The **ChucK** Tutorial

version: 1.2.x.x (dracula)

Hello ChucK:

*This tutorial was written for the command line version of ChucK (currently the most stable and widely supported). Other methods of running ChucK includes the **miniAudicle** (now on all major platforms) and the **Audicle** (in pre-pre-alpha). The code is the same, but the way to run them differs, depending the ChucK system.*

The first thing we are going to do is do generate a sine wave and send to the speaker so we can hear it. We can do this easily ChucK by connecting audio processing modules (unit generators) and having them work together to compute the sound.

We start with a blank ChucK program, and add the following line of code: (by default, a ChucK program starts executing from the first instruction in the top-level (global) scope).

```
// connect sine oscillator to D/A convertor (sound card)
SinOsc s => dac;
```

The above does several things. (1) it creates a new unit generator of type 'SinOsc' (sine oscillator), and store its reference in variable 's'. (2) 'dac' (D/A convertor) is a special unit generator (created by the system) which is our abstraction for the underlying audio interface. (3) we are using the ChucK operator (=>) to ChucK 's' to 'dac'. In ChucK, when one unit generator is ChucKed to another, we connect them. We can think of this line as setting up a data flow from 's', a signal generator, to 'dac', the sound card/speaker. Collectively, we will call this a 'patch'.

The above is a valid ChucK program, but all it does so far is make the connection (if we ran this program, it would exit immediately). In order for this to do what we want, we need to take care of one more very important thing: time. Unlike many other languages, we don't have to explicitly say "play" to hear the result. In ChucK, we simply have to "allow time to pass" for data to be computed. As we will see, time and audio data are both inextricably related in ChucK (as in reality), and separated in the way they are manipulated. But for now, let's generate our sine wave and hear it by adding one more line:

```
// connect sine oscillator to D/A convertor (sound card)
SinOsc s => dac;

// allow 2 seconds to pass
2::second => now;
```

Let's now run this (assuming you saved the file as 'foo.ck'):

```
%> chuck foo.ck
```

This would cause the sound to play for 2 seconds, during which time audio data is processed (and heard), afterwhich the program exits (since it has reached the end). For now, we can just take the second line of code to mean "let time pass for 2 seconds (and let audio compute during that time)". If you want to play it indefinitely, we could write a loop:

```
// connect sine oscillator to D/A convertor (sound card)
SinOsc s => dac;

// loop in time
```

```
while( true ) {
    2::second => now;
}
```

In ChucK, this is called a 'time-loop' (in fact this is an 'infinite time loop'). This program executes (and generate/process audio) indefinitely. Try runnig this program.

IMPORTANT: perhaps more important than how to run ChucK is how to *stop* ChucK. To stop a ongoing ChucK program from the command line, hit (ctrl - c).

So far, since all we are doing is advancing time, it doesn't really matter (for now) what value we advance time by - (we used 2::second here, but we could have used any number of 'ms', 'second', 'minute', 'hour', 'day', and even 'week'), and the result would be the same. It is good to keep in mind from this example that almost everything in ChucK happens naturally from the timing.

Now, let's try changing the frequency randomly every 100ms:

```
// make our patch
SinOsc s => dac;

// time-loop, in which the osc's frequency is changed every 100 ms
while( true ) {
    100::ms => now;
    Std.rand2f(30.0, 1000.0) => s.freq;
}
```

This should sound like computer mainframes in old sci-fi movies. Two more things to note here. (1) We are advancing time inside the loop by 100::ms durations. (2) A random value between 30.0 and 1000.0 is generated and 'assigned' to the oscillator's frequency, every 100::ms.

Go ahead and run this (again replace foo.ck with your filename):

```
%> chuck foo.ck
```

Play with the parameters in the program. change 100::ms to something else (like 50::ms or 500::ms, or 1::ms or perhaps 1::samp*(every sample)*), or change 1000.0 to 5000.0...

Run and listen:

```
%> chuck foo.ck
```

Once things work, hold on to this file - we will use it again soon.

Concurrency in ChucK:

Now let's write another (slightly longer) program: (these files can be found in the **examples/** directory, so you don't have to type them in)

```
// impulse to filter to dac
Impulse i => BiQuad f => dac;
// set the filter's pole radius
.99 => f.prad;
// set equal gain zero's
1 => f.eqzs;
// initialize float variable
```

```
0.0 => float v;

// infinite time-loop
while( true )
{
    // set the current sample/impulse
    1.0 => i.next;
    // sweep the filter resonant frequency
    Std.fabs(Math.sin(v)) * 4000.0 => f.pfreq;
    // increment v
    v + .1 => v;
    // advance time
    100::ms => now;
}
```

Name this moe.ck, and run it:

```
%> chuck moe.ck
```

Now, make two copies of moe.ck - larry.ck and curly.ck. Make the following modifications. 1) change larry.ck to advance time by 99::ms (instead of 100::ms). 2) change curly.ck to advance time by 101::ms (instead of 100::ms). 3) optionally, change the 4000.0 to something else (like 400.0 for curly).

Run all three in parallel:

```
%> chuck moe.ck larry.ck curly.ck
```

What you hear (if all goes well) should be 'phasing' between moe, larry, and curly, with curly emitting the lower-frequency pulses.

ChucK supports sample-synchronous concurrency, via the ChucK timing mechanism. Given any number of source files that uses the timing mechanism above, the ChucK VM can use the timing information to automatically synchronize all of them. Furthermore, the concurrency is 'sample-synchronous', meaning that inter-process audio timing is guaranteed to be precise to the sample. The audio samples generated by our three stooges in this examples are completely synchronized. Note that each process do not need to know about each other - it only has to deal with time locally. The VM will make sure things happen correctly and globally.

Moving forward:

For more detail on each particular aspect of the ChucK language and virtual machine environment, please see the **language specification**

A large collection of pre-made examples have been arranged and provided with this distribution in the /doc/examples directory, and are mirrored **here**