Course Project Report

Bluetooth Speaker

Clayton Ramstedt & Lehi Alcantara

April 11, 2019

Introduction

For this project, we focused on building and implementing a Bluetooth speaker using a microcontroller and creating our own Bluetooth protocol to interact with it. In this paper, we will go over our experiments trying to send audio files from a computer to a microcontroller.

Related Work

There were several tutorials for how to wire up a microcontroller to a speaker that we utilized for building the hardware of our speaker, as well as tutorials for how to establish a serial connection over Bluetooth with a microcontroller. We also found a very sophisticated GitHub project that used the same hardware we were using that implemented the A2DP and AVRCP profiles that are part of the Bluetooth specification.

In general, the A2DP and AVRCP profiles do everything and more than what our protocol is currently capable of, and if you use them, any Bluetooth device that you connect to will automatically be able to detect what purpose the Bluetooth device is designed for, however, we decided that designing our own protocol from scratch would be a much more instructive, albeit less flexible. As an added benefit, we found this to be simpler to do based on the aforementioned GitHub project, which made use of a real-time operating system, an in-depth understanding of the hardware implementation of the Bluetooth stack on the ESP32 and the flexibility to work with multiple audio codec formats.

As an interesting side note, the A2DP and AVRCP profiles both make use of a underlying communication protocol called L2CAP, which is the same protocol we rely upon as part of our Bluetooth serial connection.

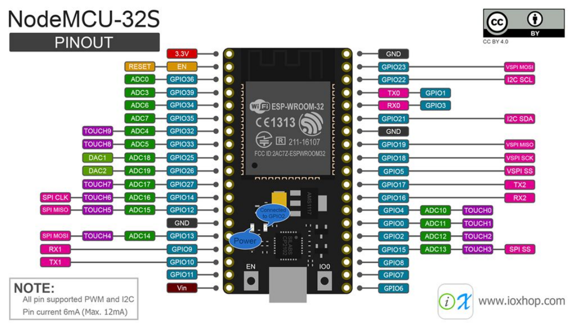
Methodology

In order to create our own Bluetooth speaker, we research several microcontrollers on the market that would be convenient for this application. Since we had had some previous experience with the ESP8266, we looked at microcontrollers in the same family, which eventually lead us to the ESP32. The ESP32 has the same features of the ESP8266 plus the hardware necessary for classic and low energy Bluetooth. On top of that, it came with an 8-bit DAC built in, which saved us another part to buy, and it is very conveniently programmable by the Arduino IDE.

We then proceeded to acquire a speaker and buttons to map play/pause, next and previous song functions. Finally, we found and downloaded a bunch of movie quotes from the internet to use for testing purposes.

Implementation and Experimentation

After assembling the necessary hardware for the microcontroller, speaker and buttons on a breadboard, we wanted to start with something simple such as: Can we play an audio file that is being sent over a USB serial connection on the ESP32?



This figure shows the ESP32 inputs available for us to interact with.

The first problem we encountered was that Windows/Mac could not see the ESP32 microcontroller because they lacked the correct drivers. After researching how to install the drivers and make them available to the Arduino IDE (which was much more of an ordeal for Windows than Unix based systems), we began developing a simple test for the speaker.

Choosing the correct format of the audio file was an important consideration for the design of our protocol. We didn't want to have to deal with compression, so the .wav became our first choice. From there we needed to make sure that the audio had been recorded by an 8-bit ADC so that we wouldn't have any problems playing it through our 8-bit DAC. Using a python script, we were able to analyze the sound files and by comparing the sampling rate and how many bits were in the file against the play time of the file, we were able to determine the bit length of the audio files.

Finally, we opted for files with a low sampling rate of 11 KHz in order to minimize the amount of data we would have to transfer, with the hope of making it easier to circumvent any timing constraints. Eventually, we resorted to getting all of our audio files from an online repository of movie quotes audio clips, as all of the clips seemed to fulfill our criteria.

When we started sending the audio file from a python script to the ESP32 via USB, we experienced some issues calculating the timing of the sampling rate that the ESP32 needed to follow in order to play the audio back recognizably. This was complicated by the unknown timing variables of the serial USB connection, which forced us to use trial and error to find the correct pace to space out playing the bits on the DAC (about 90 microseconds). This method had to be repeated when we started sending audio over a Bluetooth connection.

Now that we had our speaker working, we noticed that it was very quiet and we did not have a way to control the volume of the speaker. While we did initially intend to include volume control buttons, after looking at what the hardware requirements would be to digitally control the volume and to amplify the DAC signal to higher levels, we decided to stick to something that would be easier to breadboard.

Once we were able to send the audio file via USB serial, the next milestone was to send the same audio file to the ESP32 via Bluetooth. At this point, we start experimenting with various Bluetooth tools that were available to us, as well as researching how regular Bluetooth speakers function.

The next challenge came from how to get python to control Bluetooth on our computers. Surprisingly there are not a ton of packages that provide strong Bluetooth support. Specifically, the PyBluez library was frequently recommended as it makes use of a simple socket interface, however, we found it to be very difficult to get working on a Windows computer. As it turns out, Bluez is the name of the Bluetooth stack that is used for Linux based systems and it works best on Linux running computers. This introduced a new constraint of having our host device be a Linux running computer.

Thankfully for the ESP32 there already exists a library for setting up a Bluetooth serial connection that uses the same interface as the USB serial. However, the byte buffer for this library was hardcoded at 512 bytes. Because of the wait time that we have to undergo in order to play the audio correctly, we tended to fill up the buffer and ended up losing most of the packets. This became a major design constraint for our protocol.

Initially, our protocol simply read all 512 bytes in its buffer and then sent a *request more audio data* message requesting for another 512 bytes to be sent. However, this created a delay in the audio every 512 bytes, which gave the audio a choppy sound to it. To fix this, we had to add a series of assumptions to the *request more audio dat*a message:

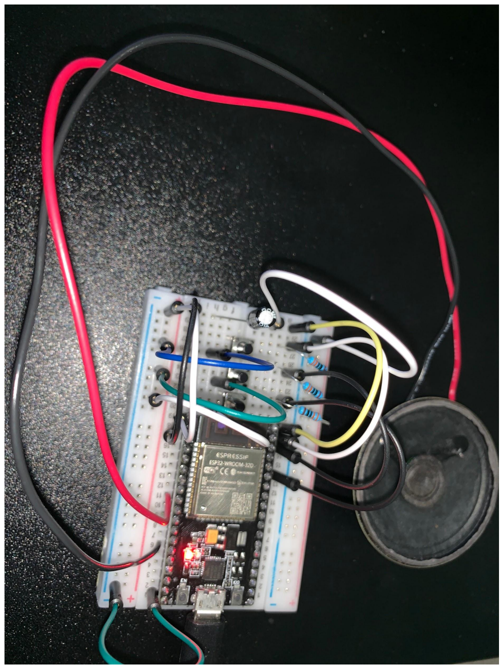
- If the ESP32 is going from a state of having no data in its buffer to having data in its buffer, it is assumed that the buffer how has 512 bytes in it.
- The ESP32 keeps count of how many bytes it processes and after processing 256 bytes, it will send the *request more audio data* message to the host.
- The host upon starting to play a new audio file will assume that the buffer of the ESP32 is empty and send 512 bytes of data initially and 256 bytes every time it receives a *request more audio data* message.

Once we got the audio streaming working well, we then augmented our protocol with commands that could be sent from the ESP32 to the host. On the ESP32 side, we have three buttons that when pressed send a single byte ASCII character to the host. Depending on the ASCII character, this would determine what button had been pressed on the ESP32. The button related part of the protocol is below:

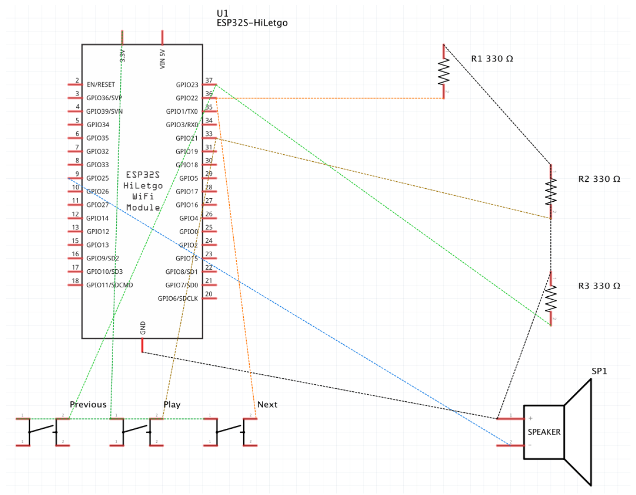
- Play/pause
    - Uses ‘p’ character
    - Toggles between music playing and not playing.
- Next song
    - Uses ‘n’ character
    - Jumps to next song.
- Previous song
    - Uses ‘q’ character
    - Jumps to previous song.

Results

In the end, we had our own simple, functional Bluetooth speaker that only works with our own custom driver as pictured below:



The physical speaker



Schematic of the circuitry

After everything was said and done, we noted that the audio quality wasn't the greatest, and so we experimented with what the theoretical limit is to the sampling rate that the speaker could perform at. The ESP32 itself is able to implement delays on the order of a few microseconds, which gives it a theoretical ability to play audio data back at a rate in the +100 KHz range, which is higher than what most audio codec support.

This leaves the speed of data transfer from the Bluetooth connection in question. To test this, we set up a feedback system that counted how many times the ESP32 was able to receive 512 bytes, process them, and send a message back to the host asking for more inside of one second. Below are the results that we found:

- Processing time per 512 bytes: **0.017 seconds**
- Approximate time per byte: **34 us**
- Max hz: **29418**

Realistically the max value is going to be less because the ESP32 will make use of a delay time to keep the audio playback rate consistent. Also, this does not take into account any time spent processing command characters.

Conclusion

In conclusion, we succeeded in building our own Bluetooth speaker and in creating our own Bluetooth protocol. There was a lot of trial and error, learning and satisfaction once it was up and working.

While this is a solved problem, jumping straight into the Bluetooth protocol was a daunting task, and our simple solution gave us an excellent opportunity to experiment with Bluetooth and the design challenges of creating new protocols.

In hindsight, if we were to go back and redo the project in a way that took advantage of the already existing audio profiles, we would focus on writing our own software for the ESP32, and rely on other people's work from, say, the Bluetooth audio capability of our cell phones as a drop-in replacement for the host. On the ESP32 especially, we were limited by the code for the Bluetooth Serial module that someone else had written and starting from scratch would be the only way to get things going as intended.

Future Work

A major limitation to our protocol is how limited it's use case is. Everything is built around the assumption that the buffer size is 512 bytes, the audio files that are being sent were sampled at 11 KHz and that the audio format is .wav.

To remedy all of these problems, we would need to change how the data is being sent to the ESP32. Instead of just all audio data, there would need to also be command characters. For example, one character could be 'the next 256 bytes are audio data'. Another could be 'What is your buffer size?' or 'the audio data is in x format'.

The biggest challenge with this is that command characters could also be interpreted as audio data. For this to work well, the protocol would have to have some sort of dropped packet tolerance to keep the system from accidentally playing a command character as sound data.

Another limitation is the sampling rate. There needs to be a system put into place that compresses the sound data so that it can be used at a higher sampling rate than the max transfer speed of the Bluetooth connection allows for with uncompressed data. Also any additions to the protocol, specifically if it adds command characters to the data that is being sent to the ESP32, is going to negatively impact how much audio data can be sent at one time, and thus limit the sampling rate. Ideally, additions to this part of the protocol should minimize how much data they use.

References

ESP32 Board Software:
https://dl.espressif.com/dl/package_esp32_index.json

ESP32 driver:
https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

Great reference for python receiving data:
https://pymotw.com/2/socket/tcp.html

Great reference for arduino bluetooth writing to BT Serial and regular Serial:
https://github.com/espressif/arduino-esp32/blob/master/libraries/BluetoothSerial/examples/SerialToSerialBT/SerialToSerialBT.ino

Internet radio bluetooth speaker using ESP32:
https://github.com/MrBuddyCasino/ESP32_MP3_Decoder

How to use serial bluetooth on ESP32:
https://circuitdigest.com/microcontroller-projects/using-classic-bluetooth-in-esp32-and-toogle-an-led