

Experiments in the Limits of Digital Audio and
MIDI

Wes Featherstun
Advised by Dr. Jennifer Burg
Senior Honors Thesis
May 1, 2008

Experiments in the Limits of Digital Audio and MIDI

Wes Featherstun
Advised by Dr. Jennifer Burg
May 1, 2008

Introduction

Digital audio pervades most computer users' experiences. Any time a user listens to music on an mp3 player or plays a MIDI keyboard, they are dealing with digital audio. While the ability to produce and listen to digital audio has been around since at least the 1970s, in the past decade, both have exploded in popularity. The most obvious example is the success of Apple's iTunes, used for downloading and listening to music. For people who want a more interactive experience with digital audio, programs like Audacity are widely available. Working with interactive digital audio can take one of at least two forms. The first form is digital signal processing. In this case, a person works on a continuous signal which listeners perceive as sound. Another form is MIDI audio, which is a discrete form of data in which a person specifies a musical note to be played, without having to worry about the underlying waveform. Whether working with continuous or discrete audio data, the most important concern is whether or not the output sounds good. For instance, is sound constantly output or are there long gaps of silence? The purpose of this paper is to explore the limits of both digital signal processing and MIDI. The end goal is to understand what happens once a program writes data to the soundcard, with specific focus on the workings and effects of software, kernel, and hardware buffers as well as the consequences of the MIDI data rate.

The History of MIDI

The acronym MIDI stands for Musical Instrument Digital Interface. MIDI was first introduced in 1983. Electronic keyboard synthesizers were first used in the 1970s. At first, the synthesizers were monophonic, meaning only one note could be played at a time. A large problem with monophonic keyboards was that notes sounded flat when played by only one keyboard. However, if multiple keyboards played the same part, the sound was more robust. Soon after this discovery, a method for controlling multiple sound sources with a single keyboard was developed. This arrangement went out the window with the advent of polyphonic keyboards. The main problem with polyphonic keyboards was that each manufacturer was very secretive about the inner workings of their devices. These secrets included the way in which notes were represented as well as how a given synthesizer implemented polyphonic playback. Of course, each manufacturer also kept the specifics of their device a secret. As a result, two synthesizers from two different companies were unable to talk to each other. Dave Smith proposed a solution to this problem in 1981. He suggested that a standard method for representing notes and implementing polyphony be created that every manufacturer would adhere to. Surprisingly, manufacturers did just that. In 1983, the manufacturers announced the MIDI standard, through which any manufacturer's keyboard could be connected to any other manufacturer's keyboard.[Kirk 1999]

The MIDI Standard

A byte is commonly used to refer to a collection of eight bits. However, in the MIDI standard, each byte is made up of 10 bits. The two extra bits are present because MIDI signals are asynchronous. In synchronous signals, a clock pulse is sent continuously at a given interval. The

MIDI standard does not support this functionality. Asynchronous signals, on the other hand, do not send data continuously, but rather in stops and starts. Thus, each MIDI byte is preceded by a bit to signal the start of a MIDI byte and is followed by a bit to tell the receiver that the byte is finished transmitting. The start and stop bits are used by the hardware and only the octet reaches the software level.[Lehrman 1993] When the MIDI device begins a message, it sends the start bit. Upon reception of the start bit, the receiver starts its clock. The receiver's clock has a bit rate much higher than the MIDI bit rate, which compensates for any pulse rate changes in the receiver. The tenth bit turns the receiver's clock off again, to be reawakened upon reception of the next start bit. This set up means the two devices must remain synchronized for only 10 bits at a time and not indefinitely.[Kirk 1999]

The MIDI data rate is 31.25 kilobits per second. As stated earlier, there are 10 bits for each byte, and thus 3,125 bytes can be sent each second. MIDI messages are usually made up of two or three MIDI bytes. This leads to a rate of about 1,000 to 1,500 messages per second. MIDI note bytes are made up of three MIDI bytes. Therefore, 31250 bits per second divided by 30 bits per MIDI note results in a rate of 1041 MIDI notes per second. [Lehrman 1993] The MIDI data rate also ensures that the sender's clock and the receiver's clock must remain synchronized for only .00032 seconds at a time. [Kirk 1999] The requirement of an off bit in combination with the MIDI data rate ensures a .6 millisecond rest between any two MIDI bytes. [Lehrman 1993] This rest means that only one note can be transmitted or played at a time. However, notes are played close enough together that it sounds as if they are played simultaneously. This compromise emulates real life. It is impossible for a human to play the three notes in a chord at precisely the same instant. Instead, the notes are played in succession quickly enough that the human ear is unable to notice a gap.

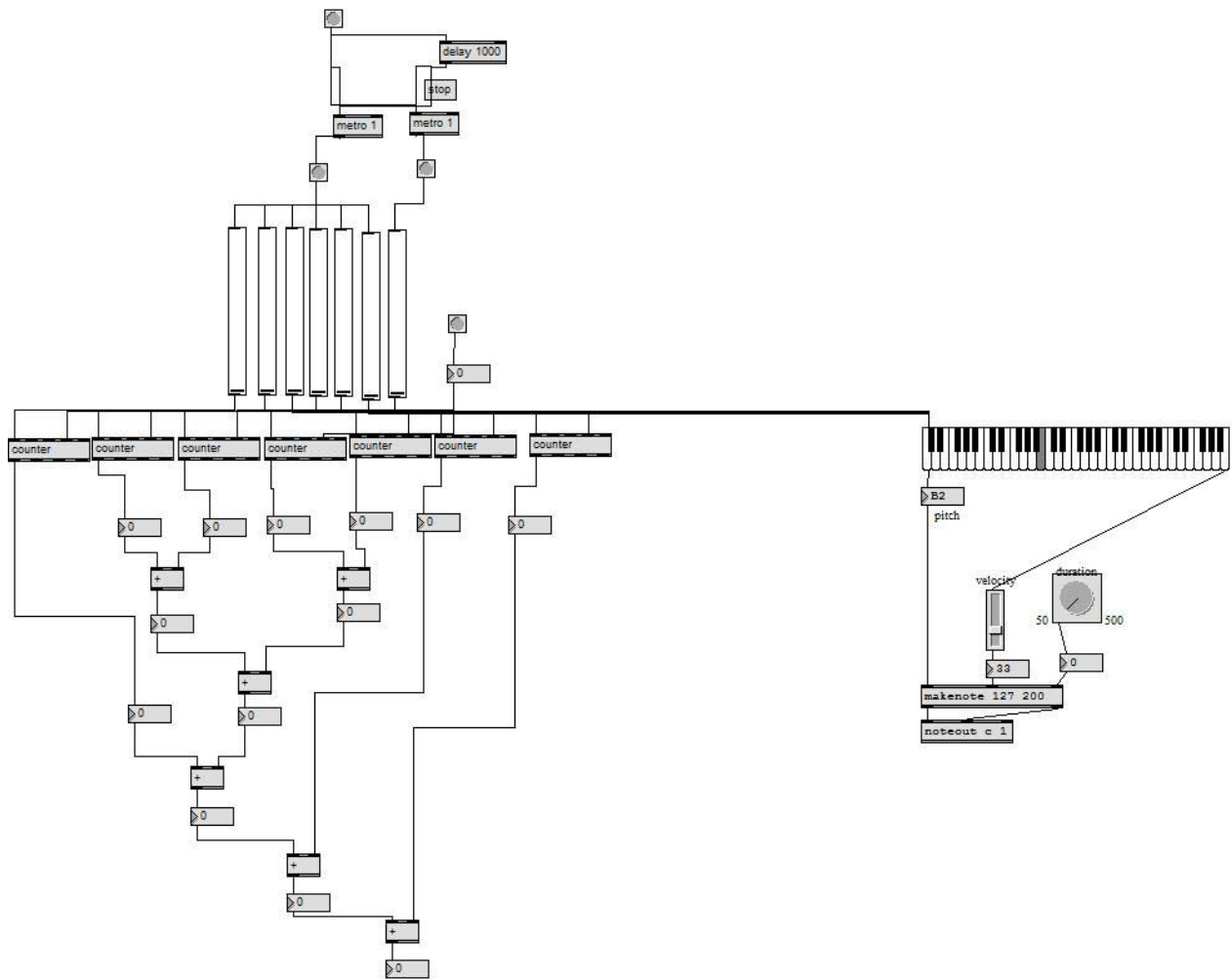
Limitations of the MIDI Standard

Two of the limitations of MIDI data are called MIDI lag and MIDI choke. MIDI lag refers to delays introduced by the MIDI data rate. MIDI lag refers to instances where transmission time adds up and becomes audible. [Rothstein 1995] All bits in a command must be received before any action can be taken. This immediately introduces a delay equal to the transmission time for 10, 20, or 30 bits. [MIDI for the Professional] A striking example requires consideration of two separate MIDI keyboards, named A and B. Suppose a musician wishes to play middle C on both keyboards at the same time and have both notes play simultaneously. Now, suppose keyboard A plays the note immediately through its speaker. If keyboard B is daisy chained to multiple other MIDI devices, however, MIDI lag will be apparent. Remember that a MIDI cable can transmit about 1041 MIDI notes per second. If keyboard B is daisy chained to 1041 MIDI devices, then it will take one second for a note played by keyboard B to reach the device which will actually play that MIDI note. Therefore, if a note is played simultaneously on keyboards A and B, then the note from keyboard B will play one second after the note from keyboard A. There are two ways to solve this problem. First, the performer could set a delay on keyboard A so that the keyboard waits one second before actually playing the note through the speaker. Second, the performer could daisy chain both keyboards to an equal number of MIDI devices, so that the delay of each note is equivalent. However, as MIDI lag is a consequence of the arrangement of MIDI devices, it is of little interest to the subject of this paper. It is presented here solely for a complete discussion of the limitations of MIDI.

MIDI choke occurs when a large number of messages is sent to a receiver within a short period of time. MIDI choke is a consequence of the MIDI data rate. When the MIDI device tries to

send more messages than the data rate can handle, the transmission medium is overloaded. [Rothstein 1995] For example, suppose a MIDI device is tasked with creating a three note chord for every one note that it receives. According to the MIDI data rate, the device can successfully receive 1041 MIDI notes per second. Since the receiver creates a three note chord for each one note it receives, the receiver must then send 3123 notes to the next device. However, since the MIDI cable can only send 1041 MIDI notes in a second, the chords will be sent over the course of three seconds. If another MIDI device then played a sequence of notes which assumed the one thousand chords were playing in the background, the choke would be noticeable when chords continued to play after the second device had finished its playback. In this case, a viable solution would be to split up the original notes among three separate devices. This would result in each device receiving 347 notes, after which the device would have to output 1041 notes a second, which is allowed by the MIDI data rate.

MIDI Choke in Windows Vista and XP



The MAX patch shown above was used to test the MIDI data rate in MAX/MSP on a Windows Vista machine. The metro object takes an integer argument which defines the gap in milliseconds between bang messages. By setting this argument to 1, the metronome will send

1,000 bangs per second. The resulting bangs are sent to an array of sliders. These bang messages cause the sliders to send their output. Each slider outputs an integer, which can be increased or decreased by moving the slider bar up or down. Slider output is connected both to a piano object and to counters. Each slider is connected to a unique counter. The piano object reads a slider's output as a MIDI note number. This number is then combined with the velocity and duration values. These three values are sent to a makenote object, which creates a MIDI note. The MIDI note is then sent to a noteout object, which actually sends the MIDI note to a specified output device. To start with, the MIDI output device was the computer's sound card. Later, MIDI output was sent to an external MIDI synthesizer. When a slider sends output, it also triggers a counter object, which increases the number output from the counter by one. These counters can be reset to zero with a single bang message. All of the various counter objects are added together to find the total number of messages sent. There is also a delay object. This object takes an argument which specifies the number of milliseconds to wait before sending a bang. In this case, the delay waits for 1,000 milliseconds, or one second, before stopping the entire patch. This set up will result in a count of how many MIDI notes are created and output in one second.

To begin with, only one slider was hooked up to metro object. The system was able to handle this with no trouble. After one second, there were 1,000 notes output. For every successful 1,000 MIDI messages sent, another slider was hooked up to the metro object. The system was able to handle 6,000 notes. However, once a seventh slider was added, the system had trouble keeping up. Fewer than 7,000 notes were actually output. In fact, the final counter indicates that an average of 5,800 notes is sent in a second. When this same patch was run on a Lenovo T60 laptop running Windows XP, the system had trouble outputting even 6,000 notes.

Recall that the MIDI data rate is 31.25 kilobits per second and can therefore send about 1,000 MIDI notes per second. While the XP laptop managed to output fewer notes than the Vista desktop, both machines produced significantly more notes than the data rate should allow. These results indicate that each computer was playing the notes as quickly as possible and were not adhering to the standard data rate. As the data rate was defined for the transfer of MIDI messages from one MIDI device to another, the standard rate refers explicitly to a MIDI cable. Therefore, the MIDI data rate only applies to hardware and not software. Since the computer is sending notes to its sound card, and therefore is not sending data over a MIDI cable, it is not restricted by the MIDI data rate. To further investigate, a MIDI keyboard was hooked up to the XP machine and set as the MIDI output for MAX. After running the patch again, the patch stopped sending notes after one second. However, the keyboard continued to output notes after the MAX patch had stopped. This is an example of MIDI choke, as MAX sent more notes than the line could handle and thus many notes were delayed. If the machine sends 6,000 notes in one second to the MIDI device, it will take the MIDI device 5.76 seconds to play back all the notes.

Since playback continues after the MAX patch has finished running, the created notes must be stored in a buffer. This buffer then empties as the notes are sent across the transfer medium to the external MIDI device. However, since both MAX/MSP and Microsoft Windows are proprietary software, no definitive conclusions can be reached about buffer sizes at either the software or operating system level. One is able to change buffer sizes of specific MAX objects, however there is no indication of whether or not MAX itself keeps a buffer of notes to be sent to the external MIDI device.

MAX obviously does not conform to the MIDI data rate while generating notes. However, it is difficult to tell whether or not it conforms to the data rate when playing a note via the soundcard. As soon as the delay object sends a bang to stop note generation, all audio stops. This

suggests that MAX does, in fact, play all the notes it generates, however the limitations of the human ear and the lack of an object that counts notes that are actually played prevents a conclusive answer. Since playback continues after the patch stops when sending data to an external MIDI device, it is reasonable to assume that all MIDI notes generated by MAX in one second are played on the external device. If this is true, then it is reasonable to conclude that all notes are also played when sending data to the soundcard, because the continued playback is a consequence of the MIDI cable, which is a penalty that is not paid when sending notes only to the soundcard.

MIDI Choke in Linux

To try and shed further light on the apparent conflict between the data rate of the machine and the data rate of the cable line, a similar experiment was performed on Linux. There are two Linux programs. The first writes MIDI notes to the computer's sound card. A counter counts how many notes are sent. This program ran for 300 seconds and the average number of notes per second was 200,929. Unfortunately, like in Windows, there is no readily apparent way to either count the number of notes actually played by the soundcard or to time how long it takes for the program to finish playing all of the notes. One can time how long it takes to generate the notes, but the time it takes to generate notes and the time it takes to actually play them are not necessarily equivalent. The only way found to suggest that the machine could play about 201,000 notes per second was the use of a sleep command and a for loop. At first, the for loop ran 201,000 times, then 402,000 times and so on up to 2,010,000 times. The sleep command caused the machine to sleep for 10 seconds and thus ensure that the program ran long enough to hear the MIDI notes being played. When 201,000 notes were sent to the soundcard, playback lasted for about one second. When 402,000 notes were sent, playback lasted for about two seconds, and so on.

To shed light on procedure used by the operating system to handle these notes, the source code for the Linux kernel version 2.6.22.15 was examined. The `sound_config.h` file reveals the size of the sequencer buffer to be 1024 bytes. When writing MIDI data to the sound card, one must use the `/dev/sequencer` library. If a process tries to write a new event into the event queue when there are already 1024 events buffered, the kernel ensures that the buffer will be drained to allow for the new event to be added to the event queue. The function used to ensure the buffer is drained plays every note in the event buffer before the 1025th note is put into the buffer. The kernel also drains the buffer at most once every 2,000 microseconds. The exact number of microseconds is a function of the tempo of the MIDI playback. Since Linux can generate and send approximately 200,000 MIDI notes to the sound card every second, the buffer fills up 195 times every second. [See Appendix A for source code]

The second Linux program was similar to the first, except instead of writing to the sound card, the program wrote MIDI notes to an external MIDI device. Since a MIDI cable can send about 1041 notes per second, it can handle about 5205 notes in five seconds. For this program, 5205 MIDI notes were generated and sent to an external MIDI synthesizer. Playback from the synthesizer took about five seconds. An interesting fact is that there were five distinct instances where many MIDI notes seemed to be played at once, followed by a moment of no notes being played. This influx of notes indicates the existence of a buffer. [See Appendix B for source code]

Once again, a look at the Linux source code provides answers as to what exactly is happening once the user program writes a MIDI note to the external device. The Linux kernel maintains a list of MIDI devices available to the user. The kernel then keeps track of two separate buffers for each MIDI device. There is one buffer for MIDI messages which are going out to an external synthesizer while another buffer keeps track of MIDI messages which are being received

by the computer. Each of these buffers holds 4000 bytes, which means each buffer can hold 4000 MIDI bytes and about 1333 MIDI notes. If the MIDI buffer is full, upon the next attempt to write data into the buffer, the MIDI buffer will drain. In addition, there is a time mechanism which empties the buffer after a certain amount of time has passed. The default value was not found, however the most frequently the buffer can be emptied is one every 2,000 microseconds. This timing mechanism explains why there is a sudden influx of notes being played and then a brief rest period. The influx occurs when the buffer is drained, either because it is full or because the time limit has been reached. There is then a pause as the buffer fills again.

While the manipulation of the MIDI buffer in the Linux kernel is known, whether or not there is a buffer on the MIDI device itself is unknown. While there is presumably a buffer on the device, there are no details available for how large the buffer may be.

The results of this Linux experiment support those of the MAX patch. Linux obviously pays no need to the MIDI data rate when generating notes, but rather generates and sends notes as quickly as instructed. However, when transmitting these notes across a cable to a MIDI device, MIDI choke occurs because the cable must obey the data rate. Thus, while the computer can generate 201,000 MIDI notes per second, the device can only 1041 MIDI notes in a second, which is equivalent to 193 seconds of playback.

Digital Signal Processing

Digital signal processing in Linux is more flexible than the MIDI standard. DSP is used for real time audio processing, such as adding a reverb effect to a note as it is played. The limits imposed on signal processing change depending on three things: the sampling rate, the time it takes to read and write to the sound card, and the size of the buffer in the application software. [Tranter 1996]

There are two different types of signal that one might wish to process. The first type is a signal read from an audio file. The computer reads the signal into a buffer and performs some sort of processing on the signal before it is written to the sound card. When the computer performs too much processing during file playback, glitches occur in the form of breaks and skips. The second signal type is a real time signal, where the computer captures audio from an external source, such as a microphone, and performs some processing before writing the signal to a file in the computer or outputting the altered signal to the sound card. Too much processing on this type of file results in audio that is not entirely recorded.

Read and Write Times

Reading from and writing to the hard drive are two of the slowest operations for a computer. Since seamless playback depends heavily on the time it takes for the computer to perform tasks between each read and write operation, it is instructive to discover how long each read and each write takes. When reading from the microphone, each read operations has to equal the amount of time it takes for a certain number of bytes to be filled with audio samples. For instance, if the buffer size is 1024 bytes and the sampling rate is 44.1 kHz, then the buffer holds .023 seconds of audio. Therefore, it takes the computer .023 seconds to read these 1024 bytes into a buffer. The following table indicates how long various buffer sizes take, in seconds, to read from streaming audio.

Buffer Size in	44.1 kHz	192 kHz
----------------	----------	---------

Bytes		
256	.0058	.0013
512	.0116	.0027
1024	.0236	.0053
2048	.0464	.0106
4096	.0929	.0213

While the time to read from the microphone can be derived via simple division, the other read and write operations are not so simple, so a program was written to discover these times. The program is a series of nested loops. The first level determines whether the program is currently testing the time to write to a file, write to the sound card, or read from a file. The second level runs through buffer sizes of 256, 512, 1024, 2048, and 4096 bytes. Finally, the program either reads from a file or writes to either a file or the sound card a certain number of times. The number of times read or written was scaled according to the buffer size. For example, a buffer size of 2 bytes went through this loop many more times than a buffer size of 4096 bytes. The only thing timed was how long it took to read or write the specified number of times. This time was measured in seconds, which allowed for derivation of the average number of seconds each read or write took. The results are in the following table.

Buffer Size	Write To Sound Card in Seconds	Write to File in Seconds	Read From File in Seconds
256	.00134	5e-06	4e-08
512	.00266	1.8e-05	4e-08
1024	.0054	4.5e-05	4e-08
2048	.0106	.0001	4e-08
4096	.0214	.0001	4e-08

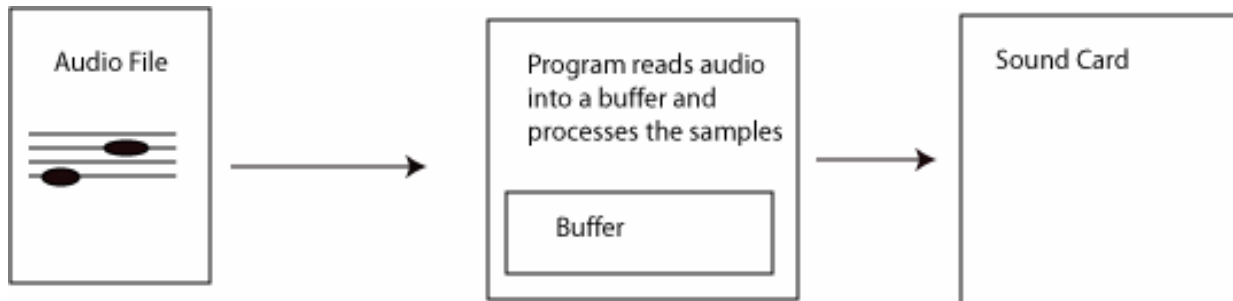
[See Appendix C for source code]

Writing to the sound card produces the results one would expect. Writing twice as many bytes takes twice as many seconds. As the buffer size increases, writing to the sound card actually takes a significant amount of time. The .0213 seconds it takes to write a 4096 byte buffer to the sound card is essentially equivalent with the .0214 seconds represented by a 4096 byte buffer using 8 bit samples at a sampling rate of 192 kHz.

In contrast, neither reading to or writing from a file takes a significant amount of time. Reading from a file stays constant. When writing to a file, doubling the buffer size actually triples the amount of time it takes to write when going from 256 bytes to 512 bytes and from 512 bytes to 1024 bytes. When the buffer size increases from 1024 bytes to 2048 bytes, the time approximately doubles, which is the expected result. Then, there does not seem to be any difference between the time it takes to write a buffer of 2048 bytes and a buffer of 4096 bytes. These disparities may be a result of Linux process scheduling. Background processes may have taken control of the processor during the for loop used to find the average time, which could skew the data. This explanation only covers the matter of the write times tripling as the buffer size doubles. Finding an explanation for the consistent write time between buffers of size 2048 bytes and 4096 bytes is a topic for future investigation.

Reading Audio From A File

The first program reads audio from a file, processes it, and subsequently outputs the signal to the sound card. The general form of this program is to open a file which is filled with audio samples. The program then reads in samples until the program's buffer is full. Once the buffer is full, it processes the signal and then writes the buffer to the sound card.



A program was used to explore the effects of increasing buffer size on an audio signal read from a file. First, the program prompts the user for two values. The first value is the minimum buffer size. The second value is the maximum buffer size. The program first runs through a loop using the minimum buffer size input by the user. Each time through the loop, the buffer size is doubled until the current buffer size exceeds the maximum sized defined by the user. Within this for loop, the program first reads samples from a file into a buffer. Once the buffer is filled, the program processes the audio signal. For this experiment, the buffer was copied to a temporary array. An insertion sort was then run on the temporary array. This technique allowed for $C \cdot n^2$ processing on the temporary array. Since the original buffer was left untouched, it could then be written to the sound card. An n^2 algorithm was chosen due to its fast growth. The rapid growth provides more dramatic changes in audio quality when running the test program. The buffer sizes used to test both this and the next program were 256, 512, 1024, 2048, and 4096 bytes. The C value was 10. Playback remained relatively smooth until the buffer size reached 1024 bytes, at which point breaks became noticeable. Once the buffer size grew to 2048 bytes, playback was so overwhelmed by gaps in the sound that playback slowed down considerably. For example, a ten second audio clip took noticeably longer than ten seconds to play due to all of the pauses in playback. The algorithm is sketched in pseudocode below.

```
for(n = minBuffSize; n <= maxBuffSize; n*=2){
    read audio samples into a software buffer of size n
    for(int i = 0; i < C; i++)
        process the samples
    write the n-byte buffer to the sound card
}
```

[See Appendix D for source code]

For this experiment, the signal had one channel which used 8 bits per sample. Sampling rates of 44.1 kHz and 192 kHz were used. Therefore, the bitrate for the 44.1 kHz signal was 352,800 bits per second. The bitrate for the 192 kHz signal was 1,536,000 bits per second.

Operation Count for $C \cdot n^2$

Buffer Size	C = 1	C = 5	C = 10
256	65,536	327,680	655,360
512	262,144	1,310,720	2,621,440
1024	1,048,576	5,242,880	10,485,760
2048	4,194,304	20,971,520	41,943,040
4096	16,777,216	83,886,080	167,772,160

The table above indicates how the number of operations performed on a buffer grows as the buffer and C values increase. Note that each time the buffer size doubles, an n^2 algorithm where $C = 1$ requires four times as many operations as the previous buffer. With a little thought, this growth makes sense. At every step, the buffer size doubles. Since the number of operations is the square of the buffer size, when the buffer size doubles, the number of operations actually grows by 2^2 , or 4. While the above table counts the number of operations for a $C \cdot n^2$ algorithm, future analysis will measure running time in seconds. This is necessary because the read and write times were measured in seconds. The following table lists the number of seconds it takes to run a $C \cdot n^2$ algorithm where $C = 1$.

Buffer Size	Seconds
256	.000118515
512	.000442455
1024	.00123749
2048	.00486316
4096	.0194105

Using the above table and the time it takes to read data from a file, a conclusion can be reached about the effect of the buffer size on reading a file. Suppose there is a file whose size is 409,600 bytes. This file will require 1,600 file reads with a buffer size of 256 bytes. If the buffer size is 4096 bytes, however, there are only 100 reads. Obviously, as the buffer size increases, the number of reads decreases. However, as demonstrated by the reading times found, the read time is a constant $4e-08$ seconds regardless of the buffer size. Now to put all of these facts together. Suppose T_n represents the processing time in seconds to run an n^2 algorithm at $C=1$ for buffer size n . Let R_n represent the time it takes to read a buffer of size n . If the file size is 409,600 bytes, then the formula

$$S = 409,600/n * (R_n + T_n)$$

indicates the total number of seconds it takes to read and process a file of size 409,600 bytes for a given buffer size. If the buffer size is 256 bytes, the total time the program performs processing on the signal is .189688 seconds. However, using a buffer size of 4096 bytes, the program performs

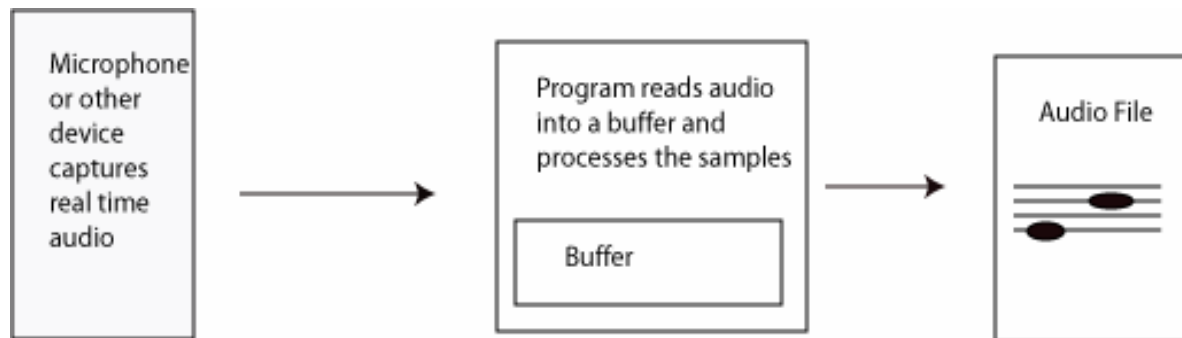
processing for 1.941054 seconds. These results indicate that despite requiring fewer reads, using a larger buffer size actually increases the total processing time of a program. Therefore, there is a trade off between reducing the number of reads and the amount of processing one can do on a signal before glitches occur.

Increased processing time results in audio glitches, specifically breaks in the audio. When the program reads audio from a file, its buffer is filled with audio samples. For flawless playback, the program must write audio to the sound card at certain intervals. Breaks occur when the program fails to maintain this rate. For a more concrete example, suppose the program reads in one second of audio at a time. After processing the audio, the program writes this second of audio to the sound card. While this second is playing, the program reads and processes a second chunk of audio. If processing this chunk takes two seconds, there will be a one second pause between the end of the first chunk's playback and the start of the second chunk's playback. This process does ensure that all of the audio is eventually played, as the program will read in and play every last byte in the file. The question is whether or not playback will be smooth. The playback of a given algorithm begins to suffer as buffer size increases. Since the number of operations grows quickly, the computer spends much more time processing a signal as the buffer size increases. The audio playback starts to break up when the number of seconds it takes to process a given buffer size is greater than the number of seconds the buffer actually represents.

Another difficulty encountered in the experiment was that sometimes the audio would seem to stutter. This always occurred at the very end of playback. This issue is a flaw specific to the test program. The program reuses a n byte array every time it reads from a file. If the size of the file is not evenly divisible by the buffer size, stuttering will occur. For example, suppose a file size of 4100 bytes and a buffer size of 4096 bytes. First, the program reads in 4096 bytes and outputs these samples to the sound card. This leaves 4 bytes that must be read. However, since the array is reused, the four bytes are written at the beginning of the array. The remaining 4092 bytes retain data from the first read, and therefore the audio "stutters" as the same audio is played twice.

Reading Real Time Audio From An Input Device

When processing real time audio from an input device, the danger lies in spending so much time processing data that a significant amount of audio is left unrecorded. For example, when the computer reads data from the microphone, it reads a certain number of bytes at a time and places these bytes in a buffer. The computer then performs processing on that buffer and writes the result to a file on the hard drive. However, while this processing and writing is occurring, there is still audio input going on. For example, someone may be talking into the microphone. While the computer is processing, the person continues talking. However, while the processing occurs, the computer fails to record some amount of audio. The question is how much audio can be missed at a time before the dropped portions become apparent.



In the above picture, the input from the microphone is constant. However, the program only reads a certain number of samples at a time and then stops reading during the last two stages. Once the output is written to the sound card, then the program begins reading from the microphone again.

To illustrate the issues faced when reading audio in through a microphone, a program very similar to the previous one was used. For this program, the sampling rate was 44.1 kHz. Again, the user was prompted for a minimum and maximum buffer size. The minimum buffer size was multiplied by two until the current buffer size was greater than the maximum buffer size. At each buffer size, 327,680 samples were read. At a sampling rate of 44.1 kHz, this corresponds to about seven seconds of audio. The program read from the microphone until the current buffer was full. Then, the program processed the data, once again running insertion sort on a temporary array. Once the processing was complete, the buffer was added to another array which contained all audio recorded thus far. Pseudocode for this program follows.

```

int runningBuffer[327680]
int currentPosition = 0
for( int n = minBuffSize; n <= maxBuffSize; n*=2){
    while fewer than 327,680 samples have been read from the microphone
        read n bytes from the microphone
        run insertion sort on the n bytes
        copy the n bytes read from the microphone into
            runningBuffer[currentPosition]
        currentPosition += n
    }
write runningBuffer to soundcard
[See Appendix E for source code]
  
```

While the problem faced with the previous program was gaps in playback, the problem when recording real time audio is actually missing samples. Consider the example of someone counting from one to twenty into the microphone. The computer is either reading in bytes from the microphone or performing processing on the bytes read. However, while the computer is processing data, the speaker keeps counting. The computer fails to record anything the speaker says while it is busy processing data. Therefore, while the speaker counted from one to twenty by ones, when the sound actually captured is played back, it may sound like the speaker skipped certain numbers. For example, maybe every other number is skipped and thus it sounds like the speaker counted from one to twenty by twos.

Like the previous experiment, the culprit for any unrecorded audio is the number of operations required by the processing algorithm. As both programs use a $C*n^2$ algorithm, the operation count table from the previous program remains accurate for this program. Therefore, the timing information from the previous program remains the same. Using the operation count from the previous table allows for an estimate of how many samples will be missed based on the processing time needed. Assume a machine can execute 100 million instructions per second (MIPS) and that the program is using a sampling rate of 44.1 kHz. Assume that every instruction a machine uses is used for this program, so if the program must do 256^2 operations to process the buffer, it takes exactly 65,536 machine instructions before the processing is complete. While this is not practical, it provides a good theoretical estimate. Suppose a buffer size of 256 bytes. This buffer will require 65,536 instructions, or .00065536 seconds to complete. For that .00065536 seconds in which the computer is processing the signal, it is not recording .00065536 seconds of what the speaker is saying. For a sampling rate of 44.1 kHz, that is equivalent to about 29 missed samples. The problem becomes more striking if a buffer size of 4096 bytes is used. In this case, the program is missing .16777 seconds of audio, which equals about 7399 samples.

It is important to note that while in the previous program all of the audio is eventually played, in this example, all of the audio is not recorded. Even under the best circumstances, all of the audio is not recorded. However, when using a $C*n^2$ algorithm, the amount of unrecorded audio grows quickly as the buffer size grows. While the main focus of this analysis has been on large gaps of missing audio, there is another interesting effect that can occur. Suppose the buffer size used is 256 bytes. At 44.1 kHz, this is equivalent to .0058 seconds. Now suppose it takes .0058 seconds to process this signal using some algorithm. For every 256 bytes the program reads from the microphone, 256 bytes will be lost. Therefore, the frequency of the signal will double. As such, while the biggest problem is missing large chunks of data while the computer is busy processing, an extension of this problem is that the overall frequency of the signal can be significantly affected.

The Linux drivers use DMA buffers to send data to the sound card. The buffer must fill up before the kernel sends the sound samples to the sound card. A typical buffer size is 64 kilobytes. [Tranter 1996] To verify that the buffer size on the Linux machine used for the previous programs was indeed 64 kilobytes, an examination of the Linux source was needed. A look at `soundcard.c` led to `sound_config.h`. Line 36 affirms that the buffer size is 64 kilobytes. At a sampling rate of 44.1 kHz, this buffer would take 1.49 seconds to fill up. The DMA buffer takes .341 seconds to fill at a sampling rate 192 kHz. The DSP buffer works similarly to the MIDI buffers. If the buffer is full, then the audio is sent to the sound card upon the next attempt to add a sample to the buffer. **[sound_config.h]**

Conclusions

A major problem encountered while trying to collect and verify results was the lack on an objective metric for measurement. With real time playback, the main concern is what the listener perceives. As a result, the computer can spend as much time reading, writing, and processing as it needs until the listener perceives an issue. As well, since neither MAX nor Linux provides a measurement for how long playback of notes sent to the sound card or an external MIDI device takes once the note is actually sent.

As evidenced by the gap between the performance of the Vista desktop and the XP laptop when running the MAX patch, much of the limitations on digital audio are machine-specific. A faster machine will simply be able to perform more operations before one notices an issue. For an

example of how striking this difference can be, consider the program which reads audio in from the microphone and then processes the audio before writing it to a file. If the computer can execute 1 MIPS and a program reads audio in 4096 byte buffer at a sampling rate of 44.1 kHz, then the computer will miss 739,875 samples while processing the buffer. 739,875 samples translates to approximately 16.8 seconds that will not be recorded. However, a compute running at 100 MIPS will lose 7,398 samples, or about .168 seconds. Finally, consider a machine that runs one billion instructions per second. In this case, the machine will only miss 739 samples and .00168 seconds of audio.

As demonstrated by the MAX patch and the two Linux programs, the MIDI data rate refers only to the line speed of a MIDI cable. When one is considering software, the only limitations on the number of notes that can be sent and played are those of the system itself, such as how fast the processor is.

The goal to understand what happens at the operating system and hardware levels once a program writes audio to the sound card was successful in some ways. The inner workings of Linux were discovered and a decent understanding of its mechanisms was gained. A lingering question remains in regards to the 64 kilobyte DMA buffer used by the `/dev/dsp` library. Supposedly, the buffer must be filled before any sound is output. But what happens if there are not 64 kilobytes of audio left to send? Presumably the buffer must also be drained frequently enough to maintain the playback frequency specified by a given program. Unfortunately, no evidence of such a mechanism was found in the source code, and therefore the question remains a topic for future consideration. Other topics for future consideration include continued attempts to crack the black boxes of Windows and MAX/MSP with the hopes of understand what occurs at the operating system and hardware levels after a noteout object is executed.

References

Print Publications

Kietntzle, Tim. A Programmer's Guide to Sound. Reading: Addison-Wesley Developers P, 1998.

Kirk, Ross, and Andy Hunt. Digital Sound Processing for Music AndMultimedia. Oxford: Focal P, 1999.

Lehrman, Paul D., and Tim Tully. MIDI for the Professional. New York: Amsco Publications, 1993.

Rothstein, Joseph. MIDI: a Comprehensive Introduction. Madison: A-R Editions, Inc, 1995.

Tranter, Jeff. Linux Multimedia Guide. Beijing: O'Reilly, 1996.

Winkler, Todd. Composing Interactive Music: Techniques and Ideas Using Max. Cambridge: The MIT P, 2001.

Web Publications

Sapp, Craig S. "Introduction to MIDI Programming in Linux." Jan. 1999. Stanford U. 11 May 2008 <<http://ccrma-www.stanford.edu/~craig/articles/linuxmidi/>>.

"The Linux Homepage." Linux Online. 11 May 2008 <<http://www.linux.org/>>.

"Max 5." Cycling '74. 11 May 2008 <<http://www.cycling74.com/products/max5>>.

Appendices

Appendix A

```
/*Thanks to Craig Sapp for the procedure of how to write MIDI notes to the soundcard
*using /dev/sequencer*/
```

```
#include <iostream>
#include <unistd.h>
#include <linux/soundcard.h>
#include <fcntl.h>
#include <stdio.h>
using namespace std;
```

```
int main(){
```

```
    char* device = "/dev/sequencer";
    unsigned char devnum = 1;
    unsigned char data[4] = {SEQ_MIDIPUTC, 0, devnum, 0};
```

```
    int fd = open(device, O_WRONLY,0);
    if(fd < 0){
        printf("Error: cannot open %s\n", device);
    }
```

```
    int start = 0;
    int end = 0;
    int count = 0;
```

```
        start = time(NULL);
```

```
        /*present are two separate loop possibilities. The for loop
        * was used in sending some number of notes to the soundcard and
        * seeing how long playback took. The while loop sent MIDI notes for
        * 300 seconds and then found the average number of notes per second
        */
```

```

        //for(int i = 0; i < 2000000; i++){
        while(end - start < 300){
            data[1] = 0x90; //note-on command
            write(fd, data, sizeof(data));
            data[1] = 60;
            write(fd, data, sizeof(data));
            data[1] = 127;
            write(fd, data, sizeof(data));
            count++;
            end = time(NULL);
        }

    sleep(10);
    close(fd);

    float avg = count / 300.0;
    cout << count << " * 3 = " << count * 3 << endl;
    cout << avg << endl;

    return 0;
}

```

Appendix B

*/*thanks to Craig Sapp for examples of how to write MIDI notes to an external device*/*

```

#include <iostream>
#include <unistd.h>
#include <linux/soundcard.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

using namespace std;

int main(){

    char* device = "/dev/midi1";
    unsigned char data[3] = {0x90, 60, 127};

    int fd = open(device, O_WRONLY,0);
    if(fd < 0){
        printf("Error: can't open /dev/midi1");
        exit(1);
    }
}

```

```

    }

    int start, end;
    start = time(NULL);
    for(int i = 0; i < 5210; i++)
        write(fd,data,sizeof(data));
    end = time(NULL);
    cout << "done with 5-ish seconds\n";
    cout << "took " << end - start << " seconds to send to device\n" << flush;
    //sleep(5);
    close(fd);

    return 0;
}

```

Appendix C

```

#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/soundcard.h>
#include <time.h>
#include <fstream>
#include <errno.h>
#include <iostream>
#include <signal.h>

#define LENGTH 30
#define RATE 44100
#define SIZE 9
#define CHANNELS 1
#define CONSTANT 12216
#define LOGN 12
#define C 0

unsigned char buf[4096];

using namespace std;
void insertionSort(char* a, int n);

int main(){

```

```

int fd, arg, status;
int i,j,k,begin,end,bufEnd;
char temp;

fd = open("/dev/dsp",O_RDWR,0);

if(fd < 0){
    perror("Opening /dev/dsp failed\n");
    exit(1);
}
//standard set up
arg = SIZE;
status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
if(status == -1)
    perror("Unable to set sample size\n");

arg = CHANNELS;
status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
if(status == -1)
    perror("Unable to set number of channels\n");

arg = RATE;
status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
if(status == -1)
    perror("Unable to set sampling rate\n");

cout << "sampling rate set to " << arg << endl;

double cycle, elapsedCycles;
//int n = 1;
//
ofstream record("ReadAndWriteTimes2.data", ios::app);
for(int i = 0; i < 4096; i++)
    buf[i] = 100;
for(int x = 3; x <= 3; x++){
    cout << "x: " << x << " ";
    for(int i = 256; i <= 4096; i*=2){
        cout << " i: " << i << endl;
        int times;

        if (i == 256)
            times = 100000000;
        else if (i == 512)
            times = 100000000;
        else if (i == 1024)

```

```

        times = 100000000;
    else if (i == 2048)
        times = 100000000;
    else if (i == 4096)
        times = 100000000;

    if(x == 1){
        int startTime,endTime;
        double startCycles,endCycles;

        startTime = time(NULL);

        for(int j = 0; j < times; j++)
            write(fd,buf, i);

        endTime = time(NULL);

        double avgSeconds = (double)endTime - startTime;
    avgSeconds = avgSeconds/times;

        record << "to SC. Bytes: " << i << " seconds: "
        << avgSeconds << endl;
    }
    else if(x == 2){

        int startTime, endTime;
        double startCycles, endCycles;

        ifstream grav("44.voc",ios::in);

        startTime = time(NULL);

        for(int j = 0; j < times; j++){
            //if(!grav.eof())
                grav.read(reinterpret_cast<char*>(buf),i);

        endTime = time(NULL);

        double avgSeconds = ((double)(endTime)
            - startTime)/times;

        record << "from File. Bytes: " << i
        << " seconds: " << avgSeconds << endl;
        grav.close();
    }

```

```

    }
    else if(x == 3){
        ofstream grav("outfile.voc", ios::out);

        int startTime, endTime;
        double startCycle, endCycle;

        char* meh = new (nothrow) char[4096];
        for(int count = 0; count < 4096; count++)
            meh[count] = buf[count];

        startTime = time(NULL);

        for(int j = 0; j < times; j++)
            grav.write(meh,i);

        endTime = time(NULL);

        double avgSeconds = ((double)endTime -
            startTime)/times;
        record << "to File. Bytes: " << i
        <<" seconds: " << avgSeconds << endl;
    }
}
}

```

Appendix D

```

#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/soundcard.h>
#include <time.h>
#include <fstream>
#include <errno.h>
#include <iostream>
#include <signal.h>
#include "timing.h"
#define LENGTH 30
#define RATE 192000
#define SIZE 9

```

```

#define CHANNELS 1
#define BUFF_SIZE 4*LENGTH*RATE*SIZE*CHANNELS/8
#define C 1
void signal_catcher(int);
unsigned char buf[8192];
using namespace std;
void insertionSort(char* a, int n);

int main(){
    clock_t start, finish;
    float seconds = 0;
    int fd, arg, status;
    int i,j,k,begin,end,bufEnd;
    char temp;
    int beginSize, endSize;
    cout << "What size buffer would you like to begin with" << endl;
    cin >> beginSize;
    cout << "What size buffer would you like to end with" << endl;
    cin >> endSize;
    fd = open("/dev/dsp",O_RDWR,0);
    if(fd < 0){
        perror("Opening /dev/dsp failed\n");
        exit(1);
    }
    arg = SIZE;
    status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
    if(status == -1)
        perror("Unable to set sample size\n");
    arg = CHANNELS;
    status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
    if(status == -1)
        perror("Unable to set number of channels\n");
    arg = RATE;
    status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
    if(status == -1)
        perror("Unable to set sampling rate\n");
    cout << "sampling rate set to " << arg << endl;
    double cycle, elapsedCycles;
    int n;
    for (int n = beginSize; n <= endSize; n=n*2) {
        seconds = 0;
        unsigned char* buf = new (nothrow) unsigned char[n];
        char* copy = new (nothrow) char[n];
        elapsedCycles = 0;
        int count = 0;
        ifstream grav("192.voc",ios::in);

```

```

//start = clock();
//start = time(NULL);
while(!grav.eof()){
    cycle = rdtsc(); //counts cycles instead of seconds
    grav.read(reinterpret_cast<char*>(buf),n);
    start = clock();
    for(int i = 0; i < C; i++){
        //start = time(NULL);
        for(int j = 0; j < n; j++)
            copy[j] = buf[j];
        insertionSort(copy, n);
    }
    finish = clock();
    seconds += finish - start;
    status = write(fd,buf, n);
    elapsedCycles += rdtsc() - cycle;
    count++;
}

//finish = time(NULL);
//seconds = (finish - start) / (float) (CLOCKS_PER_SEC);
seconds = seconds/(float)(CLOCKS_PER_SEC);
seconds = seconds / count;
cout << "loop averaged took " << seconds << "seconds." << endl;
grav.close();
int operation_count = (n * n * C) + n*C;
cout << "n^2 * c: " << n*n*C << " whole thing: " << operation_count
<< " " << (elapsedCycles)/count << endl;
}
}

```

```

//insertionSort is n^2, just used to waste time
void insertionSort(char* a, int n) {
    char temp;
    int i, j;

    for(j = 1; j < n; j++){
        //cout << j << ' ';
        temp = a[j];
        for(i = j-1; i >= 0; i--){
            if((int)(temp - '0') < (int)(a[i] - '0')){
                a[i+1] = a[i];
                if(i == 0)
                    a[0] = temp;
            }
        }
        else{

```

```

        a[i+1] = temp;
        break;
    }
}
}
}

```

Appendix E

```

//FromMicrophoneToFile.c
//Real Time Processing
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/soundcard.h>
#include <time.h>
#include <fstream>
#include <errno.h>
#include <iostream>
#include <signal.h>
#include "timing.h"
#define LENGTH 5
#define RATE 44100
#define SIZE 8
#define CHANNELS 1
#define BUFF_SIZE 4*LENGTH*SIZE*RATE*CHANNELS/8
#define C 20
void insertionSort(char* a, int n);
unsigned char buf[BUFF_SIZE];
using namespace std;
int main(){
    int fd, arg, status;
    int i,j,k,begin,end,bufEnd;
    char temp;
    clock_t start, finish;
    float seconds;
    int beginSize, endSize;

    //cout << "What size buffer do you want to begin with?" << endl;
    //cin >> beginSize;
    beginSize = 256;
    //cout << "What size buffer do you want to end with?" << endl;
    //cin >> endSize;

```

```

endSize = 4096;
fd = open("/dev/dsp",O_RDWR,0);

if(fd < 0){
    perror("Opening /dev/dsp failed\n");
    exit(1);
}
arg = SIZE;
status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
if(status == -1)
    perror("Unable to set sample size\n");

arg = CHANNELS;
status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
if(status == -1)
    perror("Unable to set number of channels\n");

arg = RATE;
status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
if(status == -1)
    perror("Unable to set sampling rate\n");

for (int n = beginSize; n <= endSize; n = n*2) {
    cout << "buffer size = " << n << endl;
    //Listen at the sound card until there's a sound loud enough to
    //register as the actual beginning of the sound that we
    //want to record.
    cout << "Waiting to record about 7 seconds worth in chunks of " << n << " bytes" <<
endl;
    status = read(fd, buf, 1);
    while(buf[0] >= 125 && buf[0] <= 131){
        status = read(fd, buf, 1);
    }
    char* copy = new (nothrow) char[n];
    cout << "Beginning recording\n";
    start = clock();
    //Read 327680 bytes of audio in chunks of n bytes. At a sampling rate
    //of 44100/sec, this is about 7.4 seconds of audio.
    for (i = 1; i <= (327680/n); i++) {
        read(fd, buf + (n*i), n);
        for(int k = 0; k < C; k++){
            for(int j = 0; j < n; j++)
                copy[j] = buf[(n*i)+j];
            insertionSort(copy, n);
        }
    }
}

```

```

        finish = clock();
        seconds = (finish - start)/(float) (CLOCKS_PER_SEC);
        cout << "Process took " << seconds << " seconds." << endl;
        status = ioctl(fd, SOUND_PCM_SYNC, 0);
        if (status == 1)
            perror("SOUND_PCM_SYNC failed\n");
        cout << "You said:" << endl;
        status = write(fd, buf, 327680);
        if (status != 327680)
            perror("Wrote wrong number of bytes.\n");
        buf[0] = 0;
    }
}

```

```

void insertionSort(char* a, int n){
    char temp;
    int i, j;

    for(j = 1; j < n; j++){
        //cout << j << ' ';
        temp = a[j];
        for(i = j-1; i >= 0; i--){
            if((int)(temp - '0') < (int)(a[i] - '0')){
                a[i+1] = a[i];
                if(i == 0)
                    a[0] = temp;
            }
            else{
                a[i+1] = temp;
                break;
            }
        }
    }
}

```