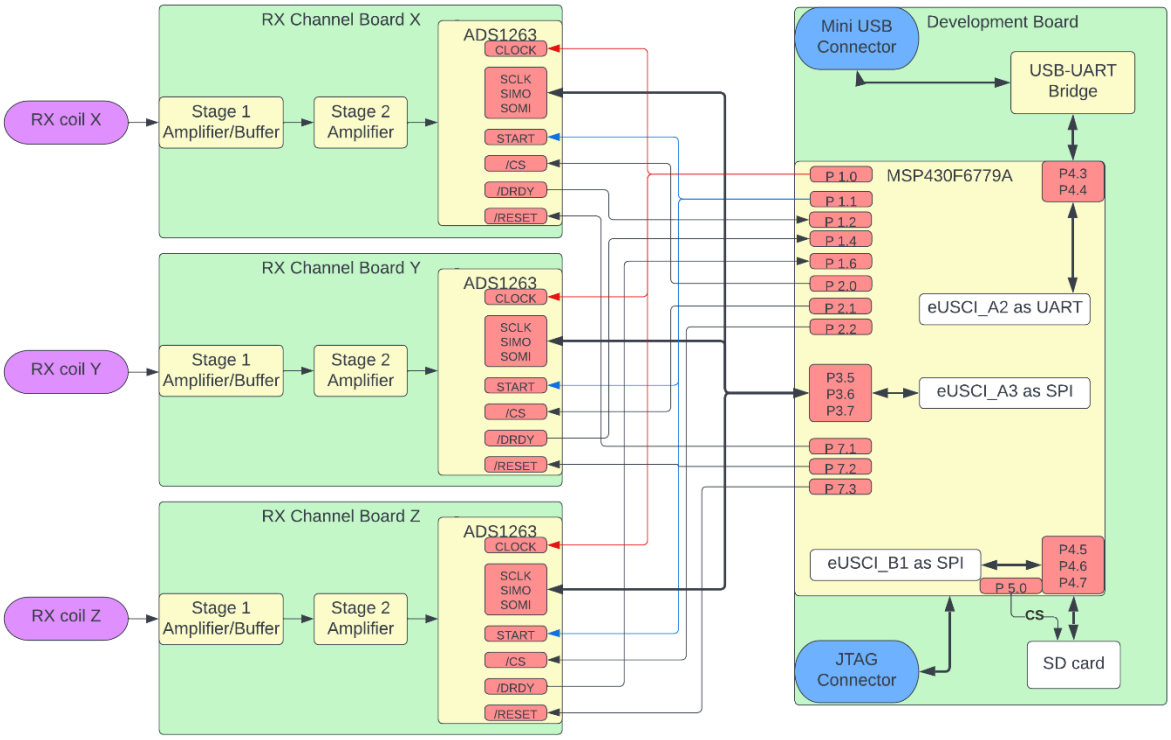

Figure 2.11: Final hardware configuration block diagram.

Chapter 3 Microcontroller Software Design

3.1 Introduction

This chapter discusses the software that runs on the MSP430F6779A (MSP) on the development board. The MSP was developed by Texas Instruments. They also developed the Code Composer Studio (CCS) integrated development environment (IDE), which was used to develop the software for the MSP using the C programming language. The debugging capability provided by CCS was used extensively in developing this program.

The software runs on the MSP and controls, communicates with, and configures the ADCs and the SD card. After downloading the program and device configuration, the device is ready for the data logging process.

The necessary software and device configuration settings are summarized below:

1. Initialize processes (3.3)
    - Setup core voltage and frequency of the MSP (3.3.1)
    - Setup SD card (3.3.2)
    - Setup communication with and configure each ADC (3.3.3)
    - Configure DMA channels to optimize data transfer (3.3.4)
    - Setup communication with a serial terminal of a computer through UART (3.3.5)
2. Data logging processes (3.4)
    - Get sample data from the ADCs using DMA (3.4.1)
    - Send the organized data to the SD card (3.4.2)

3.2 Definitions of Data Organization

Data collected by the ADCs are organized in the MSP Data Buffer for transmittal to and storage on the SD Card in units referred to as samples, packets, and frames, as described in Figure 3.1.

- ADC Sample (4 bytes) - a single sample produced from one ADC.
- Sample Packet (16 bytes) – consists of a 4-byte sample number and three simultaneous ADC Samples, one from each ADC.
- Data Frame (512 bytes) - consists of 32 Sample Packets. Data is ready to be transferred to the SD card when a frame is fully assembled. Data Frames are sized at 512 bytes so that one Data Frame fully occupies one sector on the SD Card.
- Data Buffer (30 KB) - The Data Buffer is a circular buffer large enough to store up to 60 Data Frames.

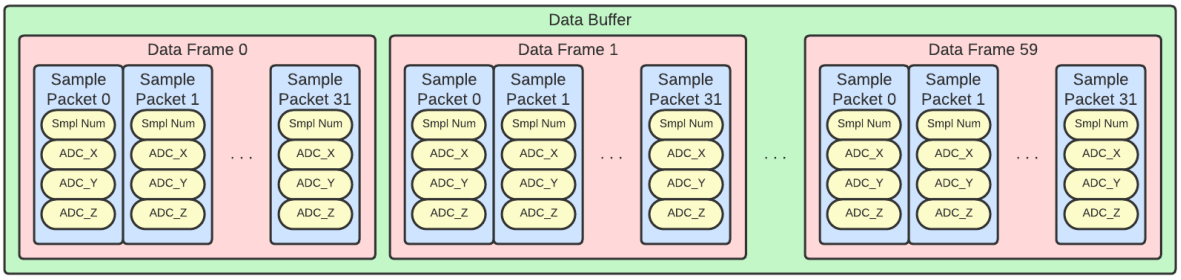

Figure 3.1: Structure of the Data Buffer. ADC_X, ADC_Y and ADC_Z represent ADC samples from each ADCs.

3.3 Initialization

When the MSP430 is powered on, the peripherals in the MSP are in a default configuration and most of them are not enabled to save power. Also, the SD card and the ADCs need to be initialized for proper functionality. This section describes important configuration changes done

to certain modules in the microcontroller and the initialization processes of the SD card and ADCs.

3.3.1 Core Voltage and Frequency Setup

The MSP can operate at four core voltage levels. The MSP powers on at the lowest possible core voltage (level 0, 1.41 V). At this power level the MSP is limited to operating at lower frequencies. As a requirement of this project, the MSP needs to operate at a 20 MHz frequency, requiring that the core voltage be increased to the highest level (level 3, 1.91 V) At this higher power level, the MSP is capable of operating at frequencies up to 25 MHz.

The central processing unit of the MSP uses the Main Clock (MCLK), while the other peripherals of the MSP use Sub-Main Clock (SMCLK) for their operation. When the MSP powers on, these clocks run at 1.048576 MHz. This clock speed is increased to 20 MHz for the purpose of this project. These clocks are controlled by the UCS module of the MSP, and the digitally controlled oscillator of this module changes those frequencies to the required value.

3.3.2 SD Card Setup

The SD card is connected to the MSP through an SPI channel. Thus, the MSP needs to allocate a USCI module to communicate with the SD card. Texas Instruments provides MSP430WARE [14] which is a resource collection that helps to build code for MSP430 microcontrollers. One of the examples in MSP430WARE provides a library for MSP430F5529 to communicate with SD card while maintaining File Allocation Table (FAT) file system. This library contains three sub-libraries:

- FATFS
- MMC
- HAL_SD

For this project, the FAT structure was removed by removing the FATFS sub-library in order to save raw data without a file structure, optimizing data transfer speed. Figure 3.2 shows the structure of the modified library (SD card driver) and the interaction between the hardware level and the main program.

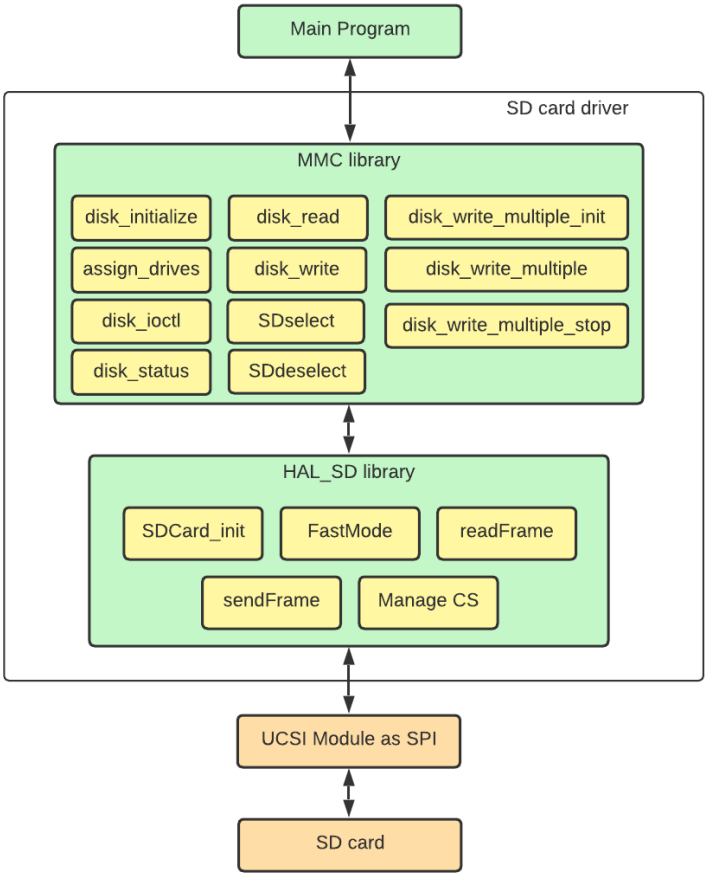

Figure 3.2: SD card driver structure.

The MMC and HAL_SD sub-libraries were modified in order to adopt the SD card driver to the MSP430F6779A and to optimize the communication with the SD card.

3.3.2.1 MMC

The MMC library contains the functions for SD card operation. The main program calls these functions to initialize and communicate with the SD card. When initialization or data writing happens, this library breaks down those functions into subtasks. For example, when the main program calls the *disk_initialize* function, several subtasks occur:

1. The USCI module needs to be configured,
2. A Sequence of commands needs to be sent at a low data rate, and
3. If the SD card responds appropriately to the above commands, the data transfer rate increases to full speed.

The *disk_write* function, available in the MMC library, can be used to write a known number of Data Frames to the SD card. In this project, as the entire sampling time varies, the exact number of Data Frames is unpredictable. Writing a single Data Frame at a time is possible with this function, but it introduces a latency to the SD card data writing process.

To optimize this data writing speed to the SD card, three new functions have been added to the MMC library:

- *disk_write_multiple_init* - initializes the writing multiple frames process by sending to the SD Card the *write multiple frame* command along with the sector number where the first Data Frame is to be stored,
- *disk_write_multiple* - sends Data Frame to the SD card, and
- *disk_write_multiple_stop* - stops the multiple frame writing process by sending the *stop_transmission* token to the SD card.

3.3.2.2 HAL_SD

The HAL_SD library provides MSP-specific functions to communicate with the SD card. This library contains the instructions to the MSP for the USCI module and the GPIO configuration for SPI bus initialization. The USCI module initializes at 400 kbps because this lower speed is required by the SD card to receive the first commands through the SPI bus. If the initialization is successful, it will be transferred to the fast mode by calling the *FastMode* function. The original *FastMode* function is designed to set the SD card communication rate to half of the SMCLK. This function has been modified to set the communication rate to the full speed of the SMCLK, which allows communication between the MSP and SD card at 20 Mbps, as the SMCLK is at 20 MHz.

When the MMC needs to send a command or data to the SD card, it usually contains multiple bytes. However, the USCI module accepts only one byte at a time. The read/write functions in the HAL_SD library resolve this problem by sending the command or data to the USCI module one byte at a time.

The HAL_SD library is not designed to use the DMA controller for data transfer. To optimize sending data to the SD card, the *sendFrame* function of the HAL_SD library is rewritten to work with one of the DMA channels. The *sendFrame* function accepts two parameters: the address of the data that needs to be sent to the SD card and the size of the data in bytes. Using these two parameters, DMA channel 2 is configured. The code section in Figure 3.3 shows the rewritten *sendFrame* function.

```
void SDCard_sendFrame (uint8_t *pBuffer, uint16_t size)
{
          DMA2SA = (unsigned short *)pBuffer; /* set DMA source address */
          DMA2SZ = size;                      /* set transfer size */

             /* Configure the DMA transfer*/
          DMA2CTL =
             DMAREQ  |                        /* start transfer */
             DMADT_0 |                        /* Single transfer mode */
             DMASBDB |                        /* Byte mode */
             DMAEN |                          /* Enable DMA */
             DMALEVEL |                       /* Level trigger */
             DMASRCINCR1 | DMASRCINCR0;       /* Increment source address */
}
```

Figure 3.3: *sendFrame* function code section.

3.3.3 ADC Setup

The ADCs have multiple connections to the MSP for communication and control purposes. These links need to be configured by the MSP to establish a connection with each ADC. Also, each ADC needs to be configured with proper settings before sampling begins. This section discusses these configurations.

All three ADCs require a clock signal with a frequency of 7.3728 MHz. In order to eliminate clock drift between each of the ADCs, the same clock signal is provided to all three ADCs by a benchtop waveform generator while performing the lab testing, although an embedded clock source will need to be included in the final design. This approach synchronizes ADC sampling and produces simultaneous $\overline{DRDY}$ signals.

3.3.3.1 MSP Configuration of ADCs

One ADC has seven Control/Data connections to the MSP. Table 3.1 shows these connections with their usage and the input/output configuration on the MSP side.

Table 3.1: ADC control and data signal connections.

Connection		Usage	Shared with all ADCs	MSP430 Input/Output
SPI	$SIMO$	Data transfer	Yes	Output
	$SOMI$			Input
	$SCLK$			Output
$\overline{CS}$		SPI Chip Select	No	Output
$\overline{RESET}$		Reset the ADC	No	Output
$\overline{DRDY}$		Data ready signal	No	Input
$START$		Start Sampling	Yes	Output

SPI and $START$ connections are shared among all three ADCs and only require four GPIO pins. The other connections need separate pins for each ADC, requiring nine more pins. For the SPI channel, the USCI_A3 module of the MSP connections are port mapped to the required pins. The rest of the pins are configured for input or output as shown in Table 3.1. Because a single SPI channel is used for all three ADCs, $\overline{CS}$ signals are used to enable communication with each ADC. The interrupts for all the $\overline{DRDY}$ pins have been enabled, and interrupts trigger when the pin transitions from high to low. All $\overline{DRDY}$ pins are connected to Port 1, and this allows the use of a single Port1_ISR for all three interrupts. This PORT1_ISR is responsible for receiving the data from each ADC when all three $\overline{DRDY}$ interrupt flags are raised.

3.3.3.2 ADC Register Configurations

Before starting sampling, ADCs are configured to have proper functionality by configuring the ADC internal registers. The MSP sends all configuration data, other than calibration data, which is internally generated, to each ADC during the initialization process including:

- select reference voltage (internal reference),
- disable status byte and checksum bytes,

- select input channels (channel 0 and 1), and
- select sampling frequency (19200 SPS).

Self-calibration of each ADC, to minimize offset error, is performed one after another. Before starting the calibration process, the input signals to the ADC are forced open using the input MUX. Continuous sampling, for the purpose of calibration, begins by raising the $START$ signal followed by sending the *Offset Self-Calibration* command to the ADC.

When calibration starts, the $\overline{DRDY}$ pin is driven high and returns to low when the calibration is finished. Subsequently, sampling is stopped by lowering the $START$ signal. During this period, the $\overline{CS}$ and $SCLK$ signals are kept low as required by the ADC, and the $\overline{DRDY}$ interrupts are kept disabled in the MSP to avoid unnecessary PORT_1_ISR triggers.

3.3.4 DMA Setup

DMA transfers operate independently from the CPU. After proper configuration, the DMA transfers a pre-configured number of bytes from one memory location to another, while the CPU can perform another task or stay in a low power state. The MSP has three DMA channels, all of which are used in this project. Table 3.2 shows the configurations for each DMA channels. By default, DMA channels are disabled until there is a data transfer requirement, at which point the relevant channel is enabled by the software. A trigger is required to transfer each byte on the enabled channel. After transferring the expected number of bytes, the channel is automatically disabled.

Table 3.2: DMA Channel Configuration.

DMA Channel	Trigger source	Transfer size (bytes)	Source Address	Destination Address
Channel 0 (DMA_RX)	ADC USCI Receive Buffer Interrupt Flag	12	ADC SPI Receive Buffer	Relevant Sample Packet
Channel 1 (DMA_TX)	ADC USCI Transmit Buffer Interrupt Flag	4	Dummy Data	ADC SPI Transmit Buffer
Channel 2 (DMA_SD)	SD USCI Transmit Buffer Interrupt Flag	Variable length	Variable pointer	SD SPI Transmit Buffer

Channel 0 (DMA_RX) is responsible for receiving the ADC sample, one byte at a time, through the SPI and placing it into the Sample Packet. When the SPI Receive Buffer receives a byte from an ADC, the Receive Buffer Interrupt Flag is raised in the USCI module and DMA_RX uses this flag as a trigger signal. A total of four bytes are received from each ADC (ADC_X, ADC_Y, ADC_Z), resulting in a total transfer size to DMA_RX of 12 bytes.

Channel 1 (DMA_TX) sends Dummy Data to each ADC to cause the ADC to return sample data. To send Dummy Data to all ADCs, DMA_TX is enabled three times, sending data to each ADC sequentially. When the USCI Transmit Buffer is free, it raises the Transmit Buffer Interrupt Flag on the USCI module. DMA_TX uses this flag as a trigger signal to send a single dummy byte to the Transmit Buffer in the USCI module.

Channel 2 (DMA_SD) sends data and commands to the SD Card. The transfer size and source address of this data is variable, depending on whether sending commands or data, so the *sendFrame* function includes parameters that specify the source address and transfer size, as shown in the code section in Figure 3.3. The DMA_SD uses the SD USCI Transmit Buffer

Interrupt Flag as the trigger signal to send individual data or command bytes to the Transmit Buffer when enabled.

3.3.5 UART Setup

To establish communication between the MSP and a computer, a UART connection is used, specifically a USCI A2 module configured to use a 115200 bps baud rate. This UART connection provides control commands to the MSP. These control commands trigger the following tasks:

- start sampling,
- stop sampling, and
- supplemental calibration of ADCs.

To use the UART channel, the MSP is configured for the receiver interrupt to be enabled. Control commands are sent through the UART channel by sending a character that corresponds to the desired task. Reception of this byte triggers the USCI_A2_ISR to perform the specified task.

3.4 Data Logging Process

When the MSP receives a start sampling command through UART, the *disk_write_multiple_start* function is called to prepare the SD card for multiple sample writing and to reserve the first sector for metadata, allowing frames to start writing to the next available sectors of the SD card. If the function returns a success status, then the start signal is given by driving the ADC START pins high. After the ADCs start the sampling process, the variable called “Running” is set to “true” to indicate that sampling has begun.

In the data logging process, the instruction to collect sample data from ADCs resides in the PORT_1_ISR. The instructions for saving the data to the SD card reside in the main loop which is a non-ending while-loop in the main function.

There are three pointers used in the program to indicate different memory locations for data organization:

- ADC Frame Pointer - points to the Data Frame that is actively receiving ADC data,
- SD Frame Pointer - points to the Data Frame that will next be sent or is currently being sent to the SD card, and
- Sample Packet Pointer - points to the Sample Packet currently being filled.

During sampling, the main loop continually checks for a new Data Frame, which is sent to the SD card when available. The $\overline{DRDY}$ interrupt occurs upon reception of the $\overline{DRDY}$ signal from an ADC. The interrupt pauses the main loop and the PORT_1_ISR starts running in order for ADC sampling to begin. The ADCs are synchronized so that when the PORT_1_ISR checks for DRDY signals, all three will be available. After collecting samples, PORT_1_ISR exits, and the main loop continues where it stopped when the interrupt was triggered. The next sections describe this process in more detail.

3.4.1 Collecting Sample Data From ADCs

PORT_1_ISR is responsible for collecting data from the ADCs and organizing the data into Sample Packets and Data Frames. The Sample Packet number is a 32-bit unsigned integer variable that keeps track of the number of samples received and increases each time the sample-receiving process occurs. This sample number is written to the Sample Packet currently being assembled, followed by 12 bytes of ADC sample data using the process shown in Figure 3.4.

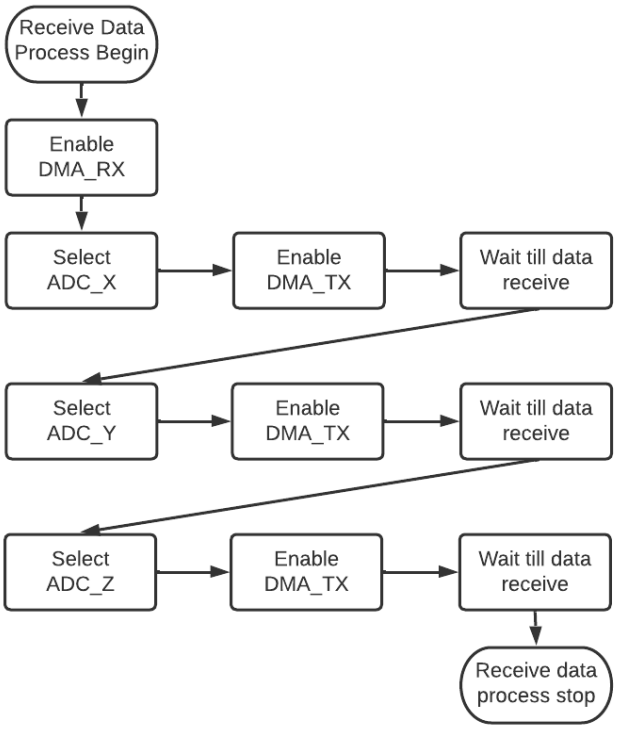

Figure 3.4: Process to Retrieve ADC Samples.

After receiving sample data from all three ADCs, the Sample Packet Pointer advances to the next location so the new sample data can be received when available. After 32 Sample Packets are received, filling the ADC Data Frame, the ADC Frame Pointer moves to the next Data Frame on the Data Buffer as shown in Figure 3.5. Another 32-bit unsigned integer named “ADC_Frames” keeps track of the number of Data Frames assembled.

This new ADC Data Frame can receive another 32 samples. After filling 60 Data Frames in the circular Data Buffer, data begins over-writing itself and the ADC Frame Pointer continues advancing with new data.

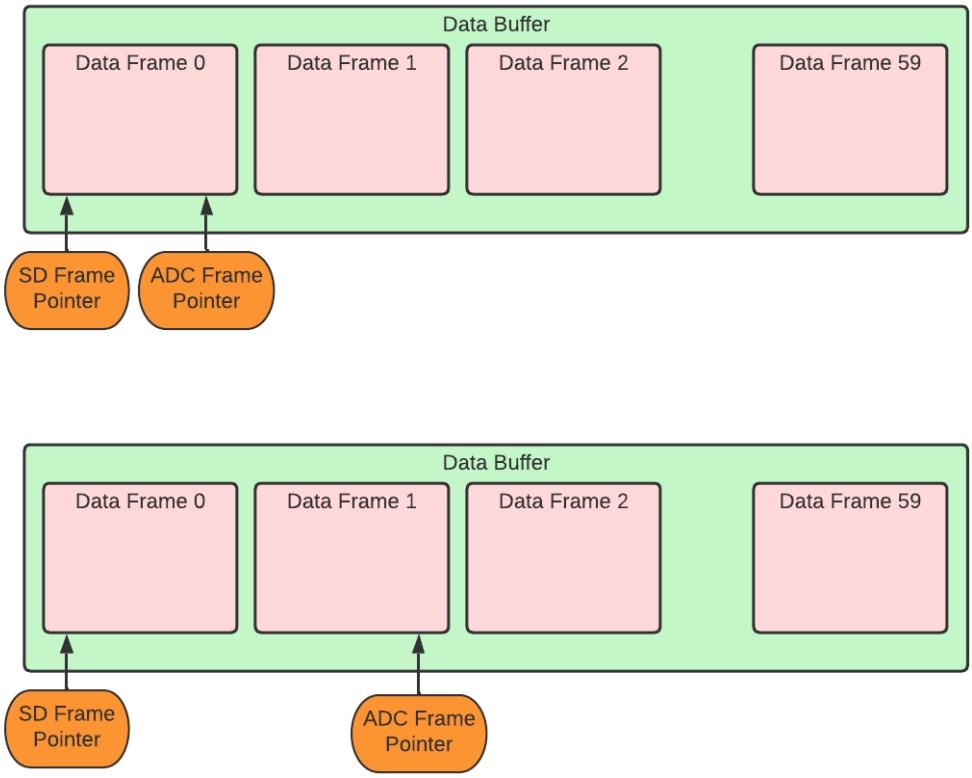

Figure 3.5: Advancement of ADC Frame Pointer.

3.4.2 Writing Data to the SD Card

Instructions to save data to the SD card reside in the main loop of the main function. The main loop compares the ADC Frame Pointer and the SD Frame Pointer to check whether new data is available to write to the SD card. If both pointers are not pointing to the same Data Frame, this indicates that the SD Frame Pointer is pointing at the Data Frame available to write to the SD card. This Data Frame is sent to the SD card using the *disk_write_multiple* function. If data writing is successful, an unsigned integer variable named “SD_Frames” is incremented to record the number of Data Frames transferred to the SD card. The SD Frame Pointer is then advanced, as shown in Figure 3.6, and the main loop repeats. The SD Frame Pointer continues advancing while writing frames to the SD card until both pointers point to the same Data Frame.

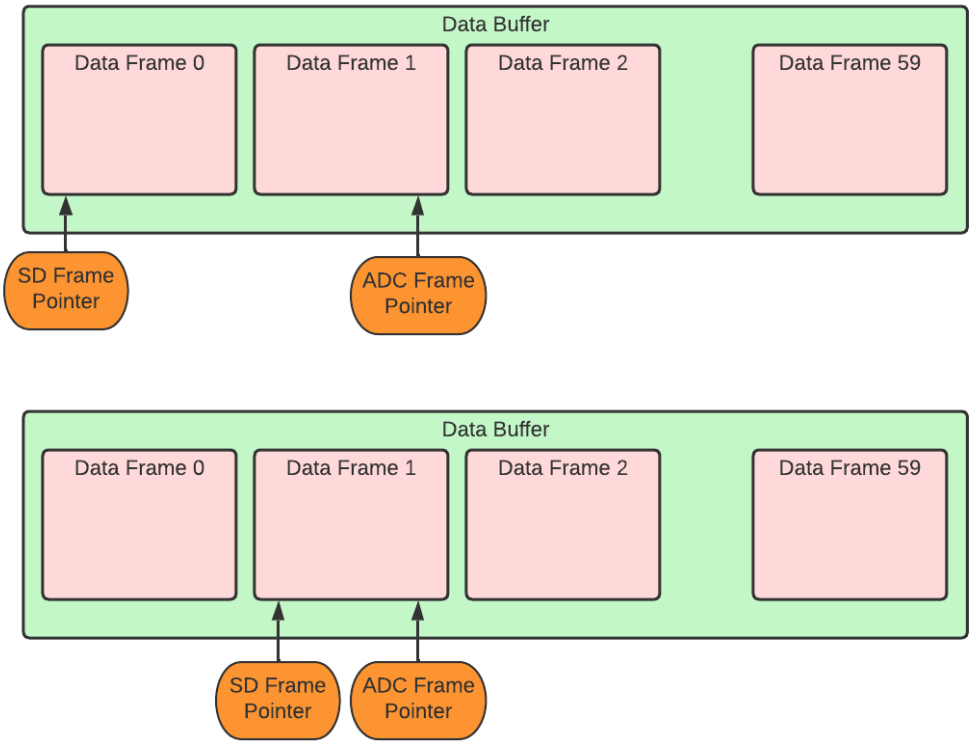

Figure 3.6: Advancement of SD Frame Pointer.

3.4.3 Ending the Sampling Process

The software of the MSP continues sampling and storing ADC data until the main routine is interrupted. When the Stop Sampling task is triggered through the UART, the UART_ISR changes the “Running” variable to “false” to indicate that sampling should cease. If this occurs while the Data Frame currently being assembled is only partially full, the PORT_1_ISR will continue running until the Data Frame is full. The PORT_1_ISR lowers the $START$ pins to stop ADC sampling, then returns to the main loop.

In the main loop, after the last Data Frame is sent to the SD card, it checks for the “Running” variable and, if it is “false”, the *disk_write_multiple_stop* function is called to stop the data

sending process. After this has occurred, the first sector of the SD card is written with metadata of the sampling process. This data includes:

- the first sector that has data available in the SD card (Start Sector),
- the number of Data Frames that was assembled by the PORT_1_ISR (ADC_Frames), and
- the number of Data Frames that was written to the SD card (SD_Frames).

Each of these values is four bytes long and is stored in the first twelve bytes of the first sector of the SD card. This information is important when data is retrieved for later analysis.

Other details of the implementation can be found in the comments of the complete code provided in the appendix.

Chapter 4 Results and Discussion

As the full LASSITOS system is still under development, sea ice measurements in the field have not yet occurred. However, to verify the system is functioning as designed, a series of lab tests were undertaken to evaluate system performance. This chapter discusses these test results, including analysis of waveforms and sampled data on the SD card. While obtaining these results, the input connections of the RX channel board were kept open-circuited.

To control the sampling process, the MSP received control commands through UART. For the lab tests, these control commands have been sent from a computer. To send these control commands Processing IDE [15] was used and a processing script was implemented. The processing script first sends the START command, waits the amount of time needed to collect a sample, and sends a STOP command through UART.

4.1 Analyzing Waveforms

This section presents the analysis of a set of digital waveforms obtained from an oscilloscope.

Figure 4.1 shows typical behavior while sampling is in progress for a period of 500 µs. The waveforms in this figure are:

- ADC_SCLK - ADC SPI clock signal ($SCLK$);
- Chip Select ($\overline{CS}$) signals for each ADC,
    - /CS_X - $\overline{CS}$ signal for ADC_X SPI interface,
    - /CS_Y - $\overline{CS}$ signal for ADC_Y SPI interface, and
    - /CS_Z - $\overline{CS}$ signal for ADC_Z SPI interface;
- Data Ready ($\overline{DRDY}$) signals from each ADC,
    - /DRDY_X - $\overline{DRDY}$ signal from ADC_X,

- /DRDY_Y - $\overline{DRDY}$ signal from ADC_Y, and
- /DRDY_Z - $\overline{DRDY}$ signal from ADC_Z; and

- SD_SCLK - SD SPI clock signal.

The $\overline{DRDY}$ signals are controlled by their respective ADCs, while the remainder of the signals listed above are controlled by the MSP.

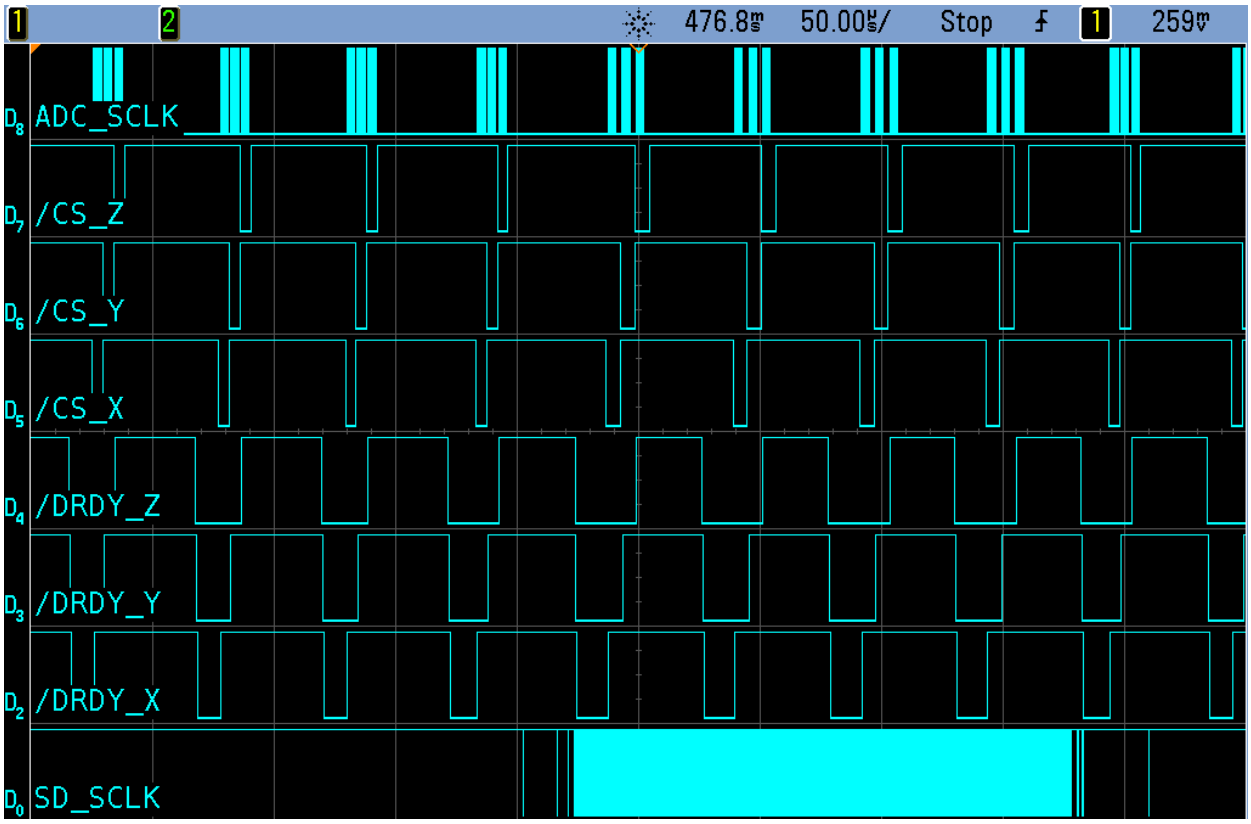

Figure 4.1: Typical waveforms while sampling.

The rectangular sections in ADC_SCLK and SD_SCLK signals in Figure 4.1 consist of high frequency clock signals. The expected ADC_SCLK clock frequency is 10 MHz, while the SD_SCLK clock frequency is 20 MHz. These clock signals can be seen in the magnified oscilloscope image, as seen in Figure 4.2. The time measured for 10-ADC_SCLK clocks and 20-

SD_SCLK clocks is 998 ± 3 ns resulting in a calculated ADC_SCLK frequency of 10.0 ± 0.1 MHz and an SD_SCLK frequency of 20.0 ± 0.3 MHz.

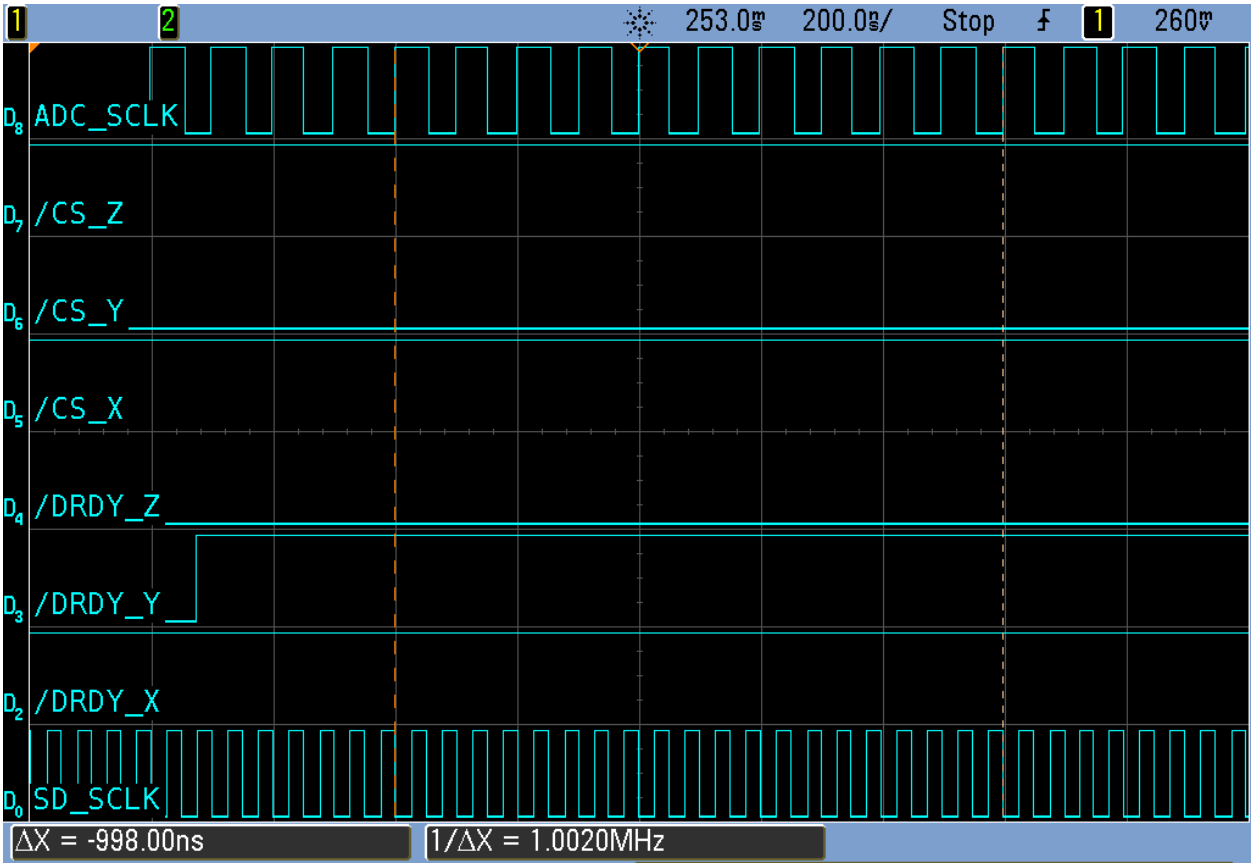

Figure 4.2: Magnified Waveforms of ADC_SCLK and SD_SCLK.

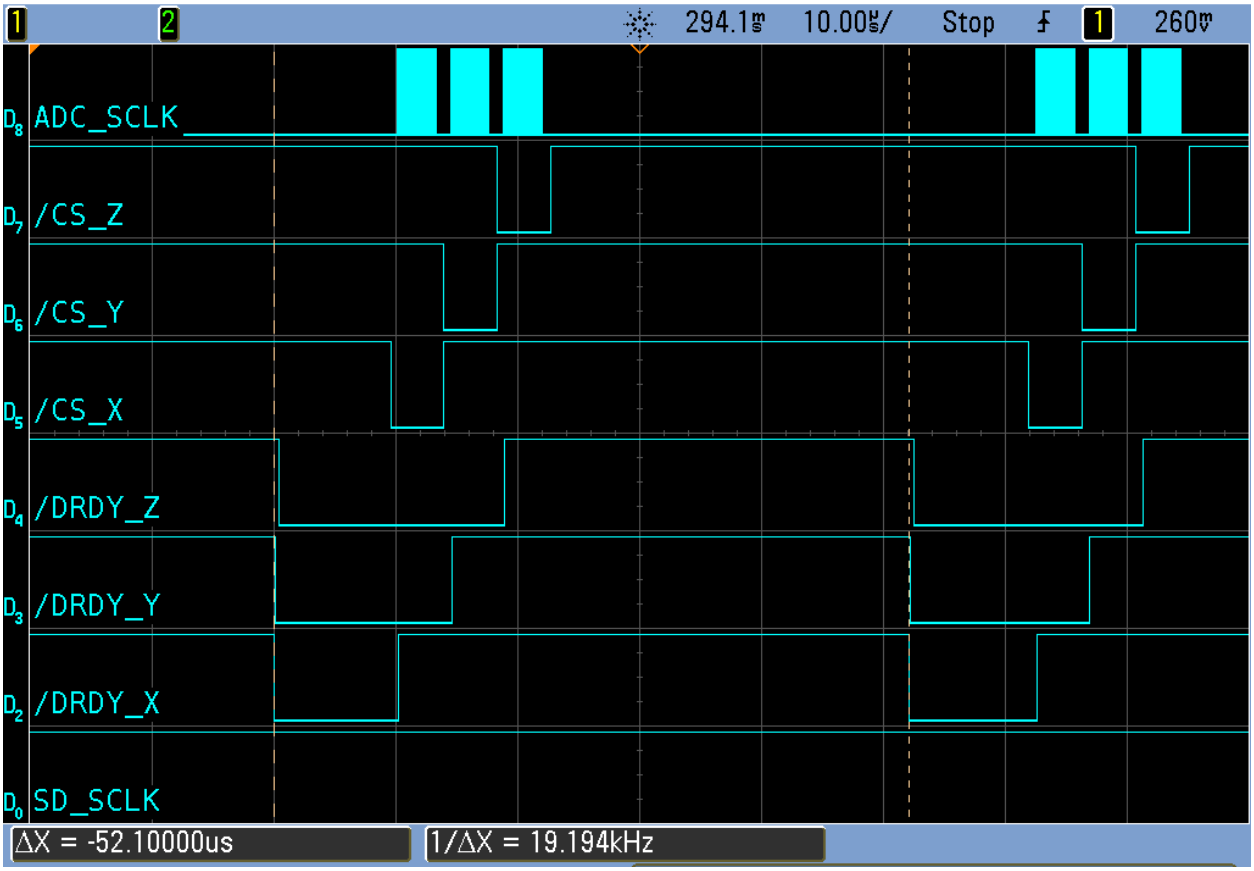

Figure 4.3: Waveform sequence of two consecutive samples.

Figure 4.3 shows the waveform sequence of two consecutive samples. Two vertical cursors show the measured time between two consecutive */DRDY_X* signals as 52.1 ± 0.1 µs. This measurement was performed twenty times. Each measurement of the period of two consecutive */DRDY_X* signals was randomly selected from a discrete sampling event, so that this data set represents twenty different sampling events. During the twenty measurements, all observed periods were either 52.0 ± 0.1 µs or 52.1 ± 0.1 µs. The number of occurrences for each of these two measurements are shown in Table 4.1.

Table 4.1: Period of two consecutive /DRDY_X signals and number of occurrences.

Period of two consecutive /DRDY_X signals (±0.1 µs)	**Number of occurrences**
52.0	5
52.1	15

The calculated mean period of two consecutive */DRDY_X* signals is 52.08 ± 0.02 µs, which results in a calculated */DRDY_X* signal frequency of 19203 ± 7 Hz. The $\overline{DRDY}$ signals generated at the same rate as the sampling rate, which is 19200 SPS. The measured frequency of the /*DRDY_X* signal is within the range of the expected $\overline{DRDY}$ frequency.

The $\overline{DRDY}$ signals are not perfectly aligned, as seen in Figure 4.3. This drift occurs only when the sampling begins and does not change during a prolonged sampling session. There is a different drift between each of the $\overline{DRDY}$ signals, and the largest of these three drifts for any given sampling session is referred to as the maximum drift. Although the initial drift on startup varies from one sampling session to another, the maximum drift between simultaneous $\overline{DRDY}$ signals can be measured, as in Figure 4.4. While investigating this maximum drift, the ADC clock drift was calculated by dividing the maximum drift by the period of the ADC clock signal. After rounding this value to the first decimal point, it shows a clear indication that the drift is a multiple of a number of ADC clocks. Table 4.2 shows ten of these measurements. While taking these measurements, the ADC clock signal is restarted each time.

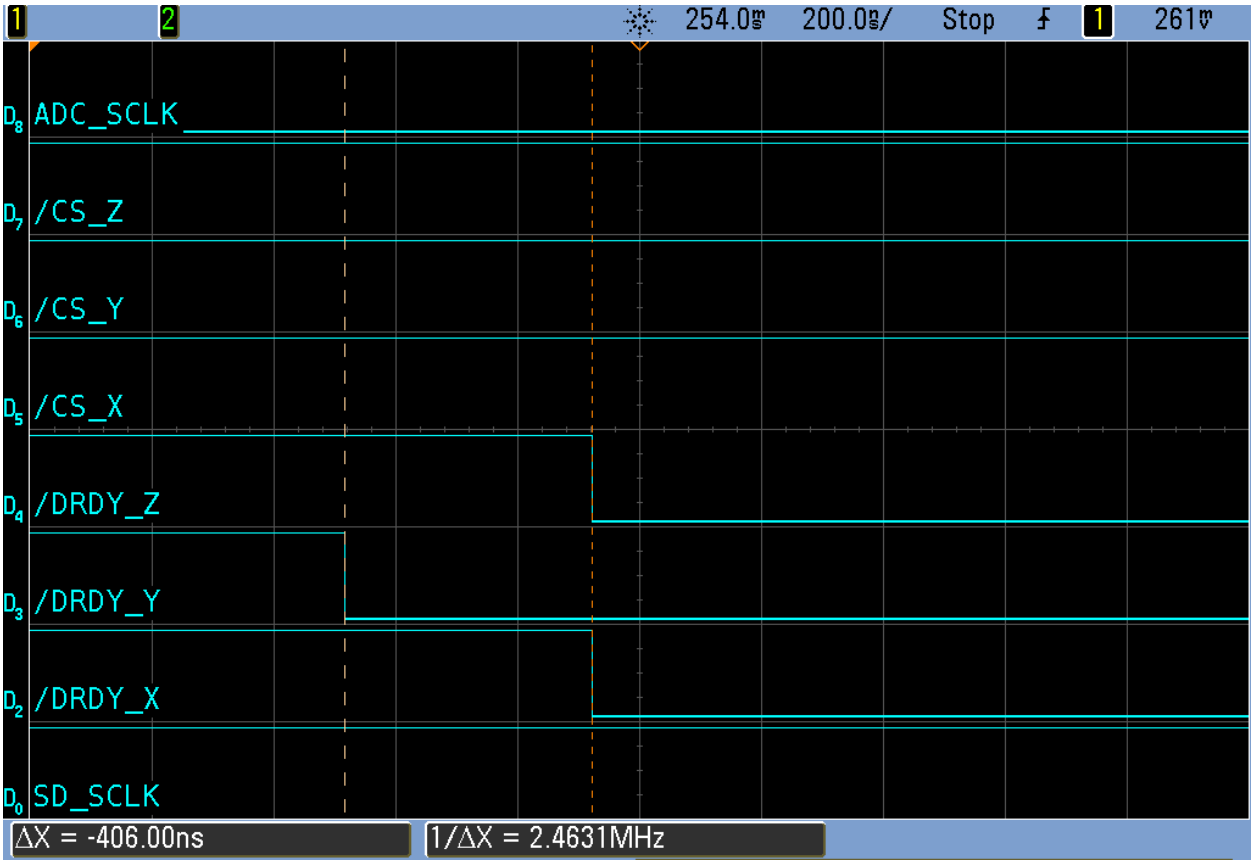

Figure 4.4: Measuring the maximum clock drift for simultaneous samples.

Table 4.2: Maximum drift measurements and the calculated number of ADC clocks.

Maximum drift between each DRDY signals (ΔT_{d_max}) ± 3 ns	$\frac{\Delta T_{d\_max}}{Period\ of\ ADC\ Clock}$
1084	8.0
680	5.0
680	5.0
680	5.0
814	6.0
272	2.0
814	6.0
406	3.0
270	2.0
950	7.0

A possible reason for this drift is that the ADCs take some time to start sampling. This may be due to a combination of propagation delay of the start signal and the ADC clock signal. More investigation is required to confirm the cause of this drift. However, for testing purposes, this initial drift is so small as to not impact data collection and the $\overline{DRDY}$ signals can be considered to be generated simultaneously.

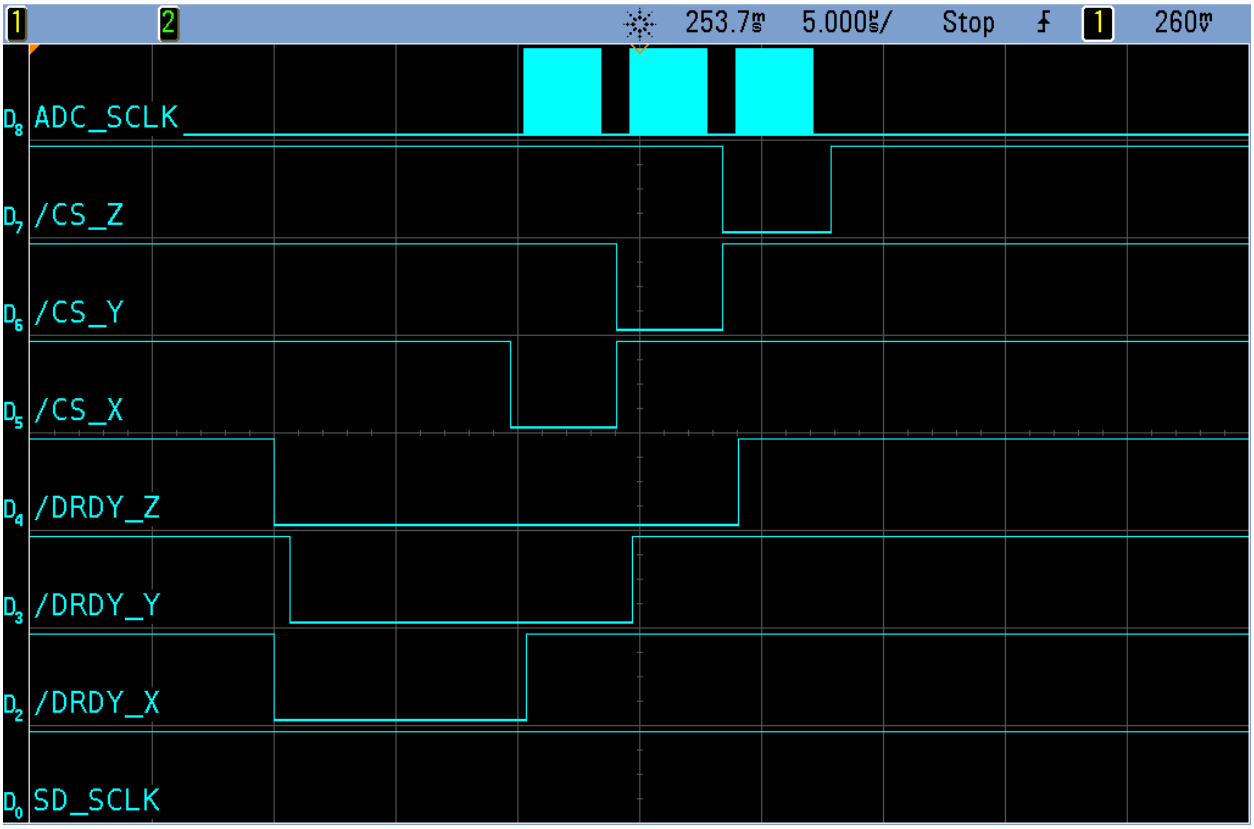

Figure 4.5: Collection of simultaneously generated ADC.

Although the $\overline{DRDY}$ signals are generated simultaneously, the MSP must collect them sequentially, as described in Chapter 3. Figure 4.5 shows the waveform of signals generated during this process, which is as follows:

i) $\overline{DRDY}$ signals are driven low simultaneously, indicating the samples are ready for collection;

ii) /CS_X is driven low, enabling the ADC_X SPI interface;

iii) 32 ADC_SCLK clocks are sent to the ADC, and ADC sends back the 32-bit sample to the MSP simultaneously;

iv) /DRDY_X is driven high immediately after the ADC_SCLK is sent to the ADC_X; and

v) /CS_X is driven high after 32 ADC_SCLK clocks, in order to disable the ADC_X SPI interface.

Steps ii) to v) repeat for collection of data from ADC_Y and ADC_Z. Oscilloscope data, such as the waveform image shown in Figure 4.5, is consistent with the above listed steps, confirming that the sample collection has happened as expected.

The expected Data Frame sending rate to the SD card can be calculated with available information. The total data rate from the ADCs is $19200 \times 4 \times 3 = 230400\ bytes/s$. An additional 4-byte sample number is added to all ADC samples, creating a sample packet of 16-bytes. Sample packets are generated at the same rate as the ADC sample rate. This results in an overall data rate transmitted to the SD card of $19200 \times 16 = 307200\ bytes/s$. Each Data

Frame is 512 bytes, and this results in $\frac{307200}{512} = 600$ Data Frames per second being sent to the SD card while sampling.

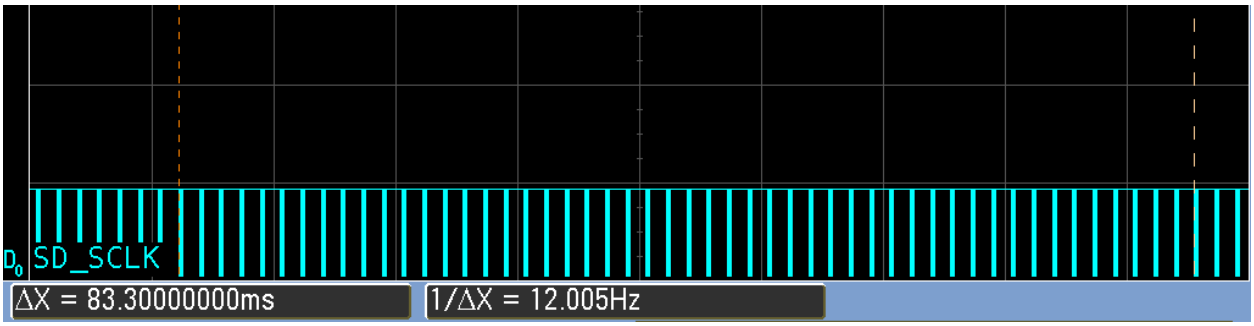

Figure 4.6: Time measured to send fifty Data Frames to the SD card.

Figure 4.6 shows the sequence of Data Frames sent to the SD card and the time measured to send fifty Data Frames, which is 83.3 ± 0.1 ms. The Data Frame sending period is 1.67 ± 0.02 ms, resulting in a calculated Data Frame sending rate of 600 ± 7 Data Frames per second, which is within the expected range.

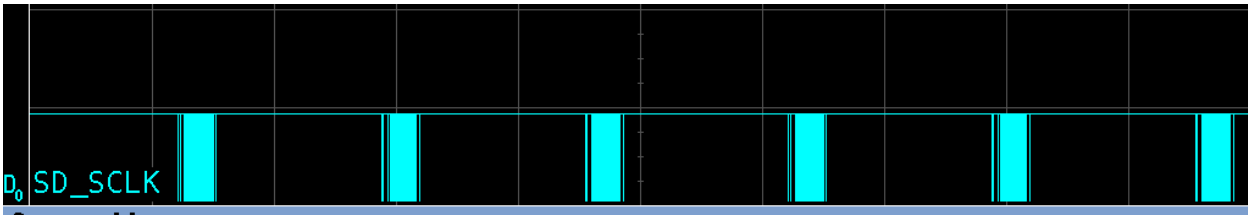

Figure 4.7: Ordinary waveform behavior of SD_SCLK for a 10 ms duration.

The ordinary behavior of the SD_SCLK waveform for a 10 ms duration is shown in Figure 4.7. However, there are situations where this is not the case. Figure 4.8 shows the SD_SCLK signal with a different waveform than expected, due to the busy state of the SD Card.

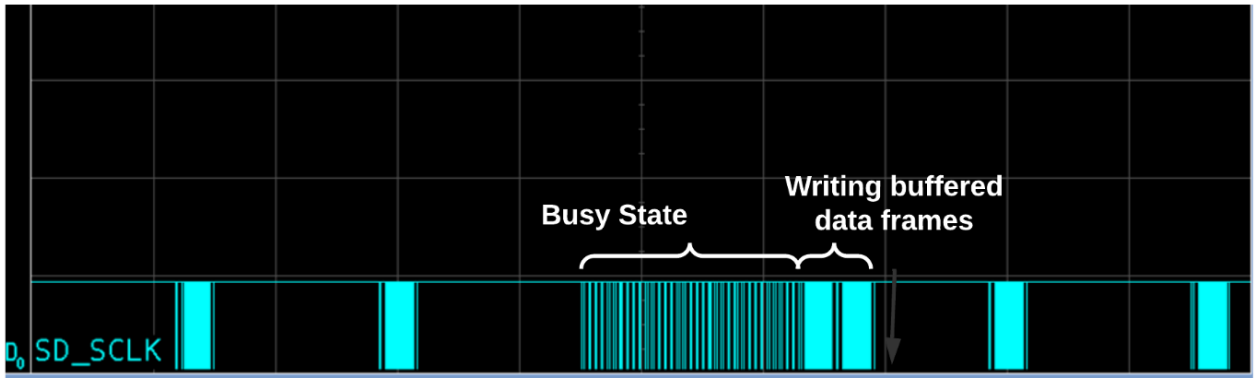

Figure 4.8: SD card busy state and buffered data writing.

SD cards have random variable length busy states while writing data, during which they cannot accept new data to write. The busy state is seen in Figure 4.8 as narrow waveform spikes, which is the MSP checking whether the SD card is available to accept data. In order to avoid data loss, a Data Buffer was implemented to store data that is ready to be sent to the SD card as soon as the SD card is available to accept new data. This Data Buffer can store sixty Data Frames, which is 100 ms of ADC samples.

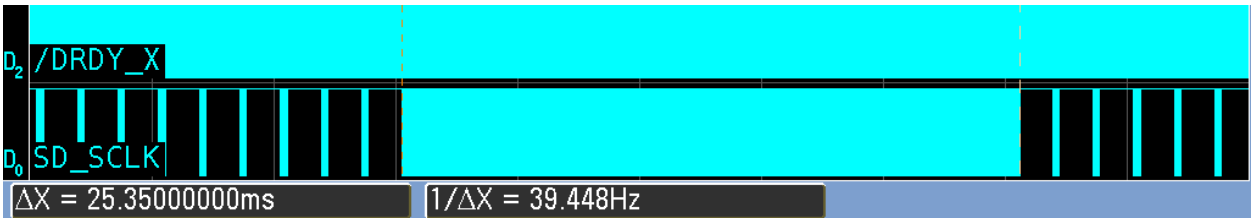

Figure 4.9: Measured SD busy period with buffered data writing time.

Figure 4.9 shows a situation where the busy period and the buffered data writing time took 20.35 ± 0.07 ms. As occurs in this example, the Data Buffer handles a delay of up to 100 ms without data loss. However, if the busy period is more than 100 ms, the entire Data Buffer is overwritten when the ADC Frame Pointer overruns the SD Frame Pointer. When this happens, data is lost in chunks of 100 ms or sixty Data Frames.

After the SD card busy period, the buffered data starts to write again. The MSP tries to send all available buffered Data Frames to the SD card as soon as possible. Depending on the SD card busy period, the number of Data Frames buffered varies. Figures 4.10, 4.11 and 4.12 show instances of sending 16, 12 and 2 buffered Data Frames after SD card busy states.

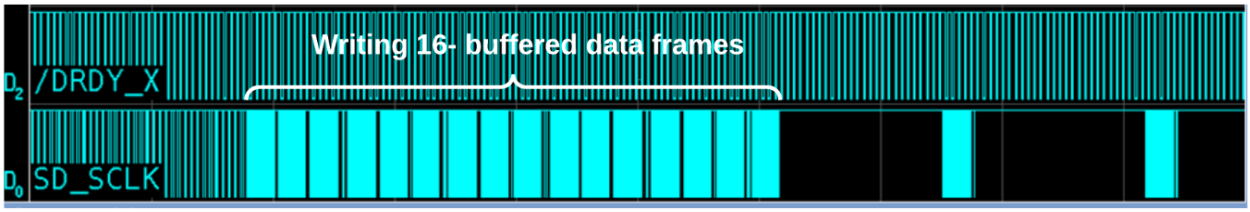

Figure 4.10: Sending sixteen buffered Data Frames to the SD card.

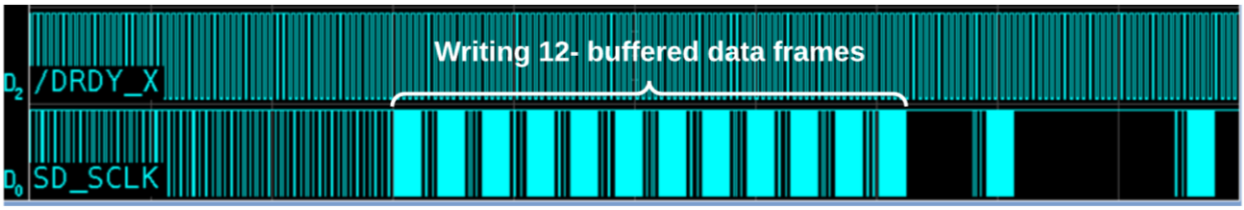

Figure 4.11: Sending twelve buffered Data Frames to the SD card.

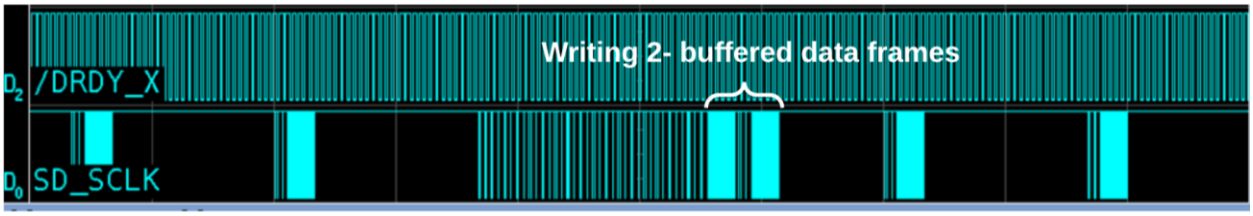

Figure 4.12: Sending two buffered Data Frames to the SD card.

4.2 Analyzing Sampled Data

Since the logger does not store data in a FAT file system, data cannot be read unless using a special computer program capable of reading individual sectors of the SD card. The initial strategy for downloading sample data from the SD card was to read data through the MSP and to send data through the UART to the computer. The processing script was modified to capture this

incoming sample and save it to a comma separated values (CSV) file on the computer. The maximum possible error free download speed was 115200 bps. With this method, downloading one second of sample data took about thirty seconds, which was unacceptably slow for this project. Thus, a different approach was required.

A Python script was developed to download data to the computer directly from the SD card by inserting the SD card into the computer and running the script. The script first reads 12-bytes of the first sector of the SD card to identify the following information:

- starting sector of the data,
- number of ADC frames, and
- number of SD frames.

The python script then starts reading the actual sample data from the starting sector and writes to a CSV file on the computer. The ADC sample data on the SD card is written in little-endian format, while sample numbers are written in big-endian format. The Python program writes both sample data and sample numbers in the little-endian format and converts sample numbers into decimal format while keeping the ADC samples in hexadecimal format.

Figure 4.13 shows the first twenty lines of a CSV file of sample data. Each line represents a sample packet and consists of, respectively, of sample number, ADC_X, ADC_Y, and ADC_Z sample data.

```
1,fffff3ba,f01514c0,fea0c30a
2,00002c05,edfa68f6,ff84df41
3,00004607,ed7a94e3,ff8e9a33
4,000060c7,eea87bfe,00729a42
5,00001e1d,f1751eec,05c360a8
6,00002827,f582ef52,081d4df4
7,fffff647,fa5effle,02b7bdc6
8,fffff8fe,ff833a72,ffe73143
9,fffffb37,0463df07,ffba4e05
10,00003bd3,087ebb46,ffba5165
11,fffff8fe,0b65f5ab,ffba2ff7
12,00001088,0cd5f28d,ffba2cac
13,ffffe70d,0cb2b0e7,ffba71d7
14,00002d42,0af02038,ffbab87d
15,00002e7f,07bdc1a6,ffbaf7cf
16,fffffb60,037243bb,ffba2fce
17,ffffff7e,fe7936le,ff419b64
18,ffffe4d4,f9571a45,fbd9fa74
19,000055ff,f499b06b,fa933e41
20,00000b98,f0c847cf,fdf0e83a
```

Figure 4.13: First twenty lines of a CSV file of sample data downloaded to the computer.

One of the tests conducted was to see whether the SD card was missing sample data due to the busy state. For this test, two SD cards were used:

- SD Card 1: SanDisk Ultra, 16 GB, microSD HC I , and
- SD Card 2: Kingston Canvas Select Plus, 32 GB, microSD HC I.

Each SD card was tested ten times, writing one hour of sample data. Table 4.3: Number of ADC Frames and SD Frames in SD card 1 & 2. Table 4.3 shows the number of ADC Frames that were compiled and the number of frames that were written to the SD card while sampling.

As shown in Table 4.3, some Data Frames were missing from the SD Card 1 on five out of the ten attempts. The missing Data Frames are in multiples of 60 Data Frames, which is the entire length of the Data Buffer. This is a result of the SD card staying in a busy state for more than 100 ms. Although the waveforms show that there are busy states for SD Card 2, there are zero

missing Data Frames. This is an indication that the busy period varies from SD card to SD card. Further testing is needed to better understand SD card performance.

Table 4.3: Number of ADC Frames and SD Frames in SD card 1 & 2.

SD Card 1		**SD Card 2**	
ADC Frames	**SD Frames**	**ADC Frames**	**SD Frames**
2159976	2159976	2159975	2159975
2159975	2159975	2159985	2159985
2159984	2159924 (60 Frames Lost)	2159984	2159984
2159985	2159925 (60 Frames Lost)	2159975	2159975
2159984	2159984	2159983	2159983
2159984	2159924 (60 Frames Lost)	2159983	2159983
2159984	2159984	2159984	2159984
2159983	2159983	2159984	2159984
2159984	2159924 (60 Frames Lost)	2159984	2159984
2159984	2159864 (120 Frames Lost)	2159985	2159985

To observe the signal conversion operation of the ADC, a set of known sinusoidal signals with the amplitude of 500 mV were provided using a “Waveform 4 Click” [16] waveform generator. The signal is directly provided to the input of the ADC section of the RX channel board bypassing the amplifiers. This signal contains a sequence of frequencies which are 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 6000, 7000, 8000 and 9000 Hz. Each frequency was provided for a one-second period starting from 100 Hz. Figure 4.14 shows this sequence of different frequency signals (upper) and the Fourier transform of the sequence of signals (lower). The Fourier transform shows the expected frequencies as spikes in the plot which confirms that the sampled signals are matched with the supplied input signals. Also, signal attenuation was observed after 1 kHz. Figure 4.15 shows a magnified section of the 1 kHz signal, which shows the sinusoidal behavior of the captured signal.

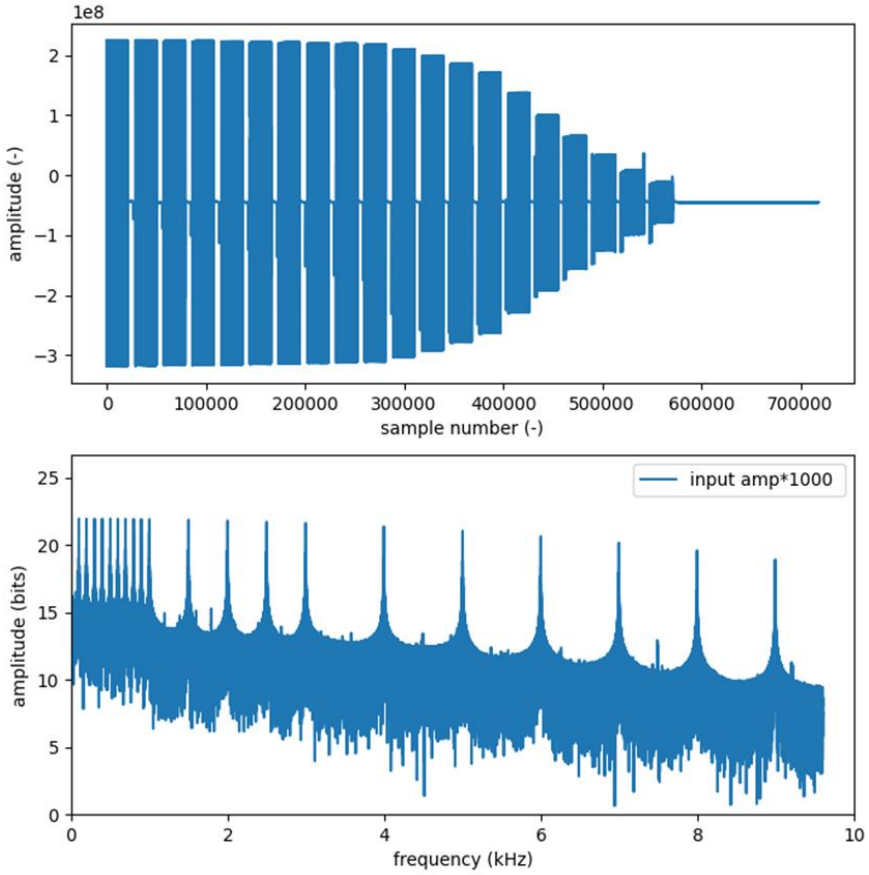

Figure 4.14: Plotted Sequence of frequency signals (upper) and the Fourier transform (lower).

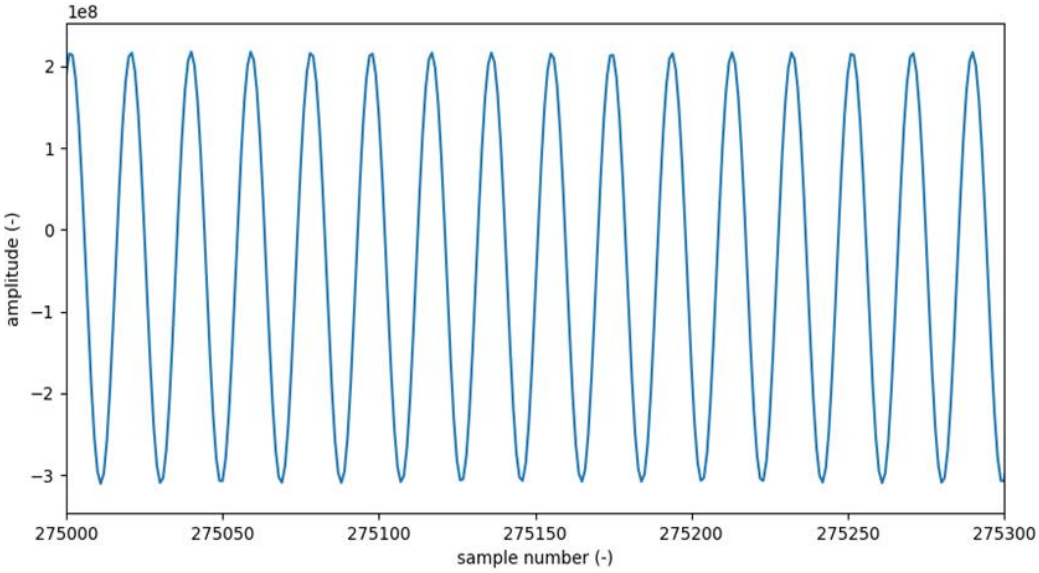

Figure 4.15: Plotted magnified 1 kHz signal.

4.3 Design Considerations

A major design consideration on the LASSITOS project was using two-stage amplification on the RX channel board to reduce ground noise, which was an improvement from the BAS-sled. The new design has separate filtered power supplies at each stage of the RX channel board. These PCBs consist of multiple layers and a separate ground plane, which is also separated between sections. By including the ADC IC in the RX channel board, the signal can be converted into a digital format without introducing additional analog noise. Additionally, the three RX channel boards were separated to further reduce noise between each signal received from the coils. However, this separation of ADCs required a synchronization mechanism, achieved by using a common ADC clock with a frequency of 7.3728 MHz. The MSP cannot provide this clock signal to the ADCs as it runs at 20 MHz, and it is also not possible to generate a 7.3728 MHz clock signal by either using a clock divider or pulse width modulation. Also, the MSP430F6779A only support a single external crystal oscillator, which is already being used for the base clock; therefore, a benchtop waveform generator was used. Future work for this project is to introduce a different MSP430 microcontroller that can have a second external crystal oscillator that can provide a 7.3728 MHz clock signal that can be deployed in the LASSITOS instrument.

After finalizing the RX channel components, primarily the ADC, the feasibility of using the MSP430F6779A and an SD card for data storage on this project was confirmed. The overall Expected Data Rate (EDR) for the SD card was calculated according to the Equation 4.1. Three ADCs were used, which generated samples simultaneously. Each sample packet contains a 4-byte sample number (SN) and three ADC samples, which are 4 bytes each, for a total of 16 bytes. The intended sample rate (SR) for this project is 19200 samples per second.

$$EDR = (Number\ of\ ADCs \times bytes\ per\ sample + \text{bytes per } SN) \times SR \qquad 4.1$$

The EDR for the SD card was calculated as 307200 bytes per second (2457600 bits per second). Therefore, the MSP needed to operate faster than 2.4576 MHz to achieve this rate. As SD card communication required overhead bytes for commands, the data rate needed to be slightly higher than the calculated value. This rate was easily achieved by the MSP430F6779A, as it can operate ten times faster than the calculated value. The SPI communication of the SD card usually has compatible speeds of up to 20-25 MHz, so this data rate is compatible with any available SD card. However, with this development board, five SD cards were tested for SPI communication and all five cards communicated at 20 MHz; therefore, the speed of the MSP was selected as 20 MHz.

Initially, separate SPI channels to the ADCs were used to communicate with the MSP. One of the primary reasons to use the MSP430F6779A was that it had the required number of SPI channels for the ADCs, SD card, and additional components. Separate USCI_ISRs were used for each SPI channel with the RX interrupt enabled. This required switching between ISRs, which consumed a significant amount of CPU time. For this reason, this technique was subsequently discarded. It was redesigned using a single ISR (PORT1_ISR) that is triggered by the DRDY signals as described in Chapter 3.

The SD card library used in this project was originally designed to maintain a FATFS on the SD card. However, using the FATFS introduced significant latency to the SD card writing operation when creating clusters and managing linked lists. For this reason, the FATFS in the SD card library was removed. Only raw data is written to the SD card.

The SD card is usually fragmented into internal sectors with a size of 512 bytes. When the SD card performs read or write operations, 512 bytes of data is written or read at a time. Therefore, 512 bytes of data needs to be buffered before sending data to the SD card. This was initially accomplished by using two Data Frames or double buffering. While one Data Frame was receiving data, the other Data Frame could be used to send the data to the SD card.

While using this technique, a significant amount of data was lost due to the random busy period of the SD card. The double buffering technique functions properly on systems that use much lower data rates; however, it was incompatible with the requirements of the LASSITOS project. Therefore, a larger Data Buffer was needed to reduce the number of lost Data Frames. This problem was solved by using a large Data Buffer and two pointers to create a circular buffer. The ADC Frame Pointer points where the ADC samples are currently being written. The SD Frame Pointer points to the data that needs to be written to the SD card or is currently being written. This Data Buffer is 60 Data Frames, which is 30 KB. In applications with lower data rates, it is simple to modify the Data Buffer.

Chapter 5 Conclusion and Future Work

This thesis describes the development of an embedded system for the receiver of the EM subsystem of the LASSITOS instrument, detailing the hardware and software design and implementation. Based on the test results described in Chapter 4, the data logging process of the prototype is functioning as designed.

The developed system can be adapted for similar data logging applications. Due to the use of a single SPI bus, the number of SPI components can be expanded. For example, additional ADCs can be added to the SPI bus.

Due to the optimized communication techniques, the time taken for sample collection is minimized. Therefore, in a multiple sensor system with lower sampling rates, the microcontroller could wake up and collect samples and then go back to sleep with a minimum amount of time to minimize power consumption.

As the LASSITOS instrument development continues, further improvements are being explored, as detailed below.

1. For this prototype, the development board designed by the SSEP lab is being used. This board has several components which are not used in this application. In order to minimize weight and size, a custom PCB needs to be designed specifically for this data logging system.
2. In this prototype, the ADC clock is provided by a benchtop waveform generator. However, for the final design, a lightweight on-board solution is required. The current MSP430F6779A microcontroller is capable of having only one external crystal oscillator for the base clock. Ideally, a different MSP430 microcontroller that meets project

constraints could be found with the capability of connecting two external crystal oscillators, the ADC clock could be provided by the second crystal oscillator, while the first crystal oscillator provides the base clock for the MSP. Otherwise, an oscillator circuit may need to be added to the PCB.

3. The data collected during testing of this device shows that the SD card busy state varies, depending on which SD card is used. Further testing needs to be done to identify a suitable SD card that is reliably achieving short busy periods in order to use it in the final version of LASSITOS instrument.
4. This design is only one subsystem in the LASSITOS instrument. However, this system needs to be synchronized with the other subsystems in order to generate accurate results. To achieve this, a synchronization signal will be implemented in the current microcontroller or else the subsystems will need to be synchronized with a signal generated from another subsystem.
5. The current microcontroller software organizes the incoming ADC samples and writes data to the SD card, consuming 94% of the available RAM of the microcontroller. Therefore, implementing data processing using the current microcontroller is challenging. Currently, sampling is intended to start when the UAS takes off, which may result in collecting unnecessary data while the UAS is not positioned over sea ice. Alternatively, the MSP could start processing data at a low ADC sample rate without logging, in order to identify quality sea ice data. It could then switch to sample at the full sample rate and start logging, avoiding unnecessary data storage. Distinguishing between high and low quality data could also potentially generate trajectory data for the UAS, optimizing UAS navigation.

In the LASSITOS team's journey towards building and deploying an unmanned aerial system for measuring snow and ice thickness over the Arctic Ocean, one of our first steps is to design a data logging and data storage system that is deployable in such an instrument. This thesis has described significant progress we have made towards that goal.

References

[1] A. Mahoney, H. Eicken, D. Raskovic, D. Thorsen and M. C. Hatfield, "MRI: Development of a Long-range Airborne Snow and Sea Ice Thickness Observing System (LASSITOS)".

[2] A. A. Pfaffhuber, S. Hendricks and Y. A. Kvistedal, "Progressing from 1D to 2D and 3D near-surface airborne electromagnetic mapping with a multisensor, airborne sea-ice explorer," *Geophysics*, vol. 77, no. 4, pp. WB109-WB117, 07 2012.

[3] C. Haas, J. Lobach, S. Hendricks, L. Rabenstein and A. Pfaffling, "Helicopter-borne measurements of sea ice thickness, using a small and lightweight, digital EM system," *Journal of Applied Geophysics*, vol. 67, no. 3, pp. 234-241, 2009.

[4] Texas Instruments, "32-bit 38-kSPS 10-ch delta-sigma ADC with PGA, VREF and auxiliary ADC for factory automation," [Online]. Available: https://www.ti.com/product/ADS1263.

[5] Texas Instruments, "Polyphase metering SoC with 7 Sigma-Delta ADCs, LCD, real-time clock, AES, 512KB Flash, 32KB RAM," [Online]. Available: https://www.ti.com/product/MSP430F6779A.

[6] Texas Instruments, "MSP430 microcontrollers," [Online]. Available: https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/msp430-microcontrollers/overview.html.

[7] SD Card Association and SD Group, "Physical Layer Simplified Specification Version 3.01," [Online]. Available: https://www.sdcard.org/.

[8] "How to Use MMC/SDC," [Online]. Available: http://elm-chan.org/docs/mmc/mmc_e.html.

[9] C. Kai, J. Sheng and S. Wang, "Electromagnetic receiver with capacitive electrodes and triaxial induction coil for tunnel exploration," *Earth, Planets and Space*, vol. 69, no. 1, 2017.

[10] R. Heli, M. Schloesser, A. Offenhäusser, S. V. Waasen and M. Schiek, "Automated electrical stimulation and recording for retina implant research by LabVIEW configured standalone data acquisition device," in *2014 Proceedings of the SICE Annual Conference (SICE)*, 2014.

[11] Y.-H. Hsiao, M.-C. Huang and C.-C. Wang, "Development of MSP430-Based Underwater Acoustic Recorder with Multi-MCU Framework," in *2007 Symposium on Underwater Technology and Workshop on Scientific Use of Submarine Cables and Related Technologies*, 2007.

[12] Analog Devices, "LTC6244 Dual 50MHz, Low Noise, Rail-to-Rail, CMOS Op Amp," [Online]. Available: https://www.analog.com/en/products/ltc6244.html#product-overview.

[13] Analog Devices, "LTC6362 Precision, Low Power Rail-to-Rail Input/Output Differential Op Amp/SAR ADC Driver," [Online]. Available: https://www.analog.com/en/products/ltc6362.html.

[14] Texas Instruments, "MSP430Ware for MSP Microcontrollers," [Online]. Available: https://www.ti.com/tool/MSPWARE.

[15] Processing, "Processing," [Online]. Available: https://processing.org/.

[16] MikroElektronika, "WAVEFORM 4 CLICK," [Online]. Available: https://www.mikroe.com/waveform-4-click.

Appendix

Source code of the MSP430F6779A main.c file.

```
/* main.c file
pin configuration
 * 1.1 - D1, D2, D3 Start signal
 * 1.2 - D1 /DRDY
 * 1.4 - D2 /DRDY
 * 1.6 – D3 /DRDY
 * 2.0 - D1 /CS
 * 2.1 - D2 /CS
 *
 * 3.5 - D1, D2 SCLK
 * 3.6 - D1, D2 SOMI
 * 3.7 - D1, D2 SIMO
 *
 * 4.5 - SD_SOMI
 * 4.6 - SD_SCLK
 * 4.7 - SD_SIMO
 *
 * 7.1 - D1 /RESET
 * 7.2 - D2 /RESET
 *
 */
#include "driverlib.h"
#include "mmc.h"
#include <string.h>
#include "diskio.h"
#include <stdlib.h>
#include "msp430.h"



#define TRUE    1
#define FALSE   0

#define BUFFERSZ    512         // Data_Frame size
#define NUM_BUFFERS  60         // Number of Data_Frames
#define ADC_SPEED   0x0E        // 19200 SPS for ADC

#define DMA_ADC_DA (DMA_BASE + DMA_CHANNEL_0 + OFS_DMA0DA)  // DMA ADC receive
channel Destination address

#define S20M                    // current Clock speed
#define UART_SPEED_115200//UART_SPEED_466800

#ifdef S16M
#define UCS_MCLK_DESIRED_KHZ   16000
#define UCS_MCLK_FLL_RATIO      488
#endif
#ifdef S20M
#define UCS_MCLK_DESIRED_KHZ   20000
```

```
#define UCS_MCLK_FLL_RATIO      610
#endif
#ifdef S22M
#define UCS_MCLK_DESIRED_KHZ   22118
#define UCS_MCLK_FLL_RATIO      674
#endif
#ifdef S24M
#define UCS_MCLK_DESIRED_KHZ   24000
#define UCS_MCLK_FLL_RATIO      732
#endif
#ifdef S25M
#define UCS_MCLK_DESIRED_KHZ   25000
#define UCS_MCLK_FLL_RATIO      762
#endif


typedef struct fileInfo{
    uint32_t startAddress;      // start Sector
    uint32_t ADC_Frames;        // number of ADC frames
    uint32_t SD_Frames;   // number of SD frames
    //uint8_t usedFile;
}fileInfo_t;


/************* Variable Declare **********************/
volatile uint8_t curADCBuf = 0, curSDBuf = 0;            // current data receiving
buffer and sd card writing buffer
volatile uint16_t dataPtr = 0;          // current data receiving pointer
volatile uint8_t newData1 = FALSE;      // is new data available

volatile uint32_t frameNum = 0;         //  record the current frame.
uint32_t SDFrames = 0;
volatile uint32_t sampleNum = 0;        // sample number

uint8_t dataADC[BUFFERSZ * NUM_BUFFERS];      // Data Buffer
uint8_t * dataADCPtrs[NUM_BUFFERS];           // pointers of the data frames


const uint8_t dummy = 0x00;             // dummy byte for SD card

uint8_t RXDataUART = 0;                 // UART data receive
unsigned int i;                         // for indexing

uint8_t Running = FALSE;                // Sampling is happening or not

DRESULT res=RES_OK;                     // disk operation results

uint8_t RWBuffer[BUFFERSZ];             // SD card read buffer

char uartBuf[100], number[15];

/*****************for sd card**********************/
uint32_t sd_blockSize, sd_sectorCount;
fileInfo_t file1;
```

```
/*************Function Definitions*********************/
void initUART(void);                                // initialize UART
void initSPIforD(void);                             // initialize the SPI bus for ADCs
void WriteUARTMsg(const char * msg);                // send UART message for debugging
void getFileInfoFromSD(void);                       // get file information from SD
void initDMA(void);                                 // initialize DMA
void WriteADCReg(uint8_t Addr, uint8_t data);       // Write to ADC registers
void WriteUARTHex(uint8_t hex);                     // Write to UART hex value
void initADC(void);                                 // initialize ADCs
void CalibrateADC(void);                            // calibrate ADCs
void WriteADCRegCalib(uint8_t Addr, uint8_t data);  // Write to ADC registers for
calibration
void initTimer(void);                               // initialize timer
void testReadWriteSD(void);                         // SD card test with read write
uint8_t ReadADCRegD1(uint8_t Addr);                 // read ADC1 register
uint8_t ReadADCRegD2(uint8_t Addr);                 // read ADC2 register
uint8_t ReadADCRegD3(uint8_t Addr);                 // read ADC3 register
/*****************************************************/

void main (void)
{
    WDT_A_hold(WDT_A_BASE);                          // Hold watchdog timer

    /*********Power level and clock configurations ******************/
    PMM_setVCore(PMMCOREV_3);               // Set VCore to level 3
    UCS_setExternalClockSource(32768, 0);   // Set XT1 frequency as 32768 and XT
Frequency as 0
    UCS_initClockSignal(UCS_FLLREF, UCS_XT1CLK_SELECT, UCS_CLOCK_DIVIDER_1);    //
Set DCO FLL Reference as XT1
    UCS_initFLLSettle(UCS_MCLK_DESIRED_KHZ, UCS_MCLK_FLL_RATIO);                //
Set the required frequency


    /******** Port Mapping ***************************************/         //
    const uint8_t port_mapping4[] = {PM_NONE, PM_NONE, PM_NONE, PM_UCA2TXD,
PM_UCA2RXD, PM_UCB1SOMI, PM_UCB1CLK, PM_UCB1SIMO};      // mapping
    PMAP_initPortsParam initPortsParam4 = {0};
    initPortsParam4.portMapping = port_mapping4;
    initPortsParam4.PxMAPy = (uint8_t *) &P4MAP01;
    initPortsParam4.numberOfPorts = 1;
    initPortsParam4.portMapReconfigure = PMAP_ENABLE_RECONFIGURATION;
    PMAP_initPorts(PMAP_CTRL_BASE, &initPortsParam4);

    const uint8_t port_mapping3[] = {PM_NONE, PM_NONE, PM_NONE, PM_NONE, PM_NONE,
PM_UCA3CLK, PM_UCA3SOMI, PM_UCA3SIMO};
    PMAP_initPortsParam initPortsParam3 = {0};
    initPortsParam3.portMapping = port_mapping3;
    initPortsParam3.PxMAPy = (uint8_t *) &P3MAP01;
    initPortsParam3.numberOfPorts = 1;
    initPortsParam3.portMapReconfigure = PMAP_ENABLE_RECONFIGURATION;
    PMAP_initPorts(PMAP_CTRL_BASE, &initPortsParam3);
```

```
    /********* GPIO Configuration ******************************/
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P3, GPIO_PIN2 + GPIO_PIN3 +
GPIO_PIN4 + GPIO_PIN5 + GPIO_PIN6 + GPIO_PIN7 +
                                               GPIO_PIN8 + GPIO_PIN9 + GPIO_PIN10 +
GPIO_PIN11 + GPIO_PIN12 + GPIO_PIN13 + GPIO_PIN14 + GPIO_PIN15);
    GPIO_setAsOutputPin(GPIO_PORT_P7,  GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN2 +
GPIO_PIN3);      // pin1,2 and 3 works as /RESET
    GPIO_setOutputHighOnPin(GPIO_PORT_P7, GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN2 +
GPIO_PIN3);

    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 +
GPIO_PIN6);  // ADC data ready signals
    GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
// Enable interrupt for data ready
    GPIO_selectInterruptEdge(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6,
GPIO_HIGH_TO_LOW_TRANSITION); // Interrupt neg edge
    GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
// clear interrupt flags

    GPIO_setAsOutputPin(GPIO_PORT_P5, GPIO_PIN0);           // SD chip select pin

    GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN3);
// ADC /CS pins
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN0 + GPIO_PIN1 + GPIO_PIN3);
// deselect ADCs

    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P1, GPIO_PIN0);
// timer output pin PWM

    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN1);           // P1.1 as the start pin
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN1);


    GPIO_setAsOutputPin(GPIO_PORT_P7,  GPIO_PIN4 + GPIO_PIN5 + GPIO_PIN6 +
GPIO_PIN7);      // for debugging


/************ Other Initializations ****************************/
    initUART();     // init UART
    initSPIforD();  // init SPI for ADCs
    initDMA();      // init DMA
    //SDselect();
    if(disk_initialize(0) == 0){
        WriteUARTMsg("\r\nSD Card initialized.\r\n");
    }else{
        WriteUARTMsg("\r\nFailed to initialize SD card.\r\n");
    }
    __delay_cycles(1000);

    memset(&dataADC[0], 0, BUFFERSZ * NUM_BUFFERS);

    for(i=0;i<NUM_BUFFERS;i++){  // setup data frame pointets
        dataADCPtrs[i] = &dataADC[i*BUFFERSZ];
    }
```

```
    __enable_interrupt(); // enable global interrupts
    WriteUARTMsg("Init Done");

/********** Main Loop*******************************/

    while(1){
        if(curADCBuf != curSDBuf)         // if a buffer is full
        {
            P7OUT |= GPIO_PIN5;           // for debugging

            if(disk_write_multiple(dataADCPtrs[curSDBuf])==RES_OK) SDFrames++; //
increment SD Frames if the write successfull

            curSDBuf++; // increment SD Frame pointer
            if(curSDBuf>=NUM_BUFFERS) curSDBuf=0;

            P7OUT &= ~GPIO_PIN5;                    // for debugging
            if((Running == FALSE) && (curADCBuf == curSDBuf)){                    //
if the Stop sample has been received
                __delay_cycles(1000);
                disk_Write_multiple_stop();         // stop multiple write to SD card

                // send number of frames
                memset(&uartBuf, 0, sizeof(uartBuf));
                ltoa(frameNum, number,10);
                strcpy(uartBuf, "\r\nData ADC frames : ");
                strcat(uartBuf, number);
                WriteUARTMsg(uartBuf);

                memset(&uartBuf, 0, sizeof(uartBuf));
                ltoa(SDFrames, number,10);
                strcpy(uartBuf, "\r\nSD data frames : ");
                strcat(uartBuf, number);
                WriteUARTMsg(uartBuf);


                file1.ADC_Frames = frameNum;
                file1.SD_Frames = SDFrames;
                memset(&RWBuffer[0],0,BUFFERSZ);
                memcpy(&RWBuffer[0], &file1, sizeof(fileInfo_t));
                if(disk_write(0, &RWBuffer[0], 0, 1)== RES_OK){
                    WriteUARTMsg("\n\rFile Created Successfully.\n\r");
                }else{
                    WriteUARTMsg("\n\rFile create failed.\n\r");
                }

            }
        }
    }
}

/*********** Interrupt Service Routines ************************/
#pragma vector=USCI_A2_VECTOR
__interrupt
```

```
void EUSCI_A2_ISR(void)
{
    switch(__even_in_range(UCA2IV, USCI_UART_UCTXCPTIFG))
    {
    case USCI_NONE: break;
    case USCI_UART_UCRXIFG:
        RXDataUART = EUSCI_A_UART_receiveData(EUSCI_A2_BASE);
        if(RXDataUART == 'S')         // start sampling
        {
            res = disk_write_multiple_init(file1.startAddress);
            if(res == RES_OK){
                GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
                GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 +
GPIO_PIN6);
                Running = TRUE;
                GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN1);// start signal for
ADCs
            }else{
                WriteUARTMsg("\r\nCannot open file\r\n");
            }
        }
        else if(RXDataUART == 'T')       // stop sampling
        {
            Running = FALSE;
        }
        else if(RXDataUART == 'E')      // Read ADC Registers
        {
            for(i=0;i<0x1A;i++)
            {
                uint8_t adcReg1 = ReadADCRegD1(i);
                uint8_t adcReg2 = ReadADCRegD2(i);
                uint8_t adcReg3 = ReadADCRegD3(i);
                WriteUARTMsg("\r\n");
                WriteUARTHex(i);WriteUARTMsg("\t");
                WriteUARTHex(adcReg1);WriteUARTMsg("\t");
                WriteUARTHex(adcReg2);WriteUARTMsg("\t");
                WriteUARTHex(adcReg3);WriteUARTMsg("\t");
            }
        }
        else if(RXDataUART == 'D')       // INIT ADC
        {
            initADC();
        }
        else if(RXDataUART == 'N')        // New sampling session
        {
            frameNum = 0;
            dataPtr = 0;
            sampleNum = 0;
            SDFrames = 0;

        }
        else if(RXDataUART == 'C')       // Calibrate ADCs
        {
            CalibrateADC();
        }
```

```
        break;

    case USCI_UART_UCTXIFG: break;
    case USCI_UART_UCSTTIFG: break;
    case USCI_UART_UCTXCPTIFG: break;
    }
}

#pragma vector=PORT1_VECTOR
__interrupt
void port1_ISR(void)
{
    //P7OUT |= GPIO_PIN3;
    if((P1IFG & GPIO_PIN2)&&(P1IFG & GPIO_PIN4)&&(P1IFG & GPIO_PIN6))
    {
        dataPtr+=16;
        if(dataPtr >= BUFFERSZ)// if buffer filled, switch current ADC data buffer
        {
            P7OUT |= GPIO_PIN4;        // debugging purposes
            dataPtr = 0;              // reset data pointer
            frameNum++;         // increment the frame number
            curADCBuf++;
            if(curADCBuf>=NUM_BUFFERS) curADCBuf = 0;
            if(Running == FALSE)      // if the stop command received stop sampling
            {
                P1OUT &= ~GPIO_PIN1;       // Stop sampling signal for ADCs
                GPIO_disableInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 +
GPIO_PIN6);
                GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
                return;
            }
            P7OUT &= ~GPIO_PIN4;                        // debugging purposes
        }
        sampleNum++;        // increment sample number
        memcpy(dataADCPtrs[curADCBuf]+dataPtr,&sampleNum, 4);   // include sample
number
        __data16_write_addr((unsigned short)(DMA_ADC_DA),(unsigned
long)dataADCPtrs[curADCBuf]+dataPtr+4);  // set receiver address on buffer

        UCA3IFG &= ~UCRXIFG;            // Clear RX interrupt flag
        DMA0CTL |= DMAEN;               // Enable DMA transfer on RX (12 bytes)
        P2OUT = GPIO_PIN1 | GPIO_PIN3;  // ADC1 Chip Select
        DMA2CTL |= DMAEN;               // Enable DMA transfer on TX ( 4 bytes)
        __delay_cycles(60);             // wait till transfer done

        P2OUT = GPIO_PIN0 | GPIO_PIN3;  // ADC2 chip select and ADC1 deselect
        DMA2CTL |= DMAEN;               // Enable DMA transfer on TX (4 bytes)
        __delay_cycles(60);             // wait till transfer done

        P2OUT = GPIO_PIN0 | GPIO_PIN1;  // ADC3 chip select and ADC2 deselect
        DMA2CTL |= DMAEN;               // Enable DMA transfer on TX (4 bytes)
        __delay_cycles(60);             // wait till transfer done
        P2OUT |= GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN3; // deselect ADC3

        P1IFG &= ~(GPIO_PIN2|GPIO_PIN4|GPIO_PIN6);  // clear pin interrupt flags
```

```
    }
}


/************* SD card functions ***********************/

void getFileInfoFromSD(void)
{
    res = disk_read(0, &RWBuffer[0], 0, 1);
    fileInfo_t fileInfo = {0};
    memcpy(&fileInfo,&RWBuffer[0], sizeof(fileInfo_t));

    memset(&uartBuf, 0, sizeof(uartBuf));
    strcpy(uartBuf, "\r\nFileName : ");
    strcat(uartBuf, fileInfo.fileName);
    strcat(uartBuf, "\r\nADCFrames : ");
    ltoa(fileInfo.ADC_Frames, number, 10);
    strcat(uartBuf, number);
    strcat(uartBuf, "\r\nSDFrames : ");
    ltoa(fileInfo.SD_Frames, number, 10);
    strcat(uartBuf, number);
    strcat(uartBuf, "\r\nStartAddress : ");
    ltoa(fileInfo.startAddress, number, 10);
    strcat(uartBuf, number);
    WriteUARTMsg(uartBuf);
}


/************ Other Functions ******************************/
void initSPIforD(void)           // initialize the SPI bus for ADCs
{
    EUSCI_A_SPI_initMasterParam paramSPI = {0};
    paramSPI.selectClockSource = EUSCI_A_SPI_CLOCKSOURCE_SMCLK;
#ifdef S16M
    paramSPI.clockSourceFrequency = 16000000;
    paramSPI.desiredSpiClock = 8000000;
#endif
#ifdef S20M
    paramSPI.clockSourceFrequency = 20000000;
    paramSPI.desiredSpiClock = 8000000;
#endif
#ifdef S22M
    paramSPI.clockSourceFrequency = 22118400;
    paramSPI.desiredSpiClock = 7372800;
#endif
#ifdef S24M
    paramSPI.clockSourceFrequency = 24000000;
    paramSPI.desiredSpiClock = 8000000;
#endif
#ifdef S25M
    paramSPI.clockSourceFrequency = 25000000;
    paramSPI.desiredSpiClock = 8000000;
#endif
    paramSPI.msbFirst = EUSCI_A_SPI_MSB_FIRST;
    paramSPI.clockPhase = EUSCI_A_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT;
```

```
    paramSPI.clockPolarity = EUSCI_A_SPI_CLOCKPOLARITY_INACTIVITY_LOW;
    paramSPI.spiMode = EUSCI_A_SPI_3PIN;
    EUSCI_A_SPI_initMaster(EUSCI_A3_BASE, &paramSPI); // configure EUSCI_A3 as SPI
master
    EUSCI_A_SPI_enable(EUSCI_A3_BASE);           // enable EUSCI_A3 for spi
    EUSCI_A_SPI_clearInterrupt(EUSCI_A3_BASE, EUSCI_A_SPI_RECEIVE_INTERRUPT);
// clear spi receive interrupt
}


void initDMA(void)                                  // initialize DMA
{
    DMA_initParam DParamADC = {0};
    DParamADC.channelSelect = DMA_CHANNEL_0;
    DParamADC.transferModeSelect = DMA_TRANSFER_SINGLE;
    DParamADC.transferSize = 12;
    DParamADC.triggerSourceSelect = DMA_TRIGGERSOURCE_25;   // UCA3RXIFG
    DParamADC.transferUnitSelect = DMA_SIZE_SRCBYTE_DSTBYTE;
    DParamADC.triggerTypeSelect = DMA_TRIGGER_HIGH;

    DMA_init(&DParamADC);
    DParamADC.transferSize = 4;
    DParamADC.channelSelect = DMA_CHANNEL_2;
    DParamADC.triggerSourceSelect = DMA_TRIGGERSOURCE_26;   // UCA3TXIFG

    DMA_init(&DParamADC);

    DMA_setSrcAddress(DMA_CHANNEL_0,
EUSCI_A_SPI_getReceiveBufferAddress(EUSCI_A3_BASE), DMA_DIRECTION_UNCHANGED);

    DMA_setDstAddress(DMA_CHANNEL_0, (uint32_t)(uintptr_t)&dataADC[0],
DMA_DIRECTION_INCREMENT);
    DMA_setSrcAddress(DMA_CHANNEL_2, (uint32_t)(uintptr_t)&dummy,
DMA_DIRECTION_UNCHANGED);
    DMA_setDstAddress(DMA_CHANNEL_2,
EUSCI_A_SPI_getTransmitBufferAddress(EUSCI_A3_BASE), DMA_DIRECTION_UNCHANGED);

    DMA_initParam DParamSD = {0};
    DParamSD.channelSelect = DMA_CHANNEL_1;
    DParamSD.transferModeSelect = DMA_TRANSFER_BLOCK;
    DParamSD.transferSize = 512;
    DParamSD.triggerSourceSelect = DMA_TRIGGERSOURCE_28;   // UCB1TXIFG
    DParamSD.transferUnitSelect = DMA_SIZE_SRCBYTE_DSTBYTE;
    DParamSD.triggerTypeSelect = DMA_TRIGGER_HIGH;
    DMA_init(&DParamSD);
    DMA_setDstAddress(DMA_CHANNEL_1,
EUSCI_B_SPI_getTransmitBufferAddress(EUSCI_B1_BASE), DMA_DIRECTION_UNCHANGED);


}

/*********** UART Functions ************************************/

void initUART(void)
{
```

```
    /****************************** UART Interface for debugging
*****************************************************/
       EUSCI_A_UART_initParam uartParam = {0};
       uartParam.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_SMCLK;
#ifdef UART_SPEED_115200
    #ifdef S16M
       uartParam.clockPrescalar = 8;
       uartParam.firstModReg = 10;
       uartParam.secondModReg = 0xF7;
    #endif
    #ifdef S20M
       uartParam.clockPrescalar = 10;
       uartParam.firstModReg = 13;
       uartParam.secondModReg = 0xAD;
    #endif
   #ifdef S22M
       uartParam.clockPrescalar = 12;
       uartParam.firstModReg = 0;
       uartParam.secondModReg = 0x55;
   #endif
   #ifdef S24M
       uartParam.clockPrescalar = 13;
       uartParam.firstModReg = 0;
       uartParam.secondModReg = 0x0B;
   #endif
   #ifdef S25M
       uartParam.clockPrescalar = 13;
       uartParam.firstModReg = 9;
       uartParam.secondModReg = 0xAB;
   #endif
#endif
       uartParam.parity = EUSCI_A_UART_NO_PARITY;
       uartParam.msborLsbFirst = EUSCI_A_UART_LSB_FIRST;
       uartParam.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT;
       uartParam.uartMode = EUSCI_A_UART_MODE;
       uartParam.overSampling = EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION;

       if(STATUS_FAIL == EUSCI_A_UART_init(EUSCI_A2_BASE, &uartParam)) return;

       EUSCI_A_UART_enable(EUSCI_A2_BASE);
       EUSCI_A_UART_clearInterrupt(EUSCI_A2_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT);
       EUSCI_A_UART_enableInterrupt(EUSCI_A2_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT);
}

void WriteUARTMsg(const char * msg)                // send UART message for debugging
{
    int j;
    for(j=0; j<strlen(msg); j++){
        //EUSCI_A_UART_transmitData(EUSCI_A2_BASE, msg[j]);
        while(!(UCA2IFG & UCTXIFG));
        UCA2TXBUF = msg[j];
    }
}

void WriteUARTHex(uint8_t hex)                     // Write to UART hex value
```

```
{
    if((hex>>4)<10)
    {
        //EUSCI_A_UART_transmitData(EUSCI_A2_BASE, (hex>>4)+'0');
        while(!(UCA2IFG & UCTXIFG));
        UCA2TXBUF = (hex>>4)+'0';
    }
    else
    {
        //EUSCI_A_UART_transmitData(EUSCI_A2_BASE, (hex>>4)+'7');
        while(!(UCA2IFG & UCTXIFG));
        UCA2TXBUF = (hex>>4)+'7';
    }

    if((hex & 0x0F)<10)
    {
        //EUSCI_A_UART_transmitData(EUSCI_A2_BASE, (hex & 0x0F)+'0');
        while(!(UCA2IFG & UCTXIFG));
        UCA2TXBUF = (hex & 0x0F)+'0';
    }
    else
    {
        //EUSCI_A_UART_transmitData(EUSCI_A2_BASE, (hex & 0x0F)+'7');
        while(!(UCA2IFG & UCTXIFG));
        UCA2TXBUF = (hex & 0x0F)+'7';
    }
}

/************************** ADC Functions ********************/

void initADC(void)                                  // initialize ADCs
{
    WriteADCReg(0x01, 0x01);
    WriteADCReg(0x02, 0x00);
    WriteADCReg(0x03, 0x00);
    WriteADCReg(0x04, 0x80);
    WriteADCReg(0x05, ADC_SPEED);
    WriteADCReg(0x06, 0x01);
    WriteADCReg(0x07, 0x00);
    WriteADCReg(0x08, 0x00);
    WriteADCReg(0x09, 0x00);
    WriteADCReg(0x0A, 0x00);
    WriteADCReg(0x0B, 0x00);
    WriteADCReg(0x0C, 0x40);
    WriteADCReg(0x0D, 0xBB);
    WriteADCReg(0x0E, 0x00);

    WriteADCReg(0x19, 0x00);
    WriteADCReg(0x1A, 0x40);

    WriteUARTMsg("Done INIT \r\n");
}

void CalibrateADC(void)                             // calibrate ADCs
{
```

```
    WriteUARTMsg("Starting Calibration \r\n");
    GPIO_disableInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
    GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);

    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0);    // chip select ADC1
    WriteADCRegCalib(0x03, 0x00);                       // select continuous mode
    WriteADCRegCalib(0x05, ADC_SPEED);                  // select ADC speed
    WriteADCRegCalib(0x01, 0x01);                       // select referance voltage
    WriteADCRegCalib(0x06, 0x01);                       // for system calibration
select input to 01
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN1);   // start conversion
    __delay_cycles(10);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x16);      // start system offset
calibration
    __delay_cycles(10);
    while(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN2));
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN1);    // stop conversion
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN0);   // deselect ADC1

    __delay_cycles(1000);

    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);    // select ADC2
    WriteADCRegCalib(0x03, 0x00);                       // select continuous mode
    WriteADCRegCalib(0x05, ADC_SPEED);                  // select ADC speed
    WriteADCRegCalib(0x01, 0x01);                       // select referance voltage
    WriteADCRegCalib(0x06, 0x01);                       // for system calibration
select input to 01
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN1);   // start conversion
    __delay_cycles(10);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x16);      // start system offset
calibration
    __delay_cycles(10);
    while(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN4));
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN1);    // stop conversion
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);   // deselect ADC1

    __delay_cycles(1000);

    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN3);    // select ADC3
    WriteADCRegCalib(0x03, 0x00);                       // select continuous mode
    WriteADCRegCalib(0x05, ADC_SPEED);                  // select ADC speed
    WriteADCRegCalib(0x01, 0x01);                       // select referance voltage
    WriteADCRegCalib(0x06, 0x01);                       // for system calibration
select input to 01
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN1);   // start conversion
    __delay_cycles(10);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x16);      // start system offset
calibration
    __delay_cycles(10);
```

```
    while(GPIO_INPUT_PIN_HIGH == GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN6));
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN1);     // stop conversion
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN3);    // deselect ADC1

    __delay_cycles(1000);

    WriteUARTMsg("Calibration completed. \r\n");

    GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
    GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN2 + GPIO_PIN4 + GPIO_PIN6);
}

void WriteADCReg(uint8_t Addr, uint8_t data)       // Write to ADC registers
{
    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x40);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, data);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN0);

    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x40);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, data);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);

    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN3);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x40);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, data);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN3);
}

void WriteADCRegCalib(uint8_t Addr, uint8_t data)  // Write to ADC registers for
calibration
{
```

```
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x40);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, data);
}


uint8_t ReadADCRegD1(uint8_t Addr)                // read ADC1 register
{
    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x20);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN0);
    return EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
}

uint8_t ReadADCRegD2(uint8_t Addr)                // read ADC2 register
{
    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x20);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);
    return EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
}

uint8_t ReadADCRegD3(uint8_t Addr)                // read ADC2 register
{
    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN3);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, Addr + 0x20);
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
```

```
    while(!EUSCI_A_SPI_getInterruptStatus(EUSCI_A3_BASE,
EUSCI_A_SPI_TRANSMIT_INTERRUPT));
    EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
    EUSCI_A_SPI_transmitData(EUSCI_A3_BASE, 0x00);
    GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN3);
    return EUSCI_A_SPI_receiveData(EUSCI_A3_BASE);
}
```