NORTHWEST NAZARENE UNIVERSITY

Creation of MakerSat Flight Code

THESIS

Submitted to the Department of Mathematics and Computer Science

In partial fulfillment of the requirements

for the degree of

BACHELOR OF ARTS

Aaron Ewing

2017

THESIS
Submitted to the Department of Mathematics and Computer Science

In partial fulfillment of the requirements

for the degree of

BACHELOR OF ARTS

Aaron Ewing

2017

Creation of MakerSat Flight Code

Author: Aaron Ewing

Aaron Ewing

Approved: [illegible]

Dr. Barry Myers, Ph.D., Professor of Computer Science,

Department of Mathematics and Computer Science, Faculty Advisor

Approved: [illegible]

Joshua Griffin, Ph.D., Assistant Professor of Electrical Engineering

Department of Engineering and Physics, Second Reader

Approved: [illegible]

Barry Myers, Ph.D., Chair,

Department of Mathematics and Computer Science

Abstract

MakerSat Flight Code Creation.

EWING, AARON (Department of Mathematics & Computer Science). MYERS, DR. BARRY (Department of Mathematics & Computer Science).

MakerSat is a nanosatellite, commonly referred to as a CubeSat, that will launch into Earth's orbit in 2017 on one of NASA's (National Aeronautics and Space Administration) ELaNa (Educational Launch of Nanosatellites) rockets. This launch will likely represent Idaho's very first satellite to go into outer space. MakerSat is a university research project to test what kinds of 3D printed plastic polymers can survive the harsh conditions of outer space over a long period of time without corrosion. The desired outcome of this project is to one day be able to 3D print satellites on the ISS (International Space Station) for quick, easy, and cheap development and deployment. The construction of MakerSat required multiple students to develop code on multiple different microcontrollers for MakerSat to reliably collect project data and send it back to Earth for processing. The responsibilities of the student code developers included running multiple project experiments, sending raw information to the "Hub" microcontroller, sending packetized data to the radio, proper life cycling (due to low power requirements), and processing collected experiment data on Earth to provide useful scientific information concerning the research project.

Acknowledgments

I would like to thank everyone who worked with me on the research project which became my senior project. Dr. Steve Parke, Dr. Joshua Griffin, Braden Grim, Mitchel Kamstra, and Connor Nogales. They made the project fun and interesting to work on, and more than anything they put everything they had into this project. Without their strength and dedication, this project would not have succeeded. I also wish to thank everyone at NearSpace Launch (NSL), who helped turn our project idea into reality. I also want to thank the several Caldwell High school students who wanted to be a part of our project, and their engineering teacher Dennis Zattiero. They all worked really hard to create their science project for outer space, and they succeeded. I, of course, would like to thank my friends and family, who encouraged me on a daily basis to get this amazing project done. They understood when I could not be with them, and their constant support was the only thing that allowed me to keep my sanity at times. Without everyone I have mentioned so far, this project would not have succeeded.

Table of Contents

Background

A CubeSat is a type of miniaturized satellite that is deployable into outer space, often for the purpose of research. These small satellites are considered U-class, meaning that a 1U CubeSat is a 10cm cube weighing approximately 1.3 kg or less. CubeSats started being developed in 1999 and has grown increasingly popular ever since then.

NNU has two MakerSat missions currently planned, MakerSat-0 and MakerSat-1. MakerSat-1's goal is to take the next step in the rapid development of microsatellites and demonstrate how they could one day be printed and assembled on the International Space Station (ISS). The ISS has recently obtained a 3D printer[3D Printer] that will work in a micro-G environment. MakerSat-1 will have rails on four of its edges to hold the CubeSat together (see Figures 1 and 2) 3D printed and then assembled on the ISS. The goal of the rails is to show that a nanosatellite can survive the environment of space and could be printed on the ISS. Once released the CubeSat will be in a similar orbit to the ISS, a 330-435 km altitude, where it will hopefully perform the experiments it was sent up to do and then burn up on re-entry up to 10 years from when it started.

MakerSat-1 main experiment involves two parts: a CMOS[CMOS] imager and the ability to measure the degradation of ULTEM polymers. The purpose of the CMOS imagers is to take images of Earth while the purpose of the polymer experiment is to measure with cantilevers[cantilever] a 3D printed test mass, the same type of polymers that 3D printed satellites would eventually be made out of.

MakerSat-0 would be the preliminary mission to test the MakerSat idea in a more traditional manor. MakerSat-0 is an NNU research project with six team members (Dr. Parke, Dr.

Griffin, Mitch Kamstra, Braden Grim, Connor Nogales, and Aaron Ewing), with the duty of creating a single 1U-class CubeSat. The satellite will likely be Idaho's very first CubeSat to be in space. MakerSat-0 would be launched with NASA's ELaNA rocket with a traditional solid metal frame with all components in a vertical stack inside.

MakerSat-0's experiment involved two parts: a radiation counter board constructed by Caldwell High School (CHS) and a polymer degradation board created by Connor Nogales. The CHS board would have a sensor that would count radiation particle hits over a certain window of time (measured in seconds), while Connor Nogales's board would measure the degradation of polymers using cantilevers. To measure the polymer degradation using cantilevers the board must be oscillating at its natural frequency of approximately 75 Hz with a surface transducer[surface transducer], keeping it in a stable oscillating state. At the end of the cantilevers there is a known sample size of 3D printed mass and as the mass of the plastic erodes away from the harsh conditions of space (including free oxygen radicals) the oscillation of the cantilevers will change noticeably. This oscillation will present itself as an easy to read voltage change.

Initial Project Concept and Progression.

The MakerSat project existed long before Aaron Ewing joined the team, and will continue to exist for a long time. The original members of the team were Dr. Parke and Dr. Griffin as the advisers. Mitch Kamstra, Braden Grim, and Keith Moilanen were the initial student researchers. Kamstra and Grim stayed with the project while Keith Moilanen ended up graduating and moved on. The goal of the project was to always create a 3D-printed CubeSat into outer space. The name, design, and overall function varied a fair bit until around the time Connor Nogales and Aaron Ewing joined the project.

Connor Nogales and Aaron Ewing began work on MakerSat in late 2015 on the science payloads, which later became known as science boards or experiment boards. There was room for four unique experiments on the science boards since the CubeSat took the form of a cube, which has six sides, and two of the sides were used for implementing the radio and Hub boards. Figure 1 and 2 show the preliminary CAD drawings of MakerSat, created by Braden Grim.

Once brought on, the newly hired researchers went to work on finding interesting science experiments. The team settled on a camera and a way to measure the polymer degradation that space would have on the 3D printed CubeSat. The rest of that calendar year was spent developing a way to measure polymer loss and finding a camera that would fit the project needs. Sensors that would measure vaporized gas were considered; since 3D printed polymer has trapped air inside, and being placed in a vacuum would release those gasses, was soon discarded because most sensors were too big, expensive, and/or required an atmosphere to function.

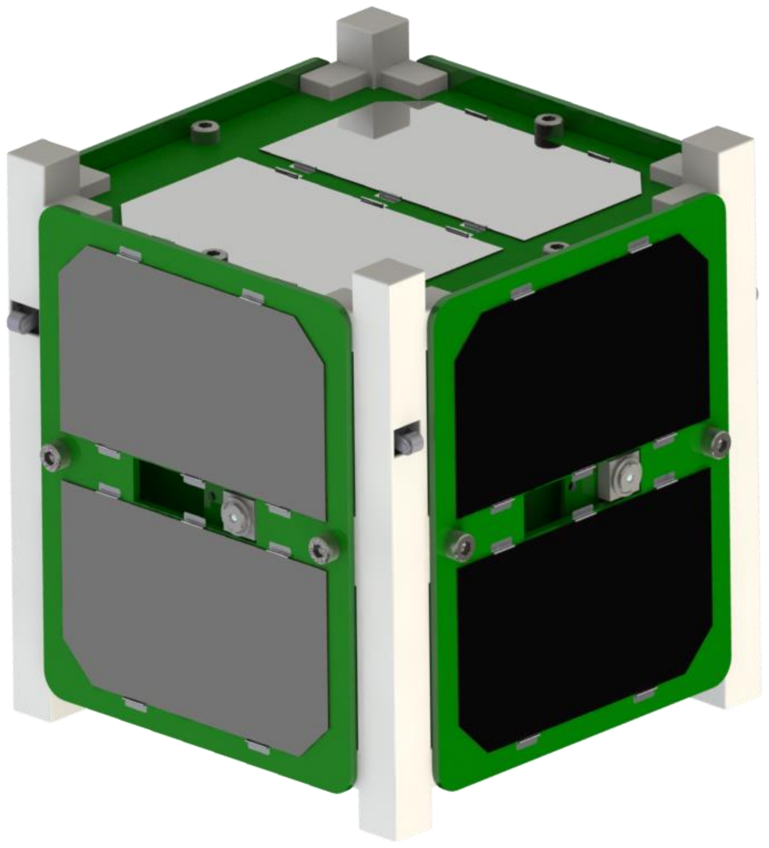


Figure 1: MakerSat CAD

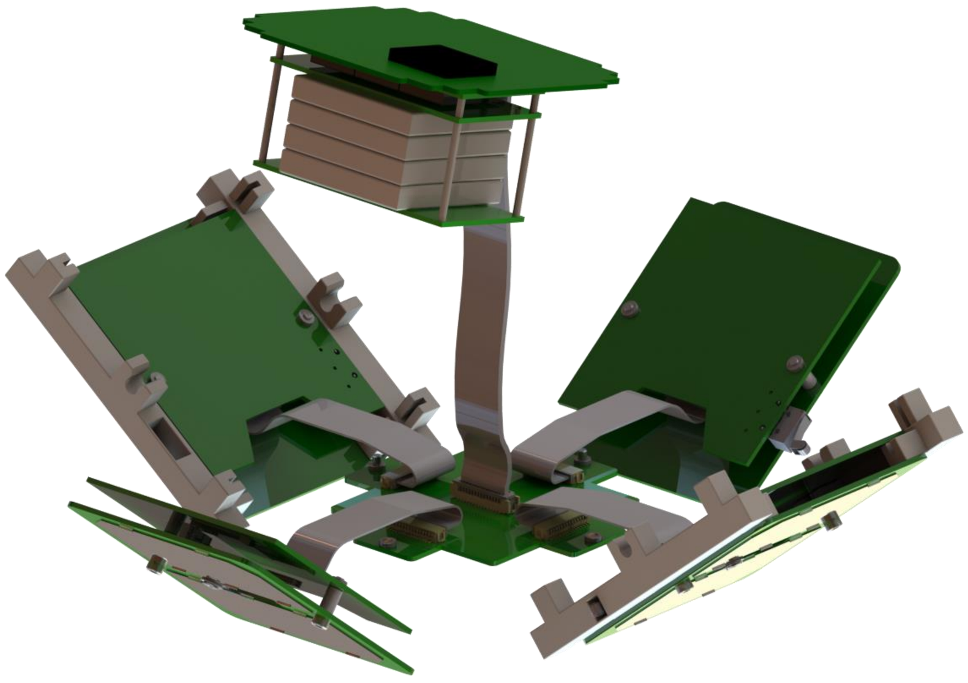


Figure 2: MakerSat CAD Exploded View

Eventually, it was determined that the polymer degradation was to be measured using a sample 3D printed polymer mass at the end of a cantilever, which was forced to vibrate at the natural frequency of a small rotating motor. The camera was proving difficult to solve, due to the quality of cameras that size for a reasonable price and the challenge of getting a decent image from that camera, through an embedded microcontroller, through a radio link back to Earth. Towards the end of the Spring 2016 semester, Aaron was mainly working on the camera development and Connor was working toward a functioning cantilever.

At this stage of the project, this research project became Aaron's senior project for the partial fulfillment of the requirements for the degree of bachelor of arts in Computer Science. The entire summer had work, specifically working with the communication pathways between all the components of MakerSat, trying to figure out the camera issue, and honing the skills in embedded programming and finite state machine design, since Aaron Ewing was largely unfamiliar with these when he started on the project. The project was ultimately altered significantly due to an earlier launch opportunity that the team accepted in early August, 2016.

Project Evolution

The focus of the project changed dramatically when the MakerSat research group was offered a position on a launch in mid-2017, instead of mid-late 2018. Considering that a delayed launch was likely, even up to 2019, the MakerSat research group decided that the best approach was to take the offer for the mid-2017 launch.

The problem that presented itself when the MakerSat team made its choice was the accelerated timeframe. For the launch to take place in mid-2017, the team needed to have the entire CubeSat completed, tested, and ready to launch in early November of 2016. Considering that the project now had to be functional within a 3-4-month time frame instead of a 15-month time frame, radical changes were made. Work on the camera ceased, Caldwell High had the choice of keeping up with the mission or be left behind, and everyone adopted an “all hands on deck” mindset. Every member of the team worked an excessive number of hours, making it their life goal to see this project through to the end, and eventually the project finished. At the present time (April, 2017), MakerSat is ready for launch.

Required Tasks and Challenges

With a little over three months to develop the basic working code, the required tasks of the project were re-evaluated. The goals of the project are thus:

1. Pass the NASA criteria for going into space.
2. Successively send MakerSat into space.
3. MakerSat successfully turns on and communicates with Earth.
4. Transmit packets of science data (science board payload data) to Earth.
5. Successfully decode that raw science packet of data and turn it into useful information.
6. Repeat steps 4 and 5 until the satellite’s eventual re-entry into the atmosphere.

NASA’s criteria are rather simple: it must not interfere with any other rocket payload in any way, it must survive the launch, and it must not emit any radio signals for more than 45 minutes after deployment (similar situation as to when an airline asks someone to turn off their phones and laptops before they lift off or land an airplane). Aaron Ewing was not involved in this part of the project, instead, it was largely the duties of NearSpace Launch and Braden Grim. NearSpace

Launch controlled the radio and Braden Grim testing the physical structure of MakerSat (since Grim was the sole Mechanical Engineering student of the group).

A successful launch was the responsibility of the those launching the rocket into outer space and our Lord and savior to oversee it.

The initial communication with Earth is initialized when power is transferred to NearSpace Launch's radio. Thankfully, NSL has had a rather successful career making CubeSat simplex and duplex radios for other interested parties. The phrase simplex and duplex just mean that the radio is one way (output) only or two ways (input and output), respectfully.

Aaron Ewing was mainly involved with the goals listed in steps 4 and 5, as mentioned above.

Goal 4 was the most important and involved the most risk. The entire finite state machine had to work properly. Figure 3 below shows the finite state machine of MakerSat.

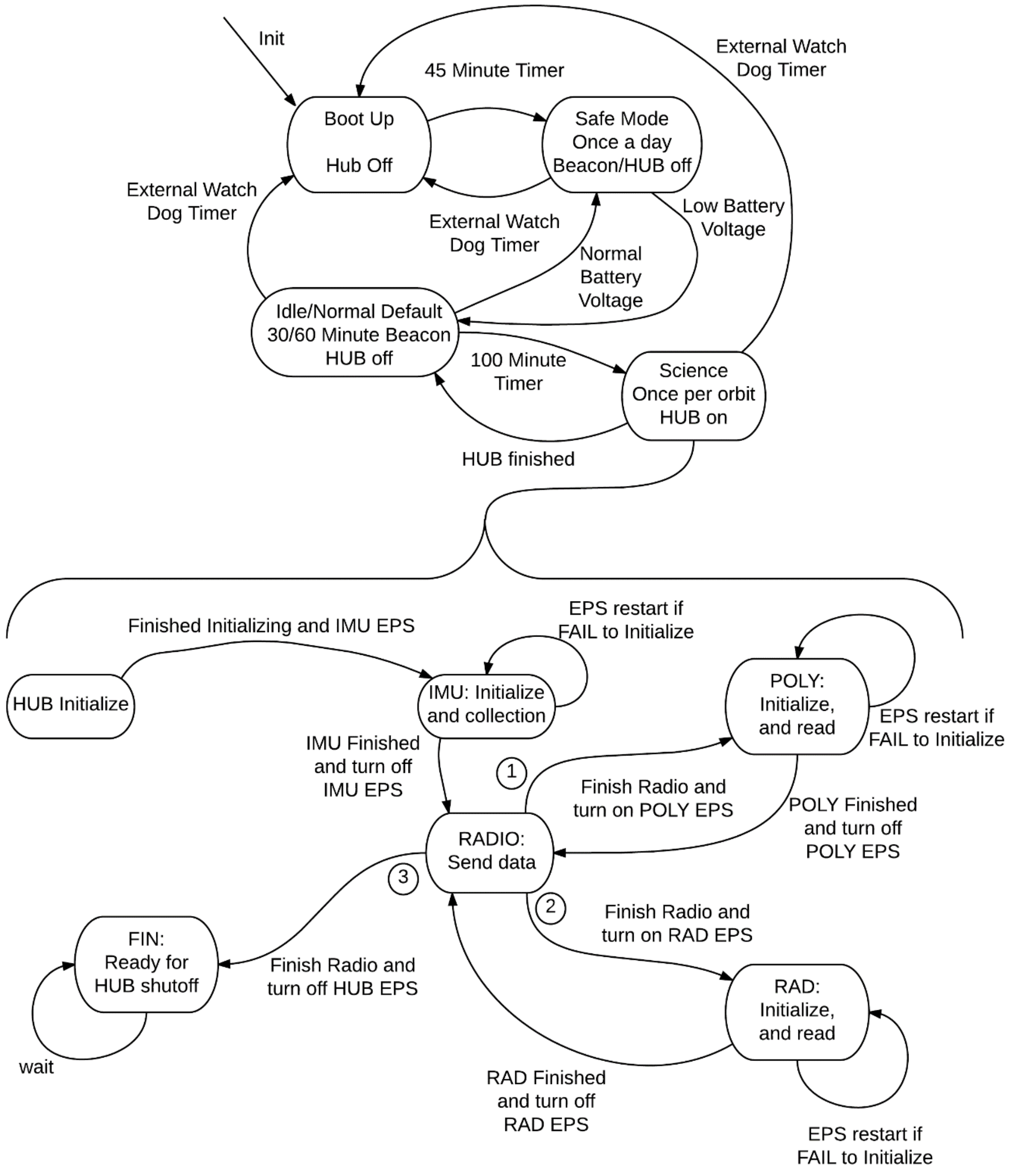

Init
45 Minute Timer
External Watch Dog Timer
Boot Up
Hub Off
Safe Mode
Once a day
Beacon/HUB off
External Watch Dog Timer
External Watch Dog Timer
Low Battery Voltage
Normal Battery Voltage
Idle/Normal Default
30/60 Minute Beacon
HUB off
100 Minute Timer
Science
Once per orbit
HUB on
HUB finished
Finished Initializing and IMU EPS
EPS restart if FAIL to Initialize
HUB Initialize
IMU: Initialize and collection
POLY: Initialize, and read
EPS restart if FAIL to Initialize
IMU Finished and turn off IMU EPS
1
Finish Radio and turn on POLY EPS
POLY Finished and turn off POLY EPS
RADIO: Send data
3
2
Finish Radio and turn on RAD EPS
FIN: Ready for HUB shutoff
Finish Radio and turn off HUB EPS
RAD: Initialize, and read
wait
RAD Finished and turn off RAD EPS
EPS restart if FAIL to Initialize

Figure 3: Finite State Machine Diagram

The order of execution of the state machine is as such:

1) The onboard solar panels charge the battery to the point it can turn on the On-Board-Computer[OBC]. Once the batteries reach an acceptable level, MakerSat will wait 45 minutes to accommodate NASA regulations, regardless if this is the first time or the n^{th} time restarting. One of the OBC's primary tasks is to manage the current voltage levels from the 5 solar panels (there is no solar panel on one of the boards because the radio takes space on that board). If the voltage drops below a certain point, the OBC will turn off MakerSat to preserve the battery and attempt to charge the battery back up to an acceptable level. The OBC will attempt to keep itself on and to transmit a single health packet a day (24-hour period, regardless of orbit) as much as possible. Doing so lets the team on Earth know that the satellite has low power, but is still operational while also saving the CubeSat from restarting the OBC without a guarantee of it turning back on.
2) Once the OBC turns on and is in a situation where there is enough power to do science experiments, it will turn on the Hub. The Hub is the microcontroller board that connects the Science Boards to the Radio (please refer to Figure 2 again for visual example). The Hub's job is to send a request to the EPS to provide power for a science board, read in the science board's experiment data, save it, request science board power to be turned off, and then send data to the radio for transmission to Earth. Once the Hub does this for all of the science boards, it will request for the OBC to turn the Hub itself off (as to save power). The Hub communication is largely where Aaron Ewing helped with the MakerSat project.
3) When the Hub requests power from the EPS to turn on a particular science board, the science board will turn on and immediately go about its science task (which is either

Connor Nogales's or Caldwell High's experiment). Once done, it will send a ready signal to the Hub, and the Hub will respond in turn, and then it will send this data to the HUB using the SPI[SPI] communication protocol. Once done sending information to the Hub board the science board drops the ready signal to 0 volts, the Hub will turn it off as to restart it in the next orbit (if there is enough power that is). The majority of the work done on the science boards was done by Connor Nogales and CHS team. Aaron Ewing assisted with the CHS team, but they did the brunt of the work and deserve all of the credit for the completion of the code. Their board revolved around a radiation counter purchased from NSL, which could measure the radiation strikes on its sensor within a given amount of time (a fraction of a second in this case) for particles per second. Description of the Polymer board is mentioned in the background section of this paper.

4) When the Hub board has the data from a science board it will transform it into a series of bytes, known as a packet, that the radio will understand. This involves prefixing a series of bytes to the front (0x50, 0x50, 0x50, 0x0C) to tell the radio that the Hub wishes to communicate and that it wants to transmit a packet to Earth. The Hub will then send a packet of 39 bytes (3 0x50 bytes to turn on radio, 0x0C byte to declare the intention of sending a packet, and 35 bytes of actual data) to the radio using the UART protocol (Universal Asynchronous Receiver/Transmitter). The hub will wait for the radio to send out the information, and when the ready line goes low (indicating that the radio can receive another packet) it will transmit the packet. Once all the desired packets are transmitted out of the radio, the Hub will delete all of its saved contents and go to the next science board. If every science board has gone through this process it will then request a power off by the OBC. As mentioned before, Aaron Ewing and Mitch Kamstra

did a large part of Hub code. Kamstra developed the circuit board and a large portion of the task scheduler of the Hub while Aaron Ewing did a smaller portion of the task scheduler and most of the communication protocols, packetizing, and other pieces of the Hub code. Both Mitch Kamstra and Aaron Ewing were heavily involved in the actual interfacing between Hub and Science Board.

5) When the radio receives a packet, it will transmit it to the GlobalStar network, a network of satellites in orbit used for satellite phones and low-speed communications that NSL uses, using a simplex radio that NSL developed. The GlobalStar Network then gathers all of the packets and transmits that information to Earth. It is important to note that the Global Star network will not receive everything that the radio transmits to it. Corruption, dropping, or losing packets often happen during communication. Precautions were taken to make sure that what data reaches Earth makes sense to the MakerSat research team, which is discussed in stage six.

6) The very last stage in the process is to convert the downloaded data into readable information. This process is the least stressful since fixing a problem is relatively simple on Earth, where problems in the other steps mentioned above cannot be fixed once the satellite is in orbit. A Matlab program was written so that the CSV (Comma Separated Values) files which are The Matlab code was originally being written by Aaron Ewing, but Dr. Griffin did the most work due to the rapidly approaching deadline and the fact that Dr. Griffin was more accustomed to Matlab programming than Aaron Ewing. Aaron Ewing started developing the code and did the end testing and alterations, but Dr. Griffin did the majority of the coding in the middle so that the project could be more rapidly produced.

Implementation

The code in Appendix A is either written in Embedded C using the Code Composer IDE[IDE] or in Matlab. The large distinction, that will become more clear as time progresses, is that Embedded code is designed for a certain type of microcontroller. It interfaces directly with the hardware, which changes between all computing systems. This means that the code will be at least slightly different for every type of microcontroller, no matter if the code is produced for the same microcontroller family from the same company. The particular microcontroller used was the MSP430FR6989 from Texas Instruments, where the MSP430 is the class of the Mixed Signal Processor (MSP) by Texas Instruments (TI), the FR stands for Ferroelectric memory (explained later), and 6989 is a specific model of MSP430. The code written in Appendix A may not be completely written by Aaron Ewing, but was where Aaron Ewing was involved. This includes the Matlab code and the Linker File[Linker] where changes had to be made for memory save locations. Figure 4 below shows a TI MSP430FR6989 Launchpad, where the majority of the programming on this project took place. The pins below are I/O ports, where GND is Ground, 3.3V and 5V are 3.3 and 5 Volt pins respectfully, and other pins denoted as Port.Pin; meaning that P2.1 would denote the Port 2 Pin 1. The black chip in the center is the microcontroller, everything else is an I/O interface to interact with and use the microcontroller.

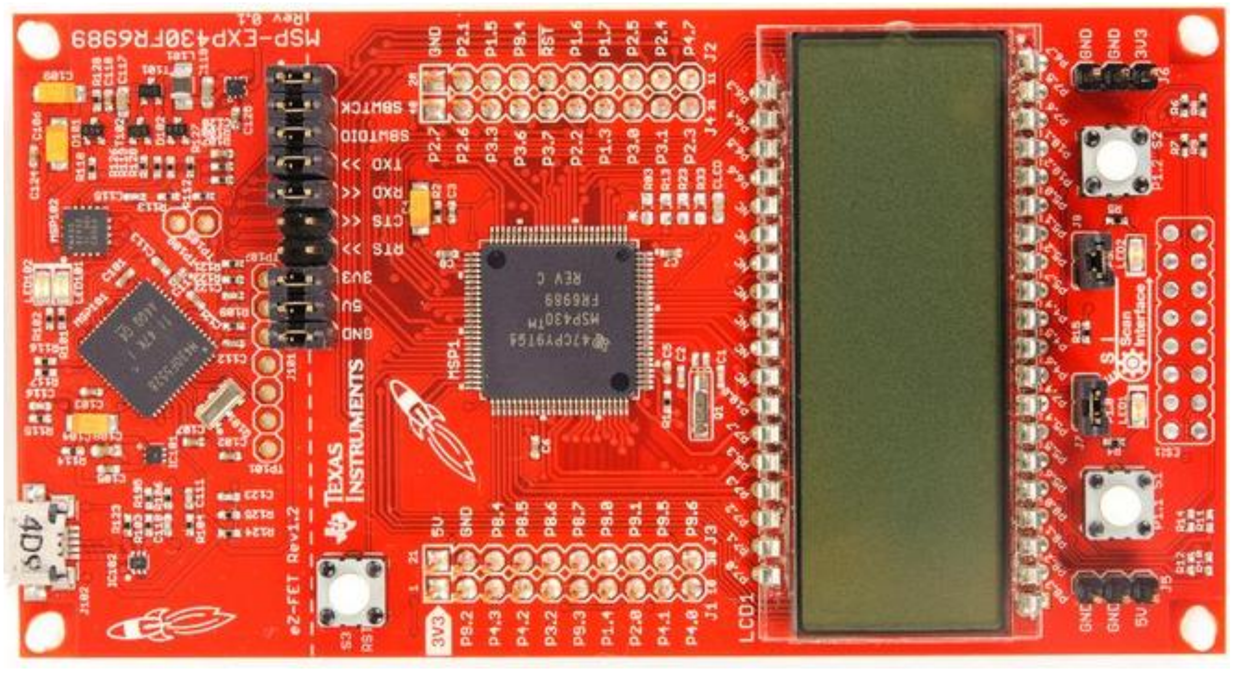


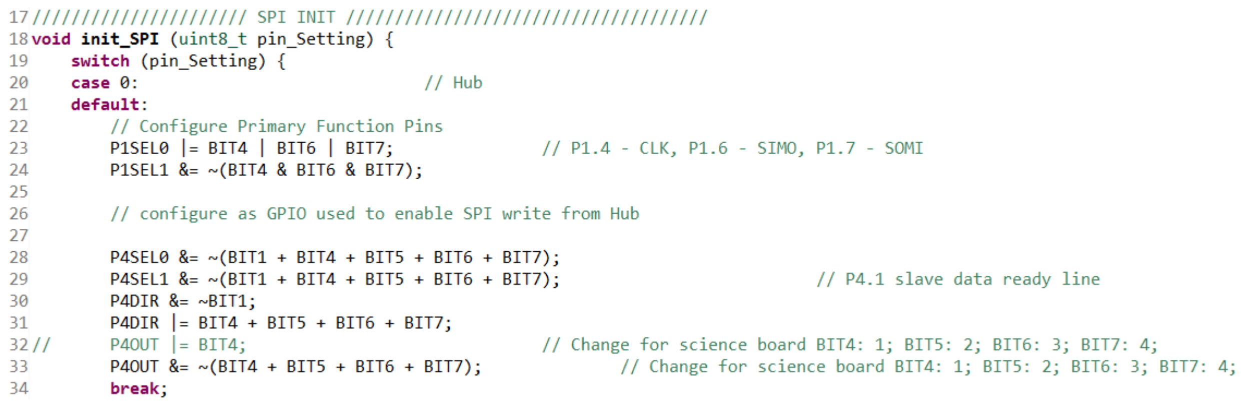
Figure 4: MSP430FR6989

Coding SPI module:

Four separate functions exist in the SPI module: initialize SPI communication (init_SPI), write 1-byte using SPI (write_uint8_SPI), 2-byte write SPI (write_uint16_SPI), and read_SPI. Please refer to Appendix A SPI_Polling for source code.

init_SPI:

The code for init_SPI was made so that multiple different microcontrollers could use it. 3 cases existed for three types of boards: Hub board (which was a board specifically designed using the MSP430FR6989 microcontroller), the Caldwell High School team (which used a Launchpad for the MSP430FR5969 provided by Texas Instruments that did not have to be uniquely designed for MakerSat), and Connor Nogales's experiment, which used SPI communication itself to communicate with the motors used to vibrate the board. The example here will focus on the Hub case (case 0), though it is very similar to the other cases.

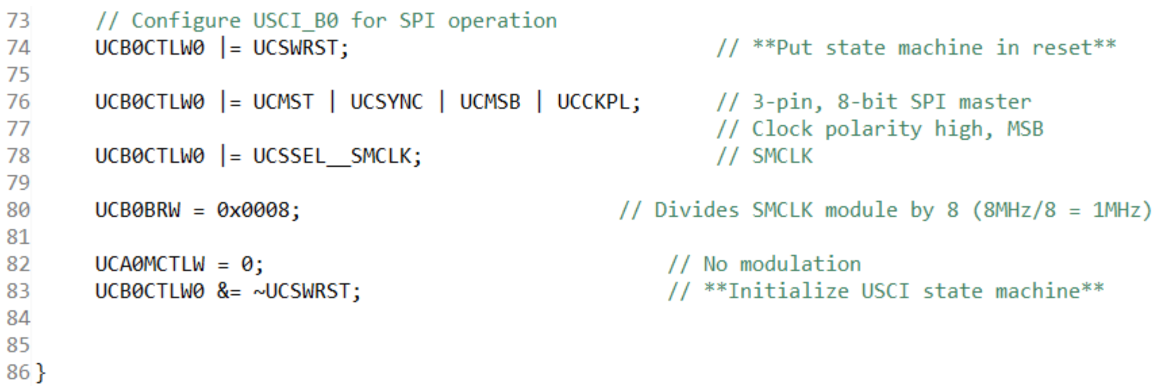


```
17 ///////////////////// SPI INIT ///////////////////////////////////
18 void init_SPI (uint8_t pin_Setting) {
19     switch (pin_Setting) {
20     case 0:                              // Hub
21     default:
22         // Configure Primary Function Pins
23         P1SEL0 |= BIT4 | BIT6 | BIT7;              // P1.4 - CLK, P1.6 - SIMO, P1.7 - SOMI
24         P1SEL1 &= ~(BIT4 & BIT6 & BIT7);
25 
26         // configure as GPIO used to enable SPI write from Hub
27 
28         P4SEL0 &= ~(BIT1 + BIT4 + BIT5 + BIT6 + BIT7);
29         P4SEL1 &= ~(BIT1 + BIT4 + BIT5 + BIT6 + BIT7);                           // P4.1 slave data ready line
30         P4DIR &= ~BIT1;
31         P4DIR |= BIT4 + BIT5 + BIT6 + BIT7;
32 //      P4OUT |= BIT4;                             // Change for science board BIT4: 1; BIT5: 2; BIT6: 3; BIT7: 4;
33         P4OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);             // Change for science board BIT4: 1; BIT5: 2; BIT6: 3; BIT7: 4;
34         break;
```

Figure 5: Init_SPI Case 0

The above code is heavily integrated with the specific microcontroller. Lines 23 and 24 set pins 4, 5, and 7 to the primary function in port 1. The primary function denoted in the 824-page userguide[1] is the state in which SPI communication can take place. The 176-page datasheet[2] denotes which pins connect to which SPI module (there are multiple different SPI channels in the event that multiple different devices need to communicate with the microcontroller). “|=” ORs bits together, while “&= ~()” does the opposite. ORing bits together will result in a1 if any of the bits (two or more) are 1, while the opposite results in a 0 if any of the bits are 0.

Lines 28-33 set up the other lines needed to communicate with SPI, where DIR sets the direction of the I/O ports (1 as output and 0 as input), and OUT as an output.

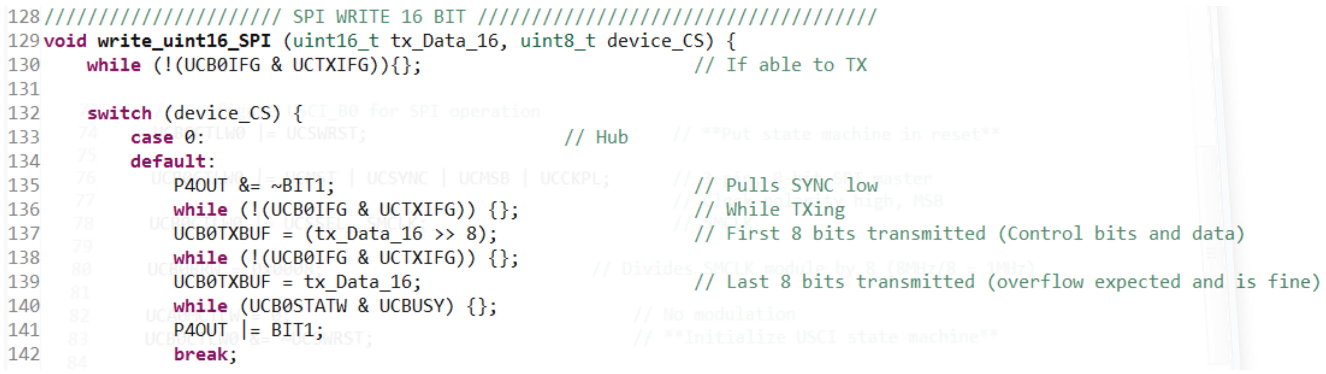


```
    // Configure USCI_B0 for SPI operation
    UCB0CTLW0 |= UCSWRST;                          // **Put state machine in reset**

    UCB0CTLW0 |= UCMST | UCSYNC | UCMSB | UCCKPL;  // 3-pin, 8-bit SPI master
                                                   // Clock polarity high, MSB
    UCB0CTLW0 |= UCSSEL__SMCLK;                    // SMCLK

    UCB0BRW = 0x0008;                      // Divides SMCLK module by 8 (8MHz/8 = 1MHz)

    UCA0MCTLW = 0;                               // No modulation
    UCB0CTLW0 &= ~UCSWRST;                       // **Initialize USCI state machine**


}
```

Figure 6: SPI_Init clock

The last part of the code places the microcontroller into a state where adjusting the internal clock is possible, starting the SPI module inside of the microcontroller, setting that SPI clock to 1MHz, and then putting the MSP back into a useful state again.

Write_uintX_SPI

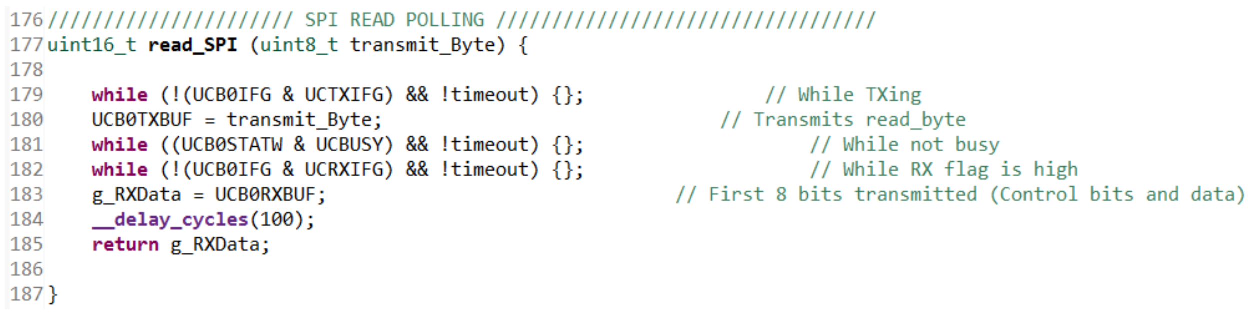


```
128 ////////////////////// SPI WRITE 16 BIT /////////////////////////////////////
129 void write_uint16_SPI (uint16_t tx_Data_16, uint8_t device_CS) {
130     while (!(UCB0IFG & UCTXIFG)){};                         // If able to TX
131 
132     switch (device_CS) {
133         case 0:                         // Hub
134         default:
135             P4OUT &= ~BIT1;                                 // Pulls SYNC low
136             while (!(UCB0IFG & UCTXIFG)) {};                // While TXing
137             UCB0TXBUF = (tx_Data_16 >> 8);                  // First 8 bits transmitted (Control bits and data)
138             while (!(UCB0IFG & UCTXIFG)) {};
139             UCB0TXBUF = tx_Data_16;                         // Last 8 bits transmitted (overflow expected and is fine)
140             while (UCB0STATW & UCBUSY) {};
141             P4OUT |= BIT1;
142             break;
```

Figure 7: write_SPI

Both the 8 and 16-bit writing SPI is the same, except that in the 16-bit version lines 137 and 138 are both included, where it is not in the 8-bit version. This function has two input parameters, a variable that holds the data to be sent through the SPI, and the device identifier (so that the SYNC pins are correct). The program first checks to see that there is no data coming through the SPI line (line 130), and then it pulls the SYNC line low (135). The SYNC line lets the other device know that the first device wishes to write to it. The function will then wait for the other device to be ready to receive (136). In the 16-bit version of the function, it will perform a bit shift and send the first byte out (lines 137 and 138). In the 8-bit version of the function, it will not perform a bit shift. The function will then send out the byte (if it is a 16 bit, the missing first byte will trigger the overflow flag, but this is the intended result). The function will then make sure that it was sent and the line is no longer busy (line 140), where it will pull the SYNC line high (line 141), generating the done signal, and leave the function.

Read_SPI



```
176 ///////////////////// SPI READ POLLING ////////////////////////////////
177 uint16_t read_SPI (uint8_t transmit_Byte) {
178
179     while (!(UCB0IFG & UCTXIFG) && !timeout) {};                // While TXing
180     UCB0TXBUF = transmit_Byte;                              // Transmits read_byte
181     while ((UCB0STATW & UCBUSY) && !timeout) {};                    // While not busy
182     while (!(UCB0IFG & UCRXIFG) && !timeout) {};                    // While RX flag is high
183     g_RXData = UCB0RXBUF;                               // First 8 bits transmitted (Control bits and data)
184     __delay_cycles(100);
185     return g_RXData;
186
187 }
```

Figure 8: read_SPI

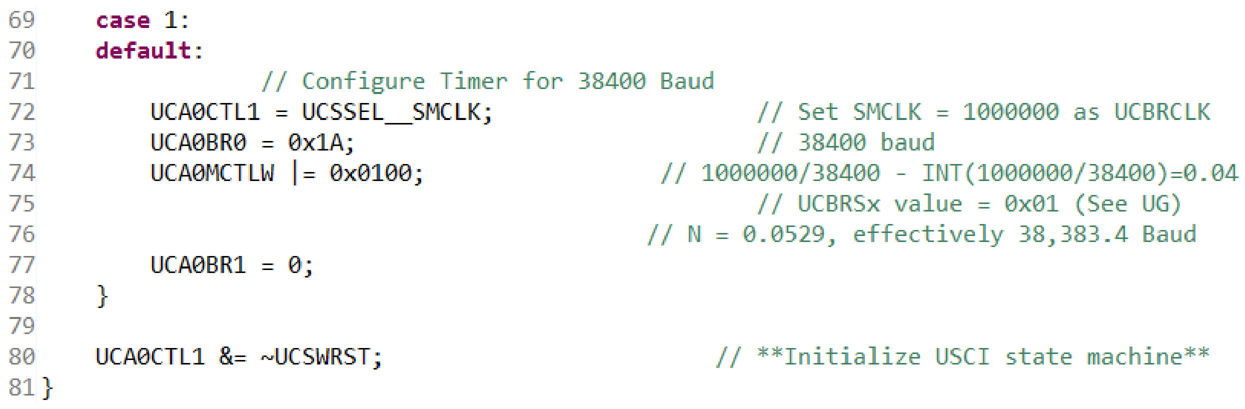
Read_SPI is unique because only the Hub uses it. SPI operatives using a Master-Slave relationship, where the Master (Hub) tells the slave (science boards) what to do. What happens then is that the Hub will write a byte to the slave, and every time that happens the Hub will receive a byte as well.

SPI is a shift register, where the receive register is linked to the write register to the other device. Essentially, when the device writes to another it will displace the 8 bits into the other device, and the 8 bits that were in the other device are sent down the other line to the sending device; they trade each other information.

When the Hub transmits the byte on the TX (Transmit) line (line 180), it will receive a byte on the RX (Receive) line. After the Hub checks that it received the information (lines 181 and 182), it will save the variable, wait for 100 clock cycles (line 184) (100 clock cycles at 1MHz is roughly a 0.0001-second delay) to have the communication lines settle to desired values, and then returns the value (line 185).

Coding UART module:

The UART module (Universal Asynchronous Receiver/Transmitter) has code that is very similar to the SPI module, even if the protocols are different. the most significant difference is that the code developer has to set a BAUD[BAUD] rate manually. Note that UART is often used for USB drive communication.



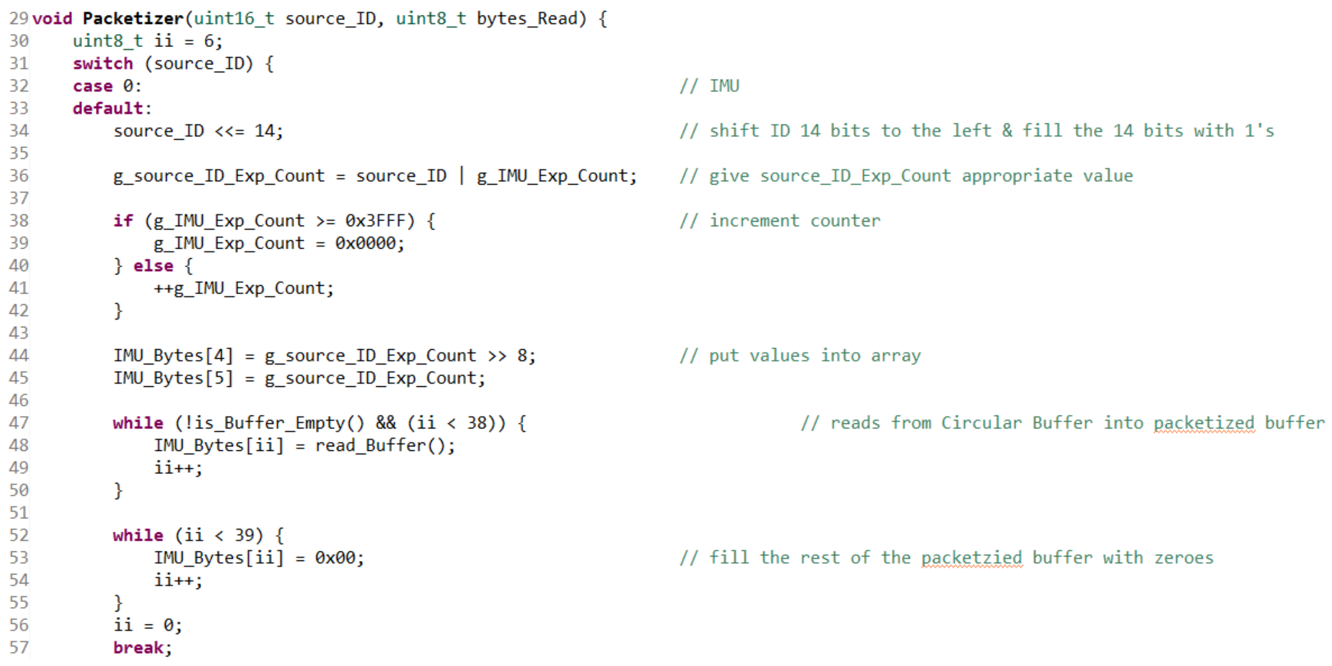
```
69    case 1:
70    default:
71                // Configure Timer for 38400 Baud
72        UCA0CTL1 = UCSSEL__SMCLK;                    // Set SMCLK = 1000000 as UCBRCLK
73        UCA0BR0 = 0x1A;                              // 38400 baud
74        UCA0MCTLW |= 0x0100;                  // 1000000/38400 - INT(1000000/38400)=0.04
75                                                     // UCBRSx value = 0x01 (See UG)
76                                             // N = 0.0529, effectively 38,383.4 Baud
77        UCA0BR1 = 0;
78    }
79
80    UCA0CTL1 &= ~UCSWRST;                          // **Initialize USCI state machine**
81 }
```

Figure 9: UART BAUD

The code in Figure 9 sets the BAUD rate to 38,400 bits per a second. It selects the 1MHz clock (line 72) and then subdivides it to 38400 using a 0x1A divisor. While this does not *exactly* equal 38,400 bits per a second (38,383.4 bits per a second instead), the MCTLW register modules the clock frequency so that on average it is equal to 38,400 bits. The other parts of the module are in Appendix A below, but the code is very similar to the SPI module.

Coding Packetizer module:

The packetizer function exists because the EyeStar radio requires information in a specific manner. The first 4 bytes must be 0x50 0x50 0x50 0x0C. This information turns on the OBC, letting it know to power on (0x50 0x50 0x50), and the 0x0C says that there is information to be sent out the radio. If the last byte was 0x0B, the command would notify the EPS (Electrical Power System) to turn on or off a science board. The next 35 bytes contains the data to be sent out of the radio. The number of bytes sent out is always constant, so if fewer than 35 bytes ought to be sent, padding bytes must be added. There was a decision made to have the next 2 bytes to represent the data source (there are multiple different experiments) and the packet number. Please see Figure 10 for visual aid.

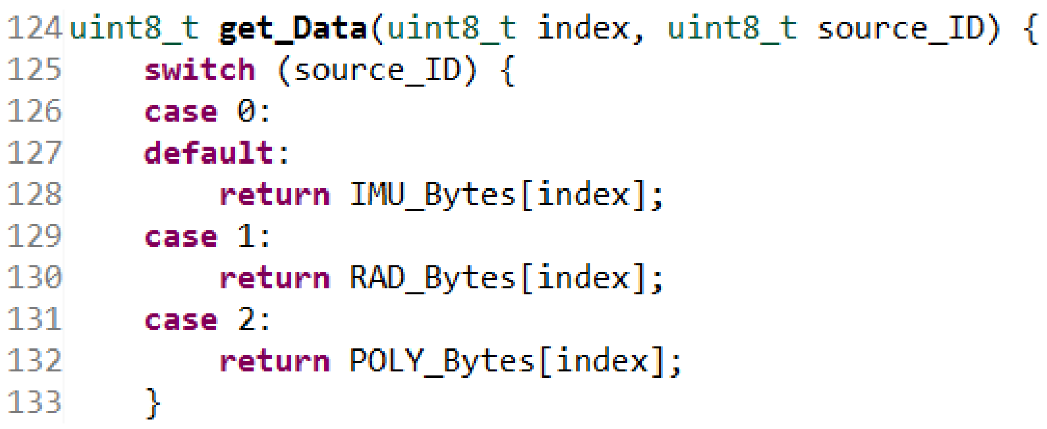


```
29 void Packetizer(uint16_t source_ID, uint8_t bytes_Read) {
30     uint8_t ii = 6;
31     switch (source_ID) {
32     case 0:                                                      // IMU
33     default:
34         source_ID <<= 14;                                        // shift ID 14 bits to the left & fill the 14 bits with 1's
35
36         g_source_ID_Exp_Count = source_ID | g_IMU_Exp_Count;     // give source_ID_Exp_Count appropriate value
37
38         if (g_IMU_Exp_Count >= 0x3FFF) {                         // increment counter
39             g_IMU_Exp_Count = 0x0000;
40         } else {
41             ++g_IMU_Exp_Count;
42         }
43
44         IMU_Bytes[4] = g_source_ID_Exp_Count >> 8;               // put values into array
45         IMU_Bytes[5] = g_source_ID_Exp_Count;
46
47         while (!is_Buffer_Empty() && (ii < 38)) {                           // reads from Circular Buffer into packetized buffer
48             IMU_Bytes[ii] = read_Buffer();
49             ii++;
50         }
51
52         while (ii < 39) {
53             IMU_Bytes[ii] = 0x00;                                // fill the rest of the packetzied buffer with zeroes
54             ii++;
55         }
56         ii = 0;
57         break;
```

Figure 10: Packetizer

The Source_ID in Figure 10 contains the data source (CHS experiment, polymer experiment, or IMU[IMU]). There is also a global variable (g_variable) that counts out how many packets have been sent. To conserve data being transmitted (due to the high financial cost of receiving information, around 10 - 20 cents per a byte), 3 bits represent the source, and 13 bits are used for the packet number. The packet number allows the team to identify any packets that were lost in transmission. 13 bits provide a sufficient number to account for any lost packets under any circumstance $((1111111111111_2) = (8191_{10}))$. If 5 bits are used $(11111_2) = (31_{10}))$, then only 32 packets could be sent without an overflow, versus 8192 packets. The source_ID replaces the 3 most significant bits of the packet counter, combining the 2 values into a useful variable that will work for the project and is more conservative than the 24 bits that would be used if no conversion took place (lines 36 – 42). Once those 6 bytes are in the array (0x50 0x50 0x50 0x0C plus 2 bytes), the data collected from the experiments are placed into the new array (line 47 – 50). An experiment can generate multiple packets worth of data, so it is likely for this function to loop multiple times, which increases the probability that there will be a need to pad the packet with “empty data.” Lines 52 – 55 are in charge of this.

This function took a while to implement in code. The decision on the size and contents of the first 6 bytes was in fluctuation for weeks. There was also a false start in the returning of the data out of the function. Figure 11 describes the final return result. Originally, a pointer was implemented so that there was no need to access the module to get every single byte of information to send out through the radio. The IDE[IDE] threw no errors, and it ran through without a problem, except the data at the pointer was corrupt somehow during transmission and neither the value it was meant to be or the value that an empty space represents (0x00). The approach in Figure 11 is currently used, since accessing the module repeatedly is a non-issue.



```
uint8_t get_Data(uint8_t index, uint8_t source_ID) {
    switch (source_ID) {
    case 0:
    default:
        return IMU_Bytes[index];
    case 1:
        return RAD_Bytes[index];
    case 2:
        return POLY_Bytes[index];
    }
```

Figure 11: Packetizer return

Coding Packet Analyzer:

The data collected needs to be parsed in order for it to be useful information, and doing it manually was not practical, so a Matlab script was made for that purpose. The Matlab program would read in CSV (Comma Separated Values) files that contain hundreds of lines that look similar to the following line:

```
0-788285,10/27/2016 8:24,465208,"""A10C40004BE9080080C54A4C4AB74B4E9600
000000000000004E5BEB45034D65495FC800""","""A1""",A1,"""A10C40004BE90800
80C54A4C4AB74B4E9600000000000000004E5BEB45034D65495FC800""",A10C400
04BE9080080C54A4C4AB74B4E9600000000000000004E5BEB45034D65495FC800
```

Figures 12 and 13 are a visual representation of same data placed into the program. The data was not gathered from the satellite but created to test the program. Figure 12 was supposed to show an increasing line when comparing frequency to same rates, and Figure 13 was supposed to the complete sinusoidal waves that a cantilever might produce on the actual mission. Figures 12 and 13 displayed the expected graphs, showing that the Matlab program worked with the test data.

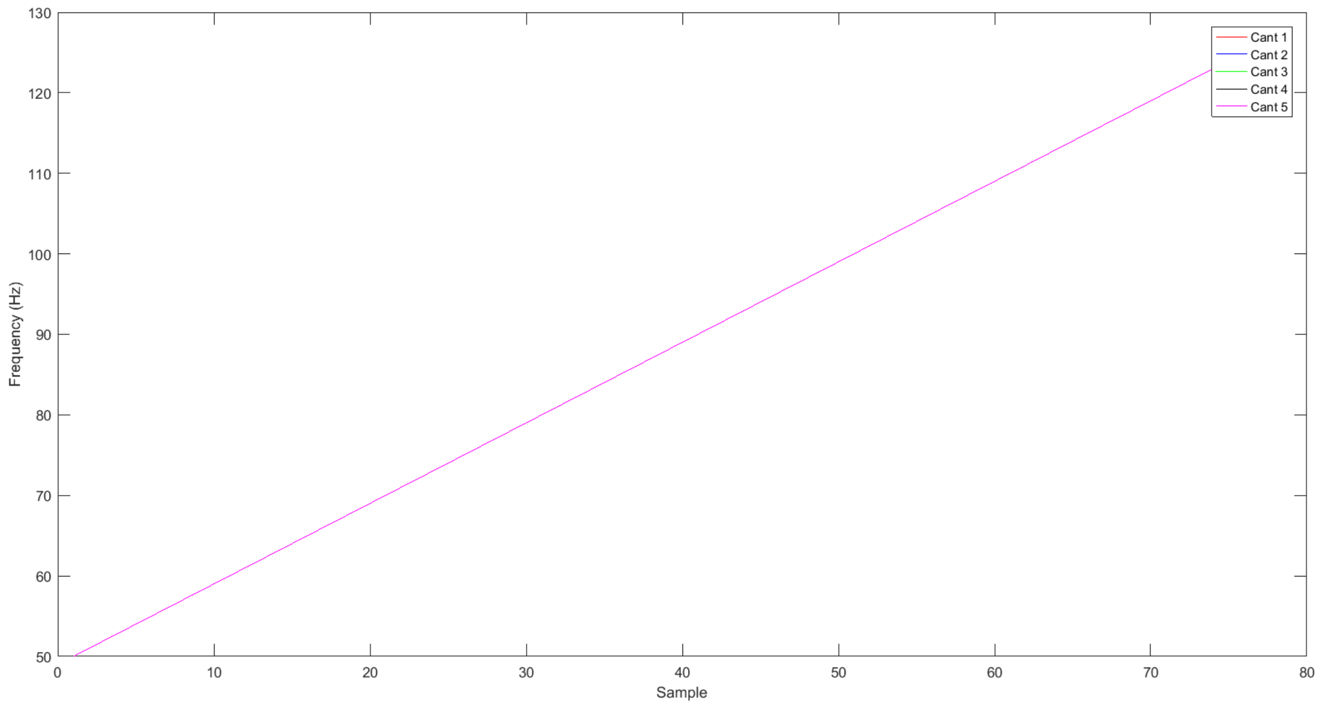


Figure 12: Polymer Period Input

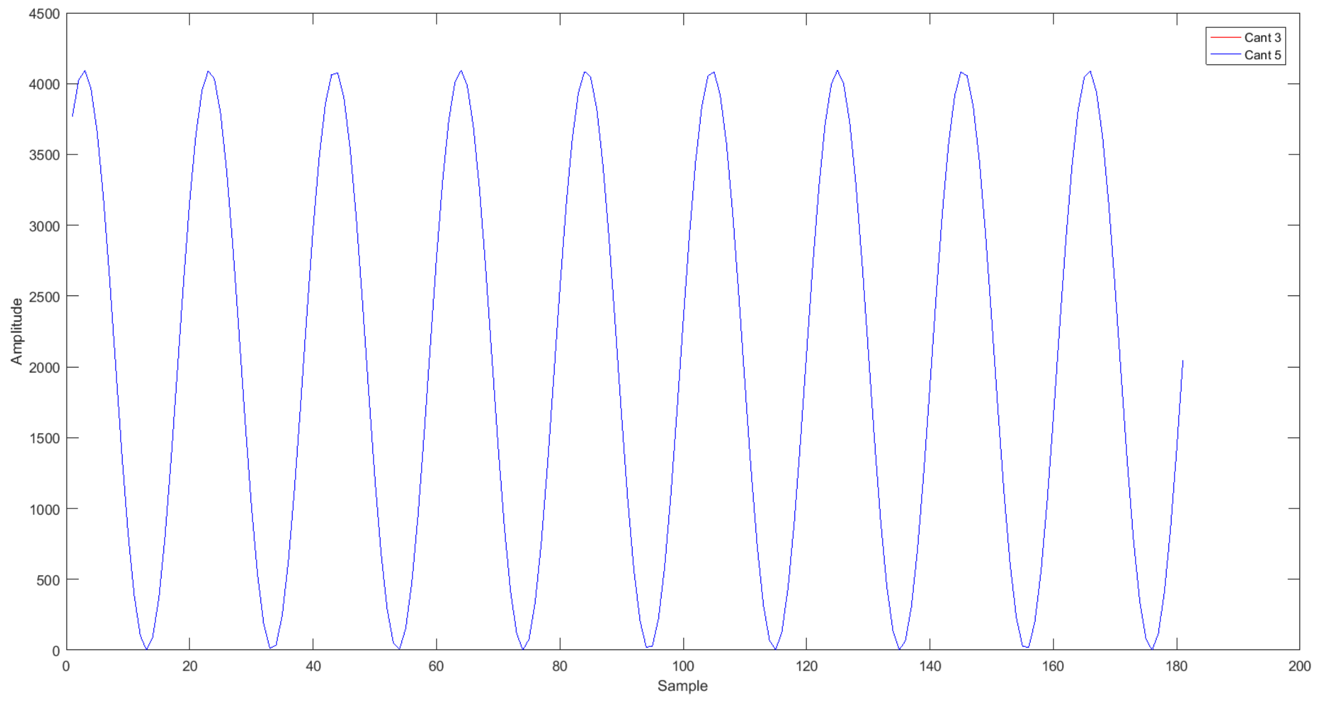


Figure 13: Polymer Waveform Input

Figures 14 and 15 display a test of the Matlab code with actual cantilever data from the satellite.

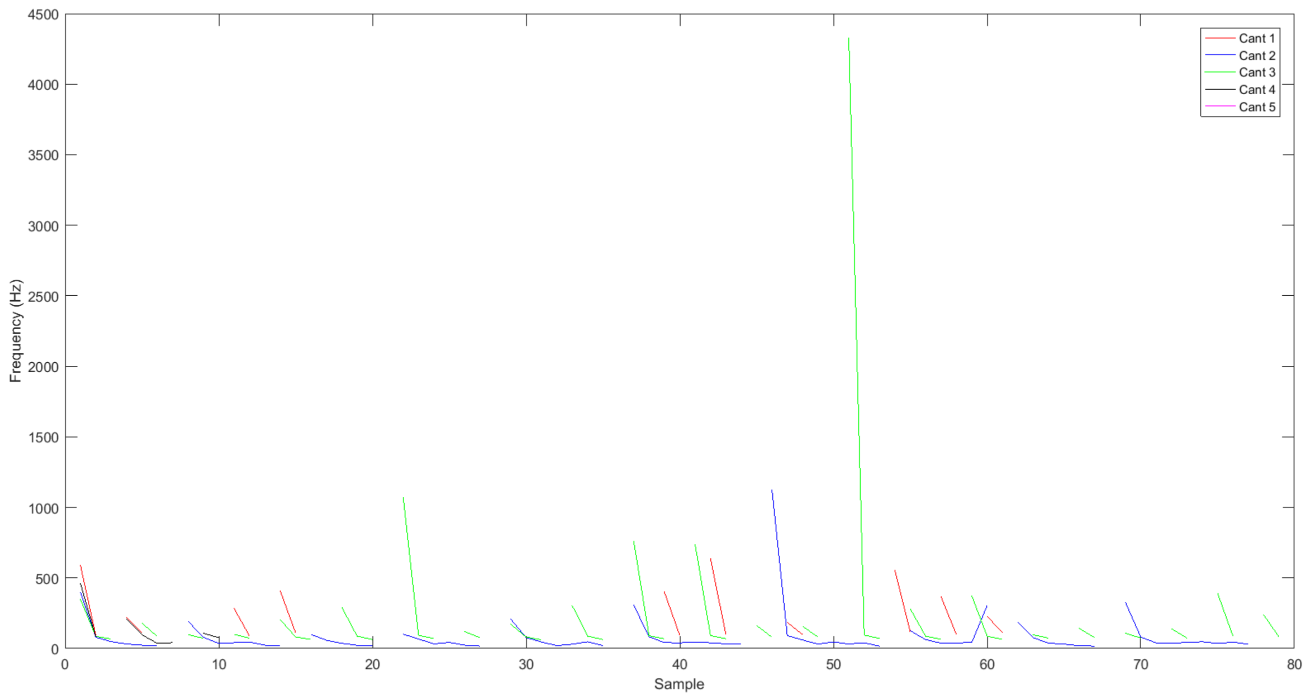


Figure 14: Polymer Period Output

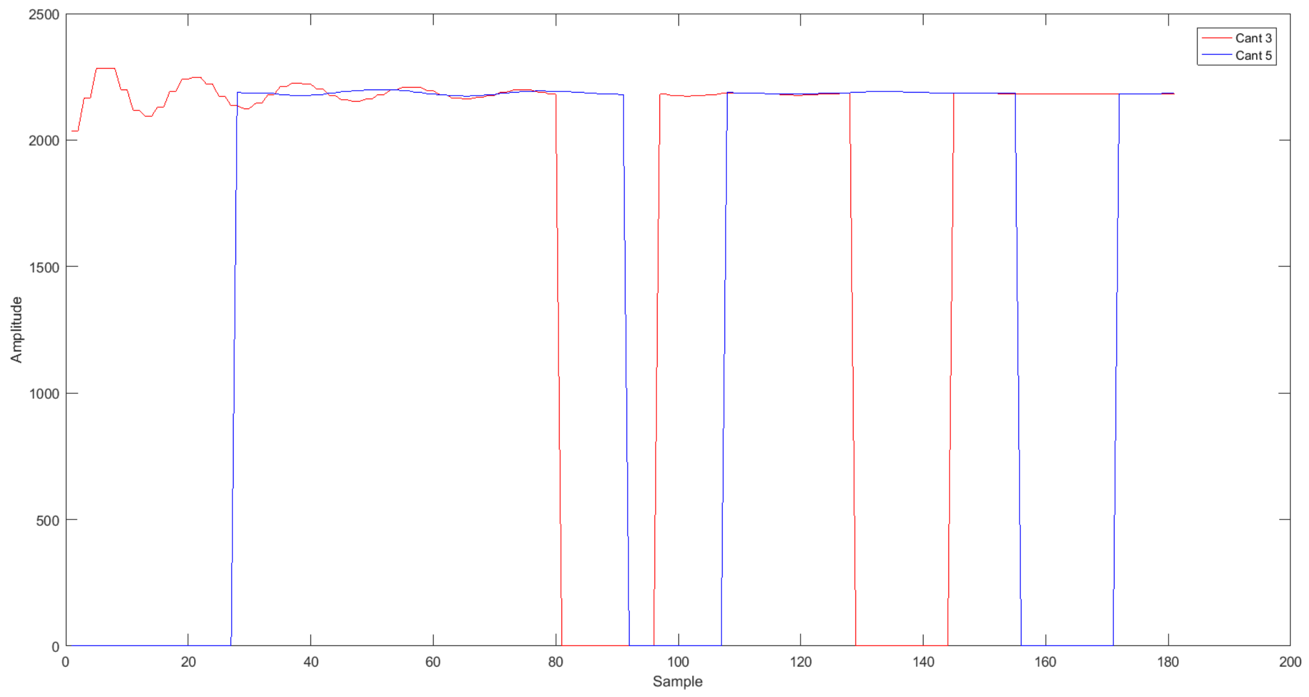
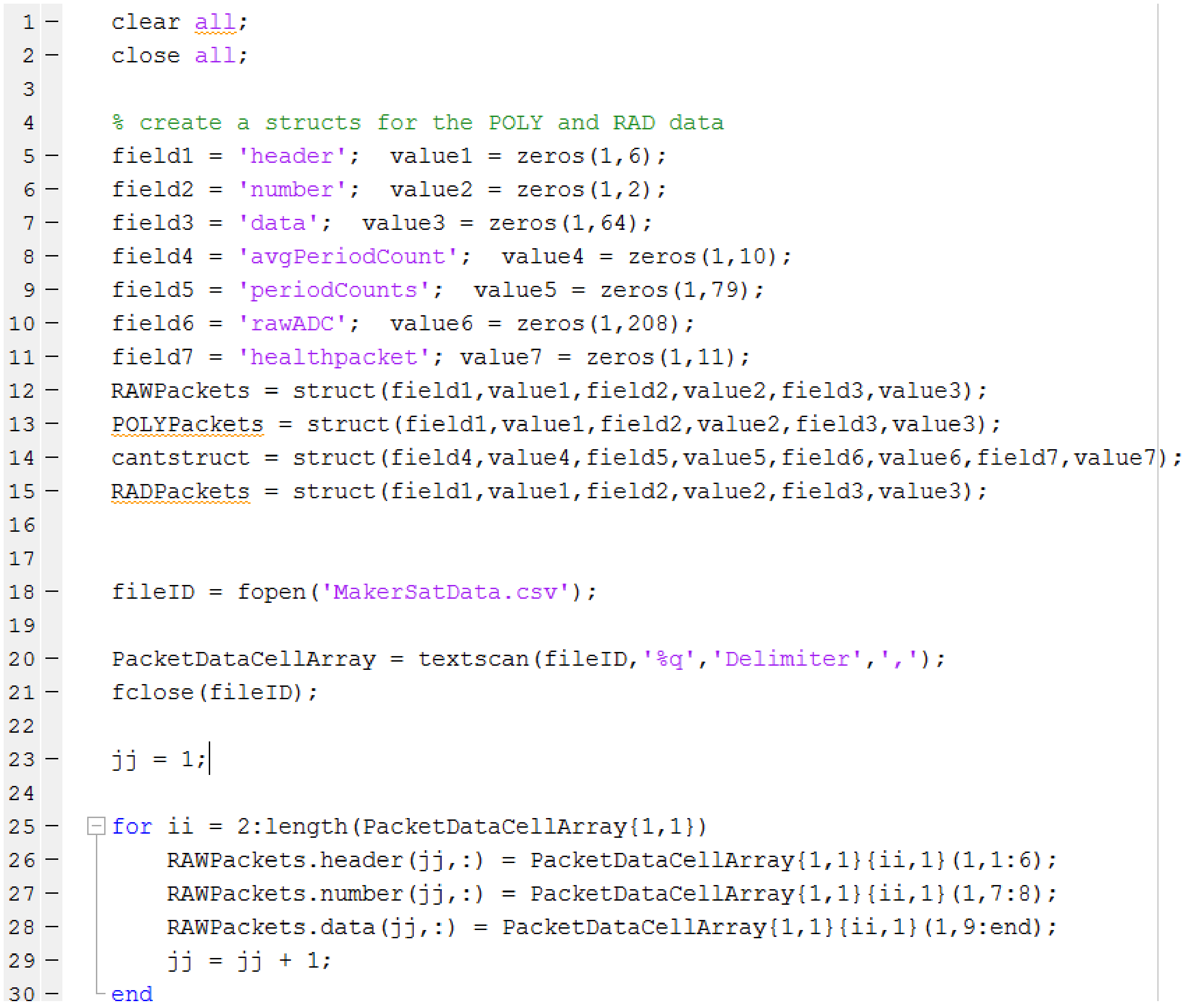


Figure 15: Polymer Waveform Input

Figures 14 and 15 accurately displayed the information from the Matlab cantilever test. It is important to note the polyer experiments were not conducted in orbit, but in the satellite lab, so the test data is expected to be skewed due to the different environments. Figure 14 shows spikes of a frequency higher than expected but drops quickly down to expected frequency ranges. Figure 15 shows the complete waveform sent from the polymer board. Zero amplitude represents missing packets, which the team wished to make obvious for testing purposes. When information is collected once the mission starts, missing packets can be represented by a straight line so that those segments could more accurately represent the missing data.

The Matlab code is shown in Figures 16 through 18, which convert the CSV files from the NSL server into useful information. Figure 16 shows the code that creates struts and fields to hold data (lines 5 – 15), takes in the CSV file (line 18), separate the values (line 20), close the file (line 21), and then separate all of the values into list of metadata (data about the data, such data timestamp) and data for the entire CSV file in a for loop (lines 25 – 30).



```
 1 -    clear all;
 2 -    close all;
 3
 4      % create a structs for the POLY and RAD data
 5 -    field1 = 'header';  value1 = zeros(1,6);
 6 -    field2 = 'number';  value2 = zeros(1,2);
 7 -    field3 = 'data';  value3 = zeros(1,64);
 8 -    field4 = 'avgPeriodCount';  value4 = zeros(1,10);
 9 -    field5 = 'periodCounts';  value5 = zeros(1,79);
10 -    field6 = 'rawADC';  value6 = zeros(1,208);
11 -    field7 = 'healthpacket'; value7 = zeros(1,11);
12 -    RAWPackets = struct(field1,value1,field2,value2,field3,value3);
13 -    POLYPackets = struct(field1,value1,field2,value2,field3,value3);
14 -    cantstruct = struct(field4,value4,field5,value5,field6,value6,field7,value7);
15 -    RADPackets = struct(field1,value1,field2,value2,field3,value3);
16
17
18 -    fileID = fopen('MakerSatData.csv');
19
20 -    PacketDataCellArray = textscan(fileID,'%q','Delimiter',',');
21 -    fclose(fileID);
22
23 -    jj = 1;
24
25 -    for ii = 2:length(PacketDataCellArray{1,1})
26 -        RAWPackets.header(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,1:6);
27 -        RAWPackets.number(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,7:8);
28 -        RAWPackets.data(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,9:end);
29 -        jj = jj + 1;
30 -    end
```

Figure 16: Packet Analyzer - reading in file

The program then calls a function to sort out the packet into the various different types of data (POLY stands for polymer experiment and RAD stands for the CHS experiment). The program finds the header that represents each (lines 4 and 16 in Figure 17) and has to do a couple of conversions between hexadecimal (base 16) and decimal (base 10) to get it into a readable format.

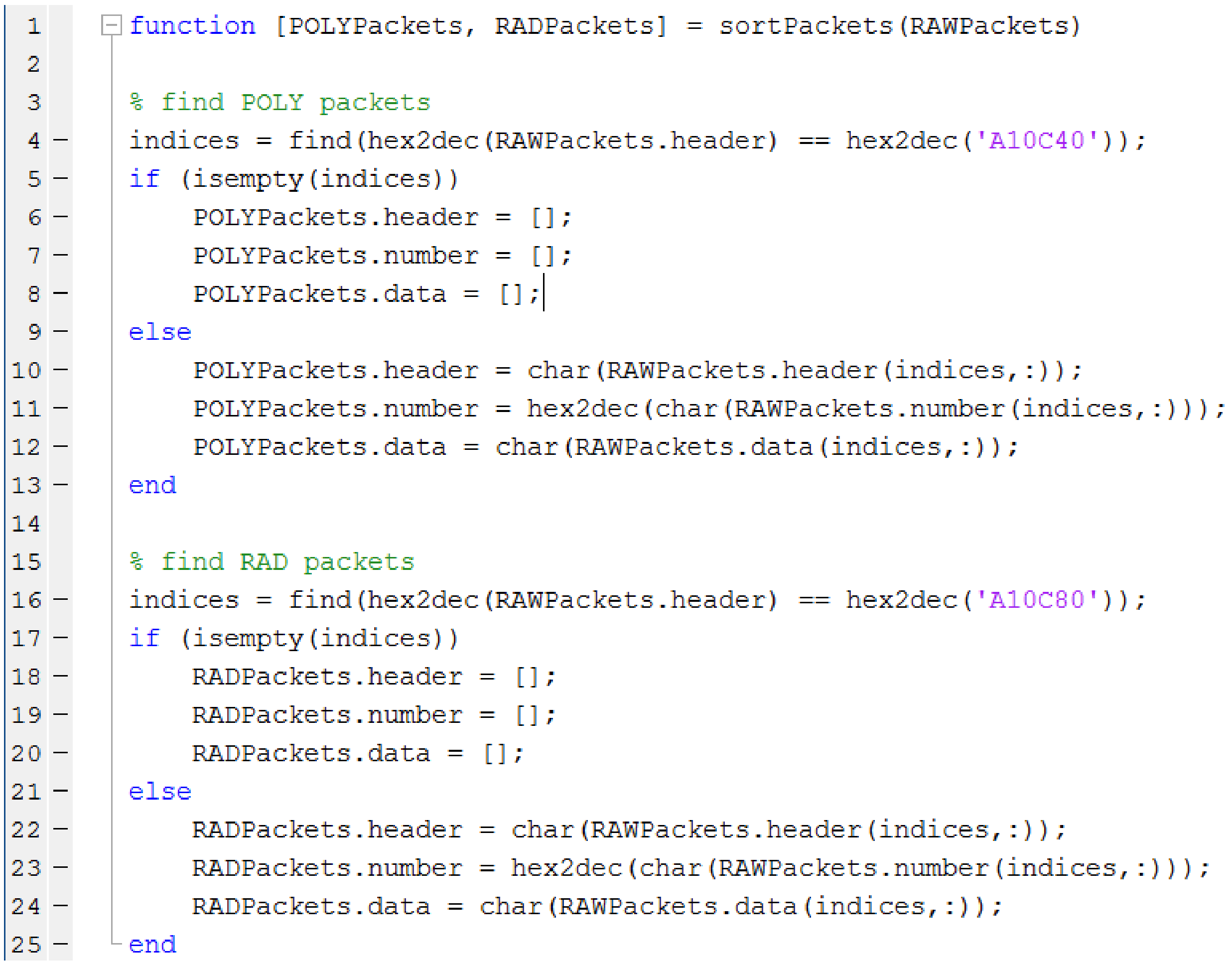
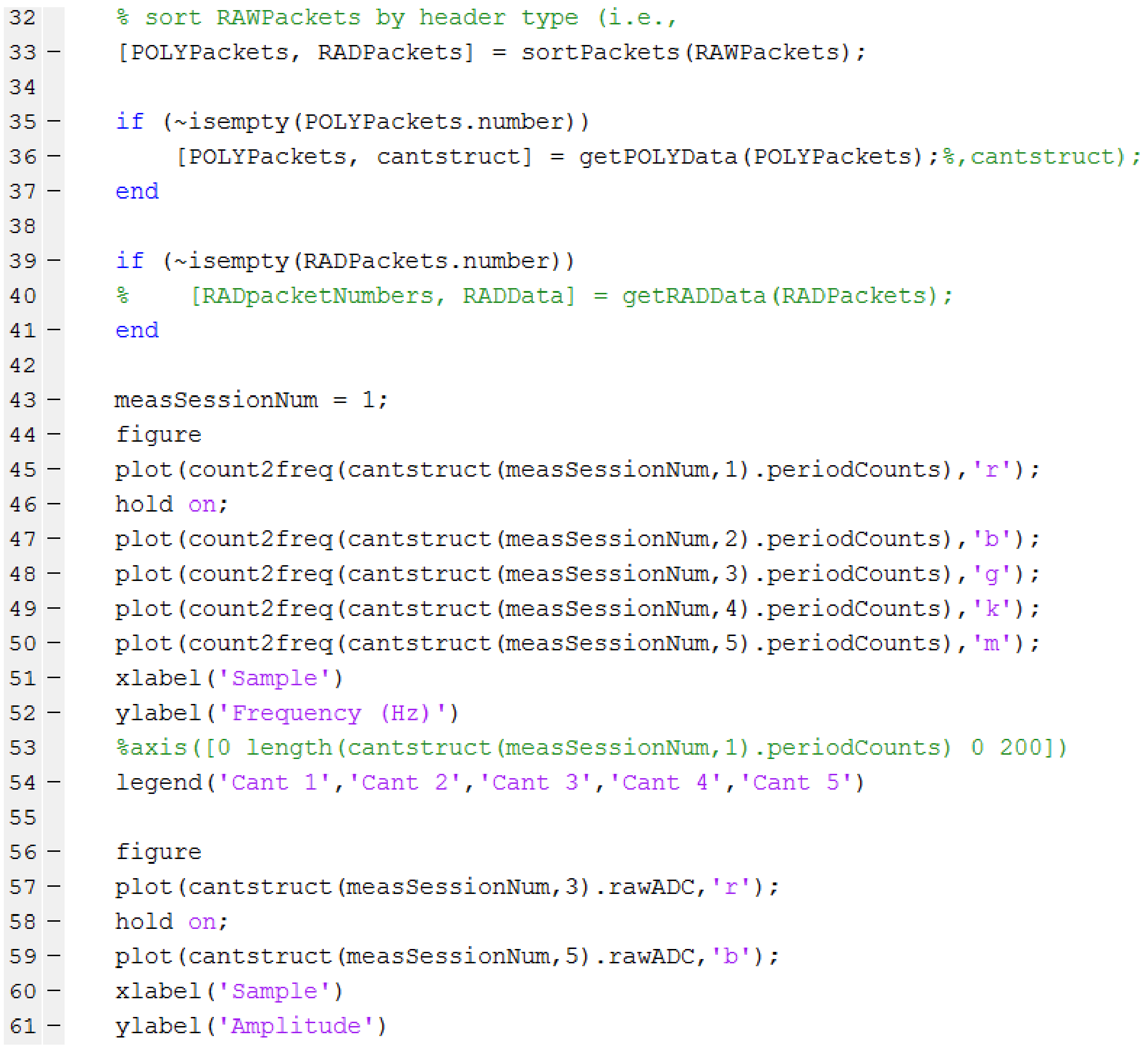

```
function [POLYPackets, RADPackets] = sortPackets(RAWPackets)

% find POLY packets
indices = find(hex2dec(RAWPackets.header) == hex2dec('A10C40'));
if (isempty(indices))
    POLYPackets.header = [];
    POLYPackets.number = [];
    POLYPackets.data = [];
else
    POLYPackets.header = char(RAWPackets.header(indices,:));
    POLYPackets.number = hex2dec(char(RAWPackets.number(indices,:)));
    POLYPackets.data = char(RAWPackets.data(indices,:));
end

% find RAD packets
indices = find(hex2dec(RAWPackets.header) == hex2dec('A10C80'));
if (isempty(indices))
    RADPackets.header = [];
    RADPackets.number = [];
    RADPackets.data = [];
else
    RADPackets.header = char(RAWPackets.header(indices,:));
    RADPackets.number = hex2dec(char(RAWPackets.number(indices,:)));
    RADPackets.data = char(RAWPackets.data(indices,:));
end
```

Figure 17: Packet Analyzer - sortPackets function

The code that is shown in Figure 18 plots the data that has been separated into all of the different types and adds proper labels. The program ends with these lines and the desired data from the CSV files are outputted to the screen. This is a good time to mention again that Aaron Ewing did not write the majority of this code, Dr. Griffin did. Please see stage 6 in the Required Tasks and Challenges section of this thesis for further explanation.



```
% sort RAWPackets by header type (i.e.,
[POLYPackets, RADPackets] = sortPackets(RAWPackets);

if (~isempty(POLYPackets.number))
    [POLYPackets, cantstruct] = getPOLYData(POLYPackets);%,cantstruct);
end

if (~isempty(RADPackets.number))
%    [RADpacketNumbers, RADData] = getRADData(RADPackets);
end

measSessionNum = 1;
figure
plot(count2freq(cantstruct(measSessionNum,1).periodCounts),'r');
hold on;
plot(count2freq(cantstruct(measSessionNum,2).periodCounts),'b');
plot(count2freq(cantstruct(measSessionNum,3).periodCounts),'g');
plot(count2freq(cantstruct(measSessionNum,4).periodCounts),'k');
plot(count2freq(cantstruct(measSessionNum,5).periodCounts),'m');
xlabel('Sample')
ylabel('Frequency (Hz)')
%axis([0 length(cantstruct(measSessionNum,1).periodCounts) 0 200])
legend('Cant 1','Cant 2','Cant 3','Cant 4','Cant 5')

figure
plot(cantstruct(measSessionNum,3).rawADC,'r');
hold on;
plot(cantstruct(measSessionNum,5).rawADC,'b');
xlabel('Sample')
ylabel('Amplitude')
```

Figure 18: Packet Analyzer – function output

Miscellaneous Code:

The only other significant thing not mentioned so far was learning what a linker file does and editing it. The MSP430FR6989 has split its memory addresses into normal DRAM and FRAM. FRAM stands for Ferroelectric Random Access Memory. Ferroelectric is different than a standard dielectric memory in the fact that it is non-volatile due to using a ferroelectric film as a capacitor instead of the dielectric film in DRAM. This means that it is more resistant to changes in the environment, specifically in regards to radiation because FRAM cannot be easily affected by magnetic or electric fields (Instruments, 2008). In this situation, it is more resistant to radiation damage that could corrupt memory in orbit. Alternatively, if the technology is older, made of less efficient but more resilient materials, it can be safe as well. However, smaller physical space, faster processing power, and other traits of new technology were needed, so FRAM was used.

```
bss        : {} > FRAM                /* Global & static vars              */
   .data      : {} > FRAM             /* Global & static vars              */
   .TI.noinit  : {} > FRAM            /* For #pragma noinit                */
   .stack     : {} > RAM (HIGH)       /* Software system stack             */
   .tinyram    : {} > TINYRAM         /* Tiny RAM                          */
```

In the above code, .bss (Basic Service Set), .data, and .TI.noinit are inside of FRAM (typically these are inside of RAM). .bb holds all uninstantiated variables (variables without a value), .data as where data is held, .TI.noinit holds not initialized variables. This protects all the information in the satellite from being corrupted from radiation strikes. An attempt at placing the stack in FRAM was pursued, was eventually abandoned due to runtime errors.

Future Work

There are a few things that would be a good idea for future work.

1) ***Correct IMU Error:*** The IMU stopped working at the last second during final CubeSat integration. The exact reason is unknown, but fixing the bug in the next one so collecting IMU data is possible would be wonderful.
2) ***Place stack in FRAM:*** To be able to put the stack inside of FRAM. It was not possible to get it to work before the deadline, but if the stack was in FRAM then there would be no weak link for possible radiation strikes against the microcontroller.

3) ***Auto-Error Correction:*** Have some sort of error correction software. Since the updating or viewing the software is not possible after launch. Using an algorithm that could at least make data collection resistant to bug errors could be useful is preserving valuable data from the satellite. This is currently being pursued by Aaron Ewing for part of his Engineering Senior Project.

Conclusion

This project was very character building for me. While I was not able to get into more elegant coding practices, it was fun to be able to work so close to the hardware (for example, most computer scientists do not worry about what kind of RAM they are using) and learning embedded systems. I wish I could have done more, but it was important for me to learn how to work efficiently with a team and how to cope with drastic scheduling changes, something that *will* happen again in my future career.

Citation:

MSP430FR6989. (2017, March). Retrieved from Texas Instruments: http://www.ti.com/product/MSP430FR6989/technicaldocuments

MSP430FR6989. (2017, March). Retrieved from Texas Instruments: http://www.ti.com/product/MSP430FR6989/datasheet

Instruments, A. O. (2008). FRAM FAQ. Retrieved from http://www.ti.com/pub/fram/fram_faq.html

Glossary

3D Printer: A device that is able to "print" a 3D object by creating multiple thin layers of some substance (often times the substance is some sort of plastic polymer). It is usually referred to as additive manufacturing in the industry, as it will slowly add material rather than remove material to make an object.

BAUD: symbols or pulses per a second is the proper definition. However, it can be loosely defined as bits per a second.

CAD: Computer-Aided Design. Allows for the construction of an object(s) in a virtual computer environment. Often used to create 3D printed objects.

Cantilever: Cantilevers used in this project are small flexible components in the shape of a diving board, where the flexible strip portion has the property of creating an increasingly larger voltage the further it is bent, known as a piezoelectric response. If there is no pressure on the flexible portion of the cantilever, there will be a zero voltage, if there is some pressure that causes it to bend, it will create some voltage level proportional to that particular cantilever.

CMOS: Complementary Metal-Oxide-Semiconductor sensors convert light into electric signals. Millions of rows of photodiodes create an image which is constructed using CMOS technology. CMOS sensors are often used in most modern digital cameras.

Embedded System: A computer system with a dedicated task that is embedded inside of a larger electrical or mechanical project.

EPS: Electrical Power System. The EPS controls which satellite subsystem is connected to the battery. It is a part of the OBC (On-Board-Computer, see OBC in Glossary), on the satellite.

Finite State Machine: A system that can be fully represented with a finite number of states, where the next state depends on the current state and the current input(s). For example, a system could have three states (State A, B, and C) and depend on a single input, and have a single output. Perhaps this system represents a light hooked up to a timer. State A could represent no input on the button, and will not light up the light. When pushing the button, however, the system will transition to State B. The system will stay in State B for as long as the button is held, and in State B the light is lit up. When releasing the button, it will transition to State C. In State C, the light will remain on, but will initialize a timer to count up to ten seconds. For those ten seconds, the light will remain on (due to it being in State C). If pushing the button happens within those ten seconds, the system will go back to State B, but if the button is not active within ten seconds it will return to state A. Refer to Finite State Diagram for a visual example.

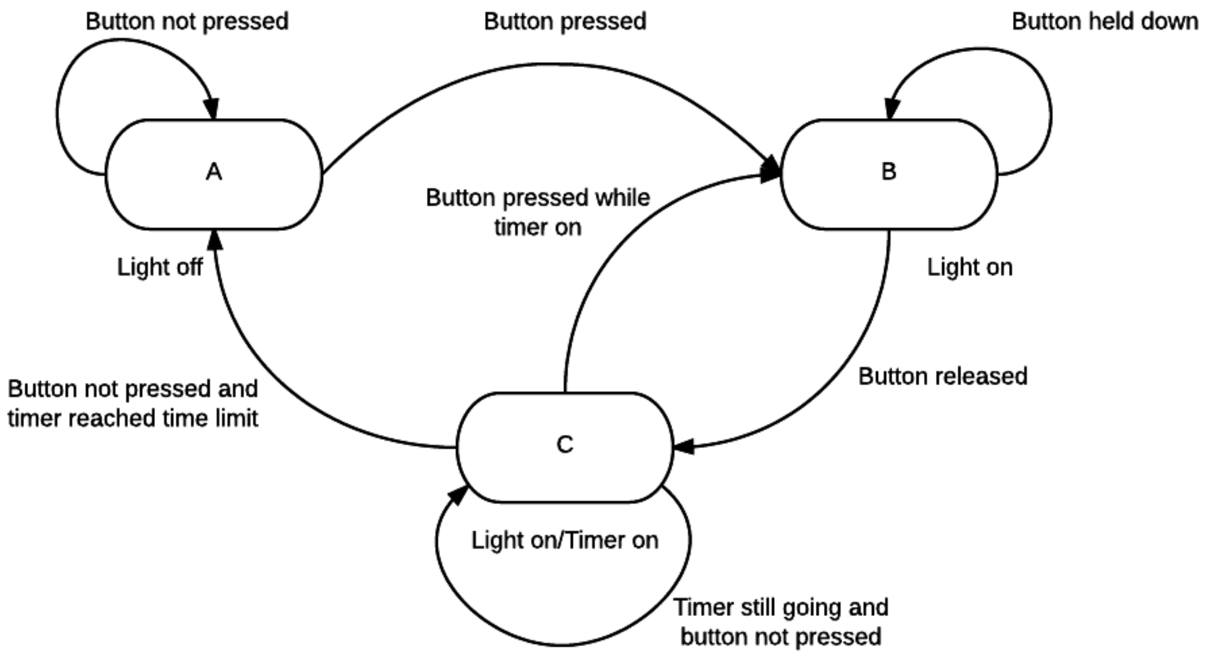

Button not pressed
Button pressed
Button held down
A
B
Button pressed while timer on
Light off
Light on
Button not pressed and timer reached time limit
Button released
C
Light on/Timer on
Timer still going and button not pressed

Figure 19: Example Finite State Diagram

Figure 18 shows multiple states (the bubbles) that have a name and a status, while an action can be made that will transition the current state to another one (this is represented by arrows pointing from one state to another). Refer to Finite State Machine example for an explanation on the diagram.

IDE: Integrated Design Environment is the software tool often used to write and test software. Such software usually contains a code editor, compiler or interpreter, and a debugger. Texas Instrument’s Code Composer was the main IDE for this project.

IMU: Inertial Measurement Unit. This particular IMU has 9 axes, 3 axes for acceleration, 3 axes for angular rotation, and 3 axes for the magnetometer. The magnetometer measures the magnetic field strength in each of the axes (X, Y, and Z).

IDE: Integrated Development Environment is an environment in which code is written and compiled for a specific hardware platform.

Linker File: A linker is a computer program that takes one or more object files generated when executing the program and combines them into a single file. It is an important step in turning code written in an IDE and turning it into something that can be executed.

Microcontroller: A microcontroller is a single integrated processor, with memory (both primary memory such as RAM for temporary use and secondary memory such as solid state for long term storage), and input-output (I/O) pins to interface with.

OBC: Stands for On-Board-Computer. It is the part of the satellite that was made by NearSpace Launch and contains the EPS (Electrical Power System, see EPS in Glossary), and the Radio.

SPI: Serial Peripheral Interface is a common communication protocol used within embedded systems. It is very good for transferring information quickly between two or more devices. A downside to SPI is the need to have several distinct communication lines.

Surface Transducer: Essentially a speaker that operates with physical contact. The surface transducer will use a solid medium to transmit vibrations through instead of air like most speakers.

Task Scheduler: A very simplistic operating system that orders the embedded systems tasks as needed, making sure that the execution of tasks will not interfere with the execution of other tasks.

UART: Universal Asynchronous Receiver/Transmitter communication is another communication protocol used on the satellite. It only requires two communication lines, but can only be used between two devices, requires calculation of BAUD rate, and is slower in communication.

Appendix A – Source Code

Packetizer:

Packetizer.h

```
/*
 * Packetizer.h
 *
 *  Created on: Sep 1, 2016
 *      Author: AaronEwing
 */

#ifndef PACKETIZER_H_
#define PACKETIZER_H_

void init_Buffers(void);

void Packetizer(uint16_t source_ID, uint8_t bytes_Read);

void write_To_Packetizer(uint8_t packet_Data, uint8_t SB_Select);

#endif /* PACKETIZER_H_ */
```

Packetizer.c

```
/*
 * Packetizer.c
 *
 *  Created on: Sep 1, 2016
 *      Author: AaronEwing
 */
#include <stdint.h>
#include <stdbool.h>
#include "Circular_Buffer.h"

uint16_t g_POLY_Exp_Count = 0;
uint16_t g_RAD_Exp_Count = 0;
uint16_t g_IMU_Exp_Count = 0;
uint16_t g_source_ID_Exp_Count = 0x0000;

uint8_t POLY_Bytes[39] = { 0x50, 0x50, 0x50, 0x0C, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
uint8_t IMU_Bytes[39] = { 0x50, 0x50, 0x50, 0x0C, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
uint8_t RAD_Bytes[39] = { 0x50, 0x50, 0x50, 0x0C, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
```

```
void Packetizer(uint16_t source_ID, uint8_t bytes_Read) {
        uint8_t ii = 6;
        switch (source_ID) {
        case 0:
                // IMU
        default:
                source_ID <<= 14;
        // shift ID 14 bits to the left & fill the 14 bits with 1's

                g_source_ID_Exp_Count = source_ID | g_IMU_Exp_Count;   // give source_ID_Exp_Count
appropriate value

                if (g_IMU_Exp_Count >= 0x3FFF) {                                      //
increment counter
                        g_IMU_Exp_Count = 0x0000;
                } else {
                        ++g_IMU_Exp_Count;
                }

                IMU_Bytes[4] = g_source_ID_Exp_Count >> 8;                  // put values into
array
                IMU_Bytes[5] = g_source_ID_Exp_Count;

                while (!is_Buffer_Empty() && (ii < 39)) {
        // reads from Circular Buffer into packetized buffer
                        IMU_Bytes[ii] = read_Buffer();
                        ii++;
                }

                while (ii < 39) {
                        IMU_Bytes[ii] = 0x00;
        // fill the rest of the packetzied buffer with zeroes
                        ii++;
                }
                ii = 0;
                break;

        case 1:
                // RAD
                source_ID <<= 14;
        // shift ID 14 bits to the left & fill the 14 bits with 1's

                g_source_ID_Exp_Count = source_ID | g_RAD_Exp_Count;  // give source_ID_Exp_Count
appropriate value

                if (g_RAD_Exp_Count >= 0x3FFF) {                                      //
increment counter
                        g_RAD_Exp_Count = 0x0000;
                } else {
                        ++g_RAD_Exp_Count;
                }

                RAD_Bytes[4] = g_source_ID_Exp_Count >> 8;                  // put values into
array
                RAD_Bytes[5] = g_source_ID_Exp_Count;
```

```
			while (!is_Buffer_Empty() && (ii < 39)) {
		// reads from Circular Buffer into packetized buffer
				RAD_Bytes[ii] = read_Buffer();
				ii++;
			}

			while (ii < 39) {
				RAD_Bytes[ii] = 0x00;
		// fill the rest of the packetzied buffer with zeroes
				ii++;
			}
			ii = 0;
			break;

		case 2:
			// POLY
			source_ID <<= 14;
		// shift ID 14 bits to the left & fill the 14 bits with 1's

			g_source_ID_Exp_Count = source_ID | g_POLY_Exp_Count;		// give
source_ID_Exp_Count appropriate value

			if (g_POLY_Exp_Count >= 0x3FFF) {					//
increment counter
				g_POLY_Exp_Count = 0x0000;
			} else {
				++g_POLY_Exp_Count;
			}

			POLY_Bytes[4] = g_source_ID_Exp_Count >> 8;			// put values into
array
			POLY_Bytes[5] = g_source_ID_Exp_Count;

			while (!is_Buffer_Empty() && (ii < 39)) {
		// reads from Circular Buffer into packetized buffer
				POLY_Bytes[ii] = read_Buffer();
				ii++;
			}

			while (ii < 39) {
				POLY_Bytes[ii] = 0x00;
		// fill the rest of the packetzied buffer with zeroes
				ii++;
			}
			break;
		}
}

uint8_t get_IMU_Data(uint8_t index) {
	return IMU_Bytes[index];
}

uint8_t get_RAD_Data(uint8_t index) {
	return RAD_Bytes[index];
}
```

```
uint8_t get_POLY_Data(uint8_t index) {
		return POLY_Bytes[index];
}

uint8_t get_Data(uint8_t index, uint8_t source_ID) {
		switch (source_ID) {
		case 0:
		default:
				return IMU_Bytes[index];
		case 1:
				return RAD_Bytes[index];
		case 2:
				return POLY_Bytes[index];
		}
}
```

SPI_Polling

SPI_Polling.h

```
/*
 * SPI_Pulling.h
 *
 *  Created on: Aug 16, 2016
 *      Author: aaronewing
 */
#include <msp430.h>
#include <stdint.h>

#ifndef SPI_POLLING_H_
#define SPI_POLLING_H_

void init_SPI (uint8_t pin_Setting);                                                              //
initalizes SPI clk rate and which pins are being used
void write_uint8_SPI (uint8_t tx_Data_8, uint8_t device_CS);                         // writes 8 bits with SPI
void write_uint16_SPI (uint16_t tx_Data_16, uint8_t device_CS);                    // writes 16 bits with SPI
uint16_t read_SPI (uint8_t transmit_Byte);
				// reads 8 bits with SPI

#endif /* SPI_POLLING_H_ */
```

SPI_Polling.c

```
/*
 * SPI_Pulling.c
 *
 *  Created on: Aug 16, 2016
 *      Author: aaronewing
 */

#include <msp430.h>
#include <stdint.h>
#include <stdbool.h>
#include "Comm.h"

uint8_t g_RXData;



//////////////////// SPI INIT //////////////////////////////////
void init_SPI (uint8_t pin_Setting) {
		switch (pin_Setting) {
		case 0:									// Hub
		default:
			// Configure Primary Function Pins
			P1SEL0 |= BIT4 | BIT6 | BIT7;			// P1.4 - CLK, P1.6 - SIMO, P1.7 - SOMI
			P1SEL1 &= ~(BIT4 & BIT6 & BIT7);

			// configure as GPIO used to enable SPI write from Hub

			P4SEL0 &= ~(BIT1 + BIT4 + BIT5 + BIT6 + BIT7);
			P4SEL1 &= ~(BIT1 + BIT4 + BIT5 + BIT6 + BIT7);
		// P4.1 slave data ready line
			P4DIR &= ~BIT1;
			P4DIR |= BIT4 + BIT5 + BIT6 + BIT7;
//			P4OUT |= BIT4;								// Change for
science board BIT4: 1; BIT5: 2; BIT6: 3; BIT7: 4;
			P4OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);				// Change for
science board BIT4: 1; BIT5: 2; BIT6: 3; BIT7: 4;
			break;

		case 1:									// MSP430FR5969
			// Configure Primary Function Pins
			P1SEL0 |= BIT6 | BIT7;		// P1.6 - SIMO, P1.7 - SOMI
			P2SEL0 |= BIT2;			// P2.2 - CLK

			// configure as GPIO used to enable SPI write to Hub
			P4SEL0 &= ~BIT1;
			P4SEL1 &= ~BIT1;						// P4.1 - SYNC/Slave Select
			P4DIR |= BIT1;
			P4OUT |= BIT1;
			break;

		case 2:									// Polymer degradation board - Pot
			// Configure Primary Function Pins
			P1SEL0 |= BIT6 | BIT7;		// P1.6 - SIMO, P1.7 - SOMI
			P2SEL0 |= BIT2;			// P2.2 - CLK

			// configure as GPIO used to enable SPI write to Hub
			P1SEL0 &= ~BIT1;
			P1SEL1 &= ~BIT1;						// P1.1 - SYNC/Slave Select
			P1DIR |= BIT1;
			P1OUT |= BIT1;
			break;
```

```
        case 3:                                                     // Polymer degradation board -
other
                // Configure Primary Function Pins
                P1SEL0 |= BIT6 | BIT7;          // P1.6 - SIMO, P1.7 - SOMI
                P2SEL0 |= BIT2;                 // P2.2 - CLK

                // configure as GPIO used to enable SPI write to Hub
                P1SEL0 &= ~BIT2;
                P1SEL1 &= ~BIT2;                                    // P1.2 - SYNC/Slave Select
                P1DIR |= BIT2;
                P1OUT |= BIT2;
                break;
        }

        // Configure USCI_B0 for SPI operation
        UCB0CTLW0 |= UCSWRST;                       // **Put state machine in reset**

        UCB0CTLW0 |= UCMST | UCSYNC | UCMSB | UCCKPL;               // 3-pin, 8-bit SPI master
                                            // Clock polarity high, MSB
        UCB0CTLW0 |= UCSSEL__SMCLK;                                 // SMCLK

        UCB0BRW = 0x0008;                                                   // Divides SMCLK
module by 8 (8MHz/8 = 1MHz)

        UCA0MCTLW = 0;                              // No modulation
        UCB0CTLW0 &= ~UCSWRST;                                  // **Initialize USCI state machine**


}

//////////////////// SPI WRITE 8 BIT ///////////////////////////////////
void write_uint8_SPI (uint8_t tx_Data_8, uint8_t device_CS) {
        while (!(UCB0IFG & UCTXIFG)){};                                                 // If able
to TX

        switch (device_CS) {                                    // Hub
                case 0:
                default:
//                      P4OUT &= ~BIT1;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                        // While TXing
                        UCB0TXBUF = tx_Data_8;
        // 8 bits transmitted
                        while (UCB0STATW & UCBUSY) {};                                  // While
not busy
//                      P4OUT |= BIT1;
                        break;

                case 1:
                        P2OUT &= ~BIT8;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                        // While TXing
                        UCB0TXBUF = tx_Data_8;
        // 8 bits transmitted
                        while (UCB0STATW & UCBUSY) {};
                        P2OUT |= BIT6;
                        break;

                case 2:
                        P3OUT &= ~BIT6;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                    // While TXing
                        UCB0TXBUF = tx_Data_8;
        // 8 bits transmitted
                        while (UCB0STATW & UCBUSY) {};
                        P3OUT |= BIT6;
                        break;

                case 3:
                        P4OUT &= ~BIT6;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                    // While TXing
                        UCB0TXBUF = tx_Data_8;
        // 8 bits transmitted
                        while (UCB0STATW & UCBUSY) {};
                        P4OUT |= BIT6;
                        break;
        }
}

//////////////////// SPI WRITE 16 BIT ///////////////////////////////////
void write_uint16_SPI (uint16_t tx_Data_16, uint8_t device_CS) {
        while (!(UCB0IFG & UCTXIFG)){};                                         // If able
to TX

        switch (device_CS) {
                case 0:                                                     // Hub
                default:
                        P4OUT &= ~BIT1;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                    // While TXing
                        UCB0TXBUF = (tx_Data_16 >> 8);                              // First 8
bits transmitted (Control bits and data)
                        while (!(UCB0IFG & UCTXIFG)) {};
                        UCB0TXBUF = tx_Data_16;
        // Last 8 bits transmitted (overflow expected and is fine)
                        while (UCB0STATW & UCBUSY) {};
                        P4OUT |= BIT1;
                        break;

                case 1:
                        P2OUT &= ~BIT8;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                    // While TXing
                        UCB0TXBUF = (tx_Data_16 >> 8);                              // First 8
bits transmitted (Control bits and data)
                        while (!(UCB0IFG & UCTXIFG)) {};
                        UCB0TXBUF = tx_Data_16;
        // Last 8 bits transmitted (overflow expected and is fine)
                        while (UCB0STATW & UCBUSY) {};
                        P2OUT |= BIT6;
                        break;

                case 2:
                        P3OUT &= ~BIT6;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                        // While TXing
                        UCB0TXBUF = (tx_Data_16 >> 8);                                  // First 8
bits transmitted (Control bits and data)
                        while (!(UCB0IFG & UCTXIFG)) {};
                        UCB0TXBUF = tx_Data_16;
        // Last 8 bits transmitted (overflow expected and is fine)
                        while (UCB0STATW & UCBUSY) {};
                        P3OUT |= BIT6;
                        break;

                case 3:
                        P4OUT &= ~BIT6;
        // Pulls SYNC low
                        while (!(UCB0IFG & UCTXIFG)) {};                        // While TXing
                        UCB0TXBUF = (tx_Data_16 >> 8);                                  // First 8
bits transmitted (Control bits and data)
                        while (!(UCB0IFG & UCTXIFG)) {};
                        UCB0TXBUF = tx_Data_16;
        // Last 8 bits transmitted (overflow expected and is fine)
                        while (UCB0STATW & UCBUSY) {};
                        P4OUT |= BIT6;
                        break;
        }
}

//////////////////// SPI READ POLLING ////////////////////////////////
uint16_t read_SPI (uint8_t transmit_Byte) {

        while (!(UCB0IFG & UCTXIFG) && !timeout) {};                    // While TXing
        UCB0TXBUF = transmit_Byte;                                              //
Transmits read_byte
        while ((UCB0STATW & UCBUSY) && !timeout) {};                            // While
not busy
        while (!(UCB0IFG & UCRXIFG) && !timeout) {};                        // While RX flag
is high
        g_RXData = UCB0RXBUF;                                                   // First 8
bits transmitted (Control bits and data)
        __delay_cycles(100);
        return g_RXData;

}
```

UART_Polling:

UART_Polling.h

```
/*
 * UART_Polling.h
 *
 *  Created on: Aug 18, 2016
 *      Author: aaronewing
 */
#ifndef UART_POLLING_H_
#define UART_POLLING_H_

void init_UART (bool baud_Rate, bool pin_Setting);             // initalizes UART clk rate and which pins are being
used
// baud_Rate = 0 - 9600, 1 - 38400 (default), pin_Setting (0 - 2.0 TX, 2.1 RX, 2.2 BUSY (default), 1 - testing)
void write_UART (uint8_t TX_Data);                                                    // writes 8 bits with SPI
uint8_t read_UART (void);
uint8_t TX_Data;

#endif /* UART_POLLING_H_ */
```

UART_Polling.c

```
/*
 * SPI_Pulling.c
 *
 *  Created on: Aug 16, 2016
 *      Author: aaronewing
 */

#include <msp430.h>
#include <stdint.h>
#include <stdbool.h>
#include "UART_Polling.h"
#include "Comm.h"

#define UART_RADIO_BUSY 0x04           // P1.4

uint8_t RX_Data;                                              // basically the radio ACK

///////////////////// UART INIT ///////////////////////////////////
void init_UART (bool baud_Rate, bool pin_Setting) {
            switch (pin_Setting) {
            case 0:
            default:
                        // Configure Secondary Function Pins
                        P2SEL0 |= BIT0 | BIT1;                                        // P2.0 - TX, P2.1 - RX
                        P2SEL1 &= ~(BIT0 | BIT1);

                        P2SEL0 &= ~UART_RADIO_BUSY;
                        P2SEL1 &= ~UART_RADIO_BUSY;                                          // P2.2 - Radio Busy line
                        P2DIR &= ~UART_RADIO_BUSY;
                        P2IN &= ~UART_RADIO_BUSY;
                        break;

            case 1:
                        // Configure Secondary Function Pins
                        P3SEL0 |= BIT4 | BIT5;                 // P3.4 - TX, P3.5 - RX
                        P4SEL1 &= ~(BIT4 | BIT5);

                        P1SEL0 &= ~UART_RADIO_BUSY;
                        P1SEL1 &= ~UART_RADIO_BUSY;                                          // P1.4 - Radio Busy line
```

```
			P1DIR &= ~UART_RADIO_BUSY;
			P1IN &= ~UART_RADIO_BUSY;
			break;
	}

	 // XT1 Setup
	/*
	CSCTL0_H = CSKEY >> 8;				// Unlock CS registers
	CSCTL1 = DCOFSEL_0;						// Set DCO to 1MHz
	CSCTL2 = SELA__LFXTCLK | SELS__DCOCLK | SELM__DCOCLK;
	CSCTL0_H = 0;			// Lock CS registers
	*/

	// Configure USCI_A0 for SPI operation
//	UCA0CTL1 |= UCSWRST;				// **Put state machine in reset**

	//UCB0CTLW0 |= UCSSEL__SMCLK;				// SMCLK
//	UCB0BRW = 0x0000;						//
Divides SMCLK module by 8 (8MHz/8 = 1MHz)

	switch (baud_Rate) {
	case 0:
				// Configure Timer for 9600 Baud
		UCA0CTL1 = UCSSEL__ACLK;		// Set ACLK = 32768 as UCBRCLK
		UCA0BR0 = 3;			// 9600 baud
		UCA0MCTLW |= 0x5300;		// 32768/9600 - INT(32768/9600)=0.41
					// UCBRSx value = 0x53 (See UG)
		UCA0BR1 = 0;
		break;

	case 1:
	default:
				// Configure Timer for 38400 Baud
		UCA0CTL1 = UCSSEL__SMCLK;			// Set SMCLK = 1000000 as UCBRCLK
		UCA0BR0 = 0x1A;							// 38400
baud
		UCA0MCTLW |= 0x0100;		// 1000000/38400 - INT(1000000/38400)=0.04
					// UCBRSx value = 0x01 (See UG)
										// N = 0.0529,
effectively 38,383.4 Baud
		UCA0BR1 = 0;
	}

	UCA0CTL1 &= ~UCSWRST;			// **Initialize USCI state machine**
}

//////////////////// UART WRITE POLLING ///////////////////////////////////
void write_UART (uint8_t TX_Data) {
	while (!(UCA0IFG & UCTXIFG)){};						// If able to TX
	while (P1IN == UART_RADIO_BUSY) {};				// If Radio is not busy
	UCA0TXBUF = TX_Data;								// 8 bits
transmitted
}

//////////////////// UART READ POLLING ///////////////////////////////
uint8_t read_UART (void) {
```

```
            while (!(UCA0IFG & UCRXIFG) && !timeout) {};                          // While RX flag is high
            RX_Data = UCA0RXBUF;                                                            // Recieve Radio
ACK
            return RX_Data;
}
```

Initialize:

Initialize.h

```
/*
 * Initialize.h
 *
 *  Created on: Aug 11, 2016
 *      Author: aaronewing
 */

// contains all functions for Initializing MSP430

#ifndef INITIALIZE_H_
#define INITIALIZE_H_

void initialize_Ports(void);
void initialize_Clocks(void);

#endif /* INITIALIZE_H_ */
```

Initialize.c

```
/*
 * Initialize.c
 *
 *  Created on: Aug 0xFFFF0xFFFF, 200xFFFF6
 *      Author: aaronewing
 */
// contains all functions for Initializing MSP430

#include <msp430fr6989.h>
#include "Initialize.h"

void initialize_Clocks(void) {                                                        // Sets all clocks
to standard position

    PJSEL0 |= BIT4 | BIT6;      // LFXT (BIT4) HFXT (BIT6)
    PJSEL1 &= ~(BIT4 + BIT6);       // LFXT (BIT4) HFXT (BIT6)

    CSCTL0_H = CSKEY >> 8;                      // Unlock CS registers
    CSCTL2 = SELA__LFXTCLK | SELS__HFXTCLK | SELM__HFXTCLK;    // ACLK-LFXT, SMCLK-HFXT,
MCLK-HFXT
            CSCTL3 = DIVA_0 | DIVS__8 | DIVM__8;      // set all dividers

    CSCTL4 = HFFREQ_1;
    CSCTL4 &= ~(HFXTOFF + LFXTOFF);

            do
```

```
        {
                CSCTL5 &= ~(LFXTOFFG + HFXTOFFG);           // Clear LFXT fault flag
                SFRIFG1 &= ~OFIFG;
        }
        while (SFRIFG1&OFIFG);               // Test oscillator fault flag
                 CSCTL0_H = 0;                       // Lock CS registers

   CSCTL0_H = 0;
}

void initialize_Ports(void){                         // sets all pins on all ports as an output (except Port 10)
        PM5CTL0 &= ~LOCKLPM5;

        P1DIR |= 0xFFFF;
        P2DIR |= 0xFFFF;
        P3DIR |= 0xFFFF;
//      P4DIR |= 0xFFFF;
        P5DIR |= 0xFFFF;
        P6DIR |= 0xFFFF;
        P7DIR |= 0xFFFF;
        P8DIR |= 0xFFFF;
        P9DIR |= 0xFFFF;
        P10DIR |= 0xFFFF;                                     // Pins 0, 1, and 2 are the only ones to exist
on Port 10

        P1OUT = 0x000;                                        // sets all pins on all ports to a low output
(redundant)
        P2OUT = 0x000;
        P3OUT = 0x000;
//      P4OUT = 0x000;
        P5OUT = 0x000;
        P6OUT = 0x000;
        P7OUT = 0x000;
        P8OUT = 0x000;
        P9OUT = 0x000;
        P10OUT = 0x000;
}
```

Matlab Packet Parser:

```
clear all;
close all;

% create a structs for the POLY and RAD data
field1 = 'header';  value1 = zeros(1,6);
field2 = 'number';  value2 = zeros(1,2);
field3 = 'data';  value3 = zeros(1,64);
field4 = 'avgPeriodCount';  value4 = zeros(1,10);
field5 = 'periodCounts';  value5 = zeros(1,79);
field6 = 'rawADC';  value6 = zeros(1,208);
field7 = 'healthpacket'; value7 = zeros(1,11);
RAWPackets = struct(field1,value1,field2,value2,field3,value3);
POLYPackets = struct(field1,value1,field2,value2,field3,value3);
cantstruct = struct(field4,value4,field5,value5,field6,value6,field7,value7);
```

```
RADPackets = struct(field1,value1,field2,value2,field3,value3);

% read RAW packet data from the .csv file downloaded from NSL
%fileID = fopen('NSL_0-788285_FnCode_A1_data.csv');
%fileID = fopen('Poly_Board_Packets_On_Ground.csv');
fileID = fopen('MakerSatData.csv');
%PacketDataCellArray = textscan(fileID,'%s %s %s %s %s %s %s','Delimiter',',');
PacketDataCellArray = textscan(fileID,'%q','Delimiter',',');
fclose(fileID);

% grab packets from file and parse them into headder bit, packet number,
% and packet data.
% The first 3 elements of each packet string and last 3 are thrown away (""" and """)
jj = 1;

% for ii = 1:2:length(PacketDataCellArray{1,4})
%     RAWPackets.header(jj,:) = PacketDataCellArray{1,4}{ii,1}(1,4:9);
%     RAWPackets.number(jj,:) = PacketDataCellArray{1,4}{ii,1}(1,10:11);
%     RAWPackets.data(jj,:) = PacketDataCellArray{1,4}{ii,1}(1,12:end-3);
%     jj = jj + 1;
% end


for ii = 2:length(PacketDataCellArray{1,1})
    RAWPackets.header(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,1:6);
    RAWPackets.number(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,7:8);
    RAWPackets.data(jj,:) = PacketDataCellArray{1,1}{ii,1}(1,9:end);
    jj = jj + 1;
end

% sort RAWPackets by header type (i.e.,
[POLYPackets, RADPackets] = sortPackets(RAWPackets);

if (~isempty(POLYPackets.number))
    [POLYPackets, cantstruct] = getPOLYData(POLYPackets);%,cantstruct);
end

if (~isempty(RADPackets.number))
%    [RADpacketNumbers, RADData] = getRADData(RADPackets);
end

measSessionNum = 1;
figure
plot(count2freq(cantstruct(measSessionNum,1).periodCounts),'r');
hold on;
plot(count2freq(cantstruct(measSessionNum,2).periodCounts),'b');
plot(count2freq(cantstruct(measSessionNum,3).periodCounts),'g');
plot(count2freq(cantstruct(measSessionNum,4).periodCounts),'k');
plot(count2freq(cantstruct(measSessionNum,5).periodCounts),'m');
xlabel('Sample')
ylabel('Frequency (Hz)')
%axis([0 length(cantstruct(measSessionNum,1).periodCounts) 0 200])
legend('Cant 1','Cant 2','Cant 3','Cant 4','Cant 5')

figure
plot(cantstruct(measSessionNum,3).rawADC,'r');
hold on;
plot(cantstruct(measSessionNum,5).rawADC,'b');
xlabel('Sample')
ylabel('Amplitude')
%axis([0 length(cantstruct(1).periodCounts) 0 100000])
legend('Cant 3','Cant 5')

figure
plot(cantstruct(measSessionNum,6).healthpacket,'r');
xlabel('total averaged values')
ylabel('Amplitude')
%axis([0 length(cantstruct(1).periodCounts) 0 100000])
legend('healthpacket')
```

Linker Command File:

```
/* ============================================================================ */
/* Copyright (c) 2016, Texas Instruments Incorporated                           */
/*  All rights reserved.                                                        */
/*                                                                              */
/*  Redistribution and use in source and binary forms, with or without          */
/*  modification, are permitted provided that the following conditions          */
/*  are met:                                                                    */
/*                                                                              */
/*  *  Redistributions of source code must retain the above copyright           */
/*     notice, this list of conditions and the following disclaimer.            */
/*                                                                              */
/*  *  Redistributions in binary form must reproduce the above copyright        */
/*     notice, this list of conditions and the following disclaimer in the      */
/*     documentation and/or other materials provided with the distribution.     */
/*                                                                              */
/*  *  Neither the name of Texas Instruments Incorporated nor the names of      */
/*     its contributors may be used to endorse or promote products derived      */
/*     from this software without specific prior written permission.            */
/*                                                                              */
/*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" */
/*  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,       */
/*  THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR      */
/*  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR            */
/*  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,       */
/*  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,         */
/*  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; */
/*  OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,    */
/*  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR     */
/*  OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,              */
/*  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.                          */
/* ============================================================================ */

/******************************************************************************/
/* lnk_msp430fr6989.cmd - LINKER COMMAND FILE FOR LINKING MSP430FR6989 PROGRAMS     */
/*                                                                            */
/*   Usage:  lnk430 <obj files...>    -o <out file> -m <map file> lnk.cmd     */
/*           cl430  <src files...> -z -o <out file> -m <map file> lnk.cmd     */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* These linker options are for command line linking only.  For IDE linking,  */
/* you should set your linker options in Project Properties                   */
/* -c                                               LINK USING C CONVENTIONS  */
/* -stack  0x0100                                   SOFTWARE STACK SIZE       */
/* -heap   0x0100                                   HEAP AREA SIZE            */
/*                                                                            */
/*----------------------------------------------------------------------------*/
/* Version: 1.192                                                             */
/*----------------------------------------------------------------------------*/

/****************************************************************************/
/* Specify the system memory map                                            */
/****************************************************************************/

MEMORY
{
    TINYRAM                 : origin = 0x0006, length = 0x001A
    PERIPHERALS_8BIT        : origin = 0x0020, length = 0x00E0
    PERIPHERALS_16BIT       : origin = 0x0100, length = 0x0100
    RAM                     : origin = 0x1C00, length = 0x0800
    INFOA                   : origin = 0x1980, length = 0x0080
    INFOB                   : origin = 0x1900, length = 0x0080
    INFOC                   : origin = 0x1880, length = 0x0080
    INFOD                   : origin = 0x1800, length = 0x0080
    FRAM                    : origin = 0x4400, length = 0xBB80
    FRAM2                   : origin = 0x10000,length = 0x14000
    JTAGSIGNATURE           : origin = 0xFF80, length = 0x0004, fill = 0xFFFF
    BSLSIGNATURE            : origin = 0xFF84, length = 0x0004, fill = 0xFFFF
    IPESIGNATURE            : origin = 0xFF88, length = 0x0008, fill = 0xFFFF
    INT00                   : origin = 0xFF90, length = 0x0002
    INT01                   : origin = 0xFF92, length = 0x0002
    INT02                   : origin = 0xFF94, length = 0x0002
    INT03                   : origin = 0xFF96, length = 0x0002
    INT04                   : origin = 0xFF98, length = 0x0002
    INT05                   : origin = 0xFF9A, length = 0x0002
    INT06                   : origin = 0xFF9C, length = 0x0002
    INT07                   : origin = 0xFF9E, length = 0x0002
    INT08                   : origin = 0xFFA0, length = 0x0002
    INT09                   : origin = 0xFFA2, length = 0x0002
    INT10                   : origin = 0xFFA4, length = 0x0002
    INT11                   : origin = 0xFFA6, length = 0x0002
    INT12                   : origin = 0xFFA8, length = 0x0002
    INT13                  : origin = 0xFFAA, length = 0x0002
    INT14                  : origin = 0xFFAC, length = 0x0002
    INT15                  : origin = 0xFFAE, length = 0x0002
    INT16                  : origin = 0xFFB0, length = 0x0002
    INT17                  : origin = 0xFFB2, length = 0x0002
    INT18                  : origin = 0xFFB4, length = 0x0002
    INT19                  : origin = 0xFFB6, length = 0x0002
    INT20                  : origin = 0xFFB8, length = 0x0002
    INT21                  : origin = 0xFFBA, length = 0x0002
    INT22                  : origin = 0xFFBC, length = 0x0002
    INT23                  : origin = 0xFFBE, length = 0x0002
    INT24                  : origin = 0xFFC0, length = 0x0002
    INT25                  : origin = 0xFFC2, length = 0x0002
    INT26                  : origin = 0xFFC4, length = 0x0002
    INT27                  : origin = 0xFFC6, length = 0x0002
    INT28                  : origin = 0xFFC8, length = 0x0002
    INT29                  : origin = 0xFFCA, length = 0x0002
    INT30                  : origin = 0xFFCC, length = 0x0002
    INT31                  : origin = 0xFFCE, length = 0x0002
    INT32                  : origin = 0xFFD0, length = 0x0002
    INT33                  : origin = 0xFFD2, length = 0x0002
    INT34                  : origin = 0xFFD4, length = 0x0002
    INT35                  : origin = 0xFFD6, length = 0x0002
    INT36                  : origin = 0xFFD8, length = 0x0002
    INT37                  : origin = 0xFFDA, length = 0x0002
    INT38                  : origin = 0xFFDC, length = 0x0002
    INT39                  : origin = 0xFFDE, length = 0x0002
    INT40                  : origin = 0xFFE0, length = 0x0002
    INT41                  : origin = 0xFFE2, length = 0x0002
    INT42                  : origin = 0xFFE4, length = 0x0002
    INT43                  : origin = 0xFFE6, length = 0x0002
    INT44                  : origin = 0xFFE8, length = 0x0002
    INT45                  : origin = 0xFFEA, length = 0x0002
    INT46                  : origin = 0xFFEC, length = 0x0002
    INT47                  : origin = 0xFFEE, length = 0x0002
    INT48                  : origin = 0xFFF0, length = 0x0002
    INT49                  : origin = 0xFFF2, length = 0x0002
    INT50                  : origin = 0xFFF4, length = 0x0002
    INT51                  : origin = 0xFFF6, length = 0x0002
    INT52                  : origin = 0xFFF8, length = 0x0002
    INT53                  : origin = 0xFFFA, length = 0x0002
    INT54                  : origin = 0xFFFC, length = 0x0002
    RESET                   : origin = 0xFFFE, length = 0x0002
}

/****************************************************************************/
/* Specify the sections allocation into memory                              */
/****************************************************************************/

SECTIONS
{
    GROUP(RW_IPE)
    {
        GROUP(READ_WRITE_MEMORY)
        {
            .TI.persistent : {}              /* For #pragma persistent            */
        .cio          : {}              /* C I/O Buffer                     */
        .sysmem       : {}              /* Dynamic memory allocation area    */
      } PALIGN(0x0400), RUN_START(fram_rw_start)

      GROUP(IPENCAPSULATED_MEMORY)
      {
        .ipestruct    : {}              /* IPE Data structure               */
        .ipe          : {}              /* IPE                              */
        .ipe_const    : {}              /* IPE Protected constants          */
        .ipe:_isr     : {}              /* IPE ISRs                         */
        .ipe_vars     : type = NOINIT{} /* IPE variables                    */
      } PALIGN(0x0400), RUN_START(fram_ipe_start) RUN_END(fram_ipe_end) RUN_END(fram_rx_start)
    } > 0x4400

    .cinit          : {}  > FRAM        /* Initialization tables            */
    .pinit          : {}  > FRAM        /* C++ Constructor tables           */
    .binit          : {}  > FRAM        /* Boot-time Initialization tables   */
    .init_array     : {}  > FRAM        /* C++ Constructor tables           */
    .mspabi.exidx   : {}  > FRAM        /* C++ Constructor tables           */
    .mspabi.extab   : {}  > FRAM        /* C++ Constructor tables           */
#ifndef __LARGE_DATA_MODEL__
    .const          : {} > FRAM         /* Constant data                    */
#else
    .const          : {} >> FRAM | FRAM2  /* Constant data                  */
#endif

    .text:_isr      : {}  > FRAM        /* Code ISRs                        */
#ifndef __LARGE_DATA_MODEL__
    .text           : {} > FRAM         /* Code                             */
#else
    .text           : {} >> FRAM2 | FRAM  /* Code                           */
#endif
#ifdef __TI_COMPILER_VERSION__
  #if __TI_COMPILER_VERSION__ >= 15009000
    #ifndef __LARGE_DATA_MODEL__
    .TI.ramfunc : {} load=FRAM, run=RAM, table(BINIT)
    #else
    .TI.ramfunc : {} load=FRAM | FRAM2, run=RAM, table(BINIT)
    #endif
  #endif
#endif

    .jtagsignature : {} > JTAGSIGNATURE     /* JTAG Signature                 */
    .bslsignature  : {} > BSLSIGNATURE      /* BSL Signature                  */

    GROUP(SIGNATURE_SHAREDMEMORY)
    {
      .ipesignature  : {}               /* IPE Signature                    */
      .jtagpassword  : {}               /* JTAG Password                    */
    } > IPESIGNATURE

    .bss        : {} > FRAM             /* Global & static vars             */
    .data       : {} > FRAM             /* Global & static vars             */
    .TI.noinit  : {} > FRAM             /* For #pragma noinit               */
    .stack      : {} > RAM (HIGH)       /* Software system stack            */
    .tinyram    : {} > TINYRAM          /* Tiny RAM                         */
```

```
.infoA    : {} > INFOA            /* MSP430 INFO FRAM  Memory segments */
.infoB    : {} > INFOB
.infoC    : {} > INFOC
.infoD    : {} > INFOD

/* MSP430 Interrupt vectors          */
.int00      : {}              > INT00
.int01      : {}              > INT01
.int02      : {}              > INT02
.int03      : {}              > INT03
.int04      : {}              > INT04
.int05      : {}              > INT05
.int06      : {}              > INT06
.int07      : {}              > INT07
.int08      : {}              > INT08
.int09      : {}              > INT09
.int10      : {}              > INT10
.int11      : {}              > INT11
.int12      : {}              > INT12
.int13      : {}              > INT13
.int14      : {}              > INT14
.int15      : {}              > INT15
.int16      : {}              > INT16
.int17      : {}              > INT17
.int18      : {}              > INT18
.int19      : {}              > INT19
.int20      : {}              > INT20
.int21      : {}              > INT21
.int22      : {}              > INT22
.int23      : {}              > INT23
.int24      : {}              > INT24
.int25      : {}              > INT25
.int26      : {}              > INT26
AES256       : { * ( .int27 ) } > INT27 type = VECT_INIT
RTC          : { * ( .int28 ) } > INT28 type = VECT_INIT
LCD_C        : { * ( .int29 ) } > INT29 type = VECT_INIT
PORT4        : { * ( .int30 ) } > INT30 type = VECT_INIT
PORT3        : { * ( .int31 ) } > INT31 type = VECT_INIT
TIMER3_A1    : { * ( .int32 ) } > INT32 type = VECT_INIT
TIMER3_A0    : { * ( .int33 ) } > INT33 type = VECT_INIT
PORT2        : { * ( .int34 ) } > INT34 type = VECT_INIT
TIMER2_A1    : { * ( .int35 ) } > INT35 type = VECT_INIT
TIMER2_A0    : { * ( .int36 ) } > INT36 type = VECT_INIT
PORT1        : { * ( .int37 ) } > INT37 type = VECT_INIT
TIMER1_A1    : { * ( .int38 ) } > INT38 type = VECT_INIT
TIMER1_A0    : { * ( .int39 ) } > INT39 type = VECT_INIT
DMA          : { * ( .int40 ) } > INT40 type = VECT_INIT
USCI_B1      : { * ( .int41 ) } > INT41 type = VECT_INIT
USCI_A1      : { * ( .int42 ) } > INT42 type = VECT_INIT
TIMER0_A1    : { * ( .int43 ) } > INT43 type = VECT_INIT
TIMER0_A0    : { * ( .int44 ) } > INT44 type = VECT_INIT
ADC12        : { * ( .int45 ) } > INT45 type = VECT_INIT
USCI_B0      : { * ( .int46 ) } > INT46 type = VECT_INIT
USCI_A0      : { * ( .int47 ) } > INT47 type = VECT_INIT
ESCAN_IF     : { * ( .int48 ) } > INT48 type = VECT_INIT
```

```
   WDT          : { * ( .int49 ) } > INT49 type = VECT_INIT
   TIMER0_B1    : { * ( .int50 ) } > INT50 type = VECT_INIT
   TIMER0_B0    : { * ( .int51 ) } > INT51 type = VECT_INIT
   COMP_E       : { * ( .int52 ) } > INT52 type = VECT_INIT
   UNMI         : { * ( .int53 ) } > INT53 type = VECT_INIT
   SYSNMI       : { * ( .int54 ) } > INT54 type = VECT_INIT
   .reset       : {}               > RESET  /* MSP430 Reset vector         */
}

/****************************************************************************/
/* MPU/IPE Specific memory segment definitons                               */
/****************************************************************************/

#ifdef _IPE_ENABLE
  #define IPE_MPUIPLOCK 0x0080
  #define IPE_MPUIPENA 0x0040
  #define IPE_MPUIPPUC 0x0020

  // Evaluate settings for the control setting of IP Encapsulation
  #if defined(_IPE_ASSERTPUC1)
      #if defined(_IPE_LOCK ) && (_IPE_ASSERTPUC1 == 0x08))
       fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPPUC |IPE_MPUIPLOCK);
      #elif defined(_IPE_LOCK )
       fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPLOCK);
    #elif (_IPE_ASSERTPUC1 == 0x08)
       fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPPUC);
    #else
       fram_ipe_enable_value = (IPE_MPUIPENA);
    #endif
  #else
    #if defined(_IPE_LOCK )
       fram_ipe_enable_value = (IPE_MPUIPENA | IPE_MPUIPLOCK);
    #else
       fram_ipe_enable_value = (IPE_MPUIPENA);
    #endif
  #endif

  // Segment definitions
  #ifdef _IPE_MANUAL                // For custom sizes selected in the GUI
    fram_ipe_border1 = (_IPE_SEGB1>>4);
    fram_ipe_border2 = (_IPE_SEGB2>>4);
  #else                          // Automated sizes generated by the Linker
    fram_ipe_border2 = fram_ipe_end >> 4;
    fram_ipe_border1 = fram_ipe_start >> 4;
  #endif

  fram_ipe_settings_struct_address = Ipe_settingsStruct >> 4;
  fram_ipe_checksum = ~((fram_ipe_enable_value & fram_ipe_border2 & fram_ipe_border1) |
(fram_ipe_enable_value & ~fram_ipe_border2 & ~fram_ipe_border1) | (~fram_ipe_enable_value &
fram_ipe_border2 & ~fram_ipe_border1) | (~fram_ipe_enable_value & ~fram_ipe_border2 & fram_ipe_border1));
#endif

#ifdef _MPU_ENABLE
  #define MPUPW (0xA500)    /* MPU Access Password */
  #define MPUENA (0x0001)   /* MPU Enable */
  #define MPULOCK (0x0002)  /* MPU Lock */
  #define MPUSEGIE (0x0010) /* MPU Enable NMI on Segment violation */

  __mpu_enable = 1;
  // Segment definitions
  #ifdef _MPU_MANUAL // For custom sizes selected in the GUI
    mpu_segment_border1 = _MPU_SEGB1 >> 4;
    mpu_segment_border2 = _MPU_SEGB2 >> 4;
    mpu_sam_value = (_MPU_SAM0 << 12) | (_MPU_SAM3 << 8) | (_MPU_SAM2 << 4) | _MPU_SAM1;
  #else // Automated sizes generated by Linker
    #ifdef _IPE_ENABLE //if IPE is used in project too
      //seg1 = any read + write persistent variables
      //seg2 = ipe = read + write + execute access
      //seg3 = code, read + execute only
      mpu_segment_border1 = fram_ipe_start >> 4;
      mpu_segment_border2 = fram_rx_start >> 4;
      mpu_sam_value = 0x1573; // Info R, Seg3 RX, Seg2 RWX, Seg1 RW
    #else
      mpu_segment_border1 = fram_rx_start >> 4;
      mpu_segment_border2 = fram_rx_start >> 4;
      mpu_sam_value = 0x1513; // Info R, Seg3 RX, Seg2 R, Seg1 RW
    #endif
  #endif
  #ifdef _MPU_LOCK
    #ifdef _MPU_ENABLE_NMI
      mpu_ctl0_value = MPUPW | MPUENA | MPULOCK | MPUSEGIE;
    #else
      mpu_ctl0_value = MPUPW | MPUENA | MPULOCK;
    #endif
  #else
    #ifdef _MPU_ENABLE_NMI
      mpu_ctl0_value = MPUPW | MPUENA | MPUSEGIE;
    #else
      mpu_ctl0_value = MPUPW | MPUENA;
    #endif
  #endif
#endif

/****************************************************************************/
/* Include peripherals memory map                                           */
/****************************************************************************/

-l msp430fr6989.cmd
```