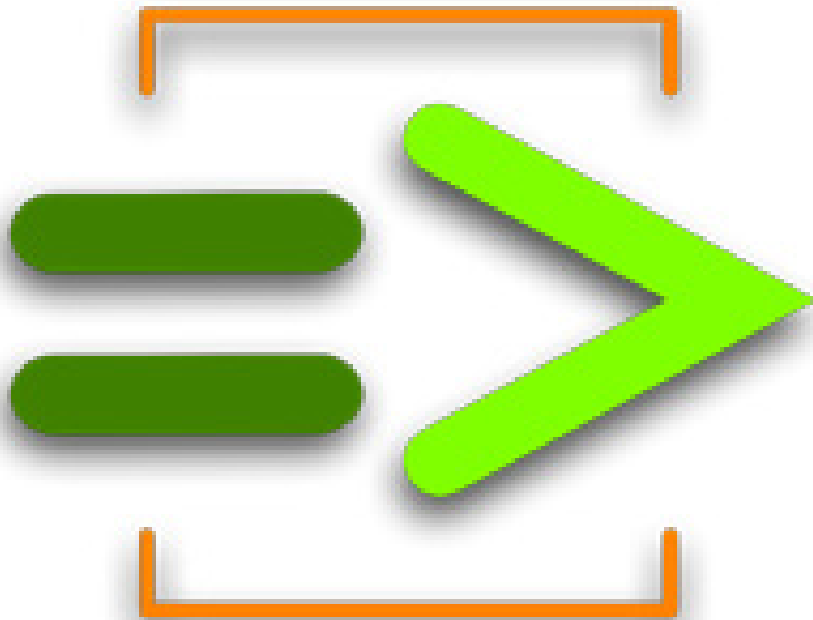


T h e C h u c K M a n u a l

1.2.0.4c



<http://chuck.cs.princeton.edu/>

Authors of ChucK

Lead Developers:

Ge Wang – gewang@cs.princeton.edu
Perry R. Cook – prc@cs.princeton.edu
Ananya Misra – amisra@cs.princeton.edu
Philip Davidson – philipd@alumni.princeton.edu
Ari Lazier – alazier@cs.princeton.edu

Documentation:

Adam Tindale – art@uvic.ca
Philip Davidson – philipd@alumni.princeton.edu
Ananya Misra – amisra@cs.princeton.edu
Ge Wang – gewang@cs.princeton.edu

Lead Testers:

Philip Davidson – philipd@alumni.princeton.edu
Ajay Kapur – akapur@alumni.princeton.edu

Many people have further contributed to ChucK by suggesting great new ideas and improvements, reporting problems, or submitting actual code. Here is a list of these people. Help us keep it complete and exempt of errors.

ThanK you!

- Andrew Appel
- Brian Kernighan
- Dan Trueman
- Ken Steiglitz
- Paul Lansky
- Szymon Rusinkiewicz
- Gary Scavone
- Nick Collins
- Tom Briggs
- Paul Calamia
- Stacken
 - Mikael Johanssons
 - Magnus Danielson
 - Rasmus Kaj
- Princeton Graphics Group

ChuckK Places

ChuckK home page:

<http://chuck.cs.princeton.edu/>

Princeton Sound Lab:

<http://soundlab.cs.princeton.edu/>

ChuckK Documentation + Tutorials:

<http://chuck.cs.princeton.edu/doc/>

For the most updated tutorial:

<http://chuck.cs.princeton.edu/doc/learn/>

For the ideas and design behind ChuckK, read the papers at:

<http://chuck.cs.princeton.edu/doc/publish/>

ChuckK Developer's Guide:

<http://chuck.cs.princeton.edu/doc/develop/>

ChuckK WIKI

<http://chuck.cs.princeton.edu/wiki>

CONTENTS

Authors of ChuckK	i
ChuckK Places	ii
1 Intro-ChuckK-tion	1
2 Installation	3
Binary Installation	3
Source Installation	5
3 ChuckK Tutorials	7
<i>The Chuck Tutorial</i>	7
Conventions	11
On-the-fly-programming	12
Modifying Basic Patches	15
LFOs and Blackhole	17
Working with MIDI	18
Writing To Disk	21
Stereo	23

4 Overview	25
running ChuckK	25
comments	26
debug print	26
reserved words	27
5 Types, Values, and Variables	31
primitive types	31
values (literals)	32
variables	32
reference types	34
6 Arrays	35
declaration	35
multi-dimensional arrays	36
lookup	37
associative arrays	39
array assignment	40
7 Operators and Operations	43
cast	46
8 Control Structures	49
if / else	49
while	50
until	50
for	51
break / continue	51
9 Functions	53

writing	53
calling	54
overloading	54
10 Concurrency and Shreds	56
sporking shreds (in code)	57
the 'me' keyword	58
using machine.add()	59
inter-shred communication	60
11 Time and Timing	61
time and duration	61
operations on time and duration (arithmetic)	62
the keyword 'now'	64
advancing time	64
12 Events	67
what they are	67
use	68
MIDI events	69
OSC events	70
creating custom events	71
13 Objects	73
introduction	73
built-in classes	74
working with objects	74
writing a class	75
members (instance data + functions)	76
class constructors	78

static (data + functions)	78
inheritance	79
Overloading	82
14 The ChucK Compiler + Virtual Machine	83
15 On-the-fly Programming Commands	86
16 Standard Libraries API	89
17 Unit Generators	95

Getting Started

CHAPTER 1

Intro-ChucK-tion

what is it: ChucK is a general-purpose programming language, intended for real-time audio synthesis and graphics/multimedia programming. It introduces a truly concurrent programming model that embeds timing directly in the program flow. Other potentially useful features include the ability to write/change programs on-the-fly.

who it is for: audio/multimedia researchers, developers, composers, and performers

supported platforms:

- MacOS X (CoreAudio)
- Linux (ALSA/OSS) (Jack Coming Soon)
- Windows/Cygwin (DirectSound)
- SGI (coming soon)

On-the-fly Programming

On-the-fly programming is a style of programming in which the programmer/performer/composer augments and modifies the program while it is running, without stopping or restarting, in order to assert expressive, programmable control for performance, composition, and experimentation at

run-time. Because of the fundamental powers of programming languages, we believe the technical and aesthetic aspects of on-the-fly programming are worth exploring.

CHAPTER 2

Installation

We tried to make ChuckK as easy as possible to build and re-use. All sources files - headers source for compiler, vm, and audio engine - are in the same directory (they run in the same address space anyway). Platforms differences are abstracted to the lowest level (in part thanks to Gary Scavone). None of the compiler/vm has any OS-depended code.

Binary Installation

The binary distributions include a directory called bin/ that contains the precompiled binary of ChuckK for your operating system. The binary distribution is a great introduction to ChuckK but once you are comfortable we suggest that you compile ChuckK on your own system so that it is optimized for your hardware.

OS X

1. The terminal is located in the Utilities/ folder in the Applications/ folder of your hard drive. In the terminal go to the bin/ directory (replace chuck-x.x.x.x-exe with the actual directory name):

```
%>cd chuck-x.x.x.x-exe/bin
```

2. Install it using the script included.

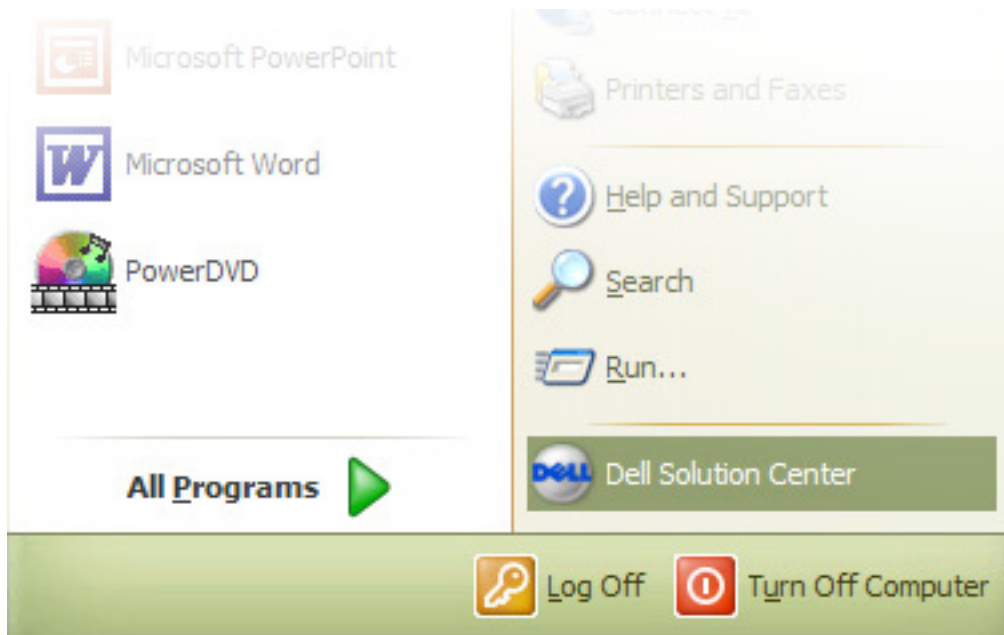
```
%>sudo make install
```

3. Test to make sure it is was installed properly.

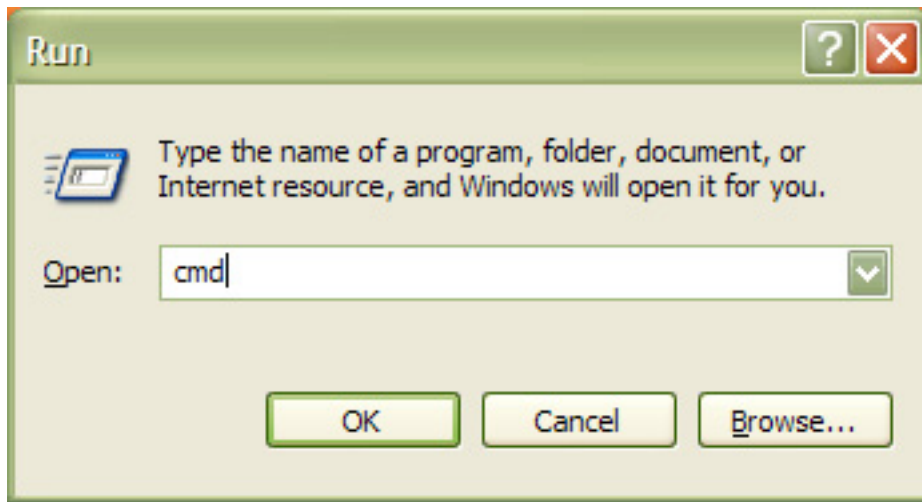
```
%>chuck
```

Windows

1. Place chuck.exe (found in the 'bin' folder) into the c:\windowss\system32\
2. Open a command window found in start->run



3. Type cmd and press return



4. Type chuck and press return

Source Installation

To build chuck from the source (from scratch):

1. Go to the src/ directory (replace chuck-x.x.x.x with the actual directory name):

```
%>cd chuck-x.x.x.x/src/
```

2. If you type 'make' here, you should get the following message:

```
%>make
[chuck] : please use one of the following configurations:
make osx, make linux-oss, make linux-alsa, make win32
```

Now, type the command corresponding to your platform...

for example, for MacOS X:

```
%>make osx
```

for example, for Windows:

```
%>make win32
```

3. If you would like to install chuck (cp into /usr/bin by default). If you don't like the destination, simply edit the makefile under 'install', or skip this step altogether. (we recommend putting it somewhere in your path, it makes on-the-fly programming easier)

```
# (optional: edit the makefile first)
%>make install
```

You may need to have administrator privileges in order to install ChuckK. If you have admin access then you can use the sudo command to install.

```
%>sudo make install
```

4. If you haven't gotten any egregious error messages up to this point, then you should be done! There should be a 'chuck' executable in the current directory. For a quick sanity check, execute the following (use './chuck' if chuck is not in your path), and see if you get the same output:

```
%>chuck [chuck]: no input files...
```

(if you do get error messages during compilation, or you run into some other problem - please let us know and we will do our best to provide support)

You are ready to ChuckK. If this is your first time programming in ChuckK, you may want to look at the documentation, or take the ChuckK Tutorial (<http://chuck.cs.princeton.edu/doc/>).

Thank you very much. Go forth and ChuckK - email us for support or to make a suggestion or to call us idiots.

Ge + Perry

CHAPTER 3

ChuckK Tutorials

The Chuck Tutorial

Hello ChuckK:

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

We start with a blank ChuckK program and add the following line of code:

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

(by default, a ChuckK program starts executing from the first instruction in the top-level (global) scope).

The above does several things:

1. it creates a new unit generator of type ‘sinosc’ (sine oscillator), and stores 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 ChuckK operator (`=>`) to ChuckK 's' to 'dac'. In ChuckK, when one unit generator is ChuckKed 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 ChuckK 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 ChuckK, 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 ChuckK (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), after which 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 ChuckK, this is called a 'time-loop' (in fact this is an 'infinite time loop'). This program executes (and generate/process audio) indefinitely. Try running this program.

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 ChuckK 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 0.0 and 800.0 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 change 400.0 to 4000.0.

Run and listen:

%>chuck foo.ck

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

Concurrency in ChuckK:

Now let’s write another (slightly longer) program:

```
// 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.

ChuckK supports sample-synchronous concurrency via the ChuckK timing mechanism. Given any number of source files that uses the timing mechanism above, the ChuckK 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.

Conventions

ChuckK compiles under many different operating systems. Each of these has their own “features” that make the experience of working with ChuckK slightly different. This chapter will outline some of these differences.

ChuckK is a terminal application (no, ChuckK is not sick, but it may cause nausea, stress, frustration...). You will need to know how to access and navigate in the terminal. Here are some hints about getting started with the terminal on your operating system.

OS X

The terminal is located in the Utilities/ folder in the Applications/ folder of your hard drive. Double click on Terminal. You can click and hold on the icon in the Dock and select the “Keep in Dock” option. Now the Terminal application will be conveniently located in the Dock.

<http://www.macdevcenter.com/pub/ct/51>

<http://www.atomiclearning.com/macosexterminalx.shtml>

Windows

The terminal is accessed by clicking on the Start Menu and then clicking on run. In the window that opens type **cmd**.

<http://www.c3scripts.com/tutorials/msdos/>

<http://www.ss64.com/nt/>

Linux

If you are reading this then you are required to return your tinfoil hat.

On-the-fly-programming

Navigate to the examples folder in the ChuckK distribution then run the following command:

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

In this case, ChuckK will run whatever is in **moe.ck**. You can replace **moe.ck** with the name of another ChuckK file. If this script is a just a loop that never ends then we need to stop ChuckK eventually. Simple press CONTROL-C (hold control and press c). This is the kill process key in the terminal.

Where is the ChuckK power that was promised to you? You can add multiple copies of the same script if you like:

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

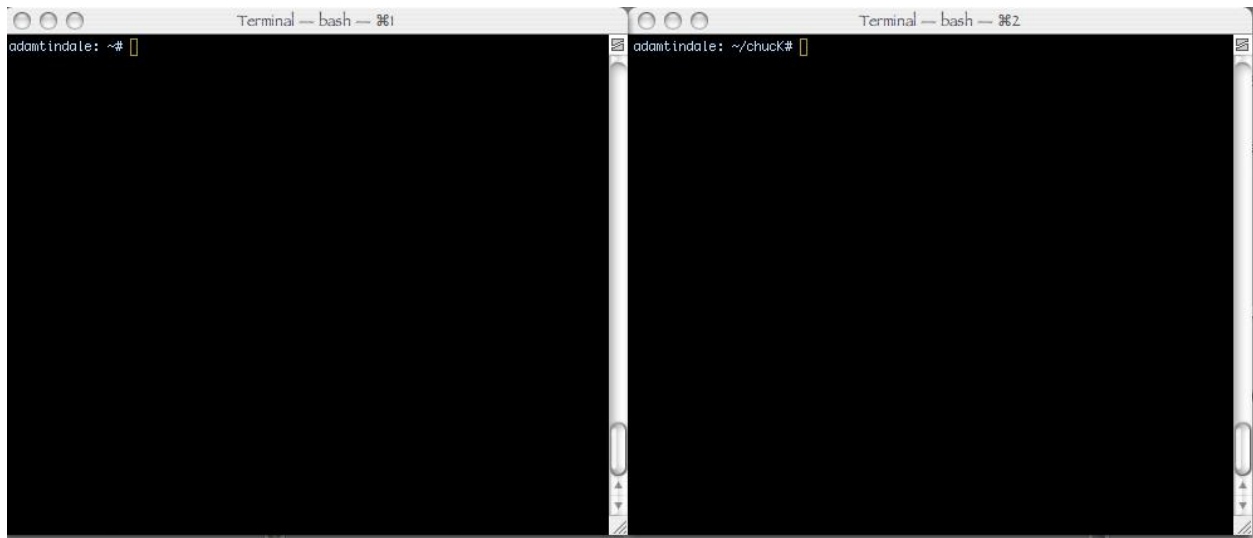
Again, if any of these scripts will go on forever then you have to use the magic CONTROL-C to halt ChuckK. Why would you do that? If the script has some random number generators or something like that then you end up with some nice ChuckK chaos not just copies of the same patch! Give it a try.

Some first things to try to test the concurrency (running multiple ChuckK files in parallel) are moe, larry, and curly. To listen to **moe.ck**, **larry.ck**, or **curly.ck** together type: They are written to go in and out of phase with each other.

Also try the improved versions of our little friends: **larry++.ck curly++.ck moe++.ck**

Two Window ChuckK

Now lets roll up our sleeves a little bit and see some real ChuckK power! We are going to run two window ChuckK, and on-the-fly! This section will walk you through a ChuckK session.



Here is what you do: open another terminal window just like this one. In this new window type:

```
%>chuck -loop
```

This will start ChuckK running. ChuckK is now waiting for something to do. Go back to your original window where you are in your ChuckK home. Be careful. If you type `chuck test1.ck` you will start a second ChuckK running `test1.ck`. What we want to do is add a script to the ChuckK that we set running in our second window. We will use the `+` operator to add a script to our ChuckK and the `-` operator to remove a script.

```
%>chuck + test1.ck
%>chuck - 1
%>chuck test.ck
%>chuck test.ck
%>chuck test.ck
```

What happened? That is the power. We added `test1.ck`. It was added as the first shred in our ChuckK. Since we knew it was shred 1 we removed it by typing `chuck - 1`. Great. Next we added three copies of the same script! Isn't that cool? You can also do this `chuck + test1.ck test1.ck test1.ck` How do you keep track of shreds?

You can ask ChuckK how he is doing by typing `chuck --status` The shortcut is `chuck ^` ChuckK will answer in the other window where we left him running. He will tell you what shreds there are and what their id numbers are. He will also tell you how long he has been running.

When you have had enough of ChuckK you can go to the other window and use your fancy `CONTROL-C` trick or you can type `chuck --kill` in your original window.

```
%>chuck --kill
```

One Window ChuckK

So you think you are pretty good? One window ChuckK is only for the hardest of hardcore. You have been warned.

The concept is pretty similar to two window ChuckK: first, you start a ChuckK going, then you manage the adding and removal of scripts to it. How do you start a ChuckK and get the command prompt to return, you ask? In your shell you can add an ampersand (&) after the command and that will tell the shell to run the command as a background process and give you the prompt back.

```
%>chuck --loop &
```

The rest is as it should be. You will have to be careful when writing your patches to not put too many print statements. When you print you will temporarily lose control of the prompt as the shell prints. This can be bad when are you are printing MIDI input. The same applies when you use the --status command to ChuckK. It can also be fun to fight for your command line. Who will win?

Modifying Basic Patches

We have a basic patch running in ChuckK but it still doesn't sound very good. In this chapter we will cover some simple ways to rectify that problem. ChuckK allows one to quickly make modifications to patches that can drastically change the sound.

First what we can do is change the type of our oscillator. There are many different oscillators available to use: `sinosc` (sine wave), `sawosc` (sawtooth), `sqrosc` (square wave) and `pulseosc` (pulse wave). We can simply change the type of oscillator just like below.

```
sawosc s => dac;
```

Try changing the oscillator to all of the different types and get a feel for how they sound. When changing the different Ugens always be sure to check the rest of your patches so that the parameter calls are valid. If you were to use the `.width` method of `pulseosc` and others on a `sinosc` ChuckK will complain. You can comment out lines that are temporarily broken by using double slashes (`//`).

Now let's add some effects to our patch. ChuckK has many different standard effects that can be added to Ugen chains. The simplest effect we can add is an amplifier. In ChuckK, this object is **gain**.

```
sawosc s => gain g => dac;
```

Now we can change the parameters of our effect. **gain** has a parameter `.gain` that can be used to change the gain of signals passing through the object. Let's go about changing the gain.

```
.5 => g.gain;
```

This is redundant. All Ugens have the ability to change their gain in a similar manner. (See the UGEN section in Reference for more information about UGEN parameters.)

```
.5 => s.gain;
```

However, this is useful when we have multiple Ugens connect to one place. If we were to connect 2 oscillators to the **dac** then we will get distortion. By default, these oscillators oscillate between -1 and 1. When they are connected to the same input they are added, so now they go between -2 and 2. This will clip our output. What to do? **gain** to the rescue!

```
sinosc s1 => gain g => dac;  
sinosc s2 => g;  
.5 => g.gain;
```

Now our oscillators are scaled between -1 and 1 and everything is right in the world.

More effects were promised, now you will see some in action. Again, one of the wonders of ChuckK is how easy it is to change Ugens. We can take the above patch and change ‘gain’ for ‘PRCRev’.

```
sinosc s1 => PRCRev g => dac;
sinsocs s2 => g;
.5 => g.gain;
```

The gain Ugen has been replaced by a reverb and the output is scaled by using the ‘.gain’ parameter that all Ugens possess. Now we can add a few spices to the recipe. ‘PRCRev’ has a ‘.mix’ parameter that can be changed between 0. and 1. If we wanted to have this parameter set to the same value as what we are ChuckKing to g.gain we can chain it along. After assignment a Ugen will return the value that was ChuckKed to it. We can use this method to propagate parameters to our oscillators.

```
.5 => g.gain => g.mix;
500 => s1.freq => s2.freq;
```

Another technique for setting parameters is to read a parameter, then modify it and then ChuckK it back to itself. Accessing parameters requires the addition of brackets () after the name of the parameter. Here is an example of doubling the frequency of an oscillator.

```
s1.freq() * 2 => s1.freq;
```

Let’s change the type of oscillators for some more fun. We can simply replace ‘sinosc’ with any other type of oscillator. Check the Ugen section in the reference for ideas. Try changing the frequency of the oscillators and the mix parameter of the reverb for each type of oscillator you try. Endless hours of fun!

LFOs and Blackhole

A common technique to add variation to synthesis is modulation. Modulation is the process of changing something, usually the parameter of a signal like frequency. A Low Frequency Oscillator (LFO) is typically used for this task because the variations are fast enough to be interesting, yet slow enough to be perceptible. When a signal is modulated quickly (ie. over 20Hz or so) it tends to alter the timbre of the signal rather than add variation.

Ok, let's use this idea. What we need to do is set up two oscillators and have one modulate a parameter of another. ChuckK does not support the connection of Ugen signal outputs to parameter inputs. This piece of code will not work:

```
sinosc s => dac;  
sinosc lfo => s.freq;
```

Foiled. What we need to do is poll our lfo at some rate that we decide on, for now we will update the frequency of s every 20 milliseconds. Remember that a sinosc oscillates between -1 and 1, so if we just put that directly to the frequency of s we wouldn't be able to hear it (unless you are using ChuckK in a tricked out civic...). What we are going to do is multiply the output of the lfo by 10 and add it to the frequency 440. The frequency will now oscillate between 430 and 450.

```
sinosc s => dac;  
sinosc lfo;  
  
// set the frequency of the lfo  
5 => lfo.freq;  
  
while (20::ms => now){  
  ( lfo.last() * 10 ) + 440 => s.freq;  
}
```

ChuckK is a smart little devil. This didn't work and now we will look into the reason. Why? Ugens are connected in a network and usually passed to the **dac**. When a patch is compiled ChuckK looks at what is connected to the **dac** and as each sample is computed ChuckK looks through the network of Ugens and grabs the next sample. In this case, we don't want our Ugen connected to the **dac**, yet we want ChuckK to grab samples from it. Enter blackhole: the sample sucker. If we connect our lfo to blackhole everything will work out just fine.

```
sinosc lfo => blackhole;
```

Play around with this patch in its current form and find interesting values for the poll rate, lfo frequency and the lfo amount. Try changing the Ugens around for more interesting sounds as well.

Working with MIDI

Adding a MIDI controller is a great way to add variety to your ChuckK patches. Conversely, ChuckK offers a simple and powerful way to utilize a MIDI controller for making music.

The first thing to do when working with MIDI is to make sure that ChuckK sees all of your devices. You can do this by using the `--probe` start flag. Like this:

```
%>chuck --probe
```

ChuckK will display a list of the connected audio and MIDI devices and their reference ID. We will assume that your controller is found to have an ID of 0. First, we must open a connection between ChuckK and the port. We can accomplish this by creating a **MidiIn** object and then connecting it to a port.

```
//create object
MidiIn min;

//connect to port 0
min.open(0);
```

If you want to send MIDI out of ChuckK you use the **MidiOut** object and then open a port.

```
//create object
MidiOut mout;

//connect to port 0
mout.open(0);
```

When opening ports it is suggested that you check whether the **.open** function returns properly. In some situations it doesn't make any sense for the shred to live on if there is no MIDI data available to be sent along. You can check the return value of the **.open** function and then exit the shred using the **me** keyword with the **exit()** function.

```
MidiIn min;
min.open(0) => int AmIOpen;

if(!AmIOpen){
    me.exit();
}
```

We can do this in fewer lines of code. We can put the `min.open(0)` in the condition of the `if` statement. This way `min.open` will return true or false (which is represented as ints with a value of 1 or 0). The `!` will give the opposite return value of `min.open`. Now the statement will mean if `min.open` doesn't return true then exit. Get it?

```
if( !min.open(0) ){me.exit();}
```

Getting MIDI

In order to receive any of the juicy data you are piping into ChuckK we need to create a `MidiMsg` object. This object is used to hold data that can be input into ChuckK or output to a MIDI port. Unless you are high skilled at managing the state of these messages (or you enjoy the headache you get from debugging) it is recommended that you create a minimum of one `MidiMsg` for each port you are using.

What we need to do is get the data from the `MidiIn` object into a message that we can use inside of ChuckK. The `MidiMsg` object is just a container with three slots: `data1`, `data2` and `data3`. We fill these slots up by putting our message in the `.recv(MidiMsg)` function of a `MidiIn` object. `MidiIn` keeps its messages in a queue so that you can poll it for messages and it will keep giving messages until it is empty. The `.recv(MidiMsg)` function returns true and false so we can put it in a while loop and it will continue to run through the loop until there are no more messages left.

```
// check for messages every 10 milliseconds
while(10::ms => now){
    //
    while( min.recv(msg) ){
        <<<msg.data1,msg.data2,msg.data3,"MIDI Message">>>;
    }
}
```

The Event system makes life a little easier and also makes sure that MIDI input is dealt with as soon as ChuckK receives it. All that has to be done is to ChuckK the `MidiIn` object to `now` and it will wait until a message is received to go further in the program.

```
while(true){
    // Use the MIDI Event from MidiIn
    min => now;
    while( min.recv(msg) ){
        <<<msg.data1,msg.data2,msg.data3,"MIDI Message">>>;
    }
}
```

Midi Output

If you have a synthesizer that you want to trigger from ChuckK you can send MIDI messages to it simply. All you have to do is have a `MidiMsg` that will serve as the container for your data and then you will hand it to `MidiOut` using the `.send(MidiMsg)` function.

```
MidiOut mout;
MidiMsg msg;
// check if port is open
if( !mout.open( 0 ) ) me.exit();

// fill the message with data
144 => msg.data1;
52  => msg.data2;
100 => msg.data3;
// bugs after this point can be sent
// to the manufacturer of your synth
mout.send( msg );
```

Writing To Disk

Here is a brief tutorial on writing to disk...

— 1. recording your ChuckK session to file is easy!

example: you want to record the following:

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

all you's got to do is ChuckK a shred that writes to file:

```
%>chuck foo.ck bar.ck rec.ck
```

no changes to existing files are necessary. an example `rec.ck` can be found in `examples/`, this guy/gal writes to “foo.wav”. edit the file to change. if you don't want to worry about overwriting the same file everytime, you can:

```
%>chuck foo.ck bar.ck rec2.ck
```

`rec2.ck` will generate a file name using the current time. You can change the prefix of the filename by

```
"data/session" => w.autoPrefix;
```

`w` is the `WvOut` in the patch.

Oh yeah, you can of course chuck the `rec.ck` on-the-fly...

from terminal 1

```
%>chuck --loop
```

from terminal 2

```
%>chuck + rec.ck
```

— 2. silent mode

you can write directly to disk without having real-time audio by using `--silent` or `-s`

```
%>chuck foo.ck bar.ck rec2.ck -s
```

this will not synchronize to the audio card, and will generate samples as fast as it can.

— 3. start and stop

you can start and stop the writing to file by:

```
1 => w.record;  // start
0 => w.record;  // stop
```

as with all thing ChuckKian, this can be done sample-synchronously.

— 4. another halting problem

what if I have infinite time loop, and want to terminate the VM, will my file be written out correctly? the answer:

Ctrl-C works just fine.

ChuckK STK module keeps track of open file handles and closes them even upon abnormal termination, like Ctrl-C. Actually for many, Ctrl-C is the natural way to end your ChuckK session. At any rate, this is quite ghetto, but it works. As for seg-faults and other catastrophic events, like computer catching on fire from ChuckK exploding, the file probably is toast.

hmmmm, toast...

— 5. the silent sample sucker strikes again

as in rec.ck, one patch to write to file is:

```
dac => gain g => WvOut w => blackhole;
```

the blackhole drives the WvOut, which in turns sucks samples from gain and then the dac. The WvOut can also be placed before the dac:

```
noise n => WvOut w => dac;
```

The WvOut writes to file, and also pass through the incoming samples.

Stereo

At long last, ChuckK is stereo! Accessing the stereo capabilities of ChuckK are relatively simple. **dac** now has three access points.

```
ugen u;
// standard mono connection
u => dac;

// simple enough!
u => dac.left;
u => dac.right;
```

adc functionality mirrors **dac**.

```
// switcheroo
adc.right => dac.left;
adc.left => dac.right;
```

If you have your great UGen network going and you want to throw it somewhere in the stereo field you can use **pan2**. You can use the **.pan** function to move your sound between left (-1) and right (1).

```
// this is a stereo connection to the dac
sinosc s => pan2 p => dac;
1 => p.pan;
while(1::second => now){
    // this will flip the pan from left to right
    p.pan() * -1. => p.pan;
}
```

You can also mix down your stereo signal to a mono signal using the **mix2** object.

```
adc => mix2 m => dac.left;
```

If you remove the **mix2** in the chain and replace it with a gain object it will act the same way. When you connect a stereo object to a mono object it will sum the inputs. You will get the same effect as if you connect two mono signals to the input of another mono signal.

Reference

CHAPTER 4

Overview

ChuckK is a strongly-typed, strongly-timed, concurrent audio and multimedia programming language. It is compiled into virtual instructions, which is immediately run in the ChuckK Virtual Machine. This guide documents the features of the Language, Compiler, and Virtual Machine for a ChuckK programmer.

running ChuckK

Some quick notes:

- you can install ChuckK (see build instructions) or run it from a local directory.
- ChuckK is a command line application called `chuck`. (also see the Audicle)
- use command line prompt/terminal to run ChuckK: (e.g. Terminal or xterm on OS X, cmd or cygwin on Windows, on Linux, you surely have your preferred terminal.)
- this is a quick overview, see VM options for a more complete guide to command line options.

To run ChuckK with a program/patch called `foo.ck` simply run `chuck` and then the name of the file:

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

To run ChuckK with multiple patches concurrently (or the same one multiple times):

```
%>chuck foo.ck bar.ck bar.ck boo.ck
```

There are several flags you can specify to control how ChuckK operates, or to find out about the system. For example, the following probes the audio system and prints out all available audio devices and MIDI devices. You may then refer to them (by number usually) from the command line or from your program. (again, see VM Options for a complete list)

```
%>chuck --probe
```

ChuckK can be run in a different terminal as a host/listener that patches may be sent to. The server should invoke the `--loop` flag to specify that the virtual machine should not halt automatically (when the current programs exit).

```
%>chuck --loop
```

(See the guide to On-the-fly Programming for more information)

If a ChuckK listener is running, we can (from a second terminal) send a program/patch to to the listener by using the `+` command line option:

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

Similarly, you can use `-` and `=` to remove/replace a patch in the listener, and use `^` to find out the status. Again, see On-the-fly Programming for more information.

To run most of the code or examples in this language specification, you only need to use the basic chuck program.

comments

Comments are sections of code that are ignored by a compiler. These help other programmers (and yourself) interpret your code. Double slashes indicate to the compiler to skip the rest of the line. For now, there is no block commenting.

```
// this is a comment
int foo; // another comment
```

debug print

For the time being, `stdout` and `chout` have been temporarily disabled for the present release. In their place we have provided a debug print syntax:

```
// prints out value of expression
<<< expression >>>;
```

This will print the values and types of any expressions placed within them. This debug print construction may be placed around any non-declaration expression (non l-value) and will not affect the execution of the code. Expressions which represent an object will print the value of that object's reference address:

```
// assign 5 to a newly declared variable
5 => int i;
// prints "5 : (int)"
<<<i>>>;

// prints "hello! : (string)"
<<<"hello!">>>; //prints "hello! : (string)"

// prints "3.5 : (float)"
<<<1.0 + 2.5 >>>=> float x;
```

For more formatted data output, a comma-separated list of expressions will print only their respective values (with one space between):

```
// prints "the value of x is 3.5" (x from above)
<<<"the value of x is" , x >>>;

// prints "4 + 5 is 9"
<<<"4 + 5 is", 4 + 5>>>;

// prints "here are 3 random numbers ? ? ?"
<<<"here are 3 random numbers",
    std.rand2(0,9),
    std.rand2(0,9),
    std.rand2(0,9) >>>;
```

reserved words

(primitive types)

- int
- float

- time
- dur
- void
- same (unimplemented)

(control structures)

- if
- else
- while
- until
- for
- break
- continue
- return
- switch (unimplemented)

(class keywords)

- class
- extends
- public
- static
- pure
- this
- super (unimplemented)
- interface (unimplemented)
- implements (unimplemented)
- protected (unimplemented)
- private (unimplemented)

(other chuck keywords)

- function
- fun
- spork
- const
- new

(special values)

- now

- true
- false
- maybe
- null
- NULL
- me
- pi

(special : default durations)

- samp
- ms
- second
- minute
- hour
- day
- week

(special : global ugens)

- dac
- adc
- blackhole

(operators)

- +
- -
- *
- /
- %
- =>
- =<
- ||
- &&
- ==
- ^
- &
- |
- ~
- ::
- ++

- --
- >
- <
- @=>
- +=>
- -=>
- *=>
- /=>
- %=>

(reserved)

- then
- before
- after
- at

CHAPTER 5

Types, Values, and Variables

ChuckK is a strongly-typed language, meaning that types are resolved at compile-time. However, it is not quite statically-typed, because the compiler/type system is a part of the ChuckK virtual machine, and is a runtime component. This type system helps to impose precision and clarity in the code, and naturally lends to organization of complex programs. At the same time, it is also dynamic in that changes to the type system can take place (in a well-defined manner) at runtime. This dynamic aspect forms the basis for on-the-fly programming.

This section deals with types, values, and the declaration and usage of variables. As in other strongly-typed programming languages, we can think of a type as associated behaviors of data. (For example, an 'int' is a type that means integer, and adding two integer is defined to produce a third integer representing the sum.) Classes and objects allow us to extend the type system with our own custom types, but we won't cover them in this section. We will focus mainly on primitive types here, and leave the discussion of more complex types for classes and objects.

primitive types

The primitive, or intrinsic types are those which are simple datatypes (they have no additional data attributes). Objects are not primitive types. Primitive types are passed by value. Primitive types cannot be extended. The primitive types in ChuckK are:

- int : integer (signed)

- float : floating point number (in ChuckK, a float is by default double-precision)
- time : ChuckKian time
- dur : ChuckKian duration
- void : (no type)

For a summary of operations on these types, go to operations and operators.

All other types are derived from 'object', either as part of the ChuckK standard library, or as a new class that you create. For specification, go to classes and objects.

values (literals)

Literal values are specified explicitly in code and are assigned a type by the compiler. The following are some examples of literal values:

int:

```
42
```

int (hexadecimal):

```
0xaf30
```

float:

```
1.323
```

dur:

```
5.5::second
```

In the above code, second is an existing duration variable. For more on durations, see the manipulating time section.

variables

Variables are locations in memory that hold data. Variables have to be declared in ChuckK before they are used. For example, to declare a variable of type int called foo:


```
// declare an 'int' called 'foo'
int foo;
```

We can assign a value to an existing variable by using the ChuckK operator (`=>`). This is one of the most commonly used operators in ChuckK, it's the way to do work and take action! We will discuss this family of operators in operators and operations.

```
// assign value of 2 to 'foo'
2 => foo;
```

It is possible to combine the two statements into one:

```
// assign 2 to a new variable 'foo' of type 'int'
2 => int foo;
```

To use a variable, just refer to it by name:

```
// debug-print the value of foo
<<< foo >>>;
```

To update the value of foo, for example:

```
// multiply 'foo' by 10, assign back to 'foo'
foo * 10 => foo;
```

You can also do the above using a `*=>`(mult-chuck):

```
// multiply 'foo' by 10, and then assign to 'foo'
10 *=> foo;
```

Here is an example of a duration:

```
// assign value of '5 seconds' to new variable bar
5::second => dur bar;
```

Once you have bar, you can inductively use it to construct new durations:

```
// 4 bar, a measure?
4::bar => dur measure;
```

Since time is central to programming ChuckK, it is important to understand time, dur, the relationship and operations between them. There is more information in the manipulating time section.

reference types

Reference types are types which inherit from the object class. Some default reference types include:

- Object : base type that all classes inherit from (directly or indirectly)
- array : N-dimensional ordered set of data (of the same type)
- Event : fundamental, extendable, synchronization mechanism
- UGen : extendable unit generator base class
- string : string (of characters)

New classes can be created. All classes are reference types. We will leave the full discussion to the objects and classes section.

CHAPTER 6

Arrays

Arrays are used represent N-dimensional ordered sets of data (of the same type). This sections specifies how arrays are declared and used in ChucK. Some quick notes:

- arrays can be indexed by integer (0-indexed).
- any array can also be used as an associative map, indexed by strings.
- it is important to note that the integer-indexed portion and the associative portion of an array are stored in separate namespaces
- arrays are objects (see objects and classes), and will behave similarly under reference assignment and other operations common to objects.

declaration

Arrays can be declared in the following way:

```
// declare 10 element array of int, called foo
int foo[10];

// since array of int (primitive type), the contents
// are automatically initialized to 0
```

Arrays can also be initialized:

```
// chuck initializer to array reference
[ 1, 1, 2, 3, 5, 8 ] @=> int foo[];
```

In the above code, there are several things to note.

- initializers must contain the same or similar types. the compiler will attempt to find the highest common base type of all the elements. if no such common element is found, an error is reported.
- the type of the initializer [1, 1, 2, 3, 5, 8] is int[]. the initializer is an array that can be used directly when arrays are expected.
- the at-chuck operator (@=>) means assignment, and is discussed at length in operators and operations.
- int foo[] is declaring an empty reference to an array. the statement assigns the initializer array to foo.
- arrays are objects.

When declaring an array of objects, the objects inside the array are automatically instantiated.

```
// objects in arrays are automatically instantiated
Object group[10];
```

If you only want an array of object references:

```
// array of null object references
Object @ group[10];
```

Check here more information on object declaration and instantiation in Chuck.

The above examples are 1-dimensional arrays (or vectors). Coming up next are multi-dimensional arrays!

multi-dimensional arrays

It is possible (and equally easy) to declare multi-dimensional arrays:

```
// declare 4 by 6 by 8 array of float
float foo3D[4][6][8];
```

Initializers work in similar ways:

```
// declare 2 by 2 array of int
[ [1,3], [2,4] ] @=> int bar[][];
```

In the above code, note the two `[]` since we make a matrix.

lookup

Elements in an array can be accessed using `[]` (in the appropriate quantities).

```
// declare an array of floats
[ 3.2, 5.0, 7 ] @=> float foo[];

// access the 0th element (debug print)
<<< foo[0] >>>; // hopefully 3.2

// set the 2nd element
8.5 => foo[2];
```

Looping over the elements of an array:

```
// array of floats again
[ 1, 2, 3, 4, 5, 6 ] @=> float foo[];

// loop over the entire array
for( 0 => int i; i < foo.cap(); i++ )
{
    // do something (debug print)
    <<< foo[i] >>>;
}
```

Accessing multi-dimensional array:

```
// 2D array
int foo[4][4];

// set an element
10 => foo[2][2];
```

If the index exceeds the bounds of the array in any dimension, an exception is issued and the current shred is halted.

```
// array capacity is 5
int foo[5];

// this should cause ArrayOutOfBoundsException
// access element 6 (index 5)
<<< foo[5] >>>;
```

a longer program: otf_06.ck from examples:

```
// the period
.5::second => dur T;
// synchronize to period (for on-the-fly synchronization)
T - (now % T) => now;

// our patch
sinosc s => JCRev r => dac;
// initialize
.05 => s.gain;
.25 => r.mix;

// scale (pentatonic; in semitones)
[ 0, 2, 4, 7, 9 ] @=> int scale[];

// infinite time loop
while( true )
{
    // pick something from the scale
    scale[ math.rand2(0,4) ] => float freq;
    // get the final freq
    std.mtof( 69 + (std.rand2(0,3)*12 + freq) ) => s.freq;
    // reset phase for extra bandwidth
    0 => s.phase;

    // advance time
    if( std.randf() > -.5 ) .25::T => now;
    else .5::T => now;
}
```

associative arrays

Any array can be used also as an associative array, indexed on strings. Once the regular array is instantiated, no further work has to be done to make it associative as well - just start using it as such.

```
// declare regular array (capacity doesn't matter so much)
float foo[4];

// use as int-based array
2.5 => foo[0];

// use as associative array
4.0 => foo["yoyo"];

// access as associative (print)
<<< foo["yoyo"] >>>;

// access empty element
<<< foo["gaga"] >>>; // -> should print 0.0
```

It is important to note (again), that the address space of the integer portion and the associative portion of the array are completely separate. For example:

```
// declare array
int foo[2];

// put something in element 0
10 => foo[0];

// put something in element "0"
20 => foo["0"];

// this should print out 10 20
<<< foo[0], foo["0"] >>>;
```

The capacity of an array relates only to the integer portion of it. An array with an integer portion of capacity 0, for example, can still have any number of associative indexes.

```
// declare array of 0 capacity
int foo[0];
```

```
// put something in element "here"
20 => foo["here"];

// this should print out 20
<<< foo["here"] >>>

// this should cause an exception
<<< foo[0] >>>
```

Note: The associative capacity of an array is not defined, so objects used in the associative namespace must be explicitly instantiated, in contrast to those in the integer namespace

Accessing an uninstantiated element of the associate array will return a null reference. Please check the class documentation page for an explanation of Chuck objects and references.

```
class Item {
    float weight;
}

Item box[10];

// integer indices ( up to capacity ) are pre-instantiated.
1.2 => box[1].weight;

// instantiate element "lamp";
new Item @=> box["lamp"];

// access allowed to "lamp"
2.0 => box["lamp"].weight;

// access causes a NullPointerException
2.0 => box["sweater"].weight;
```

array assignment

Arrays are objects. So when we declare an array, we are actually (1) declaring a reference to array (reference variable) and (2) instantiating a new array and reference assigned to the variable. A (null) reference is a reference variable that points to no object or null. A null reference to an array can be created in this fashion:

```
// declare array reference (by not specifying a capacity)
```



```

int foo[];

// we can now assign any int[] to foo
[ 1, 2, 3 ] @=> foo;

// print out 0th element
<<< foo[0] >>>;

```

This is also useful in declaring functions that have arrays as arguments or return type.

```

// our function
fun void print( int bar[] )
{
    // print it
    for( 0 => int i; i < bar.cap(); i++ )
        <<< bar[i] >>>;
}

// we can call the function with a literal
print( [ 1, 2, 3, 4, 5 ] );

// or can we can pass a reference variable
int foo[10];
print( foo );

```

Like other objects, it is possible make multiple references to a single array. Like other objects, all assignments are reference assignments, meaning the contents are NOT copied, only a reference to array is duplicated.

```

// our single array
int the_array[10];

// assign reference to foo and bar
the_array => int foo[] => int bar[];

// (the_array, foo, and bar now all reference the same array)

// we change the_array and print foo...
// they reference the same array, changing one is like changing the other
5 => the_array[0];
<<< foo[0] >>>; // should be 5
\eng{verbatim}

```

It is possible to reference sub-sections of multi-dimensional arrays.

```
\begin{verbatim}
// a 3D array
int foo3D[4][4][4];

// we can make a reference to a sub-section
foo3D[2] => int bar[][];

// (note that the dimensions must add up!)
```

CHAPTER 7

Operators and Operations

Operations on data are achieved through operators. This section defines how operators behave on various datatypes. You may have seen many of the operators in other programming languages (C/Java). Some others are native to ChuckK. We start with the family of ChuckK operators.

`=>` (the ChuckK operator)

The ChuckK operator (`=>`) is a massively overloaded operator that, depending on the types involved, performs various actions. It denotes action, can be chained, and imposes and clarifies order (always goes from left to right). The ChuckK operator is the means by which work is done in ChuckK. Furthermore, the ChuckK operator is not a single operator, but a family of operators.

`=>` (foundational ChuckK operator)

We start with the standard, plain-vanilla ChuckK operator (`=>`). It is left-associative (all ChuckK operators are), which allows us to specify any ordered flow of data/tasks/modules (such as unit generator connection) from left-to-right, as in written (English) text. What `=>` does depends on the context. It always depends on the type of the entity on the left (the chucker) and the one on the right (the chuckee), and it sometimes also depends on the nature of the entity (such as whether it is a variable or not).

Some examples:

```
// a unit generator patch - the signal flow is apparent
// (in this case, => connects two unit generators)
```

```
sinosc b => gain g => biquad f => dac;

// add 4 to foo, chuck result to new 'int' variable 'bar'
// (in this case, => assigns a value to a variable (int))
4 + foo => int bar;

// chuck values to a function == function call
// (same as math.rand2f( 30, 1000))
( 30, 1000 ) => math.rand2f;
```

There are many other well-defined uses of the ChuckK operator, depending on the context.

@=> (explicit assignment ChuckK operator)

In ChuckK, there is no standard assignment operator (=), found in many other programming languages. Assignment is carried out using ChuckK operators. In the previous examples, we have used => for assignment:

```
// assign 4 to variable foo
4 => int foo;

// assign 1.5 to variable bar
1.5 => float bar;

// assign duration of 100 millisecond to duh
100::ms => dur duh;

// assign the time "5 second from now" to later
5::second + now => time later;
```

The @=> explicit assignment ChuckK operator behaves exactly the same for the above types (int, float, dur, time). However, the difference is that @=> can also be used for reference assignments of objects (see objects and classes) whereas => only does assignment on primitive types (int, float, dur, time). The behavior of => on objects is completely context-dependent.

```
// using @=> is same as => for primitive types
4 @=> int foo;

// assign 1.5 to variable bar
1.5 @=> float bar;

// (only @=> can perform reference assignment on objects)
```

```
// reference assign moe to larry
// (such that both moe and larry reference the same object)
Object moe @=> Object @ larry;

// array initialization
[ 1, 2 ] @=> int ar[];

// using new
new Object @=> moe;
```

In its own screwed-up way, this is kind of nice because there is no confusion between assignment (@=> or =>) and equality (==). In fact the following is not a valid ChuckK statement:

```
// not a valid ChuckK statement!
int foo = 4;
```

+=> -=> *=> /=> etc. (arithmetic ChuckK operators)

These operators are used with variables (using 'int' and 'float') to perform one operation with assignment.

```
// add 4 to foo and assign result to foo
foo + 4 => foo;

// add 4 to foo and assign result to foo
4 +=> foo;

// subtract 10 from foo and assign result to foo
// remember this is (foo-10), not (10-foo)
10 -=> foo;

// 2 times foo assign result to foo
2 *=> foo;

// divide 4 into foo and assign result to foo
// again remember this is (foo/4), not (4/foo)
4 /=> foo;
```

It is important to note the relationship between the value and variable when using -=> and /=>, since these operations are not commutative.

```
// mod foo by T and assign result to foo
```

```
T %=> foo;

// bitwise AND 0xff and bar and assign result to bar
0xff &=> bar;

// bitwise OR 0xff and bar and assign result to bar
0xff |=> bar;
```

That's probably enough operator abuse for now...

+ - * / (arithmetic)

Can you add, subtract, multiply and divide? So can Chuck!

```
// divide (and assign)
16 / 4 => int four;

// multiply
2 * 2 => four;

// add
3 + 1 => four;

// subtract
93 - 89 => four;
```

cast

ChuckK implicitly casts int values to float when float is expected, but not the other around. The latter could result in a loss of information and requires an explicit cast.

```
// adding float and int produces a float
9.1 + 2 => float result;

// however, going from float to int requires cast
4.8 $ int => int foo; // foo == 4

// this function expects two floats
math.rand2f( 30.0, 1000.0 );

// this is ok because of implicit cast
math.rand2f( 30, 1000 );
```

% (modulo)

The modulo operator % computes the remainder after integer, floating point, duration, and time/duration division.

```
// 7 mod 4 (should yield 3)
7 % 4 => int result;

// 7.3 mod 3.2 floating point mod (should yield .9)
7.3 % 3.2 => float resultf;

// duration mod
5::second % 2::second => dur foo;

// time/duration mod
now % 5::second => dur bar;
```

the latter (time/duration mod) is one of many ways to dynamically synchronize timing in shreds. the examples otf_01.ck through otf_07.ck (see under examples) make use of this to on-the-fly synchronize its various parts, no matter when each shred is added to the virtual machine:

```
// define period (agreed upon by several shreds)
.5::second => dur T;

// compute the remainder of the current period ...
// and advance time by that amount
T - (now % T) => now;

// when we reach this point, we are synchronized to T period boundary

// the rest of the code
// ...
```

This is one of many ways to compute and reason about time in ChuckK. The appropriate solution(s) in each case depends on the intended functionality. Have fun!

&& || == !=> <= > >= (logic)

Logical operators - each of these need two operands. The result is an integer value of 0 or 1.

- && : and
- ||: or
- == : equals

- != : does not equal
- >: greater than
- >= : greater than or equal to
- <: less than
- <= : less than or equal to

```
// test some universal truths
if( 1 <= 4 && true )
    <<<"horray">>>;
```

>> << & | ^ (bitwise)

These are used on int values at the bit level, often for bit masking.

- >>: shift bits right (8 >>1 = 4)
- <<: shift bits left (8 <<1 = 16)
- & : bitwise AND
- |: bitwise OR
- ^: bitwise XOR

++ --(inc / dec)

Values may be incremented or decremented by appending the ++ or --operators.

```
4 => int foo;
foo++;
foo--;
```

! + - new (unary)

These operators come before one operand.

```
// logical invert
if( !true == false )
    <<<"yes">>>;

// negative
-1 => int foo;

// instantiate object
new object @=> object @ bar;
```


CHAPTER 8

Control Structures

ChuckK includes standard control structures similar to those in most programming languages. A condition (of type 'int') is evaluated and then a proceeding block is potentially executed. Blocks are separated either by semicolons or by curly brackets .

if / else

The if statement executes a block if the condition is evaluated as non-zero.

```
if( condition )
{
    // insert code here
}
```

In the above code, condition is any expression that evaluates to an int.

The else statement can be put after the if block to handle the case where the condition evaluates to 0.

```
if( condition )
{
    // your code here
}
```

```
}  
else  
{  
    // your other code here  
}
```

If statements can be nested.

while

The while statement is a loop that repeatedly executes the body as long as the condition evaluates as non-zero.

```
// here is an infinite loop  
while( true )  
{  
    // your code loops forever!  
  
    // (sometimes this is desirable because we can create  
    // infinite time loops this way, and because we have  
    // concurrency)  
}
```

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

```
do {  
    // your code executes here at least once  
} while( condition );
```

A few more points:

- while statements can be nested.
- see break/continue for additional control over your loops

until

The until statement is the opposite of while, semantically. A until loop repeatedly executes the body until the condition evaluates as non-zero.

```
// an infinite loop
until( false )
{
    // your great code loops forever!
}
```

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

```
do {
    // your code executes here at least once
} until( condition );
```

A few more points:

- until statements can be nested.
- see break/continue for additional control over your loops

for

A loop that iterates a given number of times. A temporary variable is declared that keeps track of the current index and is evaluated and incremented at each iteration.

```
// for loop
for( 0 => int foo; foo < 4 ; foo++ )
{
    // debug-print value of 'foo'
    <<<foo>>>;
}
```

break / continue

Break allows the program flow to jump out of a loop.

```
// infinite loop
while( 1 )
{
    if( condition )
```

```
        break;
    }
```

Continue allows a loop to continue looping but not to execute the rest of the block for the iteration where continue was executed.

```
// another infinite loop
while( 1 )
{
    // check condition
    if( condition )
        continue;

    // some great code that may get skipped (if continue is taken)
}
```

CHAPTER 9

Functions

Functions provide ways to break up code/common tasks into individual units. This helps to promote code re-usability and readability.

writing

Declare functions with the keyword `fun` (or `function`) followed by the return type and then the name of the function. After the name of the function parentheses must be opened to declare the types of the input arguments.

```
// define function call 'funk'
fun void funk( int arg )
{
    // insert code here
}
```

The above function returns no values (the return type is `void`). If the function is declared to return any other type of values, the function body must return a value of the appropriate type.

```
// define function 'addOne'
fun int addOne(int x)
```

```
{
    // result
    return x + 1;
}
```

calling

To call a function use the name of the function with appropriate arguments.

```
// define 'hey'
fun int hey( int a, int b )
{
    // do something
    return a + b;
}

// call the function; store result
hey( 1, 2 ) => int result;
```

You can also use the ChuckK operator to call functions!

```
// call hey
( 1, 2 ) => hey => int result;

// same
hey( 1, 2 ) => int result;

// several in a row
( 10, 3000 ) => std.math => std.abs => float foo;

// same
std.abs( std.math( 10, 3000 ) ) => float foo;
```

overloading

Overloading a function allows functions with the same name to be defined with different arguments. The function must be written in separate instances to handle the input, and the return type must agree.

```
// funk( int )
```

```
fun int add(int x)
{
    return x + x;
}

// funk( int, int )
fun int add(int x, int y)
{
    return x + y;
}

// compiler automatically choose the right one to call
add( 1 ) => int foo;
add( 1, 2 ) => int bar;
```

CHAPTER 10

Concurrency and Shreds

ChuckK is able to run many processes concurrently (the process behave as if they are running in parallel). A ChucKian process is called a shred. To spork a shred means creating and adding a new process to the virtual machine. Shreds may be sporked from a variety of places, and may themselves spork new shreds.

ChuckK supports sample-synchronous, non-preemptive concurrency. Via the timing mechanism, any number of programs/shreds can be automatically shredded and synchronized use the timing information. A shredder in the virtual machine does the shredding. The concurrency is 'sample-synchronous', meaning that inter-process audio timing is guaranteed to be precise to the sample. Note that each process/shred does not necessarily need to know about each other - it only has to deal with time locally. The virtual machine will make sure things happen correctly "across the board". Finally, concurrency - like timing - is deterministic in ChuckK.

The simplest way to to run shreds concurrently is to specify them on the command line:

```
%>chuck foo.ck bar.ck boo.ck
```

The above command runs chuck on foo.ck, bar.ck, and boo.ck concurrently. There are other ways to run shreds concurrently (see on-the-fly programming commands). Next, we show how to create new shreds from within ChuckK programs.

sporking shreds (in code)

To *spork* means to shredule a new shred.

To spork a shred, use the spork keyword/operator:

- spork dynamically sporks shred from a function call
- this operation is sample-synchronous, the new shred is shreduled to execute immediately
- the parent shred continues to execute, until time is advanced (see manipulating time) or until the parent explicitly yields (see next section).
- in the current implementation, when a parent shred exits, all child shreds all exit (this behavior will be enhanced in the future.)
- sporking a functions returns reference to the new shred, note that this operation does not return what functions returns - the ability to get back the return value at some later point in time will be provided in a future release.

```
// define function go()
fun void go()
{
    // insert code
}

// spork a new shred to start running from go()
spork ~ go();

// spork another, store reference to new shred in offspring
spork ~ go() => Shred @ offspring;
```

a slightly longer example:

```
// define function
fun void foo( string s )
{
    // infinite time loop
    while( true )
    {
        // print s
        <<< s >>>;
        // advance time
        500::ms => now;
    }
}
```

```
// spork shred, passing in "you" as argument to foo
spork ~ foo( "you" );
// advance time by 250 ms
250::ms => now;
// spork another shred
spork ~ foo( "me" );

// infinite time loop - to keep child shreds around
while( true )
    1::second => now;
```

also see function section for more information on working with functions.

the 'me' keyword

The `me` keyword (type `Shred`) refers the current shred.

Sometimes it is useful to suspend the current shred without advancing time, and let other shreds shreduled for the current time to execute. `me.yield()` does exactly that. This is often useful immediately after sporking a new shred, and you would like for that shred to have a chance to run but you do not want to advance time yet for yourself.

```
// spork shred
spork ~ go();

// suspend the current shred ...
// ... give other shreds (shreduled for 'now') a chance to run
me.yield();
```

It may also be useful to exit the current shred. For example if a MIDI device fails to open, you may exit the current shred.

```
// make a MidiIn object
MidiIn min;

// try to open device 0 (chuck --probe to list all device)
if( !min.open( 0 ) )
{
    // print error message
    <<< "can't open MIDI device" >>>;
}
```

```
    // exit the current shred
    me.exit();
}
```

You can get the shred id:

```
// print out the shred id
<<< me.id(); >>>;
```

These functions are common to all shreds, but `yield()` and `exit()` are commonly used with the current shred.

using `machine.add()`

`machine.add(string path)` takes the path to a chuck program, and sporks it. Unlike `spork`, there is no parent-child relationship between the shred that calls the function and the new shred that is added. This is useful for dynamically running stored programs.

```
// spork "foo.ck"
machine.add( "foo.ck" );
```

Presently, this returns the id of the new shred, not a reference to the shred. This will likely be changed in the future.

Similarly, you can remove shreds from the virtual machine.

```
// add
machine.add( "foo.ck" ) => int id;

// remove shred with id
machine.remove( id );

// add
machine.add( "boo.ck" ) => id

// replace shred with "bar.ck"
machine.replace( id, "bar.ck" );
```

inter-shred communication

Shreds spawned in the same file can share the same global variables. They can use time and events to synchronize to each other. (see events) Shreds spawned from different files can share data (including events). For now, this is done through a public class with static data (see classes). Static data is not completely implemented! We will fix this very soon!

CHAPTER 11

Time and Timing

ChuckK is a **strongly-timed** language, meaning that time is fundamentally embedded in the language. ChuckK allows the programmer to explicitly *reason* about time from the code. This gives extremely flexible and precise control over time and (therefore) sound synthesis.

In ChuckK:

- time and duration are native types in the language
- keyword `now` holds the current logical time
- time is *advanced* (and can only be advanced) by explicitly manipulating **now**
- you have flexible and precise control

time and duration

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**).

```
// a duration of one second
1::second => dur foo;

// a point in time (duration of foo from now)
now + foo => time later;
```

Later in this section, we outline the various arithmetic operations to perform on time and duration.

Durations can be used to construct new durations, which then be used to inductively construct yet other durations. For example:

```
// .5 second is a quarter
.5::second => dur quarter;

// 4 quarters is whole
4::quarter => dur whole;
```

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

Use these to represent any duration.

```
// the duration of half a sample
.5::samp => dur foo;

// 20 weeks
20::week => dur waithere;

// use in combination
2::minute + 30::second => dur bar;

// same value as above
2.5::minute => dur bar;
```

operations on time and duration (arithmetic)

In ChuckK, there are well-defined arithmetic operations on values of type **time** and **dur**.

example 1 (time offset):

```
// time + dur yields time
now + 10::second => time later;
```

example 2 (time subtraction):

```
// time - time yields dur
later - now => dur D;
```

example 3 (addition):

```
// dur + dur yields dur
10::second + 100::samp => dur foo;
```

example 4 (subtraction):

```
// dur - dur yields dur
10::second - 100::samp => dur bar;
```

example 5 (division):

```
// dur / dur yields number
10::second / 20::ms => float n;
```

example 6 (time mod):

```
// time mod dur yields dur
now % 1::second => dur remainder;
```

example 7 (synchronize to period):

```
// synchronize to period of .5 second
.5::second => dur T;
T - (now % T) => now;
```

example 8 (comparison on time):

```
// compare time and time
if( t1 < t2 )
    // do something...
```

example 9 (comparison on duration):

```
// compare dur and dur
if( 900::ms < 1::second )
    <<< "yay!" >>>;
```

the keyword ‘now’

The keyword **now** is the key to reasoning about and controlling time in ChuckK.

Some properties of **now** include:

- **now** is a special variable of type time.
- **now** holds the current ChuckK time (when read).
- modifying **now** has the side effects of:
 - advancing time (see below);
 - suspending the current process (called shred) until the desired time is reached - allowing other shreds and audio synthesis to compute;
 - the value of **now** only changes when it is explicitly modified.

(also see next section on advancing time).

Example:

```
// compute value that represents "5 seconds from now"
now + 5::second => time later;

// while we are not at later yet...
while( now < later )
{
    // print out value of now
    <<< now >>>;

    // advance time by 1 second
    1::second => now;
}
```

advancing time

Advancing time allows other shreds (processes) to run and allows audio to be computed in a controlled manner. There are three ways of advancing time in ChuckK:

- chucking (=>) a duration to **now**: this will advance time by that duration.
- chucking (=>) a time to **now**: this will advance time to that point. (note that the desired time must be later than the current time, or at least be equal to it.)
- chucking (=>) an **Event to now**: time will advance until the event is triggered. (also see event)

advancing time by duration

```
// advance time by 1 second
1::second => now;

// advance time by 100 millisecond
100::ms => now;

// advance time by 1 samp (every sample)
1::samp => now;

// advance time by less than 1 samp
.024::samp => now;
```

advancing time by absolute time

```
// figure out when
now + 4::hour => time later;

// advance time to later
later => now;
```

A time chunked to now will have ChuckK wait until the appointed time. ChuckK never misses an appointment (unless it crashes)! Again, the time chunked to now must be greater than or equal to now, otherwise an exception is thrown.

advancing time by event

```
// wait on event
e => now;
```

See events for a more complete discussion of using events!

The advancement of time can occur at any point in the code.

```
// our patch: sine oscillator -> dac
sinosc s => dac;

// infinite time loop
while( true )
{
```

```
// randomly choose frequency from 30 to 1000
std.rand2f( 30, 1000 ) => s.freq;

// advance time by 100 millisecond
100::ms => now;
}
```

Furthermore, there are no restrictions (other than underlying floating point precision) on how much time is advanced. So it is possible to advance time by a microsecond, a samp, 2 hours, or 10 years. The system will behave accordingly and deterministically.

This mechanism allows time to be controlled at any desired rate, according to any programmable pattern. With respect to sound synthesis, it is possible to control any unit generator at literally any rate, even sub-sample rate.

The power of the timing mechanism is extended by the ability to write parallel code, which is discussed in concurrency and shreds.

CHAPTER 12

Events

In addition to the built-in timing mechanisms for internal control, ChuckK has an event class to allow exact synchronization across an arbitrary number of shreds.

what they are

ChuckK events are a native class within the ChuckK language. We can create an event objects, and then chuck (=;) that event to now. The event places the current shred on the event's waiting list, suspends the current shred (letting time advance from that shred's point of view). When the event is triggered, one or more of the shreds on its waiting list is shredded to run immediately. This trigger may originate from another ChuckK shred, or from activities taking place outside the Virtual Machine (MIDI, OSC, or IPC).

```
// declare event
Event e;

// function for shred
fun void eventshred( Event event, string msg )
{
    // infinite loop
    while ( true )
```

```

    {
        // wait on event
        event => now;
        // print
        <<<message>>>;
    }
}

// create shreds
spork ~ eventshred ( e, "fee" );
spork ~ eventshred ( e, "fi" );
spork ~ eventshred ( e, "fo" );
spork ~ eventshred ( e, "fum" );

// infinite time loop
while ( true )
{
    // either signal or broadcast
    if( maybe )
    {
        <<<"signaling...">>>;
        e.signal();
    }
    else
    {
        <<<"broadcasting...">>>;
        e.broadcast();
    }

    // advance time
    0.5::second => now;
}

```

use

Chucking an event to now suspends the current shred, letting time advance:

```

// declare Event
Event e;

// ...

```

```
// wait on the event
e => now;

// after the event is trigger
<<< "I just woke up" >>>;
```

as shown above, events can be triggered in two ways, depending on the desired behavior.

```
// signal one shred waiting on the event e
e.signal();
```

signal() releases the first shred in that event's queue, and shredule it to run at the current time, respecting the order in which shreds were added to the queue.

```
// wake up all shreds waiting on the event e
e.broadcast();
```

broadcast() releases all shreds queued by that event, in the order they were added, and at the same instant in time

The released shreds are shreduled to run immediately. But of course they will respect other shreds also shreduled to run at the same time. Furthermore, the shred that called signal() or broadcast() will continue to run until it advances time itself, or yield the virtual machine without advancing time. (see me.yield() under concurrency)

MIDI events

ChuckK contains built-in MIDI classes to allow for interaction with MIDI based software or devices.

```
MidiIn min;
MidiMsg msg;
MidiOut mout;

//open midi receiver, exit on fail
if ( min.open(0) ) me.exit();

while( true )
{
    // wait on midi event
    min => now;
```

```
// receive midimsg(s)
while( min.recv( msg ) )
{
    // print content
    <<< msg.data1, msg.data2, msg.data3 >>>;
}

...

```

MidiIn is a subclass of Event, as as such can be Chucked to now. MidiIn then takes a MidiMsg object to its .recv() method to access the MIDI data. As a default, MidiIn events trigger the broadcast() event behavior.

OSC events

In addition to MIDI, ChuckK has OSC communication classes as well:

```
...

//create our OSC receiver
OscRecv orec;
// port 6449
6449 => orec.port;
// start listening (launch thread)
orec.listen();

function void rate_control_shred()
{
    // create an address in the receiver
    // and store it in a new variable.
    orec.event("/sndbuf/buf/rate,f") @=> OscEvent oscdata;

    while ( true )
    {
        oscdata => now; //wait for events to arrive.

        // grab the next message from the queue.
        while( oscdata.nextMesg() != 0 )
        {

```

```

        // getFloat fetches the expected float
        // as indicated in the type string ",f"
        buf.play( oscdata.getFloat() );
        0 => buf.pos;
    }
}

...

```

The `OscRecv` class listens for incoming OSC packets on the specified port. Each instance of `OscRecv` can create `OscEvent` objects using its `event()` method to listen for packets at any valid OSC address pattern.

An `OscEvent` event can then be `ChucKed` to now to wait for messages to arrive, after which the `nextMesg()` and `get{Float|String|Int}()` methods can be used to fetch message data.

creating custom events

Events, like any other class, can be subclassed to add functionality and transmit data:

```

// extended event
class TheEvent extends Event
{
    int value;
}

// the event
TheEvent e;

// handler
fun int hi( TheEvent event )
{
    while( true )
    {
        // wait on event
        event => now;
        // get the data
        <<<e.value>>>;
    }
}

```

```
// spork
spork ~ hi( e );
spork ~ hi( e );
spork ~ hi( e );
spork ~ hi( e );

// infinite time loop
while( true )
{
    // advance time
    1::second => now;

    // set data
    math.rand2( 0, 5 ) => e.value;

    // signal one waiting shred
    e.signal();
}
```


CHAPTER 13

Objects

introduction

Chuck implements an object system that borrows from both C++ and Java conventions. In our case this means:

- You can define custom classes as new types and instantiate objects
- Chuck supports *polymorphic inheritance* (this is the same model used in Java, and also known as virtual inheritance in C++)
- All object variables are *references* (like Java), but instantiation resembles C++. We will discuss this in detail below.
- There is a default class library.
- All objects inherit from the Object class (as in Java)

For the sake of clarity we will define these terms:

- a class is an *abstraction* of data (members) and behavior (methods)
- a class is a *type*.
- an object is an *instantiation* of that class
- a *reference variable* refers indirectly to an object - it is not the object itself. All Chuck *object variables* are reference variables (like in Java).

- similarly, *reference assignment* duplicates a reference to an object and assigns the reference to a reference variable. The object itself is not duplicated. All ChuckK object assignments are reference assignments.

built-in classes

ChuckK has a number of classes defined within the language.

- **Object** : base class to all ChuckK objects.
- **Event** : ChuckK's basic synchronization mechanism; may be extended to create custom Event functionality (discussed here).
- **Shred** : basic abstraction for a non-preemptive ChuckK process.
- **UGen** : base unit generator class (discussed here).

These are some of the more commonly used classes in ChuckK.

working with objects

Let's begin with some examples. For these examples, let's assume Foo is a defined class.

```
// create a Foo object; stored in reference variable bar
Foo bar;
```

The above code does two things:

- a reference variable bar is declared; its type is Foo.
- a new instance of Foo is created, and its reference is assigned to bar.

Note that in contrast to Java, this statement both declares a reference variable *and* instantiates a instance of that class and assigns the reference to the variable. Also note that in contrast to C++, bar is a reference, and does not represent the object itself.

To declare a reference variable that refers to nothing (also called a null reference):

```
// create a null reference to a Foo object
Foo @ bar;
```

The above code only declare a reference and initializes it to null. (random note: the above statement may be read as "Foo at bar")

We can assign a new instance to the reference variable:

```
// assign new instance of Foo to bar
new Foo @=> Foo @ bar;

// (this statement is equivalent to 'Foo bar', above)
```

The code above is exactly equivalent to `Foo bar`; as shown above. The `new` operator creates an instance of a class, in this case `Foo`. The `@=` operator performs the reference assignment. (see Operators chapter for more information on `@=`)

It is possible to make many references to same object:

```
// make a Foo
Foo bar;

// reference assign to duh
bar @=> Foo @ duh;

// (now both bar and duh points to the same object)
```

ChuckK objects are reference counted and garbage collection takes place automatically. (note: this is still being implemented!)

As stated above, a classes may contain data and behavior, in the form of *member variables* and *member functions*, respectively. Members are accessed by using 'dot notation' - *reference.memberdata* and *reference.memberfunc()*. To invoke a member function of an object (assuming class `Foo` has a member function called `compute` that takes two integers and returns an integer):

```
// make a Foo
Foo bar;

// call compute(), store result in boo
Foo.compute( 1, 2 ) => int boo;
```

writing a class

If a class has already been defined in the ChuckK virtual machine (either in the same file or as a public class in a different file) then it can be instantiated similar to primitive types.

Unless declared **public**, class definitions are scoped to the shred and will not conflict with identically named classes in other running shreds.

Classes encapsulate a set of behaviors and data. To define a new object type, the keyword `class` is used followed by the name of that class.

```
// define class X
class X
{
    // insert code here
}
```

If a class is defined as **public**, it is integrated into the central namespace (instead of the local one), and can be instantiated from other programs that are subsequently compiled. There can be at most one public class per file.

```
// define public class MissPopular
public class MissPopular
{
    // ...
}

// define non-public class Flarg
class Flarg
{
    // ...
}

// both MissPopular and Flarg can be used in this file
// only MissPopular can be used from another file
```

We define member data and methods to specify the data types and functionality required of the class. Members, or instance data and instance functions are associated with individual instances of a class, whereas *static* data and functions are only associated with the class (and shared by the instances).

members (instance data + functions)

Instance data and methods are associated with an object.

```
// define class X
class X
{
    // declare instance variable 'm_foo'
    int m_foo;
    // another instance variable 'm_bar'
    float m_bar;
    // yet another, this time an object
    Event m_event;

    // function that returns value of m_foo
    fun int getFoo() { return m_foo; }

    // function to set the value of m_foo
    fun void setFoo( int value ) { value => m_foo; }

    // calculate something
    fun float calculate( float x, float y )
    {
        // insert code
    }

    // print some stuff
    fun void print()
    {
        <<< m_foo, m_bar, m_event >>>;
    }
}

// instantiate an X
X x;

// set the Foo
x.setFoo( 5 );
// print the Foo
<<< x.getFoo() >>>;

// call print
x.print();
```

class constructors

In the initial release, we do not support constructors yet. However, we have a single *pre-constructor*. The code immediately inside a class definition (and not inside any functions) is run every time an instance of that class is created.

```
// define class X
class X
{
    // we can put any ChuckK statements here as pre-constructor

    // initialize an instance data
    109 => int m_foo;

    // loop over stuff
    for( 0 => int i; i < 5; i++ )
    {
        // print out message how silly
        <<< "part of class pre-constructor...", this, i >>>;
    }

    // function
    fun void doit()
    {
        // ...
    }
}

// when we instantiate X, the pre-constructor is run
X x;

// print out m_foo
<<< x.m_foo >>>;
```

static (data + functions)

Static data and functions are associated with a class, and are shared by all instances of that class – in fact, static elements can be accessed without an instance, by using the name of the class: `Classname.element`.

```
// define class X
```

```
class X
{
    // static data
    static int our_data;

    // static function
    fun static int doThatThing()
    {
        // return the data
        return our_data;
    }
}

// do not need an instance to access our_data
2 => X.our_data;
// print out
<<< X.our_data >>>;
// print
<<< X.doThatThing() >>>;

// create instances of X
X x1;
X x2;

// print out their static data - should be same
<<< x1.our_data, x2.our_data >>>;

// change use one
5 => x1.our_data;

// the other should be changed as well
<<< x1.our_data, x2.our_data >>>;
```

inheritance

Inheritance in object-oriented code allows the programmer to take an existing class and extend or alter its functionality. In doing so we can create a taxonomy of classes that all share a specific set of behaviors, while implementing those behaviors in different, yet well-defined, ways. We indicate that a new class inherits from another class using the extends keyword. The class from which we inherit is referred to as the parent class, and the inheriting class is the child class. The Child class receives all of the member data and functions from the parent class, although functions from the

parent class may be overridden (below). Because the children contain the functionality of the parent class, references to instances of a child class may be assigned to a parent class reference type.

For now, access modifiers (public, protected, private) are included but not fully implemented. Everything is public by default.

```
// define class X
class X
{
    // define member function
    fun void doThatThing()
    {
        <<<"Hallo">>>;
    }

    // define another
    fun void hey()
    {
        <<<"Hey!!!">>>;
    }

    // data
    int the_data;
}

// define child class Y
class Y extends X
{
    // override doThatThing()
    fun void doThatThing()
    {
        <<<"No! Get away from me!">>>;
    }
}

// instantiate a Y
Y y;

// call doThatThing
y.doThatThing();

// call hey() - should use X's hey(), since we didn't override
```



```
y.hey();

// data is also inherited from X
<<< y.the_data >>>;
```

Inheritance provides us a way of efficiently sharing code between classes which perform similar roles. We can define a particular complex pattern of behavior, while changing the way that certain aspects of the behavior operate.

```
// parent class defines some basic data and methods
class Xfunc
{
    int x;

    fun int doSomething( int a, int b ) {
        return 0;
    }
}

// child class, which overrides the doSomething function with an addition operation
class Xadds extends X
{
    fun int doSomething ( int a, int b )
    {
        return a + b ;
    }
}

// child class, which overrides the doSomething function with a multiply operation
class Xmuls extends X
{
    fun int doSomething ( int a, int b )
    {
        return a * b;
    }
}

// array of references to Xfunc
Xfunc @ operators[2];

// instantiate two children and assign reference to the array
new Xadds @=> operators[0];
```

```
new Xmults @=> operators[1];

// loop over the Xfunc
for( 0 => int i; i < operators.cap(); i++ )
{
    // doSomething, potentially different for each Xfunc
    <<< operators[i].doSomething( 4, 5 ) >>>;
}
```

because Xmults and Xadds each redefine doSomething(int a, int b) with their own code, we say that they have overridden the behavior of the parent class. They observe the same interface, but have potentially different implementation. This is known as polymorphism.

Overloading

Function overloading in classes is similar to that of regular functions. see functions.

CHAPTER 14

The ChuckK Compiler + Virtual Machine

Let's start with the compiler/virtual machine, both of which runs in the same process. By now, you should have built/installed ChuckK (guide), and perhaps taken the tutorial. This guide is intended to be more complete and referential than the tutorial.

SYNOPSIS (a man-esque page)

usage:

```
chuck -- [ options|commands ] [ +=^ ] file1 file2 file3 ...
[ options ] = halt|loop|audio|silent|dump|nodump|about|
             srate<N>|bufsize<N>|bufnum<N>|dac<N>|adc<N>|
             remote<hostname>|port<N>
[ commands ] = add|remove|replace|status|time|kill
[ +=^ ] = shortcuts for add, remove, replace, status
```

DESCRIPTION

[**source ChuckK files**] : ChuckK can run 1 or more processes in parallel and interactively. The programmer only needs to specify them all on the command line, and they will be compiled and run in the VM. Each input source file (.ck suffix by convention) will be run as a separate 'shred' (user-level ChuckK threads) in the VM. They can 'spork' additional shreds and interact with existing shreds. Thanks to the ChuckK timing mechanism, shreds don't necessarily need to know about each

other in order to be precisely 'shreduled' in time - they only need to keep track of they own time, so to speak.

Additionally, more shreds can be added/removed/replaced manually at run-time, using on-the-fly programming [Wang and Cook 2004] - (see publications and <http://on-the-fly.cs.princeton.edu/>).

[options] :

--halt / -h

(on by default) - tells the vm to halt and exit if there are no more shreds in the VM.

--loop / -l

tells the Chuck VM to continue executing even if there no shreds currently in the VM. This is useful because shreds can be added later on-the-fly. Furthermore, it is legal to specify this option without any input files. For example:

```
%>chuck --loop
```

the above will 'infinite time-loop' the VM, waiting for incoming shreds.

--audio / -a

(on by default) - enable real-time audio output

--silent / -s

disable real-time audio output - computations in the VM is not changed, except that the actual timing is no longer clocked by the real-time audio engine. Timing manipulations (such as operations on 'now') still function fully. This is useful for synthesizing audio to disk or network. Also, it is handy for running a non-audio program.

--dump / +d

dump the virtual instructions emitted to stderr, for all the files after this flag on the command line, until a 'nodump' is encountered (see below). For example:

```
%>chuck foo.ck +d bar.ck
```

will dump the virtual Chuck instructions for bar.ck (only), with argument values, to stderr. --dump can be used in conjunction with --nodump to selectively dump files.

--nodump / -d

(default state) cease the dumping of virtual instructions for files that comes after this flag on the command line, until a 'dump' is encountered (see above). For example:

```
%>chuck +d foo.ck -d bar.ck +d doo.ck
```

will dump foo.ck, then doo.ck - but not bar.ck.

These are useful for debug ChuckK itself, and for other entertainment purposes.

--srate(N)

set the internal sample rate to (N) Hz. by default, ChuckK runs at 44100Hz on OS X and Windows, and 48000Hz on linux/ALSA. even if the VM is running in --silent mode, the sample rate is still used by some unit generators to compute audio, this is important for computing samples and writing to file. Not all sample rates are supported by all devices!

--bufsize(N)

set the internal audio buffer size to (N) sample frames. larger buffer size often reduce audio artifacts due to system/program timing. smaller buffers reduce audio latency. The default is 512. If (N) is not a power of 2, the next power of 2 larger than (N) is used. For example:

```
%>chuck --bufsize950
```

sets the buffer size to 1024.

--dac(N)

opens audio output device #(N) for real-time audio. by default, (N) is 0.

--adc(N)

opens audio input device #(N) for real-time audio input. by default, (N) is 0.

--about / --help

prints the usage message, with the ChuckK URL

CHAPTER 15

On-the-fly Programming Commands

These are used for on-the-fly programming (see <http://on-the-fly.cs.princeton.edu>). By default, this requires that a ChuckK virtual machine be already running on the localhost. It communicates via sockets to add/remove/replace shreds in the VM, and to query VM state. The simplest way to set up a ChuckK virtual machine to accept these commands is by starting an empty VM with `--loop`:

```
%>chuck --loop
```

this will start a VM, looping (and advancing time), waiting for incoming commands. Successive invocations of ‘chuck’ with the appropriate commands will communicate with this listener VM.

(for remote operations over TCP, see below)

--add / +

adds new shreds from source files to the listener VM. this process then exits. for example:

```
%>chuck + foo.ck bar.ck
```

integrates `foo.ck` and `bar.ck` into the listener VM. the shreds are internally responsible for finding about the timing and other shreds via the timing mechanism and vm interface.

--remove / -

removes existing shreds from the VM by ID. how to find out about the id? (see `--status` below) for example:

```
%>chuck - 2 3 8
```

removes shred 2, 3, 8.

--replace / =

replace existing shred with a new shred. for example:

```
%>chuck = 2 foo.ck
```

replaces shred 2 with foo.ck

--status / ^

queries the status of the VM - output on the listener VM. for example:

```
%>chuck ^
```

this prints the internal shred start at the listener VM, something like:

```
[chuck](VM): status (now == 0h:2m:34s) ...  
    [shred id]: 1 [source]: foo.ck [sporked]: 21.43s ago  
    [shred id]: 2 [source]: bar.ck [sporked]: 28.37s ago
```

--time

prints out the value of now on the listener VM. for example:

```
%>chuck --time
```

something like:

```
[chuck](VM): the value of now:          now = 403457 (samp)  
          = 9.148685 (second)  
          = 0.152478 (minute)  
          = 0.002541 (hour)  
          = 0.000106 (day)  
          = 0.000015 (week)
```

--kill

semi-gracefully kills the listener VM - removes all shreds first.

--remote /

specifies where to send the on-the-fly command. must appear in the command line before any on-the-fly commands. for example:

```
%>chuck @192.168.1.1 + foo.ck bar.ck  
      (or)  
%>chuck @foo.bar.com -p8888 + foo.ck bar.ck
```

sends foo.ck and bar.ck to VM at 192.168.1.1 or foo.bar.com:8888

CHAPTER 16

Standard Libraries API

ChuckK Standard Libraries API these libraries are provide by default with ChuckK - new ones can also be imported with ChuckK dynamic linking (soon to be documented...). The existing libraries are organized by namespaces in ChuckK. They are as follows.

namespace: `std`

std is a standard library in ChuckK, which includes utility functions, random number generation, unit conversions and absolute value.

[example]

```
/* a simple example... */
// infinite time-loop
while( true )
{
    // generate random float (and print)
    <<< std.rand2f( 100.0, 1000.0 ) >>>;
    // wait a bit
    50::ms => now;
}
```

[function] `int abs (int value);`
returns absolute value of integer

[function] float fabs (float value);
returns absolute value of floating point number

[function] int rand ();
generates random integer

[function] int rand2 (int min, int max);
generates random integer in the range [min, max]

[function] float randf ();
generates random floating point number in the range [-1, 1]

[function] float rand2f (float min, float max);
generates random floating point number in the range [min, max]

[function] float sgn (float value);
computes the sign of the input as -1.0 (negative), 0 (zero), or 1.0 (positive)

[function] int system (string cmd);
pass a command to be executed in the shell

[function] int atoi (string value);
converts ascii (string) to integer (int)

[function] float atof (string value);
converts ascii (string) to floating point value (float)

[function] string getenv (string value);
returns the value of an environment variable, such as of "PATH"

[function] int setenv (string key, string value);
sets environment variable named 'key' to 'value'

[function] float mtof (float value);
converts a MIDI note number to frequency (Hz)
note the input value is of type 'float' (supports fractional note number)

```
[ function ] float ftom ( float value );  
converts frequency (Hz) to MIDI note number space
```

```
[ function ] float powtodb ( float value );  
converts signal power ratio to decibels (dB)
```

```
[ function ] float rmstodb ( float value );  
converts linear amplitude to decibels (dB)
```

```
[ function ] float dbtopow ( float value );  
converts decibels (dB) to signal power ratio
```

```
[ function ] float dbtorms ( float value );  
converts decibels (dB) to linear amplitude
```

namespace: machine

machine is ChuckK runtime interface to the virtual machine. this interface can be used to manage shreds. They are similar to the On-the-fly Programming Commands, except these are invoked from within a ChuckK program, and are subject to the timing mechanism.

```
[ function ] int add ( string path );  
compile and spork a new shred from file at 'path' into the VM now  
returns the shred ID  
(see example/machine.ck)
```

```
[ function ] int spork ( string path );  
same as add/+
```

```
[ function ] int remove ( int id );  
remove shred from VM by shred ID (returned by add/spork)
```

```
[ function ] int replace ( int id, string path );  
replace shred with new shred from file  
returns shred ID , or 0 on error
```

```
[ function ] int status ( );  
display current status of VM  
same functionality as status/^  
(see example/status.ck)
```

```
[ function ] void crash ( );
```

iterally causes the VM to crash. the very last resort; use with care. Thanks.

namespace: `math`

math contains the standard math functions. all trigonometric functions expects angles to be in radians.

[example]

```
// print sine of pi/2
<<<< math.sin( pi / 2.0 ) >>>;
```

```
[ function ] float sin ( float x );
```

computes the sine of x

```
[ function ] float cos ( float x );
```

computes the cosine of x

```
[ function ] float tan ( float x );
```

computes the tangent of x

```
[ function ] float asin ( float x );
```

computes the arc sine of x

```
[ function ] float acos ( float x );
```

computes the arc cosine of x

```
[ function ] float atan ( float x );
```

computes the arc tangent of x

```
[ function ] float atan2 ( float x );
```

computes the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value

```
[ function ] float sinh ( float x );
```

computes the hyperbolic sine of x

```
[ function ] float cosh ( float x );
```

computes the hyperbolic cosine of x

```
[ function ] float tanh ( float x );
```

computes the hyperbolic tangent of x

```
[ function ] float hypot ( float x, float y );
```

computes the euclidean distance of the orthogonal vectors (x,0) and (0,y)

```
[ function ] float pow ( float x, float y );
```

computes x taken to the y-th power

```
[ function ] float sqrt ( float x );
```

computes the nonnegative square root of x (x must be ≥ 0)

```
[ function ] float exp ( float x );
```

computes e^x , the base-e exponential of x

```
[ function ] float log ( float x );
```

computes the natural logarithm of x

```
[ function ] float log2 ( float x );
```

computes the logarithm of x to base 2

```
[ function ] float log10 ( float x );
```

computes the logarithm of x to base 10

```
[ function ] float floor ( float x );
```

round to largest integral value (returned as float) not greater than x

```
[ function ] float ceil ( float x );
```

round to smallest integral value (returned as float) not less than x

```
[ function ] float round ( float x );
```

round to nearest integral value (returned as float)

```
[ function ] float trunc ( float x );
```

round to largest integral value (returned as float) no greater in magnitude than x

```
[ function ] float fmod ( float x, float y );
```

computes the floating point remainder of x / y

[function] float remainder (float x, float y);

computes the value r such that $r = x - n * y$, where n is the integer nearest the exact value of x / y . If there are two integers closest to x / y , n shall be the even one. If r is zero, it is given the same sign as x

[function] float min (float x, float y);

choose lesser of two values

[function] float max (float x, float y);

choose greater of two values

[function] int nextpow2 (int n);

computes the integral (returned as int) smallest power of 2 greater than the value of x

[function] float pi ();

(currently disabled - use pi)

[function] float twopi ();

(currently disabled)

[function] float e ();

(currently disabled - use `math.exp(1)`)

CHAPTER 17

Unit Generators

Unit generators (ugens) can be connected using the ChuckK operator (`=>`)

[example]

```
adc => dac;
```

the above connects the ugen ‘adc’ (a/d convertor, or audio input) to ‘dac’ (d/a convertor, or audio output).

Ugens can also be unlinked (using `=<`) and relinked (see `examples/unchuck.ck`).

A unit generator may have 0 or more control parameters. A Ugen’s parameters can be set also using the ChuckK operator (`=>` , or `->`)

[example]

```
//connect sine oscillator to dac
sinosc osc => dac;
// set the osc’s frequency to 60.0 hz
60.0 => osc.freq;
```

(see `examples/osc.ck`)

All ugen's have at least the following three parameters:

[ctrl param]

- **.gain** - (float, READ/WRITE) - *set gain*
- **.op** - (int, READ/WRITE) - *set operation type*
 - -1 passthrough
 - 0 no processing
 - 1 normal processing
- **.last** - (float, READ/WRITE) - *the last sample computed by the unit generator*

audio output

dac

- digital/analog converter
- abstraction for underlying audio output device

adc

- analog/digital converter
- abstraction for underlying audio input device

blackhole

- sample rate sample sucker
- (like dac, ticks ugens, but no more)
- see examples/pwm.ck

gain

- gain control
- (NOTE - all unit generators can themselves change their gain)
- (this is a way to add N outputs together and scale them)
- used in examples/i-robot.ck

[ctrl param]

- **.gain** - (float, READ/WRITE) - *set gain* (all ugen's have this)
- **.op** - (int, READ/WRITE) - *set operation type*
 - -1 passthrough
 - 0 no processing
 - 1 adds inputs (default)
 - 2 subtracts inputs
 - 3 multiplies inputs
 - 4 divides inputs

[example]

```
noise n => gain g => dac;
sinosc s => g;
.3 => g.gain;
while( true ) { 100::ms => now; }
```

wave forms

noise

- white noise generator
- see examples/noise.ck examples/powerup.ck

impulse

- pulse generator - can set the value of the current sample
- default for each sample is 0 if not set

[ctrl param]

- **.value** - (float, READ/WRITE) - *set the current value* (currently broken)
- **.next** - (float, READ/WRITE) - *set value of next sample*

[example]

```
impulse i => dac;
while( true ) {
    1.0 => i.next;
    100::ms => now;
}
```

step

- step generator - like impulse, but once a value is set, it is held for all following samples, until value is set again
- see examples/step.ck

[ctrl param]

- **.value** - (float, READ/WRITE) - *set the current value*
- **.next** - (float, READ/WRITE) - *set the step value*

[example]

```
step s => dac;
-1.0 => float amp;
// square wave using step
while( true ) {
  -amp => amp => s.next;
      800::samp => now;
}
```

filters

halfrect

- half wave rectifier
- for half-wave rectification

fullrect

- full wave rectifier

zerox

- zero crossing detector
- emits a single pulse at the the zero crossing in the direction of the zero crossing
- (see examples/zerox.ck)

delays

delayp

- variable write delay

[ctrl param]

- **.delay** - (dur, READ/WRITE) - *delay before subsequent values emerge*
- **.window** - (dur, READ/WRITE) - *time for 'write head' to move*
- **.max** - (dur, READ/WRITE) - *max delay possible. trashes buffer, so do it first!*

sound files

sndbuf

- sound buffer (now interpolating)
- reads from a variety of file formats
- see examples/sndbuf.ck

[ctrl param]

- **.read** - (string, WRITE only) - *loads file for reading*
- **.write** - (string, WRITE only) - *loads a file for writing (or not)*
- **.pos** - (int, READ/WRITE) - *set position ($0 < p < .samples$)*
- **.loop** - (int, READ/WRITE) - *toggle looping*
- **.interp** - (int, READ/WRITE) - *set interpolation (0=drop, 1=linear, 2=sinc)*
- **.rate** - (float, READ/WRITE) - *playback rate (relative to the file's natural speed)*
- **.play** - (float, READ/WRITE) - *play (same as rate)*
- **.freq** - (float, READ/WRITE) - *playback rate (file loops/second)*
- **.phase** - (float, READ/WRITE) - *set phase position (0-1)*
- **.channel** - (int, READ/WRITE) - *select channel ($0 < x < .channels$) //*
- **.phase_offset** - (float, READ/WRITE) - *set a phase offset*
- **.samples** - (int, READ only) - *fetch number of samples*
- **.length** - (float, READ only) - *fetch length in seconds*
- **.channels** - (int, READ only) - *fetch number of channels*

oscillators

phasor

- simple ramp generator (0 to 1)
- this can be fed into other oscillators (with sync mode of 2)
- as a phase control. see examples/sixty.ck for an example

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to global (1) or self (0)*
- **.width** (float, READ/WRITE) *set duration of the ramp in each cycle. (default 1.0)*

sinosc

- sine oscillator
- (see examples/osc.ck)

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to self (0), global(1), input(2)*

pulseosc

- pulse oscillators
- a pulse wave oscillator with variable width.

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to self (0), global(1), input(2)*
- **.width** (float, READ/WRITE) *length of duty cycle (0 - 1)*

sqrosc

- square wave oscillator (pulse with fixed width of 0.5)

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to self (0), global(1), input(2)*

triosc

- triangle wave oscillator

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to self (0), global(1), input(2)*
- **.width** (float, READ/WRITE) *control midpoint of triangle (0 - 1)*

sawosc

- sawtooth wave oscillator (triangle, width forced to 0.0 or 1.0)

[ctrl param]

- **.freq** (float, READ/WRITE) *oscillator frequency (Hz)*
- **.sfreq** (float, READ/WRITE) *oscillator frequency (Hz), phase-matched //*
- **.phase_offset** (float, READ/WRITE) *phase offset*
- **.phase** (float, READ/WRITE) *current phase*
- **.sync** (int, READ/WRITE) *sync to self (0), global(1), input(2)*
- **.width** (float, READ/WRITE) *increasing (w > 0.5) or decreasing (w < 0.5)*

network

netout

- UDP-based network audio transmitter

[ctrl param]

- **.addr** (string, READ/WRITE) *target address*
- **.port** (int, READ/WRITE) *target port*
- **.size** (int, READ/WRITE) *packet size*
- **.name** (string, READ/WRITE) *name*

netin

- UDP-based network audio receiver

[ctrl param]

- **.port** (int, READ/WRITE) *set port to receive*
- **.name** (string, READ/WRITE) *name*

STK - Instruments

BandedWG

- Banded waveguide modeling class

This class uses banded waveguide techniques to model a variety of sounds, including bowed bars, glasses, and bowls. For more information, see Essl, G. and Cook, P. "Banded Waveguides: Towards Physical Modelling of Bar Percussion Instruments", Proceedings of the 1999 International Computer Music Conference.

Control Change Numbers:

- Bow Pressure = 2
- Bow Motion = 4
- Strike Position = 8 (not implemented)
- Vibrato Frequency = 11
- Gain = 1
- Bow Velocity = 128
- Set Striking = 64
- Instrument Presets = 16
- Uniform Bar = 0
- Tuned Bar = 1
- Glass Harmonica = 2
- Tibetan Bowl = 3

by Georg Essl, 1999 - 2002.

Modified for Stk 4.0 by Gary Scavone.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.pluck** (float, READ/WRITE) *pluck waveguide*
- **.startBowing** (float, READ/WRITE) *pluck waveguide*
- **.stopBowing** (float, READ/WRITE) *pluck waveguide*
- **.freq** (float, READ/WRITE) *strike Position*
- **.bowRate** (float, READ/WRITE) *strike Position*
- **.bowPressure** (float, READ/WRITE) *strike Position*
- **.preset** (int, READ/WRITE) *strike Position*
- **.strikePosition** (float, READ/WRITE) *strike Position*

BlowBotl

- STK blown bottle instrument class

This class implements a helmholtz resonator (biquad filter) with a polynomial jet excitation (a la Cook).

Control Change Numbers:

- Noise Gain = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Volume = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.startBlowing** (float, READ/WRITE) *noteOn*
- **.stopBlowing** (float, READ/WRITE) *noteOff*
- **.freq** (float, READ/WRITE) *frequency*
- **.rate** (float, READ/WRITE) *frequency*

BlowHole

- STK clarinet physical model with one register hole and one tonehole.

register hole and one tonehole.

This class is based on the clarinet model, with the addition of a two-port register hole and a three-port dynamic tonehole implementation, as discussed by Scavone and Cook (1998).

In this implementation, the distances between the reed/register hole and tonehole/bell are

fixed. As a result, both the tonehole and register hole will have variable influence on the playing frequency, which is dependent on the length of the air column. In addition, the highest playing frequency is limited by these fixed lengths.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Reed Stiffness = 2
- Noise Gain = 4
- Tonehole State = 11
- Register State = 1
- Breath Pressure = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.startBlowing** (float, READ/WRITE) *noteOn*
- **.stopBlowing** (float, READ/WRITE) *noteOff*
- **.freq** (float, READ/WRITE) *frequency*
- **.vent** (float, READ/WRITE) *vent frequency*
- **.tonehole** (float, READ/WRITE) *tonehole size*
- **.reed** (float, READ/WRITE) *reed stiffness*
- **.rate** (float, READ/WRITE) *rate of change*

Bowed

- STK bowed string instrument class.

This class implements a bowed string model, a la Smith (1986), after McIntyre, Schumacher, Woodhouse (1983).

This is a digital waveguide model, making its

use possibly subject to patents held by
Stanford University, Yamaha, and others.

Control Change Numbers:

- Bow Pressure = 2
- Bow Position = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Volume = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.startBowing** (float, READ/WRITE) *begin bowing instrument*
- **.stopBowing** (float, READ/WRITE) *stop bowing instrument*
- **.freq** (float, READ/WRITE)
- **.rate** (float, READ/WRITE)
- **.vibrato** (float, READ/WRITE)

Brass

- STK simple brass instrument class.

This class implements a simple brass instrument
waveguide model, a la Cook (TBone, HosePlayer).

This is a digital waveguide model, making its
use possibly subject to patents held by
Stanford University, Yamaha, and others.

Control Change Numbers:

- Lip Tension = 2
- Slide Length = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Volume = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.clear** (float, READ/WRITE) *clear instrument*
- **.startBlowing** (float, READ/WRITE) *noteOn*
- **.stopBlowing** (float, READ/WRITE) *noteOff*
- **.freq** (float, READ/WRITE) *frequency*
- **.rate** (float, READ/WRITE) *rate of change*
- **.lip** (float, READ/WRITE) *lip stiffness*

Clarinet

- STK clarinet physical model class.

This class implements a simple clarinet physical model, as discussed by Smith (1986), McIntyre, Schumacher, Woodhouse (1983), and others.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Reed Stiffness = 2
- Noise Gain = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Breath Pressure = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.clear** (float, READ/WRITE) *clear instrument*
- **.startBlowing** (float, READ/WRITE) *noteOn*
- **.stopBlowing** (float, READ/WRITE) *noteOff*

- **.freq** (float, READ/WRITE) *frequency*
- **.rate** (float, READ/WRITE) *rate of change*

Flute

- STK flute physical model class.

This class implements a simple flute physical model, as discussed by Karjalainen, Smith, Waryznyk, etc. The jet model uses a polynomial, a la Cook.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Jet Delay = 2
- Noise Gain = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Breath Pressure = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.clear** (float, READ/WRITE) *clear instrument*
- **.startBlowing** (float, READ/WRITE) *noteOn*
- **.stopBlowing** (float, READ/WRITE) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*
- **.rate** (float, READ/WRITE) *rate of change*
- **.jetReflection** (float, READ/WRITE) *rate of change*
- **.jetDelay** (float, READ/WRITE) *rate of change*
- **.endReflection** (float, READ/WRITE) *rate of change*

Mandolin

- STK mandolin instrument model class.
- see examples/mand-o-matic.ck

This class inherits from PluckTwo and uses "commuted synthesis" techniques to model a mandolin instrument.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others. Commuted Synthesis, in particular, is covered by patents, granted, pending, and/or applied-for. All are assigned to the Board of Trustees, Stanford University. For information, contact the Office of Technology Licensing, Stanford University.

Control Change Numbers:

- Body Size = 2
- Pluck Position = 4
- String Sustain = 11
- String Detuning = 1
- Microphone Position = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.pluck** (float, WRITE only) *pluck string with given amplitude*
- **.freq** (float, READ/WRITE) *frequency*
- **.pluckPos** (float, READ/WRITE) *set pluck position (0-1) along string*
- **.bodySize** (float, READ/WRITE) *modify instrument size*
- **.stringDamping** (float, READ/WRITE) *control string damping*
- **.stringDetune** (float, READ/WRITE) *control detuning of string pair*
- **.aftertouch** (float, READ/WRITE) *aftertouch*

ModalBar

- STK resonant bar instrument class.
- see examples/modalbot.ck

This class implements a number of different struck bar instruments. It inherits from the Modal class.

Control Change Numbers:

- Stick Hardness = 2
- Stick Position = 4
- Vibrato Gain = 11
- Vibrato Frequency = 7
- Direct Stick Mix = 1
- Volume = 128
- Modal Presets = 16
 - Marimba = 0
 - Vibraphone = 1
 - Agogo = 2
 - Wood1 = 3
 - Reso = 4
 - Wood2 = 5
 - Beats = 6
 - Two Fixed = 7
 - Clump = 8

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.strike** (float, READ/WRITE) *strike bar*
- **.damp** (float, READ/WRITE) *damp bar*
- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.clear** (float, READ/WRITE) *reset*
- **.preset** (float, READ/WRITE) *choose preset*
- **.freq** (float, READ/WRITE) *frequency*
- **.strikePosition** (float, READ/WRITE) *set frequency*
- **.stickHardness** (float, READ/WRITE) *set frequency*
- **.masterGain** (float, READ/WRITE) *set frequency*
- **.directGain** (float, READ/WRITE) *set frequency*
- **.mode** (int, READ/WRITE) *choose mode*
- **.modeRatio** (float, READ/WRITE) *mode edit*
- **.modeRadius** (float, READ/WRITE) *mode edit*
- **.modeGain** (float, READ/WRITE) *mode edit*

Moog

- STK moog-like swept filter sampling synthesis class
- see examples/moogie.ck

This instrument uses one attack wave, one looped wave, and an ADSR envelope (inherited from the Sampler class) and adds two sweepable formant (FormSweep) filters.

Control Change Numbers:

- Filter Q = 2
- Filter Sweep Rate = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Gain = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*
- **.modSpeed** (float, READ/WRITE) *modulation speed*
- **.modDepth** (float, READ/WRITE) *modulation depth*
- **.filterQ** (float, READ/WRITE) *filter Q value*
- **.filterSweepRate** (float, READ/WRITE) *filter sweep rate*
- **.aftertouch** (float, WRITE only) *aftertouch*

Saxofony

- STK faux conical bore reed instrument class.

This class implements a "hybrid" digital waveguide instrument that can generate a variety of wind-like sounds. It has also been referred to as the "blowed string" model. The waveguide section is essentially that of a string, with one rigid and one lossy termination. The non-linear function is a

reed table. The string can be "blown" at any point between the terminations, though just as with strings, it is impossible to excite the system at either end. If the excitation is placed at the string mid-point, the sound is that of a clarinet. At points closer to the "bridge", the sound is closer to that of a saxophone. See Scavone (2002) for more details.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Reed Stiffness = 2
- Reed Aperture = 26
- Noise Gain = 4
- Blow Position = 11
- Vibrato Frequency = 29
- Vibrato Gain = 1
- Breath Pressure = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.noteOff** (float, WRITE only) *noteOff*
- **.clear** (float, WRITE only) *clear instrument*
- **.startBlowing** (float, WRITE only) *noteOn*
- **.stopBlowing** (float, WRITE only) *noteOff*
- **.freq** (float, READ/WRITE) *frequency*
- **.rate** (float, READ/WRITE) *rate of change*
- **.blowPosition** (float, READ/WRITE) *lip stiffness*

Shakers

- PhISEM and PhOLIES class.
- see examples/shake-o-matic.ck

PhISEM (Physically Informed Stochastic Event

Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects. This class is a meta-model that can simulate a Maraca, Sekere, Cabasa, Bamboo Wind Chimes, Water Drops, Tambourine, Sleighbells, and a Guiro.

PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds) is a similar approach for the synthesis of environmental sounds. This class implements simulations of breaking sticks, crunchy snow (or not), a wrench, sandpaper, and more.

Control Change Numbers:

- Shake Energy = 2
- System Decay = 4
- Number Of Objects = 11
- Resonance Frequency = 1
- Shake Energy = 128
- Instrument Selection = 1071
 - Maraca = 0
 - Cabasa = 1
 - Sekere = 2
 - Guiro = 3
 - Water Drops = 4
 - Bamboo Chimes = 5
 - Tambourine = 6
 - Sleigh Bells = 7
 - Sticks = 8
 - Crunch = 9
 - Wrench = 10
 - Sand Paper = 11
 - Coke Can = 12
 - Next Mug = 13
 - Penny + Mug = 14
 - Nickle + Mug = 15
 - Dime + Mug = 16
 - Quarter + Mug = 17
 - Franc + Mug = 18
 - Peso + Mug = 19
 - Big Rocks = 20

- Little Rocks = 21
- Tuned Bamboo Chimes = 22

by Perry R. Cook, 1996 - 1999.

[ctrl param]

- **.freq** (float, READ/WRITE) *frequency*
- **.noteOn** (float, READ/WRITE) *start shake with a given amplitude*
- **.noteOff** (float, READ/WRITE) *stop shake*
- **.which** (int, READ/WRITE) *select instrument*

Sitar

- STK sitar string model class.

This class implements a sitar plucked string physical model based on the Karplus-Strong algorithm.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others. There exist at least two patents, assigned to Stanford, bearing the names of Karplus and/or Strong.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.pluck** (float, READ/WRITE) *noteOn*
- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.clear** (float, READ/WRITE) *clear instrument*
- **.freq** (float, READ/WRITE) *frequency*

StifKarp

- STK plucked stiff string instrument.
- see examples/stifkarp.ck

This class implements a simple plucked string algorithm (Karplus Strong) with enhancements (Jaffe-Smith, Smith, and others), including string stiffness and pluck position controls. The stiffness is modeled with allpass filters.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Pickup Position = 4
- String Sustain = 11
- String Stretch = 1

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.pluck** (float, WRITE only) *noteOn*
- **.noteOn** (float, WRITE only) *noteOn*
- **.noteOff** (float, WRITE only) *noteOff*
- **.clear** (float, WRITE only) *clear instrument*
- **.freq** (float, READ/WRITE) *frequency*
- **.pickupPosition** (float, READ/WRITE)
- **.stretch** (float, READ/WRITE)
- **.baseLoopGain** (float, READ/WRITE)

VoicForm

- Four formant synthesis instrument.
- see examples/voic-o-form.ck

This instrument contains an excitation singing wavetable (looping wave with random and periodic vibrato, smoothing on frequency, etc.), excitation noise, and four sweepable

complex resonances.

Measured formant data is included, and enough data is there to support either parallel or cascade synthesis. In the floating point case cascade synthesis is the most natural so that's what you'll find here.

Control Change Numbers:

- Voiced/Unvoiced Mix = 2
- Vowel/Phoneme Selection = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Loudness (Spectral Tilt) = 128

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.freq** (float, READ/WRITE) *frequency*
- **.phoneme** (string, READ/WRITE) *select phoneme (above)*
- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.speak** (float, READ/WRITE) *start singing*
- **.quiet** (float, READ/WRITE) *stop singing*
- **.voiced** (float, READ/WRITE) *set mix for voiced component*
- **.unVoiced** (float, READ/WRITE) *set mix for unvoiced componenet*
- **.pitchSweepRate** (float, READ/WRITE) *pitch sweep*
- **.voiceMix** (float, READ/WRITE) *voiced/unvoiced mix*
- **.setPhoneme** (int, READ/WRITE) *set phoneme by number*
- **.vibratoFreq** (float, READ/WRITE) *vibrato*
- **.vibratoGain** (float, READ/WRITE) *vibrato depth*
- **.loudness** (float, READ/WRITE) *'loudness' of voice*

STK - FM Synths

FM

- STK abstract FM synthesis base class

This class controls an arbitrary number of

waves and envelopes, determined via a constructor argument.

Control Change Numbers:

- Control One = 2
- Control Two = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.freq** (float, READ/WRITE) *frequency*
- **.noteOn** (float, READ/WRITE) *noteOn*
- **.noteOff** (float, READ/WRITE) *noteOff*
- **.modDepth** (float, READ/WRITE) *modulation depth*
- **.modSpeed** (float, READ/WRITE) *modulation speed*
- **.afterTouch** (float, READ/WRITE) *aftertouch*
- **.control1** (float, READ/WRITE) *FM control 1*
- **.control2** (float, READ/WRITE) *FM control 2*

BeeThree

- STK Hammond-oid organ FM synthesis instrument.

This class implements a simple 4 operator topology, also referred to as algorithm 8 of the TX81Z.

\code

Algorithm 8 is :

1 --.

```

                2 -\|
                  +-> Out
                3 -/|
                4 --
\endcode

```

Control Change Numbers:

- Operator 4 (feedback) Gain = 2
- Operator 3 Gain = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*

FMVoices

- STK singing FM synthesis instrument.

This class implements 3 carriers and a common modulator, also referred to as algorithm 6 of the TX81Z.

```

\code
Algorithm 6 is :
                /->1 -\
            4-|--->2 - +-> Out
                \->3 -/
\endcode

```

Control Change Numbers:

- Vowel = 2
- Spectral Tilt = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*
- **.vowel** (float, WRITE only) *select vowel*
- **.spectralTilt** (float, WRITE only) *spectral tilt*
- **.lfoSpeed** (float, WRITE only) *speed of LFO*
- **.lfoDepth** (float, WRITE only) *depth of LFO*
- **.adsrTarget** (float, WRITE only) *adsr targets*

HeavyMetl

- STK heavy metal FM synthesis instrument.

This class implements 3 cascade operators with feedback modulation, also referred to as algorithm 3 of the TX81Z.

Algorithm 3 is : 4--\
 3-->2-- + -->1-->Out

Control Change Numbers:

- Total Modulator Index = 2

- Modulator Crossfade = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*

PercFlut

- STK percussive flute FM synthesis instrument.

This class implements algorithm 4 of the TX81Z.

```
\code
Algorithm 4 is :   4->3--\
                  2-- + -->1-->Out
\endcode
```

Control Change Numbers:

- Total Modulator Index = 2
- Modulator Crossfade = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making

money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*

Rhodey

- STK Fender Rhodes-like electric piano FM
- see examples/rhodey.ck

synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

```
\code
Algorithm 5 is : 4->3--\
                  + --> Out
                  2->1--/
\endcode
```

Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*

TubeBell

- STK tubular bell (orchestral chime) FM
- extends FM

synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

\code

```
Algorithm 5 is :  4->3--\
                  + --> Out
                  2->1--/
```

\endcode

Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*

- **.freq** (float, READ/WRITE) *frequency*

Wurley

- STK Wurlitzer electric piano FM
- see examples/wurley.ck extends

synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

```
\code
Algorithm 5 is : 4->3--\
                  + --> Out
                  2->1--/
\endcode
```

Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

extends FM

[ctrl param]

- **.noteOn** (float, WRITE only) *noteOn*
- **.freq** (float, READ/WRITE) *frequency*

STK - Delay

Delay

- STK non-interpolating delay line class
- see `examples/net_relay.ck`

This protected Filter subclass implements a non-interpolating digital delay-line. A fixed maximum length of 4095 and a delay of zero is set using the default constructor. Alternatively, the delay and maximum length can be set during instantiation with an overloaded constructor.

A non-interpolating delay line is typically used in fixed delay-length applications, such as for reverberation.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.delay** (float, READ/WRITE) *length of delay*
- **.max** (float, READ/WRITE) *max delay (buffer size)*

DelayA

- STK allpass interpolating delay line class.

This Delay subclass implements a fractional-length digital delay-line using a first-order allpass filter. A fixed maximum length of 4095 and a delay of 0.5 is set using the default constructor. Alternatively, the delay and maximum length can be set during instantiation with an overloaded constructor.

An allpass filter has unity magnitude gain but variable phase delay properties, making it useful in achieving fractional delays without affecting a signal's frequency magnitude response. In order to achieve a maximally flat phase delay response, the minimum delay possible in this implementation is limited to a value of 0.5.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.delay** (float, READ/WRITE) *length of delay*
- **.max** (float, READ/WRITE) *max delay (buffer size)*

DelayL

- STK linear interpolating delay line class.
- see examples/i-robot.ck

This Delay subclass implements a fractional-length digital delay-line using first-order linear interpolation. A fixed maximum length of 4095 and a delay of zero is set using the default constructor. Alternatively, the delay and maximum length can be set during instantiation with an overloaded constructor.

Linear interpolation is an efficient technique for achieving fractional delay lengths, though it does introduce high-frequency signal attenuation to varying degrees depending on the fractional delay setting. The use of higher order Lagrange interpolators can typically improve (minimize) this attenuation characteristic.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.delay** (float, READ/WRITE) *length of delay*

- **.max** (float, READ/WRITE) *max delay (buffer size)*

Echo

- STK echo effect class.

This class implements a echo effect.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.delay** (float, READ/WRITE) *length of echo*
- **.max** (float, READ/WRITE) *max delay*
- **.mix** (float, READ/WRITE) *mix level (wet/dry)*

STK - Envelopes

Envelope

- STK envelope base class.
- see examples/sixty.ck

This class implements a simple envelope generator which is capable of ramping to a target value by a specified \e rate. It also responds to simple \e keyOn and \e keyOff messages, ramping to 1.0 on keyOn and to 0.0 on keyOff.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.keyOn** (int, WRITE only) *ramp to 1.0*
- **.keyOff** (int, WRITE only) *ramp to 0.0*
- **.target** (float, READ/WRITE) *ramp to arbitrary value*
- **.time** (float, READ/WRITE) *time to reach target*

- **.rate** (float, READ/WRITE) *rate of change*
- **.value** (float, READ/WRITE) *set immediate value*

ADSR ---

- STK ADSR envelope class.
- see examples/adrs.ck

This Envelope subclass implements a traditional ADSR (Attack, Decay, Sustain, Release) envelope. It responds to simple keyOn and keyOff messages, keeping track of its state. The \e state = ADSR::DONE after the envelope value reaches 0.0 in the ADSR::RELEASE state.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.keyOn** (int, WRITE only) *start the attack for non-zero values*
- **.keyOff** (int, WRITE only) *start the release for non-zero values*
- **.attackTime** (float, WRITE only) *attack time*
- **.attackRate** (float, READ/WRITE) *attack rate*
- **.decayTime** (float, READ/WRITE) *decay*
- **.decayRate** (float, READ/WRITE) *decay rate*
- **.sustainLevel** (float, READ/WRITE) *sustain level*
- **.releaseTime** (float, READ/WRITE) *release time*
- **.releaseRate** (float, READ/WRITE) *release rate*
- **.state** (int, READ only) *attack=0, decay=1, sustain=2, release=3,done=4*

STK - Filters ---

BiQuad ---

- STK biquad (two-pole, two-zero) filter class.

This protected Filter subclass implements a two-pole, two-zero digital filter. A method is provided for creating a resonance in the frequency response while maintaining a constant filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.b2** (float, READ/WRITE) *filter coefficient*
- **.b1** (float, READ/WRITE) *filter coefficient*
- **.b0** (float, READ/WRITE) *filter coefficient*
- **.a2** (float, READ/WRITE) *filter coefficient*
- **.a1** (float, READ/WRITE) *filter coefficient*
- **.a0** (float, READ only) *filter coefficient*
- **.pfreq** (float, WRITE only) *set resonance frequency (poles)*
- **.prad** (float, WRITE only) *pole radius (less than 1 to be stable)*
- **.zfreq** (float, WRITE only) *notch frequency*
- **.zrad** (float, WRITE only) *zero radius*
- **.norm** (float, WRITE only) *normalization*
- **.eqzs** (float, WRITE only) *equal gain zeroes*

Filter

- STK filter class.

This class implements a generic structure which can be used to create a wide range of filters. It can function independently or be subclassed to provide more specific controls based on a particular filter type.

In particular, this class implements the standard difference equation:

$$a[0]*y[n] = b[0]*x[n] + \dots + b[nb]*x[n-nb] - a[1]*y[n-1] - \dots - a[na]*y[n-na]$$

If $a[0]$ is not equal to 1, the filter coefficients

are normalized by `a[0]`.

The `gain` parameter is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0. This structure results in one extra multiply per computed sample, but allows easy control of the overall filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- `.coefs` (string, WRITE only)

OnePole

- STK one-pole filter class.

This protected Filter subclass implements a one-pole digital filter. A method is provided for setting the pole position along the real axis of the z-plane while maintaining a constant peak filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- `.a1` (float, READ/WRITE) *filter coefficient*
- `.b0` (float, READ/WRITE) *filter coefficient*
- `.pole` (float, READ/WRITE) *set pole position along real axis of z-plane*

TwoPole

- STK two-pole filter class.
- see `examples/powerup.ck`

This protected Filter subclass implements

a two-pole digital filter. A method is provided for creating a resonance in the frequency response while maintaining a nearly constant filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.a1** (float, READ/WRITE) *filter coefficient*
- **.a2** (float, READ/WRITE) *filter coefficient*
- **.b0** (float, READ/WRITE) *filter coefficient*
- **.freq** (float, READ/WRITE) *filter resonance frequency*
- **.radius** (float, READ/WRITE) *filter resonance radius*
- **.norm** (float, READ/WRITE) *toggle filter normalization*

OneZero

- STK one-zero filter class.

This protected Filter subclass implements a one-zero digital filter. A method is provided for setting the zero position along the real axis of the z-plane while maintaining a constant filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.zero** (float, READ/WRITE) *set zero position*
- **.b0** (float, READ/WRITE) *filter coefficient*
- **.b1** (float, READ/WRITE) *filter coefficient*

TwoZero

- STK two-zero filter class.

This protected Filter subclass implements a two-zero digital filter. A method is provided for creating a "notch" in the frequency response while maintaining a constant filter gain.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.b0** (float, READ/WRITE) *filter coefficient*
- **.b1** (float, READ/WRITE) *filter coefficient*
- **.b2** (float, READ/WRITE) *filter coefficient*
- **.freq** (float, READ/WRITE) *filter notch frequency*
- **.radius** (float, READ/WRITE) *filter notch radius*

PoleZero

- STK one-pole, one-zero filter class.

This protected Filter subclass implements a one-pole, one-zero digital filter. A method is provided for creating an allpass filter with a given coefficient. Another method is provided to create a DC blocking filter.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.a1** (float, READ/WRITE) *filter coefficient*
- **.b0** (float, READ/WRITE) *filter coefficient*
- **.b1** (float, READ/WRITE) *filter coefficient*
- **.blockZero** (float, READ/WRITE) *DC blocking filter with given pole position*
- **.allpass** (float, READ/WRITE) *allpass filter with given coefficient*

STK - Reverbs

JCRev

- John Chowning's reverberator class.

This class is derived from the CLM JCreve function, which is based on the use of networks of simple allpass and comb delay filters. This class implements three series allpass units, followed by four parallel comb filters, and two decorrelation delay lines in parallel at the output.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.mix** (float, READ/WRITE) *mix level*

NRev

- CCRMA's NRev reverberator class.

This class is derived from the CLM NRev function, which is based on the use of networks of simple allpass and comb delay filters. This particular arrangement consists of 6 comb filters in parallel, followed by 3 allpass filters, a lowpass filter, and another allpass in series, followed by two allpass filters in parallel with corresponding right and left outputs.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.mix** (float, READ/WRITE) *mix level*

PRCRev

- Perry's simple reverberator class.

This class is based on some of the famous Stanford/CCRMA reverbs (NRev, KipRev), which were based on the Chowning/Moorer/Schroeder reverberators using networks of simple allpass and comb delay filters. This class implements two series allpass units and two parallel comb filters.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.mix** (float, READ/WRITE) *mix level*

STK - Components

Chorus

- STK chorus effect class.

This class implements a chorus effect.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.modFreq** (float, READ/WRITE) *modulation frequency*
- **.modDepth** (float, READ/WRITE) *modulation depth*
- **.mix** (float, READ/WRITE) *effect mix*

Modulate

- STK periodic/random modulator.

This class combines random and periodic modulations to give a nice, natural human modulation function.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.vibratoRate** (float, READ/WRITE) *set rate of vibrato*
- **.vibratoGain** (float, READ/WRITE) *gain for vibrato*
- **.randomGain** (float, READ/WRITE) *gain for random contribution*

PitShift

- STK simple pitch shifter effect class.

This class implements a simple pitch shifter using delay lines.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.mix** (float, READ/WRITE) *effect dry/wet mix level*
- **.shift** (float, READ/WRITE) *degree of pitch shifting*

SubNoise

- STK sub-sampled noise generator.

Generates a new random number every "rate" ticks using the C rand() function. The quality of the rand() function varies from one OS to another.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.rate** (int, READ/WRITE) *subsampling rate*

STK - File I/O

WvIn

- STK audio data input base class.

This class provides input support for various audio file formats. It also serves as a base class for "realtime" streaming subclasses.

WvIn loads the contents of an audio file for subsequent output. Linear interpolation is used for fractional "read rates".

WvIn supports multi-channel data in interleaved format. It is important to distinguish the `tick()` methods, which return samples produced by averaging across sample frames, from the `tickFrame()` methods, which return pointers to multi-channel sample frames. For single-channel data, these methods return equivalent values.

Small files are completely read into local memory during instantiation. Large files are read incrementally from disk. The file size threshold and the increment size values are defined in `WvIn.h`.

WvIn currently supports WAV, AIFF, SND (AU), MAT-file (Matlab), and STK RAW file formats. Signed integer (8-, 16-, and 32-bit) and floating-point (32- and 64-bit) data types are supported. Uncompressed data types are not supported. If using MAT-files, data should be saved in an array with each data channel filling a matrix row.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.rate** (float, READ/WRITE) *playback rate*
- **.path** (string, READ/WRITE) *specifies file to be played*

WaveLoop

- STK waveform oscillator class.
- see examples/dope.ck

[ctrl param]

- **.freq** (float, READ/WRITE) *frequency of playback (loops/second)*
- **.addPhase** (float, READ/WRITE) *offset by phase*
- **.addPhaseOffset** (float, READ/WRITE) *set phase offset*

WvOut

- STK audio data output base class.

This class provides output support for various audio file formats. It also serves as a base class for "realtime" streaming subclasses.

WvOut writes samples to an audio file. It supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which output single samples to all channels in a sample frame, from the tickFrame() method, which takes a pointer to multi-channel sample frame data.

WvOut currently supports WAV, AIFF, AIFC, SND (AU), MAT-file (Matlab), and STK RAW file formats. Signed integer (8-, 16-, and 32-bit) and floating-point (32- and 64-bit) data types are supported. STK RAW files use 16-bit integers by definition. MAT-files will always be written as 64-bit floats. If a data type specification does not match the specified file type, the data type will automatically be modified. Uncompressed data types are not supported.

Currently, WvOut is non-interpolating and the output rate is always Stk::sampleRate().

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

[ctrl param]

- **.matFilename** (string, WRITE only) *open a matlab file for writing*
- **.sndFilename** (string, WRITE only) *open snd file for writing*
- **.wavFilename** (string, WRITE only) *open WAVE file for writing*
- **.rawFilename** (string, WRITE only) *open raw file for writing*
- **.aifFilename** (string, WRITE only) *open AIFF file for writing*
- **.closeFile** (string, WRITE only) *close file properly*

events

Event

- event handler

[ctrl param]

- **.signal()** *signals first waiting shred*
- **.broadcast()** *signals all shreds*

MidiIn

- MIDI receiver

extends Event [ctrl param]

- **.open** (int, READ/WRITE) *set port to receive*
- **.recv** (MidiMsg, READ) *receives MIDI input*

(see MIDI tutorial)

MidiOut

- MIDI sender

extends Event [ctrl param]

- **.open** (int, READ/WRITE) *set port to send*

- **.send** (MidiMsg, WRITE) *sends MIDI output*

(see MIDI tutorial)

MidiMsg

- MIDI data holder

[ctrl param]

- **.data1** (int, READ/WRITE) *first byte of data (member variable)*
- **.data2** (int, READ/WRITE) *second byte of data (member variable)*
- **.data3** (int, READ/WRITE) *third byte of data (member variable)*

(see MIDI tutorial)

OscRecv

- Open Sound Control receiver

[ctrl param]

- **.port** (int, READ/WRITE) *set port to receive*
- **.listen** () *starts object listening to port*
- **.event** (string(name), string(type), READ/WRITE) *define string for event to receive*

(see events tutorial)

OscSend

- Open Sound Control sender

[ctrl param]

- **.setHost** (string(host), int(port), READ/WRITE) *set port on the host to receive*
- **.startMsg** (string(name), string(type), READ/WRITE) *define string for event to send*

(see events tutorial)

OscEvent

- Open Sound Control event

extends Event [ctrl param]

- **.nextMsg** (int, READ) *the number of messages in queue*
- **.getFloat** (float, READ) *gets float from message*
- **.getInt** (int, READ) *gets int from message*

(see events tutorial)