VISIBLE LIGHT COMMUNICATION SYSTEMS

A Major Qualifying Project Report:

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Casey Barney

Alexander Dich

Dennis Koufos

Date: March 28, 2014

Approved: Professor Lifeng Lai, Major Advisor

Abstract

With the radio frequency spectrum becoming crowded, an alternative means to wireless communication is necessary to accommodate the exponentially increasing wireless traffic demand. Visible light communication systems provide an alternative to the current standards of wireless transfer of information, using light from LEDs as the communication medium. In these systems, light-emitting diodes blink at a rapid rate such that the human eye will not notice the change in light intensity, but a sensitive photodiode can detect the on-off behavior and decode the information embedded within it.

This project first analyzes various issues with current wireless communication systems, and discusses how visible light communications can resolve these issues. Then, the design and implementation processes of the visible light communication system are described in detail, including a value analysis of the parts used to build the prototype, as well as the necessary steps to wire and/or code each functional block of the design. The achieved results of the system, including transmission distance and speed, as well as quality of transmission and type of data are discussed.

Acknowledgements

We would like to thank our advisor, Professor Lifeng Lai, for all of the help he has given us throughout the course of this project. He has given us constant feedback on our work, as well as providing helpful suggestions to fix issues that arise.

Table of Contents

Table of Figures

Chapter 1: Introduction

With the exponentially increasing data demand but limited available radio spectrum, alternatives will be necessary to accommodate the needs of wire-free communication systems. This chapter will illustrate the problems of current wireless communication systems and alternatives to these systems, as well as motivations and possible applications for visible light communications.

1.1 Motivations

As societal dependence upon wireless systems continues to grow, wireless technology needs to expand to meet the demand. Phones, laptops, and global positioning systems are all devices that implement certain forms of wireless communication to send information to another location. However, the availability of current forms of wireless is very limited, and it is not necessarily safe to implement wireless radio, making it necessary to explore other alternatives to wireless communication to allow continued expansion upon communication systems and to ensure safe use.

Figure 1 illustrates the frequency allocations of the radio spectrum in the United States.

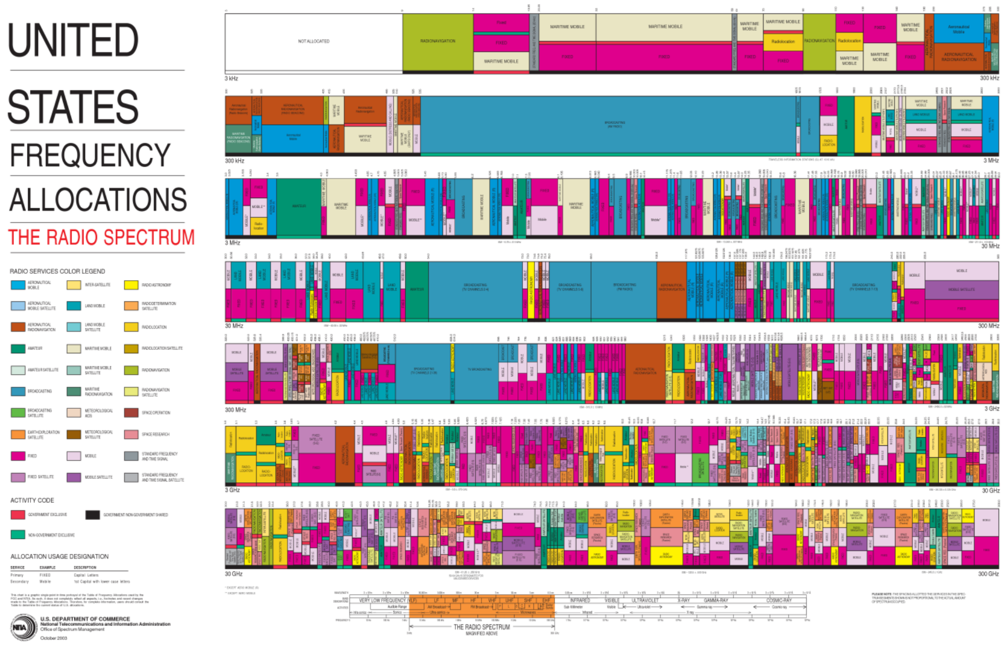


Figure 1: US Frequency Allocations
Source: http://upload.wikimedia.org/wikipedia/commons/6/65/United_States_Frequency_Allocations_Chart_2003_-_The_Radio_Spectrum.png

The Federal Communications Commission (FCC) regulates many wireless applications in the US, including radio, television, wire, satellite, and cable [1]. Each application is given a frequency band in which it is allowed to operate to allow efficient use of the available frequency spectrum. From Figure 1, it is quite evident that this spectrum is very crowded. At the same time, there is a huge growth in demand in the limited radio frequency spectrum. From Figure 2, predictions estimated that as soon as even 2013, the US could potentially be in a spectrum deficit. Therefore, a more efficient way of utilizing radio frequency is necessary.

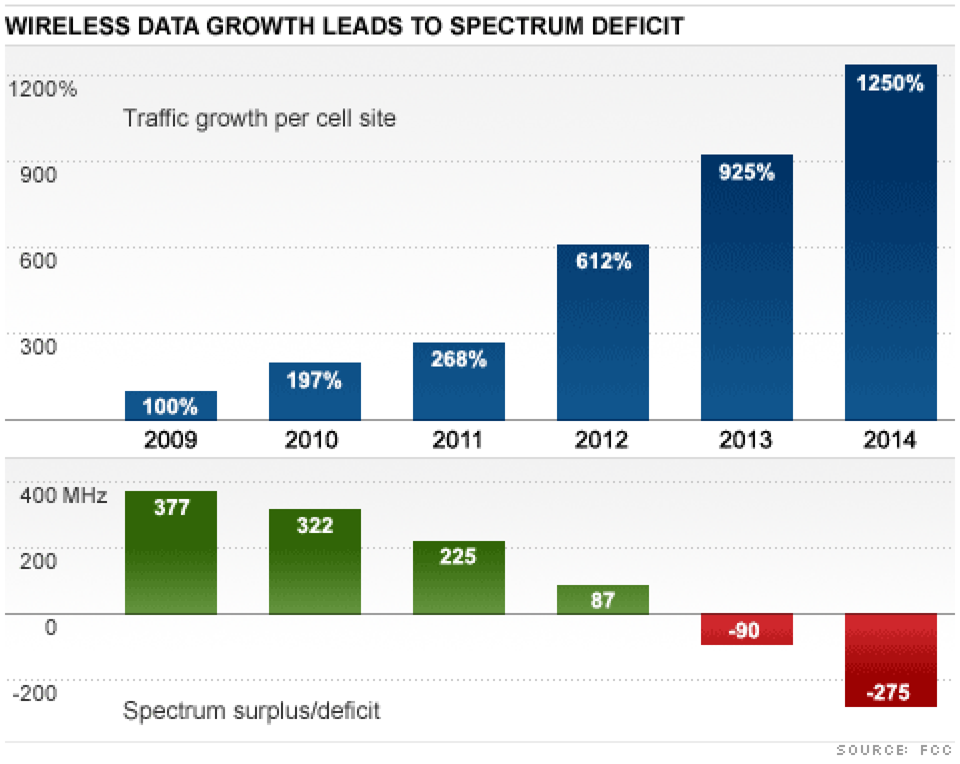


Figure 2: Wireless Data Growth Predictions
Source: http://money.cnn.com/2012/02/21/technology/spectrum_crunch/index.htm

In addition to the crowding of the frequency spectrum, interference is also a concern for many existing wireless systems. Any simultaneous use of a frequency band will cause interference due to the electromagnetic nature of most wireless devices, which could result in incorrect or loss of information for those users involved. A prime example of this is the use of mobile devices on planes, which directly affects safety. The Federal Aviation Administration (FAA) argues that these wireless devices cause interference to the aircraft's navigation and communication systems, and the Federal Communications Commission (FCC) argues that mobile devices used on aircrafts will disrupt cellular towers on the ground. Other studies indicate that use of mobile electronics on aircrafts can exceed permissible emission levels for safety with regard to some avionics [2]. Regardless of the reason, it is clear that it is not feasible to use wireless devices in certain environments in which safety, data integrity, and accuracy are highly important.

VLC systems have more flexibility and integrity than other communication systems in many regards. Since the medium for transmission in VLC systems is visible light and not RF waves that can penetrate walls, the issue of security is inherently solved because light cannot leave the room, containing data and information in one location. There is no way to retrieve and access the information unless a user is in a direct path of the light being used to transmit the data. In addition, LEDs are highly efficient and becoming more durable, adding to the integrity of these systems.

1.2 Alternatives in Progress

Currently, several alternatives to radio frequency communications exist. For example, there are cognitive radio, which utilizes radios programmed to adapt to surroundings by constantly analyzing the frequency spectrum to determine how the surrounding spectrum is currently being utilized, and laser communication systems, which transmits data through free space by shooting a laser with wavelengths close to the infrared spectrum to a receiver.

1.2.1 Cognitive Radio

Given that one major issue in wireless communication is the crowded frequency spectrum, many engineers spend their time and effort focusing on determining solutions for this issue. Since there is limited access to the frequency spectrum, these engineers are focusing on options that could optimize the spectrum. By optimizing the frequency spectrum’s usage, it would be possible to provide all end users a portion of the spectrum. As the current trend continues, devices that normally would not be able to wirelessly communicate, such as lamps or temperature sensors, will be connected to some type of wireless network. This will increase the number of end users and further crowd the frequency spectrum.

One area that engineers are focusing on to optimize the frequency spectrum involves cognitive radios. The difference between a cognitive radio and a typical radio system is that a cognitive radio is programmed to adapt to its surroundings. A cognitive radio is constantly analyzing the frequency spectrum to determine how the surrounding spectrum is being used. The system could potentially monitor the entire frequency spectrum, but that would require an antenna that has a large bandwidth. Since most antennas operate at a range of frequencies, cognitive radios will monitor that specific bandwidth and determine how it is occupied. Once the radio has determined how the spectrum is being occupied, it will choose non-occupied frequencies to transmit its information. While it is transmitting information, it continues to monitor the spectrum to determine whether other signals are attempting to access the same frequencies. If there are other signals, the radio will stop transmitting and switch to another unused frequency slot. This whole process is called Dynamic Spectrum Access and is a vital part of how a cognitive radio functions.

The idea of using cognitive radios for optimizing the use of the frequency spectrum will require the systems to focus on more than one frequency band. Since a majority of these bands have been dedicated to certain organizations, those organizations have priority or full control over the frequencies. Out of all the divided frequency bands, researchers are looking at the television bands. There are multiple television bands ranging between 54-72 MHz, 76-88 MHz, 174-216 MHz, 512-608 MHz, and 614-698 MHz which are used to provide certain television signals to the set top boxes in homes. Each band's bandwidth is then further divided to allow all channels to have access to transmission. The reason the television band is the band of focus is how the spectrum is being used. At the Illinois Institute of Technology in Chicago, IL, a team of researchers monitored the frequency spectrum over a span of three years to determine how each frequency band was occupied. The occupancy was measured by monitoring the frequency band's spectral density to a threshold. The following figure represents the occupancies of certain frequency bands [3].

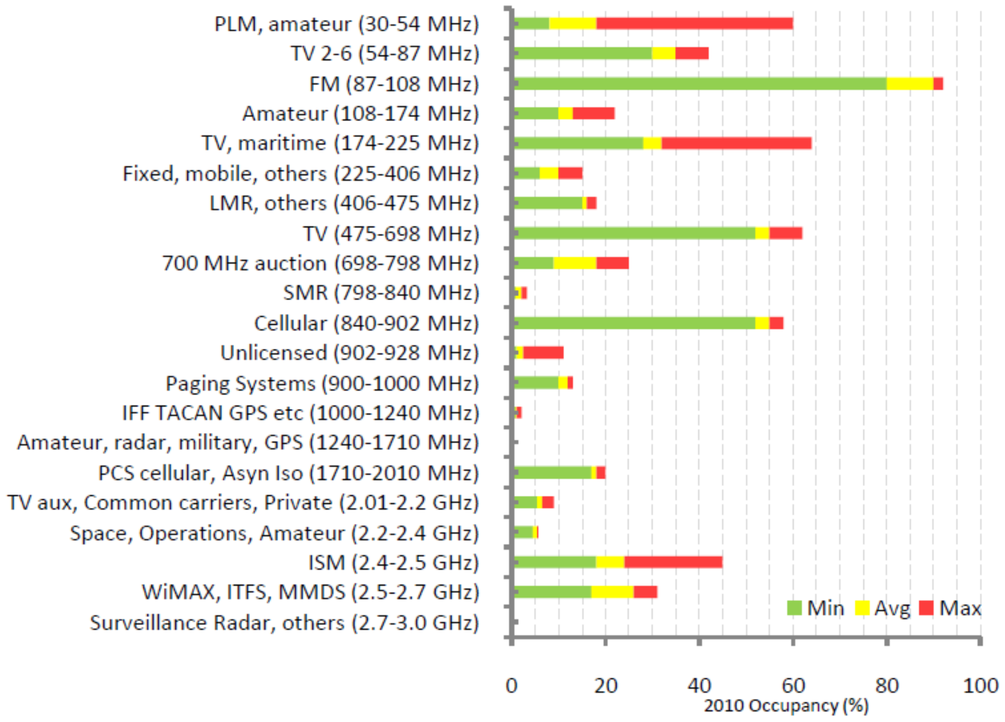


Figure 3: Estimated occupancy for 2010
Source: http://www.ece.iit.edu/~taher/dyspan11.pdf

From the data collected in 2010, the researchers determined that the three television bands, TV2-6 (54-87MHz), TV maritime (174-225MHz), and TV (475-698MHz), had an average occupancy of 32%, 30%, and 50% respectively. Given that the largest television frequency band was only occupied half the time, researchers believe that the band can be shared with other transmissions. While observing the entire spectrum, the average occupancy was measured at 14%. This low number suggests how inefficiently the radio frequency spectrum is used. With cognitive radios, other signals that are not television signals can monitor the band and use it if that specific area is not being used. As mentioned earlier, the system will have to monitor any surrounding systems to see if signals are attempting to access this specific frequency. In this case, if a radio is operating at a specific frequency that corresponds to a television channel and the channel needs to transmit, then the radio will have to stop transmitting because the television signal has a higher priority [4][5][6].

1.2.2 Laser Communication

Laser communication systems utilize wireless connections through the atmosphere, transmitting data through free space by shooting a laser. This form of wireless communication can be effective because it is not regulated by the government as it operates in a near infrared spectrum, hence avoiding any additional overcrowding of the spectrum with this form of communication. This allows for quick establishment of communication links, as it does not need to go through the various regulatory processes that would be necessary to set up an RF system. The system can work for a distance of up to 6 km with bitrates up to 1.25 Gbps[7]. The system also uses relatively low power and has a low noise ratio. It is also secure, as any sort of eavesdropping on the data transmission will require viewing directly into the transmitter path, causing an interruption in transmission.

Unfortunately, the system requires a line-of-sight path from the transmitter to receiver. This renders the two functional blocks relatively immobile. If the path is not calibrated precisely, the laser could miss the receiver by a large distance, resulting in no data transmission. In addition, although invisible to the naked eye, the lasers used could result in damage to one's eye if there is an extended exposure to the laser [8][9].

1.3 Visible Light Communications

The focus of this project will be Visible Light Communications (VLC). We aim to investigate this system by designing our own analog circuit to integrate with a computer, and then sending some form of data using visible light LEDs from a transmitter, and decoding it with a receiver.

Information will be converted into bits through some coding scheme by a microcontroller and will be transmitted with blinking LEDs. The blinking of these LEDs will not be visible to the human eye as they are blinking at a high frequency. Photodiodes on the receiving side will detect the fluctuation of the LEDs from the transmitter and will send signals to a microcontroller which is integrated with a computer to determine the originally transmitted message. The transmitting system will be powered from a wall outlet whereas the receiving system will be powered by batteries and the computer/microcontroller combination.

1.3.1 Advantages

Visible light should be considered as the medium for wireless transmission because it has a few advantages over other standard wireless transmissions. The first reason to consider is visible light's frequency spectrum bandwidth, which ranges from 430 THz to 750 THz [11]. The bandwidth is much larger than the radio frequency bandwidth, which ranges from 3 kHz to 300 GHz [1]. With a larger bandwidth it is possible to accommodate more users and potentially achieve higher transfer rates because each user can be given a larger portion of the bandwidth to transfer information. If the communication system will be used in hospitals, the transmissions will not occur in the Industrial, Scientific, and Medical (ISM) band, therefore not interfering with medical devices. On top of having a higher bandwidth, the frequency spectrum has less regulation than the radio spectrum. With little regulation, the user will be able to choose any frequency to transfer information. If visible light communication systems become more popular, regulations could be placed on these forms of data transmission for the same reasons that they were placed for the radio spectrum.

The next major advantage that visible light systems have over other communication systems is its abundance. Light sources are everywhere, and can be more efficiently used by increasing its simultaneous functionality by transmitting data in addition to lighting an area. On typical work days, company buildings, restaurants, grocery stores, etc. will have lights on for at least the duration of hours of operation, of which could be used for visible light communications.

There are also a few drawbacks to visible light in standard situations that could potentially be used as advantages for a visible light communication system. Unlike radio waves, light cannot propagate through walls. Since light cannot propagate out of an enclosed room, the only way to access the information is if the receiver is in the same room; thus, no outside sources will be able to acquire the information. Therefore, light sources are more secure than radio waves because they are not broadcasted for external sources to receive.

Visible light was chosen for a variety of reasons, but primarily because it will not add to the cluttering of the radio frequency spectrum, which is heavily regulated by the FCC, and also because it will avoid the issue of interference in sensitive settings such as hospitals and airplanes. Figure 4 shows the wavelength range of visible light.

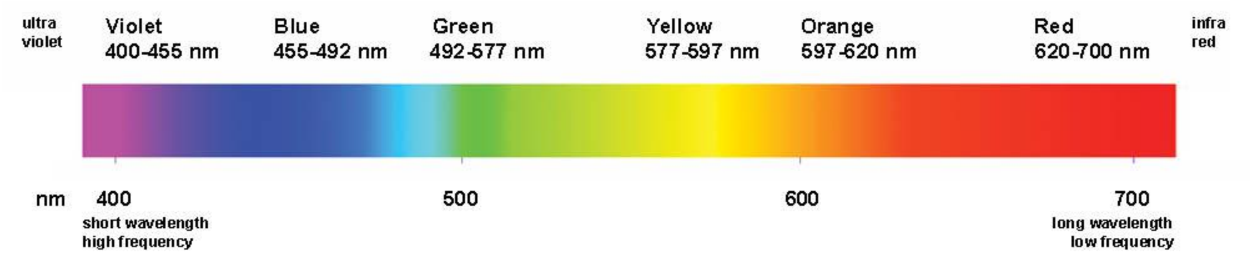


Figure 4: Visible Light Spectrum
Source: http://nextgenlite.com/images/VisibleLightSpectrumGradientForWeb.jpg

From these wavelengths, the frequency range can be calculated by the following equation:

$$f = \frac{c}{\lambda} \tag{1}$$

where f is the frequency, c is the speed of light, and λ is the wavelength. Thus, it can be shown that the range of frequencies for visible light is around 400-800 THz.

1.3.2 Disadvantages

Limitations and drawbacks that we have to consider include noise from ambient light and the line-of-sight of the system. If the intensity of ambient light is greater than that of the light from our system, the signal-to-noise ratio (SNR) is low, which will distort transmitted data. To compensate for this, the SNR will be maximized by setting thresholds on the microcontroller based on voltage signals produced by the ambient light in conjunction with the transmitter signal.

Also, the system will only be maximized when the LEDs are directly facing the sensor. If the angle is changed even slightly, the maximum range of the system will decrease significantly. The easiest solution is to ensure that the transmitter and receiver are facing directly at each other.

1.4 Potential Applications of Visible Light Communications

Lights in the visible spectrum are used everywhere, providing several opportunities to apply visible light communications. There are many applications in which data transfer via VLC systems could be useful including traffic lights, which could utilize systems to optimize traffic flow; television sets, which could supply a user with information on current show listings; and hospitals, which could utilize the systems for more secure transfer of data.

1.4.1 Traffic Lights

There are many modern applications that use visible light to portray information. Using a visible communication system in tandem with these devices can increase the devices' functionality. An example of a device that can benefit from a visible communication system is a traffic or stop light. In a busy intersection, traffic lights use visible lighting to maintain the flow of traffic. Because these lights are common in major cities, incorporating some sort of communication system in them to allow our society to stay connected and up to date with all sorts of information improves overall efficiency through multi-tasking. When dealing with traffic lights, a driver or pedestrian remains idle while waiting for their turn to proceed. The majority of the time, this time is simply wasted by remaining idle. If a visible light communication system was connected to a traffic light, the user could potentially use his/her phone or car head lights to connect to the traffic lights and retrieve some form of information. The information may be about local traffic, or even directions to a specific location. The system could even be used as a local connection to access the internet. By doing this, the user can have an alternative means of accessing data instead of his/her costly and limited 3G or 4G data connection.

While having an alternative connection point could be beneficial, it could promote drivers to use handsets while driving, which can be dangerous. If a driver would like to access information, it should be done in a manner that does not cause harm to or endanger the driver or other drivers. One way to do so is to incorporate a visible communication system in the vehicle and use the vehicle's head lights to send information. Along with this, a voice activated system could be implemented so the driver can access information hands-free.

While it may be possible to get data transmission over visible light, there are many scenarios that require consideration in order to ensure a reliable and useful system. Since traffic lights are all outdoors, natural light can become an issue and cause bad connections due to noise. One way to minimize the noise from the natural light is to use specific colored lighting to transmit information. By using a certain colored light, the photodiode used to retrieve the information can be designed to only recognize certain wavelengths and attenuate all others. Another issue that may arise is the number of users that the system can handle. One way to resolve this is to use multiple colors to transmit information. Since every visible color light has a different wavelength, they will operate at different frequencies and common communication principles can be used to minimize the interference between the different signals.

1.4.2 Television Application

Another piece of modern technology that uses visible light to portray information is a television. Unlike a traffic light, a television contains thousands of pixels that are constantly changing colors to project an image to its viewers. Because there are many individual LEDs in a television, it could be possible to allocate to a few of them the task of transmitting information through a visible light communication system. When a user is watching television, there is a possibility that the user may wish to see what else is airing on other channels. To do this with today's technology, the user will have to either constantly switch the channels to see shows that are currently airing on other channels or minimize what was being watched to bring up the TV guide. If the user has access to a smartphone or a computer, he/she could use that to look at the guide. Unfortunately, this requires internet access. Instead of using the internet connection, the smartphone or computer could also incorporate a visible communication system and retrieve the information from the television and display it on the second device, and not affect what is occurring on the television. Also, if the user is really intrigued by what he or she is currently watching but does not know what it is, they could use the communication system to transmit the program information to their other device.

One drawback to using a visible communication system on a television is the fact that a few pixels are dedicated to transmission and potentially could affect what is being displayed. To not disrupt the user experience, the LEDs must be placed somewhere that will not affect what is being displayed. One way to accomplish this is to place the LEDs away from the display, or use the LED to indicate that the television is ready to transmit the information. Similar to the traffic light scenario, the receiver will need to minimize the noise that may come from other light sources. This could be accomplished by filtering out all but a few light color frequencies.

1.4.3 Hospitals

Hospitals have many reasons to employ wireless technology. Applications of wireless technology in hospitals include updating information by wirelessly maintaining patient records, collecting data as a real-time handheld patient monitor to detect changes in a patient's condition, or even observing medical images via ultrasound. Table 1 lists proposed applications for the use of Bluetooth in hospitals:

APPLICATION	STATUS IN MARKET	DEVICE TYPE
Telemetry	Candidate	Embedded device
Heart-rate monitor	In trials	Embedded device
Ambulance crew device	Soon to be in trials	Embedded device
Ultrasound	Candidate	Embedded device
Infusion pump	Candidate	Embedded device
Glucometer	In trials	Embedded device
ECG/respiration beside monitor	In trials	Embedded device
Hearing aid programmer	In trials	Embedded device
Stethoscope	Candidate	Embedded device
Sleep monitor	Candidate	Embedded device
Epileptic brain monitor	Candidate	Embedded device
Defibrillator	Candidate	Embedded device
Handheld patient monitor	Candidate	Windows based
Data collection	Candidate	Windows based

Table 1: Proposed Bluetooth Applications in Hospitals
Source: http://www.mddionline.com/article/implementing-wireless-communication-hospital-environments-bluetooth-80211b-and-other-technol

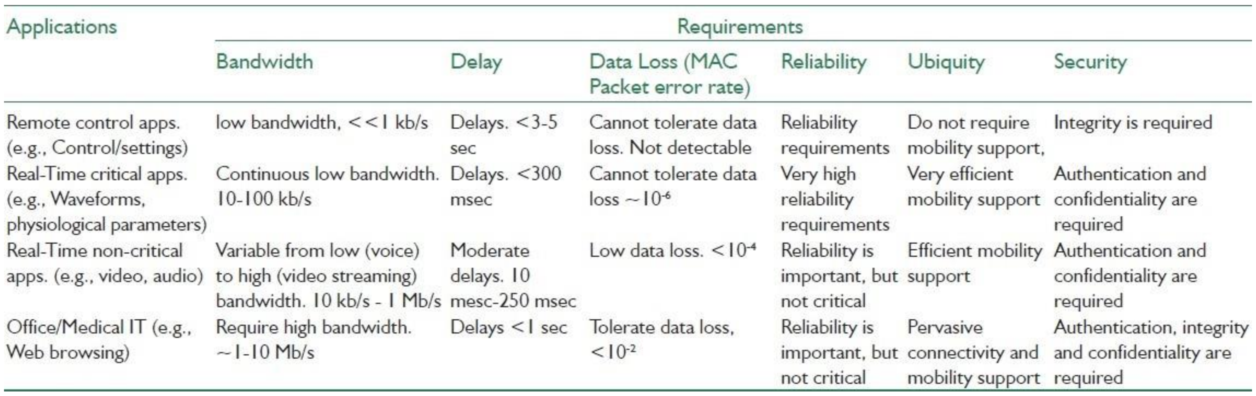
However, many concerns follow with the use of wireless technology in hospitals, and must be addressed when implementing a wireless communication system in such a sensitive environment.

Accuracy of information via wireless communication is imperative in a hospital setting. In real-time applications in which a patient's physiological conditions are monitored, data loss is intolerable with a packet error rate (PER) of less than 10^{-4}. For hospital office applications, slight data loss is tolerable, with a PER on the order of 10^{-2}, but is certainly still undesired.

Operational efficiency is necessary to ensure reliability and short delay time between two communicating devices. For real-time applications, the devices must be reliable and must have a delay of less than 300 milliseconds. For office-related applications, reliability is still important, but not critical, and delay time can be on the order of around 1 second.

Security is necessary in order to maintain confidentiality of patient records, and to ensure that only authorized personnel have access to the data being transferred wirelessly. This can be done through some authentication process with software or perhaps some sort of identification card.

Table 2 highlights some of these concerns and their required specifications:



Applications	Requirements					
	Bandwidth	Delay	Data Loss (MAC Packet error rate)	Reliability	Ubiquity	Security
Remote control apps. (e.g., Control/settings)	low bandwidth, <<1 kb/s	Delays. <3-5 sec	Cannot tolerate data loss. Not detectable	Reliability requirements	Do not require mobility support,	Integrity is required
Real-Time critical apps. (e.g., Waveforms, physiological parameters)	Continuous low bandwidth. 10-100 kb/s	Delays. <300 msec	Cannot tolerate data loss $\sim 10^{-6}$	Very high reliability requirements	Very efficient mobility support	Authentication and confidentiality are required
Real-Time non-critical apps. (e.g., video, audio)	Variable from low (voice) to high (video streaming) bandwidth. 10 kb/s - 1 Mb/s	Moderate delays. 10 mesc-250 msec	Low data loss. $<10^{-4}$	Reliability is important, but not critical	Efficient mobility support	Authentication and confidentiality are required
Office/Medical IT (e.g., Web browsing)	Require high bandwidth. ~1-10 Mb/s	Delays <1 sec	Tolerate data loss, $<10^{-2}$	Reliability is important, but not critical	Pervasive connectivity and mobility support	Authentication, integrity and confidentiality are required

Table 2: Applications and their Requirements
Source: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3592499/

Interference is perhaps the most significant concern in a hospital environment. Many medical devices are sensitive to waveform distortion, and any sort of electromagnetic interference between a wireless communication device and a medical instrument could cause an unexpected automatic shutdown or restart of the instrument.

Visible light communication systems do not allow for high mobility through obstacles, providing a relatively secure method of transferring information between a transmitter and receiver. Only those directly nodes directly facing one another will be able to obtain any information. In a hospital, there is much less regulation by the FCC which leads to much more bandwidth. These systems also have less interference with RF waves, providing a relatively reliable and accurate data transfer without risking interfering with medical instruments. Hospitals typically use fluorescent lighting, which have a visible white light wavelength of approximately 400-700 nm [10]. With such a range of wavelengths, visible light communications can also be implemented with the lighting in a hospital, increasing the overall efficiency of these lights by making them multi-purpose.

1.5 Goals and Features

The goal of this system is to ultimately be able to send data from one point to another using only visible light. Ideally, this system would be able to transfer any type of data at a high speed. However, the success of this design does not depend on the creation of a new type of communication system that will instantly replace all other means of data transfer. The objective of this system is to be able to send data reliably and accurately over a short distance at a fair speed.

Initial goals for the functionality of this system include being able to send text or pictures over a distance of approximately one meter at a data rate of at least 1 Mbps. To do this, the transmitter portion of the design would receive a signal from a computer and control the flashing of an LED to send bits to the receiver which would, with the help of another microcontroller, decode the signal and present the data back in the original format. The system would be powered by external AAA batteries to allow for more flexibility as the system will be mobile.

Additional functions that would enhance the project but are not mandatory goals of this design include sending video, sending data at a distance greater than one meter, and transmitting data at a minimum of 1 Gbps. Other features include using different colored LEDs simultaneously to increase data transfer rate and/or allow simultaneous use by multiple users, as well as somehow permitting omnidirectional transmission. The reasons for not including these features include time constraints, budget concerns, as well as stability issues. When trying to transmit at higher frequencies, stability becomes more on an issue as parts become less ideal. Also, in order to transmit at a higher frequency the quality of our design parts would have to increase which would cost more money and consume more time.

Since this project was conducted during the school year during three of the four terms, milestones were set for the completion of the design by using each term as a checkpoint. For the first term our goal was to be able to have a signal sent from our transmitter received at the receiver end of the design. This would ensure that once we increase the frequency of transmission the signal would still be received since at higher frequencies, the LEDs would not appear to be flashing to the human eye. By the end of the second term the design should have progressed to a point where the transmitter end of the circuit is fully operational and interfaced with a microcontroller. The design would come full circle by the end of the last term with both ends of the design being interfaced with microcontrollers. By this time a picture or text messages would be able to be sent from transmitter to receiver.

Our initial goals were to transmit first at a range of approximately one meter at a transmission rate of 1 kbps, and then increase this transmission rate to 1 Mbps. Ideally, this was to occur in a room that already has some ambient lighting. The system would be powered by external AAA batteries totaling approximately a 12 V drop. The data transmitted would first be text, and then audio or images. The receiver and transmitter would need to have a direct line-of-sight in order to ensure accurate data transmission with minimal errors.

In order to implement a functional prototype, some goals needed to be adjusted. The final design is capable of transmitting only text over a distance of 20-30 cm. The frequency of transmission between the transmitter and receiver is 1kHz. In order to output what the receiver has received, the data is exported to a file that is then decoded using a Matlab script. For a more instantaneous result, the Matlab script is running in the background and monitors if the file has been updated by constantly checking the file's time of modification. Once the script sees a change, it then outputs the decoded data in the Matlab console in the form of a text message.

Chapter 2: Design Approach

This chapter will discuss the specifications required for each block of the system architecture, and how it was implemented. These functional blocks are the same for both the transmitter and receiver side, but with different functionalities and implementations. The blocks include power sources, analog circuitry, a microcontroller or digital signal processing (DSP) chip, and a computer.

2.1 Functional Block Diagram

Figure 5 shows the overall functional block diagram of our system. The transmitter side consists of a signal source, a microcontroller, and analog circuitry incorporating LEDs, all of which are powered in some fashion. The receiver side is similar, containing analog circuitry incorporating photodiodes, a microcontroller and a device capable of receiving and interpreting the output, all of which are also being powered in some fashion.

The microcontroller is used as the signal source for our design by utilizing a binary system to transmit text. Each voltage maximum corresponds to a single binary 'high' digit and each voltage minimum corresponds to a single 'low' digit. This scheme is used in conjunction with the ASCII binary values, found in Appendix B, to encode a text message which is sent to the receiver side of the design utilizing LED flashes.

A power MOSFET is used to amplify the strength of the signal for increased transmission range. This particular MOSFET includes a built-in gate driver which is necessary for applications involving low-voltage logic such as the microcontroller used in this design. The device works in a way such that the signal is transmitted exactly as intended, however the logic highs and lows are inverted. To make up for this voltage inversion, the output data signal from the computer will also be inverted to produce the correct signal after the MOSFET block. The operation of this device is explained more in Section 2.2.4.3 MOSFET.

Two different power sources are utilized in the transmitter portion of the circuit. The first is used to power the MCU and is low maintenance since this source is the computer connected to the MCU. Since the MCU needs Code Composer Studio to operate the power from the USB port of the connected computer is used to power the MCU. The second power source is a 5V/2A AC/DC converter connected on one end to a wall outlet and on the other to the drain of the power MOSFET. The details of this source are covered in more detail in Section 2.4.1.2 AC/DC Converter.

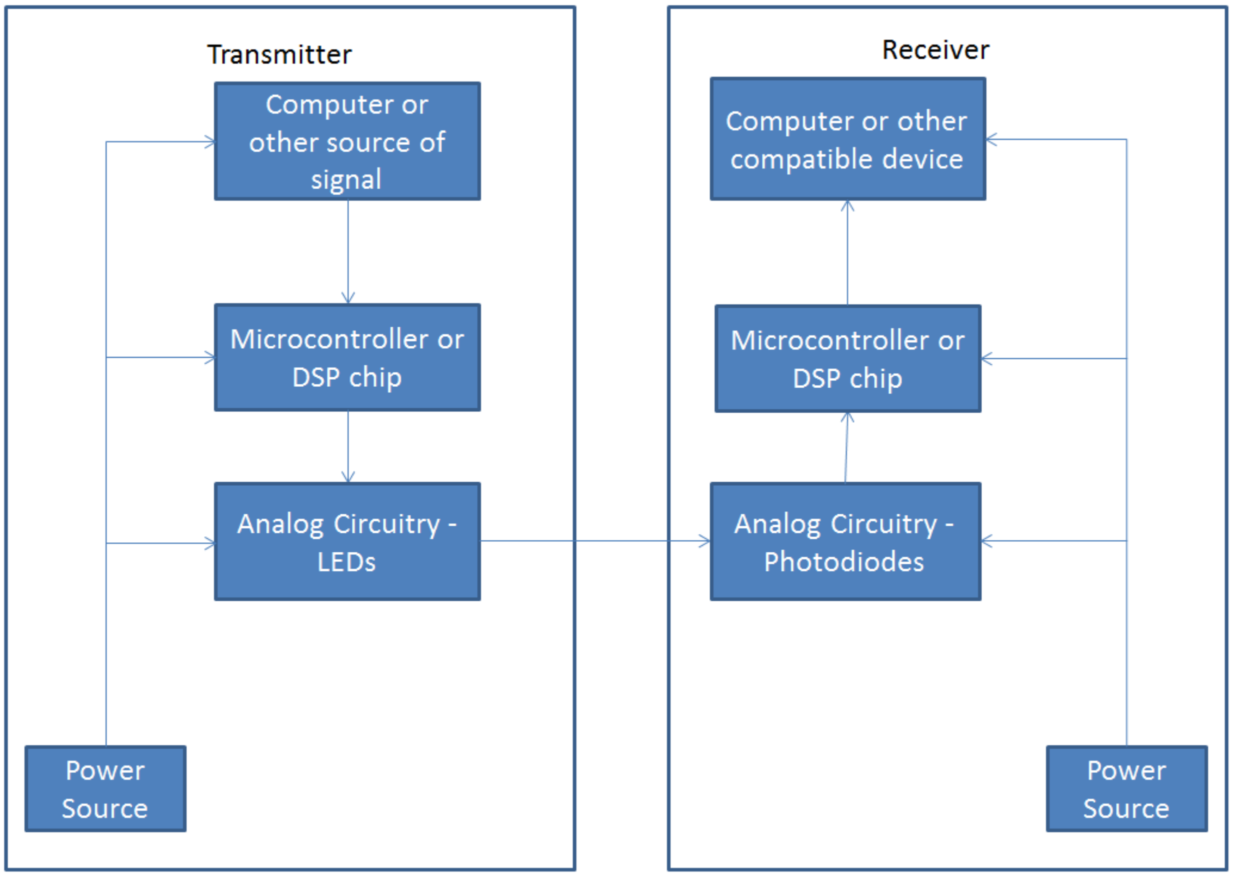

Transmitter
Computer or other source of signal
Microcontroller or DSP chip
Analog Circuitry - LEDs
Power Source
Receiver
Computer or other compatible device
Microcontroller or DSP chip
Analog Circuitry - Photodiodes
Power Source

Figure 5: Functional Block Diagram

The power source for the receiver will be AAA batteries that will supply power to the analog circuitry. The analog circuitry on the transmitter side will be powered by an outlet. The computer will also be powered by an outlet and will either provide a message on the transmission side, or read a message on the receiver side. The DSP chip will be powered by the computer and will decode the message on the transmission side to send a signal through the analog circuit containing the LEDs, or will decode the message from the analog circuit containing the photodiodes. The LEDs will be blinking at a rate corresponding to the message being sent, which the photodiodes will receive at a distance away from this transmission block.

2.2 Modules

Each block has its own specifications that need to be met in order for the system to function. The following sections will address these specifications. Many of the modules, such as the MCU, power source and analog circuitry, are present on both the transmitter and receiver.

2.2.1 Power Source

Both transmitter and receiver need some source of power; however, each component needs varying amounts of power. The transmitter end of the design utilizes the same power source but converts that power differently for different components. Both the MCU and MOSFET utilize a wall output with 120V AC output, however neither of these devices are connected directly to the AC power.

The MCU is connected to a computer using a USB connection cord which outputs 5V DC. This voltage is used to power the MCU while the actual signal is sent to the MCU using CCS. The MCU then outputs either a logic high, 3.3V, or logic low, 0V, to the gate of the next component; the MOSFET. The signal sent from the MCU is applied to the Gate of the MOSFET device which, when high, turns the device on and, when low, turns the device off effectively controlling current flow to the LEDs.

Explaining the power source for the MOSFET device is more involved than specifying a single voltage. This is due to the nature of the MOSFET where the device is only in the Active region and behaves as desired when there are differing voltages applied to both the Gate and the Drain of the device. The operation of this device is explained in greater detail in Section 2.2.4.3 MOSFET. The voltage applied to the Gate is the signal being transmitted while the voltage applied to the Drain is the converted voltage from the wall outlet. This 120V AC travels through an AC to DC converter which then outputs a 5V/2A DC signal to the Drain of the MOSFET. This scheme allows for a higher signal amplitude and therefore results in brighter LEDs. The brighter the LEDs are, the further away they can be from the photodiodes and have minimal loss in signal integrity.

On the receiver side of the design the only device that needs powering other than the MCU is the Op-Amp. The AD848 Op-Amp can have rails set to as high as +/-15V and has typical values listed in the datasheet for 5V and 15V. For our purposes the ratings at 5V were more than sufficient so this setting was chosen. In order to achieve this voltage rating, 4 AAA batteries, rated at 1.5V each, are connected to the rails of the Op-Amp. Even though the batteries at rated at 1.5V each, the typical output voltage is 1.3V which puts the rails at 5.2V instead of 6V. This means that the typical values listed in the datasheet are accurate guidelines for predicting the behavior of the device. The Op-Amp device is explained more thoroughly in Section 2.2.4.4 OP-AMP.

2.2.2 Signal Source

On the transmitter side, the signals that will be transmitted are text signals. These signals could be produced by a computer, but could potentially be produced by some other compatible device, such as a cell phone. This signal will then be sent to the microcontroller or digital signal processing (DSP) chip for processing. On the receiver side, another computer or compatible device needs to be able to interpret the original signal by taking the received signal from the receiver microcontroller or DSP chip.

2.2.3 Microcontroller

The microcontroller or DSP on the transmitter side will convert the signal from the source into bits through an “On/Off” keying scheme using logic 1s and 0s. This could require some coding with MATLAB to first convert the data into a waveform for processing. The resulting waveform will then be sent to the analog circuitry for the LEDs. On the receiver side, the microcontroller or DSP will need to be able to take the signal from the photodiodes set a threshold voltage that will offset the voltage that will be picked up from the ambient light in the room. Next, the bits will need to be decoded into voltages for the computer or compatible device. Again, MATLAB will be useful in producing these filters, and some C programming will be necessary for setting the threshold voltage and converting the bits back into voltages that can be deciphered by a computer or corresponding device. More on this block will be discussed in Section 2.5 Micro Controller.

2.2.4 Analog Circuitry

The analog circuitry on the transmitter side needs to take a pulse from the MCU and light LEDs at a rate equal to the frequency of the pulse. This can be done with a few resistors and an array of LEDs. 22 high-brightness white LEDs are used and connected in parallel for current limiting reasons. The receiver side analog circuitry will contain photodiodes to detect the fluctuations of pulses from the LEDs. Resistors and operational amplifiers are also used, as the signal produced by the photodiodes are very small, and thus need to be amplified in order to produce a signal for further process. This will be further discussed in Section 2.3 Analog Design.

2.2.4.1 LEDs

The medium being used to transmit data in our design is light with this light being provided via LEDs. The most important parameter associated with the LEDs is the brightness of the device which is measured in units of Lumens. It is important to note that the Lumens unit of light is not the same as the Candela unit.

Lumens refers to the ‘total’ amount of light that a device emits whereas Candela refers to the power emitted by a light source in a particular direction. This means that if an LED emits 1 Candela towards a photodiode but there is a wall in between them that does not allow for light to travel through it, the photodiode actually sees the LED as emitting 0 Candela.

The LEDs used in our design are measured in Lumens instead of Candelas because our LEDs emit light in a viewing angle of 15^{o} instead of a single direction. This means that not all of the power in Lumens actually reaches the photodiode, but we set up our design so that the center of the viewing angle is level with the photodiodes' 75^{o} viewing angle center of receiving light. This is possible because both the receiver and transmitter circuits are constructed on breadboards of equal size so when both LED and photodiode are angled correctly their centers align.

The next thing to note about the LEDs in this design is the amount of current, and therefore power, drawn from each device used. According to the datasheet of the LEDs selected for our design, the current draw for each device is approximately 20mA. Since 22 LEDs are used in the design to provide more light to be received by the photodiode, this means that the total current drawn from all of the LEDs is equal to approximately 440mA. Taking into account that the typical forward voltage of the LEDs is 3.2V when the forward current is equal to 20mA, the power dissipated in each LED as well as total power can be calculated. The equation used for power dissipation is as follows:

$$P_{LED} = I_F * V_F \tag{2}$$

where I_F is the forward current of a single LED and V_F is the forward voltage of a single LED. Substituting the values of 20mA and 3.2V for current and voltage respectively, the power dissipated by a single LED is found to be 64mW. Taking this value and multiplying by the total number of LEDs, 22, results in the total power dissipation of the LEDs in the system, totaling 1.408W.

An important characteristic to note about both the LEDs and photodiodes are the frequencies at which each device emit light and react to light, respectively. From Figure 6, shown below, it is clear that the LEDs being utilized in this design are most effective at a lower wavelength than the half-way point in the range of photodiode detection. This means that both devices are not ideal for one another which leads to the need for an Op-Amp to increase the amplitude of the received signal.

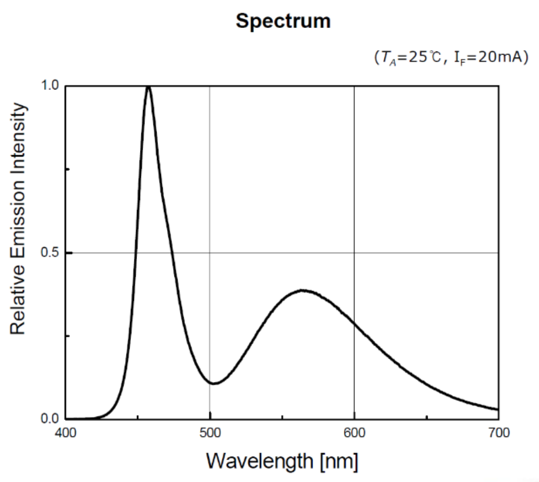


Figure 6: Emission Spectrum of LEDs for Various Frequencies

2.2.4.2 Photodiodes

In order for data transmission to have any significance there must be a way to receive the signal at the other end of the design. This is the purpose of the photodiodes as they react to the light emitted from the LEDs and allow for current to flow to the rest of the receiver circuit. When there is no light emitted from the LEDs the photodiodes do not allow current to flow through to the MCU on the receiver.

As mentioned above, the photodiode reacts to the light emitted from the LEDs to create a signal for the MCU on the receiver end of the design to process and decode. Our design works exceedingly well in a dark room, as one might expect, since there is no ambient light to interfere with the photodiodes receiving the LED signal. Our design also works in a lit room with ambient lighting that creates noise.

A simple fix is required to adjust for the ambient lighting in a lit room. The ambient lighting can be viewed as a approximately a constant signal that, when combined with LED lighting, simply adds as DC bias to our transmitted data. To resolve this problem with the photodiode, there is digital processing on the digital receiver circuit to ensure that this biasing does not affect the sampling of the MCU. More on this will be addressed in Section 2.2.6 Receiver.

Our design implements seven total photodiodes with the purpose of covering the entire breadboard such that at least two LEDs are aligned directly with each photodiode's center. According to the datasheet for the selected photodiodes, the power dissipation for each photodiode is approximately 100mW. Using the same equation used to calculate the power dissipation of each LED, the total power used by each photodiode is equal to the product of the forward current and forward voltage, 80mA and

1.3V respectively. Utilizing the exact power calculated from the datasheet, 104mW, instead of the approximate value provided, the total power dissipated from all of the photodiodes is 728mW.

Like with the LEDs, the photodiodes have a spectrum of frequencies at which they react to light more so than other frequencies. This spectrum, shown below in Figure 7, is the sensitivity of the photodiodes to each frequency of emission from the LEDs. As mentioned above, the disparity between the two peaks of these spectrums requires the use of the AD848 Op-Amp for easier signal processing purposes.

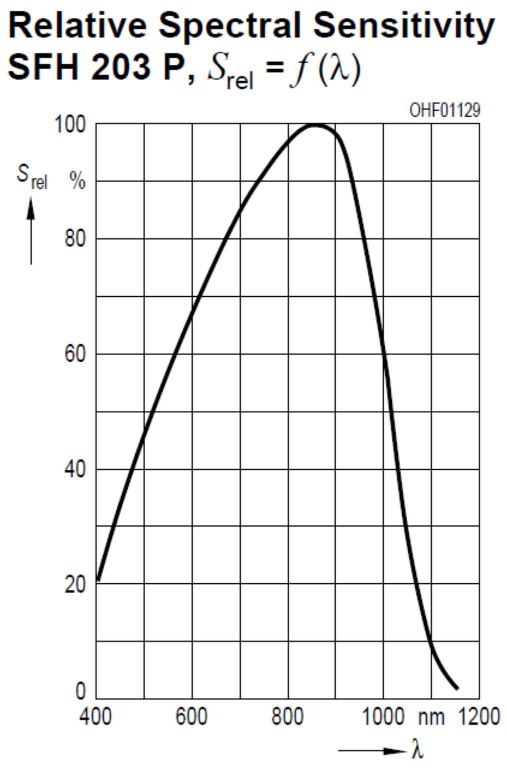


Figure 7: Relative Spectral Sensitivity of Photodiodes at Various Frequencies

2.2.4.3 MOSFET

An analog part implemented solely to amplify the transmitter signal amplitude, the N-Channel FQP30N06L MOSFET, utilizes an internal gate driver which solves all current limitation issues. In order to get the LEDs to emit a brighter light and increase the possible transmission distance of our design, it was necessary to provide more current to the LEDs since the logic output of the MCU was too low.

In order to explain how the MOSFET device increases the current provided to the LEDs, it is necessary to have a basic understanding of how such a device functions. There are three primary modes that a MOSFET device can operate in. Those three modes are Cutoff, Triode, and Active or Saturation. For this design, it is better to operate in the Active region as this region provides the most consistent measure for Drain current.

While a device is in the Active region, the Drain current is nearly constant and mainly dependent on the Gate to Source voltage. The equation for the drain current in the device is given as:

$$I_D = \frac{\mu_n C_{OX}}{2} \frac{W}{L} (V_{GS} - V_{th})^2 \tag{3}$$

This equation ignores the channel-length modulation effect that occurs, but since our device operates at lower voltages and currents this can be neglected for simplicity. The only parameter that can be controlled in our design is V_{GS} which is the voltage differential between the Gate and the Source of the MOSFET since the other parameters are determined in fabrication of the device. The above equation holds only when the MOSFET is in the active region; or when V_{GS} is greater than the threshold voltage, V_{th}, and when the Drain to Source voltage, V_{DS}, is greater than the effective voltage, ($V_{GS} – V_{th}$).

When the above conditions are not met, the device operates in either the non-ideal Triode region or is in Cutoff and does not operate at all. The problem with the Triode region is that the Drain current, I_D, is not constant with increasing values of V_{DS} but rather is ohmic and linear in nature. This makes the exact value of the drain current difficult to evaluate and makes other calculations approximations rather than accurate representations. The issue with the Cutoff region is that there is no current flowing through the Drain of the device, and therefore there is no current flowing into the LEDs.

In order to use a Power MOSFET with a low voltage logic device such as the one used in our design, there must be a Gate Driver to ensure the device can operate at the desired frequency. A MOSFET device is capable of switching on-states rapidly as long as the Gate Capacitance is charged fast enough for the device to turn on and off again. When MOSFETs are used in standard DC applications there is no need to have a Driver because once the device is on there is no need to turn it off unless the power is cut to the system. However, with our logic input being applied to the Gate of the MOSFET the V_{GS} is constantly changing from 3.3V to 0V. The MOSFET device selected for our design has a built-in Driver that takes care of the Gate charging that is needed in order for the device to switch at our desired frequency.

The power dissipation of this device is the only drawback to this device. According to the datasheet, the device has a typical power dissipation of 0.53W/^{o}C where the temperature is the ambient temperature around the device. Assuming a room temperature operation of around 25^{o}C, this would mean the MOSFET dissipates 13.25W of power. This is by far the most power consuming device in the entire analog design. However, since the wall outlet is the primary source of this device there is not much concern in the amount of power consumed through that medium since the typical wall outlet contains 120V AC in the United States which is connected to a circuit breaker of typically 15-20A. This means a standard wall outlet is capable of providing 1800-2400W.

However, because the device is switching so quickly there is a noticeable heating of the device. When a MOSFET's junction temperature is increased, the on-resistance, R_{on}, is increased which increases the power dissipation of the device. This means that while the device is left on for transmission the actual power dissipation of the device is increasing due to the increased R_{on} with the rising temperature. This phenomena is known as thermal-runaway and is common when the load of the MOSFET is a continuous current draining device such as the LEDs used in the design. The absolute maximum power dissipation for this device is listed on the datasheet at 79W which, while significant when compared to other aspects of our design, is still not enough for concern when using a wall outlet for power.

2.2.4.4 OP-AMP

Another analog part that is implemented on the receiver end of the circuit along with the photodiode is the AD848 operational amplifier. The purpose of this Op-Amp is to amplify the received signal of the photodiodes. Since the amount of current released from the photodiodes depends on how close the LEDs are with respect to the photodiodes, it is necessary to amplify the received signal when the transmission distance is increased.

Another issue with the emission and sensitivity spectrums of each device becomes apparent as well. Both of these charts are shown above in their respective sections; Section 2.2.4.1 LEDs and Section 2.2.4.2 Photodiodes. Because the peak values for these two charts are not equal, the Op-Amp is required in order to increase the received signal to an amplitude high enough to be able to perform signal processing. Although there is amplification on the transmitter end of the circuit utilizing the MOSFET, the signal is further amplified with the Op-Amp in order to allow for more accurate sampling by the MCU.

2.2.4.5 USB-B to Circuit Board

As stated in Section 2.2.4.3 MOSFET, the MOSFET device is powered utilizing an AC/DC converter from a wall outlet. However, the output of the converter is not directly able to be applied to our breadboard. This is due to the output being USB-A and not wires. In order to get around this problem we utilized a USB-B breadboard adapter.

The device, shown below in Figure 8, is an adapter that interfaces a USB-B connection to leads that can be placed on our breadboard. The device has 4 pins with the two middle unused pins being differential voltages to add to the power from the USB connection. The two pins being used are the ground and power pins with the power flowing into the Drain of the MOSFET device.

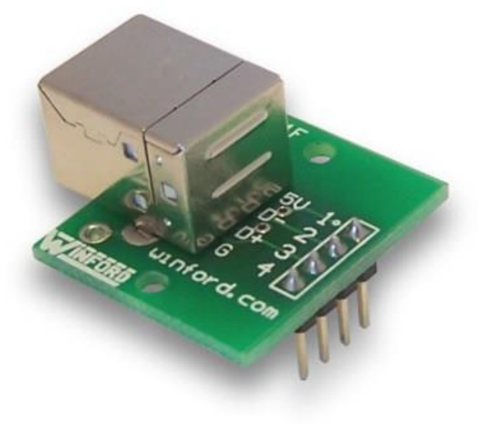


Figure 8: USB-B to Breadboard Adapter

The biggest distinction to make in this part of the design is the difference between USB-A and USB-B. A USB-A and USB-B side-by-side comparison is shown below in Figure 9. The USB-A variety is a flat, rectangular interface that holds the connection in place with friction. USB-B is more square shaped than the rectangular type A, and has slightly beveled corners on the top ends of the connector. Type A connectors are used on hosts that supply power whereas type B connectors are used on hosts that receive power. This scheme is implemented in order to prevent a user from connecting two devices that give power to one another which would lead to dangerous circuit conditions such as high currents or extreme heat resulting in a fire.

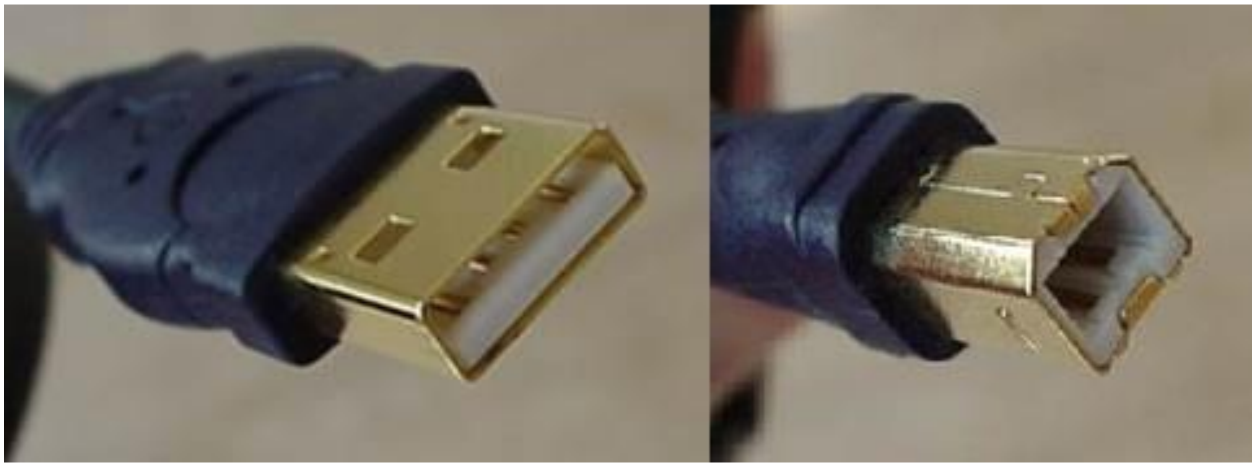


Figure 9: USB-A (left) and USB-B (right) Comparison

2.2.5 Transmitter

The entire purpose of a communication system is to send data from one location to another in order to convey information to a user on the end of the system. A transmitter, and in turn a receiver, are required to achieve this goal so that data can be sent wirelessly. Both analog and digital components are used on both ends of the system and work simultaneously to transmit and receive data using only visible light.

The transmitter, although incorporating analog parts, is mostly digital when processing the actual data itself. Most of the data manipulation is done on a computer program made for use in conjunction with the C2000 Piccolo LaunchPad being used to transmit data. The specific digital signal processing chip being used, the F28027, allows for communication between the computer and LaunchPad through micro USB connection. This board is also referred to as the LAUNCHXL-F28027.

The program used with the F28027, Code Composer Studio (CCS), allows a user to write and implement C/C+ code to the digital board. After a one-time configuration of connection settings, CCS is ready to execute all code written onto the board. Data is converted into binary in order for simple transmission in a digital sense. Once this data was entered into the transmitter C code, and processed at the specified frequency, the F28027 would output a waveform to the analog circuit board using the an edited version of the pulse-width modulator (PWM) Texas Instruments (TI) example code to create a square wave with the appropriate frequency and duty cycle.

The transmitter analog circuit leads to an array of LEDs. The LEDs would stay off when the input from the F28027 was low and would light when the input was high. This on-off behavior is not visible to the human eye so the constant switching of the LED would not bother or distract someone in the area as the light would appear to be either constantly on or off. An elaboration on this block will be in Section 2.6 Digital Side of Transmitter.

2.2.6 Receiver

The switching of the LEDs would serve as the wireless means for data transmission as the receiver analog circuit would pick up on this changing behavior. Photodiodes were used as a way to let the receiver “know” that data is being transmitted. When the photodiodes detect a change in lighting from the LEDs, current flows into the next stage of the circuit, the op-amp. The op-amp is configured in a way so that the waveform from the photodiodes would be amplified for use in the second F28027 board.

The digital aspect of the receiver is the most complicated part of the entire design because this signal is not being created but rather decoded in the F28027 board’s Analog to Digital Converter (ADC). Although initially, much of the initial testing and implementation was done with the C2000 board, we eventually chose to switch from this C2000 board to the MSP430F5529 LaunchPad Evaluation Kit. An elaboration on the functional aspects of this block will be in Section 2.7 Digital Side of Receiver, and an explanation on why we ended up switching to the MSP430F5529 will be in Section 2.5 Micro Controller.

2.3 Analog Design

In order to transmit digital information in an ordinary communication system, such as Wi-Fi, the desired information is converted to an analog signal. Once the data is converted, an antenna is used to convert the signal into electromagnetic waves to propagate through the air. To collect the propagated signal, an antenna is also needed on the receiver side. For a visible communication system, the desired information is transmitted through visible light, instead of electromagnetic waves. Because of this, transmitting and receiving the desired information must be done differently.

To design the transmitter for the visible communication system, we use LEDs to transmit the information. Once the desired information is ready to be sent, the digital data that is detected as zeros and ones can then be converted into an analog voltage and sent to the LEDs. To distinguish between the different bit values, the LED can either be turned on or off. This can be done by manipulating the voltage through the LED. If a bit has the value of one, a sufficient positive voltage will be applied to the LED to turn it on. Conversely, a bit value of zero will drop the supply voltage to the LED to zero and the LED will turn off. This whole process is referred to as ON and OFF Keying because of how the LED is constantly turning on and off to transmit information.

For the receiver, a device is needed that can distinguish the change in light from the LED. Such a device is a photodiode. A photodiode will produce a current based on how much light it absorbs. Once the signal has been received, it will need to be amplified because the output current from the photodiode is usually on the order of μA. To amplify the signal, an operational amplifier is used to convert the input current into an output voltage that is then amplified.

2.3.1 Value Analysis

Upon designing the analog circuitry for both the transmitter and receiver, the team determined which analog components were needed to complete the design. The transmitter's analog circuit consisted of LEDs, a MOSFET, and resistors. The receiver's analog circuit required photodiodes, an op-amp, resistors, and a power source. Once the type of component was determined, some research was done to determine the appropriate component for the design. A value analysis was performed on each analog component which factored in certain areas of focus such as cost and performance.

2.3.1.1 LEDs

For the value analyses of the LEDs and photodiodes, a green background represents the most ideal specification within a category. A yellow background indicates an acceptable specification for our design and red indicates a specification unsuitable for our design. The value that each category held was determined either by testing or theoretical conjecture.

LED	Luminous Intensity	Color	Price
RL5-R12008	12000 mcd	Red	$0.59
RL5-W18030	18000 mcd	White	$0.99
WP7113IT	80 mcd	Red	$0.15
LW514-BULK	32000 mcd	White	$0.66

Table 3: Categorical Value Analysis of LEDs

From initial prototype testing, the standard LED (WP7113IT) had negligible effect on the output of the photodiode being used regardless of how close the LED was in relation to the photodiode. When testing with a white-light LED of higher intensity from a smartphone, the photodiode showed significant responses. It was then determined that the intensity of the LEDs was not adequate, and so higher intensity LEDs were found. White-light was deemed most ideal color for our design as the frequency of white light was not a single peak value as with other colors such as red. Cheap parts are ideal; however, tradeoffs are necessary to meet other specifications. After the initial testing and determining that the intensity of the LED is the most important factor, this carried the most weight in our decision making process. That in mind, the LW514-BULK LED is the brightest of the selection of LEDs and is cheaper than the next brightest LED which led to selecting this device for use in our VLC system.

2.3.1.2 Photodiodes

The table shown below represents the value analysis for several photodiode devices researched during our initial prototyping phase. Several specifications were looked at with the most important ones being a very wide range for use with any type of LED and a low cost in order to use multiple of them in the design.

Photodiode	Spectral Range	Wavelength @ Peak Sensitivity	Cost	Response Time
SFH 203 P	400-1100 nm	850 nm	$0.72	5 ns
SFH 229	400-1100 nm	850 nm	$0.66	10 ns
AD800-11-TO52-S1	300-1100 nm	600 nm	$211.24	1 ns
PD70-01C/TR10	150-1200 nm	900 nm	$0.83	50 ns
PD15-22C/TR8	400-1100 nm	900 nm	$0.47	10 ns
PS1.0-6b TO	350-1100 nm	925 nm	$32.33	30 ns/10 ns
720-BPW21	400-1100 nm	550 nm	$6.70	1.5 us
PDA10A	200-1100 nm	750 nm	$283.00	150 MHz BW ~ 6.7 us

Table 4: Categorical Value Analysis of Photodiodes

Since the project focuses on visible light, it is ideal to restrict the wavelengths that the photodiode can detect to the range of wavelengths in the visible spectrum. Therefore, having the peak sensitivity near the center of this range is the most reasonable. Unfortunately, our budget is limited, so the photodiodes in the $200 range are unsuitable for our design. Because of this limiting factor the importance of the peak sensitivity was lowered and the importance of a wider range was emphasized; hence the largest ranges having a green background.

For response time, the device with the lowest response time is the best. However, our goal was to transmit data in the megabit range so all of the listed photodiodes were acceptable. Taking all the factors into account, the SFH 203 P device was chosen instead of the SFH 229 for the slightly faster response time while only costing an additional six cents per device.

2.3.1.3 MOSFET

The choice of which MOSFET device to use was not a difficult decision since certain specifications needed to be met in order for the transmitter to operate properly. Due to the lack of sufficient output current from the microprocessor, the MOSFET is needed to increase the signal strength of the transmitted signal. In order to achieve the desired signal strength, the minimum drain current of the MOSFET had to be above 440mA. This was determined after the number of LEDs being used was established. Once that number was finalized at 22 and the current draw on each LED was found to be 20mA the total current from the MOSFET had to equal 440mA.

In order to find a MOSFET that was both through-hole and had a continuous drain current of 440mA, a search in the Mouser database was done to find a suitable part. This database allowed parts to be searched with specific values for parameters specified. This allowed for the mounting type to be selected as through-hole instead of surface mount and for the exact drain current to be specified. The MOSFET that fit both requirements was the ZVN4206A N-channel MOSFET from Diodes Incorporated.

This first of these MOSFETs, however, proved to be the incorrect device for our design. Since the input to the Gate of the MOSFET device is a low power logic input, there is not enough current supplied to the device to switch it on and off quickly enough. This required the MOSFET to have an internal Gate Driver which would charge the gate capacitance to a high enough level for the device to switch on and off the same frequency of our signal. Upon further research the only device that was found to be through hole, had an internal gate driver, and operated in low power circuits was the FQP30N06L N-channel Power MOSFET from Fairchild Semiconductors.

2.3.1.4 OP-AMP

In order to select the proper Op-Amp for use in our VLC system, all specifications had to be taken into account. The Op-Amp is used on the receiver portion of the analog circuitry in order to have a gain on the received signal. Since the amplitude of the signal is not as high as desired due to the non-ideal peak sensitivity of the photodiodes in relation to the peak emission of the LEDs, it is necessary to have an op-amp connected for gain.

Several parameters were taken into consideration when determining the ideal op-amp for our VLC system but the three factors that weighed the most were the price of, the unity gain frequency of, and the voltage required to operate each op-amp. The three op-amps whose rail voltage parameter fit our specification of 5V are shown in the table below. The rail limits were determined after we had established that four AAA batteries would be used to power the analog receiver sections, as these batteries can provide up to 6 volts maximum with a typical value of 5.2V.

Op-Amp	Unity Gain Frequency (MHz)	Cost ($)	Rails (V)
AD-484	175	6.75	5-15
LT-1797	10	1.16	5
Max-473	10	2.50	3-5

Table 5: Value Analysis of Op-Amp

Table 5 displays the Op-Amp and the parameters for each. Initially, the second op-amp appears to be the most efficient for our means. However since we only require one of these op-amps rather than in bulk, this option becomes less attractive since the listed price is the price paid when bought in bulk and the minimum order was 1000 for the parts distributor, DigiKey. The third option also fit the specifications needed for our op-amp, but was no longer in stock anywhere.

The reasoning for selecting a unity gain frequency of 175MHz is that the system was initially intended to operate at a much higher frequency than it would in its final state. Although this high frequency may not look optimal, as long as the frequency we are operating at is lower or equal to this unity gain limit the op-amp will perform as intended. Therefore, the best Op-Amp for our design is the AD-484 device.

2.3.2 Initial Analog Design

For our initial step in designing the transmitter and receiver for the visible communication system, we decided to design and test the analog schematics for both end systems. Before reaching the analog components in the transmitter, the digital data will be converted to a continuous analog signal. To model the continuous signal, a signal generator was used to produce a continuous square wave. The following figure represents the remaining analog components for the transmitter.

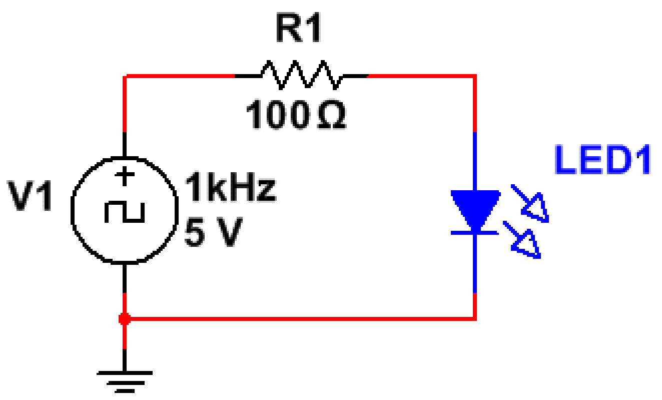


Figure 10: Basic Visible Communication Transmitter

The square wave was constantly alternating from a high to low voltage, where the high voltage was set to +5V and the low was set to 0V. The signal was then connected to a 100Ω resistor and a series of LEDs that were connected in parallel. The resistor was placed in the circuit to limit the amount of current that would flow through the LEDs. The LEDs are placed in parallel to ensure that enough voltage can be supplied to each LED. As mentioned earlier, the square wave will constantly switch the LEDs on and off. If the signal's frequency is high, the LEDs will appear to be constantly on to the human eye. Because of this, it is possible to transmit information and not hinder the main purpose of the lights.

Once the LEDs begin to propagate the light, the receiver must absorb the light and reproduce an example of the propagated signal. The following figure represents the initial receiver design used for the communication system.

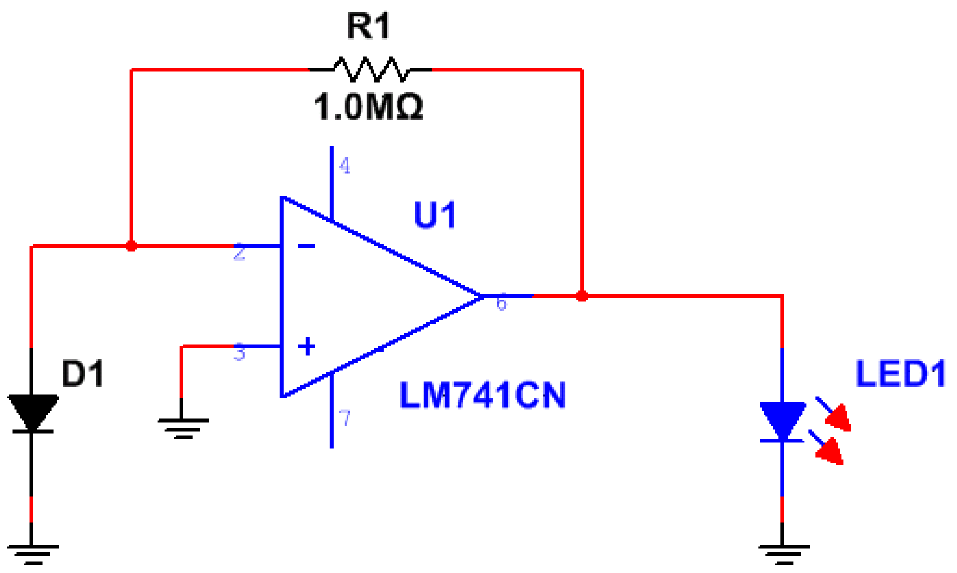


Figure 11: Basic Visible Communication Receiver with LED

The light signal is initially absorbed by the photodiode, which is labeled as D1 in Figure 11. As a photodiode absorbs light, it produces an output current. If more light is absorbed by the photodiode, it will produce more current. The current then must be converted into a voltage so it can be passed to the microcontroller. One way to do this is by using an operational amplifier. The photodiode is connected to the amplifier's negative input, while the amplifier's positive input is connected to ground. A 1MΩ resistor connects the amplifier's negative input to its output. Since the input current to the amplifier is zero, the current produced from the photodiode must flow through the resistor. Because the resistor is connected to the amplifier's output, the output is equal to the voltage drop across the resistor.

2.3.3 Final Analog Design

After several revisions of our analog design, we decided upon the design shown below in Figure 12.

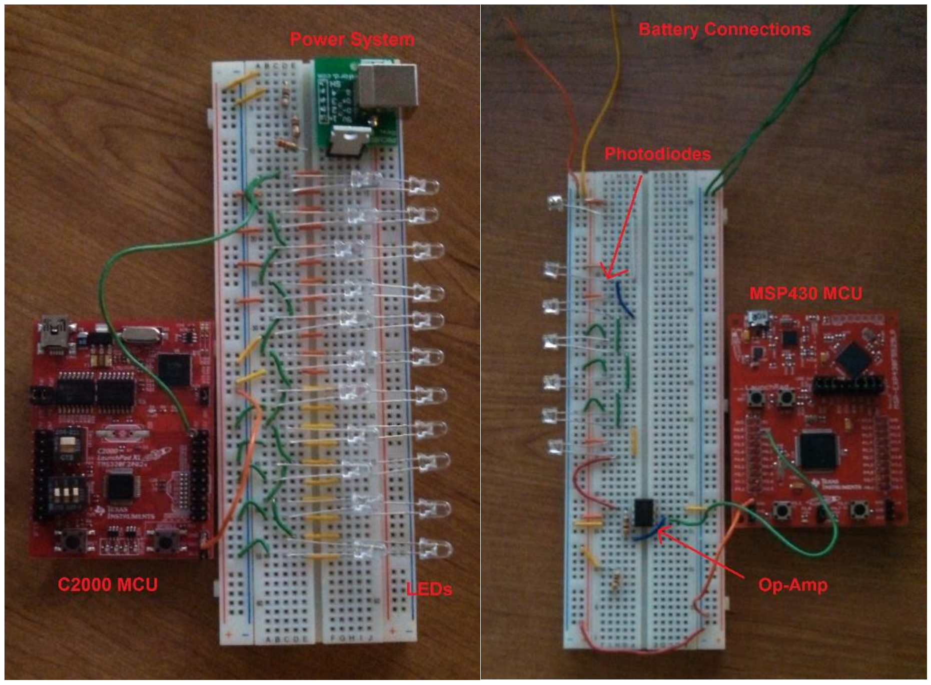


Figure 12: Prototype Implementations of Transmitter (left) and Receiver (right)

The basic schematic is similar to those of Figure 10 and Figure 11. The difference in the transmitter is that it has an array of twenty-two white LEDs connected in parallel. Because the devices are connected in parallel, there is no guarantee that each LED will draw the same current. This means that the brightness of the transmitter block does not necessarily increase proportionally with an increased number of LEDs. In order to rectify this issue the MOSFET shown was implemented to provide enough current to each LED to reach their maximum brightness. More on this solution is discussed in Section 2.2.4.3 MOSFET.

On the receiver side, the analog block has an array of seven photodiodes to increase the current flowing through the receiver block. Each photodiode was placed along the board and aligned to the transmitter to maximize the amount of light captured from the LEDs and increase the current provided from the photodiodes. The AD848JNZ Op-Amp is connected with a resistor so a voltage can be passed through the Op-Amp, outputted to the receiver MCU, and processed. More on each part of this design is found in each modules respective section.

2.4 Power Source

Our VLC system is powered with primarily two sources. The first power source is a set of batteries which powers the Op-Amp on the receiver side of the analog circuitry. The second source of power in the system is a wall outlet that is used in conjunction with a USB connection to a computer and an AC/DC converter.

The batteries being used are AAA 1.5V with four being used for a total of 6 volts. This set of batteries will be used to power the Op-Amp on the receiver. The second form of power in our VLC system comes from the USB output of the computer the DSP is hooked up to and the AC/DC converter. By using the computer as the power source for the DSP there is no need for more batteries in our system which saves on space and money.

2.4.1 Wall Outlet

2.4.1.1 Computer to MCU via USB

Of the two wall outlet conversions, this is by far the simplest since a computer is doing all of the converting instead of other analog parts. This power source is used to power the MCU with the required 5V to operate the device. The typical USB output from a desktop computer is 5V DC. Knowing this as well as needing the program Code Composer Studio, the decision to power the MCU through the computer was simple.

Once the MCU is powered on and the code is sent to the device, the MCU outputs the desired signal utilizing two pins. The first pin is a ground which is connected to the ground of the breadboard and the second pin is the output pin. Both pins are interfaced with the breadboard utilizing simple wire with crimped ends for easy use with the pins. The output signal that is sent to the breadboard is connected to the Gate of the MOSFET device with a 3.3V peak voltage representing a single ‘1’ bit and a 0V peak representing a ‘0’ bit.

2.4.1.2 AC/DC Converter

2.4.1.2.1 USB-A vs. USB-B

The second way that the wall outlet is used is by converting the AC power output to a DC power source for use with the MOSFET. In order to do this, an AC/DC adapter, which takes the 120V AC signal and converts that into a 5V/2A signal, is utilized since the converter is connected to the wall outlet. The total power available is approximately 1800-2400W, so power consumption is not an issue like it is with the batteries as described below. See Section 2.2.4.1 LEDs for details on this calculation.

2.4.2 Batteries

The other source of power in our design is present on the receiver side of the analog circuit. The Op-Amp requires a 5V rail to operate effectively so 4 AAA batteries are used to supply this voltage. As discussed earlier, the voltage maximum is 1.5V from each battery but the typical voltage is 1.3V from each which places the rails at 5.2V. Because the selected Op-Amp can operate with rails ranging from 5V to 15V the device will still work as expected and behave very closely to the typical values listed in the datasheet of the device.

The total power available from the four AAA batteries can be calculated using equation 2. Assuming a 1.3V constant voltage from each battery, the typical voltage from the datasheet, each battery discharges 100mA over an hour's time. This is within our timeframe for transmission since transmission is taking place at a 500Hz and the data being transmitted is only text. The total power available from all four batteries is 520mW for the first four batteries which is enough to power the Op-Amp on the receiver end of the circuit.

2.5 Micro Controller

Since the VLC design is transmitting information through light waves, the continuous analog signal must be discretized to be recognized by a computer. One way of accomplishing this is by using a microcontroller. The microcontroller will have an on-board Analog to Digital Converter (ADC), which would sample the incoming continuous signal. Once the signal is sampled, an assortment of processes can be performed to convert the collected data back into bits of ones and zeros. Once the binary information is recovered correctly, it is then possible to decode what was sent from the transmitter.

Another microcontroller is used on the transmitter to interface a computer to the analog circuit. The prepared message is sent to the controller and is then converted into binary bits. Once in binary form, the controller will look at one bit at a time and output a voltage pulse that corresponds to that bit. If the bit was a zero, the voltage pulse would be 0V, and if the bit was a one, the voltage pulse would be set to 3.3V.

2.5.1 Cost Analysis

2.5.1.1 Factors

In this visible communication system's design, both the receiver and the transmitter are interfaced with a computer using a microprocessor. When it came to deciding which processor to use for the design, there were a few factors to consider. Since the processor is meant to connect directly to the computer, the first factor that was considered was USB connectivity. Having a USB connection between the computer and the processor would allow the system to transfer information to and from the computer. By not having this functionality, it would not be possible to meet the goal of interfacing the communication system with the computer. To analyze if a processor met the requirement, it was given a value of five if it had USB connectivity, or a value of zero if it did not. The following table represents the breakdown of how the points were assigned for USB connectivity.

USB Connectivity	Value
No	0
Yes	5

Table 6: USB Connectivity Value Assignments

Another factor that is as crucial as the USB connectivity is the processor's sampling rate. One of the design goals was to have the system's transfer rate be in the megabits per second (Mbps) range. To meet this design, the microprocessor must have an analog to digital converter, or ADC, that can produce at least a million samples per second. The reason for this is due to the Nyquist rate, which states that a signal must be sampled at a frequency that is twice the signal's highest frequency content in order to keep the input signal intact. It is also possible to use a sampling rate that is exactly the same as the signals input and not cause interference. The following table ranks a few sampling rates into a value system ranking from zero to five.

Sampling Rate	Value
>1000ns	0
800-1000ns	1
600-800ns	2
400-600ns	3
200-400ns	4
0-200ns	5

Table 7: Sampling Rate Value Assignment

As mentioned before, since it is preferred to have a transmission rate in the mbps range, the ADC on the processor cannot have a sampling rate that is less than 1000 ns, which corresponds to 1

MHz. If the processor had a sampling rate that was slower than 1000 ns, it was not considered and given a zero. As the sampling rate decreases the corresponding value increases, which allows the system to operate at a higher rate.

The next factor that was considered was price. When deciding which processor to choose, the price of the processor and its development board were compared. Due to the project budget being set to roughly $300, the team determined that any board processor combo that was over $100 exceeded the budget and therefore were not considered. The following table describes the price breakdown for the processors and the corresponding values for them.

Cost	Value
$100+	0
$60-$80	1
$40-$60	2
$20-$40	3
$10-$20	4
< $10	5

Table 8: Price Value Assignment

The last factor that was considered for choosing the processor was its power supply. Since the visible light communication system was intended to be powered through batteries, there was a limited amount of power supply. Because of this, if the processor ran on an average power that exceeded 3.3V it was given a value of zero, and if the average was at or below 3.3V, the value was set to five. The following table represents the value breakdown for the processor's power.

Power	Value
> 3.3V	0
< 3.3V	5

Table 9: Power Value Assignment

2.5.1.2 Processors

When it came to choosing which processor is right for the design, the choices were narrowed down to four processors. The processors are all manufactured by Texas Instruments, and the manufacture names are: MSP430BT5190, MSP430F5529, TMDX28069USB, and LAUNCHXL-F28027. The following table represents each processor's sampling rate, price, power supply, and USB connectivity.

Processor	MSP430BT5190	MSP430F5529	TMDX28069USB	LAUCHXL-F28027
Sampling Rate	1000ns	1000ns	289ns	333-950ns

USB Connectivity	No	Yes	Yes	Yes
Supply Voltage	3.3V	3.3V	3.3V	3.3V
Price	$101.00	$15.24	$40.50	$17.70

Table 10: Processor Information

To get a better understanding of how the processors compare to each other, the processors' information was converted to numerical values using the value assignments. Each factor was also assigned a weight based on importance. Since the team determined that sampling rate and USB connectivity were equally important, they were assigned a weight of 50. The processors' price was also important but not as significant as the other factors so its weight was assigned to 30. The power was the least important because all processors had the same power supply; thus, the weight was set to 5.

To get the overall value for the processors, each processor's factor value was multiplied by the factor weight and then added up to achieve a total weight value. The following table represents the results in determining which processor to use for the design.

Weight	30	50	50	5	
Processor	Price	Sampling Rate	USB Connectivity	Power	Total
MSP430BT5190	0	1	0	5	75
MSP430F5529	4	1	5	5	445
TMDX28069USB	2	4	5	5	535
LAUCHXL-F28027	4	4	5	5	595

Table 11: Processor Value Analysis

Based on the results shown in the above table, the LAUCHXL-F28027 and the TMDX28069USB processors were better than the other options. Both systems use chips that come from the same line of microprocessors but the development boards are different. We decided to go with the LAUCHXL-F28027 as it had the largest total and that it was cheaper than the TMDX28069USB.

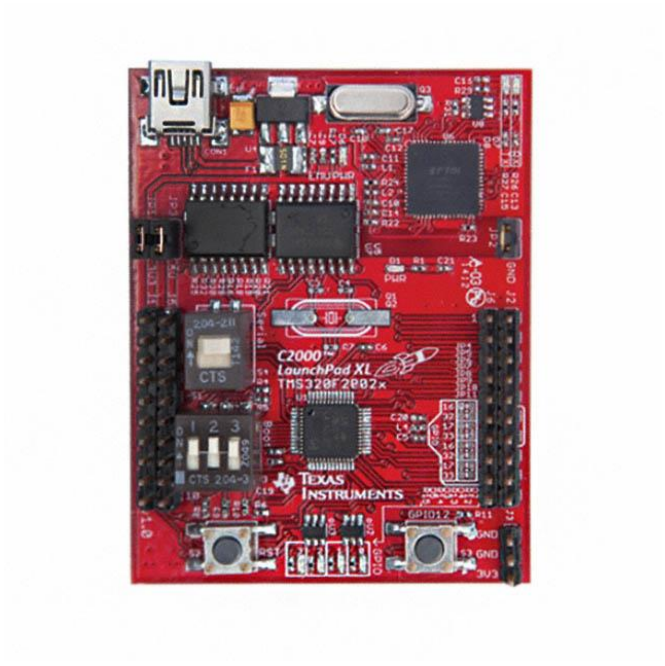


Figure 13: LAUCHXL-F28027 development board

After testing and debugging the LAUNCHXL-F28027 for several weeks on the receiver side, it was determined that the onboard ADC was not taking samples in a predictable manner. It was imperative that we could control the sampling rate in order for the receiver module to decode a message correctly. To compensate for this problem, the decision was made to switch to an actual microcontroller whose family the team was more familiar with, rather than to continue using the digital signal processing chip. Although our value analysis indicated that the TMDX28069USB would have been our second choice, we decided on the MSP430F5529 due to our familiarity and prior experience with the MSP430 microcontroller family.

For our purposes, the main significant difference between the MSP430F5529 and the LAUNCHXL-F28027 is the sampling rate of the ADC. The MSP430F5529 microcontroller lists a sampling rate of about 200 kbps, significantly less than that of the LAUNCHXL-F28027 sampling rate of 4.6 Mbps. However, due to an adjustment in goals outlined in Chapter 4, this high sampling rate was no longer necessary. Thus, we opted for a slower sampling ADC that had predictable functionality over a faster sampling ADC that had unpredictable behavior.

2.5.2 Setting up Code Composer Studio to Run the DSP

In order to connect the LAUNCHXL-F28027 to the computer, the program Code Composer Studio was used to debug and load code the DSP. When a project was created, a few settings were configured so the computer could recognize the development kit. To configure the project to function with the DSP, it was needed to navigate to the projects properties section. This was done by right clicking on the project and going all the way down to properties. Once the properties menu was loaded, under the general tab, there is a section called Connection. Next to Connection there is a drop down menu where Texas Instruments XDS100 Integrated USB is selected. Above this is a section called variant. From this drop-down menu, TMS320F28027 is selected.

After all the settings were configured in the properties menu, CCS needs to be connected to the target device. To do this, you must navigate to the top Code Composer tabs and select Window. A drop down menu will open, which is where you will select Show View. Once selected, another set of options are presented. This is where Target Configuration is selected. When it is selected, a side panel will open, and initially will only contain 'Projects' and 'User Defined' items. If you right click on 'User Defined', a set of options will be given. The first option, New Target Configuration should be selected to create a new target. When selected, a pop up menu will be shown to select the desired configurations. First, the program will ask to name the configuration, and it is up to the user. In the new menu screen, there is a section called Connection. This is the same as the 'Connection' item in the properties section and the same value, Texas Instrument XDS100 Integrated USB, should be selected. Below 'Connection', there is a section called 'Board or Device'. There will also be a list present, which will contain different boards that connect with the same connection. The list was navigated through to select Experimenter's Kit – Piccolo F28027. Once selected, navigate to 'Save Connection' and click the 'save' button.

Once the connection is set, it will be present in the 'Target Configuration' pane. To activate the connection, it was required to right-click the connection and select 'Launch Selected Configuration'. Once configured correctly, it was required to navigate to Code Composer Studio's taskbar and select Run. There the 'Connect Target' option is selected to fully connect the board to the computer.

2.5.3 Pin Configuration

On the LAUCHXL-F28027 development kit, there are a total of 40 input and output pins. Some pins are dedicated strictly for general purpose I/O pins, but some are configured to be inputs to the ADC, or Pulse Width Modulation Outputs. The MSP430F5529 Launchpad also has an assortment of input and output pins. Similar to LAUNCHXL-F28027, the MSP430F5529 has dedicated pins for its ADC inputs and general purpose I/O pins. Below are tables that represent how both development boards' pins are configured.

Mux Value				J1 Pin
3	2	1	0	
			+3.3V	1
			ADCINA6	2
TZ2	SDAA	SCIRXDA	GPIO28	3
TZ3	SCLA	SCITXDA	GPIO29	4
Rsvd	Rsvd	COMP2OUT	GPIO34	5
			ADCINA4	6
	SCITXDA	SPICLK	GPIO18	7
			ADCINA2	8
			ADCINB2	9
			ADCINB4	10

J5 Pin	Mux Value			
	0	1	2	3
1	+5V			
2	GND			
3	ADCINA7			
4	ADCINA3			
5	ADCINA1			
6	ADCINA0			
7	ADCINB1			
8	ADCINB3			
9	ADCINB7			
10	NC			

Mux Value				J6 Pin
3	2	1	0	
Rsvd	Rsvd	EPWM1A	GPIO0	1
COMP1OUT	Rsvd	EPWM1B	GPIO1	2
Rsvd	Rsvd	EPWM2A	GPIO2	3
COMP2OUT	Rsvd	EPWM2B	GPIO3	4
Rsvd	Rsvd	EPWM3A	GPIO4	5
ECAP1	Rsvd	EPWM3B	GPIO5	6
TZ2/ADCSOCA	Rsvd/EPWMSYNCI	SPISIMOA/SDAA	GPIO16/32	7
TZ3/ADCSOCB	Rsvd/EPWMSYNCO	SPISOMIA/SCLA	GPIO17/33	8
			NC	9
			NC	10

J2 Pin	Mux Value			
	0	1	2	3
1	GND			
2	GPIO19	SPISTEA	SCIRXDA	ECAP1
3	GPIO12	TZ1	SCITXDA	Rsvd
4	NC			
5	RESET#			
6	GPIO16/32	SPISIMOA/SDAA	Rsvd/EPWMSYNCI	TZ2/ADCSOCA
7	GPIO17/33	SPISOMIA/SCLA	Rsvd/EPWMSYNCO	TZ3/ADCSOCB
8	GPIO6	EPWM4A	EPWMSYNCI	EPWMSYNCO
9	GPIO7	EPWM4B	SCIRXDA	Rsvd
10	ADCINB6			

Table 12: Pin Layout for the LAUNCHXL-F28027, Provided by TI [12]

2.5.4 Configuring Transmitter and Receiver Pins

The LAUNCHXL-F28027 was used on the transmitter side as an intermediate step to convert the digital data of 1s and 0s to analog pulses, which would then be sent to the LEDs to transmit the information. Once the data was converted to analog pulses, the signal was outputted from J6 pin 1. As seen from the tables above, J6 pin 1 can be configured as a Pulse Width Modulation output. This was done to be able to send out signals that continuously changed from high to low, depending on which bit was being represented. Another pin that was used on the transmitter was a ground pin. With no reference, the circuit would not function properly.

The MSP430F5529 was configured for the receiver. The analog input coming from the photodiode and gain stage was connected to the board's ADC input pin. The P6.0 pin was used as the input to the ADC. The ground pin was also connected to the analog ground to provide a ground reference to the processor.

2.6 Digital Side of Transmitter

The American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that is widely used in representing text when transmitting bits [13]. For our encoding scheme, we used 7-bits and used an ASCII scheme containing the corresponding ASCII code for all 26 letters in upper and lower cases, the code for the 10 Arabic numeral digits, as well as code for commonly used symbols and spaces.

For the transmission of text, we used a header bit sequence corresponding to the text "This is a testThis is a test", which contains 28 characters, representing 196 bits. This header sequence is necessary in order for a receiver to know when the transmission of the message actually starts. The initial test message being sent is "Hello World", containing 11 characters, corresponding to 77 bits. This message size remains fixed for various different messages. The target data rate was 1 Mbps, but for testing purposes it has been significantly reduced to a measurable rate of < 1 kbps. Each bit that represents the header and message will be held at the transmitter's output for a given period. The period can be calculated by the following equation:

$$T_B = \left(\frac{EPW1\_TIMER\_TBPRD}{10}\right) \mu s \tag{4}$$

The parameter, EPW1_TIMER_TBPRD, is defined in the transmission code to alter the output signal's frequency. In order to get the transmission frequency of 1kHz, the parameter's value needs to be set to 10,000. From equation (4), the resulting period corresponds to 1ms when EPW1_TIMER_TBPRD is equal to 10,000.

2.7 Digital Side of Receiver

After the DSP obtains samples from the photodiode analog circuitry, the receiver needs to know when the message sequence actually starts, as well as the determination of logic highs and logic lows when the values acquired by the ADC of the DSP are discretized. Since the reference voltage of the 12-bit ADC is 3.3 V, the corresponding voltage for each discretized value is the following relationship:

$$Discretized\ Value = \frac{Input\ Voltage}{3.3\ V} * 4096 \quad (5)$$

Since there will be noise from the ambient light of the room, the logic low will not be right at 0 V. This value fluctuates, depending on the ambient light. In order to determine the threshold of low to high, we experimentally determined the range of discretized values using the watch window of the IDE Code Composer Studio, and select roughly the median as the threshold voltage. Values higher than this threshold are considered logic high, and values below the threshold are considered logic low.

2.7.1 Determining when Data is Being Transmitted

In any communication system, the transmitter and receiver are not constantly transmitting. When no transmission is being sent, both the transmitter and receiver are in an idle state. Once the transmitter finishes outputting all the necessary information, the output voltage will be set to zero to keep the LEDs off. The receiver should stay in the idle state until it determines that information is being sent, which it then will go into receive mode and store information into memory. One way to determine if any data is being transmitted is to look at the incoming energy levels. The receiver will perform a moving average by constantly sampling the incoming signal and storing a certain amount of samples into an array. The values in the array are then averaged and compared to a threshold. If the value is above the threshold, then the receiver believes data is being transmitted and starts storing the data into memory until the average goes below the threshold. When all the data is collected, the data will then be outputted to a file to be used in MATLAB to locate where the data is located and to decode it back into ASCII text.

Once all the data is collected, it is possible to use a break point to output the stored data values to a file. Normally, break points are used to halt the code at a certain line of code. However, Code Composer Studio allows the user to change the actions of the break points. By accessing the break point's properties, there is a parameter called Actions. Under this parameter, there is an option to save data to a file. A few options arise once this option is selected. The first is called file and it determines where the desired file to store information is located. The next option is format which determines which data type the data will be stored. For this project, the integer data type was used. After the format option is the start address option. The start address option allows the user to specify where the first data point is stored in the memory. The final option is length which can be adjusted depending on how many data points need to be exported. Figures 34 and 35 in Appendix D represent the break point properties menu.

2.7.2 Decoding Data

2.7.2.1 MATLAB functionality

Once the receiver determines that data is being transmitted, it will store the sampled values from the ADC into flash memory. Since the message's length in known, the transmitter will collect the given amount of samples until the entire message was sent. As mentioned earlier, once all the data is collected, the stored information is extracted to a text file. Once the data is in the text file, a MATLAB script can be executed to decode what message was sent.

The MATLAB function, importData(), can retrieve information from a text file and store the data in an array. The collected data from the MSP430F5529 is a representation of what was sent from the transmitter however it is not exact. Due to surrounding ambient light, there is a noise level that has been added as a constant DC voltage to the received signal. This DC voltage can be considered as some type of threshold. To get the data back to zeroes and ones, the collected data must be compared to a set threshold. If the value is higher than the threshold, the sample will be considered a representation of a one bit. If lower than the threshold, the value will be considered a zero. For this to function properly, the amplitude from high to low in the incoming signal needs to be sufficient enough so that samples will not be misinterpreted. When testing, the threshold was determined by adding half the amplitude to the lower voltage level. Using a 'for' loop, the collected data was compared to the threshold and a new array was created that contained only ones and zeroes.

The data still needed to be processed further because the ADC was taking multiple samples per bit. Here a form of down sampling must occur to only have one sample per bit. Since the ADC was not consistent in sampling each bit, downsampling was done by a frame by frame basis. A few parameters, old_start, start, old_stop, and stop, were used to keep track of when the received data changed from a zero to a one and vice versa. The entire data was shifted through using a 'for' loop. In the loop, the most current sample is compared to its previous and next sample. If the values were not the same, the parameters, start and stop, stored the location of the sample. The other parameters, old_start and old_stop, take on the previous starting and stopping indices.

After the indices are calculated, an 'if' statement is used that determines if old_start is not equal to start and if old_stop is not equal to stop. If the statement was true, the section of the received data between start and stop is extracted. The extracted data's length is calculated using the MATLAB function length() to determine the number of samples that were present in the data. Another if statement is used to determine if the number of samples in the section is greater than 6. Since it was known that the ADC will produce either 6 or 7 samples per bit, the extracted data’s length was divided by 7. The MATLAB function, ceil(), rounds this quotient to the next highest integer number.

The function was needed to determine the number of bits present in the selected window. For example, if the window’s length was 19, when divided by 7, the quotient is approximately 2.71. The ceil function would then round up the number to 3. If the number of samples is 19, it most likely means that two bits were sampled 6 times and one was sampled 7 times. Once the number of bits that the window represents is calculated, an array the size of the number of bits is created. The first value from the window is then stored in the array because all the values in the window have the same value. The array containing the down sampled version of the window is then added to an array that contains the previous down sampling results.

Once the data has been down sampled, the starting bit location needs to be determined to extract the sent message. Since it is known that the sent message contained the phrase “This is a test”, there is a known bit sequence in the data that can be used. To locate where the known sequence is, the xcorr function in MATLAB was used. The function xcorr can be used to correlate two sequences together. The function was used to correlate the received data with the known bit sequence. The correlated output sequence will have a maximum point when the known sequence is overlapping itself in the data. Once the data was correlated, the find function was used to locate the maximum point. In the bit sequence that represents the header phrase, there is a total of 45 one’s. The find function was used to determine the points that had a values greater than or equal to 45. Once the index for the start of the header is located, the length of the header plus one must be added to determine the starting index for the received message.

Once the starting index of the message was found, it became possible to extract the entire message bit sequence because the number of characters sent is known. In the case of sending the test message “Hello world”, there is a total of 11 characters. The ASCII coding takes each character and converts it into seven bit sequences. That means there is a total of 77 bits that correspond to the transmitted message. Knowing these two parameters allows for the extraction of the bits that represent the message. Once the data is extracted, the MATLAB function created in a prior course, bin2text(), was used to convert the data back into ASCII characters.

2.7.2.2 Testing MATLAB functionality

Before testing with data collected from the receiver, another MATLAB script was created that would mimic the received data from the receiver. A random data vector of zeros and ones, called data, was created to model any random bits that may have been collected from the receiver. This was done by using the MATLAB functions, rand and round. The function rand is a uniformly distributed number generator of values between zero and one. The round function takes those randomly distributed values and rounds them to a zero or one. The header, “This is a big testThis is a big test”, and message, “Hello world”, were converted into their binary representation using the text2bin function.

The binary data out of text2bin needed to be upsampled to model the sampled received data from the receiver. A for loop was used to go through each bit and upsample it by either a 6 or 7. The rand function was used again with some other functions to produce either a 6 or a 7. Depending on which number was selected, a vector was created that contained either 6 or 7 values of the most recent bit from the binary data. The vector was then concatenated to another vector that contained the previous vectors from the loop’s process. Once the data was upsampled, it was placed into the randomly generated data vector. This vector was then used with the decoding MATLAB script to determine if the output message would be the message, “Hello world”.

Chapter 3: Product Results

This section will describe our test results, as well as what was actually achieved with our implementation. Firstly, we will discuss our results from the analog circuitry from the transmitter and receiver. The received analog signal after the voltage gain stage at the receiver was monitored to determine what was being sent to the microprocessor. The transmitter was slowly distanced from the receiver to determine the transmission distance. By displaying the received signal on an oscilloscope, it was possible to determine when the transmitter had no effect on received signal.

3.1 Analog Results

To test that the receiver functioned properly, an LED was connected to the amplifier's output. In this test, two photodiodes were tested; the SFH203P and the SFH229. The first experiment performed was to test if the receiver can recognize a change in light from the receiver. To do this, a manual switch was used to turn the transmitters' LEDs on and off. While this occurred, the LED on the receiver was observed to determine if a change in its status occurred. Initially, LED would remain on, even when the receiver was turned off. It was then determined that the photodiode was absorbing light from the ambient light that was lighting the room. To verify this hypothesis, the lights in the room were turned off and, as expected, the receiver LEDs shut off accordingly. The experiment was then repeated with the room lights off, resulting in the receiver's LED to turn on when the transmitter turned on.

Once the receiver's LED was switched on by manually turning the transmitter on, the function generator was added to the transmitter to supply a square wave. Initially, the square wave's frequency was set to 10 Hz to determine that the LEDs were in fact constantly turning on and off. Again in a dark room, the receiver's LED flickered. Once it was determined that the receiver could recognize the transmitter, the distance between the two systems was increased until the receiver was no longer able to recognize the transmitter. With seven LEDs on the transmitter, the maximum distance for transmission was measured to be 39.5 cm.

Through experimentation, it was observed that the receiver recognized the transmitter better when the photodiodes were pointed directly to the transmitter's LEDs. This is due to the photodiode's low angle of visibility. The two photodiodes that were tested, the SFH229 and the SFH203P, have a half angle lumination detection of $\pm 17^{o}$ and $\pm 75^{o}$ respectively. While both photodiodes have similar specifications, the SFH203P functioned better due to its higher half angle. Because of this, the SFH203P photodiode was used when collecting measurements.

When the room was lighted, the added light would cause the photodiode to emit a constant current, which resulted in a DC threshold voltage at the amplifier's output. The constant voltage was enough to power the LED at the receiver. Knowing that there is a constant DC voltage, when the received signal enters the microcontroller to be sampled, some programming must be done by setting the DC threshold.

After determining that the receiver could recognize the transmitter, the amplifier's output was measured using an oscilloscope. Initially, the transmitter was set to transmit a square wave with a frequency of 10Hz. The following figure represents the receiver's output from the amplifier when the transmitters' LEDs were emitting light.

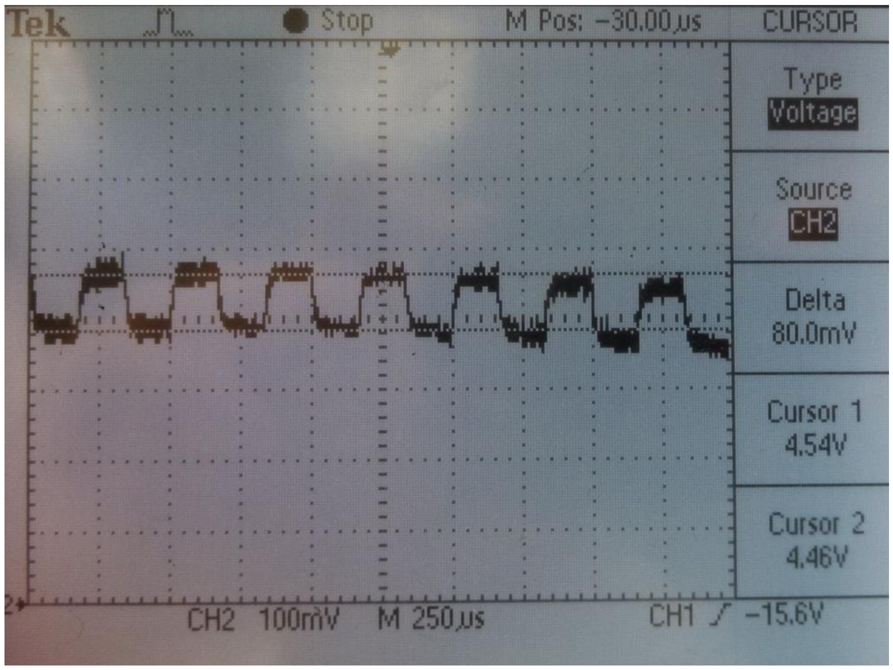


Figure 14: Square wave output measured by the receiver

The receiver was able to recreate the square wave that the transmitter sent. However, even after the amplification from the amplifier, the distance from high to low was only 80 mV. Also, the signal contained some noise, but not enough to cause the square wave signal to become indistinguishable. To get a more accurate square wave output, it was decided to replace the 1 MΩ resistor with a 5.1 MΩ resistor. The following figure represents the receiver's output when the transmitter was 27.5 cm away from the receiver.

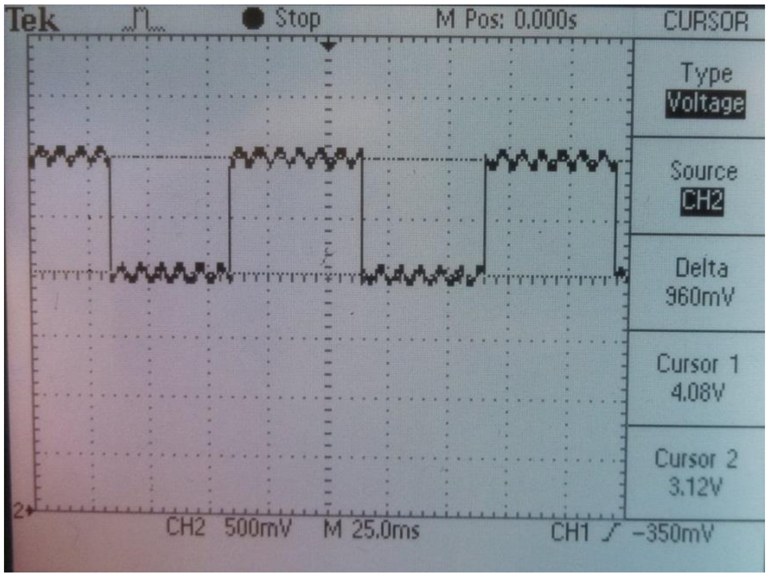


Figure 15: Receiver output with 5.1MΩ resistor

The receiver's output continued to contain some noise, but the square wave was still distinguishable. Because of a higher resistor, the amplifier contained a higher gain, thus increasing the square wave's amplitude. The square wave's amplitude was measured to be 960 mV, which is much higher than the 80 mV amplitude from before.

After the square wave was found, the distance between the transmitter and receiver was increased until the square wave was no longer distinguishable. The following figure represents the receiver's output when the distance between the two systems was 66 cm.

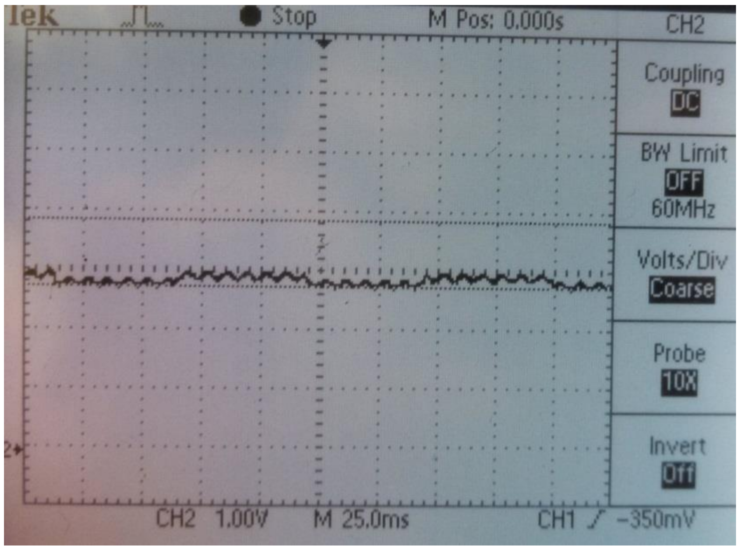


Figure 16: Max Distance when square wave is undistinguishable

From the above figure, the amount of noise present would make it difficult for the microprocessor to determine the appropriate ones and zeros. The distance measured is very close to the distance goal of 1 m we set for the prototype. One way to fix the change would be to increase the number of LEDs at the transmitter. Since we did not have enough LEDs at the time, but had many photodiodes, photodiodes were placed in parallel at the receiver end. On top of adding more photodiodes, the amount of gain was adjusted from the amplifier to determine the best design. The first adjustment we made was using two SFH203P photodiodes along with a 505 kΩ resistor. The following figure represents the receiver’s output when the adjustments were made.

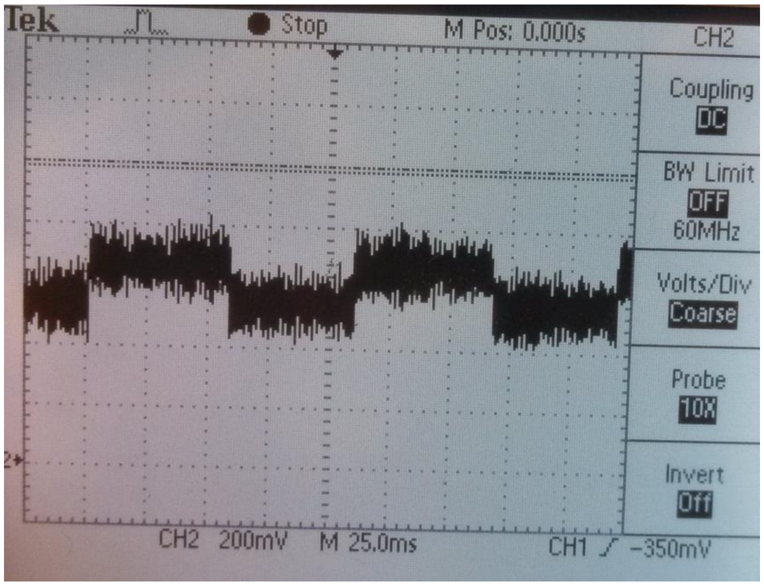


Figure 17: Receiver's output with two photodiodes and 505kΩ resistor

The receiver’s output had a smaller amplitude than before, which was expected because the gain was reduced. On top of that, there was much more noise when an extra photodiode was used. The resistor was changed to a 3.24 MΩ resistor, but the output had more noise than the initial test. Its amplitude was also smaller, about 100 mV. Because of this, we decided to use the 5.1 MΩ resistor with only one photodiode.

After determining which circuit configuration functioned better, the frequency at the transmitter was increased. When the frequency was increased to 3.2 kHz, the receiver’s output contained some ringing. The following figure represents the ringing observed at the output.

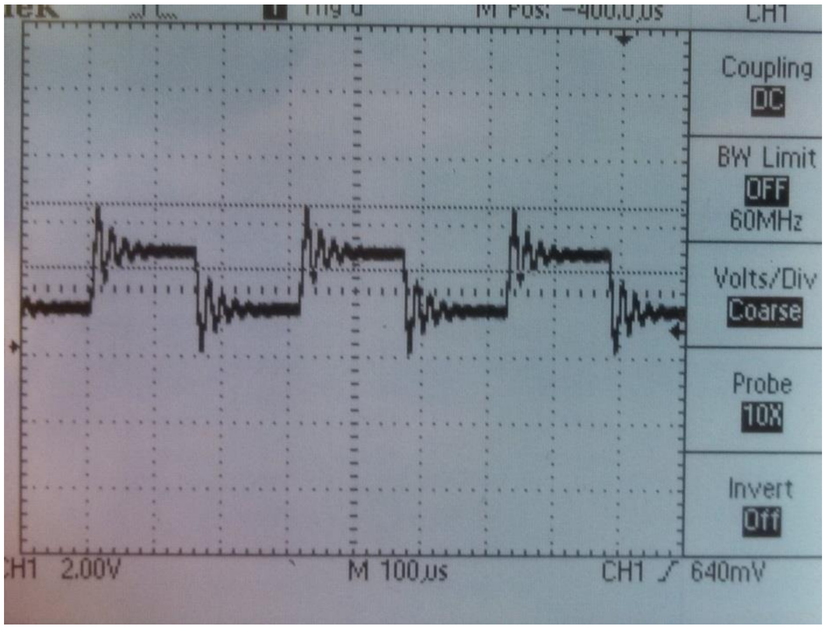


Figure 18: Ringing observed at the receiver's output

While some ringing was present, it was not enough to keep the square wave from being distinguishable. The frequency was increased more until the square wave could not be seen at the output. The following figure represents the receiver's output when the transmission frequency was set to 6 kHz.

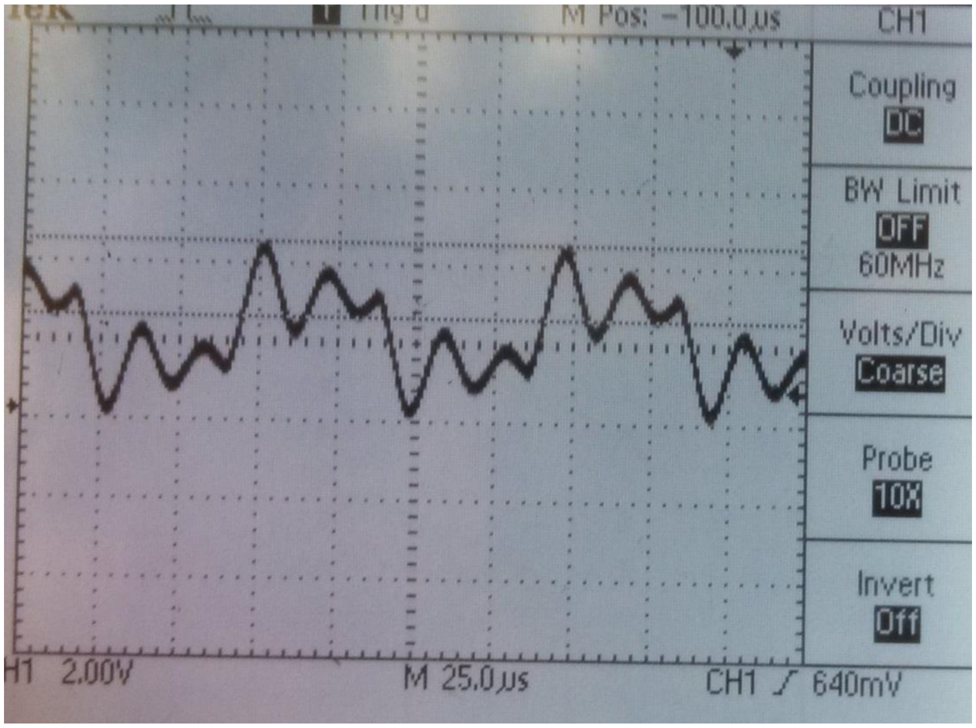


Figure 19: Noisy Receiver Output

Because the receiver's output contained a large amount of noise, which hindered the maximum frequency possible to transmit, a 10 pF capacitor was placed in parallel with the resistor. The following figure represents the new receiver circuit schematic.

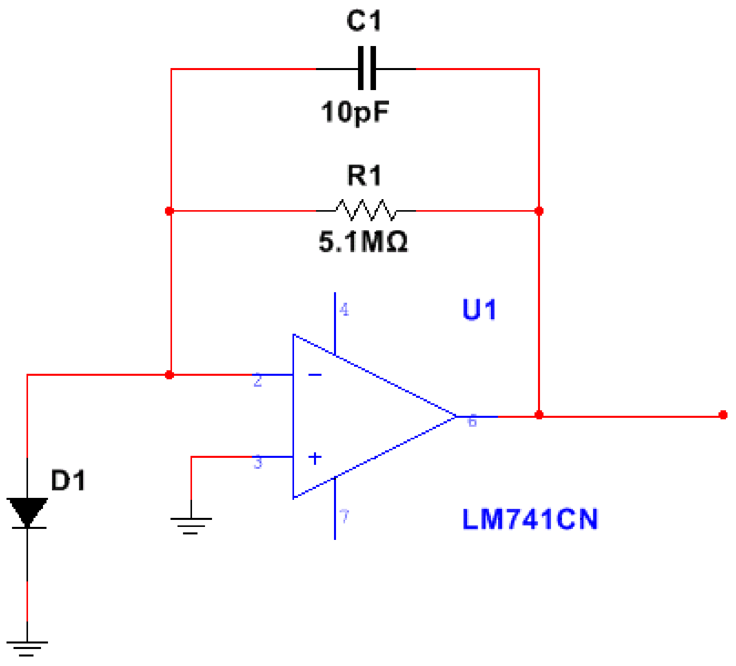


Figure 20: Receiver Schematic with capacitor

The capacitor helped eliminate the noise at the receiver's output, but there was a slight rise time that occurred by the RC time constant. The following figure represents the receiver's output when a capacitor was added.

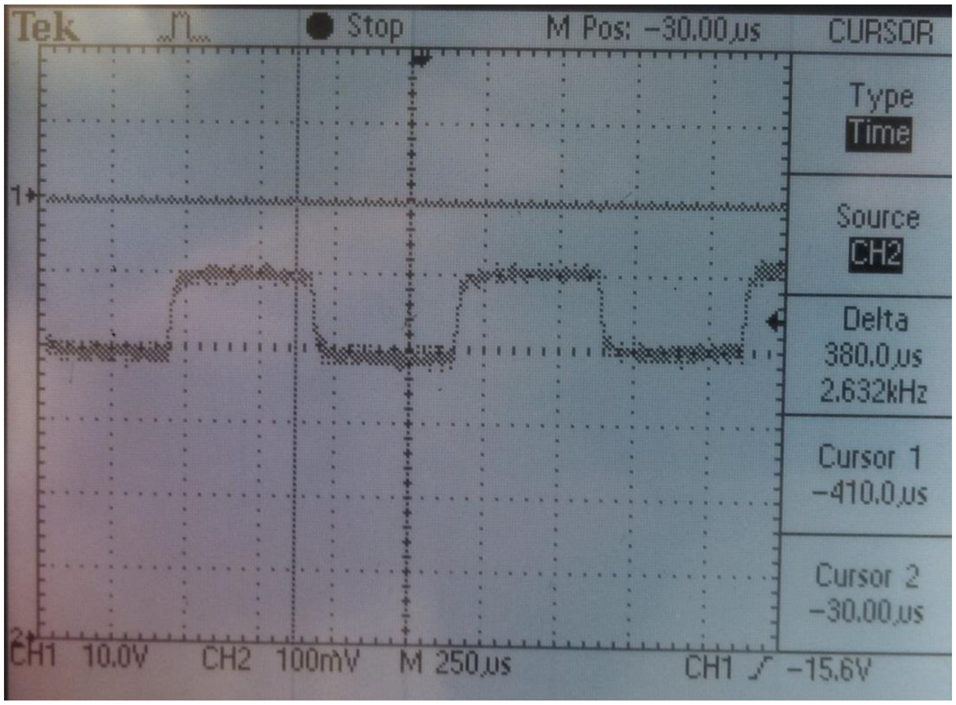


Figure 21: Receiver's output with a capacitor

Given that a 5.1MΩ resistor and a 10pF resistor were used in the circuit, the RC rise time was calculated to be:

$$t_r = RC = (5.1\ M\Omega)(10\ pF) = 51\ \mu s \qquad (6)$$

Since it took 51 µs to rise from low to high, if the transmitter would operate at a frequency higher than 20 kHz, the receiver would not be able to keep up with the transmitted data. Since 20 kHz is much smaller than our goal of at least 1MHz, we initially decided to look at other options to improve our transmission rate such as using a different op-amp with a higher unity-gain frequency.

3.2 Microcontrollers

In this section, we will discuss how the microcontrollers interact with the computer and analog circuitry for both the transmitter and receiver. On the transmitter side, the microcontroller is mainly used to convert binary values to an analog signal to drive LEDs. The microcontroller's output was monitored by an oscilloscope to get a better understanding of how the LEDs are being driven. The microcontroller on the receiver side receives an analog signal and quantizes it to retrieve the sent information. We will also discuss how the ADC was used to take the input analog signal and convert it into values of ones and zeroes. Once the discrete values were calculated, the start of the message must be located to convert the bits back into the transmitted information.

3.2.1 Transmitter

To test the functionality of the LAUNCHXL-F28027, bit values were first hardcoded into the board. Standard test values used were alternating 1s and 0s, and then the bit sequence 1000.

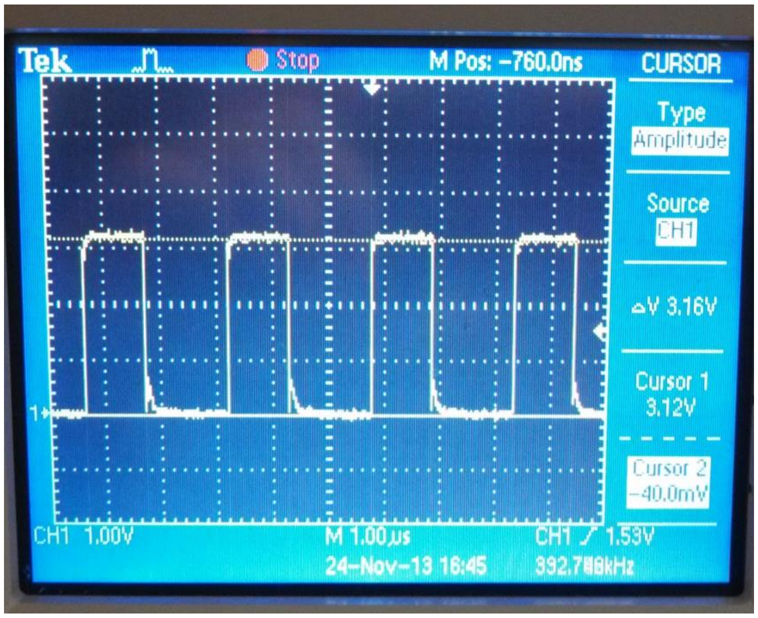


Figure 22: Output of Transmitter with Frequency of ~400 kHz

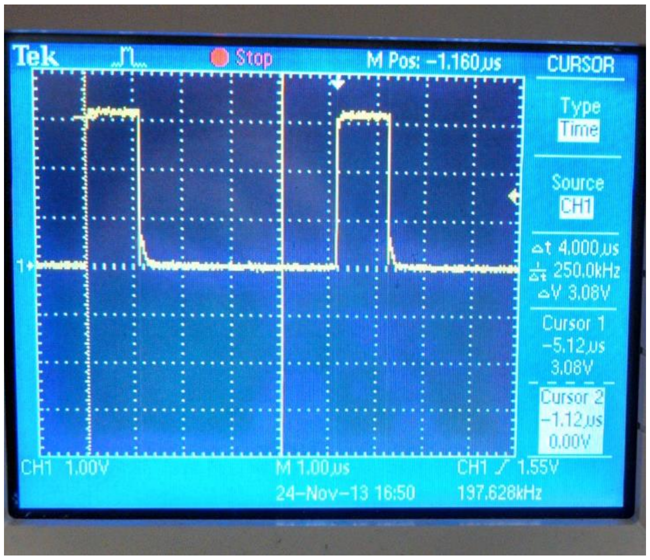


Figure 23: Transmitter Outputting 1000

The frequency of the transmitted bits is controlled by the sample epwm_up_aq code provided by TI. The values for EPWM1_TIMER_TBPRD and EPWM1_MAX_CMPA were altered to change the frequency of the pulses. Using values of 10 for each, a frequency of approximately 900 kHz was achieved. Since the board uses a reference voltage of 3.3, the output is a 0-3.3 V square pulse.

Once we created a bit scheme from the ASCII conversions of commonly used characters (see Appendix B), we were able to send a stream of varying bits, based on the message being sent. The bits being sent in Figure 24 correspond to a header of ‘Th’ and a message of ‘z’, and is being sent continuously. Eventually, the header being sent will correspond to the sequence “This is a big testThis is a big test” and the message will be “Hello World”, or other 11-character text sequences.

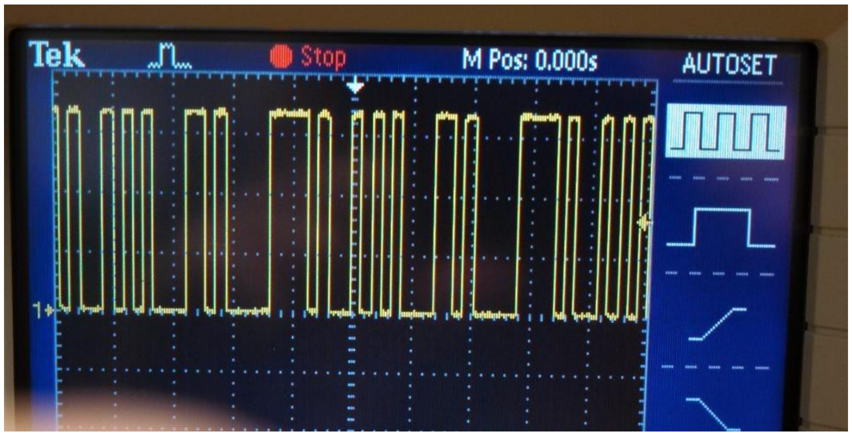


Figure 24: Transmitted Bit Scheme from ASCII Conversion

In order to observe that the transmitter was actually transmitting the bit sequence that corresponds to ‘This is a big testThis is a big testHello World’, the transmission frequency was lowered to a frequency of 1 Hz to observe when the LEDs were on and off, since the human eye can distinguish whether flickering lights are on or off for frequencies up to around 20 Hz. With such a long sequence, it was easier to time the blinking and determine if the transmitter was operating as expected using human observation rather than the oscilloscope, as an oscilloscope has a limited data viewing window.

When adjusting our transmission frequencies, equation (4) provides a fairly accurate estimate of our transmitted bit symbol period, although it is not extremely precise. Thus, we also tweaked the value of EPW1_TIMER_TBPRD slightly and measured the symbol period with the scope in order to set the symbol period to exactly what was desired.

After significant issues with the programming of the microcontroller for the receiver, our goals were modified to be more feasible, given the unexpected complications. Although we could ensure approximately 1 MHz transmission rates on the transmitter side, which met our original goal of transmitting on the rate of 1 MHz, we could not fully handle this high rate on the receiver side. The analog circuitry was able to process this quickly, but due to the ADC sampling rate issues described in Section 3.2.2 Receiver and Section 4.1 Digital Issues, we decided to lower our transmission rate to 1 kHz in conjunction with our new microcontroller, the MSP430-F5529. This 1 kHz transmission frequency worked well for initial testing with the new microcontroller; however, as we built upon the receiver, it was again necessary to lower the transmission frequency again to 500 Hz, for reasons outlined in Section 3.2.2 Receiver.

3.2.2 Receiver

Figure 25 illustrates the output from the analog portion of the receiver, prior to the signal entering the ADC.

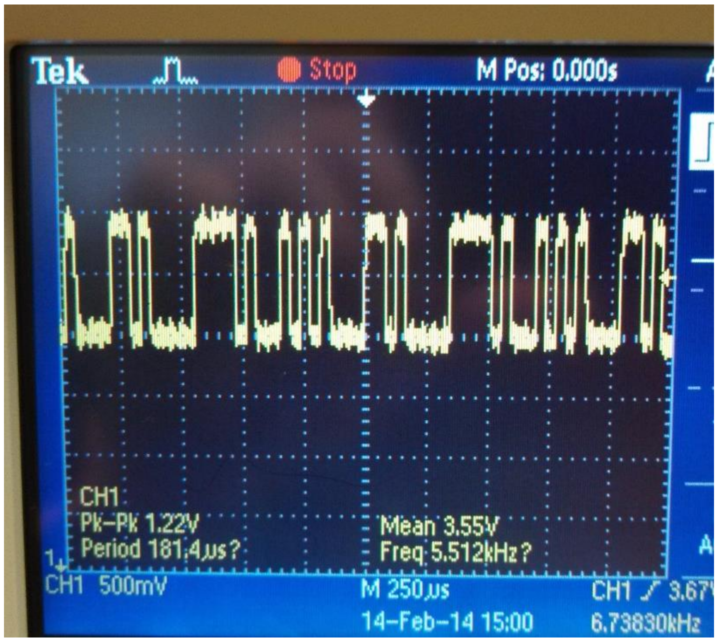


Figure 25: Analog Output of Receiver

The bit sequence can be seen to match the sequence being transmitted. There is a bit of ringing that occurs, due to the noise from the ambient light. The noise also affects the high and low voltages. Instead of a 0 V low, the lower voltage will vary depending on the amount of noise in the room. The upper voltage can also vary depending on the transmitter's signal strength. If this is the case, the amplitude between the high and low voltages will diminish.

The original chip used for the digital processing of the receiver signal was the LAUNCHXL-F28027. From testing, typical voltages that we have dealt with are around 2.2 V for logic low, and 3.3 V for logic high. Initially for testing purposes, we used the General Purpose Input/Output (GPIO) pins to display what was being detected by the microcontroller from the analog receiver block. In order for the microcontroller to distinguish between a logic high and a logic low, a threshold voltage was needed. We set this threshold voltage to be approximately the average of this range of voltages, 2.7 V. To calculate the corresponding quantization level for a certain voltage level, the equation 5 was used, resulting in a value of 3351. For our finalized design, the method of obtaining the threshold was simply to take the average of the quantized values from the ADC.

The initial plan was to store each sample from the ADC in an array. Then, these samples could be processed in several ways in order to produce a decoded message for the computer. One such plan was to take the samples from the ADC now stored in an array, correlate with a known sequence, downsample the ADC samples, and decode these samples in a similar fashion as we had originally encoded them on the transmitter side. Several issues arose from this implantation that we did not foresee.

First, memory with the board was an issue. When attempting to store any more than roughly 1000 samples, Code Composer Studio would allow compilation of the code, but would not allow execution of the code. CCS did not specifically indicate that memory was the issue, and simply gave an error that was not very intuitive in distinguishing the source of the problem. Since the error occurred upon execution rather than upon compilation, the only reasonable conclusion was that the issue was with hardware constraints rather than software constraints. When the size of the array was decreased to around 600-700 samples, the code had no issue executing. Regardless, these sized arrays were not nearly enough to store, process, and decode a reasonably length header and message from our transmitter. Our header, “This is a big testThis is a big test”, is 36 characters long, which would correspond to an ASCII bit sequence of 252 bits (7 bits per character). The message to be sent, “Hello World”, eleven characters long, is a bit sequence of 77 bits, for a total of 329 bits. It was necessary to send sample each of these bits several times (ideally, at least 10) to ensure accuracy; thus, if we only had around 700 samples to work with, we could only sample each bit roughly 4 times, which is undesirable. Since these samples used most of the resources available on the LAUNCHXL-F28027, it would then be impossible to correlate, downsample, and decode the samples on the board as well.

Secondly, the sampling frequency was not operating as we expected. When transmitting at extremely low frequencies of less than 500 Hz, we were able to sample the signal from the analog receiver module relatively accurately, as well as output this signal using the GPIO pins on the board.

When the transmission frequency was increased, the output frequency of the signal from the GPIO pins did not increase proportionally. In fact, the receiver output frequencies would cycle between 100-400 Hz as we increased the transmission frequency. It was assumed that the GPIO pins were not programmed in such a fashion that could output the samples quickly enough. Since this was not a critical part of the system, rather than using the GPIO pins to output the samples, the watch window in Code Composer Studio was utilized instead to observe proper functionality of the ADC.

The switch to troubleshooting via watch window, although an improvement in being able to measure and observe functionality, was not a cureall for the issues regarding the ADC. At frequencies on the order of 100 kHz or higher, we would observe an uneven number of ADC samples per bit. This occurring once every 200 samples or so would not be a significant issue, but this occurred at least once every 10 samples or so, resulting in undecipherable data. The frequency was lowered until it was observed that there was roughly the same number of samples per bit for an extended number of samples, after roughly 200 samples. The test inputs were square waves from the function generator with a 50% duty cycle. When these square waves were at frequencies of 20 kHz and 10 kHz, the ADC was mostly accurate except with a few blips. Unfortunately, we did not figure out a way to completely control the sampling frequency of the ADC, which was a significant handicap to the correct operation of the LAUNCHXL-F28027 as our digital receiver.

With these issues, two alternative courses of actions were proposed. First, to address the memory issue of the LAUNCHXL-F28027, the data could be exported to the computer, and MATLAB could be used to process the data with correlation, down sampling, and decoding to output the originally transmitted message. Alternatively, to address both the ADC sampling frequency issue as well as the memory issue, a MSP430 microcontroller could be used for the ADC. There are several pros to switching to an MSP430 microcontroller. In general, microcontrollers have more memory than digital signal processing chips. If there is still not enough memory on the microcontroller, then the data could still be migrated from the board to the computer for processing with MATLAB. In addition to having more memory, the team is more familiar with the MSP430 microcontroller family, making it significantly easier to control the sampling frequency.

It was decided to switch to the MSP430F5529 LaunchPad Evaluation Kit. The ADC sampling frequency was controlled by using a combination of the interrupt service routines and controlling the timers. Using the SMCLK frequency of 1.04 MHz for the timer, a counter is set to count to a value of 52 before an interrupt occurs, which corresponds to the maximum ADC sampling frequency of the

MSP430F5529, 200 kHz. After testing several transmission frequencies with our ADC, 1 kHz seemed to produce the best results, with a consistent 6 samples per bit for over 150 samples.

The memory allocation is configured differently on the MSP430F5529 than the LAUNCHXL-F28027. With the LAUNCHXL-F28027, the flash memory could be easily accessed and written into by simply switching an option in Code Composer Studio prior to building the code. In the MSP430F5529, specific lines writing into the flash needed to be included in the interrupt service routine. However, the only way we were able to make this operational was by combining our ADC interrupt service routine as well as the routine to write to flash into the timer interrupt. This expansion of the interrupt service routine slowed down our system tremendously. Although able to print the values on the console of Code Composer Studio, the ADC now was only able to output 3 samples per bit rather than 6 samples per bit at a transmission rate of 1 kHz. Our next alternative was to decrease the transmission frequency further, although this alternative is not ideal.

3.2.3 Decoding the Received Signal

The received analog signal was sampled to provide a certain number of samples per bit. Once the receiver data samples were received, the values are pulled from the memory of the MSP430-F5529 through the use of breakpoints, and stored into a .txt file so that correlation can occur through the use of the MATLAB function "xcorr". The function, xcorr, requires the two input signals to have the same number of samples. If they do not have the same length, the smaller signal is padded with zeroes to match the length of the larger signal. The resulting correlated signal has a length of 2N-1, where N is the length or the larger input signal. The correlated signals values can be represented by the following equation:

$$R_{xy}(m) = \sum_{n=0}^{N-m-1} x_{n+m} y^{*}_{n} \quad m \geq 0 \tag{7}$$

Similar to the calculation of correlation, the two signals are multiplied together and then added together. One of the input signals is conjugated and is shifted down one bit every time m increases. When correlating, the received data is correlated with the corresponding ASCII bit header sequence for "This is a big testThis is a big test". Correlation allows for the determination of the start of the transmitted message, as the output value of the correlation operation will be at its maximum at a certain index. By correlating our transmitted bit sequence by our header bit sequence, the result will be at its maximum at the start of the header.

After the start of the message is determined, the transmitted sequence, which is predefined in length, is downsampled by the same oversampled factor of the ADC, currently 6. These values, still quantized by the microcontroller ADC, are then converted into 0s and 1s by setting all values above a certain threshold to 1, and all values below a certain threshold to 0. For our purposes, this threshold was approximately 3351. The message was then ready to decode using a MATLAB script from previous coursework, 'bin2text', which converts ASCII bit sequences to characters, and will display the decoded transmitted message on the MATLAB console.

Chapter 4: Failure, Hazard Analysis, Limitations, and Future Improvements

Several issues occurred along the way of our design and implementation, causing many of our initial goals to change and adjustments were made accordingly to meet deadlines and absolutely necessary functional requirements. These ranged from power issues on the analog transmission side of the system, to digital issues on the digital receiver side of the system.

Our final system met several, but not all, of our initial design goals. While the system is operational, it is able to transmit text at a transmission frequency of 500 Hz at a transmission distance of roughly 25 cm without the implementation of our power source. Certainly, these achieved goals leave much room for improvement and extensions.

4.1 Digital Issues

There have been many issues with the programming of the C2000 Launchpad evaluation kits. TI has several versions of example code that may not be the most up-to-date set of files, which caused compilation errors, initialization errors, etc. Often times to circumvent this issue, it was necessary to re-download sets of files on different computers to achieve for any sort of functionality.

Code Composer Studio, the integrated development environment (IDE) used to program our boards would frequently have issues reading files or would not compile due to errors, but not display what the error was. One major problem was an unresolvable error that occurred consistently on line 18 of our code no matter the fix we tried. The line was commented out and the code would not compile even though it had no significance on the functionality of the code. Many times, restarting the IDE solved the issue, but sometimes a complete reinstallation was necessary.

Another problem in using the C2000 board we selected for our prototype was determining the sampling rate of the ADC as there is no simple way to specify this value. In order to set this value there had to be calculations done using different parts of the MCU to specify an approximate sampling rate. While testing, the output waveform from the ADC would not stay constant for various values of the transmitter frequency.

4.2 Analog Issues

4.2.1 LED Brightness

The original LEDs that were chosen and implemented in the prototype proved to be too dim to achieve a transmission distance of more than 20 cm. The initial goal of the design was be able to transmit data at a distance of at least one meter using solely visible light. In an attempt to achieve this goal, further research and value analysis on LEDs was conducted in hopes of finding brighter LEDs that fit the same specifications of the previous LEDs. The value analysis on LEDs can be found, and is explained more in-depth, in Section 2.3.1.1 LEDs.

Once the brighter LEDs were placed into the circuit the measurement was taken again to see if the transmission distance had improved as expected. The distance, however, did not improve by more than 10 cm. The LEDs were not receiving enough current from the MCU output to reach their brightest potential. In order to remedy this problem a Power MOSFET device was added with the purpose of supplying enough current to the LEDs, but only when the transmitted signal is 'on' or logic high. This ensures that the LEDs will only flash and activate the photodiodes when desired.

4.2.2 MOSFET Limitations

One version of our design involved using a MOSFET to increase the signal strength from the MCU that powered the LEDs on the transmitter end of the circuit. Our initial design is depicted below in Figure 26 but was quickly changed over to Figure 27 upon further investigation of how the MOSFET drain current works. In the first design the LEDs are connected in series with the drain of the MOSFET in an attempt to take the current from the MOSFET and power the LEDs when the device is on. However, being connected in series, the LEDs had no ground reference which made current flow impossible. After testing this in the lab, the design was quickly altered to that of the second picture to ensure the drain current flows through the LEDs as intended.

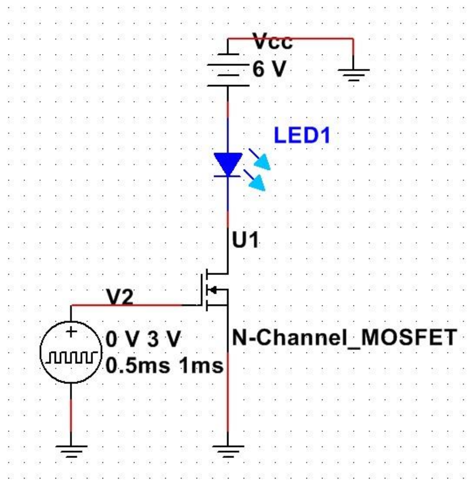


Figure 26: Original MOSFET Design Interfacing MCU Output and LEDs

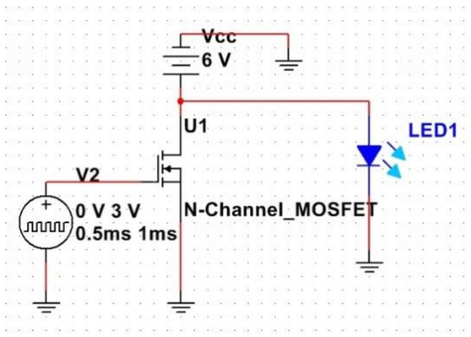


Figure 27: Updated MOSFET Design Interfacing MCU Output and LEDs

Another design fault was the lack of knowledge of using a MOSFET in a power application. Since the MOSFET is switching on and off at a high speed and has a high input capacitance, the logic output of the MCU does not supply enough current to charge the MOSFET gate fast enough. In order to bypass this problem, a Gate Driver was required to interface the two devices. This Gate Driver generates the current necessary to turn MOSFETs on and off from the input logic of a DSP or microcontroller. A lack of experience with MOSFETs in power applications was the cause of this problem and resulted in an inappropriate MOSFET for the design burning out during testing. When looking for a suitable Gate Driver it was found that the most of the devices available were surface mount which is not compatible with our design. Later in the design process, it was found the MOSFET had heating issues that caused failure in one of our boards so the MOSFET was excluded from the final design.

4.3 Future Improvements

4.3.1 Digital Improvements

Throughout the entire project, many of the issues that arose were from the digital components, the microprocessors. As mentioned earlier, it was needed to switch from the C2000 processor to the MSP430F5529 processor at the receiver because there were issues regarding sampling with the ADC on the C2000. The MSP430F5529 was a quick fix to the problem because it was familiar and available at the time of consideration. However, because its sampling rate is rather slow, it is not the best option for communication systems. In this section, we discuss other digital options such as FPGAs, and other digital signal processing chips.

4.3.1.1 FPGAs

A Field-Programmable Gate Array, or FPGA for short, is an integrated circuit that contains a large resource of logic gates and memory to implement digital computations. It is possible to customize the logic through a hardware description language such as Verilog. With an FPGA, it is possible to have parallel executions. This would allow the ADC to sample the incoming data without affecting any other process. Another process could take the data from the ADC and perform a spectral energy computation or even decoding the samples back into ASCII text.

FPGAs are also better suited for high frequency signals because the combinational logic inside the integrated chip typically can run as fast as the built in clock on the FPGA. In most cases, an FPGA's internal clock can be as high as 100MHz or higher. With the high frequency operation, it would be possible to achieve a higher transmission rate as long as the ADC that is used can sample fast enough. It is possible to choose which ADC can be used because the ADC can be an external module that will be interfaced with the rest of the FPGA development board.

Unlike a FPGA, a microcontroller performs its functionality sequentially. Since the microcontroller's ADC functions through interrupts, the time the interrupt takes to finish its process can have an effect on how fast data can be processed. In order to get a faster sampling rate, the number of computations in the interrupt needs to be done within as little processing cycles as possible or a faster processor may be needed.

While a FPGA may have been a better option than a microcontroller, there were a few major factors that deterred us from choosing one for the digital design of the transmitter and receiver. One factor was price. FPGA development boards tend to be very expensive. For example, The Xilinx NEXYS 3 development board costs a total of about $175. Given our limited budget, buying two boards would not have left any money for the analog components. The other factor was familiarity. At the time of choosing how to design the digital components for the project, the entire team was more familiar with using processors than FPGAs.

4.3.1.2 Better Processing Chips

When determining the initial microprocessors to use, the deciding factor to use the C2000 boards was their high sampling rate ADCs. Unfortunately, there were many issues that arose when using the board's ADC which is why it was discarded on the receiver side. Due to the limiting time, the MSP430F5529 was chosen to replace the C2000. While the MSP430F5529 may have not been the best option for the Visible Communication System, it was a quick fix to produce a working prototype.

If a microprocessor is going to be considered again for future Visible Communication Systems, there are a few factors that should be considered before selecting the specific chip. The first factor is the ADCs sampling rate. Without a fast sampling rate, the entire systems transmission's rate will be limited by the ADC. The second factor would be available sample code. One issue that arose with the C2000 was its lack of working sample code. By having sample code, it is much easier to design code for projects because there are models such as how to set an ADC to produce samples. The third factor is memory and processing speed. Without memory, it would be impossible to store the sampled data from the ADC. With a small amount of memory, the number of bits that can be transmitted at a time is limited because the memory has been filled. On top of having a sufficient amount of memory, it is ideal to have a fast processing processor. In order to decode the received message, the instructions to decode the received samples must run within a certain amount of samples that is not larger than the ADC's sampling rate. If it takes too long to initiate instructions, it could affect how fast the ADC is actually sampling. Since most ADCs operate using interrupts, if the interrupt takes longer to process than the ADC's sample period, the function is not operating in real time and can cause the receiver to not function properly.

4.3.1.3 Computer Interface

One of the main components of a Visible Communication System is its interface with other devices, such as computers or smartphones. A computer is an excellent source of interfacing the prototype system because the software that is used to program the digital side is a computer application. Since the processor used did not have enough memory to process the incoming data, the data had to be transferred to the computer to be processed in MATLAB. While it would have been better to process the data on the chip itself, there still needs to be a way to transfer the data to the computer. Due to the many issues that arose with transferring data to the computer, one short term fix was to export the collected ADC samples through CCS's console. Once on the console, it would be possible to copy the information to a file to be later processed by MATLAB. To make the system function better, it would be better to have the data exported instantaneously to a file. While attempting to create an instantaneous process, one source that was looked at was the MSP430f5529's sample code, emulStorageKeyboard.c.

The program was able to output data to a file depending on which button on the development board was pressed. We attempted to modify the code to run with the ADC interrupt to produce the data that would be outputted. Also the data would have been outputted to the file once all the information from the transmitter was received. Unfortunately, no progress was made with this approach. Because of this, other methods of transmitting data was looked into.

Another method that was looked into was the USB UART interface between the microcontroller and computer. With the UART interface, it was possible to send the data to a hyper terminal once all the information was received from the transmitter. A GUI interface could have then be used along with the hyper terminal to send the data to a file and initiate MATLAB to process the information. With the limited amount of time, it was decided to not pursue this option.

Ultimately, it was possible to export the received data instantaneously to a file. A break point was set in the code that would export the stored data in the flash to a specified file to be later used in Matlab. The break points functionality was altered by changing its actions in its properties menu. A more detailed description of how this was done can be found in Appendix D. Figures 34 and 35 represent screen shots of how to access the properties menu and the menu itself.

At the moment, there is not a very user-friendly way of inputting a message on the transmitter side to be sent to the receiver. Ideally, some sort of GUI can be implemented to send a text message, and these individual characters can then be stored in an array to be transmitted, either through Matlab or through Code Composer Studio.

In addition to transmitting just text, the system could be improved to transmit other forms of data, including audio data, image data, or even video data. In the GUI, there could be an option to upload the data from the computer, in order to process it. Of course, universal standards of encoding and decoding these forms of data should be observed to ensure that not only the computer on the transmitter side is able to contain the data to be transmitted, but also the computer on the receiver side is able to read the data that is processed and received. To do this, common extensions such as .mp3 or .wav could be used for audio, .jpeg or .bmp could be used for images, and .mp4 or .avi could be used for videos.

Implementation of a network would allow multiple users to simultaneously transmit or receive data. This could be done by using different colored LEDs to differentiate between users through filters, or have some array of mirrors or other ways of redirecting the light to a node through various positioning of transmitters and receivers.

References

[1] http://www.fcc.gov/what-we-do

[2] http://en.wikipedia.org/wiki/Mobile_phones_on_aircraft#The_debate_on_safety

[3] http://www.ece.iit.edu/~taher/dyspan11.pdf

[4] http://www.pwtc.eee.ntu.edu.sg/News/Documents/Spectrum%20survey%20in%20Singapore_%20Occupancy%20measurements%20and%20analyses.pdf

[5] http://cognitive2011vtm.trackchair.com/

[6] http://www2.technologyreview.com/article/405522/cognitive-radio/

[7] http://www.bakom.admin.ch/dokumentation/zahlen/00545/00547/00554/index.html?lang=en&download=NHzLpZeg7t,lnp6I0NTU042l2Z6ln1ad1IZn4Z2qZpnO2Yuq2Z6gpJCDdH18gGym162epYbg2c_JjKbNoKSn6A--

[8] http://lasercommunications.weebly.com/index.html

[9] http://www.sahyadri.edu.in/e-journal/laser.pdf

[10]http://en.wikipedia.org/wiki/Fluorescent_lamp#Advantages

[11]http://csep10.phys.utk.edu/astr162/lect/light/spectrum.html

[12]TI C2000 ADC Datasheet: http://www.ti.com/lit/ug/spruge5f/spruge5f.pdf

[13]ASCII: http://en.wikipedia.org/wiki/ASCII

Appendices

A: Parts List

Manufacture #	DigiKey #	Quantity
MCP6022-I/P	MCP6022-I/P-ND	3
LAUNCHXL-F28027	296-34797-ND	2
2482	2482K-ND	4
AD848JNZ	AD848JNZ-ND	1
OPA698ID	296-15860-5-ND	1
MSP-EXP430F5529LN	296-36506-ND	1
LW514-BULK	897-1183-ND	30

Table 13: Parts from DigiKey

Part #	Quantity
RL5-W18030: White LED	10 (0 used)
RL5-R12008: Red LED	10 (0 used)

Table 14: Parts from Super Bright LEDs

Manufacture #	Mouser #	Quantity
ZVN4206A	522-ZVN4206A	5 (1 used)

Table 15: Parts from Mouser

B: ASCII Table

!	033	0100001	.	046	0101110
"	034	0100010	/	047	0101111
#	035	0100011	0	048	0110000
$	036	0100100	1	049	0110001
%	037	0100101	2	050	0110010
&	038	0100110	3	051	0110011
'	039	0100111	4	052	0110100
(	040	0101000	5	053	0110101
)	041	0101001	6	054	0110110
*	042	0101010	7	055	0110111
+	043	0101011	8	056	0111000
,	044	0101100	9	057	0111001
-	045	0101101	:	058	0111010

;	059	0111011	T	084	1010100
<	060	0111100	U	085	1010101
=	061	0111101	V	086	1010110
>	062	0111110	W	087	1010111
?	063	0111111	X	088	1011000
@	064	1000000	Y	089	1011001
A	065	1000001	Z	090	1011010
B	066	1000010	[	091	1011011
C	067	1000011	\	092	1011100
D	068	1000100	]	093	1011101
E	069	1000101	^	094	1011110
F	070	1000110	_	095	1011111
G	071	1000111	`	096	1100000
H	072	1001000	a	097	1100001
I	073	1001001	b	098	1100010
J	074	1001010	c	099	1100011
K	075	1001011	d	100	1100100
L	076	1001100	e	101	1100101
M	077	1001101	f	102	1100110
N	078	1001110	g	103	1100111
O	079	1001111	h	104	1101000
P	080	1010000	i	105	1101001
Q	081	1010001	j	106	1101010
R	082	1010010	k	107	1101011
S	083	1010011	l	108	1101100

m	109	1101101	v	118	1110110
n	110	1101110	w	119	1110111
o	111	1101111	x	120	1111000
p	112	1110000	y	121	1111001
q	113	1110001	z	122	1111010
r	114	1110010	{	123	1111011
s	115	1110011	¦	124	1111100
t	116	1110100	}	125	1111101
u	117	1110101	~	126	1111110

C: Code

C.1 MATLAB Code

C.1.1 decoding.m

```
%Import the data collected from CCS
info = dir('receiverData.dat');
date = info.date;
begin = 0;
while begin ~= 1
    info = dir('receiverData.dat');
    date2 = info.date;
    if(strcmpi(date, date2) == 0)
        for i=1: 500000000
        end
        data = importdata('receiverData.dat', ' ', 1);
        A = data.data;
        begin = data.data(end);
    end
end

%Create the Header file to use for correlation
headString = 'This is a big test';%This is a big test';
header = text2bin(headString);

%Determine the threshold by taking the average of the received data
sum = 0;
for i=1:length(A)
    sum = sum + A(i);
end
threshold = sum/length(A);

%Convert the Voltage levels back to zeros and ones
for i=1:length(A)
    if(A(i) > threshold)
```

```
        A(i) = 1;
    else
        A(i) = 0;
    end
end

%Downsample the input data
down = 0;

%Look at the data frame by frame to see what was sent
%This is to compensate for having 6 samples instead of 7
old_start = 0;
start = 0;
old_stop = 0;
stop = 0;
count = 0;
for i = 2:length(A)-1
    if(A(i) ~= A(i-1))
        old_start = start;
        start = i;
    elseif (A(i) ~= A(i+1))
        old_stop = stop;
        stop = i;
    end

    if((old_start ~= start) && old_stop ~= stop)
        counter = length(A(start:stop));
        count = count + 1;
        if(counter >=5)
            val = A(start:stop);
            len = ceil(length(val)/6);%5.5); %/7
            bit = zeros(1,len);
            for j=1:length(bit)
                bit(j) = val(1);
            end
            down = [down bit];
        end
    end
end

down = down';

%Correlate with header 'This is a test', which has 47 ones
corr = xcorr(down,header);
maxCorr = max(corr);
ind = find(corr >= maxCorr);
startData = ind - length(down) + length(header) +1;

msg = down(startData:startData +76)';
bin2text(msg)
```

C.1.2 upSampleTest.m

```
% Create Header and Message and up sample it
% After that, set it into the random data
data = round(rand(2500,1));
headMsg = 'This is a big testThis is a big testHello world';
dataMsg = text2bin(headMsg)';
num = [6 7];
count = 0;
samp = 0;
%Upsample each bit up either 6 or 7
for i = 1: length(dataMsg)
    sel = num(1 + floor(rand()*length(num)));
    up = zeros(1, sel);
    for j = 1: length(up)
        up(j) = dataMsg(i);
    end
    samp = [samp up];
end
samp = samp';

%Remove the inital 0 from samp
upSamp = zeros(1, length(samp)-1);
for k =2:length(samp)
    upSamp(k-1) = samp(k);
end
upSamp = upSamp';

%Overwrite data with UpSamp starting at index = 100
start = 100;
for i=1:length(upSamp)
    data(start+i) = upSamp(i);
end

%Overwrite data with UpSamp starting at index = 1300
start = 1300;
for i=1:length(upSamp)
    data(start+i) = upSamp(i);
end
```

C.1.3 text2bin.m

```
% n=text2bin(textstring)
% transform text string into a vector of binary 0-1
function n=text2bin(textstring)
bintext=dec2bin(double(textstring));  % text into binary
[rp,cp]=size(bintext);
n=str2num(reshape(bintext',1,rp*cp)')';
```

C.1.4 bin2text.m

```
% ztext=bin2text(z)
% transform a vector of 7-bit binary 0-1 into a text string
function ztext=bin2text(z)
```

```
rp=floor(length(z)/7);
rez=num2str(z(1:7*rp)')';
ztext=char(bin2dec(reshape(rez,7,rp)'))';
```

C.2 C Code

C.2.1 Transmitter

```
#include "DSP28x_Project.h"     // Device Header file and Examples Include
File

#include "f2802x_common/include/clk.h"
#include "f2802x_common/include/flash.h"
#include "f2802x_common/include/gpio.h"
#include "f2802x_common/include/pie.h"
#include "f2802x_common/include/pll.h"
#include "f2802x_common/include/pwm.h"
#include "f2802x_common/include/wdog.h"
#include "text2ASCII.h"
#include "msh.h"

typedef struct
{
    PWM_Handle myPwmHandle;
    uint16_t EPwm_CMPA_Direction;
    uint16_t EPwm_CMPB_Direction;
    uint16_t EPwmTimerIntCount;
    uint16_t EPwmMaxCMPA;
    uint16_t EPwmMinCMPA;
    uint16_t EPwmMaxCMPB;
    uint16_t EPwmMinCMPB;
}EPWM_INFO;

// Prototype statements for functions found within this file.
void InitEPwm1Example(void);
void InitEPwm2Example(void);
void InitEPwm3Example(void);
interrupt void epwm1_isr(void);
interrupt void epwm2_isr(void);
interrupt void epwm3_isr(void);
void update_compare(EPWM_INFO*);
void update2(EPWM_INFO*);
void text2data(char value);

// Global variables used in this example
EPWM_INFO epwm1_info;
EPWM_INFO epwm2_info;
EPWM_INFO epwm3_info;

// Configure the period for each timer
#define EPWM1_TIMER_TBPRD   10000           // Period register
#define EPWM1_MAX_CMPA      10000           // Actual period = val/10 us
#define EPWM1_MIN_CMPA      0
#define EPWM1_MAX_CMPB      1950
#define EPWM1_MIN_CMPB      50

//Definitions are not changed, are not used
```

```
#define EPWM2_TIMER_TBPRD  2000  // Period register
#define EPWM2_MAX_CMPA     1950
#define EPWM2_MIN_CMPA       50
#define EPWM2_MAX_CMPB     1950
#define EPWM2_MIN_CMPB       50

#define EPWM3_TIMER_TBPRD  2000  // Period register
#define EPWM3_MAX_CMPA      950
#define EPWM3_MIN_CMPA       50
#define EPWM3_MAX_CMPB     1950
#define EPWM3_MIN_CMPB     1050

#define LENGTHINBUFF 7

int inbuff[LENGTHINBUFF] = {0,0,0,0,0,0,0};
int counter = 7;
int index = 0;
int ii = 0;
int headerFlag = 1;     //check to send the header file or data
int numHeader = 0;      //in case we want to extend the header length, repeat
it a number of times


// To keep track of which way the compare value is moving
#define EPWM_CMP_UP   1
#define EPWM_CMP_DOWN 0

CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm1, myPwm2, myPwm3;

void main(void)
{
    CPU_Handle myCpu;
    PLL_Handle myPll;
    WDOG_Handle myWDog;

    // Initialize all the handles needed for this application
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myPwm2 = PWM_init((void *)PWM_ePWM2_BASE_ADDR, sizeof(PWM_Obj));
    myPwm3 = PWM_init((void *)PWM_ePWM3_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));

    // Perform basic system initialization
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk);
    (*Device_cal)();
    CLK_disableAdcClock(myClk);
    //Select the internal oscillator 1 as the clock source
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Setup the PLL for x12 /1 which will yield 120Mhz = 10Mhz * 12 / 1
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_1);

    // Disable the PIE and all interrupts
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // If running from flash copy RAM only functions to RAM
#ifdef _FLASH
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
#endif

    // Initalize GPIO
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Disable);
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);

    GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_3, GPIO_PullUp_Disable);
    GPIO_setMode(myGpio, GPIO_Number_2, GPIO_2_Mode_EPWM2A);
    GPIO_setMode(myGpio, GPIO_Number_3, GPIO_3_Mode_EPWM2B);

    GPIO_setPullUp(myGpio, GPIO_Number_4, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_5, GPIO_PullUp_Disable);
    GPIO_setMode(myGpio, GPIO_Number_4, GPIO_4_Mode_EPWM3A);
    GPIO_setMode(myGpio, GPIO_Number_5, GPIO_5_Mode_EPWM3B);

    // Setup a debug vector table and enable the PIE
    PIE_setDebugIntVectorTable(myPie);
    PIE_enable(myPie);

    // Register interrupt handlers in the PIE vector table
    PIE_registerPieIntHandler(myPie, PIE_GroupNumber_3, PIE_SubGroupNumber_1,
(intVec_t)&epwm1_isr);
    PIE_registerPieIntHandler(myPie, PIE_GroupNumber_3, PIE_SubGroupNumber_2,
(intVec_t)&epwm2_isr);
    PIE_registerPieIntHandler(myPie, PIE_GroupNumber_3, PIE_SubGroupNumber_3,
(intVec_t)&epwm3_isr);

    CLK_disableTbClockSync(myClk);

    InitEPwm1Example();
    InitEPwm2Example();
    InitEPwm3Example();

    CLK_enableTbClockSync(myClk);

    // Enable CPU INT3 which is connected to EPWM1-3 INT:
    CPU_enableInt(myCpu, CPU_IntNumber_3);
    // Enable EPWM INTn in the PIE: Group 3 interrupt 1-3
    PIE_enablePwmInt(myPie, PWM_Number_1);
    PIE_enablePwmInt(myPie, PWM_Number_2);
    PIE_enablePwmInt(myPie, PWM_Number_3);

    // Enable global Interrupts and higher priority real-time debug events
    CPU_enableGlobalInts(myCpu);
    CPU_enableDebugInt(myCpu);

    for(;;) {
        asm(" NOP");
    }
}

interrupt void epwm1_isr(void)
{
    // Update the CMPA and CMPB values
    //update_compare(&epwm1_info);

    update2(&epwm1_info);

    // Clear INT flag for this timer
    PWM_clearIntFlag(myPwm1);

    // Acknowledge this interrupt to receive more interrupts from group 3
    PIE_clearInt(myPie, PIE_GroupNumber_3);

}

interrupt void epwm2_isr(void)
{

    // Update the CMPA and CMPB values
    update_compare(&epwm2_info);

    // Clear INT flag for this timer
    PWM_clearIntFlag(myPwm2);

    // Acknowledge this interrupt to receive more interrupts from group 3
    PIE_clearInt(myPie, PIE_GroupNumber_3);
}

interrupt void epwm3_isr(void)
{

    // Update the CMPA and CMPB values
    update_compare(&epwm3_info);

    // Clear INT flag for this timer
    PWM_clearIntFlag(myPwm3);

    // Acknowledge this interrupt to receive more interrupts from group 3
    PIE_clearInt(myPie, PIE_GroupNumber_3);
}

void InitEPwm1Example()
{
    CLK_enablePwmClock(myClk, PWM_Number_1);

    // Setup TBCLK
    PWM_setCounterMode(myPwm1, PWM_CounterMode_Up);          // Count up
    PWM_setPeriod(myPwm1, EPWM1_TIMER_TBPRD);                // Set timer
period
    PWM_disableCounterLoad(myPwm1);                          // Disable phase
loading
    PWM_setPhase(myPwm1, 0x0000);                            // Phase is 0
    PWM_setCount(myPwm1, 0x0000);                            // Clear counter
    PWM_setHighSpeedClkDiv(myPwm1, PWM_HspClkDiv_by_2);      // Clock ratio to
SYSCLKOUT
    PWM_setClkDiv(myPwm1, PWM_ClkDiv_by_2);

    // Setup shadow register load on ZERO
    PWM_setShadowMode_CmpA(myPwm1, PWM_ShadowMode_Shadow);
    PWM_setShadowMode_CmpB(myPwm1, PWM_ShadowMode_Shadow);
    PWM_setLoadMode_CmpA(myPwm1, PWM_LoadMode_Zero);
    PWM_setLoadMode_CmpB(myPwm1, PWM_LoadMode_Zero);

    // Set Compare values
    PWM_setCmpA(myPwm1, EPWM1_MIN_CMPA);    // Set compare A value
    PWM_setCmpB(myPwm1, EPWM1_MIN_CMPB);    // Set Compare B value

    // Set actions
    PWM_setActionQual_Zero_PwmA(myPwm1, PWM_ActionQual_Set);            //
Set PWM1A on Zero
    PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1, PWM_ActionQual_Clear);    //
Clear PWM1A on event A, up count

    PWM_setActionQual_Zero_PwmB(myPwm1, PWM_ActionQual_Set);            //
Set PWM1B on Zero
    PWM_setActionQual_CntUp_CmpB_PwmB(myPwm1, PWM_ActionQual_Clear);    //
Clear PWM1B on event B, up count

    // Interrupt where we will change the Compare Values
    PWM_setIntMode(myPwm1, PWM_IntMode_CounterEqualZero);   // Select INT on
Zero event
    PWM_enableInt(myPwm1);                                   // Enable INT
    PWM_setIntPeriod(myPwm1, PWM_IntPeriod_ThirdEvent);      // Generate INT
on 3rd event

    // Information this example uses to keep track
    // of the direction the CMPA/CMPB values are
    // moving, the min and max allowed values and
    // a pointer to the correct ePWM registers
    epwm1_info.EPwm_CMPA_Direction = EPWM_CMP_UP;   // Start by increasing
CMPA & CMPB
    epwm1_info.EPwm_CMPB_Direction = EPWM_CMP_UP;
    epwm1_info.EPwmTimerIntCount = 0;               // Zero the interrupt
counter
    epwm1_info.myPwmHandle = myPwm1;                // Set the pointer to the
ePWM module
    epwm1_info.EPwmMaxCMPA = EPWM1_MAX_CMPA;        // Setup min/max
CMPA/CMPB values
    epwm1_info.EPwmMinCMPA = EPWM1_MIN_CMPA;
    epwm1_info.EPwmMaxCMPB = EPWM1_MAX_CMPB;
    epwm1_info.EPwmMinCMPB = EPWM1_MIN_CMPB;
}

void InitEPwm2Example()
{
    CLK_enablePwmClock(myClk, PWM_Number_2);

    // Setup TBCLK
    PWM_setCounterMode(myPwm2, PWM_CounterMode_Up);     // Count up
    PWM_setPeriod(myPwm2, EPWM2_TIMER_TBPRD);           // Set timer period
    PWM_disableCounterLoad(myPwm2);                     // Disable phase
loading
    PWM_setPhase(myPwm2, 0x0000);                       // Phase is 0
    PWM_setCount(myPwm2, 0x0000);                       // Clear counter
    PWM_setHighSpeedClkDiv(myPwm2, PWM_HspClkDiv_by_2); // Clock ratio to
SYSCLKOUT
    PWM_setClkDiv(myPwm2, PWM_ClkDiv_by_2);

    // Setup shadow register load on ZERO
    PWM_setShadowMode_CmpA(myPwm2, PWM_ShadowMode_Shadow);
    PWM_setShadowMode_CmpB(myPwm2, PWM_ShadowMode_Shadow);
    PWM_setLoadMode_CmpA(myPwm2, PWM_LoadMode_Zero);
    PWM_setLoadMode_CmpB(myPwm2, PWM_LoadMode_Zero);

    // Set Compare values
    PWM_setCmpA(myPwm2, EPWM2_MIN_CMPA);    // Set compare A value
    PWM_setCmpB(myPwm2, EPWM2_MIN_CMPB);    // Set Compare B value

    // Set actions
    PWM_setActionQual_Period_PwmA(myPwm2, PWM_ActionQual_Clear);    // Clear
PWM2A on Period
    PWM_setActionQual_CntUp_CmpA_PwmA(myPwm2, PWM_ActionQual_Set);  // Set
PWM2A on event A, up count

    PWM_setActionQual_Period_PwmB(myPwm2, PWM_ActionQual_Clear);    // Clear
PWM2B on Period
    PWM_setActionQual_CntUp_CmpB_PwmB(myPwm2, PWM_ActionQual_Set);  // Set
PWM2B on event B, up count

    // Interrupt where we will change the Compare Values
    PWM_setIntMode(myPwm2, PWM_IntMode_CounterEqualZero);   // Select INT on
Zero event
    PWM_enableInt(myPwm2);                                  // Enable INT
    PWM_setIntPeriod(myPwm2, PWM_IntPeriod_ThirdEvent);     // Generate INT
on 3rd event

    // Information this example uses to keep track
    // of the direction the CMPA/CMPB values are
    // moving, the min and max allowed values and
    // a pointer to the correct ePWM registers
    epwm2_info.EPwm_CMPA_Direction = EPWM_CMP_UP;   // Start by increasing
CMPA
    epwm2_info.EPwm_CMPB_Direction = EPWM_CMP_DOWN; // and decreasing CMPB
    epwm2_info.EPwmTimerIntCount = 0;               // Zero the interrupt
counter
    epwm2_info.myPwmHandle = myPwm2;                // Set the pointer to the
ePWM module
```

```
    epwm2_info.EPwmMaxCMPA = EPWM2_MAX_CMPA;        // Setup min/max
CMPA/CMPB values
    epwm2_info.EPwmMinCMPA = EPWM2_MIN_CMPA;
    epwm2_info.EPwmMaxCMPB = EPWM2_MAX_CMPB;
    epwm2_info.EPwmMinCMPB = EPWM2_MIN_CMPB;
}

void InitEPwm3Example(void)
{
    CLK_enablePwmClock(myClk, PWM_Number_3);

    // Setup TBCLK
    PWM_setCounterMode(myPwm3, PWM_CounterMode_Up);     // Count up
    PWM_setPeriod(myPwm3, EPWM3_TIMER_TBPRD);           // Set timer period
    PWM_disableCounterLoad(myPwm3);                     // Disable phase
loading
    PWM_setPhase(myPwm3, 0x0000);                       // Phase is 0
    PWM_setCount(myPwm3, 0x0000);                       // Clear counter
    PWM_setHighSpeedClkDiv(myPwm3, PWM_HspClkDiv_by_1); // Clock ratio to
SYSCLKOUT
    PWM_setClkDiv(myPwm3, PWM_ClkDiv_by_1);

    // Setup shadow register load on ZERO
    PWM_setShadowMode_CmpA(myPwm3, PWM_ShadowMode_Shadow);
    PWM_setShadowMode_CmpB(myPwm3, PWM_ShadowMode_Shadow);
    PWM_setLoadMode_CmpA(myPwm3, PWM_LoadMode_Zero);
    PWM_setLoadMode_CmpB(myPwm3, PWM_LoadMode_Zero);

    // Set Compare values
    PWM_setCmpA(myPwm3, EPWM3_MIN_CMPA);    // Set compare A value
    PWM_setCmpB(myPwm3, EPWM3_MIN_CMPB);    // Set Compare B value

    // Set Actions
    PWM_setActionQual_CntUp_CmpA_PwmA(myPwm3, PWM_ActionQual_Set);      //
Set PWM3A on event B, up count
    PWM_setActionQual_CntUp_CmpB_PwmA(myPwm3, PWM_ActionQual_Clear);    //
Clear PWM3A on event B, up count

    PWM_setActionQual_Zero_PwmB(myPwm3, PWM_ActionQual_Toggle);         //
Toggle EPWM3B on Zero

    // Interrupt where we will change the Compare Values
    PWM_setIntMode(myPwm3, PWM_IntMode_CounterEqualZero);   // Select INT on
Zero event
    PWM_enableInt(myPwm3);                                  // Enable INT
    PWM_setIntPeriod(myPwm3, PWM_IntPeriod_ThirdEvent);     // Generate INT
on 3rd event

    // Information this example uses to keep track
    // of the direction the CMPA/CMPB values are
    // moving, the min and max allowed values and
    // a pointer to the correct ePWM registers
    epwm3_info.EPwm_CMPA_Direction = EPWM_CMP_UP;   // Start by increasing
CMPA
    epwm3_info.EPwm_CMPB_Direction = EPWM_CMP_DOWN; // and decreasing CMPB
    epwm3_info.EPwmTimerIntCount = 0;               // Zero the interrupt
counter
```

```
    epwm3_info.myPwmHandle = myPwm3;                // Set the pointer to the 
ePWM module
    epwm3_info.EPwmMaxCMPA = EPWM3_MAX_CMPA;        // Setup min/max 
CMPA/CMPB values
    epwm3_info.EPwmMinCMPA = EPWM3_MIN_CMPA;
    epwm3_info.EPwmMaxCMPB = EPWM3_MAX_CMPB;
    epwm3_info.EPwmMinCMPB = EPWM3_MIN_CMPB;
}

void update2(EPWM_INFO *epwm_info)
{

    //send header first
    if (headerFlag == 1){
        if (counter == 7){
            counter = 0;
            text2data(header[index]);
            index++;
            if (index == HEADSIZE){         //end of header
                index = 0;
                headerFlag = 0;         //set headerFlag to 0, meaning start 
transmitting msg data
            }
        }
    }
    else //if (headerFlag == 0)
    {
        //Check to see if the message has been sent
        //Output all zeros to turn off the LEDs
        if (counter == 7){
            counter = 0;
            text2data(msg[index]);
            index++;
            if (index == MSGSIZE){
                headerFlag = 1;         //set headerFlag to 1, meaning start 
transmitting header data
            }
        }
    }

    //Output a High pulse if the bit is a 1
    if(inbuff[counter] == 1)
    {
        epwm1_info.EPwmMinCMPA = EPWM1_MAX_CMPA + 1;
        epwm1_info.EPwm_CMPA_Direction = EPWM_CMP_UP;
        PWM_setCmpA(myPwm1, EPWM1_MAX_CMPA + 1);
    }
    else   //Output a Low pulse if the bit is a 0
    {
        epwm1_info.EPwmMinCMPA = EPWM1_MIN_CMPA;
        epwm1_info.EPwm_CMPA_Direction = EPWM_CMP_DOWN;
        PWM_setCmpA(myPwm1, EPWM1_MIN_CMPA);
    }
    counter++;

    //if(counter == 8)
    //  counter = 0;
```

```
}

void update_compare(EPWM_INFO *epwm_info)
{

    // Every 10'th interrupt, change the CMPA/CMPB values
    if(epwm_info->EPwmTimerIntCount == 10)
    {
        epwm_info->EPwmTimerIntCount = 0;


        // If we were increasing CMPA, check to see if
        // we reached the max value.  If not, increase CMPA
        // else, change directions and decrease CMPA
        if(epwm_info->EPwm_CMPA_Direction == EPWM_CMP_UP) {
            if(PWM_getCmpA(epwm_info->myPwmHandle) < epwm_info->EPwmMaxCMPA)
{
              PWM_setCmpA(epwm_info->myPwmHandle, PWM_getCmpA(epwm_info-
>myPwmHandle) + 1);
            }
            else {
              epwm_info->EPwm_CMPA_Direction = EPWM_CMP_DOWN;
              PWM_setCmpA(epwm_info->myPwmHandle, PWM_getCmpA(epwm_info-
>myPwmHandle) - 1);
            }
        }

        // If we were decreasing CMPA, check to see if
        // we reached the min value.  If not, decrease CMPA
        // else, change directions and increase CMPA
        else {
            if(PWM_getCmpA(epwm_info->myPwmHandle) == epwm_info->EPwmMinCMPA)
{
              epwm_info->EPwm_CMPA_Direction = EPWM_CMP_UP;
              PWM_setCmpA(epwm_info->myPwmHandle, PWM_getCmpA(epwm_info-
>myPwmHandle) + 1);
            }
            else {
              PWM_setCmpA(epwm_info->myPwmHandle, PWM_getCmpA(epwm_info-
>myPwmHandle) - 1);
            }
        }

        // If we were increasing CMPB, check to see if
        // we reached the max value.  If not, increase CMPB
        // else, change directions and decrease CMPB
        if(epwm_info->EPwm_CMPB_Direction == EPWM_CMP_UP) {
            if(PWM_getCmpB(epwm_info->myPwmHandle) < epwm_info->EPwmMaxCMPB)
{
              PWM_setCmpB(epwm_info->myPwmHandle, PWM_getCmpB(epwm_info-
>myPwmHandle) + 1);
            }
            else {
              epwm_info->EPwm_CMPB_Direction = EPWM_CMP_DOWN;
              PWM_setCmpB(epwm_info->myPwmHandle, PWM_getCmpB(epwm_info-
>myPwmHandle) - 1);
            }
        }

        // If we were decreasing CMPB, check to see if
        // we reached the min value.  If not, decrease CMPB
        // else, change directions and increase CMPB

        else {
            if(PWM_getCmpB(epwm_info->myPwmHandle) == epwm_info->EPwmMinCMPB)
{
              epwm_info->EPwm_CMPB_Direction = EPWM_CMP_UP;
              PWM_setCmpB(epwm_info->myPwmHandle, PWM_getCmpB(epwm_info-
>myPwmHandle) + 1);
            }
            else {
              PWM_setCmpB(epwm_info->myPwmHandle, PWM_getCmpB(epwm_info-
>myPwmHandle) - 1);
            }
        }
    }
    else {
      epwm_info->EPwmTimerIntCount++;
    }

    return;
}

void text2data(char value){
    switch (value){
    case 'a':
        for(ii=0; ii<7; ii++)
            inbuff[ii] = a[ii];
        break;
    case 'b':
        for(ii=0; ii<7; ii++)
            inbuff[ii] = b[ii];
        break;
    case 'c':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = c[ii];
        break;
    case 'd':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = d[ii];
        break;
    case 'e':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = e[ii];
        break;
    case 'f':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = f[ii];
        break;
    case 'g':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = g[ii];
        break;
    case 'h':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = h[ii];
        break;
    case 'i':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = i[ii];
        break;
    case 'j':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = j[ii];
        break;
    case 'k':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = k[ii];
        break;
    case 'l':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = l[ii];
        break;
    case 'm':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = m[ii];
        break;
    case 'n':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = n[ii];
        break;
    case 'o':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = o[ii];
        break;
    case 'p':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = p[ii];
        break;
    case 'q':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = q[ii];
        break;
    case 'r':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = r[ii];
        break;
    case 's':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = s[ii];
        break;
    case 't':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = t[ii];
        break;
    case 'u':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = u[ii];
        break;
    case 'v':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = v[ii];
        break;
    case 'w':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = w[ii];
        break;
    case 'x':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = x[ii];
        break;
    case 'y':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = y[ii];
        break;
    case 'z':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = z[ii];
        break;
    case 'A':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = A[ii];
        break;
    case 'B':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = B[ii];
        break;
    case 'C':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = C[ii];
        break;
    case 'D':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = D[ii];
        break;
    case 'E':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = E[ii];
        break;
    case 'F':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = F[ii];
        break;
    case 'G':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = G[ii];
        break;
    case 'H':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = H[ii];
        break;
    case 'I':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = I[ii];
        break;
    case 'J':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = J[ii];
```

```
        break;
    case 'K':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = K[ii];
        break;
    case 'L':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = L[ii];
        break;
    case 'M':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = M[ii];
        break;
    case 'N':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = N[ii];
        break;
    case 'O':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = O[ii];
        break;
    case 'P':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = P[ii];
        break;
    case 'Q':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = Q[ii];
        break;
    case 'R':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = R[ii];
        break;
    case 'S':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = S[ii];
        break;
    case 'T':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = T[ii];
        break;
    case 'U':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = U[ii];
        break;
    case 'V':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = V[ii];
        break;
    case 'W':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = W[ii];
        break;
    case 'X':
        for(ii=0; ii<7; ii++)
        inbuff[ii] = X[ii];
        break;
```

```
case 'Y':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = Y[ii];
    break;
case 'Z':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = Z[ii];
    break;
case '0':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = zero[ii];
    break;
case '1':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = one[ii];
    break;
case '2':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = two[ii];
    break;
case '3':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = three[ii];
    break;
case '4':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = four[ii];
    break;
case '5':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = five[ii];
    break;
case '6':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = six[ii];
    break;
case '7':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = seven[ii];
    break;
case '8':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = eight[ii];
    break;
case '9':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = nine[ii];
    break;
case '!':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = excl[ii];
    break;
case '"':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = quot[ii];
    break;
case '#':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = hash[ii];
    break;
case '$':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = dola[ii];
    break;
case '%':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = perc[ii];
    break;
case '&':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = amp[ii];
    break;
case '(':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = lpar[ii];
    break;
case ')':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = rpar[ii];
    break;
case '*':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = time[ii];
    break;
case '+':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = add[ii];
    break;
case ',':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = comm[ii];
    break;
case '-':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = dash[ii];
    break;
case '.':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = peri[ii];
    break;
case '/':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = slsh[ii];
    break;
case ':':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = col[ii];
    break;
case ';':
    for(ii=0; ii<7; ii++)
    inbuff[ii] = scol[ii];
    break;
case '<':
    for(ii=0; ii<7; ii++)
```

```
            inbuff[ii] = lthn[ii];
            break;
        case '=':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = equl[ii];
            break;
        case '>':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = gthn[ii];
            break;
        case '?':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = ques[ii];
            break;
        case '@':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = at[ii];
            break;
        case '{':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = lcbk[ii];
            break;
        case '}':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = rcbk[ii];
            break;
        case '~':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = til[ii];
            break;
        case ' ':
            for(ii=0; ii<7; ii++)
            inbuff[ii] = spac[ii];
            break;
        default:
            for(ii=0; ii<7; ii++)
            inbuff[ii] = 0;
            break;
        }
}

/*
 * This Header File contains the message that will be sent to the receiver
 * Also the header sequence is also defined in this file
 */


const char msg[11] = {'M', 'y', ' ', 'N', 'a', 'm', 'e', ':', 'B', 'o', 'b'};
//const char msg[11] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l',
'd'};
//const char msg[11] = {'G', 'o', 'o', 'd', 'b', 'y', 'e', ' ', ' ', ' ', '
'};
//const char msg[16] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l',
'd', ' ', ' ', ' ',' ',' '};
//const char msg[16] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l',
'd',' ','P','i','k','a'};
```

```
const char header[40] = {'T', 'h', 'i', 's', 'T', 'h', 'i', 's', ' ', 'i',
's', ' ', 'a', ' ', 'b', 'i', 'g', ' ', 't', 'e', 's', 't','T', 'h', 'i',
's', ' ', 'i', 's', ' ', 'a', ' ', 'b', 'i', 'g', ' ', 't', 'e', 's', 't'};
const int MSGSIZE = 11;
//const int MSGSIZE = 16;
const int HEADSIZE = 40;


/*
 * This Header file defines the ASCII values for all characters
 * The binary values are defined in a 8 bit array
 */

//Numbers
const int zero[8] = {0,1,1,0,0,0,0};
const int one[8]  = {0,1,1,0,0,0,1};
const int two[8]  = {0,1,1,0,0,1,0};
const int three[8]= {0,1,1,0,0,1,1};
const int four[8] = {0,1,1,0,1,0,0};
const int five[8] = {0,1,1,0,1,0,1};
const int six[8]  = {0,1,1,0,1,1,0};
const int seven[8]= {0,1,1,0,1,1,1};
const int eight[8]= {0,1,1,1,0,0,0};
const int nine[8] = {0,1,1,1,0,0,1};

//Lower Case Letters
const int a[8] = {1,1,0,0,0,0,1};
const int b[8] = {1,1,0,0,0,1,0};
const int c[8] = {1,1,0,0,0,1,1};
const int d[8] = {1,1,0,0,1,0,0};
const int e[8] = {1,1,0,0,1,0,1};
const int f[8] = {1,1,0,0,1,1,0};
const int g[8] = {1,1,0,0,1,1,1};
const int h[8] = {1,1,0,1,0,0,0};
const int i[8] = {1,1,0,1,0,0,1};
const int j[8] = {1,1,0,1,0,1,0};
const int k[8] = {1,1,0,1,0,1,1};
const int l[8] = {1,1,0,1,1,0,0};
const int m[8] = {1,1,0,1,1,0,1};
const int n[8] = {1,1,0,1,1,1,0};
const int o[8] = {1,1,0,1,1,1,1};
const int p[8] = {1,1,1,0,0,0,0};
const int q[8] = {1,1,1,0,0,0,1};
const int r[8] = {1,1,1,0,0,1,0};
const int s[8] = {1,1,1,0,0,1,1};
const int t[8] = {1,1,1,0,1,0,0};
const int u[8] = {1,1,1,0,1,0,1};
const int v[8] = {1,1,1,0,1,1,0};
const int w[8] = {1,1,1,0,1,1,1};
const int x[8] = {1,1,1,1,0,0,0};
const int y[8] = {1,1,1,1,0,0,1};
const int z[8] = {1,1,1,1,0,1,0};

//Upper Case Letters
const int A[8] = {1,0,0,0,0,0,1};
const int B[8] = {1,0,0,0,0,1,0};
const int C[8] = {1,0,0,0,0,1,1};
const int D[8] = {1,0,0,0,1,0,0};
const int E[8] = {1,0,0,0,1,0,1};
const int F[8] = {1,0,0,0,1,1,0};
const int G[8] = {1,0,0,0,1,1,1};
const int H[8] = {1,0,0,1,0,0,0};
const int I[8] = {1,0,0,1,0,0,1};
const int J[8] = {1,0,0,1,0,1,0};
const int K[8] = {1,0,0,1,0,1,1};
const int L[8] = {1,0,0,1,1,0,0};
const int M[8] = {1,0,0,1,1,0,1};
const int N[8] = {1,0,0,1,1,1,0};
const int O[8] = {1,0,0,1,1,1,1};
const int P[8] = {1,0,1,0,0,0,0};
const int Q[8] = {1,0,1,0,0,0,1};
const int R[8] = {1,0,1,0,0,1,0};
const int S[8] = {1,0,1,0,0,1,1};
const int T[8] = {1,0,1,0,1,0,0};
const int U[8] = {1,0,1,0,1,0,1};
const int V[8] = {1,0,1,0,1,1,0};
const int W[8] = {1,0,1,0,1,1,1};
const int X[8] = {1,0,1,1,0,0,0};
const int Y[8] = {1,0,1,1,0,0,1};
const int Z[8] = {1,0,1,1,0,1,0};

//Symbols
const int excl[8]  = {0,1,0,0,0,0,1};
const int quot[8]  = {0,1,0,0,0,1,0};
const int hash[8]  = {0,1,0,0,0,1,1};
const int dola[8]  = {0,1,0,0,1,0,0};
const int perc[8]  = {0,1,0,0,1,0,1};
const int amp[8]   = {0,1,0,0,1,1,0};
const int apst[8]  = {0,1,0,0,1,1,1};
const int lpar[8]  = {0,1,0,1,0,0,0};
const int rpar[8]  = {0,1,0,1,0,0,1};
const int time[8]  = {0,1,0,1,0,1,0};
const int add[8]   = {0,1,0,1,0,1,1};
const int comm[8]  = {0,1,0,1,1,0,0};
const int dash[8]  = {0,1,0,1,1,0,1};
const int peri[8]  = {0,1,0,1,1,1,0};
const int slsh[8]  = {0,1,0,1,1,1,1};
const int col[8]   = {0,1,1,1,0,1,0};
const int scol[8]  = {0,1,1,1,0,1,1};
const int lthn[8]  = {0,1,1,1,1,0,0};
const int equl[8]  = {0,1,1,1,1,0,1};
const int gthn[8]  = {0,1,1,1,1,1,0};
const int ques[8]  = {0,1,1,1,1,1,1};
const int at[8]    = {1,0,0,0,0,0,0};
const int lcbk[8]  = {1,1,1,1,0,1,1};
const int rcbk[8]  = {1,1,1,1,1,0,1};
const int til[8]   = {1,1,1,1,1,1,0};
const int spac[8]  = {0,1,0,0,0,0,0};
```

C.2.2 Receiver

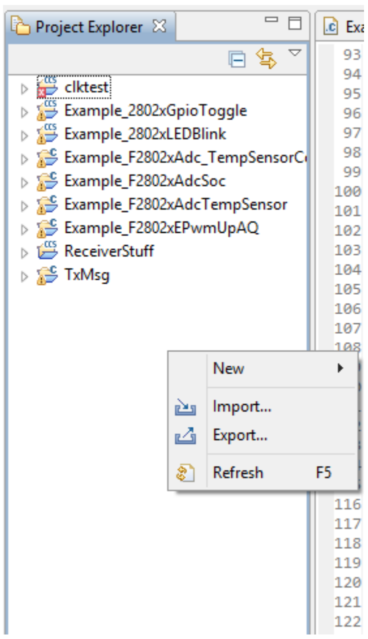
```
#include <msp430.h>
#include <stdio.h>
#include <string.h>

#define LEN 49

int count=0;
int i;
int index = LEN;
float inBuff[LEN];
int dispFlag = 0;
int *Flash_ptr;
float sum=0;

//Function Declarations
void init_tmrb();

int main(void)
{
  init_tmrb();
  WDTCTL = WDTPW + WDTHOLD;                  // Stop WDT
  ADC12CTL0 = ADC12SHT02 + ADC12ON + ADC12IE; // ADC12ON, interrupt enabled
                                              // Sampling time, ADC12 on
  ADC12CTL1 = ADC12SHP;                      // Use sampling timer
  ADC12IE = 0x01;                            // Enable interrupt
  ADC12CTL0 |= ADC12ENC;
  P6SEL |= 0x01;                             // P6.0 ADC option select
  P1DIR |= 0x01;                             // P1.0 output
  Flash_ptr = (int*)0x10000;                 //Initialize pointer at start of
Flash address

  while (1)
  {
    __bis_SR_register(LPM0_bits + GIE);      // LPM0, ADC12_ISR will force
exit
    __no_operation();                        // For debugger
  }
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
//  switch(__even_in_range(ADC12IV,34))
//  {
//
//  case  0: break;                            // Vector  0:  No interrupt
//  case  2: break;                            // Vector  2:  ADC overflow
//  case  4: break;                            // Vector  4:  ADC timing
overflow
//  case  6:                                   // Vector  6:  ADC12IFG0
    __bic_SR_register_on_exit(CPUOFF);      // Clear CPUOFF bit from 0(SR)
//
//  case  8: break;                            // Vector  8:  ADC12IFG1
//  case 10: break;                            // Vector 10:  ADC12IFG2
//  case 12: break;                           // Vector 12:  ADC12IFG3
//  case 14: break;                           // Vector 14:  ADC12IFG4
//  case 16: break;                           // Vector 16:  ADC12IFG5
//  case 18: break;                           // Vector 18:  ADC12IFG6
//  case 20: break;                           // Vector 20:  ADC12IFG7
//  case 22: break;                           // Vector 22:  ADC12IFG8
//  case 24: break;                           // Vector 24:  ADC12IFG9
//  case 26: break;                           // Vector 26:  ADC12IFG10
//  case 28: break;                           // Vector 28:  ADC12IFG11
//  case 30: break;                           // Vector 30:  ADC12IFG12
//  case 32: break;                           // Vector 32:  ADC12IFG13
//  case 34: break;                           // Vector 34:  ADC12IFG14
//  default: break;
//  }
}

void init_tmrb()
{
  // Set up to use ACLK as clock source, no divider, 16 bit, up mode
  // MAX_COUNT is 32768 for a interrupt period of 1 sec
  TBCTL = TBSSEL_2 + CNTL_0 + ID_0 + MC_1;
  TBCCR0 = 52;//32767;
  TBCCTL0 = CCIE;
}

/************* TimerB compare interrupt *********/
#pragma vector=TIMERB0_VECTOR
__interrupt void Timer_B0(void)
{
    //Check Threshold, if greater, store samples into Flash
    if (sum > 140000){//170000){
        if (count != 4600)
        {
            ADC12CTL0 |= ADC12SC;// + ENC; // Start a conversion
            FCTL3 = FWKEY;
            FCTL1 = FWKEY+WRT;
            *Flash_ptr++ = ADC12MEM0;
            FCTL1 = FWKEY;                        // Clear WRT bit
            FCTL3 = FWKEY+LOCK;                   // Set LOCK bit
            count++;
        }
        else
        {
            FCTL3 = FWKEY;
            FCTL1 = FWKEY+WRT;
            *Flash_ptr++ = 1;
            FCTL1 = FWKEY;                        // Clear WRT bit
            FCTL3 = FWKEY+LOCK;                   // Set LOCK bit
        }
    }
    else
    {
        ADC12CTL0 |= ADC12SC;
        index--;                //decrement index
        if(index < 0)            //check if index is negative
            index = LEN-1;      //if so, wrap around
        //store most recent sample at index location
        inBuff[index] = ADC12MEM0;

        sum = 0;
        //Sum all collected samples
        for(i=0; i<LEN; i++)
        {
            sum +=inBuff[i];
        }
    }
}
```

D: User Guide

Code Composer Studio must be installed on both the transmitting computer and the receiving computer in order to code and debug the various programs needed to operate the system. CCS can be downloaded from the TI Wiki page: http://processors.wiki.ti.com/index.php/Download_CCS . A license is also needed for the particular device being used. This information is also included on this webpage.

It is recommended that example code be downloaded prior to creating any projects, as projects created on CCS require several other files in order to function correctly. The best way to obtain this example code is through the TI tool, controlSUITE: http://www.ti.com/tool/controlsuite. Occasionally, the code provided on the board specific TI webpage contains out of date code, where certain files that are necessary for operation of the example code does not exist, which would cause non-descriptive errors to occur.

To import a project, right-click on an empty space in the “Project Explorer” pane and select import. Next select “Existing CCS Eclipse Projects”, make sure the “Select search-directory” option is selected, and browse for the project to be imported.



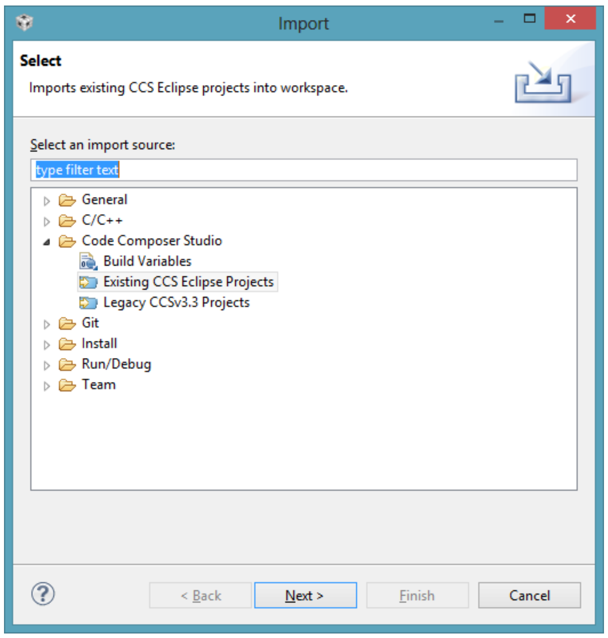


Figure 28: Project Explorer Pane and Import File Selection Window

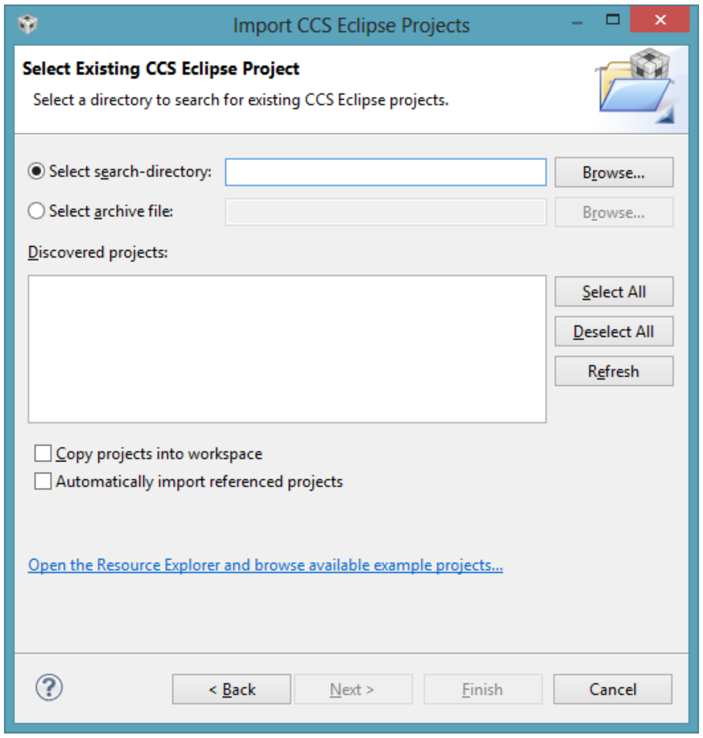
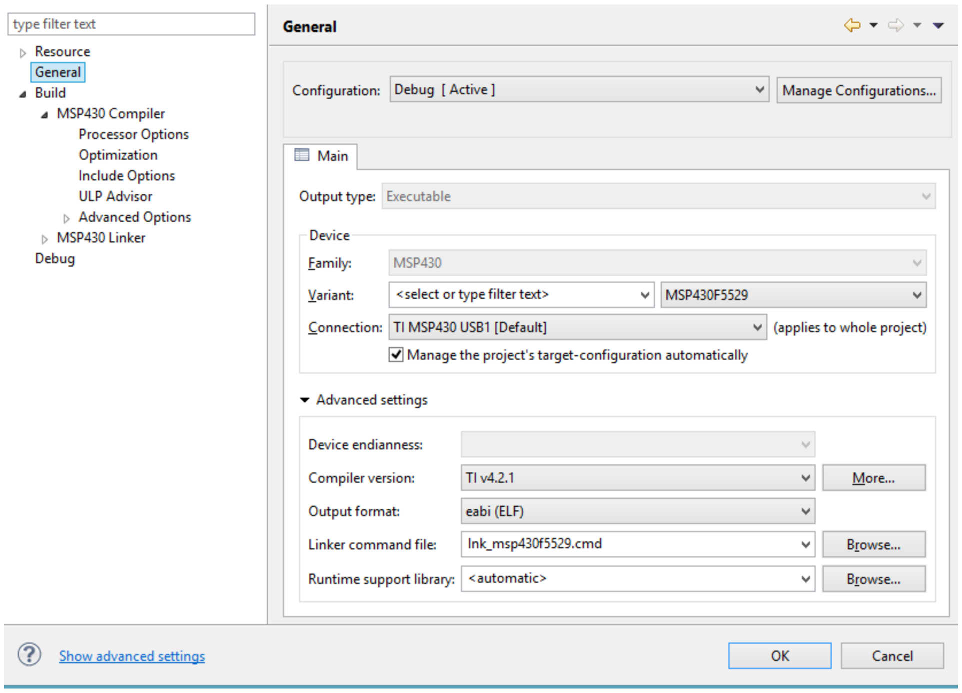

Import CCS Eclipse Projects
Select Existing CCS Eclipse Project
Select a directory to search for existing CCS Eclipse projects.
Select search-directory:
Browse...
Select archive file:
Browse...
Discovered projects:
Select All
Deselect All
Refresh
Copy projects into workspace
Automatically import referenced projects
Open the Resource Explorer and browse available example projects...
< Back
Next >
Finish
Cancel

Figure 29: Selecting Directory of Project

Alternatively, a new project can be created by selecting File > New > CCS Project. A project name will need to be specified, and the parameters under device will also need to be specified. For this particular system, the transmitter LAUNCHXL-F28027 is of the C2000 family, Experimenter's Kit – Piccolo

F28027 variant, and it uses the Texas Instruments XDS100v1 USB Emulator connection. The receiver MSP430F5529 is of the MSP430 Family, the MSP430F5529 variant, and uses the TI MSP430 USB1 connection.



Configuration: Debug [Active]

Output type: Executable

Family: MSP430

Variant: <select or type filter text> MSP430F5529

Connection: TI MSP430 USB1 [Default] (applies to whole project)

☑ Manage the project's target-configuration automatically

Device endianness:

Compiler version: TI v4.2.1

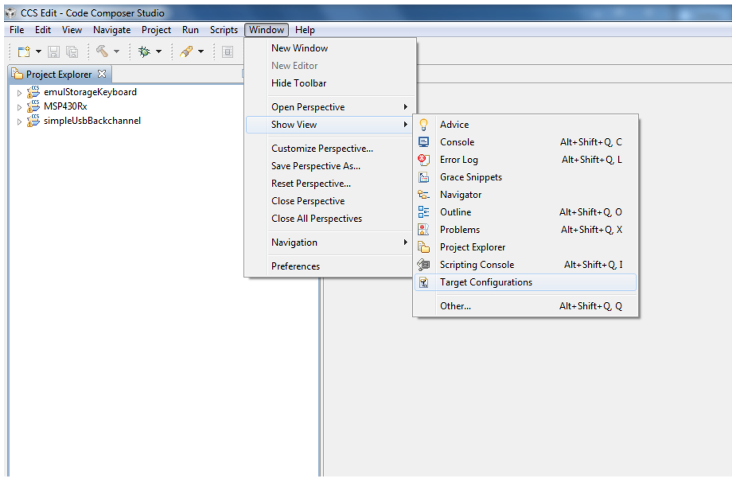
Output format: eabi (ELF)

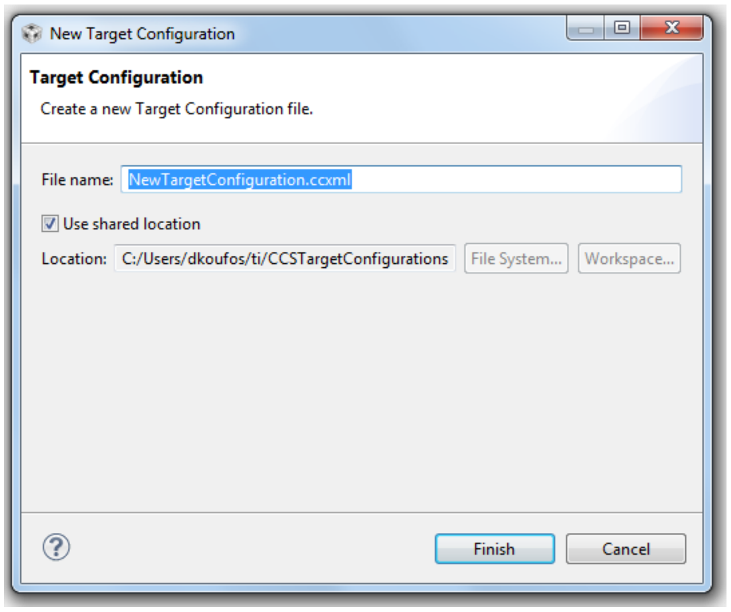
Linker command file: lnk_msp430f5529.cmd

Runtime support library: <automatic>

Figure 30: Configuring Device

Next, open the Target Configurations pane by selecting Window > Show View > Target Configurations. Right-click "User-Defined" and select "New Target Configuration". Specify each field as desired and select "Finish", which will open a general setup window to specify the target devices.




New Target Configuration

Target Configuration

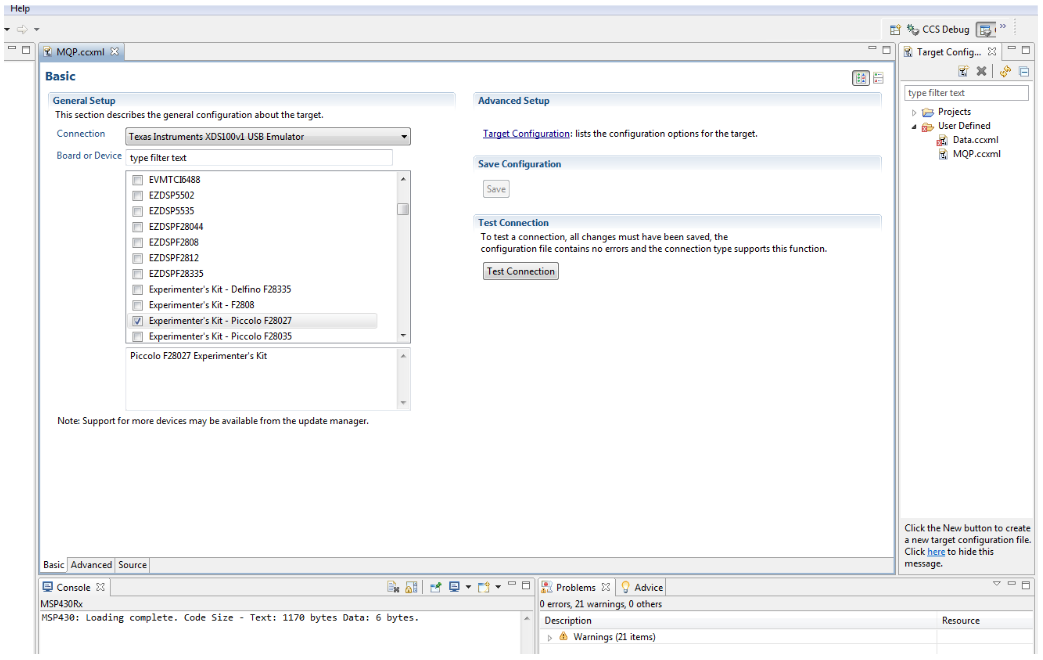
Create a new Target Configuration file.

File name: NewTargetConfiguration.ccxml

[✓] Use shared location

Location: C:/Users/dkoufos/ti/CCSTargetConfigurations

Figure 31: Setting Target Configuration



Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v1 USB Emulator

Board or Device: type filter text

- [ ] EVMTCI6488
- [ ] EZDSP5502
- [ ] EZDSP5535
- [ ] EZDSPF28044
- [ ] EZDSPF2808
- [ ] EZDSPF2812
- [ ] EZDSPF28335
- [ ] Experimenter's Kit - Delfino F28335
- [ ] Experimenter's Kit - F2808
- [✓] Experimenter's Kit - Piccolo F28027
- [ ] Experimenter's Kit - Piccolo F28035

Piccolo F28027 Experimenter's Kit

Note: Support for more devices may be available from the update manager.

Advanced Setup

Target Configuration: lists the configuration options for the target.

Save Configuration

Test Connection

To test a connection, all changes must have been saved, the configuration file contains no errors and the connection type supports this function.

type filter text

- Projects
- User Defined
  - Data.ccxml
  - MQP.ccxml

Click the New button to create a new target configuration file. Click here to hide this message.

```
MSP430Rx
MSP430: Loading complete. Code Size - Text: 1170 bytes Data: 6 bytes.
```

0 errors, 21 warnings, 0 others

Description	Resource
Warnings (21 items)	

Figure 32: General Setup of Target Configuration

The connection and board or device will be the same as those selected when creating a new project. When finished, hit "save".

Right-click the newly created target configuration and select "Launch Target Configuration". Once this process is complete, select "Run" on the taskbar and then "Connect Target". Once these steps are complete, code can be written and executed on the devices.

For the LAUNCHXL-F28027, data can be stored on the RAM or the flash memory by selecting the drop-down menu under the “Build” icon (a hammer icon) in the taskbar. For the MSP430-F5529, writing to flash must be implemented with code. Code can then be debugged by selecting the “Debug” icon (a green bug icon) in the taskbar. Once in the debugging view with executable code that contains no errors, the “Debug” pane will have options to execute, pause, or stop the code, as well as options to step through code and restart the code.

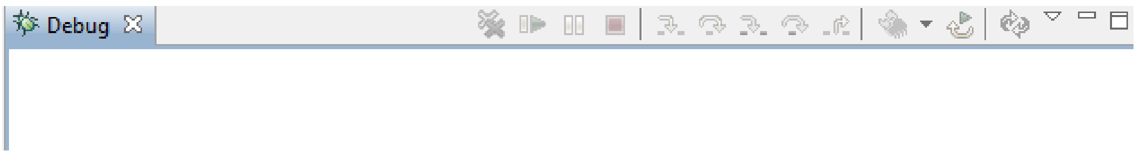
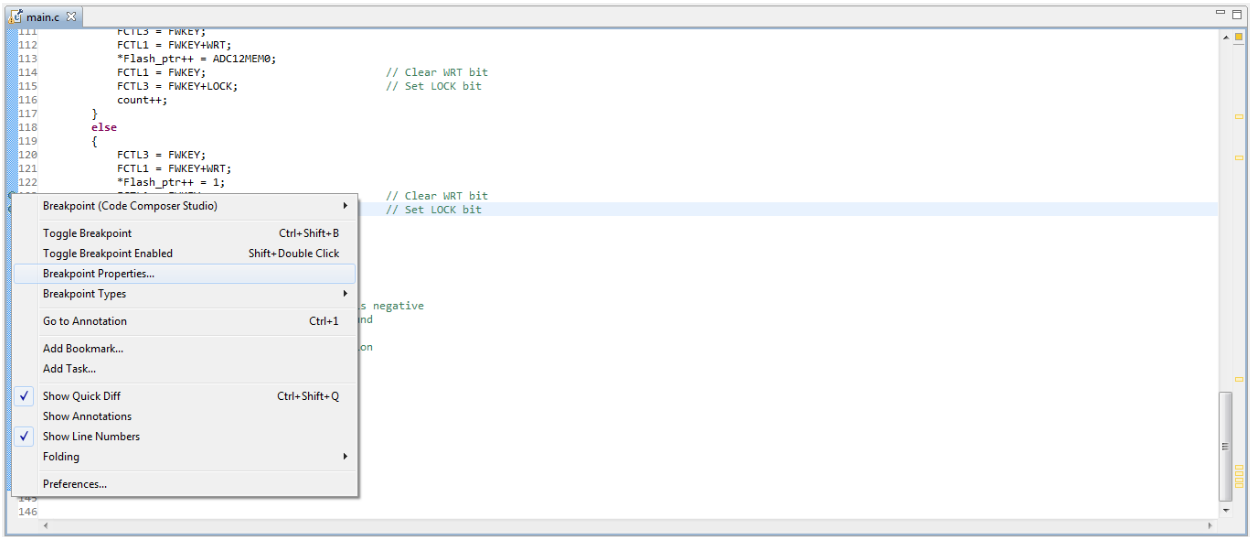


Figure 33: Debug Pane

The “Expressions” window and “Memory Browser” are good tools to determine if various sections of code are functioning as expected. In particular, these tools allow quick determination of whether variables have the correct values when breakpoints are included in the code. These tools can be found by selecting “View” in the taskbar.

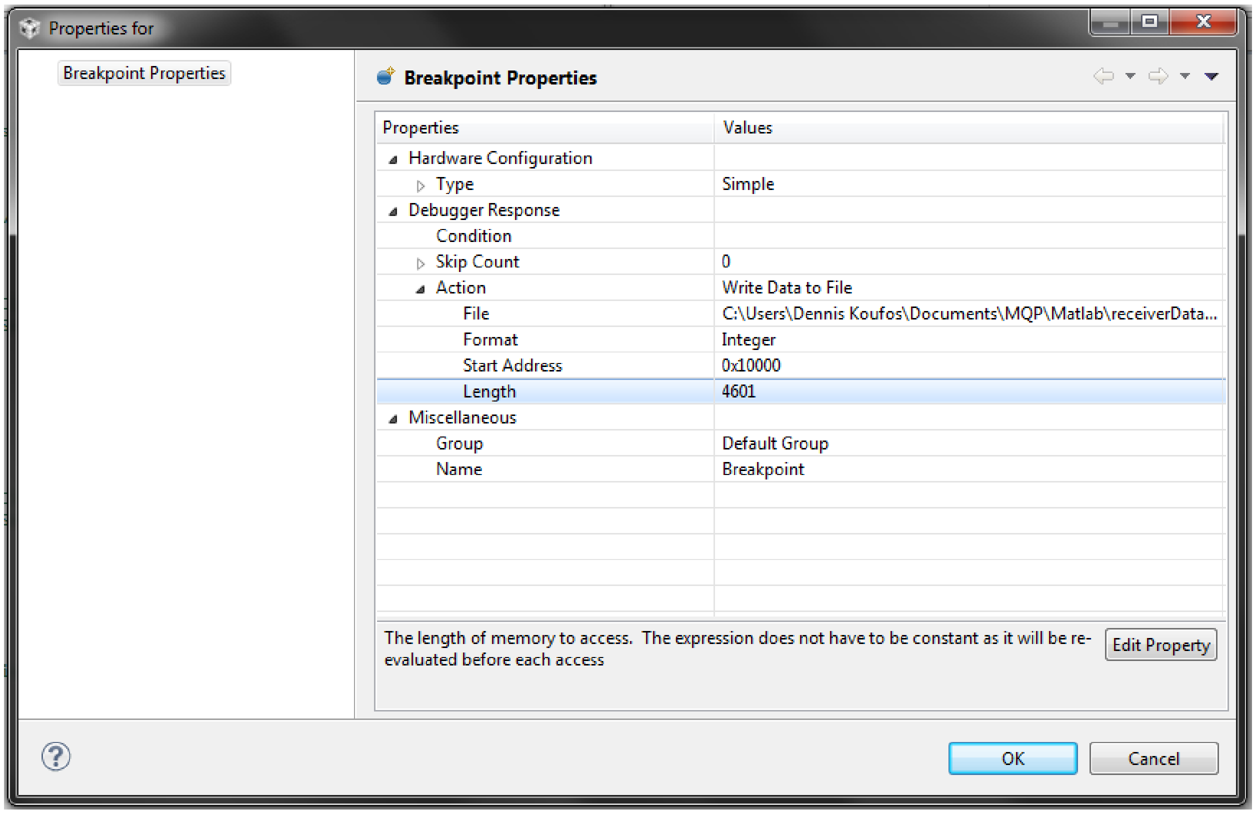
While in the debug mode, breakpoints are useful tools to use to understand how the code is running. A breakpoint can be set by right clicking the grey bar next to the line of code where the user wants to stop the program. Breakpoints can be used for more than just halting the program. Once a breakpoint is set, it is possible to left click on it to bring up a breakpoint menu. From there, it is possible to access the break points properties.



```
            FCTL3 = FWKEY;
            FCTL1 = FWKEY+WRT;
            *Flash_ptr++ = ADC12MEM0;
            FCTL1 = FWKEY;                          // Clear WRT bit
            FCTL3 = FWKEY+LOCK;                     // Set LOCK bit
            count++;
        }
        else
        {
            FCTL3 = FWKEY;
            FCTL1 = FWKEY+WRT;
            *Flash_ptr++ = 1;
                                                    // Clear WRT bit
                                                    // Set LOCK bit
```

Figure 34: Accessing Breakpoint Properties

Once the breakpoint 'properties' option is selected, a pop up menu will open that will allow the user to make changes to the breakpoint. There is an action property that determines how the breakpoint functions. By right clicking on the value section for the action property, it is possible to change the action to something more desirable. By selecting the option to 'write data to a file', the parameters are file, format, start address, and length will appear. The file option lets the user choose the file to which the data can be exported. The format option lets the user determine how the data will be exported. For this project, the data was exported as Integers, but it could have been saved as hexadecimal values or some other unit. The start address determines where in memory the data to be saved initially starts. For this project, the data was stored in flash, which starts at the address 0x10000. The length option determines the number of data points to be saved. Depending on how long of a message is sent, this value can be changed by the user.



Properties	Values
Hardware Configuration	
Type	Simple
Debugger Response	
Condition	
Skip Count	0
Action	Write Data to File
File	C:\Users\Dennis Koufos\Documents\MQP\Matlab\receiverData...
Format	Integer
Start Address	0x10000
Length	4601
Miscellaneous	
Group	Default Group
Name	Breakpoint

The length of memory to access. The expression does not have to be constant as it will be re-evaluated before each access

Figure 35: Breakpoint Properties Menu

Once Code Composer Studio is set up correctly, the various blocks of the visible light communication system can be interfaced with one another. The LAUNCHXL-F28027 is connected directly to the PC via USB to mini-USB. J6 Pin 1 of the LAUNCHXL-F28027 is connected to the anode of the LEDs, and the ground pin of the board is connected to the ground on the protoboard of the analog circuit.

From Figure 12, J6 Pin 1 is connected to C13 on the protoboard, and the ground pin of the LAUNCHXL-F28027 is connected to the negative rail of the protoboard.

To power the analog circuitry of the receiver, two 6 V sources are needed. The positive terminal of the source is connected to one of the positive rails on the protoboard, while each negative rail is connected onto the protoboard on the negative rail directly next to the corresponding positive rail. The output of the op-amp, pin C17 on the protoboard, is connected to pin 6.0 on the MSP430F5529, where the ADC will collect data on the output of the photodiodes. The ground pin of the MSP430F5529 is connected to one of the grounded rails on the protoboard.

To send a message, first run the “decoding.m” MATLAB script. Using the computer on the transmission side of the system, edit the characters in the variable “msg” in the “msh.h” file to some other 11 character text. Once this is complete, click on the debugger icon and run the code.

Two breakpoints will need to be set on the receiver code, occurring on the last two lines of the first “else” statement. The first will be the breakpoint that needs to be configured as described above. The next breakpoint will be a standard halting breakpoint. Once the breakpoints have been set on the receiver code, click on the debugger icon and run the code. The code should automatically stop when it is done collecting data, and MATLAB will output the transmitted message.