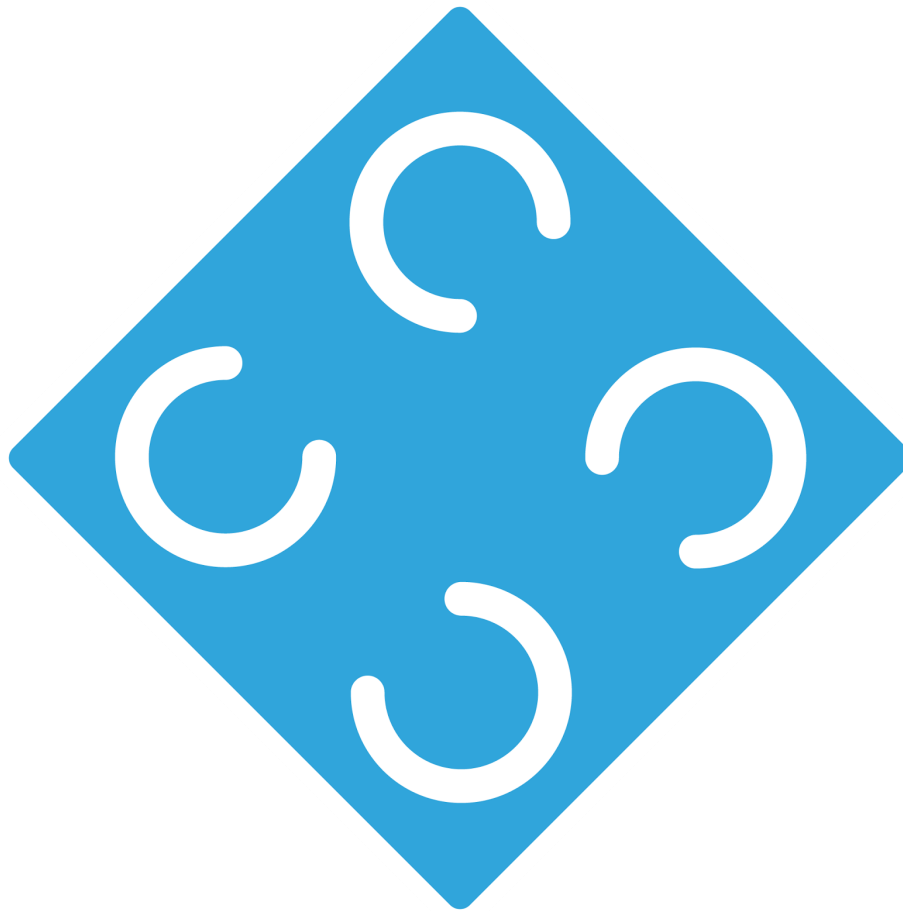


Final Design Report



"The Midfielders"

Sponsor: Dr. Tina Smilkstein
Adam Levasseur and Erik Miller
Capstone Fall 2016 - Winter 2017

Executive Summary	3
Introduction	4
Project Overview	4
Clients and Community Partners	5
Stakeholders	5
Framed Insights and Opportunities	6
Goals and Objectives	7
Outcomes and Deliverables	8
Background	8
Engineering Specifications	10
Personas	11
Use Cases	11
Design Development	13
Final Design	15
Functional Decomposition	16
Design Verification	25
FMEA	25
Design Verification Plan	25
Verification Test Results	25
System Analysis	28
Management Plan	30
References	31
Appendices	32
A1: Detailed Personas	32
A2: Arduino Sketch Source Code	34
Transmitting code:	34
Receiving code:	37
A3: UART File Transmission Source Code	40
README.md	40
Arduino-serial.c	42
Arduino-serial-lib.h	50
Arduino-serial-lib.c	51
A4: Larger Format Images	56
FMEA	56
Design Verification Plan	57

Executive Summary

The goal for this project was to wirelessly transmit data using the midfield wave antenna designed by Stanford's Dr. Ada Poon. This system was originally intended to remotely program an implanted FPGA, but was generalized to transmit data using the same antenna system. The system consists of two microcontrollers equipped with transceiver modules and sends data from one source to the other using a recreated midfield wave antenna and loop antenna.

Introduction

Project Overview

Current logic circuitry in human-implantable hardware cannot be modified without removing the device from the patient via surgery or other means. This makes development of new implanted medical devices costly, time consuming, and more dangerous for test subjects.

While initially focused on FPGA programming, this project is design around the applications of the 2014 Stanford midfield wave antenna—particularly data transfer. The original antenna was designed for improved power transmission at higher frequencies to implanted technology. While this can be useful for powering small devices embedded within the body, this project is a proof of concept that this antenna can also be used for data transfer by utilizing its tissue penetration features in order to send data to an embedded device.

Developers of more traditional electronics have long taken advantage of FPGAs, or Field Programmable Gate Arrays, which, unlike off-the-shelf microcontrollers, use digital logic that can be reprogrammed and changed as new needs arise or bugs in the “hardware” are found. Although FPGAs may significantly reduce costs and time to develop a new medical device, they are typically only able to be reprogrammed via USB cable. This means that if an off the shelf FPGA were implanted, it would still require surgery every time it needs to be reprogrammed; rendering the reduced cost and decreased time to develop the device relatively moot.

The solution is to create a wirelessly reprogrammable FPGA. Danielle Nishida, a Cal Poly alumni, has already created an antenna designed to focus power through human flesh, which we plan to use to transmit data to the remote FPGA.

Clients and Community Partners

Our primary client is Dr. Tina Smilkstein. The project began when Dr. Smilkstein was listening to project presentations from St. Jude Medical on a Pacemaker identifier, and she thought of the development of pacemakers. She connected that with a previous project she had advised, an antenna made to transmit power through flesh, which was made by Danielle Nishida for her master's thesis. Our project initially strived to use this special antenna to remotely program an FPGA, but as we assessed our time requirements and worked with our client we redirected our project to a more general approach: to transmit data through flesh using the midfield wave antenna.

Other recipients of our project's deliverables include future students (that we may not have a chance to meet), who will be using the documentation and prototype to further develop it into a biocompatible implant.

Stakeholders

First and foremost, the stakeholder of this project is the the past developer of the midfield wave antenna, Danielle Nishida (and her project advisor, Dr. Smilkstein). Nishida designed and developed a non-invasive glucose monitor based on a midfield wave antenna originally developed by Dr. Ada Poon at Stanford University. Our project revisits this antenna in order to transmit signals through the human body.

The majority of the project's stakeholders include students and future developers of this project. Beyond our scope, this project can be developed further into biomedical applications, electrical engineering adaptations, and future student projects; therefore our stakeholders may also include future students, developers, and eventually patients.

Framed Insights and Opportunities

This project is designed around the use of Nishida's implementation of the Stanford midfield wave antenna. Our task is to utilize this non-invasive wave-based transmitter in order to send data through human flesh. Once the transmitted data has been received within or beyond the human flesh, we need to process the signal into a digital format such that the new digital signal can be used to reprogram or update devices like FPGAs. This task yields opportunities for design and development.

We used an antenna that was designed to transmit power at an ultra high frequency of about 1.6 GHz; thus we planned to convert an inputted digital signal into a modulated, high frequency analog signal, receive said wireless signal, and process it back into the original digital signal. This allowed us the opportunity to design a digital to analog conversion method that is both compliant with the midfield antenna, and can be modulated in order to receive a distinguishable analog signal at the receiver end of the wireless communication.

Once we successfully transmitted our modulated midfield wave and received it on the other side, the objective was to be able to use that received signal and process it back into a digital signal for more digital usage (i.e. FPGA programming). This

process involved analog to digital conversion, error checking and correction, and signal processing in order to relay the data to other desired programmable devices.

In an interesting twist, we also needed to plan for future communications, potentially after we both have graduated. This project has the potential to be a senior project for future EE, CPE, and/or BMED students, with the goal of taking our proof of concept and developing a more biocompatible implant. While Dr. Smilkstein and other professors may be able to help them, they will likely reference our prototype and documentation. Thus, our documentation needs to be thorough enough to answer any questions they may have in the event that we cannot be reached.

Goals and Objectives

Our main goal was to investigate the data transfer application of the Stanford midfield wave antenna when transmitting through tissue.

In order to accomplish that, we:

- Obtained the midfield wave antenna and a corresponding receiving antenna
- Obtained transceiver chips to convert digital signals to analog signals
- Set up necessary signal processing components for transmitting/receiving
- Tested transceiver chips using basic antennas
- Characterized the antenna to better understand its transmission
- Compared data transmission of different antennas through the human body (or a stand in material)

Outcomes and Deliverables

The ideal outcome of this project was to create a proof of concept system that can be used to wirelessly send data to another device. This stage of development can then be taken on by another group of students (hopefully more versed in RF technologies) and improve upon our project by, for example, making it fully biocompatible.

The Key Deliverables were:

1. A modular system that can be used to send data through flesh with (2) the midfield wave antenna
2. A midfield wave antenna designed to transmit through human flesh (or an equivalent stand in material)
3. Documentation that is thorough enough for another group of students to build upon our project without having to redo most of our work

Background

Implantable technology benefits from one-way communication with minimal errors. Data protocols, like SPI, use parity bit(s) in order to discover incorrect bits. On the other hand, protocols like this can still communicate corrupt data if either the parity bit is flipped, or multiple flipped bits cause an incorrect, but equal, parity bit; therefore more complex schemes are necessary. Our project initially focussed on our data protocol for the system, particularly forward error correction (FEC) protocols. FEC

protocols are used to prevent data loss at the packet level, either due to corruption or attenuation. By sending redundant and compressed data, packets that are lost or corrupted can be fixed on the receiving end, correcting errors as the data is received—instead of afterwards. In order to transmit to highly error-sensitive implanted technology, the transmitted signal must be sent using error-free or error-correcting protocols. While we gained some momentum implementing an FEC protocol found online, our attention moved onto the transceiving chips that were designed to implement error correction protocols already.

Our ideal project makes use of an FPGA (Field Programmable Gate Array) and a microcontroller instead of just a microcontroller because of the added flexibility. A microcontroller can be reprogrammed wirelessly, but it can only take advantage of the hardware and connections that are physically included at the time of assembly. Microcontrollers, typically, can only handle one task at a time unless additional processing cores or special hardware is added. On the other hand, an FPGA is a large array of gates that can be reprogrammed to change its logical behavior without physically changing any hardware—hence the phrase “Field Programmable” in the name. This means connections can be switched around and additional logical features can be added or removed after everything has been soldered to a board. It also allows for true asynchronous computing, where multiple logical processes can occur simultaneously, without having to queue through a central processor. This means that after our system has been integrated and implanted into a patient, the entire architecture can be changed to something completely different.

Development for implanted electronic communication has advanced over the past few years. At Stanford University, Dr. Ada Poon and others within the Electrical Engineering department have been improving and designing new methods of wireless power transmission. The antenna used in this design was created based on the 2014 antenna built under Dr. Poon.

Engineering Specifications

The overall goal of this project is to wirelessly transmit data through the human body. The project must utilize the Stanford midfield wave antenna in order to send a signal through the human body and process the received signal for external use, potentially for programming another device.

Spec #	Parameter Description	Requirements w/ units	Tolerance	Risk	Compliance
1	Error Rate	0 bits (after corrections)	Max	H	A, T, I
2	Transmission Rate	1 kByte/s	Min	L	A, T, I
3	Power	2.5 W	Max	M	A, T, I, S
4	Transmission Power	1 kW	Max	L	T, I
5	Transmission Distance	15 cm through Human Tissue	Max	L	I, S

Personas

Medical Research & Development Engineer

One persona would be a medical researcher with a Biomedical, Electrical Engineering, Computer Engineering, or Physics degree (B.S. or higher) that researches new and improved methods for monitoring and/or enhancing implantable technology. This researcher must be detail-oriented, have excellent testing criteria, and be motivated to constantly improve while remaining ethical and logical in their testing. The general work environment for this persona includes dynamic challenges that push the boundary on implanted devices.

Student Developer (Capstone / Senior Project)

Another persona would be a BMED/EE/CPE student looking to use our documentation as a means to refining or redesigning our finished project. Such student would require an adequate understanding of antennas, RF, and microcontrollers. For the purposes of a senior-level project, this persona would work in an educational environment where the sum of their education is applied with the support of a structured project development course.

Use Cases

Improving the Midfield Wave Antenna's Characteristics:

An EE student or students can probably redesign the midfield wave antenna to perform better. As it stands, the midfield wave antenna that we fabricated requires soldering on four coaxial cables with SMA connectors, then needs a power splitter to get signal to all four ports on the antenna. It also

exhibits lower than expected impedance and gain, which means that it can be outperformed by a single piece of wire acting as a quarter wave antenna. The antenna would benefit massively from being redesigned with 433 MHz in mind, with a single SMA connector on the PCB itself, and with more ideal antenna characteristics. As the system stands, it would be easy to swap out our current antenna, since everything is connected with SMA connectors.

Investigating the Biological Aspects:

A biomedical student or students can investigate the effects of this antenna on flesh and compare them to the effects of a “normal” antenna. It is well known that electromagnetic signals that penetrate the flesh can heat the flesh, killing cells in the process. What is not known is how this antenna interacts with this phenomenon. The midfield wave antenna may match the impedance of the body and heat the flesh less, or it may penetrate flesh better and heat the flesh more. This would be an important question to answer if the project wants to make it to human trials.

Integration into Other Devices:

A CPE or EE student or students can use our project to build a prototype implanted device by building a new receiving module. Since the Adafruit feather chips uses an off the shelf RFM69HCW 433 MHz chip for RF communications, by building a chip from the RFM69 family into the receiving side, our transmitter

would act as a simple SPI pipe to get data to the implant. This would allow for projects like a remotely programmable FPGA.

Design Development

Our top design concepts involved communicating through the midfield wave antenna via wifi, Bluetooth, Zigbee, and an SPI-like protocol at 400 MHz. Our final design uses Adafruit Feather M0 microcontrollers that have built in RFM69HCW 433 MHz chips; though it should work with other general purpose microcontrollers and transceiver chips.

Wifi and Bluetooth are the protocols that we are most familiar with, and were some of the first to come to mind, as they are several orders of magnitude faster than what we need, there are many well documented modules to connect to them, and they build in some data integrity checks and security measures. However, they require a lot of power (though Bluetooth Low Energy does aim to address this concern), and have a lot of overhead in their packets—especially since much of their data integrity involves resending lost packets, which may be a common occurrence in lossy environments (“Bluetooth”). Critically, Wifi and Bluetooth both operate on the 2.4 GHz and 5 GHz spectrums which may not work well with the midfield wave antenna—which operates best at 1.6 GHz.

Zigbee is a data protocol for communicating with sensors, and as such uses much less power (“Zigbee”). However, it uses similar data integrity systems as wifi and

bluetooth, which means that there is high overhead in lossy environments, and again, operates at 2.4 GHz, making it less compatible with our 1.6 GHz antenna.

Finally, implementing another data protocol at 400 MHz is the most complex to implement, but offers several key advantages over the other choices. Namely, that 400 MHz can be run through a frequency mixer to reach 1.6 GHz, the native frequency of our antenna. We can also employ very low-power transmitters since we don't need very much range; potentially dropping the power consumption significantly below the other protocols. Other advantages can be conferred by the data protocol we use. We explored implementing MedRadio, as it would take advantage of forward error correction to correct errors in transmission as they are received rather than having to rely on resending lost or damaged packets and checking checksums ("Medical Devices"). However, we were not able to find a good open-source implementation, and implementing it ourselves proved unnecessary since sending raw packets at 433 MHz worked better than expected.

Back to the hardware the final design uses. We chose the Adafruit Feather RF boards because they massively simplified our design and setup. We originally intended to use Analog Devices ADF7023 RF transceiver chips, as they contained several nice features like a built in microcontroller that could do basic forward error correction and AES encryption. They came as 5x5 mm surface mount chips that required advanced soldering equipment and custom PCBs, so we decided to use an off the shelf RF chip instead. We picked the RFM69 433 MHz, as there is extensive documentation and example projects for the chipset. However, we were not aware that the chips we

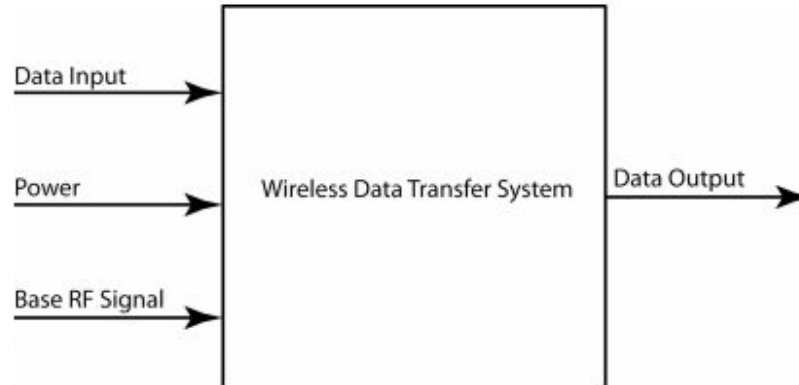
procured only operated at 3.3 V, which a normal 5 V Arduino would fry. The Adafruit Feather RF boards have extensive documentation, can handle the logic levels, provide enough power to the RF chip, and do so in a single board. We switched to these boards instead of procuring and trying to integrate logic voltage shifting chips and more powerful FTDI power delivery chips into the existing setup.

Final Design

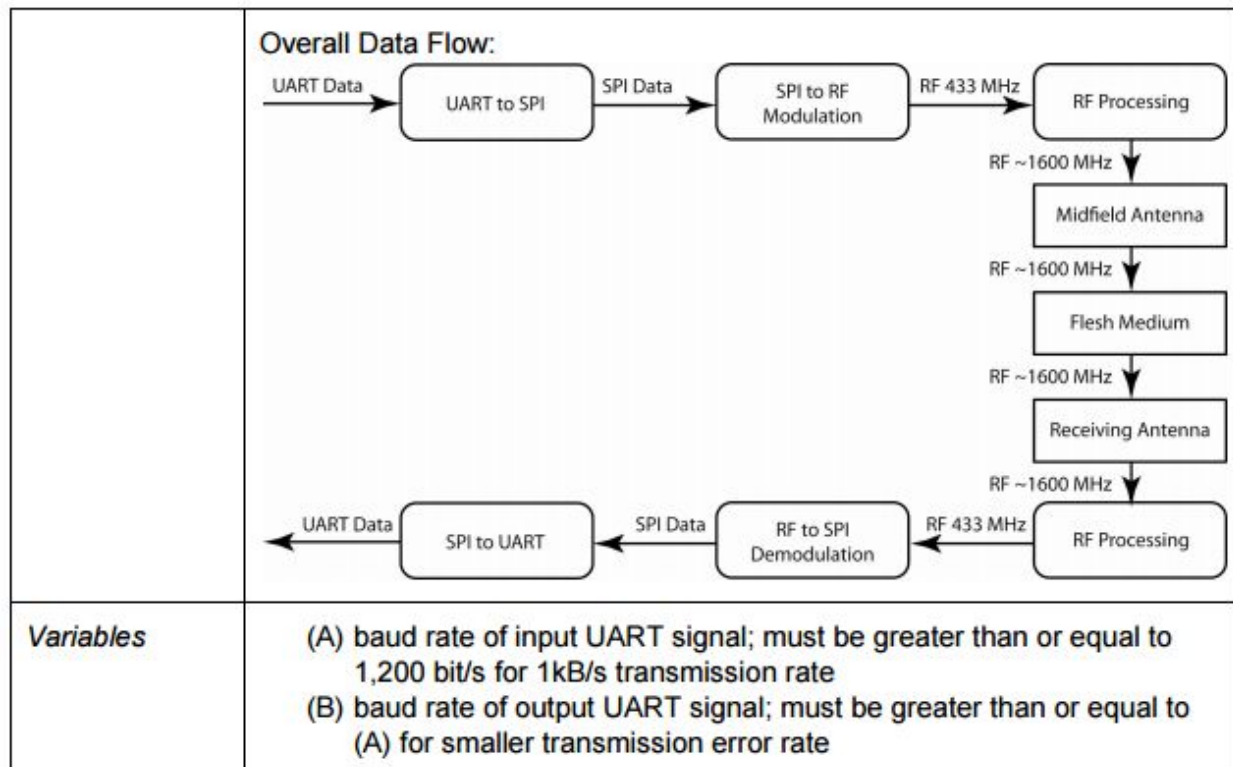
Due to its lower power usage and better compatibility with the antenna, we chose to use 400 MHz RF chips with a simple data protocol. We would like to use MedRadio with them, but the current lack of documentation and open implementation makes this difficult to do. The Adafruit Feather M0 RFM69HCW 433 MHz combine the role of microcontroller and RF chip into a single, simple board. The code to send and receive data also takes up a tiny amount of space on the Feather itself, so there are plenty of resources should a future project wish to expand on what has already been done. The RFM69 series chips have also proven reliable without needing to implement an additional data protocol on top of the packet based protocol it uses.

Functional Decomposition

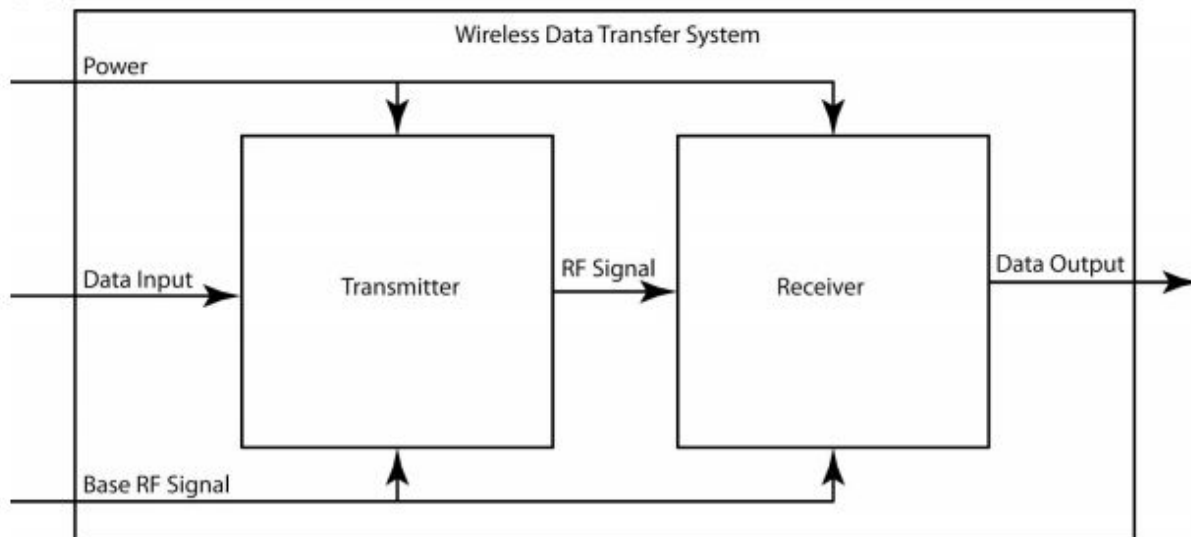
Level 0



<i>Module</i>	Wireless Data Transfer System
<i>Inputs</i>	<ul style="list-style-type: none">- Data Input: UART 8-bit data signal with baud rate of 1,200 bit/s minimum (A)- Power: 3.3VDC- Base RF Signal: 1.6GHz sinusoidal waveform
<i>Outputs</i>	<ul style="list-style-type: none">- Data Output: UART 8-bit data signal with (B) baud rate
<i>Functionality</i>	<p>Transfer digital data from one host to another using a midfield wave antenna to transmit through human flesh. The output data should be no greater than %10 different than the input data to ensure that this data can be corrected from errors and used to potentially program the receiving host.</p> <p>The overall data flow diagram can be seen below as information is inputted and outputted in UART format despite undergoing conversions into SPI, RF modulation, and RF attenuation to perform communication using the midfield antenna designed to transmit power through flesh.</p> <p>(See data flow diagram below)</p>



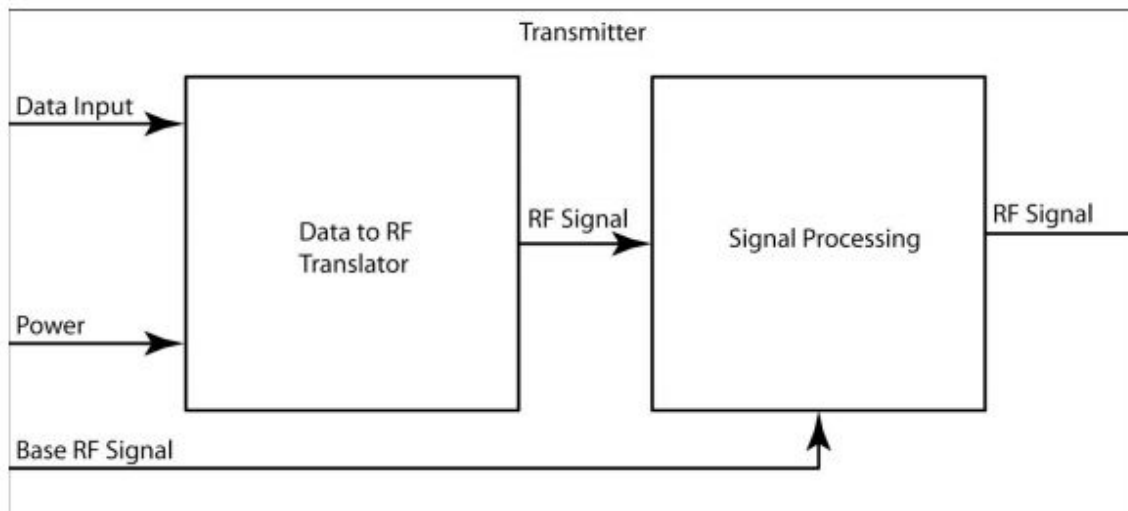
Level 1



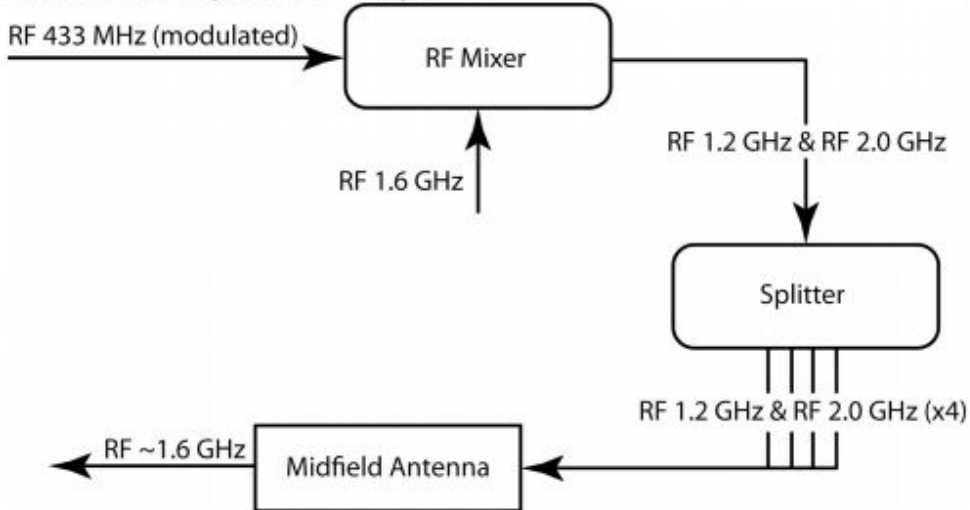
Module	Transmitter
Inputs	<ul style="list-style-type: none"> - Data Input: UART 8-bit data signal with baud rate of 1,200 bit/s minimum (A) - Power: 3.3VDC - Base RF Signal: 1.6GHz sinusoidal waveform
Outputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Functionality	Transmits modulated data signal from midfield antenna to the receiving antenna
Variables	<p>(A) baud rate of input UART signal; must be greater than or equal to 1,200 bit/s for 1kB/s transmission rate</p> <p>(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433MHz modulated data frequency</p>

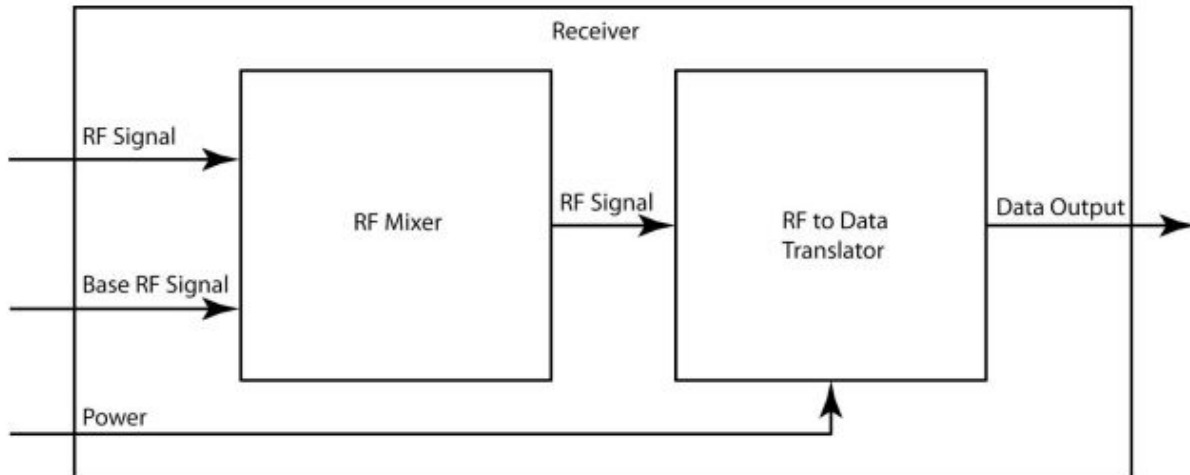
Module	Receiver
Inputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Outputs	<ul style="list-style-type: none"> - Data Output: UART 8-bit data signal with (B) baud rate
Functionality	Receives modulated data signal from midfield antenna and extracts encoded data for outputting
Variables	<p>(B) baud rate of output UART signal; must be greater than or equal to (A) for smaller transmission error rate</p> <p>(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433MHz modulated data frequency</p>

Level 2



Module	Transmitter > Data to RF Translator
Inputs	<ul style="list-style-type: none"> - Data Input: UART 8-bit data signal with baud rate of 1,200 bit/s minimum (A) - Power: 3.3V
Outputs	<ul style="list-style-type: none"> - RF Signal: Modulated 433MHz
Functionality	Modulates a 433MHz RF signal with data received
Variables	(A) baud rate of input UART signal; must be greater than or equal to 1,200 bit/s for 1kB/s transmission rate

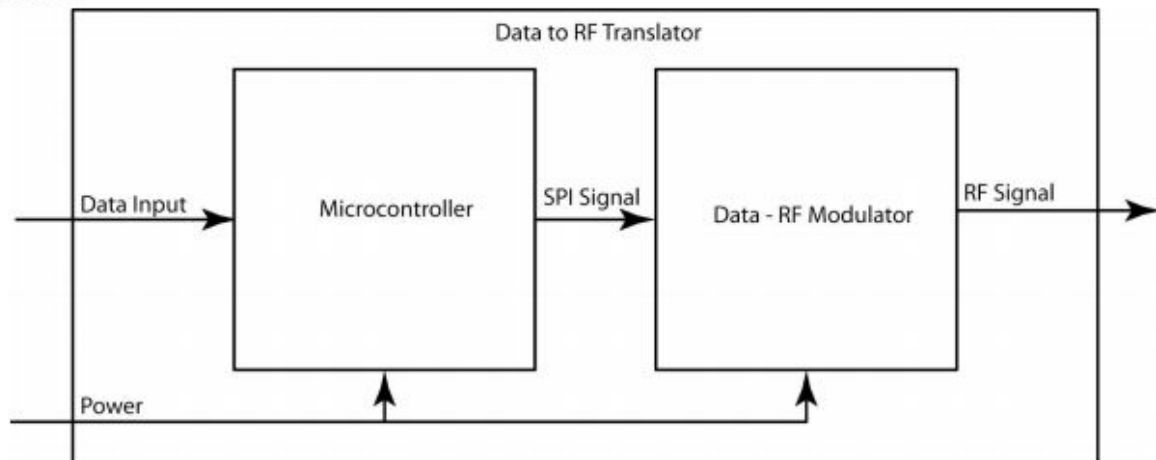
Module	Transmitter > Signal Processing
Inputs	<ul style="list-style-type: none"> - RF Signal: Modulated 433MHz RF signal - Base RF Signal: 1.6GHz RF signal
Outputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Functionality	<p>Prepares the RF signal to send modulated data across the flesh medium while using the midfield antenna.</p> <p>The process for preparing the signal after modulation requires the signal to be mixed with a 1.6GHz carrier frequency and split into four channels in order to interface with the midfield antenna.</p> <p>Data Flow for Signal Processing:</p>  <pre> graph TD A[RF 433 MHz (modulated)] --> B[RF Mixer] C[RF 1.6 GHz] --> B B --> D[RF 1.2 GHz & RF 2.0 GHz] D --> E[Splitter] E --> F[RF 1.2 GHz & RF 2.0 GHz (x4)] F --> G[Midfield Antenna] G --> H[RF ~1.6 GHz] </pre> <p>The diagram illustrates the data flow for signal processing. It starts with an input of 'RF 433 MHz (modulated)' entering an 'RF Mixer' block. A second input, 'RF 1.6 GHz', also enters the 'RF Mixer' from below. The output of the 'RF Mixer' is 'RF 1.2 GHz & RF 2.0 GHz', which then enters a 'Splitter' block. The 'Splitter' outputs four channels, labeled 'RF 1.2 GHz & RF 2.0 GHz (x4)', which are then sent to a 'Midfield Antenna' block. Finally, the 'Midfield Antenna' outputs 'RF ~1.6 GHz'.</p>
Variables	(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433MHz modulated data frequency



<i>Module</i>	Receiver > RF Mixer
<i>Inputs</i>	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal - Base RF Signal: 1.6GHz RF signal
<i>Outputs</i>	<ul style="list-style-type: none"> - RF Signal: Modulated 433MHz RF signal
<i>Functionality</i>	Separates modulated RF signal from carrier frequency for data extraction in translator
<i>Variables</i>	(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433MHz modulated data frequency

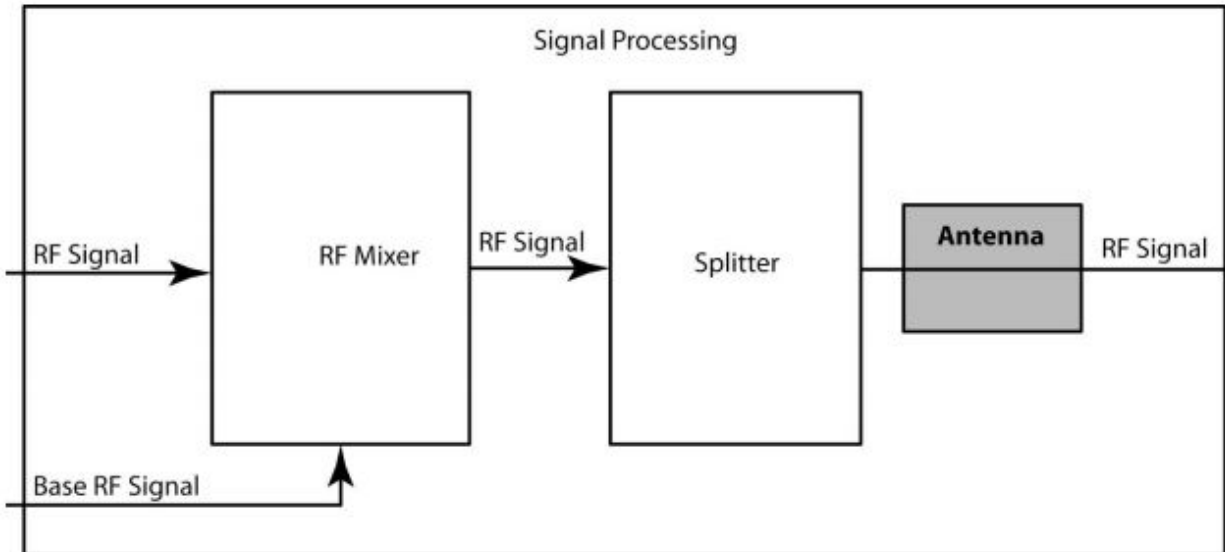
<i>Module</i>	Receiver > RF to Data Translator
<i>Inputs</i>	<ul style="list-style-type: none"> - RF Signal: Modulated 433 MHz RF signal - Power: 3.3V
<i>Outputs</i>	<ul style="list-style-type: none"> - Data Output: UART 8-bit data signal with (B) baud rate
<i>Functionality</i>	Translates RF signal into UART signal by means of extracting the data encoded in the RF signal via the RF demodulator module and a microcontroller
<i>Variables</i>	(B) baud rate of output UART signal; must be greater than or equal to (A) for smaller transmission error rate

Level 3



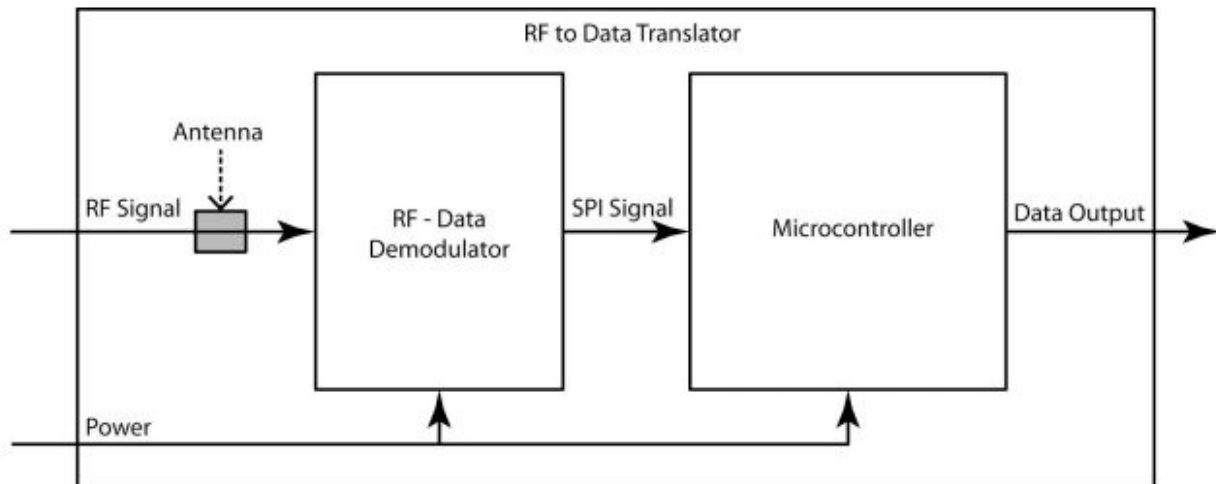
<i>Module</i>	Transmitter > Data to RF Translator > Microcontroller
<i>Inputs</i>	<ul style="list-style-type: none">- Data Input: UART 8-bit data signal with baud rate of 1,200 bit/s minimum (A)- Power: 3.3V DC
<i>Outputs</i>	<ul style="list-style-type: none">- SPI Signal: SPI signal with baud rate of (D)
<i>Functionality</i>	Converts the UART signal into an SPI signal for interfacing the Data-RF Modulator
<i>Variables</i>	(A) baud rate of input UART signal; must be greater than or equal to 1,200 bit/s for 1kB/s transmission rate (D) baud rate necessary to interface Data-RF Modulator; in our case 115200 bit/s

<i>Module</i>	Transmitter > Data to RF Translator > Data - RF Modulator
<i>Inputs</i>	<ul style="list-style-type: none">- SPI Signal: SPI signal with baud rate of (D)
<i>Outputs</i>	<ul style="list-style-type: none">- RF Signal: 433MHz Modulated RF Signal
<i>Functionality</i>	Encodes data received from SPI signal into 433MHz carrier frequency RF signal
<i>Variables</i>	(D) baud rate necessary to interface Data-RF Modulator; in our case 115200 bit/s

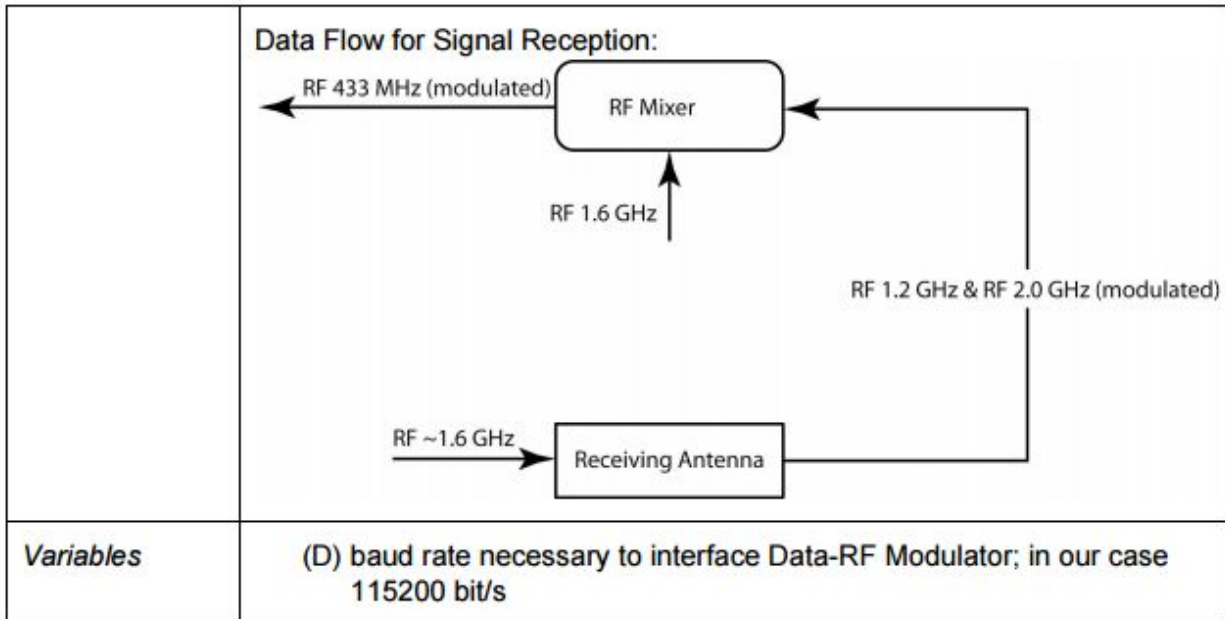


Module	Transmitter > Signal Processing > RF Mixer
Inputs	<ul style="list-style-type: none"> - RF Signal: Modulated 433MHz RF signal - Base RF Signal: 1.6GHz RF signal
Outputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Functionality	Mixes modulated RF signal with 1.6GHz carrier frequency for midfield antenna
Variables	(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433 MHz modulated data frequency

Module	Transmitter > Signal Processing > Splitter
Inputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Outputs	<ul style="list-style-type: none"> - RF Signal: Modulated (C) MHz RF signal
Functionality	Splits input RF signal into four ports to transmit with midfield antenna
Variables	(C) intermediate RF signal with 1.6GHz carrier frequency \pm 433 MHz modulated data frequency



<i>Module</i>	Receiver > RF to Data Translator > RF - Data Demodulator
<i>Inputs</i>	<ul style="list-style-type: none"> - RF Signal: Modulated 433MHz RF signal - Power: 3.3V
<i>Outputs</i>	<ul style="list-style-type: none"> - SPI Signal: SPI signal with baud rate of (D)
<i>Functionality</i>	<p>Extract data encoded in RF signal by mixing base RF signal to retrieve 433MHz signal</p> <p>The data flow diagram for receiving the RF signal from the midfield antenna is a simpler, reversed process of the data flow for transmitting. For preparing the received signal for demodulation, the signal must be received on the receiving antenna and mixed with the same carrier signal to lower the frequency back down to 433 MHz for demodulation.</p> <p>(See data flow diagram below)</p>



Module	Receiver > RF to Data Translator > Microcontroller
Inputs	<ul style="list-style-type: none"> - SPI Signal: SPI signal with baud rate of (D) - Power: 3.3V
Outputs	<ul style="list-style-type: none"> - Data Output: UART 8-bit data signal with (B) baud rate
Functionality	Converts SPI signal into UART signal for final data output
Variables	(B) baud rate of output UART signal; must be greater than or equal to (A) for smaller transmission error rate (D) baud rate necessary to interface Data-RF Modulator; in our case 115200 bit/s

Design Verification

FMEA

Process Function	Potential Failure Mode	Potential Effect(s) of Failure	Sev	Potential Cause(s)/ Mechanism(s) of Failure	Occur	Current Process Controls	Detec	RPN	Recommended Action(s)	Responsibility and Target Completion Date	Action Results				
											Actions Taken	Sev	Occ	Det	RPN
Data Transmission	Failure to Transmit	Nothing (is received)	1	Misaligned antenna, interference	3	Operator training and instructions	1	3	Operator training and instructions, interference isolation	AL and EM, March 7, 2017	Interference signal isolation	1	1	1	1
	Partial Transmission	RF receiver alerts receiving microprocessor, nothing happens	2	Misaligned antenna, interference	3	Operator training and instructions	1	6	Operator training and instructions, interference isolation	AL and EM, March 7, 2017	Interference signal isolation	2	2	1	4
	Corrupted Transmission	Hopefully, RF receiver corrects, discards if not possible	3	Interference	2	Operator training and instructions	2	12	Operator training and instructions	TBD					0
	Errant/Malicious Transmission	Bad data is transmitted	5	Interference, malicious attacker, too many being programmed too close together	2	Operator training and instructions, error correction	5	50	Encrypt communications, determine way to authenticate before programming	TBD					0

Design Verification Plan

CPE 450 DESIGN VERIFICATION PLAN AND REPORT													
Report Date		2/17		Sponsor		Dr. Smilkstein				Component/Assembly		REPORTING ENGINEER:	
TEST PLAN								TEST REPORT					
Item No	Specification [1]	Test Description [2]	Acceptance Criteria [3]	Test Responsibility [4]	Test Stage [5]	SAMPLES TESTED		TIMING		TEST RESULTS			NOTES
						Quantity	type	Start date	Finish date	Test Result [7]	Quantity Pass	Quantity Fail	
1	Error Rate	Transferred data must match up to original data as effectively as possible in order to utilize data correction and device programming, (i.e. a 1kB file received must match up with at least 900 bytes of the 1kB file sent)	< 10% difference	Adam	PV	10	ABC	2/22/2017	3/4/2017	IN PROGRESS			Occasional dropped packet
2	Transmission Rate	A 1 kB file (regardless of the contents) should be able to be transmitted in less than 1 second)	1 kB/s	Adam	DV	5	B	2/22/2017	3/4/2017	FAIL	~0.333 kB/s	-	Current programs transmit at about 333 Bytes/sec
3	Power consumption	During the testing, the transmission system (and receiving system) should not draw more than 2.5 W of power	2.5 W	Erik	PV	3	C	2/26/2017	3/6/2017	PASS	0.1 W		
4	Transmission Power	When powering the antenna, if the antenna transmits more than 1 kW of power, we risk heating and destroying flesh. Thus, the whole system should draw less than 1 kW of power (If we meet the power consumption criteria, we will meet this criteria)	1 kW	Adam	PV	3	BC	2/26/2017	3/6/2017	PASS	0.1 W		
5	Transmission Distance	The system should be able to transmit through at least 15 cm of skin, fat, and muscle from either a chicken, cow, or pig (depending on availability and cost)	15 cm through animal tissue	Erik	CV	2	AC	2/26/2017	3/6/2017	IN PROGRESS			Tested on 3 cm of tissue with success, but not 15 cm yet

Note: See Appendix for larger formats.

Verification Test Results

We conducted three major tests: error rate, antenna analysis, and data transmission. The error rate was tested using three sizes of files. The sizes were

deemed small (less than 1kB), medium (2kB), and large (200kB). These sizes were used as unit tests for our transfer system. For the most part, our error rate was negligible at the small tests, but began to drop single packets occasionally during the medium unit tests. The large unit tests took over 7 minutes long at lost about 1kB of data. While these tests were valuable, we determined that our results were not completely controlled and would benefit from retesting under stationary constraints for the antennas and in a noiseless environment (i.e. an anechoic chamber).

The antenna analysis proved to be more useful and intriguing than our error testing. When characterizing our antenna, we found that the electrical characteristics of the antenna was less than ideal. Dr. Arakaki, the consulting antenna professor, mentioned that an ideal antenna in our frequency range should have the S11 Reflection characteristics of 50 Ω impedance and -10 dB. Our antenna characteristics (by port) can be seen in the table below:

Midfield Wave Characteristics at 1.6 GHz				
Port	1	2	3	4
Reflection Impedance (Ω)	28.408	27.459	8.809	24.747

Note: Reflection Magnitude (dB) was not recorded

While the antenna we attempted to recreate appeared to be less than ideal, we continued to test the antenna using our system for transmitting data. Upon testing this antenna we found that not only could we transmit data using our midfield wave antenna, but we could also transmit through tissue without

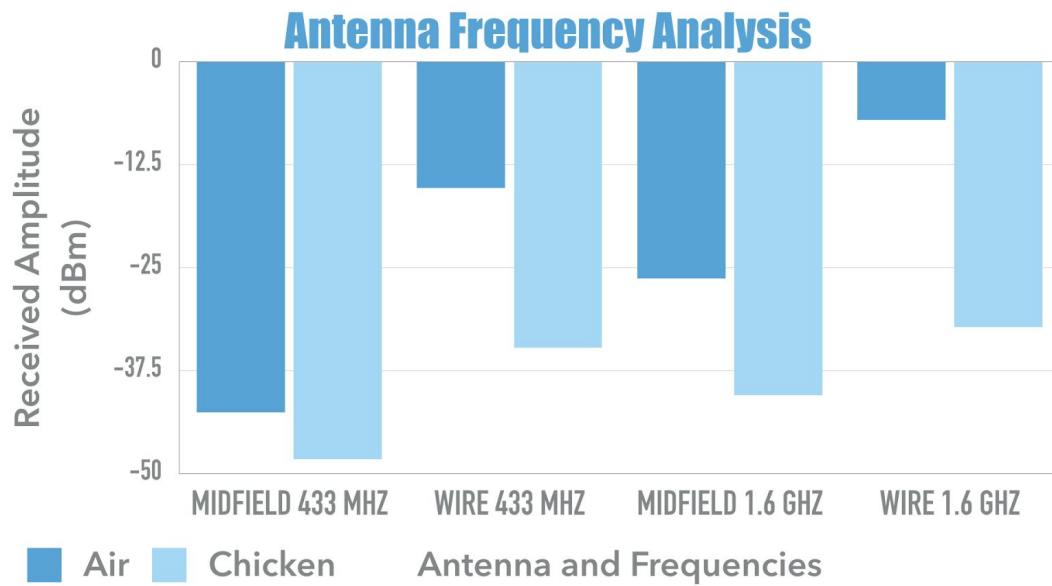
increasing the frequency to the expected 1.6 GHz it was designed for. The tests through tissue are still in progress but have been successfully tested to transmit through 3 cm of chicken breast completely surround thing the receiving antenna.

Further analysis was done after discovering that the 433 MHz quarter wavelength wire antenna could also transmit through the chicken breast. We characterized the antenna again at 433 MHz and compared the transmission between the midfield wave antenna and the wire antenna:

Midfield Wave Characteristics at 433 MHz

Port	1	2	3	4
Reflection Impedance (Ω)	28.267	21.003	16.336	23.197
Reflection Magnitude (dB)	-0.321	-0.302	-0.283	-0.319

The graph below shows the results our comparison between the midfield wave and wire antennas. In the graph, the two antennas were compared by the received power on the same loop antenna when transmitting across air and chicken at the two designed frequencies. It is important to note that the bar graph appears upside down due to the units where the negative values in dBm units equate to smaller power (mostly on the scale of μW).

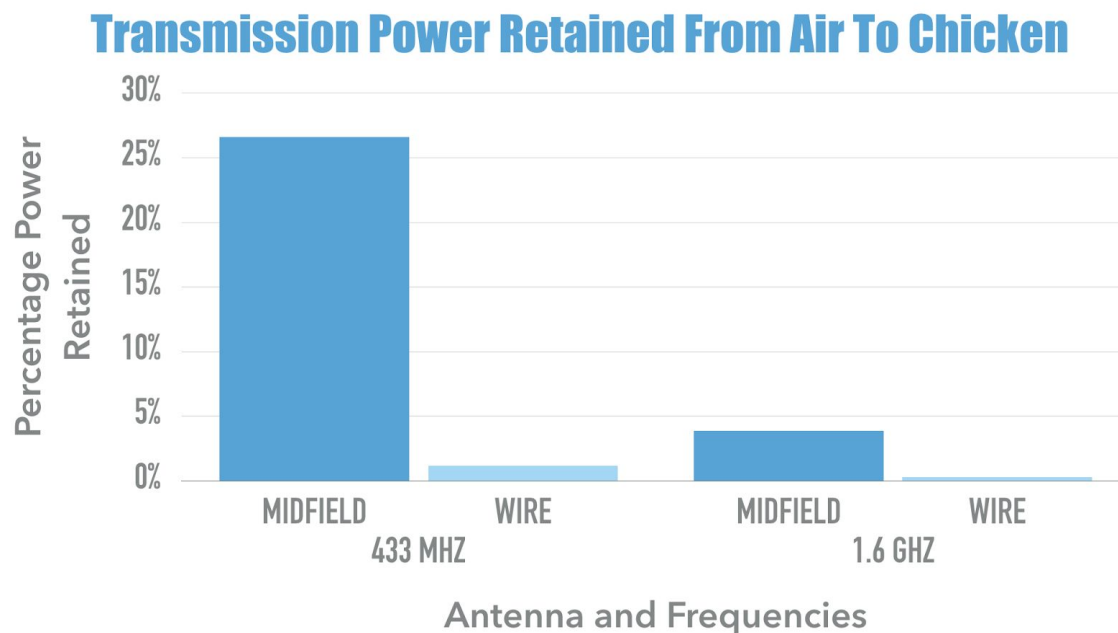


System Analysis

Overall, our system was a success, as we were able to transmit data through chicken. We handily met our requirements for power and transmission power, as our system is powered off of USB 2.0, and thus can not exceed 2.5 W. Our transmission rate was not quite as high as we had hoped; we suspect that it had something to do with the overhead of translating from UART to SPI, then back from SPI to UART, and believe that it should not be too difficult for future projects to improve on. For the most part, we are able to transmit without error. However, there were occasionally runs where we would seem to drop a packet of data. While we have not concretely identified the cause, we believe that it had something to do with either positioning of the antenna or the startup process. We were able to successfully transmit through 3 cm of chicken (on

both sides of the receiving antenna), but have not figured out the logistics of testing through 15 cm of chicken or other flesh substitutes.

Analysis of our frequency analysis also provides an interesting insight into one aspect where our antenna outperforms a normal antenna: retaining power through flesh. While not unprecedented, as this antenna is designed to retain power through flesh, the midfield wave antenna retains nearly 27% of its power at 433 MHz, and nearly 4% at 1.6 GHz, as shown in the graph below. For comparison, the wire antenna only manages 1% at 433 MHz, and 0.3% at 1.6 GHz. The drop in power transmitted at 1.6 GHz vs 433 MHz is expected, as lower frequencies are known to travel better through flesh.



This outcome sets up two directions of future research. The first is improving the antenna. While the midfield wave antenna retains power better than a simple wire antenna, in its current iteration, it is handily beaten by a

simple wire antenna in every test when it comes to raw signal strength.

Secondly, research should be done on whether better power transmission is a desirable quality. While it may be useful for powering embedded electronics, RF signals in the body will heat the flesh, killing cells that become too warm. It is possible that since the midfield antenna attempts to match the body's impedance, the amount of flesh heating that occurs is reduced. It is also plausible that the increased power of the RF signal heats the flesh even more, making this antenna potentially dangerous.

Management Plan

Our mission was to create a system that can be used to transmit data using the midfield wave antenna to another device implanted in a human body (or a similar substitute).

Since our team is so small, roles were be a bit more fluid:

Adam - Communications Officer, Financial Officer

Erik - Procurement Officer, Project Manager

References

"Bluetooth vs. Bluetooth Low Energy." Link Labs. N.p., 09 Oct. 2016. Web. 13 Dec. 2016.

"Medical Device Radiocommunications Service (MedRadio)." *Federal Communications Commission*. Federal Communications Commission, 14 June 2016. Web. 12 Dec. 2016.

Nishida, Danielle. "A Wearable 1.6GHz Non-Invasive Midfield Wave-Based Blood Glucose Sensor." *Digital Commons @ CalPoly*. CalPoly, Mar. 2015. Web. 12 Dec. 2016.

Stanford Engineer Invents Safe Way to Transfer Energy to Medical Chips in the Body. Perf. Dr. Ada S. Y. Poon and Dr. John S. Ho. *YouTube*. Stanford University, 16 May 2014. Web. 12 Dec. 2016.

"ZigBee 3.0: The Foundation for the Internet of Things Is Now Available!" The ZigBee Alliance. The Zigbee Alliance, n.d. Web. 13 Dec. 2016.

Appendices

A1: Detailed Personas

Medical Research & Development Engineer

This persona resembles a medical researcher with a Bioengineering, Electrical Engineering, or Computer Science degree (B.S. or higher) that works to research new and improved methods for monitoring and/or enhancing implantable technology. In general, this user works with implantable technology and is in charge of either considering new methods of interaction or programming and designing such systems to perform the interaction. This researcher must be detail -oriented, have excellent testing criteria, and be motivated to constantly improve while remaining ethical and logical in their testing.

Generally, they work in a dynamic testing environment where the day to day challenges are constantly changing. This researcher must also strive to solve modern problems involving implantable electronics and technology; especially using wireless signals as the method of interaction. Robust testing and alertness to error will keep this user from ethical and physical harm of their subjects. In order to propagate their research further into human testing, this researcher needs to remain vigilant and sharp towards any maleficent behavior.

Student Developer (Capstone / Senior Project)

Another persona that would use our project would be another student developer. Realistically, our project is more of a proof of concept than a fully realized project ready for human testing. The student developer, potentially as part of a Capstone class or Senior Project would focus on another aspect of the project, like miniaturization, better power use, or protecting (and protecting against) the human body.

Said student would likely be a BMED, EE, or CPE student looking to use our documentation as a stepping stone to a more complete project. They would have access to well-equipped labs and teachers, but may not have direct access to us. As such, good documentation is critical for their understanding. Work on the project may be sporadic (like the 2 hours of class every other day), or it may be in large blocks (like pulling all-nighters during finals week), so documentation must be easy to use, like a wiki or searchable, indexed document.

The students would likely need an intermediate understanding of circuits and antennas, as well as computer skills. Depending on the aspect of the project the student is focusing on, they may need a better understanding of antennas or microcontrollers; they may need a better understanding of how the device would interact with a body (human, or more likely, animal).

A2: Arduino Sketch Source Code

This code is intended to run on Adafruit Feather M0 RFM69HCW 433 MHz boards. It requires the libraries for the Adafruit Feather M0 boards as well as the LowPowerLab RFM69 and SPIFlash libraries. See this link for more detailed instructions:

<https://learn.adafruit.com/adafruit-feather-m0-radio-with-rfm69-packet-radio/setup>

The full set of code can be found in this Github Repository:

<https://github.com/letmeadam/Midfield-Data-Transfer-System>

Transmitting code:

```
/* RFM69 library and code by Felix Rusu - felix@lowpowerlab.com
// Get libraries at: https://github.com/LowPowerLab/
// Make sure you adjust the settings in the configuration section below !!!
// *****
// Copyright Felix Rusu, LowPowerLab.com
// Library and code by Felix Rusu - felix@lowpowerlab.com
// *****
// License
// *****
// This program is free software; you can redistribute it
// and/or modify it under the terms of the GNU General
// Public License as published by the Free Software
// Foundation; either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will
// be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public
// License for more details.
//
// You should have received a copy of the GNU General
```

```

// Public License along with this program.
// If not, see <<http://www.gnu.org/licenses>></http:>.
//
// Licence can be viewed at
// http://www.gnu.org/licenses/gpl-3.0.txt
//
// Please maintain this license information along with authorship
// and copyright notices in any redistribution of this code
// *****/

#include <RFM69.h>    //get it here: https://www.github.com/lowpowerlab/rfm69
#include <SPI.h>

//*****
**
// ***** IMPORTANT SETTINGS - YOU MUST CHANGE/ONFIGURE TO FIT YOUR HARDWARE
*****
//*****
**
#define NETWORKID      100 // The same on all nodes that talk to each other
#define NODEID         2   // The unique identifier of this node
#define RECEIVER       1   // The recipient of packets

//Match frequency to the hardware version of the radio on your Feather
#define FREQUENCY      RF69_433MHZ
//#define FREQUENCY      RF69_868MHZ
//#define FREQUENCY      RF69_915MHZ
#define ENCRYPTKEY      "sampleEncryptKey" //exactly the same 16 characters/bytes on all nodes!
#define IS_RFM69HCW    true // set to 'true' if you are using an RFM69HCW module

//*****
**
#define SERIAL_BAUD    115200

/* for Feather M0 Radio */
#define RFM69_CS       8
#define RFM69_IRQ      3
#define RFM69_IRQN     3 // Pin 3 is IRQ 3!
#define RFM69_RST      4

#define LED            13 // onboard blinky
//#define LED           0 //use 0 on ESP8266
#define BUFFER_SIZE    61
#define TIMEOUT_SET    20

int16_t packetnum = 0; // packet counter, we increment per xmission
char buffer[BUFFER_SIZE];

```

```

int timeout = TIMEOUT_SET;

RFM69 radio = RFM69(RFM69_CS, RFM69_IRQ, IS_RFM69HCW, RFM69_IRQN);

void setup() {
  while (!Serial); // wait until serial console is open, remove if not tethered
  //to computer. Delete this line on ESP8266
  Serial.begin(SERIAL_BAUD);

  Serial.println("Feather RFM69HCW Transmitter");

  // Hard Reset the RFM module
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, HIGH);
  delay(100);
  digitalWrite(RFM69_RST, LOW);
  delay(100);

  // Initialize radio
  radio.initialize(FREQUENCY, NODEID, NETWORKID);
  if (IS_RFM69HCW) {
    radio.setHighPower(); // Only for RFM69HCW & HW!
  }
  radio.setPowerLevel(31); // power output ranges from 0 (5dBm) to 31 (20dBm)

  radio.encrypt(ENCRYPTKEY);

  pinMode(LED, OUTPUT);
  Serial.print("\nTransmitting at ");
  Serial.print(FREQUENCY==RF69_433MHZ ? 433 : FREQUENCY==RF69_868MHZ ? 868 : 915);
  Serial.println(" MHz");
}

void loop() {
  delay(100); // Wait 0.1 second between transmits, could also 'sleep' here!

  int read_amt = 0;
  if (Serial.available() > 0) {
    read_amt = Serial.readBytes(buffer, BUFFER_SIZE);

    radio.send(RECEIVER, buffer, read_amt);
    Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
  }
}

```

Receiving code:

```
/* RFM69 library and code by Felix Rusu - felix@lowpowerlab.com
// Get libraries at: https://github.com/LowPowerLab/
// Make sure you adjust the settings in the configuration section below !!!
// *****
// Copyright Felix Rusu, LowPowerLab.com
// Library and code by Felix Rusu - felix@lowpowerlab.com
// *****
// License
// *****
// This program is free software; you can redistribute it
// and/or modify it under the terms of the GNU General
// Public License as published by the Free Software
// Foundation; either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will
// be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public
// License for more details.
//
// You should have received a copy of the GNU General
// Public License along with this program.
// If not, see <http://www.gnu.org/licenses></http:>.
//
// Licence can be viewed at
// http://www.gnu.org/licenses/gpl-3.0.txt
//
// Please maintain this license information along with authorship
// and copyright notices in any redistribution of this code
// *****/

#include <RFM69.h>    //get it here: https://www.github.com/lowpowerlab/rfm69
#include <SPI.h>

//*****
**
// ***** IMPORTANT SETTINGS - YOU MUST CHANGE/ONFIGURE TO FIT YOUR HARDWARE
*****
//*****
**
#define NETWORKID    100 //the same on all nodes that talk to each other
#define NODEID      1

//Match frequency to the hardware version of the radio on your Feather
```

```

#define FREQUENCY      RF69_433MHZ
//#define FREQUENCY    RF69_868MHZ
//#define FREQUENCY    RF69_915MHZ
#define ENCRYPTKEY      "sampleEncryptKey" //exactly the same 16 characters/bytes on all
nodes!
#define IS_RFM69HCW    true // set to 'true' if you are using an RFM69HCW module

//*****
**

#define SERIAL_BAUD    115200

/* for Feather M0 */
#define RFM69_CS        8
#define RFM69_IRQ       3
#define RFM69_IRQN      3 // Pin 3 is IRQ 3!
#define RFM69_RST       4

#define LED             13 // onboard blinky
//#define LED           0 //use 0 on ESP8266
#define BUFFER_SIZE    1024 //1kByte
#define TIMEOUT_SET    60;

RFM69 radio = RFM69(RFM69_CS, RFM69_IRQ, IS_RFM69HCW, RFM69_IRQN);

void setup() {
    while (!Serial); // wait until serial console is open, remove if not tethered to computer.
    Delete this line on ESP8266
    Serial.begin(SERIAL_BAUD);

    Serial.println("Feather RFM69HCW Receiver");

    // Hard Reset the RFM module
    pinMode(RFM69_RST, OUTPUT);
    digitalWrite(RFM69_RST, HIGH);
    delay(100);
    digitalWrite(RFM69_RST, LOW);
    delay(100);

    // Initialize radio
    radio.initialize(FREQUENCY, NODEID, NETWORKID);
    if (IS_RFM69HCW) {
        radio.setHighPower(); // Only for RFM69HCW & HW!
    }
    radio.setPowerLevel(31); // power output ranges from 0 (5dBm) to 31 (20dBm)

    radio.encrypt(ENCRYPTKEY);

```

```

pinMode(LED, OUTPUT);

Serial.print("\nListening at ");
Serial.print(FREQUENCY==RF69_433MHZ ? 433 : FREQUENCY==RF69_868MHZ ? 868 : 915);
Serial.println(" MHz");

}

void loop() {
    //check if something was received (could be an interrupt from the radio)
    if (radio.receiveDone())
    {

        for (int i = 0; i < radio.DATALEN; i++) {
            Serial.print(((char*)radio.DATA)[i]);
            if (((char*)radio.DATA)[i] == '\n')
                Serial.print('\r');
        }
        Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
    }
    Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
}

```

A3: UART File Transmission Source Code

README.md

arduino-serial -- C code to talk to Arduino
=====

Original URL: <http://todbot.com/blog/2006/12/06/arduino-serial-c-code-to-talk-to-arduino/>

Post about changes: <http://todbot.com/blog/2013/04/29/arduino-serial-updated/>

Usage

<pre>

laptop% ./arduino-serial

Usage: arduino-serial -b <bps> -p <serialport> [OPTIONS]

Options:

| | |
|-----------------------|---|
| -h, --help | Print this help message |
| -b, --baud=baudrate | Baudrate (bps) of Arduino (default 9600) |
| -p, --port=serialport | Serial port Arduino is connected to |
| -s, --send=string | Send string to Arduino |
| -S, --sendline=string | Send string with newline to Arduino |
| -i --stdininput | Use standard input |
| -y, --byte | ! Receive single byte from Arduino & print it out |
| -r, --receive | ! Receive string from Arduino & print it out |
| -n --num=num | Send a number as a single byte |
| -f --input=ifile | ! Send file as input |
| -v --output=ofile | ! Save output to file |
| -F --flush | Flush serial port buffers for fresh reading |
| -d --delay=millis | Delay for specified milliseconds |
| -e --eolchar=char | Specify EOL char for reads (default '\n') |
| -t --timeout=millis | Timeout for reads in millisecs (default 5000) |
| -q --quiet | Don't print out as much info |

Note: Order is important. Set '-b' baudrate before opening port '-p'.

Used to make series of actions: '-d 2000 -s hello -d 100 -r'

means 'wait 2secs, send 'hello', wait 100msec, get reply'

</pre>

Downloads

Midfield-Data-Transfer-System Repo:

- <https://github.com/letmeadam/Midfield-Data-Transfer-System.git>

For convenience, here's some pre-built versions of arduino-serial.

They may not be updated regularly, so compile it yourself if you can.

Click the "view raw" to get the actual zip file.

- <https://github.com/todbot/arduino-serial/blob/master/arduino-serial-macosx.zip>
- <https://github.com/todbot/arduino-serial/blob/master/arduino-serial-linux.zip>

Compilation

arduino-serial should compile on any POSIX-compatible system.

Tested on Mac OS X, Ubuntu Linux, Raspian Linux, Beaglebone Linux

(Original repo: <https://github.com/todbot/arduino-serial.git>)

To build, just check it out, make, and run it like:

```
<pre>
% git clone https://github.com/letmeadam/Midfield-Data-Transfer-System.git
% cd arduino-serial
% make
% ./arduino-serial
</pre>
```

For more details on the build process, see the Makefile.

Arduino-serial.c

```
/*
 * arduino-serial
 * -----
 *
 * A simple command-line example program showing how a computer can
 * communicate with an Arduino board. Works on any POSIX system (Mac/Unix/PC)
 *
 *
 * Compile with something like:
 *   gcc -o arduino-serial arduino-serial-lib.c arduino-serial.c
 * or use the included Makefile
 *
 * Mac: make sure you have Xcode installed
 * Windows: try MinGW to get GCC
 *
 *
 * Originally created 5 December 2006
 * 2006-2013, Tod E. Kurt, http://todbot.com/blog/
 *
 *
 * Updated 8 December 2006:
 * Justin McBride discovered B14400 & B28800 aren't in Linux's termios.h.
 * I've included his patch, but commented out for now. One really needs a
 * real make system when doing cross-platform C and I wanted to avoid that
 * for this little program. Those baudrates aren't used much anyway. :)
 *
 * Updated 26 December 2007:
 * Added ability to specify a delay (so you can wait for Arduino Diecimila)
 * Added ability to send a binary byte number
 *
 * Update 31 August 2008:
 * Added patch to clean up odd baudrates from Andy at hexapodia.org
 *
 * Update 6 April 2012:
 * Split into a library and app parts, put on github
 *
 * Update 20 Apr 2013:
 * Small updates to deal with flushing and read backs
 * Fixed re-opens
 * Added --flush option
 * Added --sendline option
 * Added --eolchar option
 * Added --timeout option
 * Added -q/-quiet option
 *
```

```

*/

/* Update
*
*/

#include <stdio.h>    // Standard input/output definitions
#include <stdlib.h>
#include <string.h>    // String function definitions
#include <termios.h>    // for tcdrain() call in new File option //AAL
#include <time.h>      // for elapsed time measuement //AAL
#include <unistd.h>    // for usleep()
#include <fcntl.h>
#include <getopt.h>

#include "arduino-serial-lib.h"

//
void usage(void)
{
    printf("Usage: arduino-serial -b <bps> -p <serialport> [OPTIONS]\n"
        "\n"
        "Options:\n"
        "  -h, --help            Print this help message\n"
        "  -b, --baud=baudrate   Baudrate (bps) of Arduino (default 9600)\n"
        "  -p, --port=serialport Serial port Arduino is connected to\n"
        "  -s, --send=string      Send string to Arduino\n"
        "  -S, --sendline=string  Send string with newline to Arduino\n"
        "  -i --stdin            Use standard input\n"
        "  -y, --byte            Receive single byte from Arduino & print it out\n"
        "  -r, --receive         Receive string from Arduino & print it out\n"
        "  -n --num=num          Send a number as a single byte\n"
        "  -f --input=ifile      Send file as input\n"
        "  -v --output=ofile     Save output to file\n"
        "  -F --flush            Flush serial port buffers for fresh reading\n"
        "  -d --delay=millis     Delay for specified milliseconds\n"
        "  -e --eolchar=char     Specify EOL char for reads (default '\\n')\n"
        "  -t --timeout=millis   Timeout for reads in millisecs (default 5000)\n"
        "  -q --quiet            Don't print out as much info\n"
        "\n"
        "Note: Order is important. Set '-b' baudrate before opening port'-p'. \n"
        "      Used to make series of actions: '-d 2000 -s hello -d 100 -r' \n"
        "      means 'wait 2secs, send 'hello', wait 100msec, get reply'\n"
        "\n");
    exit(EXIT_SUCCESS);
}

```

```

//
void error(char* msg)
{
    fprintf(stderr, "%s\n",msg);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    time_t start, end;
    const int buf_max = 256;

    int fd = -1;
    char serialport[buf_max];
    int baudrate = 9600; // default
    char quiet=0;
    char eolchar = '\n';
    int timeout = 5000;
    char buf[buf_max];
    int rc,n;
    uint8_t b;

    if (argc==1) {
        usage();
    }

    /* parse options */
    int option_index = 0, opt;
    static struct option loptions[] = {
        {"help",          no_argument,      0, 'h'},
        {"port",          required_argument, 0, 'p'},
        {"baud",          required_argument, 0, 'b'},
        {"send",          required_argument, 0, 's'},
        {"sendline",      required_argument, 0, 'S'},
        {"stdinput",      no_argument,       0, 'i'},
        {"byte",          no_argument,       0, 'y'},
        {"receive",       no_argument,       0, 'r'},
        {"flush",         no_argument,       0, 'F'},
        {"num",           required_argument, 0, 'n'},
        {"ofile",         required_argument, 0, 'v'},
        {"ifile",         required_argument, 0, 'f'},
        {"delay",         required_argument, 0, 'd'},
        {"eolchar",       required_argument, 0, 'e'},
        {"timeout",       required_argument, 0, 't'},
        {"quiet",         no_argument,       0, 'q'},
        {NULL,            0,                0, 0}
    };
};

```

```

while(1) {
    opt = getopt_long (argc, argv, "hp:b:s:S:iryFn:f:v:d:qe:t:",
                      loptions, &option_index);
    if (opt== -1) break;
    switch (opt) {
    case '0': break;
    case 'q':
        quiet = 1;
        break;
    case 'e':
        eolchar = optarg[0];
        if(!quiet) printf("eolchar set to '%c'\n",eolchar);
        break;
    case 't':
        timeout = strtol(optarg,NULL,10);
        if( !quiet ) printf("timeout set to %d millisecs\n",timeout);
        break;
    case 'd':
        n = strtol(optarg,NULL,10);
        if( !quiet ) printf("sleep %d millisecs\n",n);
        usleep(n * 1000 ); // sleep milliseconds
        break;
    case 'h':
        usage();
        break;
    case 'b':
        baudrate = strtol(optarg,NULL,10);
        break;
    case 'p':
        if( fd!= -1 ) {
            serialport_close(fd);
            if(!quiet) printf("closed port %s\n",serialport);
        }
        strcpy(serialport,optarg);
        fd = serialport_init(optarg, baudrate);
        if( fd== -1 ) error("couldn't open port");
        if(!quiet) printf("opened port %s\n",serialport);
        serialport_flush(fd);
        break;
    case 'n':
        if( fd == -1 ) error("serial port not opened");
        n = strtol(optarg, NULL, 10); // convert string to number
        rc = serialport_writebyte(fd, (uint8_t)n);
        if(rc== -1) error("error writing");
        break;
    case 'f':
        start = time(NULL);
        if( fd == -1 ) error("serial port not opened");

```

```

// n = strtol(optarg, NULL, 10); // convert string to number
FILE * ifile = fopen(optarg, "r");
if (!ifile) {
    perror("error opening input file");
    break;
}
if(!quiet) printf("opened file \"%s\"\n", optarg);

// uint8_t b;
short a = 0;
// uint8_t c[2];
serialport_flush(fd);
do {
    rc = serialport_read_byte_file(ifile, &b, 10000);
    printf("%c", b);
    // printf("rc1 %d\n", rc);
    if(rc < 0) {
        printf("input loop broke unexpectedly %d\n", rc);
        break;
    }
    else if (rc == 2) {
        // rc = serialport_writebyte(fd, b);
        printf("end of file reached\n");
        serialport_flush(fd);
        break;
    }
    rc = serialport_writebyte(fd, b);
    // serialport_flush(fd);
    // printf(">>0x%02x%02x (%c) (%c)\n", c[1], c[0], c[1], c[0]);

    // if (a % 2) {
    //     // c[0] = b;
    //     // printf(">>0x%02x%02x (%c) (%c)\n", c[1], c[0], c[1], c[0]);
    // }
    // else {
    //     // c[1] = b;
    // }
    if ((a % 60) == 0) {
        // serialport_flush(fd);
        tcdrain(fd);
        tcdrain(1); //drain STDOUT

        // usleep(500);
    }
    a++;
} while (rc == 0);
printf("\n");

```

```

        end = time(NULL);
        if(!quiet)
            printf("completed file read/input (%d bytes) (%.2f minutes OR %.2f
seconds)\n",
                a, difftime(end, start) / 60.0f, difftime(end, start));

        fclose(ifile);

        break;
    case 'v':
        if( fd == -1 ) error("serial port not opened");
        // n = strtol(optarg, NULL, 10); // convert string to number
        int ofile = open(optarg, O_WRONLY | O_CREAT | O_TRUNC, 0666);
        if (ofile < 0) {
            perror("error opening output file");
            break;
        }
        if(!quiet) printf("opened file \"%s\"\n", optarg);
        if(!quiet) printf("\twarning: 5 seconds to send file!\n");

        rc = serialport_read_byte(fd, &b, 5000);
        if(rc <= 0) {
            printf("error: no input found.\n");
            close(ofile);
            break;
        }
        else {
            printf("found input.\n");
        }
        rc = serialport_writebyte(ofile, (uint8_t)b);
        tcdrain(ofile);

        a = 1;
        // c[0] = c[1] = b;
        while (rc <= 0) {
            rc = serialport_read_byte(fd, &b, 5000);
            if(rc <= 0) {
                tcdrain(ofile);
                break;
            }
            rc = serialport_writebyte(ofile, (uint8_t)b);
            printf("%c", b);
            if ((a % 60) == 0) {
                tcdrain(ofile);
                tcdrain(1); //drain STDOUT
                // usleep(1000);
            }
            // if (a % 2) {

```

```

        // c[0] = b;
        // printf("<<0x%02x%02x (%c) (%c)\n", c[1], c[0], c[1], c[0]);
    // }
    // else {
        // c[1] = b;
    // }
    a++;
}
if(!quiet) printf("completed file save\n");

close(ofile);
break;
case 'S':
case 's':
    if( fd == -1 ) error("serial port not opened");
    sprintf(buf, (opt=='S' ? "%s\n" : "%s"), optarg);

    if( !quiet ) printf("send string:%s\n", buf);
    rc = serialport_write(fd, buf);
    if(rc==-1) error("error writing");
    break;
case 'i':
    rc=-1;
    if( fd == -1) error("serial port not opened");
    while(fgets(buf, buf_max, stdin)) {
        if( !quiet ) printf("send string:%s\n", buf);
        rc = serialport_write(fd, buf);
    }
    if(rc==-1) error("error writing");
    break;
case 'y':
    if( fd == -1 ) error("serial port not opened");
    memset(buf,0,buf_max); //
    serialport_read_byte(fd, (uint8_t*) buf, timeout);
    if( !quiet ) printf("read byte:");
    printf("0x%02x\n", *((uint8_t*) buf));
    break;
case 'r':
    if( fd == -1 ) error("serial port not opened");
    memset(buf,0,buf_max); //
    serialport_read_until(fd, buf, eolchar, buf_max, timeout);
    if( !quiet ) printf("read string:");
    printf("%s\n", buf);
    break;
case 'F':
    if( fd == -1 ) error("serial port not opened");
    if( !quiet ) printf("flushing receive buffer\n");
    serialport_flush(fd);

```



```
        break;
    }
}

exit(EXIT_SUCCESS);
} // end main
```

Arduino-serial-lib.h

```
//
// arduino-serial-lib -- simple library for reading/writing serial ports
//
// 2006-2013, Tod E. Kurt, http://todbot.com/blog/
//

#ifndef __ARDUINO_SERIAL_LIB_H__
#define __ARDUINO_SERIAL_LIB_H__

#include <stdint.h>    // Standard types
#include <stdio.h>

int serialport_init(const char* serialport, int baud);
int serialport_close(int fd);
int serialport_writebyte( int fd, uint8_t b);
int serialport_write(int fd, const char* str);
int serialport_read_until(int fd, char* buf, char until, int buf_max,int timeout);
int serialport_read_byte(int fd, uint8_t *byte, int timeout);
int serialport_read_byte_file(FILE *f, uint8_t *byte, int timeout);
int serialport_flush(int fd);

#endif
```

Arduino-serial-lib.c

```
//
// arduino-serial-lib -- simple library for reading/writing serial ports
//
// 2006-2013, Tod E. Kurt, http://todbot.com/blog/
//

#include "arduino-serial-lib.h"

#include <stdio.h>    // Standard input/output definitions
#include <unistd.h>    // UNIX standard function definitions
#include <fcntl.h>    // File control definitions
#include <errno.h>    // Error number definitions
#include <termios.h>  // POSIX terminal control definitions
#include <string.h>    // String function definitions
#include <sys/ioctl.h>

// uncomment this to debug reads
// #define SERIALPORTDEBUG

// takes the string name of the serial port (e.g. "/dev/tty.usbserial","COM1")
// and a baud rate (bps) and connects to that port at that speed and 8N1.
// opens the port in fully raw mode so you can send binary data.
// returns valid fd, or -1 on error
int serialport_init(const char* serialport, int baud)
{
    struct termios toptions;
    int fd;

    //fd = open(serialport, O_RDWR | O_NOCTTY | O_NDELAY);
    fd = open(serialport, O_RDWR | O_NONBLOCK );

    if (fd == -1) {
        perror("serialport_init: Unable to open port ");
        return -1;
    }

    //int iflags = TIOCM_DTR;
    //ioctl(fd, TIOCMBS, &iflags);    // turn on DTR
    //ioctl(fd, TIOCMBS, &iflags);    // turn off DTR

    if (tcgetattr(fd, &toptions) < 0) {
        perror("serialport_init: Couldn't get term attributes");
        return -1;
    }
    speed_t brate = baud; // let you override switch below if needed
```

```

        switch(baud) {
        case 4800:  brate=B4800;   break;
        case 9600:  brate=B9600;   break;
#ifdef B14400
        case 14400: brate=B14400;  break;
#endif
        case 19200: brate=B19200;  break;
#ifdef B28800
        case 28800: brate=B28800;  break;
#endif
        case 38400: brate=B38400;  break;
        case 57600: brate=B57600;  break;
        case 115200: brate=B115200; break;
        }
        cfsetispeed(&toptions, brate);
        cfsetospeed(&toptions, brate);

        // 8N1
        toptions.c_cflag &= ~PARENB;
        toptions.c_cflag &= ~CSTOPB;
        toptions.c_cflag &= ~CSIZE;
        toptions.c_cflag |= CS8;
        // no flow control
        toptions.c_cflag &= ~CRTSCTS;

        //toptions.c_cflag &= ~HUPCL; // disable hang-up-on-close to avoid reset

        toptions.c_cflag |= CREAD | CLOCAL; // turn on READ & ignore ctrl lines
        toptions.c_iflag &= ~(IXON | IXOFF | IXANY); // turn off s/w flow ctrl

        toptions.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // make raw
        toptions.c_oflag &= ~OPOST; // make raw

        // see: http://unixwiz.net/techtips/termios-vmin-vtime.html
        toptions.c_cc[VMIN] = 0;
        toptions.c_cc[VTIME] = 0;
        //toptions.c_cc[VTIME] = 20;

        tcsetattr(fd, TCSANOW, &toptions);
        if( tcsetattr(fd, TCSAFLUSH, &toptions) < 0) {
            perror("init_serialport: Couldn't set term attributes");
            return -1;
        }

        return fd;
    }

    //

```

```

int serialport_close( int fd )
{
    return close( fd );
}

//
int serialport_writebyte( int fd, uint8_t b)
{
    int n = write(fd,&b,1);
    if( n!=1)
        return -1;
    return 0;
}

//
int serialport_write(int fd, const char* str)
{
    int len = strlen(str);
    int n = write(fd, str, len);
    if( n!=len ) {
        perror("serialport_write: couldn't write whole string\n");
        return -1;
    }
    return 0;
}

// AAL
int serialport_read_byte(int fd, uint8_t *byte, int timeout)
{
    uint8_t b = 0x00;
    int n = 0;
    do {
        // printf("byte...\n");
        n = read(fd, &b, 1); // read 1 byte
        // printf("n = %d\n", n);
        if( n==-1) return -1; // couldn't read
        if( n==0 ) {
            usleep( 1 * 1000 ); // wait 1 msec, try again
            timeout--;
            if( timeout==0 ) return -2;
            continue;
        }
    }
#ifdef SERIALPORTDEBUG
    printf("serialport_read_byte: n=%d b=0x%02x\n",n,b); // debug
#endif
    } while(n < 1 && timeout > 0);
    *byte = b;
    return 1;
}

```

```

}

int serialport_read_byte_file(FILE *f, uint8_t *byte, int timeout)
{
    uint8_t b = 0x00;
    int n = 0;
    do {
        // printf("byte...\n");
        n = fread(&b, sizeof(uint8_t), 1, f); // read 1 byte
        // printf("n = %d\n", n);
        if( n== -1) return -1; // couldn't read
        if( n==0 ) {
            if (feof(f))
                return 2;
            usleep( 1 * 1000 ); // wait 1 msec, try again
            timeout--;
            if( timeout==0 ) return -2;
            continue;
        }
    }
#ifdef SERIALPORTDEBUG
    printf("serialport_read_byte: n=%d b=0x%02x\n",n,b); // debug
#endif
    } while(n < 1 && timeout > 0);
    *byte = b;
    return 1;
}
// AAL

//
int serialport_read_until(int fd, char* buf, char until, int buf_max, int timeout)
{
    char b[1]; // read expects an array, so we give it a 1-byte array
    int i=0;
    do {
        int n = read(fd, b, 1); // read a char at a time
        if( n== -1) return -1; // couldn't read
        if( n==0 ) {
            usleep( 1 * 1000 ); // wait 1 msec try again
            timeout--;
            if( timeout==0 ) return -2;
            continue;
        }
    }
#ifdef SERIALPORTDEBUG
    printf("serialport_read_until: i=%d, n=%d b='%c'\n",i,n,b[0]); // debug
#endif
    buf[i] = b[0];
    i++;
    } while( b[0] != until && i < buf_max && timeout>0 );
}

```

```
    buf[i] = 0; // null terminate the string
    return 0;
}

//
int serialport_flush(int fd)
{
    sleep(2); //required to make flush work, for some reason
    return tcflush(fd, TCIOFLUSH);
}
```

A4: Larger Format Images

FMEA

Process Function	Potential Failure Mode	Potential Effect(s) of Failure	Sev	Potential Cause(s)/ Mechanism(s) of Failure	Occur	Current Process Controls	Detec	RPN	Recommended Action(s)	Responsibility and Target Completion Date	Action Results			
											Actions Taken	Sev	Occ	RPN
Data Transmission	Failure to Transmit	Nothing (is received)	1	Misaligned antenna, interference	3	Operator training and instructions	1	3	Operator training and instructions, interference isolation	AL and EM, March 7, 2017	Interference signal isolation	1	1	1
	Partial Transmission	RF receiver alerts receiving microprocessor, nothing happens	2	Misaligned antenna, interference	3	Operator training and instructions	1	6	Operator training and instructions, interference isolation	AL and EM, March 7, 2017	Interference signal isolation	2	2	4
	Corrupted Transmission	Hopefully, RF reciever corrects, discards if not possible	3	Interference	2	Operator training and instructions	2	12	Operator training and instructions	TBD				0
	Errant/Malicious Transmission	Bad data is transmitted	5	Interference, malicious attacker, too many being programmed too close together	2	Operator training and instructions, error correction	5	50	Encrypt communications, determine way to authenticate before programming	TBD				0

Design Verification Plan

CPE 450 DESIGN VERIFICATION PLAN AND REPORT												
Report Date		2/17	Sponsor		Dr. Snilkstein		Component/Assembly		REPORTING ENGINEER:			
TEST PLAN					TEST REPORT							
Item No	Specification [1]	Test Description [2]	Acceptance Criteria [3]	Test Responsibility [4]	Test Stage [5]	SAMPLES TESTED		TIMING		TEST RESULTS		NOTES
						Quantity	Type	Start date	Finish date	Test Result [7]	Quantity Pass	
1	Error Rate	Transferred data must match up to original data as effectively as possible in order to utilize data correction and device programming. (i.e. a 1kB file received must match up with at least 900 bytes of the 1kB file sent)	< 10% difference	Adam	PV	10	ABC	2/22/2017	3/4/2017	PROGRESS		Occasional dropped packet
2	Transmission Rate	A 1 kB file (regardless of the contents) should be able to be transmitted in less than 1 second)	1 kB/s	Adam	DV	5	B	2/22/2017	3/4/2017	FAIL	~0.333 kB/s	
3	Power consumption	During the testing, the transmission system (and receiving system) should not draw more than 2.5 W of power	2.5 W	Erik	PV	3	C	2/26/2017	3/6/2017	PASS	0.1 W	Current programs transmit at about 333 Bytes/sec
4	Transmission Power	When powering the antenna, if the antenna transmits more than 1 kW of power, we risk heating and destroying flesh. Thus, the whole system should draw less than 1 kW of power (If we meet the power consumption criteria, we will meet this criteria)	1 kW	Adam	PV	3	BC	2/26/2017	3/6/2017	PASS	0.1 W	
5	Transmission Distance	The system should be able to transmit through at least 15 cm of skin, fat, and muscle from either a chicken, cow, or pig (depending on availability and cost)	15 cm through animal tissue	Erik	CV	2	AC	2/26/2017	3/6/2017	IN PROGRESS		Tested on 3 cm of tissue with success, but not 15 cm yet