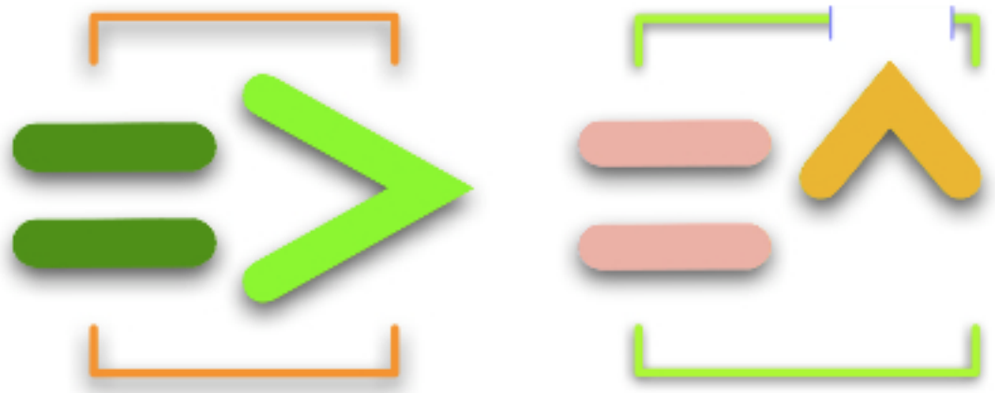
ERICA

소프트웨어학부

CSE2020 음악프로그래밍



4
Sound Files and Sound Manipulation

Contents

- How to use sound files (`.wav`, `.aif`, or others) in ChucK
  - ✦ sound files = samples
- Making your own beat-rockin' techno drum machine using samples

Sampling: turning sound into numbers

- **Sound** – the traveling of fluctuating air compressions through the air, from some source to our ears
- **Analog-to-Digital Converter (ADC)** – turns sound waveform into a stream of numbers, called digital signal
    - inspects electronically the incoming waveform (= sampling) at regular interval (sampling period = 1 / sampling rate), rounds the value to the nearest integer, and stores that amplitude.

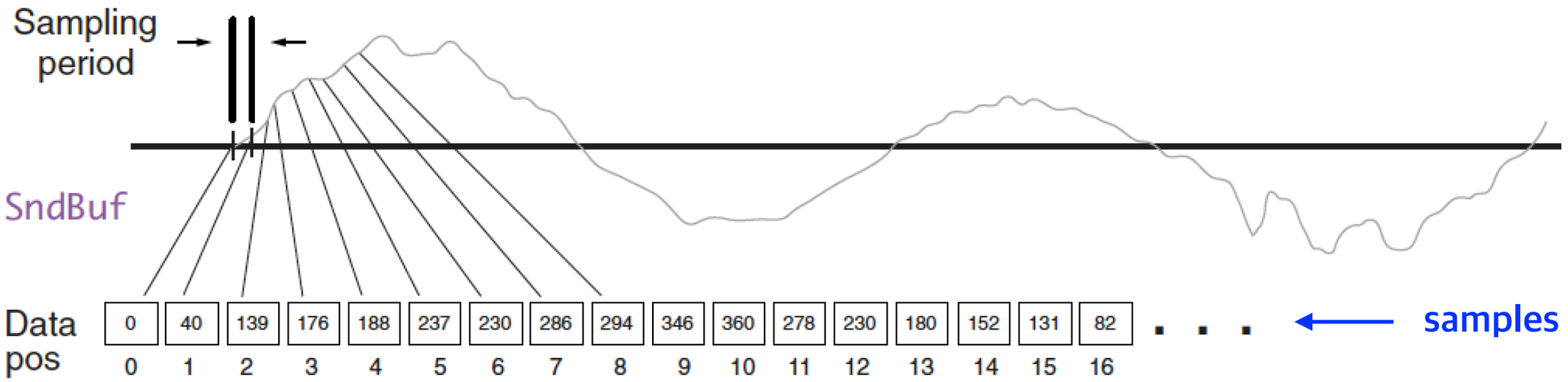


Figure 4.1 **Sampling of one period of an "eee" sound. Pressure at a microphone is turned into voltage, which is turned into a number by a process known as analog-to-digital conversion.**

    - ChucK converts each sample into float, +1.0~–1.0
- **Digital-to-Analog Converter (DAC)** – turns digital signal into sound

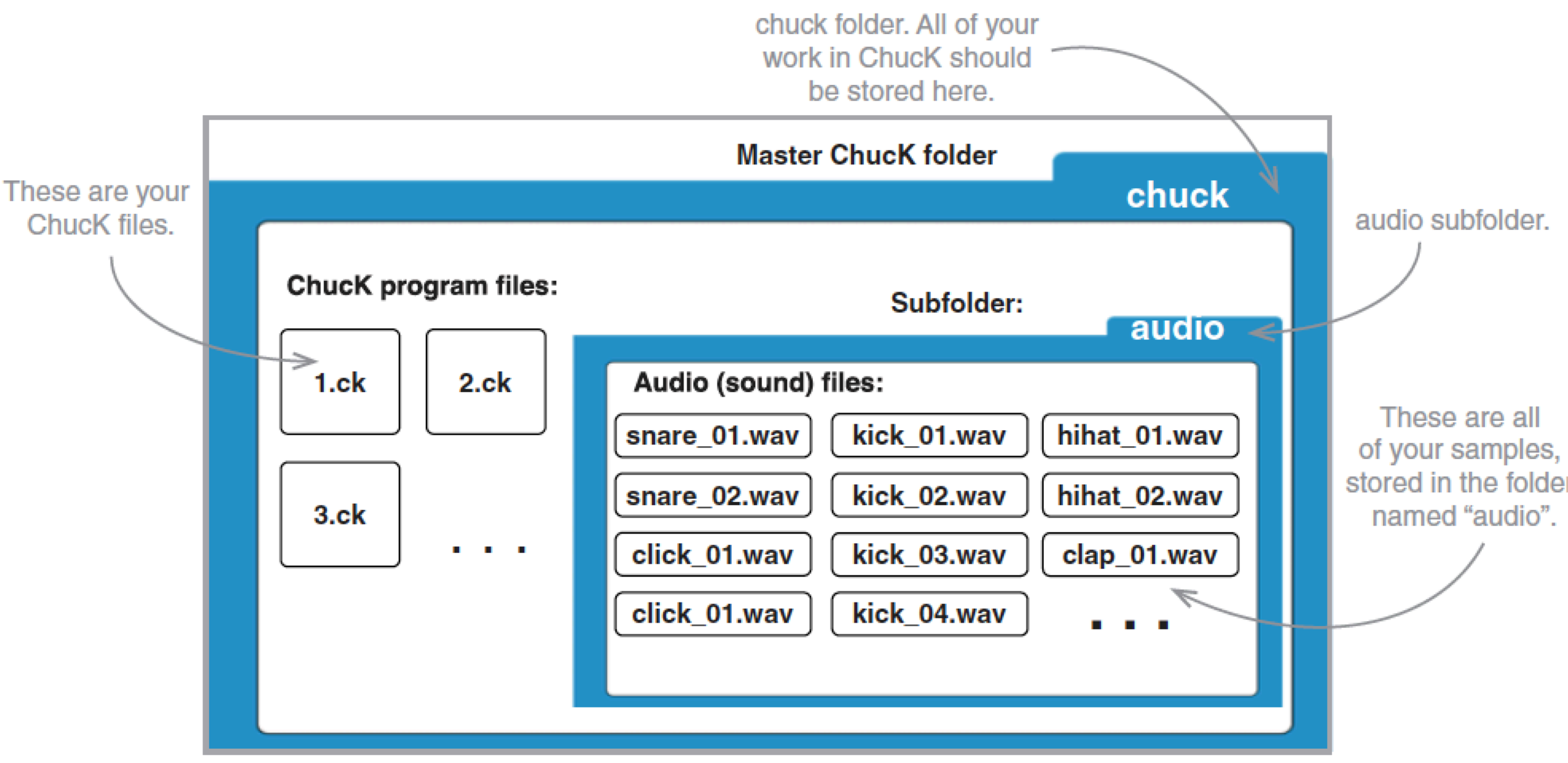
Sampling: turning sound into numbers

Sampling, word size, and sampling rate

Engineers need to consider many issues when creating and working with sound files and especially when designing hardware for ADCs and DACs. These include how much computer storage should be allotted for each individual sample value (called word size, typically 16 bits or 24 bits per sample) for music and often 8 bits for speech sounds. Also, they must choose how often the waveform is sampled (typically 44100 samples per second). These have implications for quality and memory/disk space. For example, higher sample rates and more bits per sample are generally better, up to a point. CD-quality sound, at 2 bytes per sample, 2 channels, and 44100 samples per second, consumes about 10 megabytes per minute.

SndBuf : loading and playing sound files in ChucK

- **SndBuf** (Sound Buffer)
  - the built-in ChucK UGen that can load sound files
- Recommended file structure for ChucK projects



SndBuf : loading and playing sound files in ChucK

Listing 4.1 Loading and playing sound files with SndBuf

```
// Using SndBuf to play a sound file
// by ChucK Programmer, December 2050
SndBuf mySound => dac;

// get file path
me.dir() => string path;
// sound file we want to play
"/audio/snare_01.wav" => string filename;

// + sign connects strings together!
path+filename => filename;

// tell SndBuf to read this file
filename => mySound.read;

// set gain
0.5 => mySound.gain;

// play sound from the beginning
0 => mySound.pos;

// advance time so we can hear it
second => now;
```

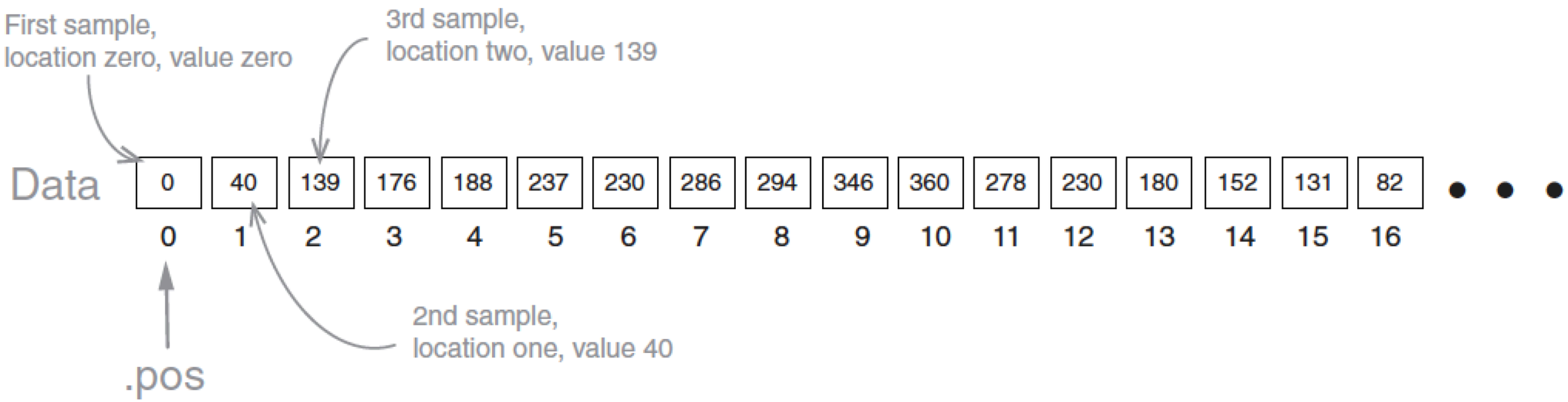
1 Gets current working directory.

2 Builds a full path/name to a sound file.

3 ChucKing a string to the .read member of SndBuf causes it to load that file.

ChucKing a number to the .pos (for position) member of SndBuf causes it to start playing from that position in the array (in this case, from the beginning).

SndBuf : loading and playing sound files in ChucK



= playhead

SndBuf method	range	
.read		
.pos	0 ~ .samples()	
.gain	0.0 ~ 1.0	
.rate	~ 1.0 ~	play forward
	~ -1.0 ~	play backward

SndBuf : loading and playing sound files in ChucK

Listing 4.2 Using a loop to repeatedly play a sound file

```
// Play a sound file repeatedly in a loop
// by ChucK Programmer, January 12, 2017
SndBuf mySound => Pan2 pan => dac;

// get file path and load file all in one line!
me.dir()+"/audio/cowbell_01.wav" => mySound.read;

// play our sound over and again in an infinite loop
while (true)
{
    // random gain, rate (pitch), and pan each time
    Math.random2f(0.1,1.0) => mySound.gain;
    Math.random2f(-1.0,1.0) => pan.pan;
    Math.random2f(0.2,1.8) => mySound.rate;

    // (re)start the sound by setting position to 0
    0 => mySound.pos;

    // advance time so we can hear it
    500.0 :: ms => now;
}
```

Connects a SndBuf through a Pan2 panner object to the DAC

Infinite loop to play this sound over and over

1 Random gain for sound file

Random pan position

2

3 Random rate (speed and pitch)

4 Sets pos to zero to start it playing

5 Hangs out a bit while it plays

SndBuf : loading and playing sound files in ChucK

Listing 4.3 Playing a sound file backward

```
// Playing sounds in reverse
// by ChucK Programmer 4102, yluJ
SndBuf mySound => dac;

me.dir()+"/audio/hihat_04.wav" => mySound.read;

mySound.samples() => int numSamples;

// play sound once forward
0 => mySound.pos;
numSamples :: samp => now;

// and once backward
numSamples => mySound.pos;
-1.0 => mySound.rate;
numSamples :: samp => now;
```

The date is backward, just like the sound soon will be.

1 Assembles file path/name and read file into SndBuf.

2 Asks the sound how long it is (in samples).

3 Lets it play for that long.

4 Sets position to end of buffer.

Sets it to play backward (5)

Play for whole duration, but backward.

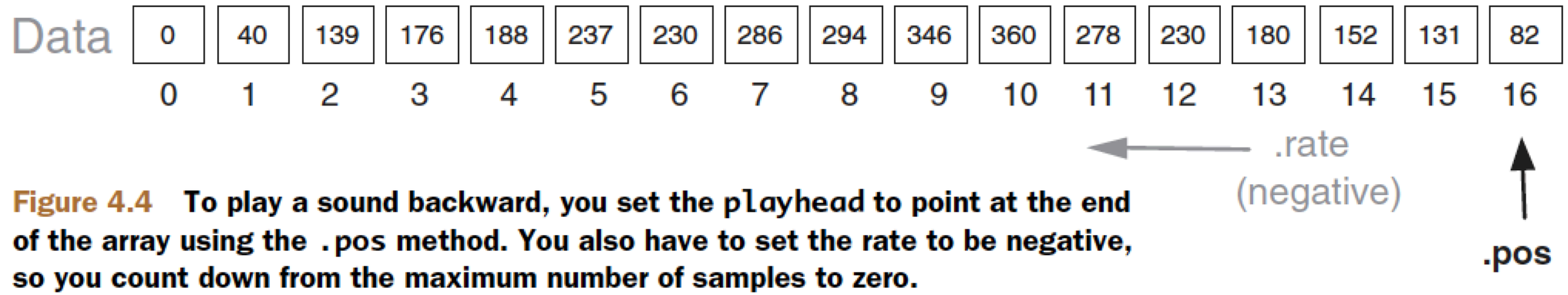


Figure 4.4 To play a sound backward, you set the playhead to point at the end of the array using the .pos method. You also have to set the rate to be negative, so you count down from the maximum number of samples to zero.

SndBuf : loading and playing sound files in ChucK

Listing 4.4 Playing different sound files with a single SndBuf

```
// Playing multiple sounds
// by ChucK Programmer July, 2023
SndBuf snare => dac;

// make and fill an array of sound file paths+names
string snare_samples[3];
me.dir()+"/audio/snare_01.wav" => snare_samples[0];
me.dir()+"/audio/snare_02.wav" => snare_samples[1];
me.dir()+"/audio/snare_03.wav" => snare_samples[2];

// infinite loop
while (true)
{
    // pick a random number between 0 and #files
    Math.random2(0,snare_samples.cap()-1) => int which;

    // load up that file
    snare_samples[which] => snare.read;

    // let it play
    0.5 :: second => now;
}
```

1 Makes and fills an array of sound file names.

2 Picks a random number between 1 and 3.

3 Loads the random file associated with that.

SndBuf : loading and playing sound files in ChucK

Listing 4.5 Playing different sound files (method 2) using multiple SndBufs

```
// Playing multiple sound files (method 2)
// by ChucK Programmer, August, 2023
SndBuf snare[3];

// for fun, let's pan them to left, right, center
snare[0] => dac.left;
snare[1] => dac;
snare[2] => dac.right;

// pre-load all sound files at the beginning
me.dir()+"/audio/snare_01.wav" => snare[0].read;
me.dir()+"/audio/snare_02.wav" => snare[1].read;
me.dir()+"/audio/snare_03.wav" => snare[2].read;

// infinite loop
while (true)
{
    // pick a random number between 0 and #files
    Math.random2(0,snare.cap()-1) => int which;

    // play that sample
    0 => snare[which].pos;

    // let it play
    0.5 :: second => now;
}
```

1 Declares an array of three SndBufs...

...connects them to left, center, right...

2 ...loads them with three different sound files.

Picks a random file to play

Play it!

Stereo sound files and playback

Listing 4.6 Loading and playing stereo sound files using SndBuf2

```
// Loading and playing stereo sound files
// by ChucK Programmer, September, 2023
SndBuf2 myStereoSound => dac;

// load up a stereo file
me.dir()+"/audio/stereo_fx_01.wav" => myStereoSound.read;

// and let it play for the right amount of time
myStereoSound.length() => now;
```

Makes a stereo SndBuf2 and connects it to the dac (stereo is automatic!).

Loads a stereo file using .read.

1 New function/method! .length returns the exact duration of the sound file.

Stereo sound files and playback

Listing 4.7 Stereo panning with stereo sound files using SndBuf2

```
// Loading and panning stereo sound files
// by ChucK Programmer, October, 2023

// declare and load up a stereo file
SndBuf2 myStereoSound;
me.dir()+"/audio/stereo_fx_03.wav" => myStereoSound.read;

// We'll use these for Stereo Panning
Gain bal[2];

// connect everything up in stereo
myStereoSound.chan(0) => bal[0] => dac.left;
myStereoSound.chan(1) => bal[1] => dac.right;
// set our soundfile to repeat forever
1 => myStereoSound.loop;

while (true)  {
     // pick a random playback rate and a random panning
     Math.random2f(0.2,1.8) => myStereoSound.rate;
     Math.random2f(-1.0,1.0) => float balance;

     // turn balance into left/right gain between 0 and 1.0
     (balance+1)/2.0 => float rightGain;
     1.0 - rightGain => float leftGain;
     leftGain => bal[0].gain;
     rightGain => bal[1].gain;

     0.3 :: second => now;
}
```

1. Makes a stereo SndBuf2 and loads a stereo sound file.
2. Makes a Gain UGen array for stereo volume control.
3. Connects left to left, right to right.
4. Repeats automatically.
5. Sets a random rate (pitch and time).
6. Sets a random balance (pan).
7. Implements the stereo balance control.

Example: making a drum machine

Listing 4.8 Making a drum machine with SndBufs in ChucK

```
//  Drum Machine, version 1.0
//  by block-rockin programmer, Dec 31, 1999

// SndBufs for kick (bass) and snare drums
SndBuf kick => Gain master => dac;
SndBuf snare => master;

// load up some samples of those
me.dir()+"/audio/kick_01.wav" => kick.read;
me.dir()+"/audio/snare_01.wav" => snare.read;

while (true)
{
    // Beat 1, play kick drum alone
    0 => kick.pos;
    0.6 :: second => now;

    // play both drums on Beat 2
    0 => kick.pos;
    0 => snare.pos;
    0.6 :: second => now;
}
```

1. SndBuf into a master mixer Gain into dac
2. Another SndBuf into master mixer
3. Loads your sound files
4. Every even beat, play kick only
5. All beats, play both kick and snare

Example: making a drum machine

Listing 4.9 Improving the `while` loop of your drum machine

```
//  Drum Machine, version 2.0
//  by block-rockin programmer, Dec 32, 1999

// SndBufs for kick (bass) and snare drums
SndBuf kick => Gain master => dac;
SndBuf snare => master;

// load up some samples of those
me.dir()+"/audio/kick_01.wav" => kick.read;
me.dir()+"/audio/snare_01.wav" => snare.read;

//  declare a new tempo variable
0.15 :: second => dur tempo;

while (true)
{
    for (0 => int beat; beat < 16; beat++)
    {
        // play kick drum on beat 0, 4, 8, and 12
        if (beat==0 || beat==4 || beat==8 || beat==12)
        {
            0 => kick.pos;
        }
        // play snare drum on beat 4, 10, 13, and 14
```

1. Kick to master mixer Gain to dac.
2. Snare also to master mixer.
3. Loads your kick and snare drum sound files.
4. Tempo duration is time between beats.
5. Loops over a "measure" of 16 beats.
6. Plays kick only on specific beats.

```
//  by block-rockin programmer, Dec 32, 1999

// SndBufs for kick (bass) and snare drums
SndBuf kick => Gain master => dac;
SndBuf snare => master;

// load up some samples of those
me.dir()+"/audio/kick_01.wav" => kick.read;
me.dir()+"/audio/snare_01.wav" => snare.read;

//  declare a new tempo variable
0.15 :: second => dur tempo;

while (true)
{
    for (0 => int beat; beat < 16; beat++)
    {
        // play kick drum on beat 0, 4, 8, and 12
        if (beat==0 || beat==4 || beat==8 || beat==12)
        {
            0 => kick.pos;
        }
        // play snare drum on beat 4, 10, 13, and 14
        if (beat==4 || beat==10 || beat==13 || beat==14)
        {
            0 => snare.pos;
        }
        tempo => now;
    }
}
```

1. Kick to master mixer Gain to dac.
2. Snare also to master mixer.
3. Loads your kick and snare drum sound files.
4. Tempo duration is time between beats.
5. Loops over a "measure" of 16 beats.
6. Plays kick only on specific beats.
7. Plays snare only on other specific beats.

Example: making a drum machine

Listing 4.10 Using arrays to further improve your drum machine

```
//  Drum Machine, version 3.0
//  by block-rockin programmer, Dec 31, 1999

// SndBufs for kick (bass) and snare drums
SndBuf kick => Gain master => dac;
SndBuf snare => master;
SndBuf hihat => master;

// load up some samples of those
me.dir()+"/audio/kick_01.wav" => kick.read;
me.dir()+"/audio/snare_01.wav" => snare.read;
me.dir()+"/audio/hihat_01.wav" => hihat.read;
0.3 => hihat.gain;

0.15 :: second => dur tempo;

// scores (arrays) to tell drums when to play
[1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0] @=> int kickHits[];
[0,0,1,0,0,0,1,0,0,0,0,0,1,1,1,1] @=> int snareHits[];

while (true)
{
    0 => int beat;
    while (beat < kickHits.cap())
    {
        // play kick drum based on array value
```

1 kick, snare, and hihat SndBufs into mixer to dac

2 Arrays to control when kick and snare play

Checks array at location beat to see if you should

```
me.dir()+"/audio/snare_01.wav" => snare.read;
me.dir()+"/audio/hihat_01.wav" => hihat.read;
0.3 => hihat.gain;

0.15 :: second => dur tempo;

// scores (arrays) to tell drums when to play
[1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0] @=> int kickHits[];
[0,0,1,0,0,0,1,0,0,0,0,0,1,1,1,1] @=> int snareHits[];

while (true)
{
    0 => int beat;
    while (beat < kickHits.cap())
    {
        // play kick drum based on array value
        if (kickHits[beat])
        {
            0 => kick.pos;
        }
        if (snareHits[beat])
        {
            0 => snare.pos;
        }
        // always play hihat
        1 => hihat.pos;
        tempo => now;
        beat++;
    }
}
```

❷ Arrays to control when kick and snare play

Checks array at location beat to see if you should play the kick drum now.

Checks array at location beat to see if the snare drum should play now.

Play hihat on every beat for now.

Advance time; lets sound synthesize.

Increments to next beat, and goes back and does the inner while loop again.

A new math/music tool: the **modulo** operator

Listing 4.11 Using the modulo operator

```
// Modulo math for music
// by musical math dude, 11/11/11

// make and load a couple of SndBufs to "sonify" modulo
SndBuf clickhi => dac;
SndBuf clicklo => dac;
me.dir()+"/audio/click_02.wav" => clickhi.read;
me.dir()+"/audio/click_01.wav" => clicklo.read;

// define a number for our modulo
4 => int MOD;

for (0 => int beat; beat < 24; beat++)  {
    // print out beat and beat modulo MOD
    <<< beat, "Modulo ",  MOD, " is: ", beat % MOD >>>;

    // click this on every beat
    0 => clickhi.pos;

    // but only click this on every "MODth" beat
    if (beat % MOD == 0)  {
        0 => clicklo.pos;
    }

    0.5 :: second => now;
}
```

1 Loads two different sound files

2 Modulo limit MOD

3 24-beat “measure”

4 High sound on every beat

5 Low sound only every MOD beat

Tying it all together: the coolest drum machine yet

Listing 4.12a SndBufs and panning connections for your big drum machine

```
//  Drum Machine, version 4.0
//  by block-rockin programmer, Jan 1, 2099
//  Here we'll use Modulo % and random to play drums


//  Define Gains for left, center, right
Gain master[3];
master[0] => dac.left;
master[1] => dac;
master[2] => dac.right;

// Declare SndBufs for lots of drums
//      hook them up to pan positions
SndBuf kick => master[1];
SndBuf snare => master[1];
SndBuf hihat => master[2];
SndBuf cowbell => master[0];

// Use a Pan2 for the hand claps,
//      we'll use random panning later
SndBuf claps => Pan2 claPan;
```

Declares an array of master gains, one for left ([0]), center ([1]) and right ([2]).

Connects left ([0]) to dac.left.

Connects right ([2]) to dac.right.

Connects center ([1]) to dac (both left and right automatically).

Connects kick drum SndBuf to center master gain.

Connects hihat to right master gain.

Connects snare drum to center also.

Connects cowbell SndBuf to left master gain.

Connects clap SndBuf to a Pan2 object.

```
//  Define Gains for left, center, right
Gain master[3];
master[0] => dac.left;
master[1] => dac;
master[2] => dac.right;
```

Defines three gains, one for left ([0]), center ([1]) and right ([2]).

Connects left ([0]) to dac.left.

Connects right ([2]) to dac.right.

Connects center ([1]) to dac (both left and right automatically).

```
// Declare SndBufs for lots of drums
//      hook them up to pan positions
SndBuf kick => master[1];
SndBuf snare => master[1];
SndBuf hihat => master[2];
SndBuf cowbell => master[0];
```

Connects kick drum SndBuf to center master gain.

Connects hihat to right master gain.

Connects snare drum to center also.

Connects cowbell SndBuf to left master gain.

```
// Use a Pan2 for the hand claps,
//      we'll use random panning later
SndBuf claps => Pan2 claPan;
claPan.chan(0) => master[0];
claPan.chan(1) => master[2];
```

Connects clap SndBuf to a Pan2 object.

Connects the left (0) channel of the Pan2 to master gain left.

Connects the right (1) channel of the Pan2 to master gain right.

Loads all the sound files.

```
// load up some samples of those
me.dir()+"/audio/kick_01.wav" => kick.read;
me.dir()+"/audio/snare_01.wav" => snare.read;
me.dir()+"/audio/hihat_01.wav" => hihat.read;
me.dir()+"/audio/cowbell_01.wav" => cowbell.read;
me.dir()+"/audio/clap_01.wav" => claps.read;
```

Tying it all together: the coolest drum machine yet

Listing 4.12b Setting up variables for your big drum machine

```
// array for controlling our cowbell strikes
[1,0,1,0, 1,0,0,1, 0,1,0,1, 0,1,1,1] @=> int cowHits[];

// controls the overall length of our "measures"
cowHits.cap() => int MAX_BEAT;    // all caps, remember?

// modulo number for controlling kick and snare
4 => int MOD;

// overall speed control
0.15 :: second => dur tempo;

// counters: beat within measures, and measure
0 => int beat;
0 => int measure;
```

1. Array to control cowbell strikes.

Determines the maximum number of beats in your measure. Define this using all capital letters.

Constant for MOD operator—you'll use this to control kick and snare drum hits.

Master speed control (tempo)—you'll use this to advance time each beat.

Two counters, one for beat and one for measure number.

Tying it all together: the coolest drum machine yet

Listing 4.12c Main loop to actually play drum patterns

```
// Main infinite drum loop
while (true)
{
    // play kick drum on all main beats (0, 4, ...)
    if (beat % 4 == 0)
    {
        0 => kick.pos;
    }
    // after a time, play snare on off beats (2, 6, ...)
    if (beat % 4 == 2 && measure %2 == 1)
    {
        0 => snare.pos;
    }
    // After a time, randomly play hihat or cowbell
    if (measure > 1)  {
        if (cowHits[beat])
        {
            0 => cowbell.pos;
        }
        else
        {
            Math.random2f(0.0,1.0) => hihat.gain;
            0 => hihat.pos;
        }
    }
    // after a time, play randomly spaced claps at end of measure
    if (beat > 11 && measure > 3)
    {
```

1. Uses MOD 4 to play kick drum every four beats
2. Plays snare only on specific beats
3. Plays cowbell and hihat only after measure 1
4. Plays cowbell, controlled by array
5. If not cowbell, then hi-hat
6. Hi-hat has random gain
7. Plays claps only on
8. Claps have random pan

```
{
    0 => kick.pos;
}
// after a time, play snare on off beats (2, 6, ...)
if (beat % 4 == 2 && measure %2 == 1)
{
    0 => snare.pos;
}
// After a time, randomly play hihat or cowbell
if (measure > 1)  {
    if (cowHits[beat])
    {
        0 => cowbell.pos;
    }
    else
    {
        Math.random2f(0.0,1.0) => hihat.gain;
        0 => hihat.pos;
    }
}
// after a time, play randomly spaced claps at end of measure
if (beat > 11 && measure > 3)
{
    Math.random2f(-1.0,1.0) => claPan.pan;
    0 => claps.pos;
}
tempo => now;
(beat + 1) % MAX_BEAT => beat;
if (beat==0)
{
    measure++;
}
}
```

1. Uses MOD 4 to play kick drum every four beats
2. Plays snare only on specific beats
3. Plays cowbell and hihat only after measure 1
4. Plays cowbell, controlled by array
5. If not cowbell, then hi-hat
6. Hi-hat has random gain
7. Plays claps only on certain measures and beats...
8. Claps have random pan
9. ...waits for one beat...
10. ...and then updates beat counter (MOD MAX)
11. Increments measure counter at each new measure