

MUSC 208 Winter 2014
John Ellinger, Carleton College

Lab 3 - Audio Files

Lab 3 will introduce methods for playing and mangling an audio file. We'll use the computer keyboard as our first HID (Human Interface Device) to trigger samples stored on disk. We'll use a little Audacity, a little Octave, and mostly ChuckK/miniAudicle in this lab. We'll use several typical programming constructs: comments, types, variables, arrays, if-else logic, and looping constructs like while, until, do, and for loops.

Setup

If you're doing these labs on your laptop, you must have Audacity, Octave, and miniAudicle installed and working. Otherwise work on an iMac in the lab.

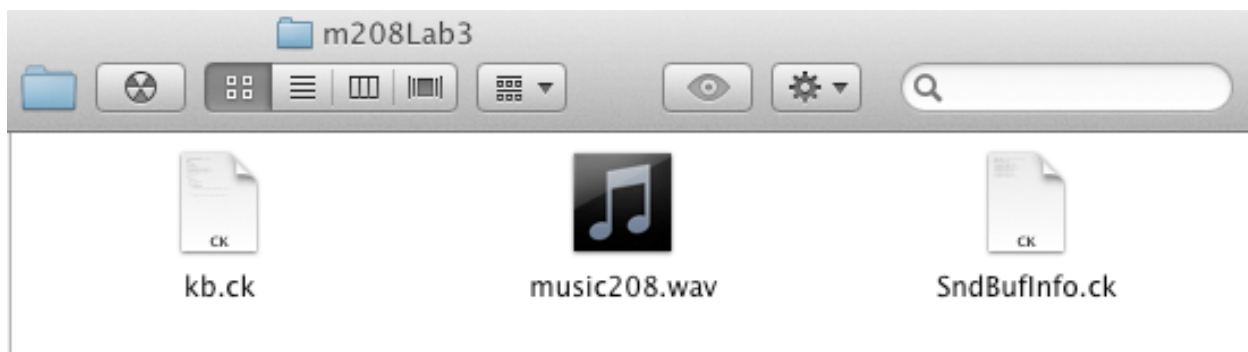
Download and unzip m208Lab3.zip to create a m208Lab3 folder. Copy it to one of these locations:

Mac: Desktop folder

Win: C:\m208\m208Lab3 folder

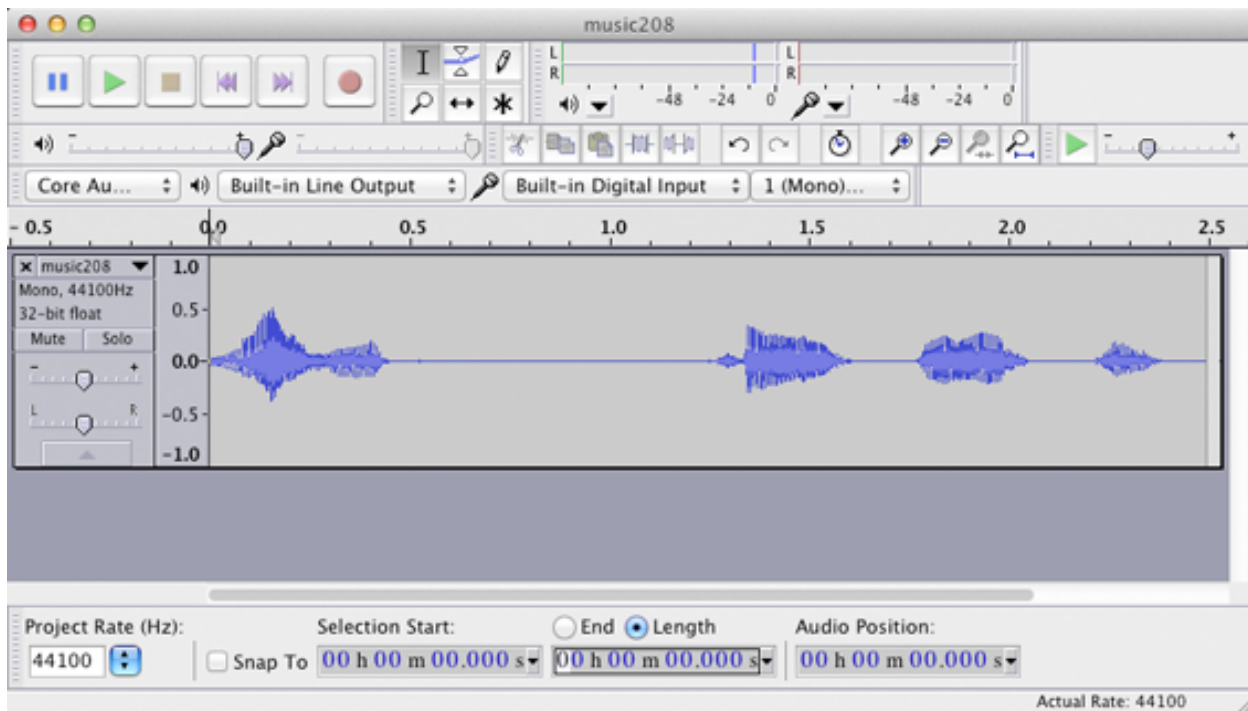
Directions in this lab refer to the folder named m208Lab3 and assume it's in one of these two locations. If you use a different location you'll have to adapt future directions to your pathname.

The m208Lab3 folder should contain these files:



Play The music208.wav File In Audacity

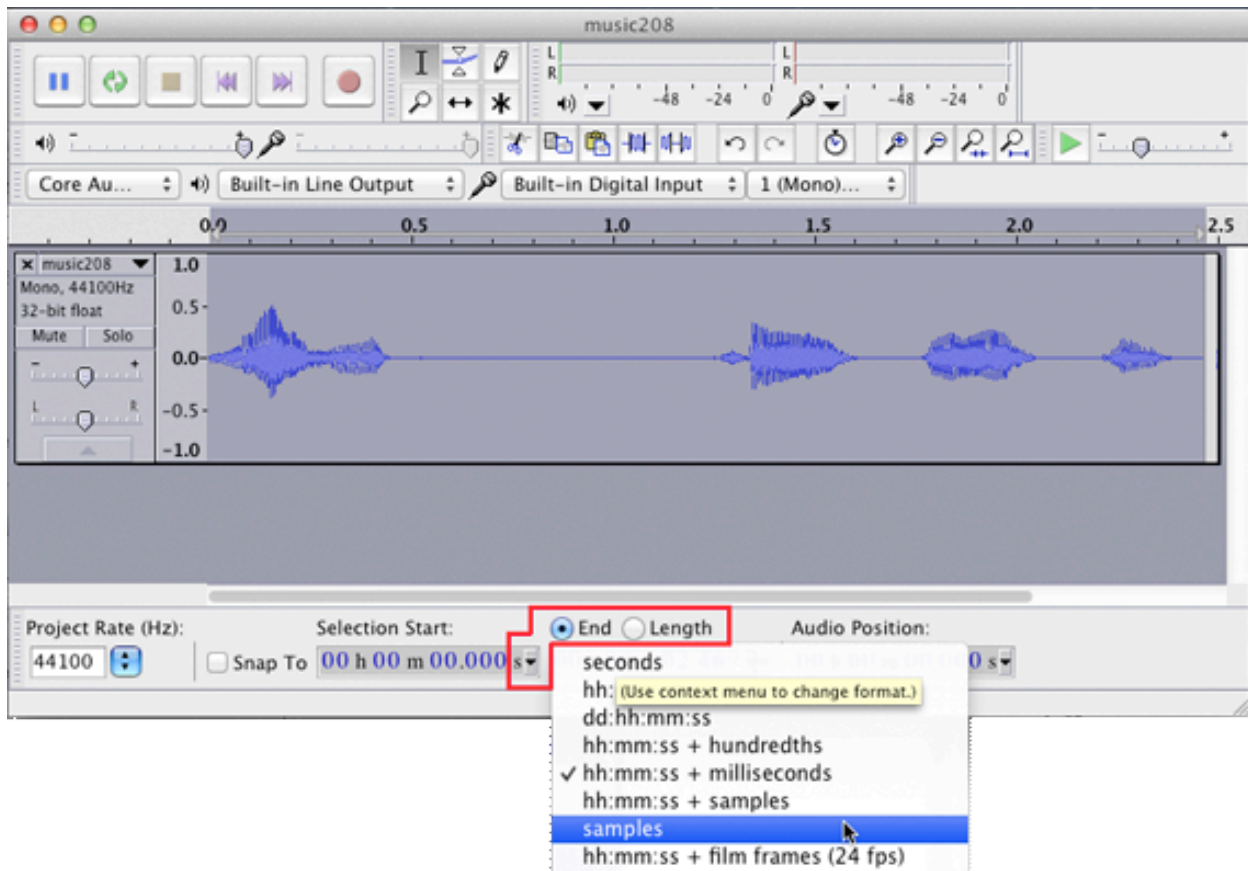
Open /Applications/Audacity and then open the music208.wav file and play it.



Lab 3 will play, manipulate, and mangle sounds in this audio file using Octave and ChuckK. To get started you'll need to find the sample starting and ending number for each of the four words in the file, as well as the and total number of samples in the whole file.

Find The Start, End, And Length Of The Entire File

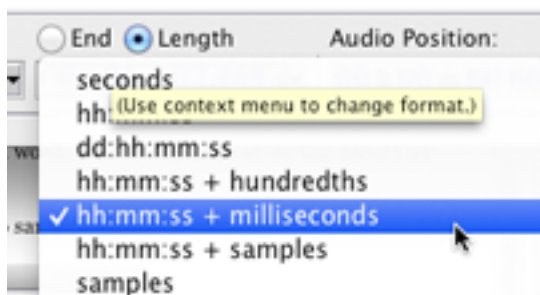
Select the entire file, turn on the End radio button, and set the popup menu to display samples.

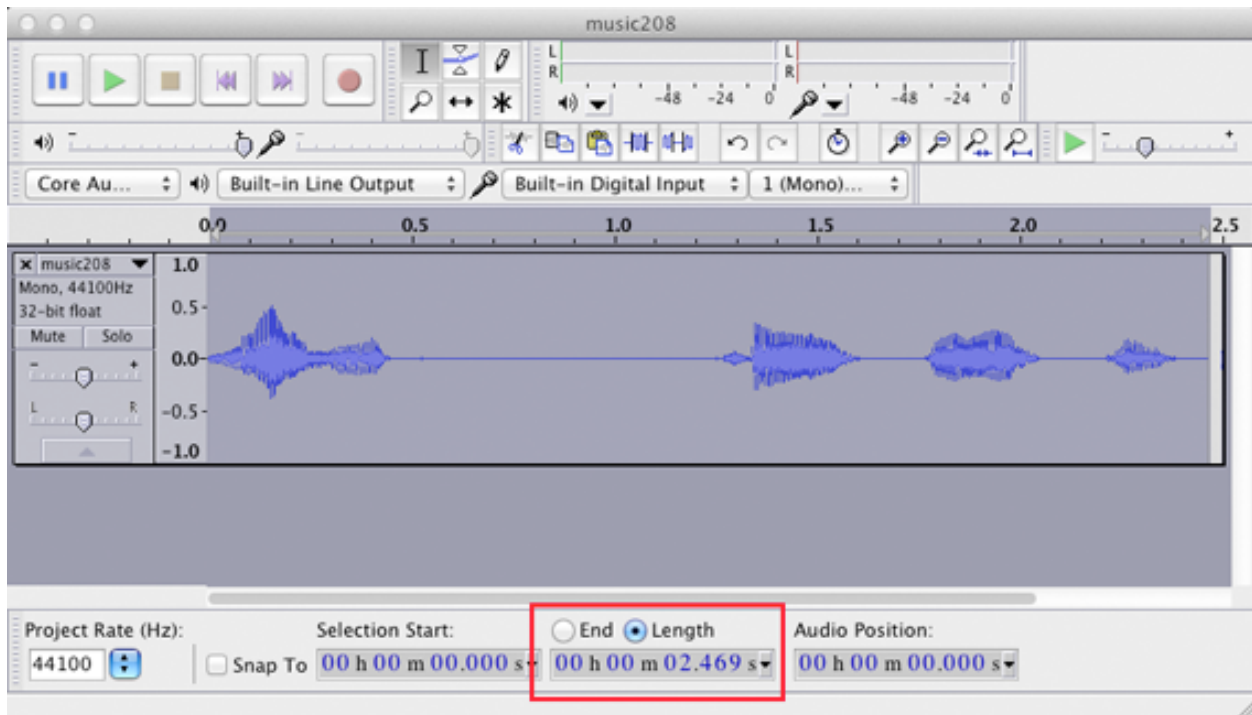


You should see that the audio starts at sample zero and ends at sample number 108,861.

Knowing the sample rate is 44100 samples per second means every sample lasts 22.676 μ s. If you know the sample number start and end times you can calculate length of that selection in seconds. Audacity will do that for you.

Turn on the Length radio button and set the popup menu to display hours, minutes, seconds + milliseconds.





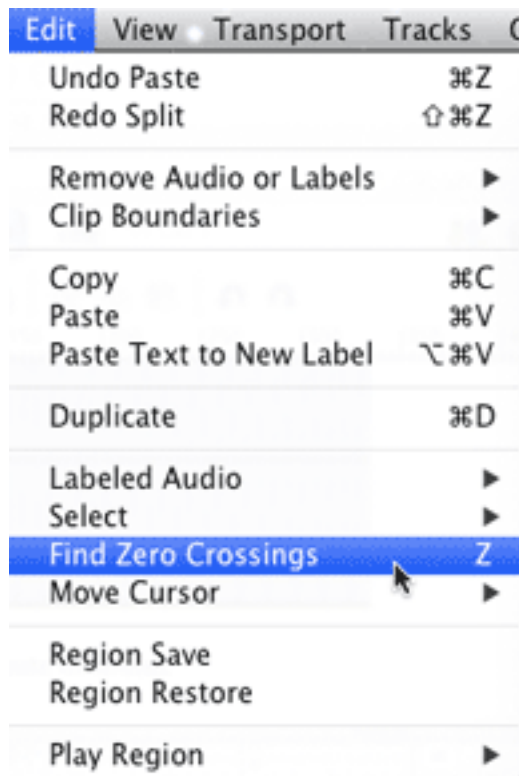
You should see that the duration of the entire file is 0 hours, 0 minutes and 2.469 seconds long

Clicks On Playback

Find Zero Crossings

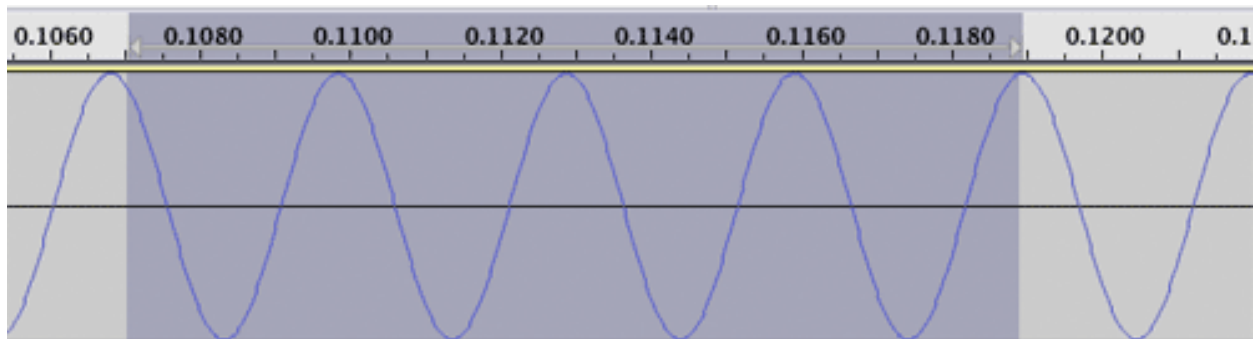
A zero crossing occurs when the amplitude of a sound file crosses the x axis (time) at which point its amplitude is zero. An amplitude of zero produces no sound. Playback clicks are caused by amplitude discontinuities when audio segments join together at different amplitudes. Audio segments that join together at zero crossings are free from clicks on playback. You should always use the Find Zero Crossings command in Audacity when locating segment boundaries and copying audio segments.

Shortcut: Select any section of audio and type Z

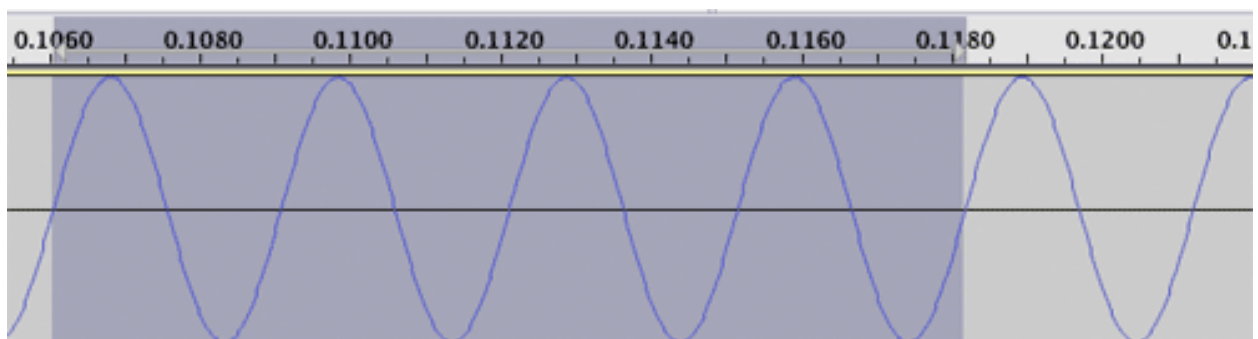


Close Up of Find Zero Crossings

Before



After



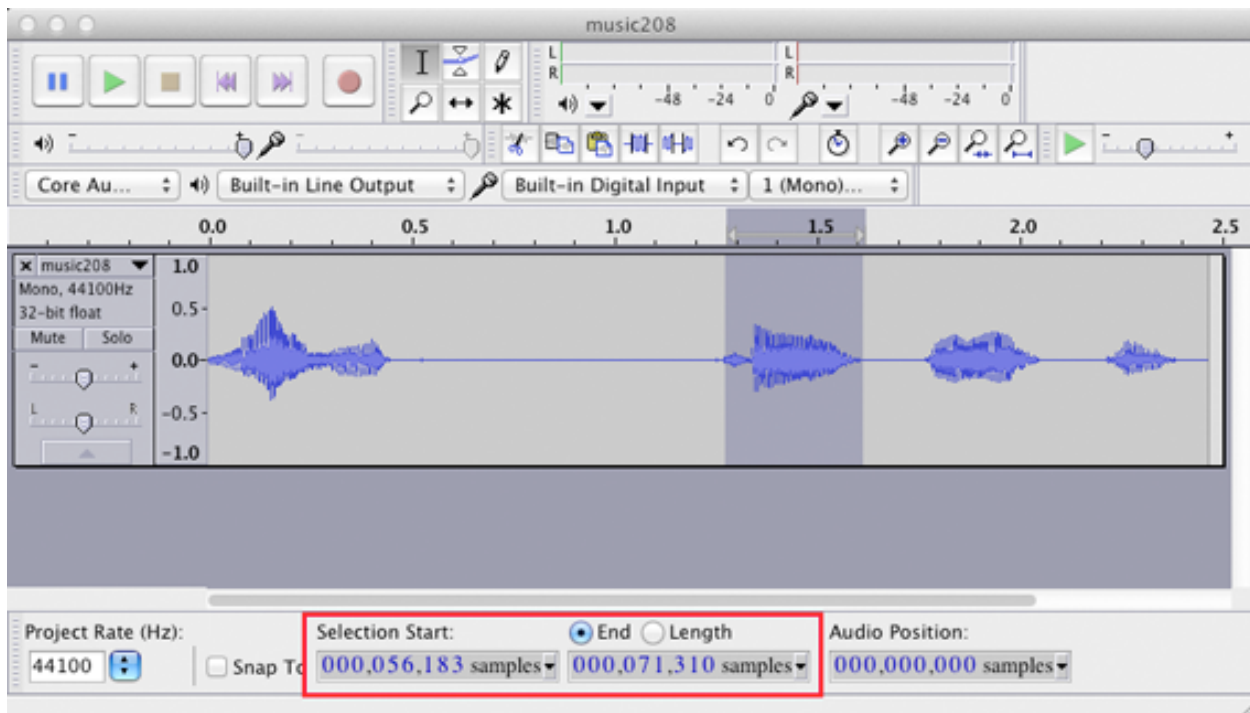
Complete This Table Of Start Times, End Times, And Length

You don't have to be rigorously exact, one sample is only 22+ μ s long. Save your results in a TextWrangler document so you can copy/paste them later.

Segment	Start time in samples	End time in samples	Length in milliseconds
Music 208 (entire file)	0	108,861	2469
Music			
Two			
Oh			
Eight			

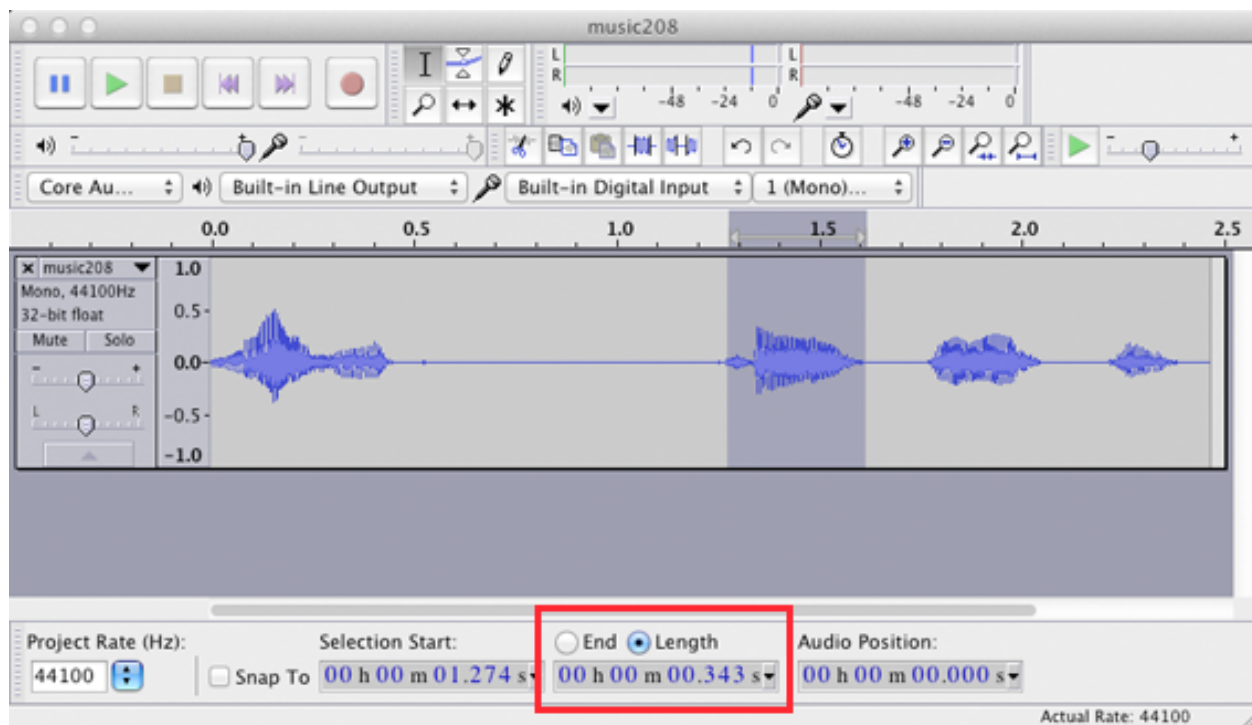
Find The Start and End Samples For The Word Two

Make sure the End radio button is selected. Select the word two and press the spacebar to play it. Position the cursor near either edge of the selection. When the cursor changes to a hand you can grab the edge to extend or shorten the selection. Type Z to set zero crossings and then read the start and end times in samples.



Find The Length In Milliseconds For The Word Two

Make sure the Length radio button is selected. Set the popup menu to "hh:mm:ss + milliseconds" and read the length.



Find the start and end times for "music", "oh", and "eight".

Make sure you've saved the times in a text file so you can copy/paste them later.

We're done with Audacity. Close and quit.

Play The Music208.Wav File In Octave

Start Octave

Start Octave and set the working directory to the m208Lab3 folder.

Set the working directory

```
cd FULL_PATH_NAME_OF_m208Lab3_FOLDER
```

The full pathname of the m208Lab3 folder will be similar to one of these examples:

Mac: /Users/labuser/Desktop/m208Lab3

The tilde (~) expands to the pathname of your home folder.

Win: C:\m208\m208Lab3

pwd

Type pwd to verify your Octave working directory is set correctly.

ls (Mac) or dir (Win)

Type ls (LiSt files) or dir to make sure Octave can find the music208.wav file. If music208.wav does not show appear, you need to fix it before proceeding.

wavread and wavwrite

We used the wavwrite function in Lab2 to save an array of samples to a .wav file. The wavread function reads the samples found in a .wav file into an array that can be played with playsamples. The course [Reference.html](#) page has links to the WAV file format.

help wavread

Type help wavread to find out how this function is used. Of the five versions of the function listed, we'll use the three underlined in red.


```

octave:6> help wavread
'wavread' is a function from the file /usr/local/Cellar/octave/3.6.4/share/
octave/3.6.4/m/audio/wavread.m

-- Function File: Y = wavread (FILENAME)
    Load the RIFF/WAVE sound file FILENAME, and return the samples in
    vector Y.  If the file contains multichannel data, then Y is a
    matrix with the channels represented as columns.

-- Function File: [Y, FS, BPS] = wavread (FILENAME)
    Additionally return the sample rate (FS) in Hz and the number of
    bits per sample (BPS).

-- Function File: [...] = wavread (FILENAME, N)
    Read only the first N samples from each channel.

-- Function File: [...] = wavread (FILENAME, N1 N2)
    Read only samples N1 through N2 from each channel.

-- Function File: [SAMPLES, CHANNELS] = wavread (FILENAME, "size")
    Return the number of samples (N) and channels (CH) instead of the
    audio data.

    See also: wavwrite

```

Functions two and three use [...] for the function return value. This means repeat the parameters of the function listed above. Don't use function 1 because Octave defaults to a sample rate of 8000 and we're using 44100.

File Information

Use the third underlined function to get information about the file.

```

octave:93> [samples, channels] = wavread('music208.wav', "size")
samples =

    108861         1

channels =  44100

```

The Octave help file is misleading. The return value [samples, channels] would seem to indicate that the function will return a two element answer. What really happens is that samples is a two element array where samples(1) is the number of samples in the file and samples(2) is the number of channels. Channels is the sample rate. A mono file will have one channel and a stereo file will have two channels.

```
octave:97> samples(1)
ans = 108861
octave:98> samples(2)
ans = 1
octave:99> channels
channels = 44100
```

whos

You can verify the return value sizes with the who's command.

```
octave:96> whos
Variables in the current scope:
```

Attr	Name	Size	Bytes	Class
====	====	====	=====	=====
	ans	1x1	8	double
	channels	1x1	8	double
	samples	1x2	16	double

Total is 4 elements using 32 bytes

[wav, FS, BITS] = wavread('music208.wav')

The first underlined function is the one you'll typically use.

```
octave:106> clear all;
octave:107> [wav, FS, BITS] = wavread('music208.wav');
octave:108> whos
Variables in the current scope:
```

Attr	Name	Size	Bytes	Class
====	====	====	=====	=====
	BITS	1x1	8	double
	FS	1x1	8	double
	wav	108861x1	870888	double

Total is 108863 elements using 870904 bytes

The wav array contains the samples, FS (Frequency of Sampling) is the sample rate, and BITS is number of bits used to store the amplitude, in this case 2^{16} . A bit depth of 16 can range from 0-65535, or from -32768-32767. In wav files these values are normalized to fall within a range of -1.0 to +1.0.

```
octave:111> wav(1:10)
```

```
ans =
```

```
-0.0098267
-0.0096741
-0.0093079
-0.0092163
-0.0087585
-0.0086975
-0.0082703
-0.0080261
-0.0078125
-0.0074158
```

```
octave:112> FS
```

```
FS = 44100
```

```
octave:113> BITS
```

```
BITS = 16
```

Play the music208.wav File

We can use the returned samples in the wav array to play the file.

```
octave:114> playsamples( wav );
```

Play The Word Two

You can use the second underlined function to play a section of wav file, in this example the word "two".

Here's what the help file states:

```
-- Function File: [...] = wavread (FILENAME, N1 N2)  
    Read only samples N1 through N2 from each channel.
```

The function return value [...] indicates that the return value is the same as the one for the function listed above. Substituting [wav, FS, BITS] for [...] and calling the function as shown in the help file results in a syntax error.

```
octave-3.4.0:29> [wav, FS, BITS] = wavread("music208.wav", 55867 71669);  
parse error:  
  
    syntax error  
  
>>> [wav, FS, BITS] = wavread("music208.wav", 55867 71669);  
                                     ^
```

Again the Octave help was misleading. You need to enclose N1 N2 in brackets.

```
octave:115> [wav, FS, BITS] = wavread('music208.wav', [55867 71669] );  
octave:116> playsamples( wav );
```

We're done with Octave. Type exit at the Octave prompt and quit Terminal.

The Chuck SndBuf Object

SndBuf

The SndBuf object is the counterpart to Octave's wavread and wavwrite methods. In ChuckK terminology, SndBuf is a ugen (Unit Generator), a class that generates or modifies sound. A class is a ChuckK object that encapsulates data (in this case the samples in sound file) and contains methods (functions) that operate on those samples. Here's the ChuckK documentation for SndBuf. http://chuck.cs.princeton.edu/doc/program/ugen_full.html

[ugen]: SndBuf

- *sound buffer (now interpolating)*
- *reads from a variety of file formats*
- *see examples: [sndbuf.ck](#)*

(control parameters)

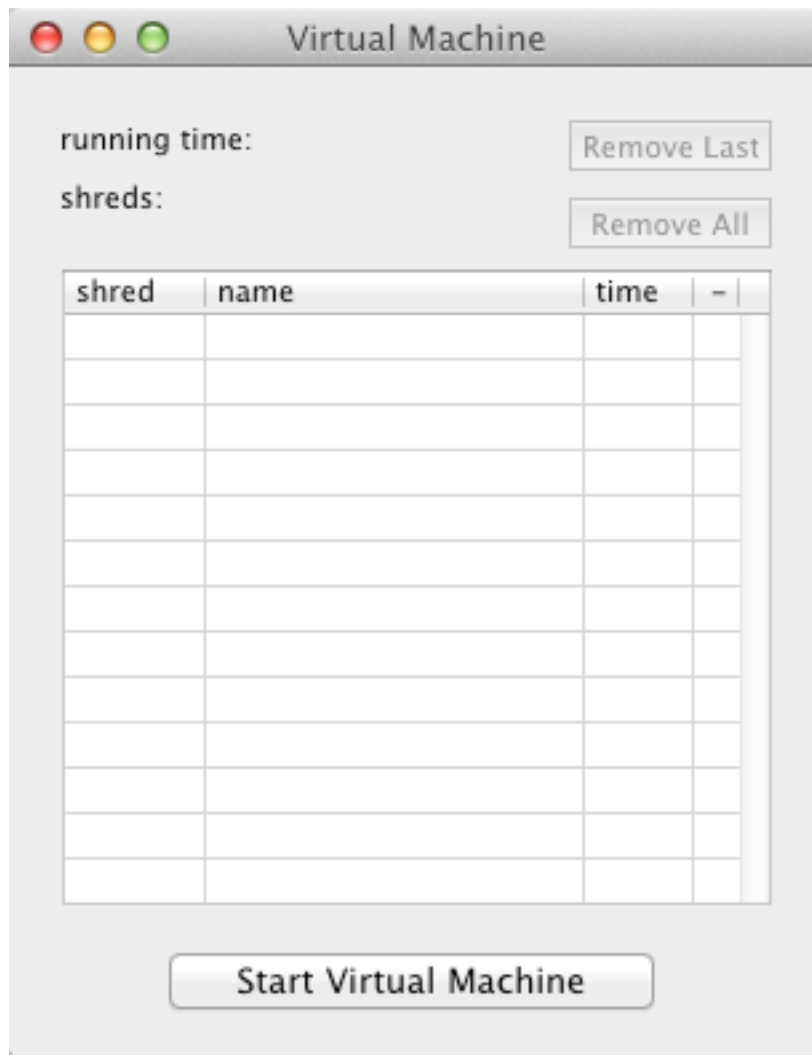
- **.read** - (string , WRITE only) - *loads file for reading*
- **.chunks** - (int , READ/WRITE) - *size of chunk (# of frames) to read on-demand; 0 implies entire file, default; must be set before reading to take effect.*
- **.samples** - (int , READ only) - *get number of samples*
- **.length** - (dur , READ only) - *get length as duration*
- **.channels** - (int , READ only) - *get number of channels*
- **.pos** - (int , READ/WRITE) - *set position (0 < p < .samples)*
- **.rate** - (float , READ/WRITE) - *set/get playback rate (relative to file's natural speed)*
- **.interp** - (int , READ/WRITE) - *set/get interpolation (0=drop, 1=linear, 2=sinc)*
- **.loop** - (int , READ/WRITE) - *toggle looping*
- **.freq** - (float , READ/WRITE) - *set/get loop rate (file loops / second)*
- **.phase** - (float , READ/WRITE) - *set/get phase position (0-1)*
- **.channel** - (int , READ/WRITE) - *sel/get channel (0 < p < .channels)*
- **.phaseOffset** - (float , READ/WRITE) - *set/get a phase offset*
- **.write** - (string , WRITE only) - *loads a file for writing (or not)*

SndBufInfo.ck

Open /Applications/miniAudicle. Open SndBufInfo.ck in the m208Lab3 folder in miniAudicle. I wrote SndBufInfo.ck to return information about the music208.wav file.

Start Chuck's Virtual Machine

Click the Start Virtual Machine button in the Virtual Machine window.

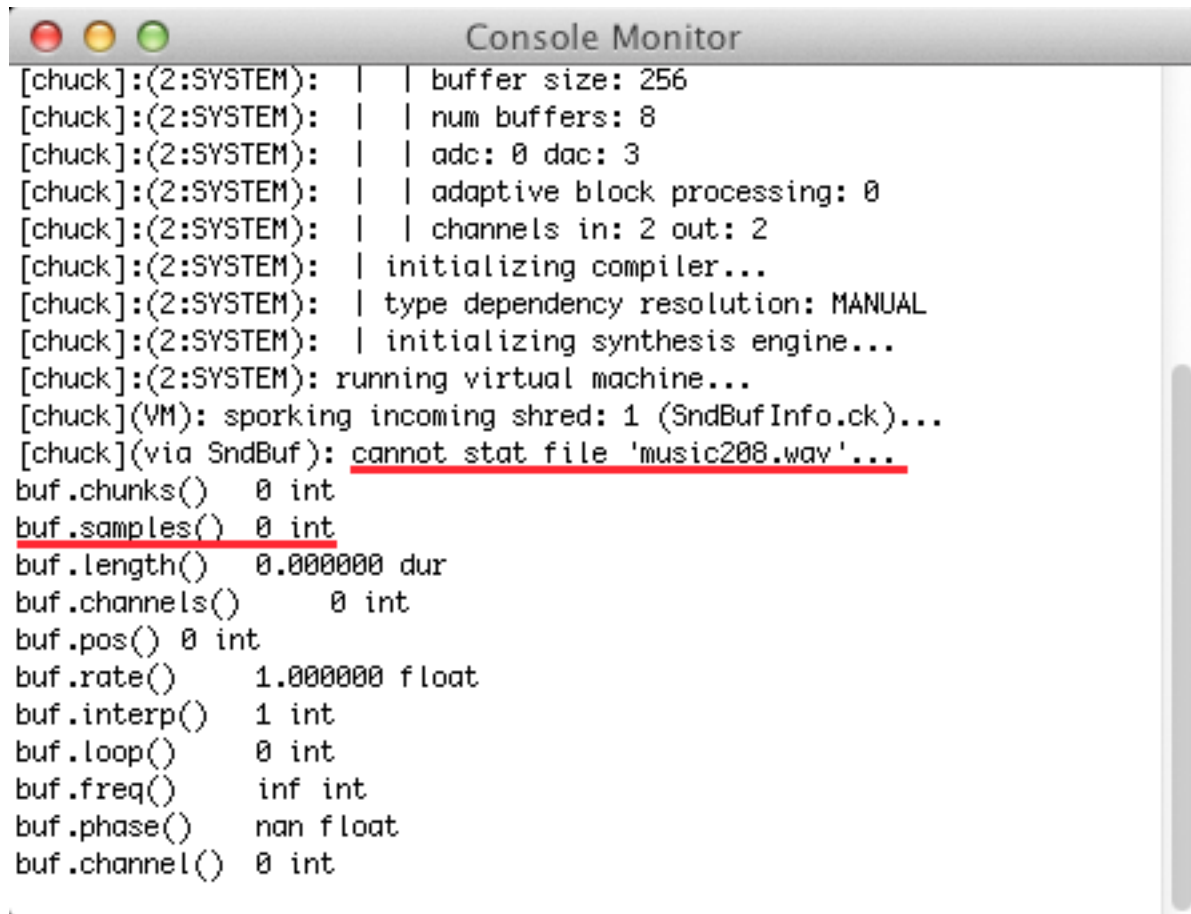


Add Shred

Click the green plus sign (Add Shred) in the miniAudicle window to execute the ChuckK code. Errors and text output will appear in the Console Monitor window.

Houston, We've Got A Problem

The Console Monitor Window reports an error: "cannot sat file 'musc208.wav' means ChuckK couldn't find the music208.wav file. All reported data values are wrong.



```

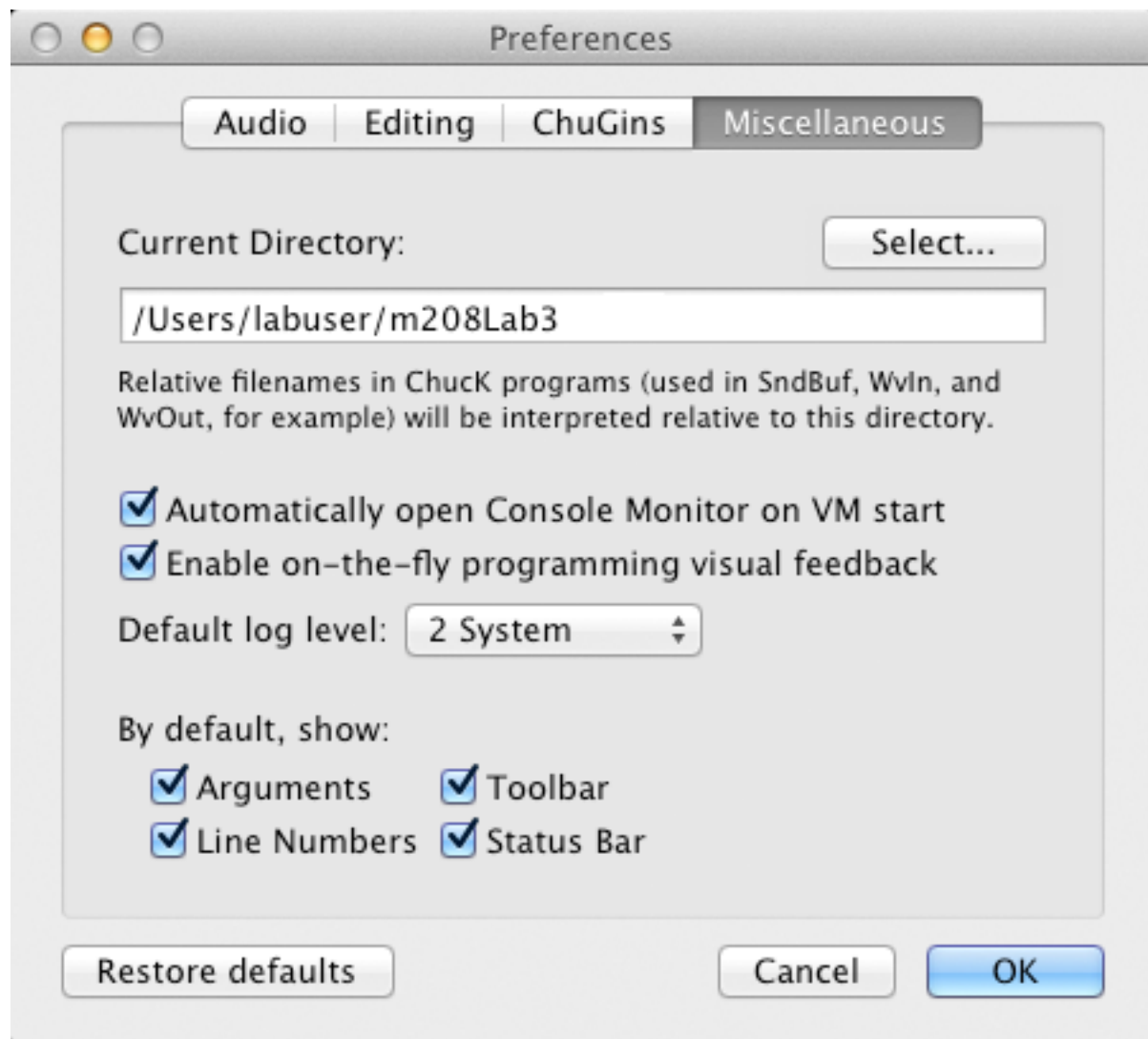
[chuck]:(2:SYSTEM): | | buffer size: 256
[chuck]:(2:SYSTEM): | | num buffers: 8
[chuck]:(2:SYSTEM): | | adc: 0 dac: 3
[chuck]:(2:SYSTEM): | | adaptive block processing: 0
[chuck]:(2:SYSTEM): | | channels in: 2 out: 2
[chuck]:(2:SYSTEM): | initializing compiler...
[chuck]:(2:SYSTEM): | type dependency resolution: MANUAL
[chuck]:(2:SYSTEM): | initializing synthesis engine...
[chuck]:(2:SYSTEM): running virtual machine...
[chuck](VM): sporking incoming shred: 1 (SndBufInfo.ck)...
[chuck](via SndBuf): cannot stat file 'music208.wav'...
buf.chunks() 0 int
buf.samples() 0 int
buf.length() 0.000000 dur
buf.channels() 0 int
buf.pos() 0 int
buf.rate() 1.000000 float
buf.interp() 1 int
buf.loop() 0 int
buf.freq() inf int
buf.phase() nan float
buf.channel() 0 int

```

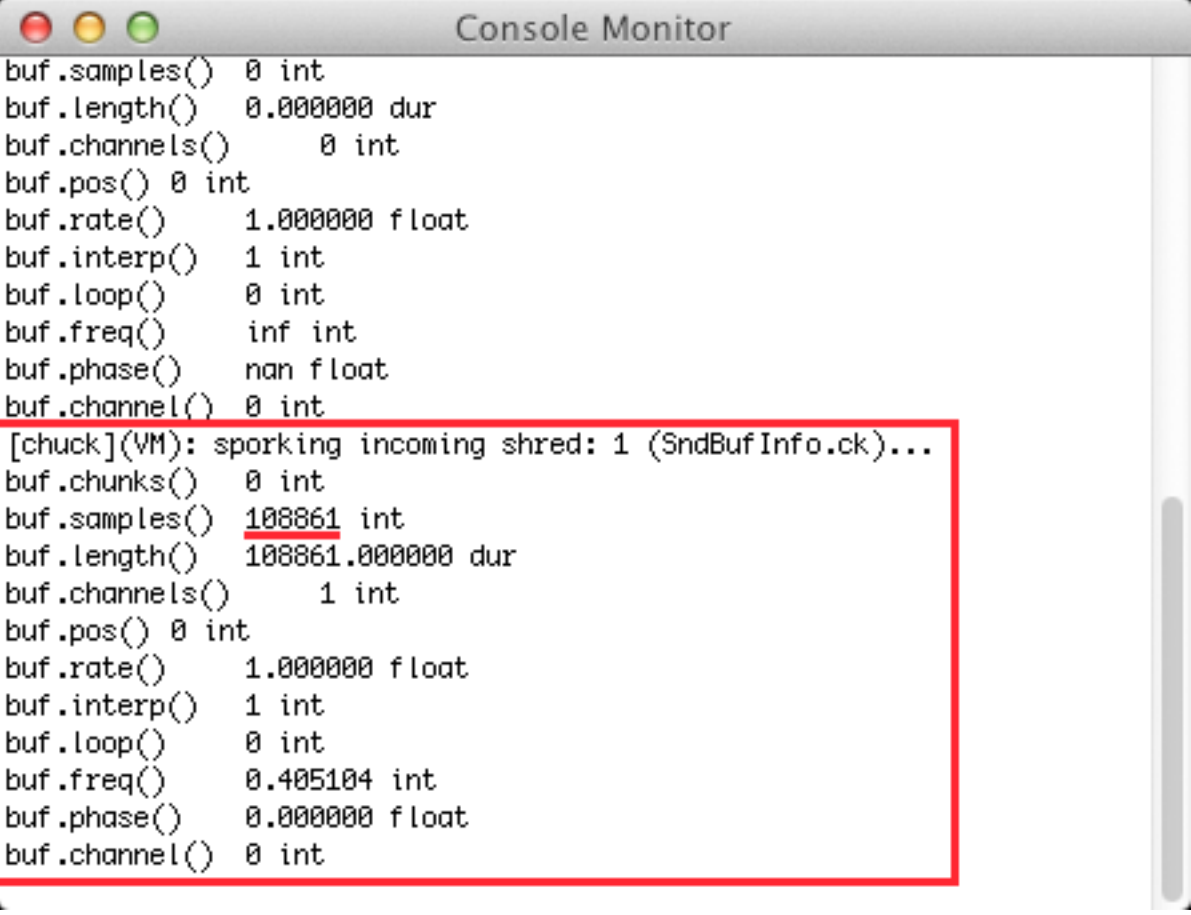
It's a working directory problem. You need to define the directory where miniAudicle searches for files.

Set the miniAudicle Working Directory

Choose Preferences from the miniAudicle menu. Select the Miscellaneous tab, click the Select button and choose the m208Lab3 folder on the Desktop.



Run the program again and the correct values should appear.



```
Console Monitor
buf.samples() 0 int
buf.length() 0.000000 dur
buf.channels() 0 int
buf.pos() 0 int
buf.rate() 1.000000 float
buf.interp() 1 int
buf.loop() 0 int
buf.freq() inf int
buf.phase() nan float
buf.channel() 0 int
[chuck](VM): sporking incoming shred: 1 (SndBufInfo.ck)...
buf.chunks() 0 int
buf.samples() 108861 int
buf.length() 108861.000000 dur
buf.channels() 1 int
buf.pos() 0 int
buf.rate() 1.000000 float
buf.interp() 1 int
buf.loop() 0 int
buf.freq() 0.405104 int
buf.phase() 0.000000 float
buf.channel() 0 int
```

Line By Line Code Commentary

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => buf.read;
4
5 /* same result as line 3
6 buf.read( "music208.wav";
7 */
8
9 <<< "buf.chunks()\t", buf.chunks(), "int" >>>;
10 <<< "buf.samples()\t", buf.samples(), "int" >>>;
11 <<< "buf.length()\t", buf.length(), "dur" >>>;
12 <<< "buf.channels()\t", buf.channels(), "int" >>>;
13 <<< "buf.pos()\t", buf.pos(), "int" >>>;
14 <<< "buf.rate()\t", buf.rate(), "float" >>>;
15 <<< "buf.interp()\t", buf.interp(), "int" >>>;
16 <<< "buf.loop()\t", buf.loop(), "int" >>>;
17 <<< "buf.freq()\t", buf.freq(), "int" >>>;
18 <<< "buf.phase()\t", buf.phase(), "float" >>>;
19 <<< "buf.channel()\t", buf.channel(), "int" >>>;

```

Line 1.

Two forward slashes indicate a single line comment. They can appear anywhere on within a line and all text to the right of the // will be ignored.

Line 2.

A SndBuf object reserves an area of memory to hold the data and functions known to the SndBuf class. The variable called buf represents one particular instance of a SndBuf object. Your code will call SndBuf functions and access SndBuf data through the buf variable. The buf variable is then chunked to dac, the speaker.

Once buf is created you can call any of the SndBuf methods by adding a period after buf followed by the method name. For example buf.samples() will tell us how many discrete samples are in the file and buf.length() will tell us how long the sound lasts.

Loading A Soundfile

Line 3.

The SndBuf class contains a method called read. When you want to load a sound file you call the read method in one of two ways:

```
"name_of_sound_file" => buf.read;
or
buf.read( "name_of_sound_file" );
```

Now that the soundfile has been chucked to buf.read, the other SndBuf methods have data to work with.

Multiline Comments

Lines 5-7.

Multiline comments are enclosed between `/*` and `*/`. Lines 5-7 show an alternative way of reading a sound file into but.

Debug Print Statements

Lines 9-19.

Triple angle brackets are used to print the values of variables for debugging purposes. The output will appear in the Console Monitor window. You can freely mix text and values separated by commas between the brackets. Text enclosed in quotes will appear in the printout, values will be calculated and the result will be displayed in the Console Monitor window. Return characters are automatically added at the end of each `<<< ... >>>` element.

\t, \n

The `\t` symbol prints a tab character to the Console Monitor window. Similarly the `\n` symbol prints a new line (return character), although it's not used here.

Types

The SndBuf object contains four variable types: int, float, dur, and string.

int - an integer, no decimal places

float - a number with decimal places

dur - a number with decimal places that represents time (time and dur will be discussed later)

string - text enclosed in quotation marks

READ/WRITE

SndBuf methods are indicated as WRITE only, READ only, or READ/WRITE. The SndBuf rate method which we'll use later in this lab is explained like this.

.rate - (float , READ/WRITE) - *set/get playback rate (relative to file's natural speed)*

READ Method Syntax

READ methods are associated with the word get. When you want to "get" a value you use the READ syntax. READ statements appear to the left of the ChuckK operator => end with parentheses, ().

```
// READ or get the current rate and chuck it to myRate
buf.rate() => float myRate;
```

WRITE Method Syntax

WRITE methods are associated with the word set. When you want to "set" a value you use the WRITE syntax. The two equivalent forms of a WRITE statement are shown below. The choice is yours.

```
// Option 1. The value is on the left side of the
// Chuck operator => and the WRITE method does not
// end with parentheses, ()
1.67 => buf.rate;
```

```
// Option 2. The value to WRITE appears inside the
// parentheses () and the Chuck operator => is not used
buf.rate( 1.67 );
```

Play the music208.wav File

Now that ChuckK has read in the data from the music208.wav file, you can play it.

In the lab examples you do not have to enter any lines beginning with two forward slashes //. ChuckK knows these lines are comments and will ignore them. The lab code comments are used to help explain what's happening. Your own code (especially in Homework and Projects) should use copious and meaningful comments. They help others (like your teacher) understand your code and they will help you remember what you were trying to do when you review the code in the future. Always choose names for functions and variables that are self documenting, and not short and obscure. Typing is cheap.

Enter this code and run the program.

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => buf.read;
4
5 // set the starting point in samples
6 0 => buf.pos;
7 // chuck the files duration to now
8 buf.length() => now;
9

```

Play the Word "Two"

The word two starts at sample 55867 and is 343 ms long.

Type comments at the beginning of lines 6 and 8.

Add lines 11-12 and run it.

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => buf.read;
4
5 // set the starting point in samples
6 // 0 => buf.pos;
7 // chuck the files duration to now
8 // buf.length() => now;
9
10 // Play the word two
11 56183 => buf.pos;
12 343::ms => now;
13

```

Play The Word "Tutu"

Change lines 10-19 as follows:

```
1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => buf.read;
4
5 // set the starting point in samples
6 // 0 => buf.pos;
7 // chunk the files duration to now
8 // buf.length() => now;
9
10 // Play the word "tutu"
11 1.0 => buf.gain;
12 56183 => buf.pos;
13 343::ms => now;
14 // again
15 .6 => buf.gain;
16 56183 => buf.pos;
17 343::ms => now;
18 // restore gain
19 1.0 => buf.gain;
20
```

Notice how the gain was lowered for the second syllable and was restored at the end.

Chuck Time and Duration

Portions of text from: <http://chuck.cs.princeton.edu/doc/language/time.html>

Time and duration are native types in ChuckK.

time represents an absolute point in time (from the beginning of ChuckK time).

dur represents a duration (with the same logical units as **time**).

By default, ChuckK provides these preset duration values:

- **samp** : duration of 1 sample in ChuckK time
- **ms** : duration of 1 millisecond
- **second** : duration of 1 second
- **minute** : 1 minute
- **hour** : 1 hour
- **day** : 1 day
- **week** : 1 week

ChuckK can perform many time and duration calculations.

Find the Sample Rate and Period

Open a new file, enter this code, save it as TimeTests.ck, and run.

```

1 // <<< ... >>>; appear in the Console Monitor window
2 // \n is line feed or print blank line
3 <<< "\n" >>>;
4 <<< "== Sample Rate and Sample Period =====>>>;
5 <<< "sampleRate = 1::second/1::samp =", 1::second/1::samp, "Hz" >>>;
6 <<< "samplePeriod = 1::samp/1::second =", 1::samp/1::second, "seconds" >>>;
7
```

Output in the Console Monitor window.

```

[chuck](VM): sporking incoming shred: 1 (timeTest1.ck)...
"
" : (string)
"== Sample Rate and Sample Period =====" : (string)
sampleRate = 1::second/1::samp = 44100.000000 Hz
samplePeriod = 1::samp/1::second = 0.000023 seconds

```

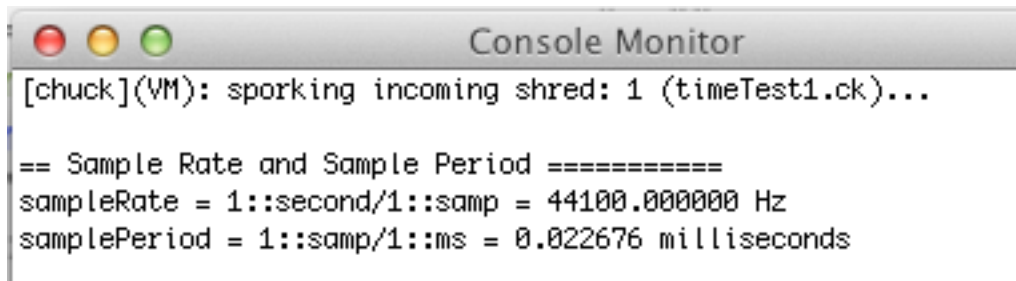
To prevent octave types like (string) from being printed modify lines 3-4 like this. Then delete original line 2, change line 5 to display period in milliseconds, and add line 6. Save and run this code.

```

1 // <<< ... >>>; appears in the Console Monitor window
2 <<< "", "" >>>;
3 <<< "== Sample Rate and Sample Period =====", "" >>>;
4 <<< "sampleRate = 1::second/1::samp =", 1::second/1::samp, "Hz" >>>;
5 <<< "samplePeriod = 1::samp/1::ms =", 1::samp/1::ms, "milliseconds" >>>;
6 <<< "", "" >>>;
7

```

Notice how the empty string "" suppresses the type (string) output.



```

[chuck](VM): sporking incoming shred: 1 (timeTest1.ck)...

== Sample Rate and Sample Period =====
sampleRate = 1::second/1::samp = 44100.000000 Hz
samplePeriod = 1::samp/1::ms = 0.022676 milliseconds

```

Convert Times

Chuck can convert times between samples, milliseconds, seconds, minutes, hours, and days. Append this code, save, and run.

```

8 // Let's find out how long the music208.wav file lasts
9 <<< "The musc208.wav file = 108,861 samples =====", "" >>>;
10 <<< "108861 samples =", 108861::samp / 1::ms, "ms" >>>;
11 <<< "                    =", 108861::samp / 1::second, "seconds" >>>;
12 <<< "                    =", 108861::samp / 1::minute, "minutes" >>>;
13 <<< "                    =", 108861::samp / 1::hour, "hours" >>>;
14 <<< "                    =", 108861::samp / 1::day, "days" >>>;
15 <<< "", "" >>>;
16

```

```

The musc208.wav file = 108,861 samples =====
108861 samples = 2468.503401 ms
                  = 2.468503 seconds
                  = 0.041142 minutes
                  = 0.000686 hours
                  = 0.000029 days

```


Time Math

Chuck tracks time at the sample level. Append this code, save, and run.

```

17 <<< "Time Math =====", "" >>>;
18 // Now lets do some time math
19 <<< "1::samp =", 1::samp, "samples", "" >>>;
20 <<< "1::ms =", 1::ms, "samples" >>>;
21 <<< "1::second =", 1::second, "samples" >>>;
22 <<< "1::minute =", 1::minute, "samples" >>>;
23 <<< "3 minutes 37 seconds =", 3::minute + 37::second, "samples" >>>;
24 <<< "1::hour =", 1::hour, "samples" >>>;
25 <<< "", "" >>>;
26

```

```

Time Math =====
1::samp = 1.000000 samples
1::ms = 44.100000 samples
1::second = 44100.000000 samples
1::minute = 2646000.000000 samples
3 minutes 37 seconds = 9569700.000000 samples
1::hour = 158760000.000000 samples

```

Fractions of a Sample

Chuck can track even time to the fraction of a sample. Append this code, save, and run.

```

27 <<< "Fractions of a sample =====", "" >>>;
28 // the value of now is accurate to fractions of a sample
29 <<< "nowStart: ", now >>>;
30 0.00001::samp => now;
31 <<< "time advanced by 0.00001::samp = ", now >>>;
32 <<< "", "" >>>;
33

```

```

Fractions of a sample =====
nowStart: 292631552.000000
time advanced by 0.00001::samp = 292631552.000010

```

Elapsed Time

Append this code, save, and run.

```

34 <<< "Process audio for 3.001 seconds =====", "" >>>;
35 // Get time now at start of this portion of the code
36 now => time nowStart;
37 <<< "nowStart:", nowStart >>>;
38 SinOsc s => dac;
39 440 => s.freq;
40 0.05 => s.gain;
41 3.001::second => dur later;
42 // add a duration to a time ("now" is time in samples).
43 now + later => now;
44 // subtracting one time from another time is of type dur (duration)
45 now - nowStart => dur elapsed;
46 <<< "nowEnd:", now, "\nelapsed samples:", elapsed >>>;
47 <<< "elapsed time:", elapsed/1::second, "seconds" >>>;

```

```

Process audio for three seconds =====
nowStart: 292631552.000010
nowEnd: 292763896.100010
elapsed samples: 132344.100000
elapsed time: 3.001000 seconds

```

Chuck Control Structures

Portions of text from: <http://chuck.cs.princeton.edu/doc/language/ctrl.html#while>

Chuck includes standard control structures similar to those in most programming languages. A logic condition (true, false) is evaluated. A block is potentially executed based on whether the condition evaluates to true or false. Blocks are separated either by semicolons or by curly brackets {}.

True and False in Chuck

The Chuck reserved words true and false are represented by integers. True is 1 and false is 0. Open a new miniAudicle window, enter and run this code.

```
1 <<< "1 < 5 evaluates to\t", 1 < 5 >>>;
2 <<< "10 < 5 evaluates to\t", 10 < 5 >>>;
3 <<< "10 == 10 evaluates to\t", 10 == 10 >>>;
4 <<< "10 == 10.00000001 evaluates to\t", 10 == 10.00000001 >>>;
5 <<< "strings: sam == sam", "sam" == "sam" >>>;
6 <<< "strings: Sam == sam", "Sam" == "sam" >>>;
```

```
1 < 5 evaluates to 1
10 < 5 evaluates to 0
10 == 10 evaluates to 1
10 == 10.00000001 evaluates to 0
strings: sam == sam 1
strings: Sam == sam 0
```

while

The while statement body (code between the curly brackets) executes repeatedly as long as the while condition evaluates to true. Open a new miniAudicle window, enter, save, and run this code.

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => string filename;
4 filename => buf.read;
5 |
6 // Play the word Two five times
7 0 => int count;
8 while (count < 5)
9 {
10     <<< "count:", count >>>;
11     56183 => buf.pos;
12     343::ms => now;
13     count + 1 => count;
14 }
15 <<< "count after while loop terminates:", count >>>;

```

Line 7

Define an integer variable named count and set its value to 0.

Line 8

Repeatedly execute the while loop as long as the value of count is less than 5. Because count was initialized to zero in the preceding statement, the loop will execute.

Line 9

The opening { marks the beginning of the block of statements to be executed.

Line 10

Print the value of count.

Line 11

Set the sample index number to sample 56183 and start playing from there.

Line 12

Play audio for 343 milliseconds.

Line 13

Add one to the value of count;

Line 14

The closing } marks the end of block of statements to be executed. Program flow returns to line 9 where the value of count will be evaluated again before processing the block of

statements between { and }. When count equals 5 the while loop ends and the next line is executed.

Line 15

Print the value of count after the while loop has terminated.

Directions for All Following Examples

Lines 1-5 (above) remain the same. Delete all lines from 6 to end and replace with following examples.

break / continue

Break allows the program flow to jump out of a loop. Enter and run this code.

```
// Play the word Two five times
0 => int count;
while (true) // repeat forever
{
    56183 => buf.pos;
    // extend the length by 250 milliseconds
    343::ms + 250::ms => now;
    count + 1 => count;
    // stop when count reaches 5
    if (count > 4)
        break;
}
```

The while loop will first check the condition, and executes the body as long as the condition evaluates as non-zero.

Line 9

The while (true) block will repeat forever. It's a very common idiom in ChuckK. The only way to terminate is with a conditional break statement.

do / while

To execute the body of the loop before checking the condition, you can use a do/while loop. This guarantees that the body gets executed at least once. Execute this code.

```
// Play oh-eight
0 => int count;
do
{
    // half second past the beginning of "two" in samples
    56183 + 22050 => buf.pos;
    // extend the length by 0.9 seconds
    343::ms + 0.9::second => now;
    // shortcut for count + 1 => count;
    count++;
} while (count < 5);
```

The Chuck cast operator \$

The cast operator is used when you need to convert a variable of one type to a different type. For example you may need to convert a float to an int, or an int to a float, etc.

```
// cast operator
pi $ int => int notpi;
<<< pi, notpi >>>;

6 + 2 => int sum;
sum $ float => float fsum;
<<< sum, fsum >>>;

3.141593 3
8 8.000000
```

until

The until statement is the semantic opposite of while. An until loop executes the body repeatedly until the condition evaluates as non-zero. Enter and run this code.

```
// A kind of "sick" sound
0 => int count;
until (false) //never happens
{
    // 1.025 seconds before "two" in samples
    56183 - (1.025::second / 1::samp) $ int => buf.pos;
    // play for 4/5 second
    900::ms => now;
    count++;
    if (count > 5)
        break;
}
```

`(1.025::second / 1::samp) $ int` calculates how many samples are in 1.025 seconds. Without the case to int, the calculation would produce a float result (number with decimal places). However, the `buf.pos` method requires an int for the sample index position and will cause an error if it's a float.

```
// this calculation results in a float
<<< 1.025::second / 1::samp >>>;

// this calculation results in an int
<<< (1.025::second / 1::samp) $ int >>>;

[chuck](VM): sporking incoming shred: 1 (Untitled)...
45202.500000 :(float)
45202 :(int)
```

The net result of the cast is that ChuckK interprets this line

```
56183 - (1.025::second / 1::samp) $ int => buf.pos;
```

as:

```
56183 - 4502 => buf.pos;
```

do / until

The `[until]` loop will first check the condition, and executes the body as long as the condition evaluates to zero. To execute the body of the loop before checking the condition, you can use a `do/until` loop. This guarantees that the body gets executed at least once. Enter and run this code.

```

// Play eight - two five times
// adjust gain of each word
0 => int count;
do
{
    // 2.1 seconds into the file
    (2.1::second / 1::samp) $ int => buf.pos;
    // make "eight" louder
    2.0 => buf.gain;
    // play for 700 milliseconds
    0.7::second => now;
    // make "two" softer
    0.4 => buf.gain;
    56183 => buf.pos;
    343::ms => now;

    /* METHOD 1
       pause for one second by setting the
       gain to zero so we don't hear anything
    */
    0.0 => buf.gain;
    1::second => now;
    // restore the gain.
    1.0 => buf.gain;

    /* METHOD 2
       pause for one second by setting the
       gain to zero so we don't hear anything
    buf.samples() => buf.pos; // set to end of file
    1::second => now;
    */

    count++;
} until ( count > 4 );

```

Note that there are two methods for playing one second of silence: Method 1 and Method 2. Multi line comments are indicated by `/* ... */`.

Try it again with Method 2 which doesn't involve saving and restoring gain.


```

/* METHOD 1
pause for one second by setting the
gain to zero so we don't hear anything
0.0 => buf.gain;
1::second => now;
// restore the gain.
1.0 => buf.gain;
*/

/* METHOD 2
pause for one second by setting the
gain to zero so we don't hear anything
*/
buf.samples() => buf.pos; // set to end of file
1::second => now;

```

for

A for loop repeats the code body a specified number of times. There are three parameters in every for loop

```
for ( index_variable; condition; adjust_index_variable )
```

The `index_variable` keeps track of the current loop number. The condition tests the state of `index_variable` (true, false). If the condition is true, the index variable is adjusted and the loop body is executed again with the new value of `index_variable`. It is then evaluated and incremented at each iteration. If the condition test is false the loop exits. Enter and run this code.

```
// Decaying buzz 5x
0.01 => float decay;
for ( 0 => int ix; ix < 100; ix++ ) // repeat 100x
{
    // start of "s" in music
    10574 => buf.pos;
    // reduce volume each time through the loop
    buf.gain() - decay => buf.gain;
    // play for 10 milliseconds
    10::ms => now;
}
```

ix++, ix--

ix++ is equivalent to ix + 1

ix-- is equivalent to ix - 1

continue

A continue statement forces the loop to begin a new iteration. Code lines following the continue statement are not executed. Enter and run this code.

```
// Create a SndBuf object called buf and send it to the speaker
SndBuf buf => dac;
"music208.wav" => string filename;
filename => buf.read;

// Play the word Two five times
// and then count to five million before exiting
0 => int count;
while (count < 6000000) // ten million
{
    count++;
    if (count < 3000001) continue;

    if (count > 3000005) continue;

    <<< "count is", count >>>;
    56183 => buf.pos;
    343::ms => now;
}
<<< "DONE!!! count reached six million: ", count >>>;
<<< "Pretty fast." >>>;
```

Chuck Functions

In Lab2 you created Octave functions.

```
[ret] = function add3( num1, num2, num3)
    ret = num1 + num2 + num3;
endfunction
```

The Octave function was saved in a separate text file named add3.m and could be called like this.

```
octave:3> add3( 1, 2, 3 )
ans = 6
```

You can also write functions in ChuckK. A single ChuckK source file can contain multiple functions. Here's an example.

```

1 // add3 take three int parameters, adds them up
2 // and returns an int
3 function int add3( int n1, int n2, int n3 )
4 {
5     n1 + n2 + n3 => int total;
6     return total;
7 }
8
9 // a function can call another function provided
10 // the called function is declared before using
11 function float average3( int n1, int n2, int n3 )
12 {
13     add3( n1, n2, n3 ) => float total;
14     return total / 3;
15 }
16
17 // a void return value is used to indicate that the
18 // function does not return a value
19 function void printPowersOf2( int maxPwr )
20 {
21     for ( 0 => int ix; ix <= maxPwr; ix++ )
22     {
23         <<< ix, "\t", Math.pow( 2, ix ) $ int, "" >>>;
24     }
25 }
26 |
27 // this section of the code tests the functions
28 add3( 1, 2, 4 ) => int sum;
29 <<< "sum =", sum >>>;
30 <<< "add3( 1, 2, 3 ) returned", add3( 1, 2, 4 ) >>>;
31
32 average3( 1, 2, 4 ) => float avg;
33 <<< "avg =", avg >>>;
34 <<< "average3 returned", average3( 1, 2, 4 ) >>>;
35
36 printPowersOf2( 16 );

```

```
[chuck](VM): sporking incoming shred: 1 (functions.ck)...\nsum = 7\nadd3( 1, 2, 3 ) returned 7\navg = 2.333333\naverage3 returned 2.333333\n0      1\n1      2\n2      4\n3      8\n4     16\n5     32\n6     64\n7    128\n8    256\n9    512\n10   1024\n11   2048\n12   4096\n13   8192\n14  16384\n15  32768\n16  65536
```

Sample Mangling

Change speed

Open a new miniAudicle window, enter, save it as "changeSpeed.ck", and run this code.

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => string filename;
4 filename => buf.read;
5
6 // normal speed from beginning of file
7 0 => buf.pos;
8 1.0 => buf.rate;
9 buf.length() => now;
10
11 // twice as fast
12 0 => buf.pos;
13 2.0 => buf.rate;
14 buf.length() => now;
15
16 // twice as slow
17 0 => buf.pos;
18 0.5 => buf.rate;
19 buf.length() => now;
20

```

Problem

The slow speed only played the word music.

When the speed is twice as fast every other sample is played, raising the pitch by one octave and shortening the duration by half. When the speed is twice as slow as each sample is played twice.

Create a playAtDifferentSpeed Function

Notice how similar the code for playing at different speeds are. All three speeds do this:

1. buf.pos to zero
2. set buf.rate
3. chuck buf.length to now

Only one parameter changes, `buf.rate` or speed of playback. You can write a function that takes one parameter for speed and plays the sound.

Speed and duration are inversely related. You can fix the duration problem by multiplying the total number of samples by the reciprocal of the speed. For example, two times the speed requires one half the samples and half speed requires twice as many samples.

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => string filename;
4 filename => buf.read;
5
6 function void playAtDifferentSpeed( float speed )
7 {
8     0 => buf.pos;
9     speed => buf.rate;
10    buf.length() * 1/speed => now;
11 }
12
13 playAtDifferentSpeed( 1.0 ); // normal
14 playAtDifferentSpeed( 2.5 ); // faster
15 playAtDifferentSpeed( 0.333 ); // slower

```

Play backwards

```

1 // Create a SndBuf object called buf and send it to the speaker
2 SndBuf buf => dac;
3 "music208.wav" => string filename;
4 filename => buf.read;
5
6 // set the position to the end of the file
7 buf.samples() => buf.pos;
8 // set the rate to -1
9 -1 => buf.rate;
10 // play it
11 buf.length() => now;
12

```

Try playing backwards at different rates.

Randomize

Enter and run this code.

```
1 // sound mangling randomized
2 SndBuf buf => dac;
3 "music208.wav" => string filename;
4 filename => buf.read;
5
6 while( true )
7 {
8     Math.random2(0, buf.samples() ) => buf.pos;
9     Math.random2f(.5, 1.0) => buf.gain;
10    Math.random2f(-2.5, 3.0) => buf.rate;
11    Math.random2(50, 500 ) => int millis;
12    millis::ms => now;
13 }
```

Modify other parameters as you wish.

While it is running click "Add Shred" again and again.

HID (Human Interface Device) Keyboard

ASCII (American Standard Code For Information Interchange)

The ASCII standard assigns numerical codes to the letters and symbols found on the computer keyboard. It's one of the ways the computer translates the character you type on the keyboard to the symbol that appears on the screen.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

<http://upload.wikimedia.org/wikipedia/commons/1/1b/ASCII-Table-wide.svg>

Open kb.ck

One of the numerous code examples that are included with ChucK and miniAudicle is kb.ck. You'll find it in the m208Lab3 folder you downloaded at the beginning of Lab 3.

Run kb.ck

Important: Activate either the Virtual Machine window or the Console Monitor window before you start typing so your typing does not affect your code in the code window.

Type these for four characters m, 2, o, and 8 (one letter followed by three numbers). The results will appear in the Console Monitor window as you type. The corresponding ASCII codes are marked with a red dot in the table above.

```

[chuck](VM): sporking incoming shred: 1 (kb.ck)...
keyboard 'Keyboard' ready
down-which 16 down-key 16 down-ascii 77 ] m
up-which: 16 up-key 16 up-ascii 77
down-which 31 down-key 31 down-ascii 50 ] 2
up-which: 31 up-key 31 up-ascii 50
down-which 39 down-key 39 down-ascii 48 ] 0
up-which: 39 up-key 39 up-ascii 48
down-which 37 down-key 37 down-ascii 56 ] 8
up-which: 37 up-key 37 up-ascii 56
[chuck](VM): removing shred: 1 (kb.ck)...

```

Copy all of the code from kb.ck and paste it into a new window. Delete lines 27-32.

```

21 // get one or more messages
22 while( hi.recv( msg ) )
23 {
24     // check for action type
25     if( msg.isButtonDown() )
26     {
27         <<< "down-which", msg.which, "down-key", msg.key, "down-ascii", msg.ascii >>>;
28     }
29
30     else
31     {
32         <<< "up-which:", msg.which, "up-key", msg.key, "up-ascii", msg.ascii >>>;
33     }
34 }
35 }
36

```

sayMusic208.ck

Save kb.ck as "sayMusic208.ck". It should look like this.

```

1 // sayMusic208.ck
2 // Chuck Example code HID/kb.ck modified for MUSC 208
3 Hid hi;
4 HidMsg msg;
5
6 // which keyboard
7 0 => int device;
8 // get from command line
9 if( me.args() ) me.arg(0) => Std.atoi => device;
10
11 // open keyboard (get device number from command line)
12 if( !hi.openKeyboard( device ) ) me.exit();
13 <<< "keyboard '" + hi.name() + "' ready", "" >>>;
14
15 // infinite event loop
16 while( true )
17 {
18     // wait on event
19     hi => now;
20
21     // get one or more messages
22     while( hi.recv( msg ) )
23     {
24         // check for action type
25         if( msg.isButtonDown() )
26         {
27
28         }
29     }
30 }

```

You'll need the start time in samples and the length in ms for the words music, two, oh, eight, that you saved at the beginning of this lab.

Create The Sndbuf Object And Load The music208.wav File

```
11 // open keyboard (get device number from command line)
12 if( !hi.openKeyboard( device ) ) me.exit();
13 <<< "keyboard '" + hi.name() + "' ready", "" >>>;
14
15 SndBuf buf => dac;
16 "music208.wav" => buf.read;
```

Define Variables For The ASCII Values of M 2 0 8

```
15 SndBuf buf => dac;
16 "music208.wav" => buf.read;
17
18 77 => int ascii_M;
19 50 => int ascii_2;
20 48 => int ascii_0;
21 56 => int ascii_8;
22
```

Create Four Functions To Respond To Key Presses

```
18 77 => int ascii_M;
19 50 => int ascii_2;
20 48 => int ascii_0;
21 56 => int ascii_8;
22
23 function void playMusic()
24 {
25     <<< "in playMusic()" >>>;
26 }
27
28 function void playTwo()
29 {
30     <<< "in playTwo()" >>>;
31 }
32
33 function void play0h()
34 {
35     <<< "in play0h()" >>>;
36 }
37
38 function void playEight()
39 {
40     <<< "in playEight()" >>>;
41 }
42
```

Add These Lines To The While Loop

```

42
43 // infinite event loop
44 while( true )
45 {
46     // wait on event208m0m82
47     hi => now;
48
49     // get one or more messages
50     while( hi.recv( msg ) )
51     {
52         // check for action type
53         if( msg.isButtonDown() )
54         {
55             if (msg.ascii == ascii_M)
56                 playMusic();
57             else if (msg.ascii == ascii_2)
58                 playTwo();
59             else if (msg.ascii == ascii_0)
60                 playOh();
61             else if (msg.ascii == ascii_8)
62                 playEight();
63         }
64     }
65 }
66

```

Run The Program

Remember to deactivate the code window before you start typing.

Type m, 2, o, 8 and watch the Console Window for debugging statements. You should see this.

```
[chuck](VM): sporking incoming shred: 1 (sayMusic208.ck)...
keyboard 'Keyboard' ready
"in playMusic()" : (string)
"in playTwo()" : (string)
"in play0h()" : (string)
"in playEight()" : (string)
```

Complete The Four Functions

Fill in the start times and durations labelled xxx with the values you obtained from Audacity.

```
23 function void playMusic()
24 {
25     xxx => buf.pos;
26     xxx::ms => now;
27 }
28
29 function void playTwo()
30 {
31     xxx => buf.pos;
32     xxx::ms => now;
33 }
34
35 function void play0h()
36 {
37     xxx => buf.pos;
38     xxx::ms => now;
39 }
40
41 function void playEight()
42 {
43     xxx => buf.pos;
44     xxx::ms => now;
45 }
46
```


Run The Program

It ran but there are four problems.

Problem 1 - Sound plays as soon as program runs

The music208 sound plays at the start of the program, before typing anything. There are at least two ways to prevent this.

1. Set the buf.gain to zero at the beginning. That would mean setting it to 1.0 in every one of the four functions because we don't know which letter will be typed first.
2. A better solution is to set buf.pos to the end of the file. Add line 17 to your code and run the file again.

```

15 SndBuf buf => dac;
16 "music208.wav" => buf.read;
17 buf.samples() => buf.pos;
18

```

Problem 2 - Words start OK but don't end OK

The words start at the correct place but they don't end properly. The sound continues until the end of the file is reached.

Solution: Add the "buf.samples() => buf.pos;" as the last line in each of the four functions. This way when the specified duration is finished, buf.pos is set to the end of the file.

```

24 function void playMusic()
25 {
26     0 => buf.pos;
27     551::ms => now;
28     buf.samples() => buf.pos;
29 }

```

It should work now. You should be able to make the computer say things like:

"music two oh eight"
 ""eight two oh music"
 "oh music ate too"
 etc.

Problem 3 - No layering of sounds

A new sound will not start until the current sound is finished. Try pressing M as fast as you can 5-10 times. There is a noticeable lag from when you finish typing until the sounds end.

Problem 4 - No polyphony

No polyphony, you can't play two different sounds at the same time. Try pressing any two keys simultaneously. The sounds come one after the other depending on which key was detected first.

Solutions

These problems can be solved in two steps.

1. Create a separate SndBuf object inside each of the four functions.
2. spork~ the functions

Spork ~

Sporking a function sends it off to execute independently, simultaneously, and in parallel with the main program. Other programming languages refer to this concept as forking, threading, or parallel processing. ChuckK is known for its play on words and its documentation calls fork, spork; thread, shred; and scheduler, shreduler. The tilde symbol (~) in spork~ must be present. It's part of the ChuckK language.

Create A Sndbuf Object Inside Each Of The Four Functions

Copy these three lines.

```

11 // open keyboard (get device number from command line)
12 if( !hi.openKeyboard( device ) ) me.exit();
13 <<< "keyboard '" + hi.name() + "' ready", "" >>>;
14
15 SndBuf buf => dac;
16 "music208.wav" => buf.read;
17 buf.samples() => buf.pos;
18
19 77 => int ascii_M;
```

Then add comments in front of the each of them.

```
15 // SndBuf buf => dac;  
16 // "music208.wav" => buf.read;  
17 // buf.samples() => buf.pos;
```

Paste the three lines at the beginning of each function. Set buf.pos to zero.

```
24 function void playMusic()  
25 {  
26     SndBuf buf => dac;  
27     "music208.wav" => buf.read;  
28     0 => buf.pos;  
29     551::ms => now;  
30     buf.samples() => buf.pos;  
31 }  
32
```

spork ~ Each of the Four Functions

Add "spork ~ " before each function call in the while loop.

```

60 // infinite event loop
61 while( true )
62 {
63     // wait on event208m0m82
64     hi => now;
65
66     // get one or more messages
67     while( hi.recv( msg ) )
68     {
69         // check for action type
70         if( msg.isButtonDown() )
71         {
72             if (msg.ascii == ascii_M)
73                 spork ~ playMusic();
74             else if (msg.ascii == ascii_2)
75                 spork ~ playTwo();
76             else if (msg.ascii == ascii_0)
77                 spork ~ playOh();
78             else if (msg.ascii == ascii_8)
79                 spork ~ playEight();
80         }
81     }
82 }
83

```

Run It

Type any combination of m, 2, o, 8 as fast as you want, simultaneously or separately, and ChuckK will keep adding sounds layered on top of one another.

Further Enhancements

You could assign other keys to increase or decrease the gain, the playback rate, and play forwards or backwards.

Done with Lab 3.