Model Rocket - Wireless Telemetry

Submitted in partial fulfillment of the requirements for the Bachelor of Science Degree

Department of Electrical Engineering
School of Engineering and Applied Science
Columbia University

Clark Granum
Duncan Jackson

EE3390 Electronic Circuits Design Final Project
# Table of Contents

1. General Overview .................................................................................................................. 3  
   1.1 Executive Summary ........................................................................................................... 3  
   1.2 Block Diagrams .............................................................................................................. 3  
       1.2.1 Transmitter System Block Diagram ....................................................................... 3  
       1.2.2 Receiver System Block Diagram ........................................................................... 4  

2. Design Details ....................................................................................................................... 5  
   2.1 Power Distribution and Power Budget / Current Draw ...................................................... 5  
   2.2 Transmission System Block Details ................................................................................. 6  
       2.2.1 Accelerometer (ADXL345) ..................................................................................... 6  
       2.2.2 Microcontroller (MSP430) .................................................................................... 6  
       2.2.3 RF Transceiver (CC1101) and Antenna ................................................................. 7  
       2.2.4 Detailed Circuit Schematic of Transmitter System .................................................. 9  
   2.3 Receiver System Block Details ......................................................................................... 10  
       2.3.1 RF Transceiver (CC1101) and Antenna ................................................................. 10  
       2.3.2 Microcontroller (MSP430) .................................................................................... 10  
       2.3.3 Arduino & Matlab Details .................................................................................... 10  
       2.3.4 Detailed Circuit Schematic of Receiver System ....................................................... 12  

3. Results .................................................................................................................................. 13  

4. Bill of Materials .................................................................................................................... 18  

5. Health, Safety & Environmental Issues ............................................................................... 18  

6. Comment on Benefits of TI parts ....................................................................................... 19  

7. Possible and Further Improvements .................................................................................... 20  

8. Criticism of the Course ........................................................................................................ 20  

9. Acknowledgements ............................................................................................................. 21  

10. References ......................................................................................................................... 21  

11. Appendix ........................................................................................................................... 22  
    11.1 Code For Transmitter MCU ....................................................................................... 22  
    11.2 Code For Receiver MCU ......................................................................................... 30  
    11.3 Arduino Slave SPI Code ............................................................................................ 37  
    11.4 Matlab Code to Interface With Arduino and Plot Data ............................................ 40
1. General Overview

1.1 Executive Summary

The main aim of this project was to design, test and build a rocket telemetry system that could be housed in the nose cone of an ESTES model rocket and is capable of wirelessly transmitting live acceleration data from the in-flight rocket to a base station up to a range of approximately 200ft. At the base station, the circuitry for the receiver (i.e. RF transceiver and microcontroller) was connected to a computer via an Arduino Duemilanove, which acted as a SPI\(^1\) slave to the microcontroller that relays data via its USB interface\(^2\), thereby producing a live plot of the rockets displacement in Matlab. This will allow for remote tracking of the rockets position in a three dimensional space with the initial launch point as a reference origin. Since the sampling rate chosen was moderately fast, it was assumed that the rocket was undergoing uniformly accelerated linear motion on all three axes during each sampling interval throughout the duration of the flight. Ultimately, Frequency-shift keying (FSK) modulation was utilized as the mode of transmission with an operating frequency of 915MHz. The data manipulation in Matlab would utilize the Data Acquisition Toolbox and result in four active graphs being displayed, three of which would display the displacement of the rocket versus time and the fourth being a 3-D line plot of the rocket’s trajectory in a 3-D space.

1.2 Block Diagrams

1.2.1 Transmitter System Block Diagram

The block diagram illustrating our implementation of the transmitter system can be found in figure 1 below and will be housed in the nose cone of the rocket. It consists of a 3-Axis Accelerometer and RF transceiver that are both connected to a MSP430, acting as a SPI master to both devices.

---

\(^1\) Serial Peripheral Interface – synchronous serial data link standard that operates in full-duplex mode

\(^2\) Arduino was programmed as a slave allowing for the utilization of the on-board FTDI chips and USB interface
The accelerometer and RF transceiver act as slaves to the microcontroller and are connected via a 4-wire SPI interface, illustrated in figure 2 below. A complication in the transmitter is that the accelerometer and RF chip have different SPI polarity and phase, meaning the polarity and phase of the microcontroller were constantly changing\(^3\). The microcontroller uses a chip select pin that chooses the desired chip to be read and/or written to. If this pin is not active on a particular chip, SPI will be passive, and the chip will not receive incoming data or send outgoing data. After initializing and enabling the accelerometer in measure mode, data is subsequently produced and stored in the FIFO registers by the 3-axis sensor on-board the accelerometer chip every 10 ms. Data is subsequently sent out on SDO\(^4\) pin after a burst read operation is requested by the microcontroller\(^5\). The microcontroller reads the accelerometer’s FIFO and writes it to the TX FIFO on the RF transceiver, which subsequently modulates and transmits the data in packets at 915 MHz after a TX command strobe is issued by the microcontroller.

![General 4-Wire SPI Connection Diagram](image)

**Figure 2: General 4-Wire SPI Connection Diagram**

### 1.2.2 Receiver System Block Diagram

Figure 3 illustrates our implementation of the receiver for the telemetry base station. It consists of the same model RF transceiver in the transmitter system and an Arduino Duemilanove, both of which act as SPI slaves to another MSP430 acting as a SPI master.

After the microcontroller initializes the RF transceiver into reception mode (RX mode), the antenna will pick up modulated signals from the air that match the expected preamble and sync word bytes. The receiver filters out the preamble and sync word bytes and stores the payload in the RX FIFO. The microcontroller then accesses this FIFO and writes the data to the SPI Data Register (SPDR) of the arduino. The arduino is programmed as a slave whose clock polarity and phase are matched for the RF chip to ensure simple compatibility. The Arduino then sends the contents of the register to the computer using the ‘Serial.print’ function and Matlab interprets the data and plots the respective displacement graphs for each axis along with its trajectory.

---

\(^3\) Polarity determines the state of the clock, high or low, when idle. Phase determines at which clock edge data is sampled
\(^4\) Serial Data Output (SDO) for SPI 4-Wire Interface
\(^5\) Multiple-byte read operation to read data on all axes of interest thereby preventing data losses due to single-byte read operation. See Section 2.2.1 for details
2. Design Details

2.1 Power Distribution and Power Budget / Current Draw

Fortunately, all chips in both the receiver and transmitter system can be operated at around 2.0V to 3.6V. The transmitter system is powered by stacking two triple AAA batteries in series. The receiver system is powered via the 3.3V pin on the arduino. A power budget was performed on the transmitter circuit, seen in Table 1, as it runs on batteries rather than the USB port of a laptop as the receiver does.

Table 1: Power budget for the transmitter system.

<table>
<thead>
<tr>
<th>Part (and state)</th>
<th>Current Draw</th>
<th>Percent of time in this state</th>
<th>Battery lifetime (time to discharge each AAA from 1.5V to 1.0V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC1101 (Idle)</td>
<td>1.7mA</td>
<td>52%</td>
<td></td>
</tr>
<tr>
<td>CC1101 (TX)</td>
<td>30.7mA</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>CC1101 (Idle to TX including calibration)</td>
<td>8.4mA</td>
<td>8%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC1101 Average Current Draw</td>
<td>13.84mA</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Accelerometer (Measure)</td>
<td>0.145mA</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>MCU (Active Mode)*</td>
<td>2mA</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Average Current Draw</td>
<td>15.981mA</td>
<td></td>
<td>~ 65-80 hours</td>
</tr>
</tbody>
</table>

*This is an estimate and includes periodic flash memory access.

Figure 3: Receiver System Block Diagram
2.2 Transmission System Block Details

2.2.1 Accelerometer (ADXL345)

The ADXL345 accelerometer is a triple axis digital accelerometer that is accessed by a three or four wire SPI interface. In this system, a 4-wire interface is used. The accelerometer is programmed to produce data at a rate of 100Hz resulting in a new data sample every 10ms. The typical current draw is 145uA at this data rate. The accelerometer also produces an active high interrupt on its INT2 pin whenever new data is available. This interrupt is useful as it notifies the MCU rather than requiring the MCU to continuously read an SPI register to find out if data is available.

The format of the data produced is two's complement and the bit resolution is determined by the g-range set by the user. This system uses a ±2g range resulting in 10-bit resolution and 4mg per least significant bit. The data is stored in six bytes, two for each axis, at the top of a 32 level FIFO. The data sheet did not make it very clear how the data is stored in the FIFO and does not explain where the high and low byte of each axis is stored when the user selects MSB versus LSB format for the FIFO. The format of the FIFO was selected to be MSB, which assumes that the first register read from the FIFO, is the high byte of any given axis (For e.g. the x-axis for the ADXL345).

2.2.2 Microcontroller (MSP430)

The MSP430F2013 combined with the EZ430 USB development tool made the completion of this project possible. The F2013 was chosen because of its simple 14-pin package and most importantly the ability to communicate with other chips using SPI. Its simplicity makes it an ideal choice for a low power telemetry system. The 14 pins were more than sufficient to handle SPI communication between the RF transceiver and accelerometer as well as handle each part’s interrupts. Code was written in C with Texas Instrument’s Code Composer Studio and flashed to the microcontroller using the EZ430 emulator. Code writing for the MSP430 was simple and was guided by the extensive example library given by TI.
Figure 6 provides a basic state diagram for the microcontroller on the transmitter side. The simplest way to explain how the MCU on the transmitter side works is to compare it to an assembly line; it waits for a data ready interrupt from the accelerometer, it then reads the data, passes it to the RF chip, waits for the data to be sent, and returns to do it all over again. This process is all completed well within the 10ms interval at which data is produced by the accelerometer. This means the MCU spends most of its time waiting for interrupts. It is certain the interval at which the accelerometer produces data can be reduced and not affect the simple operation seen in the state diagram, however, this would require a higher over the air data rate reducing the range at which data can be reliably transmitted.

```
Power Up -> Configure MCU -> Wait ~100ms

<table>
<thead>
<tr>
<th>Put accelerometer in measure mode</th>
<th>Program RF and Accelerometer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Wait for data ready interrupt from accelerometer</th>
<th>Read the acceleration data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write to RF TX FIFO</td>
</tr>
<tr>
<td></td>
<td>Issue TX command strobe</td>
</tr>
</tbody>
</table>
```

**Figure 6: State Diagram for MCU on transmitter.**

### 2.2.3 RF Transceiver (CC1101) and Antenna

The CC1101 chip on the transmitter is responsible for sending acceleration data wirelessly over the air. Figure 7 shows a detailed circuit, not including decoupling capacitors, that is present on the CC1101 evaluation module used for the telemetry system. The Digital Interface in the schematic is synonymous with the SPI interface. The diagram also includes a matching network to match the 50-Ω impedance of the antenna.

The most useful component of the CC1101 is its simple packet handling process. The CC1101 expects a certain number of bytes in each packet given by the value in the PKTLEN register and transmits them when a TX strobe is issued. However, the expected number of bytes must be available in the TX FIFO or an underflow will occur. Fortunately, this telemetry system operates in a fixed packet length environment and the MCU is always
reading 6 bytes from the accelerometer and writing them to the RF chip. This simple operation ensures underflow will not occur. When issued the TX strobe, the RF chip will automatically calibrate as specified by the initial programming of the chip and then begin to transmit a sync word. The 16-bit sync word is transmitted twice for a total of 32 bits of sync word prior to sending the packet. The RF chip sends an interrupt to the MCU when the 32 bits of sync word has been sent and begins to send the packet. CC1101 sends another interrupt to the MCU when the packet has been sent. As programmed, the RF chip returns to the IDLE state when a packet is finished.

There is also a CRC check done over the data payload and two bytes are appended to the end of the packet to be filtered out on the receiver end. Packets were also whitened using the CC1101’s data whitener and sent at 38.4kBaud using Gaussian Frequency Shift Keying (GFSK) modulation with a 915MHz carrier. Most of the suggested settings for channel filter bandwidth, deviation, and channel spacing for this data rate and modulation scheme given by SmartRF Studio were used.

The output power was programmed to 10dBm using the supplied 2dB gain, 50-Ω antenna. Initial testing showed an RSSI at the receiver end of approximately -98dBm at a range of 210 feet through significant obstruction.

Another useful feature of the CC1101 is the chip status byte sent over the SPI bus when a SPI header is issued by the MCU. The status byte tells the MCU what state the RF chip is in and how many bytes are free in the TX FIFO or how many bytes are available to be read in the RX FIFO depending on whether the SPI operation is read or write. While not explicitly used in the microcontroller code, the status byte was very useful in the debugging process.

Figure 7: CC1101 circuit for operation at 915MHz.
2.2.4 Detailed Circuit Schematic of Transmitter System

Figure 8: Detailed Circuit Schematic for Transmitter. This circuit does not include 100nF decoupling capacitors between the AVDD pins and the DVDD pin to ground on the RF chip.
2.3 Receiver System Block Details

2.3.1 RF Transceiver (CC1101) and Antenna
The same CC1101 chip and matching network seen in Figure 7 is used in the receiver system. However, the receiver differs in that it listens for sync word bytes and receives the packet after receiving a programmed correct number of sync word bits. The receiver automatically de-whitens the data and filters out the CRC bytes and writes the data payload to the RX FIFO. Optionally, two status bytes are appended to the payload. The first byte is the RSSI signal strength indicator for the received packet and the second byte indicates if the CRC check passed.

2.3.2 Microcontroller (MSP430)
Another MSP430F2013 is used in the receiver system. This MCU reads out incoming packets from the RF RX FIFO after being sent an interrupt that a packet has been received. The data, along with the RSSI value and CRC status, is sent to the Arduino. The state diagram for the microcontroller on the receiver is visible in Figure 9.

![State Diagram for MCU on Receiver](image)

**Figure 9: State diagram for MCU on receiver.**

2.3.3 Arduino & Matlab Details
To configure the Arduino as an SPI slave, the Slave Select (SS) pin is set as an input along with the SCLK and MOSI pins. The MISO pin is set as an output although due to the function of the Arduino in the circuit, this was not necessary. Additionally, the SPI logic on the arduino ranges from 0 to 5 V, which is different from the logic on the other ICs. The logic high on the arduino is anything higher than 3 V. Since the microcontroller is powered by the 3.3V pin on the arduino this means that the SPI logic high on the microcontroller ranges from 3.05 to 3.3V. Therefore MISO was disconnected to minimize potential errors. When SS is held low, the SPI is activated and when SS is driven high, all pins are inputs, and the SPI is passive, which means that it will not receive incoming data. Additionally, when
the SS pin is driven high, the SPI slave will immediately reset the send and receive logic, and drop any partially received data in the Shift Register.

The SPI control register (SPCR) on the arduino was initialized as a SPI slave by setting the MSTR bit (Master/Slave Select) to logic zero. Configuring the SS as an input however will automatically clear MSTR once driven low and thus acts as a safeguard by ensuring that the arduino is acting as a slave. The board was also initialized to SPI Data mode 0 where the clock polarity is such that SCLK is low when idle and the clock phase is such that data is sampled on the rising edge of SCLK. This was chosen to be the case as the master clocks out data on every rising edge, and subsequently receives data on the falling edge. Since SCLK was set as an input, no particular clock rate was set since it has no effect on the device configured as a slave.

In establishing a connection with Matlab, a handshake protocol is setup up where Matlab listens in on its scanner waiting for a particular initial byte. Once Matlab notifies the arduino that this byte has been received, the arduino finishes initializing in the main method and enters its primary loop. At this time a ‘slave receive’ command is called eight times during each loop in order to receive two bytes of data for each axis along with the RSSI and CRC indicator bytes. Each byte is written to the arduino’s SPI Data Register, subsequently stored in an array, and sent to Matlab using the ‘Serial.print’ function. The arduino first converts the binary data value to a two’s complement integer. After scanning each byte of data in, Matlab converts each byte to a decimal value from the two’s complement representation. Matlab then uses the equations of uniformly accelerated linear motion to deduce the displacement on each axis during each sampling interval.
2.3.4 Detailed Circuit Schematic of Receiver System

Figure 10: Detailed Circuit Schematic for Receiver. This circuit also does not include 100nF decoupling capacitors between the AVDD pins and the DVDD pin to ground on the RF chip.
3. Results

After receiving many of the parts, the testing process was lengthy and included numerous instances of re-flashing code to the microcontrollers to see if an important change occurred. Most debugging was performed using an oscilloscope to see how the SPI interface was working between the MCU and the slave devices. Figure 11 shows the SPI interface in action. In this case, the yellow signal is the clock from the MCU and the green line is data being sent from the MCU to the RF chip. In the figure, the data in hexadecimal is 0x35, which is the TX command strobe being issued.

![Figure 11: TX command strobe being issued on the SPI bus.](image)

Figure 12 shows another case of an SPI transfer. In this scope shot, the chip select line is shown in purple along with the clock in yellow and the data coming in to the microcontroller from the RF chip in green. The chip select line is low, meaning the SPI interface between the MCU and RF chip is active. The data coming in happens to be the status byte sent by the RF chip and details the state of the chip. In this case it shows hexadecimal 0x06, meaning the crystal oscillator is running, it’s in IDLE state, and it has six bytes (the data payload length) available to be read from the RX FIFO.
A complete SPI timing diagram for the transmitter circuit is visible in Figure 13. The purple line is the chip select line for the accelerometer. This line is low when the MCU is interfacing with it. The diagram shows the MCU reading the data FIFO in the accelerometer and subsequently sending it to the RF chip. This scope shot also shows the polarity of the clock shifting from operations between the RF part and accelerometer.

Figure 14 also shows a complete SPI diagram, but for the receiver circuit. The purple line is the chip select for the RF receiver. Moving from left to right in the screenshot, the first (and longest) period when the chip select line is low is data being pulled from the RX FIFO. The second and third sections where the line is low corresponds with the MCU reading the RSSI and CRC status registers. In the last section, at the very right side of the scope shot, an RX command strobe is issued telling the RF chip to begin listening for packets. This is issued only after data is written to the Arduino.

We were also able to test the RF transceiver using a spectral analyzer from the Columbia Integrated Systems Laboratory. The results confirmed that the operating frequency was 915MHz with an output power of approximately 9.89 dBm.
Figure 13: SPI timing diagram for transmitter.

Figure 14: SPI timing diagram for receiver.
The completed system, including the transmitter and receiver circuits, was successfully able to transmit acceleration data and correctly receive the same data at the receiver end. We had difficulty tracking displacement but when plotting raw acceleration data on the z-axis while moving the accelerometer up and down there was a clear correlation between the motion and the resulting time versus acceleration plot. There is also an internal offset in the accelerometer that had to be taken into account. This offset means that returned acceleration values are not exactly zero on an axis that is not moving. Figure 17 shows displacement on the z-axis when moving the z axis up and down. The sine wave-like portions correspond to this up and down motion.

Figure 15: Completed transmitter circuit. Shows accelerometer, RF, MCU, and battery holder. The red tape prevents metal from touching where wires are soldered together.
Figure 16: Completed Receiver Circuit. Shows RF, MCU and Arduino.

Figure 17: Illustration of Movement on one axis
4. Bill of Materials

<table>
<thead>
<tr>
<th>Component</th>
<th>Part Number</th>
<th>Price/Unit</th>
<th>Quantity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arduino</td>
<td>Arduino Duemilanove with ATmega328</td>
<td>$0</td>
<td>1</td>
<td>$0</td>
</tr>
<tr>
<td>3-axis Digital Accelerometer</td>
<td>EVAL-ADXL345Z</td>
<td>$39.40</td>
<td>2</td>
<td>$78.80</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>EZ430-F2013 USB Development Stick</td>
<td>$0</td>
<td>2</td>
<td>$0</td>
</tr>
<tr>
<td>Transceiver (Receiver &amp; Transmitter Combo)</td>
<td>RF-C1101 Eval Board</td>
<td>$0</td>
<td>2</td>
<td>$0</td>
</tr>
<tr>
<td>AAA batteries</td>
<td></td>
<td>$0.50</td>
<td>20</td>
<td>$9.99</td>
</tr>
<tr>
<td>Digital SPDT Switch</td>
<td>TS3A44159</td>
<td>$0</td>
<td>3</td>
<td>$0</td>
</tr>
<tr>
<td>Battery Holders</td>
<td>BC2AAAW-ND</td>
<td>$0.82</td>
<td>4</td>
<td>$3.28</td>
</tr>
<tr>
<td>.100” Pitch Cables (Female)</td>
<td>M1AXA-1436R-ND</td>
<td>$3.25</td>
<td>2</td>
<td>$6.50</td>
</tr>
<tr>
<td>.100” Pitch Cables (Male)</td>
<td>MIKXX-1436R-ND</td>
<td>$8.36</td>
<td>2</td>
<td>$16.72</td>
</tr>
<tr>
<td>.050” Pitch Cables</td>
<td>FFMD-10-S-12.00-01</td>
<td>$7.25</td>
<td>4</td>
<td>$29</td>
</tr>
</tbody>
</table>

5. Health, Safety & Environmental Issues

This product is safe to handle with no prior instruction or safety equipment. All components are low voltage (i.e. under 5 volts). All parts are secured and there is little risk of electrical shock. With regards to its operating frequency, the 915MHz band should not be used in countries outside of Region 2 (which includes the Americas) in accordance with the International Telecommunication Union Region. This band has been shared with license-free error-tolerant communications applications such as wireless LANs and cordless phones. Since unlicensed devices are already required to be tolerant of Industrial, Scientific and Medical (ISM) emissions in bands such as 915MHz, unlicensed low power uses are generally able to operate in these bands without causing problems for ISM users. Therefore our application does not break any FCC regulations and has no health or environmental issues.

---

6 Prices listed as zero were due either to its availability in lab, or due to the free Texas Instrument samples afford from the competition
6. Comment on Benefits of TI parts

RF CC1101 (Transmitter):

As we were attempting to implement a rocket telemetry system, the RF transmitter enabled the transmission of acceleration data over the air. The chip made packet handling very simple while allowing for simple error checking at the receiver (another CC1101). The data rates and output power available on the CC1101 fit well for the necessary range that was required for our system. The CC1101’s ability to automatically transition between transmit, calibrate and idle state also is helpful in saving power. The C1101 was also sufficiently small such that it could fit in the nose cone of the rocket without adversely affecting the flight of the rocket.

MSP430 F2013 (Microcontroller):

The MSP430’s SPI interface allows for communication between the accelerometer, RF parts and the arduino by acting as an SPI master. The SPI interface was very easy to use and the programming of the microcontroller’s registers was very simple. The general digital out and in pins were also used as interrupts and chip selects on the SPI interface. This allows for simple reading, processing and writing data between different chips including the CC1101.

RF CC1101 (Receiver):

While acting as a receiver, the CC1101 was able to properly handle the demodulation, reception, and filtering of data packets. The chip automatically filters out sync words and preambles, leaving the desired payload along with the signal strength indicator and CRC status. Packets are simply received by programming the expected data rate and expected packet length. The CC1101 also has a receiver sensitivity that is adequate at our data rate to receive data across a sufficient range.
7. Possible and Further Improvements

Currently our circuit is capable of transmitting accelerometer readings and is able to interface with Matlab well. The system properly received correct accelerometer readings for motion on one axis but we did not test on all three axes because it only became apparent near the end of the semester that a gyroscope would be needed in addition to the accelerometer to track a rocket. We did not get the chance to launch the rocket because of this constraint but our circuit does meet the size requirement and RF range needed for simple rocket telemetry. In short, inclusion of a gyroscope is an important future improvement.

Also to get the accelerometer and antenna oriented properly within the rocket requires significant contortion of wires whose connections are already crude. To improve on this, we would implement a PCB design that would allow the various components to be plugged into the PCB board and limit the wires and soldering needed in our first implementation of the circuit. Short wire distances are also ideal for SPI interfaces, therefore data transmission and reception losses as well as data corruption could potentially be improved upon along with the aesthetic appearance of the final product resulting in a potentially small, compact and lightweight device. In addition to PCB design, functionality and features of the device could be expanded so as to be a multipurpose telemetry system. The RF transceiver used also has a temperature sensor but due to the time constraint of the project, we were unable to effectively explore its usage, as our primary concern was the successful transmission of acceleration data.

8. Criticism of the Course

This was a self-paced and self-driven course that allowed us to express ourselves through the freedom to select our own design project. Having taken this course simultaneously with Electrical Engineering Practice (ELEN E3399), we do feel that we spent significant time at the start of the semester trying to finalize and have a project proposal that was appropriate for the expectations of the course. Needless to say, we had even less time to do background research, test and implement our design. This would have been quite useful when figuring out the proper connections between circuit elements and perhaps would have led to a PCB-implementation of the design. We probably signed up for the class without truly knowing what to expect and just treating it as a requirement, which is probably due in part to not taking ELEN E3399 previously. At times, the pacing and progress of the project was difficult due to reasons other than those mentioned prior. For instance, late orders and late arrivals of parts always seemed to be the limiting factor when it came to progress being made. It was only at the very end that we could properly test the arduino interface due to a lack of proper cables and connections.
9. Acknowledgements

We would like to acknowledge the Electrical Engineering Department at Columbia University, and in particular Professor David Vallancourt and John Kazana for providing us with good insight and support throughout the duration of the semester, with timely advice regarding the best approach that should be taken for any challenges that we encountered. We would also like to thank Professor Tsividis and his research group for allowing us access to a spectral analyzer during testing.

10. References


11. Appendix

11.1 Code For Transmitter MCU (an MSP430 F2012 was used instead of F2013)

#include <msp430x20x2.h>

void accelerometer_SPI()
{
    USICNT |= 16;
    while((USICTL1 & USIIFG) != 0x01);
}

void accelerometer_SPI8()
{
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
}

void RF_SPI()
{
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
}

void RF_SPI8()
{
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
}

void build_USISR(char high, char low)
{
    USISRH= high;
    USISRL= low;
}
void assert_Accelerometer(void)
{
    P1OUT= 0x04;
}

void assert_RF(void)
{
    P1OUT= 0x02;
}

void deassert_CS(void)
{
    P1OUT= 0x06;
}

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; //STOP WATCHDOG TIMER
    P1DIR= 0x06; //P1.1 AND P1.2 USED FOR CS PINS ON ACCELEROMETER AND RF (OUTPUT), P1.3 AND
P1.4 INTERRUPTS ACCELEROMETER AND RF (INPUT)
P1OUT= 0x06; //KEEP CS PINS HIGH WHEN NOT USING SPI AND PULL UP P1.3 AND P1.4
    _BIS_SR(OSCOFF); //TURN EXTERNAL OSCILLATOR STATUS OFF

    DCOCTL= DCO1 + DC00; //INTERNAL DCO 12.0 TO 18.5 MHz
    BCSCTL1= XT2OFF + DIVA_0 + RSEL3 + RSEL2 + RSEL1 + RSEL0; //SET ACLK 12.0 TO 18.5 MHz
    BCSCTL2= SELM0 + DIVM0 + DIVS_1; //SET MCLK= DCOCLK/2, SMCLK= DCOCLK/2

    USICTL0 |= USIPE7 + USIPE6 + USIPE5 + USIMST + USIOE; //USI CONTROL REGISTER
    USICTL1 |= USIE + USICKPH; //USI INTERRUPT ENABLED, SET PHASE= 1 FOR
RF
    USICKCTL = USIDIV_1 + USISSEL_2; //SCLK= SMCLK/2 Pol= 0 FOR RF.
    USICTL0 &= ~USISWRST;
    USICNT |= USI16B; //USI RELEASED FOR OPERATION
    //ENABLE 16 BIT OPERATION

    char x0= 0x00;
    char x1= 0x00;
    char y0= 0x00;
    char y1= 0x00;
    char z0= 0x00;
    char z1= 0x00;
```c
int packetsent = 0;
int dataready = 0;

int wait = 100;
int loop = 8000;
while (wait > 0)  // wait ~100ms
{
    while (loop > 0)
    {
        loop--;
    }
    loop = 8000;
    wait--;
}

// BEGIN RF PROGRAMMING

int RF_ready = 0;
while (!RF_ready)  // WAIT FOR CHP_RDYn on RF
{
    if(!(0x10 & P1IN))
    {
        RF_ready = 1;
    }
}

build_USISR(0x03, 0x47);  // FIFOTHR IS 0x47, 33 TX, 32 RX THRESHOLD BYTES
assert_RF();
RF_SPI();
deassert_CS();

build_USISR(0x04 | 0x40, 0xD3);  // ADDRESS FOR SYNC WORD HIGH BYTE, SYNC1 HIGH BYTE IS 0xD3
assert_RF();
RF_SPI();
build_USISR(0x91, 0x06);  // SYNC0 LOW BYTE IS 0x91, PKTLEN IS 0x06(6 payload)
RF_SPI();
build_USISR(0x00, 0x44);  // PKTCTRL1 IS 0x00, NO CRC AUTOFLUSH, PKTCTRL0 IS 0x44, DATA WHITENING, USE FIFO, CRC ON, FIXED PKT LENGTH MODE
RF_SPI();
deassert_CS();
```
build_USISR(0x0B, 0x06);  //FSCTRL1 IS 0x06, FREQUENCY IF IS 152.34KHz
assert_RF();
RF_SPI();
deassert_CS();

build_USISR(0x0D | 0x40, 0x23);  //FREQ2 IS 0x23, FCARRIER= 914.999MHz
assert_RF();
RF_SPI();
build_USISR(0x31,0x3B);  //FREQ1 IS 0x31, FREQ0 IS 0x3B
RF_SPI();
build_USISR(0xCA, 0x83);  //MDMCFG4 IS 0xCA, MDMCFG3 IS 0x83, CHANNEL FILTER BANDWIDTH IS 103kHz, DATA_RATE IS 38.4kBaud
RF_SPI();
build_USISR(0x13, 0x22);  //MDMCFG2 IS 0x13, MDMCFG1 IS 0x22, GFSK, MANCHESTER DISABLE, 32 BIT SYNC, 4 BYTE PREAMBLE
RF_SPI();
build_USISR(0xF8, 0x34);  //MDMCFG0 IS 0xF8, DEVIATN IS 0x34, CHANNEL SPACING IS 200kHz, DEVIATION IS 19kHz
RF_SPI();
deassert_CS();

build_USISR(0x17|0x40, 0x30);  //MCSM1 IS 0x30, GO TO IDLE AFTER TX OR RX PACKET FINISHED
assert_RF();
RF_SPI();
build_USISR(0x18, 0x00);  //MCSM0 IS 0x18, AUTO CALIBRATE WHEN GOING FROM IDLE TO TX
RF_SPI8();
deassert_CS();

build_USISR(0x21 | 0x40, 0x16);  //FOCCFG IS 0x16
assert_RF();
RF_SPI();
build_USISR(0x6C, 0x43);  //BSCCFG IS 0x6C, AGCCTRL2 IS 0x43
RF_SPI();
build_USISR(0x40, 0x91);  //AGCCTRL1 IS 0x40, AGCCTRL0 IS 0x91
RF_SPI();
deassert_CS();

build_USISR(0x21, 0x56);  //FREND1 IS 0x56
assert_RF();
RF_SPI();
deassert_CS();
build_USISR(0x22 | 0x40, 0x10); //FREndo IS 0x10
assert_RF();
RF_SPI();
build_USISR(0xE9, 0x2A); //FSCal3 IS 0xE9, FSCal2 IS 0x2A
RF_SPI();
build_USISR(0x00, 0x1F); //FSCal1 IS 0x00, FSCal0 IS 0x1F
RF_SPI();
deassert_CS();

build_USISR(0x2C | 0x40, 0x81); //TEST2 IS 0x81
assert_RF();
RF_SPI();
build_USISR(0x35, 0x09); //TEST1 IS 0x35, TEST0 IS 0x09
RF_SPI();
deassert_CS();

build_USISR(0x00, 0x06); //GDO2 CONFIG, SET FOR INTERRUPTS
assert_RF();
RF_SPI();
deassert_CS();

build_USISR(0x02, 0x2E); //GDO0 SET TO HIGH IMPEDANCE
assert_RF();
RF_SPI();
deassert_CS();

build_USISR(0x3E, 0xC3); //PATABLE 10dBm
assert_RF();
RF_SPI();
deassert_CS();

build_USISR(0x3B, 0x00); //FLUSH TX FIFO
assert_RF();
RF_SPI8();
deassert_CS();

//END RF PROGRAMMING

//BEGIN PROGRAMING ACCELEROMETER
USICTL1 &= ~USICKPH;  // ACCELEROMETER TAKES PHASE = 0
USICKCTL |= USICKPL;  // ACCELEROMETER TAKES POLARITY = 1
USICTNT |= 8;  // MAY NEED TO HAVE DUMMY SPI TO TAKE EFFECT

while((USICTL1 & USIIFG) != 0x01);

build_USISR(0x31, 0x08);  // PROGRAM DATA_FORMAT, FULL RESOLUTION, Sign EXTENSION, 2G RANGE
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

build_USISR(0x2C, 0x08);  // PROGRAM BW_RATE, NORMAL POWER MODE, 100Hz DATA RATE
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

build_USISR(0x38, 0x90);  // PROGRAM FIFO_CTL, STREAM MODE, 16 samples to trigger watermark
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

build_USISR(0x2F, 0x80);  // PROGRAM INT_MAP, SET DATA_READY TO INT2 PIN, ALL OTHERS TO INT1 PIN
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

build_USISR(0x2E, 0x80);  // PROGRAM INT_ENABLE, ENABLE DATA READY INTERRUPT TO INT2 PIN
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

build_USISR(0x2D, 0x08);  // PROGRAM POWER_CTL, PUT IN MEASURE MODE TAKE OUT OF STANDBY
assert_Accelerometer();
accelerometer_SPI();
deassert_CS();

// ACTUAL RUNNING CODE BEGINS HERE

while(1)
{
    packetsent = 0;
dataready=0;

while(!dataready) //WAIT FOR DATA_READY INTERRUPT FROM ACCELEROMETER TO TRIGGER MCU
{
    if((P1IN & 0x08))
    {
        dataready= 1;
    }
}

//READ THE DATA
build_USISR(0xF2, 0x00); //READ | IN BURST | FIFO ADDRESS, 0x00 ARBITRARY
assert_Accelerometer();
accelerometer_SPI8();
accelerometer_SPI();
x0 = USISRH;
x1 = USISRL;
accelerometer_SPI();
y0 = USISRH;
y1 = USISRL;
accelerometer_SPI();
z0 = USISRH;
z1 = USISRL;
deassert_CS();

//END READ THE DATA

//SEND THE DATA TO THE RF TX FIFO
USICTL1 |= USICKPH; //RF TAKES PHASE= 1
USICKCTL &= ~USICKPL; //RF TAKES POLARITY= 0
USICNT |= 8; //MAY NEED TO HAVE DUMMY SPI TO TAKE EFFECT
while((USICTL1 & USIIFG) != 0x01);
build_USISR(0x7F, 0x00); //WRITE TO TX FIFO IN BURST
assert_RF();
RF_SPI8();
build_USISR(x1, x0);
RF_SPI();
build_USISR(y1, y0);
RF_SPI();
build_USISR(z1, z0);
RF_SPI();
deassert_CS();

//SEND TX STROBE
build_USISR(0x35, 0x00);
assert_RF();
RF_SPI8();
deassert_CS();

while(!packetsent)                     //WAIT FOR SYNC WORD TO BE SENT
{
    if((0x10 & P1IN))
    {
        packetsent=1;
    }
}
packetsent=0;
while(!packetsent)                  //WAIT FOR PACKET TO BE SENT
{
    if(!(0x10 & P1IN))
    {
        packetsent=1;
    }
}

//END SEND DATA TO RF

//RETURN TO READ ACCELEROMETER

USICTL1 &= ~USICKPH;  //ACCELEROMETER TAKES PHASE= 0
USICKCTL |= USICKPL;  //ACCELEROMETER TAKES POLARITY= 1
USICNT |= 8;          //MAY NEED TO HAVE DUMMY SPI TO TAKE EFFECT
while(((USICTL1 & USIIFG) != 0x01));
11.2 Code For Receiver MCU

#include <msp430x20x3.h>

void SPI()
{
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
}

void SPI8()
{
    USICNT |= 8;
    while((USICTL1 & USIIFG) != 0x01);
}

void build_USISR(char high, char low)
{
    USISRH= high;
    USISRL= low;
}

void assert_RF(void)
{
    P1OUT= 0x02;
}

void assert_Arduino(void)
{
    P1OUT= 0x04;
}

void deassert_CS(void)
{
    P1OUT= 0x06;
}

void main(void)
{
WDTCTL = WDTPW + WDTHOLD;               //STOP WATCHDOG TIMER
P1DIR= 0x06;   //P1.1 AND P1.2 USED FOR CS PINS ON ARDUINO AND RF, P1.4 INTERRUPT FROM RF
P1OUT= 0x06;   //KEEP CS PINS HIGH WHEN NOT USING SPI
_BIS_SR(OSCOFF);    //TURN EXTERNAL OSCILLATOR STATUS OFF
DCOCTL= DCO1 + DCO0;   //INTERNAL DCO 12.0 TO 18.5 MHz
BCSCTL1= XT2OFF + DIVA_0 + RSEL3 + RSEL2 + RSEL1 + RSEL0;  //SET ACLK 12.0 TO 18.5 MHz
BCSCTL2= SELM0 + DIVM0 + DIVS_1;    //SET MCLK= DCOCLK/2, SMCLK= DCOCLK/2
USICTL0 |= USIPE7 + USIPE6 + USIPE5 + USIMST + USIOE;  //USI CONTROL REGISTER
USICTL1 |= USIIIE + USICKPH;    //USI INTERRUPT ENABLED PHASE IS 1
USICKCTL = USIDIV_1 + USISSEL_2;   //SCLK= SMCLK/2 POL= 0 FOR RF.
USICTL0 &= ~USISWRST;          //USI RELEASED FOR OPERATION
USICNT |= USI16B;             //ENABLE 16 BIT OPERATION

char  x0= 0x00;
char  x1= 0x00;
char  y0= 0x00;
char  y1= 0x00;
char  z0= 0x00;
char  z1= 0x00;
char  CRC= 0x00;
char  RSSI= 0x00;
int   packetreceived= 0;

int   RF_ready= 0;
while(!RF_ready)       //WAIT FOR CHP_RDYn on RF
{            
    if(!(0x10 & P1IN))
    {
        RF_ready=1;
    }
}
build_USISR(0x03, 0x47);  //FIFOTHR IS 0x47, 33 TX, 32 RX THRESHOLD BYTES
assert_RF();
SPI();
deassert_CS();
build_USISR(0x04 | 0x40, 0xD3);  //ADDRESS FOR SYNC WORD HIGH BYTE, SYNC1 HIGH BYTE IS 0xD3
assert_RF();
SPI();
build_USISR(0x91, 0x06); //SYNC0 LOW BYTE IS 0x91, PKTLEN IS 0x06 (6 payload)
SPI();
build_USISR(0x04, 0x44); //PKTCTRL1 IS 0x04, APPEND STATUS, NO CRC AUTOFLUSH, PKTCTRL0 IS 0x44, DATA WHITENING, USE FIFO, CRC ON, FIXED PKT LENGTH MODE
SPI();
deassert_CS();

build_USISR(0x0B, 0x06); //FSCTRL1 IS 0x06, FREQUENCY IF IS 152.34kHz
assert_RF();
SPI();
deassert_CS();

build_USISR(0x0D | 0x40, 0x23); //FREQ2 IS 0x23, FCARRIER = 914.999MHz
assert_RF();
SPI();
build_USISR(0x31, 0x3B); //FREQ1 IS 0x31, FREQ0 IS 0x3B
SPI();
build_USISR(0xCA, 0x83); //MDMCFG4 IS 0xCA, MDMCFG3 IS 0x83, CHANNEL FILTER BANDWIDTH IS 103kHz, DATA RATE IS 38.4kBaud
SPI();
build_USISR(0x13, 0x22); //MDMCFG2 IS 0x13, MDMCFG1 IS 0x22, GFSK, MANCHESTER DISABLE, 32 BIT SYNC, 4 BYTE PREAMBLE
SPI();
build_USISR(0xF8, 0x34); //MDMCFG0 IS 0xF8, DEVIATION IS 0x34, CHANNEL SPACING IS 200kHz, DEVIATION IS 19kHz
SPI();
deassert_CS();

build_USISR(0x16, 0x07); //MCSM1 IS 0x07, GO TO IDLE AFTER TX OR RX PACKET FINISHED
assert_RF();
SPI();
deassert_CS();

build_USISR(0x17 | 0x40, 0x30); //MCSM0 IS 0x30, AUTO CALIBRATE WHEN GOING FROM IDLE TO RX
assert_RF();
SPI();
deassert_CS();

SPI8();
deassert_CS();
build_USISR(0x19 | 0x40, 0x16); //FOCCFG IS 0x16
assert_RF();
SPI();
build_USISR(0x6C, 0x43); //BSCCFG IS 0x6C, AGCCTRL2 IS 0x43
SPI();
build_USISR(0x40, 0x91); //AGCCTRL1 IS 0x40, AGCCTRL0 IS 0x91
deassert_CS();

build_USISR(0x21, 0x56); //FREND1 IS 0x56
assert_RF();
SPI();
deassert_CS();

build_USISR(0x22 | 0x40, 0x10); //FREND0 IS 0x10
assert_RF();
SPI();
build_USISR(0xE9, 0x2A); //FSCAL3 IS 0xE9, FSCAL2 IS 0x2A
SPI();
build_USISR(0x00, 0x1F); //FSCAL1 IS 0x00, FSCAL0 IS 0x1F
deassert_CS();

build_USISR(0x2C | 0x40, 0x81); //TEST2 IS 0x81
assert_RF();
SPI();
build_USISR(0x35, 0x09); //TEST1 IS 0x35, TEST0 IS 0x09
deassert_CS();

build_USISR(0x00, 0x06); //GDO2 CONFIG, SET FOR INTERRUPTS
assert_RF();
SPI();
deassert_CS();

build_USISR(0x02, 0x2E); //GDO0 SET TO HIGH IMPEDANCE
assert_RF();
SPI();
deassert_CS();

build_USISR(0x3A, 0x00); //FLUSH RX FIFO
assert_RF();
SPI8();
deassert_CS();

//END RF PROGRAMMING

//BEGIN MAIN PROGRAM

while(1)
{
    packetreceived= 0;

    build_USISR(0x34, 0x00);    //SEND RX STROBE
    assert_RF();
    SPI8();
    deassert_CS();

    while(!packetreceived)     //WAIT FOR SYNC WORD TO BE RECEIVED
    {
        if((0x10 & P1IN))    //GDO2 ASSERTS WHEN SYNC WORD RECEIVED
        {
            packetreceived=1;
        }
    }
    packetreceived=0;
    while(!packetreceived)     //WAIT FOR PACKET TO BE RECEIVED
    {
        if(!(0x10 & P1IN))     //GDO2 DEASSERTS AT END OF PACKET
        {
            packetreceived=1;
        }
    }
    build_USISR(0xFF, 0x00);    //READ RX FIFO IN BURST
    assert_RF();
    SPI8();
    SPI();
x1 = USISRH;
x0 = USISRL;
    SPI();
y1 = USISRH;
y0 = USISRL;
SPI();
z1 = USISRH;
z0 = USISRL;
SPI();
RSSI = USISRH;
CRC= USISRL;
deassert_CS();

//WRITE TO ARDUINO

build_USISR(0x50, 0x00);  //SEND PREAMBLE BYTE #1
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(0x50, 0x00);  //SEND PREAMBLE BYTE #2
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(x1, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(x0, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(y1, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(y0, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(z1, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(z0, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(RSSI, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

build_USISR(CRC, 0x00);
assert_Arduino();
SPI8();
deassert_CS();

)
### 11.3 Arduino Slave SPI Code

```c
/*
 * Communications Matlab <-> Arduino
 * Arduino acting as SPI slave to MSP430
 * on-board FTDI chip provides SPI-USB interface for acceleration data
 * By: Duncan Jackson
 */

// The microcontroller communicates using SPI, so include the library:
#include <SPI.h>
#define SPI_SCK 13
#define SPI_MISO 12
#define SPI_MOSI 11
#define SPI_SS 10

void SPI_Init(void);
char buffer [6];
char SPI_SlaveReceive(void)
{
    while(!(SPSR & (1<<SPIF))); /* Wait for reception complete */
    return SPDR; /* Return Data Register */
}
void SPI_SlaveInit(void)
{
    SPCR = (1<<SPE)|(0<<MSTR); /* Enable SPI */
}
void setup()
{
    Serial.begin(9600);
    SPI.begin();
    SPI_SlaveInit();
```
for (int c=0; c<8;c++)
{
    buffer[c] = 0x00;
}

pinMode(SPI_SCK,INPUT);
pinMode(SPI_MOSI,INPUT);
pinMode(SPI_SS,INPUT);
pinMode(SPI_MISO,OUTPUT);
SPI.setDataMode((0<<CPOL) | (0 << CPHA)); // set SPI mode 0
establishContact(); // send a byte to establish contact until receiver responds

void establishContact()
{
    while (Serial.available() <= 0)
    {
        Serial.println('A', BYTE); // send a capital A
        delay(300);
    }
}

void loop()
{
    if(((int) SPI_SlaveReceive()) == 80) //PREAMBLE BYTE #1
    {
        if(((int) SPI_SlaveReceive()) == 80) //PREAMBLE BYTE #2
        {
            for (int i =0; i<8;i++)
            {
                buffer[i]= SPI_SlaveReceive();
            }
        }
    }
}
for(int k=0; k<8; k++)
{
    Serial.println((int) buffer[k]);
}
}
11.4 Matlab Code to Interface With Arduino and Plot Data

Accelerometer_Plot.m

% Communications MatLab <-- Arduino
% By Duncan Jackson

% Currently the code is written so as to display a graph of Acceleration in the z-axis versus Time and was
% used to track the movement of an elevator. This implementation would not change the orientation of
% the accelerometer and we were unable to acquire the gyroscope, which is needed for full functionality in
% all 3 axes

function[]= new_accel()
clc;
clear all
Tracking_Time=30;

d = zeros(1, 8);
ax = zeros(1, 2);
ay = zeros(1, 2);
az = zeros(1, 2);
vx = zeros(1, 2);
vy = zeros(1, 2);
vz = zeros(1, 2);
x = zeros(1, 2);
y = zeros(1, 2);
z = zeros(1, 2);
temp_data= zeros(100000, 3);
color= '*b*r';

time=.04; %interval of 10ms is the sampling rate of the accelerometer

figureHandle = figure('NumberTitle','off',...
'Name','Telemetry',...
'Visible','on');
hold on;

s1 = serial('/dev/tty.usbserial-A600agca'); % define serial port which is specific to the computer being
used e.g maybe serial(éCOM5i) on PC
s1.BaudRate=9600; % define baud rate
s1.Timeout= 20;
s1.ByteOrder = 'bigEndian';
s1.RequestToSend = 'on';
set(s1, 'terminator', 'LF'); % define the terminator for println
fopen(s1);
try % use try catch to ensure fclose
w=fscanf(s1,'%s'); % signal the arduino to start collection
if (w=='A') % must define the input % d or %s, etc.
    display('Collecting data'); % establishContact just wants
    fprintf(s1,'%s
','A'); % something in the buffer
end

k=1;
display(Tracking_Time);
t0=tic;

ax(1)= 0; %initial conditions
ay(1)= 0;
az(1)= 0;

while (toc(t0)<=Tracking_Time && k<10000) %supports ~10 minutes of data collection
    for i=1:1:8
        d(i)=fscanf(s1,'%d');
        %b(i)= d(i);
    end
    i1= mod(k, 2)+1;
i0= mod(k-1, 2)+1;
t= (k-1)*time;
{
rssi= (d(7)/2) -74;
if(d(8)>0)
    crc= 3; %CRC failed, data plotted in red
end
if(d(8)<0)
    crc= 1; %CRC passed, data plotted in blue
end
%
}
d= typecast(int8(d), 'uint8'); %get data out of two's complement
d= double(d);

for i= 2:2:6 %get data back into twos complement 16 bit
    d(i)= d(i) + d(i-1)*2^8;
    d(i)= typecast(uint16(d(i)), 'int16');
    d(i)= double(d(i));
end

%ax(i1)=d(2).*0.004*9.8;
%ay(i1)=d(4)*.004*9.8;
az(i1)=d(6)*.004*9.8-9.8-.08-.0352;

%ax_net= (ax(i1)+ax(i0))/2;
%ay_net= (ay(i1)+ay(i0))/2;
az_net= (az(i1)+az(i0))/2;
%b = ax(i1);
%c = az(i1);
%d = ay(i1);
if (k==1)
    %vx(i1) = ax_net*time;
    %vy(i1) = ay_net*time;    %assumes uniformly accelerated
    vz(i1) = az_net*time;
    %x(i1) = 0.5*ax_net*(time^2);    %assumes initial velocity and displacement is zero
    %y(i1) = 0.5*ay_net*(time^2);
    z(i1) = 0.5*az_net*(time^2);
end
if (k>1)
    %vx(i1) = vx(i0)+ ax_net*time;
    %vy(i1) = vy(i0)+ ay_net*time;
    vz(i1) = vz(i0)+ az_net*time;
    %x(i1) = %vx(i1)*time)+0.5*ax_net*(time^2)+x(i0);
    %y(i1) = %vy(i1)*time)+0.5*ay_net*(time^2)+y(i0);
    z(i1) = (vz(i0)*time)+0.5*az_net*(time^2)+z(i0);
end

temp_data(k, :) = [t, z(i1), az(i1)];
toc(t0);
k=k+1;
end
data = temp_data(1:k-1, :);
csvwrite('Telemetry.csv', data);
save('telemetry.mat', 'data');

plot(data(:,1),data(:,3))
set(figureHandle,'Visible','on');
ylabel('Acceleration in Z axis')
xlabel('Time (s)')
title('Acceleration on Z axis Vs. Time', 'FontSize',14)
% subplot(2,2,1)
plot(data(:, 1), data(:, 2), strcat(color(crc), color(crc+1))) %crc determines color of point plotted
set(figureHandle,'Visible','on');
ylabel('Displacement in X axis')
xlabel('Time (s)')
title('Displacement on X axis Vs. Time', 'FontSize',14)
subplot(2,2,2)
plot(data(:, 1), data(:, 5), strcat(color(crc), color(crc+1)))
set(figureHandle,'Visible','on');
ylabel('Displacement in Y axis')
xlabel('Time (s)')
title('Displacement on Y axis Vs. Time', 'FontSize',14)
subplot(2,2,3)
plot(data(:, 1), data(:, 8), strcat(color(crc), color(crc+1)))
set(figureHandle,'Visible','on');
ylabel('Displacement in Z axis')
xlabel('Time (s)')
title('Displacement on Z axis Vs. Time', 'FontSize',14)
subplot(2,2,4)
plot3(data(:, 2), data(:, 5), data(:, 8), strcat(color(crc), color(crc+1)))
set(figureHandle,'Visible','on');
xlabel('X Displacement')
ylabel('Y Displacement')
zlabel('Z Displacement')
title('Trajectory of Rocket', 'FontSize',14)
axis square
%
fclose(s1);

catch error
    fclose(s1); % always, always want to close s1
    display('There was an Error');
end