AWESUM: AUDIO WI-FI EMBEDDED STREAMING USING MICROCONTROLLERS

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Joseph Michael Montanino

Evan Michael Respaut

MEng Field Advisor: Bruce Robert Land

Degree Date: May 2012

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Wi-Fi Microcontroller Audio Receiver

Authors: Joseph Michael Montanino, Evan Michael Respaut

Abstract:

The goal of this project was to create a system to wirelessly broadcast an audio signal from a computer to a set of speakers using Wi-Fi. This allows one to play music files from a computer and have the sound come out of any speakers that are in range of the wireless network. The ideal use case for this product would involve the ability to have a computer in one room processing music files while multiple speakers throughout the house are actually playing the music. This would be particularly useful in a party setting where one would like to keep a computer safe in a locked room while still being able to use it to play music. Additionally, if a party is there are sets of speakers in multiple rooms, they can all be synced to the same audio source. The major components of the system are the microcontroller receiver module (an ATMega 328p) and the computer program that sends the packetized audio data.

Table of Contents

EXECUTIVE SUMMARY

The goal of this Master of Engineering project was to create a system that uses Wi-Fi to transmit audio from a source such as a laptop to a speaker system. The final product combines the use of embedded hardware, low level software programming, and the IEEE 802.11 standard protocol for wireless communication (Wi-Fi) to create a polished end device. The hardware and software was developed using a combination of original work as well as open source code and libraries.

The initial concept for the project stemmed from the idea of wirelessly transmitting audio from a laptop in a room to a central set of speakers. The goal of this project was to do this without needing to connect an external dongle to the computer. A number of different options were explored, but it was agreed that the protocol of Wi-Fi made the most sense to use as a communication method since it comes equipped in all modern laptops. After some initial research, it was discovered that no commercial products offer the service of audio over Wi-Fi, with the exception of Apple's proprietary AirPort router. Further research was conducted to determine the most suitable Wi-Fi enabled microcontroller module. This research discovered a device called a RedBack, which is a single package that integrates a microcontroller with a Wi-Fi transceiver. The RedBack proved to be the backbone of the Audio Wi-Fi Embedded Streaming Using Microcontrollers (AWESUM) project.

The workload of the project was evenly split, with one group member working on the computer transmitting end of the project and the other member working on the microcontroller receiving end. Relatively quickly, both aspects became interdependent and it was necessary for both group members to work together to finish the project. A number of difficulties were tackled, ranging from understanding foreign code libraries, to working with Linux code on a Windows machine, to discovering problems related to underpowered hardware components. Additionally, many challenges arose from the difficult nature of debugging two systems that attempt to communicate with one another. When a problem arose, it was often be very challenging to tell if the microcontroller or computer was causing the problem.

In the end, the AWESUM project was able to achieve a relatively reliable UDP packet stream of 8-bit 22 kHz uncompressed audio. One receiver module was able to play mono sound, while two receiver modules could be combined to achieve full stereo sound. With only 2 kB of onboard RAM, and only 1 kB of the RAM available for variable storage use, the audio buffer size of the receiver was very small. This meant that the audio stream had to be sent at a very consistent rate or audio anomalies would occur. Despite this small buffer size, only occasional audio defects could be heard.

Some improvements could have been made to the project with regard to both the transmitting and receiving end. Increasing the size of the RAM on the receiver would have allowed for more robust playback. The computer program on the project's transmitting end could have been improved to allow for better user interface and a wider array of playback options.

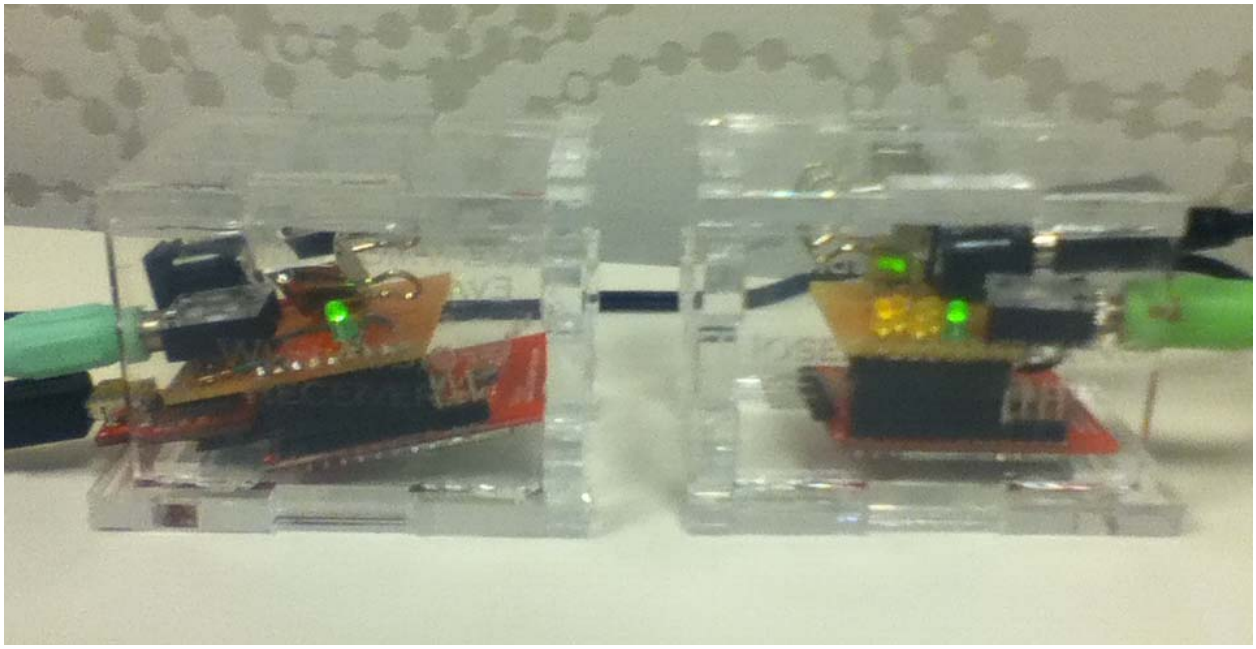


Figure 1 – Image of the Final Prototype

DESIGN PROBLEM AND REQUIREMENTS

The main goal of this project was to achieve wireless audio streaming from a computer to a set of speakers. This allows one to use speakers to play music from a computer while allowing the computer and speakers to be in different rooms in a house. This project serves to provide a feasible solution at minimum cost.

Previous Relevant Work

After doing a lot of research on the web, it was determined that little open-source work has been done in this area. The main inspiration for this project stemmed from Apple's AirPort Express router. This device allows one to transfer information from a computer to various devices (including printers and speakers) over Wi-Fi. Using a proprietary protocol called Apple AirPlay, in combination with the AirPort Express, one is able to stream music over Wi-Fi just as desired in this project. The fundamental restriction on this system is that it is proprietary. Even though it can be used with non-Apple brand speakers, it still requires the use of iTunes to stream music. If one wishes to use a different media player, the streaming will not work. Additionally, this product becomes the router for the network so it cannot interface with an existing network. Finally, the Apple AirPort Express system only supports the use of one set of speakers. This lack of flexibility is a major design flaw that this project sought to eliminate.

How Requirements Evolved

At the beginning of this project, there were many design requirements that turned out to be infeasible given the technological and time constraints. The most notable of these requirements was the desired playback quality. Initially, the goal of the system was to stream CD-quality audio (16-bit stereo sampled at 44.1 kHz). After much testing and experimentation, this turned out to be impossible given the limited size of our receiving buffer. Since the microcontroller receiver used did not have the processing ability to handle a decompression scheme, the audio was streamed in an uncompressed format. This dramatically increased the necessary bit-rate to stream audio. This high volume of data combined with the small amount of memory on the receiver, greatly limited the speed at which audio packets could be

transmitted. As such, the quality of sound was reduced to 8-bit stereo at a 22 kHz playback rate.

Another design requirement that did not make its way into the final implementation was the ability to interface directly with the sound card of a computer so that any sound from the computer could be streamed to the receiver. This feature would have been very useful as is would have allowed any media player to be used to stream music. Given the time constraints, this feature was not attempted. In the final implementation, the audio is streamed by reading WAV files from the computer and breaking them into chunks of data using a C++ script. This script then transmits each chunk as a User Datagram Protocol (UDP) packet.

List of Design Specifications

- 2 microcontroller/WiShield components
- 22 kHz sampling/playback rate
- 8-bit stereo (one channel to each microcontroller)
- Static IP address
- Home network
- Supports open, WEP, WPA, and WPA2 securities
- Uncompressed audio format (wav)

RANGE OF POSSIBLE SOLUTIONS

As with any project of this magnitude, there were several ways that this problem could be solved. The final system design was arrived at by making careful decisions at each step of the design process.

The most fundamental decision that had to be made was how to actually stream the audio from the source to the receiver. There are several methods of wireless communication. Some of these include IEEE 802.11 (Wi-Fi), Bluetooth, and Radio Frequency (RF). Each of these could handle the required data rate, however RF would require the use of a dongle that would need to be attached to the computer (via a USB port for example). This would cut down on the overall portability of the design. As far as Bluetooth is concerned, while some computers have the ability to use Bluetooth, it is far less common than Wi-Fi. Because of these reasons, we opted to use Wi-Fi. This would allow all laptops and most desktops to make use of the system without requiring any additional hardware.

After determining that Wi-Fi would be used, the next step of deciding which transport layer protocol to use. The two most common and most supported transport layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is connection-oriented and very reliable but quite slow, while UDP lacks some reliability but it very fast. Since the information of concern in this project is audio, and since there is a constant stream of it, speed was favored over reliability. UDP was chosen as the transport layer protocol, which allowed the data rate to be the fastest it could be. The only disadvantage of using UDP was that there was potential to drop packets every so often. However, this is of little concern because a single packet only contains approximately 30 milliseconds of sound, so if one is dropped, there will simply be a gap in the sound which is barely perceivable.

Another major design decision was how to receive the data on the speaker end. The main requirement of the receiver was to be able to take a stream of bytes of data and convert it into a digital signal. A microcontroller seemed like an obvious choice to do this. The microcontroller used was an ATMega328p which was interfaced directly with a wireless receiver called WiShield. An alternative to this would have been to buy the microcontroller and wireless module separately and interface the two. This would have allowed for more flexibility

in terms of microcontroller power. At the time, it was difficult to determine exactly what specifications were necessary to perform audio playback at the desired rate. The ATMega328p contained 2 KB of Random Access Memory (RAM). 1 KB of this was utilized in handshaking with the WiShield. This left 1 KB to do the buffering required for streaming. This limited buffer size is what ultimately limited the performance of the product.

The buffer size could have been increased by using external memory. However, this would require the use of the Serial Peripheral Interface (SPI) which was being used to interface with the WiShield. Given the time constraints, using external RAM did not make it into the final design. Alternatively, data could have been transmitted in a compressed format (such as mp3) and the receiver could decompress the data. In this way, more data could be transmitted given the fixed bit-rate. This scheme would require the use of an external mp3 decompression chip that would pass the uncompressed data to the microcontroller.

The computer side used code written in a Linux version of C compiled with GCC under a Cygwin environment in order to transmit the audio data. Alternatively, it would have made more sense to write and compile the code in a Windows environment, such as Visual Studios C or Java. The reason for using Linux was that the original example code provided was written in a Linux version of C and this managed to meet the needs of the system. Only toward the end of the project, when playback control was attempted to be added, was a higher-level programming language desired.

In the final design, the script running on the computer searches a directory for WAV files and plays them in order. This scheme has several limitations. Firstly, all music that is to be streamed must be in wav format. Secondly, each file must be in the same directory on the computer. A more universal way to stream would be to interface directly with the computer's sound card. This would allow the user to use any media player and still be able to stream the music.

DESIGN APPROACH AND CONTRIBUTION OF EACH MEMBER

This project was easily compartmentalized into two main components. The first is the transmitting end on the computer platform. The second is the receiving end on the microcontroller platform. The main approach used to tackle this project was to have one member (Evan) work primarily on the transmitting end, while the other member (Joe) worked primarily on the receiving end. Given the heavily interdependent nature of the project, much of the software development on both the transmitter and receiver end had to be done simultaneously. Debugging for this project was very time-consuming because it was often difficult to determine which end (transmitter or receiver) the bug was on. For this reason, a packet-sniffing program called WireShark was utilized to help diagnose some of the problems. The following sections outline the steps that were taken to get the system up and running.

The following block diagram represents the flow of information through the system:

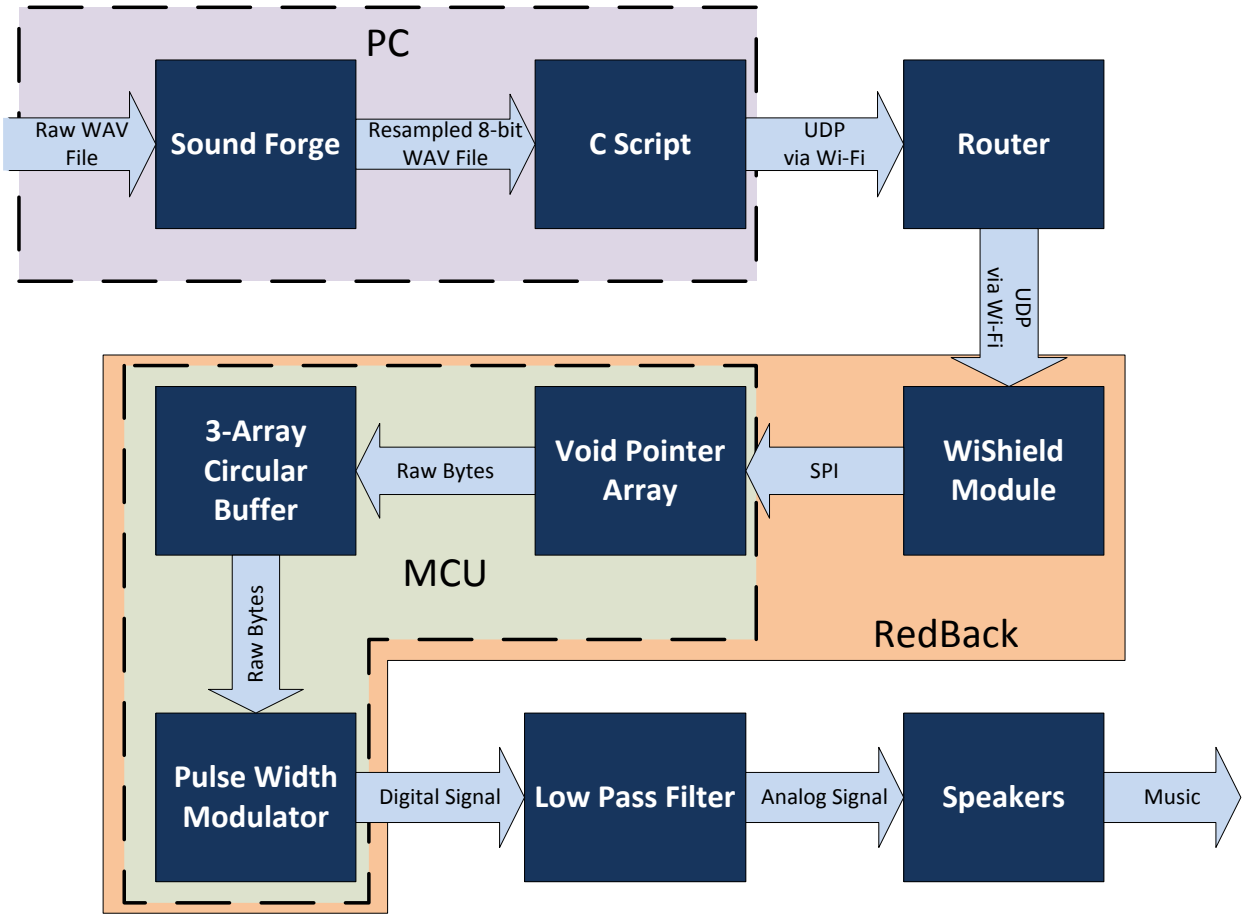


Figure 2 – Block Diagram of the AWESUM System

The Transmitter End (Evan)

The transmitter end proved to be more difficult than originally anticipated. Initial research was done on finding virtual sound card drivers that could capture audio directly from the computer. A few products existed that could do this, but none were found that could accomplish this in real time and also allow for an additional program to be added on that could then packetize the information. Next research was done on finding a preexisting program that would send a stream of audio out of the box. The media program VLC program has such a feature, however, this functionality was mainly designed for going between two computers. Furthermore, it is likely the flow control algorithms in place in this system couldn't be emulated by a simple microcontroller.

The final program used ended up being based off of a simple example given on a wiki created by AsyncLabs (the maker of the RedBack) that was written in C for Linux. At first, this code was brought into Visual Studios and external Linux libraries were added to try to get the program to compile. However, this method proved to be tedious and was not producing a compile-able result. Instead, a Linux environment emulator had to be used for windows, called Cygwin was installed, and a GCC compiler was added to the environment. Cygwin was then used to call upon the GCC compiler to compile the c code to then create a native windows executable file.

Next the code was used to verify communication between the microcontroller and computer. This "Hello world" program worked after a few tries, although proved to be rather unreliable, which was later discovered to be due to insufficient power on the microcontroller end. Once a reliable communication link between the computer and the microcontroller was established, a single 325 byte packet was sent to the microcontroller that contained a repeating sine wave. The transmission of this signal was a success (verified by the ability to play back the sine wave on a set of speakers on the receiver end).

After a single packet was successfully transferred, a sort stream of data was attempted to be sent. First a WAV file was loaded into the program, saved into an array, and was used sent to the microcontroller at a fixed interval. This successfully transferred to the microcontroller, although it was discovered from WireShark that the interval at which the

packets were being sent from the computer was not regular. A method of flow control was added on the microcontroller end that would acknowledge each incoming packet. The computer end would then wait for this acknowledgement before sending the next packet. This method proved to work quite well, except any time a packet was dropped, the audio would hang. Finally a timer was added to the computer program that would send out the next packet regardless of whether or not an acknowledgement packet was received after a set interval. This cleared up the occasional problem of a dropped packet and provided a decently reliable stream of data.

The Receiver End (Joe)

The first step was to research the technology available for communicating with Wi-Fi on an embedded platform. Since it was decided that our system would stream audio in an uncompressed format, the necessary bit-rate was determined to be approximately 1.4 Mbps in order to stream 16 bit stereo audio at 44 kHz. The chip that was purchased was called the RedBack, manufactured by a company called AsyncLabs, and claimed to have a max throughput of around 2 Mpbs. This device was essentially a printed circuit board that integrated an ATMega328p with a wireless receiver called WiShield 1.0. This chip had the ability to interface with Arduino libraries and could additionally use an open-source implementation of the TCP/IP stack for Wi-Fi. This detail was critical because it meant that the TCP/IP stack would not have to be written by hand.

The next step in the process was to program the RedBack module to connect to the network. A simple “Hello World” program was used to test the connectivity of the receiver to the network. This program relied on sample code that was provided as part of the Arduino TCP/IP libraries. Modifying this code proved to be non-trivial. A lot of work had to be done to figure out exactly which parts of the code had to be modified and how. The necessary network parameters of the router we used (including the IP address, default gateway, and subnet mask) had to be hard-coded into the program memory of the microcontroller in order to be used on the network.

Once the wireless module could successfully connect to the network, a simple handshaking algorithm was used to ensure that the transmitter and receiver could communicate with one another. This was then used to determine the maximum packet size the WiShield could receiver. This was empirically determined to be about 350 bytes. In order to increase the reliability, packets of size 325 bytes were used in the final implementation.

Additionally complexity was thrown into the mix when hardware timers were used to clock an interrupt service routine (ISR) and a pulse width modulation (PWM) function. These timers required some time to set up since the WiSheild library was using one of the three times. Furthermore, finding the proper registers to configure also took a bit more time than was expected. In the end, the performance of the Wi-Fi capabilities of the device where not compromised as a result of the hardware timers.

Pulse width modulation works by encoding analog values into a digital square wave by changing its duty cycle. The following image represents this technique:

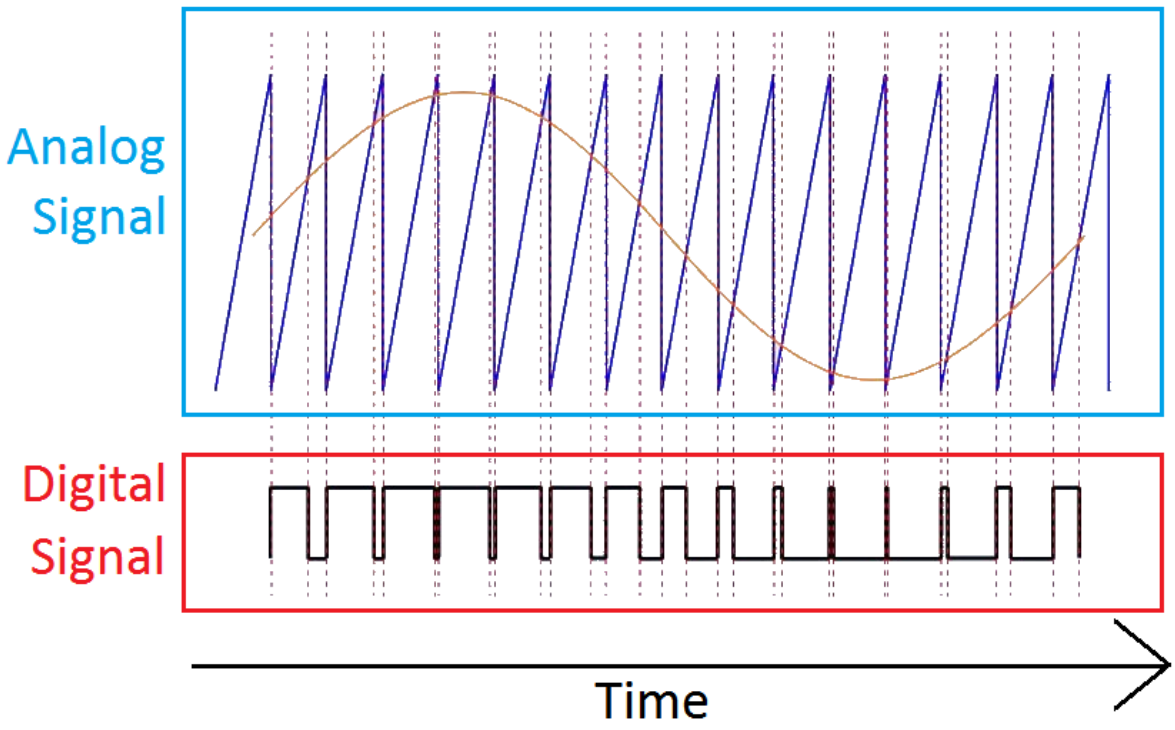


Figure 3 – Pulse Width Modulation in Action

The ISR was used to update the duty cycle of the pulse width modulator at a constant rate. These updates would occur exactly at the sampling rate (22 kHz). It was necessary to use the

hardware timers instead of timing in software because the timers are much more steady and reliable and this was a necessity for the audio playback. Adding in these interrupts to the code introduced some bugs with the wireless receiving of packets that had to be debugged.

Once the pulse width modulator was working alongside the packet receiver, a simple test was used to ensure that the PWM was doing its job. This test consisted of generating multiple samples of a sinusoid and playing these samples using the PWM. This test worked quite well, so the next step was to play the data received from the computer. The first step was to receive one period of a sinusoid and simply iterate over it on the microcontroller side.

In order to get continuous streaming working, a buffer cycle was developed on the microcontroller. Since it was discovered that the maximum UDP packet size was about 325 bytes, and there was about 1 KB of RAM available for a buffer, a three array buffer cycle was developed, with each array consisting of three 325-byte arrays. This allowed for the 1 KB of available memory to be filled up without going over the hard packet size limit of 325 bytes. As the information got depleted from one of the arrays, the next array in the sequence replaced it and the empty array was filled with new data. The following diagram helps to illustrate this process graphically:

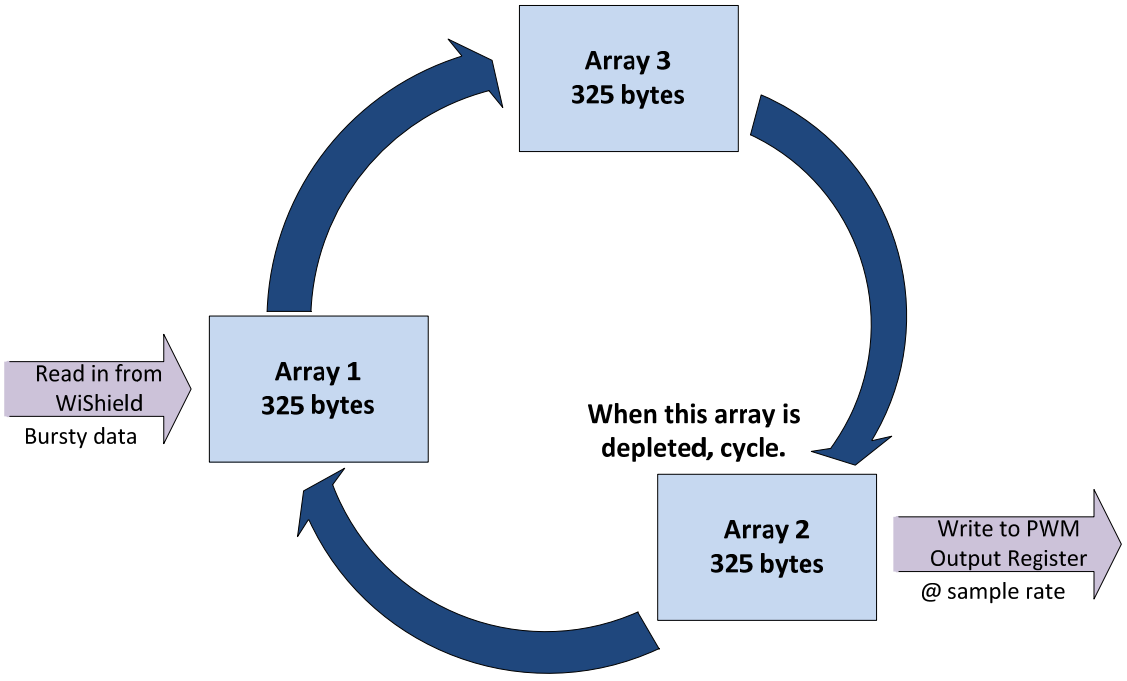


Figure 4 – Buffer Cycle

At first, the microcontroller received a one second window of music and used the PWM to convert the sequence of bytes into a digital signal. Once this functionality worked, the microcontroller could be used to receive a continuous sequence of music.

Hardware (Evan)

In comparison to the software, the hardware was not very involved in this project. The bulk of the hardware was contained on a small solder board that attached to the RedBack using header pins and sockets. LEDs were soldered into the board to use for debugging purposes. A 3.5mm audio jack was soldered on as well so that one could easily attach the receiver to a set of speakers. Leading into this audio jack was a low pass filter that was created out of a resistor and capacitor. This low pass filter served to attenuate unwanted high frequencies that were not part of the audio signal.

At first, the receiver was powered by the USB port. This was capable of supplying about 500 mA. However, this turned out to not be enough power to support the WiShield, the microcontroller, and all of the LEDs. This was discovered the hard way as sometimes when packets were received, the power would reset. It took a long time before it was realized that the system was not getting enough power. A power socket was soldered onto the board and a voltage regulator was used to bring the raw voltage down to 5 volts. The current rating on the regulator was 1.5 A so this was more than enough power to feed the receiver.

DESIGN AND IMPLEMENTATION

The project had three major components--the software running on the computer, the software running on the microcontroller end, and the hardware. The software on the computer end of the project was written in C for Linux and compiled using GCC under a Cygwin environment. The code running on the microcontroller was a combination of C and Arduino's version of C++, and was compiled using Arduino's integrated development environment. The hardware was a combination of a purchased Arduino microcontroller/Wi-Fi module (called a RedBack 1.0) and some additional supporting hardware for power and connectivity. When outputting stereo sound, the setup includes two RedBacks, one for each channel of sound.

Computer Software

The software running on the computer end of the project was written in a Linux based version of C and was compiled using GCC within a Cygwin environment. The program itself is relatively simple--it consists of a large infinite loop that loads in an uncompressed WAV file, breaks the file into 325 byte chunks, and sends these chunks though a socket in the form of UDP packets. The program then waits for a response back from the microcontroller before sending the next chunk of data. This process is repeated until the end of the song, at which point the program loads the next audio file in the directory.

The process of loading a file into the program only consists of a few lines of code. The first step the program takes is to ask the user what playlist and song is desired. The user inputs a number for the playlist desired, and this number gets translated into a folder name, with the structure "playlist#", where the # is a number picked by the user. Note that this folder must exist before the program runs. Next the program asks the user for a song number. In a similar fashion to the folder, the program will look for the WAV file named "song#.wav", where # is the number specified by the user.

The program then enters an infinite loop and initializes a couple of arrays are initialized that will be used for saving the 325 byte chunks. It then prints the number of the song and playlist currently being played and loads the file into a FILE pointer structure with the fopen function. If the file pointer cannot be found, the program errors and exits. Next the program

finds the file size and prints this to the console. It then skips the first 44 bytes of the WAV file so that the header information sent to the microcontroller.

Next the program initializes a bunch of variables and structures used for setting up the network connection with the Arduino RedBacks. This includes selecting the IP address of both of the RedBacks. Note that these IP addresses are selected in the microcontroller code and are static. The communication ports are set up and then the socket is setup and connected using the socket function. Next the computer binds the ports on the PC that the RedBack will communicate back through.

In the next section of code, the computer program runs through a large for loop that iterates over the entire WAV file in 325 byte chunks. Within this for loop, a timer is initialized to about 15ms and is used to keep the program on track in case a verification packet doesn't come back from the microcontroller on time. Next, a loop runs that loads 325 bytes into the two arrays responsible for left and right audio using the fgetc function. For an 8-bit WAV file, the audio is encoded as one byte left for the left channel and one byte for the right channel. However, if the program detects that it is nearing the end of the WAV file, it will load zeroes into the arrays so that the microcontroller outputs no sound when the song completes. Next the program sends the UDP packets to the two RedBacks using the sendto function. Since these two packets (one for left and one for right channels) are sent roughly the same time, there is no delay between the two speakers at playback. Next the program prints the current packet it is sending along with the playlist and song numbers. Finally the program blocks and waits for one of the microcontrollers to send back a response packet using the recvfrom function. It is not necessary to wait for a response from both microcontrollers since the two are in sync anyway. If the response doesn't come, the timer that was set up previously will break the program out of the blocking recvfrom function and will continue on in operation.

Microcontroller Software

The software running on the RedBack module is very complex and has many components. Included with the RedBack module was a large library (called WiShield) that supports includes a small TCP/IP stack and performs the low level networking operations. This library was very poorly documented and required a large amount of time to understand and interface with. The

main code for this project includes an Arduino program, which is top level file for the program. This Arduino file sets up the PWM and timers, and initializes and calls the wireless module libraries and but is a relatively small piece of code. In addition to the Arduino file (called UDPApp.ino), there is a c source and header file called udpapp.c and udpapp.h respectively. These files are the heart of the program, and are responsible for dictating how the microcontroller receives and transmits packets, as well as saving incoming data into arrays.

The Arduino program is the first file that the Arduino IDE looks at and compiles. As a result, this file is considered the highest level file and is used to call all of the supporting files. The first line of code in the file includes the library file WiShield.h, which in turn includes a number of other files. These supporting files are responsible for setting up the TCP/IP stack, configuring UDP settings, and setting up the SPI connection from the microcontroller to the Wi-Fi module. The next line of code includes an external header file called TimerOne. This file is responsible for defining some function calls that make setting up timer1 easier.

The next block of code in the Arduino file is responsible for configuring up the connection to the router. The desired IP address for the RedBack is selected, and the gateway IP and the subnet mask addresses of the router are inputted. Next the SSID, the security type (either open, WEP, WAP or WAP2), passcode and/or WEP keys of the desired router are set. Since these items are hard coded into the code of the microcontroller, RedBack is unable to switch routers or router settings without the code being re-compiled.

Next, the Arduino program deals with the three 325 byte arrays that hold the audio data. These arrays are first declared as volatile char arrays, since they are read in an ISR and need to be one byte in size. Next the ISR function is specified. This function is called once every sample, or 1/(22 kHz), approximately 45 us. Within this function, one byte out of one of the three arrays is read out and placed into the OCR2B register. This register is responsible for setting the duty cycle of the timer2 PWM. Since this PWM is running at approximately 62 kHz, once this signal gets low passed filtered, it looks virtually like an audio signal. A variable named globalX is responsible for keeping track of which array should be read from and written to. Every 325 runs of the ISR, the globalX variable is updated to a new value to effectively swap the

arrays' functions. Note that the globalX variable and each of the arrays are written in the udpapp.c file, and as such are declared as externs in that file.

The next block of code in the Arduino file is the setup function. This function first initializes a bunch of pins to outputs that are used for the PWM, the green LED signifying that the RedBack is connected to a router, and for a number of testing LEDs. Next the green router connectivity LED is turned off and the Wi-Fi initialize function is called. This function is part of the WiShield library and is responsible for setting up the Wi-Fi module as well as for connecting to the router and performing a number of other initial setup tasks. The next chunk of code sets up PWM on timer two and sets the frequency to 62 kHz, much faster than the highest frequency audio signals that the PWM needs to be able to emulate. Timer one is then setup and is responsible for calling the ISR function once every sample (45 us). The sei function is then called to enable the timers and the green LED is set on to signify that the receiver module is ready to receive audio from the computer. Finally the main loop function is called and it calls WiShield's run function that is responsible for running the TCP/IP stack as well as running the code in the udpapp.c file.

The udpapp.c file is responsible for performing the tasks of communicating to the computer via UDP as well as updating the globalBuff arrays that are used to store the audio data. This file also declares a protothread function, which is responsible for sending and receiving UDP packets. A protothread is a very lightweight threaded system that allows for multiple processes to be working virtually simultaneously. This is necessary in the realm of Wi-Fi since the wireless module must have CPU cycles in order to perform the low level tasks necessary to communicate wirelessly.

The udpapp.c file starts off by including a few of WiShield library files. The first of these files is the header file uip.h. This file is responsible for setting up the TCP/IP stack, where the associated source file called uip.c actually runs this stack. The UIP (micro IP) TCP/IP stack was designed for 8-bit microcontrollers by Adam Dunkels, and is very lightweight and relatively small (only about 3000 lines of code). The file udpapp.c then includes string.h, udpapp.h, and config.h. String.h is used for basic string operations, updapp.h declares a few different structures and functions, and config.h is WiShield file that is responsible for setting up WiShield

settings, such as using the UDP protocol as opposed to TCP. Next globalX and the globalBuff array buffers are declared as extern volatile variables. The extern keyword makes sure the compiler knows to look for these global variables elsewhere. The volatile keyword is necessary to tell the program that the variables might change value at any point.

Next, the udpapp.c file defines a number of functions. The first function is an initialize function that sets up the IP address and communication ports of the computer that the RedBack will be communicating to. Note that like the static IP address of the RedBack, the IP address of the computer must be set before compilation. If the computer changes IP addresses, the program must be updated, compiled, and re-uploaded to the microcontroller. The next function is a supporting function called loadBuffer that is responsible for saving each UDP packet's data into one of three arrays, depending on the value of globalX (which is set in the ISR responsible for reading out the audio data from the arrays and keeping the sampling rate consistent). Next, there is a function called sendReady. This function is used to send a single character to the computer to tell the computer that the microcontroller is ready for another packet.

The next main function in the udpapp.c file is the main protothread function, called PT_THREAD. This function starts by calling the PT_BEGIN function to tell the WiShield library that the thread is starting. Next an infinite loop is entered. This style is not exactly in line with what protothreads are designed to do, but since the purpose of this application is a constant stream of data, it doesn't make sense to ever terminate the thread. Within this infinite loop, the program first calls the macro function PT_WAIT_UNTIL with the macro function uip_newdata() as an input parameter. This function performs a non blocking wait as it waits for new data to arrive from the computer. The reason this macro function is non-blocking is because it allows the WiShield background tasks to run as the program waits for new packets to arrive. Once the WiShield receives a new packet, the protothread moves on and the UIP_NEWDATA flag is cleared so that the blocking wait statement doesn't get miss triggered. Next the data from the packet is loaded into one of three arrays using the loadBuffer() function previously described. After the array is loaded, the protothread waits for the value of globalX to change in an infinite while loop. It is ok to use a blocking wait here, since the WiShield isn't

expecting to receive any packets during the length of the blocking wait. The value of globalX will change when the ISR moves from one array to the next. Once this happens, one of the microcontrollers will sent out an acknowledgement to the PC to tell the PC it is ready for the next packet, using the function sendRead(). The program will then loop back to the non-blocking wait that waits for information from the PC.

Hardware Description

The main hardware of the project is a purchased board called a RedBack. This board includes the microcontroller, Wi-Fi module, and the header pin hookups for IO signals as well as power. Additionally the board includes a few LEDs for power and for Wi-Fi connectivity. For this project, a second board was attached to the RedBack that includes a 5V regulator, a power plug, a few LEDS, a low pass filter, and an audio jack. All of the hardware is approximately 2 inches cubed. Finally a small acrylic box designed and created using a laser cutter to keep the hardware safe.

RESULTS

The final system implementation was capable of streaming 8-bit stereo with a playback rate of 22 kHz. This audio quality sounded good enough to be usable in the home. With one microcontroller Wi-Fi receiver, 8-bit, 22 kHz uncompressed mono sound could be streamed from the computer to a set of speakers with minimal glitches. With two microcontroller modules (one for each left and right channel), 22kHz uncompressed stereo could be transferred. Each microcontroller only has 2kB of RAM on board, and after the program is fully loaded onto the microcontroller, less than 1kB of this memory is available for storing variables. As a result, the audio buffer was only 975 to 650 bytes at any given time. Therefore the actual length of the buffer in terms of time is only 45 to 30 ms. As a result, the audio receiver module is very sensitive to packets that are delayed or arrive too soon. Regardless, due to very careful and exact flow of packets from the PC to the microcontroller, only occasional blips in the output audio can be heard.

For the most part, the audio could be streamed continuously without interruption. Every once in a while (on the order of tens of seconds, an occasional crackle or pop could be heard. This was attributed to a packet being dropped. This phenomenon was unavoidable. A potential work-around for this problem would be to increase the size of our receiver buffer. This would allow each packet to be checked for continuity. If a packet was not received, there would be enough latency in the system to request the packet be transmitted again. This greatly reduced the probability of these types of audio artifacts.

Another interesting artifact was occasional stuttering. These stutters would last for approximately one second and would occur every 42 seconds. These artifacts were caused by the microcontroller not receiving new packets for an entire second. The old data output over and over again until new data was received. This problem is attributed to the router that was used for debugging the system. Every 42 seconds, the router would broadcast a packet to search for new devices on the network. During this time, it would not send or receive any packets. The computer script would continuously send packets, but they would not be received during this time. A work-around for this problem would be to increase the buffer size so that

more than a second of audio could be stored on the microcontroller. This would result in having enough new information stored to outlast the lack of incoming information.

Another interesting bug would occur quite sporadically. Essentially, the code that was in charge of streaming packets would occasionally crash. The error message returned was a memory access error. This is believed to be a problem with the way the code was reading WAV files from the computer. If other tasks on the computer attempted to use memory that our script was using, it could cause a fatal error as seen. Given that this error occurred on an irregular basis, it proved to be quite difficult to fully remove from the system.

Original Expectations

In comparison to the original expectations of the system, the final design delivered serves as a nice first prototype.

There is a lot of improvement that would need to be made for this product to be deemed fully usable. The user end of the system could use a great deal of refining and additional functionality. This would require the use of a Graphical User Interface (GUI) that would allow the user to interface music files with the streaming script in a more intuitive and effective way. Furthermore, the streaming reliability is currently less than desirable. The main issues are the risk of crashing as well as the periodic stuttering (both alluded to above). Finally, the audio quality is not quite as good as the goal. The initial desired quality was 16-bit stereo at 44.1 kHz while the final quality that was implemented was 8-bit stereo at 22 kHz. Thus, the implemented quality was a factor of 4 lower than the desired quality.

On the other hand, the system nicely demonstrates some of the key concepts. The most important of these concepts is the ability to process audio on an 8-bit microcontroller. This, in it of itself, is a huge benchmark. Another important concept is the ability to interface with Wi-Fi and UDP, especially at the high data rate that was used. Lastly, the system demonstrates the idea that the same audio information can be sent to multiple receivers in real time, which allows multiple sets of speakers to be synced to the same source.

Below is a plot of a packet stream sent from the PC to the microcontroller. The packets are represented as impulse functions and are graphed over a period of about 14 seconds. The size of the impulse is the time between delay between one packet and the next. So delayed

packets appear as larger spikes and early packets appear as smaller spikes. The left hand side of the graph is the time delay between each packet. Note that most of the packets are on the order of about 15ms or so. This is to be expected since the size in time of a single packet is about 15ms ((325 samples/packet)/ (22 samples/second)). However there are some irregularities between the packets, especially at around 170 seconds. This large one second gap is caused by the router sending a large amount of broadcast packets out (represented by the green spike). As a result of these packets, the router stops listening to the packets from the computer for a period of time, which in turn causes the audio output from the microcontroller to stutter.

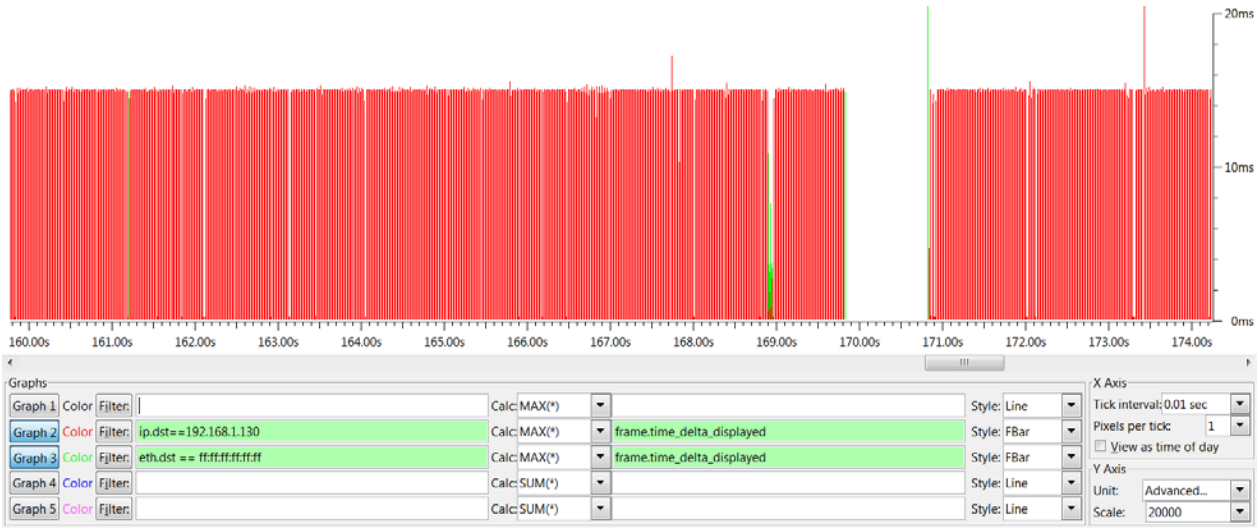


Figure 5 – Packet delay over time (from WireShark)

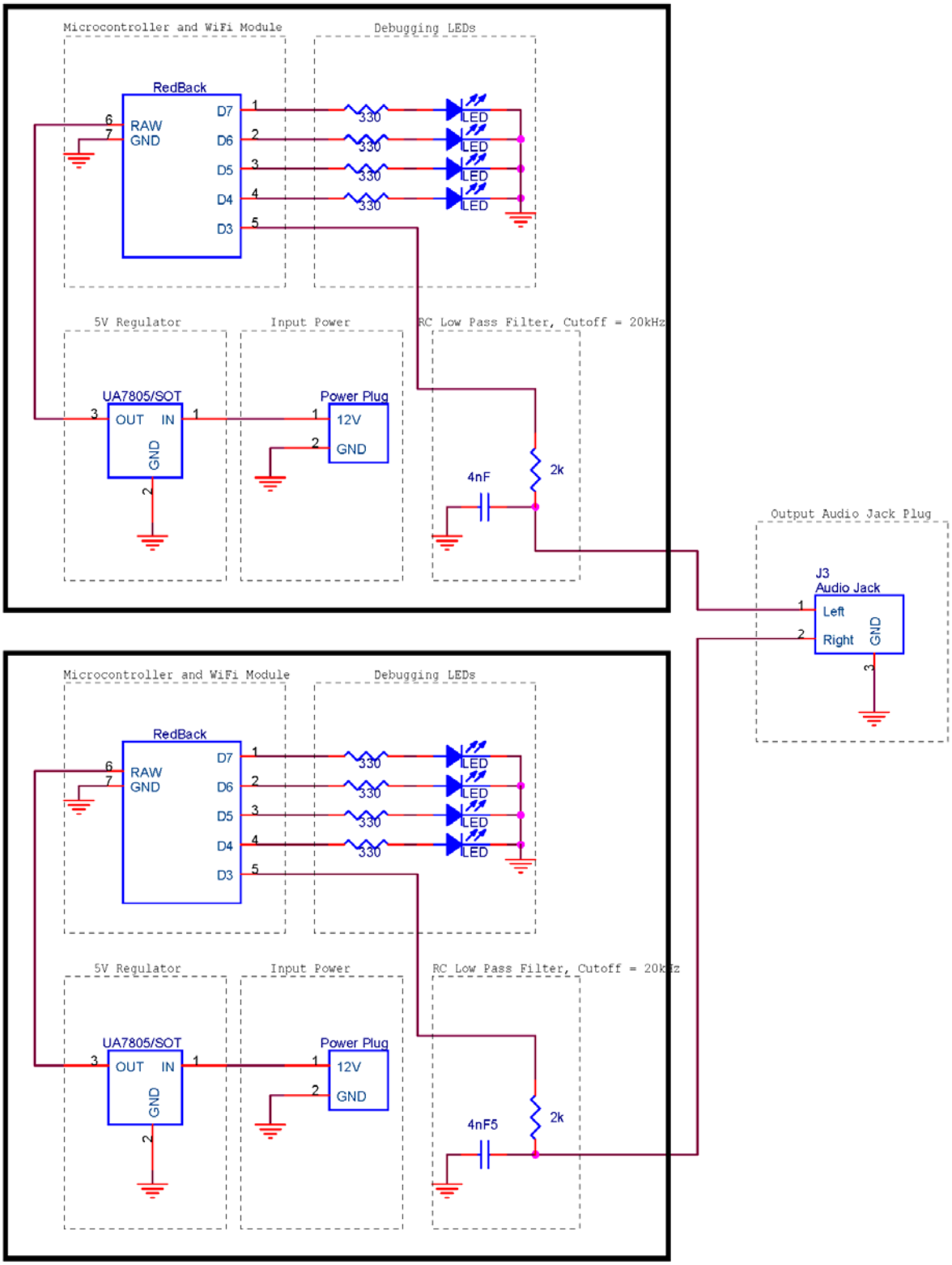
CONCLUSION

The end result from this project ended up working amazingly well considering the hard limitations in the hardware. The audio quality was very decent considering the small buffer limitation of the microcontroller being used. Furthermore the system was relatively robust and could recover from seconds of lost packets (although the audio signal would be lost for this period of time).

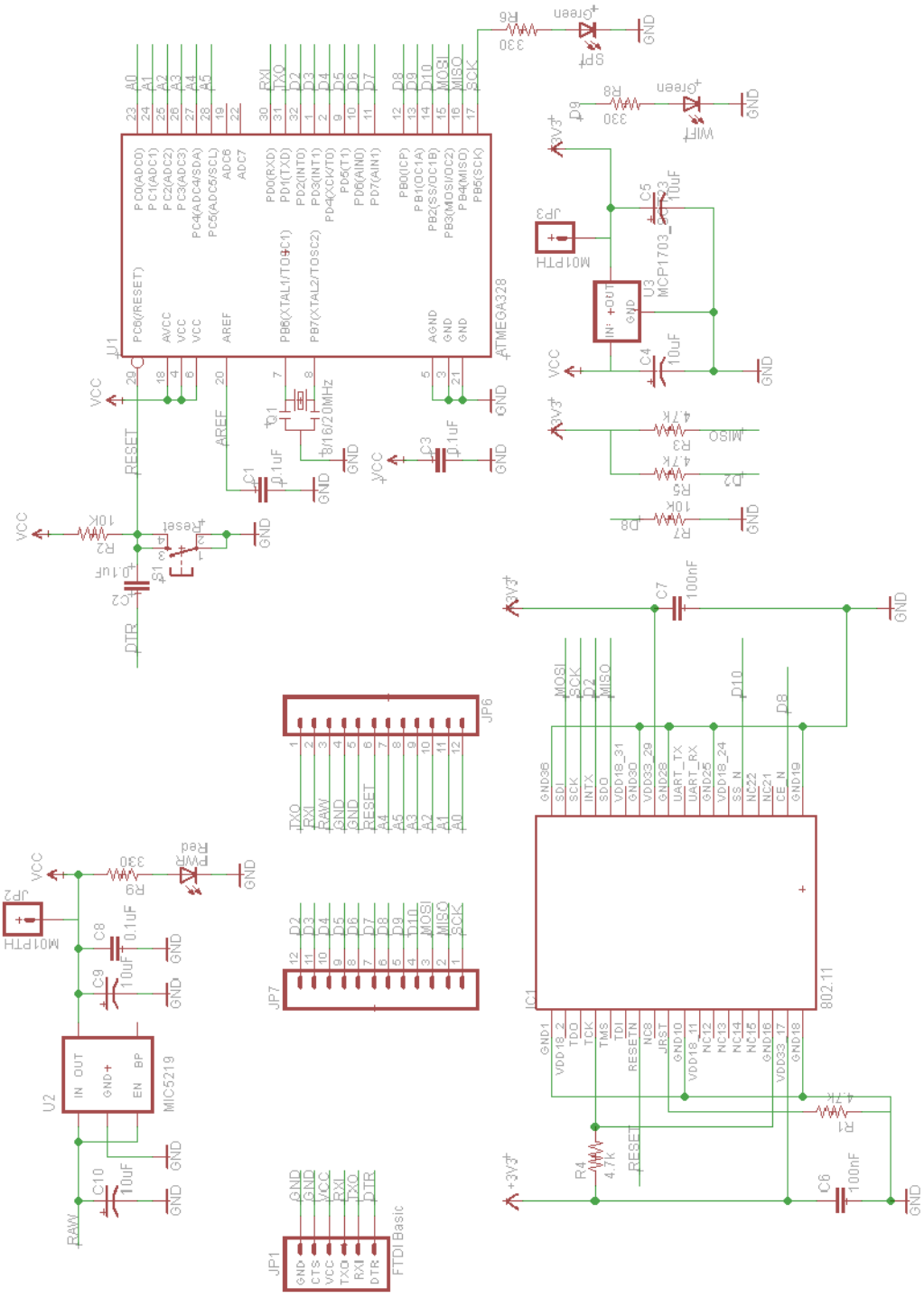
Although the outcome of the project was very impressive, there are a number of things that could be improved in this project. One of the largest improvements would have been to use a microcontroller with more onboard RAM, or add external RAM. This would have allowed the audio buffer to be longer, thus decreasing the receiver's sensitivity to untimely packets and other unexpected networking anomalies. Another possible improvement is the code on the computer end of the project. The user interface could have been drastically improved to allow for easier playback options. Furthermore, support for files other than uncompressed WAV could have been added. This could have even been taken a step further, and a virtual audio driver could have been created that would have streamed any audio being produced on the computer to the microcontroller receiver. This would have allowed for any audio program on the computer to be used.

A number of things have been learned throughout the course of this project. One of the most important lessons learned is that lots of research pays off in the end. More research should've been done in regards to the type of microcontroller being used as well as its amount of on board RAM. This being said, the RedBack microcontroller and Wi-Fi module was not inherently compatible with other microcontrollers. Another lesson learned was that one should not be afraid to rewrite code in more usable languages. This lesson applies directly to the use of Linux based C code on the computer end of the project. This code most likely could have been adapted for Java, which would have allowed for much better user input controls as well as possible support for more complex future additions such as MP3 decoding or integration with a virtual sound card driver.

APPENDIX A – RECEIVER MODULE SCHEMATIC



APPENDIX B – REDBACK SCHEMATIC



APPENDIX C – PART LIST

Part	Quantity	Price	Part Total
RedBack Microcontroller Wi-Fi Module	2	$64.99	$129.98
Solder Board	2	$1.20	$2.40
LEDs	8	$0.15	$1.20
Audio Jacks	2	$0.85	$1.70
UA7805 5V Regulator	2	$0.99	$1.98
Grand Total			**$137.26**

APPENDIX D – SOURCE CODE

The following subsections list the code that was executed on both the transmitter and receiver end of the system. The libraries and associated header files have been omitted for brevity.

Computer/Transmitter End Code

```
//Computer/Transmitter End of AWESUM System
//compile with gcc -Wl,--stack,8388608 -o udp.exe udp_endp.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/time.h>

//ports for the left channel microcontorller
#define REMOTE_PORT          12344 // port on Arduino + WiShield
#define LOCAL_PORT           12345 // port on the PC

//ports for the right channel microcontorller
#define REMOTE_PORT2    12343 // port on Arduino + WiShield
#define LOCAL_PORT2          12346 // port on the PC


volatile sig_atomic_t keep_going = 1;

//interupt for the timer
void catchInterupt(int theint){
      keep_going=0;
}

//the size in bytes of a packet
#define numSamples 325

//nubmer of songs per playlist
#define numSongs 9

int main()
{
      //have user specify the folder to look for songs in
      int playlist;
      printf("Input Desired Playlist\n[1] Classic Rock\n[2] Other\n");
      scanf("%d", &playlist);

      //user specifies the first song to play
      int song;
      printf("Input First Song Selection (1 through %d)\n",numSongs);
      scanf("%d", &song);

      //main infinite while loop
      while(1) {
            //arrays to hold each left and right packet
            unsigned char maybeL[numSamples];
            unsigned char maybeR[numSamples];
```

```
            char songTitle[35];

      sprintf(songTitle,"convertedStereo\\playlist%d\\song%d.wav",playlist,so
ng);
            printf("Playing %s\n",songTitle);

            //file pointer for the WAV file
            FILE *ifp = fopen(songTitle, "r");

            //make sure file isn't empty
            if (ifp == NULL) {
                  fprintf(stderr, "Can't open input file!\n");
                  exit(1);
            }

            //find file size note 0L is a long 0
            fseek(ifp,0L,SEEK_END);
            int fileSize = ftell(ifp);

            printf("file size = %d bytes\n",fileSize);

            //skip the header
            fseek(ifp,44L,SEEK_SET);

            //declare a bunch of things for the sockets and packet sending
functions
            char buf[99];
            char buf2[99];
            int sockfd, numbytes;
            int sockfd2, numbytes2;
            struct sockaddr_in remote, local;
            struct sockaddr_in remote2, local2;
            size_t addr_len;
            size_t addr_len2;

            memset(&remote, 0, sizeof(remote));
            memset(&remote2, 0, sizeof(remote2));

            // setup IP address of left WiShield
            remote.sin_family = AF_INET;
            remote.sin_port = htons(REMOTE_PORT);
            inet_pton(AF_INET, "192.168.1.130", &remote.sin_addr);

            // setup IP address of right WiShield
            remote2.sin_family = AF_INET;
            remote2.sin_port = htons(REMOTE_PORT2);
            inet_pton(AF_INET, "192.168.1.131", &remote2.sin_addr);

            // setup socket and connect for left channel
            if ((sockfd = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
                  perror("socket");
                  exit(0);
            }

            // setup socket and connect for right channel
            if ((sockfd2 = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
                  perror("socket");
```

```
		exit(0);
	}

	// bind to a specific port on the PC for left channel
	local.sin_family = AF_INET;
	local.sin_port = htons(LOCAL_PORT);
	local.sin_addr.s_addr = INADDR_ANY;

	// bind to a specific port on the PC for right channel
	local2.sin_family = AF_INET;
	local2.sin_port = htons(LOCAL_PORT2);
	local2.sin_addr.s_addr = INADDR_ANY;

	//bind for left channel
	if ( bind(sockfd, (struct sockaddr *)&local, sizeof(local)) == -
1) {
		perror("bind");
		exit(0);
	}

	//bind for right channel
	if ( bind(sockfd2, (struct sockaddr *)&local2, sizeof(local2)) ==
-1) {
		perror("bind");
		exit(0);
	}

	int lostCount=0;

	//this function loops over the entire file
	int indexNum;
	for(indexNum=0;indexNum<(fileSize/(2*numSamples));indexNum++){

		//setup the timer for 15ms interval to make sure we dont'
		//wait for an ack for too long
		keep_going=1;
		struct timeval timerValue={0,15000};
		struct timeval timerInterval={0,15000};
		struct itimerval timer={timerInterval,timerValue};
		setitimer(ITIMER_REAL, &timer, 0);
		signal(SIGALRM, (sighandler_t) catchInterupt);

		//last 3 or so samples of the song, make silent
		if(indexNum >= ((fileSize/(2*numSamples))-3)) {
			int i;
			for(i=0;i<numSamples;i++){
				maybeL[i]=0;
				maybeR[i]=0;
			}

		}
		//load each packet array up using the fgetc function
		else {
			int i;
			for(i=0;i<numSamples;i++){
				maybeL[i]=fgetc(ifp);
				maybeR[i]=fgetc(ifp);
```

```
                        }
                    }

                    // send stuff to left WiShield
                    if  ((numbytes  =  sendto(sockfd,  maybeL,  numSamples,  0,
(struct sockaddr *)&remote, sizeof(remote))) == -1) {
                        //perror("sendto");
                        //exit(1);
                    }

                    // send stuff to right WiShield
                    if  ((numbytes  =  sendto(sockfd2,  maybeR,  numSamples,  0,
(struct sockaddr *)&remote2, sizeof(remote2))) == -1) {
                        //perror("sendto");
                        //exit(1);
                    }
                    addr_len = sizeof(remote);

                    //print some stuff
                    printf("Packet    #    =    %d    \tPlaylist    =    %d\tSong=
%d\n",indexNum,playlist,song);

                    //wait for ack from left wisheild
                    if  ((numbytes  =  recvfrom(sockfd,  buf,  99  ,  0,  (struct
sockaddr *)&remote, &addr_len)) == -1) {
                        //perror("recvfrom");
                        //exit(1);
                    }

            }

            //close connection
            close(sockfd);
            close(sockfd2);

            //close file
            fclose(ifp);
            song++;
            if(song==numSongs+1) song=1;
      }

      return EXIT_SUCCESS;
}
```

Receiver End Main Script

```
#include <WiShield.h>
#include "TimerOne.h"
```

```
#define WIRELESS_MODE_INFRA   1
#define WIRELESS_MODE_ADHOC   2

// Wireless configuration parameters ----------------------------------------
unsigned char local_ip[] = {192,168,1,131};     // IP address of WiShield
unsigned char gateway_ip[] = {192,168,1,1};     //   router   or   gateway   IP
address
unsigned char subnet_mask[] = {255,255,255,0};  // subnet mask for the local
network
const prog_char ssid[] PROGMEM = {"AudioOverWifi"};         // max 32 bytes

unsigned char security_type = 0;    // 0 - open; 1 - WEP; 2 - WPA; 3 - WPA2

// WPA/WPA2 passphrase
const prog_char security_passphrase[] PROGMEM = {"Sufficiently Witty"}; //
max 64 characters


//90 6E BB 2D 34 8F 00000000000000
// WEP 128-bit keys
// sample HEX keys
prog_uchar wep_keys[] PROGMEM = {0x90, 0x6E, 0xEB, 0x2D, 0x34, 0x8F, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // Key 0
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, // Key 1
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, // Key 2
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00  // Key 3
                        };

// setup the wireless mode
// infrastructure - connect to AP
// adhoc - connect to another WiFi device
unsigned char wireless_mode = WIRELESS_MODE_INFRA;

unsigned char ssid_len;
unsigned char security_passphrase_len;
//---------------------------------------------------------------------------
volatile int globalX=3;

//size of a packet
#define BUFFERSIZE 325 //CHANGE THIS IN UDPAPP.c TOO

//define the buffer cycle arrays
volatile char globalBuf1[BUFFERSIZE];
volatile char globalBuf2[BUFFERSIZE];
volatile char globalBuf3[BUFFERSIZE];

//used in the ISR
volatile int index= 0;

//ISR function that changes the PWM duty cycle each sample
void sample() {
  index++;
  //if we reach the end of an array, move to the next one
```

```
  if (index==BUFFERSIZE){
      index= 0;
      if(globalX==1) globalX=2;
      else if (globalX==2) globalX=3;
      else if (globalX==3) globalX=1;
  }

  //update the PWM depending on which array we're using for reading
  if (globalX==1) OCR2B = globalBuf2[index];
  if (globalX==2) OCR2B = globalBuf3[index];
  if (globalX==3) OCR2B = globalBuf1[index];
}

//set up some stuff
void setup()
{
  pinMode(3,OUTPUT); //timer2 PWM
  pinMode(4,OUTPUT); //test LED
  pinMode(7,OUTPUT); //New Green WiFi LED
  pinMode(6,OUTPUT); //test2 LED
  pinMode(5,OUTPUT); //test2 LED

  //green WiFi LED off
  digitalWrite(7,LOW);

  //setup the WiSheild and TCP/IP stack
  WiFi.init();

  //setup the timer 2 PWM
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);  // configure
timer2 PWM
  TCCR2B = _BV(CS20); // set the prescaler (not sure how much, full speed?)
  OCR2A = 0; // pin 11 PWM is off
  OCR2B = 50; // turn on pin 3 PWM

  // set up timer1 to keep the sampling rate
  //31us for 32.258kHz
  //45us for 22.222kHz
  Timer1.initialize(45); // 125 us => 8 kHz sampling rate
  Timer1.attachInterrupt(sample);

  sei();

  //green WiFi LED on
  digitalWrite(7,HIGH);
}

void loop()
{
      //run the main WiSheild function
      //calls the TCP/IP stack and such
      WiFi.run();
}
```

Receiver End Packet Reception Code

```
//Packet code
```

```
#include "uip.h"
#include <string.h>
#include "udpapp.h"
#include "config.h"

//define the buffer size
#define BUFFERBYTESIZE 325 //CHANGE THIS IN UDPAPP.ino TOO

//external variables from the UDPApp.ino file
extern volatile char globalX;
extern volatile char globalBuf1[];
extern volatile char globalBuf2[];
extern volatile char globalBuf3[];

void dummy_app_appcall(void)
{
}


//initialize the WiSheild
void udpapp_init(void)
{
  uip_ipaddr_t addr;
  struct uip_udp_conn *c;

  //this is the IP address of the computer
  uip_ipaddr(&addr, 192,168,1,100);
  //port number for the computer
  c = uip_udp_new(&addr, HTONS(12346));
  if(c != NULL) {
      //other port number for the computer
    uip_udp_bind(c, HTONS(12343));
  }

      //initialize the protothread
  PT_INIT(&s.pt);
}

//this function loads the UDP packet's data into one of the three cicular
buffers
static unsigned char loadBuffer(void)
{
  if(globalX==1){
    memcpy(globalBuf1,uip_appdata,BUFFERBYTESIZE);
  }
  else if(globalX==2){
    memcpy(globalBuf2,uip_appdata,BUFFERBYTESIZE);
  }
  else if(globalX==3){
    memcpy(globalBuf3,uip_appdata,BUFFERBYTESIZE);
  }

  return 1;
}

//sent an ack back to the computer signifying tha twe want the next packet
static void sendReady(void){
```

```
  char str[] = "R\n";

  memcpy(uip_appdata, str, strlen(str));
  uip_send(uip_appdata, strlen(str));
}


int prevGlobalX=3;

//declare the protothread
static PT_THREAD(handle_connection(void))
{
      //startup the protothread
      PT_BEGIN(&s.pt);

  //enter infinite loop
  while(1){

    //NOTE #define uip_newdata()   (uip_flags & UIP_NEWDATA)
    //threaded wait until UIP sets the UIP_NEWDATA flag high
    PT_WAIT_UNTIL(&s.pt, uip_newdata());
    uip_flags  &=  (~UIP_NEWDATA);  //clear  the  new  data  flag  for  the  next
packet

    //load up the cicular buffers
    loadBuffer();

      //wait for the ISR to change the array number
    while(1){
      if(prevGlobalX!=globalX) break;
    }

      //update the previous value for next time we compare
    prevGlobalX=globalX;

      //send an ack back to the computer
    sendReady();
   }

  PT_END(&s.pt);
}


void udpapp_appcall(void)
{
  handle_connection();
}
```

ACKNOWLEDGEMENTS

We wish to thank the following people and groups. Without them this project would not have been possible.

Adam Dunkels for designing and writing the UIP TCP/IP stack for embedded environments

Arduino Playground for support on Arduino libraries and documentation

AsyncLabs for providing documentation on the RedBack board

Bruce Land for help with project design and debugging, and accommodating us in his lab