Bluetooth Range Extender for Audio

Yeab Wondimu

Advisor: Professor Erik Cheever

May 9, 2014
# Table of Contents

**Abstract** ........................................................................................................................................................ 3

**Background Information** .............................................................................................................................. 4
  - Project Description ........................................................................................................................................ 4
  - Bluetooth Technology ............................................................................................................................... 5

**Theory** ........................................................................................................................................................... 8
  - Range Extension Concept .......................................................................................................................... 8
  - Software Dependencies and Interaction ..................................................................................................... 9
  - Signal Processing ..................................................................................................................................... 10
  - Proof of Concept Bluetooth Scanner Program .......................................................................................... 13
  - Main Script .............................................................................................................................................. 16

**Results** .......................................................................................................................................................... 20
  - Testing Procedure ................................................................................................................................... 20

**Discussion** ................................................................................................................................................... 23

**Appendices** ................................................................................................................................................. 26
Abstract:

One of the common uses of Bluetooth is to allow for the playback of audio from a phone or other music device out to Bluetooth speakers. However, for most Bluetooth devices, the range of this connection is limited to about 10m. If one would like to make this Bluetooth connection indoors, as is usually the case, then the presence of walls in between the Bluetooth devices further limits the range of the connection. Using a Raspberry Pi microcontroller, along with some Linux packages that facilitate the connection of Bluetooth devices to the microcontroller, the range of this Bluetooth audio connection can be extended. The result of this idea is a C script run on the microcontroller that allows both the Bluetooth source device and the Bluetooth output device to connect to the Raspberry Pi and play audio. Essentially, the Raspberry Pi relays the Bluetooth audio signal from the phone to the Bluetooth speakers. Several key APIs were involved in the process: the PulseAudio¹ Simple API allowed for access and manipulation of the audio stream, the BlueZ² package was the standard Bluetooth protocol for the Linux operating system and enabled the user to open up connections to the Bluetooth devices from the microcontroller, and the Blueman³ API provided a simple graphical user interface to pair the Bluetooth devices to the RPi using function calls from the BlueZ package. The maximum range for the connection between the phone and the

---

¹ PulseAudio Version 5.0 used for this project
² BlueZ Version 5.18 used for this project
³ Blueman Version 1.0 used for this project
Bluetooth speakers without the addition of the Raspberry Pi is 11.2 meters. The maximum range with the Raspberry Pi facilitating the connection is 19.7 meters.

**Background Information:**

**Project Description:**

Nowadays, the use of Bluetooth devices to wirelessly stream audio out to speakers is commonplace. However, the range of this wireless connection is limited. My project uses a Raspberry Pi to facilitate and extend the range of this Bluetooth connection. The following two diagrams show how I propose to do this:

![Figure 1: A diagram of a standard Bluetooth connection](image-url)
Figure 2: A diagram of the addition of the Raspberry Pi to the connection and the increased range of the connection

As Figure 2 above shows, the Raspberry Pi will be inserted strategically in the middle of the connection in a spot to maximize the total range of the Bluetooth audio connection.

**Bluetooth technology:**

Bluetooth technology is used to transmit data wirelessly over short distances. Bluetooth works by transmitting data by means of low-power, short wavelength radio waves. Specifically, Bluetooth signals transmit at roughly 2.45 GHz, which has been standardized and set aside in the Industrial, Scientific, and Medical (ISM) 2.4 GHz short-range radio frequency band. The signals sent out by Bluetooth devices are very weak (about 1 Milliwatt), which is one of the ways Bluetooth devices avoid interfering...
with the radio waves of other devices. Bluetooth devices are separated into three classes: Class 1, 2 and 3. Most Bluetooth devices are class 2 devices. Consequently, the range of most devices is limited to about 32 feet or 10 meters and a maximum power output of 2.5 MW. Class 1 devices on the other hand have a maximum range of about 100m and a max power output of 100 MW. Very few devices are Class 1 devices however.

Since Bluetooth allows for up to 8 devices to be connected at once, one would think that the various radio signals would interfere with each other. However, a technique called spread-spectrum frequency hopping reduces the probability of this occurring to a small enough number that it is negligible. This technique requires each device to choose 79 randomly chosen frequencies with a bandwidth of 1MHz (called channels) within a designated range (within the small margin of frequencies that correspond to Bluetooth) and change between these channels on a regular basis. These channels run from 2402 MHz up to 2480 MHz. Bluetooth devices commonly change channels about 1600 times per second so the chances of two devices operating on the same frequency at the same time become infinitesimally small. This is due to the fact that because there are so many time frames within which a device could be operating on a certain channel and there are 79 different channels, the chance that there would be two different devices on operating on the same channel in the same timeframe is much smaller; especially given the frequency with which devices switch the channel they are operating on. When data is sent over a Bluetooth connection, the data is separated into different packets and each of these packets is sent over one of the 79 Bluetooth channels. Another added bonus of this technique is that other devices that
run in the same range of frequencies (portable phones, baby monitors) will not interfere with a Bluetooth device as the device will only be on a given frequency for a small fraction of a second.

When Bluetooth devices communicate with each other, they do so by setting up what is called a piconet. Each Bluetooth device is programmed by the manufacturer with an address that the manufacturer has established for a type of device. When a Bluetooth device is turned on, it sends out radio signals that are looking for a response from other devices within the same range of addresses. When a device within the range responds and sends signals back to the master device, a tiny network called a piconet is formed. Other devices which are not in the network are no longer able to communicate with the devices. The devices within the network start communicating with each other by transferring data packets, and each piconet is synchronously hopping randomly through the available channels. This insures that piconets are separate from each other and no interference occurs between piconets. In rare cases, a connection of two or more piconets is possible. In these cases, a device can serve as the master device (the one sending out signals to the slave devices) in one piconet while also serving as a slave in another. The figure below is a diagram of this setup, with the Raspberry Pi serving as both the slave in Piconet 1 and the master in Piconet 2.
Figure 3: A diagram of a simple connection between two piconets

Several different protocols are used to categorize the type of data transmitted by Bluetooth devices. The profile that is specifically associated with the transmission of audio data is called the Advanced Audio Distribution Profile (A2DP), and it falls under the Service Discovery Protocol (SDP). SDP allows devices to discover services (such as A2DP) that are offered by other connected devices and what parameters are needed to transmit data using these services. The A2DP profile defines how audio data is transmitted between Bluetooth devices. It is designed to transmit audio data in a 2-channel stereo audio stream and supports most audio file formats.

Theory:

Range Extension Concept:
While it is impossible to increase the range of Bluetooth devices due to technical limitations, the concept behind my project is to insert a middleman in this Bluetooth connection. Consequently both the Bluetooth audio source and the audio sink would directly connect to the intermediary device. Since both Bluetooth connections have a maximum range of about ten meters, the range of the overall connection can theoretically be doubled.

**Software Dependencies and Interaction:**

![Diagram of software dependencies and interaction](image)

Figure 4: A diagram of the various APIs and software dependencies needed to make my project functional.

The above diagram represents the handshaking that occurred between the various APIs that my project used and the steps needed to create the audio connection between the Bluetooth phone, the Raspberry Pi and the Bluetooth speakers. First of all,
a key concept of this process is that the BlueZ Linux API is the backend API that does all of the background Bluetooth connections in Linux. The Simple API for Pulse Audio as well as the Blueman API simply write code that utilizes the backend Bluetooth functionality that BlueZ provides. In the diagram above, the first step in the connection process is to pair the devices to the Raspberry Pi. The Blueman API is a graphical user interface that allows the user to select the devices to pair to the Raspberry Pi and then does the pairing for the user. The next step is to read the audio data from the Bluetooth phone which occurs using the Simple API. Further signal processing on the data is also done using function calls from the Simple API. Finally, the newly processed data is written out to the Bluetooth speakers. This final step is also done using the Simple API, and once this is completed, audio begins playing from the Bluetooth speakers.

Signal Processing:

Once the audio data was stored in the buffer, then some signal processing was done. For my project, I implemented three types of signal processing: a bass boost, a treble boost, and a balance for the left and right channels of the speakers. To implement the bass boost, each word (2 bytes) in the audio buffer was replaced with the average of it and the previous three words. This process was repeated for each of the 2048 words in the audio buffer. However, at the end of the buffer, the last three words were stored so that the average for the first word in the next set of data for the audio buffer could be calculated. If this step was skipped, the discrepancy of the first audio words in the buffer resulting in a skipping sound when played out.
The above figure shows a simplified version of this bass boost process. Rather than showing four words being averaged together, the figure only shows the average of two. However, the concept remains the same. After the audio buffer was modified, it was stored in another array of signed integers that represented an audio buffer with the new modified audio signal. The implementation of treble boost was very similar, with the only modification being that instead of replacing the word in the new audio buffer with the average of itself and the previous three words, you instead calculate this average and subtract it from the original word. The resulting signed 16 bit integer is what is placed in the new audio buffer. This simple modification changes our bass boost to a treble boost.

To create a balancing effect where the user could modify what percentage of the sound went to the left and right channels of the speakers, we used some of the built in functions in the pulse audio API. Using the `pa_channel_map_init_stereo` function, a
stereo channel map is initialized. The `pa_cvolume_set_balance` function is used to set the volumes on each of the channels in order to achieve the specified left, right balance that the user would like. However, before using this function, we must initialize a `pa_cvolume` object that contains the number of channels and the specific volumes for each channel. This object is initialized with two channels, corresponding to our desired stereo audio and the volumes for our initial channels, normally set to a constant `PA_VOLUME_NORM` which corresponds 100% of the normal device volume. After this object is created, the `pa_cvolume_set_balance` function is called with the `pa_cvolume` object, the channel map we have created, and a value between negative one and one that is used to calculate the new volumes for each channel. If the value of this number is negative one, it corresponds to the left channel being at full volume while the right channel is muted, and vice versa. Any number in between corresponds to a proportionate setting of the volumes between the left and right channels. The following table describes the arguments of the `pa_cvolume_set_balance` function and their data types.

<table>
<thead>
<tr>
<th><strong>Pa_cvolume_set_balance</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set the balance of the left and right channels</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td><strong>Data Type</strong></td>
</tr>
<tr>
<td>Structure representing channels and their volumes</td>
<td>Pa_cvolume</td>
</tr>
<tr>
<td>Channel mapping</td>
<td>Pa_channel_map</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Value between -1.0 and 1.0 to calculate new volumes</td>
<td>Float</td>
</tr>
</tbody>
</table>

Table 1: shows a description of the pa_volume_set_balance function, its arguments and their data types

**Proof of Concept Bluetooth Scanner Program:**

Audio data transferred using the A2DP protocol is represented as any normal audio signal would be, expressing the audio waveform as a sequence of binary numbers, normally separated out into bytes. The crux of my Bluetooth Range extender project is a script that opens up a connection to a Bluetooth phone which sends the audio signal over Bluetooth data packets. These packets are represented as a continuous stream of binary sent over Bluetooth to the Raspberry Pi. The script then creates a recording stream using the PulseAudio Simple API (Application Programming Interface) which writes the audio data into a buffer of signed 16 bit integers, which is essentially an array of 2048 signed 16 bit integers. As long as the audio data was being sent by the A2DP source, the audio buffer was being replaced with the new data. This audio buffer was passed into a PulseAudio playback stream that was played back on our A2DP sink device, the Bluetooth speakers. If no signal processing is done on the audio, no modification needs to be done on the audio buffer and the original audio buffer can simply be passed to the playback stream and played out to the speakers.
Setting up and manipulating a Bluetooth connection on the Raspberry Pi involved multiple Linux APIs. One of the key APIs is the BlueZ API. As the only open source Bluetooth protocol for Linux, it has become the official Bluetooth protocol on the Linux operating system. Since the BlueZ website did not have much in the way of documentation, simply posting the source code for the API, most of the initial work done familiarizing myself with the BlueZ API was done using the book Bluetooth Essentials for Programmers as well as an online article called Bluetooth for Programmers. After downloading the Linux BlueZ package onto the Raspberry Pi, I used these sources to write a proof of concept C script that simply opened up a connection to a Bluetooth device, checked to see that the connection was successful and that the script successfully extracted the MAC address of the Bluetooth device, and printed out the user-friendly name of the Bluetooth device on the screen. While this script was not used in the project itself, it was an effective testing mechanism in several ways. Firstly, in attempting to get positive results from the test, I found that the installation of several other packages related to the BlueZ package were needed. The additional package that was installed for this test specifically was the “bluez-tools” package. This package added additional command line tools for the BlueZ API as well as additional API function calls that were required to make the script functional. This test also unearthed another issue with the Bluetooth configuration of the Raspberry Pi. Previously in the report, it was explained that certain Bluetooth devices serve as A2DP sources and some serve as A2DP sinks. As the Raspberry Pi is both receiving incoming data packets (from our phone) and transmitting data packets (out to the Bluetooth speakers), it must serve as both an A2DP source as well as an A2DP sink. The default setting for
the Raspberry Pi is to serve only as an A2DP source. Consequently, opening up a Bluetooth A2DP connection and receiving data packets from our Bluetooth device requires that the Raspberry Pi serve as an A2DP sink as well. Fortunately, a simple modification to an audio configuration file allows us to make this change. This modification occurs in line 8 in Appendix I, which shows the audio.conf file. Before the modification, only “source” was enabled as a Bluetooth configuration for the RPi. Adding “sink” and “socket” allow us to use the RPi to receive A2DP (audio data) packets and the “socket” keyword allows us to open a socket to the Bluetooth device to send and receive data using other Bluetooth protocols.

Appendix II shows the C script that performs this scan and one can see that several API specific calls are made in the program. Lines 67 through 69 open up a socket to the Bluetooth device. The function `hci_get_route` finds the first available Bluetooth adapter and returns its resource number as an integer. In almost all cases, there is only one Bluetooth adapter so using `NULL` as an argument to the function finds the first available Bluetooth device. The structure used to specify a Bluetooth device address is a `bdaddr_t`, which is simply an array of 6 bytes. If a valid Bluetooth device address is passed to the function, it will find the first valid Bluetooth adapter that does not match the address given. The `hci_open_dev` function, when passed the resource ID of the Bluetooth adapter, opens up a socket to the microcontroller on the local Bluetooth adapter. However, this is not equivalent to opening up a socket to the Bluetooth device. The `hci_inquiry` function scans for available Bluetooth devices and returns some basic information about the Bluetooth them into the `inquiry_info` structure. There several pieces of information stored in the structure; however, the only one we are interested in
is the address of the detected device and in some occasions, the type of the device (whether it is a phone, computer, etc.). Once the basic information about the Bluetooth devices has been gathered, the _hci_read_remote_name_ function gets the user-friendly name of the device. It is passed the socket that was previously opened, the address of the device in question and a string for the name value. If successful, it returns a zero value and writes the name of the device to the string that was passed in. There are several lines of code that correspond the error handling in the situation that one of the steps fails and finally, the script closes the socket and returns.

**Main Script:**

The meat of my project was the “getdata.c” program which handled the task of access the A2DP audio data once the phone was paired to the Raspberry Pi, doing the audio data signal processing, and then sending the newly modified audio signal out to the Bluetooth speakers. Appendix III contains the code for this file, which begins by initializing several variables. The _pa_simple_ objects correspond to the PulseAudio structure that contains data about the audio stream such as whether it is recording audio or playing at back, the address of the Bluetooth device to either send or receive the data to, and a pointer to another object that contains the PulseAudio stream specifications. The specifications object, called a _pa_sample_spec_ object, contains data about the format of the audio data, the number of channels, and the sample rate. The first PulseAudio stream is then initialized with the _pa_simple_new_ function, and a variable telling the stream to behave as a recording stream, a parameter that passes in
the specific address of the phone as the device to receive the audio data from, and the initialized specification object.

<table>
<thead>
<tr>
<th><strong>Pa_simple_new</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create an audio stream (playback or recording)</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td>Server Name</td>
</tr>
<tr>
<td>Name for Client (application name)</td>
</tr>
<tr>
<td>Direction (playback or recording)</td>
</tr>
<tr>
<td>Sink or Source name</td>
</tr>
<tr>
<td>Stream Name</td>
</tr>
<tr>
<td>Sample type</td>
</tr>
<tr>
<td>Channel map</td>
</tr>
<tr>
<td>Buffering Attributes</td>
</tr>
<tr>
<td>Error</td>
</tr>
</tbody>
</table>

Table 2: Has a description of the arguments passed into the pa_simple_new function when creating a recording stream and their values.
This step occurs twice as we have to initialize both a recording stream to get the audio data from the input device (the phone) as well as a playback stream which sends the audio data to the output device (the Bluetooth speakers).

The next segment of my code, which is located inside a for loop, handles the signal processing of my project. The audio data is represented as a series of signed 16 bit integers placed in an array of size 2048. The for loop goes through each integer which represents two bytes of audio data and sets the value of the integer at the current index in the array to the average of itself and the previous three values, which results in a bass boost. However, adding all of the values before doing the division can result in an overflow, which means that the resulting integer value before the division could not be represented by 16 bits. The solution is to divide the integers before doing the addition. There are two lines of code that take the current array value and subtract this average from the current value. This operation corresponds to a treble boost. One thing to note is that the last three values from every audio buffer are saved so that they can be used to calculate the average for the first three values in the new audio array. This insures that you calculate the averages for the values at every index in the array. If this is not done, you hear a skipping sound that corresponds the averages not being calculated at the beginning of each array. The new values for each byte in the audio array are written to another corresponding audio buffer, named \textit{buf1}. Finally, this modified audio buffer is played out to the output device using the \textit{pa\_simple\_write} function.
Read audio data in from A2DP Source

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection stream (playback stream)</td>
<td>s1</td>
</tr>
<tr>
<td>Audio Buffer</td>
<td>buf1</td>
</tr>
<tr>
<td>Number of bytes to read</td>
<td>Sizeof(buf1)</td>
</tr>
<tr>
<td>Error Integer</td>
<td>err</td>
</tr>
</tbody>
</table>

Table 3: Shows a description of the arguments passed to the pa_simple_write function and their values

The arguments passed to the function are as follows: the playback stream that was initialized earlier in the code, the array with the audio data, the size of the array, and a variable that is written to if the playback fails. This process is encapsulated in an infinite for loop so that while the program continues to receive audio data from the phone, it continues to modify the data and write it out to the Bluetooth speakers until the script is terminated manually by the user.
Results:

The final product is a C program that opens up a socket to a Bluetooth device that acts as an A2DP audio source, and creates a recording stream. Bluetooth audio data from the phone is captured by this recording stream and sent to a playback stream which plays it out to the Bluetooth speakers. If no signal processing is done, what you hear is simply the unaltered music playing out to the Bluetooth speakers. If the code for the bass boost is active, what you hear is sound where the treble is muted compared to the bass which results in a bass boost effect. However, this effect is much harder to hear than the treble boost. When the lines of code that correspond to the treble boost were active, you could hear a significant decrease in the bass to the point where the treble greatly overpowered the bass. In effect, the treble boost sounds more like a bass reduction than it does boosting the treble but the final effect is the same. The ratio of treble to bass is much higher than without any signal processing done.

Testing Procedure:

After producing the code to do the range extension of the audio signal, the new connection range was measured. The process for doing this was as follows: first, a test of the connection range without the Raspberry Pi connecting to the devices was necessary. This was done by pairing the Bluetooth phone to the speakers, playing audio, and then separating the devices slowly until the audio connection started skipping out. This represented the fringes of the connection range and the distance when this skipping began to occur was the maximum connection range. To test the
Raspberry Pi, a similar method was used. Both devices were paired to the Raspberry Pi, and the C script was run so that audio was playing on the Bluetooth speakers to the Raspberry Pi. Then one device was moved away incrementally until the sound started skipping. Subsequently, the other device was also moved away until a skipping sound was observed. Then the distance between both devices (the phone and the speakers) was measured. The maximum range of the connection without the Raspberry Pi was measured to be 11.2 meters and the maximum connection range with the Raspberry Pi was measured to be 19.7 meters.

Another goal of my project was to test this range extension indoors and to observe if the range extension effect still occurred when there were walls in between. To test this, I used a standstill Bluetooth receiver that was connected to non-bluetooth speakers. The Bluetooth receiver was located in first floor of a three story building and was connected to speakers that were located on both the first and second floors. Initially, I connected my phone directly to the Bluetooth receiver and attempted to play audio to the speakers on the bottom floor. When the phone was also located on the bottom floor, the same floor as the Bluetooth speakers, audio could be played from anywhere in the room on the bottom floor. However, if the phone was moved upstairs to the second floor, the sound began skipping out as soon as it was moved from the spot directly above the Bluetooth receiver on the second floor. Consequently, if the user was one floor above the receiver and moved around the room, the audio would skip and sometimes lose connection altogether. The next step was to set up the Raspberry Pi on the second floor directly above where the Bluetooth receiver was located on the first floor. The Bluetooth receiver as well as the phone were connected directly to the
Raspberry Pi and audio was played through the Raspberry Pi out to the Bluetooth speakers. Once this connection was set up, audio could be played from both floors at any point on each floor. Even when the user was located a floor up from the Bluetooth receiver, audio could still be played from any spot in the room which was not the case with simply a phone to Bluetooth speaker connection. In fact, if the user moved up to the third floor, directly above where the Raspberry Pi was connected, audio could still be played out to the speakers even though the Bluetooth receiver was two floors below.

Finally, the testing of the signal processing aspects of my applications occurred. For the bass boost and treble boost, simply commenting in and out the relevant parts of the code and running my script in the lab with the Jam Classic speakers was sufficient. You were able to hear the bass boost and treble boost effects. However, testing the balance effect was more problematic. Since the Jam Classic speakers did not support stereo audio, setting the volume of the left and right channels did nothing to the audio that played on the Jam Classic speakers. So the testing of my balance function occurred on the same Bluetooth sound system used to test the indoor connection. I tested both a full left balance and a full right balance (meaning that all audio was played to the left channel only and then all audio played to the right channel only) and then tested the normal setting. The difference was easily heard. Since the system had speakers specifically set up for left and right audio, simply walking up to the left speakers during a full right balance and hearing that no audio was coming from them was confirmation that the balance function was working. Vice versa, you could listen to the sound output from the right speakers during a full left balance and hear no audio.
Discussion:

Finding a starting point for this project was challenging in several ways. While there were many similar projects done before, my research didn't find anyone who had done exactly what I was trying to do. There were several projects that implemented one half of the connection that I was to attempt to create; that is, several projects used a phone to connect to a Raspberry Pi but then played the audio out to speakers or headphones that were physically connected using the 3.5mm audio output jack. Conversely, there were several projects that loaded audio files onto the RPi and connected to the speakers using Bluetooth. However, none of these projects combined both of these Bluetooth connections as I was attempting to do and due to this, did not have to worry about both decoding the A2DP audio data as well as transmitting the A2DP signal to the Bluetooth speakers.

Many of the other difficulties that arose when working on the project stemmed for the various Linux packages associated with Bluetooth interacting with each other and with the audio packages used by Pulse Audio. Appendix III is a list of all the Linux packages that were downloaded to complete the project. Most of these packages interacted with each other in some way, and in some cases, these interactions resulted in some errors. An example of this is the bluezalso package which was originally created to simplify the interaction between the BlueZ package and ALSA, the default audio program for Linux. However, it seemed as if this package was out of date because when attempting to communicate with Bluetooth devices, I ran into several errors. The first of these errors showed up as “bt_audio_service_open: connect() failed: Connection refused (111).” Another error that also appeared was the “Cannot open
shared library /usr/lib/alsa-lib/libasound_module_pcm_bluetooth.so” error. The latter of these errors seems to suggest that there is a missing Bluetooth file in the Bluetooth library and after some searching, I found that several people had found the specific file needed online and manually placed it in the directory. However, this rarely seemed to produce results. After some more searching, I found that the real root of the error was not that there was an essential file that was missing, but rather that the most recent version of the BlueZ Bluetooth package for Linux was incompatible with the default audio player on Linux, ALSA. It followed that the bluez-alsa package would fix this compatibility issue; however, this package was also written for a previous version of BlueZ. The solution to this issue was to use PulseAudio instead of the default audio manager for Linux. It seemed that the newest version of the BlueZ package was compatible with PulseAudio.

Another issue that seemed to arise several times is that the Linux Bluetooth packages that were necessary to connect to the Bluetooth adapter as well as the Bluetooth devices needed some very specific parameters in some very specific configuration files modified. As there were usually many auxiliary files included in the directories for all of these packages, sorting through the source code and finding where to include the extra line or two of code that was needed took a great deal of time. Various web forums and source code documentation helped greatly with this process.

Finally, there is one thing to note about the main script that executes the range extension process. Currently, the MAC addresses of both the phone and the Bluetooth speakers are being hard coded into the relevant functions. What this means is that the user must know the MAC addresses of the devices that are to be used in the connection
beforehand. The script does not do any automatic discovery or pairing of devices on its own. An interesting extension of the project would be to combine the automatic pairing and retrieval of the device addresses that was completed as a proof of concept with the actual range extension part of the project. The complication in doing this arises in the fact that the address required to be passed into the `pa_simple_new` function is a specific string that comprises of the MAC address of the device as well additional characters that are meaningful to the BlueZ package. However, the MAC address retrieved in the initial C script does not contain the extra characters. Thus, the two addresses are unique and the value of one cannot be passed to the other. Given time, I would like to explore the BlueZ API further as there must be a way to scan for an available Bluetooth device, pair with it, and extract its MAC address and output it in a format that is compatible with the `pa_simple_new` function call. However, until this is done, this project is not market ready as a user must go through the tedious steps of pairing all Bluetooth devices manually and searching for each device's MAC address.
BlueZ audio.conf configuration file:

# Configuration file for the audio service

# This section contains options which are not specific to any
# particular interface
[General]
Enable=Source, Sink

# Switch to master role for incoming connections (defaults to true)
#Master=true

# If we want to disable support for specific services
# Defaults to supporting all implemented services
#Disable=Control, Source

# SCO routing. Either PCM or HCI (in which case audio is routed to/from ALSA)
# Defaults to HCI
#SCORouting=PCM

# Automatically connect both A2DP and HFP/HSP profiles for incoming
# connections. Some headsets that support both profiles will only connect the
# other one automatically so the default setting of true is usually a good
# idea.
#AutoConnect=true

# Headset interface specific options (i.e. options which affect how the audio
# service interacts with remote headset devices)
[Headset]

# Set to true to support HFP, false means only HSP is supported
# Defaults to true
HFP=true

# Maximum number of connected HSP/HFP devices per adapter. Defaults to 1
MaxConnected=1

# Set to true to enable use of fast connectable mode (faster page scanning)
# for HFP when incomming call starts. Default settings are restored after
# call is answered or rejected. Page scan interval is much shorter and page
# scan type changed to interlaced. Such allows faster connection initiated
# by a headset.
FastConnectable=false
# Just an example of potential config options for the other interfaces

[#A2DP]

#SBCSources=1

#MPEG12Sources=0
Simple Scan Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int main(int argc, char **argv) {
    inquiry_info *ii = NULL;
    int max_rsp, num_rsp;
    int dev_id, sock, len, flags;
    int i;
    char addr[19] = { 0 };
    char name[248] = { 0 };
    dev_id = hci_get_route(NULL);
    sock = hci_open_dev(dev_id);
    printf("DEV_ID: %d\n", dev_id);
    if (dev_id < 0 || sock < 0) {
        perror("Opening socket");
        exit(1);
    }
    len = 8;
    max_rsp = 255;
    flags = IREQ_CACHE_FLUSH;
    ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));
    num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
    if (num_rsp < 0) {
        perror("hci_inquiry");
    }
    for (i = 0; i < num_rsp; i++) {
        ba2str(&(ii+i)->bdaddr, addr);
        memset(name, 0, sizeof(name));
        if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name), name, 0) < 0) {
            strcpy(name, "[unknown]");
        }
    }
}
```
printf("%s %s\n", addr, name);
}
free(ii);
close(sock);
return 0;
}
Appendix III

Code for Main Bluetooth Transmitting Program:

```c
#include <config.h>

<stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <pulse/simple.h>
#include <pulse/error.h>
#include <pulse/volume.h>
#include <pulse/channelmap.h>

#define BUFSIZE 2048

int main(int argc, char* argv) {
    int check;
    printf("Input a 0 for regular audio and a 1 for treble boost\n");
    scanf("%d", &check);
    pa_simple *s;
    pa_sample_spec ss;
    pa_channel_map map;
    pa_channel_map_init_stereo(&map);
    pa_volume_t v1;
    pa_volume_t v2;
    v1 = PA_VOLUME_MUTED;
    v2 = PA_VOLUME_NORM;
    pa_cvolume volume;
    volume.channels = 2;
    volume.values[0] = v1;
    pa_cvolume_set_balance(&volume, &map, -1.0f);
    int ret = 1;
    int error;
    int error1;
    int firstTime;
```
for (;;) {
  int16_t buf[BUFSIZE];
  int16_t buf1[BUFSIZE];
  pa_simple_read(s, buf, sizeof(buf), &error);
  int i;
  int16_t previous1;
  int16_t previous2;
  int16_t previous3;
  if (firstTime == 0) {
    int16_t firstAvg;
    int16_t secondAvg;
    int16_t thirdAvg;
    firstAvg = (previous1 >> 2) + (previous2 >> 2) + (previous3 >> 2) + (buf[0] >> 2);
secondAvg = (previous2 >> 2) + (previous3 >> 2) + (buf[0] >> 2) + (buf[1] >> 2);
thirdAvg = (previous3 >> 2) + (buf[0] >> 2) + (buf[1] >> 2) + (buf[2] >> 2);
//buf1[0] = firstAvg;
//buf1[1] = secondAvg;
buf1[0] = buf[0] - firstAvg;
for (i = 3; i < BUFSIZE; i++) {
    int16_t temp;
    int16_t temp2;
    int16_t temp3;
    int16_t temp4;
    int16_t avg;
    temp = (buf[i-3]) >> 2;
    temp2 = (buf[i-2]) >> 2;
    temp3 = (buf[i-1]) >> 2;
    temp4 = (buf[i]) >> 2;
    avg = temp + temp2 + temp3 + temp4;
    //avg = avg >> 1;
    //buf1[i] = avg;
    buf1[i] = buf[i] - avg;
}
previous1 = buf[BUFSIZE-1];
previous2 = buf[BUFSIZE-2];
previous3 = buf[BUFSIZE-3];
firstTime = 0;
if (check == 0) {
    pa_simple_write(s1, buf, sizeof(buf), &error);
} else {
    pa_simple_write(s1, buf1, sizeof(buf1), &error);
}
pa_simple_free(s);
return 0;
Bluetooth for Programmers PDF:

3.1. Choosing a communication partner

A simple program that detects nearby Bluetooth devices is shown in Example 3-1. The program reserves system Bluetooth resources, scans for nearby Bluetooth devices, and then looks up the user friendly name for each detected device. A more detailed explanation of the data structures and functions used follows.

Example 3-1. simplescan.c

```c
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/socket.h> #include <bluetooth/bluetooth.h> #include <bluetooth/hci.h> #include <bluetooth/hci_lib.h>

int main(int argc, char **argv)
{
    inquiry_info *ii = NULL; int max_rsp, num_rsp; int dev_id, sock, len, flags; int i;
    char addr[19] = { 0 }; char name[248] = { 0 };

    dev_id = hci_get_route(NULL); sock = hci_open_dev(dev_id);
```
if (dev_id < 0 || sock < 0) { perror("opening socket"); exit(1); }

len = 8; max_rsp = 255; flags = IREQ_CACHE_FLUSH; ii = (inquiry_info*)malloc(max_rsp * sizeof(Inquiry_Info));

num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags); if (num_rsp < 0) perror("hci_inquiry");

for (i = 0; i < num_rsp; i++) { ba2str(&(ii+i)->bdaddr, addr); memset(name, 0, sizeof(name)); if (hci_read_remote_name(sock, & (ii+i)->bdaddr, sizeof(name), name, 0) < 0) strcpy(name, "[unknown]"); printf("%s %s\n", addr, name);
}

free(ii); close(sock); return 0;

### 3.1.1. Compiling the example

To compile our program, invoke gcc and link against libbluetooth

```sh
# gcc -o simplescan simplescan.c -lbluetooth
```

### 3.1.2. Representing Bluetooth addresses

```c
typedef struct { uint8_t b[6]; } __attribute__((packed)) bdaddr_t;

The basic data structure used to specify a Bluetooth device address is the bdaddr_t, which is simply a packed array
of six bytes. All Bluetooth addresses in BlueZ will be stored and manipulated as bdaddr_t structures. Two
convenience functions, str2ba and ba2str can be used to convert between strings and bdaddr_t structures.

```
str2ba takes a string of the form “XX:XX:XX:XXXXXX”, where each XX is a hexadecimal number specifying one byte of the 6-byte address, and packs it into a bdaddr_t. ba2str does exactly the opposite.

3.1.3. Choosing a local Bluetooth adapter

Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources. Usually, there is only one adapter or it doesn’t matter which one is used, so passing NULL to hci_get_route will retrieve the resource number of the first available Bluetooth adapter.

```c
int hci_get_route( bdaddr_t *addr );
```

This function actually returns the resource number of any adapter whose Bluetooth address does not match the one passed in as a parameter, so by passing in NULL the program essentially asks for any available adapter. If there are multiple Bluetooth adapters present, and we know which one we want, then we can use hci_devid.

```c
int hci_devid( const char *addr );
```

Unlike its counterpart, hci_devid returns the resource number of the Bluetooth adapter whose address matches the one passed in as a parameter. This is one of the few places where a BlueZ function uses a string representation to work with a Bluetooth address instead of a bdaddr_t structure.

Once the program has chosen which adapter to use in scanning for nearby devices, it must allocate resources to use that adapter. This can be done with the hci_open_dev function.

```c
int hci_open_dev( int dev_id );
```

To be more specific, this function opens a socket connection to the microcontroller on the specified local Bluetooth adapter. Keep in mind that this is not a connection to a remote Bluetooth device, and is used specifically for controlling the local adapter. Later on, in Section 3.5, we’ll see how to use this type of socket for more advanced Bluetooth operations, but for now we’ll just be using it for the device inquiry process. The result returned by hci_open_dev is a handle to the socket. On error, it returns -1 and sets errno.

**Note:** Although tempting, it is not a good idea to hard-code the device number 0, because that is not always the id of the first adapter. For example, if there were two adapters on the system and the first adapter (id 0) is disabled, then the first available adapter is the one with id 1.
3.1.4. Scanning for nearby devices

After choosing the local Bluetooth adapter to use and allocating system resources, the program is ready to scan for nearby Bluetooth devices. In the example, `hci_inquiry` performs a Bluetooth device discovery and returns a list of detected devices and some basic information about them in the variable `ii`.

```c
int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap, inquiry_info **ii, long flags);
```

Here, the function doesn’t actually use the socket opened in the previous step. Instead, `hci_inquiry` takes the resource number returned by `hci_get_route` (or `hci_devid`) as its first parameter. Most other functions we’ll see will use the socket opened by `hci_open_dev`, but this one creates its own internal socket. The inquiry lasts for at most `1.28 * len` seconds, and at most `max_rsp` devices will be returned in the output parameter `ii`, which must be large enough to accommodate `max_rsp` results. We suggest using a `max_rsp` of 255 for a standard 10.24 second inquiry.

If `flags` is set to `IRREQ_CACHE_FLUSH`, then the cache of previously detected devices is flushed before performing the current inquiry. Otherwise, if `flags` is set to 0, then the results of previous inquiries may be returned, even if the devices aren’t in range anymore.

The `inquiry_info` structure is defined as

```c
typedef __attribute__((packed))
struct {
  bdaddr_t  bdaddr;
  uint8_t   pscan_rsn_mode;
  uint8_t   pscan_period_mode;
  uint8_t   pscan_mode;
  uint8_t   dev_class[3];
  uint16_t  clock_offset;
} inquiry_info;
```

For the most part, only the first entry—the bdaddr field, which gives the address of the detected device—is of any use. Occasionally, there may be a use for the dev_class field, which gives information about the type of device detected (i.e. if it’s a printer, phone, desktop computer, etc.) and is described in the Bluetooth Assigned Numbers. The rest of the fields are used for low level communication, and are not useful for most purposes. If you’re interested, the Bluetooth specification has all the gory details.

3.1.5. Determining the user-friendly name of a nearby device

Once a list of nearby Bluetooth devices and their addresses has been found, the program determines the user-friendly names associated with those addresses and presents them to the user. The `hci_read_remote_name` function is used for this purpose.
Appendix V

Bluetooth Essentials for Programmers Excerpt:
reserves system Bluetooth resources, scans for nearby Blue-tooth devices, and then looks up the user-friendly name for each detected device. A more detailed explanation of the data structures and functions used follows.

### Compiling the Example

To compile our program, invoke `gcc` and link against `libbluetooth`:

```
# gcc -o simplescan simplescan.c -lbluetooth
```
Representing Bluetooth Addresses

    typedef struct {uint8b [6];
         } attribute((packed )) bdaddr;

The basic data structure used to specify a Bluetooth device address is a packed array of 6 bytes, and referred to as bdaddr t. All Bluetooth addresses in BlueZ will be stored and manipulated as bdaddr t structures. Two convenience functions, str2ba and ba2str, can be used to convert between strings and bdaddr t structures.

    int str2ba (const char *str , bdaddr *ba);
    int ba2str (const bdaddr *ba , char *str);

    The function str2ba takes a string of the form “XX:XX:XX:XX:XX:XX,” where each XX is a hexadecimal number specifying 1 byte of the 6-byte address, and packs it into a bdaddr t. The function ba2str does exactly the opposite.

Choosing and Opening a Local Bluetooth Adapter

Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources. Usually, there is only one adapter or it doesn’t matter which one is used, so passing NULL to hci get route will retrieve the resource number of the first available Bluetooth adapter:

    int hci_get_route (bdaddr*addr);

This function actually returns the resource number of any adapter whose Bluetooth address does not match the one passed in as a parameter, so by passing in NULL, the program essentially asks for any available adapter.

    If there are multiple Bluetooth adapters present, and we know which one we want, then we can use hci devid:

    int hci_devid (const char *addr);

Unlike its counterpart, hci devid returns the resource number of the Blue-tooth adapter whose address matches the one passed in as a parameter. This is one of the few places where a BlueZ function uses a string representation to work with a Bluetooth address instead of a bdaddr t structure.
Note: Some of the functions here use “dev” (short for “device”) in their names, while we’ve been using the term “adapter.” In general, we use device to refer to any machine capable of Bluetooth communication, while adapter refers specifically to the local (machine executing the program) Bluetooth device. BlueZ does not always make this distinction.

Once the program has chosen which adapter to use in scanning for nearby devices, it must allocate resources to use that adapter. This can be done with the hci open dev function:

```c
int hci open dev(int adapterid);
```

To be more specific, this function opens a socket connection to the microcontroller on the specified local Bluetooth adapter. Keep in mind that this is not a connection to a remote Bluetooth device, and is used specifically for controlling the local adapter. Later on, in Section 3.5, we’ll see how to use this type of socket for more advanced Bluetooth operations, but for now we’ll just be using it for the device inquiry process. The result returned by hci open dev is a handle to the socket. On error, it returns -1 and sets errno.

Note: Although tempting, it is not a good idea to hard-code the device number 0, because that is not always the ID of the first adapter. For example, if there were two adapters on the system and the first adapter (ID 0) is disabled, then the first available adapter is the one with ID 1.

Device Discovery

When an inquiry message is broadcast, nearby Bluetooth devices respond by sending their Bluetooth address. Of course, devices must be in the correct mode in order to respond, and even then they may not hear the inquiry or take too long to respond. We often say that our program “scans” for nearby devices, but it is actually scanning for devices that responded to the inquiry. A Bluetooth device discovery operation is initiated by the function hci inquiry, which issues the inquiry signal and returns a list of devices that respond along with some of their basic information placed in the variable devices that is passed as a parameter to the function:
int hci_inquiry(int adapterid , int len , int max_rsp ,
                 const uint8t *lap , inquiryinfo **devs , long flags);

A quick look at the parameters shows that the function does not actually use the socket opened in the
previous step. Rather, hci_inquiry takes the resource number returned by hci_get_route (or hci_devid) as its first parameter, and it creates its own internal socket. Nearly all other functions covered
in this chapter will use the socket opened by hci_open_dev.

The parameters control both the scanning and return information. The inquiry lasts for at most
1.28 * len seconds, and at most max_rsp responding devices will be returned in the output
parameter devs, which must be large enough to accommodate max_rsp results. We suggest using a
max_rsp of 255 for a standard 10.24-s inquiry. The devs parameter is an array of inquiry_info
structures:

typedef struct {bdaddr bdaddr;
                  *uint8scan_repmode;
                  *uint8pscan_periodmode;
                  *uint8pscanmode;
                  *uint8t devclass [3];
                  *uint16t clockoffset;
               } attribute((packed)) inquiryinfo;

The first entry, the bdaddr field, is the most useful and gives the address of the detected
device. Of the remaining fields, the dev_class field may also be of interest, as it conveys general
information about the type of device detected (if it’s a printer, phone, computer, etc.) and the services
offered (file transfer, audio, network access, etc.). The exact formatting of this field is described in the
“Assigned Numbers – Bluetooth Baseband” document, distributed on the bluetooth.org Web site. The
remaining fields are used for low-level communication, and are not usually useful. If you’re really
interested, all the gory details can be found in the Bluetooth specification.

The final parameter, flags, indicates whether or not to use previously discovered device
information or to start afresh. If it is set to IREQ_CACHE_FLUSH,
then the cache of previously detected devices is flushed before performing the current inquiry. If flags is set to 0, then the results of previous inquiries may be returned, even if the devices are no longer in range.

Name Lookup

Given the list of the addresses of nearby Bluetooth device, it is common practice to then determine their user-friendly names. The \texttt{hci read remote name} function is used for this purpose:

\begin{verbatim}
int hci read remote name(int hcisock , const bdaddr *addr ,
   int len , char *name , int timeout );
\end{verbatim}

This function attempts, for at most timeout milliseconds, to use the socket \texttt{hcisock} in order to query the device with Bluetooth address \texttt{addr} for its user-friendly name. On success, it returns 0 and the first \texttt{len} bytes of the device’s user-friendly name stored in the supplied character array: \texttt{name}.

The function \texttt{hci read remote name} only tries to resolve a single name, so a program will typically invoke it many times to get a list of all the user-friendly names of nearby Bluetooth devices.

Error Handling

So far, all the functions introduced return an integer on completion. If the function succeeds in doing whatever the program requested, then the return value is always 0. If the function fails, then the return value is -1 and the \texttt{errno} global variable is set to indicate the type of error. This is true of all the \texttt{hci} functions, as well as for almost all of the socket functions described in the next few sections. (The exceptions are \texttt{socket}, \texttt{send}, and \texttt{recv}.)

In the examples, we’ve left out error checking for clarity, but a robust program should examine the return value of each function call to check for potential failures. A simple way to incorporate error handling is to use the \texttt{strerror} function to print out what went wrong, and then exit. For example, consider the following snippet of code:

\begin{verbatim}
int devid = hci get route ( NULL );
if (devid < 0) {
  printf (stderr , "error code %d: %s\n ", errno , strerror (errno ));
  exit (1);
}
\end{verbatim}