Abstract

This project creates a serial data link using a voice call between cell phones. This is done via the creation of a simple modem that uses On-Off Keying (OOK) as the modulation scheme, and an (8, 12) Hamming code for error correction. Input and output (Tx / Rx) serial signals from a local device are modulated into an audio signal; the audio is sent over the cellular network to a remote device. Serial signals can go both ways, allowing for both remote automation and remote data collection. Any microcontroller can call any number of other microcontrollers within this system; the only limit to connection is the establishment of a voice call between cell phones. The purpose of this data link is to allow for low-cost, mobile, serial communication between microcontroller projects, without limitations on geographic distance.
# Table of contents

## Introduction
- Microcontrollers and their applications ......................................................... 3
- Serial communication ....................................................................................... 3
- Common mediums of serial communication ..................................................... 4
- Current mobile solutions for virtually infinite geographic range ...................... 6
- Motivation I: The problem to solve .................................................................. 6
- Motivation II: Anticipated savings ................................................................... 7
- Motivation III: Why voice and not data? ........................................................ 8

## Background
- The MSP430: UART, clocks, and interrupts .................................................. 9
- Dial-up modems ............................................................................................... 10
- The limits of a 2G voice call ........................................................................... 11
- Error detection and correction ....................................................................... 11

## Theory
- Maximum theoretical throughput ................................................................. 12
- Modulation schemes ....................................................................................... 12
- Forward Error Correction: the Hamming Code ......................................... 13

## Testing and development
- Materials ........................................................................................................ 15
- Procedure ......................................................................................................... 17
- Trials of different modulation schemes ....................................................... 23
- Problems and troubleshooting ....................................................................... 27

## Results
- The circuit ........................................................................................................ 28
- The circuit in context ...................................................................................... 32
- Headset jack pin assignments ......................................................................... 33
- Serial byte buffer ........................................................................................... 34
- Error rates and throughput ........................................................................... 35
- Choice of final design .................................................................................... 35
- Comparison to maximum theoretical throughput ......................................... 35

## Discussion
- Success of Forward Error Correction ............................................................ 37
- Applications .................................................................................................... 37
- Comparison to other options ......................................................................... 44

## Conclusion
- Outcomes ......................................................................................................... 45
- Future work ..................................................................................................... 45

## Acknowledgements

## Endnotes

## Appendices
- Appendix A. MSP430 code for the OOK modem ........................................... 48
- Appendix B. Serial-controlled robot ............................................................... 56
- Appendix C. Remote light sensor and local LED display .............................. 58
- Appendix D. Whistle-controlled motor .......................................................... 60
Introduction

Microcontrollers and their applications

Microcontrollers are small computers that consist of a single integrated circuit. They are often programmable in C or a similar language. Microcontrollers have general purpose in/out (GPIO) pins, which enable them to control the components of a circuit. For this reason, they are found in many digital devices.

![Figure 1. A common microcontroller used in this project, the Arduino Leonardo.](image)

In recent years, the open-source Arduino microcontroller platform has created a learning community around it. Free online resources, combined with an easy-to-learn integrated development environment (IDE), has enabled hobbyists, students, and professionals to quickly and easily create projects that are controlled by Arduino microcontrollers. Many of these project plans are available for free online. The Arduino platform, along with the BeagleBone, Zigbee, and TI MSP430 platforms, have expanded the popular interest in, and accessibility of, microcontrollers.¹

Serial communication

One thing that makes microcontrollers useful is their ability to send and receive data. Microcontrollers use serial communication to “talk” to computer terminals, peripheral
digital circuits (commonly called “boards”), or other microcontrollers. There are different architectures of serial communications, such as UART, I²C, and SPI. This project concerns itself with UART, the Universal Asynchronous Receiving/Transmitter protocol. Most common microcontrollers can use the UART protocol. UART involves a stream of binary bits, usually grouped into blocks of 8 (called “bytes”) or blocks of 7 (called “characters”). Messages often consist of long streams of bytes or characters. UART only requires one wire per communication direction, unlike other serial architectures. On a digital device communicating via UART, there is a pin for receiving data (Rx) and a pin for transmitting data (Tx). The Rx pin of one device connects to the Tx pin of another device, and vice-versa.

Serial data is represented through voltage changes, which means devices that are communicating to each other over a serial wire must share a common ground, and must agree on some threshold for “high” and “low” voltage values. In UART, no clock is needed to sync devices. The device “listens” for a voltage change on its Rx pin, and then interprets the following binary voltages change as data. In UART, the voltage of a Tx or Rx pin usually varies between 0V and +5V (or 3.3V). A “resting” channel is +5V. A “start bit” is 0V. Then, after the start bit, 0V is a binary 0, and +5V is a binary 1. The timing of these bits are determined by the pre-set baud rate of the communicating device, which can vary from a few hundred to a few hundred thousand bits per second.

Common mediums of serial communication

The simplest way to achieve UART serial communication is through a pair of wires, one for data moving in each direction between two devices. To send and receive data between a computer and a microcontroller, a USB cable is often used. There are many ways to
implement wireless serial communication: RF transmitters and receivers, WiFi, Bluetooth, and others are all readily achieved through the use of circuit boards for serial communication over those protocols. Each method of communication has its pros and cons. Ultimately, though, all of these methods are in some way limited in physical range between devices:

- Wire is reliable, high-throughput, easy, and inexpensive over short distances, yet is extremely limited in mobility.
- WiFi is inexpensive, high-throughput, and wireless, yet requires a network, and configuration with that network. If the device leaves the network range, it may encounter other networks, but will not be configured to use those networks; thus, WiFi is limited in range, as well.
- Bluetooth does not require a network, but is limited in range and throughput. RF transmitters and receivers vary widely in both cost and performance; some can communicate establish serial communication between devices many miles apart.

Devices can communicate with each other over the Internet, which allows for virtually infinite geographic range between devices. Yet, this means the devices must be within range of a WiFi or other network. Thus, connecting to the Internet does not necessarily mean a pair of devices can be mobile.
Current mobile solutions for virtually infinite geographic range

One existing solution allows for communication between mobile microcontrollers over virtually infinite geographic range. The cell phone networks, both CDMA (code-division multiple access) and GSM (Global System for Mobile communications) networks, can be accessed via CDMA or GSM boards. CDMA boards and SIM card (GSM) boards establish a data link between the microcontroller and the mobile network. The data can be sent and received on a computer terminal that links to a server operated by the telecom. Alternatively, that data can be forwarded to another board, allowing for inter-device communication over infinite range (see figure 2). Yet, this solution is relatively expensive and complicated to set up. The boards cost a minimum of $60 each. They require the purchase of a data plan subscription, which is relatively expensive compared to voice/text plans. Furthermore, each new board needs to be activated, which is a $30 fee per-device on Verizon. Thus, sending a simple, small amount of data between two microcontrollers would cost at least around $200 over CDMA. Aside from costs, working with these boards is complicated, and internet forums are rife with questions from frustrated users.

Motivation I: The problem to solve

This project was meant to solve a specific problem: how can mobile microcontrollers send small amounts of data to each other in a way that is affordable, reliable, and easy? The solution is the repurposing of cell phones as data transmitters and receivers.

This project will use a specially-designed modem to interface with the headset jack of a common cell phone. The signal path for sending data can be described as follows: UART serial data will be received by the modem on its Rx pin. That data will be modulated into audio. That audio will be input into a cell phone headset jack via the wires normally used
by a headset microphone. The audio is sent to another phone (which may be anywhere in
the world) via a voice call. On the receiving end, the received audio will be input to the
modem via the wires normally used for a headset’s earphones or earbuds. The audio will be
demodulated back into data, where it will be outputted on the modem’s Tx pin. See figure 3
for a rendering of this signal path.

Figure 3. A block diagram of the signal path, as conceived at the beginning of this project: Some circuitry,
powered by an MSP430 microcontroller, will enable serial bytes to be modulated and sent over the cell phone
network as audio; on the receiving end, audio will be demodulated back into data. One setup may be far away
(remote), while the other setup may be near a human user (local).

Motivation II: Anticipated savings

This implementation should cost considerably less than a CDMA or GSM board, mostly
due to the fact that it is designed to be lower-cost in every way. The user does not need to
pay an activation fee if they are using an old phone that has already been activated. Voice plans are far cheaper than data plans, and can be purchased on a per-minute basis. Furthermore, the hardware necessary to build this modem is fairly inexpensive. With a minimum activation balance of $15 on a prepaid Verizon phone, sending the simplest, smallest amount of data between two mobile microcontrollers with this platform would cost $30 plus the cost of hardware. That is the start-up cost for a pair of linked devices. If only one phone is dedicated to a microcontroller project, the cost of adding this mobile serial link to the project can be as low as $15 (plus the cost of the modems, which were calculated to be about $10 each later in this report), which would allow for up to 15 days of unlimited connection time between devices. Alternatively, the cost of the connection could be less than $0.0008/minute with a $35/month unlimited talk plan from Verizon. This is about 1/5th the cost of adding a CDMA or GSM board to a microcontroller project (around $110 per device).

*Motivation III: Why voice and not data?*

There are many reasons to use voice calls rather than attempting to use text messages or CDMA/GSM data. All cell phones are capable of establishing a voice call with another cell phone. Voice calls tend to be cheaper than data in many ways, as discussed above. Furthermore, there is a standard way to interface with voice. Most cell phones have a headset jack. While many cell phones can interface with the network with serial commands (such as the AT instruction set on certain Motorola phones), the specifics of these serial commands vary widely among carriers and cell phone models. There is no standardized way to interface with a phone over its USB port. The closest thing to a “standard” that exists is the AT command set, and there are very few phone models that can be easily controlled
using AT commands. This author was able to find only two out-of-production Motorola models that other experimenters on the Internet have successfully interfaced with using AT commands.  

**Background**

*The MSP430: UART, clocks, and interrupts*

The modem created in this project uses an MSP430G2553 microcontroller to receive serial data, store it in a buffer (discussed in the section *Serial byte buffer*), encode it using the Hamming code (discussed in the section *Forward error correction*), and control the modulation circuit; the MSP430 also demodulates and decodes the received audio, then outputs the received data as a serial byte. Three major features of the MSP430 enable this: the UART module, the internal clocks, and interrupts. The presence of these features, plus the high affordability of the MSP430G2553, led to the selection of the MSP430 for this project.  

![Figure 4. The MSP430G2553 microcontroller, used in this project.](image)

The MSP430 uses its UART module for serial communication. Activating the UART module two pins on the MSP430: Rx (pin 1.1), to receive signals, and Tx (pin 1.2) to transmit signals. The UART module only stores one received byte at a time, in the buffer UCA0RXBUF. It is possible to store more received bytes by creating an array of bytes (see
The UART module sends a new serial byte (on the Tx pin) whenever the sending buffer UCA0TXBUF is updated.10 Another function used in the project is the clock. The MSP430 has multiple clocks. The one used in this project (picked somewhat arbitrarily) is Clock A, or Timer A. Timer A defaults to 1 MHz, but can be divided by 2, 4, or 8. In this project, Timer A is divided by 8 and set to 125 kHz. An extremely useful function of Timer A is that the variable TA0R is always the present value of Timer A, whenever it is called.

To read the received frequencies, the MSP430 logic enters an interrupt service routine (ISR) whenever the received audio signal reaches a certain threshold. So, whenever the voltage on Pin 1.0 (selected arbitrarily as the audio signal input to the MSP430) rises above 1 V, the MSP430 enters the ISR for Pin 1.0. Once it enters the ISR, it makes a note of the last time it entered the ISR, by reading the value of TA0R. By using these interrupts, the MSP430 can determine the frequency of the audio it is receiving, or if there is any audio being received at all.

Dial-up modems

Modems have been in use for decades. The most familiar type of modem is the dial-up modem, used to connect computers over telephone landlines. The earliest dial-up modems, introduced in the 1970s, used FSK with frequencies around 1200 Hz. Later modems used more complex techniques called quadrature amplitude modulation (QAM), which involves extracting more data from an FSK signal by observing the phase changes of the signal. This is how modern dial-up can exceed 56kbps data rates.11
The limits of a 2G-voice call

One limit is the bandwidth of a cell phone call. Cell phones theoretically transmit audio between 200 and 3200 Hz, which is just enough fidelity for voice. The communication channel chosen – a 2G or 3G voice call – is prone to gaps, ticks, and noise in the signal. Cell phone networks use a range of methods to send digital voice data, including frequency hopping, and making use of multiple “cellular” towers (in different geographic cellular zones) while moving during a call. For these reasons and others, we should not expect a cell phone voice connection to be identical to a landline voice connection, as used in traditional dial-up.

Error detection and correction

There are two forms of error correcting codes: Forward Error Correction (FEC) and Automatic Repeat request (ARQ). In FEC, errors can be detected and corrected in one direction of communication. This requires that extra parity bits be sent along with information bits. In ARQ, the receiving device gives the sending device a summary of what was received, and if there is any error, the sending end re-sends the parts that had errors. Depending on the flow of information being sent, different error correction schemes make sense. Because of the relatively long delay in sending messages over a cell phone call, ARQ was not implemented for this project. Instead, FEC was used. Some forms of FEC are best for long streams of data. An FEC code such as the Reed-Solomon code would work best in this case. But since this project aims only to send small packets of data, usually just one byte long, a different type of FEC was used: the Hamming Code. The Hamming Code implementation is discussed later in the Theory section of this report.
Theory

Maximum theoretical throughput

There is a limit to the amount of data that can be sent through a channel. For this project, there are three factors that need to be accounted for: the bandwidth of the audio channel; the error introduced through ticks and gaps; and the signal-to-noise ratio (SNR). A useful place to start is the Shannon-Hartley theorem, given by:

\[ C = B \log_2 \left( 1 + \frac{S}{N} \right) \]

Where \( C \) is the maximum theoretical bitrate, \( B \) is the channel bandwidth in Hz, and \( S/N \) is the signal to noise ratio. For a standard 2G voice call, the maximum theoretical bandwidth is between 300 and 3000 Hz; however, testing with the oscilloscope has found that the actual bandwidth may be different. See the section “Calculating maximum throughput” for more details.

Modulation schemes

To send data as audio, the data must be modulated and then demodulated. This is the primary function of a modem (modulator/demodulator). There are many different modulation schemes. This project experimented with several different schemes:

- Amplitude modulation: the representation of different bit combinations as different amplitudes of audio. For example, a 0 dB pulse might represent ‘00’, -3 dB represents ‘01’, -6 dB represents ‘10’, and -9 dB represents ‘11.’

- Frequency modulation: the 16 possible half-bytes (0000 through 1111) are represented by 16 different frequencies. For example, a pulse at 1200 Hz represents ‘1111’, 1100 Hz represents ‘1110,’ 1020 Hz represents ‘1101,’ and so on.
• Frequency shift keying: two different carrier frequencies are selected to represent the two different bits. A third frequency, usually in the middle, is a neutral frequency that does not represent data. For example, the channel could have a constant neutral frequency of 1200 Hz. When data is to be represented, a ‘1’ could be represented by a 5 ms pulse at 1230 Hz, and a ‘0’ represented by a 5 ms pulse at 1170 Hz. See figure 5 for a visual representation of FSK.

• On-Off keying: in On-Off Keying (OOK), the nontransmitting (neutral) state is the absence of a carrier frequency. To represent a byte of data, OOK begins with a pulse at the carrier frequency. Then, the presence of a pulse represents a ‘1,’ and the absence of the carrier frequency represents a ‘0.’ See figure 6 for a visual representation of OOK.

![Figure 5](image-url)  
**Figure 5.** A visualization of frequency-shift keying. The binary ‘0’ and ‘1’ each correspond to different frequencies.

![Figure 6](image-url)  
**Figure 6.** A visualization of on-off keying. The binary ‘1’ corresponds to the carrier frequency being ‘on,’ and the binary ‘0’ corresponds to the carrier frequency being ‘off.’

*Forward Error Correction: The Hamming Code*
The FEC codec chosen for this project is the Hamming Code. Because the project sends one byte at a time, the Hamming Code implementation needs to send 8 bits of information. In a Hamming code implementation, the number of parity bits must be at least:

\[ p \geq \log_2(b) \]

Where \( p \) is the number of parity bits (an integer) and \( b \) is the number of information bits. Thus, for an 8-bit byte, the Hamming code must have at least 4 parity bits, meaning that the Hamming codeword is 12 bits. Thus, this is called an \((8, 12)\) Hamming code.

To use an FEC like the Hamming code, a codec, or encoder/decoder, must be used. On the encoding side, the 4 parity bits are placed in positions 1, 2, 4, and 8 of the 12-bit codeword. The values of the 4 parity bits are determined as follows:

- Parity bit \( p_1 \) is the odd parity of information bits \( b_1, b_2, b_4, b_5, \) and \( b_7 \). So if the total number of 1's in \( b_1, b_2, b_4, b_5, \) and \( b_7 \) is even, \( p_1 \) is a 0. Otherwise, it's a 1.
- \( p_2 \) is the odd parity of \( b_1, b_3, b_4, b_6, \) and \( b_7 \).
- \( p_3 \) is the odd parity of \( b_2, b_3, b_4, \) and \( b_8 \).
- \( p_4 \) is the odd parity of \( b_5, b_6, b_7, \) and \( b_8 \).

The result is a 12-bit codeword, where the 4 parity bits describe the 8 information bits. See figure 7 for a visualization of the parity encoding.

Decoding a Hamming codeword involves checking to make sure all 4 parity bits are correct. If \( p_2 \) and \( p_3 \) are incorrect, for example, we can find and fix the error: \( p_2 \) is in position 2 of the codeword, and \( p_3 \) is in position 4. Thus, the error must be in position 6 \((2+4=6)\), which is information bit \( b_3 \). The implementation of the Hamming codec can be found in the MSP430 code in Appendix A.
Figure 7. Example of a Hamming codec for the byte “10011010”.

Testing and development

Materials

The primary tools used for this project were:

- Breadboard
- Oscilloscope, for observing audio, serial, and other electrical signals
- CodeComposer, for programming and debugging the MSP430
- Ableton Live 9, for creating audio samples used during the testing of different modulation schemes.
- USB-to-serial converter, interfacing with Terminal on a MacBook at 4800 baud
- Samsung SCU-U410 prepaid cell phone, connected to the Verizon network
- iPhone 4S, connected to the Verizon network
- Eagle CAD, a free circuit design tool that allows for the construction of schematics and PCB designs

Figure 8. The broadband used for this project, at the conclusion of the project.
Figure 9. The oscilloscope used for this project.

Figure 10. The MSP430 launchpad, programmed with TI’s Code Composer.

Figure 11. A screenshot of Ableton Live 9, the audio editing program used to generate test tones for this project.
Figure 12. The USB-to-serial converter used in this project, used to read and write serial bytes from a computer. Available from Adafruit, along with drivers and instructions.

Figure 13. The Samsung SCU-U410 used for this project.

Figure 14. An iPhone 4S, used for this project.

Procedure

The procedure for creating this modem revolved mostly around repeated testing of different modulation schemes, and the precise settings for those schemes. This project took many different turns, mostly due to the fact that the author of this report was largely unfamiliar with cell phone technology, and the performance of voice calls between cell phones. The full procedure for this project was, roughly, this:
• Acquired several used phones, including
  o a Samsung SCU-U410
  o an LG Tracfone (with 300 minutes remaining)
  o a Motorola phone (unknown "brick" model)
  o an old Android phone made by Samsung
  o the author’s personal phone, an iPhone 4S.

• Acquired headsets for the LG Tracfone and iPhone, assuming that the Tracfone is the best option, due to the fact that it has 300 minutes on it. Discovered that the Tracfone headset adapter does not work. Replaced that headset adapter. Realized that the headset jack on the phone probably does not work. Abandoned the idea of using the Tracfone.

• Bought minutes for the Samsung SCU-U410 and turned it into a Verizon prepaid phone. This phone was selected because it is small, light, and has a long battery life, making it ideal for mobile projects. It also has an easily accessible headset jack.

• Realized that the Samsung SCU-U410 headset jack is a 2.5 mm jack. Bought an adapter so that it will work with the 3.5 mm headsets.

• Established a voice call between Samsung SCU-U410 and the iPhone 4S. Tested headsets, including the 2.5 mm adapter. Found that both headsets worked as expected.

• Cut the headsets apart, and spliced their wires. Attached leads to the wire ends of one headset; on the other headset, attached the microphone leads to a 3.5 mm stereo audio jack, so that audio could be played into the voice call from a computer.

• Experimented and learned about the headset jack’s audio output. Discovered that the audio input and output share a common ground. Also discovered that the TRRS (tip-ring-ring-sleeve) pin assignments are different for the two phones: on the Samsung, the orange lead is the mic V+; on the iPhone, the red lead is the mic V+.

• Discovered that the voltage of the black “ground” wire on the headset changes voltage in the opposite direction of the audio output wires.
  o For example, if the desired audio output is a 1 kHz sine wave with a peak-to-peak voltage of 100 mV, the common ground lead and the audio output leads will each have a 1 kHz sine wave at 50 mV, but 180 degrees out-of-phase which each other. The result is a potential difference between the common ground and the audio out wires that is a 100 mV sine wave.
  o This appears to be a smart design on the part of the phone manufacturers, since it cuts the power consumed by audio playback.
However, when these audio leads are plugged into a circuit board, it leads to a problem: if the voltage output is too high, any circuit the audio is plugged into acts unpredictably.

- To solve this problem, created a differential amplifier for the audio input to the circuit board. The output of this differential amplifier is the absolute voltage value of any audio coming from the headset jack, relative to the breadboard’s ground, not the headset’s ground.

- Realized that the TLV rail-to-rail op-amp is a better choice for the differential amplifier, since this op-amp does a better job of outputting voltages closer to the input power (which will likely be +5V and GND, since that is the standard peripheral power option for Arduino and MSP430 microcontrollers).

- Using the TLV op-amp, gave this differential amplifier the highest gain possible, so that the relatively small audio signal can be read by the MSP430.

- Began testing various modulation schemes:
  - Using Ableton Live and the 3.5 mm audio cable, sent various sounds over a phone call.
  - Using the oscilloscope and the other spliced headset jack, observe the audio signal that is received.
  - From these readings, can determine which modulation scheme works best.
  - For each new modulation scheme, new test audio needs to be generated on Ableton Live. Sometimes, new circuitry needs to be created as well.
  - This process lasted several weeks, and was informed by trial-and-error, along with information from the Internet. Many weeks were wasted with a modulation scheme that was ultimately rejected in the final weeks of the project.
  - For a more expanded view of what modulation schemes were tried, see the section “Trials of different modulation schemes.”
  - Eventually, an On-Off Keying (OOK) modulation scheme was selected as the most promising.
    - The procedure, from this point on, is based around the creation of an OOK modem.
    - The actual procedure for this project involved much more trial-and-error for the different modulation schemes.

- Added a comparator to the signal path, so that the signal being received is either 0V or 5V. Thus, the signal is more definitively “on” or “off”.

- Wired the output of the comparator to Pin 1.0 on the MSP430 Launchpad.

- Set up Timer A on the MSP430.
• Set up a rising-edge interrupt on the MSP430’s Pin 1.0.
  o In the ISR (interrupt service routine), assigned a variable to be the current value of Timer A, using TA0R.
  o Each time the ISR is entered, got a period reading by comparing the current and previous values of TA0R. This period is the inverse of the frequency being read. Since Clock A is set to 125 kHz, each integer in the period length signifies 8 microseconds. A period of length 104 clock units, for example, is 832 microseconds long, which translates to a frequency of roughly 1200 Hz.
  o Set the observed period value to a variable called ‘PeriodRead.’
  o At this point, the demodulation hardware circuit was completed: frequencies coming from the phone can be read by the MSP430.

• Now, began construction of the demodulation hardware circuit. The plan for the demodulation circuit was this:
  o an MSP430 pin will trigger an oscillator. The frequency output by the oscillator is the carrier frequency.
  o Put this signal through a lowpass filter (LPF) to make it more like a sine wave.
  o Give it a gain of -30dB so that it can be input into the phone’s headset jack.

• Constructed an oscillator using a 555 timer. Tuned it to approximately 1200 Hz by selecting appropriate capacitors and resistors. 1200 Hz was selected because it is a frequency that comes through a 2G phone call clearly. Used complex formulas to determine capacitor and resistor values. See endnote 15 for the derivation of these formulas. Selected for 1 uF capacitors because they are abundant in the lab; let the formulas determine the resistor values.

• Wired the MSP430’s Pin 1.4 to the ‘reset’ pin of the 555 timer. When Pin 1.4 is high, the oscillator will output a frequency; when it is 0 V, the oscillator will output a static DC voltage.

• Determined the input voltage required for the mic input wire to the phone’s headset jack. Using Ableton Live, sent a 1200 Hz test tone. Square waves, sine waves, and sawtooth waves were be tested. Observed that sine waves sound the least-distorted on the receiving end. Tested various amplitudes of the audio input into the phone. Found that if the audio signal is too strong, some internal circuit cuts off the sound. This is likely due to the fact that the headset is expecting a very low-power voltage reading from a passive microphone. Found that the ideal sine wave amplitude is 120 mV, peak-to-peak.
• Constructed a passive LPF for the 555 timer output. Experimented with different cutoff frequencies (ie., resistor and capacitor values) until it is discovered the 1200 Hz 5V square wave has become a 100-150 mV sine wave. This was observed using the oscilloscope.

• Found that this passive LPF has a high DC gain. This high DC gain appears to affect the mic input circuitry on the phone. Continue tweaking the circuit, using different capacitor and resistor values, until the DC gain is minimized.

• Found that, although the audio-input circuit works well for the Samsung phone, the iPhone 4S cannot accept audio from the circuit.
  o After some research, it was found that the iPhone defaults to setting the mic input pin to GND, if an unauthorized headset is plugged into its headset jack.
  o Found that the Apple hardware development standard requires a certain audio-processing chip for creating authorized headsets.
  o Found that the TRRS standard for the iPhone is shared by new Samsung phones, as well. Realized that there must be some circuit out there that enables headsets to work with smart phones.
  o Finally, found a circuit online that allows a line-input through the mic pins on a smartphone. Built the “iPhone adapter” and attached it to the breadboard, between the LPF and the spliced headset pins. Audio from the breadboard can now be sent from an iPhone through the headset jack.

• Programmed, debugged, and re-programmed the MSP430 until the code was finalized. This included:
  o Implemented a buffer to store bytes received on the MSP430’s Rx pin, since the modem will send modulated audio slower than it might receive serial data to send.
  o Established a baud rate of 4800 baud, which is readily attainable on most microcontrollers.
  o Implemented an (8,12) Hamming codec.
  o Wrote all the code necessary for modulating a 12-bit Hamming codeword as audio:
    • A byte is received on the Rx pin (Pin 1.1).
    • The byte is encoded using the (8,12) Hamming codec.
    • To output audio from the modem, the 555 timer is triggered by Pin 1.4 on the MSP430.
    • Each 12-bit message starts with a “start bit,” for a total of 15 bits. Each bit is 15 ms, for a total of 135 ms per message.
    • For each ‘0’ (off), the 555 timer is deactivated by setting Pin 1.4 to GND (digital LOW).
• For each ‘1’ (on), the 555 timer is activated by setting Pin 1.4 to +3.3V (digital HIGH).
  o Wrote all the code necessary for demodulating the audio received:
    • Once the ISR is entered the first time, begin “listening” to more “pings” using the ISR.
    • Each single 1200 Hz waveform that is received (in other words, each time the comparator output goes from low to high) is added to a buffer.
    • This modulation scheme initially uses 15 ms bits.
    • If it’s 0-15 ms since the first ping is received, ignore it.
    • If it’s 15-30 ms since the first ping is received, add 1 to “ReadHits[0]” for each ping received during this period.
    • If it’s 30-45 ms since the first ping is received, add 1 to “ReadHits[1]” for each ping received during this period.
    • ... and so on, until it’s been 135 milliseconds (15 * (1+12)) since the first ping was received.
    • ReadHits[0] corresponds to bit n of the received codeword, the 12 bits of the codeword numbered 0 – 11.
    • If ReadHits[0] is greater than 10, the ReadBits[0] is interpreted as being a “1” value. If it’s less than 10, ReadBits[0] is interpreted as being a “0” value.
    • These array ReadBits[] is now the received 12-bit codeword. It is the interpreted and corrected by the Hamming codec.
    • The decoded 8-bit byte is the outputted on the Tx pin of the MSP430. This means that:
      • the Tx pin of the modem is essentially the Rx pin of the MSP430 (bytes that are to be sent out go into the MSP430 on its Rx pin)
      • the Rx pin of the modem is the Tx pin of the MSP430 (bytes that are received come out of the MSP430 on its Tx pin).
• Replicated the breadboard circuit on a protoboard (a half-sized circuit board).
• Loaded the MSP430 code onto multiple different MSP430G2553 chips using the MSP430 Launchpad.
• Added a 3.3V voltage regulator, a pull-up resistor, a pull-down capacitor, and a reset button so that the MSP430 chip can work outside of the Launchpad.
• Added 0.1” pins to the protoboard for +5V, GND, Rx and Tx, so that the modem can be easily added to any microcontroller project, just like other common peripheral boards.

• Added 0.1” pins to the protoboard so that the headset jack can be plugged in. Did not hard-wire the headset jack to the board because, as discussed, 2.5 mm and 3.5 mm headsets have different TRRS configurations. It is up to the user to plug-in the wire leads into the right spots on the protoboard.

• Soldered the finished circuit.

• Tested the finished circuit.

• Modified the code on the MSP430 to include a “secret feature:” the ASCII character ‘*’ is outputted by the modem whenever it receives a frequency at 510 Hz (+/- 2 Hz).

• Built example applications for the modem, including:
  o a small robot car, controlled by computer terminal
  o a photosensor that remotely triggers an LED display on a different microcontroller
  o a motor that opens a small box when the 510 Hz frequency is received

• Constructed a schematic of the finished circuit in Eagle CAD.

• Constructed a board (a .brd file) of the finished circuit in Eagle CAD.

• Sent the .brd file to OSH Park circuit board printers.

Trials of different data modulation schemes

The bulk of the testing for this project revolved around trying different data modulation schemes: or, different ways to represent data as sound. While there are many ways to represent data as an audio signal, the challenge was to find a modulation scheme that works well over a 2G voice call. To evaluate the performance of different schemes, an oscilloscope was used to visualize the signal at the receiving end of the call. Comparing the sent (original) audio to the received audio gives a sense of what works and what does not.

To generate the test audio, Ableton Live 9 was used. A computer’s headphone jack interfaced directly with a phone’s headset jack to send audio from the computer (Ableton Live 9) to a phone (see figure 15).
The following modulation schemes were tried, with varying levels of success:

- The first modulation scheme tested was the representation of segments of bytes as blips of audio at varying amplitudes. For example, a blip at 0 dB might represent ‘00,’ -3 dB might represent ‘01,’ etc. This scheme did not appear effective, because a constant-amplitude test tone did not come out the other end of the voice call at a constant amplitude. There was too much variation in amplitude for this scheme to work, as seen in figure 16.

- The second modulation scheme involved using 16 different frequencies to represent 16 different 4-bit values; i.e., 700 Hz might represent ‘0000,’ 800 Hz might represent ‘0001,’ and so on. At first, this scheme seemed promising, because the frequency at the receiving end was pretty consistently unchanged. See figure 17 for an oscilloscope shot of a constant-frequency tone at the receiving end of a voice call. However, to differentiate between 16 different frequencies between 550 and 2500 Hz, there needs to be a significant amount of time for the frequency to settle, which took as long as 500 ms for the frequencies tested. This meant that the transmission rate was roughly 1 byte per second. The error rate was also very high, as the MSP430 could only read frequencies with a limited degree of precision. This scheme worked but was not ideal.

- Frequency-Shift Keying (FSK) was another modulation scheme that showed promise, but was too hard to implement effectively. As shown in figure 18, an FSK scheme alternating between 600 and 1200 Hz produces a noticeable difference in frequencies at the receiving end. When filtered through the comparator (signal in
green), the two different frequencies became even more apparent. However, there were many small blips in the signal that suggested frequencies that were neither 600 or 1200 Hz were being read by the MSP430, which made it very difficult to demodulate. This scheme may be promising if explored further.

- Finally, On-Off Keying (OOK) was tested. Immediately it appeared promising on the oscilloscope, especially when analyzing the “cleaned-up” signal after the comparator. At first, each bit was represented by 15-ms periods. The carrier frequency chosen was 1200 Hz, because it was found to be a frequency that came through the voice call with relatively high audible clarity. The oscilloscope results showed that the received audio signal resembled the sent audio signal enough to try implementing with the MSP430 (see figure 19).

**Figure 16.** The audio received over a 2G voice call for a test of amplitude modulation. Two 2400 Hz test tones were sent at a constant 0 dB (top) and a constant -7 dB (bottom). Notice that the received audio varies greatly in amplitude, suggesting this is not an effective modulation scheme for the voice call medium.

**Figure 17.** The audio received over a 2G voice call for a test of frequency modulation. A constant test tone was sent, and although the amplitude varies somewhat, the frequency received is highly consistent.
Figure 18. The audio received over a 2G voice call for a test of frequency shift keying (FSK). The comparator output (in green) highlights the fact that different frequencies are indeed received; however, at these settings (5 ms per bit), the two carrier frequencies (600 Hz and 1200 Hz) are not clearly recognizable.

Figure 19. The audio received over a 2G voice call for a test of on-off keying (OOK). The audio received over the 2G voice call (top) and the “processed” audio (the comparator output) at the bottom of the image highlights the clear distinctions that can be made between “on” and “off” on the receiving end. As we can see in the comparator output (which is sent to the MSP430), there are some gaps in the audio where there should be audio. A “deciding” protocol was implemented to decide between ‘0’ and ‘1’ bits.

Because the voice call introduced a great deal of imprecision, a “deciding” protocol was used to determine if a received bit was a ‘1’ or a ‘0’. With the OOK modulation scheme, for each 15-ms bit, the MSP430 counted the number of interrupts it received from each sine wave pulse. In 15 ms, a 1200 Hz carrier frequency has exactly 18 periods \((0.015 / \frac{1}{1200}) = 18\). So, if a given 15-ms period has at least 9 counted pulses (half of 18), the MSP430 counts it as a 1. If it has less than 9 pulses, it counts it as a 0. See figure 19 for an
illustration of why this deciding scheme is important. There are some pulses where there shouldn’t be, and there are some gaps where there should be pulses.

Problems and troubleshooting

Throughout the course of this project, many unprecedented problems came to the fore, including but not limited to:

- No working headset could be found for the LG Tracfone, so it could not be used, despite the fact that it had 300 minutes on it. All phones in this project were recycled phones, so they were all free. This phone, though, was the only phone that had minutes on it.
  - Solution: bought minutes for the Samsung phone.
- During soldering onto the proto board, wires and contacts often crossed by accident, causing short-circuiting.
  - Solution: careful un-soldering and re-soldering.
- The iPhone has a strange, unknown internal circuit for its headset jack, meaning that the mic pin is automatically set to ground when anything besides an authorized headset is plugged into it.
  - Solution: found a circuit online that somehow, inexplicably, solves this problem. When built, this circuit worked flawlessly. There was no explanation for why the iPhone required this circuit, or how it activates the mic input. The circuit resembles a lowpass filter, and is shown in figure 20.
- The comparator output has a significant amount of jitter, which leads to rapid, frequent, undesirable interrupts on the MSP430. This leads to completely unreliable frequency readings.
  - Solution: put a 4th-order Butterworth lowpass filter at the output of the comparator. Tune the filter to cut off frequencies about 3 kHz, just above maximum audible frequency on a 2G voice call.
- The buffer built into the MSP430 code does successfully store bytes to-be-transmitted, yet, when it receives new bytes while transmitting modulated audio, the audio output is interrupted, leading to inaccuracies in the transmitted data.
  - Solution 1: input new bytes into the modem one at a time, with 400 ms between each byte.
  - Solution 2: when sending long streams of bytes, such as text strings, add a few unnecessary characters at the beginning, such as spaces. These spaces will be corrupted, but everything that is sent after the buffer is no longer being added to is unaffected by this problem.
Figure 20. The iPhone adapter circuit, found online. See endnote #16 for the source.

Results

The circuit

The choice of OOK as a modulation scheme resulted in a circuit with two primary signal paths: a path from the Rx pin to the phone for modulation of data; and a path from the phone to the Tx pin for the demodulation of sound.

For the modulation signal path (shown at top in figure 22): a byte is received by the MSP430 on its Rx pin. The data is encrypted using the Hamming code and turned into a 12-
bit codeword. Then a 555 timer, tuned to 1200 Hz, is triggered using Pin 1.3, in a series of On/Off blips that represents the codeword using OOK.

For the demodulation signal path (shown at bottom in figure 22): audio from the phone’s headset is amplified using the differential amplifier (the left-most op amp); the amplified signal is sent to a comparator that compares it to a constant 1.6 V DC signal (the LM311 comparator); the signal outputted by the comparator is now binary. A 2nd-order Butterworth low-pass filter eliminates any jitter that may falsely trigger the MSP430’s ISR. Originally, this was a 4th-order LPF, but because one of the op-amps soldered onto the IC broke somehow, the filter was adapted to be a 2nd-order filter (tuned to a cutoff frequency of 4500 Hz), which works just as well at eliminating jitter. Then the jitter-free LPF output is sent to Pin 1.0 on the MSP430, where it triggers the ISR. As soon as the ISR is triggered for the first time, the MSP430 begins to “listen” for thirteen 15-ms periods (1 start bit + 8 data bits + 4 parity bits) for a total of 0.195 seconds.

![Figure 22. Block diagram of the modulation signal path (top) and demodulation signal path (bottom).](image-url)
The voltage regulator supplies 3.3 V to the MSP430; the 47k Ohm resistor and the capacitor attached to the RST pin provide the necessary voltage to activate the MSP430 via its reset pin; the pushbutton switch is like a “reset” button for the MSP430, and connects the reset pin directly to ground. The two 4-by-1 pins act as port for the microcontroller (+V, GND, Rx and Tx) and for the phone (mic, common ground, left earpiece, right earpiece). The two op-amps are part of the same IC, a TLV2772 (not represented by the schematic, because Eagle CAD did not have that exact op-amp package).

The schematic was also re-created in Eagle CAD, shown in figure 23. A circuit board layout was created as a .brd file, shown in figure 24. That board design was sent to OSH Park for fabrication, resulting in the PCB shown in figure 25. OSH Park also fabricated a small circuit board for the iOS/smartphone adapter circuit, shown in figure 26.

**Figure 23.** Schematic of the final circuit, generated in Eagle CAD.
Figure 24. Circuit board layout of the final circuit, generated in Eagle CAD.

Figure 25. PCB of the final circuit, printed by OSH Park and designed in Eagle CAD.
**Figure 26.** Schematic (top) and circuit board layout (middle) of the smartphone adapter circuit, generated in Eagle CAD; PCB (bottom) of the smartphone adapter circuit printed by OSH Park and designed in Eagle CAD.

*The circuit in context*

The block diagram in figure 27 shows where the OOK modem sits on the signal path as part of a larger system involving multiple microcontrollers.

**Figure 27.** The circuit in context: two copies of the final circuit are plugged into two separate phones, which are on a call with each other. This enables two-way serial communication between microcontrollers (or any other serial-enabled device) over a virtually infinite geographic range.
Headset jack pin assignments

Different kinds of cell phone have different headset jack pin assignments. Using a typical TRRS (tip-ring-ring-sleeve) headset jack, a standard phone has the following pin assignments for tip/ring/ring/sleeve, respectively: left speaker, right speaker, ground, and mic. In figure 28, a standard 3.5 mm headset jack pinout is plugged into the board (mic = red, ground = black, left speaker = brown, right speaker = orange). In figure 29, a 2.5 mm or Nokia phone headset jack pinout is plugged into the board (mic = orange, ground = black, left speaker = brown, right speaker = red). The board is designed so that the user can switch between 2.5 mm (Nokia) and 3.5 mm (standard) headsets.

Figure 28. Headset jack pin arrangement for a standard 3.5 mm headset. The red wire, for the sleeve of the TRRS jack, is plugged into the mic pin on the left.
A serial byte buffer was implemented in the MSP430 code. This was needed because the modem may receive new bytes to send while it is in the process of sending a byte. This happens when sending a string of characters, for example. It takes a few hundred milliseconds to send each byte via OOK, and at 4800 baud, as many as 600 bytes per second per may be received by the modem. The serial byte buffer stores all received bytes in a buffer, and sends each byte in the order they are received. Every new byte received on UCA0RXBUF gets put into the buffer. The buffer holds up to 50 bytes.

Unfortunately, the buffer does not work perfectly. As discussed in the section Problems and troubleshooting, at this point, the receipt of new bytes corrupts the sending process slightly. It can be worked around, though, as discussed in Problems and troubleshooting.
Error rates and throughput

The error rate was calculated by sending 200 characters from one computer terminal to another, using a USB-to-serial converter, two OOK modems, and an iPhone and a Samsung phone on a 2G voice call with each other. Different modulation schemes produced different error rates. Each modulation scheme tested also took different amounts of time to send each byte. This resulted in different throughputs for the different modulation schemes:

<table>
<thead>
<tr>
<th>Modulation scheme</th>
<th>Success rate (% correct out of 200-byte test)</th>
<th>Time to send 1 byte of data (ms)</th>
<th>Throughput (bits per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-pitch keying (for comparison)</td>
<td>50% (approx.)</td>
<td>1000</td>
<td>4 (approx.)</td>
</tr>
<tr>
<td>15-ms OOK without FEC</td>
<td>22%</td>
<td>135</td>
<td>13.03</td>
</tr>
<tr>
<td>28-ms OOK without FEC</td>
<td>57.5%</td>
<td>252</td>
<td>18.25</td>
</tr>
<tr>
<td>28 ms OOK with Hamming code</td>
<td>98.5%</td>
<td>364</td>
<td>21.64</td>
</tr>
</tbody>
</table>

Table 1. The results of a 200-character test reveal the success rates of these three variations of On-Off Keying, compared to a fourth scheme, 16-pitch keying. From these success rates, along with the time each scheme needs to send 8 data bits (plus one start bit, for the OOK schemes), the approximate throughput of each scheme can be found.

Choice of final design

Ultimately, the specific OOK protocol decided upon was the one with the highest throughput and the lowest error rate: an OOK with 28-ms bits (instead of 15 ms) and with an (8,12) Hamming code.

Comparison to maximum theoretical throughput

Using the Shannon-Hartley theorem, discussed previously in the Theory section, the maximum theoretical throughput of the communication channel can be approximated.
Testing with the oscilloscope has found that the actual audio bandwidth appears to be between 575 and 2400 Hz (see figures 30 and 31). This results in a bandwidth of approximately 1825 Hz.

![Figure 30. Oscilloscope shot of the lowest stable, clear frequency achieved on a 2G voice call: 575 Hz.](image1)

![Figure 31. Oscilloscope shot of the highest stable, clear frequency achieved on a 2G voice call: 2400 Hz.](image2)

The SNR can be found by observing noise seen in the audio output of the 2G voice call, shown in figure 32. The maximum signal appears to be 1500 mV, while the maximum noise appears to be around 187 mV. This results in an SNR of approximately 8.

![Figure 32. An oscilloscope shot of the received audio on a 2G voice call, to illustrate an approximate SNR of 8. The maximum signal, 1500 mV, is roughly 8 times the maximum noise of 187 mV.](image3)
Thus, we should expect that the maximum theoretical throughput, without the introduction of an error correction scheme, or accounting for gaps and ticks, is approximately:

\[ 1825 \times \log_2(1 + 8) = 5785 \text{ bits per second} \]

This suggests that the best throughput achieved (21.64 bps) is about 0.374% of the best theoretical throughput. Assuming the best theoretical throughput would also have an (8,12) Hamming code, along with a start bit, the maximum theoretical throughput changes. This means 13 bits are required for every 8 bits of data. The maximum throughput changes to:

\[ 1825 \times \log_2(1 + 8) \times \frac{8}{13} = 3560 \text{ bps} \]

This means the 21.64 bps achieved is 0.607% the maximum throughput achieved when using a start bit and an (8,12) Hamming code.

**Discussion**

*Success of Forward Error Correction*

The Hamming code appears to have drastically decreased the error rate (boosting the success rate from 57.5% to 98.5%). This may be due to the fact that the vast majority of errors in each 8-bit message were 1-bit errors. The (8,12) Hamming code can correct for up to 1 bit error. Thus, it would appear that, for the 28 ms OOK without FEC, 57.5% of bytes went through without error; 41% had one-bit errors; and 1.5% had more than one bit errors.
Applications

There are three major use cases for this modem:

- Computer—microcontroller communication (with a serial monitor or Terminal application on the computer)
- Microcontroller—microcontroller communication
- Human-microcontroller communication

![Figure 33. Block diagrams of the three use cases for this project: computer-to-microcontroller communication; microcontroller-to-microcontroller communication; and human-to-microcontroller communication.](image)

The human—microcontroller communication makes use of a special “Easter Egg” feature included in the MSP430’s code: whenever a frequency of approximately 510 Hz is received by the modem, it outputs the ASCII character ‘*’ on the Tx pin. This can be triggered by whistling at that frequency.
To display each use case, three different example applications were constructed for this project. To demonstrate computer—microcontroller communication, an Arduino-controlled car was built. When the Arduino receives the characters ‘w,’ ‘a,’ ‘d,’ ‘z,’ and ‘s,’ the car moves forward, left, right, backwards, and stops, respectively. The controlling computer outputs serial characters from Terminal via the USB-to-serial converter. These characters go into one modem board, which is plugged into a phone. The phone is on a call with a phone inside the robot, which is connected to a second modem board (the “remote modem”). The remote modem outputs data on its Tx pin to the Arduino, which controls the motors accordingly. See Appendix B for the Arduino code, for the Terminal commands, and for a link to a video demonstration of the serial-controlled robot.

![Figure 34. Demonstration of computer-to-microcontroller communication: a serial controlled robot. Note the phone, the Arduino microcontroller, and the blue cylindrical battery inside the robot.](image-url)
To demonstrate microcontroller—microcontroller communication, a system was built in which an LED display reveals information about the status of a (theoretically) far-away box. When the box is open, a light sensor inside the box sends a ‘HIGH’ voltage to an Arduino UNO; that Arduino outputs a ‘y’ character to an OOK modem; that modem is plugged into a Samsung phone; the Samsung phone is on a call with an iPhone, which acts as the “local” phone; the local phone is plugged into a local modem, which is plugged into another Arduino (specifically an Arduino Leonardo); when the local Arduino receives a ‘y,’ it blinks some red LEDs. Similarly, when the box is closed, the light sensor outputs a ‘LOW’ voltage, which ultimately sends an ‘n’ character to the local microcontroller, which stops the LED blinking and activates a blue LED (to denote that the box is closed). See Appendix C for the Arduino UNO code, the Arduino Leonardo code, and for a link to a video demonstration.
Figure 36. Demonstration of microcontroller-to-microcontroller communication: a remote light sensor (bottom picture, attached to the Arduino by yellow wires) and a “local” LED display (top, plugged into the Arduino) that describes the status of the box (open or closed; lights ON or lights OFF). Note that the phone, the Arduino microcontroller, and the blue cylindrical battery make the remote circuit entirely mobile.
To demonstrate human—microcontroller communication, a system was created in which a whistle at 510 Hz triggers a motor to open a box, revealing something of value, perhaps a key. Only one modem and one microcontroller is needed for this use case. A remote phone is plugged into a remote microcontroller, which uses a motor control board to control a stepper motor, which can open the box. When the remote modem “hears” a frequency of 510 Hz on the voice call, it sends the ASCII symbol ‘*’ to the Arduino, which triggers the Arduino to turn on the motor and open the box. The 510 Hz frequency is achieved by whistling. See Appendix D for the Arduino code and for a link to a video demonstration.
Figure 38. Demonstration of human-to-microcontroller communication: the motor turns on when the Arduino microcontroller receives a ‘*’ character from the OOK modem (on the white protoboard). The OOK modem outputs a ‘*’ character when the Samsung phone receives a tone at 510 Hz, achieved by whistling into the iPhone (in the foreground) at that frequency.

Figure 39. Block diagram of human-to-microcontroller communication: a whistle-controlled motor that opens a box.

There are many other potential uses for this project, such as:

- Real-time GPS tracking for an important shipment
- Real-time GPS tracking for a fleet of taxis, couriers, etc.
- Controlling an RC plane or car
- Mobile credit card reader
- Open / close electronic locks for shared spaces
Comparisons to other options

Ultimately, this modem should be used only for applications that require very long-distance, mobile communications; the 22 bps data rate is very slow compared to WiFi, Bluetooth, or RF communications. But, the plug-and-play nature of the design make it far easier to use than a CDMA or GSM board: it basically acts like a wire between two devices.

The error rate can be unpredictable, depending on the strength of the cell phone connection. Thus, this connection should not be used for projects that require a highly reliable throughput. It should only be used for low-bandwidth, non-time-critical projects, such as home automation, temperature sensors, or pinging for GPS data.

This modem is far cheaper than the CDMA/GSM boards that are currently available. The cost of the MSP430 ($2.79 from DigiKey), plus the printed circuit board ($12.50 for 3 boards from OSH Park), plus the other components (generic components such as the LM311 comparator, TLV2772 op-amp, 555 timer, resistors, capacitors, pushbutton, voltage regulator, and wires) brings the total cost per modem to approximately $9.50. This excludes the cost of a prepaid cell phone with a voice plan, which could be as low as $5/month on a family plan.
Conclusion

Outcomes

Several things were achieved in doing this project:

- A low-bandwidth, mobile, serial modem was created that uses cell phone voice calls as a communication channel
- The modem was created on a relatively small budget, and the cost of materials per device is less than $10.
- The effectiveness and value of the Hamming code was clearly demonstrated.
- The MSP430's UART module was shown to seamlessly integrate with serial communications to/from other devices, including the USB-to-serial converter, the Arduino Uno, and the Arduino Leonardo.
- The author of this project learned a great deal about different communication protocols; manufacturer specifications for phones; voltage regulators; the MSP430; designing circuits using many different IC's; oscilloscopes; designing PCB's and schematics in Eagle CAD; error correction schemes; troubleshooting; and much more.

Future work

A major limitation of the project, as it stands now, is the fact that phone calls do not answer themselves. In order for the connection to be made, something or somebody needs to accept the incoming call on the remote microcontroller. For a more universal application, a small enclosure can be made that uses a solenoid to push the “accept call” button on the phone. A piezo disc, taped to the cell phone, would act as a sensor to determine when the phone was vibrating. That sensor information would be sent to the MSP430, which would trigger the solenoid to press the “accept call” button when the phone vibrates.

Additionally, the serial buffer built-into the MSP430 code could use some improvement. As it stands now, the serial buffer stores up to 50 bytes, waiting to be sent by the OOK protocol, which takes 364 ms to send each byte (hence, a buffer is useful). If the modem
receives new bytes while in the process of sending a byte, the Rx buffer has some unknown impact on the audio output, resulting in corrupt data being sent. This needs to be investigated and fixed.

Additionally, the choice of modulation scheme and error-correction scheme could be further investigated and refined. Implementation problems may have been the cause of abandoning FSK (and instead choosing OOK), and FSK may ultimately be a better modulation scheme for this noisy channel. The reason the author suspects this is because dial-up modems have successfully used FSK for decades. Also, the Hamming code is a fairly simple error correction code, and more complex implementations may be better.

Finally, the mic-input circuit could be causing some kind of distortion or interference, leading to higher-than average error rates. The author of this project did not do significant research into audio line-input channels; the mic input is simply an oscillating voltage. Perhaps there are better, more complex circuits for inputting audio into a mic jack.

Acknowledgements

The author of this report would like to thank Professor Lynne Molter, the advisor for this project, who was very helpful with guidance, support, and troubleshooting throughout the semester; Professor Erik Cheever, Professor Alan Moser, and Guatem Mohan for additional troubleshooting advice; and Ed Jaoudi for hardware supplies and support.
Endnotes

1. For more information on the Arduino maker community, see <www.arduino.cc>
3. ibid.
4. The JMT Sim900 Gprs GSM Shield Development Board Module sells on Amazon for $68.21 at the time of this writing, Amazon product ID number B00C19VMAK
6. Many forum posts about troubleshooting with confusing GSM and CDMA board interfaces exist, including this one: http://forum.arduino.cc/index.php?topic=219829.0
9. The 20-pin DIP package of the MSP430G2553 can be purchased on DigiKey.com for $2.79 at the time of this writing.
15. For the derivation of the formulas used, see “555 Timer Oscillator (1kHz)” by jmorgan@ehelion.com, January 9, 2012. <http://ehelion.net/projects/digitalclock/555timer.html>
   The formulas used were, roughly:
   \[ R2 = \frac{.69}{(f \ast c1)} \]
   \[ R1 = R2/10, \text{ for a } 50\% \text{ duty cycle.} \]
   \( C1 \) is chosen by the circuit designer; 0.01 \( \text{uF} \) was chosen for this project.
17. Printed circuit boards are available from: <www.OSHPark.com>. Pricing information and design rules are available on their website.
Appendix A. MSP430 code for the OOK modem.

/*
 * ======== Standard MSP430 includes ========
 */
#include <msp430.h>

//variables for demodulating
unsigned int outputByte = 0x00;
unsigned int newChar = 0x00;
unsigned int lastTime = 0;
unsigned int timeGap = 0;
unsigned int periodRead = 0;
unsigned int isReadingByte = 0;
unsigned int readHits[12];
unsigned int readBits[12] = {0,0,0,0,0,0,0,0,0,0,0,0};
unsigned int paritySumsReading[4] = {0,0,0,0};
unsigned int sumOfParityErrors = 0;
unsigned int readLoopEnterer = 0;
unsigned int timeofByteReadStart = 0;
unsigned int timeSinceByteReadStart;
unsigned int isSendingByte = 0;
unsigned int timeofByteSendStart = 0;
unsigned int timeSinceByteSendStart = 0;
unsigned int sentBits = 0;
unsigned char RXvalue = 0;
unsigned char currentByteSending = 0;
unsigned int HammingCodeToSend[12] = {0,0,0,0,0,0,0,0,0,0,0,0};
unsigned int paritySumsSending[4] = {0,0,0,0};
unsigned int recByteBuffer[50];
unsigned int recByteBufferPos = 0;
unsigned int posOfByteToBeSent = 0;
unsigned int byteSenderBusy = 0;

//variables for debugging
unsigned int periodReadStorage[5];
unsigned int i = 0;

/*
 * ======== Grace related includes ========
 */
#include <ti/mcu/msp430/Grace.h>

/*
 * ======== main ========
 */
int main(void)
{
    Grace_init(); // Activate Grace-generated configuration
```c
while (1) {
    timeSinceByteReadStart = (TAOR - timeofByteReadStart);
    timeSinceByteSendStart = (TAOR - timeofByteSendStart);

    //Whenever there's a byte to send:
    if (isSendingByte == 1) {
        if ((sentBits == 0) && (timeSinceByteSendStart < 3500)) {
            P10UT |= BIT3;
            P20UT = currentByteSending;
            currentByteSending = recByteBuffer[posOfByteToBeSent];
            posOfByteToBeSent = posOfByteToBeSent + 1;
            if (posOfByteToBeSent == 50) posOfByteToBeSent = 0;

            // Turn the received byte into a Hamming symbol
            // First assign the information bits to their correct positions

            if ((currentByteSending & BIT0) > 0) { HammingCodeToSend[2]=1; }
            if ((currentByteSending & BIT1) > 0) { HammingCodeToSend[4]=1; }
            if ((currentByteSending & BIT2) > 0) { HammingCodeToSend[5]=1; }
            if ((currentByteSending & BIT3) > 0) { HammingCodeToSend[6]=1; }
            if ((currentByteSending & BIT4) > 0) { HammingCodeToSend[8]=1; }
            if ((currentByteSending & BIT5) > 0) { HammingCodeToSend[9]=1; }
            if ((currentByteSending & BIT6) > 0) { HammingCodeToSend[10]=1; }
            if ((currentByteSending & BIT7) > 0) { HammingCodeToSend[11]=1; }

            // Now determine the parity bits in positions 0, 1, 3, and 7.

            if (((paritySumsSending[0] == 1) || (paritySumsSending[0] == 3) ||
                 (paritySumsSending[0] == 5)) { HammingCodeToSend[0]=1; })
                 (paritySumsSending[1] == 5)) { HammingCodeToSend[1]=1; })
```

paritySumsSending[0]=0;
paritySumsSending[1]=0;
paritySumsSending[2]=0;
paritySumsSending[3]=0;
sentBits = 1;
}
else if ((sentBits == 1) && (timeSinceByteSendStart > 3500) &&
(timeSinceByteSendStart < 7000)) {
    sentBits = 2;
    if (HammingCodeToSend[0]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 2) && (timeSinceByteSendStart > 7000) &&
(timeSinceByteSendStart < 10500)) {
    sentBits = 3;
    if (HammingCodeToSend[1]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 3) && (timeSinceByteSendStart > 10500) &&
(timeSinceByteSendStart < 14000)) {
    sentBits = 4;
    if (HammingCodeToSend[2]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 4) && (timeSinceByteSendStart > 14000) &&
(timeSinceByteSendStart < 17500)) {
    sentBits = 5;
    if (HammingCodeToSend[3]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 5) && (timeSinceByteSendStart > 17500) &&
(timeSinceByteSendStart < 21000)) {
    sentBits = 6;
    if (HammingCodeToSend[4]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 6) && (timeSinceByteSendStart > 21000) &&
(timeSinceByteSendStart < 24500)) {
    sentBits = 7;
    if (HammingCodeToSend[5]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 7) && (timeSinceByteSendStart > 24500) &&
(timeSinceByteSendStart < 28000)) {
    sentBits = 8;
    if (HammingCodeToSend[6]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 8) && (timeSinceByteSendStart > 28000) &&
(timeSinceByteSendStart < 31500)) {
    sentBits = 9;
    if (HammingCodeToSend[7]==1){P1OUT |= BIT3;}
    else{P1OUT &= ~BIT3;}
}
else if ((sentBits == 9) && (timeSinceByteSendStart > 31500) && (timeSinceByteSendStart < 35000)) {
    sentBits = 10;
    if (HammingCodeToSend[8] == 1) {P1OUT |= BIT3;}
    else {P1OUT &= ~BIT3;}
}
else if ((sentBits == 10) && (timeSinceByteSendStart > 35000) && (timeSinceByteSendStart < 38500)) {
    sentBits = 11;
    if (HammingCodeToSend[9] == 1) {P1OUT |= BIT3;}
    else {P1OUT &= ~BIT3;}
}
else if ((sentBits == 11) && (timeSinceByteSendStart > 38500) && (timeSinceByteSendStart < 42000)) {
    sentBits = 12;
    if (HammingCodeToSend[10] == 1) {P1OUT |= BIT3;}
    else {P1OUT &= ~BIT3;}
}
else if ((sentBits == 12) && (timeSinceByteSendStart > 42000) && (timeSinceByteSendStart < 45500)) {
    sentBits = 13;
    if (HammingCodeToSend[11] == 1) {P1OUT |= BIT3;}
    else {P1OUT &= ~BIT3;}
}
else if (timeSinceByteSendStart > 45500) {
    P1OUT &= ~BIT3;
    if (recByteBufferPos == posOfByteToBeSent) {
        isSendingByte = 0;
    }
    sentBits = 0;

    HammingCodeToSend[0] = 0;
    HammingCodeToSend[1] = 0;
    HammingCodeToSend[2] = 0;
    HammingCodeToSend[3] = 0;
    HammingCodeToSend[4] = 0;
    HammingCodeToSend[5] = 0;
    HammingCodeToSend[6] = 0;
    HammingCodeToSend[7] = 0;
    HammingCodeToSend[8] = 0;
    HammingCodeToSend[9] = 0;
    HammingCodeToSend[10] = 0;

    byteSenderBusy = 0;
    P2OUT = 0;

}
// Every time there's a freq within range:
if (readLoopEnterer==1){
    readLoopEnterer=0;
    // For debugging purposes:
    periodReadStorage[i] = periodRead;
    i = i + 1;
    if (i == 5){
        i = 0;
    }

    // This is a special case, for the "whistle" functionality
    // If someone whistles a C5 note, it will output the byte 2A, the ASCII symbol *
    if (((periodReadStorage[0]>115) && (periodReadStorage[0]<125) &&
         (periodReadStorage[1]>115) && (periodReadStorage[1]<125) &&
        isReadingByte = 0;
        readLoopEnterer=0;
        P1OUT ^= BIT4;

        readHits[0] = 0; readHits[1] = 0; readHits[2] = 0; readHits[3] = 0;
        outputByte = 0x2A; // Special case outputs the ASCII symbol '}',
                       // an arbitrarily chosen character
        P2OUT = outputByte;
        UCA0TXBUF = outputByte;
        while(UCA0STAT & UCBUSY);
    }

    // If a gap is read, isReadingByte = 1
    // Every 15 ms, increase bitCount
    // Every periodRead, add 1 to bitCount register
    // If a space in a bitCount register is greater than 5, its a 0. Otherwise its 1.
    if (isReadingByte == 0) {
        isReadingByte = 1;
        timeofByteReadStart = TA0R;
        timeSinceByteReadStart = 0;
        P1OUT ^= BIT4;
    }

    if (((isReadingByte == 1) && (timeSinceByteReadStart > 3500) &&
         (timeSinceByteReadStart < 7000)) { 
        readHits[0] = readHits[0] + 1;
    } else if (((isReadingByte == 1) && (timeSinceByteReadStart > 7000) &&
                 (timeSinceByteReadStart < 10500)) { 
        readHits[1] = readHits[1] + 1;
    } else if (((isReadingByte == 1) && (timeSinceByteReadStart > 10500) &&
                 (timeSinceByteReadStart < 14000)) { 


else if ((isReadingByte == 1) && (timeSinceByteReadStart > 14000) && (timeSinceByteReadStart < 17500)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 17500) && (timeSinceByteReadStart < 21000)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 21000) && (timeSinceByteReadStart < 24500)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 24500) && (timeSinceByteReadStart < 28000)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 28000) && (timeSinceByteReadStart < 31500)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 31500) && (timeSinceByteReadStart < 35000)) {
    readHits[8] = readHits[8] + 1;
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 35000) && (timeSinceByteReadStart < 38500)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 38500) && (timeSinceByteReadStart < 42000)) {
}

else if ((isReadingByte == 1) && (timeSinceByteReadStart > 42000) && (timeSinceByteReadStart < 45500)) {
}
}

if ((isReadingByte == 1) && (timeSinceByteReadStart > 45500)) {
    isReadingByte = 0;
    readLoopEnterer=0;
    PLOUT ^= BIT4;

    if (readHits[0] > 15) {readBits[0] = 1;}
    if (readHits[1] > 15) {readBits[1] = 1;}
    if (readHits[8] > 15) {readBits[8] = 1;}
    if (readHits[9] > 15) {readBits[9] = 1;}
    if (readHits[10] > 15) {readBits[10] = 1;}
}
// Decode Hamming symbol into outputByte here

// Now determine the EXPECTED parity bits values in positions 0, 1, 3, and 7.

                     readBits[6] + readBits[11];
                     readBits[10] + readBits[11];

if ((paritySumsReading[0] == 1) || (paritySumsReading[0] == 3) ||
    (paritySumsReading[0] == 5)) {
    sumOfParityErrors = sumOfParityErrors + 1;
}
if ((paritySumsReading[1] == 1) || (paritySumsReading[1] == 3) ||
    (paritySumsReading[1] == 5)) {
    sumOfParityErrors = sumOfParityErrors + 2;
}
    (paritySumsReading[2] == 5)) {
    sumOfParityErrors = sumOfParityErrors + 4;
}
    (paritySumsReading[3] == 5)) {
    sumOfParityErrors = sumOfParityErrors + 8;
}

// Now (sumOfParityErrors - 1) = the position of any single-bit errors.
// Now flip any one-bit errors.
if ((sumOfParityErrors > 0) && (sumOfParityErrors < 12)) {
    sumOfParityErrors = sumOfParityErrors - 1;
    if (readBits[sumOfParityErrors] == 0) {readBits[sumOfParityErrors] = 1;}
    else {
        readBits[sumOfParityErrors] = 0;
    }
}

paritySumsReading[0] = 0;
paritySumsReading[1] = 0;
paritySumsReading[2] = 0;
paritySumsReading[3] = 0;
sumOfParityErrors = 0;

outputByte = readBits[2]*1 + readBits[4]*2 + readBits[5]*4 + readBits[6]*8 +
             readBits[8]*16 + readBits[9]*32 + readBits[10]*64 + readBits[11]*128;

readBits[0] = 0;
readBits[1] = 0;
readBits[2] = 0;
readBits[3] = 0;
readBits[4] = 0;
readBits[5] = 0;
readBits[6] = 0;
readBits[7] = 0;
readBits[8] = 0;
readBits[9] = 0;
readBits[10] = 0;
readBits[11] = 0;
readHits[0] = 0;
readHits[1] = 0;
readHits[2] = 0;
readHits[3] = 0;
readHits[4] = 0;
readHits[5] = 0;
readHits[6] = 0;
readHits[7] = 0;
readHits[8] = 0;
readHits[9] = 0;
readHits[10] = 0;
readHits[11] = 0;

P2OUT = outputByte;
UCA0TXBUF = outputByte;
while(UCA0STAT & UCBUSY);
Appendix B. Serial-controlled robot.

A video demonstration of this robot can be found at the following URL:

www.youtube.com/watch?v=VELVpFxUDqU

The USB-to-serial converter was purchased from Adafruit, and requires drivers that can be downloaded from the Adafruit website. Once the converter is plugged into a computer, communication with the converter must be initialized at 4800 baud. On a MacBook Pro, the Terminal command might look similar to this (but with a different USB port name):

Chriss-MacBook-Pro:~ chris$ screen /dev/cu.NoZAP-PL2303-00002014 4800

Where “NoZAP-PL2303” is the name of the converter’s drivers, “00002014” is the USB port name, and “4800” sets the baud rate.

Code for the Arduino Leonardo that controls the robot:

```cpp
#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"

// Create the motor shield object with the default I2C address
Adafruit_MotorShield AFMS = Adafruit_MotorShield();

// Select 2 DC motors on ports 3 and 4
Adafruit_DCMotor *LeftMotor = AFMS.getMotor(3);
Adafruit_DCMotor *RightMotor = AFMS.getMotor(4);

byte incomingByte;  // a variable to read incoming Serial1 data into

void setup() {     // a variable to read incoming Serial1 data into
    Serial1.begin(4800);  // set up Serial communication at 4800 baud

    AFMS.begin();  // initialize motor board

    // Set the speed to start, from 0 (off) to 255 (max speed)
    LeftMotor->setSpeed(120);
    RightMotor->setSpeed(120);
    LeftMotor->run(FORWARD);
    RightMotor->run(FORWARD);
    // turn on motor
    LeftMotor->run(RELEASE);
    RightMotor->run(RELEASE);
}
```
void loop() {
    uint8_t i;

    if (Serial1.available() > 0) {
        // Every time a new byte is received:
        incomingByte = Serial1.read();

        if (incomingByte == 'w') {
            // move robot forward
            LeftMotor->run(FORWARD);
            RightMotor->run(FORWARD);
            LeftMotor->setSpeed(255);
            RightMotor->setSpeed(255);
            delay(50);
        }

        if (incomingByte == 'z') {
            // move robot backward
            LeftMotor->run(BACKWARD);
            RightMotor->run(BACKWARD);
            LeftMotor->setSpeed(255);
            RightMotor->setSpeed(255);
            delay(50);
        }

        if (incomingByte == 'a') {
            // turn robot left
            LeftMotor->run(BACKWARD);
            RightMotor->run(FORWARD);
            LeftMotor->setSpeed(255);
            RightMotor->setSpeed(255);
            delay(50);
        }

        if (incomingByte == 'd') {
            // turn robot right
            LeftMotor->run(FORWARD);
            RightMotor->run(BACKWARD);
            LeftMotor->setSpeed(255);
            RightMotor->setSpeed(255);
            delay(50);
        }

        if (incomingByte == 's') {
            // stop robot
            LeftMotor->run(RELEASE);
            RightMotor->run(RELEASE);
            delay(50);
        }
    }
}
Appendix C. Remote light sensor and local LED display.

Arduino code for the remote light sensor:

```cpp
void setup() {
    Serial.begin(4800);
    pinMode(3, INPUT);  // The light sensor is plugged into pin 3.
}
void loop() {
    if (digitalRead(3) == 1) {
        Serial.print('y');
        delay(600);  // output serial character ‘y’
    }
    if (digitalRead(3) == 0) {
        Serial.print('n');
        delay(600);  // output serial character ‘n’
    }
    delay(100);
}
```

Arduino code for the local LED display:

```cpp
byte incomingByte;
int leftSideOn = 0;
int enterBlink = 0;

void setup() {
    Serial1.begin(4800);
    pinMode(6, OUTPUT);
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);
    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(12, OUTPUT);
    pinMode(13, OUTPUT);  // LEDs plugged into pins 6–12 are
    pinMode(13, OUTPUT);  // LED in pin 13 is blue, is ON
}
void loop() {
    if (Serial1.available() > 0) {  // Every time a new byte is received,
        incomingByte = Serial1.read();  // enter loop.
    }
    if (incomingByte == 'n') {  // If no light detected, receive ‘n’.
        enterBlink = 0;  // No blinking LEDs; set enterBlink to 0.
        digitalWrite(13, HIGH);  // Blue LED is on, all others are off.
        digitalWrite(9, LOW), digitalWrite(10, LOW), digitalWrite(11, LOW),
        digitalWrite(12, LOW);
        digitalWrite(6, LOW), digitalWrite(7, LOW), digitalWrite(8, LOW);
        delay(50);
    }
```
if (incomingByte == 'y') {  // If the new character is a 'y',
enterBlink = 1;          // enter the blink loop.
}

if (enterBlink == 1){  // In the 'blink loop,' the left half
    if (leftSideOn == 1){ // and the right half of the LEDs flip
        leftSideOn=0;    // on and off every 200 ms.
        digitalWrite(13, LOW);
        digitalWrite(6, HIGH),digitalWrite(7, HIGH),digitalWrite(8, HIGH);
        digitalWrite(9, LOW),digitalWrite(10, LOW),digitalWrite(11, LOW),digitalWrite(12, LOW);
        delay(200);
    }
}

else if (leftSideOn == 0){
    leftSideOn=1;
    digitalWrite(13, LOW);
    digitalWrite(6, LOW),digitalWrite(7, LOW),digitalWrite(8, LOW);
    digitalWrite(9, HIGH),digitalWrite(10, HIGH),digitalWrite(11, HIGH),digitalWrite(12, HIGH);
    delay(200);
}

// The blink loop is exited when an 'n' is received
// and enterBlink is reset to 0.
Appendix D. Whistle-controlled motor.

A video demonstration, in which two wrong frequencies are whistled before the correct 510 Hz frequency is whistled, can be found at: www.youtube.com/watch?v=skQY4kGdnqY

Arduino code, in which the motor, plugged into an Adafruit motor control board, is triggered by the receipt of the ASCII symbol ‘*’:

```c
#include <Wire.h>
#include <Adafruit_MotorShield.h> // Load the Adafruit MotorShield library
#include "utility/Adafruit_PWMServoDriver.h"

// Create the motor shield object with the default I2C address
Adafruit_MotorShield AFMS = Adafruit_MotorShield();

// Connect a stepper motor with 200 steps per revolution (1.8 degree)
// to motor port #2 (M3 and M4)
Adafruit_StepperMotor *myMotor = AFMS.getStepper(200, 2);

byte incomingByte; // a variable to read incoming Serial1 data into

void setup() {
  Serial1.begin(4800); // set up Serial communication at 4800 baud
  AFMS.begin(); // initialize motor board
  myMotor->setSpeed(10); // set the speed of the motor to 10 rpm
}

void loop() {
  if (Serial1.available() > 0) { // Enter when any byte is received.
    incomingByte = Serial1.read(); // Store received byte as incomingByte.
    if (incomingByte == '*') {
      // If the received byte is the char *, turn on motor for 500 steps,
      // which is 2.5 revolutions, which takes 15 seconds.
      myMotor->step(500, FORWARD, DOUBLE);
    }
  }
}
```