4-27-2015

Refinement of a Novel Hearing Assistive Device Algorithim for Lowering Frequencies in Real Time

Nicholas Mycroft Christensen

Refinement of a Novel Hearing Assistive Device Algorithm for Lowering Frequencies in Real Time

by

Nicholas Mycroft Christensen

An Honors Thesis
submitted in partial fulfillment of the requirements
for the Honors Diploma
to

The Honors College

of

The University of Alabama in Huntsville

April 27, 2015

Honors Thesis Advisor: Dr. Richard Coleman
Computer Science Lecturer

_____
Student Date

_____
Advisor Date

_____
Department Chair Date

_____
Honors College Director Date

Table of Contents

Abstract

In 2009, working with electrical engineer John H. Jacobs, I designed an internationally award-winning algorithm that lowers voice frequencies in real time for a novel hearing assistive device. The prototype device included analog-to-digital and digital-to-analog converters, operating with a PIC18F8722 microprocessor and a customized circuit board. The algorithm, originally coded in Pascal for pre-testing and later in PIC assembly, omits sound wave cycles at adjustable intervals and expands the remaining cycles to fit into the original time domain, thereby lowering the frequency, improved speech recognition for people with hearing loss. The testing proved the device to be beneficial, especially for those with significant high-frequency hearing loss.

The objectives of this effort are to refine the algorithm, rewriting the code in C for more efficiency and portability, implement more precise and variable frequency adjustments, and try to test it in commercially available platforms. Additionally, if time permits, implementation of Bluetooth support will be explored. The end goal of this effort is to make the algorithm portable and thus commercially viable for use with various hearing devices.

Introduction

Prevalence of Hearing Loss

According to the National Institute on Deafness and Other Communication Disorders (NIDCD): "Approximately 15% of American adults (37.5 million) aged 18 and over report some trouble hearing" (April 20, 2015). The most common type of hearing loss is presbycusis, defined as high-frequency loss associated with aging (Gates and Mills, 2005). The hair cells within the human cochlea, called stereocilia, transmit different frequencies at different locations as shown in Figure 1 below. In fact, sounds within the 10 to 20 kHz range are picked up at the base of the cochlea, near the outer ear, where there are likely to be damaged hair cells that do not regenerate (Grau). If enough are damaged, no sounds will be heard in this range. Amplifying the sounds does not help. It is like turning the brightness of a light bulb up for a blind person; he still cannot see.

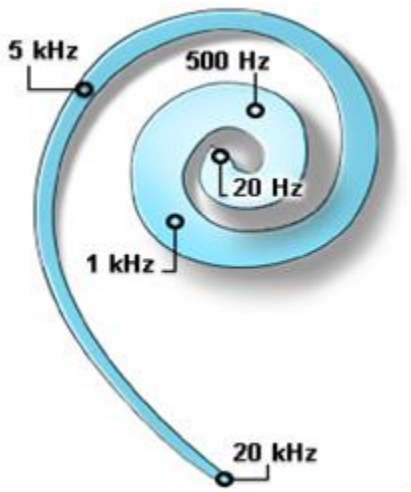

Figure 1. Guy Rebillard and Rémy Pujol. *Frequency distribution along the basilar membrane of a human cochlea: passive tonotopy*. Available from Journey into the world of Hearing, http://www.cochlea.eu/en/cochlea/function

Even partial hearing loss causes communication problems because many voiceless phonemes, such as s, th, ch, and sh are in the high-frequency range. Even worse, in English we use morphological endings such as –s for plural, possessive, and verb agreement or –ed to form past tense. These sounds are difficult for a hearing-impaired person to distinguish. Figure 2 below, a chart scanned during an interview with audiologist Dr. Jane Laseter, shows the phonemes that hearing loss tends to affect.

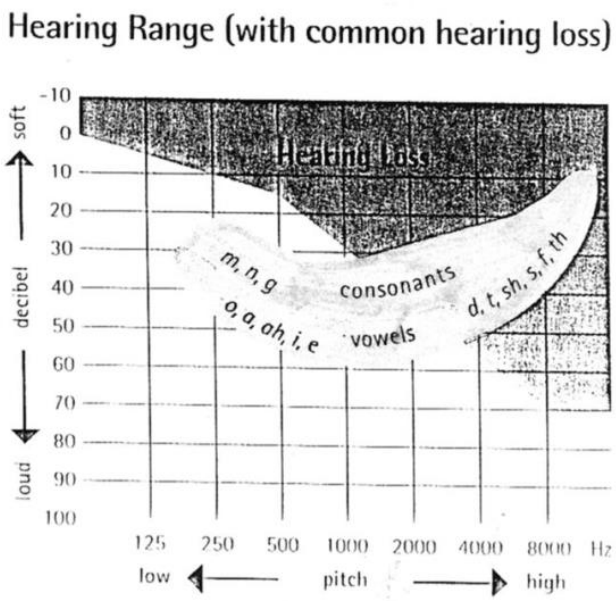

Figure 2. Dr. Jane Laseter. *Hearing Range (with common hearing loss)*. Scanned from personal interview on June 11, 2007.

Why Traditional Hearing Aids Do Not Work

A typical hearing aid has three parts: a microphone, an amplifier, and a speaker, which merely amplify sound waves. Digital hearing aids amplify by digitizing, converting the electrical signal into numbers for processing (Ricketts, accessed April 25, 2015). The most useful feature of digital hearing aids is their ability to be programmed to fit individual hearing loss because they can be adjusted to amplify a particular frequency range instead of amplifying all sounds equally. While this is helpful, it still does not address the entire problem. Many people have areas with dead stereocilia; no amount of amplification helps. What is needed is hearing assistive technology that lowers frequency, moving sounds to other areas of the cochlea that are not as damaged.

Brief Summary of Algorithm Development

For the first phase of my project, conducted in 2007-08, I created an algorithm to omit intermittent wave cycles of voice signals, and then stretch the wave by resampling to conform to the standard 44.1 kHz sampling rate of .wav files. This algorithm was implemented in Pascal. This lowered the frequency from 12.5% to 25% because when a wave is stretched out, the wavelength increases, and thus the frequency lowers as shown in Figure 3 on the next page. The re-digitized file became what humans perceive as lower pitch. Subjects were tested for word recognition when listening to the lowered words with voiceless phonemes; the results were encouraging, with 6% overall improvement and individual gains as high as 65%.

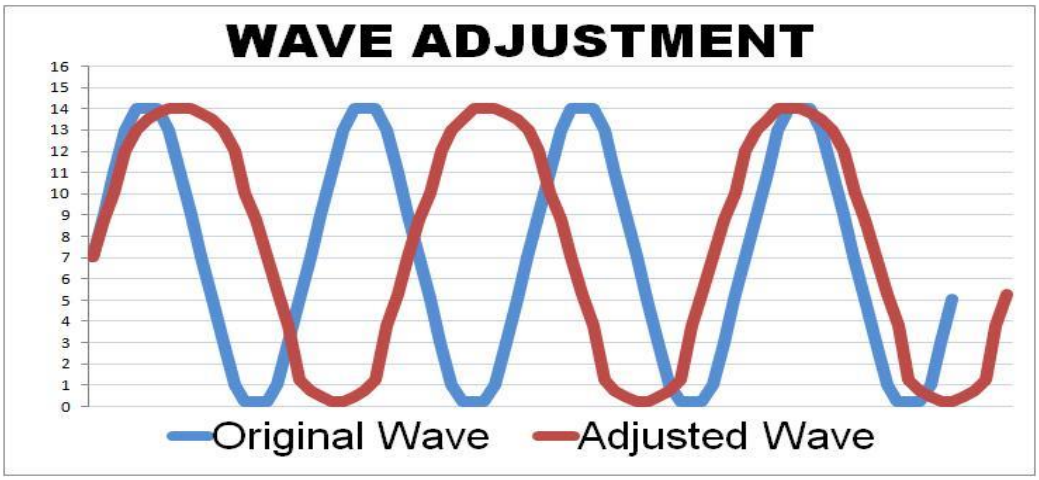

Figure 3. Nicholas Christensen. *Lowered frequency from increased wavelength.* Created with Microsoft Excel software.

The second phase of the project, conducted in 2008-09, was implementing the algorithm in assembly on a custom-built circuit board with an 8/12-bit PIC processor, specifically the PIC18F8722, to make it work in real time. The clocks were synchronized with the same percentage of time adjustment as of frequency adjustment, as the digital-to-analog converter (DAC) outputs the results. The input had to be sampled with an analog-to-digital converter (ADC) at a sample rate that is at least 40,000 samples per second because that is twice the maximum frequency for human hearing at 20 kHz. According to the Nyquist-Shannon sampling theorem, to fully reproduce a function of a specific frequency, it must be sampled at a minimum of twice that frequency, e.g. if a function has a frequency of 10 Hz, then a sample rate of 20 samples per second is required to fully reproduce that function with samples (Shannon, p. 10).

The algorithm distinguishes the cycles of waves by counting the number of zero crossings (every time a zero is crossed twice, a full cycle of the wave has been processed). The algorithm will omit cycles intermittently; to be precise, it omits samples intermittently, and

then outputs the non-omitted samples at a reduced time rate to the DAC which after being converted back to an analog sound wave is heard as a lower frequency.

Now, for the third phase of the project, the algorithm, which was proven to not only work, but be useful to those with high-frequency hearing loss, is rewritten in the C programming language. The benefit of C is that, unlike assembly, which is tied to specific processors, the language is portable between machines, such as Bluetooth.

Case for using Bluetooth

The reason Bluetooth is an excellent choice to adapt as a hearing assistive device is that there is no social stigma against Bluetooth devices, unlike hearing aids.

> People with hearing loss often have negative stereotypes and prejudices attributed to them that result in a deleterious effect on how they are perceived by others. People with hearing loss are often perceived as senile (Oyer & Oyer, 1985), uninteresting (Heine & Browning, 2002; Hétu, 1996) and/or undesirable communication partners (Jones, Farina, Hastorf, Miller, & Scott, 1984). Studies on the hearing aid effect have repeatedly shown that people are rated more negatively by others if they are seen wearing a hearing aid…. (Gagné, J. P., M.B. Jennings, and K. Southall, pg. 203)

Bluetooth technology, such as Bluetooth earbuds or headsets, is popular among younger people and has no stigma attached. That makes Bluetooth devices a prime candidate for implementing an algorithm that can help people with hearing loss be able to hear those high-frequency sounds that they would otherwise miss. Instead of being perceived negatively due to wearing normal hearing aids, which would not be helpful anyways, a person with a

Bluetooth device with the frequency-lowering algorithm would receive the benefit of a hearing assistive device and not be stigmatized for wearing an obvious hearing aid.

Purchasing Hardware and Setting up IDE

After researching various audio development boards, I settled on the PIC32 Bluetooth Audio Development Kit (part number DV320032). Because mentor John Jacobs is using a PIC32MX350F256L for a personal project and the development kit uses a PIC32MX470F512L, which is in the same processor family, the same timing code can be used for both processors. The only difference between the two is the pin mappings (the 470F512L has USB support while the 350F256L does not). This should simplify trying to work out how to get the timing code to work; when it works on one processor, it will work on the other.

Determining Compiler

Considering the compilers available for use, I determined that, for the purpose of this project, the free version of Microchip's XC32 compiler would be sufficient. The PRO version would make the compiled code smaller, but not faster, and it is not anticipated that the code will use up much space of the processor's 512 KB of program memory.

Configuring Include Files

Including the master header file for PIC32 peripherals (plib.h) proved to be a troublesome endeavor. Simply adding it to the project from its location in C:\Microchip\xc32\v1.33\pic32mx\include does not work; however, copying the header file directly into the project folder works just fine. Unlike other IDEs (such as Visual Studio), MPLAB X IDE does not automatically copy a source file from its original location into the project folder when that file is added to a project. This is fine, except that the header files included in plib.h cause an error message to be sent to the user about their own header files not being included, and if those header files are included, then the compiler balks about recursively adding header

files, even with standard #ifndef-#define-#endif preprocessor tags that should stop a header file from being included multiple times.

To fix this particular problem, the following steps were performed:

1) Go to Run -> Set Project Configuration -> Customize
2) Underneath Conf: [XC32_PIC32MX795F512L] (note: the device is set to PIC32MX470F512L) go to XC32 (Global Options) -> xc32-gcc
3) Go to Option categories: Preprocessing and messages
4) Add the following directories to the "Include directories":
    a. ..\Microchip\xc32\v1.33\pic32mx\include
    b. ..\Microchip\xc32\v1.33\pic32mx\include\machine
    c. ..\Microchip\xc32\v1.33\pic32mx\include\peripheral

This tells MPLAB to include those directories, which contain the header files that plib.h references, when using the plib.h file.

Connecting Debugger and Programming Processor

Before the development kit could be programmed, it had to be properly connected via the PICKit 3 debugger and configured within MPLAB X IDE.

The IDE was able to run a programming sequence, but the program did not actually seem to run properly on the development kit; it was supposed to flash LEDs in a predetermined sequence, but it did not have all the LED's because the program loaded was for a different board. Nor was the IDE able to run a debug mode as the debug configuration bits were not properly set. Erasing the program memory and reprogramming did not fix the issue, so the problem may lie in the code not working properly for this particular LED set.

In further testing and research, it turned out that the LED message sample code was for a different board, but the program was able to be compiled and programmed onto the development kit. The timers were clearly running because the LEDs were blinking in sync. The program memory was able to be erased and reprogrammed successfully, meaning that the goal of getting MPLAB X IDE, xc32 C compiler, and the development kit all up and running has been successfully achieved.

Calculations for Timers

One problem is that the audio development kit does not have a standard audio input. It can use Bluetooth audio in or USB audio in, so a Bluetooth microphone might work. Alternatively, an extra audio-in board could be installed via the I2C expansion port. Fortunately, the BT Audio Demo 9 is able to connect to a phone via Bluetooth and an audio stream from the phone could reach the DAC. The next major step was to get some source code in to intercept the stream and perform processing before it reaches the DAC.

The values to set the timers to can be calculated using the following formulas (PIC32 Tutorial 2 – Timers, accessed October 2014):

Period = 1 / frequency (sample rate)

Accuracy = 1 / f_input (the processor clock)

Resolution = XX bits*accuracy

In this case, the period of 1/40 kHz = 25 µs is constrained by the preferred sample rate for speech due to the Nyquist-Shannon sampling theorem stating that to achieve perfect reconstruction of an analog wave with samples is to have the sample rate be twice the highest frequency, and since human hearing maxes out at 20 kHz, the sample rate should be 40,000

samples per second. The processor clock is 80MHz, so accuracy is 1/80 MHz = 12.5 µs. Timer 1 is 16 bits, so XX = 2^16 or 65,536 while timers 2 & 3 and 4 & 5 can be combined to make 32-bit timers, so XX = 2^32 or 4,294,967,296.

Resolution refers to how long counter can increment using just the timer. Using a prescaler sacrifices accuracy to gain resolution (can count higher). A prescaler can be used to increase resolution at the expense of accuracy by dividing the input by the second number (for example, a 1:8 prescaler increments output by 1 for every 8 ticks of the input). In essence, it slows down the clock for the timer. The following pre-scale options are available to the PIC32MX470F512L: 1:1, 1:8, 1:64, and 1:256, according to the datasheet (pg. 180).

Because it is necessary to have a 40 kHz timer for the input on timer 1, the period should be 25 µs**.** A table of values is given below.

Pre-scale	**Accuracy**	**Resolution**
1:1	12.5 µ**s**	819.2 m**s**
1:8	100 µ**s** or 0.1 m**s**	6.554 s
1:64	800 µ**s** or 0.8 m**s**	52.43 s
1:256	3.2 m**s**	210 s

Table 1: Timer calculations based on 40 kHz input, not dividing processor frequency by four.

While not clear from my research, some of the PIC processor families, like the PIC32 family, may have the frequency divided by four for purposes of timer calculations. This would mean that the accuracy is

$$1/(80\text{MHz}/4) = 1/(20\text{MHz}) = 50\ \mu\text{s}$$

Assuming processor frequency is divided by four, the table of values for a 40 kHz timer would be

Pre-scale	**Accuracy**	**Resolution**
1:1	50 µs	3.277 s
1:8	400 µs or 0.4 ms	26.21 s
1:64	3.2 ms	210 s
1:256	12.8 ms	838.9 s

Table 2: Timer calculations based on 40 kHz input, dividing processor frequency by four.

The next major step was to figure out how to intercept the Bluetooth data stream. The files loaded onto the development kit from the Microchip website are .HEX files, which are precompiled and thus are not useful in trying to figure out how to implement a Bluetooth data stream. At this time, the source code was unavailable to see how they implemented the Bluetooth data stream.

Bluetooth Promiscuous Mode and Obtaining Disassembly from Memory Dump

Because the .HEX files were precompiled, giving no information except raw assembly in working memory, I then tried to research Bluetooth promiscuous mode on this Bluetooth Audio Development Kit. However, it is apparently not implemented in the hardware and thus, not available.

Next I used Microchip's contact form to contact someone in the company about my intended use of this software suite, with questions about the source code, but nobody responded.

In order to try to get the raw disassembly code from working memory, I followed the following instructions: "Use the IMPORT function of MPLAB found under the File menu to import your code as hex. You can then view the ASM listing in the Program Memory window" (Seriani, accessed October, 2014).

At that point, I found the documentation of MPLAB Assembler for PIC32 microcontrollers and started attempting to learn how to read the assembly code of a PIC32 processor.

The demo program "BT Audio Demo 9" was able to be successfully loaded onto the development kit; however, when the option *Window -> Output -> Disassembly Listing File* is selected to view the assembly code of the precompiled demo program, it generates the following error message: "Unable to generate the disassembly listing file. Please make sure that you have built the project with symbol information." Clearly the symbol information is missing from the precompiled demo program. A quick Google search on the error message turned up nothing useful.

The disassembly code was eventually able to be viewed in Execution Memory by selecting *Window -> PIC Memory Views -> Execution Memory* and expanding the window up to view the code after selecting Format: Code instead of Data. From here, the disassembly code was able to be exported to a well-formatted text file and then imported into a spreadsheet. When imported into a raw text file, the file was 11.8 MB in size and nearly impossible to read.

For BT Audio Demo 9, the program memory starts at line 32770 with the disassembly value being "Program Memory," and it ends at line 81433 with the disassembly value being "CACHE 8, 12384 (GP)". All opcodes after this line are set to FFFFFFFF whereas all opcodes before line 32770 are set to 00000000. That means that there are 48663 lines of assembly code.

According to the PIC32MX330/350/370/430/450/470 datasheet, the instruction set complies with the MIPS32 Release 2 instruction set architecture. It provided a link to download the manual for the instruction set titled "MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set" (pg. 265). Attempting to read the disassembly code and translating it to lines of C code proved to be impossible, so the next step was to purchase the source code directly.

Obtaining Source Code and Implementing Algorithm

PIC32 Bluetooth Break-In Mode + SBC Audio Software Suite 3 v3.5 is not available for immediate purchase, so it required contacting MCU32 marketing to be allowed to purchase it. A form was filled out to contact them; however, there was no reply for two weeks, so instead the PIC32 Bluetooth SBC Only Audio Software Suite 1 v3.5 *(*part number SW320014-1) was purchased, for $299, to use BT Audio Demo 7 (Bluetooth SBC + Graphics Equalizer) or BT Audio Demo 5 (Bluetooth Stack (A2DP + AVRCP + SPP + SBC decoder) + Graphics).

The following code snippets and source code files are contained in Bluetooth audio demo 5, “Bluetooth Audio with SBC.” The different signals that tell the processor to do a specific task are defined in bttask.h as follows:

```
typedef enum _BTTASK_SIGNALS
{
    BTTASK_SIG_RX              = 0x0001,
    BTTASK_SIG_TX              = 0x0002,
    BTTASK_SIG_BTSIGNAL        = 0x0004,
    BTTASK_SIG_TIMER           = 0x0008,
    BTTASK_SIG_WAKEUP          = 0x0010,
    BTTASK_SIG_BUTTONS         = 0x0020,
    BTTASK_SIG_BUTTON_REPEAT   = 0x0040,
    BTTASK_SIG_KEYBOARD        = 0x0080,
    BTTASK_SIG_STORAGE         = 0x0100,
    BTTASK_SIG_AUDIO           = 0x0200,
    BTTASK_SIG_ACP             = 0x0400,
```

```
        BTTASK_SIG_DISPLAY          = 0x0800,
        BTTASK_SIG_VOLUME           = 0x1000
    } BTTASK_SIGNALS;
```

In the **main.c** source file, there is the main() function, which is the central function for all C programs. In main(), the first instruction called is boardInit() to initialize the board. The next instruction, since USE_DISPLAY is defined, is to call the DISPLAY_MESSAGE_Initialize() function to initialize the display. Next up are the following initialization function calls which initialize various features of the board: storageInit(), bluetoothInit(), app_display_Initialize(), displayInit(), buttonsInit(), audioInit(), and volumeInit().

Once all features of the board are initialized, main() enters a while-true loop to execute the following tasks continuously until the board is shut down: bluetoothTask(), audioTask(), displayTask(), buttonsTask(), and volumeTask(). These task functions are responsible for everything from interpreting signals to changing states based on user input via buttons.

The first task is bluetoothTask(), which is defined in **bttask.c** and is responsible for interpreting various signals (wake up signal, signal process pending, audio signal handler, receive signal, transmit signal, signal timer, repeat button signal, storage signal, display signal, and volume signal). Within this function are various if-conditionals that compare the signal with a known signal defined in the _BTTASK_SIGNALS enumeration mentioned earlier.

The next task is audioTask(), which is defined in **audio.c** and is responsible for reading different kinds of messages. It has a switch-case on the message type if the processor has received a message from the Audio Message Queue (MSG_CONFIG_PLAYBACK, MSG_START_PLAYBACK, MSG_STOP_PLAYBACK, MSG_PAUSE_PLAYBACK,

MSG_RESUME_PLAYBACK, MSG_SET_VOLUME, MSG_PLAY_PACKET, and MSG_FILL_PLAY_BUFFER). While the names of these messages seem to indicate what they are for, what is actually happening in the code is unclear because there are very few explanatory comments.

The third task is displayTask(), which is defined in **display_tasking.c** and is responsible for managing how messages are displayed on the display using a switch case statement (MSG_OUTTEXT, MSG_IMAGE, MSG_BAR, MSG_BEVEL, MSG_FILLED_BEVEL, MSG_PROGRESS_BAR, MSG_THROTTLE_UP, MSG_THROTTLE_DOWN, and MSG_SCREEN_SET). However, if the throttle timer is in use and has not timed out, the function will simply return and not do anything. Only if the next message is available will the function update what is on the display. Like the messages in audioTask(), it is not clear what the code is actually doing.

The fourth task is buttonsTask(), which is defined in **buttons.c** and is responsible for changing states based on which button is pressed.

The fifth and final task is volumeTask(), which is defined in **audio_volume.c**. This task checks to see if a variable for mVolumeEvent is 1; if so, then set that variable to 0 and call the button_pollVolume() function. The button_pollVolume() function is defined in **audio_volume.c** and calls the bttask_setSignal() function with the BTTASK_SIG_VOLUME enumeration.

In the **bttask.c** source file, there is a function called bttask_setSignal() that takes in a signal as an argument and disables interrupts, then it bitwise OR's the signal with itself and assigns it to mSignals. The mSignalsFlag variable is set to 1 and then interrupts are restored. There is another function called btaudio_configurePlayback() that sets sampling frequencies for

various audio codecs. The sampling frequencies for AVDTP_CODEC_TYPE_SBC are as follows: 16000, 32000, 44100, and 48000.

In the **audio_volume.c** source file, the bttask_setVolumeSignalHandler() function pointer is set to the address of the volumeSignalHandler() function defined within the audio_volume.c source file. This is where the mVolumeSignalHandler function within the bluetoothTask() function in **bttask.c** is ultimately pointing to. Basically, volumeSignalHandler() simply saves the old (current) volume, gets the new volume value from the ADC and converts it from ADC bits into a volume number. It then compares the old volume with the new one. If they are not the same, then the volume is updated to the new volume.

The next step was to add at least two files: **nmc_flprocess.h** and **nmc_flprocess.c** to the project. These files, if successful, would define and implement the frequency lowering algorithm. In an attempt to minimize the amount of editing of Microchip source code, at most, the **main.c** file would have to be edited to include a task function, nmc_ProcessTask(), inside the main while loop that is responsible for the frequency lowering.

Implementing the Algorithm part 1 – Circular Buffers

Two circular buffers would be used to store the incoming and outgoing samples while processing the sample data. A circular buffer, also known as a ring buffer, is a type of queue (itself a first-in, first-out (FIFO) data structure), where the head and tail are connected in such a way that when visualized, the buffer looks like Figure 4 on the next page.

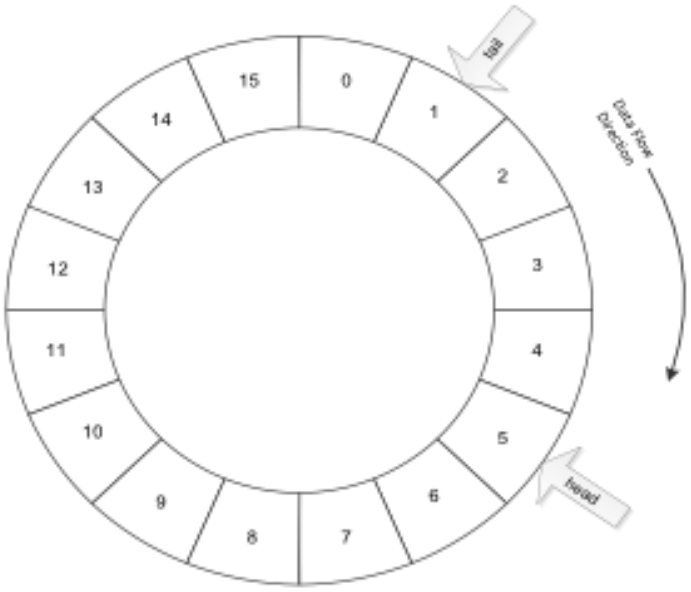

Figure 4. Ken Wada, *Structure of a Ring Buffer*, Available from Embedded, http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer

In order to understand how to implement a circular buffer, I wrote some sample C code to implement a basic circular buffer of length 16 (note that this is a power of 2, specifically 2^4). Sample code for a circular buffer published on Doug Brown's blog was modified to implement a circular buffer for testing purposes in Code::Blocks IDE with the gcc compiler. The following comment snippet from the sample circular buffer code notes the specific errors that were encountered.

```
/*************************************************************
***********
*      Compiled with GNU GCC Compiler 4.8.1 in Code::Blocks
*************************************************************
***********
* KNOWN ERRORS:
*   1) Garbage data is being input into the buffers
*   2) Buffer is acting like it is empty when removing data
from it
*   3) Crash with status -1073741819 on ending program
*      (Likely this is from not freeing memory somewhere)
*************************************************************
**********/
```

At first, it seemed that these errors were likely from mismanaging memory somewhere in the code. It turned out that the errors were from passing the address of pointers to the functions rather than passing the pointers themselves, which would lead to incorrect data being put into the buffer (the data being passed was the address of the pointer rather than the data the pointer was pointing to). Once these errors were corrected, the buffer code was implemented in **nmc_flprocess.c**. Once the buffer was implemented, it was time to work on figuring out how to read the analog-to-digital converter to obtain incoming data from the Bluetooth receiver.

Implementing the Algorithm part 2 – Reading the ADC

The analog-to-digital converter (ADC) is how the data from the input stream gets converted into a form that the processor can deal with, specifically from an analog sound wave to sampled digitized wave. Typically in C code, one has to perform a multitude of steps to initialize the ADC and then enter a loop to continuously read data and output it in some manner. First the ports must be configured to be either input or output, which is done by setting the associated TRIS or ANSEL bits. Next the channel must be selected, which is where the data will come in through. The ADC voltage reference must be set as well, and then the conversion clock must be set to a value that will allow enough time for the data to be converted from analog to digital (Giovanetti, accessed April 25, 2015).

Because the volume gets its data from the ADC, looking in the source code titled audio_volume.c for the function titled *volumeInit(void)* shed some light on how the ADC is set up to handle volume input. The first line in this function sets the potentiometer pin to an analog value (the potentiometer changes the volume). This is done via the *ANSELSET()* macro that calls the macro *CONCAT3* on the *ANSEL* macro, the given port number, and a SET bit. Looking up this macro shows that it does an operation on three variables, say *var1*, *var2*, and *var3* as follows: *var1##var2##var3*. According to Kernigan and Ritchie in their book, *The C Programming Language: 2nd Edition*, "The rules for nested uses of ## are arcane; further details may be found in Appendix A" (pg. 91, 1988). Going to Appendix A, section A12.3 on Macro Definition and Expansion, the rules for the ## operator is as follows:

> Second, if the definition token sequence for either kind of macro contains a ## operator, then just after replacement of the parameters, each ## is deleted,

> together with any white space on either side, so as to concatenate the adjacent tokens and form a new token. The effect is undefined if invalid tokens are produced, or if the result depends on the order of processing of the ## operators. Also, ## may not appear at the beginning or end of a replacement token sequence. (pg.230, 1988)

ANSEL is the analog select register. Because the ports are labeled A, B, C, D, and so on, if one uses *ANSELSET* on a specific pin, say B, the result of the macro *ANSELSET(B)* will give *ANSELBSET* which corresponds to setting port B.

The next two lines involve setting the ADC control registers (*AD1CON2* and *AD1CON3*). Typically, this would be done by passing a binary number to the *AD1CON2/AD1CON3* variable, which will set or clear specific pins. In the *volumeInit()* function, these two functions are passed an ORed series of macros defined to be 0 or bit-shifted binary values; all these macros are defined in adc10.h. The macros passed to *AD1CON2* are called *ADC_VREF_AVDD_AVSS* (the positive voltage reference is AVdd and negative voltage reference is AVss), *ADC_ALT_INPUT_OFF* (use MUXA input sample mode only as opposed to alternating between MUXA and MUXB modes), *ADC_BUF_16* (buffer is configured as a single 16-word buffer), *ADC_SAMPLES_PER_INT_2* (ADC interrupt occurs every 2nd sample), and *ADC_SCAN_ON* (set input scan selections for CH0 during SAMPLE A).

Variable AD1CON3 has the following two macros ORed together and passed to it: *ADC_SAMPLE_TIME_31* (auto sample time bit – 31 Tad) and *ADC_CONV_CLK_3Tcy* (Conversion Clock Select bits). A Tad is the time needed to complete a one-bit conversion, which is defined to be 1.6 μs for PIC microcontrollers (Verle, 2008).

When setting a new volume, it gets an ADC reading by setting *newVolume* to the macro *BT_VOL_ADC_BUFFER*, which according to their hardware profile definitions is defined as ADC1BUF0 in **HardwareProfile_PIC32_Bluetooth_Audio_Development_Board.h**, which itself is set as an extern volatile unsigned int with attribute: __attribute__((section("sfrs"))) in **p32mx450f256l.h**. This attribute specifies that the ADC1BUF0 variable is located in the section for Special Function Registers (SFRs), which helps speed up processing by letting the processor search only a specific subspace dedicated to the section for the various Special Function Registers instead of the entire memory space ("6.38 Specifying Attributes of Variables", accessed March 25, 2015).

Because the only clear indication of setting up for an ADC reading was in the volume task, the code from **audio_volume.c** was copied into **nmc_flprocess.c** as a structure of how to set up for an ADC reading. The specific macros corresponding to the volume pin and volume port will need to be changed to custom macros corresponding to reading the ADC for audio data coming in from the Bluetooth.

Unfortunately, at this point, implementing the algorithm proved to be far more challenging than originally anticipated due to lack of familiarity with Bluetooth technology and unclear documentation of the obtained source code. Another problem was the way the variables refer to other variables in the code. There was a very non-linear tracing through the various source code files. The overall flow of control and lack of explanatory documentation in sections was atypical. I was not able to find where to hook into the code to get my frequency-lowering algorithm operating on the data stream. Even though the variable names were clear, there was not enough labeling of sections and there was no key or explanation available online

or with the printed documentation. I had checked every possible route of learning and could not do so without specific Bluetooth training. Even trying to trace the code back to the pins via the datasheet was not useful.

Conclusion

It is disappointing that I was not able to actually achieve what I set out to do, that is, get my algorithm up and running on a Bluetooth development board and testing how well it works. However, it has been an illuminating experience to learn about Bluetooth technology and closely study an example of professional code to try to figure out how it works. Ultimately, it was unfeasible for me to work with the code I was given because the documentation was not explanatory either within the program or in the datasheets. I would need further experience with Bluetooth before being able to actually hook into their code. However, I have confidence the algorithm works, from earlier testing. Now that I have developed the C code, my next step may be to design my own development board with Bluetooth and test the code that way. I will not be giving up on this project and plan to continue until I can implement the algorithm in commercial devices. The number of people with high-frequency hearing loss will continue to rise over time, and new hearing assistive devices will be necessary. This one has the benefit of being a very portable, small program that should be easy to implement.

References

"6.38 Specifying Attributes of Variables." *GCC, the GNU Compiler Collection*. Accessed March 21, 2015. https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

Brown, Doug. "Microcontrollers: Interrupt-safe ring buffers." (blog) *Downtown Doug Brown.* Last modified January 01, 2013. http://www.downtowndougbrown.com/2013/01/microcontrollers-interrupt-safe-ring-buffers/

Gagné, J. P., M.B. Jennings, and K. Southall. "Understanding the Stigma Associated with Hearing Loss in Older Adults." *Hearing care for adults* (2009): 203-212.

Gates, G. A., and J. H. Mills. 2005. "Presbycusis." *Lancet* 366, no. 9491: 1111-1120. *CINAHL Plus with Full Text, EBSCOhost* (accessed April 25, 2015).

Giovanetti, Kevin. "Electronics 372 an Introduction to Microprocessors." *James Madison University*. Accessed April 25, 2015. csma31.csm.jmu.edu/physics/giovanetti/EE/ADC.pdf

Grau, Harold J. "Hearing." *biologyreference.com.* Accessed April 25, 2015. www.biologyreference.com/Gr-Hi/Hearing.html

Kernighan, B. W. and D. M. Ritchie. *The C Programming Language: 2nd Edition*. Englewood Cliffs, NJ: Prentice Hall, 1988.

"PIC32MX330/350/370/430/450/470." Datasheet. *Microchip.* 2012-2013. http://ww1.microchip.com/downloads/en/DeviceDoc/60001185D.pdf

"PIC32 Tutorial 2 – Timers." *EE Design Labs: The Engineering Tutorial & Project Blog.* (blog) Last modified August 16, 2014. Accessed October 2014. http://web.archive.org/web/20140816061148/http://www.eedesignlabs.com/tutorials/pic-microcontroller/pic32/pic32-tutorial-2-timers

"Quick Statistics." *National Institute on Deafness and Other Communication Disorders (NIDCD)*. Last modified April 20, 2015. http://www.nidcd.nih.gov/health/statistics/pages/quick.aspx

Ricketts, Todd. A. "Digital Hearing Aids: Current 'State-of-the-Art.'" *American Speech-Language-Hearing Association*. 2015. Accessed April 25, 2015. http://www.asha.org/public/hearing/treatment/digital_aid.htm

Seriani, Richard. "PIC Micro Controller Dissassemblers." *piclist.org.* 2014. Accessed October 2014. http://web.archive.org/web/20140421131817/http://piclist.org/techref/microchip/dissassemblers.htm

Shannon, Claude E. 1949. "Communication in the presence of noise." *Proceedings of the Institute of Radio Engineers* 37, no. 1: 10-21.

Verle, Milan. *PIC Microcontrollers.* Belgrade: mikroElektronika, 2008. http://www.mikroe.com/chapters/view/8/chapter-7-analog-modules/

Wada, Ken. “Ring buffer basics.” *Embedded.* Last modified August 07, 2013. http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer