the SuperCollider \Help "Book"

// typeset from the help files

The Beaver brought paper, portfolio, pens,
And ink in an unfailing supplies:
While strange creepy creatures came out of their dens
And watched them with wondering eyes.

(L. Carroll, *The Hunting of the Snark*, Fit the Fifth, The Beaver's Lesson)

This book is a collection of all the SuperCollider help files available in the standard distribution. It has been typeset with ConTeXt, a TeX-based typesetting system for document preparation which allows high-quality automated typesetting with native PDF output. The ConTeXt code has been generated by TheCollidingChampollion, a Python module which analyses and translates the SuperCollider html help files. As TheCollidingChampollion is in its first version, it is still very experimental: in particular, syntax colorizing presents some bugs and some features are lacking (e.g. sometimes colors are misplaced and indentation is missing). The situation partly depends also on sources: the html files reproduces all the errors of the rtf files from which they are generated, and these are reproduced again in the ConTeXt files. More, the original rtf documentation lacks structure and an explicit markup would be needed to achieve better results (the best example is Thread help file). The book tries to reproduce the original hierarchy of the Help folder. As depth in the Help subfolders is changing, the book structure is a bit messy. A general criterium is that if files are at the same level of folders they are all listed before the folders. Note that:

1. in each document the original path is indicated in the header
2. each document has a unique ID

Other Meta-data and stylistic improvements are on the ToDo list. Japanese tutorial is missing. This work has been possible thanks to three open-source projects.

- **SuperCollider**: `http://supercollider.sourceforge.net/`
- **ConTeXt**: `http://www.pragma-ADE.com/`
- **Python**: `http://www.python.org`

Please report bugs, suggestions, cries & whispers to:
`andrea.valle@unito.it`
Compiled on January 16, 2007.
Andrea Valle
`http://www.semiotiche.it/andrea/`

1 3vs2

ID: 1

Backwards Compatibility

There are a number of classes and methods that have been added to allow for backwards compatibility with SC2 code. The most notable of these is Synth.play, which is basically a wrapper for Function.play.

```
  SinOsc                        // creates an arbitrarily named SynthDef and a Synth to play it
Synth       SinOsc                    // in SC3 just a wrapper for Function.play with fewer args
```

Both of these will create synth nodes on the default server. Note that neither requires the use of an Out.ar ugen; they simply output to the first audio bus. One can however add an Out to Function.play in order to specify.

```
Synth.play({ Out.ar(1, SinOsc.ar(440, 0, 0.5)) });
```

In general, one should be aware of this distinction when using this code. When copying such code for reuse with other SC3 classes (for example in a reusable SynthDef), it will usually be necessary to add an Out.ar. Although useful for quick testing these methods are generally inferior to SynthDef.play, as the latter is more direct, requires no modifications for general reuse, has greater general flexibility and has slightly less overhead. (Although this is insignificant in most cases, it could be relevant when large numbers of defs or nodes are being created.)

Like SynthDef.play, Function.play returns a Synth object which can then be messaged, etc. However, since Function.play creates an arbitrarily named SynthDef, one cannot reuse the resulting def, at least not without reading its name from the post window, or getting it from the Synth object.

```
//The following examples are functionally equivalent
x = { arg freq = 440; Out.ar(1, SinOsc.ar(freq, 0, 0.5)) }.play(fadeTime:  0);
      \freq          // you can set arguments
    Synth                  // get the arbitrary defname from x
x.free;
y.free;

x = SynthDef("backcompat-sine", { arg freq = 440; Out.ar(1, SinOsc.ar(freq, 0, 0.5)) }).play;
x.set(\freq, 880);
    Synth     "backcompat-sine"
```

```
x.free;
y.free;
```

Function.play is in general superior to both its SC2 equivalent and Synth.play. It has a number of significant features such as the ability to specify the output bus and fade times as arguments. See the **Function** helpfile for a more in-depth discussion.

A number of other classes and methods have also been added to improve compatibility. These are listed below. In general there are equivalent or better ways of doing the same things in SC3.

Synth *play use Function.play or SynthDef.play
GetFileDialog use CocoaDialog
GetStringDialog
Synth *stop use Server.freeAll
Synth *isPlaying Server.numSynths (this will include non-running nodes)
Mix *ar *arFill use Mix *new and *fill
SimpleNumber.rgb
Rawarray.write

ID: 2

Client versus Server Operations

Unlike in SC2 where language and synthesis were unified in a single application, SC3 divides its operations between a language application (the SuperCollider app on which you double-clicked to startup SC) and a synthesis-server application (a UNIX command-line application called scsynth, which is started when you press the boot button on the 'local' server window that is created by default at startup, or when you boot a Server from code). The two applications communicate between each other through UDP or TCP using a subset of CNMAT's Open Sound Control.

This is a radical departure from the previous architecture (a more detailed discussion of this and other matters can be found in the file **sc3 intro 2** in the Examples folder) and yields several important advantages:

The server can be controlled by programs other than the language app.
The language app can crash and synthesis will not stop.
The server can crash and the language will not.
The language and server apps can be running on different machines, even in different parts of the world. This allows for efficient 'division of labour' and network interactivity.

There is one notable drawback: The messaging process introduces a small amount of latency. This should not be confused with audio latency which can be quite low. It only means that there is a small, usually insignificant delay between the one side sending a message and the other receiving it and acting upon it. (This can be minimized by using the 'internal' server. See **Server** for more detail.)

What is crucial to understand is the distinct functions of each side. The server app is a lean and efficient program dedicated to audio functions. It knows nothing about SC code, objects, OOP, or anything else to do with the SC language. It has (at least for the moment) little programmatic ability.

When one creates a Synth object in the language app, that object is only the clientside *representation* of a node on the server. The language app provides you with convienent OOP functionality to keep track of and manipulate things on the server. All of this functionality is possible to do 'by hand' using the sendMsg method of **Server**, and other similar messages. For instance:

```
s = Server.default;
```

```
s.boot;

// this
n = s.nextNodeID;
 s.sendMsg("/s_new"  "default"      // use the SynthDef "default"
s.sendMsg("/n_free", n);

// is equivalent to
 x = Synth "default"   // create a synth on the default server (s) and allocate an ID for it
 x.free;    // free the synth, its ID and resources
```

The latter method gives you certain functionality. It gets a node ID for you automatically, it allows you to control the Synth in syntactically elegant and efficient ways (see the **Synth** and **Node** helpfiles), and to access all the advantages of object oriented programming while doing so. Encapsulating the complexities and bookeeping greatly reduces the chance of bugs in your own code.

It also has a small amount of overhead. It requires clientside CPU cycles and memory to create and manipulate an object. Normally this is not significant, but there may be times when you would prefer to use the less elegant, and less expensive first method, for instance when creating large numbers of grains which will simply play and then deallocate themselves.

Thus it is possible to create synth nodes on the server without actually creating Synth objects, providing you are willing to do the required housekeeping yourself. The same is true of group nodes, buffers, and buses. A more detailed discussion of these concepts can be found in the **NodeMessaging** helpfile.

In conclusion, the crucial thing to remember is the distinction between things like nodes, busses, buffers, and servers and the objects that *represent* them in the language app (i.e. instances of **Node**, **Bus**, **Buffer**, and **Server**). Keeping these conceptually distinct will help avoid much confusion.

ID: 3

SuperCollider 3 versus SuperCollider 2

There are a number of ways in which SuperCollider 3 (or SCServer) is very different from SC2.

A discussion of this is organised in the following documents:

ClientVsServer - Separate language and synthesis apps.

SynthDefsVsSynths - The use of precompiled SynthDefs as opposed to always compiling on the fly.

Spawning - The lack of the Spawn and TSpawn ugens and their various convienence classes.

Sync-Async - The problem of simultaneous synchronous and asynchronous execution.

Backwards-Compatibility - A discussion some classes and methods which have been added to improve compatibility with SC2 code, and their limitations.

(Select the bold text and type cmd-shift-? to open the corresponding file.)

Note that these documents are not intended to be exhaustive tutorials, just an introduction to some of the differences. Close examination of the helpfiles of relevant classes should help to fill in the details. These files may be of some use to beginners as well.

ID: 4

"Spawning" and "TSpawning"

In SC2, Spawn and TSpawn were two of the most powerful and commonly used UGens. In SC3 the idea of a top level Synth in which everything is spawned is no longer valid. Synthesis is always running (at least as long as a server is) and new Synths can be created on the fly. This arrangement results in even greater flexibility than in SC2, but requires a slightly different approach.

In SC3 one can create Synths at any time simply by executing blocks of code.

```
// do this
(
       SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
  }).play; // SynthDef-play returns a Synth object.
)

// then do this
(
SynthDef("help-Rand", { arg out=0;
Out.ar(out,
FSinOsc.ar(
    Rand                 // frequency between 200 and 400 Hz
0, Line.kr(0.2, 0, 1,
doneAction:2))   // frees itself
)
}).play(s);
)

x.free;
```

Clocks, such as **SystemClock**, provide a way to schedule things at arbitrary points in the future. This is similar to Synth.sched in SC2.

```
  (
SystemClock.sched(2.0,{
  "2.0 seconds later"         // this could be any code, including Synth creation
```

```
  nil // this means don't repeat
});
)
```

In SC3 time-based sequences of events can be implemented using Routines. A **Routine** which yields a number can be scheduled using a clock:

```
(
var w, r;
w = SCWindow("trem", Rect(512, 256, 360, 130));
w.front;
r = Routine({ arg time;
60.do({ arg i;
    0.05.yield; // wait for 0.05 seconds
{
w.bounds = w.bounds.moveBy(10.rand2, 10.rand2);
w.alpha = cos(i*0.1pi)*0.5+0.5;
}.defer;
});
   1.yield; // wait for 1 second before closing w
w.close;
});
 SystemClock
)
```

Note that this implementation avoids one of the stranger aspects of the SC2 approach: The need to start a Synth to schedule time-based behavior, even if no audio is involved.

Both **SystemClock** and **AppClock** (a less accurate version which can call Cocoa primitives) have only class methods. Thus one does not create instances of them. If you need to have an individual clock to manipulate (for instance to manipulate the tempi of different sequences of events) you can use **TempoClock**.

A simple SC2 Spawn example is shown below, followed by its translation into SC3 style code.

```
// This will not execute in SC3
(

Synth.play({
```

```
Spawn.ar({
EnvGen.ar(Env.perc) * SinOsc.ar(440,0,0.1)
},
  1, // one channels
  1) // new event every second
}))

// The same example in SC3 (will execute)

s = Server.default;
s.boot;
(
 SynthDef "help-EnvGen"   arg
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s);
)

(
r = Routine.new({ { Synth.new("help-EnvGen"); 1.yield; }.loop }); // loop every one second
 SystemClock
)
```

Note that the above example uses a precompiled **SynthDef**. This results in a lower CPU spike when Synths are created than SC2-style Spawning. It is possible to create SynthDefs on the fly, if this is necessary, but a great deal of variation can be achieved with arguments, or with ugens such as **Rand** and **TRand**. See the helpfile **SynthDefsVsSynths** for more detail.

```
// SynthDefs on the fly

s = Server.default;
s.boot;
(
t = TempoClock.new;
r = Routine.new({
10.do({
   // could be done with an argument instead of a new def, but proves the point
```

```
SynthDef("help-EnvGen" ++ i,{ arg out=0;
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(100 + (100 * t.elapsedBeats),0,0.1)
)
}).play(s);
1.yield;
});
 }).play(t); // Note the alternative syntax:  Routine.play(aClock)
)
```

Note the alternative syntax for playing a Routine. aClock.play(aRoutine) and aRoutine.play(aClock) are functionally equivalent. The two make different things more or less convienent, like sending messages to the **Routine** or **Clock**. (See the **play** helpfile for a more detailed discussion.) For instance:

```
(
 // this, that and the other
r = Routine.new({var i = 0; { ("this:  " ++ i).postln; i = i + 1; 1.yield; }.loop });
q = Routine.new({var i = 0; { ("that:  " ++ i).postln; i = i + 1; 1.yield; }.loop });
 t = Routine.new({var i = 0; { ("the other:  " ++ i).postln; i = i + 1; 1.yield; }.loop });
)

 SystemClock           // start this
 SystemClock           // start that
 SystemClock           // start the other

 r.stop;     // stop this but not that or the other
 q.reset;    // reset that while playing

 c = TempoClock      // make a TempoClock
 r.reset;    // have to reset this because it's stopped
 c.play(r);   // play this in the new clock; starts from the beginning
 c.tempo = 16;   // increase the tempo of this

 SystemClock         // clear EVERYTHING scheduled in the SystemClock; so that and the other
      // but not this

 c.clear;    // clear everything scheduled in c, i.e.  this
 c.play(r);   // since it wasn't stopped, we don't have to reset this
```

```
    // and it picks up where it left off

c.stop    // stop c, destroy its scheduler, and release its OS thread
```

For convenience pauseable scheduling can be implemented with a **Task**. Task.new takes two arguments, a function and a clock, and creates it's own **Routine**. If you don't specify a clock, it will create a **TempoClock** for you. Since you don't have to explicitly create a Clock or Routine, use of Task can result in code that is a little more compact.

```
(
t = Task.new({
inf.do({ arg i;
i.postln;
0.5.wait
});
});
)

t.start;     // Start it
t.stop;    // Stop it
 t.start;    // Start again from the beginning
 t.reset;    // Reset on the fly
 t.stop;    // Stop again
 t.resume;    // Restart from where you left off
                      // Get the Task's clock and change the tempo.  This works since the
    // default is a TempoClock.
  t.pause;    // Same as t.stop
```

TSpawn's functionality can be replicated with **SendTrig** and **OSCresponder** or **OSCresponderNode**. See their individual helpfiles for details on their arguments and functionality.

```
s = Server.default;
s.boot;

(
 // this Synth will send a trigger to the client app
 SynthDef "help-SendTrig"
  SendTrig
```

```
   Dust          // trigger could be anything, e.g.  Amplitude.kr(AudioIn.ar(1) > 0.5)
0,0.9
);
}).send(s);
)

(
 // this recieves the trigger on the client side and 'Spawns' a new Synth on the server
OSCresponder(s.addr,'/tr',{
SynthDef("help-EnvGen",{ arg out=0;
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).play(s);
}).add;

 // Start 'spawning'
 Synth "help-SendTrig"
)
```

ID: 5

Synchronous and Asynchronous Execution

Using a program such as SuperCollider introduces a number of issues regarding timing and order of execution. Realtime audio synthesis requires that samples are calculated and played back at a certain rate and on a certain schedule, in order to avoid dropouts, glitches, etc. Other tasks, such as loading a sample into memory, might take arbitrary amounts of time, and may not be needed within a definite timeframe. This is the difference between synchronous and asynchronous tasks.

Problems can arise when synchronous tasks are dependent upon the completion of asynchronous ones. For instance trying to play a sample that may or may not have been completely loaded yet.

In SC2 this was relatively simple to handle. One scheduled synchronous tasks during synthesis, i.e. within the scope of a Synth.play. Asynchronous tasks were executed in order, outside of synthesis. Thus one would first create buffers, load samples into them, and then start synthesis and play them back. The interpreter made sure that each step was only done when the necessary previous step had been completed.

In SC3 the separation of language and synth apps creates a problem: How does one side know that the other has completed necessary tasks, or in other words, how does the left hand know if the right is finished? The flexibility gained by the new architecture introduces another layer of complexity, and an additional demand on the user.

A simple way to deal with this is to execute code in blocks. In the following code, for instance, each block or line of code is dependent upon the previous one being completed.

```
// Execute these one at a time

// Boot the local Server
(
s = Server.local;
s.boot;
)

// Compile a SynthDef and write it to disk
(
 SynthDef "Help-SynthDef"
```

```
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).writeDefFile;
)

// Load it into the server
s.loadSynthDef("Help-SynthDef");

// Create a Synth with it
x = Synth.new("Help-SynthDef", s);

// Free the node on the server
x.free;

// Allow the client-side Synth object to be garbage collected
x = nil;
```

In the previous example it was necessary to use interpreter variables (the variables a-z, which are declared at compile time) in order to refer to previously created objects in later blocks or lines of code. If one had declared a variable within a block of code (i.e.`var mySynth;`) than it would have only persisted within that scope. (See the helpfile **Scope** for more detail.)

This style of working, executing lines or blocks of code one at a time, can be very dynamic and flexible, and can be quite useful in a performance situation, especially when improvising. But it does raise the issues of scope and persistence. Another way around this that allows for more descriptive variable names is to use environment variables (i.e. names that begin with , so mysynth; see the **Environment** helpfile for details). However, in both methods you become responsible for making sure that objects and nodes do not persist when you no longer need them.

```
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s);
)

 // make a Synth and assign it to an environment variable
mysynth = Synth.new("Help-SynthDef", s);
```

```
 // free the synth
mysynth.free;

 // but you've still got a Synth object
mysynth.postln;

 // so remove it from the Environment so that the Synth will be garbage collected
currentEnvironment.removeAt(\mysynth);
```

But what if you want to have one block of code which contains a number of synchronous and asynchronous tasks. The following will cause an error, as the SynthDef that the server needs has not yet been received.

```
// Doing this all at once produces the error "FAILURE /s_new SynthDef not found"
(
var name;
 name = "Rand-SynthDef"                // use a random name to ensure it's not already loaded
SynthDef(name,
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s);

Synth.new(name, s);
)
```

A crude solution would be to schedule the dependant code for execution after a seemingly sufficient delay using a clock.

```
// This one works since the def gets to the server app first
(
var name;
name = "Rand-SynthDef" ++ 400.0.rand;
SynthDef(name,
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s);

SystemClock.sched(0.05, {Synth.new(name, s);}); // create a Synth after 0.05 seconds
)
```

Although this works, it's not very elegant or efficient. What would be better would be to have the next thing execute immediately upon the previous thing's completion. To explore this, we'll look at an example which is already implemented.

You may have realized that first example above was needlessly complex. SynthDef-play will do all of this compilation, sending, and Synth creation in one stroke of the enter key.

```
// All at once
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).play(s);
)
```

Let's take a look at the method definition for SynthDef-play and see what it does.

```
play { arg target,args,addAction=\addToTail;
var synth, msg;
target = target.asTarget;

synth = Synth.basicNew(name,target.server); // create a Synth, but not a synth node
msg = synth.newMsg(target, addAction, args);// make a message that will add a synth node
  this                          // ** send the def, and the message as a completion message
  ^synth // return the Synth object
}
```

This might seem a little complicated if you're not used to mucking about in class definitions, but the important part is the second argument to `this.send(target.server, msg);`. This argument is a completion message, it is a message that the server will execute when the send action is complete. In this case it says create a synth node on the server which corresponds to the **Synth** object I've already created, when and only when the def has been sent to the server app. (See the helpfile **Server-Command-Reference** for details on messaging.)

Many methods in SC have the option to include completion messages. Here we can use SynthDef-send to accomplish the same thing as SynthDef-play:

```
// Compile, send, and start playing
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s, ["s_new", "Help-SynthDef", x = s.nextNodeID]);
  // this is 'messaging' style, see below
)
s.sendMsg("n_free", x);
```

The completion message needs to be an OSC message, but it can also be some code which when evaluated returns one:

```
// Interpret some code to return a completion message.  The .value is needed.
// This and the preceding example are essentially the same as SynthDef.play
(
   SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s, {x = Synth.basicNew("Help-SynthDef"); x.newMsg; }.value); // 'object' style
)
x.free;
```

If you prefer to work in 'messaging' style, this is pretty simple. If you prefer to work in 'object' style, you can use the commands like newMsg, setMsg, etc. with objects to create appropriate server messages. The two proceeding examples show the difference. See the **NodeMessaging** helpfile for more detail.

In the case of **Buffer** objects a function can be used as a completion message. It will be evaluated and passed the **Buffer** object as an argument. This will happen after the Buffer object is created, but before the message is sent to the server. It can also return a valid OSC message for the server to execute upon completion.

```
(
SynthDef("help-Buffer",{ arg out=0,bufnum;
Out.ar( out,
PlayBuf.ar(1,bufnum,BufRateScale.kr(bufnum))
)
}).load(s);
```

```
y = Synth.basicNew("help-Buffer"); // not sent yet
 b = Buffer        "sounds/a11wlk01.wav"
completionMessage:  { arg buffer;
   // synth add its s_new msg to follow
   // after the buffer read completes
y.newMsg(s,\addToTail,[\bufnum,buffer.bufnum])
  }); // .value NOT needed, unlike in the previous example

)

 // when done...
y.free;
b.free;
```

The main purpose of completion messages is to provide OSC messages for the server to execute immediately upon completion. In the case of Buffer there is essentially no difference between the following:

```
(
b = Buffer.alloc(s, 44100,
completionMessage:  { arg buffer; ("bufnum:" + buffer.bufnum).postln; });
)
 // this is equivalent to the above
(
b = Buffer.alloc;
("bufnum:" + b.bufnum).postln;
)
```

One can also evaluate a function in response to a 'done' message, or indeed any other one, using an **OSCresponder** or an **OSCresponderNode**. The main difference between the two is that the former allows only a single responder per command, where as the latter allows multiple responders. See their respective helpfiles for details.

```
(
SynthDef "help-SendTrig"
SendTrig.kr(Dust.kr(1.0), 0, 0.9);
}).send(s);

// register to receive this message
a = OSCresponderNode(s.addr, '/done', { arg time, responder, msg;
```

```
("This is the done message for the SynthDef.send:" + [time, responder, msg]).postln;
                        // remove me automatically when done
b = OSCresponderNode(s.addr, '/tr', { arg time, responder, msg;
[time, responder, msg].postln;
}).add;
c = OSCresponderNode(s.addr, '/tr', { arg time, responder, msg;
"this is another call".postln;
}).add;
)


x = Synth      "help-SendTrig"
b.remove;
c.remove;
x.free;
```

ID: 6

SynthDefs versus Synths

In SC2 Synth.play was the standard way to compile a ugenGraphFunc and play it. Each time you executed Synth.play, or Spawned a new event, that function was compiled anew. SC3 on the other hand, makes use of what are called SynthDefs. A **SynthDef** takes a ugenGraphFunc and compiles it to a kind of bytecode (sort of like Java bytecode) which can be understood by the server app. The server reads the SynthDef and creates a synth node based upon it.

SynthDefs can be precompiled and saved to disk. Any def saved in the synthdefs/ directory (or in any directory set in the environment variable SC_SYNTHDEF_PATH) will be loaded into memory by a local **Server** when it is booted. If the def being used in a new **Synth** is already compiled and loaded, there is much less of a CPU spike when creating a new **Synth** than there was in SC2.

SynthDefs can also be compiled and loaded into the Server without writing them to disk. This can be done while performing.

The downside of this is that precompiled SynthDefs lack some of the programmatic flexibility that was one of SC2's great strengths. Much of this flexibility is gained back however, through the ability to set and change arguments (which you build into your ugenGraphFunc), and through new ugens such as **Rand** and **TRand**.

When maximum flexibility is required, it is still possible to compile and send SynthDefs 'on the fly', albeit with SC2-like CPU spikes and a small amount of messaging latency.

It is important to understand that creating and sending SynthDefs is *asynchronous.* This means that it is impossible to determine precisely how long it will take to compile and send a **SynthDef**, and thus when it will be available for creating new Synths. A simple way around this is to execute code in blocks, selecting them one at a time. More complicated is to use completion messages. SynthDef.play takes care of this for you, and returns a Synth object which you can then manipulate. See the example below.

Another important distinction is between Synth in SC2 and **Synth** in SC3. The latter is a client-side object which *represents* a synth node on the server. Although it has some of the same methods, it does not function in the same way. There is no top level Synth in SC3, within which all scheduling and creation of other Synths occurs. There are only Synth objects which represent synth nodes on the server. These can be created at any

time, within any scope.

Examples

```
(
s = Server.local;
s.boot;
)

// Compile a SynthDef and write it to disk
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).writeDefFile;
)

// Compile, write, and load it to the server
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).load(s);
)

// Load it to the server without writing to disk
(
 SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).send(s);
)

// Create a Synth with it
x = Synth.new("Help-SynthDef", s);
x.free;

// Shorthand method to compile and write a SynthDef, and then play it in a Synth when done.
// Look familiar?
(
```

```
      SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).play(s);
)
 // The above only starts the new Synth after the def has been sent to the server.
  // Note that SynthDef.play returns a Synth object

  x.set(\out      // change one of the arguments
x.free;

// SynthDef with a parameter that will be randomly determined each time a new Synth is created
// (try it several times to hear the differences)

(
 SynthDef "help-RandFreq"    arg
Out.ar(out,
FSinOsc.ar(
    Rand                // frequency between 200 and 400 Hz
0, Line.kr(0.2, 0, 1, doneAction:2))
)
}).play(s);
)
```

2 BinaryOps

ID: 7

absdif absolute value of the difference

BinaryOperator

absdif(a, b)
a absdif: b
a.absdif(b)

Return the value of abs(a - b). Finding the magnitude of the difference of two values is a common operation.

```
// creates a rhythm
(
{
var mul;
mul = 0.2 absdif:  FSinOsc.ar(2, 0, 0.5);
FSinOsc.ar(440, 0, mul);
}.play;)
```

ID: 8

+ addition

BinaryOperator

a + b

```
{ FSinOsc.ar(800, 0, 0.1) + PinkNoise.ar(0.1) }.play;

// DC offset; add:  0.1 would be more efficient
{ FSinOsc.ar + 0.1 }.play
```

ID: 9

amclip two quadrant multiply

BinaryOperator

amclip(a, b)
a amclip: b
a.amclip(b)

0 when b <= 0, a*b when b > 0

```
{ WhiteNoise.ar.amclip(FSinOsc.kr(1,0.2)) }.play; // makes a sine envelope
```

ID: 10

atan2 arctangent

BinaryOperator

atan2(y, x)
y atan2: x
y.atan2(x)

Returns the arctangent of y/x.

See also **hypot**.

OK, now we can add a pan to the **hypot** doppler examples by using atan2 to find the azimuth,
or direction angle, of the sound source.
Assume speakers at +/- 45 degrees and clip the direction to between those.

```
(
{
var x, y, distance, velocity, pitchRatio, amplitude, azimuth, panValue;
 // object travels 200 meters in 6 secs (=120kph) passing 10 meters
 // from the listener
x = 10;
y = LFSaw.kr(1/6, 0, 100);
distance = hypot(x, y);
velocity = Slope.kr(distance);
pitchRatio = (344 - velocity) / 344;  // speed of sound is 344 meters/sec
amplitude = 10 / distance.squared;
azimuth = atan2(y, x); // azimuth in radians
panValue = (azimuth / 0.5pi).clip2(1);
Pan2.ar(FSinOsc.ar(1000 * pitchRatio), panValue, amplitude)
}.play)

(
{
var x, y, distance, velocity, pitchRatio, amplitude, motorSound,
azimuth, panValue;
 // object travels 200 meters in 6 secs (=120kph) passing 10 meters
```

```
 // from the listener
x = 10;
y = LFSaw.kr(1/6, 0, 100);
distance = hypot(x, y);
amplitude = 40 / distance.squared;
motorSound = RLPF.ar(FSinOsc.ar(200, LFPulse.ar(31.3, 0, 0.4)), 400, 0.3);
azimuth = atan2(y, x); // azimuth in radians
panValue = (azimuth / 0.5pi).clip2(1); // make a value for Pan2 from azimuth
Pan2.ar(DelayL.ar(motorSound, 110/344, distance/344), panValue, amplitude)
}.play)
```

ID: 11

BinaryOpUGen

superclass: UGen

BinaryOpUGens are created as the result of a binary operator applied to a UGen.

```
(SinOsc.ar(200) * ClipNoise.ar).dump;
(SinOsc.ar(200).thresh(0.5)).dump;
```

The use of the binary operators ***** and **thresh** above each instantiate a BinaryOpUGen. Do not confuse the operators themselves (which are methods) with the resulting BinaryOpUGen, which is an object. When applied to other classes they may not return new objects, and can behave in a more straightforward manner. See for example **SimpleNumber**.

There are helpfiles for each the different operators, listed below.

The operators >, >=, <, and <= are particularly useful for triggering. They should not be confused with their use in conditionals. Compare

```
(1 > 0).if({"1 is greater than 0".postln}); // > returns a boolean
```

with

```
  // trigger an envelope
{
var trig;
trig = SinOsc.ar(1) > 0.1;
Out.ar(0,
EnvGen.kr(Env.perc, trig, doneAction:  0)
* SinOsc.ar(440,0,0.1)
)
            // > outputs 0 or 1
```

See the individual helpfiles for more detail.

The following operators have their own helpfiles:

+ - * / ** absdif amclip atan2 clip2 difsqr excess fold2 hypot hypotApx max min ring1 ring2 ring3 ring4 round scaleneg sqrdif sqrsum sumsqr thresh trunc wrap2

ID: 12

clip2 bilateral clipping

BinaryOperator

clip2(a, b)
a clip2: b
a.clip2(b)

clips input wave a to +/- b

```
Server.internal.boot;

{ FSinOsc.ar(400).clip2(0.2) }.scope; // clipping distortion

{ FSinOsc.ar(1000).clip2(Line.kr(0,1,8)) }.scope;
```

ID: 13

difsqr difference of squares

BinaryOperator

difsqr(a, b)
a difsqr: b
a.difsqr(b)

Return the value of (a*a) - (b*b). This is more efficient than using separate unit generators for each operation.

```
{ (FSinOsc.ar(800) difsqr:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
((a * a) - (b * b)) * 0.125
}.play)
```

ID: 14

/ division

BinaryOperator

a / b

Division can be tricky with signals because of division by zero.

```
{ PinkNoise.ar(0.1) / FSinOsc.kr(10, 0.5, 0.75) }.play;
```

ID: 15

excess clipping residual

BinaryOperator

excess(a, b)
a excess: b
a.excess(b)

Returns the difference of the original signal and its clipped form: (a - clip2(a,b)).

```
{ FSinOsc.ar(1000).excess(Line.kr(0,1,8)) }.play;
```

ID: 16

** exponentiation

BinaryOperator

a ** b

When the signal is negative this function extends the usual definition of exponentiation and returns neg(neg(a) ** b). This allows exponentiation of negative signal values by noninteger exponents.

```
(
{
var a;
a = FSinOsc.ar(100);
[a, a**10]
}.play
)
```

ID: 17

fold2 bilateral folding

BinaryOperator

fold2(a, b)
a fold2: b
a.fold2(b)

folds input wave a to +/- b

```
{ FSinOsc.ar(1000).fold2(Line.kr(0,1,8)) }.scope;
```

ID: 18

>= greater than or equal

BinaryOperator

a >= b

Result is 1 if a >= b, otherwise it is 0. This can be useful for triggering purposes, among other things:

```
s = Server.local;
s.boot;

  // trigger an envelope
{
var trig;
trig = SinOsc.ar(1) >= 0;
Out.ar(0,
EnvGen.kr(Env.perc, trig, doneAction:  0)
* SinOsc.ar(440,0,0.1)
)
}.play(s);)

  // trigger a synth
SynthDef "help-EnvGen"   arg
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s);

// This synth has no output.  It only checks amplitude of input and looks for a transition from < 0.2

// to > 0.2

SynthDef "help->= trig"
SendTrig.kr(Amplitude.kr(AudioIn.ar(1)) >= 0.2);
}).play(s);
```

```
// responder to trigger synth
OSCresponderNode(s.addr,'/tr',{ "triggered".postln; Synth.new("help-EnvGen") }).add;
)
```

ID: 19

> greater than

BinaryOperator

a > b

Result is 1 if a > b, otherwise it is 0. This can be useful for triggering purposes, among other things:

```
s = Server.local;
s.boot;

( // trigger an envelope
{
var trig;
trig = SinOsc.ar(1) > 0;
Out.ar(0,
EnvGen.kr(Env.perc, trig, doneAction:  0)
* SinOsc.ar(440,0,0.1)
)
}.play(s);)

( // trigger a synth
SynthDef "help-EnvGen"   arg
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s);

// This synth has no output.  It only checks amplitude of input and looks for a transition from < 0.2

// to > 0.2

SynthDef "help-> trig"
SendTrig.kr(Amplitude.kr(AudioIn.ar(1)) > 0.2);
}).play(s);
```

```
// responder to trigger synth
OSCresponderNode(s.addr,'/tr',{ "triggered".postln; Synth.new("help-EnvGen") }).add;
)
```

ID: 20

hypot hypotenuse

BinaryOperator

hypot(x, y)
x hypot: y
x.hypot(y)

Returns the square root of the sum of the squares of a and b. Or equivalently, the distance from the origin
to the point (x, y).
See also **atan2**.

In this example, hypot is used to calculate a doppler shift pitch and amplitude based on distance.

```
(
{
var x, y, distance, velocity, pitchRatio, amplitude;
 // object travels 200 meters in 6 secs (=120kph) passing 10 meters
 // from the listener
x = 10;
y = LFSaw.kr(1/6, 0, 100);
distance = hypot(x, y);
velocity = Slope.kr(distance);
pitchRatio = (344 - velocity) / 344;  // speed of sound is 344 meters/sec
amplitude = 10 / distance.squared;
FSinOsc.ar(1000 * pitchRatio, 0, amplitude)
}.play)
```

The next example uses the distance to modulate a delay line.

```
(
{
var x, y, distance, velocity, pitchRatio, amplitude, motorSound;
 // object travels 200 meters in 6 secs (=120kph) passing 10 meters
 // from the listener
x = 10;
```

Where: Help→BinaryOps→Hypot

```
y = LFSaw.kr(1/6, 0, 100);
distance = hypot(x, y);
amplitude = 40 / distance.squared;
motorSound = RLPF.ar(FSinOsc.ar(200, 0, LFPulse.ar(31.3, 0, 0.4)), 400, 0.3);
DelayL.ar(motorSound, 110/344, distance/344, amplitude)
}.play)
```

ID: 21

hypotApx hypotenuse approximation

BinaryOperator

hypotApx(x, y)
x hypotApx: y
x.hypotApx(y)

Returns an approximation of the square root of the sum of the squares of x and y.

The formula used is :

abs(x) + abs(y) - ((sqrt(2) - 1) * min(abs(x), abs(y)))

hypotApx is used to implement Complex method magnitudeApx.
This should not be used for simulating a doppler shift because it is discontinuous. Use hypot.

See also **hypot**, **atan2**.

ID: 22

<= less than or equal

BinaryOperator

a <= b

Result is 1 if a <= b, otherwise it is 0. This can be useful for triggering purposes, among other things:

```
s = Server.local;
s.boot;

  // trigger an envelope
{
var trig;
trig = SinOsc.ar(1) <= 0;
Out.ar(0,
EnvGen.kr(Env.perc, trig, doneAction:  0)
* SinOsc.ar(440,0,0.1)
)
}.play(s);)

( // trigger a synth
SynthDef "help-EnvGen"   arg
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s);

// This synth has no output.  It only checks amplitude of input and looks for a transition from > 0.2

// to < 0.2

SynthDef "help-<= trig"
SendTrig.kr(Amplitude.kr(AudioIn.ar(1)) <= 0.2);
}).play(s);
```

```
// responder to trigger synth
OSCresponderNode(s.addr,'/tr',{ "triggered".postln; Synth.new("help-EnvGen") }).add;
)
```

ID: 23

< less than

BinaryOperator

a < b

Result is 1 if a < b, otherwise it is 0. This can be useful for triggering purposes, among other things:

```
s = Server.local;
s.boot;

  // trigger an envelope
{
var trig;
trig = SinOsc.ar(1) < 0;
Out.ar(0,
EnvGen.kr(Env.perc, trig, doneAction:  0)
* SinOsc.ar(440,0,0.1)
)
}.play(s);)

  // trigger a synth
SynthDef "help-EnvGen"   arg
Out.ar(out,
EnvGen.kr(Env.perc,1.0,doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s);

// This synth has no output.  It only checks amplitude of input and looks for a transition from > 0.2

// to < 0.2

SynthDef "help-< trig"
SendTrig.kr(Amplitude.kr(AudioIn.ar(1)) <= 0.2);
}).play(s);
```

```
// responder to trigger synth
OSCresponderNode(s.addr,'/tr',{ "triggered".postln; Synth.new("help-EnvGen") }).add;
)
```

ID: 24

max maximum

BinaryOperator

max(a, b)
a max: b
a.max(b)

```
(
{
var z;
z = FSinOsc.ar(500);
z max:  FSinOsc.ar(0.1);
        // modulates and envelopes z
```

ID: 25

min minimum

BinaryOperator

min(a, b)
a min: b
a.min(b)

```
(
{
var z;
z = FSinOsc.ar(500);
z min:  FSinOsc.ar(0.1);
        // distorts and envelopes z
```

ID: 26

% modulo

BinaryOperator

a % b

Outputs a modulo b.

```
FSinOsc                    // results in a sawtooth wave
```

ID: 27

* multiplication

BinaryOperator

a * b

```
// Same as mul:  0.5
{ SinOsc.ar(440) * 0.5 }.play;

// This creates a beating effect (subaudio rate).
{ FSinOsc.kr(10) * PinkNoise.ar(0.5) }.play;

// This is ring modulation.
{ SinOsc.ar(XLine.kr(100, 1001, 10), 0, 0.5) * SyncSaw.ar(100, 200, 0.5) }.play;
```

ID: 28

ring1 ring modulation plus first source

BinaryOperator

Return the value of ((a*b) + a). This is more efficient than using separate unit generators for the multiply and add.

See also *****, **ring1**, **ring2**, **ring3**, **ring4**.

```
{ (FSinOsc.ar(800) ring1:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
((a * b) + a) * 0.125
}.play)
```

normal ring modulation:

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
(a * b) * 0.125
}.play)
```

ID: 29

ring2 ring modulation plus both sources

BinaryOperator

Return the value of ((a*b) + a + b). This is more efficient than using separate unit generators for the multiply and adds.

See also *, **ring1**, **ring2**, **ring3**, **ring4**.

```
{ (FSinOsc.ar(800) ring2:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
((a * b) + a + b) * 0.125
}.play)
```

ID: 30

ring3 ring modulation variant

BinaryOperator

Return the value of (a*a *b). This is more efficient than using separate unit generators for each multiply.

See also *****, **ring1**, **ring2**, **ring3**, **ring4**.

```
{ (FSinOsc.ar(800) ring3:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
(a * a * b) * 0.125;
}.play)
```

ID: 31

ring4 ring modulation variant

BinaryOperator

Return the value of ((a*a *b) - (a*b*b)). This is more efficient than using separate unit generators for each operation.

See also *****, **ring1**, **ring2**, **ring3**, **ring4**.

```
{ (FSinOsc.ar(800) ring4:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
((a * a * b) - (a * b * b)) * 0.125
}.play)
```

ID: 32

round rounding

BinaryOperator

round(a, b)
a round: b
a.round(b)

Rounds a to the nearest multiple of b.

ID: 33

scaleneg scale negative part of input wave

BinaryOperator

scaleneg(a, b)
a scaleneg: b
a.scaleneg(b)

a*b when a < 0, otherwise a.

```
{ FSinOsc.ar(500).scaleneg(Line.ar(1,-1,4)) }.play;
```

ID: 34

sqrdif square of the difference

BinaryOperator

Return the value of (a - b)**2. This is more efficient than using separate unit generators for each operation.

```
{ (FSinOsc.ar(800) sqrdif:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b, c;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
c = a - b;
(c * c) * 0.125
}.play)
```

ID: 35

sqrsum square of the sum

BinaryOperator

Return the value of (a + b)**2. This is more efficient than using separate unit generators for each operation.

```
{ (FSinOsc.ar(800) sqrsum:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b, c;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
c = a + b;
(c * c) * 0.125
}.play)
```

ID: 36

- subtraction

BinaryOp

a - b

Subtracts the output of a ugen from something else.

```
(
{
var z;
z = FSinOsc.ar(800,0.25);
      // results in silence
}.play)
```

ID: 37

sumsqr sum of squares

BinaryOperator

Return the value of (a*a) + (b*b). This is more efficient than using separate unit generators for each operation.

```
{ (FSinOsc.ar(800) sumsqr:  FSinOsc.ar(XLine.kr(200,500,5))) * 0.125 }.play;
```

same as :

```
(
{
var a, b;
a = FSinOsc.ar(800);
b = FSinOsc.ar(XLine.kr(200,500,5));
((a * a) + (b * b)) * 0.125
}.play)
```

ID: 38

thresh signal thresholding

BinaryOperator

thresh(a, b)
a thresh: b
a.thresh(b)

0 when a $<$ b, otherwise a.

```
{ LFNoise0.ar(50, 0.5) thresh:  0.45 }.play // a low-rent gate
```

ID: 39

trunc truncation

BinaryOperator

trunc(a, b)
a trunc: b
a.trunc(b)

Truncate a to a multiple of b.

ID: 40

wrap2 bilateral wrapping

BinaryOperator

wrap2(a, b)
a wrap2: b
a.wrap2(b)

wraps input wave to +/- b

```
{ FSinOsc.ar(1000).wrap2(Line.kr(0,1.01,8)) }.scope;
```

3 Collections

ID: 41

Array

Superclass: ArrayedCollection

Arrays are ArrayedCollections whose slots may contain any object. Arrays have a fixed maximum size beyond which they cannot grow. For expandable arrays, use the List class.

An array can be created with a fixed maxiumum capacity:

```
z = Array.new(size);
```

Which will return an array of size 0, but the capability to add up to 32 objects.

```
z = z.add(obj);
```

z now has a size of 1.

For Arrays, the 'add' method may or may not return the same Array object. It will add the argument to the receiver if there is space, otherwise it returns a new Array object with the argument added. Thus the proper usage of 'add' with an Array is to always assign the result as follows:

```
z = z.add(obj);
```

This allows an efficient use of resources, only growing the array when it needs to. The List class manages the Array for you, and in many cases in more suitable.

An array can be created with all slots filled with nils:

```
z = Array.newClear(size);
```

Elements can be put into an existing slot:

```
a.put(2,obj);
```

And accessed :

```
  a.at(2); // these are equivalent
a[2];
```

See **[ArrayedCollection]** for the principal methods:
at
put
clipAt, wrapAt etc.

Literal Arrays can be created at compile time, and are very efficient. See **[Literals]** for information.

Class Methods

*with(... args)

Create a new Array whose slots are filled with the given arguments.
This is the same as the method in ArrayedCollection, but is reimplemented here to be more efficient.

```
Array.with(7, 'eight',  9).postln;
```

Instance Methods

reverse

Returns a new Array whose elements are reversed. The receiver is unchanged.

```
[1, 2, 3].reverse.postln;

(
x = [1, 2, 3];
z = x.reverse;
x.postln;
z.postln;
)
```

scramble

Returns a new Array whose elements have been scrambled. The receiver is unchanged.

```
[1, 2, 3, 4, 5, 6].scramble.postln;
```

mirror

Return a new Array which is the receiver made into a palindrome.
The receiver is unchanged.

```
[1, 2, 3, 4].mirror.postln;
```

mirror1

Return a new Array which is the receiver made into a palindrome with the last element removed.
This is useful if the list will be repeated cyclically, the first element will not get played twice.
The receiver is unchanged.

```
[1, 2, 3, 4].mirror1.postln;
```

mirror2

Return a new Array which is the receiver concatenated with a reversal of itself.
The center element is duplicated. The receiver is unchanged.

```
[1, 2, 3, 4].mirror2.postln;
```

stutter(n)

Return a new Array whose elements are repeated n times. The receiver is unchanged.

```
[1, 2, 3].stutter(2).postln;
```

rotate(n)

Return a new Array whose elements are in rotated order. Negative n values rotate left, postive n values
rotate right. The receiver is unchanged.

```
[1, 2, 3, 4, 5].rotate(1).postln;
```

```
[1, 2, 3, 4, 5].rotate(-1).postln;

[1, 2, 3, 4, 5].rotate(3).postln;
```

pyramid

Return a new Array whose elements have been reordered via one of 10 "counting" algorithms.
The algorithms are numbered 1 through 10. Run the examples to see the algorithms.

```
[1, 2, 3, 4].pyramid(1).postln;

(
10.do({ arg i;
[1, 2, 3, 4].pyramid(i + 1).asCompileString.postln;
});
)
```

lace(length)

Returns a new Array whose elements are interlaced sequences of the elements of the receiver's subcollections, up to size **length**. The receiver is unchanged.

```
(
x = [ [1, 2, 3], 6, List["foo", 'bar']];
y = x.lace(12);
x.postln;
y.postln;
)
```

permute(nthPermutation)

Returns a new Array whose elements are the **nthPermutation** of the elements of the receiver. The receiver is unchanged.

```
(
x = [ 1, 2, 3];
6.do({| i| x.permute(i).postln;});
)
```

wrapExtend(length)

Returns a new Array whose elements are repeated sequences of the receiver, up to size **length**. The receiver is unchanged.

```
(
x = [ 1, 2, 3, "foo", 'bar' ];
y = x.wrapExtend(9);
x.postln;
y.postln;
)
```

foldExtend(length)

Same as **lace** but the sequences fold back on the list elements.

```
(
x = [ 1, 2, "foo"];
y = x.foldExtend(9);
x.postln;
y.postln;
)
```

slide(windowLength, stepSize)

Return a new Array whose elements are repeated subsequences from the receiver. Easier to demonstrate than explain.

```
[1, 2, 3, 4, 5, 6].slide(3, 1).asCompileString.postln;

[1, 2, 3, 4, 5, 6].slide(3, 2).asCompileString.postln;

[1, 2, 3, 4, 5, 6].slide(4, 1).asCompileString.postln;
```

containsSeqColl

Returns true if the receiver Array contains any instance of SequenceableCollection

```
[1, 2, 3, 4].containsSeqColl.postln

[1, 2, [3], 4].containsSeqColl.postln
```

multiChannelExpand

Used by UGens to perform multi channel expansion.

source

Some UGens return Arrays of OutputProxy when instantiated. This method allows you to
get at the source UGen.

```
(
z = Pan2.ar;
z.postln;
z.source.postln;
)
```

fork(join, clock, quant, stackSize)

Used within Routines and assumes an array of functions, from which subroutines are created. The subroutines are played while the outer Routine carries on. The **join** parameter expresses after how many subroutines complete the outer Routine is allowed to go on. By default this happens after all subroutines have completed.

```
// an array of routine functions:
(
a = [
{ 1.wait; \done_one.postln },
{ 0.5.wait; \done_two.postln },
{ 0.2.wait; \done_three.postln }
];
)

// join after 0
```

```
(
Routine
 "join = 0."
a.fork(0); \doneAll.postln;
}.play;
)
// join after 1
(
Routine
 "join = 1."
a.fork(1); \doneAll.postln;
}.play;
)
// join after all
(
Routine
 "join = a.size (default)."
a.fork; \doneAll.postln;
}.play;
)
```

atIdentityHash(argKey)

This method is used by IdentitySet to search for a key among its members.

atIdentityHashInPairs(argKey)

This method is used by IdentityDictionary to search for a key among its members.

asShortString

Returns a short string representing the Array that only shows how many elements it contains

asString

Returns a string representing the Array. May not be compileable due to ellision (...) of excessive arguments.

asCompileString

Returns a string that will compile to return an Array equal to the receiver.

isValidUGenInput

Returns true. Arrays are valid UGen inputs.

ID: 42

ArrayedCollection

Superclass: SequenceableCollection

ArrayedCollection is an abstract class, a subclass of SequenceableCollections whose elements are held in a vector of slots. Instances of ArrayedCollection have a fixed maximum size beyond which they may not grow.

Its principal subclasses are Array (for holding objects), and RawArray, from which Int8Array, FloatArray,Signal etc. inherit.

Class Methods

*with(... args)

Create a new ArrayedCollection whose slots are filled with the given arguments.

```
Array.with(7, 'eight',  9).postln;
```

*series(size, start, step)

Fill an ArrayedCollection with an arithmetic series.

```
Array.series(5, 10, 2).postln;
```

*geom(size, start, grow)

Fill an ArrayedCollection with a geometric series.

```
Array.geom(5, 1, 3).postln;
```

Instance Methods

at(index)

Return the **item** at **index**.

clipAt(index)

Same as **at**, but values for **index** greater than the size of the ArrayedCollection will be clipped to the last index.

```
y = [ 1, 2, 3 ];
y.clipAt(13).postln;
```

wrapAt(index)

Same as **at**, but values for **index** greater than the size of the ArrayedCollection will be wrapped around to 0.

```
y = [ 1, 2, 3 ];
                    // this returns the value at index 0
                    // this returns the value at index 1
```

foldAt(index)

Same as **at**, but values for **index** greater than the size of the ArrayedCollection will be folded back.

```
y = [ 1, 2, 3 ];
                    // this returns the value at index 1
                    // this returns the value at index 0
                    // this returns the value at index 1
```

swap(i, j)

Swap the values at indices i and j.

```
[ 1, 2, 3 ].swap(0, 2).postln;
```

put(index, item)

Put **item** at **index**, replacing what is there.

clipPut(index, item)

Same as **put**, but values for **index** greater than the size of the ArrayedCollection will be clipped to the last index.

wrapPut(index, item)

Same as **put**, but values for **index** greater than the size of the ArrayedCollection will be wrapped around to 0.

foldPut(index)

Same as **put**, but values for **index** greater than the size of the ArrayedCollection will be folded back.

removeAt(index)

Remove and return the element at **index**, shrinking the size of the ArrayedCollection.

```
y = [ 1, 2, 3 ];
y.removeAt(1);
y.postln;
```

takeAt(index)

Same as **removeAt**, but reverses the order of the items following those that which was taken.

```
y = [ 1, 2, 3, 4 ];
y.takeAt(1);
y.postln;
```

add(item)

Adds an item to an ArrayedCollection if there is space. If there is not any space left in the object then this method returns a new ArrayedCollection. For this reason, you should always assign the result of add to a variable - never depend on add changing the receiver.

```
(
// z and y are the same object
```

```
var y, z;
z = [1, 2, 3];
y = z.add(4);
z.postln;
y.postln;
)

(
// in this case a new object is returned
var y, z;
z = [1, 2, 3, 4];
y = z.add(5);
z.postln;
y.postln;
)
```

addAll(aCollection)

Adds all the elements of aCollection to the contents of the receiver, possibly returning a new collection.

```
(
// in this case a new object is returned
var y, z;
z = [1, 2, 3, 4];
y = z.addAll([7, 8, 9]);
z.postln;
y.postln;
)
```

fill(value)

Inserts the item into the contents of the receiver, possibly returning a new collection. Note the difference between this and Collection's *fill.

```
(
var z;
z = List[1, 2, 3, 4];
z.fill(4).postln;
z.fill([1,2,3,4]).postln;
```

```
)
```

insert(index, item)

Inserts the item into the contents of the receiver, possibly returning a new collection.

```
(
// in this case a new object is returned
var y, z;
z = [1, 2, 3, 4];
y = z.insert(1, 999);
z.postln;
y.postln;
)
```

addFirst(item)

Inserts the item before the contents of the receiver, possibly returning a new collection.

```
(
// in this case a new object is returned
var y, z;
z = [1, 2, 3, 4];
y = z.addFirst(999);
z.postln;
y.postln;
)
```

pop

Remove and return the last element of the ArrayedCollection.

```
(
var z;
z = [1, 2, 3, 4];
z.pop.postln;
z.postln;
)
```

grow(sizeIncrease)

Increase the size of the ArrayedCollection by **sizeIncrease** number of slots, possibly returning a new collection.

copyRange(start, end)

Return a new ArrayedCollection which is a copy of the indexed slots of the receiver from start to end.

```
(
var y, z;
z = [1, 2, 3, 4, 5];
y = z.copyRange(1,3);
z.postln;
y.postln;
)
```

copySeries(first, second, last)

Return a new ArrayedCollection consisting of the values starting at **first**, then every step of the distance between **first** and **second**, up until **last**.

```
(
var y, z;
z = [1, 2, 3, 4, 5, 6];
y = z.copySeries(0, 2, 5);
y.postln;
)
```

putSeries(first, second, last, value)

Put **value** at every index starting at **first**, then every step of the distance between **first** and **second**, up until **last**.

```
(
var y, z;
z = [1, 2, 3, 4, 5, 6];
y = z.putSeries(0, 2, 5, "foo");
y.postln;
)
```

++ aCollection

Concatenate the contents of the two collections into a new ArrayedCollection.

```
(
var y, z;
z = [1, 2, 3, 4];
y = z ++ [7, 8, 9];
z.postln;
y.postln;
)
```

reverse

Return a new ArrayedCollection whose elements are reversed.

```
(
var y, z;
z = [1, 2, 3, 4];
y = z.reverse;
z.postln;
y.postln;
)
```

do(function)

Iterate over the elements in order, calling the function for each element. The function is passed two arguments, the element and an index.

```
['a', 'b', 'c'].do({ arg item, i; [i, item].postln; });
```

reverseDo(function)

Iterate over the elements in reverse order, calling the function for each element. The function is passed two arguments, the element and an index.

```
['a', 'b', 'c'].reverseDo({ arg item, i; [i, item].postln; });
```

windex

Interprets the array as a list of probabilities which should sum to 1.0 and returns a random index value based on those probabilities.

```
(
Array.fill(10, {
[0.1, 0.6, 0.3].windex;
}).postln;
)
```

normalizeSum

Returns the Array resulting from :

```
(this / this.sum)
```

so that the array will sum to 1.0.

This is useful for using with windex or wchoose.

```
[1, 2, 3].normalizeSum.postln;
```

performInPlace(selector, from, to, argList)

performs a method in place, within a certain region [from..to], returning the same array.

```
a = (0..10);
a.performInPlace(\normalizeSum, 3, 6);
```

ID: 43

Association

superclass: Magnitude

Associates a key with a value.
Associations can be created via the -> operator which is defined in class Object.

```
(
x = 'name' -> 100;
x.postln;
)
```

Accessing

<>key

the key object.

<>value

the value object.

Creation

*new(key, value)

Create an Association between two objects.
key - any object.
value - any object.

Testing

== anAssociation

Compare the keys of two Associations.

< anAssociation

Compare the keys of two Associations.

hash

Compute the hash value of the Association.

Streams

printOn(stream)

Write a string representation to the stream.

storeOn(stream)

Write a compileable string representation to the stream.

ID: 44

Bag

Superclass: Collection

A Bag is an unordered collection of objects. In some languages it is referred to as a counted set. A Bag keeps track of the number of times objects are inserted and requires that objects be removed the same number of times. Thus, there is only one instance of an object in a Bag even if the object has been added to the Bag multiple times.

Most of Bag's methods are inherited from Collection.
The contents of a Bag are unordered. You must not depend on the order of items in a set.

Adding and Removing:

add(anObject)

Add anObject to the Bag. A Bag may contain multiple entries of the same object.

```
Bag[1, 2, 3].add(4).postln;

Bag[1, 2, 3].add(3).postln;

Bag["abc", "def", "ghi"].add("jkl").postln;

Bag["abc", "def", "ghi"].add("def").postln;
```

remove(anObject)

Remove anObject from the Bag.

```
Bag[1, 2, 3].remove(3).postln;
```

Iteration:

do(function)

Evaluates function for each item in the Bag.
The function is passed two arguments, the item and an integer index.

```
Bag[1, 2, 3, 300].do({ arg item, i; item.postln });
```

ID: 45

Collection

superclass: Object

Collections are groups of objects. Collection is an abstract class. You do not create direct instances of Collection. There are many types of Collections including List, Array, Dictionary, Bag, Set, SortedList, etc. See the **Collections** overview for a complete listing of all subclasses.

Class Methods:

*fill(size, function)

Creates a Collection of the given size, the elements of which are determined by evaluation the given function. The function is passed the index as an argument.

```
Array.fill(4, {arg i; i * 2}).postln;
```

Accessing:

size

Answers the number of objects contained in the Collection.

```
List[1, 2, 3, 4].size.postln;
```

isEmpty

Answer whether the receiver contains no objects.

```
List[].isEmpty.postln;
```

Adding and Removing:

add(anObject)

Add anObject to the receiver.

```
List[1, 2].add(3).postln;
```

addAll(aCollection)

Add all items in aCollection to the receiver.

```
List[1, 2].addAll(List[3, 4]).postln;
```

remove(anObject)

Remove anObject from the receiver. Answers the removed object.

```
(
var a;
a = List[1, 2, 3, 4];
a.remove(3).postln;
a.postln;
)
```

removeAll(aCollection)

Remove all items in aCollection from the receiver.

```
List[1, 2, 3, 4].removeAll(List[2, 3]).postln;
```

removeAllSuchThat(function)

Remove all items in the receiver for which function answers true. The function is passed two arguments, the item and an integer index. Answers the objects which have been removed.

```
(
var a;
a = List[1, 2, 3, 4];
a.removeAllSuchThat({ arg item, i; item < 3 }).postln;
a.postln;
)
```

Testing:

includes(anObject)

Answer whether anObject is contained in the receiver.

```
List[1, 2, 3, 4].includes(3).postln;
```

includesAny(aCollection)

Answer whether any item in aCollection is contained in the receiver.

```
List[1, 2, 3, 4].includesAny(List[4, 5]).postln;
```

includesAll(aCollection)

Answer whether all items in aCollection are contained in the receiver.

```
List[1, 2, 3, 4].includesAll(List[4, 5]).postln;
```

Iteration:

do(function)

Evaluates function for each item in the collection. The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].do({ arg item, i; item.postln });
```

collect(function)

Answer a new collection which consists of the results of function evaluated for each item in the collection. The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].collect({ arg item, i; item + 10 }).postln;
```

select(function)

Answer a new collection which consists of all items in the receiver for which function answers true. The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].select({ arg item, i; item.even }).postln;
```

reject(function)

Answer a new collection which consists of all items in the receiver for which function answers false. The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].reject({ arg item, i; item.even }).postln;
```

detect(function)

Answer the first item in the receiver for which function answers true.
The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].detect({ arg item, i; item.even }).postln;
```

inject(initialValue, function)

In functional programming, the operation known as a fold.
inject takes an initial value and a function and combines the elements of the collection by applying the function to the accumulated value and an element from the collection. The function takes two arguments and returns the new value. The accumulated value is initialzed to initialValue.

```
[1,2,3,4,5].inject(0, _+_);
```

```
15
```

```
[1,2,3,4,5].inject(1, _*_);
```

```
120
```

```
[1,2,3,4,5].inject([], {| a,b| a ++ b.squared }); // same as .collect(_.squared)
```

```
[ 1, 4, 9, 16, 25 ]
```

```
[1,2,3,4,5].inject([], {| a,b| [b] ++ a ++ [b]});
```

```
[ 5, 4, 3, 2, 1, 1, 2, 3, 4, 5 ]
```

```
[1,2,3,4,5].inject([], {| a,b| a ++ b ++ a});
```

```
[ 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1 ]
```

```
[1,2,3,4,5].inject([], {| a,b| a ++ a ++ b});
```

```
[ 1, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 1, 2, 3, 4, 1, 1, 2, 1, 1, 2, 3, 1, 1, 2, 1, 1, 2, 3, 4, 5 ]
```

any(function)

Answer whether function answers true for any item in the receiver.
The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].any({ arg item, i; item.even }).postln;
```

every(function)

Answer whether function answers true for every item in the receiver.
The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].every({ arg item, i; item.even }).postln;
```

count(function)

Answer the number of items for which function answers true.
The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].count({ arg item, i; item.even }).postln;
```

occurencesOf(anObject)

Answer the number of items in the receiver which are equal to anObject.

```
List[1, 2, 3, 3, 4, 3, 4, 3].occurencesOf(3).postln;
```

sum(function)

Answer the sum of the results of function evaluated for each item in the receiver.
The function is passed two arguments, the item and an integer index.

```
List[1, 2, 3, 4].sum.postln;

(0..8).sum { | i| 1 / (2 ** i) };
```

maxItem(function)

Answer the maximum of the results of function evaluated for each item in the receiver.
The function is passed two arguments, the item and an integer index.
If function is nil, then answer the maximum of all items in the receiver.

```
List[1, 2, 3, 4].maxItem({ arg item, i; item + 10 }).postln;
```

minItem(function)

Answer the minimum of the results of function evaluated for each item in the receiver.
The function is passed two arguments, the item and an integer index.
If function is nil, then answer the minimum of all items in the receiver.

```
List[1, 2, 3, 4].minItem({ arg item, i; item + 10 }).postln;
```

Conversion:

asBag

Answer a Bag to which all items in the receiver have been added.

```
List[1, 2, 3, 4].asBag.postln;
```

asList

Answer a List to which all items in the receiver have been added.

```
List[1, 2, 3, 4].asList.postln;
```

asSet

Answer a Set to which all items in the receiver have been added.

```
List[1, 2, 3, 4].asList.postln;
```

asSortedList

Answer a SortedList to which all items in the receiver have been added.

```
List[2, 1, 4, 3].asSortedList.postln;
```

printOn(stream)

Print a representation of the collection to a stream.

storeOn(stream)

Write a compileable representation of the collection to a stream.

printItemsOn(stream)

Print a comma separated compileable representation of the items in the collection to a stream.

storeItemsOn(stream)

Write a comma separated compileable representation of the items in the collection to a stream.

ID: 46

Collections

SuperCollider has a rich hierarchy of Collection subclasses. Collection's class subtree is detailed below. Subclasses of a given class are indented and enclosed in (possibly nested) square brackets. Most of these subclasses have their own helpfiles. Classes labelled abstract are not for direct use, but classes lower down the tree may inherit methods from them. For this reason it is important to consult the helpfiles of classes farther up the tree in order to get a complete list of available methods.

Collection abstract superclass of all Collection subclasses
many methods are inherited from this class
[
Array2D a two dimensional array

Range ranges of values

Interval ranges of Integers with a fixed Interval between them

MultiLevelIdentityDictionary a tree of IdentityDictionaries

[Library] a unique global MultiLevelIdentityDictionary

Set an unordered collection of unequal objects
[
Dictionary an unordered associative collection mapping
keys to values
[
IdentityDictionary a Dictionary wherein keys match only if identical
(rather than if simply equal)
[
Environment an IdentityDictionary, one of which is always current;
useful for creating sets of persistent variables

[Event] a Dictionary mapping names of musical parameters
to their values

NameDictionary an IdentityDictionary for adding named objects

(objects with a .name method) such that
name -> namedObject
]
]
IdentitySet an unordered collection of unidentical objects
(compare to Set)
]
Bag an unordered collection of objects

Pair Lisp-like two element cells

TwoWayIdentityDictionary an IdentityDictionary which allows easy searching by both key and value; faster than IdentityDictionary on
reverse lookup, but with more memory overhead

[ObjectTable] associates Integer ids with objects

SequenceableCollection abstract superclass of collections whose objects can be indexed by integer
[
Order SequenceableCollection with an indices instance
variable

LinkedList a doubly linked list

List an expandable SequenceableCollection
(compare to ArrayedCollection and Array)

[SortedList] a List whose items are kept in a sorted order

ArrayedCollection abstract superclass of Collections of fixed maximum size whose elements are held in a vector of slots
[
RawArray abstract superclass of array classes that hold
raw data values
[
DoubleArray a RawArray of double precision floats

FloatArray a RawArray of floats
[

Wavetable a special format FloatArray

Signal a FloatArray that represents a sampled function of time buffer
]
String an array of characters

SymbolArray a RawArray of symbols

Int32Array a RawArray of 32 bit Integers

Int16Array a RawArray of 16 bit Integers

Int8Array a RawArray of 8 bit Integers
]
Array an ArrayedCollection whose slots may contain any object; more efficient than List
]
]
]

ID: 47

Dictionary

Superclass: Set

A Dictionary is an associative collection mapping keys to values.
Two keys match if they are equal. i.e. == returns true.

The contents of a Dictionary are unordered.
You must not depend on the order of items in a Dictionary.

Creation:

*new(n)

Creates a Dictionary with an initial capacity for n key value mappings.

Adding and Removing:

add(anAssociation)

Add anAssociation to the Dictionary.
If the key value pair already exists in the Dictionary, the key's value will be replaced.

```
(
    Dictionary
d.add('monkey' -> 0).postln;
// Add robot as a key with a value of 1
d.add('robot' -> 1).postln;
// Replaces the value for the key monkey with 2
d.add('monkey' -> 2).postln;
)
```

put(key, obj)

Associate two objects and add them to the Dictionary.
key - key to associate with object
obj - an object

```
d.put("abc", 10).postln;
```

removeAt(key)

Remove the key and the value associated with it from the Dictionary.

```
d.removeAt('monkey').postln;
```

Accessing:

at(key)

Access the value associated with the key.

```
// Get the value associated with key
d.at('robot');
// Key doesn't exist
     'monkey'
```

matchAt(item)

The dictionary's keys are used as conditions against which the arbitrary item is matched.

Note: if an item matches multiple criteria, the value returned is arbitrary. This is because a dictionary is an unordered collection. It's the user's responsibility to make sure that criteria are mutually exclusive.

If the key is an object, the item will be matched by identity (if key === item, the value will be returned).
If the key is a collection, the item is matched if it's contained in the collection.
If the key is a function, the function is evaluated with the item as an argument and the item is matched if the function returns true.

```
    Dictionary
0 -> 'zero',
 'abc'   'alpha'
[1, 2, 3, 5, 8, 13, 21] -> 'fibonacci',
{ | x|  x.even } -> 'even'
];
```

```
d.matchAt(0)
d.matchAt(1)
              // matches both 'fibonacci' and 'even', but 'fibonacci' is returned
d.matchAt(4)
d.matchAt('abc')
```

ID: 48

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 49

Environment

superclass: IdentityDictionary

An Environment is an IdentityDictionary mapping Symbols to values. There is always one current Environment which is stored in the currentEnvironment class variable of class Object.

Symbol and value pairs may be put into the current Environment as follows:

```
currentEnvironment.put(\myvariable, 999);
```

and retrieved from the current Environment as follows:

```
currentEnvironment.at(\myvariable).postln;
```

The compiler provides a shorthand for the two constructs above .

```
myvariable = 888;
```

is equivalent to:

```
currentEnvironment.put(\myvariable, 888);
```

and:

```
myvariable.postln;
```

is equivalent to:

```
currentEnvironment.at(\myvariable).postln;
```

Making an Environment

Environment has a class method **make** which can be used to create an Environment and fill it with values. What **make** does is temporarily replace the current Environment with a new one, call your function where you fill the Environment with values, then it replaces the previous current Environment and returns you the new one.

```
(
var a;
a = Environment.make({
a = 100;
b = 200;
c = 300;
});
a.postln;
)
```

Using an Environment

The instance method **use** lets you temporarily replace the current Environment with one you have made. The **use** method returns the result of your function instead of the Environment like **make** does.

```
(
var a;
a = Environment.make({
a = 10;
b = 200;
c = 3000;
});
a.use({
a +  b +  c
}).postln;
)
```

There is also a **use** class method for when you want to make and use the result from an Environment directly.

```
(
var a;
    Environment
a = 10;
b = 200;
c = 3000;
a +  b +  c
}).postln;
```

```
)
```

Calling Functions with arguments from the current Environment

It is possible to call a Function and have it look up any unspecified argument values from the current Environment. This is done with the **valueEnvir** and **valueArrayEnvir** methods. These methods will, for any unspecified argument value, look in the current Environment for a symbol with the same name as the argument. If the argument is not found then whatever the function defines as the default value for that argument is used.

```
(
var f;

// define a function
f = { arg x, y, z; [x, y, z].postln; };

Environment
x = 7;
y = 8;
z = 9;

f.valueEnvir(1, 2, 3); // all values supplied
 f.valueEnvir(1, 2); // z is looked up in the current Environment
 f.valueEnvir(1);  // y and z are looked up in the current Environment
 f.valueEnvir;   // all arguments are looked up in the current Environment
 f.valueEnvir(z:  1); // x and y are looked up in the current Environment
});
)
```

Now here is how this can be used with an instrument function. Environments allow you to define instruments without having to worry about argument ordering conflicts. Even though the three functions below have the freq, amp and pan args declared in different orders it does not matter, because valueEnvir looks them up in the environment.

```
s.boot;

(
var a, b, c, orc;
```

```
a = { arg freq, amp, pan;
Pan2.ar(SinOsc.ar(freq), pan, amp);
};
b =  { arg amp, pan, freq;
Pan2.ar(RLPF.ar(Saw.ar(freq), freq * 6, 0.1), pan, amp);
};
c =  { arg pan, freq, amp;
Pan2.ar(Resonz.ar(GrayNoise.ar, freq * 2, 0.1), pan, amp * 2);
};
orc = [a, b, c];
// 'reverb'
{ var in; in = In.ar(0, 2); CombN.ar(in, 0.2, 0.2, 3, 1, in); }.play(addAction:  \addToTail);

{ loop({
 Environment
   // set values in the environment
freq = exprand(80, 600);
amp = 0.1;
pan = 1.0.rand2;

   // call a randomly chosen instrument function
   // with values from the environment

x = { orc.choose.valueEnvir; }.play(fadeTime:  0.2, addAction:  \addToHead);
0.2.wait;
x.release(0.2);
});
}) }.fork;

)
```

ID: 50

Event

superclass: Environment

Events are dictionaries matching Symbols representing names of parameters for a musical event to their values.

For more information on Events see:
[Streams-Patterns-Events4] and **[Streams-Patterns-Events5]**

ID: 51

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 52

IdentityDictionary

Superclass: Dictionary

An IdentityDictionary is an associative collection mapping keys to values.
Two keys match only if they are identical.

The contents of an IdentityDictionary are unordered.
You must not depend on the order of items in a IdentityDictionary.

Often IdentityDictionaries are used with Symbols as the keys since
Symbols are guaranteed to be identical if they have the same character representation
(i.e. they are equal). Two equal Strings on the other hand might not be identical.

ID: 53

IdentitySet

Superclass: Set

An IdentitySet is collection of objects, no two of which are the same object (aka. "identical").
Most of its methods are inherited. Look in the Collection class for the most of the relevant methods.
The contents of an IdentitySet are unordered.
You must not depend on the order of items in an IdentitySet.

IdentitySets are faster than Sets because testing for identity is much faster than testing for
equality. Different classes may implement equality in different ways, but identity can be determined
just by comparing the object addresses. This allows some methods of IdentitySet to be implemented
by fast primitives.

Adding and Removing:

add(anObject)

Add anObject to the IdentitySet.
An object which is equal to an object already in the IdentitySet will not be added.

```
IdentitySet[1, 2, 3].add(4).postln;

IdentitySet[1, 2, 3].add(3).postln;

// the two strings are equal but not identical
IdentitySet["abc", "def", "ghi"].add("def").postln;

// symbols are guaranteed to be identical if they are equal
IdentitySet['abc', 'def', 'ghi'].add('def').postln;

IdentitySet['abc', 'def', 'ghi'].add('jkl').postln;
```

remove(anObject)

Remove anObject from the IdentitySet.

```
IdentitySet[1, 2, 3].remove(3).postln;
```

Iteration:

do(function)

Evaluates function for each item in the IdentitySet.
The function is passed two arguments, the item and an integer index.

```
IdentitySet[1, 2, 3, 300].do({ arg item, i; item.postln });
```

ID: 54

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 55

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 56

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 57

Interval

superclass: Collection

An Interval is a range of integers from a starting value to an ending value by some step value.

Creation

*new(start, end, step)

Create a new Interval.

```
Interval.new(10, 30, 4).postln;
```

Instance Variables

<>start

The starting value of the interval.

<>end

The ending value of the interval.

<>step

The step value of the interval.

Instance Methods

size

Return the number of items in the interval.

```
Interval.new(10, 30, 4).size.postln;
```

at(index)

Return the indexed item in the interval.

```
Interval.new(10, 30, 4).at(3).postln;
```

do(function)

Evaluates function for each item in the interval.
The function is passed two arguments, the item and an integer index.

```
Interval.new(10, 30, 4).do({ arg item, i; item.postln });
```

ID: 58

Library

superclass: MultiLevelIdentityDictionary

This is a single global MultiLevelIdentityDictionary.
There is only one of them ever.

The last argument to put is the object being inserted:

```
Library     \multi \level \addressing \system "i'm the thing you put in here"

Library    \multi \level \addressing \system

Library         \multi \level \addressing \system
```

postTree
post a formatted description of the entire library

```
Library.postTree;
```

ID: 59

LinkedList

Superclass: SequenceableCollection

LinkedList implements a doubly linked list.

Instance Methods

Most methods are inherited from the superclasses.

addFirst(obj)

Add an item to the head of the list.

add(obj)

Add an item to the tail of the list.

remove(obj)

Remove an item from the list.

pop

Remove and return the last item in the list.

popFirst

Remove and return the first item in the list.

first

Return the first item in the list.

last

Return the last item in the list.

at(index)

Return the item at the given index in the list.
This requires a scan of the list and so is O(n).

put(index, obj)

Put the item at the given index in the list.
This requires a scan of the list and so is O(n).

removeAt(index)

Remove and return the item at the given index in the list.
This requires a scan of the list and so is O(n).

ID: 60

LinkedListNode

Superclass: Object

LinkedListNode is used to implement the internal nodes of the LinkedList class. You should not need to deal with a LinkedListNode directly.

ID: 61

List

Superclass: SequenceableCollection

List is a subclass of **SequenceableCollection** with unlimited growth in size. Although not a subclass of **Array** or its superclass **ArrayedCollection** it uses an Array in its implementation and is in many cases interchangeable with one. (List implements many of the same methods.)

Arrays have a fixed maximum size. If you add beyond that size a new Array is created and returned, but you must use an assignment statement or the new array will be lost. (See the **Array** helpfile.) List has no size limitation and is thus more flexible, but has slightly more overhead.

```
(
x = Array.new(3);
y = List.new(3);
5.do({arg i; z = x.add(i); y.add(i);});
x.postln; z.postln; y.postln;
)
```

Many of List's methods are inherited from **SequenceableCollection** or **Collection** and are documented in those helpfiles.

Creation

*new(size)

Creates a List with the initial capacity given by **size**.

*newClear(size)

Creates a List with the initial capacity given by **size** and slots filled with nil.

*copyInstance(aList)

Creates a List by copying **aList's** array variable.

*newUsing(anArray)

Creates a List using **anArray**.

Instance Methods

asArray

Returns a new **Array** based upon this List.

array

Returns the List's Array, allowing it to be manipulated directly. This should only be necessary for exotic manipulations not implemented in List or its superclasses.

```
(
x = List[1, 2, 3];
x.array.add("foo");
x.postln;
)
```

array_(anArray)

Sets the List's Array.

at(index)

Return the **item** at **index**.

```
List[ 1, 2, 3 ].at(0).postln;
```

clipAt(index)

Same as **at**, but values for **index** greater than the size of the List will be clipped to the last index.

```
y = List[ 1, 2, 3 ];
y.clipAt(13).postln;
```

wrapAt(index)

Same as **at**, but values for **index** greater than the size of the List will be wrapped around to 0.

```
y = List[ 1, 2, 3 ];
                    // this returns the value at index 0
                    // this returns the value at index 1
```

foldAt(index)

Same as **at**, but values for **index** greater than the size of the List will be folded back.

```
y = List[ 1, 2, 3 ];
                    // this returns the value at index 1
                    // this returns the value at index 0
                    // this returns the value at index 1
```

put(index, item)

Put **item** at **index**, replacing what is there.

clipPut(index, item)

Same as **put**, but values for **index** greater than the size of the List will be clipped to the last index.

wrapPut(index, item)

Same as **put**, but values for **index** greater than the size of the List will be wrapped around to 0.

foldPut(index)

Same as **put**, but values for **index** greater than the size of the List will be folded back.

add(item)

Adds an **item** to the end of the List.

addFirst(item)

Inserts the **item** at the beginning of the List.

insert(index, item)

Inserts the **item** into the contents of the List at the indicated **index**.

pop

Remove and return the last element of the List.

grow(sizeIncrease)

Increase the size of the List by **sizeIncrease** number of slots.

removeAt(index)

Remove and return the element at **index**, shrinking the size of the List.

```
y = List[ 1, 2, 3 ];
y.removeAt(1);
y.postln;
```

fill(value)

Inserts the item into the contents of the receiver, possibly returning a new collection. Note the difference between this and Collection's *fill.

```
(
var z;
z = List[1, 2, 3, 4];
z.fill(4).postln;
z.fill([1,2,3,4]).postln;
)
```

do(function)

Iterate over the elements in order, calling the function for each element. The function is passed two arguments, the element and an index.

```
List['a', 'b', 'c'].do({ arg item, i; [i, item].postln; });
```

reverseDo(function)

Iterate over the elements in reverse order, calling the function for each element. The function is passed two arguments, the element and an index.

```
List['a', 'b', 'c'].reverseDo({ arg item, i; [i, item].postln; });
```

copyRange(start, end)

Return a new List which is a copy of the indexed slots of the receiver from start to end.

```
(
var y, z;
z = List[1, 2, 3, 4, 5];
y = z.copyRange(1,3);
z.postln;
y.postln;
)
```

copySeries(first, second, last)

Return a new List consisting of the values starting at **first**, then every step of the distance between **first** and **second**, up until **last**.

```
(
var y, z;
z = List[1, 2, 3, 4, 5, 6];
y = z.copySeries(0, 2, 5);
y.postln;
)
```

putSeries(first, second, last, value)

Put **value** at every index starting at **first**, then every step of the distance between **first** and **second**, up until **last**.

```
(
```

```
var y, z;
z = List[1, 2, 3, 4, 5, 6];
y = z.putSeries(0, 2, 5, "foo");
y.postln;
)
```

reverse

Return a new List whose elements are reversed.

```
(
var y, z;
z = List[1, 2, 3, 4];
y = z.reverse;
z.postln;
y.postln;
)
```

scramble

Returns a new List whose elements have been scrambled. The receiver is unchanged.

```
List[1, 2, 3, 4, 5, 6].scramble.postln;
```

mirror

Return a new List which is the receiver made into a palindrome.
The receiver is unchanged.

```
List[1, 2, 3, 4].mirror.postln;
```

mirror1

Return a new List which is the receiver made into a palindrome with the last element removed.
This is useful if the list will be repeated cyclically, the first element will not get played twice.
The receiver is unchanged.

```
List[1, 2, 3, 4].mirror1.postln;
```

mirror2

Return a new List which is the receiver concatenated with a reversal of itself. The center element is duplicated. The receiver is unchanged.

```
List[1, 2, 3, 4].mirror2.postln;
```

stutter(n)

Return a new List whose elements are repeated n times. The receiver is unchanged.

```
List[1, 2, 3].stutter(2).postln;
```

rotate(n)

Return a new List whose elements are in rotated order. Negative n values rotate left, postive n values rotate right. The receiver is unchanged.

```
List[1, 2, 3, 4, 5].rotate(1).postln;

List[1, 2, 3, 4, 5].rotate(-1).postln;

List[1, 2, 3, 4, 5].rotate(3).postln;
```

pyramid

Return a new List whose elements have been reordered via one of 10 "counting" algorithms.
The algorithms are numbered 1 through 10. Run the examples to see the algorithms.

```
List[1, 2, 3, 4].pyramid(1).postln;

(
10.do({ arg i;
List[1, 2, 3, 4].pyramid(i + 1).asCompileString.postln;
});
)
```

lace(length)

Returns a new List whose elements are interlaced sequences of the elements of the receiver's subcollections, up to size **length**. The receiver is unchanged.

```
(
x = List[ [1, 2, 3], 6, List["foo", 'bar']];
y = x.lace(12);
x.postln;
y.postln;
)
```

permute(nthPermutation)

Returns a new List whose elements are the **nthPermutation** of the elements of the receiver. The receiver is unchanged.

```
(
x = List[ 1, 2, 3];
6.do({| i| x.permute(i).postln;});
)
```

wrapExtend(length)

Returns a new List whose elements are repeated sequences of the receiver, up to size **length**. The receiver is unchanged.

```
(
x = List[ 1, 2, 3, "foo", 'bar' ];
y = x.wrapExtend(9);
x.postln;
y.postln;
)
```

foldExtend(length)

Same as **lace** but the sequences fold back on the list elements.

```
(
x = List[ 1, 2, "foo"];
```

```
y = x.foldExtend(9);
x.postln;
y.postln;
)
```

slide(windowLength, stepSize)

Return a new List whose elements are repeated subsequences from the receiver. Easier to demonstrate than explain.

```
List[1, 2, 3, 4, 5, 6].slide(3, 1).asCompileString.postln;

List[1, 2, 3, 4, 5, 6].slide(3, 2).asCompileString.postln;

List[1, 2, 3, 4, 5, 6].slide(4, 1).asCompileString.postln;
```

dump

Dump the List's Array.

clear

Replace the List's Array with a new empty one.

ID: 62

```
"This text is the result of a postln command which was loaded and executed by loadPaths".postln;
```

ID: 63

MultiLevelIdentityDictionary

superclass: Collection

A tree of IdentityDictionaries. Addresses within the tree are specified with a series of keys. Library is its most useful subclass.

at(key1,key2 ... keyN)
retrieves a leaf node or nil if not found.

put(key1,key2 ... keyN, item)
puts the item as a leaf node, internally creating new branches as needed to accommodate the list of keys.

choose
choose a branch at each level, descend the tree until a leaf is chosen.

choose(key1,key2 ... keyN)
starting at an address within the tree, descend the tree until a leaf is chosen.

putTree(key1,[
key2a, item1-2a,
key2b, item1-2b,
[
key3, item1-3
] // etc...
]);

A way to insert objects into the tree with a syntax similar to the organization of the tree itself.

ID: 64

ObjectTable associate objects with IDs

superclass: TwoWayIdentityDictionary

An ObjectTable is used to associate an id with an object. This is useful
for enabling references to objects on remote systems via Open Sound Control.

*init

Create the main ObjectTable. This is called in Main::startUp.

*add(obj)

Put an object in the main ObjectTable and generate an Integer id.
obj - the object to put in the table.

add(obj)

Put an object in an ObjectTable and generate an Integer id.
obj - the object to put in the table.

*put(key, obj)

Put an object in the main ObjectTable under a specific key.
key - a Symbol.
obj - the object to put in the table.

*at(id)

Get an object in the main ObjectTable.
id - an Integer or Symbol.

*getID(obj)

Get the ID of an object in the table.
obj - an object in the table.

ID: 65

PriorityQueue

superclass: Object

PriorityQueue implements a priority queue data structure, which is used to build schedulers.
It allows you to put in items at some arbitrary time and pop them in time order.

Instance Methods:

put(time, item)

Puts the item in the queue at the given time.

topPriority

Returns the time of the earliest item in the queue.

pop

Returns the earliest item in the queue.

clear

Empty the queue.

isEmpty

Return a Boolean whether the queue is empty.

notEmpty

Return a Boolean whether the queue is not empty.

Example:

```
(
var p;
    PriorityQueue

p.put(0.1, \a);
p.put(2.0, \b);
p.put(0.5, \c);
p.put(0.2, \d);
p.put(1.0, \e);

while ({ p.notEmpty },{
[p.topPriority, p.pop].postln;
});


p.pop.postln;
p.pop.postln;
p.pop.postln;

)

[ 0.1, a ]
[ 0.2, d ]
[ 0.5, c ]
[ 1, e ]
[ 2, b ]
nil
nil
nil
```

ID: 66

RawArray

Superclass: ArrayedCollection

RawArray is the abstract superclass of a group of array classes that hold raw data values.

Class Methods

*read(path)

Reads a file into a subclass of RawArray and returns the array.

Instance Methods

write(path)

Writes the array as a file.

putFile(prompt, defaultName)

Opens a save file dialog to save the array to a file.

ID: 67

Int8Array, Int16Array, Int32Array, RGBArray, FloatArray, DoubleArray

Superclass: RawArray

These classes implement arrays whose indexed slots are all of the same type.

Int8Array - 8 bit integer
Int16Array - 16 bit integer
Int32Array - 32 bit integer
RGBArray - 32 bit color
FloatArray - 32 bit floating point
DoubleArray - 64 bit floating point

These classes implement only one method.

***readNew(file)**

Called by *read to read the entire file as an array of this class' type and return a new instance.

ID: 68

SequenceableCollection

Superclass: Collection

SequenceableCollection is a subclass of Collection whose elements can be indexed by an Integer. It has many useful subclasses; **Array** and **List** are amongst the most commonly used.

Class Methods

*series(size, start, step)

Fill a SequenceableCollection with an arithmetic series.

```
Array.series(5, 10, 2).postln;
```

*geom(size, start, grow)

Fill a SequenceableCollection with a geometric series.

```
Array.geom(5, 1, 3).postln;
```

*rand(size, minVal, maxVal)

Fill a SequenceableCollection with random values in the range minVal to maxVal.

```
Array.rand(8, 1, 100).postln;
```

*rand2(size, val)

Fill a SequenceableCollection with random values in the range -val to +val.

```
Array.rand2(8, 100).postln;
```

*linrand(size, minVal, maxVal)

Fill a SequenceableCollection with random values in the range minVal to maxVal with a linear

distribution.

```
Array.linrand(8, 1, 100).postln;
```

Instance Methods

first

Return the first element of the collection,

last

Return the first element of the collection,

indexOf(item)

Return the index of item in the collection, or nil if not found.

indexIn(val)

returns the closest index of the value in the collection (collection must be sorted)

```
[2, 3, 5, 6].indexIn(5.2)
```

indexInBetween(val)

returns a linearly interpolated float index for the value (collection must be sorted)
inverse operation is **blendAt**

```
x = [2, 3, 5, 6].indexInBetween(5.2)
[2, 3, 5, 6].blendAt(x)
```

blendAt(floatIndex)

returns a linearly interpolated value between the two closest indices
inverse operation is **indexInBetween**

```
x = [2, 5, 6].blendAt(0.4)
```

copyRange(start, end)

Return a new SequenceableCollection which is a copy of the indexed slots of the receiver from **start** to **end**.

```
(
var y, z;
z = [1, 2, 3, 4, 5];
y = z.copyRange(1,3);
z.postln;
y.postln;
)
```

copyToEnd(start)

Return a new SequenceableCollection which is a copy of the indexed slots of the receiver from **start** to the end of the collection.

copyFromStart(end)

Return a new SequenceableCollection which is a copy of the indexed slots of the receiver from the start of the collection to **end**.

remove(item)

Remove item from collection.

flat

Returns a collection from which all nesting has been flattened.

```
[[1, 2, 3],[[4, 5],[[6]]]].flat.postln;
```

flatten(numLevels)

Returns a collection from which numLevels of nesting has been flattened.

```
[[1, 2, 3],[[4, 5],[[6]]]].flatten(1).asCompileString.postln;
```

```
[[1, 2, 3],[[4, 5],[[6]]]].flatten(2).asCompileString.postln;
```

flop

Invert rows and columns in a two dimensional collection.

```
[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]].flop.asCompileString.postln;
```

choose

Choose an element from the collection at random.

```
[1, 2, 3, 4].choose.postln;
```

wchoose

Choose an element from the collection at random using a list of probabilities or weights. The weights must sum to 1.0.

```
[1, 2, 3, 4].wchoose([0.1, 0.2, 0.3, 0.4]).postln;
```

sort(function)

Sort the contents of the collection using the comparison function argument.
The function should take two elements as arguments and return true if the first argument should be sorted before the second argument.
If the function is nil, the following default function is used.

```
{ arg a, b; a < b }
```

```
[6, 2, 1, 7, 5].sort.postln;
```

```
[6, 2, 1, 7, 5].sort({ arg a, b; a > b }).postln; // reverse sort
```

swap(i, j)

Swap two elements in the collection at indices i and j.

doAdjacentPairs(function)

Calls function for every adjacent pair of elements in the SequentialCollection.
The function is passed the two adjacent elements and an index.

```
[1, 2, 3, 4, 5].doAdjacentPairs({ arg a, b; [a, b].postln; });
```

separate(function)

Separates the collection into sub-collections by calling the function for each adjacent pair of elements.
If the function returns true, then a separation is made between the elements.

```
[1, 2, 3, 5, 6, 8, 10].separate({ arg a, b; (b - a) > 1 }).asCompileString.postln;
```

clump(groupSize)

Separates the collection into sub-collections by separating every groupSize elements.

```
[1, 2, 3, 4, 5, 6, 7, 8].clump(3).asCompileString.postln;
```

clumps(groupSizeList)

Separates the collection into sub-collections by separating elements into groupings whose size
is given by integers in the groupSizeList.

```
[1, 2, 3, 4, 5, 6, 7, 8].clumps([1,2]).asCompileString.postln;
```

curdle(probability)

Separates the collection into sub-collections by separating elements according to the given probability.

```
[1, 2, 3, 4, 5, 6, 7, 8].curdle(0.3).asCompileString.postln;
```

Math Support

Unary Messages:

All of the following messages send the message performUnaryOp to the receiver with the unary message selector as an argument.

neg, reciprocal, bitNot, abs, asFloat, asInt, ceil, floor, frac, sign, squared, cubed, sqrt
exp, midicps, cpsmidi, midiratio, ratiomidi, ampdb, dbamp, octcps, cpsoct, log, log2,
log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, rand, rand2, linrand, bilinrand,
sum3rand, distort, softclip, nyqring, coin, even, odd, isPositive, isNegative,
isStrictlyPositive, real, imag, magnitude, magnitudeApx, phase, angle, rho, theta,
asFloat, asInteger

performUnaryOp(aSelector)

Creates a new collection of the results of applying the selector to all elements in the receiver.

```
[1, 2, 3, 4].neg.postln;
```

```
[1, 2, 3, 4].reciprocal.postln;
```

Binary Messages:

All of the following messages send the message performBinaryOp to the receiver with the
binary message selector and the second operand as arguments.

+, -, *, /, div, %, **, min, max, <, <=, >, >=, &, | , bitXor, lcm, gcd, round, trunc, atan2,
hypot, », +», fill, ring1, ring2, ring3, ring4, difsqr, sumsqr, sqrdif, absdif, amclip,
scaleneg, clip2, excess, <!, rrand, exprand

performBinaryOp(aSelector, theOperand)

Creates a new collection of the results of applying the selector with the operand to all elements
in the receiver.
If the operand is a collection then elements of that collection are paired with elements of
the receiver.

```
([1, 2, 3, 4] * 10).postln;

([1, 2, 3, 4] * [4, 5, 6, 7]).postln;
```

ID: 69

Set

Superclass: Collection

A Set is collection of objects, no two of which are equal.
Most of its methods are inherited from Collection.
The contents of a Set are unordered. You must not depend on the order of items in a set.

Adding and Removing:

add(anObject)

Add anObject to the Set. An object which is equal to an object already in the Set will not be added.

```
Set[1, 2, 3].add(4).postln;

Set[1, 2, 3].add(3).postln;

Set["abc", "def", "ghi"].add("jkl").postln;

Set["abc", "def", "ghi"].add("def").postln;
```

remove(anObject)

Remove anObject from the Set.

```
Set[1, 2, 3].remove(3).postln;
```

Iteration:

do(function)

Evaluates function for each item in the Set.
The function is passed two arguments, the item and an integer index.

```
Set[1, 2, 3, 300].do({ arg item, i; item.postln });
```

ID: 70

Signal sampled audio buffer

Superclass: FloatArray

A Signal is a FloatArray that represents a sampled function of time buffer.
Signals support math operations.

Creation

*sineFill(size, amplitudes, phases)

Fill a Signal of the given size with a sum of sines at the given amplitudes and phases.
The Signal will be normalized.
size - the number of samples in the Signal.
amplitudes - an Array of amplitudes for each harmonic beginning with the fundamental.
phases - an Array of phases in radians for each harmonic beginning with the fundamental.

```
Signal.sineFill(1000, 1.0/[1,2,3,4,5,6]).plot;
```

*chebyFill(size, amplitudes, phases)

Fill a Signal of the given size with a sum of Chebyshev polynomials at the given amplitudes
for use in waveshaping by the Shaper ugen.
The Signal will be normalized.
size - the number of samples in the Signal.
amplitudes - an Array of amplitudes for each Chebyshev polynomial beginning with order 1.

```
Signal.chebyFill(1000, [1]).plot;
```

```
Signal.chebyFill(1000, [0, 1]).plot;
```

```
Signal.chebyFill(1000, [0, 0, 1]).plot;

Signal.chebyFill(1000, [0.3, -0.8, 1.1]).plot;
```

*hanningWindow(size, pad)

Fill a Signal of the given size with a Hanning window.
size - the number of samples in the Signal.
pad - the number of samples of the size that is zero padding.

```
Signal.hanningWindow(1024).plot;

Signal.hanningWindow(1024, 512).plot;
```

*hammingWindow(size)

Fill a Signal of the given size with a Hamming window.
size - the number of samples in the Signal.
pad - the number of samples of the size that is zero padding.

```
Signal.hammingWindow(1024).plot;

Signal.hammingWindow(1024, 512).plot;
```

*welchWindow(size)

Fill a Signal of the given size with a Welch window.
size - the number of samples in the Signal.
pad - the number of samples of the size that is zero padding.

```
Signal.welchWindow(1024).plot;

Signal.welchWindow(1024, 512).plot;
```

*rectWindow(size)

Fill a Signal of the given size with a rectangular window.
size - the number of samples in the Signal.
pad - the number of samples of the size that is zero padding.

```
Signal.rectWindow(1024).plot;
```

```
Signal.rectWindow(1024, 512).plot;
```

Instance Methods

plot(name, bounds)

Plot the Signal in a window. The arguments are not required and if not given defaults will be used.
name - a String, the name of the window.
bounds - a Rect giving the bounds of the window.

```
Signal.sineFill(512, [1]).plot;
```

```
Signal.sineFill(512, [1]).plot("Signal 1", Rect(50, 50, 150, 450));
```

play(loop, mul, numChannels, server)

loads the signal into a buffer on the server and plays it.
returns the buffer so you can free it again.
loop - A Boolean whether to loop the entire signal or play it once. Default is to loop.
mul - volume at which to play it, 0.2 by default.
numChannels - if the signal is an interleaved multichannel file, number of channels, default is 1.
server - the server on which to load the signal into a buffer.

```
b = Signal.sineFill(512, [1]).play(true, 0.2);
        // free the buffer again.
```

waveFill(function, start, end)

Fill the Signal with a function evaluated over an interval.
function - a function that should calculate the value of a sample.
The function is called with two arguments.
x - the value along the interval.
i - the sample index.
start - the starting value of the interval
end - the ending value of the interval.

```
(
s = Signal.newClear(512);
s.waveFill({ arg x, i; sin(x).max(0) }, 0, 3pi);
s.plot;
)
```

asWavetable

Convert the Signal into a Wavetable.

```
Signal.sineFill(512, [1]).asWavetable.plot;
```

fill(val)

Fill the Signal with a value.

```
Signal.newClear(512).fill(0.2).plot;
```

scale(scale)

Scale the Signal by a factor **in place**.

```
a = Signal[1,2,3,4];
a.scale(0.5); a;
```

offset(offset)

Offset the Signal by a value **in place**.

```
a = Signal[1,2,3,4];
a.offset(0.5); a;
```

peak

Return the peak absolute value of a Signal.

```
Signal[1,2,-3,2.5].peak;
```

normalize

Normalize the Signal **in place** such that the maximum absolute peak value is 1.

```
Signal[1,2,-4,2.5].normalize;
Signal[1,2,-4,2.5].normalize(0, 1); // normalize only a range
```

normalizeTransfer

Normalizes a transfer function so that the center value of the table is offset to zero and the absolute peak value is 1. Transfer functions are meant to be used in the Shaper ugen.

```
Signal[1,2,3,2.5,1].normalizeTransfer;
```

invert

Invert the Signal **in place**.

```
a = Signal[1,2,3,4];
a.invert(0.5); a;
```

reverse(beginSamp, endSamp)

Reverse a subrange of the Signal **in place**.

```
a = Signal[1,2,3,4];
a.reverse(1,2); a;
```

fade(beginSamp, endSamp, beginLevel, endLevel)

Fade a subrange of the Signal **in place**.

```
a = Signal.fill(10, 1);
a.fade(0, 3);  // fade in
a.fade(6, 9, 1, 0); // fade out
```

integral

Return the integral of a signal.

```
Signal[1,2,3,4].integral;
```

overDub(aSignal, index)

Add a signal to myself starting at the index.
If the other signal is too long only the first part is overdubbed.

```
a = Signal.fill(10, 100);
a.overDub(Signal[1,2,3,4], 3);

  // run out of range
a = Signal.fill(10, 100);
a.overDub(Signal[1,2,3,4], 8);

a = Signal.fill(10, 100);
a.overDub(Signal[1,2,3,4], -4);

a = Signal.fill(10, 100);
a.overDub(Signal[1,2,3,4], -1);
```

```
a = Signal.fill(10, 100);
a.overDub(Signal[1,2,3,4], -2);

a = Signal.fill(4, 100);
a.overDub(Signal[1,2,3,4,5,6,7,8], -2);
```

overWrite(aSignal, index)

Write a signal to myself starting at the index.
If the other signal is too long only the first part is overdubbed.

```
a = Signal.fill(10, 100);
a.overWrite(Signal[1,2,3,4], 3);

  // run out of range
a = Signal.fill(10, 100);
a.overWrite(Signal[1,2,3,4], 8);

a = Signal.fill(10, 100);
a.overWrite(Signal[1,2,3,4], -4);

a = Signal.fill(10, 100);
a.overWrite(Signal[1,2,3,4], -1);

a = Signal.fill(10, 100);
a.overWrite(Signal[1,2,3,4], -2);

a = Signal.fill(4, 100);
a.overWrite(Signal[1,2,3,4,5,6,7,8], -2);
```

blend(aSignal, blend)

Blend two signals by some proportion.

```
Signal[1,2,3,4].blend(Signal[5,5,5,0], 0);
Signal[1,2,3,4].blend(Signal[5,5,5,0], 0.2);
Signal[1,2,3,4].blend(Signal[5,5,5,0], 0.4);
Signal[1,2,3,4].blend(Signal[5,5,5,0], 1);
```

```
Signal[1,2,3,4].blend(Signal[5,5,5,0], 2);
```

Fourier Transform:

fftCosTable(size)

Fill a Signal with the cosine table needed by the FFT methods.

```
Signal.fftCosTable(512).plot;
```

fft(imag, cosTable)

Perform an FFT on a real and imaginary signal **in place**.

```
(
var size = 512, real, imag, cosTable, complex;

real = Signal.newClear(size);
  // some harmonics
real.sineFill2([[8], [13, 0.5], [21, 0.25], [55, 0.125, 0.5pi]]);
  // add a little noise
real.overDub(Signal.fill(size, { 0.2.bilinrand }));

imag = Signal.newClear(size);
cosTable = Signal.fftCosTable(size);

complex = fft(real, imag, cosTable);
[real, imag, (complex.magnitude) / 100 ].flop.flat
.plot("fft", Rect(0,0, 512 + 8, 500), numChannels:  3);
)
```

ifft(imag, cosTable)

Perform an inverse FFT on a real and imaginary signal **in place**.

```
(
var size = 512, real, imag, cosTable, complex, ifft;

real = Signal.newClear(size);
  // some harmonics
real.sineFill2([[8], [13, 0.5], [21, 0.25], [55, 0.125, 0.5pi]]);
  // add a little noise
real.overDub(Signal.fill(size, { 0.2.bilinrand }));

imag = Signal.newClear(size);
cosTable = Signal.fftCosTable(size);

complex = fft(real, imag, cosTable).postln;
ifft = complex.real.ifft(complex.imag, cosTable);

[real, ifft.real].flop.flat.plot("fft and back", Rect(0,0, 512 + 8, 500), numChannels:  2);
)
```

Unary Messages:

Signal will respond to unary operators by returning a new Signal.

neg, abs, sign, squared, cubed, sqrt
exp, log, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh,
distort, softclip, nyqring, isPositive, isNegative,
isStrictlyPositive

```
x = Signal.sineFill(512, [0,0,0,1]);
[x, x.neg, x.abs, x.sign, x.squared, x.cubed,
x.asin.normalize, x.exp.normalize, x.distort].flop.flat.plot(numChannels:  9);
```

Binary Messages:

Signal will respond to binary operators by returning a new Signal.

+, -, *, /, div, %, **, min, max, ring1, ring2, ring3, ring4, difsqr, sumsqr, sqrdif, absdif, amclip, scaleneg, clip2, excess, <!

```
(
x = Signal.fill(512, { rrand(0.0, 1.0) });
y = Signal.fill(512, { | i| (i * pi / 64).sin });
[x, y, (x + y) * 0.5, x * y, min(x, y), max(x, y) ].flop.flat.plot(numChannels:  6);
)
```

ID: 71

SortedList

Superclass: List

SortedList is a Collection whose items are kept in a sorted order.

Creation

*new(size, function)

Creates a SortedList with the initial capacity given by **size** and a comparison **function**.

Instance Methods

add(item)

Adds an item in the SortedList at the correct position.

```
SortedList[1, 2, 5, 6].add(4).postln;
```

addAll(aCollection)

Adds all the items in the collection into the SortedList.

```
SortedList[1, 2, 5, 6].addAll([0, 3, 4, 7]).postln;
```

ID: 72

String

Superclass: RawArray

String represents an array of characters.
Strings can be written literally using double quotes:

```
"my string".class.postln;
```

Class Methods

*readNew(file)

Read the entire contents of a **File** and return them as a new String.

Instance Methods

at(index)

Strings respond to .at in a manner similar to other indexed collections. Each element is a **Char**.

```
"ABCDEFG".at(2).postln;
```

compare(aString)

Returns a -1, 0, or 1 depending on whether the receiver should be sorted before the argument,
is equal to the argument or should be sorted after the argument. This is a case sensitive compare.

< aString

Returns a Boolean whether the receiver should be sorted before the argument.

== aString

Returns a Boolean whether the two Strings are equal.

post

Prints the string to the current post window.

postln

Prints the string and a carriage return to the current post window.

postc postcln

As post and postln above, but formatted as a comment.

```
"This is a comment."
```

postf

Prints a formatted string with arguments to the current post window. The % character in the format string is replaced by a string representation of an argument. To print a % character use \\% .

```
	"this % a %.  pi = %, list = %\n", "is"  "test"

this is a test.  pi = 3.1416, list = [ 1, 2, 3, 4 ]
```

format

Returns a formatted string with arguments. The % character in the format string is replaced by a string representation of an argument. To print a % character use \\% .

```
	"this % a %.  pi = %, list = %\n", "is"  "test"

this is a test.  pi = 3.1416, list = [ 1, 2, 3, 4 ]
```

error

Prepends an error banner and posts the string

warn

Prepends a warning banner and posts the string.

inform

Posts the string.

++ aString

Return a concatenation of the two strings.

+ aString

Return a concatenation of the two strings with a space between them.

compile

Compiles a String containing legal SuperCollider code and returns a Function.

```
(
var f;
f = "2 + 1".compile.postln;
f.value.postln;
)
```

asCompileString

Returns a String formatted for compiling.

```
(
var f;
    "myString"
f.postln;
f.asCompileString.postln;
)
```

postcs

As postln, but posts the compileString of the reciever

```
List[1, 2, ["comment", [3, 2]], { 1.0.rand }].postcs;
```

interpret

Compile and execute a String containing legal SuperCollider code, returning the result.

```
"2 + 1".interpret.postln;
```

interpretPrint

Compile, execute and print the result of a String containing legal SuperCollider code.

```
"2 + 1".interpretPrint;
```

asSymbol

Return a Symbol derived from the String.

```
(
var z;
z = "myString".asSymbol.postln;
z.class.postln;
)
```

asInteger

Return an Integer derived from the String. Strings beginning with non-numeric characters return 0.

```
"4".asInteger.postln;
```

asFloat

Return a Float derived from the String. Strings beginning with non-numeric characters return 0.

```
"4.3".asFloat.postln;
```

catArgs(... args)

Concatenate this string with the following args.

```
"These are some args:  ".catArgs(\fish, SinOsc.ar, {4 + 3}).postln;
```

scatArgs(... args)

Same as catArgs, but with spaces in between.

```
"These are some args:  ".scatArgs(\fish, SinOsc.ar, {4 + 3}).postln;
```

ccatArgs(... args)

Same as catArgs, but with commas in between.

```
"a String".ccatArgs(\fish, SinOsc.ar, {4 + 3}).postln;
```

catList(list) scatList(list) ccatList(list)

As catArgs, scatArgs and ccatArgs above, but takes a Collection (usually a List or an Array) as an argument.

```
"a String".ccatList([\fish, SinOsc.ar, {4 + 3}]).postln;
```

split(separator)

Returns an Array of Strings split at the separator. The separator is a **Char**, and is not included in the output array. The default separator is $/, handy for Unix paths.

```
"This/could/be/a/Unix/path"
"These are several words"
```

find(string)

Returns the index of the string in the receiver, or nil if not found.

```
"These are several words"      "are"
"These are several words"      "fish"
```

findAll(string)

Returns the indices of the string in the receiver, or nil if not found.

```
"These are several words which are fish"        "are"
"These are several words which are fish"        "fish"
```

contains(string)

Returns a **Boolean** indicating if the String contains string.

```
"These are several words"          "are"
"These are several words"          "fish"
```

containsi(string)

Same as contains, but case insensitive.

```
"These are several words"           "ArE"
```

containsStringAt(index, string)

Returns a **Boolean** indicating if the String contains string beginning at the specified index.

```
"These are several words".containsStringAt(6, "are").postln;
```

icontainsStringAt(index, string)

Same as containsStringAt, but case insensitive.

escapeChar(charToEscape)

Add the escape character (\) at the location of your choice.

```
"This will become a Unix friendly string"
```

tr(from, to)

Transliteration. Replace all instances of from with to.

```
":-(:-(:-("                    //turn the frowns upside down
```

printOn(stream)

Print the String on stream.

```
"Print this on Post"         Post

// equivalent to:
Post   "Print this on Post"
```

storeOn(stream)

Same as printOn, but formatted asCompileString.

```
"Store this on Post"         Post

// equivalent to:
Post    "Store this on Post"
```

inspectorClass

Returns class StringInspector.

stripRTF

Returns a new String with all RTF formatting removed.

```
(
// same as File-readAllStringRTF
    File "/code/SuperCollider3/build/Help/UGens/Chaos/HenonC.help.rtf" "r"
g.readAllString.stripRTF.postln;
g.close;
)
```

Unix Support

Where relevant, the current working directory is the same as the location of the SuperCollider app and the shell is the Bourne shell (sh). Note that the cwd, and indeed the shell itself, does not persist:

```
"echo $0"          // print the shell (sh)
"pwd".unixCmd;
"cd Help/"
"pwd".unixCmd;

"export FISH=mackerel"
"echo $FISH"
```

It is however possible to execute complex commands:

```
"pwd; cd Help/; pwd"
"export FISH=mackerel; echo $FISH"
```

Should you need an environment variable to persist you can use **setenv** (see below).

unixCmd

Execute the String on the command line using the Bourne shell (sh) and send stdout to the post window. See man sh for more details.

```
"ls Help".unixCmd;
```

setenv(value)

Set the environment variable indicated in the string to equal the String **value**. This value will persist until it is changed or SC is quit. Note that if **value** is a path you may need to call standardizePath on it (see below).

```
// all defs in this directory will be loaded when a local server boots
"SC_SYNTHDEF_PATH"        " /scwork/"
"echo $SC_SYNTHDEF_PATH"
```

getenv

Returns the value contained in the environment variable indicated by the String.

```
"USER".getenv;
```

pathMatch

Returns an **Array** containing all paths matching this String. Wildcards apply, non-recursive.

```
Post << "Help/*".pathMatch;
```

loadPaths

Perform pathMatch (see above) on this String, then load and execute all paths in the resultant **Array**.

```
"Help/Collections/loadPaths example.rtf"            //This file posts some text
```

load

Load and execute the file at the path represented by the receiver.

standardizePath

Expand to your home directory, and resolve symbolic links. See **PathName** for more complex needs.

```
" "                 //This will print your home directory
```

basename

Return the filename from a Unix path.

```
"Imaginary/Directory/fish.rtf"
```

dirname

Return the directory name from a Unix path.

```
"Imaginary/Directory/fish.rtf"
```

splitext

Split off the extension from a filename or path and return both in an **Array** as [path or filename, extension].

```
"fish.rtf".splitext;
"Imaginary/Directory/fish.rtf"
```

Document Support

newTextWindow(title, makeListener)

Create a new **Document** with this.

```
"Here is a new Document"
```

openDocument

Create a new **Document** from the path corresponding to this. Returns the Document.

```
(
    "Help/Help.help.rtf"
d.class;
)
```

openTextFile(selectionStart, selectionLength)

Create a new **Document** from the path corresponding to this. The selection arguments will preselect the indicated range in the new window. Returns this.

```
(
d = "Help/Help.help.rtf".openTextFile(0, 20);
d.class;
)
```

findHelpFile

Returns the path for the helpfile named this, if it exists, else returns nil.

```
"Document".findHelpFile;
"foobar".findHelpFile;
```

openHelpFile

Performs foundHelpFile(above) on this, and opens the file it if it exists, otherwise opens the main helpfile.

```
"Document".openHelpFile;
"foobar".openHelpFile;
```

Drawing Support

The following methods must be called within an SCWindow-drawHook or a SCUserView-drawFunc function, and will only be visible once the window or the view is refreshed. Each call to SCWindow-refresh SCUserView-refresh will 'overwrite' all previous drawing by executing the currently defined function.

See also: **[SCWindow]**, **[SCUserView]**, **[Color]**, and **[Pen]**.

draw

Draws the String at the current 0@0 **[Point]**. If not transformations of the graphics state have taken place this will be the upper left corner of the window. See also **[Pen]**.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
 "abababababa\n\n\n"
};
w.refresh
)
```

drawAtPoint(point, font, color)

Draws the String at the given **[Point]** using the **[Font]** and **[Color]** specified.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
"abababababa\n\n\n".scramble.drawAtPoint(
100@100,
Font(" Gadget", 30),
```

```
Color.blue(0.3, 0.5))
};
w.refresh
)
```

drawInRect(rect, font, color)

Draws the String into the given **[Rect]** using the **[Font]** and **[Color]** specified.

```
(
w = SCWindow.new.front;
r = Rect(100, 100, 100, 100);
w.view.background_(Color.white);
w.drawHook = {
"abababababa\n\n\n".scramble.drawInRect(r, Font(" Gadget", 30), Color.blue(0.3, 0.5));
Pen.strokeRect(r);
};
w.refresh
)
```

// drawCenteredIn(inRect)

draws the String into the center of the given rect with font and color into the window.
Unfortunately does not work for now...

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
"abababababa\n\n\n".scramble.drawCenteredIn(
Rect(100, 100, 100, 100),
Font(" Gadget", 30),
Color.blue(0.3, 0.5)
)
};
w.refresh
)
```

// drawLeftJustIn(inRect)

Unfortunately does not work for now...

```
(
w = SCWindow.new.front;
```

```
w.view.background_(Color.white);
w.drawHook = {
"abababababa\n\n\n".scramble.drawLeftJustIn(
Rect(100, 100, 100, 100),
Font(" Gadget", 30),
Color.blue(0.3, 0.5)
)
};
w.refresh
)
```

// drawRightJustIn(inRect)

Unfortunately does not work for now...

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
"abababababa\n\n\n".scramble.drawLeftJustIn(
Rect(100, 100, 100, 100),
Font(" Gadget", 30),
Color.blue(0.3, 0.5)
)
};
w.refresh
)
```

ID: 73

Wavetable

Superclass: FloatArray

A Wavetable is a FloatArray in a special format used by SuperCollider's interpolating oscillators. Wavetables cannot be created by **new**.

Creation

*sineFill(size, amplitudes, phases)

Fill a Wavetable of the given size with a sum of sines at the given amplitudes and phases. The Wavetable will be normalized.
size - must be a power of 2.
amplitudes - an Array of amplitudes for each harmonic beginning with the fundamental.
phases - an Array of phases in radians for each harmonic beginning with the fundamental.

```
Wavetable.sineFill(512, 1.0/[1,2,3,4,5,6]).plot;
```

*chebyFill(size, amplitudes, phases)

Fill a Wavetable of the given size with a sum of Chebyshev polynomials at the given amplitudes
for use in waveshaping by the Shaper ugen.
The Wavetable will be normalized.
size - must be a power of 2.
amplitudes - an Array of amplitudes for each Chebyshev polynomial beginning with order 1.

```
Wavetable.chebyFill(512, [1]).plot;

Wavetable.chebyFill(512, [0, 1]).plot;

Wavetable.chebyFill(512, [0, 0, 1]).plot;

Wavetable.chebyFill(512, [0.3, -0.8, 1.1]).plot;
```

Instance Methods

plot(name, bounds)

Plot the Wavetable in a window. The arguments are not required and if not given defaults will be used.
name - a String, the name of the window.
bounds - a Rect giving the bounds of the window.

```
Wavetable.sineFill(512, [1]).plot;
```

```
Wavetable.sineFill(512, [1]).plot("Table 1", Rect.newBy(50, 50, 150, 450));
```

play(name)

Plays the Wavetable in a Mixer channel.
name - a Symbol or String giving the name of the mixer channel.

```
Wavetable.sineFill(512, [1]).play;
```

```
Wavetable.sineFill(512, [1]).play("Table 1");
```

asSignal

Convert the Wavetable into a Signal.

```
Wavetable.sineFill(512, [1]).asSignal.plot;
```

4 Control

ID: 74

CmdPeriod register objects to be cleared when Cmd-. is pressed

Superclass: Object

CmdPeriod allows you to register objects to perform an action when the user presses Cmd-. These objects must implement a method called -cmdPeriod which performs the necessary tasks. (You can add such a method in your custom classes.) Note that since **[Function]** implements -cmdPeriod as a synonym for -value, it is possible to register any function (and thus any arbitrary code) for evaluation when Cmd-. is pressed.

N.B. Cmd-. uses an **[IdentitySet]** to store its registered objects. For this reason you cannot rely on the order in which the objects will be cleared. If you need -cmdPeriod to be called on a set of objects in a given order then it is best to wrap those within a **[Function]** and register that. (See example below.)

Class Methods

*add(object)

Registers an object to be cleared when Cmd-. is pressed. This object will stay registered until it is explicitly removed, and will thus respond to additional presses of Cmd-.

*remove(object)

Removes an object that was previously registered.

*doOnce(object)

Registers an Object o be evaluated once, and then unregistered.

Examples

```
(
f = {"foo".postln };
g = {"bar".postln };
CmdPeriod
CmdPeriod
)
```

```
// Now press Cmd-.

CmdPeriod.remove(g);

// Now press Cmd-.  Only f executes

CmdPeriod            // must explicitly cleanup

//// Controlling order of execution
(
f = {"first".postln };
g = {"second".postln };

// order is arbitrary, so wrap in a function

h = { f.cmdPeriod; g.cmdPeriod;};

CmdPeriod
)

// Now press Cmd-.

CmdPeriod.remove(h);



// note that often you want to automatically remove the function once it is evaluated
// instead of

f = { "foo".postln; CmdPeriod.remove(f) };
CmdPeriod

// you can write:

CmdPeriod.doOnce { "foo".postln }

// a typical example:
(
w = SCWindow.new("close on cmd-.").front;
CmdPeriod.doOnce { w.close };
```

)

ID: 75

ContiguousBlockAllocator

A more robust replacement for the default server block allocator, PowerOfTwoAllocator. May be used in the Server class to allocate audio/control bus numbers and buffer numbers.

To configure a server to use ContiguousBlockAllocator, execute the following:

```
aServer.options.blockAllocClass = ContiguousBlockAllocator;
```

Normally you will not need to address the allocators directly. However, ContiguousBlockAllocator adds one feature not present in PowerOfTwoAllocator, namely the *reserve* method.

*new(size, pos = 0)

Create a new allocator with *size* slots. You may block off the first *pos* slots (the server's audioBusAllocator does this to reserve the hardware input and output buses).

alloc(n = 1)

Return the starting index of a free block that is *n* slots wide. The default is 1 slot.

free(address)

Free a previously allocated block starting at *address*.

reserve(address, size = 1)

Mark a specific range of addresses as used so that the alloc method will not return any addresses within that range.

ID: 76

ControlSpec specification for a control input

superclass: Spec

The original, and most common spec. (see [Spec])

ControlSpec.new(minval, maxval, warp, step, default,units);

The most common way to create one is by

anObject.asSpec

```
// nil becomes a default ControlSpec
nil.asSpec.dump
Instance of ControlSpec {    (0313FF18, gc=00, fmt=00, flg=00, set=03)
instance variables [6]
minval :  Float 0
maxval :  Float 1
warp :  Symbol 'linear'
step :  Float 0
default :  Float 0
}


// array is used as arguments to ControlSpec.new( minval, maxval, warp, step, default)
[300,3000,\exponential,100].asSpec.dump
Instance of ControlSpec {    (0313FC08, gc=00, fmt=00, flg=00, set=03)
instance variables [6]
minval :  Integer 300
maxval :  Integer 3000
    warp :  Symbol'exponential'
step :  Integer 100
default :  Integer 300
}
// partially specified ...
[-48,48].asSpec.dump
Instance of ControlSpec {    (0313FF18, gc=00, fmt=00, flg=00, set=03)
instance variables [6]
```

```
minval :  Integer -48
maxval :  Integer 48
warp :  Symbol 'linear'
step :  Float 0
default :  Integer -48
}
```

constrain (value)
clips and rounds the value to within the spec

map (value)
maps a value from [0..1] to spec range

unmap (value)
maps a value from the spec range to [0..1]

```
 // example

 // make a frquency spec with an exponential range from 20 to 20000,
 // give it a rounding of 30 (Hz)
a = \freq.asSpec;
a.step = 100;

 // equivalent:
a = [20, 20000, 'exp', 100, 440].asSpec;
a.dump;

 a.constrain(800); // make sure it is in range and round it.
  a.constrain(803); // make sure it is in range and round it.

a.map(0.5);
 a.map(0.0); // returns min
 a.map(1.5); // exceeds the area:  clip, returns max
```

```
a.unmap(4000);
a.unmap(22.0);

 // using spec for sliders:
(
var w, c, d;
w = SCWindow("control", Rect(128, 64, 340, 160));
w.front;
c = SCSlider(w, Rect(10, 10, 300, 30));
d = SCStaticText(w, Rect(10, 40, 300, 30));
c.action = {
  d.string = "unmapped value"
   + c.value.round(0.01)
   + "......"
   + "mapped value"
+ a.map(c.value)
};
)
```

ID: 77

DebugNodeWatcher

Posts when these messages are received from the server:

n_go n_end n_off n_on

```
s = Server.default;
s.boot;

    DebugNodeWatcher
d.start;

y = Group.new;

y.free;

d.stop;
```

ID: 78

Env envelope specification

superclass: Object

An Env is a specification for a segmented envelope. Envs can be used both server-side, by an **EnvGen** within a SynthDef, and clientside, with methods such as **at** and **asStream**, below. An Env can have any number of segments which can stop at a particular value or loop several segments when sustaining. An Env can have several shapes for its segments.

Basic Creation

*new(levels, times, curves, releaseNode, loopNode)

Create a new envelope specification.
levels - an array of levels. The first level is the initial value of the envelope.
times - an array of durations of segments in seconds. There should be one fewer duration than there are levels.
curve - this parameter determines the shape of the envelope segments.
The possible values are:
'step' - flat segments
'linear' - linear segments, the default
'exponential' - natural exponential growth and decay. In this case, the levels must all be nonzero
and the have the same sign.
'sine' - sinusoidal S shaped segments.
'welch' - sinusoidal segments shaped like the sides of a Welch window.
a Float - a curvature value for all segments.
An Array of Floats - curvature values for each segments.
releaseNode - an Integer or nil. The envelope will sustain at the release node until released.
loopNode - an Integer or nil. If not nil the sustain portion will loop from the releaseNode to the loop node.

```
    s.boot;
  // different shaped segments:
Env.new([0,1, 0.3, 0.8, 0], [2, 3, 1, 4],'linear').test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001], [2, 3, 1, 4],'exponential').test.plot;
```

```
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],'sine').test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001],[2,3,1,4],'welch').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],'step').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], -2).test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], 2).test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], [0, 3, -3, -1]).test.plot;
```

If a release node is given, and the gate input of the EnvGen is set to zero, it outputs the nodes after the release node:

```
        //release node is node 2; releases after 5 sec
Env.new([0.001,1,0.3,0.8,0.001],[2,3,1,4] * 0.2, 2, 2).test(5).plot;
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,3,1,2,2,1] * 0.2, 2, 2).test(5).plot;
  //instant release
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,3,1,2,2,1] * 0.2, 2, 2).test(0.1).plot;
```

If a loop node is given, the EnvGen outputs the nodes between the release node and the loop node until it is released:

```
        //release node is node 3, loop node is node 1
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,1,1,2,3,1] * 0.1, 'lin', 3, 1).test(3).plot;
```

Note:

The starting level for an envelope segment is always the level you are at right now. For example when the gate is released and you jump to the release segment, the level does not jump to the level at the beginning of the release segment, it changes from the whatever the current level is to the goal level of the release segment over the specified duration of the release segment.

There is an extra level at the beginning of the envelope to set the initial level. After that each node is a goal level and a duration, so node zero has duration equal to times[0] and goal level equal to levels[1].

The loop jumps back to the loop node. The endpoint of that segment is the goal level for that segment and the duration of that segment will be the time over which the level changed from the current level to the goal level.

***newClear(numSegments)**

Creates a new envelope specification with **numSegments** for filling in later. This can be useful when passing Env parameters as args to a **[Synth]**. Note that the maximum number of segments is fixed and cannot be changed once embedded in a **[SynthDef]**. Trying to set an Env with more segments than then this may result in other args being unexpectedly set.

```
(
SynthDef("Help-Env-newClear", { arg i_outbus=0, t_gate ;
var env, envctl;
   // make an empty 4 segment envelope
env = Env.newClear(4);
   // create a control argument array
envctl = Control.names([\env]).kr( env.asArray );
Out.ar(i_outbus, SinOsc.ar(EnvGen.kr(envctl, t_gate), 0, 0.3));
}).send(s);
)
(
s.makeBundle(nil, {
   // must not have more segments than the env above
e = Env([700,900,900,800], [1,1,1], \exp); // 3 segments
   x = Synth "Help-Env-newClear"   \t_gate
x.setn(\env, e.asArray);
});
)
(
  // reset then play again
e = Env([800,300,400,500,200], [1,1,1,1], \exp); // 4 segments
x.setn(\env, e.asArray);
x.set(\t_gate, 1);
)
x.free;
```

Standard Shape Envelope Creation Methods

The following class methods create some frequently used envelope shapes based on supplied durations.

*linen(attackTime, sustainTime, releaseTime, level, curve)

Creates a new envelope specification which has a trapezoidal shape.

attackTime - the duration of the attack portion.
sustainTime - the duration of the sustain portion.
releaseTime - the duration of the release portion.
level - the level of the sustain portion.
curve - the curvature of the envelope.

```
s.boot;
Env.linen(1, 2, 3, 0.6).test.plot;
Env.linen(0.1, 0.2, 0.1, 0.6).test.plot;
Env.linen(1, 2, 3, 0.6, 'sine').test.plot;
Env.linen(1, 2, 3, 0.6, 'welch').test.plot;
Env.linen(1, 2, 3, 0.6, -3).test.plot;
Env.linen(1, 2, 3, 0.6, -3).test.plot;
```

*triangle(duration, level)

Creates a new envelope specification which has a triangle shape.
duration - the duration of the envelope.
level - the peak level of the envelope.

```
Env.triangle(1, 1).test.plot;
```

*sine(duration, level)

Creates a new envelope specification which has a hanning window shape.
duration - the duration of the envelope.
level - the peak level of the envelope.

```
Env.sine(1,1).test.plot;
```

*perc(attackTime, releaseTime, peakLevel, curve)

Creates a new envelope specification which (usually) has a percussive shape.
attackTime - the duration of the attack portion.
releaseTime - the duration of the release portion.
peakLevel - the peak level of the envelope.
curve - the curvature of the envelope.

```
Env.perc(0.05, 1, 1, -4).test.plot;
Env.perc(0.001, 1, 1, -4).test.plot; // sharper attack
Env.perc(0.001, 1, 1, -8).test.plot; // change curvature
Env.perc(1, 0.01, 1, 4).test.plot; // reverse envelope
```

Sustained Envelope Creation Methods

The following methods create some frequently used envelope shapes which have a sustain segment.

*adsr(attackTime, decayTime, sustainLevel, releaseTime, peakLevel, curve)

Creates a new envelope specification which is shaped like traditional analog attack-decay-sustain-release (adsr) envelopes.
attackTime - the duration of the attack portion.
decayTime - the duration of the decay portion.
sustainLevel - the level of the sustain portion as a ratio of the peak level.
releaseTime - the duration of the release portion.
peakLevel - the peak level of the envelope.
curve - the curvature of the envelope.

```
Env.adsr(0.02, 0.2, 0.25, 1, 1, -4).test(2).plot;
Env.adsr(0.001, 0.2, 0.25, 1, 1, -4).test(2).plot;
  //release after 0.45 sec
Env.adsr(0.001, 0.2, 0.25, 1, 1, -4).test(0.45).plot;
```

*dadsr(delayTime, attackTime, decayTime, sustainLevel, releaseTime, peakLevel, curve)

As ***adsr** above, but with it's onset delayed by **delayTime** in seconds. The default delay is 0.1.

*asr(attackTime, sustainLevel, releaseTime, peakLevel, curve)

Creates a new envelope specification which is shaped like traditional analog attack-sustain-release (asr) envelopes.
attackTime - the duration of the attack portion.
sustainLevel - the level of the sustain portion as a ratio of the peak level.
releaseTime - the duration of the release portion.

peakLevel - the peak level of the envelope.
curve - the curvature of the envelope.

```
Env.asr(0.02, 0.5, 1, 1, -4).test(2).plot;
Env.asr(0.001, 0.5, 1, 1, -4).test(2).plot; // sharper attack
Env.asr(0.02, 0.5, 1, 1, 'linear').test(2).plot; // linear segments
```

*cutoff(releaseTime, level, curve)

Creates a new envelope specification which has no attack segment. It simply sustains at the peak level until released. Useful if you only need a fadeout, and more versatile than **[Line]**.
releaseTime - the duration of the release portion.
level - the peak level of the envelope.
curve - the curvature of the envelope.

```
Env.cutoff(1, 1).test(2).plot;
Env.cutoff(1, 1, 4).test(2).plot;
Env.cutoff(1, 1, 'sine').test(2).plot;
```

Instance Methods

blend(anotherEnv, blendFraction)

Blend two envelopes. Returns a new Env.
anotherEnv - an Env.
blendFraction - a number from zero to one.

```
a = Env([0, 0.2, 1, 0.2, 0.2, 0], [0.5, 0.01, 0.01, 0.3, 0.2]).test.plot;
b = Env([0, 0.4, 1, 0.2, 0.5, 0], [0.05, 0.4, 0.01, 0.1, 0.4]).test.plot;

(
Task({
f = (0, 0.2 ..  1);
f.do { | u|
blend(a, b, u).test.plot;
2.wait;
SCWindow.allWindows.pop.close; // close last opened window
```

```
}
}).play(AppClock);
)

  // in a SynthDef
(
SynthDef("Help-EnvBlend", { arg fact = 0;
Out.ar(0, EnvGen.kr(Env.perc.blend(Env.sine, fact), 1.0, doneAction:  2)
* SinOsc.ar(440,0,0.1)
)
}).send(s));

(
{
f = (0, 0.1..1);
f.do({| fact|  Synth("Help-EnvBlend", [\fact, fact.postln]); 1.wait;});

}.fork;)
```

delay(delay)

Returns a new Env based on the receiver in which the start time has been offset by adding a silent segment at the beginning.
delay - The amount of time to delay the start of the envelope.

```
a = Env.perc(0.05, 1, 1, -4);
b = a.delay(2);
a.test.plot;
b.test.plot;
```

test(releaseTime)

Test the envelope on the default **[Server]** with a **[SinOsc]**.
releaseTime - If this is a sustaining envelope, it will be released after this much time in seconds. The default is 3 seconds.

plot(size)

Plot this envelope's shape in a window.
size - The size of the plot. The default is 400.

asSignal(length)

Returns a Signal of size **length** created by sampling this Env at **length** number of intervals.

asArray

Converts the Env to an Array in a specially ordered format. This allows for Env parameters to be settable arguments in a **[SynthDef]**. See example above under ***newClear**.

isSustained

Returns true if this is a sustaining envelope, false otherwise.

Client-side Access and Stream Support

Sustain and loop settings have no effect in the methods below.

at(time)

Returns the value of the Env at **time**.

```
Env.triangle(1, 1).at(0.5);
```

embedInStream

Embeds this Env within an enclosing **[Stream]**. Timing is derived from `thisThread.beats`.

asStream

Creates a Routine and embeds the Env in it. This allows the Env to function as a **[Stream]**.

```
(
{
e = Env.sine.asStream;
5.do({
e.next.postln;
0.25.wait;
```

```
})}.fork
)
```

ID: 79

HIDDeviceService

A Service that provides access to Human Interface Devices like joysticks and gamepads.

This service was mainly designed to use gamepads as control input. The name is derived from the mac osx specifications.
The HIDDeviceService handles all the primitive calls. HIDDevice only stores information about a device and holds an array of HIDElements, which store information about the controllers of the device.
A HIDDevice's information consists out of:
the manufacturer, the product, the usage, the vendorID, the productID and the locID.
The last three are used to identify the device. The vendorID and the productID are static for each device, the locID depends on the (usb) port the device is connected to.
A HIDDeviceElement's information consists out of:
the type, the usage, the cookie, the minimum and the maximum value.
the cookie is a number that can be used to identify an element of a device.

There are two ways of getting values from the device: One is to poll a value, the other one is to start an eventloop that pushes every new value into the language and calls an action (like MIDIIn).
To set up an eventloop follow these steps:
1. initialize the service by calling:

```
HIDDeviceService.buildDeviceList;
```

2. now the information about the devices can be found:

```
(
HIDDeviceService.devices.do({arg dev;
[dev.manufacturer, dev.product, dev.vendorID, dev.productID, dev.locID].postln;
dev.elements.do({arg ele;
[ele.type, ele.usage, ele.cookie, ele.min, ele.max].postln;
});
});
)
```

3. the device needs to be queued, that means that the eventloop actually uses this

device to push values.

```
HIDDeviceService.devices.at(0).queueDevice;
```

4. set an action that is called by the incoming events. In addition to the value the events also deliver the productID, the vendorID and the locID of the device and the cookie of the element.

```
(
HIDDeviceService.action_({arg productID, vendorID, locID, cookie, val;
[productID, vendorID, locID, cookie, val].postln;
});
)
```

5. start the eventloop:

```
        HIDDeviceService.
```

6. stop the eventloop:

```
    HIDDeviceService.
```

buildDeviceList(usagePage, usage)

It is also possible to search for devices in other usage pages. (look in the class file)
the default is: page: GenericDesktop usage: Joystick. if a nil is passed in all devices are listed.

//HIDDeviceServis by jan trutzschler v. falkenstein

deviceSpecs

you can add to the classvar deviceSpecs the specs of your device.
the key used has to be the **product name** derived from the device info.

here is a collection of specs:

```
//wingman
(
HIDDeviceService                'WingMan Action Pad'
   IdentityDictionary
\a -> 0, \b-> 1, \c-> 2,
\x-> 3, \y-> 4, \z-> 5,
    \l      //front left
    \r      //front right
\s-> 8,
\mode-> 9,
\xx-> 10,
\yy-> 11,
\slider-> 12,
\hat-> 13
])
)
//cyborg
(
HIDDeviceService                \cyborg   //not the right product name yet, so this doesn't work.
      IdentityDictionary
\trig -> 0, \a-> 1, \b -> 2, \c -> 3,
\f1-> 4, \f2-> 5, \f3-> 6, \f4 -> 7,
\l -> 8, \r -> 9, // arrow buttons
\hu -> 10, \hl -> 11, \hr -> 12, \hd -> 13, // hat positions
\x -> 14, \y -> 15, \z -> 16, // axes
\slider-> 17,
\hat-> 18
]);
)
```

ID: 80

MIDIIn

A popular 80s technology

This document explains technical details of the MIDI hardware interface class, MIDIIn.

Note that the interface in this class has significant limitations. MIDI responders created directly in MIDIIn cannot be created and destroyed dynamically, which significantly limits the ability to create flexible MIDI configurations. For general programming, **users should not use the MIDIIn class directly**. Instead, use the MIDIResponder classes (see helpfile: **[MIDIResponder]**).

Certain MIDI messages are supported only through MIDIIn. These are: polytouch, program, sysex, sysrt, smpte.

See the **[UsingMIDI]** helpfile for practical considerations and techniques for using MIDI in SC.

The MIDIIn class

MIDIIn links MIDI input received from the operating system to a set of user defined functions.
Only one set of MIDI input handling functions can be active at a time, they are stored in the
following class variables:

noteOff, noteOn, polytouch, control, program, touch, bend, sysex, sysrt, smpte

The first argument these functions receive is an unique identifier that specifies the source of the data.

Quick start for 1 port:

```
(
 MIDIIn          // init for one port midi interface
// register functions:
MIDIIn.noteOff = { arg src, chan, num, vel;  [chan,num,vel / 127].postln; };
MIDIIn.noteOn = { arg src, chan, num, vel;  [chan,num,vel / 127].postln; };
```

```
MIDIIn.polytouch = { arg src, chan, num, vel;  [chan,num,vel / 127].postln; };
MIDIIn.control = { arg src, chan, num, val;  [chan,num,val].postln; };
MIDIIn.program = { arg src, chan, prog;   [chan,prog].postln; };
MIDIIn.touch = { arg src, chan, pressure;  [chan,pressure].postln; };
MIDIIn.bend = { arg src, chan, bend;    [chan,bend - 8192].postln; };
MIDIIn.sysex = { arg src, sysex;    sysex.postln; };
MIDIIn.sysrt = { arg src, chan, val;    [chan,val].postln; };
MIDIIn.smpte = { arg src, chan, val;    [chan,val].postln; };
)
```

Quick start for 2 or more ports:

```
(
var inPorts = 2;
var outPorts = 2;
 MIDIClient                          // explicitly intialize the client
inPorts.do({ arg i;
 MIDIIn.connect(i, MIDIClient.sources.at(i));
});
)
```

class methods:

*noteOn_(function)

function is evaluated whenever a MIDI noteOn message is received, it is passed the following arguments:
uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
keyNumber 0 - 127
velocity 0 - 127

*noteOff_(function)

uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
keyNumber 0 - 127
velocity 0 - 127, typically 64 unless noteOff velocity is supported

*polytouch_(function)

uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15

keyNumber 0 - 127
pressure 0 - 127

***control_(function)**
uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
controllerNumber 0 - 127
value 0 - 127

***program_(function)**
uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
programNumber 0 - 127

***touch_(function)**
uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
pressure 0 - 127

***bend_(function)**
uid unique identifier of the MIDI port
MIDIchannel ranges from 0 to 15
bend 0..16384, the midpoint is 8192

***sysex_(function)**
uid unique identifier of the MIDI port
system exclusive data an Int8Array (includes f0 and f7)
see manufacturer references for details
note: The current implementation assembles a complete system exclusive packet before evaluating the function.

***sysrt_(function)**
uid unique identifier of the MIDI port
index ranges from 0 to 15
data 0 - 127

#020202 **index data message**
#020202 2 14bits song pointer
#020202 3 7bits song select
#020202 8 midiclock
#020202 10 start
#020202 11 continue
#020202 12 stop

***smpte**
uid unique identifier of the MIDI port

index ranges from 0 to 7
data 4 bits

Over MIDI, SMPTE is transmitted at 1/4 frame intervals four times faster than the frame rate.
index data
0 frames low nibble
1 frames hi nibble
2 seconds low nibble
3 seconds hi nibble
4 minutes low nibble
5 minutes hi nibble
6 hours low nibble
7 hours hi **bit** OR'ed with frameRate
0 -> 24fps
2 -> 25 fps
4 -> 30 fps drop frame
6 -> 30 fps
Nibbles are sent in ascending order,

```
(
MIDIIn.connect;
s = Server.local;
s.boot;
s.latency = 0;

SynthDef("sik-goo", { arg freq=440,formfreq=100,gate=0.0,bwfreq=800;
var x;
x = Formant.ar(
SinOsc.kr(0.02, 0, 10, freq),
formfreq,
bwfreq
);
x = EnvGen.kr(Env.adsr, gate,Latch.kr(gate,gate)) * x;
Out.ar(0, x);
}).send(s);

    Synth "sik-goo"
```

```
//set the action:
MIDIIn.noteOn = {arg src, chan, num, vel;
x.set(\freq, num.midicps / 4.0);
x.set(\gate, vel / 200 );
x.set(\formfreq, vel / 127 * 1000);
};
MIDIIn.noteOff = { arg src,chan,num,vel;
x.set(\gate, 0.0);
};
MIDIIn.bend = { arg src,chan,val;
    //(val * 0.048828125).postln;
x.set(\bwfreq, val * 0.048828125 );
};
)
```

//i used this and got acceptable latency for triggering synths live.
//The latency might actually be less than sc2, but i haven't used it enough
//to tell for sure yet.
//Powerbook G4, 512mb ram.
- matrix6k@somahq.com

```
writing to the bus rather than directly to the synth

s = Server.local;
s.boot;

(
s.latency = 0;
SynthDef("moto-rev", { arg ffreq=100;
var x;
x = RLPF.ar(LFPulse.ar(SinOsc.kr(0.2, 0, 10, 21), [0,0.1], 0.1),
ffreq, 0.1)
.clip2(0.4);
Out.ar(0, x);
}).send(s);
```

```
b = Bus.control(s);

    Synth "moto-rev"

// map the synth's first input (ffreq) to read
// from the bus' output index
x.map(0,b.index);



MIDIIn.connect;
//set the action:
MIDIIn.noteOn = {arg src, chan, num, vel;
b.value = num.midicps.postln;
};

MIDIIn.control = {arg src, chan, num, val;
[chan,num,val].postln;
};
MIDIIn.bend = {arg src, chan, val;
val.postln;
};
)

// cleanup
x.free;
b.free;
```

KeyboardSplitfor two voices
pbend to cutoff, mod to rez, 7 to amp
// - matrix6k@somahq.com

prepare

```
s.boot;
(
SynthDef("funk",{ arg freq = 700, amp = 0.2, gate = 1, cutoff = 20000, rez = 1, lfospeed=0;
var e,x,env,range,filterfreq;
e = Env.new([0, 0.1, 0.1, 0], [0, 0.1, 0.1], 'linear', 2);
env=Env.adsr(0.3,1,1,1);
```

```
range = cutoff -1;
filterfreq = SinOsc.kr(lfospeed,0, range, cutoff).abs;
x = RLPF.ar(Mix.ar([
Mix.arFill(2, {Saw.ar(freq *2 + 0.2.rand2, amp)}),
Mix.arFill(2, {Saw.ar(freq *4+ 0.2.rand2, amp)})
]),
EnvGen.kr(env,gate)*filterfreq,
rez);
Out.ar([0,1],x * EnvGen.kr(e, gate, doneAction:  2))

}).send(s);

SynthDef("strings",{ arg freq = 700, amp = 0.2, gate = 1;
var x,enve;
enve = Env.new([0, 0.1, 0.1, 0], [2, 0.1, 1], 'linear', 2);
x = RLPF.ar(Mix.ar([
Mix.arFill(2, {Saw.ar(freq +2.rand2,0.6)}),
Mix.arFill(2, {Saw.ar(freq *0.5 + 2.rand2,0.6)})
]),
6000,1);
Out.ar([0,1],x * EnvGen.kr(enve, gate, doneAction:  2))

}).send(s);

)
```

then...

```
(
var keys, cutspec, cutbus, rezspec, rezbus, lfospec, lfobus;
keys = Array.newClear(128);

MIDIClient
MIDIIn.connect(0, MIDIClient.sources.at(0));

g = Group.new;

cutspec = ControlSpec(100,10000,\linear,0.001);
cutbus = Bus.new(\control,1,1,s);
cutbus.value = 10000;

rezspec = ControlSpec(1,0,\linear,0.001);
```

```
rezbus = Bus.new(\control,2,1,s);
rezbus.value = 1.0;

lfospec = ControlSpec(0,50,\linear,0.001);
lfobus = Bus.new(\control,3,1,s);

MIDIIn.control = {arg src, chan, num, val;
if(num == 1,{
rezbus.value = rezspec.map(val/127.0);
});
if(num == 7,{
lfobus.value = lfospec.map(val/127.0).postln;
});
};
MIDIIn.bend = {arg src, chan, val;
cutbus.value = cutspec.map(val/16383.0);
};

MIDIIn.noteOn = {arg src, chan, num, vel;
var node;
if(num < 60, {
node = Synth.tail(g, "funk", [\freq, num.midicps, \amp, vel/255]);
  node.map("cutoff"   "rez"   "lfospeed"
//  node = Synth.basicNew("funk",s);
//  s.sendBundle(nil,
//   node.addToTailMsg(g,[\freq, num.midicps, \amp, vel/255]),
//   node.mapMsg("cutoff",1,"rez",2,"lfospeed",3)
//  );
keys.put(num, node)
},{
node = Synth.tail(g, "strings", [\freq, num.midicps, \amp, vel/255]);
keys.put(num, node)
});
};

MIDIIn.noteOff = {arg src, chan, num, vel;
var node;
node = keys.at(num);
if (node.notNil, {
keys.put(num, nil);
```

```
s.sendMsg("/n_set", node.nodeID, "gate", 0);
        // or node.release
        // then free it ...  or get the NodeWatcher to do it
});
};

)
```

ID: 81

MIDIOut

MIDIout objects interface MIDI output ports defined by the operating system to the language.
from the operating system to a set of user defined functions.

methods

```
noteOn ( chan, note, veloc )
noteOff ( chan, note, veloc )
polyTouch ( chan, note, val )
control ( chan, ctlNum, val )
program ( chan, num )
touch ( chan, val )
bend ( chan, val )
allNotesOff ( chan )
smpte ( frames, seconds, minutes, hours, frameRate )
songPtr ( songPtr )
songSelect ( song )
midiClock (  )
startClock (  )
continueClock (  )
stopClock (  )
reset (  )
sysex ( uid, Int8Array )
send ( outport, uid, len, hiStatus, loStatus, a, b, latency )
```

```
MIDIClient.init;

m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
m.noteOn(16, 60, 60);

MIDIIn.connect;
MIDIIn.sysex = { arg uid, packet; [uid,packet].postln };
MIDIIn.sysrt = { arg src, chan, val;  [src, chan, val].postln; };
MIDIIn.smpte = { arg src, chan, val;  [src, chan, val].postln; };

m.sysex(MIDIClient.destinations.at(0).uid, Int8Array[ 16rf0, 0, 0, 27, 11, 0,16rf7])
```

```
m.smpte (24,16)
m.midiClock
m.start
m.continue
m.stop
```

ID: 82

NodeWatcher **notify sc-lang side node objects of their server sided state**

Node instances (Synths and Groups) can be registered with the NodeWatcher.
It watches for server node status messages:

```
n_go
n_end
n_off
n_on
```

and sets the isPlaying and isRunning variables on the Node instance accordingly. A Node that ends is unregistered at that time.

In some cases this can be an invaluable service. The use of an independant object to maintain the state keeps the implementation of the Node classes simple.
Note that server notification should be on. (this is default. see: aServer.notify)

the most common use:
NodeWatcher.register(aNode);

*new(server)
create a new instance listening to the server's address

*newFrom(server)
create a new instance listening to the server's address
if there is one present already return that one

*register(aNode, assumePlaying)
aNode can be a Group or a Synth.
the NodeWatcher is created internally
assumePlaying: if true, the node's **isPlaying** field is set to true

*unregister(aNode)
remove the node from the list of nodes.

this happens also when a node is freed.

start
add the OSCresponderNode to listen to the address

stop
remove the OSCresponderNode to stop listen to the address

```
// example:

(
b = s.makeBundle(false, {
  a = Group         //create a node object
  NodeWatcher              // register before creating on the server
});
)
a.isPlaying;
 s.listSendBundle(nil, b); //start the node on the server
a.isPlaying;
a.isRunning;
a.run(false);
a.isRunning;
 s.freeAll; //free all nodes
a.isPlaying;
a.isRunning;
```

DebugNodeWatcher

for debugging, it can be useful to see every node start and end
it doesn't require registration, reacts to each message.

```
// example:
  n = DebugNodeWatcher
n.start;
```

```
x = Group(s);
x.run(false);
x.free;
n.stop;
```

ID: 83

ObjectSpec : Spec

Allows any kind of object to be specified as a default for an Instr argument.

The object should be a kind of object that does not have an Editor in the Patch system... e.g., for Env, the spec should be an EnvSpec. The object will not be editable after patch creation.

Suitable objects for ObjectSpec are static Arrays, other data structures used to build parallel or serial structures, or even Functions that provide additional UGens to the Patch.

***new(obj)**

obj is the object that will be used as the default when the Patch is built.

defaultControl
defaultControl_

Access or change the object.

Example:

In this patch, the Instr defines a filter structure, but leaves the choice of exciter up to the user. If the user doesn't provide an exciter, a default will be used.

Since the Formlet filter's impulse response is a sine wave, formHarmRatios and formHarmAmps accept arrays that create an additive array of Formlets. Formlet is a very efficient UGen, so the Patch is still CPU cheap!

The result resembles CHANT (IRCA/M, 1979).

```
(
// define the Instr
Instr  \analog  \voxlet     | freq, gate, exciterFunc, detune, formfreq, ffreq, env, formfreqenv, at-
tacktime, decaytime, vsens, fenvsens, formHarmRatios, formHarmAmps|
var amp, sig;
formfreq = formfreq * ((EnvGen.kr(formfreqenv, gate) * fenvsens) + 1);
```

```
amp = (Latch.kr(gate, gate)-1) * vsens + 1;
sig = exciterFunc.value(freq, detune); // this func is user supplied
sig = Formlet.ar(sig,
formHarmRatios.notNil.if({ formfreq * formHarmRatios }, { formfreq }),
attacktime, decaytime, mul:  formHarmAmps ??  { 1 });
 // formlet is a bit volatile, so limit its amplitude
(Limiter.ar(LPF.ar(Mix.ar(sig), ffreq), 0.9, 0.06)
* EnvGen.kr(env, gate, doneAction:2)) !  2
}, [
 \freq
 \amp
  // default func is an audio-rate impulse to provide the base frequency
  // override this with a func for a different exciter
  // your func may have a frequency and detune argument
  // it should output 1 channel only
 ObjectSpec   | fr| Impulse
 \mydetune
 \freq
#[20, 20000, \exp, 0, 1200],
EnvSpec(Env.adsr(0.07, 0.2, 0.8, 0.11)),
EnvSpec(Env(#[0, 0], [1])),
#[0.0001, 1, \exp, 0, 0.01],
#[0.0001, 1, \exp, 0, 0.1],
 \amp
 \amp
 ObjectSpec nil   // arrays by default are nil -- ugenfunc fills in the true default here
 ObjectSpec nil
]);
)

// use the default exciter
p = Patch([\analog, \voxlet], [Patch({ MouseX.kr(20, 20000, 1, 0.1) }), 0.5, nil, nil, Patch({ MouseY.kr(20,
20000, 1, 0.1) }), nil, nil, nil, nil, nil, 1, 0]);
p.play;
// move the mouse to control base freq and formant freq
// watch the volume--amplitude can spike at times in this patch

// when done:
p.free;
```

```
// free the patch ("free" button) and try this to change the exciter
p = Patch([\analog, \voxlet], [Patch({ MouseX.kr(20, 20000, 1, 0.1) }), 0.25, { | fr, detune|  Mix.ar(Saw.ar([fr, fr*detune])) }, nil, Patch({ MouseY.kr(20, 20000, 1, 0.1) }), nil, nil, nil, nil, nil, 1, 0]);
p.play;

p.free;

// now let's add some additiveness to the filters
p = Patch([\analog, \voxlet], [Patch({ MouseX.kr(20, 20000, 1, 0.1) }), 0.25, { | fr, detune|  Mix.ar(Saw.ar([fr, fr*detune])) }, nil, Patch({ MouseY.kr(20, 20000, 1, 0.1) }), nil, nil, nil, nil, nil, 1, 0, (1..6), (1..6).reciprocal]);
p.play;

p.free;
```

ID: 84

OSCBundle networkbundle object

superclass: Object

a bundle object that allows to add preparation messages for async processes.
if this feature is not needed, a list object can be used instead.

add(msg) add an osc message to the bundle

addAll(array) add an array of osc messages to the bundle

addPrepare(msg) add a preparation osc message, which is sent
before the bundle is sent.

send(server, latency) send the bundle to a server. If preparation messages are given, they are sent, the process waits for their reception abd then sends the
bundle.

schedSend(server, clock, quant)
like send, but the sending is synced to a given clock (TempoClock) to the next beat.
quant can be a pair of values: [quant, offset]

```
// example


 // create a new, empty instance
a = OSCBundle.new;

 // a synthdef that needs to be sent to the server, an operation that is asynchronous,
 // i.e.  we have to wait until it is finished.
x = SynthDef("test", { OffsetOut.ar(0, BPF.ar(Impulse.ar(4) * 10, Rand(9000, 1000), 0.1)) });
 // this is why addPrepare is used.
```

```
a.addPrepare(["/d_recv", x.asBytes]);
 // add is used with synchronous operations, like starting synths.
a.add(["/s_new", "test", -1]);

 // the bundle has now the synchronous separated from the asynchronous bundles:
a.oscMessages;
a.preparationMessages;

 // this can be simply sent - the bundle takes care of the server client communication
 // like waiting for the synthdef to be loaded.  the synth is started when the preparation
 // is finished.

  s.boot; // boot the server
a.send(s);

 s.freeAll; // free all nodes on the server

 // scheduled sending:  the synths are started on the next beat.

a.schedSend(s, TempoClock.default, 1);
a.schedSend(s, TempoClock.default, 1);
a.schedSend(s, TempoClock.default, 1);

 s.freeAll; // free all nodes on the server

 // the bundle can contain several preparation messages and messages at a time.
  // the proparationMessages are sent first and only when they are all completed,
 // the other bundles are sent.
  // the bundle can also be reused, if there is no specific allocated buffers/node ids.
```

ID: 85

OSCpathResponder client side responder

superclass: OSCresponder

Register a function to be called upon receiving a command with a specific path.

***new(addr,cmdName,action);**

addr
an instance of NetAddr, usually obtained from your server: server-addr
an address of nil will respond to messages from anywhere.

cmdName
a command path, such as ['\c_set', bus index]

action
a function that will be evaluated when a cmd of that name is received from addr.
args: time, theResponder, message
note that OSCresponderNode evaluates its function in the system process.
in order to access the application process (e.g. for GUI access) use { **...** }**.defer**;

Command paths

OSC commands sometimes include additional parameters to specify the right responder.

For example **/tr** commands, which are generated on the server by the **SendTrig** Ugen create
an OSC packet consisting of: **[/tr, nodeID, triggerID, value]**
This array actually specifies the source of value : **[/tr, nodeID, triggerID].**
We will refer to that array as a **command path.**

To create an OSCpathResponder for a specific trigger, the **cmdName** parameter is simply replaced by
the complete command path.

Path defaults

Any element of the command path array can be set to nil to create a responder that will handle multiple command paths.

For example, setting the commandpath = **['/tr', nil, triggerID]** makes a responder that
responds to /tr messages from any Synth but with a specific triggerID.
/tr messages from one Synth but with any triggerID.

```
//Here is an example:

s.boot;

(
var s, commandpath, response, aSynth, nodeID, triggerID;
s = Server.local;
s.boot;
triggerID = 1;
aSynth = { arg freq = 1, triggerID = 1; SendTrig.kr(SinOsc.kr(freq), triggerID, 666); }.play;
nodeID = aSynth.nodeID;
commandpath = ['/tr', nodeID, triggerID];
response = { arg time, responder, message; message.postln };

o = OSCpathResponder(s.addr, commandpath, response);
o.add;

)

// switch on and off:
o.remove;
o.add;
```

Buffer-getn

ID: 86

OSCresponder client side network responder

Register a function to be called upon receiving a specific command from a specific OSC address.

Examples: see [OSCresponderNode]

***new(addr,cmdName,action);**

addr
the address the responder **receives from** (an instance of NetAddr, e.g. Server.default.addr)

an address of **nil** will respond to messages from anywhere.

cmdName
an OSC command eg. '/done'

action
a function that will be evaluated when a cmd of that name is received from addr.
arguments: **time, theResponder, message, addr**
note that OSCresponder evaluates its function in the system process.
in order to access the application process (e.g. for GUI access) use { **...** }**.defer**;

Note:
A single OSCresponder may be set up for each addr and cmdName combination.
Subsequent registrations will overwrite previous ones. See **[OSCresponderNode]**.

Whenever an OSC message is sent to the SuperCollider application (the language, not the server), either Main-recvOSCmessage or Main-recvOSCbundle is called. There, the messages are forwarded to the OSCresponder class using the OSCresponder-respond class method.

add
add this responder instance to the list of active responders.
The OSCresponder is not active until this is done.

remove
remove and deactivate the OSCresponder

removeWhenDone
remove and deactivate the OSCresponder when action is done.

```
//syntax:
OSCresponder(addr,cmdName,action).add.removeWhenDone;
```

***add(oscResponder)**
add the responder instance

***remove(oscResponder)**
remove the responder instance

***removeAddr(addr)**
remove all OSCresponders for that addr.

ID: 87

OSCresponderNode client side network responder

Register a function to be called upon receiving a specific command from a specific OSC address.
same interface like [**OSCresponder**], but allows**multiple responders to the same command**.

note that OSCresponderNode evaluates its function in the system process.
in order to access the application process (e.g. for GUI access) use { **...** }**.defer**;

Setting up OSCresponder for listening to a remote application

```
// example:  two SuperCollider apps communicating

// application 1:
    NetAddr "127.0.0.1"          // the url should be the one of computer of app 2 (or nil)

o = OSCresponder(n, '/chat', { | t, r, msg|  msg[1].postln }).add;

// application 2:
    NetAddr "127.0.0.1"          // the url should be the one of computer of app 1
          "/chat"  "Hello App 1"

// end application 2:
m.disconnect;

// end application 1:
n.disconnect; o.remove;
```

Sending data from server to client

```
// example from SendTrig

(
```

```
s = Server.local;
s.boot;
s.notify;
)

(
SynthDef "help-SendTrig"
SendTrig.kr(Dust.kr(1.0), 0, 0.9);
}).send(s);

// register to receive this message
a = OSCresponderNode(s.addr, '/tr', { arg time, responder, msg;
[time, responder, msg].postln;
}).add;
b = OSCresponderNode(s.addr, '/tr', { arg time, responder, msg;
"this is another call".postln;
}).add;
)


x = Synth     "help-SendTrig"
a.remove;
b.remove;
x.free;
```

Watching for something specific

```
// end of group message

s.boot;

a = OSCresponderNode(s.addr,'/n_end',{ arg time,responder,msg;
[time, responder, msg].postln;
if(msg[1] == g.nodeID,{
  "g is dead !"
  // g = Group.new;
});
}).add;
```

```
g = Group.new;

g.free;

a.remove;
```

Watching for errors

```
// example from ServerErrorGui in crucial lib

f = OSCresponderNode(s.addr, '/fail', { arg time, responder, msg;
{
var mins,secs;
mins = (time/60).round(1);
secs = (time%60).round(0.1);
if(secs<10,{ secs = "0"++secs.asString; },{ secs=secs.asString;});
// put this on a gui
  //errors.label = msg[1].asString + msg[2].asString + "("++(mins.asString++":"++secs)++")";
  //errors.stringColor = Color.white;
(msg[1].asString + msg[2].asString + "("++(mins.asString++":"++secs)++")").postln;
}.defer
});
f.add;

// cause a failure
Synth("gthhhhpppppp!");

f.remove
```

ID: 88

Score score of timed OSC commands

Score encapsulates a list of timed OSC commands and provides some methods for using it, as well as support for the creation of binary OSC files for non-realtime synthesis. See **Non-Realtime-Synthesis** for more details.

The list should be in the following format, with times in ascending order. Bundles are okay.

```
[
[beat1, [OSCcmd1]],
[beat2, [OSCcmd2], [OSCcmd3]],
...
[beat_n, [OSCcmdn]],
[beatToEndNRT, [\c_set, 0, 0]] // finish
]
```

For NRT synthesis the final event should a dummy event, after which synthesis will cease. It is thus important that this event be timed to allow previous events to complete.

Score scheduling defaults to **TempoClock**. A setting of **TempoClock**.default.tempo = 1 (60 beats per minute), may be used to express score events in seconds if desired.

Class Methods

***new(list)** - returns a new Score object with the supplied list. **list** can be an **Array**, a **List**, or similar object.

***newFromFile(path)** - as ***new**, but reads the list in from a text file. **path** is a string indicating the path of the file. The file must contain a valid SC expression.

***play(list, server)** - as ***new** but immediately plays it. (See also the instance method below.) If no value is supplied for **server** it will play on the default **Server**.

***playFromFile(path, server)** - as ***play**, but reads the list from a file.

***write(list, oscFilePath, clock)** - a convenience method to create a binary OSC file for NRT synthesis. Does not create an instance. **oscFilePath** is a string containing the desired path of the OSC file. Use **clock** as a tempo base. **TempoClock**.default if **clock** is nil.

***writeFromFile(path, oscFilePath, clock)** - as *write but reads the list from a file. Use **clock** as a tempo base. **TempoClock**.default if **clock** is nil.

***recordNRT(list, oscFilePath, outputFilePath, inputFilePath, sampleRate, headerFormat, sampleFormat, options)** - a convenience method to synthesize **list** in non-realtime. This method writes an OSC file to **oscFilePath** (you have to do your own cleanup if desired) and then starts a server app to synthesize it. For details on valid headerFormats and sampleFormats see **SoundFile**. Use **TempoClock**.default as a tempo base. Does not return an instance.
oscFilePath - the path to which the binary OSC file will be written.
outputFilePath - the path of the resultant soundfile.
inputFilePath - an optional path for an input soundfile.
sampleRate - the sample rate at which synthesis will occur.
headerFormat - the header format of the output file. The default is 'AIFF'.
sampleFormat - the sample format of the output file. The default is 'int16'.
options - an instance of **ServerOptions**. If not supplied the options of the default **Server** will be used.

Instance Methods

play(server, clock, quant) - play the list on **server** use **clock** as a tempo base and quantize start time to **quant**. If **server** is nil, then on the default server. **TempoClock**.default if **clock** is nil. now if **quant** is 0.

stop - stop playing.

write(oscFilePath, clock) - create a binary OSC file for NRT synthesis from the list. Use **clock** as a tempo base. **TempoClock**.default if **clock** is nil.

score - get the list.

score_(list) - set the list.

add(bundle) - adds bundle to the list.

sort - sort the score time order.
This is recommended to do **before recordNRT or write** when you are not sure about the packet order

recordNRT(oscFilePath, outputFilePath, inputFilePath, sampleRate, headerFormat, sampleFormat, options) - synthesize the score in non-realtime. For details of the arguments see ***recordNRT** above.

saveToFile(path) - save the score list as a text file to **path.**

NRT Examples:

```
// A sample synthDef
(
SynthDef("helpscore",{ arg freq = 440;
Out.ar(0,
SinOsc.ar(freq, 0, 0.2) * Line.kr(1, 0, 0.5, doneAction:  2)
)
}).load(s);
)

// write a sample file for testing
(
var f, g;
TempoClock.default.tempo = 1;
g = [
[0.1, [\s_new, \helpscore, 1000, 0, 0, \freq, 440]], [0.2, [\s_new, \helpscore, 1001, 0, 0, \freq, 660]],

[0.3, [\s_new, \helpscore, 1002, 0, 0, \freq, 220]],
[1, [\c_set, 0, 0]] // finish
];
    File "score-test" "w"
f.write(g.asCompileString);
f.close;
)
```

```
//convert it to a binary OSC file for use with NRT
Score               "score-test"  "test.osc"
```

From the command line, the file can then be rendered from within the build directory:

```
./scsynth -N test.osc _ test.aif 44100 AIFF int16 -o 1
```

Score also provides methods to do all this more directly:

```
(
var f, o;
g = [
[0.1, [\s_new, \helpscore, 1000, 0, 0, \freq, 440]], [0.2, [\s_new, \helpscore, 1001, 0, 0, \freq, 660],

[\s_new, \helpscore, 1002, 0, 0, \freq, 880]],
[0.3, [\s_new, \helpscore, 1003, 0, 0, \freq, 220]],
[1, [\c_set, 0, 0]] // finish
];
o = ServerOptions.new.numOutputBusChannels = 1; // mono output
Score.recordNRT(g, "help-oscFile", "helpNRT.aiff", options:  o); // synthesize
)
```

Real-time Examples:

```
        // boot the default server

// A sample synthDef
(
SynthDef("helpscore",{ arg freq = 440;
Out.ar(0,
SinOsc.ar(freq, 0, 0.2) * Line.kr(1, 0, 0.5, doneAction:  2)
)
}).load(s);
)

// write a sample file for testing
```

```
(
var f, g;
TempoClock.default.tempo = 1;
g = [
[0.1, [\s_new, \helpscore, 1000, 0, 0, \freq, 440]], [0.2, [\s_new, \helpscore, 1001, 0, 0, \freq, 660],

[\s_new, \helpscore, 1002, 0, 0, \freq, 880]],
[0.3, [\s_new, \helpscore, 1003, 0, 0, \freq, 220]],
[1, [\c_set, 0, 0]] // finish
];
    File "score-test" "w"
f.write(g.asCompileString);
f.close;
)

z = Score.newFromFile("score-test");

// play it on the default server
z.play;

// change the list
(
x = [
[0.0, [ \s_new, \helpscore, 1000, 0, 0,  \freq, 1413 ]],
[0.1, [ \s_new, \helpscore, 1001, 0, 0,  \freq, 712 ]],
[0.2, [ \s_new, \helpscore, 1002, 0, 0,  \freq, 417 ]],
[0.3, [ \s_new, \helpscore, 1003, 0, 0,  \freq, 1238 ]],
[0.4, [ \s_new, \helpscore, 1004, 0, 0,  \freq, 996 ]],
[0.5, [ \s_new, \helpscore, 1005, 0, 0,  \freq, 1320 ]],
[0.6, [ \s_new, \helpscore, 1006, 0, 0,  \freq, 864 ]],
[0.7, [ \s_new, \helpscore, 1007, 0, 0,  \freq, 1033 ]],
[0.8, [ \s_new, \helpscore, 1008, 0, 0,  \freq, 1693 ]],
[0.9, [ \s_new, \helpscore, 1009, 0, 0,  \freq, 410 ]],
[1.0, [ \s_new, \helpscore, 1010, 0, 0,  \freq, 1349 ]],
[1.1, [ \s_new, \helpscore, 1011, 0, 0,  \freq, 1449 ]],
[1.2, [ \s_new, \helpscore, 1012, 0, 0,  \freq, 1603 ]],
[1.3, [ \s_new, \helpscore, 1013, 0, 0,  \freq, 333 ]],
[1.4, [ \s_new, \helpscore, 1014, 0, 0,  \freq, 678 ]],
[1.5, [ \s_new, \helpscore, 1015, 0, 0,  \freq, 503 ]],
[1.6, [ \s_new, \helpscore, 1016, 0, 0,  \freq, 820 ]],
```

```
[1.7, [ \s_new, \helpscore, 1017, 0, 0,  \freq, 1599 ]],
[1.8, [ \s_new, \helpscore, 1018, 0, 0,  \freq, 968 ]],
[1.9, [ \s_new, \helpscore, 1019, 0, 0,  \freq, 1347 ]],
[2.0, [\c_set, 0, 0]] // finish
];

z.score_(x);
)

// play it
z.play;

// play and stop after one second
(
z.play;
SystemClock.sched(1.0, {z.stop;});
)
```

creating Score from a pattern

```
SynthDescLib


// new pattern
(
p = Pbind(

\dur, Prand([0.3, 0.5], inf),
\freq, Prand([200, 300, 500],inf)
);
)

// make a score from the pattern, 4 beats long
z = p.asScore(4.0);

z.score.postcs;
```

```
z.play;
```

rendering a pattern to sound file directly:

```
// render the pattern to aiff (4 beats)

        "asScore-Help.aif"
```

ID: 89

Spec specification of an input datatype

This is an abstract class. Specs specify what kind of input is required or permissible, and what the range of those parameters are.

common subclasses:
AudioSpec
ControlSpec
ScalarSpec

Spec has subclasses which depict different kinds of possible inputs.
This is of interest to functions, to gui interface objects (sliders etc.) and others.

The class Spec itself holds a master Dictionary of common specifications.

ID: 90

StartUp register functions to be evaluated after the startup is finished

Superclass: Object

StartUp allows you to register functions to perform an action after the library has been compiled, and after the startup file has run. This is used for creating SynthDef in the **initClass** function of class files in order to be able to make the synthdef directory customizable by the startup script.

Class Methods

*add(function)

Registers an function to be evaluated after startup is finished.

*remove(function)

Removes a function that was previously registered.

*run

runs the functions in order.

Examples

```
*initClass {
StartUp.add {
  // something to do...
}


}
```

ID: 91

Notes on MIDI support in SuperCollider

Contents

Introduction

SuperCollider's out of the box MIDI support is fairly thorough (although not as complete as you'll find in commercial sequencers). All MIDI devices accessible to CoreMIDI are accessible to SuperCollider.

Note: This document is written from an OSX perspective. The essential behavior of the MIDI interface classes should be the same on other platforms, despite my continual reference to CoreMIDI here.

SuperCollider does not impose much higher-level structure on MIDI functionality. The core classes are little more than hardware abstractions (see also the **[MIDI]** helpfile):

MIDIClient: represents SuperCollider's communications with CoreMIDI
MIDIIn: receives MIDI messages and executes functions in response to those messages
MIDIOut: sends MIDI messages out to a specific port and channel
MIDIEndPoint: a client-side representation of a CoreMIDI device, containing three variables (name, device and uid, which is a unique identifier assigned by the system)

In most cases, each physical MIDI connection (pair of in/out jacks on the MIDI interface) has one MIDIEndPoint object to represent it in the client.

Receiving MIDI input: MIDIIn

The MIDIIn class provides two ways to receive MIDI input: MIDI response functions, and routines that wait for MIDI events.

1. MIDI response functions

MIDIIn has a number of class variables that are evaluated when a MIDI event comes in. Technical details on each function can be found in the MIDIIn help file.

noteOn
noteOff
control
bend
touch
polyTouch
program
sysex
sysrt
smpte

To assign a response to a particular kind of MIDI message, assign a function to the class variable:

```
MIDIIn.connect;
MIDIIn.noteOn = { | port, chan, note, vel|  [port, chan, note, vel].postln };
MIDIIn          nil   // stop responding
```

MIDIIn provides the responding functions with all the information coming in from CoreMIDI:

source (src): corresponds to the uid of the MIDIEndPont from which the message is coming.
channel (chan): integer 0-15 representing the channel bits of the MIDI status byte

... with subsequent arguments representing the data bytes. The MIDIIn help file details all the supported messages along with the arguments of the responding function for the message.

Because these are class variables, you can have only one function assigned at one time. A common usage is to assign a function that looks up responses in a collection. For example, you could have a separate set of response functions for each channel.

```
noteOn = Array.fill(16, IdentityDictionary.new);
```

```
MIDIIn.noteOn = { | port, chan, num, vel|   noteOn[chan].do(_.value(port, chan, num, vel)) };

   // this function will respond only on channel 0
noteOn[0].put(\postNoteOn, { | port, chan, num, vel|  [port, chan, note, vel].postln });
noteOn[0].removeAt(\postNoteOn);  // stop responding
```

The advantage of this approach over using "if" or "case" statements in the response function is that you can add and remove responses without having to change the MIDIIn function. The MIDIIn function can serve as a "hook" into another structure that distributes the MIDI events to the real responders.

Third-party frameworks exist to handle this bookkeeping automatically. See the "Third party libraries" section at the bottom of this file.

2. Routines that wait for MIDI events

As of December 2004, there is an alternate technique to supply multiple responses for the same MIDI event type. This routine waits for a MIDI event, then posts information about the event. After your routine receives the MIDI event, it can take any other action you desire.

```
    Routine
var event;
loop {
event = MIDIIn.waitNoteOn;
[event.status, event.b, event.c].postln;
}
}).play;

        // stop responding
```

Supported MIDI event waiting methods are:

waitNoteOn
waitNoteOff
waitControl
waitBend
waitTouch
waitPoly

You can have multiple routines assigned to the same MIDI event type. The MIDI wait method lets you specify conditions for the routine to fire based on the arguments of the corresponding MIDI responder function:

```
event = MIDIIn.waitNoteOn(nil, [2, 7], (0, 2..126), { | vel|  vel > 50 });
```

This would respond to note on messages from any port, channels 2 and 7 only, even numbered note numbers only, and only velocity values greater than 50.

Use caution when creating a large number of MIDI response routines with very specific conditions. For each incoming MIDI event, SuperCollider will iterate over the entire list for that event type, which incurs a CPU cost. If you have 500 MIDI controller routines, and an incoming event should trigger only 2, all 500 sets of conditions have to be evaluated.

In that case it may be more efficient to create a smaller number of routines and evaluate some of the conditions inside routines, either using branching statements or by looking up functions inside collections.

Playing notes on your MIDI keyboard

The technical problem is that every note on needs to save its synth object so that the note off message can end the right server-side node.

```
s.boot;

(
var notes, on, off;

MIDIIn.connect;

        Array                // array has one slot per possible MIDI note

on = Routine({
var event, newNode;
loop {
event = MIDIIn.waitNoteOn; // all note-on events
   // play the note
newNode = Synth(\default, [\freq, event.b.midicps,
   \amp                        // 0.00315 approx.  == 1 / 127 * 0.4
```

```
notes.put(event.b, newNode); // save it to free later
}
}).play;

off = Routine({
var event;
loop {
event = MIDIIn.waitNoteOff;
   // look up the node currently playing on this slot, and release it
notes[event.b].set(\gate, 0);
}
}).play;

q = { on.stop; off.stop; };
)

// when done:
q.value;
```

The MIDIIn help file contains a more elaborate example.

SuperCollider does not have a built-in class to handle this automatically. However, dewdrop_lib, one of the third party libraries mentioned below, includes a small suite of classes designed for exactly this purpose. Users interested in this functionality may wish to examine that library.

Sending MIDI out

See the **[MIDIOut]** helpfile. Unlike MIDIIn, with MIDIOut you create an instance of the MIDIOut class with a port and uid. You can have multiple MIDIOut objects to send MIDI to different physical devices.

Many users have reported timing issues with MIDIOut. When the CPU is busy, especially during graphics updates, outgoing MIDI messages may be delayed. Use with caution in a performance situation.

MIDI synchronization

MIDI synchronization may be performed using MIDIIn's sysrt or smpte response functions. It's up to the user to implement the desired kind of synchronization.

For sysrt, external MIDI clocks output 24 pulses per quarter note. The responder should count the incoming pulses and multiply the rhythmic value into 24 to determine how many pulses to wait:

0.25 wait 6 pulses (16th note)
0.5 wait 12 pulses (8th note)
2 wait 48 pulses (half note)

dewdrop_lib (third party library) includes a class, MIDISyncClock, that receives MIDI clock messages and allows events to be scheduled to keep time with an external MIDI device. See the **[MIDISyncClock]** helpfile for details.

There are significant limitations, discussed in the helpfile. This is not really a fully supported class, but it's there for users who are desperate for the functionality.

Third party libraries

The crucial library (included in the main distribution) includes a couple of classes (NoteOnResponder, NoteOffResponder, CCResponder) that simplify the use of multiple responders when all ports and channels should respond identically. Multichannel MIDI applications are not possible using these classes.

dewdrop_lib is a third party library providing a number of useful performance features, available from <#0000ffhttp://www.dewdrop-world.net>. The library provides a user-extensible framework of MIDI responder classes designed for multiport, multichannel applications.

Among its features:

- user-extensible: simple functions may be used, and frequently-needed responses can be written into classes that inherit from the framework (see **[BasicMIDISocket]** and **[BasicMIDIControl]**)

- easy to use classes for playing MIDI notes and assigning MIDI controllers to synthesis parameters

- a user-configurable array of MIDI controller numbers, to simplify assignment of events to hardware controllers

The framework is not part of the main distribution. Interested users need to download the tarball from the website above and follow the installation instructions.

5 Core

5.1 Kernel

ID: 92

Class

superclass: Object

A Class describes the structure and implementation of a set objects which are its instances.

Utilities

browse

Open a graphical browser for this Class. (OSX only). Shows methods, arguments, variables, subclasses, and has buttons for navigating to the superclass, source, helpfile, cvs, etc.

findMethod(methodName)

Find the Method referred to by name. If not found, return nil.

findRespondingMethodFor(methodName)

As above, but climb the class tree to see if the method is inherited from a superclass. If not found, return nil.

dumpAllMethods

Post all instance methods which instances of this class responde too, including inherited ones. this.class.dumpAllMethods will post all class methods which this class responds to.

dumpByteCodes(methodName)

Dump the byte codes of the named method.

dumpClassSubtree

Post the tree of all Classes that inherit from this class.

dumpInterface

Post all the methods defined by this Class and their arguments.

dumpFullInterface

Post all the class and instance methods that this class responds to (i.e. those defined in this class and those inherited by it).

openHelpFile

Opens the help file for this Class if it exists.

helpFilePath

Returns the path of this Class's helpfile as a String.

helpFileForMethod(methodSymbol)

Opens the helpfile for the class in which the responding method is implemented.

```
Array                   'select'   // This will open the Collection helpfile
```

Conversion

asClass

Return this.

asString

Return the name of the class as a String.

Accessing

name

A Symbol that is the name of the class.

nextclass

The next class in a linked list of all classes.

superclass

The Class from which this class directly inherits.

superclasses

An Array of this class's superclasses, going back to Object.

subclasses

An Array of the direct subclasses of this.

allSubclasses

An Array of all subclasses of this.

methods

An Array of the methods of this class.

instVarNames

An Array of the names of the instance variables for this class.

classVarNames

An Array of the names of the class variables for this class.

iprototype

An Array of the initial values of instance variables.

cprototype

An Array of the initial values of class variables.

filenameSymbol

A Symbol which is a path to the file which defines the Class.

ID: 93

Frame

superclass: Object

Frames are used to contain the arguments, variables and other information for active Functions.

There are no instance variables or methods.

Since Frames are often created on the stack, it is too dangerous to allow access to them. Dangling pointers could result.

Frame instances are inaccessible to the user.

For error handling routines, the relevant information from a Frame can be transferred into a DebugFrame object which can safely be inspected.

```
this.getBackTrace.inspect
```

ID: 94

Function

superclass: AbstractFunction

A Function is a reference to a [FunctionDef] and its defining context [Frame]. When a FunctionDef is encountered in your code it is pushed on the stack as a Function. A Function can be evaluated by using the 'value' method. See the [Functions] help file for a basic introduction.

Because it inherits from [AbstractFunction], Functions can respond to math operations by creating a new Function. For example:

```
(
var a, b, c;
a = { [100, 200, 300].choose }; // a Function
b = { 10.rand + 1 }; // another Function
            // c is a Function.
                // evaluate c and print the result
)
```

See [AbstractFunction] for function composition examples.

Accessing

def

Get the FunctionDef definition of the Function.

Evaluation

value(...args)

Evaluates the FunctionDef referred to by the Function. The Function is passed the args given.

```
{ arg a, b; (a * b).postln }.value(3, 10);
```

valueArray(..args..)

Evaluates the FunctionDef referred to by the Function. If the last argument is an Array or List, then it is unpacked and appended to the other arguments (if any) to the Function. If the last argument is not an Array or List then this is the same as the 'value' method.

```
{ arg a, b, c; ((a * b) + c).postln }.valueArray([3, 10, 7]);

{ arg a, b, c, d; [a, b, c, d].postln }.valueArray([1, 2, 3]);

{ arg a, b, c, d; [a, b, c, d].postln }.valueArray(9, [1, 2, 3]);

{ arg a, b, c, d; [a, b, c, d].postln }.valueArray(9, 10, [1, 2, 3]);
```

valueEnvir(...args)

As value above. Unsupplied argument names are looked up in the current **Environment**.

```
(
Environment
a = 3;
b = 10;
{ arg a, b; (a * b).postln }.valueEnvir;
});
)
```

valueArrayEnvir(..args..)

Evaluates the FunctionDef referred to by the Function. If the last argument is an Array or List, then it is unpacked and appended to the other arguments (if any) to the Function. If the last argument is not an Array or List then this is the same as the 'value' method. Unsupplied argument names are looked up in the current **Environment**.

loop

Repeat this function. Useful with **Task** and Clocks.

```
t = Task({ { "I'm loopy".postln; 1.wait;}.loop });
t.start;
t.stop;
```

defer(delta)

Delay the evaluation of this Function by **delta** in seconds. Uses **AppClock**.

```
  "2 seconds have passed."
```

dup(n)

Return an Array consisting of the results of n evaluations of this Function.

```
x = { 4.rand; }.dup(4);
x.postln;
```

! n

equivalent to dup(n)

```
x = { 4.rand } !  4;
x.postln;
```

sum(n)

return the sum of n values produced.

```
{ 4.rand }.sum(8);
```

bench(print)

Returns the amount of time this function takes to evaluate. **print** is a boolean indicating whether the result is posted. The default is true.

```
{ 1000000.do({ 1.0.rand }); }.bench;
```

fork(clock, quant, stackSize)

Returns a Routine using the receiver as it's function, and plays it in a **TempoClock**.

```
{ 4.do({ "Threadin...".postln; 1.wait;}) }.fork;
```

block

Break from a loop. Calls the receiver with an argument which is a function that returns from the method block. To exit the loop, call .value on the function passed in. You can pass a value to this function and that value will be returned from the block method.

```
block {| break|
100.do {| i|
i.postln;
if (i == 7) { break.value(999) }
};
}
```

thunk

Return a Thunk, which is an unevaluated value that can be used in calculations

```
x = thunk { 4.rand };
x.value;
x.value;
```

flop

Return a function that, when evaluated with nested arguments, does multichannel expansion by evaluting the receiver function for each channel.

```
f = { | a, b|  if(a > 0) { a + b } { -inf } }.flop;
f.value([-1, 2, 1, -3.0], [10, 1000]);
f.value(2, 3);
```

flopEnvir

like flop, but implements an environment argument passing (valueEnvir).
Less efficient in generation than flop, but not a big difference in evaluation.

```
f = { | a|  if(a > 0) { a + 1 } { -inf } }.envirFlop;
e = (a:  [20, 40]);
e.use { f.value }
```

case(cases)

Function implements a **case** method which allows for conditional evaluation with multiple cases. Since the receiver represents the first case this can be simply written as pairs of test functions and corresponding functions to be evaluated if true. Unlike Object-switch, this is inlined and is therefore just as efficient as nested if statements.

```
(
var i, x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
i = z.choose;
x = case
{ i == 1 }   { \no }
{ i == 1.1 } { \wrong }
{ i == 1.3 } { \wrong }
{ i == 1.5 } { \wrong }
{ i == 2 }   { \wrong }
{ i == 0 }   { \true };
x.postln;
)
```

Exception Handling

For the following two methods a return ^ inside of the receiver itself cannot be caught. Returns in methods called by the receiver are OK.

try(handler)

Executes the receiver. If an exception is thrown the catch function **handler** is executed with the error as an argument. **handler** itself can rethrow the error if desired.

protect(handler)

Executes the receiver. The cleanup function **handler** is executed with an error as an argument, or nil if there was no error. The error continues to be in effect.

Examples:

```
// no exception handler
value { 8.zorg; \didnt_continue.postln; }

try { 8.zorg } {| error|  error.postln; \cleanup.postln; }; \continued.postln;

protect { 8.zorg } {| error|  error.postln; }; \didnt_continue.postln;

try { 123.postln; 456.throw; 789.postln } {| error|  [\catch, error].postln };

try { 123.postln; 789.postln } {| error|  [\catch, error].postln };

try { 123.postln; nil.throw; 789.postln } {| error|  [\catch, error].postln };


protect { 123.postln; 456.throw; 789.postln } {| error|  [\onExit, error].postln };

protect { 123.postln; 789.postln } {| error|  [\onExit, error].postln };

(
try {
```

```
protect { 123.postln; 456.throw; 789.postln } {| error|  [\onExit, error].postln };
} {| error|  [\catch, error].postln };
)

value { 123.postln; 456.throw; 789.postln }

value { 123.postln; Error("what happened?").throw; 789.postln }

(
    \aaa  \bbb  \ccc  \ddd
a[1].postln;
a[\x].postln;
a[2].postln;
)

(
try {
 a = [\aaa  \bbb  \ccc  \ddd
a[1].postln;
a[\x].postln;
a[2].postln;
} {| error|  \caught.postln; error.dump }
)

(
try {
 a = [\aaa  \bbb  \ccc  \ddd
a[1].postln;
a[\x].postln;
a[2].postln;
} {| error|  \caught.postln; error.dump; error.throw }
)

(
protect {
 a = [\aaa  \bbb  \ccc  \ddd
a[1].postln;
a[\x].postln;
a[2].postln;
} {| error|  \caught.postln; error.dump }
```

)

Audio

play(target, outbus, fadetime, addAction)

This is probably the simplest way to get audio in SC3. It wraps the Function in a **SynthDef** (adding an **Out** ugen if needed), creates and starts a new **Synth** with it, and returns the Synth object. A **Linen** is also added to avoid clicks, which is configured to allow the resulting Synth to have its \gate argument set, or to respond to a release message. Args in the function become args in the resulting def.

target - a Node, Server, or Nil. A Server will be converted to the default group of that server. Nil will be converted to the default group of the default Server.
outbus - the output bus to play the audio out on. This is equivalent to Out.ar(outbus, theoutput). The default is 0.
fadeTime - a fadein time. The default is 0.02 seconds, which is just enough to avoid a click. This will also be the fadeout time for a release if you do not specify.
addAction - see **Synth** for a list of valid addActions. The default is \addToHead**.**

```
x = { arg freq = 440; SinOsc.ar(freq, 0, 0.3) }.play; // this returns a Synth object;
 x.set(\freq        // note you can set the freq argument
 x.defName; // the name of the resulting SynthDef (derived from the Functions hash value)
 x.release(4); // fadeout over 4 seconds
```

Many of the examples in SC3 make use of the Function.play syntax. Note that reusing such code in a **SynthDef** requires the addition of an **Out** ugen.

```
  // the following two lines produce equivalent results
{ SinOsc.ar(440, 0, 0.3) }.play(fadeTime:  0.0);
SynthDef("help-FuncPlay", { Out.ar(0, SinOsc.ar(440, 0, 0.3))}).play;
```

Function.play is often more convienent than SynthDef.play, particularly for short examples and quick testing. The latter does have some additional options, such as lagtimes for controls, etc. Where reuse and maximum flexibility are of greater importance, SynthDef and its various methods are usually the better choice.

scope(numChannels, outbus, fadeTime, bufsize, zoom)

As **play** above, but plays it on the internal **Server**, and calls Server-scope to open a scope window in which to view the output. Currently only works on OSX.

numChannels - The number of channels to display in the scope window, starting from **outbus**. The default is 2.
outbus - The output bus to play the audio out on. This is equivalent to Out.ar(outbus, theoutput). The default is 0.
fadeTime - A fadein time. The default is 0.02 seconds, which is just enough to avoid a click.
bufsize - The size of the buffer for the ScopeView. The default is 4096.
zoom - A zoom value for the scope's X axis. Larger values show more. The default is 1.

```
{ FSinOsc.ar(440, 0, 0.3) }.scope(1)
```

plot(duration, server, bounds)

Calculates **duration** in seconds worth of the output of this function, and plots it in a GUI window. Currently only works on OSX. Unlike **play** and **scope** it will not work with explicit Out Ugens, so your function should return a UGen or an **Array** of them. The plot will be calculated in realtime.

duration - The duration of the function to plot in seconds. The default is 0.01.
server - The **Server** on which to calculate the plot. This must be running on your local machine, but does not need to be the internal server. If nil the default server will be used.
bounds - An instance of **Rect** or **Point** indicating the bounds of the plot window.

```
{ SinOsc.ar(440) }.plot(0.01, bounds:  SCWindow.screenBounds);
```

```
{ {| i|  SinOsc.ar(1 + i)}.dup(7) }.plot(1);
```

Conversion

asSynthDef(rates, prependArgs, outClass, fadetime)

Returns a SynthDef based on this Function, adding a **Linen** and an **Out** ugen if needed.

rates - An Array of rates and lagtimes for the function's arguments (see **SynthDef** for more details).
outClass - The class of the output ugen as a symbol. The default is \Out.
fadeTime - a fadein time. The default is 0.

asDefName

Performs asSynthDef (see above), sends the resulting def to the local server and returns the SynthDefs name. This is asynchronous.

```
x = { SinOsc.ar(440, 0, 0.3) }.asDefName; // this must complete first
y = Synth.new(x);
```

asRoutine

Returns a **Routine** using this as its func argument.

ID: 95

FunctionDef

superclass: Object

FunctionDefs contain code which can be executed from a Function.

Accessing

Even though it is possible to change the values in the various arrays that define the FunctionDef,
you should not do it, unless you like to crash.

code

Get the byte code array.

prototypeFrame

Get the array of default values for argument and temporary variables.

context

Get the enclosing FunctionDef or Method.

argNames

Get the Array of Symbols of the argument names.

varNames

Get the Array of Symbols of the local variable names.

Utilities

dumpByteCodes

"Disassemble" and post the FunctionDef's byte code instructions to the text window.

ID: 96

FunctionList multiple function

superclass: AbstractFunction

A **FunctionList** is a function that composes multiple functions into one. This allows allow to deal transparently with several functions as if they were one and to append new functions at a later point.

See the [Functions] help file for a basic introduction.

***new(functions)**
create a new instance. *functions* is an array of functions or objects

addFunc(function, function ..)
This message is used to be able to add to **an Object**, to a **Function**, or to a **FunctionList**.
nil.addFunc returns a function, if only one function is passed in the argument.
function.addFunc then returns a FunctionList.

removeFunc(function), remove a function from the list.
Object returns **nil** when appropriate.

```
// example

a = nil;
a = a.addFunc { | x="", y=""|  "this % is an % example\n".postf(x, y); 1 };
a.postln;
a = a.addFunc { | x="", y=""|  "there is no % that is %\n".postf(x, y); 2 };
a.value;
        "text"  "extraordinary well written"
a.valueArray(["x", "y"]);
```

```
(
().use {
x = "array";
  y = "ominous"
a.valueEnvir;
a.valueEnvir("list");
}
)


// removing a function
x = { "removal test".postln };
a.addFunc(x);
a.value;
a = a.removeFunc(x);
a.value;

// mathematics
a = x.addFunc({ 1.0.rand }).addFunc({ [0, 1].choose });
a = a.squared.linexp(0, 1, 1.0, 500);

a.value;


// compatibility with function multichannel expansion
a = nil;
a = a.addFunc { | x=0|  if(x > 0) { 7 } { 1000.rand } };
a = a.addFunc { | x=0|  if(x < 0) { 17 } { -1000.rand } };
a.value

a = a.flop;
a.value
a.value([-1, 1])



// typical usage (similar in action functions for views)

d = Document.current;
d.keyDownAction = { "You touched the keyboard.".postln };
```

```
d.keyDownAction = d.keyDownAction.addFunc {:x, x<-(1..), ::  "already % times\n\n".postf(x) };
d.keyDownAction = nil;

// even if you don't know if there is already an action defined
// one can add one.

(
d.keyDownAction = nil;
d.keyDownAction = d.keyDownAction.addFunc {:x, x<-(1..), ::  "already % times\n\n".postf(x) };

);

d.keyDownAction = nil;
```

ID: 97

*initClass

When SuperCollider starts up, just after it has compiled the library, it initializes all the classes from Object down, a depth first traversal of subclasses.

In this method, any class that requires it may initalize classvars or other resources.

In some cases you will require another class to be initalized before you can initalize your own. You may depend on its resources (a classvar). This can be accomplished by:

```
YourClass
*initClass {
Class.initClass(OtherClass);

..

    //

..
}

..

}
```

Each class will be inited once, and the OtherClass will have all of its subclasses inited before the method returns.

-felix

ID: 98

Interpreter

superclass: Object

The interpreter defines a context in which interactive commands are compiled and executed.

In the interpreter, this refers to the interpreter itself, e.g.:

```
this.postln
```

Accessing

The interpreter defines instance variables 'a' through 'z' which are always available in the interpreter. By convention, the variable 's' is used to hold the default **Server**. Assigning another value to 's' may cause some of the examples in the documentation to fail.

clearAll

set the values of the variables 'a' through 'z' to nil.

```
(
x = 123;
x.postln;
this.clearAll;
x.postln;
)
```

Compile & Interpret

interpret(aString)

Compile and execute a String.

```
this.interpret("(123 + 4000).postln");
```

interpretPrint(aString)

Compile and execute a String, printing the result.

```
this.interpretPrint("123 + 4000");
```

compile(aString)

Compile a String and return a Function.

```
(
z = this.compile("(123 + 4000).postln");
z.postln;
z.value;
)
```

compileFile(pathName)

Reads the file at pathName, compiles it and returns a Function.
The file must contain a valid SuperCollider expression, naturally.
This will not compile class definitions, only expressions.

executeFile(pathName)

Reads the file at pathName, compiles it and executes it, returning the result.
The file must contain a valid SuperCollider expression, naturally.
This will not compile class definitions, only expressions.

ID: 99

Process

superclass: Object

A Process is the runtime environment for the virtual machine and interpreter.
It has a subclass named Main which is where you should override the methods
of Process. There are two methods of interest. One is named 'startUp' and is
called after the class library has been compiled. The other is named 'run' and
is called when the user chooses the Run menu command.

startUp

called after the class library has been compiled. Override this in class Main to do whatever you want.

run

called when the user chooses the Run menu command. Override this in class Main to do whatever you want.

ID: 100

Method

superclass: Function

A Method is code that is a part of the set of operations upon instances of a Class.

Accessing

ownerClass

The Class for which the method is part of the implementation.

name

A Symbol which is the name of the Method.

primitiveName

A Symbol which contains the name of the primitive function that implements the Method, if
there is one.

ID: 101

Process

superclass: Object

A Process is the runtime environment for the virtual machine and interpreter.
It has a subclass named Main which is where you should override the methods
of Process. There are two methods of interest. One is named 'startUp' and is
called after the class library has been compiled. The other is named 'run' and
is called when the user chooses the Run menu command.

startUp

called after the class library has been compiled. Override this in class Main to do whatever you want.

run

called when the user chooses the Run menu command. Override this in class Main to do whatever you want.

***tailCallOptimize**

Returns a Boolean indicating whether tail call optimization is on. The default is on.

***tailCallOptimize_(aBoolean)**

Turns tail call optimization on or off. Setting this to false can help with debugging by including intermediate levels in an error backtrace.

ID: 102

random generator seed

Every Thread in sclang has a (pseudo-) random number generator that is responsible for all randomization within this thread. Each randgen has its own seed (starting point) from which the series of values is generated. This seed can be set and after that, the randgen (being strictly deterministic) produces exactly the same numbers again.

In order to save diskspace, you can reproduce any sequence of randomized data just by one integer number that you can write down in your notebook..

see also: [**RandSeed**][**Pseed**]

```
//every thread, also a Routine, has a random generator seed:
(
    Routine
loop({#[1,2,3,4,5].choose.yield })
});
r.randSeed = 1923;
)

//using the routine to fill an array
Array.fill(7, r);

//setting the random generator seed back to our initial seed
r.randSeed = 1923;

//causes this array to be identical
Array.fill(7, r);
```

Inheriting Seeds

Also it is possible to set the seed of the running thread that all threads started within will inherit.

```
thisThread.randSeed = 1923;

//create a function that returns a routine

r = { Routine({
loop({#[1,2,3,4,5].choose.yield })
}) };

Array.fill(7, r.value);

//reset the seed
thisThread.randSeed = 1923;

Array.fill(7, r.value);

//use the seed to completely reproduce a sound:
(
SynthDef("help-randomSeed", { arg out=0, freq=440;
Out.ar(out,
Line.kr(1, 0, 0.3, doneAction:2) *
Resonz.ar(
Dust2.ar([10, 10], 270) + WhiteNoise.ar(4),
freq, 0.01)
)
}).send(s);

SynthDef("help-setRandomSeed", { arg seed=1956, tbus=0.0;
RandSeed.kr(tbus, seed);
}).send(s);
)

//run a patch
(
```

```
    Synth "help-setRandomSeed"
    Routine
loop({
Synth("help-randomSeed", [\freq, rrand(440, 700)]);
0.25.wait;
})
}).play;
)

//make a reset task

(
d = 1250;//seed
t = Task({
loop({
x.set(\seed, d, \tbus, 1.0); r.randSeed = d;
0.1.wait;
x.set(\tbus, 0.0);
1.9.wait;
})
});
)

//sound starts to loop
t.start;

          //different loop
d = 1925;


//sound is just like random again, not interested in anything..
t.stop;
```

ID: 103

RawPointer

superclass: Object

A class used to hold raw pointers from the host environment.
No instance variables, no methods.

ID: 104

Routine

Superclass: Thread

Routines are functions that can return in the middle and then resume where they left off when called again. Routines can be used to implement co-routines as found in Scheme and some other languages.
Routines are useful for writing things that behave like Streams.
Routines inherit behaviour for math operations and filtering from **[Stream]**.

*new(func, stackSize, seed)

Creates a Routine instance with the given function.
The stackSize and random seed may be overridden if desired.

```
(
a = Routine.new({ 1.yield; 2.yield; });
a.next.postln;
a.next.postln;
a.next.postln;
)
```

value(inval)
resume(inval)
next(inval)

These are all synonyms for the same method.

The Routine function is either started if it has not been called yet, or it is resumed from where it left off. The argument inval is passed as the argument to the Routine function if it is being started, or as the result of the yield method if it is being resumed from a yield. The result of the method will be what the Routine yields.

There are basically 2 conditions for a Routine: one is when the routine starts. The other case is
that the routine continues after it has yielded.

When the routine starts (by calling the above methods), you are passing in a first inval. This inval is accessible as the routine function argument:

```
(
Routine { arg inval;
inval.postln;
        "hello routine"
)
```

When there is a yield in the routine, the next time you call *next* (or synonym), the routine continues
from there, and you get a chance to pass in a value from the outside. To access that value within the
continuing routine, you have to assign the **result of the yield call** to a variable:

```
(
r = Routine { arg inval;
var valuePassedInbyYield;
inval.postln;
valuePassedInbyYield = 123.yield;
valuePassedInbyYield.postln;

}
)

        "hello routine"
        "goodbye world"
```

Typically the name *inval* (or inevent) is reused, instead of declaring a variable like "valuePassedInbyYield":

```
(
r = Routine { arg inval;
inval.postln;
inval = 123.yield;
inval.postln;
}
)
```

```
"hello routine"
"goodbye world"
```

Typically a routine uses a multiple yield, in which the inval is reassigned repeatedly:

```
(
r = Routine { arg inval;
inval.postln;
5.do { arg i;
inval = (i + 10).yield;
inval.postln;
}
}
)
(
5.do {
r.value("hello routine").postln;
}
)
```

reset

Causes the Routine to start from the beginning next time it is called.
A Routine cannot reset itself except by calling the yieldAndReset method.

See also in class **Object :**
yield(outval)
yieldAndReset(outval)
alwaysYield(outval)

If a Routine's function returns then it will always yield nil until reset.

play(clock, quant)

clock: a Clock, **TempoClock** by default
quant: either a number**n** (quantize to **n** beats)
or an array**[n, m]** (quantize to **n** beats, with offset **m**)

In the SuperCollider application, a Routine can be played using a Clock, as can any Stream.
every time the Routine yields, it should do so with a float, the clock will interpret that, usually
pausing for that many seconds, and then resume the routine, passing it it the clock's current time.

Accessible instance variables

Routine inherits from Thread, which allows access to some of its state:

beats
return the elapsed beats (logical time) of the routine. The beats do not proceed when the routine is
not playing.

seconds
return the elapsed seconds (logical time) of the routine. The seconds do not proceed when the routine is
not playing, it is the converted beat value.

clock
return the thread's clock. If it hs not played, it is the SystemClock.

```
(
r = Routine { arg inval;
loop {
  // thisThread refers to the routine.
    postf("beats:  % seconds:  % time:  % \n"
```

```
thisThread.beats, thisThread.seconds, Main.elapsedTime
);
1.0.yield;

}
}.play;
)

r.stop;
r.beats;
r.seconds;
r.clock;
```

Routine Example:

```
(
var r, outval;
r = Routine.new({ arg inval;
("->inval was " ++ inval).postln;
inval = 1.yield;
("->inval was " ++ inval).postln;
inval = 2.yield;
("->inval was " ++ inval).postln;
inval = 99.yield;
});

outval = r.next('a');
("<-outval was " ++ outval).postln;
outval = r.next('b');
("<-outval was " ++ outval).postln;
r.reset; "reset".postln;
outval = r.next('c');
("<-outval was " ++ outval).postln;
outval = r.next('d');
("<-outval was " ++ outval).postln;
outval = r.next('e');
("<-outval was " ++ outval).postln;
```

```
outval = r.next('f');
("<-outval was " ++ outval).postln;
)



(
var r;
r = Routine.new({
10.do({ arg a;
a.postln;
  // Often you might see Wait being used to pause a routine
  // This waits for one second between each number
1.wait;
});
 // Wait half second before saying we're done
0.5.wait;
"done".postln;
});

SystemClock
)
```

ID: 105

Thread

state

0 = not started

3 = ?

7 = running

8 = stopped

5.2 Miscellanea

ID: 106

Boolean

superclass: Object

Boolean is an abstract class whose instances represent a logical value.
Boolean is the superclass of True and False which are the concrete realizations.

xor(aBoolean)

Answers the exclusive or of the receiver and another Boolean.

and(function)
and: function

If the receiver is true then answer the evaluation of function.
If the receiver is false then function is not evaluated and the message answers false.

or(function)
or: function

If the receiver is false then answer the evaluation of function.
If the receiver is true then function is not evaluated and the message answers true.

&& aBoolean

Answers true if the receiver is true and aBoolean is true.

| | aBoolean

Answers true if either the receiver is true or aBoolean is true.

not

Answers true if the receiver is false, and false if the receiver is true.

if(trueFunc, falseFunc)

If the receiver is true, answer the evaluation of the trueFunc. If the receiver is false,

answer the evaluation of the falseFunc.

binaryValue

Answer 1 if the receiver is true, and 0 if the receiver is false.

ID: 107

Char ascii characters

Chars may be written as literals using the $sign. For example $a, $b, $c.
See section [01 Literals]

Chars may be created from Integers using the Integer methods asAscii and asDigit.

Conversion

ascii

answers the integer ascii value of a Char.

digit

answers an integer value from 0 to 9 for chars $0 to $9, and values 10 to 35 for chars $a to $z
or $A to $Z.

toUpper

answers the upper case version of a char. Nonalphabetic chars return themselves.

toLower

answers a lower case version of a char. Nonalphabetic chars return themselves.

Testing

isAlpha

answers whether the char is an alphabetic character.

isAlphaNum

answers whether the char is an alphabetic or numeric character.

isPrint

answers whether the char is printable.

isPunct

answers whether the char is a punctuation character

isSpace

answers whether the char is white space.

isDecDigit

answers whether the char is a decimal digit $0 to $9.

isFileSafe

answers whether the char is safe for use as in a filename.
excludes the path separators / and :

```
for(0,255,{ arg i;
var a;
[i,a = i.asAscii,a.isAlphaNum,a.isPrint,a.isPunct,a.isControl].postln;
});
```

ID: 108

False

see [Boolean]

ID: 109

if

if(boolean, trueFunc, falseFunc)

see also: [Control-Structures]

the functions will be inlined, which plucks the code from the functions and uses a more efficient jump statement.

```
{
if( 6 == 9,{
"hello".postln;
},{
"hello".postln;
})
}.def.dumpByteCodes

BYTECODES: (18)
0   FE 06    PushPosInt 6
2   FE 09    PushPosInt 9
  4   E6      SendSpecialBinaryArithMsg'=='
5   F8 00 06 JumpIfFalse 6  (14)
8   42       PushLiteral "hello"
9   A1 00    SendMsg 'postln'
11   FC 00 03 JumpFwd 3  (17)
14   41       PushLiteral "hello"
15   A1 00    SendMsg 'postln'
17   F2       BlockReturn
  FunctionDef          FunctionDef
```

failure to inline due to variable declarations

{

```
if( 6 == 9,{
var notHere;
"hello".postln;
},{
"hello".postln;
})

}.def.dumpByteCodes

WARNING: FunctionDef contains variable declarations and so will not be inlined.
    in file 'selected text'
line 4 char 14 :
var notHere;●
"hello".postln;
-----------------------------------
BYTECODES: (12)
0   FE 06    PushPosInt 6
2   FE 09    PushPosInt 9
  4   E6      SendSpecialBinaryArithMsg'=='
5   04 00    PushLiteralX instance of FunctionDef in closed FunctionDef
7   04 01    PushLiteralX instance of FunctionDef in closed FunctionDef
  9   C3 B    SendSpecialMsg'if'
11   F2       BlockReturn
  FunctionDef          FunctionDef
```

```
{
if( 6 == 9,{
"hello".postln;
},{
"hello".postln;
})
}.def.dumpByteCodes

BYTECODES: (18)
0   FE 06    PushPosInt 6
```

```
2   FE 09    PushPosInt 9
  4   E6      SendSpecialBinaryArithMsg'=='
5   F8 00 06 JumpIfFalse 6  (14)
8   42       PushLiteral "hello"
9   A1 00    SendMsg 'postln'
11   FC 00 03 JumpFwd 3  (17)
14   41       PushLiteral "hello"
15   A1 00    SendMsg 'postln'
17   F2       BlockReturn
  FunctionDef          FunctionDef
```

UGens can also use if

the condition ugen is 0 / 1

```
(
{
if( LFNoise1.kr(1.0,0.5,0.5) , SinOsc.ar, Saw.ar )
}.play
)
```

ID: 110

loop / repeat

create an object that behaves like a stream that returns values for a limited (or infinite) number of times.

Function-loop

repeats the function forever.

```
f = { 3.yield };
x = Routine({ f.loop });
10.do({ x.next.postln })
```

Object-repeat(n)

repeat to yield the object

```
x = 5;
y = x.repeat(6);
y.nextN(8);
```

Pattern-repeat(n)

```
x = Prand([1, 2]).repeat(6).asStream;
x.nextN(8);
```

Pattern-loop

```
x = Prand([1, 2]).loop.asStream;
x.nextN(8);
```

Stream-repeat(n)

embeds the stream repeatedly

```
x = Routine({ 3.do({ arg i; i.yield }) }).repeat(6);
x.nextN(8);
```

Stream-loop

embeds the stream repeatedly

```
x = Routine({ 3.do({ arg i; i.yield }) }).loop;
x.nextN(8);
```

ID: 111

Nil

Superclass: Object

Nil has a single instance named nil and is used to represent uninitialized data, bad values, or terminal values such as end-of-stream.

Instance Methods

isNil

Answers true because this is nil. In class Object this message is defined to answer false.

notNil

Answer false. In class Object this message answers true.

? anObject

? means return first non-nil argument. Since this IS nil then return anObject. In class Object, ? is defined to answer the receiver.

?? aFunction

If the receiver is nil, value the function and return the result. Since this IS nil, then value the function and return the result. In class Object, ?? is defined to answer the receiver.

Dependancy

All the messages for the Dependancy protocol (See class Object) are defined in class Nil to do nothing. This eliminates the need to check for nil when sending dependancy messages.

ID: 112

Object

superclass: nil

Object is the root class of all other classes. All objects are indirect instances of class Object.

Class membership:

class

Answer the class of the object.

```
5.class.name.postln;
```

respondsTo(selector)

Answer a Boolean whether the receiver understands the message selector.
Selector must be a Symbol.

```
5.respondsTo('+').postln;
```

isKindOf(aClass)

Answer a Boolean whether the receiver is a direct or indirect instance of aClass.
Use of this message in code must be questioned, because it often indicates a missed opportunity to exploit object polymorphism.

```
5.isKindOf(Magnitude).postln;
```

isMemberOf(aClass)

Answer a Boolean whether the receiver is a direct instance of aClass.
Use of this message in code is almost always a design mistake.

```
5.isMemberOf(Integer).postln;
```

Accessing:

size

Different classes interpret this message differently. Object always returns 0.

Copying:

copy

Make a copy of the receiver. The implementation of this message depends on the object's class. In class Object, copy calls shallowCopy.

shallowCopy

Makes a copy of the object. The copy's named and indexed instance variables refer to the same objects as the receiver.

deepCopy

Recursively copies the object and all of the objects contained in the instance variables, and so on down the structure. This method works with cyclic graphs.

Equality, Identity:

== anotherObject

Answer whether the receiver equals anotherObject. The definition of equality depends on the class
of the receiver. The default implementation in Object is to answer if the two objects are identical (see below).

=== anotherObject

Answer whether the receiver is the exact same object as anotherObject.

!= anotherObject

Answer whether the receiver does not equal anotherObject.
The default implementation in Object is to answer if the two objects are not identical (see below).

!== anotherObject

Answer whether the receiver is not the exact same object as anotherObject.

hash

Answer a code used to index into a hash table. This is used by Dictionaries and Sets to implement fast object lookup. Objects which are equal == should have the same hash values. Whenever == is overridden in a class, hash should be overridden as well.

identityHash

Answer a code used to index into a hash table. This method is implemented by a primitive and is not overridden. Objects which are identical === should have the same hash values.

Testing:

isNil

Answer whether the receiver is nil.

notNil

Answer whether the receiver is not nil.

isNumber

Answer whether the receiver is an instance of Number.

isInteger

Answer whether the receiver is an instance of Integer.

isFloat

Answer whether the receiver is an instance of Float.

pointsTo(anObject)

Answer whether one of the receiver's instance variables refers to anObject.

? anObject

If the receiver is nil then answer anObject, otherwise answer the receiver.

?? aFunction

If the receiver is nil, value the function and return the result.

switch(cases)

Object implements a **switch** method which allows for conditional evaluation with multiple cases. These are implemented as pairs of test objects (tested using if this == test.value) and corresponding functions to be evaluated if true. In order for switch to be inlined (and thus be as efficient as nested if statements) the matching values must be literal Integers, Floats, Chars, Symbols and the functions must have no variables or arguments.

```
(
var x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
switch (z.choose.postln,
1,   { \no },
1.1, { \wrong },
1.3, { \wrong },
1.5, { \wrong },
2,   { \wrong },
0,   { \true }
).postln;
)
```

or:

```
(
var x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
x = switch (z.choose)
{1}   { \no }
{1.1} { \wrong }
{1.3} { \wrong }
{1.5} { \wrong }
{2}   { \wrong }
{0}   { \true };
x.postln;
)
```

Messaging:

perform(selector ... args)

The selector argument must be a Symbol.
Sends the method named by the selector with the given arguments to the receiver.

performList(selector, ...args..., listOrArray)

The selector argument must be a Symbol.
Sends the method named by the selector with the given arguments to the receiver. If the last
argument is a List or an Array, then its elements are unpacked and passed as arguments.

performMsg(listOrArray)

The argument must be a List or Array whose first element is a Symbol representing a method selector.
The remaining elements are unpacked and passed as arguments to the method named by the selector

Printing:

post

Print a string representation of the receiver.

postln

Print a string representation of the receiver followed by a newline.

dump

Print a detailed low level representation of the receiver.

Dependancy:

addDependant(aDependant)

Add aDependant to the receiver's list of dependants.

removeDependant(aDependant)

Remove aDependant from the receiver's list of dependants.

dependants

Answer an IdentitySet of all dependants of the receiver.

changed(theChanger)

Notify the receiver's dependants that it has changed. The object making the change should be passed
as theChanger.

update(theChanged, theChanger)

An object upon which the receiver depends has changed. theChanged is the object that changed and
theChanger is the object that made the change.

release

Remove all dependants of the receiver. Any object that has had dependants added must be

released in order for it or its dependants to get garbage collected.

Routines

yield

Must be called from inside a **Routine.** Yields control to the calling thread. The receiver is the result
passed to the calling thread's method. The result of yield will be the value passed to the Routine's **next**
method the next time it is called.

yieldAndReset

Must be called from inside a **Routine.** Yields control to the calling thread. The receiver is the result
passed to the calling thread's method. The Routine is reset so that the next time it is called, it will
start from the beginning. yieldAndReset never returns within the Routine.

alwaysYield

Must be called from inside a **Routine.** Yields control to the calling thread. The receiver is the result
passed to the calling thread's method. The Routine, when called subsequently
will always yield the receiver until it is reset. alwaysYield never returns within the Routine.

ID: 113

Ref a reference to a value

superclass: AbstractFunction

A Ref instance is an object with a single slot named 'value' that serves as a holder of an object. Ref.new(object) one way to create a Ref, but there is a syntactic shortcut. The backquote ‘ is a unary operator that is equivalent to calling Ref.new(something).

example:

```
x = Ref.new(nil);
                    // method puts something in reference
                     // retrieve value and use it
```

Ref is also used as a quoting device to protect against multi channel expansion in certain UGens that require Arrays.

Class methods:

new(anObject)

create a Ref of an object.

‘anObject

create a Ref of an object.

Instance methods:

value

Answer the value.

value_(aValue)

set the value.

get

Answer the value.

set(aValue)

set the value.

dereference

Answer the value. This message is also defined in class Object where it just returns the receiver. Therefore `anything.dereference` will remove a Ref if there is one. This is slightly different than the value message, because value will also cause functions to evaluate themselves whereas dereference will not.

asRef

Answers the receiver. In class Object this message is defined to create a Ref of the object.

ID: 114

Symbol

superclass: Object

A Symbol is a name that is guaranteed to be unique. They can be used to represent symbolic constant values, Dictionary keys, etc.

Symbols are represented syntactically as literals which are described in [01 Literals] .

Testing

isClassName

Answer whether the symbol is a class name.

isSetter

Answer whether the symbol has a trailing underscore.

Conversion

asString

Convert to a String

asClass

Answer the Class named by the receiver.

asSetter

Return a symbol with a trailing underscore added.

asGetter

Return a symbol with a trailing underscore removed.

Math

Symbols respond to all unary and binary math operations by returning themselves. The result of any math operation between a Number or other math object and a Symbol is to return the Symbol. This allows operations on lists of notes which contain 'rest's to preserve the rests.

ID: 115

True

see [Boolean]

6 Crucial

6.1 Constraints

ID: 116

AbstractConstraint

subclasses:
Constraint SeenBefore IsIn IsNotIn Every Not Any Xor CountLimit IsEven IsOdd IsNil NotNil

Constraints let you specify conditions in an OOP fashion.
You can perform logical operations on the constraint object itself to further filter or refine your query.

```
(
// Create a constraint.
c =
 Constraint({ arg obj; obj.even  })
and:  Constraint({ arg obj; obj % 4 == 0 })
and:  (Constraint({ arg obj; obj == 8 }).not);
)
```

c is now a constraint object that can be used to validate that an input is even, divisible by 4 and is not the number 8.

```
c.value(3)

c.value(8)

c.value(4)

c.value(12)
```

This can be used any place a function that returns true/false is required.
eg. select, reject, every, any

```
(
// run numbers through it
50.do({
n = 40.rand;
```

```
[n,c.value(n)].postln
});
)
```

It can be used in place of a function for SequenceableCollections Streams Patterns

```
(
// filter a collection
Array.fill(100,{ 100.rand })
 .select(c) // acts like a function
.do({ arg num; num.postln; });
)
```

```
(
// Use to filter a Pattern
p = Pseries(0,1,100)
.select(c);
)

// Unfiltered
Pseries(0,1,100).asStream.all.do({arg num; num.postln;})

// Filtered
p.asStream.all.do({ arg num; num.postln });
```

```
(
// and here is everybody that gets rejected by the constraint
p = Pseries(0,1,100)
.reject(c);
)

p.asStream.all.do({ arg num; num.postln });
```

The below example is expressed using only Constraint.

```
(
c =
Constraint({ arg obj; obj.even  })
and:  Constraint({ arg obj; obj % 4 == 0 })
and:  (Constraint({ arg obj; obj == 8 }).not);
)
```

It could also be expressed this way

```
(
c =
IsEven.new and:   Constraint({ arg obj; obj % 4 == 0 })
and:  Constraint({ arg obj; obj != 8 });
)
```

Constraints respond to

```
.not
.or(aConstraint)
.and(aConstraint)
.xor(aConstraint)
.reject(aConstraint)
.select(aConstraint)
```

by returning a new compound constraint that expresses that logic.

```
(
c = IsEven.new;
d = Constraint({ arg num; num == 3 });

 e = c or:  d; // if its even or it is the number 3
)
(
c = IsEven.new;
d = Constraint({ arg num; num == 4 });

 e = c.reject(d); // if its even and also reject it if it is the number 4
)
```

ID: 117

Constraint

superclass: AbstractConstraint

Constraint takes a function that it evaluates whenever the constraint itself is evaluated. Its main benefit then over a simple function is that it responds to .not .or .and .xor .reject .select by returning the logical compound constraint.

see [**AbstractConstraint**] for examples of Constraint

ID: 118

CountLimit

**counts how many items have been presented to it. returns true until the limit has
been reached, thereafter returns false.**

superclass: AbstractConstraint

```
(
s = CountLimit.new(10);

100.do({ arg i;
var r;
r = 100.rand;
if(s.value(r),{
[i,r].postln;
})
});


[ 0, 58 ]
[ 1, 37 ]
[ 2, 39 ]
[ 3, 82 ]
[ 4, 99 ]
[ 5, 93 ]
[ 6, 27 ]
[ 7, 38 ]
[ 8, 20 ]
[ 9, 77 ]
)
```

ID: 119

IsEven

answers whether the item is even.

superclass: AbstractConstraint

```
(
s = IsEven.new;

100.do({ arg i;
if(s.value(i),{
i.postln
})
});

)
```

The class IsEven itself will respond to *value just as an instance will. So it can be used in place of functions in the same manner. This is faster than constructing a FunctionDef { }, and probably executes faster.

```
(

Array.fill(20,{rand(100)})
.select(IsEven)
.postln

[ 12, 76, 76, 8, 18, 26, 30, 44, 24, 84 ]
)
```

ID: 120

IsNotIn answers whether an item is not included in a collection

superclass: AbstractConstraint

```
(

a = [1,3,6,8,9,0];
[1,2,3,4,5,6,7,8,9,0].select(IsNotIn(a)).postln

[ 2, 4, 5, 7 ]
)
```

ID: 121

SeenBefore

**keeps a history of all items and answers whether an item has been seen by the constraint
object before.**

superclass: AbstractConstraint

```
(
    SeenBefore

100.do({ arg i;
var r;
r = 100.rand;
if(s.value(r).not,{
[i,r].postln;
})
});

[ 0, 88 ]
[ 1, 19 ]
[ 2, 71 ]
[ 3, 83 ]
[ 4, 56 ]
[ 5, 97 ]
[ 6, 98 ]
[ 7, 78 ]
[ 8, 65 ]
[ 9, 63 ]
[ 10, 7 ]
[ 11, 5 ]
[ 12, 30 ]
[ 13, 53 ]
[ 14, 15 ]
[ 15, 70 ]
[ 16, 74 ]
[ 17, 44 ]
```

```
[ 18, 18 ]
[ 19, 66 ]
[ 22, 6 ]
[ 23, 60 ]
[ 24, 42 ]
[ 25, 95 ]
[ 26, 62 ]
[ 27, 96 ]
[ 28, 29 ]
[ 29, 81 ]
[ 30, 49 ]
[ 31, 13 ]
[ 32, 47 ]
[ 33, 59 ]
[ 34, 61 ]
[ 36, 34 ]
[ 37, 1 ]
[ 38, 0 ]
[ 43, 25 ]
[ 44, 3 ]
[ 46, 20 ]
[ 50, 16 ]
[ 51, 76 ]
[ 54, 87 ]
[ 55, 99 ]
[ 56, 90 ]
[ 57, 36 ]
[ 59, 57 ]
[ 60, 67 ]
[ 61, 45 ]
[ 62, 94 ]
[ 66, 86 ]
[ 69, 92 ]
[ 71, 80 ]
[ 72, 91 ]
[ 75, 89 ]
[ 79, 69 ]
[ 81, 35 ]
[ 84, 10 ]
[ 86, 73 ]
```

```
[ 87, 2 ]
[ 89, 8 ]
[ 93, 33 ]
[ 95, 31 ]
[ 98, 23 ]
)
```

6.2 Control

ID: 122

MIDIResponder

Register multiple functions to be evaluated when MIDI events occur.

MIDIResponder is an abstract class. These subclasses should be used for specific midi work.

CCResponder Respond to control messages
NoteOnResponder Respond to note-on messages
NoteOffResponder Respond to note-off messages
BendResponder Respond to pitch bend messages
TouchResponder Respond to aftertouch messages

Creation and initialization:

CCResponder(function, src, chan, num, value, install = true)
NoteOnResponder(function, src, chan, num, veloc, install = true)
NoteOffResponder(function, src, chan, num, veloc, install = true)
BendResponder(function, src, chan, value, install = true)
TouchResponder(function, src, chan, value, install = true)

function: The function to execute when the incoming MIDI event matches the responder. The function takes the arguments src, chan, A, B (or for Bend and Touch, src, chan, value).
src: If a number is given, the responder will fire only for messages coming in from this port. The number may be the system UID (obtained from MIDIClient.sources[index].uid) or the index itself. If nil, the responder will match any port.
chan: The MIDI channel(s) to match.
num: The control or note number(s) to match.
value: The value(s) to match.
veloc: The velocities to match.
install: If true, install the responder automatically. If false, return the responder but don't install it (it will be inactive).

Any of the matching values may be one of the following:

Nil: Match anything.
Integer: Match only this specific number.

Array: Match any item in the array. Any kind of Collection will work here.
Function: Evaluate the function with the incoming value as the argument. The function should return true or false.

For instance, this would respond to note on messages from any port, channels 2 and 7 only, even numbered note numbers only, and only velocity values greater than 50.

```
NoteOnResponder({ | src, chan, num, vel|  [src, chan, num, vel].postln },
nil,
[2, 7],
 (0, 2..126),  // could also be { | num| num.even } or _.even
{ | vel|  vel > 50 });
```

MIDIResponders automatically initialize the MIDIClient with 1 standard device. If you have more devices, simply initialize the MIDIClient yourself before using any MIDIResponders.

Removal:

Just call .remove on the responder.

```
    CCResponder                    // respond to any modwheel

          // stop this responder
```

6.3 Editors

ID: 123

BooleanEditor

```
b = BooleanEditor(false);

b.gui;

b.value.postln;
```

ID: 124

```
IrNumberEditor

used with Patch or InstrSpawner to specify an .ir rate control

a float or integer will not create a synth arg input.   the number will be passed
directly into the synth def.



(
InstrSpawner({ arg sample,pchRatio=1,start=0,dur=0.2;

Pan2.ar(
PlayBuf.ar(sample.numChannels,
sample.bufnumIr,
sample.bufRateScaleIr * pchRatio,
startPos:  start * sample.bufFramesIr,
loop:  0
),Rand(0,1),0.3)
* EnvGen.kr( Env.perc(0.01,1.0,1,3), timeScale:  dur, doneAction:  2)

},[
Sample("sounds/ a11wlk01.wav"),
IrNumberEditor(1,[-4,4]),
IrNumberEditor(0,[0,1]),
IrNumberEditor(0.2,[0.05,0.5])
],
NumberEditor(0.1,[0.05,0.2])
).gui

)
```

ID: 125

KrNumberEditor

superclass: NumberEditor

This is the defaultControl for a ControlSpec. a KrNumberEditor is like its superclass, except that if used in a Patch it will be a continously modulateable control. You can move the slider and it sends its

IMPORTANT

if a KrNumberEditor is connected to a Patch playing on a server, the message to the server happens when the KrNumberEditor gets the .changed message and sends .update to all its depedants. This includes any NumberEditorGui and also any UpdatingScalarPatchOut, which is what actually sends the message to the server.

```
/*
s.boot;
(
// works as a stream ..  convienient for patterns
 n=NumberEditor(440.0,\freq);
 n.gui;

 Pbind(
  \freq,n
 ).play
)
*/
```

from an email:

KrNumberEditors support lag. You can set lag to nil for no Lag.

I have added NoLagControlSpec, whose defaultControl sets the lag to nil.

I would prefer to just have a lag preference in ControlSpec

(clients that do lag eg. sliders can tell from the spec if they should do lag or not).

as Jan pointed out a while ago, binary things don't like lag.

\binary, \loop is registred as a NoLagControlSpec, and thus doesn't use any lag at all. you can register others, or use a NoLagControlSpec when writing the Instr.

I am experimenting with different kinds of inertia Lag (hysterisis ?), so I'm not using LagControl
right now, but it might switch to that.
either way it violates the contract : it should be a function on the input object, not at the receiving end
inside of the function. but its more efficient.

ID: 126

NumberEditor holds a float for editing

NumberEditor.new(value,spec)
value - initial value
spec - ControlSpec or StaticSpec. see [Spec]

like all editors,
.value
.asCompileString
.next

all return the float value, not the editor itself.

This is the default control view for a StaticSpec.

If used in a Patch, it will return its initial value when the patch starts, but will not be modulateable after that. See KrNumberEditor for modulateable.

NumberEditor can also be used in Pbind, since it returns its float value in response to .next or .value

```
(
n = NumberEditor(2.3,[0,10]);
n.value = 5.6;
n.asCompileString.postln;
5.6
)
```

```
(
//note that the .gui message returns a NumberEditorGui
n = NumberEditor(440.0,\freq).gui;
```

```
n.insp;
)

(
// so make sure you get the NumberEditor
  NumberEditor       \freq
n.gui;
n.insp;
)



(
 f=MultiPageLayout

n=NumberEditor(440.0,\freq);
n.topGui(f);

ActionButton(f,"post value",{ n.value.postln });
 // it compiles as its value
 ActionButton   "post NumberEditor asCompileString"
n.asCompileString.postln
});
f.resizeToFit.front;
)

// programatically set it
n.value = 100
n.changed; // now the slider moves
// and sends to the server !




// controlling the display
(
Sheet({ arg f;
f.startRow;
NumberEditor(440.0,\freq).gui(f); // default

NumberEditor(440.0,\freq).smallGui(f); // smallGui never has slider
```

```
NumberEditor(440.0,\freq).gui(f,nil, false); //use gui,nil bounds, slider:  false

f.startRow;
NumberEditor(440.0,\freq).gui(f,60@10,true); // slider 60 by 10
f.startRow;
NumberEditor(440.0,\freq).gui(f, 200@40, true); // slider 200 by 40
f.startRow;



NumberEditor(440.0,\freq).smallGui(f);
NumberEditor(440.0,\freq).smallGui(f);
NumberEditor(440.0,\freq).smallGui(f);
NumberEditor(440.0,\freq).smallGui(f);

f.startRow;
NumberEditor(440.0,\freq).gui(f,20@100,true); // verticle, with slider
NumberEditor(440.0,\freq).gui(f,20@100,true); // verticle, with slider
})
)
bug:  verticle not working yet

Putting them on a Sheet
(
w = Sheet({ arg h;
c = Array.fill(10,{ arg i;
var n;
n = NumberEditor(0,\amp);
h.startRow;
n.gui(h);
n
});
});

)

Putting them on a MultiPageLayout
(
w = MultiPageLayout.new;
c = Array.fill(10,{ arg i;
var n;
```

```
n = NumberEditor(0,\amp);
w.startRow;
n.gui(w);
n
});
w.front;
)

Putting them on normal windows
(
w = SCWindow.new;
w.front;
c = Array.fill(10,{ arg i;
var n;
n = NumberEditor(0,\amp);
n.gui(w,Rect(10,25 * i, 150,13));
n
});

)

using a MultiPageLayout on a window
(

w = SCWindow.new;
w.front;
p = MultiPageLayout.on(w);
c = Array.fill(10,{ arg i;
var n;
n = NumberEditor(0,\amp);
n.gui(p);
p.startRow;
n
});

)


put them on a FlowView
(
```

```
w = SCWindow.new;
w.front;
p = FlowView(w,Rect(10,10,500,500));
c = Array.fill(10,{ arg i;
var n;
n = NumberEditor(0,\amp);
n.gui(p);
p.startRow;
n
});

)
```

```
// a nice glitch display
//verticle not working yet
(
w = SCWindow.new;
w.front;
c = Array.fill(10,{ arg i;
var n;
n = NumberEditor(0,\amp);
n.gui(w,Rect(10 + (15 * i),25, 13,150));
n
});

)

// in SCVLayout not working yet either
(
w = SCWindow.new;
w.front;
v = SCVLayoutView.new(w,w.view.bounds);
c = Array.fill(10,{ arg i;
```

```
var n;
n = NumberEditor(0,\amp);
n.gui(v,Rect(0,0,100,20));
n
});

)


//works with sliders
(
w = SCWindow.new;
w.front;
v = SCVLayoutView.new(w,w.view.bounds);
c = Array.fill(10,{ arg i;
var n;
n = SCSlider(v,Rect(0,0,100,20));
n
});

)
```

6.4 Gui

ID: 127

ActionButton

Superclass: Object

crucial dependencies: PageLayout

*new(layout, title, function)

if layout is nil, a new PageLayout will be created.
the size of the button will scale to that of the string

```
(
 ActionButton nil "hit me"
  "yeah baby"
});

)

(
//   bigger title...  bigger button
 ActionButton nil "hit me hit me hit me hit me hit me hit me hit me hit me hit me hit me hit me "
  "yeah baby"
});
)



(
// set minimum sizes for x and y
// longer text will still cause it to expand
ActionButton nil "hit me"
 "yeah baby"
},200,40,Color.white,Color.black);

)



// accepting drags by setting the view.receiveDrag handler
the list view by default gives an integer when dragging from it.
```

here i am making the action button accept dragged integers.

```
(
Sheet   arg

a = SCListView(f,100@100);
a.items = ["a","b","c"];

 b = ActionButton   "i accept integers"
"was hit".postln
});
b.view.canReceiveDragHandler = { SCView.currentDrag.isNumber };
b.view.receiveDragHandler = {
a.items[ SCView.currentDrag.asInteger ].postln;
};
})
)
```

here the list view is made to export a string when dragged from.
the action button is accepting strings dragged to it.

```
(
Sheet   arg
a = SCListView(f,100@100);
a.items = ["a","b","c"];
a.beginDragAction = { arg listView;
listView.items[ listView.value ].debug("begin dragging");
};

 b = ActionButton   "i accept strings"
"butt hit".postln
});
b.view.canReceiveDrag = { SCView.currentDrag.isString };
b.view.receiveDrag = {
SCView.currentDrag.postln;
};
})
)
```

ID: 128

CXMenu

A pop up menu that does its action and closes itself after you select an item.

The difference between this and PopUp is that here there are separate functions for each menu item, and with PopUp there is one action.

```
(
    CXMenu
\soup->{ "soup".postln; },
\pork->{ "pork".postln; },
\duck->{ "duck".postln; },
\tiramisu->{ "tiramisu".postln; }
);


m.gui(nil);

)

(
    CXMenu
\myName->{ "you hit myName".postln; },
 \yourName    "you hit yourName"
);

m.closeOnSelect = false;

m.gui;

)
```

On another layout

```
(
Sheet({ arg f;
CXLabel(f,"partials:");
```

```
f.startRow;
 m = CXMenu.newWith
Array.fill(15,{ arg i;
i.asString.asSymbol -> { i.postln }
})
);
m.closeOnSelect = false;
m.gui(f);
})
)
```

You can add things to the menu above

```
m.add(\more->{ "more".postln; });
```

On a normal SCWindow

```
(


w = SCWindow.new;
w.front;


x = CXMenu( \a -> { "a".postln },\b -> { "b".postln },\c -> { "c".postln });
x.closeOnSelect = false;
x.gui(w);


)
```

Note that the arrow keys work to navigate once you are focused on any of the buttons.

ID: 129

FlowView

superclass: SCLayoutView

an SCCompositeView with a FlowLayout inside of it.

In one respect this is simply a lazy contraction of this :

w = SCWindow.new;
w.view.decorator = FlowLayout.new(w.bounds);
w.front;

crucial style gui objects and normal sc views can easily coexist here.

FlowView(parent,bounds)

```
(
// makes a window automatically
f = FlowView.new;

//lazy crucial gui objects work
ActionButton   "a"

// normal sc views are flowed
SCSlider(f,Rect(0,0,100,100));

// flow within a flow
g = f.flow({ arg g;
 ActionButton   "a"
SCSlider(g,Rect(0,0,100,100)).backColor_(Color.rand);
}).background_(Color.black);
 // shrinks to fit the contents afterwards

// rather than this :   f.decorator.nextLine
// talk to the FlowView
f.startRow;
// it will insert a StartRow object as a pseudo-view,
```

```
// this will keep the flow as you specified it for views that follow it :
ActionButton   "next line"
// even after you resize a parent view etc.

)




// tests
(
FlowView.new.flow({ arg f;
// b = ActionButton(f,"hi",minWidth:140)
}).background_(Color.grey)

)
(
FlowView.new.flow({ arg f;
b = ActionButton(f,"hi",minWidth:140);
}).background_(Color.grey)

)
(
FlowView.new.flow({ arg f;
b = SCSlider(f,Rect(0,0,100,100));
}).background_(Color.grey)

)
(
FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
  //b = ActionButton(f,"hi",minWidth:140)
}).background_(Color.white)
}).background_(Color.grey)

)
(
```

```
FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
b = ActionButton(f,"hi",minWidth:140)
}).background_(Color.white)
}).background_(Color.grey)

)

(

FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
b = ActionButton(f,"hi",minWidth:140);
}).background_(Color.white)
}).background_(Color.grey)


)

(

FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
b = ActionButton(f,"hi",minWidth:140);
}).background_(Color.white)
}).background_(Color.grey)


)
(
```

```
FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
b = ActionButton(f,"hi",minWidth:140);
f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
}).background_(Color.white)
}).background_(Color.grey)

)

(

FlowView.new.flow({ arg f;
g = f;
f.flow({ arg f;
b = SCSlider(f,Rect(0,0,140,20));
f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
}).background_(Color.white)
}).background_(Color.grey)


)


(

FlowView.new.flow({ arg f;
b = SCSlider(f,Rect(0,0,140,20));
f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
}).background_(Color.grey)


)
```

```
(

FlowView.new.flow({ arg f;
g = f;
w = f.flow({ arg f;
b = f.flow({ arg f;
ActionButton(f,"hello",minWidth:100);
}).background_(Color.blue);
ActionButton(f,"hi",minWidth:140);
}).background_(Color.white)
}).background_(Color.grey)

)

b.remove(true);
w.resizeToFit(true,true);


// add something big back in
ActionButton(w,"i'm back",minWidth:  200);
//its messed up, outside of the bounds
w.resizeToFit(true,true);
```

ID: 130

GUI Classes Overview

The following GUI classes have individual helpfiles. There are a number of undocumented GUI classes listed in **Undocumented-Classes**.

Color
Document
Font
SC2DSlider
SC2DTabletSlider
SCButton
SCCompositeView
SCEnvelopeView
SCFuncUserView
SCHLayoutView
SCMultiSliderView
SCNumberBox
SCPopUpMenu
SCRangeSlider
SCTabletView
SCTextField
SCVLayoutView
SCView
SCWindow
resize

ID: 131

MLIDbrowser

MultiLevelIdentityDictionary browser

From any node, you can browse down to the leaves.

***new(name1,name2 ... nameN , onSelect)**

name1,name2 ... nameN -
the name of the node you wish to start browsing at.
if nil, it will browse from the top of Library.

onSelect -
the function that is executed when you click on a leaf node.
if nil, it will supply a function that guis the item.

```
(
 // what exactly is in Library right now ?
  MLIDbrowser
)

(
// put in something to library
Library.put(\test,"hello");
MLIDbrowser.new(\test);
)


(
// browse all currently loaded instruments
// if you have no Instr loaded, then Library.at(Instr) will return nil
 Instr("help-MLIDbrowser",{ arg freq=440,phase=0.0,amp=0.2;
SinOsc.ar(freq,phase,amp);
});
```

```
 //make a Patch when you select an instr
MLIDbrowser(\Instr,{ arg instr; Patch(instr.name).topGui });
)
```

To browse all the Instr in your Instr folder, you need to load each one of them.

Simply by accessing each one by its first name (filename and first symbol in the name list), you will force it to load.

```
[\oscillOrc,\synths].do({ arg name; Instr.at(name) });
```

ID: 132

ModalDialog

```
(
 ModalDialog   arg          // pops a small window for you

var choices;

choices= [
   BooleanEditor  true,false].choose
   BooleanEditor  true,false].choose
   BooleanEditor  true,false].choose
   BooleanEditor  true,false].choose
   BooleanEditor  true,false].choose
   BooleanEditor  true,false].choose
];

choices.do({ arg c,i;
c.gui(layout);
});

  choices // return your objects

 },{ // ok function
arg choices; // receives those objects
choices.sum({ arg boolgui,i;
boolgui.value.binaryValue * (2**i)
}).postln
});
)


see also  [Sheet]
```

ID: 133

MultiPageLayout

A MultiPageLayout creates one or more windows with FlowViews on them.
It can be used as the specified parent for any view. It will place the view on the current FlowVIew, creating a new window as necessary to handle overflow.

```
(
var f;
 f=MultiPageLayout     "flow"

800.do({ arg i;
SCButton( f, Rect(0,0,30,30 ) )
.states_([[i.asString,Color.black,Color.white]])
});
f.front;
)
```

The windows are treated as a group. When one closes, they all close.

.window
returns the current window being written to.

.view
the FlowView on the current window

.close
close all windows created by the MultiPageLayout

.focus(index)
focus the view on the current window

.front
cascades all windows owned by MultiPageLayout to front

.resizeToFit

resizes all windows to fit their contents, brings them all to front

```
(
var f,sliders;
 f= MultiPageLayout     "a vaguely practical example"

sliders=Array.fill(rrand(16,32),{ arg i;
SCSlider( f, 10@150 );
});

f.flow({ arg subf;
SCSlider( subf, 30@30 );
SCSlider( subf, 30@30 );
SCSlider( subf, 30@30 );
SCSlider( subf, 30@30 );
}, 50@100 );

f.resizeToFit;
f.front;
)



//
//// layout within a layout
//(var f;
// f=MultiPageLayout.new("flow");
//
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
// // html joke, but useful
// f.hr;
//
// // allocate space for a small layout within
// // a verticle strip
// f.within( 100,300,{ arg subLayout;
//  5.do({ arg i;
//   RangeView(subLayout.win, subLayout.layRight(100.rand,100.rand),"",0,1,0,1)
```

```
//  })
// });
//
// // more sliders to the right of the strip
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
// // continuing with a new section below
// f.hr;
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
//)
```

A nice way to work with MultiPageLayout is with [**Sheet**]

A MultiPageLayout closes all of its windows when any of its windows is closed.
When a MultiPageLayout closes, it sends the \didClose notification (see NotificationCenter).

You can register to receive that notification:

```
NotificationCenter.registerOneShot( f,\didClose, yourObject,{
   // stop the model from playing, clean house,
   // unregister a keydown etc.
});
```

This notification unregisters itself after being called.

removeOnClose(controller)
this method adds the controller to the MultiPageLayout's autoremove list. when the window closes, all items on the list will be sent the **.remove** message. this allows them to release any of their own dependancies, clean the poop out of their cage etc.

eg: Updater, ObjectGui (and subclasses), and NotificationRegistration

A Note about MVC/Dependencies

this is some under-the-hood stuff that you don't need to know about but might be interested in.

When a subclass of ObjectGui is added to a MultiPageLayout it adds itself as a dependant on that layout. (ObjectGui::guify).

When MultiPageLayout creates a SCWindow, it sets the window's onClose function to call MultiPageLayout::release. this calls .**release** on all of the MultiPageLayout's dependants.

The MultiPageLayout calls .**release** on all the gui objects when the SCWindow closes. The gui objects release their dependancy on their models, thus severing the link between the two, allowing garbage collection.

in sc all Views must actually be on a window. when a View is created, it is added to a window, and the SCWindow keeps track of them in its **view.children** array.

this gui system has Gui classes that build and maintain an interface/gui on a specific model. They are Controllers, and they create actual View objects and control the interface between those and the Model.

these controllers add themselves as dependants on the models.
eg. a PatchGui is a dependant of a Patch

when a SCWindow is shut, it calls .**release** on every no*rmal v*iew (StringView, RangeView etc.), to cause each view to sever its dependency relationship with its model. Otherwise, even though the window is gone and the view is not visible (and neither the window, the view or the model have any direct reference to each other), it will not get garbage collected because there is still an entry in the dependants dictionary (Object classvar) listing that view as a dependant on the model. there is still a link between model and view.

The Gui objects (controllers) need to know when the window closes so they can release themselves from the model.

in Object::guify this happens:

```
layout.removeOnClose(guiObject)
```

ID: 134

ObjectGui

The guiClass is the class used to build a gui for an object.

In the MVC architecture it is the Controller, which creates Views for manipulating the properties of your Model, and receives messages from the View and enacts the changes on the Model.

The default guiClass for an Object is ObjectGui.

Many subclasses overide the **guiClass** method to specify a different class. All gui classes should inherit from ObjectGui.

```
see [gui]
```

It is the simplest display, just the the object asString.

if you click on the "nameplate", you will open object's inspector.

an example gui class

```
YourSimpleGuiClass   ObjectGui

guiBody { arg layout;

  // we refer to the model and
  // access its variable howFast.
    // if its a simple number, it will display
  // using the default ObjectGui class, which
  // will simply show its value as a string.
model.howFast.gui(layout);
}
}
```

```
// more complex
YourGuiClass   ObjectGui

var numberEditor;

 //for example
guiBody { arg layout;
var r;
  // the object you are making a gui for is referred to as the model

  // display some param on screen.
    // here we assume that someParam is something that
  //  has a suitable gui class
  // implemented, or that the default ObjectGui is sufficient.
model.someParam.gui(layout);

  // using non 'gui' objects
r = layout.layRight(300,300); // allocate yourself some space
ButtonView(layout.win,r)
.action_({ arg butt;
model.goApeShit;
});

numberEditor = NumberEditor(model.howFast,[0,100])
.action_({ arg val;
model.howFast = val;
model.changed(this);
    // tell the model that this gui changed it
});
numberEditor.gui(layout);
}

 // your gui object will have update called any time the .changed message
 // is sent to your model
update { arg changed,changer;

if(changer !== this,{
   /* if it is this gui object that changed the value
    using the numberEditor, then we already have a correct
    display and don't need to waste cpu to update it.
```

```
        if anyone else changed anything about the model,
    we will update ourselves here.
      */
numberEditor.value = model.howFast;
/*
    note that
     numberEditor.value = model.howFast;
    is passive, and does not fire the numberEditor's action.

          numberEditor.activeValue = model.howFast
    would fire the action as well, resulting in a loop that would
    probably crash your machine.
      */
}
}

}


 // you can gui an object more than once.
// they are both active interfaces to the object.

  n = NumberEditor

 Sheet   arg
n.gui(f);
n.gui(f);
})
)
```

When the PageLayout window closes that your gui object (Controller) is on, it will be removed as a dependent on the Model, so it will no longer be sent the update message, and will then be free for garbage collection.

ID: 135

PageLayout

A PageLayout is a window that manages the layout of views added to it.

You request a rectangle using **layout.layRight(x,y)**

the layout manager moves its internal cursor, wrapping to a new line, then to a new window as necessary.

Wraps to the next line

```
(
var f;
f=PageLayout.new("flow");

504.do({ arg i;
SCButton( f.window, f.layRight(30,30) )
.states_([[i.asString,Color.black,Color.white]])
});

f.front;
)
```

Exceeding the bottom of the window wraps to a new window

```
(
var f;
f=PageLayout.new("flow");

800.do({ arg i;
var r;
  r= f.layRight(30,30); // obtain the rectangle first
      // in case we cascade to the next window
SCButton( f.window, r )
.states_([[i.asString,Color.black,Color.white]])
});
f.front;
)
```

The windows are treated as a group. When one closes, they all close.

.window
returns the current window being written to.

.layRight(x,y)
allocates space and returns a Rect of size (x,y)

.close
close all windows created by the PageLayout

.focus(index)
focus the view on the current window

.toFront
cascades all windows owned by PageLayout to front

.resizeToFit
resizes all windows to fit their contents, brings them all to front

```
(
 var f,sliders;
f= PageLayout.new("a vaguely practical example");

sliders=Array.fill(rrand(16,32),{ arg i;
SCSlider( f.window, f.layRight(10, 150));
});

f.within( 50,150,{ arg subf;
SCSlider( subf.window, subf.layDown( 30,30));
SCSlider( subf.window, subf.layDown( 30,30));
SCSlider( subf.window, subf.layDown( 30,30));
SCSlider( subf.window, subf.layDown( 30,30));
});

f.resizeToFit;
```

```
f.front;
)


//
//// layout within a layout
//(var f;
// f=PageLayout.new("flow");
//
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
// // html joke, but useful
// f.hr;
//
// // allocate space for a small layout within
// // a verticle strip
// f.within( 100,300,{ arg subLayout;
//  5.do({ arg i;
//   RangeView(subLayout.win, subLayout.layRight(100.rand,100.rand),"",0,1,0,1)
//  })
// });
//
// // more sliders to the right of the strip
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
// // continuing with a new section below
// f.hr;
// 30.rand.do({ arg i;
//  SliderView(f.win, f.layRight(80.rand,80.rand))
// });
//
//)
```

A nice way to work with PageLayout is with [**Sheet**]

A PageLayout closes all of its windows when any of its windows is closed.
When a PageLayout closes, it sends the \didClose notification (see NotificationCenter).

You can register to receive that notification:

```
NotificationCenter.registerOneShot( f,\didClose, yourObject,{
   // stop the model from playing, clean house,
   // unregister a keydown etc.
});
```

This notification unregisters itself after being called.

removeOnClose(controller)
this method adds the controller to the PageLayout's autoremove list. when the window closes, all items on the list will be sent the **.remove** message. this allows them to release any of their own dependancies, clean the poop out of their cage etc.

eg: Updater, ObjectGui (and subclasses), and NotificationRegistration

A Note about MVC/Dependencies

this is some under-the-hood stuff that you don't need to know about but might be interested in.

When a subclass of ObjectGui is added to a PageLayout it adds itself as a dependant on that layout. (ObjectGui::guify).

When PageLayout creates a SCWindow, it sets the window's onClose function to call PageLayout::release. this calls .**release** on all of the PageLayout's dependants.

The PageLayout calls .**release** on all the gui objects when the SCWindow closes. The gui objects release their dependancy on their models, thus severing the link between the two, allowing garbage collection.

in sc all Views must actually be on a window. when a View is created, it is added to a window, and the SCWindow keeps track of them in its **view.children** array.

this gui system has Gui classes that build and maintain an interface/gui on a specific model. They are Controllers, and they create actual View objects and control the interface between those and the Model.

these controllers add themselves as dependants on the models.
eg. a PatchGui is a dependant of a Patch

when a SCWindow is shut, it calls .**release** on every no*rmal v*iew (StringView, RangeView etc.), to cause each view to sever its dependency relationship with its model. Otherwise, even though the window is gone and the view is not visible (and neither the window, the view or the model have any direct reference to each other), it will not get garbage collected because there is still an entry in the dependants dictionary (Object classvar) listing that view as a dependant on the model. there is still a link between model and view.

The Gui objects (controllers) need to know when the window closes so they can release themselves from the model.

in Object::guify this happens:

```
layout.removeOnClose(guiObject)
```

ID: 136

SaveConsole

a tool for managing .asCompileString based archiving of objects.

does a no-clobber check, makes a .bak copy of any previously existing file it finds. saves inside a TaskIfPlaying, so you don't have to stop play.

SaveConsole(object,path)

object:
anything that can meaningfully respond to .asCompileString
path:
if the object you are supplying was already loaded from disk and has a known path, tell the SaveConsole of this. (makes a difference for save/save as behavior)

```
(
 SaveConsole
Array.rand(16,10,200),
  nil  // no previous path
  nil  //no layout
 .print // prints object
.save
.saveAs

)
```

ID: 137

SelectButtonSet

Radio button style set, fashioned as a single object.

**SelectButtonSet.new(layout,
labels, // array of labels or a quantity
action,
color, // or color function
selectedColor, // or selectedColor function
x,y // optional size of each button
)**

(

```
 SelectButtonSet  nil     "one"  "two" "three" "four"
{ arg selectedIndex,selectButtonSet;
[ selectedIndex, selectButtonSet ].postln;
}
)
```

)

(

```
 SelectButtonSet  nil
16 , //  make 16 buttons
{ arg selectedIndex,selectButtonSet;
[ selectedIndex, selectButtonSet ].postln;
}
)
```

)

.selected

selected index
.**selectedLabel**
label of the selected
select(index)
passiveSelect(index)
action not performed

color and selectedColor may be either a Color object or a function that will be passed the selected index when valued.

on various kinds of layouts/windows/nil :

```
SelectButtonSet
nil,
["1", "2"],
{| i|  ("Input" + i).postln},
x:  40, y:  30
);


SelectButtonSet
FlowView.new,
["1", "2"],
{| i|  ("Input" + i).postln},
x:  40, y:  30
);

SelectButtonSet
SCWindow.new.front,
["1", "2"],
{| i|  ("Input" + i).postln},
x:  40, y:  30
);

SelectButtonSet
SCHLayoutView.new(SCWindow.new.front,400@400),
["1", "2"],
{| i|  ("Input" + i).postln},
x:  40, y:  30
```

```
);
```

ID: 138

Sheet

a simple PageLayout that auto-resizes to fit its contents.

```
A better name might be Page.


(
 Sheet   arg
  ActionButton   "selectionStart"
  ActionButton   "selectionEnd"
});
)


see also  [ModalDialog]
```

ID: 139

SynthConsole

*new(object, layout)

convenient buttons for common utilities:
play, record, stop, tempo etc.

each method adds another button.

This may be used on its own and it is also a component used in AbstractPlayerGui for all Players.

An easy way to use it:

```
(
 Sheet   arg

  SynthConsole   arg
SinOsc.ar(300,0,0.3)
  },f) // if no layout provided it will create one
.play
.scope
.fftScope
  .record("SoundFiles/testy.aiff"  // sets defaultPath for the prompt dialog
  .write(20) // 20 seconds
  .pauseableRecord // | ,|   toggle recording on and off while you play
.stop
.formats
  .tempo // gui the default Tempo

})
)
```

certain controls are not yet enabled in sc3, so the button will not appear.

note: the play button sends the .play message to your object.

see also [**FunctionPlayer**]

SynthConsole sends notifications that you can register to recieve through NotificationCenter:

```
NotificationCenter.register(yourSynthConsole,\didStop,you,{
  // do something like
  true                 // turn back on the wacom mouse
});

NotificationCenter.register(yourSynthConsole,\didRecordOrWrite,you,{
arg path; // path is passed in with the notification
savedTo = path;
});
```

ID: 140

ToggleButton

ToggleButton.new(layout,title,onFunction,offFunction,initialState)

```
    ToggleButton nil "push me"
"on".postln;
},{
"off".postln;
  false

t.toggle;
```

6.5 Instr

ID: 141

ControlPrototypes automatically create inputs for Patches

This is a registery of controls, cataloged by the Spec of their output. It was used by Patch to procure suitable control objects to satisfy an argument to a function.

In other words: you give it a Spec, it gives you some suitable Player object to use as an input.

In this distribution, no special controls are registered for specific specs. You could use this
to customise your "auto input creation" in any number of ways.

In your Main-startUp method:

```
    ControlPrototypes

\trig -> {[
      StreamKrDur Pseq Array
]},
//   \freq -> {[
//     ModalFreq(StreamKrDur(Pseq(Array.fill(16,{ rrand(0,11).round }),0.25)
//       ..  etc...
//    ]},
EnvSpec -> {[
EnvEditor.new(Env.adsr)
]}
);
```

*define(associations...)

Keys are either symbols or classes, the values are functions that return arrays of prototypes. Patch simply selects the first in the list. Other methods of ControlPrototypes use the full list.

The function is valued each time so that the control is a unique instance.

You may freely change or redefine control protypes while working/composing without recompiling.

This class also serves to decouple Spec from knowing of the various controls, gadgets and widgets.

ID: 142

Instr - Instrument

An Instrument is a named sound function that is stored in the Library.

Storing:

```
Instr(\sin, { arg freq,amp;
SinOsc.ar(freq,0.0, amp)
});
```

Retreiving:

```
Instr    \sin
```

or:

```
Instr \sin
```

If the Instr is not found in the Library it will search in the Instr directory and **load it from a file**.

By default the directory Instr.dir is "build/Instr" or you may set Instr.dir in your startup.

```
Instr       " /Documents/SuperCollider/Instr"
```

Specify by **dot notation** to look for a file named 'oscillators' :

```
Instr  "oscillators.sin"
Instr  "folder.subfolder.oscillators.sin"
```

It will look for the files oscillators.rtf, oscillators.txt, oscillators.sc or oscillators

it expects to find in one of those files an Instr named "oscillators.sin"

The older form **array notation** also works:

```
Instr  \oscillators \sin
```

Instr(name,function,inputSpecs.outputSpec);

name - \sin
"oscillators.sin"
in file oscillators
"folder.subfolder.oscillators.sin"
in folder/subfolder/oscillators
[\oscillators, \sin]

function - the Instr's ugenFunc

When using your Instrument with Patch THERE IS NO NEED TO USE Out.ar
though you may explicitly use it if you wish.
It will be appended to your ugen graph func if needed.

inputSpecs
Specify what kind of input is required, and the working range of values.
somewhat optional - these can be guessed from the argnames.

see [Spec] and [ControlSpec]

if no spec is supplied, Instr will use the function's argument name to
lookup a spec in Spec.specs. eg arg freq -> looks for Spec.specs.at(\freq)
If nothing is found there, it will default to a ControlSpec with a range of 0 .. 1

These specs are used by Patch to determine the suitable type of input.
They may be used in many other situations, and you will find many uses
where you will wish to query the specs.

The default/initial value for the Spec is taken from the functiondefaults.

different lazy ways to specify the spec...

```
(
Instr("minimoog.one",{ arg freq=440,int1=5,int2 = -5,
width1=0.1,width2=0.1,width3=0.1,
ffreqInterval=0,rq=0.4;
```

```
var p;
p=Pulse.ar([ freq  * int1.midiratio, freq, freq * int2.midiratio],
[ width1,width2,width3],0.3);
RLPF.ar(Mix.ar(p),freq * ffreqInterval.midiratio,rq)
},#[
  nil  // nil, so use function's arg name (\freq)
   // to look up in Spec.specs
  [-48,48,\linear     // as(Spec,[-48,48,\linear,1])
       //   =>   ControlSpec.new(-48,48,\linear,1)
[-48,48,\linear,1],
  \unipolar    // try Spec.specs.at(\unipolar)
  nil     // try the argname width2, that fails,
      //  so the default is ControlSpec(0,1,\linear)
  \unipolar
[-48,48,\linear,1]
]);
)
```

outSpec

optional - InstrSynthDef can determine the outputSpec by evaluating the ugenGraph and finding what the spec of the result is.
An Instr can be .ar, .kr or can even return an object, a player, or a pattern.

Playing

```
(
Instr.new("minimoog.two",{ arg freq=440,int1=5,int2 = -5,
width1=0.1,width2=0.1,width3=0.1,
ffreqInterval=0,rq=0.4;
var p;

p=Pulse.ar([ freq  * int1.midiratio, freq, freq * int2.midiratio],
[ width1,width2,width3],0.3);

RLPF.ar(Mix.ar(p),freq * ffreqInterval.midiratio,rq)

    // specs
 \freq
```

```
[-48,48,\linear,1],
[-48,48,\linear,1],
 \unipolar
 \unipolar
 \unipolar
[-48,48,\linear,1]
]);

)
```

```
(
 Instr.at("minimoog.two").play
)

(
{
"minimoog.two".ar( LFNoise1.kr(0.1,300,700) )
}.play
)

(
{
Instr.at("minimoog.two")
.ar( LFNoise1.kr(0.1,300,700) );
}.play
)

(
{
Instr.ar(
"minimoog.two",
[ LFNoise1.kr(0.1,300,700) ]
);
}.play
)
```

but by far the best way is to use Patch :

```
Patch("minimoog.two",[1000]).play
```

```
Patch "minimoog.two"
```

Patterns

```
(
Instr([\minimoog,\two],{ arg freq=440,int1=5,int2 = -5,width1=0.1,width2=0.1,width3=0.1,
ffreqInterval=0,rq=0.4;
var p;

p=Pulse.ar([ freq  * int1.midiratio, freq, freq * int2.midiratio],
[ width1,width2,width3],0.3);

RLPF.ar(Mix.ar(p),freq * ffreqInterval.midiratio,rq)
* EnvGen.kr(Env.perc,doneAction:  2)

});

 p = Patch  \minimoog \two    // no args, Patch automagically creates KrNumberEditors

SynthDescLib.global.read;
d = p.asSynthDef.store;

 Pbind
\instrument, d.name,
  // note is converted to freq by things in NotePlayer
\note,Prand([10,20,30],inf),
  // args are passed into the function
\int1, Prand([-3,0,7,11,13],inf)
).play
)
```

see also InstrGateSpawner and InstrSpawner

An Instr is not a SynthDef

An Instr creates an InstrSynthDef (a subclass of SynthDef)

Each argument in the function for a SynthDef creates a Control input to the Synth that will eventually play on the server.

An Instr can also include extra arguments that will be used in building the synth def, but will not be Control inputs in the final synth.

For instance an Integer may be passed in:

```
// caution: mind the feedback. AudioIn
(
Instr(\qAllpassA,{ arg audio,decay=1,maxDelay=0.3, quantity=4,chanDiff=0.1;

( quantity.asInteger).do({
var x;
audio =
AllpassL.ar(audio, maxDelay,
[rrand(0.01,maxDelay),rrand(0.01,maxDelay)],
decay)
});
audio
});

Patch \qAllpassA
{ AudioIn.ar([1,2]) },
1,
0.3,
8
]).play

)
```

The first input to the synth is a stereo audio rate input, the others were fixed floats that did not require synth inputs.

Envelopes, fixed floats, fixed integers, Samples etc. can be passed into Instr functions.

When Patch is used to specify the inputs to the function some of these inputs will be reduced to fixed values (integers, floats, Envelopes etc.), and the resulting SynthDef will contain those inputs hard-coded. Using different Patches, it is possible to write many SynthDefs from the same Instr.

```
Instr(\rlpf,{ arg input,ffreq=400,rq=0.1;
RLPF.ar( input, ffreq, rq )
});
```

If the input supplied is stereo, the synthDef produced will be stereo.

```
(
 Patch \rlpf
Patch({ Saw.ar([400,440],0.1) }) // in stereo
]).play
)
```

It is possible to play another Instr inside of your Instr:

```
(
Instr(\sawfilter,{ arg freq,ffreq,rq;
Instr.ar(\rlpf, [ Saw.ar(freq,0.1) , ffreq, rq ])
})
)
```

and thus get further reuse out of your library of functions. Here the \rlpf that is used inside doesn't produce a synth def, but is used as a simple function in the ugenGraph of the \sawfilter Instr which does make a synthDef.

It is not generally possible to use the .ar method on a player inside of an Instrument function. This was possible in sc2. You cannot use a sound file player in this way:

sfp = SFP("path/to/soundfile");

```
Instr('no-can-do',{ arg sfp,amp=1.0;
sfp.ar * amp
});
```

Because an SFP (soundfile player) will require a buffer, a bus, and various stages of preparation. It is a complex process that cannot be compiled into a SynthDef.

the better approach is :

```
Instr("can-do",{ arg input,amp=1.0;
input * amp
});
```

```
Patch("can-do",[
SFP("path/to/soundfile")
])
```

.gui

The gui for Instr is a simple display of the arguments and specs.

```
  // default gui display for Instr
Instr.at("minimoog.one").gui
```

see [Patch] [InstrGateSpawner]

ID: 143

InstrAt

a compile-string-saveable, reloadable reference to an Instr by its name

a Patch already saves the name of the Instr as its means of addressing it.

InstrAt could be used in Pbind or Pwhile or anywhere a { } function is needed.

ID: 144

InstrSpawner

superclass: Patch

InstrSpawner(instr , args, delta)

instr - as per Patch, may be a function or an Instr name.
args - as per Patch, nil args will be auto-created.
args that are Players will play continously in their own synths and be patched into each spawn event synth.
args that are of rate \stream (all Patterns) will be streamed.
args that are of rate \scalar (floats, Envs, samples) will be passed into the instr function and are subsequently fixed.
delta - a float or pattern.
in **seconds**
see **InstrGateSpawner** for beats and for variable legato

```
// start and pchRatio are streamed
(
InstrSpawner({ arg sample,start=0,pchRatio=1.0,env;
 PlayBuf
sample.numChannels,
sample.bufnumIr,
pchRatio,
1,
start * sample.bufFramesIr,
1
) * EnvGen.kr(env,doneAction:  2)
},[
 Sample "a11wlk01.wav"
Pbrown(0,1,0.1,inf),
 Prand
Array.fill(4,{
Pseries(rrand(-20,30),[2,-2].choose,rrand(5,20))
}),inf).midiratio,
Env.sine(0.2,0.4)
```

```
],0.06).play

)


// pchRatio will not stream, is fixed at -1
(

InstrSpawner({ arg sample,start=0,pchRatio=1.0,env;
PlayBuf.ar( sample.numChannels, sample.bufnumIr,pchRatio,1,start * sample.bufFramesIr,1 )
* EnvGen.kr(env,doneAction:  2)
},[
 Sample "a11wlk01.wav"
Pbrown(0,1,0.1,inf),
-1,
Env.sine(0.2,0.4)

],0.125).play

)
```

the Patchin the width input plays continuously and is patched into each spawn event

```
(
InstrSpawner({ arg freq,rq,width,fenv,fenvmod,envperc;
width.debug("width");  // an OutputProxy
RLPF.ar(
Pulse.ar(
freq,
width
),
EnvGen.kr(fenv,levelScale:  fenvmod),
rq)
* EnvGen.kr(envperc, 1.0,0.3,doneAction:  2)
},[
Pfunc({  15.rand.degreeToKey([ 0, 2, 3, 5, 7, 8, 10 ]).midicps * 3 }),
0.1,
Patch({ FSinOsc.kr(0.05).range(0.01,0.99) }),
Env.asr,
6000,
```

```
Env.perc(releaseTime:  0.8)
],0.125).play
)
note:  for beats see InstrGateSpawner
```

the stereo Patchin the width input causes the InstrSpawnerto expand to stereo

```
(
InstrSpawner({ arg freq,rq,width,fenv,fenvmod,envperc;
width.debug("width");  // an OutputProxy
RLPF.ar(
Pulse.ar(
freq,
width
),
EnvGen.kr(fenv,levelScale:  fenvmod),
rq)
* EnvGen.kr(envperc, 1.0,0.3,doneAction:  2)
},[
Pfunc({  15.rand.degreeToKey([ 0, 2, 3, 5, 7, 8, 10 ]).midicps * 3 }),
0.1,
Patch({
[ FSinOsc.kr(0.05,0.0).range(0.01,0.99),
FSinOsc.kr(0.05,0.5).range(0.01,0.99),
]
}),
Env.asr,
6000,
Env.perc(releaseTime:  0.8)
],0.125).play
)
```

```
(
Instr(\InstrSpawner,{ arg freq=1000,amp=0.1,env;
SinOsc.ar(freq,mul:  amp)
```

```
* EnvGen.kr(env,doneAction:  2)
});

    InstrSpawner \InstrSpawner
Pbrown(45,90,3,inf).midicps,
0.1,
    Env      // does not get streamed
],
0.1
);

i.play

)


sliders are polled on the gui
(

Instr(\InstrSpawner,{ arg freq=1000,amp=0.1,env;

SinOsc.ar(freq,mul:  amp)
* EnvGen.kr(env,doneAction:  2)
});

InstrSpawner \InstrSpawner
    nil, // accept a default KrNumberEditor
    nil  // accept a default KrNumberEditor
    Env      // does not get streamed
],
NumberEditor(0.1,[0.05,1.0]) // polled each time
).gui
)




// how to get eventCount like sc2 Spawn
(
```

```
InstrSpawner({ arg eventCount=0, freq,rq,width,fenv,fenvmod,envperc;

  // do something with eventCount if you need it...

RLPF.ar(
Pulse.ar(
freq,
width
),
EnvGen.kr(fenv,levelScale:  fenvmod),
rq)
* EnvGen.kr(envperc, doneAction:  2)
},[

  Pseries     inf   // infinite counting

  //aeolian
Pfunc({  15.rand.degreeToKey([ 0, 2, 3, 5, 7, 8, 10 ]).midicps * 3 }),
0.1,
Patch({ LFTri.kr(0.1,[0.0,0.5],0.5,0.5) }),
Env.asr,
6000,
Env.perc(releaseTime:  0.1)
],0.25).play
)
```

this is more flexible, is only on when you need it, and lets you do wrapping or scaling etc.
of the event count all in the pattern domain.

ID: 145

InstrSpawner2

InstrSpawner2.new(name,args,noteOn, beatsPerStep,tempo)

name
the instr name
args
each argument is taken .asStream and the stream is iterated during play

noteOn is a stream of values meaning:
1 noteOn
arg streams are iterated and sent to a new synth
0 rest
-1 legato
arg streams are interated and sent to the last synth

beatsPerStep (default 0.25)
how many beats to wait between each step
tempo (default is global Tempo)
the Tempo object used for conversions

```
(

Instr(\InstrSpawner,{ arg freq=1000,amp=1.0,env;

Saw.ar(freq,mul:  amp)
* EnvGen.kr(env,doneAction:  2)
});

    InstrSpawner2 \InstrSpawner
Pbrown(40,90,3,inf).midicps,
0.2,
    Env      // does not get streamed
],
Pseq([1,-1,-1,0,0,0,0,0,0,1,0,0,0],inf),
   0.25 // 16th notes
```

```
);

z.play;

)

z.stop;


z.gui
```

ID: 146

InstrSynthDef

this is how Patch performs its magic.

how it works :

InstrSynthDef.build(instr, objects, outClass)

each object is asked to initForSynthDef(synthDef,argIndex)
BufferProxy classes use the argument index to build a unique key
so they dont conflict with other BufferProxies.
all other classes need do nothing.

each object is asked to addToSynthDef(synthDef,argName)
depending on the objects rate it asks the synthDef to
addKr(argName,value,lag)
addIr(argName,value)
addInstrOnlyArg(argName,value)

the object may choose which initial value to pass. if a Player does not yet
know its bus assignment, it can safely pass a 0 as it will be asked what bus
it is playing on when the actual synth is being started.
objects such as Env or Float or Array will pass themselves as an instrOnly arg,
thus fixing them in the synthDef
objects such as Sample can be used in the course of the ugenGraph evaluation
to add 'secret' additional inputs. (see below)

the synthDef builds Control objects from the kr and ir inputs.

each object is asked to return an instrArgFromControl(control,argIndex)
Players return In.ar(control, this.numChannels) or In.kr
the control is a kr OutputProxy that indicates the bus that input
should listen to.
KrNumberEditors return either the raw control or wrap it in a Lag
Stream2Trig returns an InTrig.kr(control,1)
Object returns itself
so a Sample or an Env is passed into the instr function, and in
fact the control it was passed is nil since it would have used

addInstrOnlyArg

the instr function is valued with those objects.
during the course of the ugenFunc evaluation,
BufferProxy objects (Sample etc.) may request to addSecretIr or addSecretKr
arguments. this is to request additional inputs to the synth that do not
have arguments in the instr function. thus a Sample can pass itself to the
instr function as an object, and then request a synth input that will be used
to get the bufnum dynamically passed in when the synth actually plays.

this can also be used to request a tempo bus input (not yet implemented)

the Out.ar and a kr control of the name \out are added bassed on the rate and numChannels
of the result of the ugenFunc.

the unique synthDef name is calculated. the name should represent the instr name
and the fixed inputs that were used to evaluate it. any Patch that uses the
same fixed inputs (SimpleNumbers) should be able to share the synthDef,
and all the variable inputs should be of compatible rate and numChannels.

ID: 147

Interface

This sets up an environment in which you can build a player,
build a gui for that player, and respond to midi and keyboard control.

The gui is quite optional, and in fact non-screen-staring is one of its primary goals.

GUI

You can set a custom gui function.
This can use any combination of .gui style and normal SCViews
The Interface can be placed on any other windows of any style.
You may decline to customize your gui.

MIDI

If you set any of these handler functions:

`onNoteOn`

`onNoteOff`

`onPitchBend`

`onCC`

then appropriate midi responders will be enabled when the player starts to play and disabled when it stops. This includes if the player is being started/stopped by external mixers, PlayerPool etc.

KeyDown/KeyUp

`keyDownAction`

`keyUpAction`

(only when guied, only when focus is within the MetaPatch's views)

Interface is great for having no gui at all. Personally I use the gui stuff to set up parameters for performing, and then when performing I use no gui, only MIDI controllers and key commands.

simplistic example

```
(
```

```
    Interface
 // an environment is in place here
freq = KrNumberEditor(400,[100,1200,\exp]);
syncFreq = KrNumberEditor(800,[100,12000,\exp]);
amp = KrNumberEditor(0.1,\amp);

Patch({ arg freq,syncFreq,amp=0.3;
SyncSaw.ar(syncFreq,freq) * amp
},[
freq,
syncFreq,
amp
])

});

// setting the gui
m.gui = { arg layout,metaPatch;
var joy;

 // the same environment is again in place
freq.gui(layout);

 ActionButton        "twitch"
var x,y;
  // action button now remembers the environment !
freq.setUnmappedValue( x = 1.0.rand );
syncFreq.setUnmappedValue( y = 1.0.rand );
joy.x_(x).y_(y).changed;
});

joy = SC2DSlider(layout, 100 @ 100)
.action_({ arg sl;
   // at this time not in environment
   metaPatch.use({ // use the metaPatch's environment
freq.setUnmappedValue(sl.x);
syncFreq.setUnmappedValue(sl.y );
})
});
```

```
EZNumber(layout,30 @ 30,"amp",[0.01,0.4,\exp],{ arg ez;
metaPatch.use({
amp.value_(ez.value).changed;
})
});

};

// creating a gui
m.gui



)
```

You can place them on any window

```
(
w = SCWindow.new("other",Rect(20,20,700,200));
w.front;

g = m.gui(w,Rect(30,30,500,200));

g.background = Color.blue(alpha:0.4);
)
```

MIDI handler installed on play

takes some seconds to start up, then play your midi keyboard

```
(

Instr([\klankperc,\k2a],{ arg trig=0.0,sfreqScale=1.0,sfreqOffset=0.0,stimeScale=1.0,foldAt=0.1;
Klank.ar(
`[
FloatArray[ 87.041, 198.607 ],
nil,
FloatArray[ 0.165394, 0.15595 ]
],
K2A.ar(trig),
sfreqScale,sfreqOffset,stimeScale
).squared.fold2(foldAt)
```

```
},[
 nil
[0.01,100],
[0,10000],
[0.01,100]
]);

// Create 5 patches, cycle through them on each midi key
Interface({ arg quantity=5;

quantity = quantity.poll;
a = Array.fill( quantity,{ arg i;
  Patch      \klankperc \k2a
[
   BeatClockPlayer
i * (3.midiratio),
i * (3.midiratio),
1.0,
foldAt = KrNumberEditor(0.1,[1.0,0.01])
]);
});

pool = PlayerPool(  a,
env:  Env.asr(0.01,releaseTime:  0.01),
round:  0.25);


}).onNoteOn_({ arg note,vel;
 // the same environment is in place here
 // foldAt.setUnmappedValue(vel / 127.0).changed;
pool.select(note %  quantity)
}).play

)
// fast triggering still trips it up.   working on it.
```

Simple CC example

```
(

Interface

freq = KrNumberEditor(400,[100,1200,\exp]);
syncFreq = KrNumberEditor(800,[100,12000,\exp]);
amp = KrNumberEditor(0.1,\amp);

Patch({ arg freq,syncFreq,amp=0.3;
SyncSaw.ar(syncFreq,freq) * amp
},[
freq,
syncFreq,
amp
])

}).onCC_({ arg src,chan,num,value;
if(num == 80,{   freq.setUnmappedValue(value/127);});
if(num == 81,{   syncFreq.setUnmappedValue(value/127);});
if(num == 82,{   amp.setUnmappedValue(value/127);});
})
.play

)
(

Interface

freq = KrNumberEditor(400,[100,1200,\exp]);
syncFreq = KrNumberEditor(800,[100,12000,\exp]);
amp = KrNumberEditor(0.1,\amp);

Patch({ arg freq,syncFreq,amp=0.3;
SyncSaw.ar(syncFreq,freq) * amp
},[
freq,
syncFreq,
amp
])
```

```
}).onCC_(
 ResponderArray
  // these normally install themselves immediately, but the Interface will be handling that
CCResponder(80,{ arg value;  freq.setUnmappedValue(value/127);},install:  false),
CCResponder(81,{ arg value;  syncFreq.setUnmappedValue(value/127);},install:  false),
CCResponder(82,{ arg value;  amp.setUnmappedValue(value/127);},install:  false)
)
)
.play

)



(
// beat juggler

Interface   arg

beatStart1 = NumberEditor(0.0,[0.0,8.0,\lin,0.25]);
beatStart2 = NumberEditor(0.0,[0.0,8.0,\lin,0.25]);
durations = [ 2.0,2.0];

patch = InstrGateSpawner({ arg sample,dur, pchRatio,beatStart,amp=0.3,envadsr,tempo;

var gate;
gate = Trig1.kr(1.0,dur / tempo);

pchRatio = pchRatio * sample.pchRatioKr;
beatStart = beatStart * sample.beatsizeIr;

PlayBuf.ar(sample.numChannels,
sample.bufnumIr,
pchRatio,
1.0,
beatStart,
1.0)

* EnvGen.kr(envadsr, gate,amp,doneAction:  2 )
},[
```

```
sample,
   // dur uses a Pfunc to ask the delta till the next event
Pfunc({ arg igs;  (igs.delta * 0.9) }),

   // get an .ir input into the synth function
pchRatio = IrNumberEditor(1.0,[-2,2,\lin,0.25]),

   // patterns naturally create an .ir input
   Pswitch1
beatStart1,
beatStart2
   ],Pseq       inf   // juggle back and forth

],
  // stream of beat durations
Pseq( durations,inf));

patch

},[
 // a blank sample
 Sample     nil
])
        arg        // we are given a FlowView

var env,ddsp,bdsp;

 CXLabel        "Click the sample path (nil) to browse for a sample.  You can choose new samples even
while playing."
layout.startRow;
 /* the environment from the build function above is available here */
sample.gui(layout,500@100);

 /* but when view actions fire you will be in a different environment
  so save it here in a variable for use later */
env = currentEnvironment;

 // .vert returns an SCVLayoutView so we can stack each 2d over its caption
layout.vert({ arg layout;
SC2DSlider(layout,100@100)
```

```
.action_({ arg sl;
env.use({
     // set a 0..1 value, map it to the spec ranges of the NumberEditors
beatStart1.setUnmappedValue(sl.x);
beatStart2.setUnmappedValue(sl.y);
bdsp.object_( [ beatStart1.value, beatStart2.value]).changed;
})
});
SCStaticText(layout,100@13).object_("Beat starts:");
bdsp = SCStaticText(layout,100@13).object_([ beatStart1.value, beatStart2.value].asString);
},100@120);

layout.vert({ arg layout;
SC2DSlider(layout,100@100)
.action_({ arg sl;
env.use({
var stride,part;
stride = 2 ** [3,4,5,6].at((sl.x * 3).asInteger) * 0.125;
part = (stride * (1.0 - sl.y)).round(0.25).clip(0.25,stride - 0.25);
durations.put(0,part);
durations.put(1,stride - part);
ddsp.object_( durations.sum.asString + "=" +  durations).changed;
});
});
SCStaticText(layout,100@13).object_("beats");
ddsp = SCStaticText(layout,100@13).object_( durations.sum.asString + "=" +  durations);
},100@120);
CXLabel(layout,"pchRatio:");
pchRatio.gui(layout);
})
.gui

)
```

<>onCC

the control change handler is installed (via CCResponder) when play starts and unistalled when

play stops.

It can be a simple function:

```
interface.onCC = { arg src,chan,num,value;
[num,value].postln;
};
```

a CCResponder that responds on a specific number.
(note: tell it NOT to install itself, because the Interface
will install and uninstall it when play starts or stops)

```
onCC = CCResponder(num,{ },install:  false);
```

or a custom class:

```
      KorgMicroKontrolCC
[\slider,0,{  }],
[\slider,1,{  }],
[\encoder,0,{  }],
[\encoder,1,{  }],
[\x,{ }],
[\y, { }]
);
```

whatever it is will be asked to respond to 'value' :

```
thing.value(src,chan,num,value);
```

<>onPlay
<>onStop
<>onFree

```
(

Interface
```

```
freq = KrNumberEditor(400,[100,1200,\exp]);
amp = KrNumberEditor(0.1,[0.01,0.4,\exp]);

Patch({ arg freq,amp;
SinOsc.ar(freq) * amp
},[
freq,
amp
])

})
.onPlay_({
"playing".postln;
})
            // also on command-.
"stopping".postln;
})
.onFree_({
"freeing".postln;
}).play

)
```

InterfaceDef

the function that builds the player is actually an InterfaceDef. These can be created and stored in the same fashion as Instr and kept in the same folder. You can then address them by name, supply paramaters as you do for Patch and you will get an Interface which will use the gui and midi functions from the InterfaceDef.

ID: 148

Patch

A Patch connects an Instr function with input arguments to that function.

```
Patch(instr,args)

(
Instr(\bubbles, { arg amp=1.0;
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4);
zout * amp
});

    Patch \bubbles

// default server will be booted, def written and loaded
p.play;

)



p.stop;

// command-.  will also stop all sounds

p.play;

p.run(false);

p.run(true);

p.insp; //inspect it

p.gui;

// close the window
```

```
// open it again
p.gui;
```

```
(
Instr(\bubbles, { arg amp=1.0;
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4);
zout * amp
});

Instr(\rlpf,{ arg audio=0,freq=500,rq=0.1;
RLPF.ar(audio, freq, rq)
});

p = Patch(\rlpf,[
 q = Patch \bubbles
]);

p.gui

)
```

Instr can be specified as

an Instr

```
(
i = Instr("help-Patch",{ arg freq=100,amp=1.0;
SinOsc.ar([freq,freq + 30],0,amp)
});
p = Patch(i,[ 500, 0.3 ]);
p.gui
)
```

a Function

```
(
p = Patch({ arg freq=100,amp=1.0;
SinOsc.ar([freq,freq + 30],0,amp)
},[
500,
0.3
]);
p.gui
)
```

or by the Instr name

```
// the Instr stores itself when created
(
Instr(\bubbles, { arg amp=1.0;
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4);
zout * amp
});

 // the patch retrieves it
p = Patch(\bubbles,[0.4] );
p.gui
)
```

Patch can be easily saved to disk, and can make use of a large library of Instr functions. An Instr can produce many different possible SynthDefs, expanding the number of output channels or , varying in output rate or number of channels or internal structure.

Automatic Input Creation

For any nil arguments, a default control will be created. The spec returns this from the defaultControl method.

ControlSpec : KrNumberEditor
StaticSpec : NumberEditor (for quantities or max delay times etc.)
EnvSpec : EnvEditor

SampleSpec : Sample

see the implementations of defaultControl

This gives the impression that Patch is "an automatic gui for an Instr / SynthDef". If you do not supply arguments, it will make default ones, simple ones, **but the real power of Patch is to supply functions with complex and varied inputs**. Sitting there with 5 sliders on a 1 dimensional Instrument isn't really the height of synthesis.

Most simple synth inputs (ControlSpec) will end up being KrNumberEditors. Setting their values will control the playing synth.

```
aPatch.set( index, value)
```

I recommend experimenting with factory methods to create your patches, supplying them with inputs useful for what you are working on.
If you use a certain physical controller or wacom :

```
buildPatch  = { arg instrName;
var i;
i = Instr.at(instrName);

Patch(instrName,[
{ i.specAt(0).map( JoyAxis.kr(0,1,axis:5) ) },
{ i.specAt(1).map( JoyAxis.kr(0,1,axis:5) ) },
])
};

buildPatch.value( \boingboing );
```

you can even keep your factories themselves in Instrument libraries:

```
Instr(\joysticker,[ arg instrName;

var i;
i = Instr.at(instrName);
```

```
Patch(instrName,[
{ i.specAt(0).map( JoyAxis.kr(0,1,axis:5) ) },
{ i.specAt(1).map( JoyAxis.kr(0,1,axis:5) ) },
])
});


patch = Instr(\joysticker).value( \simple );
```

this Instr is not used for audio, its just used to build and return a Patch

You could choose different controllers for different common inputs,
you can query the argument name and the spec.
Keep files in databases, load other Patches or soundfiles from disk.
You could flip coins and choose from soundfiles, audio in, other saved patches or randomly chosen net radio feeds.

Patch inside Patch

```
(
// lets collect the cast...
Instr(\bubbles, { arg amp=1.0;
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4);
zout * amp
});


// note that the same function will work as stereo or mono
// depending on what gets put into it
Instr(\rlpf,{ arg audio=0,freq=500,rq=0.1;
RLPF.ar(audio, freq, rq)
});

// put bubbles into the filter
p = Patch(\rlpf,[
 q = Patch \bubbles
```

```
]);

)


// watch the ugen count in the default server window
// and also the error window results

p.play;



// stops the parent and the child q
p.stop;


// allocates new everything
p.play;

// additional play
// does not start an additional process
p.play;



// stop q, but the filter p is still reading from its old bus
q.stop;

// should still have 9 ugens playing

// sent the play message, q defaults to play out of the main outputs
// not through the filter p
q.play;

// stopping p now still stops q because it is still a child of p
p.stop;

// replaying the whole structures
p.play;
```

```
// note the AudioPatchOut and the inputs:  PatchIn classes
p.insp;
q.insp;
```

Fixed Arguments

Floats and other scalar values including Envelopes, are transparently dealt with by Patch. These items are passed to the Instr function, but not to the SynthDef or the Synth. They are not modulateable.

```
(
// fixing arguments

Instr([\jmcExamples,'moto-rev'],{ arg lfo=0.2,freq=1000,rq=0.1;
RLPF.ar(LFPulse.ar(SinOsc.kr(lfo, 0, 10, 21), [0,0.1], 0.1), freq, rq).clip2(0.4);
});

    Patch  \jmcExamples 'moto-rev'
0.2
]);

q.gui;

)
```

You can design Instr to take parameters that are used only in the building of the SynthDef. This can be used to select from different kinds of filters or to .

```
Instr(\upOrDown, {arg upDown=0;
var line;
if (upDown>0,
  {line = Line             // upDown>0 ==> pitch goes up
  {line = Line             // upDown 0 or less ==> pitch goes down
```

```
);
SinOsc.ar(440*line,0,0.2);
},[
 StaticIntegerSpec
]);

Patch(\upOrDown, [ 0]).play
```

The upDown param acts as a switch between different synth def architectures. If your Instr library is well designed you can acheive very sophisticated sound structures with automatic optimizations and code reuse.

Note that the Patch would assume upDown to be a modulatable control input (with a default of 0.0) without the StaticIntegerSpec making it clear that its a static integer.

Busses

```
(

s.boot;

a = Group.new;

b = Group.after(a);

    Bus.audio(s,1);

p=Patch({ arg in,ffreq;
// the Bus is passed in as In.ar(bus.index,bus.numChannels)
LPF.ar(in,ffreq)
},[
c,
KrNumberEditor(3000,[200,8000,\exp])
]).play(group:  b);

// play something onto this bus in a group before that of the filter
y = Patch({ Saw.ar(400) * 0.1  }).play(group:  a, bus:  c );


z = Patch({ Saw.ar(500) * 0.1  }).play(group:  a, bus:  c );
```

```
z.stop;

y.stop;

)

// you can make the bus play to a main audio output
c.play

//command-.  to stop all


(
s.boot;

a = Group.new;

b = Group.after(a);

// no index, not yet allocated
c = Bus(\audio,nil,2);

y = Patch({ arg in,ffreq;
LPF.ar(in,ffreq)
},[
 c, // a proxy, the bus is yet to be allocated
KrNumberEditor(3000,[200,8000,\exp])
]).play(group:  b);

// now that the patch has played, the bus allocated itself
c.insp

// play onto this bus in a group before that of the filter
z = Patch({ Saw.ar([400,401]) * 0.1  }).play(group:  a, bus:  c )
```

Mapping values

you can use a spec to map a signal :

```
(
var spec;
spec = [ 100,18000,\exp].asSpec;
{

SinOsc
spec.map(  SinOsc.kr(0.1).range(0,1) )
)

}.play

)
```

you can use that as an input to a patch:

```
(
var spec;
spec = [ 100,18000,\exp].asSpec;

Patch({ arg freq;
SinOsc.ar(freq)
},[
spec.map( { SinOsc.kr(0.1).range(0,1) } )
]).play
)
```

or map another player's output and then use that as an input to a patch :

```
(
var spec;
spec = [ 100,18000,\exp].asSpec;

Patch({ arg freq;
SinOsc.ar(freq)
},[
spec.map( Patch({ SinOsc.kr(0.1).range(0,1) }) ).debug("i am a")
]).play

)
```

so spec.map is taking the player (which is a kind of function :
a potential piece of music once it is valued)
and creating a BinaryOpFunction out of it.
that is to say if you do math on functions you get another function.

Spawning

```
still wrong.   a should be playing already and b should just patch it in each time.
 //
//
//a = Patch({
// SinOsc.ar(800,0.0)
//});
//
//c = Bus.audio;
//a.play(bus:  c);
//
//b = Patch({ arg tone;
// var gate;
// gate = Trig1.kr(1.0,0.25);
// tone = In.ar(tone,1);
// tone * EnvGen.kr(Env([0,1,0],[0.05,0.05],\welch,1),gate,doneAction:  2)
//}[
// c.index
//]);
//
//b.prepareForPlay(s);
//
//
//Routine({
// 1.0.wait;
// 100.do({
//  b.spawn(atTime:  0.1);
//  0.25.wait
// })
//}).play(SystemClock)
```

```
//
)
```

ID: 149

StreamSpec

superclass: HasItemSpec

StreamSpec(specOfItemsReturnedInTheStream)

a StreamSpec specifies an input that will be used in a stream or pattern.
The default control is an IrNumberControl, though usually you will be more
intereseted in using Patterns as inputs.

The most common use is for InstrSpawner and InstrGateSpawner. An IrNumberControl
or a Pattern (any object that returns a rate of \stream) will result in the creation
of an .ir rate input to the synth function. Then on each spawning, the synth
is created and on that .ir rate input is passed in the next value of the stream.

```
StreamSpec( [ 0.01, 8.0,\exp ] )
```
a stream of values between 0.01 and 8.0
any control should use an exponential fader

```
StreamSpec  EnvSpec Env
```
a stream of envelopes.
the default envelope is an Env.linen, though
the stream may return any kind of envelope.

6.6 Introspection

ID: 150

TestCase Unit Testing

Rather than write full classes that test each method of your class, this uses a simple instance.

```
TestCase Object
'==' -> {
var a;
a = Object.new;
  a == a // test passes if it returns true
}
);
```

TestCases for each class can be stored in "TestingAndToDo/Tests" as Classname.test.rtf (note that if you check "hide extension" it will just say Classname.test).

Any class can find and run its test:

```
Float.test;
```

If not found, a message is posted.

All classes can try and run their tests if they have them:

```
TestCase.runAll;
```

An individual test case that you are working on can be run:

```
     TestCase Object
'!==' -> {
var a;
a = Object.new;
   a !== a // a deliberate failure
}
).run;
```

You can click to open the class file or the test file.

6.7 Miscellanea

ID: 151

CrucialLibrary

Higher level interfaces for managing musical objects.

AbstractPlayer
This encapsulates things that play. you can stop them, start them, record them and save them
to disk. They handle all the messy details of loading their resources to the server, and getting
them off again when you are done.

Patch
The most common and well known Player. This patches inputs into a function. Inputs can be floats,
other players, Envs (envelopes), Samples, or SFP (sound file player).
A Patch plays continously and infinitely (until you stop it).

InstrSpawner, InstrGateSpawner
a Patch that plays successive events. The inputs are taken as streams and interated.

Interface
specify a player and interface elements and how they are connected.
allows custom gui and midi functions.

introspection tools
.insp - a debugging tool
"any object".insp
class browser
AbstractPlayer.gui

It is not advisable to use "the whole library"–don't think of it as a single entity. Try out one or two things and work them into what you are normally doing; as time goes on, learn some more objects.
It should not be an all or nothing commitment. If you perceive it as that, then you are approaching it wrong.

Although there is a convienient system for instantly making guis for instrument functions, **the purpose of the library is not "a gui system".** It is flexible, and can be

used in any coding situation.

INSTALLATION

crucial is distributed with SC and should not needed further installation. frequent updates are maintained through CVS.

[double click to select, command-h to open help file]

Instr
Patch

InstrSpawner
InstrGateSpawner

Interface

gui

SynthConsole

NumberEditor

AbstractPlayer

StreamKrDur
Tempo
TempoBus
TempoPlayer
BeatClockPlayer
Stream2Trig
PlayerEfxFun

PlayerPool

What you can do for Me:

send me a copy of any record/CD you put out using this stuff
write Instr and offer them to me or to everyone
Write Usable Classes and share them with me
suggest or implement improvements
build things i wouldn't build
identify issues
report bugs
fix bugs
hook me up with gigs. i rock the house. honest.
buy my album

ID: 152

Crucial

superclass: Object

If you are looking for the intro to the library, see [**CRUCIAL-LIBRARY**].

This class initializes some lightweight resources relevant to crucial library and loads some useful utilities into the Library (a shared dictionary of reusable functions).

```
Crucial.menu;
```

You can put this in Main if you like:

```
  run { // called by command-R
Crucial.menu;
}
```

You should set personal preferences in Main-startUp, as this file (Crucial.sc) will get overwritten
by CVS updates from time to time.

In Main-startUp you can set:

```
  // you can move all of your documents to your home directory
 Document       " /Documents/SuperCollider/"
  see Document

 Instr       " /Documents/SuperCollider/Instr/"
see Instr

 // this would allow your sounds to be shared by other applications
Sample.soundsDir = " /Sounds/";
see Sample
  // copy a11wlk01.wav to there for use in helpfiles !
```

everything is called in *initClass

preferences: Colors, root paths.

creates some useful common Specs

installs some general use Library functions.

see [**Library**]

6.8 Patching

ID: 153

Patching

PatchIn
AudioPatchIn
ControlPatchIn
ScalarPatchIn

PatchOut
ControlPatchOut
AudioPatchOut
ScalarPatchOut

These objects hold information about a Player's inputs and outputs. They are used to make connections to other Players, and to manage that connection and all subsequent repatching or disconnections.

PatchIn objects hold NodeControl objects that enable communication to one input of a Node (Synths and Groups), either for setting the value or mapping the input to read from a Bus.

PatchOut objects hold either Bus objects (for audio and control) or sources (for control and scalar). Bus objects hold the information about the Bus that the Player is playing onto. sources are client side objects that produce float values to send to the inputs of Nodes. Possible sources are gui objects like sliders or Patterns and other automatons. Anything that can respond to .value with a Float can be a source for a ScalarPatchOut.

aPatchIn.connectTo(aPatchOut)
aPatchOut.connectTo(aPatchIn)
connect the input on the node to the output of the bus or source.

Both PatchIn and PatchOut remember who they are connectedTo. PatchOuts may be connected to multiple PatchIns.

play(patchIn)

server

plays on top group at head
group
plays at group of head
integer
default server, audio output number

they should be even gui-able, a drag drop

maybe even a player makes ins and outs when created

prepare(group)
sets the patchOut group, where it will be created

patchIn
also hold source ?, search for synth

```
examples:
drag output of player to some input
patchOut.connectTo(patchIn)
sets nodeControl if playing,
else when it plays it will get the value

if player is not playing anywhere else
if input is reading from something else
insert mixer


drag filter to player

if player is not playing anywhere else
node after
same buss for out
filter reads from that buss

this is a kind of player

if player is playing to another filter
```

remove other

node after

same buss for out

filter reads from that buss

ID: 154

PlayerInputProxy

represents an audio or control input for a player that is to be dynamically patchable at runtime.
eg. an input to an effect.
a potential controller/interface input

```
 PlayerInputProxy(spec)


p = Patch({ arg audio,ffreq;
RLPF.ar(audio,ffreq)
},[
  PlayerInputProxy
]);
```

it holds the place. if the patch is played, it is silent.

```
p.play
```

other classes can detect these inputs and patch into them.

```
notes:

/*

XFadeEfxFunc     "Patches/breaks/felaoopkunnzz"
 Patch       'efxFilters3'  'combN'
  PlayerInputProxy
StreamKrDur.new(PbrownGUI.new(600, 1000, 100, inf), 8, 1), -0.02 ]
),
```

```
0.5,
1,
1)

*/
```

the XFadeEfxFunc would know numChannels and rate of input,
and sets it on the PlayerInputProxy

so pip can save blank

if pip is played (silent), it has defaults and would play it as mono

```
PlayerInputProxy.new.play
```

if the channels changes while playing, it would have to respawn

6.9 Players

1 Miscellanea

2 SFP

6.10 Sample

ID: 155

ArrayBuffer

superclass: BufferProxy

Passes an array into a Patch for use in UGens which need an array supplied as a buffer index.
If saved as a compile string to disk, saves the array with it.

ArrayBuffer.new(array)

```
(
// modal space - jmc
// mouse x controls discrete pitch in dorian mode

Patch({ arg scale;

var mix;

mix =

 // lead tone
 SinOsc
(
   DegreeToKey
scale.bufnumIr,
    MouseX            // mouse indexes into scale
    12,     // 12 notes per octave
1,     // mul = 1
    72     // offset by 72 notes
)
   + LFNoise1                // add some low freq stereo detuning
  ).midicps,      // convert midi notes to hertz
0,
0.1)

 // drone 5ths
+ RLPF.ar(LFPulse.ar([48,55].midicps, 0.15),
```

```
SinOsc.kr(0.1, 0, 10, 72).midicps, 0.1, 0.1);

 // add some 70's euro-space-rock echo
CombN.ar(mix, 0.31, 0.31, 2, 1, mix)
},[
ArrayBuffer(FloatArray[0, 2, 3, 5, 7, 9, 10])
]).gui

)
```

ID: 156

BufferProxy

Allocates and supplies a buffer for use in Patches. The buffer is unfilled, suitable for recording. See Sample (a subclass of BufferProxy) if you need to load soundfiles.

```
(

Instr([\recordPlay,\JemAudioIn], {arg buffer, input, trigRate = 0.5,offOn = 1, pitch = 1, start = 0;
var offset,trig;
trig = Impulse.kr(trigRate);
RecordBuf.ar(input,buffer.bufnumIr, run:  offOn,trigger:  trig);
offset = start * buffer.bufFramesKr;
PlayBuf.ar(buffer.numChannels,buffer.bufnumIr,pitch,trig,offset,loop:  1);

},#[
 \buffer  // == BufferProxySpec(44100,2)
 \stereo  // == AudioSpec.new(2),
[0.25,10,\linear],
 \unipolar
[-5,5,\linear],
 \unipolar
]);



Patch  \recordPlay \JemAudioIn
BufferProxy(44100 * 4, 2), // 4 secs in stereo
 AudioInPlayer
]).gui

)
```

Make sure your audio input and buffer numChannels match.

an argName of spec symbol of \buffer will create a BufferProxySpec with the default 44100 frames (1 second).
you can place a BufferProxySpec and specify any default size you would like.

this is the DEFAULT that will be used if you DON'T specify an input to a Patch. Usually you pass in a BufferProxy to the patch that is the size that you wish.

ID: 157

Sample

superclass: AbstractSample

This class can be used as an argument to a Patch. It will take care of all the troubles of loading, allocating, measuring, and even beat synchronizing of a small sound file. It will not clear the copyright.

```
Sample.new(soundFilePath,tempo)
```

It will not play by itself, but it holds all the intelligence to allow other things to play it very easily.

```
(

p = Patch   arg

PlayBuf.ar(sample.numChannels,
sample.bufnumIr,
sample.bufRateScaleKr,
1.0,
0.0,
1.0)

},[
Sample("a11wlk01.wav")
]);

p.gui;

)
```

Notice that the path to the sample is relative to the sounds/ directory, not to SuperCollider's own directory. You can set the Sample.soundsDir to the directory of your choice (eg, /Library/Sounds/ or /Sounds/). Copy a11wlk01.wav to your own sounds directory so you can still play examples.

Within the Instr function you use these methods on your Sample object

bufnumIr
at the start of the synth, this will get the dynamic bufferID of your Sample object. this Instr will reuse SynthDefs where possible. Multiple synths may use the same basic sample synthDef for many voices with no need to compile new SynthDefs and send to the server.

sampleRate
a float of the current sample's sample rate, embedded into the SynthDef as a constant. the def will be resuable for all samples of that sample rate, and will be slightly more efficient.
sampleRateKr
a kr rate signal that will change if you load a different sample into the buffer,even while playing.
sampleRateIr
a ir rate signal that will NOT change if you load a different sample into the buffer. use when you know the sample will not change, or if you know that all samples are the same sampleRate anyway.

bufRateScaleKr
the nominal pitchRatio value needed to play at the original pitch
bufRateScaleIr
the nominal pitchRatio value needed to play at the original pitch. will NOT change if you load a different sample into the buffer.

bufFramesKr
a kr rate signal with the number of frames of the current sample
bufFramesIr
an ir rate signal with the number of frames of the sample

bufSamplesKr
a kr rate signal with the number of samples of the current sample
bufSamplesIr
an ir rate signal with the number of samples of the current sample

duration
duration in seconds of current sample, embedded into SynthDef as a constant.
bufDurKr

duration in seconds
bufDurIr
duration in seconds

numChannels
integer, number of channels of the current sample. this will be embedded into the SynthDef as a constant. the SynthDef will still be reusable for all samples of the same numChannels.
bufChannelsKr
number of channels of the current sample. you cannot use this to modulate a PlayBuf.
bufChannelsIr
number of channels of the sample. you cannot use this to modulate a PlayBuf.

You can swap the samples while playing. Click on the name of the sample (in black font) and browse for a stereo sample. Then start play, and you can browse for more and change it while playing.

```
(

Instr("help-Sample",{ arg sample,pchRatio=0.50;
PlayBuf.ar(sample.numChannels,
    sample.bufnumIr, // finds the buffer number when the synth starts
sample.bufRateScaleKr * pchRatio,
1.0,0.0,1.0);
});

p = Patch "help-Sample"
 Sample "pick a stereo sample..."
]);

p.gui

)
```

The def name was : help-SampleO8NEut

You can build up a library of Instr functions and exploit them with Patch.

```
(
Instr([\help,\Sample],{ arg sample,pchRatio=1.0,start=0.0;
PlayBuf.ar(sample.numChannels,
    sample.bufnumIr, // finds the buffer number when the synth starts
sample.sampleRateKr / 44100 * pchRatio,
1.0,
start * sample.bufFramesKr,
1.0); // loop
});
)
```

Patch object:

```
(
 p = Patch  \help \Sample
[
  Sample "a11wlk01.wav"
]);

 // edit controls on the gui
p.gui
)
```

save it, and this will fully restore the complete sound.

```
(
Patch.new(
[ 'help', 'Sample' ], [ Sample.new("a11wlk01.wav", 1.6347258775994),
0.46511627906977, 0.17441860465116 ]
).play
)
```

BeatLock

**This will embed the sample's tempo into the SynthDef as a constant. Tempo's tempo can vary, but
what the monkey thinks the music in the sample is will remain fixed in the SynthDef.**

```
(

 //beatlock
Instr([\help,\Sample],{ arg sample;
  PlayBuf
sample.numChannels,
sample.bufnumIr,
sample.pchRatioKr,
1.0,0.0,1.0);
},[
  \sample
  \tempo
]);

 p = Patch  \help \Sample
[
  Sample "a11wlk01.wav"
]);

// move the tempo slider
p.gui

)

This is roughly equivalent to this:
(
 //beatlock
Instr([\help,\Sample],{ arg sample,tempo;
  PlayBuf
sample.numChannels,
sample.bufnumIr,
sample.sampleRateIr / 44100 * tempo * sample.tempo.reciprocal,
1.0,0.0,1.0);
},[
  \sample
```

```
  \tempo
]);


 p = Patch  \help \Sample
[
  Sample "a11wlk01.wav"
  TempoPlayer
]);

// move the tempo slider
p.gui

)
```

soundFilePath
end
signal.size - 1
the last indexable position in the signal
duration
totalMemory
numFrames * numChannels

The following methods are relevant if the sample is some kind of loop.

tempo
beats per second the original recording is regarded to have.
beats
number of beats
beatsize
the number of samples per beat

```
/***

(
// hit load and select a rhythm
// will stay beat locked and the beat will flow despite the cutting
q = rrand(8,32);

Patch({arg gate,env,sample,pchRatio;
 var pchRatioKr,start;
 pchRatioKr = sample.pchRatioKr * pchRatio;
 start = LFSaw.kr(GetTempo.kr * sample.beats.reciprocal, sample.end * 0.5, sample.end * 0.5);

 ReTrigger2.ar({
  PlayBuf.ar(sample.signal,sample.sampleRate,pchRatioKr,start.poll,0,sample.end);
 },gate,env,sample.numChannels)
},
[
 Stream2Trig(
  1.0,
  Pseq(Array.geom(8.rand,2 ** -5,  2.0).scramble,inf)
 ),
 Env.asr(release:0.1),
 Sample(":Sounds:floating_1"),
 StreamKrDur(
  Pslide(Array.series(q,0.0,4.0 / q),inf,rrand(3,5),rrand(1,6)),
  rrand(0.125,0.5)
 )
]).topGui

)


( // will stay beat locked and the beat will flow despite the cutting
q = rrand(8,32);
```

```
Patch({arg gate,env,sample,pchRatio;
 var pchRatioKr,start;
 pchRatioKr = sample.pchRatioKr * pchRatio;
 start = LFSaw.kr(GetTempo.kr * sample.beats.reciprocal, sample.end * 0.5, sample.end * 0.5);

 ReTrigger2.ar({
  PlayBuf.ar(sample.signal,sample.sampleRate,pchRatioKr,start.poll,0,sample.end);
 },gate,env,sample.numChannels)
},
[
 Stream2Trig(
  1.0,
  Pseq(Array.geom(8.rand,2 ** -5,  2.0).scramble,inf)
 ),
 Env.asr(release:0.1),
 Sample(":Sounds:floating_1"),
 StreamKrDur(
  Pslide(Array.series(q,-2.0,2.0 / q).scramble,inf,rrand(3,5),rrand(2,5)),
  rrand(0.125,1.0)
 )
]).topGui

)


(
Patch({arg gate,env,sample;
 var p;
 p = PlayBuf.ar(sample.signal,sample.sampleRate,sample.pchRatioKr,0,0,sample.end);
 Enveloper2.ar(p,gate,env,sample.numChannels)
},
[
 Stream2Trig(‘([1,0,1,0,0,1,0,1]),‘(Array.fill(8,{ 2 ** rrand(-5,-1) }))),
 Env.perc(release:0.2),
 Sample(":Sounds:floating_1")
]).topGui

)

(
```

```
Patch({arg gate,env,sample,startBeat;
 var p,s,e;
 p = sample.pchRatioKr;
 s = startBeat * sample.beatsize;
 e = s + LFNoise1.kr(0.2,9000.0,5000.0);
 Enveloper.ar({ PlayBuf.ar(sample.signal,sample.sampleRate,p,s,s,e); },gate,env,4,sample.numChannels)

},
[
 Stream2Trig(‘(Array.fill(128.rand,{[1,0.125,0,0].choose})),‘(Array.fill(128.rand,{ 2 ** rrand(-7,-1)
}))),
 Env.perc(release:3.0),
 s = Sample(":Sounds:floating_1"),
 StreamKrDur( Pfunc({ s.beats.rand.round(0.25) }),Pfunc({ 2 ** rrand(-4,2)}))
]).topGui

)


***/
```

6.11 Scheduling

ID: 158

atTime

this is an argument for many methods.
it specifies when the bundle or event should occur

Nil : immediately

Float : that many seconds from now
if time is greater than server latency,
it will be scheded in sclang and only sent close to the time

// start in 4.5 seconds
(

```
Patch({ arg tempo;
Impulse.ar( tempo )
},[
TempoPlayer.new

]).play(atTime:  4.5)
```

)

Integer : according to TempoClock on the next
1 bar
2 half bar
4 beat
8 8th note
16 16th note
etc.

execute the following several times. they will each start at the start of the next bar.
(

```
Patch({ arg tempo;
Impulse.ar( tempo )
},[
TempoPlayer.new
```

```
]).play(atTime:  1)
```

)

Date : at that time on that date if in the future
Date has to have raw seconds set to work !
use Date.localtime or Date.getDate to create a Date object with the raw seconds set.
and then make relative changes to that date.

ie. you can't make a Date.new(year,month ....) and expect that to work.
note: a getRawSeconds primitive would solve this problem.

```
(
d = Date.getDate;
// 10 seconds in the future
d.rawSeconds = d.rawSeconds + 10;
Patch({ arg tempo;
Impulse.ar( tempo )
},[
TempoPlayer.new

]).play(atTime:  d)

)
```

ID: 159

BeatSched

A beat capable function scheduler

Functions can be scheduled for precise execution using relative seconds, relative beats, absolute seconds or absolute beats. This class uses TempoClock for scheduling, and has some overlap of capabilities with that.

The TempoClock class is used to specify what the tempo is, and is used for all beat <-> second calculations. The default global TempoClock object is used, or you can use a specific TempoClock instance.

There is a default global BeatSched that can be addressed through class methods. It uses the SystemClock and the default global TempoClock. You may also create individual instances that maintain their own scheduling queues, tempii, and time epochs.

If using BeatSched instances you can clear the queue, only affecting your own events. If using the global BeatSched class, clearing the queue will affect everybody.

All of these methods exist as both class (the default global scheduler)

```
BeatSched.tsched(seconds,function)
```

and instance methods (a specific scheduler).

```
beatSched = BeatSched.new;
beatSched.tsched(seconds,function)
```

The default clock used is the SystemClock, but you may also specify to use the AppClock.

tsched(seconds,function)
time based scheduling
delta specified in seconds
xtsched(seconds,function)
exclusive time based schedule
any previous messages scheduling using xtsched, xsched or xqsched will
be cancelled. this is incredibly useful in situations where several messages

might be sent and you only want the last one to be used as the final answer.
example: a monkey is hitting many buttons, changing his mind about which sound to play next. this would result in many messages being stacked up all at the same time, and the server would choke trying to execute them all.
this is a kind of enforced monophony.
another example: a sequence plays successive notes, all using xsched,
you then switch to a different sequence. you don't want the note that was scheduled from the previous sequence to happen. using one of the x-methods, you don't have to worry about it, the old notes will be cleared when new ones are scheduled.

sched(beats,function)
delta specified in beats
xsched(beats,function)
exclusive beat based scheduling

qsched(quantize,function)
will happen at the next even division (4.0 means on the downbeat of a 4/4 bar), or immediately if you happen to be exactly on a division.
xqsched(quantize,function)
exclusive quantized beat based scheduling

tschedAbs(time,function)
will happen at the time specified in seconds
schedAbs(beat,function)
will happen at the beat specified.

xblock
blocks any and all pending x-scheduled messages.

time
get the scheduler's time
time_(seconds)
set the scheduler's time
beat
get the scheduler's current beat
beat_(beat)
set the scheduler's current beat.
this is also used to start a "song": zero the beat, and all absolute times previously scheduled events will be unpredictable

deltaTillNext(quantizeDivision)

returns the number of seconds untiil the next quantizeDivision.
4.0 means the next even bar
16.0 means the next 4 bar cycle
0.25 means the next 16th note

clear

clear all scheduled events.

In the examples, remember to execute groups of code lines together.

```
b = BeatSched.new;

b.qsched(4.0,{ "hello".postln; });

b.qsched(4.0,{ "hello".postln; });


b.time; // since the app started

b.time = 0.0; // reset the time

b.time;

b.beat;

TempoClock.default.tempo = 2;
b.beat.postln;
TempoClock.default.tempo = 1;
b.beat.postln;


b.time = 0.0;
b.tschedAbs(4.0,{ "4 seconds absolute time".postln; });
b.tschedAbs(2.0,{ "2 seconds absolute time".postln; });
```

```
b.xsched(4.0, { "4 beats later".postln });
// cancels previous xsched
b.xsched(2.0, { "2 beats later".postln });
```

A little rounding error

```
(
TempoClock.default.tempo = 120 / 60.0;
    Routine
20.do({
var t;
t = BeatSched.global.tdeltaTillNext(4.0);
t.postln;
t.wait;
});
});
SystemClock.play(d);
)
```

```
at 5206.432346276  we ask for deltaTillNext 4
[ 5204, 4, 5206.432346276 ]
1.5676537239997

that would be
5206.432346276 + 1.5676537239997
// at 5208
5208


// but when the scheded event comes due:
[ 5204, 4, 5207.999072862 ]
0.00092713799949706

its appears to be slightly ahead of schedule, due
to rounding errors in the several math ops that have happened.

so the right answer is 0.00092713799949706
as far as BeatSched is concerned.
```

but if you try to loop like this, you will suffer from rounding errors.

mostly you would never set up a loop like this, mostly
you just want to know when the next even beat is so you can get your groove on.

```
Tempo.bpm_(120);
    Routine
 "wait for the downbeat..."
OSCSched.global.tdeltaTillNext(4.0).wait;

32.do({ arg i;
[i,BeatSched.beat].postln;
Tempo.beats2secs(1.0).wait;
});
});
SystemClock
```

ID: 160

OSCSched

A scheduler for sending OSC bundles to servers.

The bundle is kept on the client until the last possible moment, and then actually sent to the server in a time stamped bundle, just before it is due to be executed.

Bundles can be scheduled for precise execution using relative seconds, relative beats, absolute seconds or absolute beats. Bundles can be scheduled on multiple servers, with exact simultaneous execution times.

Bundles can be easily cancelled up until the time they are sent to the server. They are sent to the server just before execution.

The Tempo class is used to specify what the tempo is, and is used for all beat <-> second calculations. A default global Tempo object is used, or you can create a specific Tempo instance if y'all got your own separate grooves.

There is a default global OSCSched that can be addressed through class methods. It uses the SystemClock and the default global Tempo. You may also create individual instances that maintain their own scheduling queues, tempii, and time epochs.

The default clock used is the SystemClock, but you may also specify to use the AppClock.

An optional clientSideFunction can also be supplied that will be evaluated on the client at the exact time as the OSC bundle is scheduled to happen. This could be used to show a change in the gui or to update some setting on a client side object.

All of these methods exist as both class (the default global scheduler)

```
OSCSched.tsched(seconds,server,bundle,clientSideFunction)
```

and instance methods (a specific scheduler).

```
oscSched = OSCSched.new;
oscSched.tsched(seconds,server,bundle,clientSideFunction)
```

tsched(seconds,server,bundle,clientSideFunction)
time based scheduling
delta specified in seconds
xtsched(seconds,server,bundle,clientSideFunction)
exclusive time based schedule
any previous bundles scheduled using xtsched, xsched or xqsched will
be cancelled. this is incredibly useful in situations where several bundles
might be sent and you only want the last one to be used as the final answer.
example: a monkey is hitting many buttons, changing his mind about which
sound to play next. this would result in many bundles being stacked up all
at the same time, and the server would choke trying to execute them all.
so this forces a kind of monophony of bundles.
another example: a sequence plays successive notes, scheduling each one
when it plays the previous one.
you then switch to a different sequence. you don't want the note that was
scheduled from the previous sequence to happen. using one of the
x-methods, you don't have to worry about it, it will just be cancelled.

sched(beats,server,bundle,clientSideFunction)
delta specified in beats
xsched(beats,server,bundle,clientSideFunction)
exclusive beat based scheduling

qsched(quantize,server,bundle,clientSideFunction)
will happen at the next even division (4.0 means on the downbeat of a 4/4 bar),
or immediately if you happen to be exactly on a division.
xqsched(quantize,server,bundle,clientSideFunction)
exclusive quantized beat based scheduling

tschedAbs(time,server,bundle,clientSideFunction)
will happen at the time specified in seconds
schedAbs(beat,server,bundle,clientSideFunction)
will happen at the beat specified. this uses TempoClock for scheduling

xblock
blocks any and all pending x-scheduled bundles.

time
get the scheduler's time
time_(seconds)

set the scheduler's time

beat

get the scheduler's current beat

beat_(beat)

set the scheduler's current beat.
this is also used to start a "song": zero the beat, and all absolute times previously scheduled events will be unpredictable

deltaTillNext(quantizeDivision)

returns the number of seconds untiil the next quantizeDivision.
4.0 means the next even bar
16.0 means the next 4 bar cycle
0.25 means the next 16th note
This value is only correct so long as you don't change the tempo.
For scheduling, use the beat based scheduling methods.

clear

clear all scheduled events.

All of the x-methods establish a block such that a subsequent schedule using another x-method will cause the previous one to be cancelled. This is particularily useful when you are controlling something from a gui or process, and change your mind before the event you have triggered comes due. Another example is a pattern that returns delta beat values, repeatedly scheduling its next note at the time of playing the current one. To switch the pattern with another or abruptly start it over, simply do so: if you used xsched, then the previously scheduled event will be cancelled.
In most cases, you will wish to create a private instance of OSCSched when using this technique.

warning: older examples, not tested recently

load all of these for use in all following examples

```
s = Server.local;
s.boot;
(
SynthDef "bubbles"
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
```

```
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4);
Out.ar(0, zout);
}).send(s);

i = [ '/s_new', "bubbles", 1002, 1, 0 ];
o = [ '/n_free', 1002 ];

c = OSCSched.new;
)



// unitialised, the scheduler's time is number of seconds
// since SC itself started up
c.time.postln;

// defaults to 1.0 beats per second
Tempo.tempo.postln;

// number of beats since SC itself started up
c.beat.postln;

// set the default global Tempo
Tempo.bpm = 96;

// how many of those beats since time started
c.beat.postln;

// tell the scheduler what beat we think it is
c.beat = 0.0;

// how beats since time started
c.beat.postln;



// start in 2.0 beats
c.sched(2.0,s,i,{
 "starting"
});
```

```
// free the synth on the next even bar
c.qsched(4.0,s,o,{
 c.beat.postln; // note the floating point imprecision
});


// start in 4.0 seconds
c.tsched(4.0,s,i,{
 "starting"
});
```

Absolute Beat / Time scheduling

```
c.clear;


(
               // we are starting at beat 32
c.schedAbs(36.0,s,i); // in
c.schedAbs(39.0,s,o); // out
c.schedAbs(41.0,s,o); // out
c.schedAbs(40.0,s,i); // but first in
c.schedAbs(43.0,s,i,{
c.schedAbs(42.0,s,o,{
  "this will never happen, its in the past"
});
c.schedAbs(46.0,s,o);
});
)
```

Exclusive

```
(
c.xsched(4.0,s,i,{
"4.0".postln;
});
```

```
c.sched(8.0,s,o); // not affected
// changed my mind...
                    // the x-methods are exclusive
"3.0".postln;
});
)
```

ID: 161

Tempo tempo calculations

This class represents the concept of tempo. It can be used for translations between seconds, beats and bars. It holds an instance of TempoClock which it sets to its own tempo whenever that is changed.

A TempoBus can be started on the server, and it will keep the Tempo object's tempo as a float value on a Bus on the server. UGens can use this for scaling their frequencies for beat based rhythms etc.

It can be used to convert beats <-> seconds, but this value is only accurate at the time you make the computation. If the tempo is changed the value is of course no longer valid. TempoBus adds itself as a dependant to the Tempo object, so when the tempo changes, it is informed, and it updates the value on the bus accordingly.

```
Tempo.bpm = 170;
Tempo               // in beats per second

Tempo      // there is a gui class

Tempo.bpm = 170;
Tempo.beats2secs(4.0).postln;

Tempo.bpm = 10;
Tempo.beats2secs(4.0).postln;
```

All class methods refer to the default global tempo.
You can create an instance of Tempo if you need individual, separate tempii.

```
t = Tempo.new;
u = Tempo.new;

t.bpm = 170;
                // in beats per second
t.gui;
```

All of the following methods exist as class methods (the default tempo)

and as instance methods.

bpm
bpm_(beatsPerMinute)
Tempo.bpm = 96;
or
Tempo.bpm_(96);
tempo
in beats per second
tempo_(beatsPerSecond)
Tempo.tempo = 2.0;
or
Tempo.tempo_(2.0);

beats2secs(beats)
secs2beats(seconds)
bars2secs(bars)
you can change the beats per bar:
Tempo.beatsPerBar = 7.0;
secs2bars(seconds)

sched(delta,function)
Schedule a function to be evaluated delta beats from now.

If you change the tempo after scheduling, your function will still be evaluated at the time originally calculated. A more sophisticated solution will be presented later.

schedAbs(beat,function)
Schedule a function to be evaluated at an absolute beat, as measured from the time SuperCollider first booted up. Use OSCsched for more sophisticated control (able to reset the beat).

If you change the tempo after scheduling, your function will still be evaluated at the time originally calculated. A more sophisticated solution will be presented later.

ID: 162

TempoBus

A Bus whose value is set by a Tempo. It can be used as a multiplier in any Synth on the Server that needs to know the Tempo. It is used by BeatClockPlayer. Any Inst/Patch that needs a tempo input should use a TempoPlayer.

TempoBus.new(server,tempo)
TempoBus.new
default server, default tempo

There is one TempoBus per server, per tempo. After the first time it is created, the shared instance will be returned for any subsequent requests.

```
(
a = TempoBus.new;
b = TempoBus.new;
 a === b // they are the same object
)
```

```
(
s = Server.local;
t = TempoBus.new(s);

t.index.postln;

Tempo.bpm = 60;

SynthDef("help-TempoBus",{ arg out=0,tempoBus;
var tempo,trig,amp;
tempo = In.kr(tempoBus);
trig = Impulse.kr(tempo);
amp = Decay2.kr(trig,0.01,0.1).clip2(1.0);
Out.ar(out,
amp * SinOsc.ar(300)
```

```
)
}).play(s,[0,0,1,t.index]);


Tempo.bpm = 40;

Tempo.bpm = 100;

Tempo.bpm = 666;

Sheet   arg
 Tempo                  // move the slider, it works
})

)


see TempoPlayer
```

ID: 163

TempoPlayer

Outputs the current tempo in beats per seconds. All TempoPlayers share the same TempoBus, and so don't incur any additional resources.

Move the tempo slider.

```
(
 Instr \helpTempoPlayer   arg
Impulse.ar( tempo )
},[
  \tempo
]);

Patch(\helpTempoPlayer
,[
  TempoPlayer
]).gui
)
```

A TempoBus belongs to a specific server for its whole object-lifetime. A TempoPlayer is only told which server it is to play on when it is asked to prepare for play by its parent object. A TempoPlayer can be saved in a larger musical structure and that structure is capable of being played on disparate servers.

the symbol \tempo is registered in Spec.specs as a TempoSpec

```
\tempo.asSpec.insp
```

whose defaultControl is a TempoPlayer

```
\tempo.asSpec.defaultControl.insp
```

so that the argname tempo in an Instr would by default result in a TempoPlayer for a Patch using that Instr.

```
Patch({ arg tempo;
Impulse.ar( tempo )
```

```
}).gui



execute this many times
(

Patch({ arg tempo;
Impulse.ar( tempo )
},[
TempoPlayer.new

]).play(atTime:  1)

)
```

see BeatClockPlayer

6.12 Sequencers

ID: 164

ModalFreq

Control rate player

For backwards compatiblity and convenience. This actually returns a Patch on the pseudo-ugen ModalFreqUGen.

Takes floats or player inputs and puts out control rate signal of frequency.

ModalFreq.new(degree, scaleArray, root, octave, stepsPerOctave)

Used as a kr rate Player.

```
(
m =  ModalFreq.new(
StreamKrDur.new(Pbrown(1,12,2,inf), 0.25, 0.1),
FloatArray[ 0, 1, 2, 3, 4, 7,10 ],
StreamKrDur.new(Pseq(#[ 7, 6,1,10 ], inf), 0.25, 0.1),
StreamKrDur(Pbrown(2,6,1),Prand([0.25,1.0,4.0,8.0])),
12
);

Patch({ arg freq=200;
Pulse.ar(
  [freq, freq * 0.5]
LFNoise1.kr([0.3,0.1],0.5,0.5),
0.2)
},[
m
]).play

)
```

Used as a Stream. Can also be used with Pbind or any other Pattern.
Not as efficient as using it directly as a Player.

```
(

Patch({ arg freq=100,amp=1.0;
```

```
SinOsc.ar([freq,freq + 30],0,amp)
},[
  StreamKrDur
ModalFreq(Pseq([1,2,3,4],inf)),
Pseq([0.25,0.5,0.75],inf),
0.1),
1
]).play;

)
```

ID: 165

PlayerSeqTrack

Play a series of Players in a sequence, with individual control over duration,
attack,release and level.

This is only designed to work with Players that already saved to disk.

The primary design difference between this and CyclePlayers is that PlayerSeqTrack does not need to know all of its players before play/editing.

They can be added and removed dynamically, even while playing.

PlayerSeqTrack(arrayOfPlayerPaths, arrayOfBeatDurations,arrayOfEnvelopes, loop)

arrayOfPlayerPaths - or nil (players can be added later)
or they can be actual players. if you use paths, then identical paths in the array will play the identical player (only one copy is loaded).
so the sequence can contain repeats:
[path1, path2, path1, path2, path3 ...]

(

```
// locate some players on your hard drive and add them to a list
// hit cancel when you have enough
l = List.new;
f = {
 GetFileDialog   arg
if(ok,{
l.add(loadPath(path));
f.value;
})
});
};
f.value
```

)

```
(

    PlayerSeqTrack

6.do({
p.insertLast(Properties(\working).wchoose.asString.loadDocument);
});

p.setDuration(0,4);
p.setDuration(1,4);
p.setDuration(2,4);
p.setDuration(3,4);
p.setDuration(4,4);
p.setDuration(5,4);

p.gui
)



p.play;

p.free;
)

p.insp
```

PlayerSeqTrackGui

now while watching the pretty visuals:

```
(
p.gui;
 p.changed; // update the gui
)


// while playing is fine, weve already loaded the players
p.insert(  3, l.choose ).changed;
```

```
p.insert( 8.rand, l.choose ).changed;



(
5.do({ arg i;
p.insert(i, l.choose );
});
p.changed;
)
(
5.do({ arg i;
p.setDuration( p.playerSeq.size.rand, [4,8,16,32].choose );
});
p.changed;
)
(
5.do({ arg i;
p.setRelease( i, rrand(0.01,5.0) );
});
// no gui display of release times
)
(
5.do({ arg i;
p.setLevel( i, rrand(0.1,2.0) );
});
// no gui display of levels
)



p.deleteAt(6).changed;
```

Note that the duration display changes also. It also changes when you change the tempo.

GUI Hot-Keys

While selected on any of the sequences these keys are active:

<- select previous

-> select next
opt -> move selected left
opt <- move selected right
up increase dur by 1 bar
down decrease dur by 1 bar
opt-up double duration
opt-down half duration
shift-opt-up double durations of all steps
shift-opt-down half durations of all steps
‘ relocate (while playing) to this step
delete delete this step
g gui the player at this step
i open information window

escape focus on the first focusable view in this window that is not one of the sequence steps

The information window

in the information window you can edit
duration, attack,decay,level, envelope
in three selectable scopes:
this step, all steps with this player, all steps

It is also possible to embed the info window on the same layout:

```
(
 Sheet  arg
PlayerSeqTrack.new.topGui(f).infoGui(f);
})
)

Note that topGui returns the PlayerSeqTrackGui object, which responds to infoGui(layout);
```

Its turtles all the way down

It is of course possible to put a PlayerSeqTrack inside of another PlayerSeqTrack.

Multiple tracks are obtainable via the use of PlayerMixer, though they won't easily
remained synchronized if you relocate while playing.
And the gui will be not lined up verticle.
Eventually i will write a multi-track version that holds and synchronizes multiple PlayerSeqTrack.

Live Insert of Players

```
( // insert players at the selected step, even while playing
Sheet   arg


 p = PlayerSeqTrack
 g = p.topGui(f); // return the gui itself


Label(f,"insert:");
l.do({ arg player;
ActionButton(f,player.name,{ p.insert(g.selected ?  0 , player).changed })
})
})
)


(
// insert players NOW at the presently playing step
// 808 style
Sheet   arg


 p = PlayerSeqTrack
p.topGui(f);


Label(f,"insertNow:");
l.do({ arg player;
ActionButton(f,player.name,{ p.insertNow( player,round:  1.0).changed })
})
})

)
```

insert(step,player,duration,env)
step- index to insert at
player - the player object or path to insert
in the example above, i used the actual player because its faster, you share the same player instance, and it doesn't have to load anything from disk while
its playing.
duration - the number of beats it should play for
if nil,
use the duration of any previous appearance of this player in the sequence
if that is nil,
use the natural beatDuration of the player
if that is nil,
use 128.0 beats
env - the envelope to use
if nil,
use the envelope of any previous appearance of this player in the sequence
if that is nil,
use default envelope

All players that you insert must have a path (must have been loaded from disk).

When this example saves, notice that the steps 0 and 3 repeat the same player. On reloading, they will
share the same copy.

```
PlayerSeqTrack
[ ":Patches:footfist:dawhohop"  ":Patches:footfistwhisker:dhallooo"  ":Patches:footfist:dawhohop"

 ":Patches:footfistwhisker:buggercraft"  ":Patches:footfistwhisker:basscl"  ":Patches:footfist:simp"

[ 16, 16, 16, 16, 16, 16 ],
[ Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1,
0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([
0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -
2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil) ]
```

```
)
```

Actually PlayerSeqTrack could play players without a path, but the gui would display them
all as "nil" "nil" etc.
And it would save as something like this:

```
  PlayerSeqTrack
[ Patch.new(
   [ 'minimoog'  'detune'
[ 440, -4, 0, 0.4, 1 ]
), Patch.new(
   [ 'synths'  'stereo'  'SyncSaw'
[  BeatClockPlayer(16)
, 440, 447.214, 0.5, 0.4, Env.new([ 0, 1, 0.5, 0 ], [ 0.01, 0.3, 1 ], -4, 2, nil), 4, 2 ]
), Patch.new(
   [ 'minimoog'  'detune'
[ 440, -4, 0, 0.4, 1 ]
) ],
[ 16, 16, 16 ],
[ Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1,
0.1 ], [ -2, -2, -2 ], 2, nil), Env.new([ 0, 1, 1, 0 ], [ 0.01, 1, 0.1 ], [ -2, -2, -2 ], 2, nil) ]
)
```

And on reload the identical steps 0 and 3 would not be able to share the same copy.

ID: 166

Stream2Trig generate a trigger signal from a stream

superclass: StreamKrDur

Stream2Trig(levels,deltas)

Take a pattern and use it as a stream of values for a trigger. This plays the pattern in real time on the client, and sends messages to the server.

levels - A stream of values for the level of each trigger

```
1.0
Prand([1,0],inf)
{ 1.0.rand }
‘[ 1,0,1,0,] // Ref converted into Pseq([1,0,1,0],inf)
```

deltas - A stream of values for the delta in beats between each trigger.

```
0.25
Prand([0.25,0.5],inf)
FuncStream({ rrand(0.125,16.0) })
‘[1.0,0.25,0.5] // Ref
```

```
(
Instr(\Stream2Trig, {arg gate=0.0,freq=440,env;
p = Pulse.ar(freq,LFNoise1.kr(0.1).abs,0.5);
Enveloper2.ar(p,gate,env,2,2);
},[
 nil
 nil
 \envperc
]);

Patch     \Stream2Trig
[
 Stream2Trig
```

```
Pseq([
Prand([1,1,1,1,1,0,0,0,0],2)],inf),
Pseq([0.25,0.25,0.25,0.125,0.125],inf)
),
StreamKrDur(Pseq( Array.rand(16,30,50).midicps,inf ), 0.25,0.1 )
]).gui

)



(

Patch     \Stream2Trig
[
 Stream2Trig
Pseq([
Pn(Pshuf([1,0,1,0],4),3),
Pshuf([0.2,0,0.2,0],4)],inf),
Pseq([0.25,0.25,0.25,0.125,0.125],inf)
),
84
]).gui
)
```

A nice sequel would be to make a class that also takes a gate duration stream (in beats or legato), so that the gate will stay open for that long.

ID: 167

StreamKrDur

superclass: StreamKr ... AbstractPlayer

StreamKrDur.new(pattern,durations,lagTime)

This plays a pattern in real time on the client, and sends those values as /c_set messages to its bus on the server.
Durations are specified in beats by a second pattern.

Its timing are exact, its messages are sent to the server with a small latency period before they should be played. That is to say when you start playing a StreamKrDur, it will run slightly and exactly ahead of time.
It is cancellable and stoppable within the value of Server-latency.

It is extremely efficient in CPU usage on the client, using less than it takes to switch from one text window to another.

pattern
a Pattern or Stream of floats
durations
a float specifying the beat constant duration
a stream or pattern of beat durations
or a ref to an array of beat durations eg '[0.25,0.5,1]
lag
lag time for a Lag.kr that will be added to the output.
if 0.0 (default) no Lag will be used.

dalek mating season I

```
(
 var freq;

        StreamKrDur
Prand(Array.fill(rrand(4,16),{ rrand(20,80).midicps }),inf),
    0.25,// a float
0.1);
```

```
 Tempo            // times are in beats

Patch({ arg freq;
Saw.ar(freq,0.2)
},[
freq
]).play
)



(
 var freq;

freq =  StreamKrDur(
Pbrown(40,100,100,inf),
‘[ 1,2, 0.25], // an array
0.05);

 Tempo            // times are in beats
Patch({ arg freq=440;
SinOsc.ar(freq,SinOsc.ar(600,0,0.3))
//PMOsc.ar(freq,100,1.0,0,0.3)
},[
freq
]).play
)
```

dalek mating season II

```
(
var freq,freq2,pmindex;

        StreamKrDur
Prand(Array.fill(rrand(4,16),{ rrand(20,80).midicps }),inf),
    0.25,// a float
0.1);

freq2 =   StreamKrDur(
Pbrown(40,100,100,inf),
```

```
‘[ 1,2, 0.25], // an array
0.05);

pmindex = StreamKrDur(
Pbrown(1.5,3.0,0.1,inf),
Prand([0.25,0.125,1.0,3.0],inf), // a pattern
0.1);

 Tempo            // times are in beats
Patch({ arg freq,freq2,pmindex;
PMOsc.ar(freq,freq2,pmindex,0,0.3)
},[
freq,
freq2,
pmindex
]).gui
)
```

same thing with a separated Instr

```
(
Instr([\oscillOrc,\pmosc],{ arg freq=400, freq2=500,pmindex=0,phasemod=0.0,amp=1.0;
PMOsc.ar(freq,freq2,pmindex,phasemod,amp)
});

 p=Patch       ’oscillOrc’  ’pmosc’
[
  StreamKrDur
Prand(Array.fill(rrand(4,16),{ rrand(20,80).midicps }),inf),
    0.25,// a float
0.1),
    StreamKrDur
Pbrown(40,100,100,inf),
‘[ 1,2, 0.25], // an array
0.05),
    StreamKrDur
Pbrown(1.5,3.0,0.1,inf),
Prand([0.25,0.125,1.0,3.0],inf), // a pattern
0.1),
0,
```

```
0.3
]);

p.gui;
)
```

A metronome

```
(
var nome,layout;
nome = Patch({ arg beat,freq,amp;
Decay2.ar(
K2A.ar(beat), 0.01,0.11,
SinOsc.ar( freq, 0, amp )
)
},[
  BeatClockPlayer
  StreamKrDur
Pseq([ 750, 500, 300, 500, 750, 500, 400, 500, 750, 500, 400, 500, 750, 500, 400, 500 ],inf),
1.0),
  StreamKrDur
Pseq([1,0.25,0.5,0.25,0.75,0.25,0.5,0.25,0.75,0.25,0.5,0.25,0.75,0.25,0.5,0.25] * 0.01,inf),
1.0)
]);
layout = FlowView.new;

ToggleButton        "Nome"
if(nome.isPlaying.not,{ nome.play(atTime:  4) })
},{
if(nome.isPlaying,{ nome.stop })
},minWidth:  250);

)
```

In this case since the beat clock, and both stream kr durs are running all at the same tempo, you could better use an InstrGateSpawner and do separate events. Only one scheduler then instead of 2. See the example there.

But if you wanted them in different syncs, different streams and a continous running synth (have fun with the Decay), then this would be a good starting point.

with a Pbind you are creating a new event for every note and creating new UGens and a new Synth each time. this is not so efficient. what I am doing here is like an analog synth: the oscillators play constantly and it is gated to create notes. its more fluid, liquid and uses significantly less cpu. this can even be done with samples.
(

```
Tempo.bpm = 130;

Instr([\oscillOrc,\trigged, \pmosc],{ arg trig=0.0,freq=400, freq2=500,pmindex=0,phasemod=0.0,amp=1.0;

PMOsc.ar(
freq,
freq2,
pmindex,
phasemod,
Decay2.kr(trig)
)
});

 p=Patch      \oscillOrc \trigged  \pmosc
[
  BeatClockPlayer

  StreamKrDur
Prand(Array.fill(rrand(4,16),{ rrand(20,80).midicps }),inf),
    0.25,// a float
0.1),
    StreamKrDur
Pbrown(40,100,100,inf),
`[ 1,2, 0.25], // an array
0.05),
    StreamKrDur
Pbrown(1.5,3.0,0.1,inf),
Prand([0.25,0.125,1.0,3.0],inf), // a pattern
0.1),
0,
0.3
]);
```

```
p.gui;
)



//durations will be multiples of trigger pulse
//
//stream kr dur has to find the lowest common multiple
//
//l = [0.125,0.25,1.0,1.5];
//
//l = [0.25,0.25 + 0.125, 1.0,1.5];
//
////fails
//l = Array.fill(8,{ rrand(0.1,0.5) });
//
//(l % (l.minItem)) every is 0
//or
//
//g = (l % (l.minItem)).reject({ arg t; t==0 }).minItem
//
//l / g  should be all integers
// or it fails
//
```

6.13 ServerTools

ID: 168

SharedBus

superclass: Bus

This simple class solves a problem with sharing busses between players. When a player is stopped or freed, it frees its Bus, which frees the Bus number for reallocation.

In the case of PlayerMixer, several players are using the same bus. Only the PlayerMixer itself should be able to free the Bus.

In the case of PlayerSocket, each player that plays in the socket is given the bus, and they will free that Bus when the player is replaced by a new player in the socket. Again, only the PlayerSocket itself should be able to finally free the Bus.

A SharedBus will not free its bus number until given explicit permission by its owner by calling .releaseBus

If you are writing a class that for instance hosts various Players and you wish them to share busses or to keep ahold of the bus you are playing them on without having the Player free it:

```
sbus = bus.as(SharedBus);

.... use it ....

sbus.releaseBus; // releases and frees the Bus index
```

6.14 UncoupledUsefulThings

ID: 169

Enveloper2

gate the audio input with an Envelope;

```
(

b = Bus.control(Server.local);
{
var p,qnty=4,gate;
gate = In.kr(b.index);
p = Mix.arFill(qnty.asInteger,{ arg i;
SinOsc.ar(440 + rrand(1,5))
}) * qnty.reciprocal;

Enveloper2.ar(p,gate,Env.adsr)
}.play;

)



// On
b.value = 1.0;

// Off
b.value = 0.0;

b.value = 0.2;

b.value = 0.0;

b.free;
```

ID: 170

KeyCodeResponder

Handles multiple registrations for keycode and modifier combinations.
This object should be used as the keydown or keyup action for a view in place of a function.

KeyCodes are hardware dependant, and change from machine to machine.
Its a simple way to hook up keys to respond, but it won't transfer to other people's computers.

see also [**UnicodeResponder**] which matches based on the unicode, though the physical location
of the key may still vary according zee nationality ov der keyboard.
#1322ff
#1322ff
see [**SCView**]

Use this to post the keycodes
(

```
KeyCodeResponder
```

)
it prints

```
//  shift :
k.register(   41   ,   true, false, false, false, {

});
```

```
(
#1322ff
    KeyCodeResponder

// match single modifiers exclusively
```

```
k.normal( 36 -> {  "return".postln });
k.option( 36 -> { "option return".postln });
k.shift( 36 -> { "shift return".postln });
// overwrites previous registration
k.shift( 36 -> { "SHIFT RETURN only".postln; });

#1322ff
#1322ff
// match multiple modifier combinations
 //        shift caps opt   control
k.register( 36, true,  nil,  true, false, {
 //         yes   either  yes   no
 "return:  shift-option regardless of CAPS"
});


k.registerKeycode(KeyCodeResponder.normalModifier, 52 , { "enter".postln; });

// arrow keys are considered function keys and must be bit ORd with function key modifier
k.registerKeycode(KeyCodeResponder.normalModifier | KeyCodeResponder.functionKeyModifier , 123 , {  "<-
".postln;
});

k.registerKeycode(KeyCodeResponder.controlModifier | KeyCodeResponder.functionKeyModifier , 123 , {
 "control <-"
});

w = SCWindow.new.front;
v = SCSlider.new(w,Rect(10,10,100,100));
v.keyDownAction = k;
v.focus;

)
```

register(keycode, shift, caps, opt, cntl, function)

for shift, caps,opt,cntl

true indicates a required modifier
false indicates an excluded modifier
nil expresses that you really don't care one way or the other

normal(keycode -> function)
normal(keycode -> fuction, keycode2 -> function2 , ... keycodeN -> functionN)

note the association (key -> value)

shift(keycode -> function)
shift(keycode -> fuction, keycode2 -> function2 , ... keycodeN -> functionN)

option(keycode -> function)
option(keycode -> fuction, keycode2 -> function2 , ... keycodeN -> functionN)

control(keycode -> function)
control(keycode -> fuction, keycode2 -> function2 , ... keycodeN -> functionN)

Any player's gui can have its keyDownAction set

```
(


p = Patch({ arg freq=440; SinOsc.ar(freq,mul:  0.1) });
g = p.gui;
g.keyDownAction =  {
 "you touched me"
};


)
```

focus on the slider. notice that every key stroke is passed,
the slider does not swallow them.

or you can use KeyCodeResponder

```
      KeyCodeResponder
kcr.option( 36 -> { "option 36".postln });
kcr.shift( 36 -> { "shift 36".postln });
aPatchGui = aPatch.gui;
aPatchGui.keyDownAction = kcr;
```

This means that when ever the player is focused (any of its controls is in focus), these keys will be active providing that the view that is actually in focus doesn't handle the key event (it should have a nil keyDownAction function or pass nil).

You can concatenate KeyCodeResponders using ++

```
global keydowns not yet tested....
(
KeyCodeResponder
/*
this will fire on shift-'r'
 shift must be held down
 caps must NOT be down
 cntl or opt status is irrelevant
*/
KeyCodeResponder.register(15,true,false,nil,nil,{ "shift, no caps".postcln });

/*
this will fire on shift-'r'
 shift must be held down
 caps may or may not be down
 cntl or opt status is irrelevant
*/
```

```
KeyCodeResponder.register(15,true,nil,nil,nil,{ "shift, yes/no caps".postcln });
Sheet({ arg f; ActionButton(f).focus });


)
```

This is very useful when using CAPS-LOCK to switch interface modes etc.

Only one function per deny/require mask combination is possible per keycode:
(
// hit shift - r

```
KeyCodeResponder.register(15,true,nil,nil,nil,{ "shift r".postcln });
KeyCodeResponder.register(15,true,nil,nil,nil,{ "overwrote the previous one".postcln });
Sheet({ arg f; ActionButton(f).focus });

)
```

The simpler, older method is :

```
KeyCodeResponder.registerKeycode(2,28,{      });// *
```

whereby that modifier and only that modifier will fire the funtion.
see SCView for modifier values or use this :

```
(

 KeyCodeResponder

)



// using characters
KeyCodeResponder.registerChar(0,$q,{  }); // q no modifier
```

Gotcha: it is easy to forget that you registered a function with KeyCodeResponder that holds a reference to a large object. Garbage Collection will not remove the object until
you have released your reference.

solution:
// place this at the top of your performance code to start with a clean slate

```
KeyCodeResponder
```

ID: 171

Mono forces the input to a mono signal if needed

Mono.ar(input)

This is useful for inputs to compressors, amplitude detectors or audio functions where you need to ensure that the input is mono. It takes the first channel and discards others. If the input is already mono, it passes through unscathed.

It saves asking the input if its an array or not.

ID: 172

NotificationCenter

Objects can send notifications to the NotificationCenter, and all functions that were registered for
that notification will be excecuted.

implements the Notification pattern.
This is similar to MVC, except here the model object does not ever
know anything about who is dependant on it.
This allows any interested client object to be notified of special events such as the object being
saved to disk, the object recording a sound file version of itself etc.

For instance when a Sample is saved it emits a \didSave notification:

```
NotificationCenter(notify,Sample,\didSave,soundFilePath);
```

You can listen for this:

```
  NotificationCenter          Sample \didSave \sampleWatcher    arg
path.postln;
});
```

```
//in the following examples this is the interpreter
this.postln
    Interpreter
```

```
(
// nothing is yet registered
// so notify finds nothing to do
NotificationCenter        this \didRecord

// register a function
NotificationCenter          this \didRecord \theRequestingObject    "hello"

// now it has something to do
NotificationCenter        this \didRecord
```

```
hello

// unregister, thus releasing yourself for GC
NotificationCenter            this \didRecord \theRequestingObject

// theRequestingObject is no longer interested in this notification
NotificationCenter        this \didRecord


)
```

The listener argument is somewhat unimportant as far as you the client is concerned. It is used to identify the notification and to find it later to remove it when needed.

There can only be one notification per listener, but you may use anything for the listener object, such
as an arbitrary symbol.

```
 // two symbols
NotificationCenter          this \didRecord \thingOne    "this will get overwritten by the next regis-
tration"
NotificationCenter.register(this,\didRecord,\thingOne, { "do this".postln; });
NotificationCenter.register(this,\didRecord,\thingTwo, { "do this also".postln; });

NotificationCenter        this \didRecord

do this
do this also
)


(
NotificationCenter.register(this,\didRecord,this, { arg path,comments; path.postln; comments.postln; });


// after the addressing, an array of arguments can be supplied to be passed into the function
NotificationCenter        this \didRecord,[":SoundFiles:blurb.aiff",'yo mama']
:SoundFiles:blurb.aiff
yo mama
```

```
)
```

You can also remove the Notification registration by getting the layout to remove you when the window closes:

```
guiBody { arg layout;
layout.removeOnClose(
NotificationCenter.register(model,\didRecord,this, {
   // do things
});
);
}
```

ID: 173

NumChannels

ensures the output has the stated number of channels, regardless of the number of input channels.

NumChannels.ar(input, numChannels, mixdown)

input - the audio signal
numChannels - an integer
mixdown - true/false, whether you want to mixdown or just use the first channel

```
(
#1322ff {
  NumChannels
SinOsc.ar(100,0,0.2), // 1#1322ff becomes 2
2)
}.play
)

(
#1322ff {
  NumChannels
SinOsc.ar([100,200,300],0,0.2), // 3#1322ff becomes 2
2)
}.play
)

(
#1322ff {
  NumChannels
SinOsc.ar([100,200,300,100],0,0.2), // 4#1322ff becomes 2

2)
}.play
)
```

mono input is copied
multi-channels clumped and
if mixdown is true
mixed down
else
first channel used

see also [Mono]

ID: 174

PathName file and directory path utilities

superclass: Object

PathName is a utility Class for manipulating file names and paths.
It expects a path to a file, and lets you access parts of that file path.

Creation

```
*new(path)
```

path is a String which likely contains one or more / as typical for folder separation.

```
PathName     "MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
```

will be converted to your fully addressed home directory. Symbolic Links will be expanded, as per String-standardizePath.

Class Methods

tmp
tmp_(aPath)

Get or set the global temp directory as a **String**. This is used by **Buffer**, etc. By default this is "/tmp/" for Linux and OSX, and "/WINDOWS/TEMP/" for Windows.

Instance Methods

fileName

returns just the name of the file itself; i.e.
everything after the last slash in the full path.

```
(
var myPath;
myPath = PathName.new("MyDisk/SC 2.2.8 f/Sounds/FunkyChicken");
myPath.fileName.postln;
)
```

pathOnly

returns the full path up to the file name itself; i.e.
everything up to and including the last slash.
This is handy e.g. for storing several files in the same folder.

```
(
var myPath;
myPath = PathName.new("MyDisk/SC 2.2.8 f/Sounds/FunkyChicken");
myPath.pathOnly.postln;
)
```

isRelativePath
isAbsolutePath
asRelativePath

you MUST have correctly initialized the scroot classvar for this
to know what it is relative to !

folderName

returns only the name of the folder that the file is in;
i.e. everything in between the last but one and the last slash.

```
(
var myPath;
        PathName     "MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
myPath.folderName.postln;
)
```

fullPath

returns the full path name that PathName contains.

```
(
var myPath;
        PathName     "MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
myPath.fullPath.postln;
)
```

allFolders

returns a list of all the folder names in the pathname.

```
(
var myPath;
        PathName     "MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
myPath.allFolders.postln;
)
```

diskName

if path is an absolute path, returns the disk name; else a blank string.

```
(
var myPath;
        PathName     "MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
myPath.diskName.postln;
)

  // note the / at the start
var myPath;
        PathName     "/MyDisk/SC 2.2.8 f/Sounds/FunkyChicken"
myPath.diskName.postln;
)
```

endNumber

returns a number at the end of PathName.
Returns zero if there is no number.

```
PathName("floating1").endNumber.postln;

PathName("floating").endNumber.postln;
```

noEndNumbers

returns fullPath without any numbers at the end.

```
PathName("floating1").noEndNumbers.postln;

PathName("floating").noEndNumbers.postln;
```

nextName
generates a sensible next name for a file
by incrementing a number at the end of PathName,
or by adding one if there is none.

```
PathName("floating34").nextName.postln;

PathName("floating").nextName.postln;

PathName "floating12_3A4X_56.7"
```

This is useful for recording files with consecutive names,
and e.g. to generate a new filename when you don't want to
overwrite an existing file with the current name.

Here is an example that uses many instance methods.
Just pick any file to see all the parts of its path.

```
/*
(
GetFileDialog.new(
 { arg ok, path;
 var myPathName;
 if (ok,
  {
    myPathName = PathName.new(path);

    "New PathName object/   ".postc;
    myPathName.postln;

    "fileName only/   ".postc;
    myPathName.fileName.postln;

    "path up to file only/   ".postc;
    myPathName.pathOnly.postln;
```

```
    "folder Name/   ".postc;
    myPathName.folderName.postln;
  }
 )
 }
)
)
```

Choose a soundfile to put into the library,
using its foldername and filename/

```
(
GetFileDialog.new(
 { arg ok, path;
 var myPathName, myFile;
 if (ok,
  {
   myPathName = PathName.new(path);

   // read your file from disk, e.g.  a soundFile/

   myFile = SoundFile.new;
   if (myFile.readHeader(path),
    {
     Library.put(
      [ myPathName.folderName.asSymbol, myPathName.fileName.asSymbol ],
      myFile);
     ("Check LibMenu/ " ++ myPathName.folderName ++ " please.").postln;
    },
    { ("Could not read soundfile " ++ path ++ ".").postln; }
   );
  }
 )
 }
)
)
```

Save three tables in the same folder:

```
 Note:  The file name chosen in the dialog is ignored!
 The files are always named table1, table2, table3.
(
var table1, table2, table3;

table1 = Wavetable.sineFill(1024, [1,2,3]);
table2 = Signal.newClear.asWavetable;
table3 = Wavetable.sineFill(1024, Array.rand(64, 0.0, 1.0));

PutFileDialog.new(
 "Pick a folder for tables1-3/", "table1",
 { arg ok, path;
 var myPathName, myPathOnly;
 if (ok,
  {
   myPathName = PathName.new(path);

   myPathOnly = myPathName.pathOnly;

   table1.write(myPathOnly ++ "table1");
   table2.write(myPathOnly ++ "table2");
   table3.write(myPathOnly ++ "table3");
  }
 )
 }
)
)


*/
```

ID: 175

PlayPathButton

Superclass: ActionButton

*new(layout, path,minx)

when clicked, loads the object at *path* and plays it.
if already playing, stops.

minx is the minimum x width of the button.

// if you were to have something at :Patches:dhalf this would work

```
(
PlayPathButton(nil, "Patches/sc3batch1/crazycrazySUnj" )
)
```

ID: 176

UnicodeResponder

This can be used to replace a function in a view's keydownAction.
It matches modifier/unicode combinations and .values functions.

This is the best way to accurately match the exact modifier combination you want.

register(unicode, shift, caps, option, control, function)
true/false/nil:
must be present
should not be present
doesn't matter

```
(
    UnicodeResponder

// option down arrow
k.register( 63233, false,false,true,false, {
"option down".postln;
});

// shift-option down arrow
k.register( 63233 , true,false,true,false, {
 "shift option down"
});

w = SCWindow.new.front;
v = SCSlider.new(w,Rect(10,10,100,100));

v.keyDownAction = k;

v.focus;
)
```

normal(unicode -> function [, unicode -> function])
shift(unicode -> function [, unicode -> function])
control(unicode -> function [, unicode -> function])

option(unicode -> function [, unicode -> function])

```
The view in this example is merely to have something to focus on, it
does nothing else.
(
var w, l;
w= SCWindow("test").front;
l= SCListView(w, Rect(10, 10, 350, 350))
.items_({"eggs".scramble}.dup(12))
.focus
.keyDownAction_(
  UnicodeResponder
.normal(63232 -> {
    "normal arrow"
})
.shift(63232 -> {
    "shift arrrow"
})
.register( 63232, true, nil, false, true, {
    "shift control, with or without CAPS"
})
.normal( 97 -> {
"normal a".postln
})
.shift( $A -> {
"shift a".postln
})
)
)
Note that to match shift-a you have to specify "A", not "a"
```

You can also specify with ascii characters

```
(

var w, l;
w= SCWindow("test").front;
l= SCListView(w, Rect(10, 10, 350, 350))
.items_({"eggs".scramble}.dup(12))
.focus
```

```
.keyDownAction_(
  UnicodeResponder
.normal(
$a -> {
"a ".postln;
},
$b -> {
"b".postln;
},
$; -> {
";".postln;
},
$' -> {
"'".postln;
}
)
.shift(
$A -> {
"shift a".postln;
},
$B -> {
"shift b".postln;
},
$:  -> {
"shift ;".postln;
},
$" -> {
"shift '".postln;
}
);

)
)
```

see also KeyCodeResponder

If you merely check the modifier like so:
(modifer & optionModifier) == optionModifier
you will detect the presence of the options key,
but not if only the option key is present (eg. for shift-option)

ID: 177

ZArchive

superclass: File

A safe binary archive format. Supports large file sizes.

Compresses strings and symbols using a string lookup table.
(limit: 65536 different strings/symbols maximum)

Compresses repeated values
(limit: 4294967296 consecutive repeated items)

The text archive written by Object.writeArchive will store an object and
restore all of its internal variables. However, it will break with large file sizes
because it actually writes compilable code and there is a limit to that.

The binary archives written by Object.writeBinaryArchive will break if the
instance variables change in any of the classes you have archived.

This class is designed for raw data storage that you manually and explictly control.

You manually write to the archive and then should read from the archive in the same
order you wrote in. You could also write a Dictionary and then not worry about order.
This would also let you add or remove fields later without breaking the archives.

***write(path)**
open an archive for writing

writeItem(item) -
this will write a character specifying the class type of the object and then will write the object
in the smallest possible format.
Floats, Integers, Strings, Symbols, SequenceableCollections and Dictionaries all have support to write to the archive. eg. a Float writes a float to the file.
Strings and Symbols write using a string table, so your 10000 Events with \degree in them will only
need to save the word "degree" once.
All other objects will be saved asCompileString.

writeClose
finish the write session and close the file.

***read(path)**
open an archive for reading

readItem(expectedClass) -
read an item from the file. If expectedClass is non-nil, it will throw an error if the item is of
a different class.

<>version -
you may set the version number so that your various objects can check the version of the archive.
you need to store and retrieve the version number yourself, the ZArchive won't do it automatically.
Its just a convienient variable.

(

```
a = ZArchive.write("archiveTest");

a.writeItem(1.0.rand);
a.writeItem([1,2,3,4,5].choose);
a.writeItem("hello");
a.writeItem(
Event.make({
a = \a;
b = \b;
})
);
a.writeItem([1,2,3,4,5]);
a.writeItem( Ref(4.0) );
a.writeItem([
Event[
('time' -> 149.797), ('delta' -> 0.453), ('m' -> [ 'setVolAt', 0, 0.415356 ])
],
Event[
```

```
('time' -> 150.25), ('delta' -> 0.478), ('m' -> [ 'setVolAt', 0, 0.37382 ])
],
Event[
('time' -> 150.728), ('delta' -> 0.428), ('m' -> [ 'setVolAt', 0, 0.336438 ])
]
]);
a.writeItem([
IdentityDictionary[
\a -> "b",
"b" -> \c
]
]);
a.writeClose;
```

)

(

```
b =  ZArchive.read("archiveTest");
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;
b.readItem.postln;

b.close;
```

)

(

```
a = ZArchive.write("archiveTest");

a.writeItem(5);
```

```
a.writeItem( [ Event[
('time' -> 0), ('delta' -> 7.68278), ('m' -> [ 'state_', Environment[
('efxpath' -> ":Patches:justefx:4subtleDisturb er"), ('mixes' -> [ 0, 0.328532, 1, 0 ]), ('subject' -
> Environment[
('subject' -> Environment[
('paths' -> [ ":Patches:splash:chewy", ":Patches:twisters:wahfunk", ":Patches:riddims:slowrollzghet",
nil ]), ('amps' -> [ 0.177931, 0.42807, 0.219667, 0.7 ])
]), ('filterObjects' -> [ nil, nil, nil, nil ])
])
] ])
], Event[
('time' -> 7.68278), ('delta' -> 2.0898), ('m' -> [ 'selectByPath', 2, ":Patches:riddims:geekslut" ])

], Event[
('time' -> 9.77257), ('delta' -> 0.41796), ('m' -> [ 'setVolAt', 2, 0.197701 ])
], Event[
('time' -> 10.1905), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 2, 0.177931 ])
], Event[
('time' -> 10.5853), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 2, 0.160138 ])
], Event[
('time' -> 10.98), ('delta' -> 0.32508), ('m' -> [ 'setVolAt', 2, 0.144124 ])
], Event[
('time' -> 11.3051), ('delta' -> 8.75393), ('m' -> [ 'setVolAt', 2, 0.129711 ])
], Event[
('time' -> 20.059), ('delta' -> 8.96291), ('m' -> [ 'selectByPath', 2, ":Patches:riddims2:jRunnin" ])

], Event[
('time' -> 29.0219), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 2, 0.142683 ])
], Event[
('time' -> 29.4167), ('delta' -> 5.61923), ('m' -> [ 'setVolAt', 2, 0.156951 ])
],
Event[
('time' -> 35.0359), ('delta' -> 0.41796), ('m' -> [ 'setVolAt', 2, 0.172646 ])
], Event[
('time' -> 35.4539), ('delta' -> 2.71674), ('m' -> [ 'setVolAt', 2, 0.18991 ])
], Event[
('time' -> 38.1706), ('delta' -> 1.36998), ('m' -> [ 'setMixOnVoice', 2, 1 ])
], Event[
('time' -> 39.5406), ('delta' -> 0.3483), ('m' -> [ 'setMixOnVoice', 2, 0.85 ])
], Event[
```

```
('time' -> 39.8889), ('delta' -> 0.41796), ('m' -> [ 'setMixOnVoice', 2, 0.722501 ])
], Event[
('time' -> 40.3068), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.614126 ])
], Event[
('time' -> 40.6784), ('delta' -> 1.161), ('m' -> [ 'setMixOnVoice', 2, 0.522007 ])
], Event[
('time' -> 41.8394), ('delta' -> 2.85606), ('m' -> [ 'setMixOnVoice', 1, 1 ])
], Event[
('time' -> 44.6954), ('delta' -> 1.7415), ('m' -> [ 'setMixOnVoice', 1, 1 ])
], Event[
('time' -> 46.4369), ('delta' -> 2.85606), ('m' -> [ 'wakeEffectByPath', ":Patches:justefx:pitchCasStereoSprd"
])
],
Event[
('time' -> 49.293), ('delta' -> 0.41796), ('m' -> [ 'setMixOnVoice', 1, 0.85 ])
], Event[
('time' -> 49.7109), ('delta' -> 0.696599), ('m' -> [ 'setMixOnVoice', 1, 0.7225 ])
], Event[
('time' -> 50.4075), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 1, 0.385263 ])
], Event[
('time' -> 50.8023), ('delta' -> 0.44118), ('m' -> [ 'setVolAt', 1, 0.346736 ])
], Event[
('time' -> 51.2435), ('delta' -> 11.4707), ('m' -> [ 'setVolAt', 1, 0.312063 ])
], Event[
('time' -> 62.7141), ('delta' -> 1.46286), ('m' -> [ 'selectByPath', 0, ":Patches:clouds:newjetengine"
])
], Event[
('time' -> 64.177), ('delta' -> 0.673379), ('m' -> [ 'setVolAt', 0, 0.160138 ])
], Event[
('time' -> 64.8504), ('delta' -> 0.51084), ('m' -> [ 'setVolAt', 0, 0.144124 ])
], Event[
('time' -> 65.3612), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 0, 0.129711 ])
], Event[
('time' -> 65.7559), ('delta' -> 8.89325), ('m' -> [ 'setVolAt', 0, 0.11674 ])
],
Event[
('time' -> 74.6492), ('delta' -> 4.50468), ('m' -> [ 'setVolAt', 0, 0.128414 ])
], Event[
('time' -> 79.1539), ('delta' -> 1.92726), ('m' -> [ 'selectByPath', 0, ":Patches:clouds:screamspac" ])
```

```
], Event[
('time' -> 81.0811), ('delta' -> 10.449), ('m' -> [ 'setVolAt', 0, 0.115573 ])
], Event[
('time' -> 91.5301), ('delta' -> 9.84527), ('m' -> [ 'sleepVoice', 0 ])
], Event[
('time' -> 101.375), ('delta' -> 0.3483), ('m' -> [ 'setVolAt', 2, 0.208902 ])
], Event[
('time' -> 101.724), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 2, 0.229792 ])
], Event[
('time' -> 102.118), ('delta' -> 2.06658), ('m' -> [ 'setVolAt', 2, 0.252771 ])
], Event[
('time' -> 104.185), ('delta' -> 0.32508), ('m' -> [ 'setMixOnVoice', 2, 0.443706 ])
], Event[
('time' -> 104.51), ('delta' -> 0.39474), ('m' -> [ 'setMixOnVoice', 2, 0.377151 ])
], Event[
('time' -> 104.905), ('delta' -> 2.322), ('m' -> [ 'setMixOnVoice', 2, 0.320578 ])
],
Event[
('time' -> 107.227), ('delta' -> 1.161), ('m' -> [ 'setMixOnVoice', 2, 0.272492 ])
], Event[
('time' -> 108.388), ('delta' -> 1.95048), ('m' -> [ 'setMixOnVoice', 2, 1 ])
], Event[
('time' -> 110.338), ('delta' -> 0.41796), ('m' -> [ 'setMixOnVoice', 1, 0.614125 ])
], Event[
('time' -> 110.756), ('delta' -> 0.928799), ('m' -> [ 'setMixOnVoice', 1, 0.73695 ])
], Event[
('time' -> 111.685), ('delta' -> 10.1471), ('m' -> [ 'setMixOnVoice', 1, 1 ])
], Event[
('time' -> 121.832), ('delta' -> 1.71828), ('m' -> [ 'setVolAt', 1, 0.280856 ])
], Event[
('time' -> 123.55), ('delta' -> 2.0898), ('m' -> [ 'setVolAt', 1, 0.252771 ])
], Event[
('time' -> 125.64), ('delta' -> 6.13007), ('m' -> [ 'setVolAt', 1, 0.227494 ])
], Event[
('time' -> 131.77), ('delta' -> 1.99692), ('m' -> [ 'selectByPath', 2, ":Patches:plusefx:musiqueConcrete"
])
], Event[
('time' -> 133.767), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 1 ])
],
Event[
```

```
('time' -> 134.139), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.85 ])
], Event[
('time' -> 134.51), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.722501 ])
], Event[
('time' -> 134.882), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.614126 ])
], Event[
('time' -> 135.253), ('delta' -> 0.41796), ('m' -> [ 'setMixOnVoice', 2, 0.522007 ])
], Event[
('time' -> 135.671), ('delta' -> 0.30186), ('m' -> [ 'setMixOnVoice', 2, 0.443706 ])
], Event[
('time' -> 135.973), ('delta' -> 0.3483), ('m' -> [ 'setMixOnVoice', 2, 0.377152 ])
], Event[
('time' -> 136.321), ('delta' -> 0.3483), ('m' -> [ 'setMixOnVoice', 2, 0.32058 ])
], Event[
('time' -> 136.67), ('delta' -> 0.3483), ('m' -> [ 'setMixOnVoice', 2, 0.272493 ])
], Event[
('time' -> 137.018), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.231619 ])
], Event[
('time' -> 137.39), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 2, 0.196877 ])
],
Event[
('time' -> 137.761), ('delta' -> 0.39474), ('m' -> [ 'setMixOnVoice', 2, 0.167345 ])
], Event[
('time' -> 138.156), ('delta' -> 0.39474), ('m' -> [ 'setMixOnVoice', 2, 0.142243 ])
], Event[
('time' -> 138.551), ('delta' -> 1.8576), ('m' -> [ 'setMixOnVoice', 2, 0.120907 ])
], Event[
('time' -> 140.408), ('delta' -> 0.3483), ('m' -> [ 'setVolAt', 2, 0.278048 ])
], Event[
('time' -> 140.756), ('delta' -> 0.32508), ('m' -> [ 'setVolAt', 2, 0.305853 ])
], Event[
('time' -> 141.082), ('delta' -> 0.37152), ('m' -> [ 'setVolAt', 2, 0.336438 ])
], Event[
('time' -> 141.453), ('delta' -> 0.3483), ('m' -> [ 'setVolAt', 2, 0.370082 ])
], Event[
('time' -> 141.801), ('delta' -> 0.37152), ('m' -> [ 'setVolAt', 2, 0.40709 ])
], Event[
('time' -> 142.173), ('delta' -> 2.73996), ('m' -> [ 'setVolAt', 2, 0.447799 ])
], Event[
('time' -> 144.913), ('delta' -> 60.5577), ('m' -> [ 'setVolAt', 2, 0.492579 ])
```

```
],
Event[
('time' -> 205.471), ('delta' -> 1.48608), ('m' -> [ 'setVolAt', 2, 0.541837 ])
], Event[
('time' -> 206.957), ('delta' -> 1.90404), ('m' -> [ 'setVolAt', 2, 0.59602 ])
], Event[
('time' -> 208.861), ('delta' -> 0.39474), ('m' -> [ 'setMixOnVoice', 1, 1 ])
], Event[
('time' -> 209.255), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 1, 0.85 ])
], Event[
('time' -> 209.627), ('delta' -> 0.37152), ('m' -> [ 'setMixOnVoice', 1, 0.7225 ])
], Event[
('time' -> 209.998), ('delta' -> 1.20744), ('m' -> [ 'setMixOnVoice', 1, 0.614126 ])
], Event[
('time' -> 211.206), ('delta' -> 0.41796), ('m' -> [ 'setMixOnVoice', 1, 0.522007 ])
], Event[
('time' -> 211.624), ('delta' -> 0.719819), ('m' -> [ 'setMixOnVoice', 1, 0.443706 ])
], Event[
('time' -> 212.344), ('delta' -> 0.39474), ('m' -> [ 'setMixOnVoice', 1, 0.37715 ])
], Event[
('time' -> 212.738), ('delta' -> 0.32508), ('m' -> [ 'setMixOnVoice', 1, 0.320578 ])
],
Event[
('time' -> 213.063), ('delta' -> 0.32508), ('m' -> [ 'setMixOnVoice', 1, 0.272492 ])
], Event[
('time' -> 213.389), ('delta' -> 0.3483), ('m' -> [ 'setMixOnVoice', 1, 0.231618 ])
], Event[
('time' -> 213.737), ('delta' -> 0.39474), ('m' -> [ 'setVolAt', 1, 0.204744 ])
], Event[
('time' -> 214.132), ('delta' -> 0.37152), ('m' -> [ 'setVolAt', 1, 0.18427 ])
], Event[
('time' -> 214.503), ('delta' -> 0.32508), ('m' -> [ 'setVolAt', 1, 0.165843 ])
], Event[
('time' -> 214.828), ('delta' -> 0.3483), ('m' -> [ 'setVolAt', 1, 0.149259 ])
], Event[
('time' -> 215.176), ('delta' -> 0.37152), ('m' -> [ 'setVolAt', 1, 0.134333 ])
], Event[
('time' -> 215.548), ('delta' -> 0.44118), ('m' -> [ 'setVolAt', 1, 0.1209 ])
], Event[
('time' -> 215.989), ('delta' -> 1.92726), ('m' -> [ 'setVolAt', 1, 0.10881 ])
```

```
], Event[
('time' -> 217.916), ('m' -> [ 'setVolAt', 1, 0.0979286 ])
]] );

a.writeClose;
```

)

(

```
b =  ZArchive.read("archiveTest");
b.readItem.postln;
b.readItem.postln;

b.close;
```

)

Repetition compression

identical values or objects that repeat are compressed.

(

```
a = ZArchive.write("archiveTest");

a.writeItem( [
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
```

```
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2
]);

a.writeClose;
```

)
this is about 42 bytes.

Identical objects get reconstituted as equal but independant objects.

(

```
b = ZArchive.read("archiveTest");
b.readItem.postln;
b.close;
```

)

(

```
a = ZArchive.write("archiveTest");


a.writeItem(nil);
a.writeItem("word");
a.writeClose;
```

)

(

```
b =  ZArchive.read("archiveTest");
b.readItem.postln;
b.readItem.postln;
// one more
b.readItem.postln;



b.close;
```

)

asZArchive

relative to your Document directory

(

```
a = "archiveTest".asZArchive;



a.writeItem(1.0.rand);
a.writeItem([1,2,3,4,5].choose);
a.writeItem("hello");
a.writeItem(
Event.make({
a = \a;
b = \b;
})
);
a.writeItem([1,2,3,4,5]);
a.writeItem( Ref(4.0) );
```

```
a.writeItem([
Event[
('time' -> 149.797), ('delta' -> 0.453), ('m' -> [ 'setVolAt', 0, 0.415356 ])
],
Event[
('time' -> 150.25), ('delta' -> 0.478), ('m' -> [ 'setVolAt', 0, 0.37382 ])
],
Event[
('time' -> 150.728), ('delta' -> 0.428), ('m' -> [ 'setVolAt', 0, 0.336438 ])
]
]);
a.writeItem([
IdentityDictionary[
\a -> "b",
"b" -> \c
]
]);
a.writeClose;
```

)

adding support for your custom class

```
SomeClass {

writeZArchive { arg akv;
// turn a path into an archive if needed
akv = akv.asZArchive;

akv.writeItem(columns.size);
// we know the column objects have their own support
columns.do({| c| c.writeZArchive(akv) });

akv.writeItem(name);
akv.writeItem(beats);
akv.writeItem(beatsPerBar);
}
readZArchive { arg akv;
columns = Array.fill( akv.readItem(Integer) ,{ arg i;
```

```
// call the custom column object read
Tracker.readZArchive(akv)
});

name = akv.readItem; // could be an Integer or a String !
beats = akv.readItem(Float);
beatsPerBar = akv.readItem(Float);
}
```

When you have large arrays of floats etc., you can bypass writeItem (which will write a character to specify the class type) and directly use putFloat, putInt32 etc. (methods of File)

but you must call saveCount before starting to use any of these methods. This is because any counted repetitions need to be closed off and written to the file before you start manually putting floats etc.

7 Files

ID: 178

File

Superclass: UnixFILE

A class for reading and writing files. Not sound files.

see also the superclass for further docs.

*new(pathname, mode)

Create a File instance and open the file. If the open fails, isOpen will return false.

pathname

a String containing the path name of the file to open.

mode
a String indicating one of the following modes:

"r" - read only text
"w" - read and write text
"a" - read and append text
"rb" - read only binary
"wb" - read and write binary
"ab" - read and append binary

"r+" - read only text
"w+" - read and write text
"a+" - read and append text
"rb+" - read only binary
"wb+" - read and write binary
"ab+" - read and append binary

open

Open the file. Files are automatically opened upon creation, so this call is only necessary if you are closing and opening the same file object repeatedly.

NOTE: it is possible when saving files with a standard file dialog to elect to "hide the extension"
and save it as RTF. When opening the file you must specify the real filename: "filename.rtf",
even though you can't see in file load dialogs or in the Finder.

close
Close the file.

***exists(pathName)**
answers if a file exists at that path.

***delete(pathName)**
deletes the file at that path.
use only for good, never for evil.

***openDialog(prompt,sucessFunc,cancelFunc)**

***saveDialog("hello",{},{})**
not yet implemented

***getcwd**
POSIX standard 'get current working directory'.

```
    // example;
File.getcwd
```

***use(function)**
open the file, evaluate the function with the file and close it.

readAllString
Reads the entire file as a String.

readAllStringRTF

Reads the entire file as a String, stripping RTF formatting.

Examples:

```
// write some string to a file:
(
var f, g;
    File "test" "w"
        "Does this work?\n is this thing on ?\n"
f.close;
)

// read it again:
(
    File "test" "r"
g.readAllString.postln;
g.close;
)

// try the above with File.use:

File     "test"  "w"    | f|         "Doesn't this work?\n is this thing really on ?\n"
File.use("test", "r", { | f|  f.readAllString.postln })


// more file writing/reading examples:
(
var h, k;
    File "test2"  "wb"
h.inspect;
h.write( FloatArray[1.1, 2.2, 3.3, pi, 3.sqrt] );
h.close;

    File "test2"  "rb"
(k.length div:  4).do({ k.getFloat.postln; });
k.close;
)


(
```

```
var f, g;
    File "test3" "w"
100.do({ f.putChar([$a, $b, $c, $d, $e, $\n].choose); });
f.close;

    File "test3" "r"
g.readAllString.postln;
g.close;

g = File("test3","r");
g.getLine(1024).postln;
"*".postln;
g.getLine(1024).postln;
"**".postln;
g.getLine(1024).postln;
"***".postln;
g.getLine(1024).postln;
"****".postln;
g.close;

)

(
//var f, g;
    File "test3" "wb"
f.inspect;
100.do({ f.putFloat(1.0.rand); });

f.inspect;
f.close;

    File "test3" "rb"
100.do({
g.getFloat.postln;
});
g.inspect;
g.close;

)
```

```
(
//var f, g;
    File "test3" "r"
f.inspect;

f.getLine(1024).postln;

f.close;


)
```

ID: 179

Pipe

superclass: UnixFILE

Pipe stdin to, or stdout from, a unix shell command. Pipe treats the shell command as if it were a **UnixFILE**, and returns nil when done. See **UnixFILE** for details of the access methods. Pipe must be explicitly closed. Do not rely on the garbage collector to do this for you!

Note: due to a bug in the current os x (10.3) , unix commands like pipe do not work when the server is booted.
for now one has to quit the server, otherwise sc crashes.

***new(commandLine, mode)**

commandLine - A String representing a valid shell command.
mode - A string representing the mode. Valid modes are "w" (pipe to stdin) and "r" (pipe from stdout).

close
Closes the pipe. You must do this explicitly before the Pipe object is garbage collected.

Examples

```
// quit the server
s.quit;

// this pipes in stdout from ls
(
var p, l;
    Pipe     "ls -l"  "r"        // list directory contents in long format
l = p.getLine;        // get the first line
while({l.notNil}, {l.postln; l = p.getLine; }); // post until l = nil
                // close the pipe to avoid that nasty buildup
)
```

A more time-intensive request:

```
(
var p, l;
    Pipe     "ping -c10 sourceforge.net"  "r"        // list directory contents in long format
l = p.getLine;        // get the first line
while({l.notNil}, {l.postln; l = p.getLine; }); // post until l = nil
                  // close the pipe to avoid that nasty buildup
)
```

ID: 180

SoundFile

In most cases you will wish to send commands to the server to get it to load SoundFiles directly into Buffers. You will not use this class for this. See **Server-Command-Reference.**

This class is used to check the size, format, channels etc. when the client needs this information about a SoundFile.

Some manipulation of the sound file data is possible. Soundfile data can be read and written incrementally, so with properly designed code, there is no restriction on the file size.

```
(

f = SoundFile.new;

            "sounds/a11wlk01.wav"

f.inspect;

f.close;


)
```

Creating

new

Creates a new SoundFile instance.

Read/Write

openRead(inPathname)

Read the header of a file. Answers a Boolean whether the read was successful.
sets the numFrames,numChannels and sampleRate. does not set the headerFormat and sampleFormat.
inPathname - a String specifying the path name of the file to read.

readData(rawArray)

Reads the sample data of the file into the raw array you supply. You must have already called openRead.

The raw array must be a FloatArray. Regardless of the sample format of the file, the array will be populated with floating point values. For integer formats, the floats will all be in the range -1..1.

The size of the FloatArray determines the maximum number of single samples (*not* sample frames) that will be read. If there are not enough samples left in the file, the size of the array after the readData call will be less than the original size.

When you reach EOF, the array's size will be 0. Checking the array size is an effective termination condition when looping through a sound file. See the method channelPeaks for example.

openWrite(inPathname)

Write the header of a file. Answers a Boolean whether the write was successful.
inPathname - a String specifying the path name of the file to write.

writeData(rawArray)

Writes the rawArray to the sample data of the file. You must have already called openWrite.

The raw array must be a FloatArray or Signal, with all values between -1 and 1 to avoid clipping during playback.

Example:

```
(
f = SoundFile.new.headerFormat_("AIFF").sampleFormat_("int16").numChannels_(1);
```

```
            "sounds/sfwrite.aiff"
 // sawtooth
b = Signal.sineFill(100, (1..20).reciprocal);
 // write multiple cycles (441 * 100 = 1 sec worth)
441.do({ f.writeData(b) });
f.close;
)
```

isOpen
answers if the file is open

close
closes the file

duration
the duration in seconds of the file

Normalizing

***normalize(path, outPath, newHeaderFormat, newSampleFormat, startFrame, numFrames, maxAmp, linkChannels, chunkSize)**

normalize(outPath, newHeaderFormat, newSampleFormat, startFrame, numFrames, maxAmp, linkChannels, chunkSize)

Normalizes a soundfile to a level set by the user. The normalized audio will be written into a second file.

Note: While the normalizer is working, there is no feedback to the user. It will look like SuperCollider is hung, but it will eventually complete the operation.

Arguments:

path: a path to the source file
outPath: a path to the destination file
newHeaderFormat: the desired header format of the new file; if not specified, the header format of the source file will be used
newSampleFormat: the desired sample format of the new file; if not specified, the sample format of the source file will be used
startFrame: an index to the sample frame to start normalizing (default 0)

numFrames: the number of sample frames to copy into the destination file (default nil, or entire soundfile)
maxAmp: the desired maximum amplitude. Provide a floating point number or, if desired, an array to specify a different level for each channel. The default is 1.0.
linkChannels: a Boolean specifying whether all channels should be scaled by the same amount. The default is **true**, meaning that the peak calculation will be based on the largest sample in any channel. If false, each channel's peak will be calculated independently and all channels will be scaled to maxAmp (this would alter the relative loudness of each channel).
chunkSize: how many samples to read at once (default is 4194304, or 16 MB)

Using the class method (SoundFile.normalize) will automatically open the source file for you. You may also openRead the SoundFile yourself and call normalize on it. In that case, the source path is omitted because the file is already open.

The normalizer may be used to convert a soundfile from one sample format to another (e.g., to take a floating point soundfile produced by SuperCollider and produce an int16 or int24 soundfile suitable for use in other applications).

Instance Variables

<**path**

Get the pathname of the file. This variable is set via the openRead or openWrite calls.

<>**headerFormat**

This is a String indicating the header format which was read by openRead and will be written by openWrite. In order to write a file with a certain header format you set this variable.

Sound File Format symbols:
header formats:
read/write formats:
"AIFF", - Apple's AIFF
"WAV","RIFF" - MicrosSoft .WAV
"Sun", - NeXT/Sun
"IRCAM", - old IRCAM format
"none" - no header = raw data

A huge number of other formats are supported read only.

<>sampleFormat

A String indicating the format of the sample data which was read by openRead and will be written by openWrite. Not all header formats support all sample formats. The possible header formats are:

sample formats:
"int8", "int16", "int24", "int32"
"mulaw", "alaw",
"float32"

not all header formats support all sample formats.

<numFrames

The number of sample frames in the file.

<numChannels

The number of channels in the file.

<>sampleRate

The sample rate of the file.

ID: 181

UnixFILE

superclass: IOStream

An abstract class. See File and Pipe

(docs incomplete)

isOpen

returns whether the file is open. An open request can fail if a file cannot be found for example.
This method lets you test that the open call succeeded.

length

Answer the length of the file.

pos

Answer the current file position

seek(offset, origin)

Seek to an offset from the origin.
offset - an offset in bytes.
origin - one of the following Integers:

0 - seek from beginning of file.
1 - seek from current position in file.
2 - seek from end of file.

write(item)

Writes an item to the file.
item - one of the following:
Float
Integer,
Char,
Color,
Symbol - writes the name of the Symbol as a C string.
RawArray - write the bytes from any RawArray in big endian.

getLine

Reads and returns a String up to lesser of next newline or 1023 chars.

getChar

read one byte and return as a Char

getInt8

read one byte and return as a Integer.

getInt16

read two bytes and return as an Integer.

getInt32

read four bytes and return as an Integer.

getFloat

read four bytes and return as a Float.

getDouble

read eight bytes and return as a Float.

putChar

write a Char as one byte.

putInt8

write an Integer as one byte.

putInt16

write an Integer as two bytes.

putInt32

write an Integer as four bytes.

putFloat

write a Float as four bytes.

putDouble

write a Float as eight bytes.

putString

write a null terminated String.

readAllString

Reads the entire file as a String.

readAllInt8

Reads the entire file as an Int8Array.

readAllInt16

Reads the entire file as an Int16Array.

readAllInt32

Reads the entire file as an Int32Array.

readAllFloat

Reads the entire file as an FloatArray.

readAllDouble

Reads the entire file as an DoubleArray.

8 Geometry

ID: 182

Point Cartesian point

Superclass: Object

Defines a point on the Cartesian plane.

Creation

new(inX, inY)

defines a new point.

Accessing

x

get the x coordinate value.

y

get the y coordinate value.

x_(aValue)

set the x coordinate value.

y_(aValue)

set the y coordinate value.

set(inX, inY)

Sets the point x and y values.

Testing

== aPoint

answers a Boolean whether the receiver equals the argument.

hash

returns a hash value for the receiver.

Math

+ aPointOrScalar

Addition.

- aPointOrScalar

Subtraction.

*** aPointOrScalar**

Multiplication.

/ aPointOrScalar

Division.

translate(aPoint)

Addition by a Point.

scale(aPoint)

Multiplication by a Point.

rotate(angle)

Rotation about the origin by the angle given in radians.

abs

Absolute value of the point.

rho

return the polar coordinate radius of the receiver.

theta

return the polar coordinate angle of the receiver.

dist(aPoint)

return the distance from the receiver to aPoint.

transpose

return a Point whose x and y coordinates are swapped.

round(quantum)

round the coordinate values to a multiple of quantum.

trunc(quantum)

truncate the coordinate values to a multiple of quantum.

Conversion

asPoint

returns the receiver.

asComplex

returns a complex number with x as the real part and y as the imaginary part.

asString

return a string representing the receiver.

asShortString

return a short string representing the receiver.

paramsCompileString

represent parameters to 'new' as compileable strings. (See Object::asCompileString)

ID: 183

Rect rectangle

Class methods:

new(inLeft, inTop, inWidth, inHeight)

return a new Rect with the given upper left corner and dimensions.

newSides(inLeft, inTop, inRight, inBottom)

return a new Rect with the given boundaries.

fromPoints(inPoint1, inPoint2)

return a new Rect defined by the given Points.

Instance methods:

left
top
right
bottom

Get the value of the boundary.

left_(aValue)
top_(aValue)
right_(aValue)
bottom_(aValue)

Set the value of the boundary.

set(inLeft, inTop, inRight, inBottom)

set the boundaries to the given values.

setBy(inLeft, inTop, inWidth, inHeight)

set the upper left corner and dimensions.

setExtent(inWidth, inHeight)

set the dimensions.

width

return the width.

height

return the height,

width_(aValue)

set the width.

height_(aValue)

set the height.

origin

return the upper left corner as a Point.

corner

return the lower right corner as a Point.

extent

return a Point whose x value is the height and whose y value is the width.

leftTop

return the upper left corner as a Point.

rightTop

return the upper right corner as a Point.

leftBottom

return the lower left corner as a Point.

rightBottom

return the lower right corner as a Point.

moveBy(x, y)

returns a new Rect which is offset by x and y.

moveTo(x, y)

returns a new Rect whose upper left corner is moved to (x, y).

moveToPoint(aPoint)

returns a new Rect whose upper left corner is moved to aPoint.

resizeBy(x, y)

returns a new Rect whose dimensions have been changed by (x, y).

resizeTo(x, y)

returns a new Rect whose dimensions are (x, y).

insetBy(x, y)

returns a new Rect whose boundaries have been inset by (x, y).

insetAll(insetLeft, insetTop, insetRight, insetBottom)

returns a new Rect whose boundaries have been inset by the given amounts.

contains(aPoint)

answers whether aPoint is in the receiver.

union(aRect)
| **aRect**

returns a new Rect which contains the receiver and aRect.

sect(aRect)
& aRect

returns a new Rect which is the intersection of the receiver and aRect.

9 Getting-Started

ID: 184

Buffers

Buffers represent server buffers, which are ordered arrays of floats on the server. 'float' is short for floating point number, which means a number with a decimal point, like 1.3. This is in contrast to integers, which are positive or negative whole numbers (or zero), and are written without decimal points. So 1 is an integer, but 1.0 is a float.

Server buffers can be single or multichannel, and are the usual way of storing data server-side. Their most common use is to hold soundfiles in memory, but any sort of data that can be represented by floats can be stored in a buffer.

Like busses, the number of buffers is set before you boot a server (using **[ServerOptions]**), but before buffers can be used, you need to allocate memory to them, which is an asynchronous step. Also like busses, buffers are numbered, starting from 0. Using Buffer takes care of allocating numbers, and avoids conflicts.

You can think of buffers as the server-side equivalent of an Array, but without all the elegant OOP functionality. Luckily with Buffer, and the ability to manipulate data in the client app when needed, you can do almost anything you want with buffer data. A server's buffers are global, which is to say that they can be accessed by any synth, and by more than one at a time. They can be written to or even changed in size, *while* they are being read from.

Many of Buffer's methods have numerous arguments. Needless to say, for full information see the **[Buffer]** help file.

Making a Buffer Object and Allocating Memory

Making a Buffer object and allocating the necessary memory in the server app is quite easy. You can do it all in one step with Buffer's alloc method:

```
s.boot;
b = Buffer                    // allocate 2 channels, and 100 frames
b.free;      // free the memory (when you're finished using it)
```

The example above allocates a 2 channel buffer with 100 frames. The actual number of values stored is numChannels * numFrames, so in this case there will be 200 floats. So each frame is in this case a pair of values.

If you'd like to allocate in terms of seconds, rather than frames, you can do so like this:

```
b = Buffer.alloc(s, s.sampleRate * 8.0, 2); // an 8 second stereo buffer
 b.free;
```

Buffer's 'free' method frees the memory on the server, and returns the Buffer's number for reallocation. You should not use a Buffer object after doing this.

Using Buffers with Sound Files

Buffer has another class method called 'read', which reads a sound file into memory, and returns a Buffer object. Using the UGen PlayBuf, we can play the file.

```
    // read a soundfile
  b = Buffer         "sounds/a11wlk01.wav"


  // now play it
(
x = SynthDef("tutorial-PlayBuf",{ arg out = 0, bufnum;
Out.ar( out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum))
)
}).play(s,[\bufnum, b.bufnum ]);
)
x.free; b.free;
```

PlayBuf.ar has a number of arguments which allow you to control various aspects of how it works. Take a look at the **[PlayBuf]** helpfile for details of them all, but for now lets just concern ourselves with the first three, used in the example above.

```
      PlayBuf
   1,      // number of channels
   bufnum,      // number of buffer to play
   BufRateScale             // rate of playback
)
```

Number of channels: When working with PlayBuf you must let it know how many channels any buffer it will read in will have. *You cannot make this an argument in the SynthDef and change it later.* Why? Remember that SynthDefs must have a fixed

number of output channels. So a one channel PlayBuf is *always* a one channel PlayBuf. If you need versions that can play varying numbers of channels then make multiple SynthDefs or use Function-play.

Bufffer Number: As noted above, Buffers are numbered, starting from zero. You can get a Buffer's number using its 'bufnum' method. This is done at the end of the SynthDef above, where it is passed in as an argument to the resulting Synth. (Note that SynthDef-play allows you to include an array of arguments, just like Synth-new.)

Rate of Playback: A rate of 1 would be normal speed, 2 twice as fast, etc. But here we see a UGen called BufRateScale. What this does is check the samplerate of the the buffer (this is set to correspond to that of the soundfile when it is loaded) and outputs the rate which would correspond to normal speed. This is useful because the soundfile we loaded (a11wlk01.wav) actually has a samplerate of 11025 Hz. With a rate of 1, PlayBuf would play it back using the sampling rate of the server, which is usually 44100 Hz, or four times as fast! BufRateScale thus brings things back to normal.

Streaming a File in From Disk

In some cases, for instance when working with very large files, you might not want to load a sound completely into memory. Instead, you can stream it in from disk a bit at a time, using the UGen DiskIn, and Buffer's 'cueSoundFile' method:

```
(
  SynthDef "tutorial-Buffer-cue"   arg
Out.ar(out,
DiskIn.ar( 1, bufnum )
)
}).send(s);
)


b = Buffer.cueSoundFile(s,"sounds/a11wlk01-44_1.aiff", 0, 1);
y = Synth.new("tutorial-Buffer-cue", [\bufnum,b.bufnum], s);


b.free; y.free;
```

This is not as flexible as PlayBuf (no rate control), but can save memory.

More on Instance Variables and Action Functions

Now a little more OOP. Remember that individual Objects store data in *instance variables*. Some instance variables have what are called getter or setter methods, which allow you to get or set their values. We've already seen this in action with Buffer's 'bufnum' method, which is a getter for its buffer number instance variable.

Buffer has a number of other instance variables with getters which can provide helpful information. The ones we're interested in at the moment are numChannels, numFrames, and sampleRate. These can be particularly useful when working with sound files, as we may not have all this information at our fingertips before loading the file.

```
  // watch the post window
 b = Buffer          "sounds/a11wlk01.wav"
b.bufnum;
b.numFrames;
b.numChannels;
b.sampleRate;
b.free;
```

Now (like with the example using an action function in our Bus-get example; see **[Busses]**) because of the small messaging latency between client and server, instance variables will not be immediately updated when you do something like read a file into a buffer. For this reason, many methods in Buffer take action functions as arguments. Remember that an action function is just a Function that will be evaluated after the client has received a reply, and has updated the Buffer's vars. It is passed the Buffer object as an argument.

```
    // with an action function
  // note that the vars are not immediately up-to-date
(
b = Buffer.read(s, "sounds/a11wlk01.wav", action:  { arg buffer;
("numFrames after update:" + buffer.numFrames).postln;
x = { PlayBuf.ar(1, buffer.bufnum, BufRateScale.kr(buffer.bufnum)) }.play;
});


  // Note that the next line will execute BEFORE the action function
("numFrames before update:" + b.numFrames).postln;
)
x.free; b.free;
```

In the example above, the client sends the read command to the server app, along with

a request for the necessary information to update the Buffer's instance variables. It then cues the action function to be executed when it receives the reply, and continues executing the block of code. That's why the 'Before update...' line executes first.

Recording into Buffers

In addition to PlayBuf, there's a UGen called RecordBuf, which lets you record into a buffer.

```
b = Buffer.alloc(s, s.sampleRate * 5, 1); // a 5 second 1 channel Buffer


 // record for four seconds
(
x = SynthDef("tutorial-RecordBuf",{ arg out=0,bufnum=0;
var noise;
  noise = PinkNoise           // record some PinkNoise
  RecordBuf                     // by default this loops
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)


 // free the record synth after a few seconds
x.free;


 // play it back
(
SynthDef("tutorial-playback",{ arg out=0,bufnum=0;
var playbuf;
playbuf = PlayBuf.ar(1,bufnum);
  FreeSelfWhenDone              // frees the synth when the PlayBuf has played through once
Out.ar(out, playbuf);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
b.free;
```

See the **[RecordBuf]** help file for details on all of its options.

Accessing Data

Buffer has a number of methods to allow you to get or set values in a buffer. Buffer-get

and Buffer-set are straightforward to use and take an index as an argument. Multichannel buffers interleave their data, so for a two channel buffer index 0 = frame1-chan1, index 1 = frame1-chan2, index 2 = frame2-chan1, and so on. 'get' takes an action function.

```
b = Buffer.alloc(s, 8, 1);
 b.set(7, 0.5);     // set the value at 7 to 0.5
 b.get(7, {| msg|               // get the value at 7 and post it when the reply is received
b.free;
```

The methods 'getn' and 'setn' allow you to get and set ranges of adjacent values. 'setn' takes a starting index and an array of values to set, 'getn' takes a starting index, the number of values to get, and an action function.

```
b = Buffer.alloc(s,16);
b.setn(0, [1, 2, 3]);       // set the first 3 values
b.getn(0, 3, {| msg|  msg.postln});    // get them
b.setn(0, Array.fill(b.numFrames, {1.0.rand})); // fill the buffer with random values
b.getn(0, b.numFrames, {| msg|  msg.postln});  // get them
b.free;
```

There is an upper limit on the number of values you can get or set at a time (usually 1633 when using UDP, the default). This is because of a limit on network packet size. To overcome this Buffer has two methods, 'loadCollection' and 'loadToFloatArray' which allow you to set or get large amounts of data by writing it to disk and then loading to client or server as appropriate.

```
(
 // make some white noise
v = FloatArray.fill(44100, {1.0.rand2});
b = Buffer.alloc(s, 44100);
)
(
 // load the FloatArray into b, then play it
b.loadCollection(v, action:  {| buf|
x = { PlayBuf.ar(buf.numChannels, buf.bufnum, BufRateScale.kr(buf.bufnum), loop:  1)
* 0.2 }.play;
});
)
x.free;
```

```
 // now get the FloatArray back, and compare it to v; this posts 'true'
 // the args 0, -1 mean start from the beginning and load the whole buffer
b.loadToFloatArray(0, -1, {| floatArray|  (floatArray == v).postln });
b.free;
```

A FloatArray is just a subclass of Array which can only contain floats.

Plotting and Playing

Buffer has two useful convenience methods: 'plot' and 'play'.

```
  // see the waveform
 b = Buffer        "sounds/a11wlk01.wav"
b.plot;


 // play the contents
 // this takes one arg:  loop.  If false (the default) the resulting synth is
 // freed automatically
 b.play;     // frees itself
 x = b.play(true    // loops so doesn't free
x.free; b.free;
```

Other Uses For Buffers

In addition to being used for loading in sound files, buffers are also useful for any situation in which you need large and/or globally accessible data sets on the server. One example of another use for them is as a lookup table for waveshaping.

```
b = Buffer.alloc(s, 512, 1);
b.cheby([1,0,1,1,0,1]);
(
x = play({
Shaper.ar(
b.bufnum,
SinOsc.ar(300, 0, Line.kr(0,1,6)),
0.5
)
});
)
```

```
x.free; b.free;
```

The Shaper UGen performs waveshaping on an input source. The method 'cheby' fills the buffer with a series of chebyshev polynomials, which are needed for this. (Don't worry if you don't understand all this.) Buffer has many similar methods for filling a buffer with different waveforms.

There are numerous other uses to which buffers can be put. You'll encounter them throughout the documentation.

For more information see:

[Buffer] [PlayBuf] [RecordBuf] [SynthDef] [BufRateScale] [Shaper]

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 185

Busses

Now a little bit more about busses on the server. Busses are named after the busses or sends in analog mixing desks, and they serve a similar purpose: Routing signals from one place to another. In SC this means to or from the audio hardware, or between different synths. They come in two types: audio rate and control rate. As you've probably guessed, the former routes audio rate signals and the latter routes control rate signals.

The control rate busses are fairly simple to understand, each one has an index number, starting from 0.

Audio rate busses are similar, but require slightly more explanation. A server app will have a certain number of output and input channels. These correspond to the first audio busses, with outputs coming before inputs.

For example, if we imagine a server with two output channels and two input channels (i.e. stereo in and out) then the first two audio busses (index 0 and index 1) will be the outputs, and the two immediately following those (index 2 and index 3) will be the inputs. Writing audio out to one of the output busses will result in sound being played from your speakers, and reading audio in from the input busses will get sound into SC for things like recording and processing (providing you have a source such as a microphone connected to your computer's or audio interface's inputs).

The remaining audio busses will be 'private'. These are used simply to send audio and control signals between various synths. Sending audio to a private bus will not result in sound in your speakers unless you reroute it later to one of the output busses. These 'private' busses are often used for things like an 'effects send', i.e. something that requires further processing before it's output.

The number of control and audio busses available, as well as the number of input and output channels is set at the time the server app is booted.(See **[ServerOptions]** for information on how to set the number of input and output channels, and busses.)

Writing to or Reading from Busses

We've already seen Out.ar, which allows you to write (i.e. play out) audio to a bus. Recall that it has two arguments, an index, and an output, which can be an array of UGens (i.e. a multichannel output) or a single UGen.

To read in from a bus you use another UGen: In. In's 'ar' method also takes two arguments: An index, and the number of channels to read in. If the number of channels is greater than one, than In's output will be an Array. Execute the following examples, and watch the post window:

```
In          // this will return 'an OutputProxy'
In          // this will return an Array of 4 OutputProxies
```

An OutputProxy is a special kind of UGen that acts as a placeholder for some signal that will be present when the synth is running. You'll probably never need to deal with one directly, so don't worry about them, just understand what they are so that you'll recognise them when you see them in the post window and elsewhere.

In and Out also have 'kr' methods, which will read and write control rate signals to and from control rate busses. Note that Out.kr will convert an audio rate signal to control rate (this is called 'downsampling'), but that the reverse is not true: Out.ar needs an audio rate signal as its second arg.

```
// This throws an error.  Can't write a control rate signal to an audio rate bus
{Out.ar(0, SinOsc.kr)}.play;

// This will work as the audio rate signal is downsampled to control rate
Server.internal.boot;
{Out.kr(0, SinOsc.ar)}.scope;
```

(This limitation is not universal amongst audio rate UGens however, and most will accept control rate signals for some or all of their arguments. Some will even convert control rate inputs to audio rate if needed, filling in the extra values through a process called interpolation.)

You'll note that when multiple Synths write to the same bus, there output is summed, or in other words, mixed.

```
(
SynthDef("tutorial-args", { arg freq = 440, out = 0;
Out.ar(out, SinOsc.ar(freq, 0, 0.2));
}).send(s);
)
// both write to bus 1, and their output is mixed
```

```
x = Synth "tutorial-args"   "out"     "freq"
y = Synth "tutorial-args"   "out"     "freq"
```

Creating a Bus Object

There is a handy client-side object to represent server busses: Bus. Given that all you need is an In or Out Ugen and an index to write to a bus, you might wonder what one would need a full-fledged Bus object for. Well, much of the time you don't, particularly if all you're doing is playing audio in and out. But Bus does provide some useful functionality. We'll get to that in a second, but first lets look at how to make one.

Just as many UGens have ar and kr methods, Bus has two commonly used creation methods: Bus-audio and Bus-control. These each take two arguments: a Server object, and the number of channels.

```
b = Bus                 // Get a two channel control Bus
   Bus              // Get a one channel private audio Bus (one is the default)
```

You may be wondering what a 'two channel' bus is, since we haven't mentioned these before. You should recall that when Out has an Array as its second argument it will write the channels of the Array to consecutive busses. Recall this example from **[SynthDefs and Synths]**:

```
(
 SynthDef     "tutorial-SinOsc-stereo"    var
outArray = [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)];
  Out                  // writes to busses 0 and 1
}).play;
)
```

The truth is that there aren't multichannel busses per se, but Bus objects are able to represent a series of busses with consecutive indices. The *encapsulate* several adjacent sever-side busses into a single Bus object, allowing you to treat them as a group. This turns out to be rather handy.

When you're working with so-called 'private' busses (i.e. anything besides the input and output channels; all control busses are private) you generally want to make sure that that bus is only used for exactly what you want. The point after all is to keep things separate. You could do this by carefully considering which indices to use, but Bus allows for this to be done automatically. Each Server object has a bus allocator, and when you

make a Bus object, it reserves those private indices, and will not give them out again until freed. You can find out the index of a Bus by using its 'index' method.

```
 s.reboot; // this will restart the server app and thus reset the bus allocators

b = Bus                // a 2 channel control Bus
b.index;     // this should be zero
b.numChannels   // Bus also has a numChannels method
c = Bus.control(s);
c.numChannels;  // the default number of channels is 1
c.index;    // note that this is 2; b uses 0 and 1
```

So by using Bus objects to represent adjacent busses, you can guarantee that there won't be a conflict. Since the indices are allocated dyamically, you can change the number of channels of a Bus in your code (for instance because you now need to route a multichannel signal), and you're still guaranteed to be safe. If you were simply 'hard allocating' busses by using index numbers, you might have to adjust them all to make room for an extra adjacent channel, since the indices need to be consecutive! This is a good example of the power of objects: By encapsulating things like index allocation, and providing a *layer of abstraction*, they can make your code more flexible.

You can free up the indices used by a Bus by calling its 'free' method. This allows them to be reallocated.

```
b = Bus.control(s, 2);
b.free; // free the indices.  You can't use this Bus object after that
```

Note that this doesn't actually make the bus on the server go away, it's still there. 'free' just lets the allocator know that you're done using this bus for the moment, and it can freely reallocate its index.

Now here's another advantage when working with private audio rate busses. As we said above, the first few busses are the output and input channels. So if we want to use the first private bus, all we need to do is add those together, right? Consider our server app with 2 output and 2 input channels. The first private audio bus is index 4. (0, 1, 2, 3 ... 4!) So we write our code, and give the appropriate Out UGen 4 as its index arg.

But what happens if we later decide to change the number of output channels to 6? Now everything that was written to our private bus is going out one of the output channels! A Server's audio bus allocator will only assign private indices, so if you change the number

of input or output channels it will take this into account when you execute your code. Again this makes your code more flexible.

Busses in Action

So here are two examples using busses. The first is with a control rate bus.

```
(
SynthDef("tutorial-Infreq", { arg bus, freqOffset = 0;
  // this will add freqOffset to whatever is read in from the bus
Out.ar(0, SinOsc.ar(In.kr(bus) + freqOffset, 0, 0.5));
}).send(s);

SynthDef("tutorial-Outfreq", { arg freq = 400, bus;
Out.kr(bus, SinOsc.kr(1, 0, freq/40, freq));
}).send(s);

b = Bus.control(s,1);
)

(
x = Synth.new("tutorial-Outfreq", [\bus, b.index]);
y = Synth.after(x, "tutorial-Infreq", [\bus, b.index]);
z = Synth.after(x, "tutorial-Infreq", [\bus, b.index, \freqOffset, 200]);
)
x.free; y.free; z.free; b.free;
```

Both y and z read from the same bus, the latter just modifies the frequency control signal by adding a constant value of 200 to it. This is more efficient than having two separate control oscillators to control frequency. This sort of strategy of connecting together synths, each of which does different things in a larger process, can be very effective in SC.

Now an example with an audio bus. This is the most complicated example we've seen so far, but should give you some idea of how to start putting all the things we've learned together. The code below will use two Synths as sources, one creating pulses of PinkNoise (a kind of Noise which has less energy at high frequencies than at low), and another creating pulses of Sine Waves. The pulses are created using the UGens **[Impulse]** and **[Decay2]**. These are then reverberated using a chain of **[AllpassC]**, which is a kind of delay.

Note the construction 16.do({ ... }), below. This makes the chain by evaluating the function 16 times. This is a very powerful and flexible technique, as by simply changing the number, I can change the number of evaluations. See **[Integer]** for more info on Integer-do.

```
(
 // the arg direct will control the proportion of direct to processed signal
SynthDef("tutorial-DecayPink", { arg outBus = 0, effectBus, direct = 0.5;
var source;
  // Decaying pulses of PinkNoise.  We'll add reverb later.
source = Decay2.ar(Impulse.ar(1, 0.25), 0.01, 0.2, PinkNoise.ar);
  // this will be our main output
Out.ar(outBus, source * direct);
  // this will be our effects output
Out.ar(effectBus, source * (1 - direct));
}).send(s);

SynthDef("tutorial-DecaySin", { arg outBus = 0, effectBus, direct = 0.5;
var source;
  // Decaying pulses of a modulating Sine wave.  We'll add reverb later.
source = Decay2.ar(Impulse.ar(0.3, 0.25), 0.3, 1, SinOsc.ar(SinOsc.kr(0.2, 0, 110, 440)));
  // this will be our main output
Out.ar(outBus, source * direct);
  // this will be our effects output
Out.ar(effectBus, source * (1 - direct));
}).send(s);

SynthDef("tutorial-Reverb", { arg outBus = 0, inBus;
var input;
input = In.ar(inBus, 1);

  // a low rent reverb
  // aNumber.do will evaluate it's function argument a corresponding number of times
  // {}.dup(n) will evaluate the function n times, and return an Array of the results
  // The default for n is 2, so this makes a stereo reverb
16.do({ input = AllpassC.ar(input, 0.04, { Rand(0.001,0.04) }.dup, 3)});

Out.ar(outBus, input);
}).send(s);
```

```
 b = Bus              // this will be our effects bus
)

(
x = Synth.new("tutorial-Reverb", [\inBus, b.index]);
y = Synth.before(x, "tutorial-DecayPink", [\effectBus, b.index]);
z = Synth.before(x, "tutorial-DecaySin", [\effectBus, b.index, \outBus, 1]);
)

 // Change the balance of wet to dry
 y.set(\direct      // only direct PinkNoise
 z.set(\direct      // only direct Sine wave
 y.set(\direct      // only reverberated PinkNoise
 z.set(\direct      // only reverberated Sine wave
x.free; y.free; z.free; b.free;
```

Note that we could easily have many more source synths being processed by the single reverb synth. If we'd built the reverb into the source synths we'd be duplicating effort. But by using a private bus, we're able to be more efficient.

More Fun with Control Busses

There are some other powerful things that you can do with control rate busses. For instance, you can map any arg in a running synth to read from a control bus. This means you don't need an In UGen. You can also write constant values to control busses using Bus' 'set' method, and poll values using its 'get' method.

```
(
 // make two control rate busses and set their values to 880 and 884.
b = Bus.control(s, 1); b.set(880);
c = Bus.control(s, 1); c.set(884);
 // and make a synth with two frequency arguments
x = SynthDef("tutorial-map", { arg freq1 = 440, freq2 = 440;
Out.ar(0, SinOsc.ar([freq1, freq2], 0, 0.1));
}).play(s);
)
 // Now map freq1 and freq2 to read from the two busses
x.map(\freq1, b.index, \freq2, c.index);
```

```
 // Now make a Synth to write to the one of the busses
y = {Out.kr(b.index, SinOsc.kr(1, 0, 50, 880))}.play(addAction:  \addToHead);

 // free y, and b holds its last value
y.free;

 // use Bus-get to see what the value is.  Watch the post window
b.get({ arg val; val.postln; f = val; });

 // set the freq2, this 'unmaps' it from c
x.set(\freq2, f / 2);

 // freq2 is no longer mapped, so setting c to a different value has no effect
c.set(200);

x.free; b.free; c.free;
```

Note that unlike audio rate busses, control rate busses hold their last value until something new is written.

Also note that Bus-get takes a Function (called an action function) as an argument. This is because it takes a small amount of time for the server to get the reply and send it back. The function, which is passed the value (or Array of values in the case of a multichannel bus) as an argument, allows you to do something with the value once its come back.

This concept of things taking a small amount of time to respond (usually called *latency*) is quite important to understand. There are a number of other methods in SC which function this way, and it can cause you problems if you're not careful. To illustrate this consider the example below.

```
  // make a Bus object and set its values
b = Bus.control(s, 1); b.set(880);

 // execute this altogether
(
 f = nil  // just to be sure
b.get({ arg val; f = val; });
f.postln;
)
```

```
// f equals nil, but try it again and it's as we expected!
f.postln;
```

So why was f nil the first time but not the second time? The part of the language app which executes your code (called the *interpreter*), does what you tell it, as fast as it can, when you tell it to. So in the block of code between the parentheses above it sends the 'get' message to the server, schedules the Function to execute when a reply is received, and then moves on to posting f. Since it hasn't received the reply yet f is still nil when it's posted the first time.

It only takes a tiny amount of time for the server to send a reply, so by the time we get around to executing the last line of code f has been set to 880, as we expected. In the previous example this wasn't a problem, as we were only executing a line at a time. But there will be cases where you will need to execute things as a block, and the action function technique is very useful for that.

Getting it all in the Right Order

In the examples above, you may have wondered about things like Synth.after, and addAction: \addToHead. During each cycle (the period in which a block of samples is calculated) the server calculates things in a particular order, according to its list of running synths.

It starts with the first synth in its list, and calculates a block of samples for its first UGen. It then takes that and calculates a block of samples for each of its remaining UGens in turn (any of which may take the output of an earlier UGen as an input.) This synth's output is written to a bus or busses, as the case may be. The server then moves on to the next synth in its list, and the process repeats, until all running synths have calculated a block of samples. At this point the server can move on to the next cycle.

The important thing to understand is that *as a general rule*, when you are connecting synths together using busses it is important that synths which write signals to busses are earlier in the server's order than synths which read those signals from those busses. For instance in the audio bus example above it was important that the 'reverb' synth is calculated *after* the noise and sine wave synths that it processes.

This is a complicated topic, and there are some exceptions to this, but you should be aware that ordering is crucial when interconnecting synths. The file **[Order-of-execution]** covers this topic in greater detail.

Synth-new has two arguments which allow you to specify where in the order a synth is added. The first is a *target*, and the second is an *addAction*. The latter specifies the new synth's position in relation to the target.

```
x = Synth("default", [\freq, 300]);
 // add a second synth immediately after x
y = Synth("default", [\freq, 450], x, \addAfter);
x.free; y.free;
```

A target can be another Synth (or some other things; more on that soon), and an addAction is a symbol. See **[Synth]** for a complete list of possible addActions.

Methods like Synth-after are simply convenient ways of doing the same thing, the difference being that they take a target as their first argument.

```
      // These two lines of code are equivalent
y = Synth.new("default", [\freq, 450], x, \addAfter);
y = Synth.after(x, "default", [\freq, 450]);
```

For more information see:

[Bus] [In] [OutputProxy] [Order-of-execution] [Synth]

Suggested Exercise:

Experiment with interconnecting different synths using audio and control busses. When doing so be mindful of their ordering on the server.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Groups]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 186

First Steps

Hello World, I'm SuperCollider

It is traditional when learning a new programming language to start with a simple program called 'Hello World'. This just makes the program print the text 'Hello World!' to well, wherever it prints text. In SC that's a place called the post window. The post window is the one that opened up when you first started SC, and a bunch of stuff was printed there which looks something like this:

```
init_OSC
compiling class library..
NumPrimitives = 587
compiling dir:  '/Applications/SC3/SCClassLibrary'
pass 1 done
Method Table Size 3764776 bytes
Number of Method Selectors 3184
Number of Classes 1814
Number of Symbols 7595
Byte Code Size 180973
compiled 296 files in 1.34 seconds
compile done
RESULT = 256
Class tree inited in 0.14 seconds
```

Don't worry too much about what all that means just now, just keep in mind that this is where SC will send you information. It's also where we'll get the result of our Hello World program, which you can see below:

```
"Hello World!"
```

To execute it, simply click to place the cursor on the same line as the code and then press the enter key. Note that the 'enter' key is not the same as the 'return' key. The 'enter' key is the one that is on the number pad. On Mac laptops there is usually a separate enter key down at bottom of the keyboard towards the right, or you can hold down the 'fn' or function key, and press 'return'. Try this now.

If all went well, you should see this in the post window.

```
Hello World!
Hello World!
```

Now let's take a closer look at the code. The first bit, `"Hello World!"`, is a kind of *Object*, called a String. An object is basically just a way of representing something in the computer, for instance a bit of text, or an oscillator, that allows you to control it and send messages to it. More about that later, but for now just understand that a String is a way of representing a bit of text.

The second bit, `.postln;`, says 'print me (or a meaningful description of me) to the post window.' Remember postln, it's your friend. You can apply it to almost anything in SC and get something meaningful back. This can be very handy when tracking down bugs in your code.

Why did it print twice? Well, when you execute code in SC, it always posts the last thing executed. So in this case we didn't really need the `postln` bit. But in the following example we would. Select both lines of text by clicking and dragging over them, and then press enter.

```
     "Hello World!"
 "Hello SC!"
```

The first line, 'Hello World' would not have printed if we didn't have the explicit postln. Note also that each line of code ends with a semi-colon. This is how you separate lines of code in SC. If we didn't have a semi-colon between the two lines we would get an error.

In general when you are meant to execute several lines of code at the same time they will be surrounded by parentheses, as in the example below. This is convenient as it allows you to select the whole block of code by double clicking just inside one of the parentheses. Try it out on the example below.

```
(
 "Call me,"
 "Ishmael."
)
```

When code is not surrounded by parentheses it is generally intended to be executed one line at a time.

Note that each of the lines within the block of code ends with a semi-colon. This is very important when executing multiple lines of code, as it's how SC knows where to separate commands. Without a semi-colon above, you would get an error posted.

```
(
 "Call me?"
 "Ishmael."
)
```

Executing the code above results in a 'Parse Error'. With an error of this kind, the dot • in the error message shows you where SC ran into trouble. Here it happens just after `"Ishmael.".`

```
• ERROR: Parse error
in file 'selected text'
line 3 char 11 :
"Ishmael."•.postln;
```

Usually the problem actually occurs a little before that, so that's where you should look. In this case of course, it's the lack of a semi-colon at the end of the previous line.

Using semi-colons it's possible to have more than one line of code in the same line of text. This can be handy for execution.

```
  "Call me "       "Ishmael?"
```

A couple of more notes about the post window. It's very useful to be able to see it, but sometimes it can get hidden behind other windows. You can bring it to the front at any time by holding down the Command key, and pressing \. The Command key is the one with the ■ and ■ symbols on it.

By convention this kind of key sequence is written Cmd - /.

As well, sometimes the post window becomes full of stuff and hard to read. You can clear it at any time by pressing Cmd-shift-k (hold down the command key and the shift key, and then press k).

The World According to SuperCollider

SuperCollider is actually two programs: The language or 'client' app, which is what you're looking at now, and the server, which does the actual synthesis and calculation of audio. The former is a graphical application with menus, document windows, nice GUI features and a sophisticated programming language; and the latter is a mean, lean, efficient UNIX command line application (meaning it runs without a nice modern GUI).

The two communicate by a protocol called Open Sound Control (OSC), over either UDP or TCP, which are network protocols also used on the internet. Don't think from this that the two applications must run on different computers (they can, which can have definite performance advantages), or that they need to be connected to the internet (although it is possible to have clients and servers in diffferent parts of the world communicating!!). Most of the time they will be running on the same machine, and the 'networking' aspect of things will be relatively transparent for you.

You can only communicate with the server using OSC messages over the network, but luckily the language app has lots of powerful objects which represent things on the server and allow you to control them easily and elegantly. Understanding how exactly that works is crucial to mastering SC, so we'll be talking about that in some depth.

But first let's have a little fun, and make some sound...

For more information see:

[How-to-Use-the-Interpreter] [Literals] [String] [ClientVsServer] [Server-Architecture]

Suggested Exercise:

Open a new window by pressing Cmd-n or selecting 'New' from the File menu.Copy some of the posting code from the examples above and paste it into the new document. (The standard Mac Cmd-c and Cmd-v work for copy and paste, or use the Edit menu.)

SC will let you edit the help files and documentation, so it's always a good idea to copy text over before changing it so as to avoid accidentally saving altered files!

Experiment with altering the text between the quotes to print different things to the post window. Do this with both blocks of text wrapped in parentheses, and single lines.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Start Your Engines]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 187

Functions and Other Functionality

The easiest way to get sound from SC is to use a Function. Below is a simple example of this. Execute this (after making sure the server is booted), and when you're sick of it, press Cmd - . (that's hold down the command key and press the period or fullstop key) to stop the sound. This will always stop all sound in SC. You'll be using it a lot, so commit it to memory.

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play;
```

Not too inspiring? Don't worry, we're just getting started, and this is just a simple example to demonstrate Functions and sound. We'll take it apart a bit below.

Before we get to doing that though, let's learn a little about Functions in general.

A Function is just a reusable bit of code. You define a Function by enclosing code in 'curly brackets': { }. Here's an example:

```
"Function evaluated"
```

The stuff within the curly brackets is what will get executed each time you reuse, or evaluate the Function. Note that this is written like an equation, i.e. `f = {...}`. This is not an equation in the mathematical sense, it's what's called an assignment. Basically it allows me to name the Function I've created, by storing it in a variable called '`f`'. A variable is just a name representing a slot in which we can store things, such as a Function, a number, a list, etc. Execute the following lines one at a time and watch the post window:

```
"Function evaluated"
f;
```

Both times it should say '`a Function`'. Now whenever we want to refer to our Function we can just use the letter f. That's in fact what makes it reusable! Otherwise we'd need to type the Function in every time.

So how do we reuse it? Execute the following lines one at a time and watch the post window:

```
  f = { "Function evaluated"
f.value;
f.value;
f.value;
```

Our Function is an object, (i.e a thing that does something or represents something), which we have defined and stored in the variable '`f`'. The bit of code that says '`.value`' says evaluate this function now. This is an example of sending a message to an object. This follows the syntax someObject.someMessage. The dot must go in between.

Now this next bit is a little bit tricky. In a given object, each *message* calls (calls means executes) a particular *method*. Different types of objects may have methods with the same name, and thus respond to the same message in different ways. Whoah, get that? Read it again slowly, as this is pretty important:

Different types of objects may have methods with the same name, and thus respond to the same message in different ways.

What's interesting about this is that the actual methods may differ in what they do, but as long as they implement a method with that name, they become interchangeable in your code.

A good example is 'value'. All objects in SC respond to the message 'value'. When you 'call' a method, it always 'returns' something, such as a value or a result. When you call the method 'value' on a Function it will evaluate and return the result of its last line of code. The example below will return the number 5.

```
f = { "Evaluating...".postln; 2 + 3; };
f.value;
```

Often methods simply return the object itself. This is the case with most objects and the message 'value'. The example below demonstrates this. (Everything to the right of the `//` is a 'comment', which means that SC just ignores it. Comments are a good idea to make your code clearer.)

```
              // Here I make f equal to a number
f.value;   // Post window says:  3, i.e it returns itself
f.value;   // Still says 3

f = { 3.0.rand; }; // Here it's a Function.
```

```
f.value;   // 3.0.rand means return a random value from 0 to 3.0 exclusive.
f.value;   // something different
f.value;   // something different
```

This means that by using the 'value' method Functions and other objects can be interchangeable in your code. This is an example of *polymorphism*, which is one of the powerful features of what's called Object Oriented Programming. Polymorphism just means that different objects are interchangeable (at least providing they return something sensible for what you're doing) if they respond to the same message. Object Oriented Programming (or OOP, as it's called for short) just means programming with objects. Simple, yes? Here's another short example showing this in action:

```
f = { arg                  // call 'value' on the arg; polymorphism awaits!
f.value(3);    // 3.value = 3, so this returns 3 + 3 = 6
g = { 3.0.rand; };
f.value(g);    // here the arg is a Function.  Cool, huh?
f.value(g);    // try it again, different result
```

Start to see how this could be useful?

Functions can also have what are called arguments. These are values which are passed into the Function when it is evaluated. The example below demonstrates how this works. See if you can guess what the result will be before executing it.

```
(
f = { arg a, b;
a - b;
};
f.value(5, 3);
)
```

Arguments are declared at the beginning of the Function, using the keyword '`arg`'. You can then refer to them just like variables. When you call value on a Function, you can pass in arguments, in order, by putting them in parentheses: `someFunc.value(arg1, arg2)`. This is the same with any method that takes arguments, not just value.

You can specify different orders by using what are called keyword arguments:

```
f = { arg a, b; a / b; };  // '/' means divide
f.value(10, 2);    // regular style
```

```
f.value(b:  2, a:  10);   // keyword style
```

You can mix regular and keyword style if you like, but the regular args must come first:

```
f = { arg a, b, c, d; (a + b) * c - d };
f.value(2, c:3, b:4, d:  1); // 2 + 4 * 3 - 1
```

(Note that SC has no operator precedence, i.e. math operations are done in order, and division and multiplication are not done first. To force an order use parentheses. e.g. 4 + (2* 8))

Sometimes it's useful to set default values for arguments. You can do this like so:

```
f = { arg a, b = 2; a + b; };
f.value(2);      // 2 + 2
```

Default values must be what are called literals. Literals are basically numbers, strings, symbols (more on these later), or collections of them. Don't worry if that doesn't totally make sense, it will become clearer as we go on.

There is an alternate way to specify args, which is to enclose them within two vertical lines. (On most keyboards the vertical line symbol is Shift-\) The following two Functions are equivalent:

```
f = { arg a, b; a + b; };
g = { | a, b|  a + b; };
f.value(2, 2);
g.value(2, 2);
```

Why have two different ways? Well some people like the second one better and consider it a shortcut. SC has a number of syntax shortcuts like this, which can make writing code a little faster. In any case you will encounter both forms, so you need to be aware of them.

You can also have variables in a Function. These you need to declare at the beginning of the Function, just after the args, using the keyword '`var`'.

```
(
f = { arg a, b;
var firstResult, finalResult;
```

```
firstResult = a + b;
finalResult = firstResult * 2;
finalResult;
};
 f.value(2, 3);  // this will return (2 + 3) * 2 = 10
)
```

Variable and argument names can consist of letters and numbers, but must begin with a lower-case letter and cannot contain spaces.

Variables are only valid for what is called their scope. The scope of a variable declared in a Function is that Function, i.e. the area between the two curly brackets. Execute these one at a time:

```
f = { var foo; foo = 3; foo; };
f.value;
 foo;      // this will cause an error as 'foo' is only valid within f.
```

You can also declare variables at the top of any block of code which you execute altogether (i.e. by selecting it all). In such a case that block of code is the variable's scope. Execute the block (in parentheses) and then the last line.

```
(
var myFunc;
myFunc = { | input|  input.postln; };
myFunc.value("foo");  // arg is a String
myFunc.value("bar");
)

 myFunc;    // throws an error
```

You may be wondering why we haven't needed to declare variables like '`f`', and why they don't seem to have any particular scope (i.e. they keep there values even when executing code one line at a time). The letters a to z are what are called interpreter variables. These are pre-declared when you start up SC, and have an unlimited, or 'global', scope. This makes them useful for quick tests or examples. You've already encountered one of these, the variable '`s`', which you'll recall by default refers to the localhost server.

For more information see:

[Functions] [Function] [Assignment] [Intro-to-Objects] [Literals] [Scope]

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Functions and Sound]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 188

And What About Functions and Sound?

I've probably bored you enough with technical details, so let's get back to making noise, which I assume is why you're reading this after all. Trust me though, all this work will pay off later, and believe it or not, we've already covered a fair amount of the basics of the language, at least in passing.

Let's go back to our sound example, or rather a slightly simplified version of it. Check that the localhost server is running, execute the code below and then press Cmd-. when you've had enough.

```
{ SinOsc.ar(440, 0, 0.2) }.play;
```

In this case we've created a Function by enclosing some code in curly brackets, and then called the method 'play' on that Function. To Functions 'play' means evaluate yourself and play the result on a server. If you don't specify a server, you'll get the default one, which you'll recall is stored in the variable 's' and is set at startup to be the localhost server.

We didn't store the Function in a variable, so it can't be reused. (Well, actually you could just execute the same line of code again, but you know what I mean...) This is often the case when using Function-play, as it is useful as a quick way of getting something to make noise, and is often used for testing purposes. There are other ways of reusing Functions for sounds, which are often better and more efficient as we will see.

Lets look at what's between the curly brackets. We're taking something called a 'SinOsc' and we're sending it the message ar, with a few arguments. It turns out that SinOsc is an example of something called a *class*. To understand what a class is, we need to know a little more about OOP and objects.

In a nutshell, an object is some data, i.e. some information, and a set of operations that you can perform on that data. You might have many different objects of the same type. These are called instances. The type itself is the object's class. For instance we might have a class called Student, and several instances of it, Bob, Dave and Sue. All three will have the same types of data, for instance they might have a bit of data named gpa. The value of each bit of data could be different however. They would also have the same methods to operate on the data. For instance they could have a method called calculateGPA, or something similar.

An object's class defines its set of data (or *instance variables* as they are called) and methods. In addition it may define some other methods which only you send only to the class itself, and some data to be used by all of its instances. These are called class methods and class variables.

All classes begin with upper-case letters, so it's pretty easy to identify them in code.

Classes are what you use to make objects. They're like a template. You do this through class methods such as 'new', or, in the case of our SinOsc class above, 'ar'. Such methods return an object, an instance, and the arguments affect what its data will be, and how it will behave. Now take another look at the example in question:

```
SinOsc.ar(440, 0, 0.2)
```

This tells the class SinOsc to make an instance of itself. All SinOscs are an example of what are called unit generators, or UGens. These are objects which produce audio or control signals. SinOsc is a sine wave oscillator. This means that it will produce a signal consisting of a single frequency. A graph of it's waveform would look like this:

<img src="Pasted%20Graphic.tiff" alt="Pasted Graphic.tiff"> (don't worry about the 'index' and 'value' stuff; it's not important just now)

This waveform loops, creating the output signal. 'ar' means make the instance *audio rate*. SuperCollider calculates audio in groups of samples, called *blocks*. Audio rate means that the UGen will calculate a value for each sample in the block. There's another method, 'kr', which means *control rate*. This means calculate a single value for each block of samples. This can save a lot of computing power, and is fine for (you guessed it) signals which control other UGens, but it's not fine enough detail for synthesizing audio signals.

The three arguments to SinOsc-ar given in the example determine a few things about the resulting instance. I happen to know that the arguments are frequency, phase, and mul. (We'll get to how I know that in a second.) Frequency is just the frequency of the oscillator in Hertz (Hz), or cycles per second (cps). Phase refers to where it will start in the cycle of its waveform. For SinOsc (but not for all UGens) phase is given in radians. If you don't know what radians are, don't worry, just understand that it's a value between 0 and 2 * pi. (You can look at a trigonometry text if you really want

more detail.) So if we made a SinOsc with a phase of (pi * 0.5), or one quarter of the way through its cycle, the waveform would look like this:

<img src="Pasted%20Graphic%201.tiff" alt="Pasted Graphic 1.tiff">

Make sense? Here are several cycles of the two side by side to make the idea clearer:

<img src="Pasted%20Graphic%202.tiff" alt="Pasted Graphic 2.tiff">

So what about 'mul'? Mul is a special argument that almost all UGens have. It's so ubiquitous that it's usually not even explained in the documentation. It just means a value or signal by which the output of the UGen will be multiplied. It turns out that in the case of audio signals, this affects the amplitude of the signal, or how loud it is. The default mul of most UGens is 1, which means that the signal will oscillate between 1 and -1. This is a good default as anything bigger would cause clipping and distortion. A mul of 0 would be effectively silent, as if the volume knob was turned all the way down.

To make clearer how mul works, here is a graph of two SinOscs, one with the default mul of 1, and one with a mul of 0.25:

<img src="Pasted%20Graphic%203.tiff" alt="Pasted Graphic 3.tiff">

Get the idea? There's also another similar arg called 'add' (also generally unexplained in the doc), which (you guessed it) is something which is added to the output signal. This can be quite useful for things like control signals. 'add' has a default value of 0, which is why we don't need to specify something for it.

Okay, with all this in mind, let's review our example, with comments:

```
(
{     // Open the Function
 SinOsc     // Make an audio rate SinOsc
  440,  // frequency of 440 Hz, or the tuning A
  0,   // initial phase of 0, or the beginning of the cycle
```

```
   0.2)  // mul of 0.2
 }.play;   // close the Function and call 'play' on it
)
```

Some More Fun with Functions and UGens

Here's another example of polymorphism, and how powerful it is. When creating Functions of UGens, for many arguments you don't have to use fixed values, you can in fact use other UGens! Below is an example which demonstrates this:

```
(
{ var ampOsc;
ampOsc = SinOsc.kr(0.5, 1.5pi, 0.5, 0.5);
SinOsc.ar(440, 0, ampOsc);
}.play;
)
```

Try this. (Again, use Cmd-. to stop the sound.)

What we've done here is plugged the first SinOsc (a *control rate* one!) into the mul arg of the second one. So its output is being multiplied by the output of the second one. Now lets look at the first SinOsc's arguments.

Frequency is set to 0.5 cps, which if you think about it a bit means that it will complete one cycle every 2 seconds. (1 / 0.5 = 2)

Mul and add are both set to 0.5. Think for a second about what that will do. If by default SinOsc goes between 1 and -1, then a mul of 0.5 will scale that down to between 0.5 and -0.5. Adding 0.5 to that brings it to between 0 and 1, a rather good range for mul!

The phase of 1.5pi (this just means 1.5 * pi) means 3/4 of the way through its cycle, which if you look at the first graph above you'll see is the lowest point, or in this case, 0. So the ampOsc SinOsc's waveform will look like this:

<img src="Pasted%20Graphic%204.tiff" alt="Pasted Graphic 4.tiff">

And what we have in the end is a SinOsc that fades gently in and out. Shifting the

phase just means that we start quiet and fade in. We're effectively using ampOsc as what is called an amplitude *envelope*. There are other ways of doing the same thing, some of them simpler, but this demonstrates the principal.

Patching together UGens in this way is the basic way that you make sound in SC. For an overview of the various types of UGens available in SC, see **[UGens]** or **[Tour_of_UGens]**.

For more information see:

[Functions] [Function] [UGens] [Tour_of_UGens]

Suggested Exercise:

Experiment with altering the Functions in the text above. For instance try changing the frequencies of the SinOsc, or making multi-channel versions of things.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Presented in Living Stereo]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 189

Getting Help

This is probably a good point to stop and explore some methods of finding further information. You're already familiar with the clickable links that have been used so far in this tutorial. Here's an example:

[Help]

Clicking on this link will open the main help window, which contains a number of links to other help files. At some point, it would be a good idea to familiarise yourself with some of these. The ones under the headings 'Essential Topics' and 'Language' are of particular import. Again don't worry if everything doesn't immediately make complete sense to you. Learning a computer language is sometimes a little like slowly zeroing in on something, rather than getting it immediately, and some information you can just file away for future reference.

Classes and Methods

By now we've learned enough OOP theory that we know that we have classes, which are like templates for objects, and instances, which are objects which have been created from those templates. We also have class and instance methods, which may take arguments. Class methods do things like create instances (as well as some convenience functions that don't require an actual instance), and instance methods control and manipulate instances. There are also instance variables, which are the data specific to each instance, and class variables, which are data in common between all instances.

Recall that anything in the code that begins with an uppercase letter is a class. Most classes have help files. If you select a class by double-clicking on it, and press Cmd - ? (that's hold down the Cmd key, hold down shift and press ?) the help file for that class will open if it exists. (If not you'll get the main help window.) Try it with this example below:

```
SinOsc
```

You should have gotten a window with a brief description of the class and what it does, a list of some methods, and a description of their arguments. (Remember that 'mul' and 'add' are usually not explained.)

Beneath that are some examples of the class in action. These can be very useful for making it clear exactly what the class does, and can serve as starting points for your own work. It's a good idea to cut and paste these to a new window, and then play around with modifying them. (Remember that SC won't stop you from saving any modified files, including this tutorial!) This is a great way to learn.

You may be wondering how to access the helpfiles for Function and Array, since they often appear in code as {...} and [...]. They are also named classes, so by typing in the following, you can also select and Cmd-? on them.

```
     Function
Array
```

Some methods also have helpfiles, and there are a number of ones on general topics. Most of these are listed in the main help window.

Syntax Shortcuts

Remember the example of Mix(...) vs. Mix.new(...)? SC has a number of such shorthand forms or alternate syntaxes. A common example is the distinction between Functional and receiver notation. This means that the notation `someObject.someMethod(anArg)` is equivalent to `someMethod(someObject, anArg)`. Here's a concrete example. Both of these do exactly the same thing:

```
{ SinOsc.ar(440, 0, 0.2) }.play;

play({ SinOsc.ar(440, 0, 0.2) });
```

You will find numerous other examples of syntax shortcuts throughout SC's documentation. If you see something you don't recognize, a good place to check is **[Syntax-Shortcuts]**, which gives examples of most of these.

Snooping, etc.

SC has numerous other ways of tracking down information on classes, methods, etc. Most of these won't be too helpful for you at this point, but are good to know about for future use. Information on these can be found in the files **[More-On-Getting-Help]** and **[Internal-Snooping]**.

For more information see:

[More-On-Getting-Help] [Internal-Snooping] [Syntax-Shortcuts]

Suggested Exercise:

Go back over the examples in the previous tutorials, and try opening up the helpfiles for the various classes used. Try out the examples, and if you like open up the help files for any unfamiliar classes used in those examples. Get used to Cmd-?, you'll be using it a lot. :-)

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[SynthDefs and Synths]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 190

Getting Started With SuperCollider

by Scott Wilson

Table of Contents

In OSX click on the links below to take you to the corresponding document.

ID: 191

Groups

Our discussion about the order of synths on the server brings us to the topic of groups. Synths on the server are a type of what are called *nodes*. There's one other type of node: groups. Groups are simply collections of nodes, and can contain synths, other groups, or combinations of both. They are mostly useful in two ways: First they are very helpful in controlling order, second, they allow you to easily group together nodes and send them messages all at once. As you've probably guessed, there's a handy Server abstraction object to represent group nodes in the client app: Group.

Groups as Ordering Tools

Groups can be quite helpful in terms of controlling order. Like synths they take targets and addActions as arguments, which makes it easy to put them in position.

```
g = Group.new;
h = Group.before(g);
g.free; h.free;
```

This can be very helpful for things like keeping effects or processing separate from sound sources, and in the right order. Let's reconsider our reverb example from the previous section.

```
(
 // a stereo version
SynthDef("tutorial-DecaySin2", { arg outBus = 0, effectBus, direct = 0.5, freq = 440;
var source;
  // 1.0.rand2 returns a random number from -1 to 1, used here for a random pan
source = Pan2.ar(Decay2.ar(Impulse.ar(Rand(0.3, 1), 0, 0.125), 0.3, 1,
SinOsc.ar(SinOsc.kr(0.2, 0, 110, freq))), Rand(-1.0, 1.0));
Out.ar(outBus, source * direct);
Out.ar(effectBus, source * (1 - direct));
}).send(s);

SynthDef("tutorial-Reverb2", { arg outBus = 0, inBus;
var input;
input = In.ar(inBus, 2);
16.do({ input = AllpassC.ar(input, 0.04, Rand(0.001,0.04), 3)});
```

```
Out.ar(outBus, input);
}).send(s);
)

 // now we create groups for effects and synths
(
sources = Group.new;
effects = Group.after( sources);  // make sure it's after
     Bus                  // this will be our stereo effects bus
)

 // now synths in the groups.  The default addAction is \addToHead
(
x = Synth("tutorial-Reverb2", [\inBus, b.index],  effects);
y = Synth("tutorial-DecaySin2", [\effectBus, b.index, \outBus, 0],  sources);
z = Synth("tutorial-DecaySin2", [\effectBus, b.index, \outBus, 0, \freq, 660],  sources);
)

 // we could add other source and effects synths here

  sources.free;  effects.free; // this frees their contents (x, y, z) as well
 b.free;
```

Note that we probably don't care what order the sources and effects are within the groups, all that matters is that all effects synths come after the source synths that they process.

If you're wondering about the names ' sources' and ' effects', placing a tilde ( ) in front of a word is a way of creating an *environment variable*. For the moment, all you need to know about them is that they can be used in the same way as interpreter variables (you don't need to declare them, and they are persistent), and they allow for more descriptive names.

All the addActions

At this point it's probably good to cover the remaining add actions. In addition to \addBefore and \addAfter, there is also the (rarely) used \addReplace, and two add actions which apply to Groups: \addToHead and \addToTail. The former adds the receiver to the beginning of the group, so that it will execute first, the latter to the end of the group, so that it will execute last. Like the other addActions, \addToHead and

\addToTail have convenience methods called 'head' and 'tail'.

```
g = Group.new;
 h = Group              // add h to the head of g
 x = Synth          "default"   // add x to the tail of h
 s.queryAllNodes;    // this will post a representation of the node hierarchy
x.free; h.free; g.free;
```

'queryAllNodes' and node IDs

Server has a method called 'queryAllNodes' which will post a representation of the server's node tree. You should have seen something like the following in the post window when executing the example above:

```
nodes on localhost:
a Server
Group(0)
Group(1)
Group(1000)
Group(1001)
Synth 1002
```

When you see a Group printed here, anything below it and indented to the right is contained within it. The order of nodes is from top to bottom. The numbers you see are what are called node IDs, which are how the server keeps track of nodes. Normally when working with Server abstraction objects you won't need to deal with node IDs as the objects keep track of them, assigning and freeing them when appropriate.

You may have been wondering why there were four groups posted above when we only created two. The first two, with the IDs 0 and 1, are special groups, called the RootNode and the 'default group'.

The Root Node and the Default Group

When a server app is booted there is a special group created with a node ID of 0. This represents the top of the server's node tree. There is a special server abstraction object to represent this, called RootNode. In addition there is another group created with an ID of 1, called the default group. This is the default target for all Nodes and is what you will get if you supply a Server as a target. If you don't specify a target or pass in nil, you will get the default group of the default Server.

```
Server.default.boot;
a = Synth      \default    // creates a synth in the default group of the default Server
a.group; // Returns a Group object.  Note the ID of 1 (the default group) in the post window
```

The default group serves an important purpose: It provides a predictable basic Node tree so that methods such as Server-scope and Server-record (which create nodes which *must* come after everything else) can function without running into order of execution problems. In the example below the scoping node will come after the default group.

```
Server.internal.boot;

{ SinOsc.ar(mul:  0.2) }.scope(1);

// watch the post window;
Server.internal.queryAllNodes;

// our SinOsc synth is within the default group (ID 1)
// the scope node ('stethoscope') comes after the default group, so no problems

Server.internal.quit;
```

In general you should add nodes to the default group, or groups contained within it, and *not* before or after it. When adding an 'effects' synth, for instance, one should resist the temptation to add it after the default group, and instead create a separate source group within the default group. This will prevent problems with scoping or recording.

default group [
source group [
source synth1
source synth2
]
effects synth
]
recording synth

Groups as, well, groups...

The other major use of groups is to allow you to easily treat a number of synths as a whole. If you send a 'set' message to a group, it will apply that message to all nodes

contained within it.

```
g = Group.new;

 // make 4 synths in g
 // 1.0.rand2 returns a random number from -1 to 1.
4.do({ { arg amp = 0.1; Pan2.ar(SinOsc.ar(440 + 110.rand, 0, amp), 1.0.rand2) }.play(g); });

 g.set("amp"          // turn them all down

g.free;
```

Groups, their Inheritance, and More on Tracking Down Help

Now for a little more OOP theory. Both Group and Synth are examples of what are called *subclasses*. You can think of subclasses as being children of a parent class, called their *superclass*. All subclasses inherit the methods of their superclass. They may *override* some methods with their own implementation (taking advantage of *polymorphism*), but in general subclasses respond to all the methods of their superclass, and some other ones of their own. Some classes are *abstract classes*, which means that you don't actually make instances of them, they just exist to provide a common set of methods and variables to their subclasses.

We might for instance imagine an abstract class called Dog, which has a number of subclasses, such as Terrier, BassetHound, etc. These might all have a 'run' method, but not all would need a 'herdSheep' method.

This way of working has certain advantages: If you need to change an inherited method, you can do so in one place, and all the subclasses which inherit it will be changed too. As well, if you want to extend a class to make your own personal variant or enhanced version, you can automatically get all the functionality of the superclass.

Inheritance can go back through many levels, which is to say that a class' superclass may also have a superclass. (A class cannot, however have more than one immediate superclass.) All objects in SC in fact inherit from a class called Object, which defines a certain set of methods which all its subclasses either inherit or override.

Group and Synth are subclasses of the abstract class **[Node]**. Because of this, some of their methods are defined in Node, and (perhaps more practically important) *are documented in Node's helpfile*.

So if you're looking at a helpfile and can't find a particular method that a class responds to, you may need to go to the helpfile for that class' superclass, or farther up the chain. Most classes have their superclass listed at the top of their helpfile. You can also use the following methods for getting this kind of info and tracking down documentation (watch the post window):

```
  Group                    // this will return 'Node'
Group.superclass.openHelpFile;
Group.findRespondingMethodFor('set');  // Node-set
Group.findRespondingMethodFor('postln'); // Object-postln;
 Group                   'postln'     // opens class Object help file
```

For more information see:

[Group] [Node] [default_group] [RootNode] [Intro-to-Objects] [Order-of-execution] [Synth] [More-On-Getting-Help] [Internal-Snooping]

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Buffers]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 192

Introductory Remarks

The following text is intended to serve as an introduction to SuperCollider 3, an object-oriented language for sound synthesis and digital signal processing (DSP). This tutorial does not assume a background in computer science, but does assume basic familiarity with your computer and its OS, as well as a basic knowledge of acoustics and digital audio. (I'm assuming here that words like frequency and sample will not cause any confusion.)

The tutorial is written from a Mac OSX perspective, but much of it should apply to linux and windows as well. The parts which specifically differ have mostly to do with GUI aspects (Graphical User Interface).

I should acknowledge that this tutorial is 'by' me in only a limited sense. In writing it I have drawn freely upon the general documentation, which was written by a number of people. This document is not intended to replace those (often more detailed) sources, and refers the reader to them constantly for further information.

A full list of those who have contributed to SuperCollider and its documentation can be seen at:

#1104ffhttp://supercollider.sourceforge.net

Links

Within the text, and at the end of each section there might be a list links to other documents, that will look something like this:

See also:**[Some other document]**

Most of these are meant to expand upon what you have just read, but some just point you in the direction of further information which you will probably need in the future. Some of the linked documents are written in fairly technical language, and may duplicate information which is presented in this tutorial in a more casual form. Often they are designed as reference documents for people already familiar with SC, so don't worry if everything in them doesn't immediately make sense. You won't need to have seen and/or fully understood them in order to continue with the tutorial.

Code Examples

Code examples within the text are in a different font:

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play;
```

This is a common convention in documentation of computer languages, and one that is followed throughout SC's doc. The different colours you'll see in code are just to make things clearer, and have no effect on what the code does.

You are encouraged to copy the code examples to another window and play around with modifying them. This is a time honoured way of learning a new computer language! SC will allow you to mofify the original tutorial documents, but if you do so you should be careful not to save them (for instance if prompted when closing them). It's safest to copy things to a new document before changing them.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[First Steps]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 193

Mix it Up

We've already seen that multiplication changes the level of something, but what about mixing UGens together? This turns out to be equally simple. All we need is addition:

```
{ PinkNoise.ar(0.2) + SinOsc.ar(440, 0, 0.2) + Saw.ar(660, 0.2) }.play;
```

Saw is another type of oscillator, with a waveform that looks like a sawtooth. Note that we use a low value for mul, thus ensuring that the final output will be between -1 and 1, and not clip.

There's another handy class called Mix, which will mix an array of channels down to a single channel or an array of arrays of channels down to a single array of channels. Watch the post window to see Mix's results.

```
  // one channel
{ Mix.new([SinOsc.ar(440, 0, 0.2), Saw.ar(660, 0.2)]).postln }.play;

 // combine two stereo arrays
(
{
var a, b;
a = [SinOsc.ar(440, 0, 0.2), Saw.ar(662, 0.2)];
b = [SinOsc.ar(442, 0, 0.2), Saw.ar(660, 0.2)];
Mix([a, b]).postln;
}.play;
)
```

In the first case we get a 'BinaryOpUGen' (in this case this means the two UGens added together), and in the second we get an Array of two BinaryOpUGens.

Note that in the first example we use Mix.new(...), but in the second we use Mix(...). The latter is a shorthand for the former. 'new' is the most common class method for creating a new object. In some cases objects have more than one class method for creating objects, such as the 'ar' and 'kr' methods of UGens. (Mix, however, is actually just a 'convenience' class, and doesn't actually create Mix objects, it just returns the results of its summing, either a BinaryOpUGen or an Array of them.)

Mix also has another class method called fill, which takes two arguments. The first is a number, which determines how many times the second argument, a Function, will be evaluated. The results of the evaluations will be summed. Confusing? Take a look at the following example:

```
(
var n = 8;
{ Mix.fill(n, { SinOsc.ar(500 + 500.0.rand, 0, 1 / n) })  }.play;
)
```

The Function will be evaluated n times, each time creating a SinOsc with a random frequency from 500 to 1000 Hz (500 plus a random number between 0 and 500). The mul arg of each SinOsc is set to 1 / n, thus ensuring that the total amplitude will not go outside -1 and 1. By simply changing the value of n, you can have vastly different numbers of SinOscs! (Try it!) This sort of approach makes this code extremely flexible and reusable.

Each time the Function is evaluated it is passed the number of times evaluated so far as an argument. So if 'n' is 8 the Function will be passed values from 0 to 7, in sequence, counting up. By declaring an argument within our Function we can use this value.

```
  // Look at the post window for frequencies and indices
(
var n = 8;
{
Mix.fill(n, { arg index;
var freq;
index.postln;
freq = 440 + index;
freq.postln;
SinOsc.ar(freq , 0, 1 / n)
})
}.play;
)
```

By combining addition and multiplication (or indeed almost any mathematical procedure you could imagine!) with the use of classes like Mix, we have the tools we need to combine multichannel sources of sound into complex mixes and submixes.

For more information see:

[Mix] [BinaryOpUGen] [Operators] [Syntax-Shortcuts]

Suggested Exercise:

Experiment with altering the Functions in the text above. For instance try changing the frequencies of the SinOsc, or making multi-channel versions of things.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Scoping and Plotting]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 194

Presented in Living Stereo

Okay, but what about our first, unsimplified example? Remember:

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play;
```

This also has two SinOscs, but in a different arrangement, between two square brackets `[]`, and with a comma in between. Just like the curly brackets indicate a Function, square brackets define something called an Array. An Array is a type of Collection, which is (you guessed it) a collection of Objects. Collections themselves are Objects, and most types of Collections can hold any types of objects, mixed together, including other Collections! There are many different types of Collections in SC, and you will come to learn that they are one of the SC's most powerful features.

An Array is a particular type of Collection: An ordered collection of limited maximum size. You can make one as we have above, by putting objects in between two square brackets, with commas in between. You can get the different elements of an Array using the method 'at', which takes an index as an argument. Indices correspond to the order of objects in the Array, and start from 0.

```
          "foo"  "bar"   // "foo" is at index 0; "bar" is at index 1
a.at(0);
a.at(1);
 a.at(2);    // returns "nil", as there is no object at index 2

 // there's a shorthand for at that you'll see sometimes:
          // same as a.at(0);
```

In addition to being used to hold collections of objects, Arrays also have a special use in SC: They are used to implement multichannel audio! If your Function returns an Array of UGens (remember that Functions return the result of their last line of code) then the output will be a number of channels. How many depends on the size of the Array, and each channel will correspond to an element of the Array. So in our example:

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play;
```

What we end up with is stereo output, with a SinOsc at 440Hz in the left channel, and a SinOsc at 442Hz in the right channel. We could have even more channels of output

by having a larger array.

Now watch carefully, because this next bit involves a little slight of hand, but shows another way in which SC makes things very interchangeable. Because the arguments for phase and mul are the same for both SinOscs, we can rewrite the code for our example like this:

```
{ SinOsc.ar([440, 442], 0, 0.2) }.play;
```

We've replaced the frequency argument with an Array. This causes something called 'multichannel expansion', which means that if you plug an Array into one of a UGen's arguments, you get an Array of that UGen instead of a single one. Now consider this:

```
(
{ var freq;
freq = [[660, 880], [440, 660], 1320, 880].choose;
SinOsc.ar(freq, 0, 0.2);
}.play;
)
```

Try executing it several times, and you'll get different results. 'choose' is just a method which randomly selects one of the elements of the Array. In this case the result may be a single number or another Array. In the case of the latter you'll get stereo output, in the case of the former, monophonic. This sort of thing can make your code very flexible.

But what if you want to 'pan' something, crossfading it between channels? SC has a number of UGens which do this in various ways, but for now I'll just introduce you to one: Pan2. Pan2 takes an input and a position as arguments and returns an Array of two elements, the left and right or first and second channels. The position arg goes between -1 (left) and 1 (right). Take a look at this example:

```
{ Pan2.ar(PinkNoise.ar(0.2), SinOsc.kr(0.5)) }.play;
```

This uses a SinOsc to control the position (remember it outputs values from -1 to 1, or left to right), but uses a different UGen as the input to the Pan2, something called PinkNoise. This is just a kind of noise generator, and it has a single argument: mul. You can of course also used fixed values for the position arg.

```
{ Pan2.ar(PinkNoise.ar(0.2), -0.3) }.play; // slightly to the left
```

For more information see:

[MultiChannel] [Collections] [Pan2]

Suggested Exercise:

Experiment with altering the Functions in the text above. For instance try changing the frequencies of the SinOsc, or making multi-channel versions of things.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Mix it Up]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 195

Scoping Out Some Plots

Function has two other useful audio related methods. The first you've already seen some results of, Function-plot:

```
{ PinkNoise.ar(0.2) + SinOsc.ar(440, 0, 0.2) + Saw.ar(660, 0.2) }.plot;
```

This makes a graph of the signal produced by the output of the Function. You can specify some arguments, such as the duration. The default is 0.01 seconds, but you can set it to anything you want.

```
{ PinkNoise.ar(0.2) + SinOsc.ar(440, 0, 0.2) + Saw.ar(660, 0.2) }.plot(1);
```

This can be useful to check what's happening, and if you're getting the output you think you're getting.

The second method, Function-scope, shows an oscilloscope-like display of the Function's output. This only works with what is called the internal server, so you'll need to boot that before it will work. You can do this using the internal server window

<img src="Pasted%20Graphic.tiff" alt="Pasted Graphic.tiff">

or you can do it in code, like so:

```
Server.internal.boot;
```

BTW, clicking on the '-> default' button on the localhost or internal server's window sets that server to be the default, and stores it in the variable 's'. Thereafter, that will be the server on which all audio is played, unless you specify another one. Since Function-scope only works with the internal server, however, it will always play on it.

So let's try to scope some audio:

```
{ PinkNoise.ar(0.2) + SinOsc.ar(440, 0, 0.2) + Saw.ar(660, 0.2) }.scope;
```

This should open a window which looks something like this:

<img src="Pasted%20Graphic%201.tiff" alt="Pasted Graphic 1.tiff">

This also works for multiple channels:

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.scope;
```

Scope also has a zoom argument. Higher values 'zoom out'.

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.scope(zoom:  10);
```

Like Function-plot, Function-scope can be useful for testing purposes, and to see if you're actually getting out what you think you are.

Scoping on Demand

You can also scope the output of the internal server at any time, by calling 'scope' on it.

```
{ [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)] }.play(Server.internal);
 Server                // you could also use 's' if the internal is the default
```

You can do the same thing by clicking on the internal server window and pressing the 's' key.

Local vs. Internal

If you're wondering what's the difference between the local and the internal servers, it's relatively straightforward: The internal server runs as a process within the client app; basically a program within a program. The main advantage of this is that it allows the two applications to share memory, which allows for things like realtime scoping of audio. The disadvantage is that the two are then interdependent, so if the client crashes, so does the server.

For more information see:

[Function] [Server] [Stethoscope]

Suggested Exercise:

Experiment with scoping and plotting some of the Function examples from earlier sections, or some Functions of your own creation. Try experimenting with different duration or zoom values.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Getting Help]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 196

Start Your Engines

Before we can make any sound, we need to start or 'boot' a server application. The easiest way to do this is to use one of the server windows which is automatically created by the client app. These can be found in the bottom left-hand corner of your screen. Look for the one that says 'localhost server'. It should look like this:

<img src="Pasted%20Graphic.tiff" alt="Pasted Graphic.tiff">

'localhost' just means on your local computer, as opposed to running on a different computer connected by a network. To start the server click on the 'Boot' button, or click on the window and press the space bar. After a second or two it should look something like this:

<img src="Pasted%20Graphic_1.tiff" alt="Pasted Graphic_1.tiff">

Notice that the name has lit up red, and that the 'Boot' button has changed to 'Quit'. This indicates that the server is running. As well the window provides you with some information about CPU usage, and some other things which probably aren't too clear yet. More about them soon.

Also take a look at the post window, where SC has given you some info, and let you know that it booted okay:

```
booting 57110
SC_AudioDriver:  numSamples=512, sampleRate=44100.000000
start   UseSeparateIO?:  0
PublishPortToRendezvous 0 57110
SuperCollider 3 server ready..
notification is on
```

If for some reason it had failed to boot, there would be some information indicating that.

By default you can refer to the localhost server in your code by using the letter `s`. You

can thus send messages to start and stop it like so:

```
s.quit;
s.boot;
```

Try this out and then leave the server running. Many examples in the documentation have `s.boot` at the beginning, but in general you should make sure the server is running before using any examples that generate audio, or otherwise access the server. In general the examples in this tutorial assume that the server is running.

You can also refer to the localhost server with the text '`Server.local`', for example:

```
Server.local.boot;
```

For more information see:

[Server]

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Functions and Other Functionality]**

Click here to return to the table of Contents: **[Getting Started With SC]**

ID: 197

SynthDefs and Synths

Now that we've covered some basic information we're going to start looking at server abstractions, which are the various classes in the language app which represent things on the server. When looking at these it is important to understand that these objects are just client-side *representations* of parts of the server's architecture, and should not be confused with those parts themselves. Server abstraction objects are simply conveniences.

Distinguishing between the two can be a little confusing, so in general I refer herein to the client-side classes with uppercase names, and the corresponding aspects of server architecture with lowercase names, i.e. Synth vs. synth.

You've already met one kind of server abstraction, class Server itself. The objects referred to by Server.local and Server.internal (and whichever one is stored in the interpreter variable 's' at any given moment) are instances of Server.

Now it's time to get familiar with the rest of them. The first thing we'll look at is the class SynthDef, which is short for 'synth definition'.

Meet the SynthDef

Up until now we've been using Functions to generate audio. This way of working is very useful for quick testing, and in cases where maximum flexibility is needed. This is because each time we execute the code, the Function is evaluated anew, which means the results can vary greatly.

The server, however, doesn't understand Functions, or OOP, or the SC language. It wants information on how to create audio output in a special form called a synth definition. A synth defintion is data about UGens and how they're interconnected. This is sent in a kind of special optimised form, called 'byte code', which the server can deal with very efficiently.

Once the server has a synth definition, it's can very efficiently use it to make a number of synths based on it. Synths on the server are basically just things that make or process sound, or produce control signals to drive other synths.

This relationship between synth definitions and synths is something like that between

classes and instances, in that the former is a template for the latter. But remember that the server app knows nothing about OOP.

Luckily for us there are classes in the language such as SynthDef, which make is easy to create the necessary byte code and send it to the server, and to deal with synth definitions in an object oriented way.

Whenever you use any of Function's audio creating methods what happens is that a corresponding instance of SynthDef is created 'behind the scenes', so to speak, and the necessary byte code is generated and sent to the server, where a synth is created to play the desired audio. So Function's audio methods provide a kind of convenience for you, so that you don't have to take care of this.

So how do you make a SynthDef yourself? You use its 'new' method. Let's compare a by now familiar Function based example, and make an equivalent SynthDef. Like Function, SynthDef also has a convenient play method, so we can easily confirm that these two are equivalent.

```
  //first the Function
{ SinOsc.ar(440, 0, 0.2) }.play;


 // now here's an equivalent SynthDef
SynthDef.new("tutorial-SinOsc", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).play;
```

SynthDef-new takes a number of arguments. The first is a name, usually in the form of a String as above. The second is in fact a Function. This argument is called a UGen Graph Function, as it tells the server how to connect together its various UGens.

SynthDefs vs. Functions

This UGen Graph Function we used in the second example above is similar to the Function we used in the first one, but with one notable difference: It has an extra UGen called Out. Out writes out an ar or kr signal to one of the server's busses, which can be thought of as mixer channels or outputs. We'll discuss busses in greater detail later, but for now just be aware that they're used for playing audio out of the computer, and for reading it in from sources such as microphones.

Out takes two arguments: The first is the index number of the bus to write out on. These start from 0, which on a stereo setup is usually the left output channel. The second is either a UGen or an Array of UGens. If you provide an array (i.e. a multichannel

output) then the first channel will be played out on the bus with the indicated index, the second channel on the bus with the indicated index + 1, and so on.

Here's a stereo example to make clear how this works. The SinOsc with the frequency argument of 440 Hz will be played out on bus 0 (the left channel), and the SinOsc with the frequency argument of 442 Hz will be played out on bus 1 (the right channel).

```
(
 SynthDef     "tutorial-SinOsc-stereo"    var
outArray = [SinOsc.ar(440, 0, 0.2), SinOsc.ar(442, 0, 0.2)];
Out.ar(0, outArray)
}).play;
)
```

When you use Function-play an Out UGen is in fact created for you if you do not explicitly create one. The default bus index for this Out UGen is 0.

Both Function-play and SynthDef-play return another type of object, a Synth, which represents a synth on the server. If you store this object by assigning it to a variable you can control it's behaviour in various ways. For instance the method 'free' causes the synth on the server to stop playing and its memory and cpu resources to be freed.

```
x = { SinOsc.ar(660, 0, 0.2) }.play;
y = SynthDef.new("tutorial-SinOsc", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).play;
 x.free; // free just x
 y.free; // free just y
```

This is more flexible than Cmd-., which frees all synths at once.

SynthDef has two methods which cause the corresponding byte code to be sent to the server app without immediately creating a synth: send and load. The difference between these two is that send streams the definition over the network, and load writes the definiton to disk as a file so that the server can load it. Such a file will have the extension .scsyndef (so for example tutorial-SinOsc.scsyndef), and will be written into the synthdefs/ directory within the main SC directory. This will remain there until you specifically delete it, and will be loaded automatically whenever you boot a server.

In general should use 'send' unless you're going to reuse the def all the time. It is however sometimes necessary to use 'load' with very large or complicated defs, due to limits on packet size on the network.

You can create many, many Synths using the same Function or SynthDef, but using SynthDef has certain advantages, as well as certain limitations.

Once you have a def in a server app, you can create many synths from it with a relatively low overhead of CPU. You can do this with Synth's new method, which takes a def's name as its first argument.

```
SynthDef.new("tutorial-PinkNoise", { Out.ar(0, PinkNoise.ar(0.3)) }).send(s);
 x = Synth      "tutorial-PinkNoise"
 y = Synth      "tutorial-PinkNoise"
x.free; y.free;
```

This is more efficient than repeatedly calling play on the same Function, as it saves the effort of evaluating the Function, compiling the byte code, and sending it multiple times. In many cases this saving in CPU usage is so small as to be largely insignificant, but when doing things like 'mass producing' synths, this can be important.

A corresponding limitation to working with SynthDefs directly is that the UGen Graph Function in a SynthDef is evaluated *once and only once*. (Remember that the server knows nothing about the SC language.) This means that it is somewhat less flexible. Compare these two examples:

```
      // first with a Function.  Note the random frequency each time 'play' is called.
f = { SinOsc.ar(440 + 200.rand, 0, 0.2) };
x = f.play;
y = f.play;
z = f.play;
x.free; y.free; z.free;

 // Now with a SynthDef.  No randomness!
SynthDef("tutorial-NoRand", { Out.ar(0, SinOsc.ar(440 + 200.rand, 0, 0.2)) }).send(s);
 x = Synth "tutorial-NoRand"
 y = Synth "tutorial-NoRand"
 z = Synth "tutorial-NoRand"
x.free; y.free; z.free;
```

Each time you create a new Synth based on the def, the frequency is the same. This is because the Function (and thus `200.rand`) is only evaluated only once, when the SynthDef is created.

Creating Variety with SynthDefs

There are numerous ways of getting variety out of SynthDefs, however. Some things, such as randomness, can be accomplished with various UGens. One example is **[Rand]**, which calculates a random number between low and high values when a synth is first created:

```
  // With Rand, it works!
SynthDef("tutorial-Rand", { Out.ar(0, SinOsc.ar(Rand(440, 660), 0, 0.2)) }).send(s);
 x = Synth "tutorial-Rand"
 y = Synth "tutorial-Rand"
 z = Synth "tutorial-Rand"
x.free; y.free; z.free;
```

The **[UGens]** overview lists a number of such UGens.

The most common way of creating variables is through putting arguments into the UGen Graph Function. This allows you to set different values when the synth is created. These are passed in an array as the second argument to Synth-new. The array should contain pairs of arg names and values.

```
(
SynthDef("tutorial-args", { arg freq = 440, out = 0;
Out.ar(out, SinOsc.ar(freq, 0, 0.2));
}).send(s);
)
 x = Synth "tutorial-args"          // no args, so default values
 y = Synth "tutorial-args"   "freq"           // change freq
 z = Synth "tutorial-args"   "freq"       "out"       // change freq and output channel
x.free; y.free; z.free;
```

This combination of args and UGens means that you can get a lot of mileage out of a single def, but in some cases where maximum flexibility is required, you may still need to use Functions, or create multiple defs.

More About Synth

Synth understands some methods which allow you to change the values of args after a synth has been created. For now we'll just look at one, 'set'. Synth-set takes pairs of

arg names and values.

```
Server.default = Server.internal;
 s = Server.default;
s.boot;
(
SynthDef.new("tutorial-args", { arg freq = 440, out = 0;
Out.ar(out, SinOsc.ar(freq, 0, 0.2));
}).send(s);
)
 s.scope; // scope so you can see the effect
 x = Synth.new "tutorial-args"
x.set("freq", 660);
x.set("freq", 880, "out", 1);
x.free;
```

Some Notes on Symbols, Strings, SynthDef and Arg Names

SynthDef names and argument names can be either a String, as we've seen above, or another kind of literal called a Symbol. You write symbols in one of two ways, either enclosed in single quotes: `'tutorial-SinOsc'`or preceded by a backslash: `\tutorial-SinOsc`. Like Strings Symbols are made up of alpha-numeric sequences. The difference between Strings and Symbols is that all Symbols with the same text are guaranteed to be identical, i.e. the exact same object, whereas with Strings this might not be the case. You can test for this using '===='. Execute the following and watch the post window.

```
     "a String"    "a String"   // this will post false
\aSymbol    'aSymbol'   // this will post true
```

In general in methods which communicate with the server one can use Strings and Symbols interchangeably, but be aware that this is not necessarily true in general code.

```
     "this"    \this   // this will post false
```

For more information see:

[SynthDef] [Synth] [String] [Symbol] [Literals] [Randomness] [UGens]

Suggested Exercise:

Try converting some of the earlier Function based examples, or Functions of your own, to SynthDef versions, adding Out UGens. Experiment with adding and changing arguments both when the synths are created, and afterwards using 'set'.

This document is part of the tutorial **Getting Started With SuperCollider**.

Click here to go on to the next section: **[Busses]**

Click here to return to the table of Contents: **[Getting Started With SC]**

10 GUI

ID: 198

Color

superclass: Object

Creation

Each component has a value from 0.0 to 1.0, except in new255.

***new(red, green,blue, alpha)**
***new255(red, green,blue, alpha)**
create new color in RGB mode. ***new255** takes 8-bit values as arguments.

***fromArray(array)**
create new color in RGB mode from a 3-4dim. array.

***hsv(hue, sat, val, alpha)**
create new color in HSV mode.

***rand(hue, sat, val, alpha)**
create new random color.

***black**
create black.
***white**
create white.
***clear**
create translucent.
***red(val = 1.0, alpha = 1.0)**
create red.
***green { arg val = 1.0, alpha = 1.0)**
create green.
***yellow { arg val = 1.0, alpha = 1.0)**
create yellow.
***blue { arg val = 1.0, alpha = 1.0)**
create blue.

***grey(grey = 0.5, alpha = 1.0)**

*gray(gray = 0.5, alpha = 1.0)
create gray/grey

Accessing

<>red
Get the red component.

```
Color.new(0.47, 0.94, 0.31, 1).red.postln;
```

<>green
Get the green component.

```
Color.new(0.47, 0.94, 0.31, 1).green.postln;
```

<>blue
Get the blue component.

```
Color.new(0.47, 0.94, 0.31, 1).blue.postln;
```

<>alpha
Get the alpha component.

```
Color.new(0.47, 0.94, 0.31, 1).alpha.postln;
```

<asHSV
returns an Array of [Hue, Sat, Val].

```
Color.rand.asHSV.postln;
```

<asArray
returns an Array of [R,G,B].

```
Color.rand.asArray.postln;
```

Manipulating

scaleByAlpha
uses the alpha-value to scale all other colorsParts; alpha then is 1.

blend(arg that, blend)
blends two Colors.

vary(val=0.1, lo=0.3, hi=0.9, alphaVal=0)
varies a Color.

GUI drawing

The following methods must be called within an SCWindow-drawHook or a SCUserView-drawFunc function, and will only be visible once the window or the view is refreshed. Each call to SCWindow-refresh SCUserView-refresh will 'overwrite' all previous drawing by executing the currently defined function.

See also: **[SCWindow]**, **[SCUserView]**, **[Pen]** and **[String]**

setStroke
sets the stroke color.

setFill
sets the fill color.

set
sets the stroke and the fill color.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
 // set the Color
Pen.translate(200, 200);
30.do{
Pen.width = 3.0.rand;
Color.blue(rrand(0.0, 1), rrand(0.0, 0.5)).setStroke;
Color.red(rrand(0.0, 1), rrand(0.0, 0.5)).setFill;
Pen.addAnnularWedge(
0@0,
rrand(10, 50),
rrand(51, 100),
2pi.rand,
2pi.rand
);
Pen.perform([\stroke, \fill].choose);
}
};
w.refresh;
```

```
)
```

X-windows colors :

```
'alice blue' -> Color.new255(240, 248, 255),
'AliceBlue' -> Color.new255(240, 248, 255),
'antique white' -> Color.new255(250, 235, 215),
'AntiqueWhite' -> Color.new255(250, 235, 215),
'AntiqueWhite1' -> Color.new255(255, 239, 219),
'AntiqueWhite2' -> Color.new255(238, 223, 204),
'AntiqueWhite3' -> Color.new255(205, 192, 176),
'AntiqueWhite4' -> Color.new255(139, 131, 120),
'aquamarine' -> Color.new255(127, 255, 212),
'aquamarine1' -> Color.new255(127, 255, 212),
'aquamarine2' -> Color.new255(118, 238, 198),
'aquamarine3' -> Color.new255(102, 205, 170),
'aquamarine4' -> Color.new255(69, 139, 116),
'azure' -> Color.new255(240, 255, 255),
'azure1' -> Color.new255(240, 255, 255),
'azure2' -> Color.new255(224, 238, 238),
'azure3' -> Color.new255(193, 205, 205),
'azure4' -> Color.new255(131, 139, 139),
'beige' -> Color.new255(245, 245, 220),
'bisque' -> Color.new255(255, 228, 196),
'bisque1' -> Color.new255(255, 228, 196),
'bisque2' -> Color.new255(238, 213, 183),
'bisque3' -> Color.new255(205, 183, 158),
'bisque4' -> Color.new255(139, 125, 107),
'black' -> Color.new255(0, 0, 0),
'blanched almond' -> Color.new255(255, 235, 205),
'BlanchedAlmond' -> Color.new255(255, 235, 205),
'blue' -> Color.new255(0, 0, 255),
'blue violet' -> Color.new255(138, 43, 226),
'blue1' -> Color.new255(0, 0, 255),
'blue2' -> Color.new255(0, 0, 238),
'blue3' -> Color.new255(0, 0, 205),
'blue4' -> Color.new255(0, 0, 139),
'BlueViolet' -> Color.new255(138, 43, 226),
'brown' -> Color.new255(165, 42, 42),
```

```
'brown1' -> Color.new255(255, 64, 64),
'brown2' -> Color.new255(238, 59, 59),
'brown3' -> Color.new255(205, 51, 51),
'brown4' -> Color.new255(139, 35, 35),
'burlywood' -> Color.new255(222, 184, 135),
'burlywood1' -> Color.new255(255, 211, 155),
'burlywood2' -> Color.new255(238, 197, 145),
'burlywood3' -> Color.new255(205, 170, 125),
'burlywood4' -> Color.new255(139, 115, 85),
'cadet blue' -> Color.new255(95, 158, 160),
'CadetBlue' -> Color.new255(95, 158, 160),
'CadetBlue1' -> Color.new255(152, 245, 255),
'CadetBlue2' -> Color.new255(142, 229, 238),
'CadetBlue3' -> Color.new255(122, 197, 205),
'CadetBlue4' -> Color.new255(83, 134, 139),
'chartreuse' -> Color.new255(127, 255, 0),
'chartreuse1' -> Color.new255(127, 255, 0),
'chartreuse2' -> Color.new255(118, 238, 0),
'chartreuse3' -> Color.new255(102, 205, 0),
'chartreuse4' -> Color.new255(69, 139, 0),
'chocolate' -> Color.new255(210, 105, 30),
'chocolate1' -> Color.new255(255, 127, 36),
'chocolate2' -> Color.new255(238, 118, 33),
'chocolate3' -> Color.new255(205, 102, 29),
'chocolate4' -> Color.new255(139, 69, 19),
'coral' -> Color.new255(255, 127, 80),
'coral1' -> Color.new255(255, 114, 86),
'coral2' -> Color.new255(238, 106, 80),
'coral3' -> Color.new255(205, 91, 69),
'coral4' -> Color.new255(139, 62, 47),
'cornflower blue' -> Color.new255(100, 149, 237),
'CornflowerBlue' -> Color.new255(100, 149, 237),
'cornsilk' -> Color.new255(255, 248, 220),
'cornsilk1' -> Color.new255(255, 248, 220),
'cornsilk2' -> Color.new255(238, 232, 205),
'cornsilk3' -> Color.new255(205, 200, 177),
'cornsilk4' -> Color.new255(139, 136, 120),
'cyan' -> Color.new255(0, 255, 255),
'cyan1' -> Color.new255(0, 255, 255),
'cyan2' -> Color.new255(0, 238, 238),
```

```
'cyan3' -> Color.new255(0, 205, 205),
'cyan4' -> Color.new255(0, 139, 139),
'dark goldenrod' -> Color.new255(184, 134, 11),
'dark green' -> Color.new255(0, 100, 0),
'dark khaki' -> Color.new255(189, 183, 107),
'dark olive green' -> Color.new255(85, 107, 47),
'dark orange' -> Color.new255(255, 140, 0),
'dark orchid' -> Color.new255(153, 50, 204),
'dark salmon' -> Color.new255(233, 150, 122),
'dark sea green' -> Color.new255(143, 188, 143),
'dark slate blue' -> Color.new255(72, 61, 139),
'dark slate gray' -> Color.new255(47, 79, 79),
'dark slate grey' -> Color.new255(47, 79, 79),
'dark turquoise' -> Color.new255(0, 206, 209),
'dark violet' -> Color.new255(148, 0, 211),
'DarkGoldenrod' -> Color.new255(184, 134, 11),
'DarkGoldenrod1' -> Color.new255(255, 185, 15),
'DarkGoldenrod2' -> Color.new255(238, 173, 14),
'DarkGoldenrod3' -> Color.new255(205, 149, 12),
'DarkGoldenrod4' -> Color.new255(139, 101, 8),
'DarkGreen' -> Color.new255(0, 100, 0),
'DarkKhaki' -> Color.new255(189, 183, 107),
'DarkOliveGreen' -> Color.new255(85, 107, 47),
'DarkOliveGreen1' -> Color.new255(202, 255, 112),
'DarkOliveGreen2' -> Color.new255(188, 238, 104),
'DarkOliveGreen3' -> Color.new255(162, 205, 90),
'DarkOliveGreen4' -> Color.new255(110, 139, 61),
'DarkOrange' -> Color.new255(255, 140, 0),
'DarkOrange1' -> Color.new255(255, 127, 0),
'DarkOrange2' -> Color.new255(238, 118, 0),
'DarkOrange3' -> Color.new255(205, 102, 0),
'DarkOrange4' -> Color.new255(139, 69, 0),
'DarkOrchid' -> Color.new255(153, 50, 204),
'DarkOrchid1' -> Color.new255(191, 62, 255),
'DarkOrchid2' -> Color.new255(178, 58, 238),
'DarkOrchid3' -> Color.new255(154, 50, 205),
'DarkOrchid4' -> Color.new255(104, 34, 139),
'DarkSalmon' -> Color.new255(233, 150, 122),
'DarkSeaGreen' -> Color.new255(143, 188, 143),
'DarkSeaGreen1' -> Color.new255(193, 255, 193),
```

```
'DarkSeaGreen2' -> Color.new255(180, 238, 180),
'DarkSeaGreen3' -> Color.new255(155, 205, 155),
'DarkSeaGreen4' -> Color.new255(105, 139, 105),
'DarkSlateBlue' -> Color.new255(72, 61, 139),
'DarkSlateGray' -> Color.new255(47, 79, 79),
'DarkSlateGray1' -> Color.new255(151, 255, 255),
'DarkSlateGray2' -> Color.new255(141, 238, 238),
'DarkSlateGray3' -> Color.new255(121, 205, 205),
'DarkSlateGray4' -> Color.new255(82, 139, 139),
'DarkSlateGrey' -> Color.new255(47, 79, 79),
'DarkTurquoise' -> Color.new255(0, 206, 209),
'DarkViolet' -> Color.new255(148, 0, 211),
'deep pink' -> Color.new255(255, 20, 147),
'deep sky blue' -> Color.new255(0, 191, 255),
'DeepPink' -> Color.new255(255, 20, 147),
'DeepPink1' -> Color.new255(255, 20, 147),
'DeepPink2' -> Color.new255(238, 18, 137),
'DeepPink3' -> Color.new255(205, 16, 118),
'DeepPink4' -> Color.new255(139, 10, 80),
'DeepSkyBlue' -> Color.new255(0, 191, 255),
'DeepSkyBlue1' -> Color.new255(0, 191, 255),
'DeepSkyBlue2' -> Color.new255(0, 178, 238),
'DeepSkyBlue3' -> Color.new255(0, 154, 205),
'DeepSkyBlue4' -> Color.new255(0, 104, 139),
'dim gray' -> Color.new255(105, 105, 105),
'dim grey' -> Color.new255(105, 105, 105),
'DimGray' -> Color.new255(105, 105, 105),
'DimGrey' -> Color.new255(105, 105, 105),
'dodger blue' -> Color.new255(30, 144, 255),
'DodgerBlue' -> Color.new255(30, 144, 255),
'DodgerBlue1' -> Color.new255(30, 144, 255),
'DodgerBlue2' -> Color.new255(28, 134, 238),
'DodgerBlue3' -> Color.new255(24, 116, 205),
'DodgerBlue4' -> Color.new255(16, 78, 139),
'firebrick' -> Color.new255(178, 34, 34),
'firebrick1' -> Color.new255(255, 48, 48),
'firebrick2' -> Color.new255(238, 44, 44),
'firebrick3' -> Color.new255(205, 38, 38),
'firebrick4' -> Color.new255(139, 26, 26),
'floral white' -> Color.new255(255, 250, 240),
```

```
'FloralWhite' -> Color.new255(255, 250, 240),
'forest green' -> Color.new255(34, 139, 34),
'ForestGreen' -> Color.new255(34, 139, 34),
'gainsboro' -> Color.new255(220, 220, 220),
'ghost white' -> Color.new255(248, 248, 255),
'GhostWhite' -> Color.new255(248, 248, 255),
'gold' -> Color.new255(255, 215, 0),
'gold1' -> Color.new255(255, 215, 0),
'gold2' -> Color.new255(238, 201, 0),
'gold3' -> Color.new255(205, 173, 0),
'gold4' -> Color.new255(139, 117, 0),
'goldenrod' -> Color.new255(218, 165, 32),
'goldenrod1' -> Color.new255(255, 193, 37),
'goldenrod2' -> Color.new255(238, 180, 34),
'goldenrod3' -> Color.new255(205, 155, 29),
'goldenrod4' -> Color.new255(139, 105, 20),
'gray' -> Color.new255(190, 190, 190),
'gray0' -> Color.new255(0, 0, 0),
'gray1' -> Color.new255(3, 3, 3),
'gray10' -> Color.new255(26, 26, 26),
'gray100' -> Color.new255(255, 255, 255),
'gray11' -> Color.new255(28, 28, 28),
'gray12' -> Color.new255(31, 31, 31),
'gray13' -> Color.new255(33, 33, 33),
'gray14' -> Color.new255(36, 36, 36),
'gray15' -> Color.new255(38, 38, 38),
'gray16' -> Color.new255(41, 41, 41),
'gray17' -> Color.new255(43, 43, 43),
'gray18' -> Color.new255(46, 46, 46),
'gray19' -> Color.new255(48, 48, 48),
'gray2' -> Color.new255(5, 5, 5),
'gray20' -> Color.new255(51, 51, 51),
'gray21' -> Color.new255(54, 54, 54),
'gray22' -> Color.new255(56, 56, 56),
'gray23' -> Color.new255(59, 59, 59),
'gray24' -> Color.new255(61, 61, 61),
'gray25' -> Color.new255(64, 64, 64),
'gray26' -> Color.new255(66, 66, 66),
'gray27' -> Color.new255(69, 69, 69),
'gray28' -> Color.new255(71, 71, 71),
```

```
'gray29' -> Color.new255(74, 74, 74),
'gray3' -> Color.new255(8, 8, 8),
'gray30' -> Color.new255(77, 77, 77),
'gray31' -> Color.new255(79, 79, 79),
'gray32' -> Color.new255(82, 82, 82),
'gray33' -> Color.new255(84, 84, 84),
'gray34' -> Color.new255(87, 87, 87),
'gray35' -> Color.new255(89, 89, 89),
'gray36' -> Color.new255(92, 92, 92),
'gray37' -> Color.new255(94, 94, 94),
'gray38' -> Color.new255(97, 97, 97),
'gray39' -> Color.new255(99, 99, 99),
'gray4' -> Color.new255(10, 10, 10),
'gray40' -> Color.new255(102, 102, 102),
'gray41' -> Color.new255(105, 105, 105),
'gray42' -> Color.new255(107, 107, 107),
'gray43' -> Color.new255(110, 110, 110),
'gray44' -> Color.new255(112, 112, 112),
'gray45' -> Color.new255(115, 115, 115),
'gray46' -> Color.new255(117, 117, 117),
'gray47' -> Color.new255(120, 120, 120),
'gray48' -> Color.new255(122, 122, 122),
'gray49' -> Color.new255(125, 125, 125),
'gray5' -> Color.new255(13, 13, 13),
'gray50' -> Color.new255(127, 127, 127),
'gray51' -> Color.new255(130, 130, 130),
'gray52' -> Color.new255(133, 133, 133),
'gray53' -> Color.new255(135, 135, 135),
'gray54' -> Color.new255(138, 138, 138),
'gray55' -> Color.new255(140, 140, 140),
'gray56' -> Color.new255(143, 143, 143),
'gray57' -> Color.new255(145, 145, 145),
'gray58' -> Color.new255(148, 148, 148),
'gray59' -> Color.new255(150, 150, 150),
'gray6' -> Color.new255(15, 15, 15),
'gray60' -> Color.new255(153, 153, 153),
'gray61' -> Color.new255(156, 156, 156),
'gray62' -> Color.new255(158, 158, 158),
'gray63' -> Color.new255(161, 161, 161),
'gray64' -> Color.new255(163, 163, 163),
```

```
'gray65' -> Color.new255(166, 166, 166),
'gray66' -> Color.new255(168, 168, 168),
'gray67' -> Color.new255(171, 171, 171),
'gray68' -> Color.new255(173, 173, 173),
'gray69' -> Color.new255(176, 176, 176),
'gray7' -> Color.new255(18, 18, 18),
'gray70' -> Color.new255(179, 179, 179),
'gray71' -> Color.new255(181, 181, 181),
'gray72' -> Color.new255(184, 184, 184),
'gray73' -> Color.new255(186, 186, 186),
'gray74' -> Color.new255(189, 189, 189),
'gray75' -> Color.new255(191, 191, 191),
'gray76' -> Color.new255(194, 194, 194),
'gray77' -> Color.new255(196, 196, 196),
'gray78' -> Color.new255(199, 199, 199),
'gray79' -> Color.new255(201, 201, 201),
'gray8' -> Color.new255(20, 20, 20),
'gray80' -> Color.new255(204, 204, 204),
'gray81' -> Color.new255(207, 207, 207),
'gray82' -> Color.new255(209, 209, 209),
'gray83' -> Color.new255(212, 212, 212),
'gray84' -> Color.new255(214, 214, 214),
'gray85' -> Color.new255(217, 217, 217),
'gray86' -> Color.new255(219, 219, 219),
'gray87' -> Color.new255(222, 222, 222),
'gray88' -> Color.new255(224, 224, 224),
'gray89' -> Color.new255(227, 227, 227),
'gray9' -> Color.new255(23, 23, 23),
'gray90' -> Color.new255(229, 229, 229),
'gray91' -> Color.new255(232, 232, 232),
'gray92' -> Color.new255(235, 235, 235),
'gray93' -> Color.new255(237, 237, 237),
'gray94' -> Color.new255(240, 240, 240),
'gray95' -> Color.new255(242, 242, 242),
'gray96' -> Color.new255(245, 245, 245),
'gray97' -> Color.new255(247, 247, 247),
'gray98' -> Color.new255(250, 250, 250),
'gray99' -> Color.new255(252, 252, 252),
'green' -> Color.new255(0, 255, 0),
'green yellow' -> Color.new255(173, 255, 47),
```

```
'green1' -> Color.new255(0, 255, 0),
'green2' -> Color.new255(0, 238, 0),
'green3' -> Color.new255(0, 205, 0),
'green4' -> Color.new255(0, 139, 0),
'GreenYellow' -> Color.new255(173, 255, 47),
'grey' -> Color.new255(190, 190, 190),
'grey0' -> Color.new255(0, 0, 0),
'grey1' -> Color.new255(3, 3, 3),
'grey10' -> Color.new255(26, 26, 26),
'grey100' -> Color.new255(255, 255, 255),
'grey11' -> Color.new255(28, 28, 28),
'grey12' -> Color.new255(31, 31, 31),
'grey13' -> Color.new255(33, 33, 33),
'grey14' -> Color.new255(36, 36, 36),
'grey15' -> Color.new255(38, 38, 38),
'grey16' -> Color.new255(41, 41, 41),
'grey17' -> Color.new255(43, 43, 43),
'grey18' -> Color.new255(46, 46, 46),
'grey19' -> Color.new255(48, 48, 48),
'grey2' -> Color.new255(5, 5, 5),
'grey20' -> Color.new255(51, 51, 51),
'grey21' -> Color.new255(54, 54, 54),
'grey22' -> Color.new255(56, 56, 56),
'grey23' -> Color.new255(59, 59, 59),
'grey24' -> Color.new255(61, 61, 61),
'grey25' -> Color.new255(64, 64, 64),
'grey26' -> Color.new255(66, 66, 66),
'grey27' -> Color.new255(69, 69, 69),
'grey28' -> Color.new255(71, 71, 71),
'grey29' -> Color.new255(74, 74, 74),
'grey3' -> Color.new255(8, 8, 8),
'grey30' -> Color.new255(77, 77, 77),
'grey31' -> Color.new255(79, 79, 79),
'grey32' -> Color.new255(82, 82, 82),
'grey33' -> Color.new255(84, 84, 84),
'grey34' -> Color.new255(87, 87, 87),
'grey35' -> Color.new255(89, 89, 89),
'grey36' -> Color.new255(92, 92, 92),
'grey37' -> Color.new255(94, 94, 94),
'grey38' -> Color.new255(97, 97, 97),
```

```
'grey39' -> Color.new255(99, 99, 99),
'grey4' -> Color.new255(10, 10, 10),
'grey40' -> Color.new255(102, 102, 102),
'grey41' -> Color.new255(105, 105, 105),
'grey42' -> Color.new255(107, 107, 107),
'grey43' -> Color.new255(110, 110, 110),
'grey44' -> Color.new255(112, 112, 112),
'grey45' -> Color.new255(115, 115, 115),
'grey46' -> Color.new255(117, 117, 117),
'grey47' -> Color.new255(120, 120, 120),
'grey48' -> Color.new255(122, 122, 122),
'grey49' -> Color.new255(125, 125, 125),
'grey5' -> Color.new255(13, 13, 13),
'grey50' -> Color.new255(127, 127, 127),
'grey51' -> Color.new255(130, 130, 130),
'grey52' -> Color.new255(133, 133, 133),
'grey53' -> Color.new255(135, 135, 135),
'grey54' -> Color.new255(138, 138, 138),
'grey55' -> Color.new255(140, 140, 140),
'grey56' -> Color.new255(143, 143, 143),
'grey57' -> Color.new255(145, 145, 145),
'grey58' -> Color.new255(148, 148, 148),
'grey59' -> Color.new255(150, 150, 150),
'grey6' -> Color.new255(15, 15, 15),
'grey60' -> Color.new255(153, 153, 153),
'grey61' -> Color.new255(156, 156, 156),
'grey62' -> Color.new255(158, 158, 158),
'grey63' -> Color.new255(161, 161, 161),
'grey64' -> Color.new255(163, 163, 163),
'grey65' -> Color.new255(166, 166, 166),
'grey66' -> Color.new255(168, 168, 168),
'grey67' -> Color.new255(171, 171, 171),
'grey68' -> Color.new255(173, 173, 173),
'grey69' -> Color.new255(176, 176, 176),
'grey7' -> Color.new255(18, 18, 18),
'grey70' -> Color.new255(179, 179, 179),
'grey71' -> Color.new255(181, 181, 181),
'grey72' -> Color.new255(184, 184, 184),
'grey73' -> Color.new255(186, 186, 186),
'grey74' -> Color.new255(189, 189, 189),
```

```
'grey75' -> Color.new255(191, 191, 191),
'grey76' -> Color.new255(194, 194, 194),
'grey77' -> Color.new255(196, 196, 196),
'grey78' -> Color.new255(199, 199, 199),
'grey79' -> Color.new255(201, 201, 201),
'grey8' -> Color.new255(20, 20, 20),
'grey80' -> Color.new255(204, 204, 204),
'grey81' -> Color.new255(207, 207, 207),
'grey82' -> Color.new255(209, 209, 209),
'grey83' -> Color.new255(212, 212, 212),
'grey84' -> Color.new255(214, 214, 214),
'grey85' -> Color.new255(217, 217, 217),
'grey86' -> Color.new255(219, 219, 219),
'grey87' -> Color.new255(222, 222, 222),
'grey88' -> Color.new255(224, 224, 224),
'grey89' -> Color.new255(227, 227, 227),
'grey9' -> Color.new255(23, 23, 23),
'grey90' -> Color.new255(229, 229, 229),
'grey91' -> Color.new255(232, 232, 232),
'grey92' -> Color.new255(235, 235, 235),
'grey93' -> Color.new255(237, 237, 237),
'grey94' -> Color.new255(240, 240, 240),
'grey95' -> Color.new255(242, 242, 242),
'grey96' -> Color.new255(245, 245, 245),
'grey97' -> Color.new255(247, 247, 247),
'grey98' -> Color.new255(250, 250, 250),
'grey99' -> Color.new255(252, 252, 252),
'honeydew' -> Color.new255(240, 255, 240),
'honeydew1' -> Color.new255(240, 255, 240),
'honeydew2' -> Color.new255(224, 238, 224),
'honeydew3' -> Color.new255(193, 205, 193),
'honeydew4' -> Color.new255(131, 139, 131),
'hot pink' -> Color.new255(255, 105, 180),
'HotPink' -> Color.new255(255, 105, 180),
'HotPink1' -> Color.new255(255, 110, 180),
'HotPink2' -> Color.new255(238, 106, 167),
'HotPink3' -> Color.new255(205, 96, 144),
'HotPink4' -> Color.new255(139, 58, 98),
'indian red' -> Color.new255(205, 92, 92),
'IndianRed' -> Color.new255(205, 92, 92),
```

```
'IndianRed1' -> Color.new255(255, 106, 106),
'IndianRed2' -> Color.new255(238, 99, 99),
'IndianRed3' -> Color.new255(205, 85, 85),
'IndianRed4' -> Color.new255(139, 58, 58),
'ivory' -> Color.new255(255, 255, 240),
'ivory1' -> Color.new255(255, 255, 240),
'ivory2' -> Color.new255(238, 238, 224),
'ivory3' -> Color.new255(205, 205, 193),
'ivory4' -> Color.new255(139, 139, 131),
'khaki' -> Color.new255(240, 230, 140),
'khaki1' -> Color.new255(255, 246, 143),
'khaki2' -> Color.new255(238, 230, 133),
'khaki3' -> Color.new255(205, 198, 115),
'khaki4' -> Color.new255(139, 134, 78),
'lavender' -> Color.new255(230, 230, 250),
'lavender blush' -> Color.new255(255, 240, 245),
'LavenderBlush' -> Color.new255(255, 240, 245),
'LavenderBlush1' -> Color.new255(255, 240, 245),
'LavenderBlush2' -> Color.new255(238, 224, 229),
'LavenderBlush3' -> Color.new255(205, 193, 197),
'LavenderBlush4' -> Color.new255(139, 131, 134),
'lawn green' -> Color.new255(124, 252, 0),
'LawnGreen' -> Color.new255(124, 252, 0),
'lemon chiffon' -> Color.new255(255, 250, 205),
'LemonChiffon' -> Color.new255(255, 250, 205),
'LemonChiffon1' -> Color.new255(255, 250, 205),
'LemonChiffon2' -> Color.new255(238, 233, 191),
'LemonChiffon3' -> Color.new255(205, 201, 165),
'LemonChiffon4' -> Color.new255(139, 137, 112),
'light blue' -> Color.new255(173, 216, 230),
'light coral' -> Color.new255(240, 128, 128),
'light cyan' -> Color.new255(224, 255, 255),
'light goldenrod' -> Color.new255(238, 221, 130),
'light goldenrod yellow' -> Color.new255(250, 250, 210),
'light gray' -> Color.new255(211, 211, 211),
'light grey' -> Color.new255(211, 211, 211),
'light pink' -> Color.new255(255, 182, 193),
'light salmon' -> Color.new255(255, 160, 122),
'light sea green' -> Color.new255(32, 178, 170),
'light sky blue' -> Color.new255(135, 206, 250),
```

```
'light slate blue' -> Color.new255(132, 112, 255),
'light slate gray' -> Color.new255(119, 136, 153),
'light slate grey' -> Color.new255(119, 136, 153),
'light steel blue' -> Color.new255(176, 196, 222),
'light yellow' -> Color.new255(255, 255, 224),
'LightBlue' -> Color.new255(173, 216, 230),
'LightBlue1' -> Color.new255(191, 239, 255),
'LightBlue2' -> Color.new255(178, 223, 238),
'LightBlue3' -> Color.new255(154, 192, 205),
'LightBlue4' -> Color.new255(104, 131, 139),
'LightCoral' -> Color.new255(240, 128, 128),
'LightCyan' -> Color.new255(224, 255, 255),
'LightCyan1' -> Color.new255(224, 255, 255),
'LightCyan2' -> Color.new255(209, 238, 238),
'LightCyan3' -> Color.new255(180, 205, 205),
'LightCyan4' -> Color.new255(122, 139, 139),
'LightGoldenrod' -> Color.new255(238, 221, 130),
'LightGoldenrod1' -> Color.new255(255, 236, 139),
'LightGoldenrod2' -> Color.new255(238, 220, 130),
'LightGoldenrod3' -> Color.new255(205, 190, 112),
'LightGoldenrod4' -> Color.new255(139, 129, 76),
'LightGoldenrodYellow' -> Color.new255(250, 250, 210),
'LightGray' -> Color.new255(211, 211, 211),
'LightGrey' -> Color.new255(211, 211, 211),
'LightPink' -> Color.new255(255, 182, 193),
'LightPink1' -> Color.new255(255, 174, 185),
'LightPink2' -> Color.new255(238, 162, 173),
'LightPink3' -> Color.new255(205, 140, 149),
'LightPink4' -> Color.new255(139, 95, 101),
'LightSalmon' -> Color.new255(255, 160, 122),
'LightSalmon1' -> Color.new255(255, 160, 122),
'LightSalmon2' -> Color.new255(238, 149, 114),
'LightSalmon3' -> Color.new255(205, 129, 98),
'LightSalmon4' -> Color.new255(139, 87, 66),
'LightSeaGreen' -> Color.new255(32, 178, 170),
'LightSkyBlue' -> Color.new255(135, 206, 250),
'LightSkyBlue1' -> Color.new255(176, 226, 255),
'LightSkyBlue2' -> Color.new255(164, 211, 238),
'LightSkyBlue3' -> Color.new255(141, 182, 205),
'LightSkyBlue4' -> Color.new255(96, 123, 139),
```

```
'LightSlateBlue' -> Color.new255(132, 112, 255),
'LightSlateGray' -> Color.new255(119, 136, 153),
'LightSlateGrey' -> Color.new255(119, 136, 153),
'LightSteelBlue' -> Color.new255(176, 196, 222),
'LightSteelBlue1' -> Color.new255(202, 225, 255),
'LightSteelBlue2' -> Color.new255(188, 210, 238),
'LightSteelBlue3' -> Color.new255(162, 181, 205),
'LightSteelBlue4' -> Color.new255(110, 123, 139),
'LightYellow' -> Color.new255(255, 255, 224),
'LightYellow1' -> Color.new255(255, 255, 224),
'LightYellow2' -> Color.new255(238, 238, 209),
'LightYellow3' -> Color.new255(205, 205, 180),
'LightYellow4' -> Color.new255(139, 139, 122),
'lime green' -> Color.new255(50, 205, 50),
'LimeGreen' -> Color.new255(50, 205, 50),
'linen' -> Color.new255(250, 240, 230),
'magenta' -> Color.new255(255, 0, 255),
'magenta1' -> Color.new255(255, 0, 255),
'magenta2' -> Color.new255(238, 0, 238),
'magenta3' -> Color.new255(205, 0, 205),
'magenta4' -> Color.new255(139, 0, 139),
'maroon' -> Color.new255(176, 48, 96),
'maroon1' -> Color.new255(255, 52, 179),
'maroon2' -> Color.new255(238, 48, 167),
'maroon3' -> Color.new255(205, 41, 144),
'maroon4' -> Color.new255(139, 28, 98),
'medium aquamarine' -> Color.new255(102, 205, 170),
'medium blue' -> Color.new255(0, 0, 205),
'medium orchid' -> Color.new255(186, 85, 211),
'medium purple' -> Color.new255(147, 112, 219),
'medium sea green' -> Color.new255(60, 179, 113),
'medium slate blue' -> Color.new255(123, 104, 238),
'medium spring green' -> Color.new255(0, 250, 154),
'medium turquoise' -> Color.new255(72, 209, 204),
'medium violet red' -> Color.new255(199, 21, 133),
'MediumAquamarine' -> Color.new255(102, 205, 170),
'MediumBlue' -> Color.new255(0, 0, 205),
'MediumOrchid' -> Color.new255(186, 85, 211),
'MediumOrchid1' -> Color.new255(224, 102, 255),
'MediumOrchid2' -> Color.new255(209, 95, 238),
```

```
'MediumOrchid3' -> Color.new255(180, 82, 205),
'MediumOrchid4' -> Color.new255(122, 55, 139),
'MediumPurple' -> Color.new255(147, 112, 219),
'MediumPurple1' -> Color.new255(171, 130, 255),
'MediumPurple2' -> Color.new255(159, 121, 238),
'MediumPurple3' -> Color.new255(137, 104, 205),
'MediumPurple4' -> Color.new255(93, 71, 139),
'MediumSeaGreen' -> Color.new255(60, 179, 113),
'MediumSlateBlue' -> Color.new255(123, 104, 238),
'MediumSpringGreen' -> Color.new255(0, 250, 154),
'MediumTurquoise' -> Color.new255(72, 209, 204),
'MediumVioletRed' -> Color.new255(199, 21, 133),
'midnight blue' -> Color.new255(25, 25, 112),
'MidnightBlue' -> Color.new255(25, 25, 112),
'mint cream' -> Color.new255(245, 255, 250),
'MintCream' -> Color.new255(245, 255, 250),
'misty rose' -> Color.new255(255, 228, 225),
'MistyRose' -> Color.new255(255, 228, 225),
'MistyRose1' -> Color.new255(255, 228, 225),
'MistyRose2' -> Color.new255(238, 213, 210),
'MistyRose3' -> Color.new255(205, 183, 181),
'MistyRose4' -> Color.new255(139, 125, 123),
'moccasin' -> Color.new255(255, 228, 181),
'navajo white' -> Color.new255(255, 222, 173),
'NavajoWhite' -> Color.new255(255, 222, 173),
'NavajoWhite1' -> Color.new255(255, 222, 173),
'NavajoWhite2' -> Color.new255(238, 207, 161),
'NavajoWhite3' -> Color.new255(205, 179, 139),
'NavajoWhite4' -> Color.new255(139, 121, 94),
'navy' -> Color.new255(0, 0, 128),
'navy blue' -> Color.new255(0, 0, 128),
'NavyBlue' -> Color.new255(0, 0, 128),
'old lace' -> Color.new255(253, 245, 230),
'OldLace' -> Color.new255(253, 245, 230),
'olive drab' -> Color.new255(107, 142, 35),
'OliveDrab' -> Color.new255(107, 142, 35),
'OliveDrab1' -> Color.new255(192, 255, 62),
'OliveDrab2' -> Color.new255(179, 238, 58),
'OliveDrab3' -> Color.new255(154, 205, 50),
'OliveDrab4' -> Color.new255(105, 139, 34),
```

```
'orange' -> Color.new255(255, 165, 0),
'orange red' -> Color.new255(255, 69, 0),
'orange1' -> Color.new255(255, 165, 0),
'orange2' -> Color.new255(238, 154, 0),
'orange3' -> Color.new255(205, 133, 0),
'orange4' -> Color.new255(139, 90, 0),
'OrangeRed' -> Color.new255(255, 69, 0),
'OrangeRed1' -> Color.new255(255, 69, 0),
'OrangeRed2' -> Color.new255(238, 64, 0),
'OrangeRed3' -> Color.new255(205, 55, 0),
'OrangeRed4' -> Color.new255(139, 37, 0),
'orchid' -> Color.new255(218, 112, 214),
'orchid1' -> Color.new255(255, 131, 250),
'orchid2' -> Color.new255(238, 122, 233),
'orchid3' -> Color.new255(205, 105, 201),
'orchid4' -> Color.new255(139, 71, 137),
'pale goldenrod' -> Color.new255(238, 232, 170),
'pale green' -> Color.new255(152, 251, 152),
'pale turquoise' -> Color.new255(175, 238, 238),
'pale violet red' -> Color.new255(219, 112, 147),
'PaleGoldenrod' -> Color.new255(238, 232, 170),
'PaleGreen' -> Color.new255(152, 251, 152),
'PaleGreen1' -> Color.new255(154, 255, 154),
'PaleGreen2' -> Color.new255(144, 238, 144),
'PaleGreen3' -> Color.new255(124, 205, 124),
'PaleGreen4' -> Color.new255(84, 139, 84),
'PaleTurquoise' -> Color.new255(175, 238, 238),
'PaleTurquoise1' -> Color.new255(187, 255, 255),
'PaleTurquoise2' -> Color.new255(174, 238, 238),
'PaleTurquoise3' -> Color.new255(150, 205, 205),
'PaleTurquoise4' -> Color.new255(102, 139, 139),
'PaleVioletRed' -> Color.new255(219, 112, 147),
'PaleVioletRed1' -> Color.new255(255, 130, 171),
'PaleVioletRed2' -> Color.new255(238, 121, 159),
'PaleVioletRed3' -> Color.new255(205, 104, 137),
'PaleVioletRed4' -> Color.new255(139, 71, 93),
'papaya whip' -> Color.new255(255, 239, 213),
'PapayaWhip' -> Color.new255(255, 239, 213),
'peach puff' -> Color.new255(255, 218, 185),
'PeachPuff' -> Color.new255(255, 218, 185),
```

```
'PeachPuff1' -> Color.new255(255, 218, 185),
'PeachPuff2' -> Color.new255(238, 203, 173),
'PeachPuff3' -> Color.new255(205, 175, 149),
'PeachPuff4' -> Color.new255(139, 119, 101),
'peru' -> Color.new255(205, 133, 63),
'pink' -> Color.new255(255, 192, 203),
'pink1' -> Color.new255(255, 181, 197),
'pink2' -> Color.new255(238, 169, 184),
'pink3' -> Color.new255(205, 145, 158),
'pink4' -> Color.new255(139, 99, 108),
'plum' -> Color.new255(221, 160, 221),
'plum1' -> Color.new255(255, 187, 255),
'plum2' -> Color.new255(238, 174, 238),
'plum3' -> Color.new255(205, 150, 205),
'plum4' -> Color.new255(139, 102, 139),
'powder blue' -> Color.new255(176, 224, 230),
'PowderBlue' -> Color.new255(176, 224, 230),
'purple' -> Color.new255(160, 32, 240),
'purple1' -> Color.new255(155, 48, 255),
'purple2' -> Color.new255(145, 44, 238),
'purple3' -> Color.new255(125, 38, 205),
'purple4' -> Color.new255(85, 26, 139),
'red' -> Color.new255(255, 0, 0),
'red1' -> Color.new255(255, 0, 0),
'red2' -> Color.new255(238, 0, 0),
'red3' -> Color.new255(205, 0, 0),
'red4' -> Color.new255(139, 0, 0),
'rosy brown' -> Color.new255(188, 143, 143),
'RosyBrown' -> Color.new255(188, 143, 143),
'RosyBrown1' -> Color.new255(255, 193, 193),
'RosyBrown2' -> Color.new255(238, 180, 180),
'RosyBrown3' -> Color.new255(205, 155, 155),
'RosyBrown4' -> Color.new255(139, 105, 105),
'royal blue' -> Color.new255(65, 105, 225),
'RoyalBlue' -> Color.new255(65, 105, 225),
'RoyalBlue1' -> Color.new255(72, 118, 255),
'RoyalBlue2' -> Color.new255(67, 110, 238),
'RoyalBlue3' -> Color.new255(58, 95, 205),
'RoyalBlue4' -> Color.new255(39, 64, 139),
'saddle brown' -> Color.new255(139, 69, 19),
```

```
'SaddleBrown' -> Color.new255(139, 69, 19),
'salmon' -> Color.new255(250, 128, 114),
'salmon1' -> Color.new255(255, 140, 105),
'salmon2' -> Color.new255(238, 130, 98),
'salmon3' -> Color.new255(205, 112, 84),
'salmon4' -> Color.new255(139, 76, 57),
'sandy brown' -> Color.new255(244, 164, 96),
'SandyBrown' -> Color.new255(244, 164, 96),
'sea green' -> Color.new255(46, 139, 87),
'SeaGreen' -> Color.new255(46, 139, 87),
'SeaGreen1' -> Color.new255(84, 255, 159),
'SeaGreen2' -> Color.new255(78, 238, 148),
'SeaGreen3' -> Color.new255(67, 205, 128),
'SeaGreen4' -> Color.new255(46, 139, 87),
'seashell' -> Color.new255(255, 245, 238),
'seashell1' -> Color.new255(255, 245, 238),
'seashell2' -> Color.new255(238, 229, 222),
'seashell3' -> Color.new255(205, 197, 191),
'seashell4' -> Color.new255(139, 134, 130),
'sienna' -> Color.new255(160, 82, 45),
'sienna1' -> Color.new255(255, 130, 71),
'sienna2' -> Color.new255(238, 121, 66),
'sienna3' -> Color.new255(205, 104, 57),
'sienna4' -> Color.new255(139, 71, 38),
'sky blue' -> Color.new255(135, 206, 235),
'SkyBlue' -> Color.new255(135, 206, 235),
'SkyBlue1' -> Color.new255(135, 206, 255),
'SkyBlue2' -> Color.new255(126, 192, 238),
'SkyBlue3' -> Color.new255(108, 166, 205),
'SkyBlue4' -> Color.new255(74, 112, 139),
'slate blue' -> Color.new255(106, 90, 205),
'slate gray' -> Color.new255(112, 128, 144),
'slate grey' -> Color.new255(112, 128, 144),
'SlateBlue' -> Color.new255(106, 90, 205),
'SlateBlue1' -> Color.new255(131, 111, 255),
'SlateBlue2' -> Color.new255(122, 103, 238),
'SlateBlue3' -> Color.new255(105, 89, 205),
'SlateBlue4' -> Color.new255(71, 60, 139),
'SlateGray' -> Color.new255(112, 128, 144),
'SlateGray1' -> Color.new255(198, 226, 255),
```

```
'SlateGray2' -> Color.new255(185, 211, 238),
'SlateGray3' -> Color.new255(159, 182, 205),
'SlateGray4' -> Color.new255(108, 123, 139),
'SlateGrey' -> Color.new255(112, 128, 144),
'snow' -> Color.new255(255, 250, 250),
'snow1' -> Color.new255(255, 250, 250),
'snow2' -> Color.new255(238, 233, 233),
'snow3' -> Color.new255(205, 201, 201),
'snow4' -> Color.new255(139, 137, 137),
'spring green' -> Color.new255(0, 255, 127),
'SpringGreen' -> Color.new255(0, 255, 127),
'SpringGreen1' -> Color.new255(0, 255, 127),
'SpringGreen2' -> Color.new255(0, 238, 118),
'SpringGreen3' -> Color.new255(0, 205, 102),
'SpringGreen4' -> Color.new255(0, 139, 69),
'steel blue' -> Color.new255(70, 130, 180),
'SteelBlue' -> Color.new255(70, 130, 180),
'SteelBlue1' -> Color.new255(99, 184, 255),
'SteelBlue2' -> Color.new255(92, 172, 238),
'SteelBlue3' -> Color.new255(79, 148, 205),
'SteelBlue4' -> Color.new255(54, 100, 139),
'tan' -> Color.new255(210, 180, 140),
'tan1' -> Color.new255(255, 165, 79),
'tan2' -> Color.new255(238, 154, 73),
'tan3' -> Color.new255(205, 133, 63),
'tan4' -> Color.new255(139, 90, 43),
'thistle' -> Color.new255(216, 191, 216),
'thistle1' -> Color.new255(255, 225, 255),
'thistle2' -> Color.new255(238, 210, 238),
'thistle3' -> Color.new255(205, 181, 205),
'thistle4' -> Color.new255(139, 123, 139),
'tomato' -> Color.new255(255, 99, 71),
'tomato1' -> Color.new255(255, 99, 71),
'tomato2' -> Color.new255(238, 92, 66),
'tomato3' -> Color.new255(205, 79, 57),
'tomato4' -> Color.new255(139, 54, 38),
'turquoise' -> Color.new255(64, 224, 208),
'turquoise1' -> Color.new255(0, 245, 255),
'turquoise2' -> Color.new255(0, 229, 238),
'turquoise3' -> Color.new255(0, 197, 205),
```

```
'turquoise4' -> Color.new255(0, 134, 139),
'violet' -> Color.new255(238, 130, 238),
'violet red' -> Color.new255(208, 32, 144),
'VioletRed' -> Color.new255(208, 32, 144),
'VioletRed1' -> Color.new255(255, 62, 150),
'VioletRed2' -> Color.new255(238, 58, 140),
'VioletRed3' -> Color.new255(205, 50, 120),
'VioletRed4' -> Color.new255(139, 34, 82),
'wheat' -> Color.new255(245, 222, 179),
'wheat1' -> Color.new255(255, 231, 186),
'wheat2' -> Color.new255(238, 216, 174),
'wheat3' -> Color.new255(205, 186, 150),
'wheat4' -> Color.new255(139, 126, 102),
'white' -> Color.new255(255, 255, 255),
'white smoke' -> Color.new255(245, 245, 245),
'WhiteSmoke' -> Color.new255(245, 245, 245),
'yellow' -> Color.new255(255, 255, 0),
'yellow green' -> Color.new255(154, 205, 50),
'yellow1' -> Color.new255(255, 255, 0),
'yellow2' -> Color.new255(238, 238, 0),
'yellow3' -> Color.new255(205, 205, 0),
'yellow4' -> Color.new255(139, 139, 0),
'YellowGreen' -> Color.new255(154, 205, 50)
```

ID: 199

Document

Document(title, text, isPostWindow);

```
Document     "this is the title"  "this is the text"
```

Document.open(path);

```
Document      "Help/Help.help.rtf"
```

Document.allDocuments
array where all open documents are stored.

Document.current
returns the current Document.

Document.hasEditedDocuments
returns true if there are unsaved changes in one of the open Document.

Document.closeAll(leavePostOpen)
by default the postWindow stays open.

Document.closeAllUnedited(leavePostOpen)
by default the postWindow stays open.

background_
set the background color of a Document

```
(
    Document "background", "'hardly see anything"
a.background_(Color.blue(alpha:0.8));
)
```

stringColor_
set the text color of a Document

```
(
    Document "background", "where are my glasses?"
```

```
a.background_(Color.red(alpha:0.8));
a.stringColor_(Color.red);
)
```

font_(font, rangestart, rangesize)
set font. if rangestart = -1 for the whole document

bounds_(Rect)
set bounds

close
close a Document

```
(
Task
var doc;
 doc = Document "background", "closing in 2 seconds"
doc.stringColor_(Color.red);
1.wait;
doc.background_(Color.red(alpha:0.8));
1.wait;
doc.close;
}).play(AppClock);

)
```

isEdited
returns true if a Document has unsaved changes unless it is the postWindow

```
(
Document.current.isEdited.postln;
)
```

syntaxColorize
same as command'

```
(
a = Document.allDocuments.at(0).syntaxColorize;
)
```

selectLine

```
(
Document.current.selectLine(1);
)
```

selectionStart

get the current position in the text

```
(
Document.current.selectionStart.postln;
)
```

selectionSize

get the current size of selection in the text

```
(
Document.current.selectionSize.postln;
)
```

selectRange(start, length)

```
(
Document.current.selectRange(Document                              );
)
```

string(rangestart, rangesize)

get the text of a document. If no rangestart is applied the whole text is returned.

```
(
Document.current.string;
)
```

selectedString

get the currently selected text.

```
(
var doc;
      Document.current;
doc.selectRange(doc.selectionStart-40, 10);
doc.selectedString.postln;
```

```
)
```

string_(string, rangestart, rangesize)

set the text of a document. if rangestart is -1 (default) the whole text is replaced

```
(
var doc;
      Document(string:"");
doc.string_("set a String")
)
```

selectedString_

insert text at the current position

```
(
var doc, txt;
doc = Document.current;
      "doc.postln; \n"
doc.selectedString_(txt);
)
```

front

```
(
Document.allDocuments.at(0).front;
)
```

keyDownAction_

register a keyDownAction. this is useful for macros

```
(
var doc, txt;
      Document.current;
doc.              ({arg doc, key, modifiers, num;
 [doc, key, modifiers].postln
});
)
(
Document.current.              (nil);
)
```

toFrontAction_

called when the window is clicked to front

```
example:
associate a proxyspace to two different windows.
(
s = Server.local;
s.boot;
q = ProxySpace
q.pop;
r = ProxySpace.push(r);
r.pop;
a = Document("proxy r", "//this is proxyspace r \n x =  out.play; \n  out = { SinOsc.ar([400, 500]*0.9,
0, 0.2) };");
a.background_(Color(0.8, 1.0, 1.0));

b = Document( "proxy q", "//this is proxyspace q \n x =  out.play; \n  out = { SinOsc.ar([1400, 1500]*0.9,
0, 0.2) };");
b.background_(Color(1.0, 1.0, 0.8));

b.toFrontAction_({
if(currentEnvironment == r,{r.pop});
q.push;
});
a.toFrontAction_({
if(currentEnvironment == q,{q.pop});
r.push;

});
)
(
//need to pop proxyspace from other examples
q.pop
r.pop
)
```

onClose_

register a close - action.

```
(
Document.current.onClose_({
```

```
var doc;
 doc = Document "before closing","did you call me?"

 Task

doc.stringColor_(Color.red);
0.1.wait;
doc.background_(Color.red(alpha:0.8));
0.3.wait;
doc.close;
}).play(AppClock);

})
)
```

mouseDownAction_

```
(

//add a mouse action to this document:
//example:  easy button:
//when you click in front of a 17 a SinOsc will start up;
Server.local.boot;
Document.current.mouseDownAction_({arg doc;
var char;
char = doc.rangeText(doc.selectionStart, 2);
if(char == "17",{
{EnvGen.kr(Env.perc, doneAction:2) * SinOsc.ar([600,720,300].choose, 0, 0.5)}.play;
});
if(char == "23",{
{EnvGen.kr(Env.perc, doneAction:2) * PinkNoise.ar(0.2)}.play;
});
})

)
```

test here and click in front of the number:

17

23

unfocusedFront_

```
(
Document.allDocuments.at(0).unfocusedFront
)



(

var doc;
doc = Document("", "| | ");
doc.background_(Color.blue(alpha:  1.0.rand));
Task
1000.do({
doc.setFont(size:  [7,8,9,24].choose);
0.08.wait;
})
}).play(AppClock);
Task
100.do({
1.01.wait;
doc.stringColor_([Color.red(alpha:  1.0.rand), Color.green(alpha:  1.0.rand)].choose);
})
}).play(AppClock);
Task
100.do({
1.01.wait;
doc.selectedString_(["\"\n#","| | ","-", "--"].choose);
})
}).play(AppClock);
Task({
var co, mul;
co = 0.1;
mul = 1.02;
100.do({
0.16.wait;
co = co * mul;
if(co > 0.99, { co = 0.1 });
doc.background_(Color.blue(alpha:  co));
});
```

```
doc.close;
}).play(AppClock)

)
```

//
Utilities and settings for dealing with documents such as super collider code files. By default the document directory is SuperCollider's application directory.

In Main-startUp you can set this to a more practical directory:

```
  Document       " /Documents/SuperCollider"
```

*standardizePath
if it is a relative path, expand it to an absolute path relative to your document directory. expand tildes in path (your home directory), resolve symbolic links (but not aliases). also converts from OS9 macintosh path format.

```
Document.standardizePath(
  ":Patches:newfoots:fastRuckAndTuck"
)
/Volumes/Macintosh HD/Users/cruxxial/Documents/SC3docs/Patches/newfoots/fastRuckAndTuck



Document.standardizePath(
  " /Documents/SC3docs/Patches/newfoots/fastRuckAndTuck"
)
Patches/newfoots/fastRuckAndTuck



Document.standardizePath(
  "Patches/newfoots/fastRuckAndTuck"
)
Patches/newfoots/fastRuckAndTuck
```

*abrevPath
reduce the path relative to your document directory if possible.

ID: 200

SuperCollider autocompletion v0.3

Usage:

To open a text window with the auto-complete feature enabled, execute the following in SuperCollider:

```
Document.autoComplete
```

(ac is a shortcut for Auto-complete, to make it easier to type.)

To open a file by pathname:

```
Document.openFileAutoComplete("myPath.rtf");
Document                      "*.sc"   // wildcards are supported
```

To bring up an open-file dialog:

```
Document.openAutoComplete
```

Autocompletion will be integrated more tightly into the code editor.

Summary:

While editing code in an auto-complete code window, the following keystrokes initiate special actions:

(– attempt to match the preceding identifier to method names containing that string, and display a list of methods with their defining classes. Making a selection will insert a method template into your document.

(will also match classnames, with the .new method: Rect(will show you a method template for Rect-*new.

. – attempt to match the preceding identifier to an exact class name, and present a list of class methods (not instance methods). Your selection will insert a method template into the document.

ctrl-. – attempt to match the preceding identifier to class names containing the identifier, and present a list of those class names. Your selection will open a class browser. You can navigate through the class tree to find the method you want, and press enter in the method list to insert a method template.

Shortcut in the class browser: type ^ in the method list to go to the superclass. This allows speedier location of methods inherited from superclasses.

Special behavior for ctrl-. – when you choose a method in a class browser, its class will be compared to the class you chose in the opening list. If the initial class responds to the method, the initial class will be put into the document; otherwise, the class from the class browser.

Feature description:

When you type a dot, SuperCollider will to check the previous text to see if it refers to a valid class. If so, a window will be presented with all the class methods (not instance methods) of the class.

So, for example, if you type:

```
SinOsc
```

the window will display the options:

ar(freq, phase, mul, add)
buildSynthDef()
buildSynthDef_()
....

If you type the first few letters into the text box, the list will reduce itself to the matching entries. If you type 'a', then the list will contain only:

ar(freq, phase, mul, add)

Press enter or return, and the method name with all its arguments will be added to your document, leaving the text:

```
SinOsc.ar(freq, phase, mul, add)
```

You can also click on the item you want in the list (or move through the list with the up and down arrow keys), and then press return.

Pressing escape or closing the window will cancel the auto-complete. Text typed into the text box prior to canceling will be added to the document–so, if you keep typing while the box comes up and you want to ignore it, your text will not be lost.

Similar behavior for method names: when you type an open parenthesis '(', SuperCollider will display a list of all classes that define this method. Type the first few letters of the class name (don't forget to capitalize) to choose the right one.

This treatment is necessary because variables in SuperCollider are not typed. If you enter 'func.value(', the text editor has no way to know what kind of object will be contained in func at the time of execution. So, it presents you with all possible options and allows you to choose.

New: The autocompleter now supports partial string matching for methods (triggered by typing open-paren) and classes (not by typing dot, but by typing ctrl-dot). In the case of classes, you will be given a list of classes matching the string typed. After you choose from the list, a full class browser will be opened. When you select a method and press enter, a method template will be dropped into the current document.

Because the class browser does not show methods defined by superclasses, you may press ^ to go to the superclass.

Further configuration:

Place the startup.rtf file in a folder called scwork in your home user directory. You can define class names and method names to be excluded from the browsers. I like to exclude the most common flow of control mechanisms (while, do, if, etc.).

Quirks and caveats:

The auto complete features will be lost from all documents when recompiling the class library.

Because of the way the document class works, the identifiers will not be extracted correctly if the cursor is at the very end of the document. You should leave a couple of empty lines below what you're typing for the feature to work.

The method browser does not handle inheritance. If you're browsing a method like 'add', you won't find Array in the list (but you will find its superclass ArrayedCollection).

Comments or questions to jamshark70@yahoo.com, please. No SPAM!

ID: 201

EZSlider wrapper class for label, slider, number box

EZSlider(window, dimensions, label, controlSpec, action, initVal, initAction, labelWidth, numberWidth)

EZSlider is a wrapper class for managing a label, slider and number box.

window - the SCWindow containing the views.
dimensions - a Point giving the width and height of the combined views.
label - a String
controlSpec - the ControlSpec for the value.
action - a function called when the value changes. The function is passed the EZSlider instance as its argument.
initVal - the value to initialize the slider and number box with. If nil, then it uses the ControlSpec's default value.
initAction - a Boolean indicating whether the action function should be called when setting the initial value. The default is false.
labelWidth - number of pixels width for the label. default is 80.
numberWidth - number of pixels width for the number box. default is 80.

The contained views can be accessed via the EZSlider instance variables:
labelView, **sliderView**, **numberView**

Another useful instance variable is **round**, the rounding precision for the number box display. The default value for **round** is 0.001 .

Example:

```
(
// start server
s = Server.internal;
Server.default = s;
s.boot;
)

(
// define a synth
```

```
SynthDef("window-test", { arg note = 36, fc = 1000, rq = 0.25, bal=0, amp=0.4, gate = 1;
var x;
x = Mix.fill(4, {
LFSaw.ar((note + {0.1.rand2}.dup).midicps, 0, 0.02)
});
x = RLPF.ar(x, fc, rq).softclip;
x = RLPF.ar(x, fc, rq, amp).softclip;
x = Balance2.ar(x[0], x[1], bal);
x = x * EnvGen.kr(Env.cutoff, gate, doneAction:  2);
Out.ar(0, x);
}, [0.1, 0.1, 0.1, 0.1, 0.1, 0]
).load(s);
)

(
var w, startButton, noteControl, cutoffControl, resonControl;
var balanceControl, ampControl;
var id, cmdPeriodFunc;

                    // generate a note id.

// make the window
w = SCWindow("another control panel", Rect(20, 400, 440, 180));
          // make window visible and front window.
w.view.decorator = FlowLayout(w.view.bounds);

w.view.background = HiliteGradient(Color.rand(0.0,1.0),Color.rand(0.0,1.0),
[\h,\v].choose, 100, rrand(0.1,0.9));

// add a button to start and stop the sound.
startButton = SCButton(w, 75 @ 24);
startButton.states = [
["Start", Color.black, Color.green],
["Stop", Color.white, Color.red]
];
startButton.action = {| view|
if (view.value == 1) {
   // start sound
s.sendMsg("/s_new", "window-test", id, 0, 0,
"note", noteControl.value,
```

```
"fc", cutoffControl.value,
"rq", resonControl.value,
"bal", balanceControl.value,
"amp", ampControl.value.dbamp);
};
if (view.value == 0) {
   // set gate to zero to cause envelope to release
s.sendMsg("/n_set", id, "gate", 0);
};
};

// create controls for all parameters
w.view.decorator.nextLine;
noteControl = EZSlider(w, 400 @ 24, "Note", ControlSpec(24, 60, \lin, 1),
{| ez|  s.sendMsg("/n_set", id, "note", ez.value); }, 36);

w.view.decorator.nextLine;
cutoffControl = EZSlider(w, 400 @ 24, "Cutoff", ControlSpec(200, 5000, \exp),
{| ez|  s.sendMsg("/n_set", id, "fc", ez.value); }, 1000);

w.view.decorator.nextLine;
resonControl = EZSlider(w, 400 @ 24, "Resonance", ControlSpec(0.1, 0.7),
{| ez|  s.sendMsg("/n_set", id, "rq", ez.value); }, 0.2);

w.view.decorator.nextLine;
balanceControl = EZSlider(w, 400 @ 24, "Balance", \bipolar,
{| ez|  s.sendMsg("/n_set", id, "bal", ez.value); }, 0);

w.view.decorator.nextLine;
ampControl = EZSlider(w, 400 @ 24, "Amp", \db,
{| ez|  s.sendMsg("/n_set", id, "amp", ez.value.dbamp); }, -6);


// set start button to zero upon a cmd-period
cmdPeriodFunc = { startButton.value = 0; };
CmdPeriod.add(cmdPeriodFunc);

// stop the sound when window closes and remove cmdPeriodFunc.
w.onClose = {
s.sendMsg("/n_free", id);
```

```
CmdPeriod.remove(cmdPeriodFunc);
};

)
```

ID: 202

Font

Font(name, size)

command-T to look for Font names.

SCStaticText, SCButton and their subclasses (SCNumberBox, SCDragView, SCDragSink, SCDragBoth) can set their fonts.

```
(
var w,f;

w = SCWindow("Fonts", Rect(128, 64, 340, 360));
w.view.decorator = f = FlowLayout(w.view.bounds,Point(4,4),Point(4,2));

[
"Helvetica-Bold",
"Helvetica",
"Monaco",
"Arial",
"Gadget",
"MarkerFelt-Thin "

].do({ arg name;
var v, s, n, spec, p, height = 16;

v = SCStaticText(w, Rect(0, 0, 56, height+2));
v.font = Font(name, 13);
v.string = name;

s = SCButton(w, Rect(0, 0, 140, height+2));
s.font = Font(name, 13);
s.states = [[name]];

n = SCNumberBox(w, Rect(0, 0, 56, height+2));
n.font = Font(name, 13);
n.object = pi;
```

```
f.nextLine;
});

w.front;

)

(
var w,f,i=0;

w = SCWindow("Fonts", Rect(128, 64, 800, 760));
w.view.decorator = f = FlowLayout(w.view.bounds,Point(4,4),Point(4,2));

Font.availableFonts.do({ arg name;
var v, s, n, spec, p, height = 16,font;
font = Font(name,13);

v = SCStaticText(w, Rect(0, 0, 56, height+2));
v.font = font;
v.string = name;

s = SCButton(w, Rect(0, 0, 140, height+2));
s.font = font;
s.states = [[name]];
s.action = { font.asCompileString.postln; };

n = SCNumberBox(w, Rect(0, 0, 56, height+2));
n.font = font;
n.object = pi;
if( (i = i + 1) % 3 == 0,{
f.nextLine;
});
});

w.front;

)
```

ID: 203

GUI Classes Overview

The following GUI classes have individual helpfiles. There are a number of undocumented GUI classes listed in **Undocumented-Classes**.

Color
Document
Font
SC2DSlider
SC2DTabletSlider
SCButton
SCCompositeView
SCEnvelopeView
SCFuncUserView
SCHLayoutView
SCMultiSliderView
SCNumberBox
SCPopUpMenu
SCRangeSlider
SCTabletView
SCTextField
SCVLayoutView
SCView
SCWindow
resize

ID: 204

GUI Classes Overview

The following GUI classes have individual helpfiles. There are a number of undocumented GUI classes listed in **Undocumented-Classes**.

Color
Document
Font
SC2DSlider
SC2DTabletSlider
SCButton
SCCompositeView
SCEnvelopeView
SCFuncUserView
SCHLayoutView
SCMultiSliderView
SCNumberBox
SCPopUpMenu
SCRangeSlider
SCTabletView
SCTextField
SCVLayoutView
SCView
SCWindow
resize

ID: 205

Pen draw on an SCWindow

superclass: Object

A class which allows you to draw on a **[SCWindow]**. It has no instance methods.

The following methods must be called within an SCWindow-drawHook or a SCUserView-drawFunc function, and will only be visible once the window or the view is refreshed. Each call to SCWindow-refresh SCUserView-refresh will 'overwrite' all previous drawing by executing the currently defined function.

See also: **[SCWindow]**, **[SCUserView]**, **[Color]**, and **[String]**

Drawing Methods

The following methods define paths. You will need to call ***stroke** or ***fill** to actually draw them.

***moveTo (point)**
Move the Pen to **point**. **point** is an instance of **[Point]**.

***lineTo (point)**
Draw a line (define a path) from the current position to **point**. **point** is an instance of **[Point]**.

***line (p1, p2)**
Draw a line (define a path) from p1 to p2. Current position will be p2. **p1** and **p2** are instances of **[Point]**.

// *curveTo(point, cpoint1, cpoint2)
draws an interpolated curve from the current position to **point**.
cpoint1, cpoint2 are help-points determining the curves curvature.
(Splines, B-Splines, Nurbs?)
Unfortunately not working for now...

// *quadCurveTo(point, cpoint1)
draws an interpolated curve from the current position to **point**.
cpoint1 is a help-point determining the curves curvature.

Unfortunately not working for now...

*addArc(center, radius, startAngle, arcAngle)

Draw an arc around the **[Point] center**, at **radius** number of pixels. **startAngle** and **arcAngle** refer to the starting angle and the extent of the arc, and are in radians [0..2pi].

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Pen.translate(100, 100);
10.do{
Color.red(rrand(0.0, 1), rrand(0.0, 0.5)).set;
Pen.addArc((100.rand)@(100.rand), rrand(10, 100), 2pi.rand, pi);
Pen.perform([\stroke, \fill].choose);
}
};
w.refresh;
)
```

*addWedge(center, radius, startAngle, arcAngle)

Draw a wedge around the **[Point] center**, at **radius** number of pixels. **startAngle** and **arcAngle** refer to the starting angle and the extent of the arc, and are in radians [0..2pi].

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Pen.translate(100, 100);
10.do{
Color.blue(rrand(0.0, 1), rrand(0.0, 0.5)).set;
Pen.addWedge((100.rand)@(100.rand), rrand(10, 100), 2pi.rand, 2pi.rand);
Pen.perform([\stroke, \fill].choose);
}
};
w.refresh;
)
```

*addAnnularWedge (center, innerRadius, outerRadius, startAngle, arcAngle)

Draw an annular wedge around the **[Point] center**, from **innerRadius** to **outerRadius**

in pixels. **startAngle** and **arcAngle** refer to the starting angle and the extent of the arc, and are in radians [0..2pi].

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Pen.translate(100, 100);
1000.do{
Color.green(rrand(0.0, 1), rrand(0.0, 0.5)).set;
Pen.addAnnularWedge(
(100.rand)@(100.rand),
rrand(10, 50),
rrand(51, 100),
2pi.rand,
2pi.rand
);
Pen.perform([\stroke, \fill].choose);
}
};
w.refresh;
)
```

// *addRect(rect)

adds a rectangle to the drawing;
Unfortunately not working for now...

*stroke

outline the previous defined path.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.red.set;
Pen.moveTo(200@100);

Pen.lineTo(250@200);
Pen.lineTo(300@200);
```

```
Pen.lineTo(200@250);
Pen.lineTo(100@200);
Pen.lineTo(150@200);
Pen.lineTo(200@100);

Pen.stroke
};
w.refresh;
)
```

*fill

fill the previous defined path.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.red.set;
Pen.moveTo(200@100);

Pen.lineTo(250@200);
Pen.lineTo(300@200);
Pen.lineTo(200@250);
Pen.lineTo(100@200);
Pen.lineTo(150@200);
Pen.lineTo(200@100);

Pen.fill
};
w.refresh;
)
```

These methods do require separate calls to ***stroke** or ***fill**.

*strokeRect(rect)

Strokes a rectangle into the window. **rect** is an instance of **[Rect]**.

```
(
w = SCWindow("strokeRect", Rect(128, 64, 360, 360));
w.drawHook = {
```

```
var h, v, r;
v = h = 300.0;
r = Rect(100, 100, 160, 80);
Color.black.alpha_(0.8).set;
Pen.strokeRect(r);
};
w.front;
)
```

*fillRect(rect)
Draws a filled rectangle into the window. **rect** is an instance of **[Rect]**.

*strokeOval(rect)
Strokes an oval into the window. **rect** is an instance of **[Rect]**.

```
(
w = SCWindow("strokeOval", Rect(128, 64, 360, 360));
w.drawHook = {
var h, v, r;
v = h = 300.0;
r = Rect(100, 100, 160, 80);
Color.black.alpha_(0.8).set;
Pen.strokeOval(r);
};
w.front;
)
```

*fillOval(rect)
Draws a filled oval into the window. **rect** is an instance of **[Rect]**.

// *drawAquaButton(rect, type=0, down=false, on=false)

Graphics State Methods

The following commands transform the graphics state, i.e. they effect all subsequent drawing commands. These transformations are cumulative, i.e. each command applies to the previous graphics state, *not* the original one.

*translate(x=0, y=0)
translate the coordinate system to have its origin moved by **x**,**y**

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.blue.set;
Pen.translate(200,100);

   // 0@0 is now 200@100
Pen.moveTo(0@0);

Pen.lineTo(50@100);
Pen.lineTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);

Pen.stroke
};
w.refresh;
)

  // cumulative translations
(
w = SCWindow.new.front;
w.view.background_(Color.clear);
w.drawHook = {
   // set the Color
Color.black.set;
   // draw 35 lines
35.do {
Pen.moveTo(0@0);
Pen.lineTo(50@350);
    // shift 10 to the right every time
Pen.translate(10, 0);
Pen.stroke
}
};
```

```
w.refresh;
)
```

*scale (x=0, y=0)

Scales subsequent drawing. **x** and **y** are scaling factors (i.e. 1 is normal, 2 is double size, etc.).

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.green.set;
Pen.translate(200,100);
Pen.scale(0.5, 2);
   // you have to set a starting point...
Pen.moveTo(0@0);

Pen.lineTo(50@100);
Pen.lineTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);

Pen.stroke
};
w.refresh;
)
```

*skew (x=0, y=0)

Skews subsequent drawing. **x** and **y** are skewing factors (i.e. 1 is normal).

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.green(0.5, 0.8).set;
Pen.translate(200,100);
Pen.skew(0.5, 0.2);
```

```
   // you have to set a starting point...
Pen.moveTo(0@0);

Pen.lineTo(50@100);
Pen.lineTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);

Pen.fill
};
w.refresh;
)
```

*rotate (angle=0, x=0, y=0)

Rotates subsequent drawing around the **Point** x@y by the amount **angle** in radians [0..2pi].

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
c = 0;
w.drawHook = {
Pen.translate(220, 200);

10.do({
Pen.translate(0,10);
    // set the Color for all "real" drawing
Color.hsv(c.fold(0, 1), 1, 1, 0.5).set;

    // you have to set a starting point...
Pen.moveTo(0@0);

Pen.lineTo(50@100);
Pen.lineTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);
```

```
Pen.fill;
Pen.rotate(0.2pi);

c = c + 0.1;
});
};
w.refresh;
)
```

*matrix_ (array)

transforms coordinate system.
array = [a, b, c, d, x, y]
a zoomX
b shearingX
c shearingY
d zoomY
x translateX
y translateY

```
(
var controlWindow, w;
var r, a, b, c, d, matrix = [1, 0, 0, 1, 10, 10];
var sliders, spex, name;

w = SCWindow.new.front;
w.view.background_(Color.white);

  // create a controller-window
controlWindow = SCWindow("matrix controls", Rect(400,200,350,120));
controlWindow.front;

  // determine the rectangle to be drawn
r = Rect.fromPoints(a = 0 @ 0, c = 180 @ 180);
b = r.leftBottom;
d = r.rightTop;

  // the drawHook
w.drawHook = {
Color.red.set;
Pen.matrix = matrix;
Pen.width = 5;
```

```
Pen.strokeRect(r);
Pen.strokeOval(r);
Color.blue.set;
Pen.width = 0.1;
Pen.line(a, c);
Pen.line(b, d);
Pen.stroke;

"A".drawAtPoint(a - 6, Font("Helvetica-Bold", 12));
"B".drawAtPoint(b - 6, Font("Helvetica-Bold", 12));
"C".drawAtPoint(c - (0@6), Font("Helvetica-Bold", 12));
"D".drawAtPoint(d - (0@6), Font("Helvetica-Bold", 12));

"a matrix test".drawInRect(r.moveBy(50,50),   Font("Helvetica", 10));
};

controlWindow.view.decorator = sliders = FlowLayout(controlWindow.view.bounds);
spex = [
[0, 2.0].asSpec,
[0, 2.0].asSpec,
[0, 2.0].asSpec,
[0, 2.0].asSpec,
[0, 200.0].asSpec,
[0, 200.0].asSpec
];
name = #[zoomX, shearingX, shearingY, zoomY, translateX, translateY];
6.do { | i|
EZSlider(controlWindow, 300 @ 14, name[i], spex[i], { | ez|  var val;
val = ez.value;
[i, val.round(10e-4)].postln;
matrix.put(i, val);
              w.refresh; // reevaluate drawHook function
}, matrix[i]);
sliders.nextLine;
};
w.refresh;
)
```

*width_(width=1)

sets the width of the Pen for the whole stroke

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.blue(0.5, 0.5).set;
Pen.translate(200,100);
Pen.width = 10;
   // you have to set a starting point...
Pen.moveTo(0@0);

Pen.lineTo(50@100);
Pen.lineTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);

Pen.stroke
};
w.refresh;
)
```

*use (function)

Draw **function**, and then revert to the previous graphics state. This allows you to make complex transformations of the graphics state without having to explicitly revert to get back to 'normal'.

```
(
  // modified by an example of Stefan Wittwer
  w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   //paint origin
Color.gray(0, 0.5).set;
Pen.addArc(0@0, 20, 0, 2pi);
Pen.fill;
Pen.width = 10;
```

```
   Pen       // draw something complex...
Pen.width = 0.5;
Pen.translate(100,100);
Color.blue.set;
Pen.addArc(0@0, 10, 0, 2pi);
Pen.fill;
20.do{
Pen.moveTo(0@0);
Pen.lineTo(100@0);
Color.red(0.8, rrand(0.7, 1)).set;
Pen.stroke;
Pen.skew(0, 0.1);
};
};

   // now go on with all params as before
   // translation, skewing, width, and color modifications do not apply
Pen.line(10@120, 300@120);
Pen.stroke;
};
w.refresh
)
```

*path(function)

make a path, consisting of the drawing made in **function.**
Unfortunately not working for now...
(there's no Meta_Pen-endPath which currently is used in this method)

*beginPath

Discard any previous path.

```
        // incomplete arrow
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
   // set the Color
Color.blue.set;
Pen.translate(200,100);
Pen.moveTo(0@0);
```

```
Pen.lineTo(50@100);
Pen.lineTo(100@100);

   // forget what we just drew
Pen.beginPath;
Pen.moveTo(100@100);
Pen.lineTo(0@150);
Pen.lineTo(-100@100);
Pen.lineTo(-50@100);
Pen.lineTo(0@0);

Pen.stroke
};
w.refresh;
)
```

*clip

Use the previously defined path as a clipping path.

```
(
w = SCWindow.new.front;
w.view.background_(Color.white);
w.drawHook = {
         // outline the clipping path
Pen.moveTo(110@110);
Pen.lineTo(290@110);
Pen.lineTo(290@240);
Pen.lineTo(110@240);
Pen.lineTo(110@110);
   // now clip
Pen.clip;

   // everything else we draw is now clipped
Color.yellow.set;
Pen.fillRect(Rect(0,0,400,400));
Color.red.set;
Pen.moveTo(200@100);

Pen.lineTo(250@200);
```

```
Pen.lineTo(300@200);
Pen.lineTo(200@250);
Pen.lineTo(100@200);
Pen.lineTo(150@200);

Pen.fill;
};
w.refresh;
)
```

Examples

```
(
// simple rotating and scaling
w = SCWindow("Pen Rotation and Scaling", Rect(128, 64, 360, 360));
w.drawHook = {
var h, v;
v = h = 300.0;
Pen.use {
   // use the same rect for everything, just scale and rotate
var r = Rect(0,0,200,80);
Color.black.set;
   // offset all subsequent co-ordinates
Pen.translate(80,20);
Pen.fillRect(r);
Color.red.set;
   // scale all subsequent co-ordinates
Pen.scale(0.8, 0.8);
Pen.translate(8,10);
   // rotate all subsequent co-ordinates
Pen.rotate(0.1pi);
Pen.fillRect(r);
Color.blue.set;
   // lather, rinse, repeat
Pen.scale(0.8, 0.8);
Pen.rotate(0.1pi);
Pen.width = 3;
Pen.strokeRect(r);
Color.yellow(1,0.5).set;
```

```
Pen.scale(0.8, 0.8);
Pen.rotate(0.1pi);
Pen.translate(20,-20);
Pen.fillOval(r);
}
};

w.front;
)

// redraw at random interval
// different every time
(
var w, run = true;
w = SCWindow("my name is...  panel", Rect(128, 64, 800, 800));
w.view.background = Color.white;
w.onClose = { run = false; };
w.front;
w.drawHook = {
Pen.use {
Pen.width = 0.2;
400.do {
Pen.beginPath;
Pen.moveTo(Point(10.rand * 80 + 40, 10.rand * 80 + 40));
Pen.lineTo(Point(10.rand * 80 + 40, 10.rand * 80 + 40));
Pen.stroke;
};
};
};

{ while { run } { w.refresh; 1.0.rand.wait } }.fork(AppClock)

)

(
var w, run = true;
w = SCWindow("my name is...  panel", Rect(128, 64, 800, 500));
w.view.background = Color.white;
w.onClose = { run = false; };
w.front;
```

```
w.drawHook = {
Pen.use {
Pen.width = 2;
80.do {
Pen.width = rrand(0,4) + 0.5;
Pen.beginPath;
Pen.moveTo(Point(800.rand, 500.rand));
Pen.lineTo(Point(800.rand, 500.rand));
Pen.stroke;
};
};
};

{ while { run } { w.refresh; 1.0.rand.wait } }.fork(AppClock)

)

// Animation

// Uses random seed to 'store' data
// By reseting the seed each time the same random values and shapes are generated for each 'frame'
// These can then be subjected to cumulative rotation, etc., by simply incrementing the phase var.
(
// By James McCartney
var w, h = 700, v = 700, seed, run = true, phase = 0;
w = SCWindow("wedge", Rect(40, 40, h, v), false);
w.view.background = Color.rand(0,0.3);
                    false   // stop the thread on close
w.front;
// store an initial seed value for the random generator
seed = Date.seed;
w.drawHook = {
Pen.width = 2;
Pen.use {
  // reset this thread's seed for a moment
thisThread.randSeed = Date.seed;
  // now a slight chance of a new seed or background color
if (0.006.coin) { seed = Date.seed; };
if (0.02.coin) { w.view.background = Color.rand(0,0.3); };
  // either revert to the stored seed or set the new one
```

```
thisThread.randSeed = seed;
  // the random values below will be the same each time if the seed has not changed
  // only the phase value has advanced
Pen.translate(h/2, v/2);
  // rotate the whole image
  // negative random values rotate one direction, positive the other
Pen.rotate(phase * 1.0.rand2);
  // scale the rotated y axis in a sine pattern
Pen.scale(1, 0.3 * sin(phase * 1.0.rand2 + 2pi.rand) + 0.5 );
  // create a random number of annular wedges
rrand(6,24).do {
Color.rand(0.0,1.0).alpha_(rrand(0.1,0.7)).set;
Pen.beginPath;
Pen.addAnnularWedge(Point(0,0), a = rrand(60,300), a + 50.rand2, 2pi.rand
+ (phase * 2.0.rand2), 2pi.rand);
if (0.5.coin) {Pen.stroke}{Pen.fill};
};
};
};

// fork a thread to update 20 times a second, and advance the phase each time
{ while { run } { w.refresh; 0.05.wait; phase = phase + 0.01pi;} }.fork(AppClock)


)


(
var w, phase = 0, seed = Date.seed, run = true;
w = SCWindow("my name is...  panel", Rect(128, 64, 800, 800));
w.view.background = Color.blue(0.4);
w.onClose = { run = false; };
w.front;
w.drawHook = {
Pen.use {
if (0.02.coin) { seed = Date.seed; };
thisThread.randSeed = seed;
Color.white.set;
200.do {
var a = 4.rand;
var b = 24.rand;
var r1 = 230 + (50 * a);
```

```
var a1 = 2pi / 24 * b + phase;
var r2 = 230 + (50 * (a + 1.rand2).fold(0,3));
var a2 = 2pi / 24 * (b + (3.rand2)).wrap(0,23) + phase;
Pen.width = 0.2 + 1.0.linrand;
Pen.beginPath;
Pen.moveTo(Polar(r1, a1).asPoint + Point(400,400));
Pen.lineTo(Polar(r2, a2).asPoint + Point(400,400));
Pen.stroke;
};
thisThread.randSeed = Date.seed;
40.do {
var a = 4.rand;
var b = 24.rand;
var r1 = 230 + (50 * a);
var a1 = 2pi / 24 * b + phase;
var r2 = 230 + (50 * (a + 1.rand2).fold(0,3));
var a2 = 2pi / 24 * (b + (3.rand2)).wrap(0,23) + phase;
Pen.width = 0.2 + 1.5.linrand;
Pen.beginPath;
Pen.moveTo(Polar(r1, a1).asPoint + Point(400,400));
Pen.lineTo(Polar(r2, a2).asPoint + Point(400,400));
Pen.stroke;
};
};
};

{ while { run } { w.refresh; 0.1.wait; phase = phase + (2pi/(20*24)) } }.fork(AppClock)

)


(
var w, h = 800, v = 600, seed = Date.seed, phase = 0, zoom = 0.7, zoomf = 1, run = true;
w = SCWindow("affines", Rect(40, 40, h, v));
w.view.background = Color.blue(0.4);
w.onClose = { run = false };
w.front;
w.drawHook = {
thisThread.randSeed = Date.seed;
if (0.0125.coin) { seed = Date.seed; phase = 0; zoom = 0.7; zoomf = exprand(1/1.01, 1.01); }
```

```
{ phase = phase + (2pi/80); zoom = zoom * zoomf; };
thisThread.randSeed = seed;
Pen.use {
var p1 = Point(20.rand2 + (h/2), 20.rand2 + (v/2));
var p2 = Point(20.rand2 + (h/2), 20.rand2 + (v/2));
var xscales = { exprand(2** -0.1, 2**0.1) } !  2;
var yscales = { exprand(2** -0.1, 2**0.1) } !  2;
var xlates = { 8.rand2 } !  2;
var ylates = { 8.rand2 } !  2;
var rots = { 2pi.rand + phase } !  2;
var xform;
xscales = (xscales ++ (1/xscales)) * 1;
yscales = (yscales ++ (1/yscales)) * 1;
xlates = xlates ++ xlates.neg;
ylates = ylates ++ xlates.neg;
rots = rots ++ rots.neg;
xform = {| i|  [xlates[i], ylates[i], rots[i], xscales[i], yscales[i]] } !  4;
Color.grey(1,0.5).set;
Pen.width = 8.linrand + 1;
Pen.translate(400, 400);
Pen.scale(zoom, zoom);
Pen.translate(-400, -400);
1200.do {
var p, rot, xlate, ylate, xscale, yscale;
Pen.width = 8.linrand + 1;
Pen.beginPath;
#rot, xlate, ylate, xscale, yscale = xform.choose;
Pen.translate(xlate, ylate);
Pen.rotate(rot, h/2, v/2);
Pen.scale(xscale, yscale);
Pen.moveTo(p1);
Pen.lineTo(p2);
Pen.stroke;
};
};
};

{ while { run } { w.refresh; 0.05.wait; } }.fork(AppClock)


)
```

—-

NodeBox vs. SC3 (modified from a mailinglist-post of James McCartney)

—-

rect() Pen.strokeRect, Pen.fillRect
oval() Pen.strokeOval, Pen.fillOval
line() Pen.line – or use Pen.moveTo, Pen.lineTo
arrow()
star()

beginpath() Pen.beginPath
moveto() Pen.moveTo
lineto() Pen.lineTo
curveto() not now
endpath() Pen.stroke, Pen.fill
(paths don't need to be stored as data because you can compose them functionally).
drawpath() Pen.stroke, Pen.fill
beginclip()
endclip()

transform() – not needed since rotate lets you specify the center point.
translate() Pen.translate
rotate() Pen.rotate
scale() Pen.scale
skew() Pen.skew
Pen // private method???
Pen // private method???
reset() Pen.matrix = [0,0,0,0,0,0]

colormode() not necessary use hsv or rgb as needed. missing CMYK though. easy to add.
color() Color.hsv(h,s,v) or Color(r,g,b)
fill() color.setFill
nofill() use Pen.stroke or Pen.fill as needed.
stroke() color.setStroke
nostroke() use Pen.stroke or Pen.fill as needed.
strokewidth() Pen.width

font() Font(name, size)
fontsize() Font(name, size)
text() string.drawAtPoint
textpath()
textwidth() string.bounds – currently commented out but should work once reenabled.
textheight() string.bounds
textmetrics()
lineheight()
align() use string.drawCenteredIn, string.drawLeftJustIn, string.drawRightJustIn

image() not yet
imagesize() not yet

size() – all of these are covered by other mechanisms in SC
var()
random()
choice()
grid()
open()
files()
autotext()

ID: 206

resize

resize behavior for SCView

```
1  2  3
4  5  6
7  8  9

1 - fixed to left, fixed to top
2 - horizontally elastic, fixed to top
3 - fixed to right, fixed to top

4 - fixed to left, vertically elastic
5 - horizontally elastic, vertically elastic
6 - fixed to right, vertically elastic

7 - fixed to left, fixed to bottom
8 - horizontally elastic, fixed to bottom
9 - fixed to right, fixed to bottom
```

ID: 207

SC2DSlider

superclass: SCSliderBase

```
(
var window;
var slider;

window = SCWindow("SC2DSlider", Rect(100,100, 140 ,140));
window.front;

slider = SC2DSlider(window, Rect(20, 20,80, 80))
.x_(0.5).y_(1);

)
```

<>x
<>y

drag and drop returns and accepts Points.
hold command key to initiate a drag.

ID: 208

SC2DTabletSlider

superclass: SC2DSlider

a 2D slider with support for extended wacom data

```
(
var window;
var slider;

window = SCWindow("SC2DSlider", Rect(100,100, 140 ,140));
window.front;

slider = SC2DTabletSlider(window, Rect(20, 20,80, 80))
.x_(0.5).y_(1);
slider.mouseDownAction = { arg view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount;
["down",view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount].postln;
};
slider.action = { arg view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount;
[view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount].postln;
};
slider.mouseUpAction = { arg view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount;
["up",view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount].postln;
};
slider.setProperty(\clipInBounds,0)
)
```

<>x 0..1 value
<>y 0..1 value

action
mouseDownAction
mouseUpAction

all actions are passed the following wacom tablet values:
view - the view
x - 0..1 value

y - 0..1 value
pressure - 0..1
tiltX - 0..1 where available
tiltY - 0..1 where available
deviceID - will be used to look up if the tip or the eraser is used
buttonNumber - 0 left, 1 right, 2 middle wheel click
clickCount - double click, triple click ...
most relevant for the mouseDown, but still valid for the dragged and mouseUp
absoluteZ - the wheel on the side of some mice
rotation - in degrees, only on the 4d mice

Properties

clipInBounds- 0 or 1
by default the x/y values are clipped to 0..1
by setting this to 0, the values will exceed this as you drag from
inside the view to outside. This is useful in that you can have a small view
in which to start your movement and then go all over as long as you don't lift the pen.

drag and drop returns and accepts Points.
hold command key to initiate a drag.

```
(
SynthDef("help-SC2DTabletSlider",  arg
ffreqInterval=0,rq=0.4,gate=0.0;
var p,c,d,f;
c=LFNoise1.kr(0.1,0.45,0.55);
d=LFNoise1.kr(0.1,0.45,0.55);
f=LFNoise1.kr(0.1,2);
p=Pulse.ar([ freq  * int1.midiratio + f , freq, freq * int2.midiratio - f],
[c,d,c],0.2);
Out.ar(0,
RLPF.ar(Mix.ar(p),freq * ffreqInterval.midiratio,rq)
* EnvGen.kr(Env.adsr, gate, gate)
)
},[0.1,0.1,0.1,0.1,0.1,nil]).send(s);
)
```

```
(
var w, v,freq,int,synth;
 synth = Synth("help-SC2DTabletSlider");
w = SCWindow.new.front;

freq = ControlSpec(100,3000,\exp);
int = ControlSpec(-48,48,\linear,1);

v = SCTabletView(w,Rect(10,10,200,200));
v.background = Color.white;
v.action = { arg view,x,y,pressure,tiltx,tilty;
synth.set(
\int1, int.map(x),
\int2, int.map(y),
\ffreqInterval, int.map(pressure),
\gate, pressure.postln
);
};

v.mouseDownAction = { arg view,x,y,pressure;
synth.set(
\freq , rrand(30,80).midicps,
\gate, pressure.postln
)
};
v.mouseUpAction = { arg view,x,y,pressure;
synth.set( \gate, 0.postln )
};
)
```

ID: 209

SCButton

```
each state:
  [ name, text color, background color ]

(

w = SCWindow.new;
w.front;

b = SCButton(w,Rect(20,20,340,30));
b.states = [
["suffering",Color.black,Color.red],
   ["origins of suffering" Color       Color
   ["cessation of creating suffering" Color     Color
   ["the path to cessation of creating suffering"
Color.blue,Color.clear]
];
b.action = { arg butt;
butt.value.postln;
};
)
```

Failure to set any states at all results in an invisible button.

```
// does not do action
b.value = 2;

// does action if it results in a change of value
b.valueAction =  3;

// clips to size of states
b.valueAction =  -1;

// floats no problem
b.valueAction =  3.3;
```

```
(

w = SCWindow.new;
w.front;

b = SCButton(w,Rect(20,20,340,30));
b.states = [
["suffering",Color.black,Color.red]
];


// new state doesn't take effect until ...
b.states = [
   ["cessation of suffering" Color     Color
];
//window is refreshed
w.refresh;

//or the view itself is refreshed
b.refresh;

)
```

ID: 210

SCCompositeView

A view that contains other views.

grouping by background color

```
(
w = SCWindow.new;

c = SCCompositeView(w,Rect(0,0,300,300));

a = SC2DSlider(c,Rect(0,0,100,100));
b = SC2DSlider(c,Rect(100,100,100,100));

c.background = Gradient(Color.rand,Color.rand);

w.front;
)
```

Coordinates are the same as that for the window, not relative to the origin of the composite view (as in other gui frameworks).

```
(
w = SCWindow.new;

c = SCCompositeView(w,Rect(50,0,300,300));

a = SC2DSlider(c,Rect(0,0,100,100));
b = SC2DSlider(c,Rect(100,100,100,100));

c.background = Gradient(Color.rand,Color.rand);

w.front;
)
```

keydown bubbling

Note that the keyDown action is assigned to the composite view. If c and d do not have keyDown actions themselves, the event is passed to b, the parent.

```
(
w = SCWindow.new;

c = SCCompositeView(w,Rect(0,0,500,500));

a = SC2DSlider(c,Rect(0,0,100,100));
b = SC2DSlider(c,Rect(100,100,100,100));

w.front;

c.keyDownAction = {
 "keydown bubbled up to me"
};

//d is on window w,  not on composite view c
d = SC2DSlider(w,Rect(200,200,100,100));
)
```

click on the different views and hit keys on the keyboard.

decorators

a 'decorator' object can be set to handle layout management. all views added to the composite view will now be placed by the decorator.

```
(
a = SCWindow.new;

b = SCCompositeView(a,Rect(0,0,500,500));
b.decorator = FlowLayout(b.bounds);

// adding views to b automatically use the decorator
// no need to use parent.decorator.place
c = SC2DSlider(b,Rect(0,0,100,100)); // size matters
d = SC2DSlider(b,Rect(0,0,100,100)); // origin doesn't

a.front;
)
```

hiding / swapping

```
(
a = SCWindow.new;
q = 3;

e = SCButton(a,Rect(0,0,160,20));

e.states = Array.fill(q,{ arg i;
[i.asString,Color.black,Color.white]
});

e.action = { arg butt;
p.visible = false;
p = c.at(butt.value);
p.visible = true;
};

c = Array.fill(q,{ arg i;
b = SCCompositeView(a,Rect(0,25,300,300));
b.decorator = FlowLayout(b.bounds);
c = SC2DSlider(b,Rect(0,0,100,100));
c.x = 1.0.rand;
d = SC2DSlider(b,Rect(0,0,100,100));
d.y = 1.0.rand;
b.visible = false;
b
});

p = c.at(0); // previous
            true  // show first one

a.front;

)
```

removing

```
(
```

```
w = SCWindow.new;
c = SCCompositeView(w,Rect(0,0,300,300));
a = SC2DSlider(c,Rect(0,0,100,100));
b = SC2DSlider(c,Rect(100,100,100,100));
c.background = Gradient(Color.rand,Color.rand);
w.front;
)


a.remove;
c.refresh;
```

resize contraints

resize the window to see how the contents behave

```
(
w = SCWindow.new;

c = SCCompositeView(w,Rect(0,0,300,300));
c.background = Gradient(Color.rand,Color.rand);

c.resize = 5; // elastic

a = SC2DSlider(c,Rect(0,0,100,100));
a.resize = 1; // fixed

b = SC2DSlider(c,Rect(100,100,100,100));
b.resize = 2; // x elastic
b.setProperty(\minWidth,30); // up to a point
b.setProperty(\maxWidth,200);
w.front;

)
(bug:  composite view should get limited by it's contents' limitations)
```

ID: 211

SCEnvelopeView

value_([times,values])
where times and values are all 0..1

value
[times,values]
where times and values are all 0..1

action
function is passed the view

index
the current or last moved node

drawRect_(boolean)
set wether to show the points or not

drawLines_(boolean)
draw lines between the point

setThumbSize(index, size)
set the size of a point for the specified index, if the index is -1 set the size for all points

thumbSize_(size)
set the size of all points

setThumbWidth(index, width)
set the width of a point for the specified index, if the index is -1 set the width for all points

thumbWidth_(width)
set the width of all points

setThumbHeight(index, heigth)
set the height of a point for the specified index, if the index is -1 set the height for all

points

thumbHeight_(heigth)
set the height of all points

setEditable(index, boolean)
makes a specified point unmovable

editable_(boolean)
makes all points unmovable

selectionColor_(color)
set the color of the point when selected

setFillColor(index, color)
set the point color

fillColor_(color)
set the color for all points

setString(index, string)
give a point a string

connect_(index, arrayofpoints)
connect a point to others and do not use the standart connection scheme

```
use as envelope view
(
//use shift-click to keep a node selected
a = SCWindow("envelope", Rect(200 , 450, 250, 100));
a.view.decorator =  FlowLayout(a.view.bounds);

b = SCEnvelopeView(a, Rect(0, 0, 230, 80))
.drawLines_(true)
.selectionColor_(Color.red)
.drawRects_(true)
.resize_(5)
```

```
.action_({arg b; [b.index,b.value].postln})
 .thumbSize_
.value_([[0.0, 0.1, 0.5, 1.0],[0.1,1.0,0.8,0.0]]);

a.front;
)
(
//make the first point unmoveable
b.setEditable(0,false);
)

(
//use shift click to select/unselect the points
a = SCWindow("test", Rect(200 , 450, 450, 150));
a.view.decorator =  FlowLayout(a.view.bounds);

b = SCEnvelopeView(a, Rect(0, 0, 350, 100))
.thumbSize_(5)
.drawLines_(true)
.fillColor_(Color.green)
.selectionColor_(Color.red)
.drawRects_(true)
.value_([(0.0, 0.1 ..  1.0), (0.0, 0.1 ..  1.0)])
.setEditable(0,false);

a.front;
)
(
    Routine
var j = 0;
20.do({ arg i;
b.select((b.size -1).rand.abs);

0.1.wait;
b.x_(1.0.rand.abs);
b.y_(1.0.rand.abs);
});
b.select(-1);

});
```

```
AppClock.play(r);
)

c = b.xvalues;




//show boxes with a string in it:
(
a = SCWindow("text-boxes", Rect(200 , 450, 450, 450));
a.view.decorator =  FlowLayout(a.view.bounds);

b = SCEnvelopeView(a, Rect(0, 0, 440, 440))
 .thumbWidth
 .thumbHeight
.drawLines_(true)
.drawRects_(true)
.selectionColor_(Color.red)
.value_([[0.1, 0.4, 0.5, 0.3], [0.1, 0.2, 0.9, 0.7]]);
//b.setStatic(0,true);
4.do({arg i;
b.setString(i, ["this", "is", "so much", "fun"].at(i));
b.setFillColor(i,[Color.yellow, Color.white, Color.green].choose);
});
a.front;
)



the text objects can be connected:
(
b.connect(3, [2.0,0.0,1.0]);
b.connect(0,[2.0,3.0,1.0]);
b.drawLines_(true);
)
```

ID: 212

SCFuncUserView

not working yet

```
(
a = SCWindow.new;

b = SCFuncUserView(a,Rect.new(0,0,100,100));

b.drawFunc = { };
a.front;

)
```

ID: 213

SCHLayoutView

```
(
q = 10;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));

Array.fill(q,{ arg i;
SCSlider(h,Rect(0,0,20,75)).value_(i / q)
});

w.front
)
```

elastic

```
(
q = 10;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.background = Color.red(alpha:0.1);
h.resize = 5; //   elastic
Array.fill(q,{  arg i;
var s;
s = SCSlider(h,Rect(0,0,20,75));
s.value = i / q;
s
});

w.front
)

Contents are elastic
(
```

```
q = 10;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.resize = 5; //   elastic
Array.fill(q,{  arg i;
var s;
s = SCSlider(h,Rect(0,0,20,75));
s.resize = 5; // elastic
s.value = i / q;
s
});

w.front
)

set minWidth on contents
(
q = 5;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.background = Color.red(alpha:0.2);
h.resize = 5; //   elastic

Array.fill(q,{  arg i;
var s;
s = SCSlider(h,Rect(0,0,20,75));
s.value = i / 5;
if(i < 2,{
s.resize = 5; // some elastic
s.setProperty(\minWidth,20);
},{
  s.resize = 1; // some not elastic
});
s
});

w.front
)
```

```
(
q = 5;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.resize = 5; //   elastic

Array.fill(q,{  arg i;
var s;
s = SCSlider(h,Rect(0,0,20,75));

s.value = i / 5;
s.resize = 5;
s.setProperty(\minWidth,20);
s.setProperty(\maxWidth,40);
s
});

w.front
)
```

Text flows

```
(
q = 5;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.resize = 5; //   elastic

Array.fill(q,{  arg i;
var s;
s =  SCStaticText(h,120@20).string_("abcdefg");

s.resize = 5;
s.setProperty(\minWidth,10);
s.setProperty(\maxWidth,80);

// not working
s.setProperty(\maxHeight,10);
```

```
s.setProperty(\minHeight,10);

s.background = Color.white;
s
});

w.front
)
```

spacing

```
(
q = 10;
w = SCWindow.new;

h = SCHLayoutView(w,Rect(0,0,300,300));
h.setProperty(\spacing,0);

Array.fill(q,{
SCSlider(h,Rect(0,0,20,75))
});

w.front
)
```

ID: 214

SCListView

```
(

w = SCWindow.new.front;
l = [
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative",
  "absolute" "relative"
];
v = SCListView(w,Rect(10,10,180,50));
v.items = l;
v.background_(Color.white);
v.action = { arg sbs;
[sbs.value, l.at(sbs.value)].postln; // .value returns the integer
};

)

v.value = 16.rand
```

ID: 215

SCMovieView

can play movies such as .mov and mpg,
and image files like jpg, png, tiff and others.
(currently, it gets stuck on picts.)
This is basically a wrapper for a Cocoa Quicktime view.

```
(
w = SCWindow("mov").front;
b = SCButton(w, Rect(0, 0, 150, 20))
.states_([["pick a file"]])
.action_({ File.openDialog("", { | path|  m.path_(path) }) });
m = SCMovieView(w, Rect(0,20,360, 260));
)

 // random-pick a tiff from the Help folder
m.path_("Help/*/*/*.tiff".pathMatch.choose);

 // or point it to a movie (you may have that one too):
        "/Library/Application\Support/iDVD/Tutorial/Movies/Our\First\Snowman.mov"


m.start;   // playback
m.muted_(true); // thank god
m.stop;

 //rate
m.rate_(0.2);
 // backwards
m.gotoEnd.rate_(-1).start;

 // select a range on the controller and play it
m.rate_(1).playSelectionOnly_(true).start;

 // loopModes:
                // only one direction
```

```
m.loopMode_(0).start; // back and forth



m.stop;
m.gotoBeginning;

 // single steps
m.stepForward;

10.do { m.stepForward; };
m.stepBack;

// select with shift-drag, copy paste between movieviews or quicktime player
m.editable_(true);


m.showControllerAndAdjustSize(true, true);
 // resize compared to image size:
m.resizeWithMagnification(0.75);

//goto time (in seconds)
m.currentTime_(1);

  // not there yet, but would be nice to have:
    // startFrame, length
m.setSelection_(20, 15);


m.frame_(frame); // jump to frame
                // poll current frame pos
```

ID: 216

SCMultiSliderView

isFilled = true

looks like a candlestick graph

```
(
//use as table
var size;
size = 350 / 6;
a = SCWindow("test", Rect(200 , 450, 450, 150));
a.view.decorator =  FlowLayout(a.view.bounds);
b = SCMultiSliderView(a, Rect(0, 0, 350, 100));

c = Array.new;
size.do({arg i;
c = c.add(0.01);
});
c = c.reverse;
b.value_(c);

b.isFilled_(true);

// width in pixels of each stick
b.indexThumbSize_(2.0);
// spacing on the value axis
b.gap_(4);

a.front;
)
```

flip by 90 degree

```
(
b.indexIsHorizontal_(false);

a.bounds_(Rect(200 , 450, 150, 430));
b.bounds_( Rect(10, 0, 100, 390));
```

```
b.background_(Color.black);
b.strokeColor_(Color.white);
b.fillColor_(Color.white);

b.gap = 1;

a.front;
)
```

isFilled = false

individual squares for each point

```
(
//use as multislider
var size;
size = 12;
a = SCWindow("test", Rect(200 , 450, 10 + (size * 17), 10 + (size * 17)));
a.view.decorator =  FlowLayout(a.view.bounds);
b = SCMultiSliderView(a, Rect(0, 0, size * 17, size * 17));
b.action = {arg xb; ("index:  " ++ xb.index ++" value:  " ++ xb.currentvalue).postln};

c = Array.new;
size.do({arg i;
c = c.add(i/size);
});
b.value_(c);

b.isFilled = false;

b.xOffset_(5);
b.thumbSize_(12.0);

// value axis size of each blip in pixels
b.valueThumbSize_(15.0);
// index axis size of each blip in pixels
b.indexThumbSize_( b.bounds.width / c.size );
b.gap = 0;

b.strokeColor_(Color.blue);
b.fillColor_(Color.blue);
```

```
a.front;
)
```

read only mode

```
(
b.readOnly = true;
// show an area as selected, used like a cursor
b.showIndex = true;
// move the selection index
b.index = 4;
// 1 item wide
b.selectionSize = 1;

)

b.index;
b.selectionSize;

(
//use it as sequencer
b.setProperty(\showIndex, true);

    Routine
var j = 0;
20.do({ arg i;
0.1.wait;
b.index_(j);
if (j < 11 ,{j = j + 1},{j = 0});
});
0.1.wait;
20.do({ arg i;
[0.1,0.2].choose.wait;
b.index_(b.size.rand);
});
});
AppClock.play(r);

)
```

Note: this forces the entire view to redraw at each step and will spend a lot of CPU.

drawLines

```
(
//use as multislider II with lines
var size;
size = 12;
a = SCWindow("test", Rect(200 , 450, 450, 150));
a.view.decorator =  FlowLayout(a.view.bounds);
b = SCMultiSliderView(a, Rect(0, 0, size * 17, 50));
a.view.decorator.nextLine;
//e = SCDragBoth(a , Rect(0, 0, size * 17, 50));
e = SCMultiSliderView(a, Rect(0, 0, size * 17, 50));
c = Array.new;
size.do({arg i;
c = c.add(i/size);
});
b.value_(c);

b.xOffset_(18);
b.thumbSize_(1);
b.strokeColor_(Color.blue);
b.drawLines_(true);
b.drawRects_(true);
b.indexThumbSize_(1);
b.valueThumbSize_(1);

a.front;
)
c = Array.newClear(12);
b.getProperty(\referenceValues, Array.newClear(12));
c.size;



(
//press shift to extend the selection
//use as waveView:  scrubbing over the view returns index
//if showIndex(false) the view is not refreshed (faster);
//otherwise you can make a selection with shift - drag.
var size, file, maxval, minval;
```

```
size = 640;
a = SCWindow("test", Rect(200 , 140, 650, 150));
a.view.decorator =  FlowLayout(a.view.bounds);
b = SCMultiSliderView(a, Rect(0, 0, size, 50));
b.readOnly_(true);
a.view.decorator.nextLine;



d = Array.new;
c = FloatArray.newClear(65493);

e = SCSlider( a, Rect(0, 0, size, 12));
e.action = {arg ex; b.setProperty(\xOffset, (ex.value * 4) + 1 )};

file = SoundFile.new;
              "sounds/a11wlk01.wav"
file.numFrames.postln;
file.readData(c);
//file.inspect;
file.close;
minval = 0;
maxval = 0;
f = Array.new;
d = Array.new;
c.do({arg fi, i;
if(fi < minval, {minval = fi});
if(fi > maxval, {maxval = fi});

 //f.postln;
if(i % 256 == 0,{
d = d.add((1 + maxval ) * 0.5 );
f = f.add((1 + minval ) * 0.5 );

minval = 0;
maxval = 0;
});
});

                //this is used to draw the upper part of the table
```

```
b.value_(f);

e = SCSlider( a, Rect(0, 0, size, 12));
e.action = {arg ex; b.setProperty(\startIndex, ex.value *f.size )};

//b.enabled_(false);
b.action = {arg xb; ("index:  " ++ xb.index).postln};
b.drawLines_(true);
b.drawRects_(false);
b.isFilled_(true);
b.selectionSize_(10);
b.index_(10);
b.thumbSize_(1);
b.gap_(0);
b.colors_(Color.black, Color.blue(1.0,1.0));
b.showIndex_(true);
a.front;

)
```

the "index" is also the "selection"

```
setting showIndex = true will allow selections.
shift click and drag will select an area.
setting selectionSize will set that selection area.
this dispay may also be used to look like an index as in the above sequencer example.
(
var size;
size = 12;
a = SCWindow("test", Rect(200 , 450, 10 + (size * 17), 10 + (size * 17)));
a.view.decorator =  FlowLayout(a.view.bounds);
b = SCMultiSliderView(a, Rect(0, 0, size * 17, size * 17));
b.action = { arg xb; ("index:  " ++ xb.index ++" value:  " ++ xb.currentvalue).postln};

c = Array.new;
size.do({ arg i;
c = c.add(i/size);
});
b.value_(c);
```

```
b.xOffset_(5);
b.thumbSize_(12.0);
b.strokeColor_(Color.blue);
b.fillColor_(Color.blue);

b.drawLines(false);

b.showIndex = true;
b.index_(4);
a.front;

)

// this means the x-dimension size in pixels
b.indexThumbSize = 40

// not the selection size

// value pixels, the y-dimension
b.valueThumbSize = 100
```

ID: 217

SCNumberBox

superclass: SCStaticTextBase

```
(
w = SCWindow("SCNumberBox Example", Rect(100, 500, 400, 60));
//w.view.decorator = FlowLayout(w.view.bounds);
b =  SCNumberBox(w, Rect(150, 10, 100, 20));
b.value = rrand(1,15);
b.action = {arg numb; numb.value.postln; };
w.front
)

b.value = rrand(1,15);

b.setProperty(\boxColor,Color.grey);
b.setProperty(\stringColor,Color.white);
b.setProperty(\align,\center);
```

ID: 218

SCPopUpMenu

The 8-fold noble path

```
(
var sbs;
w = SCWindow.new.front;
l = [
  "right view" "right thinking" "right mindfulness" "right speech"
  "right action" "right diligence" "right concentration" "right livelihood"
];
sbs = SCPopUpMenu(w,Rect(10,10,180,20));
sbs.items = l;
sbs.background_(Color.white);
sbs.action = { arg sbs;
[sbs.value, l.at(sbs.value)].postln; // .value returns the integer
};
)
```

The underlying OS X graphics system gives special meanings to some characters
- divider line

```
(

var sbs;
w = SCWindow.new.front;
l = [
  "1 absolute"
"-replaced by a divider", // starting with a -
  "3 relative",

  "4 fore <= aft", // fore aft ( <= disappears )
  "5 fore <hello aft", // fore ello aft
```

```
  "6 something -> else", // ok

  "7 fore –hello aft", // fore hello aft
  "8 fore –-hello aft", // fore -hello aft (one - )
  "9 fore -<hello aft", // fore ello aft

  "10 something (else)", // item greyed out
  "11 something \(else)", // item still greyed out
  "12 something [else]", // ok
  "13 something {else}", // ok

  "14 something | else" // ok

];
sbs = SCPopUpMenu(w,Rect(10,10,180,20));
sbs.items = l;
sbs.background_(Color.white);
sbs.action = { arg sbs;
[sbs.value, l.at(sbs.value)].postln; // .value returns the integer
};

)

also these:
<
=
(
```

ID: 219

SCRangeSlider

modifier keys:

command
begin drag
control
move whole range
shift
move lo point
alt
move hi point
normal
set value

ID: 220

SCSoundFileView

```
(
w = SCWindow.new("soundfile test", Rect(200, 200, 800, 400));
a = SCSoundFileView.new(w, Rect(20,20, 700, 60));

f = SoundFile.new;
            "sounds/a11wlk01.wav"
f.inspect;

a.soundfile = f;
a.read(0, f.numFrames);
a.elasticMode = true;

a.timeCursorOn = true;
a.timeCursorColor = Color.red;
a.timeCursorPosition = 2050;
a.drawsWaveForm = true;
a.gridOn = true;
a.gridResolution = 0.2;

w.front;

)
```

ID: 221

SCStaticText

A non-editable textfield

string_(string)
set the text.

font_(font)
set the font.

stringColor_(color)
set the color of the string.

Examples

```
(
w = SCWindow.new.front;
a = SCStaticText(w, Rect(10, 10, 100, 20));
            "Rolof's Rolex"
)

// adjust bounds
a.bounds = Rect(5, 5, 100, 20)


///// dynamic

(
w = SCWindow.new.front;
a = Array.fill(20, {SCStaticText(w, Rect(w.bounds.extent.x.rand, w.bounds.extent.y.rand, 100, 16))
.string_("Rolof's Rolex".scramble)
.stringColor_(Color.rand)
.font_(Font([
  "Helvetica-Bold",
"Helvetica",
"Monaco",
" Arial",
"Gadget",
```

"MarkerFelt-Thin "

```
].choose, 16))
});
)

r = {inf.do{| i|
thisThread.randSeed_(1284);
a.do{| item|
{item.bounds = Rect(5+w.bounds.extent.x.rand * (cos(i*0.01)).abs, w.bounds.extent.y.rand * sin(i*0.001),
100, 20)}.defer;
};
0.1.wait;
}}.fork

r.stop
```

ID: 222

SCTabletView

superclass: SCView

An otherwise featureless view that receives extended wacom tablet data. It can also be used with a normal mouse but with less resolution.

action - dragging the mouse inside the view
mouseDownAction
mouseUpAction

Each of the three actions are passed the following wacom tablet values:

view - the view
x - subpixel location in view
y - subpixel location in view
pressure - 0..1
tiltX - 0..1
tiltY - 0..1
deviceID - will be used to look up if the tip or the eraser is used
buttonNumber - 0 left, 1 right, 2 middle wheel click
clickCount - double click, triple click ...
most relevant for the mouseDown, but still valid for the dragged and mouseUp
absoluteZ - the wheel on the side of some mice
rotation - in degrees, only on the 4d mice

If using a mouse (even a wacom) rather than a pen, the x and y will be integer pixel values, rather than subpixel floats. Wacom stylus devices have higher resolution than the screen. Pressure will be 1 for mouse down, 0 for mouse up.

Properties

clipToBounds - by default the x y values are clipped to the bounds of the view.
it set to 0, it is possible to drag from inside to outside the view, and the x y values will exceed the bounds accordingly.

```
(
w = SCWindow.new;
t = SCTabletView(w,Rect(40,40,300,300));
t.background = Color.white;
w.front;

t.mouseDownAction = { arg  view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation;

["down",x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation].postln;
};

t.action = { arg  view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation;

["dragging", x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation].postln;
t.background = Color(x / 300,y / 300,tiltx,pressure);
};

t.mouseUpAction = { arg  view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation;

["up",x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount,absoluteZ,rotation].postln;
};

)
```

Assign the same function to each action

```
(
w = SCWindow.new;
t = SCTabletView(w,Rect(40,40,300,300));
t.background = Color.white;
w.front;

f = { arg  view,x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount;
[x,y,pressure,tiltx,tilty,deviceID, buttonNumber,clickCount].postln;
t.background = Color(x / 300,y / 300,tiltx,pressure);
};

t.mouseDownAction = f;
t.action = f;
```

```
t.mouseUpAction = f;

)
```

An example using crucial library

```
(
Instr([\minimoog,\loose],{ arg freq=440,int1=5,int2 = -5,
ffreqInterval=0,rq=0.4,gate=0.0;
var p,c,d,f;
c=LFNoise1.kr(0.1,0.45,0.55);
d=LFNoise1.kr(0.1,0.45,0.55);
f=LFNoise1.kr(0.1,2);
p=Pulse.ar([ freq  * int1.midiratio + f , freq, freq * int2.midiratio - f],
[c,d,c],0.2);

RLPF.ar(Mix.ar(p),freq * ffreqInterval.midiratio,rq)
* EnvGen.kr(Env.adsr, gate, Latch.kr(gate,gate))

},#[
 nil
[[-48,48,\linear,1]],
[[-48,48,\linear,1]],
[[-48,48,\linear,1]]
]);

p = Patch.new([ 'minimoog', 'loose' ],[
 nil nil nil nil nil
 KrNumberEditor     \gate  // override the default control
]);

Sheet   arg
var v,freq,int;
freq = ControlSpec(100,3000,\exp);
int = [-48,48,\linear,1].asSpec;

p.topGui(f);
```

```
v = SCTabletView(f,Rect(0,0,200,200));
v.background = Color.white;
v.action = { arg view,x,y,pressure,tiltx,tilty;
p.args.at(1).value_( int.map( x / 200 ) ).changed;
p.args.at(2).value_( int.map( y / 200 ) ).changed;
p.args.at(3).value_( int.map( pressure ) ).changed;
};
v.mouseDownAction = { arg view,x,y,pressure;
p.args.at(0).value_( rrand(30,80).midicps ).changed;
p.args.at(5).value_( pressure ).changed;
};
v.mouseUpAction = { arg view,x,y,pressure;
p.args.at(5).value_( 0.0 ).changed;
};
});
```

)

move the box

(buggy)

```
(
w = SCWindow.new;
t = SCTabletView(w,Rect(40,40,30,30));
t.background = Color.white;
w.front;

t.action = { arg view,x,y;
var b;
b = t.bounds;
b.left = b.left + x;
//b.top = y - b.top;
t.bounds = b;
};

)
```

ID: 223

SCTextField

superclass: SCNumberBox

```
Sheet   arg
b =  SCTextField(l,Rect(0,0,150,30));
b.string = "hi there";
b.action = {arg field; field.value.postln; }
});


// does not do the action
b.value = "yo";


b.setProperty(\boxColor,Color.grey);
b.setProperty(\stringColor,Color.white);
b.setProperty(\align,\center);
```

Does not handle composed character sequences (é ø etc.)

option-e appears to freeze it ?

ID: 224

SCTextView a text editor

superclass: SCView
see also: Document

***new(window, bounds);**

string_ set the text
string get the text
setString(string, rangestart, rangesize)
set text into a range
selectedString get the selected text only
selectionStart get the current position in the text
selectionSize get the current size of selection in the text
stringColor_ set the color of the whole text
setStringColor(color, start, size)
set the color of a selection of text
setStringColor(color, start, size)
autoHideScrollers_
hasVerticalScroller_
hasHorizontalScroller_
textBounds_
font_
usesTabToFocusNextView_
enterInterpretsSelection_

```
//examples
```

```
(
var win, txtv;

win = SCWindow.new.front;
win.view.decorator_(FlowLayout(win.view.bounds));
txtv = SCTextView(win,Rect(0,0, 300,200))
```

```
.hasVerticalScroller_(true)
.autohidesScrollers_(true)
.focus(true);
)
```

ID: 225

SCUserView user-definable view

superclass: SCView

SCUserView is a user-definable View intended mainly for use with Pen and drawHooks.

See also: **[SCWindow]**, **[Pen]**, **[Color]**, and **[String]**

keyDownFunc_

Set the function which should be evaluated if the view is in focus and a key is pressed. This function will be passed four arguments: the View, the key pressed as a **Char**, modifier keys (shift, alt, etc.), and the unicode value. See **[SCView]** for more details.

```
(
 // select the window, type something and watch the post window
w = SCWindow.new("select this window and type something");
  c = SCUserView   w.view.bounds
c.keyDownFunc = { arg view,char,modifiers,unicode;
[char, modifiers, unicode].postln;
c.drawFunc = {
char.asString.drawAtPoint(180@150, Font(" Gadget", 70), Color.blue(0.3, 0.5))
};
w.refresh;
};
w.front; c.focus;
)
```

drawFunc_

Set the function which should be evaluated if the view is refreshed. This happens every time the whole window is refreshed (manually by calling SCWindow-refresh or e.g. by selecting the view or resizing the window).

```
(
var func;

func = {| me|
```

```
Pen.use{
   // clipping into the boundingbox
Pen.moveTo((me.bounds.left)@(me.bounds.top));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@0));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@me.bounds.height));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (0@me.bounds.height));
Pen.lineTo((me.bounds.left)@(me.bounds.top));
Pen.clip;

   // draw background
Color.gray(0.5).set;
Pen.moveTo((me.bounds.left)@(me.bounds.top));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@0));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@me.bounds.height));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (0@me.bounds.height));
Pen.lineTo((me.bounds.left)@(me.bounds.top));
Pen.fill;

Pen.translate(100, 100);
10.do{
Color.red(rrand(0.0, 1), rrand(0.0, 0.5)).set;
Pen.addArc((400.exprand(2))@(100.rand), rrand(10, 100), 2pi.rand, pi);
Pen.perform([\stroke, \fill].choose);
}
}
};

 w = SCWindow      "DrawFunc Examples"
w.view.background_(Color.white);
3.do{| i|
v = SCUserView(w, Rect(20+(i*120), 100, 100, 100));
v.drawFunc = func;
};
w.refresh;
```

```
)
```

mouseBeginTrackFunc_

Set the function which should be evaluated if the mouse is at the beginning of tracking (mouse-down). This function will be passed four arguments: theView, x coordinate, y coordinate, and keyboard modifiers.

mouseTrackFunc_

Set the function which should be evaluated if the mouse is tracked. This function will be passed four arguments: theView, x coordinate, y coordinate, and keyboard modifiers.

mouseEndTrackFunc_

Set the function which should be evaluated if the mouse is at the end of tracking (mouse-up). This function will be passed four arguments: theView, x coordinate, y coordinate, and keyboard modifiers.

```
(
var drawFunc, beginTrackFunc, endTrackFunc, trackFunc, sat = 0, absX;

drawFunc = {| me|
Pen.use{
   // clipping into the boundingbox
Pen.moveTo((me.bounds.left)@(me.bounds.top));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@0));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@me.bounds.height));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (0@me.bounds.height));
Pen.lineTo((me.bounds.left)@(me.bounds.top));
Pen.clip;

   // draw background
Color.gray(sat).set;
Pen.moveTo((me.bounds.left)@(me.bounds.top));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
```

```
+ (me.bounds.width@0));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (me.bounds.width@me.bounds.height));
Pen.lineTo(((me.bounds.left)@(me.bounds.top))
+ (0@me.bounds.height));
Pen.lineTo((me.bounds.left)@(me.bounds.top));
Pen.fill;

Pen.translate(100, 100);
10.do{
Color.red(rrand(0.0, 1), rrand(0.0, 0.5)).set;
Pen.addArc((400.exprand(2))@(100.rand), rrand(10, 100), 2pi.rand, pi);
Pen.perform([\stroke, \fill].choose);
}
}
};
beginTrackFunc = {| me, x, y, mod|
absX = x;
  postf("begin path:  x=%\n"
};
endTrackFunc = {| me, x, y, mod|
  postf("end path:  (absX-x)=%\n"
};
 trackFunc = {| me, x, y, mod|
sat = ((absX-x)/100);
me.refresh;
};

w = SCWindow.new.front;
w.view.background_(Color.white);
3.do{| i|
v = SCUserView(w, Rect(20+(i*120), 100, 100, 100));
  //v.background_(Color.white); // not affecting anything...
v.drawFunc = drawFunc;
v.mouseBeginTrackFunc = beginTrackFunc;
v.mouseEndTrackFunc = endTrackFunc;
v.mouseTrackFunc = trackFunc;
};
w.refresh;
)
```

```
// draw on the view

(
var w, txt, tmppoints, all;
tmppoints = [];

w = SCWindow("draw on me", Rect(128, 64, 340, 360));
w.drawHook_{
  Pen.use {
Pen.width = 1;
Pen.beginPath;

tmppoints.do{ | p, i|
if(i == 0){
Pen.moveTo(p);
}{
Pen.lineTo(p);
}
};
all.do{| points|
points.do{| p, i|
if(i == 0){
Pen.moveTo(p);
}{
Pen.lineTo(p);
}
};
};
Pen.stroke;
};
};
v = SCUserView(w,Rect(0, 0, 340, 360))
.mouseTrackFunc_({| v,x,y|
tmppoints = tmppoints.add(x@y);
w.refresh;
})
.mouseEndTrackFunc_({| v,x,y|
all = all.add(tmppoints.copy);
tmppoints = [];
```

```
w.refresh;
});

w.front;
)
```

ID: 226

SCView

superclass: Object

SCView is the abstract superclass for all SC GUI widgets. Currently this system is OSX only. On Linux there is another GUI implementation, SCUM, which has its own documentation. Several key methods and variables are defined in SCView and inherited in its subclasses.

resize_(int)

This setting controls how the widget will behave when it's window or enclosing view is resized. This is illustrated graphically below:

1 2 3
4 5 6
7 8 9

1 - fixed to left, fixed to top
2 - horizontally elastic, fixed to top
3 - fixed to right, fixed to top

4 - fixed to left, vertically elastic
5 - horizontally elastic, vertically elastic
6 - fixed to right, vertically elastic

7 - fixed to left, fixed to bottom
8 - horizontally elastic, fixed to bottom
9 - fixed to right, fixed to bottom

resize

Return an Integer corresponding to the current resize behaviour (see above).

keyDownAction_(aFunction)

Register a **Function** to be evaluated when a keystroke is received and this view is in focus.

```
(
// select the slider, type something and watch the post window
w = SCWindow.new;
c = SCSlider(w,Rect(0,0,100,30));
c.keyDownAction = { arg view,char,modifiers,unicode,keycode;
[char,modifiers,unicode,keycode].postln;
};
w.front;
)
```

If you return nil from your function, or you have no function registered, the event will bubble up to the parent view which may then respond. It will continue to bubble up unless something responds or it hits the topView of the window. You may register a function in the window's topView to respond to all unhandled events for the window.

There are default keyDownActions for some views, which will be overridden when you set a keydown action.

When called, the function will be passed the following arguments:

view - The receiving instance of SCView.

char - The character pressed, possibly unprintable. Character sequences (for example é) get passed as two characters, the first one blank ( ), the second one is the unmodified character (e). This will also vary depending on the nationality the keyboard is set to.

modifiers - An integer bit field indicating the modifier keys in effect. You can examine individual flag settings using the C bitwise AND operator.

65536 NSAlphaShiftKeyMask
Set if Caps Lock key is pressed.

131072 NSShiftKeyMask
Set if Shift key is pressed.

262144 NSControlKeyMask
Set if Control key is pressed.

524288 NSAlternateKeyMask

Set if Option or Alternate key is pressed.

1048576 NSCommandKeyMask
Set if Command key is pressed.

2097152 NSNumericPadKeyMask
Set if any key in the numeric keypad is pressed. The numeric keypad is generally on the right side of the keyboard.

4194304 NSHelpKeyMask
Set if the Help key is pressed.

8388608 NSFunctionKeyMask
Set if any function key is pressed. The function keys include the F keys at the top of most keyboards (F1, F2, and so on) and the navigation keys in the center of most keyboards (Help, Forward Delete, Home, End, Page Up, Page Down, and the arrow keys).

arrow keys have an extra modifier value of 10485760
so for a shift arrow key do a bitwise 'or' with the shift mask:
10485760 | 131072
= 10616832 // this is the mask for shift arrow key

unicode - The unicode integer, identical to the char.

keycode - The hardware dependent keycode indicating the physical key. This will vary from machine to machine, but is useful for building musical interfaces using the computer keyboard. In order to play little melodies, this code will identify which key you consider to be special.

N.B.: Function key modifier may change the keycode.

For various reasons these don't make it through cocoa:
most command modifiers
cntl-tab
cntl-escape
tab and shift tab are currently trapped by SC itself for cycling the focus through the views.
(we could change this)

keyDownAction

Return the current keyDownAction function for this view if there is one, otherwise return nil.

*globalKeyDownAction_(func)

A function that is evaluated for every keyDown event on every SCView. See **keyDownAction_** for details.

focus

Brings this view into focus.

```
(
w = SCWindow.new;
c = SCSlider(w,Rect(0,0,100,30));
d = SCSlider(w,Rect(0,30,100,30));
w.front;
)

c.focus;
d.focus;
w.close;
```

refresh

Under certain circumstances a view will not automatically update its appearance. This forces a redraw.

```
(
w = SCWindow.new;
c = SCButton(w,Rect(0,0,100,30));
c.states = [["a",Color.black,Color.red]];
d = SCButton(w,Rect(0,30,100,30));
d.states = [["a",Color.black,Color.red]];
w.front;
)

 // won't display change...
c.states = [["b",Color.red,Color.black]];
```

```
d.states = [["b",Color.red,Color.black]];

 //until
c.refresh;

 //needs separate refresh
d.refresh;

 // in some cases might be better to refresh the whole window
 // which does refresh on all damaged areas (it keeps track, doesn't redraw whole thing)

c.states = [["a",Color.black,Color.red]];
w.refresh;
w.close;
```

drag and drop

Each view subclass has a default object that it exports when dragged from. For sliders its the value of the slider, for lists it is the currently selected numeric index etc.

By setting the beginDragAction handler you can return a different object based on the context and your application.

beginDragAction(theView) - return the object you wish your view to export by dragging

```
aView.beginDragAction =  { arg theView;  someList[ theView.value ]  }
```

The current dragged thing can be found in the classvar SCView.currentDrag. Objects dragged from within
SuperCollider are also in SCView.currentDragString as a compile string. Text dragged from other applications is in SCView.currentDragString and the results of attempting to compile that as sc code is in SCView.currentDrag

Each view subclass has a defaultCanReceiveDrag method that determines if the current object being dragged is possible for this view to accept, and a defaultReceiveDrag method for actually receiving the drag. Sliders accept numbers, simple text labels do not accept drags etc. After receiving the drag, the SCView.currentDrag is set to nil.

By setting the canReceiveDragHandler and receiveDragHandler you can make any view

accept and receive objects based on the context and your application. (Note: currently not possible for SCStaticText)

canReceiveDrag(theView) - return true/false if you are willing to accept the current drag.

```
aView.canReceiveDrag = false; // no, leave me alone
aView.canReceiveDrag = { SCView.currentDrag.isString };
```

receiveDrag(theView) - accept the drag.

```
aView.receiveDrag = {
SCView.currentDrag.postln;
}
```

The default drag object from a list view is the currently selected integer index.
Here a list view is made to export a string.

```
(
f = SCWindow.new.front;
a = SCListView(f,100@100);
 a.items = ["eh?" "bee!" "sea."
a.beginDragAction = { arg listView;
listView.items[ listView.value ].debug("begun dragging");
};

c = nil;
b = SCButton(f,Rect(0,200,200,20));
b.states = [["Drop stuff on me"]];
b.canReceiveDragHandler = { SCView.currentDrag.isString };
b.receiveDragHandler = {
b.states = [[SCView.currentDrag]];
c = SCView.currentDrag;
};
b.action = {
c.postln
};

)
```

ID: 227

SCVLayoutView

```
(
q = 10;
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));

Array.fill(q,{ arg i;
SCSlider(v,Rect(0,0,75,20)).value_(i / q)
});

w.front
)
```

elastic

```
resize the window ...  oooh
(
q = 10;
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));
v.resize = 5; //   elastic
Array.fill(q,{  arg i;
var s;
s = SCSlider(v,Rect(0,0,75,20));
s.value = i / q;
s
});

w.front
)

(
q = 10;
```

```
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));
v.resize = 5; //   elastic
Array.fill(q,{  arg i;
var s;
s = SCSlider(v,Rect(0,0,75,20));
s.resize = 5; // elastic
s.value = i / q;
s
});

w.front
)

(
q = 5;
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));
v.resize = 5; //   elastic

Array.fill(q,{  arg i;
var s;
s = SCSlider(v,Rect(0,0,75,20));
s.value = i / 5;
if(i < 2,{
s.resize = 5; // some elastic
s.setProperty(\minHeight,20);
},{
  s.resize = 1; // some not elastic
});
s
});

w.front
)

(
q = 5;
```

```
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));
v.resize = 5; //   elastic

Array.fill(q,{  arg i;
var s;
s = SCSlider(v,Rect(0,0,75,20));

s.value = i / 5;
s.resize = 5;
s.setProperty(\minHeight,20);
s.setProperty(\maxHeight,40);
s
});

w.front
)
```

spacing

```
(
q = 10;
w = SCWindow.new;

v = SCVLayoutView(w,Rect(10,10,300,300));
v.setProperty(\spacing,0);

Array.fill(q,{
SCSlider(v,Rect(0,0,75,20))
});

w.front
)
```

ID: 228

SCWindow user interface window

***new(name, bounds, resizable, border);**

bounds: a Rect(
distance from left,
distance from bottom,
width,
height
)
***closeAll** closes all windows
***allWindows** a list of all windows

fullScreen fullscreeen mode, no way to close it then. so don't forget the button
endFullScreen end the fullscreen mode

userCanClose_ if set to false, window is uncloseable
close close the window
front display the window, bring it to the front.

refresh sometimes this has to becalled so the views are updated
alpha_ tranparency channel value (0...1)
bounds_ set the bounds to a Rect

onClose_ can be set to a function

```
//examples


//how to add views
(

var w;
w = SCWindow("my name is...  panel", Rect(128, 64, 340, 360));
```

```
32.do({ arg i;
b = SCButton(w, Rect(rrand(20,300),rrand(20,300), 75, 24));
b.states = [["Start "++i, Color.black, Color.rand],
["Stop "++i, Color.white, Color.red]];
});

w.front;

)
```

view

every window has an SCTopView instance, which contains all the other views.

```
(

var w;
w = SCWindow("my name is...  panel", Rect(128, 64, 340, 360));

w.view.decorator = FlowLayout(w.view.bounds);
w.view.background = Color(0.6,0.8,0.8);
w.front;

32.do({ arg i;
b = SCButton(w, Rect(rrand(20,300),rrand(20,300), 75, 24));
b.states = [["Start "++i, Color.black, Color.rand],
["Stop "++i, Color.white, Color.red]];
});

w.front;

)
```

bounds_(aRect)

set the bounds of the window

```
(

x = SCWindow.new;
x.front;
```

```
x.bounds_(Rect(10,10,100,30));

)
```

Note that the setting of the bounds doesn't happen until the application finishes its current application event cycle. Thus, if you check the bounds in the same chunk of code, the SCWindow will not yet have it updated.

```
  // execute this all at once
w = SCWindow.new("test");
w.front;
w.bounds = Rect(50, 50, 50, 50);
w.bounds.postln;
{ w.bounds.postln; nil }.defer(0.1); // next application event cycle
```

setInnerExtent(width,height)

Changes the size of the window while keeping the top left corner fixed. This is the usual desired behavior, but quick draw and Rect have flipped coordinate systems.

userCanClose_(boolean)

Set this to true to prevent command-w from closing the window. window.close will still close it, and it will still close on recompiling the library.

border argument

```
 SCWindow            false         //can't be closed, as it has no buttons, also cmd-w not.
SCWindow.closeAll;
```

onClose

get the current onClose function.

onClose_

set a function that will be evaluated when the window is closed.

```
//close the window and the synth plays
(
x = SCWindow.new.front;
x.alpha = 0.8;
x.onClose_({ Synth.new(\default) });
)
```

ID: 229

Stethoscope **scope window**

a graphical interface to navigate on buses

works only with internal server

the scope window can be controlled by the following keys:

J one channel back
K switch rate (audio vs. control)
L one channel forward

O jump to first hardware output channel and adjust numChannels to hardware
I jump to first hardware input channel and adjust numChannels to hardware

space run, if not running anyway.
. (period) stop.
M toggle screen size

+ / - zoom horizontally
*** / _** zoom vertically
S change style between parallel and overlay

shift S change style to lissajou (use only with fast computer and small buffer size)
shift A allocate buffer size so it fills the screen (to next power of two) (this can be dangerous, might crash)

instance creation:

***new(server, numChannels, index, bufsize, zoom, rate, view)**
returns a new instance of Stethoscope.

by the message**.scope**:

aServer.scope(numChannels, index, bufsize, zoom, rate)
opens a scope window for the server, stores it in the server instance var **scopeWindow**

aBus.scope(bufsize, zoom)
displays buffer channels in scope

aFunction.scope(numChannels, outbus, fadeTime, bufsize, zoom)
plays a function and shows output in scope, returns synth object, like { }**.play**

instance methods:

allocBuffer(size)
(re)allocate the buffer to a given size
run
start it if not playing anyway
free
end it, free the buffer

numChannels_
change the number of channels displayed
index_
change the offset index
rate_
change the rate (\audio or \control)
size_
set the window size (default: 222)
zoom_
set horizontal zoom

setProperties(numChannels, index, bufsize, zoom, rate)
any of these given will adjust the scope accordingly:
e.g. x.setProperties(zoom:8) will only zoom.

```
// examples:

(
```

```
Server.default = Server.internal;
s = Server.default;
s.boot;
)
(
{
SinOsc.ar([225, 450, 900], 0, 0.2)
+ LPF.ar(
LFPulse.ar(226 * [1, 2, 5],[0,0.1,0.1],0.2, 0.2),
MouseX.kr(20, 10000, 1)
)
}.scope;
)

// server.scope only changes the properies explicitly given:

s.scope(numChannels:5);
s.scope(index:12);
s.scope(zoom:4);
s.scope(index:0);

s.scopeWindow.size = 600;
s.scopeWindow.size = 222;

// scoping buses:

a = Bus.audio(s, 4);
{ WhiteNoise.ar(0.2.dup(4)) }.play(s, a.index);

a.scope;

c = Bus.control(s, 3);
{ WhiteNoise.kr(1.dup(4) * MouseX.kr) }.play(s, c.index);

c.scope;

// note that scoping control rate buses shows block size interpolation (this is due to the
// fact that ScopeOut.kr doesn't work yet.)
```

external use: you can pass your own view in to add a stethoscope to it;

```
w = SCWindow.new("my own scope", Rect(20, 20, 400, 500));
w.view.decorator = FlowLayout(w.view.bounds);
c = Stethoscope.new(s, view:w.view);
w.onClose = { c.free }; // don't forget this
w.front;
```

11 Help-scripts

ID: 230

```
// trolls the help extension help directories and compiles a doc with links

var path, doc, result, headingIndices, headingFont, excluded, addFunc;
var underlineStarts, underlineRanges, titleString, thirdParty, thirdPartyIndex;
var extensions, extensionsIndex, extensionFunc, extensionsRoots, extensionsFolders;
var undoc, undocIndex;

path = PathName.new("Help/");

headingIndices = List.new;

              "A Generated List of all Documented Classes"
        "*Show All Undocumented Classes"

                        Char       Char       "Below is an automatically generated list of all documented
classes, sorted by directory.  For a list of undocumented classes click here:"  Char

undocIndex = result.size;

result = result ++ undoc ++ Char.nl ++ Char.nl;

// put included third party libraries at the end
            PathName "Help/crucial"   PathName "Help/JITLib"

// this func trolls the directory and harvests the descriptions
           | folderPathName|
var classFiles, heading, currentFile, currentFileString;
var removeIndices, spaceIndices, removePairs, lastSpace = 0;
classFiles = "";

folderPathName.files.do({| item|
var nameString, nameIndex, end;
nameString = item.fileName.split($.).at(0);
if(nameString.asSymbol.asClass.notNil, {
currentFile = File(item.fullPath, "r");
currentFileString = currentFile.readAllString;
   // fix accent acute (remove it)
currentFileString.findAll("\'8").reverseDo({ | i|
```

```
currentFileString = currentFileString.copyFromStart(i-2) ++
currentFileString.copyToEnd(i+2);
});
currentFile.close;
   // strip RTF gunk
currentFileString = currentFileString.stripRTF;
nameIndex = currentFileString.find(nameString);
if(nameIndex.notNil, {
currentFileString = currentFileString.drop(nameIndex);
end = currentFileString.find("\n");
if( end.notNil, {
end = end - 1;
currentFileString = currentFileString.copyFromStart(end);
});

    // remove tab stops
currentFileString = currentFileString.reject({| item| item == $\t});
    // remove commas, hyphens, and spaces
while({(currentFileString[nameString.size] == $,) | |
(currentFileString[nameString.size] == $) | |
(currentFileString[nameString.size] == $-)},
{currentFileString = currentFileString.copyFromStart(nameString.size -1) ++
currentFileString.copyToEnd(nameString.size + 1);
}
);
if(currentFileString.size > nameString.size, {
currentFileString = currentFileString.insert(nameString.size, "\t");
});},
{
currentFileString = nameString;
}
);
   // add square brackets
currentFileString = currentFileString.insert(nameString.size, "]");
currentFileString = currentFileString.insert(0, "[");

classFiles = classFiles ++ Char.tab ++ currentFileString ++ Char.nl;
});
});
if(classFiles.size > 0, {
```

```
  //heading = folderPathName.fileName;
heading = folderPathName.fullPath;
headingIndices.add([result.size, heading.size]);

result = result ++ heading ++ Char.nl ++ Char.nl ++ classFiles ++ Char.nl;
});
folderPathName.foldersWithoutCVS.do({| folder|
if(excluded.detect({| item|  item.fileName == folder.fileName; }).isNil,
{addFunc.value(folder);}
);
});
};

addFunc.value(path);

// Check for Extensions Folders and add if they exist

                  PathName "/Library/Application Support/SuperCollider/Extensions"
 PathName " /Library/Application Support/SuperCollider/Extensions"

extensionsRoots.any({| item|  item.pathMatch.size > 0 }).if({

extensionsFolders = List.new;
extensionFunc = { | path|
path.folders.do({| item|
item.fullPath.containsi("help").if({ extensionsFolders.add(item)},{
extensionFunc.value(item);});
});
};

extensionsRoots.do({| item|  extensionFunc.value(item); });

 result = result ++ "\n------------------------\n\n"
extensions = "Extensions:";
extensionsIndex = result.size;
result = result ++ extensions + Char.nl + Char.nl;
extensionsFolders.do({| item|  addFunc.value(item);});

});
```

```
// Third Party Libraries
                    "\n------------------------\n\n"

              "Included Third Party Libraries:"
thirdPartyIndex = result.size;

result = result ++ thirdParty + Char.nl + Char.nl;

excluded.do({| item|  addFunc.value(item); result = result ++ "\n------------\n\n";});

//doc = Document.new("Documented Classes");

// this sets basic tab stops and line spacing
      Document      File           "/"    "Help/help-scripts/tab-template.rtf"

             "Documented Classes"

// set the fonts
doc.setFont(Font("Helvetica", 12));
doc.string = result;

doc.setFont(Font("Helvetica-Bold", 18), 0, titleString.size);

doc.setFont(Font("Helvetica-Bold", 16), thirdPartyIndex, thirdParty.size);

extensionsIndex.notNil.if({
doc.setFont(Font("Helvetica-Bold", 16), extensionsIndex, extensions.size);
});

headingFont = Font("Helvetica-Bold", 14);
headingIndices.do({| item|  doc.setFont(headingFont, *item)});

// find the underlines for help links.  Apparently faster than storing them above.
underlineStarts = doc.string.findAll("[").reverse + 1;
underlineRanges = doc.string.findAll("]").reverse - underlineStarts;

underlineStarts.do({| item, i|  doc.selectRange(item, underlineRanges[i]); doc.underlineSelection;});

doc.selectRange(undocIndex, undoc.size);
doc.underlineSelection;
```

```
doc.selectRange(0,0);
doc.editable_(false);

// keeps window title as it should be!
doc.mouseDownAction = { {doc.title = "Documented Classes";}.defer(0.00001) };

{doc.removeUndo;}.defer(0.001);
```

ID: 231

```
// trolls the help extension help directories and compiles a doc with links

var path, doc, result, headingIndices, headingFont, excluded, addFunc;
var underlineStarts, underlineRanges, titleString, thirdParty, thirdPartyIndex;
var extensions, extensionsIndex, extensionFunc, extensionsRoots, extensionsFolders;
var undoc, undocIndex;

path = PathName.new("Help/");

headingIndices = List.new;

              "A Generated List of all Documented Extension Classes"
        "*Show All Undocumented Classes"

                        Char       Char       "Below is an automatically generated list of all documented
extension classes (i.e.  those whose class and help files are in /Library/Application Support/SuperCollider/Extensions
or  /Library/Application Support/SuperCollider/Extensions), sorted by directory.\n\nFor a list of un-
documented classes click here:"  Char

undocIndex = result.size;

result = result ++ undoc ++ Char.nl ++ Char.nl;

// put included third party libraries at the end
            PathName "Help/crucial"   PathName "Help/JITLib"

// this func trolls the directory and harvests the descriptions
           | folderPathName|
var classFiles, heading, currentFile, currentFileString, temp;
classFiles = "";

folderPathName.files.do({| item|
var nameString, nameIndex, end;
nameString = item.fileName.split($.).at(0);
if(nameString.asSymbol.asClass.notNil, {
currentFile = File(item.fullPath, "r");
currentFileString = currentFile.readAllString;
   // fix accent acute (remove it)
```

```
currentFileString.findAll("\'8").reverseDo({ | i|
currentFileString = currentFileString.copyFromStart(i-2) ++
currentFileString.copyToEnd(i+2);
});
currentFile.close;
   // strip RTF gunk
currentFileString = currentFileString.stripRTF;
nameIndex = currentFileString.find(nameString);
if(nameIndex.notNil, {
currentFileString = currentFileString.drop(nameIndex);
end = currentFileString.find("\n");
if( end.notNil, {
end = end - 1;
currentFileString = currentFileString.copyFromStart(end);
});
    // remove tab stops
currentFileString = currentFileString.reject({| item| item == $\t});
    // remove commas, hyphens, and spaces
while({(currentFileString[nameString.size] == $,) | |
(currentFileString[nameString.size] == $) | |
(currentFileString[nameString.size] == $-)},
{currentFileString = currentFileString.copyFromStart(nameString.size -1) ++
currentFileString.copyToEnd(nameString.size + 1);
}
);
if(currentFileString.size > nameString.size, {
currentFileString = currentFileString.insert(nameString.size, "\t");
});},
{
currentFileString = nameString;
}
);
   // add square brackets
currentFileString = currentFileString.insert(nameString.size, "]");
currentFileString = currentFileString.insert(0, "[");

classFiles = classFiles ++ Char.tab ++ currentFileString ++ Char.nl;
});
});
if(classFiles.size > 0, {
```

```
  //heading = folderPathName.fileName;
heading = folderPathName.fullPath;
headingIndices.add([result.size, heading.size]);

result = result ++ heading ++ Char.nl ++ Char.nl ++ classFiles ++ Char.nl;
});
folderPathName.foldersWithoutCVS.do({| folder|
if(excluded.detect({| item|  item.fileName == folder.fileName; }).isNil,
{addFunc.value(folder);}
);
});
};

//addFunc.value(path);

// Check for Extensions Folders and add if they exist

                  PathName "/Library/Application Support/SuperCollider/Extensions"
 PathName " /Library/Application Support/SuperCollider/Extensions"

extensionsRoots.any({| item|  item.pathMatch.size > 0 }).if({

extensionsFolders = List.new;
extensionFunc = { | path|
path.folders.do({| item|
item.fullPath.containsi("help").if({ extensionsFolders.add(item)},{
extensionFunc.value(item);});
});
};

extensionsRoots.do({| item|  extensionFunc.value(item); });
result = result ++ "\n\n";
// result = result ++ "\n-----------------------\n\n";
// extensions = "Extensions:";
// extensionsIndex = result.size;
// result = result ++ extensions + Char.nl + Char.nl;
extensionsFolders.do({| item|  addFunc.value(item);});

});
```

```
// Third Party Libraries
//result = result ++ "\n-----------------------\n\n";
//
//thirdParty = "Included Third Party Libraries:";
//thirdPartyIndex = result.size;
//
//result = result ++ thirdParty + Char.nl + Char.nl;
//
//excluded.do({| item| addFunc.value(item); result = result ++ "\n------------\n\n";});

//doc = Document.new("Documented Classes");

// this sets basic tab stops and line spacing
      Document      File           "/"    "Help/help-scripts/tab-template.rtf"

            "Documented Extension Classes"

// set the fonts
doc.setFont(Font("Helvetica", 12));
doc.string = result;

doc.setFont(Font("Helvetica-Bold", 18), 0, titleString.size);

//doc.setFont(Font("Helvetica-Bold", 16), thirdPartyIndex, thirdParty.size);

extensionsIndex.notNil.if({
doc.setFont(Font("Helvetica-Bold", 16), extensionsIndex, extensions.size);
});

headingFont = Font("Helvetica-Bold", 14);
headingIndices.do({| item|  doc.setFont(headingFont, *item)});

// find the underlines for help links.  Apparently faster than storing them above.
underlineStarts = doc.string.findAll("[").reverse + 1;
underlineRanges = doc.string.findAll("]").reverse - underlineStarts;

underlineStarts.do({| item, i|  doc.selectRange(item, underlineRanges[i]); doc.underlineSelection;});

doc.selectRange(undocIndex, undoc.size);
doc.underlineSelection;
```

```
doc.selectRange(0,0);
doc.editable_(false);

// keeps window title as it should be!
doc.mouseDownAction = { {doc.title = "Documented Classes";}.defer(0.00001) };

{doc.removeUndo;}.defer(0.001);
```

ID: 232

```
// Generates a list of all classes for which there are no help files.

var paths, doc, result, addFunc;
var titleString, infoString;
var documentedClasses, undocumentedClasses, classesStartIndex;
var documented, documentedIndex;

        PathName "Help/"   PathName "/Library/Application Support/SuperCollider/Extensions"
 PathName " /Library/Application Support/SuperCollider/Extensions"

             "A Generated List of all Undocumented Classes"

            "Below is an alphabetical list of all classes which have no help files.  This includes classes
from CRUCIAL-LIBRARY, JITLib, and other third party libraries you may have installed.  Note that many
of these are either private classes not intended for direct use, abstract superclasses (such as Clock),
or currently non-functioning or vestigial classes (such as the image synthesis classes from SC3d5).  Nev-
ertheless this is a good place to look for undocumented functionality.  Note that some of these classes
are covered in overviews, tutorials, etc.\n\nClicking on any of the Class Names below will open a Class
Browser.  For a list of documented classes click here:\n\n"

result = titleString ++ Char.nl ++ Char.nl ++ infoString;

            "*Show All Documented Classes"
documentedIndex = result.size;

result = result ++ documented ++ Char.nl ++ Char.nl;

documentedClasses = List.new;

// compile list of documented classes and compare with master class list
// WAY faster than searching for individual files
          | folderPathName|
folderPathName.fullPath.containsi("help").if({
folderPathName.files.do({| item|
var nameString;
nameString = item.fileName.split($.).at(0);
if(nameString.asSymbol.asClass.notNil, {
documentedClasses.add(nameString.asSymbol.asClass);
```

```
});
});
});

folderPathName.foldersWithoutCVS.do({| folder|
addFunc.value(folder);
});
};

paths.do(addFunc);

undocumentedClasses = Class.allClasses.difference(documentedClasses);

classesStartIndex = result.size;

undocumentedClasses.do({| item|
var name;
name = item.name;
 // weed out metaclasses
name.isMetaClassName.not.if({
result = result ++ Char.nl ++ name.asString;
});
});

result = result ++ Char.nl;

      Document     "Undocumented Classes"
//doc = Document.open("Help/help-scripts/tab-template.rtf");
//doc.title = "Undocumented Classes";

doc.setFont(Font("Helvetica", 12));
doc.string = result;

doc.setFont(Font("Helvetica-Bold", 18), 0, titleString.size);

doc.selectRange(documentedIndex, documented.size);
doc.underlineSelection;

// Click on name opens class browser
doc.mouseDownAction = { arg document;
```

```
var line;
line = document.currentLine;
if((document.selectionStart > classesStartIndex) &amp;&amp; (line.size > 0), {
(line ++ ".browse").interpret
});
};

doc.selectRange(0,0);
doc.editable_(false);
{doc.removeUndo;}.defer(0.1);
```

ID: 233

12 JITLib

12.1 Environments

ID: 234

EnvironmentRedirect base class for environment redirects

superclass: Object

Environment that redirects access (**put**) and assignment (**at**).

***new(envir)** create new redirect, if envir is given it is used.
envir return the source environment
envir_ replace the source environment

Overriding the following methods, one can redirect where objects go when they are assigned to
the space: **at, put, localPut, removeAt.**
This is done for example in**[LazyEnvir]** and **[ProxySpace]**.

EnvironmentRedirect implements some of the
interface of [**Environment**], which it can replace where needed:

***push, *pop, push, pop, make, use, do, clear, keysValuesDo, keysValuesArrayDo,**
findKeyForValue, sortedKeysValuesDo, choose, <>know, doesNotUnderstand

Networking:

EnvironmentRedirect and its subclasses can be used to dispatch assignment over a network.
To do this, a **dispatch** function can be supplied - see**[Public]**.

ID: 235

LazyEnvir lazy environment

superclass: EnvironmentRedirect

Environment with deferred evaluation and default values.

put(key, val) sets the value of the reference at **key**
at(key) returns a reference to the object at **key.**
If none is present, the default value is returned (integer **1**)

Consequently, calculations can be done with nonexisting objects which can then be assigned later.

```
// examples

l = LazyEnvir.push;

// default objects are created on access
a;
          // defaults to 1

// operations on placeholders
(
c =  a +  b;

          // defaults to 2.
)



// variables can be assigned later
(
a = 800;
b = { 1.0.rand };

c.value;
```

```
)


// variables can be exchanged later
(
b = { 1000.rand };
c.value;
)
```

ID: 236

ProxySpace **an environment of references on a server**

superclass: LazyEnvir

Generally a proxy is a placeholder for something, which in this case
is something playing on a server that writes to a limited number of busses.
(this can be for example a synth or an event stream)

When accessed, ProxySpace returns a**[NodeProxy].**

The rate is determined in a lazy way from the first object put into this environment.
Once it is created it can only be set to a function that returns the same rate and a number
of channels equal to the intial one or smaller. see [**the_lazy_proxy]**

if the ugen function's number of channels is smaller, the offset in 'put' can be used to offset the ugens
if the number of channels is larger, the outputs will wrap around and mix accordingly.

```
// note that the two expressions are equivalent:
out = something;
currentEnvironment.put(\out, something);
```

a proxyspace can be created when its server is not running and played later.

#a3925a

#a3925a

#a3925a

Note:
The following examples can be executed line by line, usually in any order.
code that should be evaluated together is set in parentheses.

#a3925a

#a3925a

#a3925a

#a3925a
class methods

*new(server, name, clock)

server: a Server object. note that on remote computers the clock must be in sync
name: a symbol. if a name is given, the proxy space is **stored** in ProxySpace.all under this name.
clock: for event-based or beat-sync playing use a TempoClock.

*push(server, name, clock)

replace the currentEnvironment with a new ProxySpace and **clear** the current one, if it is a ProxySpace (this is to avoid piling up proxy spaces in performance).

In order to move to another ProxySpace while keeping the current,
use **pop** and then **push** a new one.To have multiple levels of proxy spaces,
use **.new.push;**

*pop
restore the previous currentEnvironment

instance methods

play(key)
returns a group that plays the NodeProxy at that key.
default key: \out

record(key, path, headerFormat, sampleFormat)

returns a RecNodeProxy that records the NodeProxy at that key

ar(key, numChannels, offset)

kr(key, numChannels, offset)

returns a NodeProxy output that plays the NodeProxy at that key,
to be used within a function used as input to a node proxy

wakeUp
when the proxyspace is created without a running server this method can be used
to run it (internally this is done by **play(key)** as well.

fadeTime_ set the fadetime of all proxies as well as the default fade time

clock_ set the clock of all proxies as well as the default clock.

free(fadeTime) free all proxies (i.e. free also the groups)

release(fadeTime) release all proxies (i.e. keep the groups running)

clear(fadeTime) clear the node proxy and remove it from the environment.
this frees all buses. If a fadeTime is given, first fade out, then clear.

***clearAll** clear all registered spaces

"garbage collecting":

clean(exclude)
free and remove all proxies that are not needed in order to play the
ones passed in with 'exclude'. if none are passed in, all proxies
that are monitoring (with the .play message) are kept as well as their parents etc.

reduce(to)
free all proxies that are not needed in order to play the
ones passed in with 'to'. if none are passed in, all proxies
that are monitored (with the play message) are kept as well as their parents etc.

storing

document(keys)

creates a new document with the current proxyspace state. This does not allow open functions as proxy sources. see: [jitlib_asCompileString]
keys: list of keys to document a subset of proxies

for more examples see: [proxyspace_examples] [jitlib_basic_concepts_01]

```
// examples

(
s = Server.local;
s.boot;
p = ProxySpace
)

out.play;

out = { SinOsc.ar([400, 407] * 0.9, 0, 0.2) };

out = { SinOsc.ar([400, 437] * 0.9, 0, 0.2) * LFPulse.kr([1, 1.3]) };

out = { SinOsc.ar([400, 437] * 0.9, 0, 0.2) *  x.value };

x = { LFPulse.kr([1, 1.3] * MouseX.kr(1, 30, 1)) };

out = { SinOsc.ar([400, 437] * Lag.kr(0.1 +  x, 0.3), 0, 0.2) *  x };

p.fadeTime = 5;

out = { SinOsc.ar([400, 437] * 1.1, 0, 0.2) *  x.kr(2) };
```

```
// end all in 8 sec.


   // remove all, move out.
```

12.2 Extras

ID: 237

History keeps a history of interpreted lines of code

History keeps track of all code lines that are being executed, in order to forward them to other players, to easily reuse earlier versions, or to store and reproduce a performance. Since it records everything that is interpreted, there is only one instance of History. (adc 2006)

***start / *stop** start/stop adding interpreted code to history

***clear** remove all items from history

***enter(obj)** add an entry by hand

***document** post the history in a new document

***drop(n)** drop the newest n lines from history. if n is negative, drop the oldest n lines

***keep(n)** keep only the newest n lines from history. if n is negative, keep the oldest n lines

***saveCS(path, forward)**
store the history as one compileString

***saveStory(path)** store in a file, in historical order as individual code snippets

```
// example

History.start;

a = { | freq=440|  SinOsc.ar(freq, 0, 0.2) }.play;

a.set(\freq, 450);

a.free;
```

```
History           // create a document with all the changes

History          // gui window

History       // stop collecting




// removing and adding lines

History       "2 + 2"    // make a simple entry by hand.

History           // drop the oldest memory

History          // drop the newest memory




// more examples

History.start;


         // code line gets stored

 nil             // error lines  are ignored

 // comment-only is kept, empty lines not:


         // do some things
p = ProxySpace.push;
```

```
(
test = { Blip.ar(50 * [1, 1.02], 8) * 0.1 };
test.playN;
)

test = { Blip.ar(LFNoise1.kr(3 !  2, 200, 500), 4) * 0.1 };
test = { Blip.ar(LFNoise1.kr(3 !  2, 200, 500), LFNoise1.kr(5 !  2, 5, 8)) * 0.1 };


test.vol_(0.1);

test.stop;

test.clear;

History.document;
History               // clear last line
History                   // clear last 4 lines


History             // post most recent line
History             // go back 4 lines

History.keep(4);  // keep newest 4
History


History         // save as compilestring for reloading.

History.saveCS(" /Desktop/testHist.txt", forward:  true);
   // save with special name, in forward time order.


History            // write all to file in historical order

History.saveStory(" /Desktop/myTest.sc"); // ...  with given filename.

History        // start over with a clean slate.
```

```
// To Do:
    // History Reloaded - from a saved file:
// History.loadCS(" /Desktop/testHist.sc");

  // auto-save by appending to a file ...
// History.autoSave_(true);

  //  save as one runnable task/script.
// History.saveScript;
// History.saveScript(" /Desktop/histScript.sc"); // with given filename.
```

ID: 238

SkipJacka utility for background tasks that survive cmd-.

***new(name, updateFunc, dt, stopTest)**
updateFunc the function to repeat in the background
dt the time interval at which to repeat
stopTest a test whether to stop the task now
name is only used for posting information
clock the clock that plays the task.
default is AppClock, so SkipJack can call GUI primitives.
If you need more precise timing, you can supply your own clock,
and use defer only where necessary.

```
w = SkipJack({ "watch...".postln; }, 0.5, name:  "test");
SkipJack.verbose = true; // post stop/wakeup logs

w.stop;
w.start;

//  now try to stop with cmd-.  :  SkipJack always restarts itself.
thisProcess.stop;

w.stop;

 // use stopTest:
a = 5;
w = SkipJack({ "watch...".postln; }, 0.5, { a == 10 }, "test");
a = 10; // fulfil stopTest


// Typical use:  SkipJack updates a window displaying the state
// of some objects every now and then.
   // example is mac-only
d = (a:  12, b:  24);
d.win = SCWindow("dict", Rect(0,0,200,60)).front;
d.views = [\a, \b].collect { | name, i|
SCStaticText(d.win, Rect(i * 100,0,96,20))
.background_(Color.yellow).align_(0).string_(name);
};
```

```
    SkipJack
"...".postln;
[\a, \b].do { | name, i|
d.views[i].string_(name ++ ":" + d[name])
}
},
0.5,
{ d.win.isClosed },
 "showdict"
);
)

           // updates should be displayed
d.b = \otto;
              // when window closes, SkipJack stops.

   // the same example, but written in x´cross-platform gui style:
d = (a:  12, b:  24);
d.win = GUI(\window).new("dict", Rect(0,0,200,60)).front;
d.views = [\a, \b].collect { | name, i|
GUI(\staticText).new(d.win, Rect(i * 100,0,96,20))
.background_(Color.yellow).align_(0).string_(name);
};
    SkipJack
"...".postln;
[\a, \b].do { | name, i|
d.views[i].string_(name ++ ":" + d[name])
}
},
0.5,
{ d.win.isClosed },
 "showdict"
);
)


// I prefer this 'lazy' gui idea to a dependency model:
// Even when lots of changes happen fast, you don't choke your
// cpu on gui updating, you still see some intermediate states.
```

```
// if you need to get rid of an unreachable skipjack
SkipJack({ "unreachable, unkillable...".postln }, name:  "jack");

SkipJack.stopAll // do this to stop all;

SkipJack.stop("jack"); // reach it by name and stop
```

12.3 Miscellanea

ID: 239

Just In Time Programming

"Passenger to taxtidriver: take me to number 37. I'll give you the street name when we are there."
(an austrian math teacher's joke)

Disclaimer: there is no time, really; punctuality is your personal responsibility though.

Just in time programming (or: *live coding* ¹, *on-the fly-programming*, *interactive programming* ²) is a paradigm that includes the programming activity itself in the program's operation. This means a program is not seen as a tool that is made first, then to be productive, but a dynamic construction process of description and conversation - writing code becomes a closer part of musical practice. SuperCollider, being a dynamic programming language, provides several possibilities for modification of a running program - this library attempts to extend, simplify and develop them, mainly by providing placeholders*(proxies)* that can be modified and used in calcuations while playing. There is some specific networking classes which are made to simplify the distribution of live coding activity.

Jitlib consists of a number of *placeholders* (server side and client side) and *schemes of access*.
These two aspects of space corresponding to *inclusion* and *reference,* depend on their context - here the placeholders are like roles which have a certain behaviour and can be fulfilled by certain objects.
It is useful to be aware of the three aspects of such a placeholder: a certain set of elements can be their source, they can be used in a certain set of contexts and they have a certain default source, if none is given.

Tutorial: Interactive Programming with SuperCollider and jitlib

This tutorial focusses on some basic concepts used in JITLib. There are many possibilities,
such as server messaging and pattern proxies which are not covered in tutorial form presently.

content:

Overview of the different classes and techniques:

• One way or style of access is the *'def'* classes (Pdef, Ndef etc.)
it binds a symbol to an object in a specific way:
Pdef(\name) returns the proxy
Pdef(\name, object) sets the source and returns the proxy.
the rest of the behaviour depends on its use.

client side: **[Pdef] [Pdefn], [Tdef], [Pbindef]**
server side: **[Ndef]**

• Another way, for server side NodeProxies, is an *environment* that returns placeholders on demand:
ProxySpace.push
out = { ...}
helpfile: **[ProxySpace]** for the use together with other environments, see [jitlib_basic_concepts_02]

• there is also direct access *without* using the access schemes: NodeProxy, TaskProxy etc. provide it.
internally the former use these as base classes.

client side: [**PatternProxy**], [**EventPatternProxy**], [**TaskProxy**], [**PbindProxy**], [**Pdict**]
server side: [**NodeProxy**], [**RecNodeProxy**]

• in remote and local networks thanks to sc-architecture node proxies can be used on any server,
as long as it notifies the client and has a correctly initialized default node.
note that the client id should be set.
using the network classes, groups of participants can interfere into each other's composition
by sharing a common server, using SharedNodeProxy and exchanging code and comments
in a chat (see Client)
networking classes:
experimental:
[Public] distribute environments over networks.
[public_proxy_space] how to distribute a ProxySpace
[**Client**] simplifies client-to-client networking.

stable:
[**BroadcastServer**]
[OSCBundle]

tutorials: **[proxyspace_examples]**
[jitlib_efficiency]
[the_lazy_proxy]
[jitlib_fading]
[jitlib_asCompileString]
[recursive_phrasing]
[jitlib_networking]

live coding without jitlib: **[basic_live_coding_techniques]**

storage: [NodeMap] storing control setting
[Order] ordered collection

unconnected other classes:

UGens
[TChoose]
[TWChoose]

for suggestions / comments contact me
Julian Rohrhuber, rohrhuber@uni-hamburg.de

Thanks a lot for all the feedback and ideas!

[1] see for example *http://toplap.org*
[2] *dynamic programming* would have been a good term, but it is in use for something else already
compare:
http://en.wikipedia.org/wiki/Dynamic_programming
http://en.wikipedia.org/wiki/Dynamic_programming_language
#a3925a

RECENT CHANGES:

2005
october node proxy: developments with alberto de campo:
added playN method for larger multichannel setups

getKeysValues and controlKeys

september
node proxy:
unset works (not tested for setn)
nodeproxy.at now returns source, not play control object
control rate proxies now crossfade really linearly.

NodeProxy (BusPlug) can be passed a parentGroup now.
also ProxySpace can play in a given group (proxyspace.group = ...)

may
pattern proxies: added de Campo suggestion:
pattern proxies have now an update condition. reset method, if stuck.
there is also an .endless method that will embed the proxy endlessly.

node proxy multichannel expansion with arrays of numbers works now more like expected
x = [a, b, c]; expands into 3 channels
x[0..] = [a, b, c]; expands into 3 slots (mixing the inputs)

april
networking: removed PublicProxySpace, added a Dispatcher class [**Public**] that does the same for any EnvironmentRedirect.

march
pattern proxies: all can have an environment, and take functions as arguments as well.
Tdef/TaskProxy take also a pattern as argument (for time streams).
Tdef .**fork**: fork a thread.
defaults now end (no looping by default)
nodeproxy.clear and proxyspace.have a fadeTime argument now

nodeproxy / nodemap: **mapEnvir** now takes not an array of keys, but multiple arguments
(.mapEnvir(\a, \b, \c) instead of .mapEnvir([\a, \b, \c])

febuary
improved network classes, PublicProxySpace
added envir variable to pattern proxies, the respond to set / map now.

january
nodeproxy.source now returns the sources
added tutorial

2004
december
fixed a bug when loading node proxy if the server is fresh.
node proxy now removes its synthdefs from the server.
Pdef: quant can now have the form [quant, offset]
networking:
removed Router class. see BroadcastServer helpfile.
experimental network classes added, old ones changed.

october
fixed a bug in Tdef/Pdef where the clock wasn't passed in (.play) properly when it was set before.
NodeProxy is more efficient in building SynthDefs.
ProxySynthDef can be used independent of NodeProxy now

september
refactored Pdef: new subclasses
improvements in NodeMap and NodeProxy
added role scheme (helpfiles to come)

august
nodeProxy.mapn expands to multiple controls. (not by arrays of keys anymore!)
some small efficiency improvements

july
Pdef/Tdef: pause and resume are now beat quantized
embedInStream is more efficient now, fadeTime is only applied for transitions
added recursive phrasing event
corrected Client bundle size calculation
NodeProxy remove/stop now differentiated, to allow ressource freeing in future (mainly cx players)
nodeMap_( ) fixed (unmaps synth now)
filter method for node proxy

june
removed N / Debug class to reduce number of classes.
NodeProxy/ProxySpace:
wrapForNodeProxy improved (Bus works now, NodeProxy more tolerant)
simplified object wrapping
added monitor class, separated monitoing from BusPlug
channel expansion works with .play now (converting number of channels)
removed toggle method.
onClear, monitorGroup, vol slots are removed (monitorGroup, vol is in monitor)
node order in multi object node proxy works more efficiently
multiple put: a[2..5] = { ... }
Pdef:
added fadeTime
Pdefn default is 1.0 now, which is safer for use in time patterns.
avoid occasional duplicate notes when changing pattern
Tdef is more error proof.
ProxySpace has a parent. the proto slot is used to store the tempo proxy.

bugfixes:
SynthDef-hasGateControl fixed
InBus static float index works now
bugfix in Order-detect
fix unset/unmap bug
supplementNodeMap also works for non functions.

may
_loading large proxyspaces now works - no late messages anymore
fixed bug in NodeProxy.clear when using patterns
fixed group order
added crossfade support for Patterns (EventStreams)
added crossfaded input offset (using e.g. .ar(1, offset))
removed Channel ugen, added XIn Ugens
lineAt, xlineAt messages work now
fixed the read example in [jitlib_efficiency]
nodeProxy.send now sends all if no index is given
refactoring, readability improved.

april

fixed bug on ProxySpace-reduce that caused infinite recursion in some cases
fixed bug in BinaryOpPlug (account for function.rate returning \stream)

new version of Pdef/Pdefn/Tdef

experimental:
added .storeOn, .document methods for ProxySpace. see [jitlib_asCompileString]

march
fixed Pattern support for NodeProxy/ProxySpace due to change in Event implementation
fixed Prefn
removed Penvir, Sfin etc, Ensemble etc. (removed classes can be found on sc-swiki)
simplifications

feb
added garbage collector to ProxySpace
fixed unmap
control rate proxies do not wrap channels anymore.

jan
_Pref / Prefn are tested, update properly. I'm thinking of merging them with Pdef.
fixed wakeUp bug in NodeProxy.
fixed bug in load when no server is present

embed control proxies in streams as bus index, allow multichannel mapping in patterns
added possibility to use \offset key in event streams to have offset index in multichannel proxy
fixed bug in lazy init

2003

nov
_Pdef, Tdef work now, also with NodeProxy.
NodeProxy-releaseAndFree is replaced by .end.

oct
symbol input is assumed to represent a synthdef that has a gate and frees.
functions and synthdefs are searched for their inner envelopes and the appropriate

method of freeing / releasing is chosen accordingly.

earlier:
input channels now wrap. so any number of channels can be set to any proxy.

ar and kr take a second argument for the channel offset within the proxy
checked all helpfiles

put now has a different arg order! a.put(0, obj) or a[0] = obj
prependargs were removed for now.
lag can be set by name a.lag(\freq, 1)
a shortcut was added to efficiently read from another proxy: a.read(b) or a.read([b, c])

added granular / xtexture functionality: gspawner / spawner (experimental, might change)

a.play now does not create multiple synths if evaluated repeatedly (unless multi: true)
lazy init works now.

bin/unops work fully now again like any math op.

12.4 Networking

ID: 240

BroadcastServer dispatches osc messages to multiple servers

superclass: Object (adaptor class)

***new(name, homeAddr, options, clientID)**
create a new instance. name, homeAddr, options and clientID are used for the home server's properties. The other servers are represented by their addresses (allAddr) in multiclient situation, **clientID** needs to be different with each participant!

***for(homeServer, allAddr)**
like *new, but directly pass in an already existing server.

addresses_(list of net addr)
set the addresses the server is supposed to broadcast to.
This usually should include the home address.

homeServer
return the home server, which is a server that is used for id allocation and all normal functions of a individual server model.

name
returns the name of the server. The name is always the homeServer's name extended by "broadcast"

at(index)
returns the nth web address.

wrapAt(index)
returns the nth web address, if index too large, starts from beginning.

do(function)
iterate over all addresses

```
// example


(
x = NetAddr("127.0.0.1", 57201);
y = NetAddr("127.0.0.1", 57202);

a = BroadcastServer(\broad1, x, nil, 0).addresses_( [x, y]);
b = BroadcastServer(\broad2, y, nil, 1).addresses_( [x, y]);

a.boot;
b.boot;
a.makeWindow;
b.makeWindow;
)


          "/s_new"  "default"         // create 2 synths, one on each server
          "/n_set"        "freq"        // set both their freq control
                    "/n_set"        "freq"        // set only the home server's synth control

// set them to different freqs, from a
(
a.do { arg addr;
addr.sendMsg("/n_set", 1980, "freq", 450 + 100.rand2);
}
)


// set them to different freqs, from b
(
b.do { arg addr;
addr.sendMsg("/n_set", 1980, "freq", 450 + 100.rand2);
}
)
```

```
b.sendMsg("/n_set", 1980, "gate", 0.0); // release all, from b
```

ID: 241

Client **represents a remote sclang application**

Client and **LocalClient** together represent a sender / reciever pair for sclang side osc messages.
Using SynthDef like global function definitions, **ClientFunc**, an sclang application can evaluate code on a remote sclang app.

Class Methods

***new(name, netAddr)**
returns a new instance and stores it in a global dictionary
the port is set to defaultPort, which is hardcoded presently.
if no address is passed in, localhost is used.

Instance Methods

send(key, args ...)
evaluates a client function that is stored at that key

password_ (symbol)
set the password for interpreter access

cmdName_
set the cmdName under which the client sends (default: **'/client'**)
this cmdName must be the same like the LocalClient reciever cmdName

interpret(string)
if the remote client has the same password, it interprets the string

LocalClient represents a listener to a remote sclang application

superclass: Client

Note that passing a **nil address** to LocalClient will make it respond to **any** remote client and try to match any message that is sent from a client object.
If it is expected to listen to a specific remote client, the address of that client should be used.

Instance Methods

start
start listening to the world

stop
stop listening to the world

remove
remove from client list

isListening
returns whether it is listening to the world

password_ (symbol)
set the password for interpreter access from outside

cmdName_
set the cmdName under which the client recieves (default: **'/client'**)
this cmdName must be the same like the Client sender cmdName

allowInterpret
open the interpreter to the world (potential hacking danger)

disallow
close the interpreter access

ClientFunc similar to SynthDef - represents a client side stored function

*new(name, func)

global function that is accessed by LocalClient when a message is recieved.
the key sent is a key of the ClientFunc that is evaluated.
the other args are passed to the function: time, responder, args...

Note:

for accessing a gui element or a document from an OSCResponder such as the one in LocalClient, one has to defer the action:

ClientFunc(\ok, { defer({ ... }) });

```
// example
// instantiate a remote-local pair (in this case both are local of course)
    LocalClient           // this one listens to any remote client and evaluates the functions.
    Client       // this one sends the messages

// eqivalent to the above defaults:
    LocalClient \default  nil   //addr is nil :  listen to all
b = Client(\default, NetAddr("127.0.0.1", 57120));

// store some client functions to be accessible from outside (analogous to SynthDef)
ClientFunc(\ok, { arg ...  args; args.postln });
ClientFunc(\yes, { arg ...  args; \ok2.postln });

// start the local client
a.start;

// send messages
b.send(\ok, "rrere", 39);
b.send(\ok, "rrxxxre");
```

```
b.send(\ok, 2, 3, 4, 5, 6);
b.send(\yes, "rrere", 39);
b.send(\yes);
```

opening remote interpreter access is risky, because
anyone can access the interpreter (also unix commands)
if you do not set the password, this is not possible.

```
// open interpreter
a.password = \xyz;
b.password = \xyz;
a.allowInterpret;

// remote interpret
            " Array.fill(8, { [1,0].choose }).postln "
            " String.fill(80, { [$u, $n].choose }).postln"

// remote GUI
            " SCWindow.new(\"                                    ").front;{ SinOsc.ar(500, 0, LFPulse.kr(4))
}.play;"

// close interpret
a.disallow

//test:  this should not interpret
            " String.fill(8, { [$u, $n].choose }).postln"

        //stop local responder
```

writing a chat

```
(
// hit tab for sending
var n, d, e, b;
    "John"
d = Document("chat").background_(Color.rand).bounds_(Rect(30, 10, 400, 200));
```

```
e = Document("chat-write").background_(Color.rand).bounds_(Rect(30, 210, 400, 50));


a = LocalClient.default.start;
b = Client.new;
ClientFunc(\chat, { arg str; { d.string =  d.string ++ str ++ "\n" }.defer });


e.keyDownAction_({ arg doc, char;
var string;
if(char === Char.tab)
{
string = n + ":" + e.currentLine;
b.send(\chat, string.copy);
   AppClock                          ""    // time is the original crime.  remove the tab.
}
});
e.onClose_({ AppClock.sched(0.1, { a.remove; d.close; nil }) }); // sched, otherwise sc crashes
)
```

ID: 242

Public dispatch system

superclass: EnvirDispatch

experimental.

Dispatchers like Public can be used for any EnvironmentRedirect, such as LazyEnvir and ProxySpace. They cause a mirroring of parts of the environment to multiple locations. This done by**sending the code itself**, which is very flexible and leightweight but it also means that one has to be **careful** not to do harm to the other systems. Code is only sent if it interprets without error on the sender side. Timing is still not yet synchronized, although it works pretty well for not too costly code.

see also: **[public_proxy_space]**

class methods:

***new(envir)** create a new instance (with an *EnvironmentRedirect*)

***all** a dictionary of all available dispatchers. dispatchers with the same name send to and receive from each other (alternative: provide a sendToName)

***startListen(addr)** start to receive messages from the network
addr: whom to listen to. **nil**: listen to all. (default)

***stopListen** stop listening

instance methods:

addresses_(list) a list of addresses (NetAddr) to which to send to.
this list can contain also the sender's address, which it then sends to,
but does not get evaluated.

sendingKeys_(list) keys from which to send (list of symbols). If **nil**, do not send, if **all**, send to all.

listeningKeys_(list) keys at which to receive (list of symbols). If **nil**, do not send, if **all**, send to all.

put(key, obj) put an object in the space (see superclass).
if this key is sending, send object to all. object must be reproducible
as a **compileString!**. (closed functions, patterns with no open functions)

at(key) returns an object from the space at that key (see superclass).

join(channel, nickname) join a channel with a nickname

leave leave the channel

public_(bool) if public is set to **false**, no broadcasting happens.

basicSafety_(bool) if true (default), the "worst" commands are rejected - like unixCmd etc.

logSelf_(bool) if logSelf is set to true, my own changes are passed into
the action function (e.g. the log window)

logAll_(bool) if logAll is true, I can see all messages that are coming in,
even if they do not effect me (listeningKeys != \all).
This is only allowed if sendingKeys are set to \all

channel_(name) set / get channel name

nickname_(string) set / get nickname

action_(func) action to be evaluated when receiving a message (optional) function
args: *dispatch, nickname, key, receivedString*

note: if you want to call the os from this action (e.g. for GUI), you need to use defer { }

makeLogWindow create a log window.

lurk / boss / merge change behaviour diametrically (just try it out)

```
// example



(
var addresses;

Public              // start an osc responder to dispatch the messages
             NetAddr "127.0.0.1"           // this is loopback for now.  Port must be 57120 (sc-lang)

    Public      // create 2 new instances (one "remote" one "local")
e = Public.new;
                         // set the addresses - this can be done at any time laterto add new ones.
e.addresses = addresses;

       \waitingroom  \eve   // join a channel, provide a unique nickname.
       \waitingroom  \ade
                \all   // if keys are set to \all, the spaces are enitirelly open.
d.sendingKeys = \all;
d.listeningKeys = \all;
e.listeningKeys = \all;

// create two new environment redirect (works basically like an environment)

    EnvironmentRedirect
    EnvironmentRedirect

// set their dispatch variables.  the envir is registered in the dispatch implicitly

a.dispatch = d;
b.dispatch = e;
```

```
)

(
                // see what is going on
d.makeLogWindow;
)



// using the environment

a[\x] = 5;
  \x   // 5 is in b now as well.

b[\x] = { 1.0.rand };
a[\x].postcs;

a[\x] = Array.rand(20, 0, 10);
b[\x];

  \x    "hi adam"

  \x    "hi eve"


// more to come...

a.clear;
b.clear;
```

ID: 243

public ProxySpace distributed system

this is an example how to create a networked proxyspace.

Using a dispatch such as **Public**, a changes to a **ProxySpace** can be transmitted by code to a remote (or local) other ProxySpace and is compiled there as well. This is very flexible and leightweight but it also means that one has to be careful not to do harm to other systems. Code is only sent if it interprets without error on the sender side. Timing is still not yet synchronized, although it works pretty well for not too costly code.

see **[Public]** help for more about the configuration.

```
// example



(
var addresses;

Public.startListen;
addresses = [NetAddr("127.0.0.1", 57120)];

    ProxySpace
    ProxySpace

d = Public(a);
e = Public(b);

a.dispatch = d;
b.dispatch = e;


d.addresses = addresses;
e.addresses = addresses;
```

```
        \waitingroom  \eve
        \waitingroom  \ade
e.sendingKeys = \all;
d.sendingKeys = \all;
d.listeningKeys = \all;
e.listeningKeys = \all;
)

(
                // see what is going on
e.makeLogWindow;
)

// modify space

        // boot server


  \out        // play here
a[\out] = { PinkNoise.ar(0.1 !  2) }; // set here
  \out        // play here too
b[\out] = { SinOsc.ar(rrand(300, 400)) !  2 * 0.1 }; // two different tones
b[\out] = { SinOsc.ar(300) !  2 * 0.1 }; // same tone
d.public = false; // be private
b[\out] = { PinkNoise.ar(0.1 !  2) };
d.public = true;

// you can also enter the space:
a.push;
out.free;
a.pop; // exit



// now you can type into both of them, just as in examples in ProxySpace.help:
(
var str;
      "\n  out = { LFNoise2.ar(3000 + exprand(1.0, 2000), 0.1) }; "  // example string
EnvirDocument(a, "a", str);
EnvirDocument(b, "b", str);
```

)

ID: 244

SharedProxySpace distributed system

experimental.

superclass: ProxySpace

***new(broadcastServer, name, clock)**
return a new instance. the server should be a fresh, unused **BroadcastServer**

***push(broadcastServer, name, clock)**
return a new instance and make it the currentEnvironment

broadcast
return the broadcast server

server
return the home server

makeSharedProxy(key, rate, numChannels)
create a SharedNodeProxy with a group id that is derived from the
key: only short strings work. This should be done with a fresh, unused server.

addSharedKeys(controlKeys, audioKeys, firstAudioKey)
create multiple SharedNodeProxies.
This should be done with a fresh, unused server
controlKeys: names of the control rate proxies
audioKeys: names of the audio rate proxies
firstAudioKey: if the above are not given, generate them automatically.
using the letters a-z.
this is a character (default: $s) of the first audio key in alphabet

```
// examples:

// prepare and boot two servers
(
x = NetAddr("127.0.0.1", 57201);
y = NetAddr("127.0.0.1", 57202);

a = BroadcastServer(\B1, x, nil, 0).allAddr_( [x, y]);
b = BroadcastServer(\B2, y, nil, 1).allAddr_( [x, y]);

a.boot;
b.boot;
a.makeWindow;
b.makeWindow;
)



(
    SharedProxySpace        "Elizabeth"
    SharedProxySpace        "George"

// this has to be done with the fresh servers:
p.makeSharedProxy(\ensemble, \audio, 1);
r.makeSharedProxy(\ensemble, \audio, 1);
)

        \ensemble   // returns a shared node proxy.


// Elizabeth pushes her proxspace:
p.push;
currentEnvironment === p;

           // play locally

// play sine tone with freq dependant on ensemble, but a little random.
out = { SinOsc.ar(440 + (300 *  ensemble.ar) + Rand(0, 10), 0, 0.1) };
               // tune up a bit;
```

```
ensemble = { Line.ar(0, 2, 10) };


// Henry pushes his proxyspace
r.push;
out.play;
out = { Ringz.ar(Impulse.ar(9, 0, 0.8),  ensemble.ar * 300 + 400, 0.2) };
                // now both depend on the "same" bus.
ensemble = { SinOsc.ar(0.4, 0, 0.6) };

// Elizabeth

p.push;

freq = 300 *  ensemble + 400;
out = { SinOsc.ar( freq.ar * [1, 1.02], SinOsc.ar( freq.ar, 0, pi)) * 0.1 };

          // stop only locally

// Henry

r.push;
           // stop only locally


// finish:
r.clear;
p.clear;


// quit the servers
a.quit;
b.quit;
```

12.5 Nodeproxy

ID: 245

BusPlug a listener on a bus

superclass: AbstractFunction

a superclass to node proxy that listens to a bus.

it is mainly a basic subclass of **NodeProxy**, but it can be used as well for other things. for most methods see NodeProxy.help.

monitor

returns the current monitor (see [Monitor])

```
//using as a control bus listener

s.boot;
z = Bus.control(s, 16);
a = BusPlug.for(z);

m = { Mix(SinOsc.ar(a.kr(16), 0, 0.1)) }.play;

z.setn(Array.rand(16, 300, 320).put(16.rand, rrand(500, 1000)));
z.setn(Array.rand(16, 300, 320).put(16.rand, rrand(500, 1000)));
z.setn(Array.rand(16, 300, 320).put(16.rand, rrand(500, 1000)));

m.free;


m = { SinOsc.ar(a.kr(2, MouseX.kr(0, 19)), 0, 0.1) }.play; //modulate channel offset

z.setn(Array.rand(16, 300, 1320).put(16.rand, rrand(500, 1000)));


m.free; z.free;

//using as a audio monitor
```

```
p = BusPlug.audio(s,2);
d = { Out.ar(p.index, PinkNoise.ar([0.1, 0.1])) }.play;


        //monitor whatever plays in p (the execution order does not matter)



d.free;
d = { Out.ar(p.index, PinkNoise.ar([0.1, 0.1])) }.play;

p.stop;
p.play;

//also p can play to another bus:

p.stop;
p.play(12);

//listen to that bus for a test:
x = { InFeedback.ar(12,2) }.play;
x.free;
```

ID: 246

Monitor link between busses

superclass: Object

play(fromIndex, fromNumChannels, toIndex, toNumChannels, target, multi, volume, fadeTime)
plays from a bus index with a number of channels to another index with a number of channels, within a target group, or a server.
multi: keep old links and add new one
volume: volume at which to monitor
fadeTime: fade in fade out time

isPlaying returns true if the group is still playing

stop(fadeTime)
stops within the fadeTime

vol_ set the volume
out_ set the output index. doesn't work right now.

playToBundle(bundle, ... (same as .play))
adds all playing osc messages to the bundle. the bundle should support the message**.add**

```
//example

Server.default = s = Server.internal;
s.boot;
s.scope(16);

{ Out.ar(87, SinOsc.ar(MouseX.kr(40, 10000, 1) * [1, 2, 3], 0, 0.2)) }.play;
x = Monitor.new;
x.play(87, 3, 1, 2);
x.out = 0;
x.stop(3.0);
```

```
 x.play(87, 1, 0, 1); // in > out :  now mixes down (wrapping)
 x.play(89, 1, 0, 2); // in < out :  now distributes to 2 channels
 x.stop;

 // multiple play
x.play(87, 1, 0, 2, multi:true);
x.play(88, 1, 0, 2, multi:true);
x.play(89, 1, 0, 2, multi:true);
x.stop;


 // experimental, might change:
  // multichannel method
 // playN(outs, amps, ins, mastervol, fadeTime, group)
 // outs, amps can be nested arrays

x.playN([0, 1, 4], [0.1, 0.4, 0.3], [87, 88, 89]); // play:  87 -> 0,  88 -> 1, 89 -> 4
x.playN([0, [1, 3, 2], 4], [0.1, [0.4, 0.2, 0.1], 0.3], [87, 88, 89]);
  // play:  87 -> 0,  88 -> [1, 3, 2],  89 -> 4
  // volumes 0.1, [0.4, 0.2, 0.1], and 0.3
 x.playN(vol:0.0, fadeTime:4);
```

ID: 247

Ndef node proxy definition

superclass: NodeProxy

reference to a proxy, forms an alternative to ProxySpace

Ndef(key) returns the instance, Ndef(key, obj) stores the object and returns the instance, like Tdef and Pdef.
see also **ProxySpace**, **NodeProxy**

*new(key, object)
craete a new node proxy and store it in a global dictionary under key.
if there is already an Ndef there, replace its object with the new one.
The object can be any supported class, see NodeProxy help.
if key is an association, it is interpreted as **server -> key.**

*clear
clear all proxies

*at(server, key)
return an instance at that key for that server

defaultServer_(a server)
set the default server (default: Server.local)

```
// examples

s = Server.local.boot;

Ndef(\sound).play;
Ndef(\sound).fadeTime = 1;
Ndef(\sound, { SinOsc.ar([600, 635], 0, SinOsc.kr(2).max(0) * 0.2) });
Ndef(\sound, { SinOsc.ar([600, 635] * 3, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
Ndef(\sound, { SinOsc.ar([600, 635] * 2, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
```

```
Ndef(\sound, Pbind(\dur, 0.17, \freq, Pfunc({ rrand(300, 700) })) );

Ndef(\lfo, { LFNoise1.kr(3, 400, 800) });
Ndef \sound      \freq  Ndef \lfo
Ndef(\sound, { arg freq; SinOsc.ar([600, 635] + freq, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
Ndef(\lfo, { LFNoise1.kr(300, 400, 800) });

Ndef        //clear all
```

recursion:

Ndefs can be used recursively.
a structure like the following works:

```
Ndef(\sound, { SinOsc.ar([600, 635], Ndef(\sound).ar * 10, LFNoise1.kr(2).max(0) * 0.2) });
Ndef(\sound).play;
```

this is due to the fact that there is a feedback delay (the server's block size), usually 64 samples,
so that calculation can reiterate over its own outputs.

ID: 248

NodeMap

object to store control values and bus mappings independant of of a specific node.

set(key1, value1, ...) set arguments of a node
map(key1, busindex1, ...) set bus mappings of a node
unset(key1, key2, ...) remove settings
unmap(key1, key2, ...) remove mappings
setn(key1, valueArray1, ...) set ranges of controls
mapn(key1, busindex1, numChan...)
map num busses mappings to node

at(index) return setting at that key.
sendToNode(aTarget, latency) apply a setting to a node by sending a bundle
send(server, nodeID, latency) apply a setting to a node by sending a bundle
addToBundle(aBundle) add all my messages to the bundle

```
s.boot;

(
SynthDef "modsine"
{ arg freq=320, amp=0.2;
Out.ar(0, SinOsc.ar(freq, 0, amp));
}).send(s);
SynthDef "lfo"
{ arg rate=2, busNum=0;
Out.kr(busNum, LFPulse.kr(rate, 0, 0.1, 0.2))
}).send(s);
)

//start nodes
(
b = Bus.control(s,1);
    Synth "modsine"
```

```
y = Synth.before(x, "lfo", [\busNum, b.index]);
)

//create some node maps
(
h = NodeMap.new;
h.set(\freq, 800);
h.map(\amp, b.index);

k = NodeMap.new;
k.set(\freq, 400);
k.unmap(\amp);
)

//apply the maps

                //the first time a new bundle is made
k.sendToNode(x);

                //the second time the cache is used
k.sendToNode(x);

h.set(\freq, 600);

                //when a value was changed, a new bundle is made

//free all
x.free; b.free; y.free;
```

ID: 249

NodeProxy **a reference on a server**

superclass: BusPlug

Generally a proxy is a placeholder for something, which in this case
is something playing on a server that writes to a limited number of busses.

(this can be for example a synth or an event stream). The rate and number
of channels is determined either when the instance is created (.control/.audio)
or by lazy initialisation from the first source.[**the_lazy_proxy**]
These objects can be replaced, mapped and patched and used for calculation.

ProxySpace returns instances of NodeProxy. all the examples below apply to ProxySpace accordingly:

a = NodeProxy(s) is equivalent to a;
a.source = ... is equivalent to a = ...
a[3] = ... is equivalent to a[3] = ...

see also: [**jitlib_efficiency**]

note that NodeProxy plays on a private bus.
if you want to hear the output, use p.**play** and p.**stop**.
free only the inner players: p.**free**
for free inner players and stop listen: p.**end**
entirely removing all inner settings: p.**clear**

instance creation

***new(server)**
***audio(server, numChannels)**
***control(server, numChannels)**

reading from the bus

play(index, numChannels, group, multi)
play output on specified bus index (default: public busses)
this works like a monitor.
if multi is set to true it can create multiple monitors
stop(fadeTime)
stop to play out public channels (private channels keep playing as others might listen still)
this stop the monitoring. to stop the objects playing, use **free, release**
fadeTime: decay time for this action

end(fadeTime)
releases the synths and stops playback
fadeTime: decay time for this action

ar(numChannels)
kr(numChannels)

return a link to my output, which is limited by [numChannels]
causes an uninitialized proxy to create a matching bus.

normally ar defaults to stereo, kr to mono. this can be set in the classvars:
defaultNumAudio, defaultNumControl

supported inputs

```
NodeProxy     played by reading from
Function     interpreted as UGenFunc
SimpleNumber    used to write to bus continously
Bus      reads from that bus
SynthDef     plays a synth from the def
Symbol     plays a synth from the def with this name
```

```
Pattern     played as event pattern
Stream     played as event stream
nil      removes all objects
Pdef      played like a stream
Task      played, no output assigned
Tdef      played like Task
AbstractPlayer   started in a separate bus, mapped to this bus
Instr     converted to player and started

Associations:
(\filter -> func)   filter previous input
(\set -> event pattern)  set controls
```

setting the source:

source_(anObject)
play a new synth through me, release old one.
anObject can be one of the supported inputs (see above)
[only if the used synthdef (applies also to patterns) has the right number of channels and an out argument, this can be used to do filtering.
if you supply a gate, the nodeProxy will assume doneAction 2 and fade out].

add(anObject, channelOffset, extraArgs)
play a new synth, add it to the present ones

removeAt(index)
remove the synth at index i and its player definition

removeLast
remove the last synth and its player definition

put(index, anObject, channelOffset, extraArgs)
set the source by index.
index:
where the object should be placed in the internal order.
if **-1**, all objects are freed
anObject:
can be a Function, an Instr, any valid UGen input

a pattern can be used if it returns an EventStream.
channelOffset:
using a multichannel setup it can be useful to set this.
when the objects numChannels is smaller than the proxy

extraArgs: extra arguments that can be sent with the object directly (not cached)

put can be used as array indexing: **a[0]** = { SinOsc.ar }
one can put an object at any index, only the order of indices is relevant.
if the index equals an existing index, the object at this index is replaced.
using multiple index expands into multiple objects: **a[0..3]** = ... or **a[[0, 4, 6]]** = [.., ..., ..]

pause
pause all objects and set proxy to paused

resume
if paused, start all objects

group-like behaviour:

set(key, val, ...)
I behave like my nodeMap: see [NodeMap]
set, setn, unset, unmap

map(key(s), proxy, ...)
map the arguments in keys to the subsequent channels of a **control proxy**
(keys can be a symbol or a number)
if the proxy has **multiple channels**, subsequent channels of the control,
if present, are mapped (mapn)
note that you should not map to more channels than the control has.

setn(key, list, ...)
set ranges of controls

run(flag)

pause/unpause all synths in the group

extended group-like behaviour:

xset(key, val, ...)
set with crossfade into new setting

x**map(keys, proxy)**
map with crossfade into new setting

xsetn()
untested

lag(key, val, ...)
set the lag values of these args (identical to setRates)
to remove these settings, use: **lag(\key1, nil, key2, nil, ...)**

setRates(key, rate1, ...)
set the default rate (\tr, \ir, numerical) for synthDef arg
rate of **nil** removes setting

bus-like behaviour:

line(value, dur)
set my bus to the new value in dur time linearly
x**line(value, dur)**
set my bus to the new value in dur time exponentially
gate(value, dur)
gate my bus to the level value for dur time

// do not work properly yet !
lineAt(key, value, dur)
set the control value to the new value in dur time linearly
x**lineAt(key, value, dur)**
set control value to the new value in dur time exponentially

gateAt(key, value, dur)
gate my control to the level value for dur time.
if the control was not set before, stay at the new value

sending synths to server
(normally the source_() message does the sending already, but it can be used for spawning)

wakeUp
until the proxy is not used by any output (either .play or .ar/.kr)
it is not running on the server. you can wake it up to force it playing.
normally this is not needed.

send(argList, index, freeLast)
send a new synth without releasing the old one.
the argument list is applied to the synth only.
freeLast: if to free the last synth at that index
if index is nil, sends all

sendAll(argList, freeLast)
send all synths without releasing the old one.
the argument list is applied to all synths.
freeLast: if to free present synths

release and cleaning up:

free(fadeTime)
release all my running synths and the group
fadeTime: decay time for this action

release(fadeTime)
release running synths
fadeTime: decay time for this action

clear(fadeTime)
reset everything to nil, neutralizes rate/numChannels
if a fadeTime is given, first fade out, then clear.

setting properties:

fadeTime_(time)
set the attack/release time

clock_(aClock)
use a tempo clock for scheduling beat accurate

misc:

record(path, headerFormat, sampleFormat)
record output to file (returns a [RecNodeProxy] that you can use for control)
returns a [RecNodeProxy]

***defaultNumAudio_(n)**
set the default channel number for audio busses

***defaultNumControl_(n)**
set the default channel number for control busses

---

for more examples see [ProxySpace]

```
// examples
s = Server.local;
s.boot;
```

using node proxy with ugen functions

```
a = NodeProxy.audio(s, 2);
        // play to hardware output, return a group with synths

// setting the source
a.source = { SinOsc.ar([350, 351.3], 0, 0.2) };

// the proxy has two channels now:
a.numChannels.postln;
a.source = { SinOsc.ar([390, 286] * 1.2, 0, 0.2) };

// exeeding channels wrap:
a.source = { SinOsc.ar([390, 286, 400, 420, 300] * 1.2, 0, 0.2) };

// other inputs
a.source = { WhiteNoise.ar([0.01,0.01]) };
a.source = 0;
           \default  // synthDef on server
a.source = SynthDef("w", { arg out=0; Out.ar(out,SinOsc.ar([Rand(430, 600), 600], 0, 0.2)) });
           nil  //  removes any object

// feedback
a.source = { SinOsc.ar(a.ar * 7000 * LFNoise1.kr(1, 0.3, 0.6) + 200, 0, 0.1) };
a.source = { SinOsc.ar(a.ar * 6000 * MouseX.kr(0, 2) + [100, 104], 0, 0.1) };

// fadeTime
a.fadeTime = 2.0;
a.source = { SinOsc.ar([390, 286] * ExpRand(1, 3), 0, 0.2) };


// adding nodes
a.add({ SinOsc.ar([50, 390]*1.25, 0, 0.1) });
a.add({ BrownNoise.ar([0.02,0.02]) });

// setting nodes at indices:
a[0] = { SinOsc.ar( 700 * LFNoise1.kr(1, 0.3, 0.6) + 200, 0, 0.1) };
a[1] = { LFPulse.kr(3, 0.3) * SinOsc.ar(500, 0, 0.1) };
```

```
a[2] = { LFPulse.kr(3.5, 0.3) * SinOsc.ar(600, 0, 0.1) };
a[3] = { SinOsc.ar([1,1.25] * 840, 0, 0.1) };

// filtering:  the first argument is the previous bus content.  more args can be used as usual.
a[3] = \filter -> { arg in; in * SinOsc.ar(Rand(100,1000)) };
a[2] = \filter -> { arg in; in * MouseY.kr(0,1) };
a[8] = \filter -> { arg in; in * MouseX.kr(0,1) };
a[4] = \filter -> { arg in; in * SinOsc.ar(ExpRand(1,5)).max(0) };



// setting controls
a.fadeTime = 2.0;
a.source = { arg f=400; SinOsc.ar(f * [1,1.2] * rrand(0.9, 1.1), 0, 0.1) };
a.set(\f, rrand(900, 300));
a.set(\f, rrand(1500, 700));
a.xset(\f, rrand(1500, 700)); // crossfaded setting
a.source = { arg f=400; RLPF.ar(Pulse.ar(f * [1,1.02] * 0.05, 0.5, 0.2), f * 0.58, 0.2) };

// control lags
a.fadeTime = 0.01;
      \f        // the objects are built again internally and sent to the server.
a.set(\f, rrand(1500, 700));
a.lag(\f, nil);
a.set(\f, rrand(1500, 700));
a.fadeTime = 1.0;

// mapping controls to other node proxies

c = NodeProxy.control(s, 2);
c.source = { SinOsc.kr([10,20] * 0.1, 0, 150, 1300) };
a.map(\f, c);
a[0] = { arg f=400; RHPF.ar(Pulse.ar(f * [1,1.2] * 0.05, 0.5, 0.2), f * 0.58, 0.2) };
c.source = { SinOsc.kr([10,16] * 0.02, 0, 50, 700) };
c.source = { Line.kr(300, 1500, 10) + SinOsc.kr(20 * [1,2], 0, 100) };
a[1] = { arg f; LFPar.ar(f % MouseX.kr(1, 40, 1) * 4 + 360, 0, 0.2) };

// map multiple channels of one proxy to multiple controls of another
// recently changed behaviour!
```

```
a.source = { arg f=#[400, 400]; LPF.ar(Pulse.ar(f[0] * [0.4,1], 0.2, 0.2), f[1] * 3) };
      \f      // multichannel proxy c is mapped to multichannel control of a
a.source = { arg f=#[400, 400]; LPF.ar(Pulse.ar(f, 0.2, 0.2), f[1]) };
a.source = { arg f=#[400, 400]; Formant.ar(140, f * 1.5, 100, 0.1)  };
c.source = { SinOsc.kr([Line.kr(1, 30, 10), 1], 0, [100, 700], [300, 700]) };
c.source = 400;


c.fadeTime = 5.5;
c.source = { LFNoise0.kr([2.3, 1.0], [100, 700], [300, 1700]) };
c.source = { SinOsc.kr([2.3, 1.0], 0, [100, 700], [300, 1700]) };
c.source = 400;


// behave like a sc2 plug
c.gate(1400, 0.1);
c.gate(1000, 0.1);
c.line(1000, 1);

// direct access
a.lineAt(\f, 300, 2);
a.xlineAt(\f, 600, 0.3);
a.gateAt(\f, 1600, 0.3);


// changing nodeMaps
a.unmap(\f);
n = a.nodeMap.copy;
n.set(\f, 700);
a.fadeToMap(n);
n = a.nodeMap.copy;
n.set(\f, 400);
a.fadeTime = 1.0;
               \f    // linear interpolation to new map:  experimental
      \f      // restore mapping


// sending envelopes (up to 8 levels)
w = Env.new(Array.rand(3, 400, 1000),Array.rand(2, 0.3, 0.001), -4);
c.env(w);
```

```
c.env(w);
w = Env.new(Array.rand(8, 400, 1000),Array.rand(7, 0.03, 0.1));
c.env(w);
c.env(w);

// stop synthesis, then wake up proxies:

        // stop the monitor
        // start the monitor
       // release the synths and stop the monitor
         // free the control proxy c
```

channel offset/object index

```
a = NodeProxy.audio(s,2);
a.play;
a[0] = { Ringz.ar(Impulse.ar(5, 0, 0.1), 1260) };
a.put(1, { Ringz.ar(Impulse.ar(5.3, 0, 0.1), 420) }, 1);
a.put(0, { Ringz.ar(Dust.ar([1,1]*15.3,  0.1), 720) }, 1);
a.put(1, { Ringz.ar(Impulse.ar(5.3, 0, 0.1), 420) }, 1);
a.end;
```

beat accurate playing

```
a = NodeProxy.audio(s,2);
a.play;

a.clock = TempoClock(2.0).permanent_(true); // round to every 2.0 seconds
a.source = { Ringz.ar(Impulse.ar(0.5, 0, 0.3), 3000, 0.01) };
a[1] = { Ringz.ar(Impulse.ar([0.5, 1], 0, 0.3), 1000, 0.01) };
a[2] = { Ringz.ar(Impulse.ar([3, 5]/2, 0, 0.3), 8000, 0.01) };
a[3] = { Ringz.ar(Impulse.ar([3, 5]*16, 0, 0.3), 5000, 0.01) * LFPulse.kr(0.5, 0, 0.05) };

a.removeLast;
a.removeAt(2);
```

```
a.clear;
```

using patterns - event streams

```
(
// must have 'out' or 'i_out' argument to work properly
SynthDef("who", { arg freq, gate=1, out=0, ffreq=800, amp=0.1;
var env;
env = Env.asr(0.01, amp, 0.5);
Out.ar(out, Pan2.ar(
Formant.ar(freq, ffreq, 300, EnvGen.kr(env, gate, doneAction:2)), Rand(-1.0, 1.0))
)
}).store;

)


(
s.boot;
a = NodeProxy.audio(s, 2);
a.fadeTime = 2;
b = NodeProxy.audio(s,2);
b.fadeTime = 3;
)

        // monitor output

// play the pattern silently in b
b.source = Pbind(\instrument, \who, \freq, 500, \ffreq, 700, \legato, 0.02);

// play b out through a:
a.source = b;

// filter b with ring modulation:
a.source = {  b.ar  * SinOsc.ar(SinOsc.kr(0.2, 300, 330))  }; // filter the input of the pattern
a.source = {  b.ar * LFCub.ar([2, 8], add:  -0.5)  }; // filter the input of the pattern
```

```
a.source = b;

// map b to another proxy
c = NodeProxy.control(s, 1).fadeTime_(1);
c.source = { SinOsc.kr(2, 0, 400, 700) };


// now one can simply embed a control node proxy into an event pattern.
// (this works not for \degree, \midinote, etc.)
// embedding in other patterns it will still return itself.


b.source = Pbind(\instrument, \who, \freq, 500, \ffreq, c, \legato, 0.02);

c.source = { SinOsc.kr(SinOsc.kr(0.2, 0, 10, 10), 0, 400, 700) };

c.source = { LFNoise1.kr(5, 1300, 1500) };
c.source = { MouseX.kr(100, 5500, 1) };

(
b.source = Pbind(
   \instrument  \who
\freq, Pseq([600, 350, 300],inf),
\legato, 0.1,
\ffreq, Pseq([c, 100, c, 100, 300, 600], inf), // use proxy in a pattern
\dur, Pseq([1, 0.5, 0.75, 0.25] * 0.4, inf),
\amp, Pseq([0.2, 0.2, 0.1, 0.1, 0.2], inf)
);
)



b[2] = Pbind(\instrument, \who, \freq, 620, \ffreq, Prand([500,c],inf), \legato, 0.1, \dur, 0.1);
b[3] = Pbind(\instrument, \who, \ffreq, 5000, \freq, Pseq([720, 800],inf), \legato, 0.1, \dur, 0.1, \amp,
0.01);
b[4] = Pbind(\instrument, \who, \freq, Pseq([700, 400],inf), \legato, 0.1, \ffreq, 200);
b[1] = { WhiteNoise.ar([0.01,0.01]) };
b[4] = { arg ffreq=800; Resonz.ar(WhiteNoise.ar([1,1]), ffreq, 0.05) };
```

```
	\ffreq	// map the control to the proxy
b.removeLast;
b.removeLast;
a.source = {  b.ar * WhiteNoise.ar(0.1, 1)  };
a.source = {  b.ar * WhiteNoise.ar(0.1, 1) + (b.ar * SinOsc.ar(SinOsc.kr(0.01, 0, 50, 330)))  };

c.source = { XLine.kr(1900, 10, 10) };

a.clear; b.clear; c.clear; // clear all, free bus
```

ID: 250

Order an order of elements with an integer index

superclass: SequenceableCollection

keeps elements in an order.

put and at are slower than IdentityDictionary/PriorityQueue, do is faster.

inherits all methods from superclass.

the following messages change the content of the collection without returning a new one.
apart from this they work like collect/reject/select

collectInPlace(func)

selectInPlace(func)

rejectInPlace(func)

```
//example

a = Order.new;

a[0] = \z;
a[0] = \y;
a[5] = \five;
a[4] = \four;

a.collectInPlace({ arg item, i; 700 + i });
a[0] = \z;
a[5] = \five;
a[4] = \four;

a.indices;
a.selectInPlace({ arg item; item.asString[0] === $f });
```

```
a.indices;

a[9] = 100;
a.rejectInPlace({ arg item; item.isNumber.not });
a.indices;
```

ID: 251

ProxySynthDef synth def that wraps ugen graph

superclass: SynthDef

***new(name, func, rates, prependArgs, makeFadeEnv, channelOffset=0, chanConstraint)**
name, func, rates, prependArgs: like in SynthDef.new
todo: add variants.

makeFadeEnv
if true it constructs a fader envelope and adds controls for gate and fadeTime
channelOffset
a constant offset that is added to the out number
chanConstraint
max numChannels for the synthdef. If ugenfunc returns a larger array, it wraps

***sampleAccurate_**
use OfsetOut, if set to true (default: false)

for inner workings see [**jitlib_fading**]

```
// example


a = ProxySynthDef("xtest", { SinOsc.ar(400) * 0.1 });

a.send(s);

x = Synth("xtest");
x.release;


/*
```

```
if the resulting number of channels is larger than a given channelConstraint,
it behaves according to the rate:  audio rate signals are wrapped around
a smaller channel size, control rate signals are not (the exceeding channels are left out)


*/
```

ID: 252

RecNodeProxy

superclass: NodeProxy

a node proxy that can record

instance creation:

***new(server) / *audio(server, numChannels)**
see superclass

***newFrom(inproxy, numChannels)**
instantiate a new proxy that listens to the in proxy.

access:

open(path, headerformat, sampleformat)
open new file and initialize buffer on server

record(paused)
start the recording synth, if paused is false start recording immediately
default: true

close
stop recording, close file

pause/unpause
pause recording / unpause recording

isRecording
see if recording right now

wakeUp
until the proxy is not used by any output (either .play or .ar/.kr)
it is not running on the server. you can wake it up to force it playing.

---

examples

```
s = Server.local;
s.boot;

a = RecNodeProxy.audio(s, 2);
a.source = { SinOsc.ar([400,500], 0, 0.1) };
        //monitor;
       "xproxySpace.aif"
a.record(false);

a.source = { SinOsc.ar([400,700], 0, 0.1) };
a.source = { SinOsc.ar([410,510], 0, 0.1) };
a.source = { SinOsc.ar([LFNoise1.kr(80, 100, 300),500], 0, 0.1) };

//stop recording and close file
a.close;

//monitor off
a.stop;
```

recording from some bus

```
a = Bus.audio(s, 2);

SynthDef("test", { arg out; Out.ar(out, { WhiteNoise.ar(0.1) }.dup(2)) }).send(s);
x = Synth("test", [\out, a.index]);



n = RecNodeProxy.audio(s, 2);
n.source = { InFeedback.ar(a.index, 2) };

       //monitor
       //turn off monitor
```

```
        "noise.aif"
n.record;
n.unpause;

n.close;
```

instance creation from an existent node proxy

```
b = NodeProxy.audio(s, 2);
        //listen to b
b.source = { SinOsc.ar([400,500], 0, 0.1) }; //play something

r = RecNodeProxy.newFrom(b);
        "recproxy514.aif"   //open file
           //start recorder (paused)

            //start recording

b.source = { SinOsc.ar([430,500], 0, 0.1) };
b.source = { SinOsc.ar([410,510], 0, 0.1) };
b.source = { SinOsc.ar([LFNoise1.kr(80, 100, 300), 500], 0, 0.1) };
r.pause;
b.source = { WhiteNoise.ar(0.01) };
r.unpause;
r.pause;


//stop recording and close file
r.close;
        //stop listen to b
```

instance creation from an existent node proxy II

```
b = NodeProxy.audio(s, 2);
```

```
        //listen to b
b.source = { SinOsc.ar([400,500], 0, 0.1) }; //play something

    b.record "recproxy101.aiff"   //start recorder (paused)
           //start recording
          //end recording, close file
        //stop listen
```

recording from other sources

```
s = Server.local;
s.boot;

a = RecNodeProxy.audio(s, 2);
             //get the bus index;
a.play;   //monitor;
       "xproxySpace.aif"
a.record;
a.unpause;

(
Routine
var id;
loop({
id = s.nextNodeID;
s.sendMsg("/s_new", "default", id,0,0, \out, b, \freq, rrand(400, 800));
0.2.wait;
s.sendMsg("/n_set", id, \gate, 0);
0.2.wait;
})
}).play;
)


//stop recording and close file
```

```
a.close;

//monitor off
a.stop;
```

12.6 Patterns

ID: 253

EventPatternProxy event stream reference

superclass: TaskProxy

keeps a reference to a stream that can be replaced while playing.
Multiple streams are thus handled without creating dependancies.

related: [**Pdef**]

***basicNew(source)**
create a new instance with a pattern (the source).
the pattern should be an *event pattern* (see **Pdef**)
(*new is implemented for Pdef to avoid a dispatch)

***default**
a default source, if none is given. the default is a Pbind with resting notes of 1.0 beat duration

source_(obj)
set the source (a pattern). If a quantization is given, schedule this change to the next beat
(**pattern_**(..) is equivalent)

quant_(beats)
set the quantization value. can be an array [quant, offset, outset]

quant
get the quantization value

***defaultQuant_(beats)**
set the default quantization value for the class. (default: 1.0)

fadeTime_(beats)

when the synthdefs that are used contain an \amp control, the patterns are replaced by crossfading the previous with the new over this time (in beats)

envir_(event)
provide a default event for the Pdef. It is used to filter the incoming stream before it is passed to the source pattern. This is similar to NodeProxy-nodeMap.
When set for the first time, the pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the default event. If there is none, it is created and the pattern is rebuilt.

a) using as stream reference

_asStream
returns an instance of RefStream, which updates its stream as soon as the pattern is changed.

embedInStream(inval)
just like any pattern, embeds itself in stream

b) using as EventStreamPlayer

play(clock, protoEvent, quant)
starts the EventPatternProxy and creates a player.
if you want to play multiple instances, use **.playOnce(clock, protoEvent, quant)**
quant can be an array of [quant, phase]

stop
stops the player

player

the current player (if the Pdef is simply used in other streams this is nil)

pause / resume / reset / mute / unmute
perform player method

isPlaying
returns true if Pdef is running.
if a Pdef is playing and its stream ends, it will schedule a stream for playing as soon as a new one is assigned to it.

a) embedding EventPatternProxy in streams:

```
(
SynthDef("Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
env = EnvGen.kr(Env.perc(0.03, sustain), 1, doneAction:2);
5.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
)
s.boot;

#a, b, c, m = {EventPatternProxy.basicNew} !  4;

m.play;
           Pbind \instrument  \Pdefhelp  \dur     \degree      \legato

a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 5, 4, 3]));
b.source = Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[7, 8, 7, 8]));
c.source = Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 1, 2], 2));

x = Pseq([a, b, c], inf).play;
```

```
c.source = Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[4, 3, 1, 2]*3));


// infinite loops are scheduled (to ths clock's next beat by default) and released:

a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.753, \degree, Pseq(#[0, 5, 4, 3, 2], inf));
a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[0, 5, 4, 3] + 1, 1));
a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 5, 4, 3] - 1, 1));

a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[0, 5] - 1, 1));
a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.753, \degree, Pshuf(#[0, 5, 4, 3, 2], inf));

x.stop;
m.stop;

// EventPatternProxy can be used in multiple patterns

(
x = Ppar
 Pbindf Pn a  inf
\gtranspose, Pstutter(8, Pseq(#[0, 2, 0, 3],inf))
),
 Pbindf Pn a  inf
\gtranspose, Pstutter(8, Pseq(#[7, 4, 0, 3],inf)),
\dur, 0.6
),
 Pbindf Pn a  inf
\degree, Pseq(#[0, 5, 4, 3, 2, 3, 2], 1)
)
]).play;
)

a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.1, \degree, Pseq(#[0, 1, 0, 1, 2], inf));

a.source = Pbind \instrument  \Pdefhelp  \dur       \degree  Pseq         inf

a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.2, \degree, Pseq([0, 4, Prand([6, 8b],2)], inf));
```

```
a.source = Pbind(\instrument, \Pdefhelp, \dur, 0.1, \degree, Pseq(#[0, 1b, 1, 2b, 2, 3, 4b, 4, 5], inf));


	\detune		// set environment
a.set(\detune, 0);

x.stop;
```

b) playing EventPatternProxy

```
(
// load a synthdef
s.boot;
SynthDef "gpdef"
{ arg out=0, freq=440, dur=0.05, amp=0.1;
var env;
env = EnvGen.kr(Env.perc(0.01, dur), doneAction:2) * amp;
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)


#x, y = {EventPatternProxy.basicNew} !  2;

	// play them.  A silent resting pattern is used.
y.play;


// assign various patterns to it:

x.source = Pbind \dur		\instrument  \gpdef
x.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6], inf), \instrument, \gpdef);
x.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]+1, inf), \instrument, \gpdef);
y.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]-1, inf), \instrument, \gpdef);
y.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b]-2, inf), \instrument, \gpdef);
```

```
// using fadeTime:

y.fadeTime = 8.0;
y.source = Pbind(\dur, 0.125, \degree, Pseq([3, 4, 5b, 6]+4.rand, inf), \instrument, \gpdef);
y.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]-2, inf), \instrument, \gpdef);

(
x.source = Pbind
\dur, 1 / 6,
\degree, Pseq([3, 4, Prand([8, 2, 3, 9, 10],1) - 5, 6]+1, inf),
  \instrument  \gpdef
)
);
)
(
x.source = Pbind
\dur, 0.25,
\degree, Pseq([3, 4, Prand([8, 2, 3, 9, 10],1), 6], inf),
  \instrument  \gpdef
);
)
x.stop;



// tempo change
TempoClock.default.tempo = 1.3;
y.source = Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5, 6]+1, inf), \instrument, \gpdef);

// drop in ending patterns

x.play;
x.fadeTime = nil;

x.source = Pbind(\dur, 0.25, \degree, Pseq([3, [7,4], 5, 6]-2), \instrument, \gpdef);
x.source = Pbind(\dur, 0.125, \degree, Pseq([3, [7,4], 5, 4]-3), \instrument, \gpdef);
x.source = Pbind(\dur, 0.35, \degree, Pseq([3, [7,4], 5, 4, 3]-3), \instrument, \gpdef);
x.source = Pbind(\dur, 0.25, \degree, Pshuf([3, [7,4], 5, 6]-2), \instrument, \gpdef);
```

```
TempoClock.default.tempo = 1.0;
x.stop;
y.stop;
```

ID: 254

PatternProxy stream reference

superclass: Pattern

keeps a reference to a stream that can be replaced while playing.
Multiple streams are thus handled without creating dependancies.

related: [**Pdefn**]

***basicNew(source)**
create a new instance with a pattern (the source).
the pattern should be a *value pattern* (see **Pdefn**)
(*new is implemented for Pdefn to avoid a dispatch)
for event pattern proxy, see: *EventPatternProxy*
instead of a pattern, a **function** can be passed in, creating a routine.

***default**
a default source, if none is given. the default is 1.0 (it is not 0.0 in order to make it safe for durations)

source_(obj)
set the source. If a quantization is given, schedule this change to the next beat

quant_
set the quantization value

quant
get the quantization value

***defaultQuant_**
set the default quantization value for the class. (default: nil)

condition_(func)
provide a condition under which the pattern is switched when a new one is inserted.
the stream value and a count is passed into the function

the methods **count_(n)** simply counts up to n and switches the pattern then

reset
switch the pattern immediately. (stuck conditions can be subverted by this)

embedInStream(inval)
just like any pattern, embeds itself in stream

PatternProxy implements some methods for the benefits of its subclasses Pdefn/Pdef/Tdef which
are not useful for PatternProxy, EventStreamProxy and TaskProxy.

envir_(event)
provide a default environment for the proxy.
If given, it is used as an environment for the routine
function. When set for the first time, the routine pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the environment.
If there is none, it is created and the pattern is rebuilt.

endless
returns a Proutine that plays the proxy endlessly, replacing **nil** with a **default** value (1). This allows to create streams that idle on until a new pattern is inserted.

```
// example

a = PatternProxy.basicNew(Pseq           inf

x = Pseq([0, 0, a], inf).asStream;

t = Task({ loop({ x.next.postln; 0.3.wait }) }).play;



a.source = Pseq              inf
a.source = Pseq([55, 66, 77],1);

t.stop;
```

```
// PatternProxy, like Pdefn can be accessed in multiple streams

(
SynthDef("Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
env = EnvGen.kr(Env.perc(0.03, sustain), 1, doneAction:2);
5.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
s.boot;
)

(
x = PatternProxy.basicNew;
x.source = Pseq([0, 3, 2],inf);

Pset \instrument  \Pdefhelp
 Ppar
Pbind(\degree, x),
Pbind(\degree, x, \dur, 1/3)
])
).play;
)

x.source = Prand([0, 3, [1s, 4]],inf);

x.source = Pn(Pshuf([0, 3, 2, 7, 6],2),inf);



// if quant is set, the update is done at the next beat or whatever is specified:

x.quant = 4;
x.source = Pn Pseries          inf

x         nil  // reactivate immediacy

(
```

```
x.source = Prout {
loop {
4.do { | i|
#[2, 3, 4].choose.yield;
#[5, 0, 11].choose.yield;
#[6, 3, 4].choose.do { | j|  (i % j).yield };
}
}
})
)
```

ID: 255

Pbindef incremental event pattern reference definition

superclass: Pdef

keeps a reference to a Pbind in which single keys can be replaced, just like in **PbindProxy**.
It plays on when the old stream ended and a new stream is set and schedules the changes to the beat.

the difference to Pdef is that it allows to incrementally change the the elementary patterns (patternpairs)
of a Pbind - also of an already existing Pbind in a Pdef.

*new(key, paramKey1, pattern1, ...)

store the pattern in the global dictionary of **Pdef** under key.
if there is already a Pdef there, replace its pattern with the new one.
If there is already a **Pbindef** there, set the parameters only, or add a new one (the whole pattern is replaced).

*new(key)

acess the pattern at that key (if none is there, a default pattern is created)
see **Pdef**

```
// example:

(
SynthDef(#5f5f5f"Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
```

```
env = EnvGen.kr(Env.perc(0.01, sustain), 1, doneAction:2);
5.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
)
s.boot;


#007200Pbindef \a  \instrument  \Pdefhelp
Pbindef(#007200\a, #007200\degree, Pseq([0, 2, 5b, 1b], inf));
Pbindef(#007200\a, #007200\dur, 0.1);
Pbindef(#007200\a, #007200\degree, Pseq([1b, 5, 3, 1b, 6, 2, 5, 0, 3, 0, 2], inf));
Pbindef(#007200\a, #007200\legato, Prand([1.0, 2.4, 0.2], inf), #007200\mtranspose, -3);
Pbindef #007200\a  #007200\mtranspose  nil   // remove key

Pbindef(#007200\a, #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 1));
Pbindef(#007200\a, #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 3), #007200\dur, 0.02);
Pbindef(#007200\a, #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 3), #007200\dur, 0.1);

// apart from this Pbindef behaves like Pdef:

Pbindef(#007200\a).quant = 0.0;
Pbindef(#007200\a, #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 1));

Pbindef(#007200\a).stop;
Pbindef(#007200\a, #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 1)); // does not resume now

Pbindef #007200\a            // play single instance
Pseq  Pbindef #007200\a   Pdef #007200\a          // same here (Pdef(\a) is the same pattern as Pbindef))


Pbindef #007200\a      Pdef #007200\a  // identical.



// an already existing Pdef can be incrementally changed

#007200Pdef \x  Pbind \instrument  \Pdefhelp  \dur
Pdef(#007200\x).play;
```

```
Pbindef(#007200\x, #007200\degree, 7.rand);
Pbindef(#007200\x, #007200\degree, Pseq([0, 7, 3, 7, 4], inf), #007200\dur, Pn(Pseries(0.2, -0.02, 10)));

#007200Pbindef \x  \stretch
```

ID: 256

PbindProxy incremental event pattern reference

superclass: Pattern

keeps a reference to a Pbind in which single keys can be replaced.
It plays on when the old stream ended and a new stream is set and schedules the changes to the beat.

related [**Pbindef**][**Pdef**]

*new(key1, pattern1, key2, pattern2, ...)
create a new instance of PbindProxy with the given patternpairs

source
returns the wrapper for the Pbind

set(key1, pattern1, key2, pattern2, ...)
set the given patternpairs.

at(key)
return a pattern at that key. this can be used to set quant value individually,
so different elementary patterns can be quantized differently.

```
x.at(#007200\freq).quant = 2;
```

quant_(val)
set the quant of all elementary patterns

quant

return the quant value of the source pattern

```
// example:

(
SynthDef(#5f5f5f"Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
env = EnvGen.kr(Env.perc(0.03, sustain), 1, doneAction:2);
5.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
)
s.boot;


    PbindProxy
#007200      \instrument  \Pdefhelp

x.play;

x.set(#007200\degree, Pseq([0, 2, 5b, 1b], inf));
x.set(#007200\dur, 0.1);
x.set(#007200\degree, Pseq([1b, 5, 3, 1b, 6, 2, 5, 0, 3, 0, 2], inf));
x.set(#007200\legato, Prand([1.0, 2.4, 0.2], inf), #007200\mtranspose, -3);
      #007200\mtranspose  nil   // remove key

x.set(#007200\degree, Pseq([1, 2, 3, 4, 5, 6], 1));
x.play;

x.set( #007200\degree, Pseq([1, 2, 3, 4, 5, 6], 3), #007200\dur, 0.02);
x.play;

x.set(#007200\degree, Pseq([1, 2, 3, 4, 5, 6], 3), #007200\dur, 0.1);
x.play;


// embed in other patterns:
(
```

```
x.set(#007200\degree, Pseq([1b, 5, 3, 1b, 6, 2, 5, 0, 3, 0, 2], inf));
Ppar
x,
#007200Pbindf    \ctranspose
]).play;
)


x.set(#007200\degree, Pseq([1b, 5, 1b, 4, 0], inf), #007200\dur, 0.4);
```

ID: 257

Pdef stream reference definition

superclass: EventPatternProxy

keeps a reference to a stream that can be replaced while playing, just like its superclass. It plays on when the old stream ended and a new stream is set and schedules the changes to the beat.
It is very similar to [**EventPatternProxy],** but handles the storing of global instances:

Pdef(key) returns the instance, **Pdef(key, pat)** stores the pattern and returns the instance, like Tdef and Ndef.

It can be used to store event Patterns globally. Changes in this global library have effect immediately, or if quant/offset values are given, at the next time step. For value patterns (numerical, non event patterns)
[**Pdefn]** is used.

note that exchanging the source of a Pdef while playing does not work with Pmono and Pfx yet,
due to their incopatibility with Pfindur.

***new(key, pattern)**
store the pattern in a global dictionary under key.
if there is already a Pdef there, replace its pattern
with the new one.
if the pattern is a **function**, Pdef creates a PlazyEnvir internally
that dynamically creates the pattern returned from the function,
applying the arguments from the inevent.
***new(key)**
acess the pattern at that key (if none is there, a default pattern is created)

***default**
a default source, if none is given. the default is a Pbind with resting notes of 1.0 beat duration

***removeAll**
remove all streams
***all**
environment (IdentityDictionary) that stores all Pdefs.

***all_(envir)**
set the global environment

quant_(beats)
set the quantisation time for beat accurate scheduling.
can be an array [quant, offset, outset]

***defaultQuant_(beats)**
set the default quantisation for new instances (default: 1.0)
can be an array [quant, offset, outset]

condition_(func)
provide a condition under which the pattern is switched when a new one is inserted.
the stream value and a count is passed into the function (see example)
the methods **count_(n)** simply counts up to n and switches the pattern then

reset
switch the pattern immediately. (stuck conditions can be subverted by this)

fadeTime_(beats)
when the synthdefs that are used contain an \amp control, the patterns are replaced by crossfading the previous with the new over this time (in beats)

envir_(event)
provide a default event for the Pdef. It is used to filter the incoming stream before it is passed to the source pattern. This is similar to NodeProxy-nodeMap.
When set for the first time, the pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the default event. If there is none, it is created and the pattern is rebuilt.

map(key, pdefKey, key, pdefKey ...)
map Pdefn to the keys in the event.

source
set the pattern (internally done by *new(key, pattern)
(**pattern_**(..) is equivalent)

endless
returns a Proutine that plays the proxy endlessly, replacing **nil** with a **default** value (silent event). This allows to create streams that idle on until a new pattern is inserted.

a) using it as stream reference

embedInStream(inval)
just like any pattern, embeds itself in stream

b) using it as EventStreamPlayer

play(clock, protoEvent, quant)
starts the Pdef and creates a player.
if you want to play multiple instances, use **.playOnce(clock, protoEvent, quant)**
quant can be an array of [quant, phase]

stop
stops the player

player
the current player (if the Pdef is simply used in other streams this is nil)

pause / resume / reset / mute / unmute

perform player method

isPlaying
returns true if Pdef is running.
if a Pdef is playing and its stream ends, it will schedule a stream for playing as soon as a new one is assigned to it.

for another use of Pdef see also [**recursive_phrasing**]

a) embedding Pdef into a stream:

```
(
SynthDef("Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
env = EnvGen.kr(Env.perc(0.03, sustain), 1, doneAction:2);
3.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
)
s.boot;

Pdef \metronom  Pbind \instrument  \Pdefhelp  \dur     \degree      \legato

x = Pseq([Pdef(\a), Pdef(\b), Pdef(\c)], inf).play;

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 5, 4, 3])));
Pdef(\b, Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[7, 8, 7, 8])));
Pdef(\c, Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 1, 2], 2)));
```

```
Pdef(\c, Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[4, 3, 1, 2]*3)));


// infinite loops are scheduled (to ths clock's next beat by default) and released:

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.753, \degree, Pseq(#[0, 5, 4, 3, 2], inf)));
Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[0, 5, 4, 3] + 1, 1)));
Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.25, \degree, Pseq(#[0, 5, 4, 3] - 4, 1)));

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.125, \degree, Pseq(#[0, 5] - 1, 1)));
Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.753, \degree, Pshuf(#[0, 5, 4, 3, 2], inf)));

x.stop;
Pdef \metronom

// Pdef can be used in multiple patterns:

(
x = Ppar
 Pbindf Pn Pdef \a   inf
\gtranspose, Pstutter(8, Pseq(#[0, 2, 0, 3],inf))
),
 Pbindf Pn Pdef \a   inf
\gtranspose, Pstutter(8, Pseq(#[7, 4, 0, 3],inf)),
\dur, 0.6
),
 Pbindf Pn Pdef \a   inf
\degree, Pseq(#[0, 5, 4, 3, 2, 3, 2], 1)
)
]).play;
)

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.1, \degree, Pseq(#[0, 1, 0, 1, 2], inf)));

Pdef \a  Pbind \instrument  \Pdefhelp  \dur       \degree  Pseq         inf

Pdef \a  Pbind \instrument  \Pdefhelp  \dur     \degree  Pseq         inf

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.2, \degree, Pseq([0, 4, Prand([6, 8b],2)], inf)));
```

```
Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.1, \degree, Pseq(#[0, 1b, 1, 2b, 2, 3, 4b, 4, 5], inf)));


// using a fade time, the above changes are crossfaded
Pdef(\a).fadeTime = 2;

Pdef(\a, Pbind(\instrument, \Pdefhelp, \dur, 0.2, \degree, Pseq([0, 4, Prand([6, 8b],2)], inf)));

// ...

Pdef \a      \detune        // set environment
Pdef(\a).set(\detune, 0);

x.stop;
```

b) playing Pdef

```
(
// load a synthdef
s.boot;
SynthDef "gpdef"
{ arg out=0, freq=440, dur=0.05, amp=0.1;
var env;
env = EnvGen.kr(Env.perc(0.01, dur), doneAction:2) * amp;
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)


Pdef \x   // creates a Pdef with a default pattern.


Pdef \x        // play it.  A silent resting pattern is used.
Pdef \y        // play a second one (automatically instantiated)


// assign various patterns to it:
```

```
Pdef \x  Pbind \dur        \instrument  \gpdef
Pdef(\x, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6], inf), \instrument, \gpdef));
Pdef(\x, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]+1, inf), \instrument, \gpdef));
Pdef(\y, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]-1, inf), \instrument, \gpdef));
Pdef(\y, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b]-2, inf), \instrument, \gpdef));

// using fadeTime:

Pdef(\y).fadeTime = 8.0;
Pdef(\y, Pbind(\dur, 0.125, \degree, Pseq([3, 4, 5b, 6]+4.rand, inf), \instrument, \gpdef));
Pdef(\y, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5b, 6]-2, inf), \instrument, \gpdef));

(
Pdef \x  Pbind
\dur, 1 / 6,
\degree, Pseq([3, 4, Prand([8, 2, 3, 9, 10],1) - 5, 6]+1, inf),
  \instrument  \gpdef
)
);
)
(
Pdef \x  Pbind
\dur, 0.25,
\degree, Pseq([3, 4, Prand([8, 2, 3, 9, 10],1), 6], inf),
  \instrument  \gpdef
);
)
Pdef(\x).stop;

Pdef(\x).play;

// tempo change
TempoClock.default.tempo = 1.3;
Pdef(\y, Pbind(\dur, 0.25, \degree, Pseq([3, 4, 5, 6]+1, inf), \instrument, \gpdef));

// drop in ending patterns

Pdef(\x, Pbind(\dur, 0.25, \degree, Pseq([3, [7,4], 5, 6]-2), \instrument, \gpdef));
Pdef(\x, Pbind(\dur, 0.125, \degree, Pseq([3, [7,4], 5, 4]-3), \instrument, \gpdef));
```

```
Pdef(\x, Pbind(\dur, 0.35, \degree, Pseq([3, [7,4], 5, 4, 3]-3), \instrument, \gpdef));
Pdef(\x, Pbind(\dur, 0.25, \degree, Pshuf([3, [7,4], 5, 6]-2), \instrument, \gpdef));


// clear all.
Pdef(\x).clear;
Pdef(\y).clear;
TempoClock.default.tempo = 1.0;
```

recursion:

Pdefs can be used recursively under the condition that the stream call structure allows it.
a structure like the following works:

```
Pdef \x  Pseq  Pbind \instrument  \gpdef)  Pdef \x    inf
Pdef(\x).play;
```

but the following would crash, because .embedInStream is called recursively with no limit:

```
// Pdef \y  Pseq  Pdef \y   Pbind \instrument  \gpdef)   inf
```

outset

When quantizing to a larger number of beats, the changes become very slow if one has to wait for the next beat. Providing an outset quant value is a way to make the change so that it appears as if it had been done at the previous grid point already. The stream is fast forwarded to the current position relative to the quant grid.
Providing a number larger than zero, the next possible quant point is used as outset.

For example, if quant is 32, and one has just missed the first beat when changing the pattern,

one has to wait for 32 beats until the change happens. Using an outset of 1, it is assumed that you had already
changed the pattern at the first beat, the stream is fast forwarded to the time it would be at now if you had done so. The new pattern is inserted at the next beat (outset=1).

quant can be: [quant, offset, outset]

```
// examples
(
Pdef(\x).quant_([8, 0, 1]);
Pdef(\y).quant_([8, 0.5, 1]); // offset by half a beat
Pdef(\x).play;
Pdef(\y).play;
)

Pdef(\x, Pbind(\degree, Pseq((0..7)+2, inf)));
Pdef(\y, Pbind(\degree, Pseq((0..7)-2, inf)));
Pdef(\x, Pbind(\degree, Pseq((0..7)+2, inf), \dur, 0.5));
Pdef(\y, Pbind(\degree, Pseq((0..7).scramble-2, inf), \dur, 0.25, \legato, 0.3));
Pdef(\x, Pbind(\degree, Pseq((0..7), inf)));

Pdef(\x, Pbind(\degree, Pseq([ 1, 5, 6, 7, 0, 3, 2, 4 ], inf), \dur, 1));
Pdef(\x, Pbind(\degree, Pseq([ 0, 2, 2, 4, 0, 4, 0, 4 ], inf), \dur, 1));

Pdef \x                      // offset by 1/6 beat relative to y
Pdef(\x, Pbind(\degree, Pseq([ 1, 1, 1, 7, 0, 2, 2, 4 ], inf), \legato, 0.1));
Pdef(\x, Pbind(\degree, Pseq([ 3, 3, 3, 4b ], inf), \legato, 0.1));
Pdef(\y, Pbind(\degree, Pseq((0..7).scramble-4, inf), \dur, 0.25, \legato, 0.3));
```

note

this fast forwarding might create a cpu peak if the pattern is very complex/fast or quant is very long. This is hard to avoid, so it simply has to be taken into account.

```
// some testing
(
var                    // quantise to 8 beats, no offset, insert quant to 1 beat
Pdef(\x).quant_(quant);
Pdef(\x).play;
Routine { loop { 8.do { | i|  ("uhr:"+i).postln; 1.wait } } }.play(quant:quant);
Pbind(\degree, Pseq((0..7), inf)).play(quant:quant);
)

Pdef(\x, Pbind(\degree, Pseq((0..7)+2, inf)).trace(\degree));
Pdef(\x, Pbind(\degree, Pseq((0..7), inf) + [0, 3]).trace(\degree));
Pdef(\x, Pbind(\degree, Pseq((0..7), inf) + [0, 6], \dur, 0.5).trace(\degree));



Pdef(\x).fadeTime = 8;

Pdef(\x, Pbind(\degree, Pseq((0..7), inf)).trace(\degree));
Pdef(\x, Pbind(\degree, Pseq((0..7).reverse, inf) + [0, 6], \dur, 0.5));

Pdef(\x).fadeTime = nil;
Pdef(\x).quant = 1;

Pdef(\x, Pbind(\degree, Pseq((0..7), inf)).trace(\degree));

Pdef(\x).quant = 8;
Pdef(\x, Pbind(\degree, Pseq((0..7), inf)).trace(\degree));
```

update condition

In order to be able to switch to a new pattern under a certain condition, the instance variable
condition can be set to a function that returns a boolean. Value and a count index are passed to the function.
The condition is always valid for the **next pattern** inserted. For stuck conditions, the **reset** message can be used.

As counting up (such as *"every nth event, a swap can happen"*) is a common task, there

is a method for this,
called **count(n).**

```
Pdef(\x).play;
Pdef \x             // we don't want quant here.
Pdef(\x, Pbind(\degree, Pseq((0..5), inf), \dur, 0.3)).condition_({ | val, i|  i.postln % 6 == 0 });
Pdef(\x, Pbind(\degree, Pseq((0..7) + 5.rand, inf), \dur, 0.3)).condition_({ | val, i|  (i % 8).postln
== 0 });

// the above is equvalent to:
Pdef(\x, Pbind(\degree, Pseq((0..7) + 5.rand, inf), \dur, 0.3)).count(8);

// the value that is sent in is the event, so decisions can be made dependent on the event's fields
```

reset

```
// reset to change immediately:
Pdef(\x).reset;
```

ID: 258

Pdefn value-stream reference definition

superclass: PatternProxy

access and assignment are done by *new

keeps a reference to a task that can be replaced while playing.

Pdefn(key) returns the instance, **Pdefn(key, pat)** defines the pattern
and returns the instance, like Pdef, Tdef and Ndef.
it is very similar to [**PatternProxy**]

it can be used to store value patterns globally (for event patterns, see **Pdef**).

***new(key, pattern)**
store the pattern in a global dictionary under key.
the **pattern** can be anything that embeds in a stream.
instead of a pattern, a **function** can be passed in, creating a routine. (see example below).

***new(key)**
acess the pattern at that key (if none is there, a default pattern is created)

***default**
a default source, if none is given.
the default is 1.0 (it is not 0.0 in order to make it safe for durations)
***removeAll**
remove all patterns
***all**
dict that stores all Pdefn

***all_(envir)**
set the global environment

quant_(beats)
set the quantisation time for beat accurate scheduling
can be a pair [quant, offset]

***defaultQuant_(beats)**
set the default quantisation for new instances (default: nil)
can be a pair [quant, offset]

condition_(func)
provide a condition under which the pattern is switched when a new one is inserted.
the stream value and a count is passed into the function (see example)
the methods **count_(n)** simply counts up to n and switches the pattern then

source
set the pattern (internally done by *new(key, pattern).
if quant is not nil, the change is scheduled to the beat
(**pattern_**(..) is equivalent)

embedInStream(inval)
just like any stream, embeds itself in stream.

reset
switch the pattern immediately. (stuck conditions can be subverted by this)

envir_(event)
provide a default environment for the proxy.
If given, it is used as an environment for the routine
function. When set for the first time, the routine pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the environment.
If there is none, it is created and the pattern is rebuilt.

map(key, pdefKey, key, pdefKey ...)
map one Pdefn to the other. the patterns can be accessed via the currentEnvironment

endless
returns a Proutine that plays the proxy endlessly, replacing **nil** with a **default**

value (1). This allows to create streams that idle on until a new pattern is inserted.

Pdefn is similar to [Pdef] and [Tdef] . see the other helpfiles for comparison.

Pdefn in expressions

```
Pdefn \c  Pdefn \a    Pdefn \b

    Pdefn \c            // create a stream from Pdefn(\c)

         // default value for a Pdefn is 1, so that it is a good time value default.

Pdefn \a        // (re)define Pdefn(\a) as 100

t.value;

Pdefn \b  Pseq            inf    // (re)define Pdefn(\b) as Pseq([1, 2, 3], inf)

3.do { t.value.postln };

Pdefn \c  Pdefn \a    Pdefn \b    Pdefn \a    // (re)define Pdefn(\c)

8.do { t.value.postln };

Pdefn \a  Prand            inf     // (re)define Pdefn(\a)
```

Embedding Pdefn in other patterns

```
Pdefn(\x, Pseq([1, 2, 3],inf));

x = Pseq([0, 0, Pdefn(\x)], inf).asStream;

t = Task({ loop({ x.next.postln; 0.3.wait }) }).play;


Pdefn(\x, Pseq([55, 66, 77],inf));
Pdefn(\x, Pseq([55, 66, 77],1));

t.stop;



// Pdefn can be accessed in multiple streams

(
SynthDef("Pdefhelp", { arg out, freq, sustain=1, amp=1, pan;
var env, u=1;
env = EnvGen.kr(Env.perc(0.03, sustain), 1, doneAction:2);
5.do { var d; d = exprand(0.01, 1); u = SinOsc.ar(d * 300, u, rrand(0.1,1.2) * d, 1) };
Out.ar(out, Pan2.ar(SinOsc.ar(u + 1 * freq, 0, amp * env), pan));

}).store;
s.boot;
)

(
Pdefn(\deg, Pseq([0, 3, 2],inf));

Pset \instrument  \Pdefhelp
 Ppar
  Pbind \degree  Pdefn \deg
Pbind(\degree, Pdefn(\deg), \dur, 1/3)
])
).play;
)

Pdefn(\deg, Prand([0, 3, [1s, 4]],inf));
```

```
Pdefn(\deg, Pn(Pshuf([0, 3, 2, 7, 6],2),inf));

(
Pdefn(\deg, Plazy { var pat;
pat = [Pshuf([0, 3, 2, 7, 6],2), Pseries(0, 1, 11), Pseries(11, -1, 11)].choose;
Pn(pat, inf)
});
)
```

Timing: when does the definition change?

```
// if quant is set, the update is done at the next beat or whatever is specified:

Pdefn(\deg).quant = 4;
Pdefn(\deg, Pn(Pseries(0, 1, 8),inf));

Pdefn \deg          nil  // activate immediately again

(
Pdefn \deg
loop {
5.do { | i|
#[1, 3, 4].choose.yield;
#[5, 0, 12].choose.yield;
#[14, 3, 4].choose.do { | j|  (i % j).postln.yield };
}
}
})
)
```

update condition

In order to be able to switch to a new pattern under a certain condition, the instance

variable
condition can be set to a function that returns a boolean. Value and a count index are passed to the function.
The condition is always valid for the **next pattern** inserted. For stuck conditions, the **reset** message can be used.

As counting up (such as *"every nth event, a swap can happen"*) is a common task, there is a method for this,
called **count(n).**

```
z = Pbind(\degree, Pdefn(\x), \dur, 0.25).play;
Pdefn(\x, Pseq((0..5), inf)).condition_({ | val, i|  i.postln % 6 == 0 });
Pdefn(\x, Pseq((0..8), inf)).condition_({ | val, i|  i.postln % 9 == 0 });


// the above is equvalent to:
Pdefn(\x, Pseq((0..8), inf)).count(9);
```

reset

```
// reset to change immediately:
Pdefn(\x).reset;
```

Functions as arguments to Pdefn:
(experimental, bound to change!)

```
Pdefn(\deg, { loop { yield(0.1.rand.round(0.01) + [2, 3, 9].choose) } });

// equivalent to:

Pdefn(\deg, Proutine { loop { yield(0.1.rand.round(0.01) + [2, 3, 9].choose) } });

// this is not exactly true, see below..
```

The (inner) environment

when passing a function to

```
// set() creates a local environment that overrides the outer currentEnvironment

Pdefn(\z).set(\a, 1, \b, 5);
(
Pdefn \z    | e|
loop { yield((e.a + e.b) + 0.1.rand.round(0.01)) }
})
   // [1]

t = Pdefn(\z).asStream;

t.nextN(3);

(
Pdefn \z    | e|
 //(e.a + e.b) + 0.1.rand.round(0.01) 1
Pseq([1, 2, e.b], 1)
})
);

Pdefn(\z, Pseq([1, 2, 3], 1));

e = Pdefn \z
d

Pdefn(\z).set(\a, 3);

t.next;

Pdefn(\z).set(\a, Pseq([1, 2, 3], inf));

t.nextN(3);
```

```
Pdefn \z          // post the envir



// using the "map" message one can map one Pdefn to the other:

Pdefn(\z).map(\a, \other);

            // Pdefn default value (1) is used

Pdefn \other  Prand                  inf     // assign a pattern to \other

t.nextN(3);


// if you want to keep using the currentEnvironment at the same time,
// assign the currentEnvironment to the envir's parent (or proto) field
// (this shouldn't be a proxy space of course.)

Pdefn(\z).envir.parent = currentEnvironment;
a = 9;
b = 10;

t.nextN(3);
```

ID: 259

Pdict pattern that embeds patterns from a dictionary

superclass: Penvir

*new(dict, keyPattern, repeats, default)

```
// example

SynthDescLib

(
e = (
a:  Pbind(\dur, 0.1, \degree, Pseq([0, 5, 4, 3, 2])),
b:  Pbind(\dur, 0.06, \degree, Pseq([7, 8, 7, 8])),
c:  Pbind(\dur, 0.3, \degree, Pseq([0, 1, 2], 2))
);

x = Pdict(e, Pseq([
\a, \b,
Prand([\a, \c])
], 4)
);
x.play;
)
```

ID: 260

StreamClutch

superclass: Stream

buffers a streamed value

StreamClutch.new(pattern, connected)

pattern a pattern or stream to be buffered
connected
if true it wil call the next stream value for each time next is called
if false it returns the last value

```
//example:

SynthDescLib

a = Pseq([1, 2, 3], inf);
    StreamClutch

6.do({ b.next.postln });
b.connected = false;
6.do({ b.next.postln });


//statistical clutch
a = Pseq([1, 2, 3], inf);
b = StreamClutch(a, { 0.5.coin });

12.do({ b.next.postln });


//sound example:
(
var clutch, pat, decicion;
```

```
decicion = Pseq([Pn(true,10), Prand([true, false], 10)], inf).asStream;
pat = Pbind(\freq, Pseq([200, [300, 302], 400, 450], inf), \dur, 0.3);
clutch = StreamClutch(pat, decicion);
clutch.asEventStreamPlayer.play;
)

// independant stepping
(
var clutch, pat, decicion;
pat = Pbind(\freq, Pseq([200, [300, 302], 400, 450], inf), \dur, 0.3);
    StreamClutch
b.connected = false;
b.asEventStreamPlayer.play;
)

b.step;
```

ID: 261

TaskProxy event stream reference

superclass: PatternProxy

Keeps a reference to a task (time pattern) that can be replaced while playing.
It plays on when the old stream ended and a new stream is set and schedules the changes to the beat.

related: [**Tdef**]

***basicNew(source)**
create a new instance with a function (the source).
the source should be a **routine function** (see **Tdef**) or a **pattern** of time values.
(*new is implemented for Tdef to avoid a dispatch)

***default**
a default source, if none is given. the default is a loop that does nothing with a 1.0 beat wait time

source_(obj)
set the source. If a quantization is given, schedule this change to the next beat
the object is a **routine function,** which is evaluated in a protected way, so that failure will notify the proxy that it has stopped.
The object can also be a **pattern** of time values.

quant_(beats)
set the quantization value. can be a pair [quant, offset]

quant
get the quantization value

***defaultQuant_(beats)**
set the default quantization value for the class. (default: 1.0)

can be a pair [quant, offset]

condition_(func)
provide a condition under which the pattern is switched when a new one is inserted.
the stream value and a count is passed into the function.
the methods **count_(n)** simply counts up to n and switches the pattern then

reset
switch the pattern immediately. (stuck conditions can be subverted by this)

envir_(event)
provide a default environment for the proxy.
If given, it is used as an environment for the routine
function. When set for the first time, the routine pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the environment.
If there is none, it is created and the routine pattern is rebuilt.

endless
returns a Proutine that plays the proxy endlessly, replacing **nil** with a **default**
value (1 s. wait time). This allows to create streams that idle on until a new pattern is inserted.

a) using it as stream reference

source
set the routine function / pattern (internally done by *new(key, obj)

embedInStream(inval)
just like any stream, embeds itself in stream

b) using it as EventStreamPlayer

play(clock, protoEvent, quant)
starts the TaskProxy and creates a player.
if you want to play multiple instances, use **.playOnce(clock, protoEvent, quant)**
quant can be an array of [quant, phase]

stop
stops the player

player
the current player (if the TaskProxy is simply used in other streams this is nil)

pause / resume / reset / mute / unmute
perform player method

isPlaying
returns true if TaskProxy is running.
if a TaskProxy is playing and its stream ends, it will schedule a stream for playing as soon as a new one is assigned to it.

a) using it as a task player

```
// create an empty Tdef and play it.
x = TaskProxy.basicNew;
x.play;


x.source =  { loop { "ggggggggggggggggg9999ggg999ggg999gg".scramble.postln; 0.5.wait; } };


x.source = { loop { "---------////----------------------".scramble.postln; 0.25.wait; } };
x.source = { loop { thisThread.seconds.postln; 1.wait; } };
x.source = { loop { thisThread.seconds.postln; 1.01.wait; } };

TempoClock.default.tempo = 2;
```

```
x.source = { "the end".postln };
x.source = { "one more".postln };
x.source = { loop { "some more".scramble.postln; 0.25.wait; } };

TempoClock.default.tempo = 1;

x.stop;
x.play;
x.stop;




// sound example

(
// load a synthdef
s.boot;
SynthDef "pdef_grainlet"
{ arg out=0, freq=440, dur=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, dur, 0.3), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
x.play;

(
x.source = {
loop {
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, rrand(600, 640));
0.1.wait;
}
}
)

(
x.source = {
var x;
```

```
x = Pseries(300, 20, 100).loop.asStream;
loop {
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, x.next);
0.05.wait;
}
}
)

(
x.source = {
var x;
x = Plazy { Pseries(300 + 300.rand, 10 + 30.rand, 10 + 30.rand) }.loop.asStream;
loop {
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, x.next);
0.05.wait;
}
}
)

// metronome
(
y = TaskProxy.basicNew {
loop { s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, 1500); 1.wait; }
};
y.play;
)

// play ending stream once
(
x.source = {
var x, dt;
dt = [0.1, 0.125, 0.05].choose;
x = Plazy { Pseries(1300 + 300.rand, 110 + 130.rand, 16) }.asStream;
x.do { arg item;
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, item);
dt.wait;
}
}
)
```

```
...  and so on ...

x.stop;
y.stop;
```

b) embedding TaskProxy into other Tasks / Routines

```
(
#a, c = { TaskProxy.basicNew } !  2;
a.source = { "one".postln; 1.wait; "two".postln };
c.source = { var z; z = Synth(\default); 0.5.wait; z.release };
r = Task {
 "counting..."
2.wait;
a.embedInStream;
1.wait;
c.embedInStream;
"done.".postln;
};
)

        // play a stream

c.source = { var z; z = Synth(\default, [\freq, 300]); 1.5.wait; z.release }; // change the def

r.reset;
r.play;

// of course TaskProxies can be used in other Tdefs:
(
b = TaskProxy.basicNew;
b.source = {
 "counting..."
2.wait;
a.embedInStream;
1.wait;
```

```
c.embedInStream;
"done.".postln;
};
)
b.playOnce;

// if one wants to branch off a stream in a separate thread, asStream is used.
(
Routine
c.asStream.play;
0.1.wait;
c.asStream.play;
0.1.wait;
a.asStream.play;

}.play;
)
```

ID: 262

Tdef task reference definition

superclass: TaskProxy

keeps a reference to a task (**time pattern**) that can be replaced while playing.
It plays on when the old stream ended and a new stream is set and schedules the changes to the beat.

Tdef(key) returns the instance, Tdef(key, pat) defines the pattern and returns the instance, like Pdef and Ndef.
it is very similar to [**TaskProxy**]

***new(key, obj)**
store the task in a global dictionary under key.
if there is already a Tdef there, replace its task
with the new one.
obj is a function or a pattern of time values.

***new(key)**
acess the task at that key (if none is there, a default task is created)

***default**
a default source, if none is given. the default is a loop that does nothing with a 1.0 beat wait time

***removeAll**
remove all tasks

***all**
dict that stores all Tdefs

***all_(envir)**
set the global environment

quant_(beats)
set the quantisation time for beat accurate scheduling
can be a pair [offset, quant]

***defaultQuant_(beats)**
set the default quantisation for new instances (default: 1.0)
can be a pair [offset, quant]

condition_(func)
provide a condition under which the pattern is switched when a new one is inserted.
the stream value and a count is passed into the function.
the methods **count_(n)** simply counts up to n and switches the pattern then

reset
switch the pattern immediately. (stuck conditions can be subverted by this)

envir_(event)
provide a default environment for the proxy.
If given, it is used as an environment for the routine
function. When set for the first time, the routine pattern is rebuilt.

set(key, val, key2, val2, ...)
set arguments in the environment.
If there is none, it is created and the routine pattern is rebuilt.

endless
returns a Proutine that plays the proxy endlessly, replacing **nil** with a **default** value (1 s. wait time). This allows to create streams that idle on until a new pattern is inserted.

a) using it as stream reference

embedInStream(inval)
just like any stream, embeds itself in stream.
see example for usage.

b) using it as Task

play(clock, doReset, quant)
starts the Pdef and creates a player.
if you want to play multiple instances, use **.playOnce(clock, doReset, quant)** **quant** can be an array of [quant, phase]

stop
stops the task

player
the current task (if the Tdef is simply used in other streams this is nil)

pause / resume / reset / mute / unmute
perform player method

isPlaying
returns true if Tdef is running.
if a Tdef is playing and its stream ends, it will schedule a task for playing as soon as a new function is assigned to it.

a) using Tdef as a task player

```
Tdef \x        // create an empty Tdef and play it.

Tdef(\x, { loop({ "gggggggggggggggggg9999ggg999ggg999gg".scramble.postln; 0.5.wait; }) });


Tdef(\x, { loop({ "---------////----------------------".scramble.postln; 0.25.wait; }) });
Tdef(\x, { loop({ thisThread.seconds.postln; 1.wait; }) });
Tdef(\x, { loop({ thisThread.seconds.postln; 1.01.wait; }) });
```

```
TempoClock.default.tempo = 2;

Tdef(\x, { "the end".postln });
Tdef(\x, { "one more".postln });
Tdef(\x, { loop({ "some more".scramble.postln; 0.25.wait; }) });

TempoClock.default.tempo = 1;

Tdef(\x).stop;
Tdef(\x).play;

Tdef(\x)




// sound example

(
// load a synthdef
s.boot;
SynthDef "pdef_grainlet"
{ arg out=0, freq=440, dur=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, dur, 0.3), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
Tdef(\x).play;

(
Tdef(\x, {
loop({
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, rrand(600, 640));
0.1.wait;
})
})
)

(
```

```
Tdef(\x, {
var x;
x = Pseries(300, 20, 100).loop.asStream;
loop({
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, x.next);
0.05.wait;
})
})
)

(
Tdef(\x, {
var x;
x = Plazy({ Pseries(300 + 300.rand, 10 + 30.rand, 10 + 30.rand) }).loop.asStream;
loop({
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, x.next);
0.05.wait;
})
})
)

// metronome
Tdef(\y, { loop({ s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, 1500); 1.wait; }) }).play;

// play ending stream once
(
Tdef(\x, {
var x, dt;
dt = [0.1, 0.125, 0.05].choose;
x = Plazy({ Pseries(1300 + 300.rand, 110 + 130.rand, 16) }).asStream;
x.do({ arg item;
s.sendMsg("/s_new", "pdef_grainlet", -1,0,0, \freq, item);
dt.wait;
})
})
)

...  and so on ...

Tdef(\x).stop;
```

```
Tdef.removeAll;
```

b) embedding Tdef into other Tasks / Routines

```
(
Tdef(\a, { "one".postln; 1.wait; "two".postln });
Tdef(\c, { var z; z = Synth(\default); 0.5.wait; z.release });
r = Task({
 "counting..."
2.wait;
Tdef(\a).embedInStream;
1.wait;
Tdef(\c).embedInStream;
"done.".postln;
});
)

        // play a stream

Tdef(\c, { var z; z = Synth(\default, [\freq, 300]); 1.5.wait; z.release }); // change the def

r.reset;
r.play;

// of course Tdefs can be used in other Tdefs:
(
Tdef(\b, {
 "counting..."
2.wait;
Tdef(\a).embedInStream;
1.wait;
Tdef(\c).embedInStream;
"done.".postln;
});
)
Tdef(\b).asStream.play;
```

```
// if one wants to branch off a stream in a separate thread, asStream is used.
// also the method playOnce can be used
(
Routine
Tdef(\c).asStream.play;
0.1.wait;
Tdef(\c).asStream.play;
0.1.wait;
Tdef(\a).asStream.play;

}).play;
)
```

Tdef as a time pattern

Instead of using a Pdefn for time values, it can be useful to use a Tdef.
When changing its source, it keeps the stream of values synchronized to its clock.

```
(
// load a synthdef
s.boot;
SynthDef "pdef_grainlet"
{ arg out=0, freq=440, dur=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, dur, 0.3), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
```

```
Tdef(\z, Pseq([1, 1, 1, 0.5, 0.5], inf));
```

```
(
 Pset \instrument  \pdef_grainlet
Ppar([
   Pbind
\dur, Tdef(\z),
\note, Pseq([1, 3, 2, 1, 0], inf),
\x, Pfunc { TempoClock.default.elapsedBeats.postln } // posts the onset times
),
   Pbind
    \dur     // reference beat
\sustain, 0.1,
\note, 8
)
])
).play(quant:1);
)


Tdef(\z, Prand([1, 1, 0.23, 0.5, 0.5], inf)); // exchange time pattern
Tdef \z  Pseq               inf    // pattern stays in sync.
Tdef \z  Pseq                      inf    // but might be in different order (
         // to avoid this, set the quant to a appropriate value.
```

12.7 Tutorials

ID: 263

```
// basic live coding techniques ("object style")
// without the use of JITLib

// more to come..


// using a simple environment.  this looks just like ProxySpace, but works differently.
// for the difference, see [jitlib_basic_concepts_01] and  [jitlib_basic_concepts_02]



        // create a new environment
        // push it to current

// this synthdef can be changed on the fly, but the synth will
// not change from this.  use expression [1] for replacing a given synth
(
SynthDef \x    | freq=440|
Out.ar(0,
Ringz.ar(Dust.ar(40), freq, 0.1)
)
}).send(s);
)

// send a first synth:
s1 = Synth(\x);

// [1]
// now you can play around with these lines, as well as with the synth def above
s1 = Synth.replace( s1, \x, [\freq, 3000]);
s1.set(\freq, 4000);

// add a bus:

b1 = Bus.control(s);
b1.set(350);
```

```
s1.map(\freq,  b1);

// set the bus to different values:

b1.set(100);
b1.xline(800, 5);

s3 = { Out.kr( b1.index, MouseX.kr(300, 900, 1)) }; // add some mouse control on the fly
          // remove it again.



// finish:

b1.free;
d.clear;
d.pop;
```

ID: 264

```
{ 10 + 6 *  harry }.asCompileString;
```

many objects understand **.storeOn**, which a way to post their string that is needed to reproduce them by compilation.
sometimes one wants to store a certain configuration of a proxy space, which can be done
if all functions used are closed functions.

```
// an example how ProxySpace can document its current state:

p = ProxySpace.push(s);



(
ctl1 = {
var z = 1;
4.do { | i|  z = z * SinOsc.kr(i.sqrt, i+[0,0.2]) };
z
};

ctl2[0] = { LFNoise2.kr([20,20],20) };
ctl2[1] = {
LFNoise2.kr([20,20],20) * LFNoise0.kr([20,20],20)
};

out = {
SinOsc.ar( freq.kr, 0, 0.1)
};

freq[0] = {  ctl1.kr(2) +  ctl2.kr(2) + 400 };
freq[5] =  ctl1.wrap2( ctl2) *  ctl1 / ( ctl2 +  ctl1);

pat = Pbind(\freq, Pfunc({ 1.2.rand }));
z = 9;
out.set(\freq, 760, \ffreq, 20);
)
```

```
p.asCompileString;

// the document message creates a new document which it posts the code into

              // document everything
            \out     // document all dependants of  out
            \ctl1    // document all dependants of  ctl1
```

ID: 265

some placeholders in supercollider 1

this helpfile explains some basic concepts of interactive programming
with supercollider and proxy space.

previous: [JITLib] next: [jitlib_basic_concepts_02]

a) What is a proxy?

A proxy is a place holder that is often used to operate on something that does not yet exist.
For example, an *OutputProxy* is used to represent multiple outputs of a ugen, even if only
one ugen is created eventually.
Any object can have proxy behaviour (it may delegate function calls to different objects for example)
but specially functions and references can be used as operands while they keep their referential quality.

see also: [OutputProxy] [Function] [Ref]

using a Ref as a proxy:

```
// reference example

// create a new Ref object
y = `(nil);

// you can start to calcuate with y, even if its value is not yet given:
            // returns a function
```

```
// now the source can be set:
y.value = 34;

// the function z can be evaluated now.
z.value


// the same without a reference does not work:

    nil  // empty y first

            // this fails.

// also an array does not provide this referentiality:

     nil   // array with nil as element

            // this fails.

// an environment without sufficient defaults has the same problem:

currentEnvironment.postln; // anEnvironment
    // access the enironment:  there is nothing stored:  nil
        // store something
     // now 9 is stored
          // calculate with it

                          // the value is stored in the environment

         // cause an error:   y is nil.
y = -90; // set  y

         // this works.
```

using a Function as a proxy:

```
// a function can serve the same purpose

    nil  // empty y first
                // this does not fail, instead it creates a new function, which
   // does not fail when evaluating it after y is set to 34.

y = 34;
z.value;


see also client side proxies like [Tdef][Pdefn][Pdef]
```

b) NodeProxy

For interactive programming it can be useful to be able to use something before it is there - it makes evaluation order more flexible and allows to postpone decisions to a later moment. Some preparations have to be done usually - like above, a reference has to be created. In other situations this sort of preparation is not enough, for example if one wants to do maths with running processes on the server.

Audio output on the server has mainly two properties - a *calculation rate* (audio or control) and a certain *number of channels*. These are the main static properties of a node proxy, which cannot be changed while it is in use.

```
// boot the server
s.boot;

// two proxies on a server.  calculation rate is audio rate, number of channels is 2
y = NodeProxy.audio(s, 2);
z = NodeProxy.audio(s, 2);

// use them in calculation
z.play;
z.source = y.sin * 0.2;
```

```
// set its source now.
y.source = { Saw.ar([300, 301], 4*pi) };

// the source can be of various type, one of them would be a number:
y.source = 0.0;

// post the source
y.source;

// end them, free their bus number
y.clear;
z.clear;
```

In order to provide a simple way of creating node proxies, a proxy space can be used. So the above reads like this:

```
    ProxySpace              // store proxy space in p so it can be accessed easily.
z.play;


z =  y.sin * 0.2;


y = { Saw.ar([300, 301], 4*pi) };


// clear the space (all proxies)
p.clear;

// move out of the proxyspace.
p.pop;
```

further readings: [NodeProxy] [ProxySpace] [Ndef]

next: [jitlib_basic_concepts_02]

ID: 266

proxy space - basic concepts 2

external structure of the node proxy, referencing in proxyspace and environments.

previous: [jitlib_basic_concepts_01] next: [jitlib_basic_concepts_03]

a) normal environment lookup
b) proxyspace as an environment
c) using the proxyspace to change processes on the fly
d) when are the node proxies initialized?
e) moving out of the proxy space
f) using ProxySpace together with other Environments

a) normal environment lookup

```
                        // anEnvironment (if not, you haven't left it from last helppage..)

    // access the environment:  there is nothing stored:  nil
        // store something
     // now 9 is stored
          // calculate with it

                        // the value is stored in the environment

         // cause an error:   y is nil.
b = -90; // set  y

         // this works.
```

```
// note that you can always access environments (or ProxySpaces) from outside as well:

x = currentEnvironment;
  \a      \b  // equivalent to  b +  a

// or, if "know" is true,
x.know = true;
x.a + x.b;
```

further readings: [Environment]

b) proxyspace as an environment

one can replace the current environment with a special type of environment, a proxy space
this environment represents processes that play audio on a server.

```
    ProxySpace          // create a new environment, store it in variable p for now.
          // push it, so i becomes the current environment.
currentEnvironment.postln;
currentEnvironment === p; // this is identical.

      // accessing creates a NodeProxy (uninitialized) automatically.
          // this works immediately, because the lookup does not return nil,
     // but a placeholder (proxy) instead

          // now there is two placeholders in the environment.
```

c) using the proxyspace to change processes on the fly

```
//  boot the server
s.boot;


// as soon as a sound function (or any compatible input) is assigned to a proxy
// this sound plays on its own private bus (so it is not audible yet.)
(
x = {

RLPF.ar(Impulse.ar(4) * 20, [850, 950], 0.2)

}
)

// the proxy has been initialized by its first assignment.
// it plays at audio rate (because we have assigned an audio rate ugen function)
// and it has two channels (because the function has stereo output)

          // what bus index is it?  this posts the index to the postwindow
  // before it was .ir(nil), now it is initialized to .ar(2)
       // what bus is it?


         // now listen to it.  a monitor is created (see [Monitor]) that plays
  // the signal onto a public bus.  This is independent of the proxy itself.
    // for further info see:  [jitlib_basic_concepts_03] (part c)



// the sound function can be changed at any time:
(
x = {
RLPF.ar(Impulse.ar([5, 7]) * 5, [1450, 1234], 0.2)
```

```
}
)

// You can tune a sound function to your liking very easily
// by replacing it with little (or big) variations:

    // filter freqs higher:
x = { RLPF.ar(Impulse.ar([5, 7]) * 5, [1800, 2000], 0.2) }

  // same pulse ratio (5/8), different pulse tempo:
x = { RLPF.ar(Impulse.ar([5, 8] * 3.2) * 5, [1800, 2000], 0.2) }

  // different filter:
x = { Ringz.ar(Impulse.ar([5, 8] * 3.2), [1800, 2000], 0.05) }

// and if you set the proxy's fadeTime, you can create little
// textures by hand:

x.fadeTime = 3;
  // different filter freqs every time:
x = { Ringz.ar(Impulse.ar([5, 8] * rrand(0.5, 1.5)) * 0.5, ({ exprand(200, 4000) } !  2), 0.05) }



// here is another proxy:
y = { Pan2.ar(Dust.ar(20), 0) };

        // it has two channels, just as the  x., but it plays on another (private) bus.

// note that  y is not audible directly,
// but it can be used in any other proxy:
(
x = {

RLPF.ar( y.ar * 8,  [1450, 1234], 0.2)
}
)

// when the proxy changes, the result changes dynamically:
```

```
y = { Impulse.ar(MouseX.kr(2, 18, 1)) * [1, 1] };

y = { PinkNoise.ar(MouseX.kr(0, 0.2) * [1, 1]) };

y = { Impulse.ar([MouseX.kr(2, 18, 1), MouseY.kr(2, 18, 1)]) };



// stop listening.  the proxies run in the background.

x.stop;

        //  y is playing on another bus.
        // than  x

// we can also listen to  y directly:
y.play;

// to remove an input, nil can be used:

y = nil;

// stop listening
y.stop;
```

further readings: [proxyspace_examples] [Bus] [AbstractFunction] [UGens]

d) when are the node proxies initialized?

bus initialization of a node proxy happens as soon as it is used for the first time. later inputs are adjusted to this bus, as far as it is possible.

```
        LFNoise0                    // a four channel control rate proxy
z2.bus.postln;
```

```
            // a constant value causes a single channel control rate proxy.
z100.bus.postln;

            // the first access allocates the bus
                 // a 3 channel audio proxy

// these initializations can be removed by using clear:
z34.clear;
z34.bus.postln;
```

This initialisation happens whenever the proxy is first used. Later, the proxy can be accessed with other rate/numChannels combinations as needed (rates are converted, numChannels are extended by wrapping).

Note that this might cause ambiguous initialisation in which case the proxy should be always initialized first. A typical problem is demonstrated here:

```
                // initialize 2 audio channels (default).  0 is the output bus number.
        // if the proxy is not inititialized, play defaults to 2 channels.
        // here it is explicitly given only to make it more clear.
u = { PinkNoise.ar(0.2) }; // use only one
u.numChannels; // 2 channels
u.clear;
```

if evaluated the other way round, only one channel is used:

```
       PinkNoise            // initialize 1 audio channel
                // play 2 channels:  the 1 channel is expanded into 2.
        // numChannels of .play defaults to the proxy's numChannels.
        // here it is explicitly given, so to expand the channels
u.numChannels; // 1 channel
u.clear;
```

Thus it can be useful to explicitly initialize proxies that use variable type inputs:

```
                 // explicit initialisation
          // post the whole proxy space
```

e) moving out of the proxy space:

```
// play the audio:
x.play;

x = { PinkNoise.ar(0.5) };

// p is the proxy space:
p.postln;

// to end all processes in p, use end:
        // 2 seconds fade out.

// to remove all bus objects and free them from the bus allocato, use clear:
p.clear;

currentEnvironment.postln;

// restore original environment:

p.pop;

currentEnvironment.postln;

        // the old values are still here.

                        // this is not the case anymore.

// remove the content, so the garbage collector can release their memory.
p.clear;

// note that if you use this kind of accessing scheme, the objects are not garbage collected
// until the keys are set to nil.  This is a common mistake when using normal environments.

// clear all in the normal environment:

currentEnvironment.clear;
```

f) using ProxySpace together with other Environments

using proxy space as an access scheme for node proxies can get in the way of the normal use of environments as pseudo variables. Here is some ways to cope with this.

```
// if you want to keep using the current environment as usual, you can restrict the
// scope of proxyspace to one document (note:  this is mac-only currently)

EnvirDocument    "proxyspace"    // to test this, check for currentEnvironment here
       // and in the envir document.

// you can also access the proxy space indirectly:
p[\x].play;
p[\x] = { SinOsc.ar(450, 0, 0.1) };

// or:  when the proxyspace is pushed, you can use a normal environment indirectly:
p.push;
d = ();
d[\buffer1] = Buffer.alloc(s, 1024);
                                    // for more than one access to the environment, use use



// to access the inner environment of proxy space directly,
// without creating new proxies, use .envir:

p.envir.postln;
p.envir[\x].postln;

// this can be useful for lookup, when you want to know if a certain proxy exists already.
// direct access would create that proxy, which would not make sense in that case.
```

previous: [jitlib_basic_concepts_01] next: [jitlib_basic_concepts_03]

ID: 267

proxyspace - basic concepts 3

internal structure of the node proxy, node order and the parameter context

a) slots
b) fadeTime
b) play/stop, send/release, pause/resume, clear
c) the parameter context

A NodeProxy has two internal contexts in which the objects are inserted:
The group, which is on the server, and the nodeMap, which is a client side parameter context. As the group can contain an order of synths, there is a client side representation, in which the source objects are stored (see [Order]).

previous: [jitlib_basic_concepts_02] next: [jitlib_basic_concepts_04]

```
// make new space
p = ProxySpace.push(s.boot);
                // explicitly initialize proxies
```

a) NodeProxy slots:

One node proxy can hold several objects in an execution order. The index can be any positive integer.

```
// the initial slot (0) is used when assigning directly.
//  y is still unused, we will add it later.

z = ( y * pi).sin * 0.1 * { LFSaw.kr(LFNoise1.kr(0.1 !  3).sum * -18).max(0.2) };
```

```
// other slot numbers are accessed by positive integers:

y[1] = { Saw.ar([400, 401.3], 0.4) };
y[0] = { Saw.ar([300, 301], 0.4) };

// to remove one of them, nil is used:

y[0] = nil;

// what is to be found at index 1?
      // a playing interface
                  // the function that was put in.
           // this returns objects in the slots.
```

multiple assignment

```
// the function is assigned to th slots from 1 to 4
z[1..4] = { SinOsc.ar(exprand(300, 600), 0, LFTri.kr({exprand(1, 3)} !  3).sum.max(0)) * 0.1 };


// the function is assigned to the slots 1, 2 and 3 (subsequent)
z[1..]  = [ {SinOsc.ar(440) * 0.1 }, { SinOsc.ar(870) * 0.08 }, { SinOsc.ar(770) * 0.04 }];

// if no slot is given, all other slots are emptied
z = { OnePole.ar(Saw.ar([400, 401.3], 0.3), 0.95) };

z.end;
y.end;
```

b) fade time:

```
// setting the fadeTime will allow cross fades.
// in case of an audio rate proxy the fade is pseudo-gaussian
// in case of a control rate proxy it is linear.

z.play;
```

```
z.fadeTime = 5.0; // 5 seconds
z = { max(SinOsc.ar([300, 301]), Saw.ar([304, 304.3])) * 0.1 };
z = { max(SinOsc.ar(ExpRand(300, 600)), Saw.ar([304, 304.3])) * 0.1  };

// the fadeTime can be set effectively at any time
z.fadeTime = 0.2;
z = { max(SinOsc.ar(ExpRand(3, 160)), Saw.ar([304, 304.3])) * 0.1 };
```

note that the fadeTime is also used for the operations xset and xmap.(see below)

c) play/stop, send/free, pause/resume

there are a couple of messages a NodeProxy understands that are related to play, stop etc.
Here is what they do.

play/stop

this pair of messages is related to the monitoring function of the proxy.
play starts monitoring, stop ends the monitoring.
if the proxy group is playing (this can be tested with .isPlaying), play will not affect the proxie's internal behaviour in any way. Only if it is not playing (e.g because one has freed the group by cmd-period) it starts the synths/objects in the proxy.
Stop never affects the internal state of the proxy.

```
     // first hit cmd-period.
z =  { max(SinOsc.ar(ExpRand(3, 160)), Saw.ar([304, 304.3])) * 0.1 };
  z.play; // monitor the proxy
  z.stop; // note that now the proxy is still playing, but only in private
  z.isPlaying; // is the group playing?  yes.
   z.monitor.isPlaying; // is the monitor playing?  no.
```

You can pass a vol argument to play to adjust the monitor volume without affecting the proxy internal bus volume.

```
z.play(vol:0.3);
 // while playing you can set the volume also:
z.vol = 0.8;
```

send / release

this pair of messages controls the synths within the proxy. It does not affect the monitoring (see above). send starts a new synth, release releases the synth. **send** by default releases the last synth. if the synth frees itself (doneAction 2) **spawn** can be used.

```
  // first hit cmd-period.
   z.play; // monitor.  this starts also the synth, if the group wasn't playing.

z = { SinOsc.ar(ExpRand(20, 660) !  2) * Saw.ar(ExpRand(200, 960) !  2) * 0.1 };

  z.release; // release the synth.  the current fadeTime is used for fade out

  z.send; // send a new synth.  the current fadeTime is used for fade in

  z.send; // send another synth, release the old

z.release;

z.stop;

  z.play; // monitor.  as the group was still playing, this does _not_ start the proxy.
```

in order to free the synths and the group together, **free** is used:

```
   z.free; // this does also not affect the monitoring.
   z.play; // monitor.  as the group was not playing, this starts the proxy.
```

in order to free the synths and the group, stop playback, **end** is used.

```
   z.end(3); // end in 3 sec
```

in order to rebuild the synthdef on the server, use **rebuild**.

this can make sense when the synthdef has a statistic architecture (but of course this is far less efficient than *send*)

```
 (
  z = {
sum(
 SinOsc.ar(Rand(300,400) + ({exprand(1, 1.3)} !  rrand(1, 9)))
* LFCub.ar({exprand(30, 900)} !  rrand(1, 9))
* LFSaw.kr({exprand(1.0, 8.0)} !  rrand(1, 9)).max(0)
* 0.1
)
};
)


  z.play;
z.rebuild;
  z.send; // send just creates a new synth
  z.rebuild; // rebuild the synthdef
z.end;
```

pause / resume

when paused, a node proxy still stays active, but every synth that is started is paused until
the proxy is resumed again.

```
z.play;


  z.pause; // pause the synth.


z = { SinOsc.ar({ExpRand(300, 660)} !  2) * 0.1 };  // you can add a new function,
            // which is paused.


   z.resume; // resume playing.
```

Note that pause/resume causes clicks with audio rate proxies, which do not happen when pauseing control rate proxies.

clear

clear removes all synths, the group, the monitor and releases the bus number.

```
z.clear;
z.bus;   // no bus
  z.isNeutral; // not initialized.
```

note that when other processes use the nodeproxy these are not notified. So clearing has to
be done with regard to this.

d) The parameter context

what happens to function arguments?

```
y.play;
y = { arg freq=500; SinOsc.ar(freq * [1, 1.1]) * 0.1 };
```

now the argument 'freq' is a control in the synth (just like in SynthDef) which you can change by the '**set**' message.

```
y.set(\freq, 440);

 // unlike in synths, this context is kept and applied to every new synth:

y = { arg freq=500; Formant.ar(50, freq * [1, 1.1], 70) * 0.1 };
```

xset is a variant of set, to crossfade the change using the current fadeTime:

```
y.fadeTime = 3;
y.xset(\freq, 600);

 // the same context is applied to all slots:

y[2] = { arg freq=500; SinOsc.ar(freq * [1, 1.1]) * LFPulse.kr(Rand(1, 3)) * 0.1 };
y.xset(\freq, 300);
```

the parameter context also can keep bus mappings. a control can be mapped to any *control proxy*:

```
c = { MouseX.kr(300, 800, 1) };
y.map(\freq,  c);

 // also here the context is kept:

y = { arg freq=500; Formant.ar(4, freq * [1, 1.1], 70) * 0.1 };
```

xmap is a variant of map, to crossfade the change using the current fadeTime:

```
y.set(\freq, 440);
y.xmap(\freq,  c);
```

to remove a setting or a mapping, use **unmap / unset**.

```
y.unmap;
```

a multichannel control can be mapped to a multichannel proxy using **mapn**:

```
c2 =  { [MouseX.kr(300, 800, 1), MouseY.kr(300, 800, 1)] };
y = { arg freq=#[440, 550]; SinOsc.ar(freq) * SinOsc.ar(freq + 3) * 0.05 };
y.mapn(\freq,  c2);
```

the parameter context can be examined:

```
y.nodeMap;

 // apart from the parameters explicitly set,
 // it contains the bus index and the fadeTime

 // for more information, see [NodeMap]


            // clear the whole proxy space, in 8 secs.
```

previous: [jitlib_basic_concepts_02] next: [jitlib_basic_concepts_04]

ID: 268

Timing in NodeProxy 4

Changes that happen to NodeProxy, most importantly setting its source, are normally done
whenever the put method is called (or, in ProxySpace, the assignment operation =).
Sometimes it is desirable to time these changes relative to a clock.

previous: [jitlib_basic_concepts_03] next: [JITLib]

a) clock
b) quant and offset
c) client and server tempo
d) sample accurate output

a) clock

generally, every node proxy can have its own time base, usually
a tempo clock. the clock is responsible for the timing of insertion
of new functions, per default at the next beat of the clock.

```
p = ProxySpace.push(s.boot);
x.play;  y.play;

// these two synths are started at the time when they are inserted:
x = { Ringz.ar(Impulse.ar(1), 400, 0.05) };
y = { Ringz.ar(Impulse.ar(1), 600, 0.05) };

// adding a common clock:
x.clock = TempoClock.default;
y.clock = TempoClock.default;

// now they are in sync
```

```
x = { Ringz.ar(Impulse.ar(1), 400, 0.05) };
y = { Ringz.ar(Impulse.ar(1), 600, 0.05) };

// for simplicity, one can provide a clock for a whole proxy space:

p.clock = TempoClock.default;
y = { Ringz.ar(Impulse.ar(1), 800, 0.05) };

z.play;
z = { Ringz.ar(Impulse.ar(1), [500, 514], 0.8) };
z = { Ringz.ar(Impulse.ar(1), exprand(300, 400 !  2), 0.8) };
z = { Ringz.ar(Impulse.ar(2), exprand(300, 3400 !  2), 0.08) };
z.end;
```

sequence of events:
When inserting a new function into the proxy, the synthdef is built, sent to the server who sends back a message when it has completed. Then the proxy waits for the next beat to start
the synth. When using node proxies with patterns, the patterns are played using the clock as a scheduler.

b) quant and offset

In order to be able to control the offset/quant point of insertion, the 'quant' instance variable can be used,
which can be either a number or an array of the form [quant, offset], just like in pattern.play(quant).

```
                    // offset of 0.3, quant of 1.0
y = { Ringz.ar(Impulse.ar(1), 600, 0.05) };
```

quant and offset scheduling is used for the following operations:
play, put, removeAt, setNodeMap, wakeUp, rebuild (and the rebuild operations lag, setRates, bus_)

c) connecting client and server tempo

a ProxySpace has the method makeTempoClock, which creates an instance of TempoBusClock
together with a node proxy (tempo) which it keeps in sync.

```
                    // create a new tempoclock with 2 beats/sec

            // set the quant back to 1 and the offset to 0
y = { Ringz.ar(Impulse.ar( tempo.kr), 600, 0.05) }; // impulse uses tempo
x = Pbind(\instrument, \default, \freq, Pseq([300, 400], inf)); // pattern uses tempoclock

p.clock.tempo = 1.0; // set the tempo to 1
                    // set the tempo to 2.2

x.free;
y.free;
```

d) sample accurate output

for efficiency, NodeProxy uses a normal Out UGen for writing to its bus. If sample accurate
playback is needed (OffsetOut), the ProxySynthDef class variable sampleAccurate can be set to true:

```
// example

ProxySynthDef.sampleAccurate = false;

x.play;
```

```
// the grain frees itself
x = { SinOsc.ar(800) * EnvGen.ar(Env.perc(0.001, 0.03, 0.4), doneAction:2) };


// jittery tone.
(
    Routine
loop {
200.do { arg i;
x.spawn;
(0.005).wait;
};
1.wait;
}
}.play;
)

ProxySynthDef.sampleAccurate = true;

// steady tone, because sample accurate.

x.rebuild;

r.stop;

        // remove all.
```

previous: [jitlib_basic_concepts_03]

ID: 269

Efficient coding with NodeProxy

NodeProxy (and, in disguise ProxySpace) represent "pronouns", placeholders for all kinds of sound producing objects that are able to write to a specific bus on the server.

To prepare such an object for playing, different objects require different preparation, some very little, some more. As working with the placholders does not show directly which actions take hardly any calculations and which are expensive, it is shown here more in detail.
This is also important if you want to automate certain processes - some things in this library are not meant to be used in certain ways (A) and better solutions should be used instead then, others are much more efficient (B, C)

```
a = NodeProxy.audio;
ProxySpace.push;

a.source = ...  is equivalent to  a = ...
a.add(...)  a.put(0,...)  a[0] = ...   a[0] = ...  are equivalent in cpu load.
```

A) rebuild and send: manual rate
the following code requires a rebuild and send of a SynthDef and is thus most cpu-expensive.
though fine for slower use (esp.hand-use) for automatisation it is better to build a synthdef and assign it

```
a = { someUGenFunction };
a = Patch(instrname, args);
a = SynthDef(\name, { someUGenFunction });


 // the same applies to rebuilding the graphs:
a.rebuild

 // this rebuild is also used when setting one of the following properties:
server, bus, setRates
```

B) starting synths and tasks

the following code sends commands to the server to start synths, which is load mainly on the server
and depends on the characteristics of the synthdef:

```
   a = \synthDefName  // the synthdef is already on the server
a = Pbind(\instrument, name, \freq, ...);
a = Routine({ loop({ s.sendMsg("/s_new", name, ...)})  });

  a.refresh;  a.wakeUp; // waking up a stopped proxy does not require a resend
```

these resend the synth with new properies

```
  a.send(...)  // directly sends a message.  the mapping bundle of the proxy is cached
a.sendAll(...)

        // for the following the bundle is recalculated if a new key is assigned.
        // if you use the same key with a different value, the bundle is modified

a.xset(...)   a.xmap(...)
a.nodeMap_(a map)
a.fadeToMap(a map)

 // synthdefs for these things are on the server already.

a.gate,  a.env,  a.line,  a.xline

 // some more calculations have to be made on client side, so if automated, it is better to use
 // the above or a lag in the synth.

a.lineAt(key),  a.xlineAt(key)
```

C) sending messages to running synths: for these the least calculation has to be done

```
a.set(\freq, 400, \dt, 0.2);  a.unset(\freq); // if running the bundle will be recalculated
a.map(\freq,  lfo);  a.unmap(\freq);
a.fadeTime = 2;
```

```
a.gateAt(key)

 // for avoiding bundle recalculation you can directly talk to the group.
  // this setting will not be kept when you exchange the synth
a.group.set(\freq, 500);
```

switching audio

control rate sources can be easily and efficiently switched using **map** or **xmap**.
here is an example of how already running audio rate inputs can be switched.
it is about as efficient as (B) - first example (setting a defname)
it works only for 1 or 2 channels right now.

```
(
s = Server.local;
p = ProxySpace.push(s.boot);
)



out.play;

s1 = { Blip.ar(Rand(32,15), 100, 0.5) };
s2 = { SinOsc.ar(740, 0, 0.1) };
s3 = { Pulse.ar(140, 0.2, 0.1) };


out = {  Pan2.ar( mix.ar(1), MouseX.kr(-1,1)) };

mix.read( s1);
mix.read( s2);
mix.read( s3);

//resetting the source stops reading
mix = \default;

//now you can also crossfade audio efficiently:
```

```
mix.fadeTime = 1.5;

mix.read( s1);
mix.read( s2);
mix.read( s3);

// automation:
(
t = Task({
var dt;
loop({
dt = rrand(0.01, 0.3);
mix.fadeTime = dt;
mix.read([ s1,  s2,  s3].choose);
dt.wait;
});
});
)

       SystemClock


// change the sources meanwhile:
s1 = { Blip.ar(105, 100, 0.2) };
s2 = { SinOsc.ar(350, 0, 0.1) };
s3 = { Pulse.ar(60, 0.2, 0.1) };

freq = { MouseX.kr(200, 600, 2) };

s1 = { Blip.ar( freq.kr * 0.3, 10, 0.2) };
s2 = { SinOsc.ar( freq.kr, 0, 0.1) };
s3 = { Pulse.ar( freq.kr * 0.2, 0.2, 0.1) };


t.stop;

// note that when restarting  out, the inputs have to be woken up.
// to avoid this, you can add the inputs to the mix nodeMap parents:

mix.nodeMap.parents.putAll( (s1:   s1, s2:   s2, s3:   s3) );
```

```
// also the task can be added to the proxy:
(
mix.task = Routine({
loop({
mix.fadeTime = rrand(0.01, 0.1);
mix.read([ s1,  s2,  s3].choose);
[0.2, 0.4].choose.wait;
});
});
)
```

ID: 270

Fade envelope generation and crossfading

NodeProxy (ProxySynthDef) looks for inner envelopes in your definition function to find out whether **a fade envelope is needed or not**. In case there is no other inner possibility of freeing the synth, either

a) a fade envelope is created (audio / control rate output)

b) the synth is freed directly with no fading (scalar output or doneAction 1)

c) if you provide a gate arg and a doneAction 2 to your ugenGraph function, this is supposed
to be a fade envelope for the synth

d) if a synthdef name is used, case c) is supposed

... so in most cases, there is not much to worry about, just these two points ar important,
if one wants to use a self releasing synth or a different out ugen:

e) own responsibility:
if the function creates a ugengraph that can be freed by trigger or other things, it waits for this action instead of the node proxy freeing the synth.

f) own out channel with 'out' arg: the control ugen with the name 'out' is set to the output channel
number of the proxy.

```
p = ProxySpace.push(s.boot);


out.play;


// note that you can use this functionality also when using ProxySynthDef directly:


d = ProxySynthDef("test", { arg freq=440; SinOsc.ar(freq) }).send(s);
s.sendMsg("/s_new", "test", 1980, 1, 1, \freq, 340);
```

```
s.sendMsg("/n_set", 1980, \freq, 240);
s.sendMsg("/n_set", 1980, \fadeTime, 4);
s.sendMsg("/n_set", 1980, \gate, 0);
```

a) automatic fade envelope generation

```
// no inner envelope and audio / control rate output
(
out = { PinkNoise.ar([1,1]*0.1) };
)

(
kout = { PinkNoise.kr([1,1]*0.1) };
)
```

b) automatic free instead of crossfade

```
// inner envelope that cannot free the synth, the synth is freed when a new
// function is assigned.
(
out = { arg t_trig; EnvGen.kr(Env.asr, t_trig) * PinkNoise.ar([1,1]) };
)
out.group.set(\t_trig, 1);

(
out = { arg t_trig; EnvGen.kr(Env.asr, t_trig) * SinOsc.ar([1,1]*400) };
)
out.group.set(\t_trig, 1);


// for a scalar output also no fade env is created, but the synth is freed (without fading)
(
out = { Out.ar(0, SinOsc.ar(Rand(440,550),0,0.2)) };
)
```

c) custom fade envelope

```
// when a gate arg is provided, and the env can free the synth, this envelope
```

```
// is supposed to be the fade envelope for the synth:  no extra fade env is created.
(
out = { arg gate=1; EnvGen.kr(Env.asr, gate, doneAction:2) * 0.2 * SinOsc.ar([1,1]*Rand(440,550)) };
)
```

d) SynthDef name assignment

```
// if a symbol is used as input, the defname of a def on the server is supposed
// to represent a SynthDef that has a gate, an out input and can free itself.
(
out = \default;
)

// this is the minimal requirement arguments for such a use (similar to Pbind)
(
SynthDef("test", { arg gate=1, out;
Out.ar(out, Formant.ar(300, 200, 10) * EnvGen.kr(Env.asr, gate, doneAction:2))
}).send(s);
)

// you can also provide a fadeTime arg, whic is set by the proxy:
(
SynthDef("test", { arg gate=1, out, fadeTime=1;
Out.ar(out,
Formant.ar(Rand(20,40), 600, 10, 0.2)
* EnvGen.kr(Env.asr(fadeTime,1,fadeTime), gate, doneAction:2)
)
}).send(s);
)

out = \test;
out.fadeTime = 3;
```

note that the **number of channels** is your own responsibility when using symbols,
as a symbol carries no channel information!
(in all other cases the number of channels is wrapped or expanded to fit the proxy)

```
// if the synthdef has a fixed duration envelope, there is a FAILURE /n_set Node not found message.
// with no further significance
(
```

```
SynthDef("test", { arg gate=1, out;
Out.ar(out,
Formant.ar(Rand(20,40), 600, 10, 0.6)
* EnvGen.kr(Env.perc, gate, doneAction:2)
)
}).send(s);
)

out = \test;
```

e) own free responsibility

```
//inner envelope that can free the synth, no extra fade env is created:
(
out = { arg t_trig; EnvGen.kr(Env.asr, t_trig, doneAction:2) * PinkNoise.ar([1,1]) };
)
out.group.set(\t_trig, 1); //end it

           //start a new synth
out.group.set(\t_trig, 1); //end it again


// if there is a ugen that can free the synth, no extra fade env is created either,
// but it supposes the synth frees itself, so if a new function is assigned it does
// not get freed.
(
out = {  arg t_trig;
 FreeSelf    t_trig
PinkNoise.ar([1,1]*0.3);
};
)
out.group.set(\t_trig, 1);
```

f) own output responsibility

```
// the arg name 'out' can be used to output through the right channel.
// of course with this one can get problems due to a wrong number of channels
```

```
// or deliberate hacks.

//left speaker
(
out = { arg out;
OffsetOut.ar(out, Impulse.ar(10,0,0.1))
}
)

//both speakers
(
out = { arg out;
OffsetOut.ar(out, Impulse.ar([10, 10],0,0.1))
}
)

//this plays out into another adjacent bus:  this is a possible source of confusion.
(
out = { arg out;
OffsetOut.ar(out, Impulse.ar([10, 10, 10],0,0.1))
}
)
```

ID: 271

networked programming

please note any problems, I'll try to add solutions here.

1) using ProxySpace with more than one client, with separate bus spaces

Note: if only one client is using a remote server, only step (a) and step (d) are relevant. The clientID argument can be left out then.

before you start:
remember to synchronize your system clocks. This can be done by:
in OS X: SystemPreferences>Date&Time: set *"Set Date & Time automatically"* to true.
in linux: set the ntp clock
a local time server is better than the apple time server.
if you cannot sync the time, you can set the server latency to nil.
This will break the pattern's functionality though.

a) boot the (remote) server and create a local model
(you cannot directly boot a remote server instance)

```
s = Server("serverName", NetAddr(hostname, port), clientID);
```

serverName can be any name
hostname is an ip address, or if you have a name resolution, a network name
port the port on which the server is listening. default is 57110
clientID for each client (each sc-lang) *a different integer number has to be given*
see [Server]

b) from each client, initialize the default node and set notify to true:

```
s.boot; // this will initialize the tree and start notification

    // if needed, a server window can be created:
s.makeWindow;
```

c) preallocate a range of busses in each client.
If there is conflicts, increase the number of busses in the server options before booting:

```
(s.options.numAudioBusChannels = 1024;)

(
var numberOfParticipants, n;
numberOfParticipants = 4;


n = s.options.numAudioBusChannels / numberOfParticipants;
n = n.floor.asInteger * s.clientID;
s.audioBusAllocator.alloc(n);


n = s.options.numControlBusChannels / numberOfParticipants;
n = n.floor.asInteger * s.clientID;
s.controlBusAllocator.alloc(n);
)
```

d) now create a ProxySpace from the server:

```
p = ProxySpace.push(s);
```

2) using ProxySpace with more than one client, with a partly shared bus space

step a, b like in (1), skip (d)

c) before allocating a number of busses for each client, create shared busses:

```
p = ProxySpace.push(s);
shared1.ar(2);
shared2.ar(2);
sharedkr.kr(1); // or other names.
```

then do (c) like in (1), just take care that the shared busses are taking number space
already, so the easiest is to increase the numberOfParticipants by one, so no overrun happens.

3) writing a chat

see example in**[Client]**

see also [Public]

ID: 272

proxy space examples

preparation of the environment

```
(
s = Server.local;
s.boot;
p = ProxySpace
)
```

playing and monitoring

```
// play some output to the hardware busses, this could be any audio rate key.
out.play;

out = { SinOsc.ar([400, 408]*0.8, 0, 0.2) };


// replacing the node.  the crossfade envelope is created internally.
out = { SinOsc.ar([443, 600-Rand(0,200)], 0, 0.2) };
out = { Resonz.ar(Saw.ar(40+[0,0.2], 1), [1200, 1600], 0.1) + SinOsc.ar(60*[1,1.1],0,0.2) };
out = { Pan2.ar(PinkNoise.ar(0.1), LFClipNoise.kr(2)) };
```

setting the node controls

```
out = { arg rate=2; Pan2.ar(PinkNoise.ar(0.1), LFClipNoise.kr(rate)) };
out.set(\rate, 30);
out = { arg rate=2; Pan2.ar(Dust.ar(2000, 0.2), LFClipNoise.kr(rate)) };
out.set(\rate, 2);
```

referencing between proxies

```
lfo = { LFNoise2.kr(30, 300, 500) };
out = { SinOsc.ar( lfo.kr, 0, 0.15)  };
out = { SinOsc.ar( lfo.kr * [1, 1.2], 0, 0.1) * Pulse.ar( lfo.kr * [0.1, 0.125], 0.5) };
```

```
lfo = { LFNoise1.kr(30, 40) + SinOsc.kr(0.1, 0, 200, 500) };
out = { SinOsc.ar( lfo.kr * [1, 1.2], 0, 0.1)  };
lfo = 410;
```

math

```
// unary operators
lfo2 = { SinOsc.kr(0.5, 0, 600, 100)  };
lfo =  lfo2.abs;
lfo2 = { SinOsc.kr(1.3, 0, 600, 100)  };


// binary operators
lfo3 = { LFTri.kr(0.5, 0, 80, 300) };
lfo =  lfo2 +  lfo3;
lfo =  lfo3;
lfo = ( lfo3 / 50).sin * 200 + 500 * { LFTri.kr( lfo.kr * 0.0015, 0, 0.1 *  lfo3.kr / 90, 1) };
lfo3 = { Mix( lfo2.kr * [1, 1.2]) };

currentEnvironment.free; // free all node proxies
          // free the playback synth.
```

waking up a network of proxies

```
// hit cmd-.  to stop all nodes
// start again
out.play;
```

feeding back (one buffer size delay)

```
out = { SinOsc.ar([220, 330],  out.ar(2).reverse * LFNoise2.kr(0.5, 4*pi), 0.4) };

// there is no immediacy:  hear the buffer size cycle
out = { Impulse.ar(1 !  2) + ( out.ar(2) * 0.99) };



// supercollider 'differential equations'
```

```
out = { SinOsc.ar(Slope.ar( out.ar) * MouseX.kr(1000, 18000, 1)) * 0.1 + SinOsc.ar(100, 0, 0.1) };

(
out = { var z, zz;
z = Slope.ar( out.ar);
zz = Slope.ar(z);
SinOsc.ar(Rand(300,410), z) *
SinOsc.ar(zz * 410)
* 0.1 + Decay2.ar(Pan2.ar(Dust.ar(600), MouseX.kr(-1,1)), 0.01, 0.05);
}
)
```

multiple control

```
(
out = { arg freqOffest;
var ctl;
ctl = Control.names(\array).kr(Array.rand(8, 400, 1000));
Pan2.ar(Mix(SinOsc.ar(ctl + freqOffest, 0, 0.1 / 8)), LFNoise0.kr(2))
};
)

out.setn(\array, Array.exprand(8, 400, 2000));
out.set(\freqOffest, rrand(300,200));
out.map(\freqOffest,  lfo);

// a simpler short form for this is:
(
out = { arg freqOffest=0, array = #[ 997, 777, 506, 553, 731, 891, 925, 580 ];
Pan2.ar(Mix(SinOsc.ar(array + freqOffest, 0, 0.1 / 8)), LFNoise0.kr(2))
};
)
```

mixing

```
out1 = { SinOsc.ar(600, 0, 0.1) };
```

```
out2 = { SinOsc.ar(500, 0, 0.1) };
out3 = { SinOsc.ar(400, 0, 0.1) };
out =  out2 +  out1 +  out3;

out =  out1 +  out2;
out =  out1;

// another way is:
out = { SinOsc.ar(600, 0, 0.1) };
out.add({ SinOsc.ar(500, 0, 0.1) });
out.add({ SinOsc.ar(400, 0, 0.1) });

// or with direct access:
out[1] = { SinOsc.ar(500 * 1.2, 0, 0.1) };
out[2] = { SinOsc.ar(400 * 1.2, 0, 0.1) };
```

restoring / erasing

```
           // this frees the group, not the play synth x
           // resends all synths
out.free;
          nil      // this sends at index 1 only
out.send;

// removing:
out.removeLast;
out.removeAt(0);

// cleaning up, freeing the bus:
            // this neutralizes the proxy, and frees its bus
```

for more on the proxy slots see: [jitlib_basic_concepts_03]

garbage collecting

```
// often there are proxies playing that are not used anymore - this is good,
// because they might be used again at any time.
// this shows how to free unused proxies, such as  out1,  out2.

out.play;
out = { Pan2.ar(SinOsc.ar( lfo.kr, 0, 0.2), sin( lfo.kr / 10)) }; //  lfo is kept, as its parents.
lfo = { LFNoise2.kr(3, 160, 400) };

p.keysValuesDo { arg key, proxy; [key, proxy.isPlaying].postln };
          // all monitoring proxies (in this case  out) are kept.  equivalent:  p.reduce(to:  [ out]);

p.keysValuesDo { arg key, proxy; [key, proxy.isPlaying].postln };

// to remove everything else:
p.postln;
         // all monitoring proxies (in this case  out) are kept.
p.postln;

// after  out is stopped, it is removed, too:
           // stop monitor
p.clean;
          // empty space.
```

execution order

```
// you can .play .kr or .ar also a name that is not yet used.
// the rate is guessed as far as possible.  on this topic see also:  [the_lazy_proxy]

             // play some key (audio rate is assumed)

// the rate is determined from the first access:
// like this  lfo becomes control rate

myOut = { SinOsc.ar( freq.kr * 2, 0, 0.1) };
freq = 900;
freq = { SinOsc.kr(115, 0, 70, 220) }
```

```
myOut = { SinOsc.ar( otherFreq.ar * 2, 0, 0.1) };
otherFreq = { SinOsc.ar(115, 0, 70, 220) };

                        // clear every  proxy in this environment and remove them.  (same:  p.clear)
```

setting the xfade time

```
out.play;

out.fadeTime = 4;
out = { SinOsc.ar(Rand(800, 300.0)*[1,1.1], 0, 0.1) };
out = { SinOsc.ar(Rand(800, 300.0)*[1,1.1], 0, 0.1) };
out.fadeTime = 0.01;
out = { SinOsc.ar(Rand(800, 300.0)*[1,1.1], 0, 0.1) };
out = { SinOsc.ar(Rand(800, 300.0)*[1,1.1], 0, 0.1) };

            // release the synths and the group with a given fadeTime without changing proxy time
        // stop monitor
```

setting and mapping arguments

```
out.play;

out = { arg freq=500, ffreq=120; SinOsc.ar(freq*[1,1.1], SinOsc.ar(ffreq, 0, pi), 0.2) };
out.set(\freq, 400+100.rand2);
out.set(\freq, 400+100.rand2);
out.set(\ffreq, 30+20.rand2);
        \freq  \ffreq    // remove the setting
out.set(\ffreq, 30+10.rand2, \freq, 500 + 200.rand2);


// argument settings and mappings are applied to every new function
```

```
out = { arg freq=100, ffreq=20; SinOsc.ar(freq, SinOsc.ar(SinOsc.ar(ffreq)*ffreq, 0, pi), 0.2) };

// mapping to other proxies
lfo = { SinOsc.kr(0.3, 0, 80, 100) };
out.map(\ffreq,  lfo);

out = { arg freq=300, ffreq=20; Pulse.ar(freq*[1,1.1]+ SinOsc.ar(ffreq, 0, freq), 0.3, 0.1) };
out = { arg freq=300, ffreq=20; BPF.ar(LFSaw.ar(ffreq*[1,1.1], 0, 1), freq, 0.2) };

lfo = { FSinOsc.kr(0.3, 0, 30, 200) + FSinOsc.kr(10, 0, 10) };
out = { arg freq=300, ffreq=20; SinOsc.ar(freq*[1,1.1], SinOsc.ar(ffreq, 0, pi), 0.1) };


// crossfaded setting and mapping:  fadeTime is used
out.fadeTime = 2;
out.xset(\freq, 9000);
out.xset(\freq, rrand(400, 700));

lfo = { FSinOsc.kr(0.1, 0, 30, 100) };
lfo2 = { LFClipNoise.kr(3, 100, 200) };
lfo3 = StreamKrDur(Pseq([Prand([530, 600],1), 700, 400, 800, 500].scramble, inf) / 3, 0.2);

out.xmap(\ffreq,  lfo2);
out.xmap(\ffreq,  lfo);
out.xmap(\ffreq,  lfo3);

// argument rates:  just like a synthdef has input 'rates' (like \ir or \tr), a nodeproxy control
// can be given a rate.  this rate is used for each function passed into the proxy.

// trigger inputs
out = { arg trig, dt=1; Decay2.kr(trig, 0.01, dt) * Mix(SinOsc.ar(7000*[1.2, 1.3, 0.2])) }
out.setRates(\trig, \tr);

// set the group, so the node proxy does not store the new value
out.group.set(\trig, 0.1, \dt, 0.1);
out.group.set(\trig, 0.4, \dt, 0.31);
out.group.set(\trig, 0.13, \dt, 2);

// lagging controls:
        \xfreq      // equivalent to  out.setRates(\xfreq, 1);
```

```
(
out = { arg trig, dt=1, xfreq=700;
Decay2.kr(trig, 0.01, dt) * Mix(SinOsc.ar(xfreq*[1.2, 1.3, 0.2]))
};
)
out.group.set(\trig, 0.1, \dt, 1, \xfreq, rrand(2000,9000));
out.group.set(\trig, 0.1, \dt, 0.5, \xfreq, rrand(2000,9000));
out.group.set(\trig, 0.1, \dt, 1, \xfreq, rrand(2000,9000));

// changing the lag, the synth is reconstructed with the new lag:

out.lag(\xfreq, 0.01);
out.group.set(\trig, 0.1, \dt, 1, \xfreq, rrand(2000,9000));
out.group.set(\trig, 0.1, \dt, 1, \xfreq, rrand(2000,9000));
out.group.set(\trig, 0.1, \dt, 1, \xfreq, rrand(2000,9000));

// removing the trig rate:
out.setRates(\trig, nil);

// note that the same works with the i_ and the t_ arguments, just as it does in SynthDef
```

---

other possible inputs

using a synthdef as input

```
// for a more systematic overview see:  [jitlib_fading]

// you have the responsibility for the right number of channels and output rate
// you have to supply an 'out' argument so it can be mapped to the right channel.

out.play;
```

```
out = SynthDef("w", { arg out=0; Out.ar(out,SinOsc.ar([Rand(430, 600), 600], 0, 0.2)) });
out = SynthDef("w", { arg out=0; Out.ar(out,SinOsc.ar([Rand(430, 600), 500], 0, 0.2)) });


// if you supply a gate it fades in and out.  evaluate this several times
(
out = SynthDef("w", { arg out=0, gate=1.0;
Out.ar(out,
SinOsc.ar([Rand(430, 800), Rand(430, 800)], 0, 0.2) * EnvGen.kr(Env.asr(1,1,1), gate, doneAction:2)
)
});
)

// once the SynthDef is sent, it can be assigned by name.
// using this method, a gate argument should be
// provided that releases the synth.  (doneAction:2)
// this is very efficient, as the def is on the server already.

// if the synth def is in the synthdesc lib (.store) its gate is detected.

(
SynthDef("staub", { arg out, gate=1;
Out.ar(out, Ringz.ar(Dust.ar(15), Rand(1, 3) * 3000*[1,1], 0.001) * EnvGen.kr(Env.asr, gate, doneAction:2))

}).send(s);
)

out = \staub;



// if you supply an envelope that frees itself, no bundle is sent to free it
(
out = SynthDef("w", { arg out, lfo, f0=430;
Out.ar(out,
SinOsc.ar([Rand(f0, 800), Rand(f0, 800)]+lfo, 0, 0.2) * EnvGen.kr(Env.perc(0.01, 0.03), doneAction:2)

)
});
)
```

```
out.spawn;
out.spawn([\f0, 5000]);
fork { 5.do {  out.spawn([\f0, 5000 + 1000.0.rand]); 0.1.wait; } }


// when the synth description in the SynthDescLib is found for the symbol,
// the proxy can determine whether to release or to free the synth.
// so if there is no 'gate' arg provided and the def has a desc, the synth is
// freed and not released.

(
SynthDef("staub", { arg out;
Out.ar(out, Ringz.ar(WhiteNoise.ar(0.01), 1000*[1,1], 0.001))
          // store the synth def so it is added to the SynthDescLib
)


out = \staub;
       \staub  // watching the synth count shows that the old synth is freed.
            // now out plays continuous stream of zero.
       nil  // removes object and stops it.
```

using patterns

```
// example

(
SynthDef("who", { arg amp=0.1, freq=440, detune=0, gate=1, out=0, ffreq=800;
var env;
env = Env.asr(0.01, amp, 0.5);
Out.ar(out, Pan2.ar(
Formant.ar(freq + detune, ffreq, 30, EnvGen.kr(env, gate, doneAction:2)), Rand(-1.0, 1.0))
)
}).store;

)

out.play;
```

```
out = Pbind(\instrument, \who, \freq, [600, 601], \ffreq, 800, \legato, 0.02);



// embed a control node proxy into an event pattern:
// this does not work for indirect assignment as \degree, \midinote, etc.,
// because there is calculations in the event!  if needed, these can be done in the SynthDef.

lfo = { SinOsc.kr(2, 0, 400, 700) };
out = Pbind(\instrument, \who, \freq, 500, \ffreq,  lfo, \legato, 0.02);

lfo = { SinOsc.kr(SinOsc.kr(0.2, Rand(0,pi), 10, 10), 0, 400, 700) };

lfo = { LFNoise1.kr(5, 1300, 1500) };
lfo = { MouseX.kr(100, 5500, 1) };

(
out = Pbind(
   \instrument  \who
\freq, Pseq([500, 380, 300],inf),
\legato, 0.1,
\ffreq, Pseq([ lfo, 100,  lfo, 100, 300, 550], inf), // use it in a pattern
\dur, Pseq([1, 0.5, 0.75, 0.125]*0.4, inf)
);
)

// note that when you use a proxy within a non-event pattern it gets embedded as an object,
// so this functionality is still standard

// works only with control rate proxies.  multichannel control rate proxies cause
// multichannel expansion of the events:

lfoStereo = { LFNoise1.kr([1, 1], 1300, 1500) }; // 2 channel control rate proxy
out = Pbind(\instrument, \who, \freq, 1500, \detune,  lfoStereo, \legato, 0.02);
lfoStereo = { [MouseX.kr(100, 15500, 1), SinOsc.kr(SinOsc.kr(0.2, 0, 10, 10), 0, 400, 700)] }

// btw:  setting the clock will cause the pattern to sync:
p.clock = TempoClock.default;
```

```
p.clock.tempo = 2.0;
p.clock.tempo = 1.0


// patterns also crossfade, if an \amp arg is defined in the synthdef:
// (evaluate a couple of times)
out.fadeTime = 3.0;
(
out = Pbind(
   \instrument  \who
\freq, Pshuf([500, 380, 200, 510, 390, 300, 300],inf) * rrand(1.0, 2.0),
\legato, 0.1,
\ffreq, Pshuf([ lfo, 100,  lfo, 100, 300, 550], inf),
\dur, 0.125 * [1, 2, 3, 2/3].choose
);
)
```

using instruments and players

```
// note that you'll get late messages (which don't cause problems)
// pause and resume do not work yet.


// store an instrument
(
Instr \test
{ arg dens=520, ffreq=7000; Ringz.ar(Dust.ar(dens, [1,1]*0.1), ffreq, 0.02) }
);
)

out = Patch([\test], [10, rrand(5000, 8000)]);



(
out = InstrSpawner({ arg freq=1900,env,pan;
Pan2.ar(SinOsc.ar(freq, 0.5pi, 0.3) * EnvGen.kr(env, doneAction:  2), pan)
},[
Prand([1500, 700, 800, 3000] + 170.rand2, inf),
```

```
Env.perc(0.002,0.01),
Prand([-1,1],inf)
],0.125)
)

out.clear;



// does not work (yet).
// out.set(\dens, 120);
// out.xset(\dens, 1030);
// out.unmap(\ffreq);
// out.set(\ffreq, 500);
```

---

client side routines

spawning

```
out.play;
                    // allow sound object assignment without immediate sending

// putting an synthdef into the node proxy without playing it right away
// the synthdef has an envelope that frees by itself.
(
out = SynthDef("a", { arg out=0, freq=800, pmf=1.0, pan;
var env, u;
env = EnvGen.kr(Env.perc(0.001, 0.04, 0.4),doneAction:2); // envelope
u = SinOsc.ar(freq * Rand(0.9, 1.1), SinOsc.ar(pmf, 0, pi), env);
Out.ar(out, Pan2.ar(u, pan))
})
);


// create a task to repeatedly send grains
(
t = Task.new({
```

```
loop({
  // starts a synth with the current synthdef at index 0
out.spawn([\pmf, [1, 20, 300].choose, \pan, [0, -1, 1].choose]);
[0.1, 0.01, 0.25].choose.wait;
})
});
)

t.start;
t.stop;
t.start;

// note:  if you want to avoid using interpreter variables (single letter, like "t"),
// you can use Tdef for this.  (see Tdef.help)

// set some argument
out.set(\freq, 300);
out.set(\freq, 600);
out.map(\freq,  lfo);
lfo = { SinOsc.kr(0.1, 0, 3000, 4000) };
lfo = { SinOsc.kr(0.1, 0, 600, 700) };
lfo.add({ Trig.kr(Dust.kr(1), 0.1) * 3000 });
lfo = 300;

// change the definition while going along
(
out = SynthDef("a", { arg out, freq=800;
var env;
env = EnvGen.kr(Env.perc(0.01, 0.1, 0.3),doneAction:2);
Out.ar(out, Pulse.ar(freq * Rand([0.9,0.9], 1.1), 0.5, env) )
});
)



t.stop;
             true  // don't forget this
//  free all synths in this current ProxySpace
currentEnvironment.clear;
```

granular synthesis: efficient code

see also [jitlib_efficiency]

```
out.play;

(
SynthDef("grain", { arg i_out = 0, pan;
var env;
env = EnvGen.kr(Env.perc(0.001, 0.003, 0.2),doneAction:2);
Out.ar(i_out, Pan2.ar(FSinOsc.ar(Rand(1000,10000)), pan) * env)
}).send(s);
)

// a target for the grains
                    // initialize to 2 channels audio
out =  someInput;

(
t = Task({
loop({
s.sendMsg("/s_new","grain",-1,0,0,
   \i_out                    // returns the bus index of the proxy
\pan, [1, 1, -1].choose * 0.2
);
[0.01, 0.02].choose.wait;
})
});
)
t.play;

// different filters;

out.fadeTime = 1.0;

out = { BPF.ar( someInput.ar, MouseX.kr(100, 18000, 1), 0.1) };

out = { CombL.ar( someInput.ar * (LFNoise0.ar(2) > 0), 0.2, 0.2, MouseX.kr(0.1, 5, 1)) };
```

```
out = { RLPF.ar( someInput.ar, LFNoise1.kr(3, 1000, 1040), 0.05) };


t.stop;



//_________

out.stop;
currentEnvironment.clear;
ProxySpace      // restore original environment
```

---

using multiple proxyspaces

note that this can be done while the server is not running: with p.wakeUp or p.play the environment can be played back.

```
// quit server:

s.quit;


// create two proxyspaces without a running server
(
    ProxySpace
    ProxySpace

p.use({
out = { Resonz.ar( in.ar,  freq.kr, 0.01) };
in = { WhiteNoise.ar(0.5) };
freq = { LFNoise2.kr(1, 1000, 2000) };
});
```

```
q.use({

in = { Dust.ar(20, 0.1) };
out = { Resonz.ar( in.ar * 450,  freq.kr, 0.005) };
freq = { LFNoise2.kr(1, 400, 2000) };
});
)

// wait for the booted  server
s.boot;

// play the proxy at \out
p.play(\out);
        // out is the default output
```

external access

```
q[\in][1] = { Impulse.ar(2, 0, 0.5) }; // adding a synth at index 1

// equivalent to
q.at(\in).put(1, { Impulse.ar(7, 0, 0.5) });
```

connecting two spaces (must be on one server)

```
(
q.use({
freq =  100 + p[\freq] / 2;
})
)
```

recording output (see also: [RecNodeProxy])

```
r = p.record(\out, "proxySpace.aiff");

// start recording
```

```
r.unpause;

// pause recording
r.pause;

// stop recording
r.close;
```

push/pop

```
// make x the currentEnvironment
p.push;

freq = 700;
freq = 400;
freq = { p.kr(\freq) + LFNoise1.kr(1, 200, 300) % 400 }; // feedback
freq = 400;

	// restore environment


// make y the currentEnvironment
q.push;

freq = 1000;
in = { WhiteNoise.ar(0.01) };

	// restore environment

q.clear;
p.clear;
```

---

some more topics

nodeproxy with numbers as input:

```
p = ProxySpace.push(s.boot);


out = { SinOsc.ar( a.kr * Rand(1, 2), 0, 0.1) };
out.play;

a = 900;

// these add up:
a[0] = 440;
a[1] = 220;
a[2] = 20;

a.fadeTime = 2;

            // now there is a crossfade.
a[1] = { SinOsc.kr(5, 0, 20) };
a[2] = { SinOsc.kr(30, 0, 145) };



// internally a numerical input is approximately replaced by:
// (pseudocode)
SynthDef("name", { arg out, fadeTime;
Out.kr(out,
Control.kr(Array.fill(proxy.numChannels, { the number }))
* EnvGate.new(fadeTime:fadeTime)
)
});
```

ID: 273

Recursive phrases and granular composite sounds

Pdef can be used as a global storage for event patterns. Here a way is provided by which these
definitions can be used as an instrument that consists of several events (a *phrase*), such as a cloud of short grains. Furthermore, this scheme can be applied recursively, so that structures like a cloud of clouds can be constructed.

when the event type **phrase** is passed in, the event looks for a pattern in **Pdef.all** if it can find a definition
- if it finds one it plays this pattern in the context of the outer pattern's event.
If there is no definition to be found there, it uses a SynthDef with this name, if present.

When passing a *function* to Pdef it creates a PlazyEnvir internally.
Its function is evaluated in the context of the outer environment (see **[PlazyEnvir]**) which should return
a pattern or a stream.

```
(
s.boot;

SynthDef "pgrain"
{ arg out = 0, freq=800, sustain=0.001, amp=0.5, pan = 0;
var window;
window = Env.sine(sustain, amp);
Out.ar(out,
Pan2.ar(
SinOsc.ar(freq),
pan
) * EnvGen.ar(window, doneAction:2)
)
}
).store;

SynthDef "noiseGrain"
```

```
{ arg out = 0, freq=800, sustain=0.001, amp=0.5, pan = 0;
var window;
window = Env.perc(0.002, sustain, amp);
Out.ar(out,
Pan2.ar(
Ringz.ar(PinkNoise.ar(0.1), freq, 2.6),
pan
) * EnvGen.ar(window, doneAction:2)
)
}
).store;



Pdef(\sweep, { arg sustain=1, n=8, freq=440, ratio=0.1;
 Pbind
  \instrument  \pgrain
\dur, sustain.value / n,
  \freq  Pseq                                  // freq is a function, has to be evaluated
)
});
Pdef(\sweep2, { arg sustain=1, n=8, freq=440, ratio=0.1;
 Pbind
  \instrument  \noiseGrain
  \dur                       // sustain is also a function, has to be evaluated
\freq, Pseq((1..n).scramble) * ratio + 1 * freq.value,
  \recursionLevel
)
});
Pdef(\sweep3, { arg freq=440;
 Pbind
  \type  \phrase
  \instrument  \sweep
\freq, Pfunc({ rrand(0.8, 1.3)  })  * freq.value,
\dur, 0.3,
\legato, 1.3,
\n, 5
)
});
)
```

```
// the pattern that is found in Pdef.all (or your own defined library) is truncated in time
// using the sustain provided by the outer pattern.
(
Pbind
 \type  \phrase  // phrase event from global library
 \instrument  \sweep
\n, 15,
\degree, Pseq([0, 4, 6, 3], inf),
\sustain, Pseq([1.3, 0.2, 0.4],inf)
).play
)

// multichannel expansion is propagated into the subpatterns
(
Pbind
 \type  \phrase  // phrase event from global library
 \instrument  \sweep
\n, 15,
\degree, Pseq([0, 0, 6, 3], inf) + Prand([0, [0, 3], [0, 5], [0, 15]], inf),
\ratio, Prand([ 0.1, 0.1, [0.1, -0.1] ], inf)
).play
)

// various instruments and synthdefs can be used on the same level
(
Pbind
  \type  \phrase
  \instrument  Pseq  \sweep  \default  \sweep2  \sweep3  \pgrain  \pgrain   inf
\degree, Pseq([0, 3, 2], inf),
\dur, Pseq([1, 0.5], inf) * 0.7,
\n, Pseq([4, 6, 25, 10], inf),
\ratio, Prand([0.03, 0.1, 0.4, -0.1],inf) + Pseq([0, 0, [0, 0.02]], inf),
\legato, Pseq(#[0.5, 1, 0.5, 0.1, 0.1],inf)
).play;
)
```

```
// of course also a patten can be used directly in a Pdef

(
Pdef \sweep
 Pbind
  \instrument  Pseq  \pgrain  \noiseGrain  inf
\dur, Pseq([1, 2, 1, 3, 1, 4, 1, 5]) * 0.05,
\legato, Prand([0.5, 0.5, 3],inf)
)

)
)

// play directly, emebdded in stream (see Pdef.help)
Pn(Pdef(\sweep), 2).play;
Pdef \sweep        // play without changing player state (see [Pdef.help])


// play within a pattern
(
Pbind
 \type  \phrase
 \instrument  \sweep
\degree, Pseq([0, 1b, 4, 2, 3, 1b], inf),
\pan, Pfunc(#{ 1.0.rand2 })
).play
)



//////// recursion examples //////////

// the given pattern can be recursively applied to itself
// resulting in selfsimilar sound structures, like lindenmeyer systems (see also Prewrite)
// special care is taken so that no infinite loops can happen.
// just like with non recursive phrasing, new values override old values,
// any values that are not provided within the pattern definition
// are passed in from the outer event.
```

```
(
Pdef(\sweep, { arg dur=1, n=4, freq=440, ratio=0.3;
 Pbind
  \instrument  \pgrain
  \dur                 // now dur is dependant on outer dur, not on sustain
\freq, Pseq((1..n)) * ratio + 1 * freq.value
)
});
)


// no recursion
(
 Pbind
  \type  \phrase
  \instrument  \sweep
  \degree
).play;
)

// no recursion, with legato > 1.0 and varying notes
// note how subpatterns are truncated to note length
// provided by outer pattern (in this case determined by legato)
(
 Pbind
  \type  \phrase
  \instrument  \sweep
\degree, Pseq((0..5),inf),
\legato, Prand([1.2, 2.8, 0.3], inf)
).play;
)


// recursion over one level
(
 Pbind
  \type  \phrase
  \instrument  \sweep
```

```
  \degree
  \recursionLevel
).play
)

// recursion over one level:  legato is recursively applied
(
 Pbind
  \type  \phrase
  \instrument  \sweep
  \degree
\legato, Pseq([0.5, 1, 2, 4, 10], inf),
  \recursionLevel
).play
)


// to block this recursion, or modify it, assign legato explicitly:
(
Pdef(\sweep, { arg dur=1, n=4, ratio=0.5, freq=440;
var legato;
freq = freq.value;
legato = freq % 200 / 200 * 3 + 0.2;
 Pbind
  \instrument  \pgrain
\dur, dur.value / n,
\legato, legato,
\freq, Pseq((1..n) * ratio + 1 * freq)
)
});
)

// recursion over one level:  degree is assigned to each phrase,
// because freq is calculated internally and overrides degree on the second level
(
 Pbind
  \type  \phrase
  \instrument  \sweep
\degree, Pseq((0..10),inf),
  \recursionLevel
```

```
).play
)



// recursion over two levels
(
Pbind
 \type  \phrase
 \instrument  \sweep
 \degree
 \recursionLevel
).play
)

// recursion over three levels with variable number of grains
(
Pbind
 \type  \phrase
 \instrument  \sweep
 \degree
\n, Pseq([1, 2, 3],inf),
 \recursionLevel
).play
)


// "zoom" in
TempoClock.default.tempo = 0.2;
TempoClock.default.tempo = 1.0;


// recursion over variable levels
(
Pbind
 \type  \phrase
 \instrument  \sweep
\n, Prand([2, 7, 3], inf),
 \degree
\recursionLevel, Prand([0, 1, 2],inf)
```

```
).play
)



// replace the frequency based pattern with a degree based pattern
(
Pdef(\sweep, { arg sustain=1, n=8, degree=0, ratio=1;
 Pbind
  \instrument  \pgrain
\dur, sustain.value / n,
\degree, Pseq((1..n)) * ratio + 1 + degree.value
)
});
)


// drunken master
(
Pbind
 \type  \phrase
 \instrument  \sweep
\n, Prand([2, 4, 3, 8], inf),
\degree, Pseq([-5, 0, -2], inf),
\legato, Pseq([1.4, 0.5, 2], inf),
\scale, #[0, 2, 5, 7, 10],
\recursionLevel, Prand([0, 1, 2],inf)
).play
)



// pass in a synthDef from the outside
(
Pdef(\sweep, { arg sustain=1, n=8, degree=0, ratio=1;
 Pbind
\dur, sustain.value / n,
\degree, Pseq((1..n) * ratio + 1 + degree.value) )
});
```

```
)


(
Pbind
 \type  \phrase
 \instrument  \sweep
 \synthDef  Prand  \pgrain  \default  \noiseGrain  inf
\n, Prand([2, 4, 3, 8], inf),
\degree, Pseq([-5, 0, -2], inf),
 \recursionLevel  Prand        inf
).play
)


// use a different parent event in the inner pattern
(
e = Event.default;
e.use {  sustain = { 2.0.exprand(0.05) } };
Pdef(\sweep, { arg sustain=1, n=8, degree=0, ratio=1;
 Pbind
  \parent     // replace by some other event
  \instrument  \pgrain
\dur, sustain.value / n,
\degree, Pseq((1..n)) * ratio + 1 + degree.value
)
});
)


(
Pbind
 \type  \phrase
 \instrument  \sweep
\n, Prand([2, 4, 3, 8], inf),
\degree, Pseq([-5, 0, -2], inf),
 \recursionLevel  Prand        inf
).play
)
```

```
// pass in a pattern from outside

(
Pdef(\sweep, { arg sustain=1, n=8, degree=0, ratio=1;
n = n.value;
 Pbind
  \instrument  \pgrain
\dur, sustain.value / n,
\degree, Pseq([ 1, 2, 3, 4, 5 ] * ratio + 1 + degree.value) )
});
)



(
Pbind
 \type  \phrase
 \instrument  \sweep
 \n    Pshuf                       inf     //  use a function to insulate from embedInStream
\degree, Pseq([-5, 0, -2], inf),
 \recursionLevel  Prand        inf
).play
)




// recursion inside the pattern definition

(
Pdef(\sweep2, { arg sustain=1, n=2, degree=0, ratio=1;
 Pbind
  \type, \phrase,
  \instrument  \sweep
\dur, sustain.value / n,
\degree, Pseq((1..5).scramble * ratio + 1 + degree.value),
  \recursionLevel
)
});
)
```

```
(
Pbind
 \type, \phrase,
 \instrument  \sweep2
\n, 3,
\degree, Pseq([-5, 0, -2], inf)
).play
)


// instruments do not crossfade while they play (to make phrasing more efficient).


(
Pbind
 \type, \phrase,
 \instrument  \sweep
\n, 3,
\degree, Pseq([0, 2b, 3, 4], inf),
\dur, 2,
 \legato
).play
)

// change pattern definition while playing:
(
Pdef \sweep
 Pbind
  \instrument  \pgrain
\dur, exprand(0.01, 0.1),
\legato, rrand(0.01, 2.0),
\octave, rrand(5, 7)
)
)
)


// koch "snowflake"
(
```

```
Pdef(\koch, { arg dur=1, freq=440;
 Pbind
\dur, dur.value / 3,
\freq, freq.value * Pseq([1, 1.2, 1])
)
});
)

(
 Pbind
  \type, \phrase,
  \instrument  \koch
  \synthDef  \pgrain
\dur, 9,
  \recursionLevel
\legato, 1.1
).play
)

(
 Pbind
  \type, \phrase,
  \instrument  \koch
  \synthDef  \pgrain
\dur, 9,
  \recursionLevel
\legato, 1.1
).play
)

(
Pdef(\koch, { arg dur=1, degree=0;
 Pbind
\dur, dur.value / 3,
\degree, degree + Pseq([0, 2, 0])
)
});
)
```

```
// soundfile example


(
SynthDef(\play_from_to, { arg out, bufnum, from=0.0, to=1.0, sustain=1.0;
var env;
env = EnvGen.ar(Env.linen(0.01, sustain, 0.01), 1, doneAction:2);
Out.ar(out,
BufRd.ar(1, bufnum,
Line.ar(from, to, sustain) * BufFrames.kr(bufnum)
) * env
)


}).store;
)

          "/b_allocRead"       "sounds/a11wlk01.wav"



// this plays it straight
(
Pdef(\poch, { arg sustain=1.0, from=0.0, to=1.0, n=3;
var step;
sustain = sustain.value;
step = (to - from) / n;
  Pbind
   \instrument  \play_from_to
\from, Pseries(from, step, n),
\to, Pseries(from, step, n) + step,
\legato, 1.0,
\dur, sustain / n
)
})
)
```

```
(
Pbind
 \type, \phrase,
 \instrument  \poch
 \recursionLevel
\from, 0,
\to, 1,
\dur, 3,
 \bufnum
).play
)

// now turn round the middle part
(
Pdef(\poch, { arg sustain=1.0, from=0.0, to=1.0, n=3;
var step, f, t, i;
sustain = sustain.value;
step = (to - from) / n;
f = Array.series(n, from, step) +.t [0.0, step];
i = n div:  2;
f[i] = f[i].reverse;
  Pbind
   \instrument  \play_from_to
[\from, \to], Pseq(f),
\legato, 1.0,
\dur, sustain / n
)
})
)


// varying recursion
(
Pbind
 \type  \phrase
 \instrument  \poch
\recursionLevel, Prand([0, 1, 2, 3], inf),
\from, 0,
\to, Prand([-1, 1], inf),
\dur, 3,
```

```
\n, Prand([1, 2, 3], inf),
\bufnum, 170,
\amp, 0.2
).play
)
```

ID: 274

the lazy proxy

NodeProxy (and BusPlug) uses a lazy evaluation scheme to derive its appropriate rate and numChannels from the first meaningful input that is assigned to it.
see [NodeProxy] and [ProxySpace] helpfiles for basic info.
So as long as the source is not set, the proxy is **neutral**:

```
p = ProxySpace.push;
x.isNeutral;
```

as soon as the first time the source is set, it derives its bus arguments from that input

```
x = { Array.fill(14, { SinOsc.kr(1.0.rand, 0, 100) }) }; // x is now 14 channels control rate
```

in order to reset these properties, clear is used:

```
x.clear;
//note that no other proxy should be reading from  x when this is done:
//for simplicity nodeproxy currently does not care for its children, only for its parents.
```

for a quick initialisation, also defineBus can be used:

```
x.defineBus(\control, 5);
// or in another way:
x.kr(5)
```

the properties are also set when some other proxy reads from it:

```
	LFPulse			//the first arg to kr / ar is the default number of channels
```

if no number of channels is passed in, the default value is used:

```
test.ar; //2
krtest.kr; //1
```

the default can be set in the class NodeProxy:

```
NodeProxy.defaultNumAudio = 3;
NodeProxy.defaultNumControl = 13;

test3.ar; //3
krtest3.kr; // 13
```

also if a proxy is used as a map source, control rate is assumed:

```
u;
x.map(\zzz,  u);
u;
```

when unary or binary operations are performed, the highest rate / numChannels is used to initialize all uninitialized proxies:

```
x.clear;
x.defineBus(\control, 5);
x =  e +  f;

x.clear;  e.clear;  f.clear;
e.defineBus(\audio, 1);
x =  e +  f.squared +  r;

x.clear;  e.clear;  f.clear;
e.defineBus(\audio, 3);
x =  e;
```

if a rate1 proxy is used as rate2 input, the rate is converted
and the channels are expanded in the ususal multichannel expansion pattern:

```
f.defineBus(\control);
f.ar(2);

f.defineBus(\audio);
f.kr(2);

// if the number of channels passed in is less, it only uses the first n channels
f.defineBus(\audio, 8);
```

```
f.ar(2);
```

an offset can be passed in as second argument to ar/kr

```
//modulate offset:
p = ProxySpace.push(s.boot);

out.play;
src = { SinOsc.ar(Array.rand(5, 400, 500.0), SinOsc.ar(Array.exprand(5, 2.1, 500.0)), 0.1) };
out = {  src.ar(1, MouseX.kr(0, 5)) };
         Mix            MouseX              //the wrapping offset is moved accordingly
```

12.8 Ugens

ID: 275

Channel

output a fixed number of channels of a larger input array that can be moved across. In this way it is similar to the In ugen.

if wrap is set to true (default) the index wraps across the input array, otherwise it clips. Channel does not multi channel expand. For a channel mixer see **NumChannels**

***ar(array, offset, numChannels, wrap)**
***kr(array, offset, numChannels, wrap)**

```
(
{
var a;
a = Array.fill(8, { SinOsc.ar(Rand(500, 1800) * [1, 1.5], 0, 0.1) });
Channel.ar(a, MouseX.kr(0, 20), 2);

}.play;
)

(
{
var a;
a = Array.fill(8, { SinOsc.kr(Rand(0.1, 8) * [1, 2], 0, Rand(0, 80), Rand(300, 900)) });
SinOsc.ar(Channel.kr(a, MouseX.kr(0, 20), 2), 0, 0.1);

}.play;
)

// without wrapping
(
{
var a;
a = Array.fill(8, { SinOsc.ar(Rand(500, 1800) * [1, 1.5], 0, 0.1) });
Channel.ar(a, MouseX.kr(0, 20), 2, false);

}.play;
```

```
)

//  when the offset is fixed, Channel returns a fixed array accordingly.
// the othe ugens keep playing, so this makes sense only in certain cases.
// (similar to Select ugen)
(
{
var a, b;
a = Array.fill(8, { SinOsc.ar(Rand(300, 1800) * [1, 1.5] * LFNoise1.kr(0.01)) });
b = Channel.ar(a, 3, 2);
5.do({ b = b * Channel.ar(a, 8.rand, 2) });
b *  0.1

}.play;
)

(
{
var a, b, m;
a = Array.fill(8, { SinOsc.ar(Rand(300, 1800) * [1, 1.5] * LFNoise1.kr(0.01)) });
b = Channel.ar(a, 3, 2);
m = MouseX.kr(0, a.size);
5.do({ b = b * Channel.ar(a, 8.rand + m, 2) });
b * 0.1

}.play;
)
```

Note: all the input ugens are continously running. This may not be the most efficient way if each input is cpu-expensive.

ID: 276

TWChoose

The output is selected randomly on recieving a trigger from an array of inputs.
the weights of this choice are determined from the weights array.
If normalize is set to 1 the weights are continuously normalized, which means an extra calculation overhead.
when using fixed values the **normalizeSum** method can be used to normalize the values
TWChoose is a composite of TWindex and Select

***ar(trig,array,weights,normalize)**
***kr(trig,array,weights,normalize)**

```
(
{
var a;
a = [
SinOsc.ar,
Saw.ar,
Pulse.ar
];
TWChoose.ar(Dust.ar(MouseX.kr(1, 1000, 1)), a, [0.99, 0.05, 0.05].normalizeSum) * 0.2

}.play;
)
```

Note: all the ugens are continously running. This may not be the most efficient way if each input is cpu-expensive.

13 Language

ID: 277

Assignment Statements

Single Assignment

A single assignment assigns the value of an expression on the right hand side to a variable on the left hand side. A single assignment is in the form:

<variable> = <an expression>

examples:

```
x = [1, 2, 3, 4].rotate(1);
c = a + b;
```

Multiple Assignment

A multiple assignment statement assigns the elements of a Collection which is the result of an expression on the right hand side, to a list of variables on the left hand side. A multiple assignment statement is preceeded by the symbol #. If the last variable on the left is preceeded by three dots, then the entire remainder of the collection is assigned to that variable. There must be at least one variable name before the ellipsis.

The form of a multiple assignment is:

<list of variables> = <expression>
– or –
<list of variables> ... <variable> = <expression>

examples:

```
# a, b, c = [1, 2, 3, 4, 5, 6]; // afterwards a=1, b=2, c=3

                                 // afterwards a=1, b=2, c = [3, 4, 5, 6]

                               // ILLEGAL, just use:      a = [1, 2, 3, 4, 5, 6];
```

Multiple assignment is implemented using the 'at' method and the 'copyToEnd' method. Your right hand side expression can return any object that responds to these messages.

Instance Variable Assignment

The basic syntax for setting the value of an instance variable is to use the variable's setter method which is the name of the variable with an underscore appended.

```
            // set point's x coordinate to 5
```

An alternative syntax is to use instance variable assignment.

```
point.x = 5;
```

This type of assignment is translated to the first form by the compiler. The two syntaxes are equivalent.

Series Assignment to an ArrayedCollection or List

There is a special syntax for doing assignments to a range of values in an ArrayedCollection or List.

```
a = (0,10..200);
               // series stepping by 1

a = (0,10..200);
                // a series by any step size

a = (0,10..200);
             // from zero to n

a = (0,10..200);
               // from n to the end of the array

a = (0,10..200);
                // a series to the end of the array
```

ID: 278

Classes

All objects in SuperCollider are members of a class that describes the objects' data and interface. The name of a class must begin with a capital letter. Class names are the only global values in the SC language. Since classes are themselves objects, the value of a class name identifier is the object representing that class.

Instance Variables

The data of an object is contained in its instance variables. Instance variables are of two kinds, named and indexed. Each object contains a separate copy of its instance variables.

Some classes' instances have no instance variables at all but instead have an atomic value. Classes whose instances consist of an atomic value are Integer, Float, Symbol, True, False, Nil, Infinitum, Char, Color.

Instance variables declaration lists occur after the open curly bracket of the class definition and are preceeded by the reserved word **var**. Instance variables names in the list may be initialized to a default literal value by using an equals sign. Instance variables which are not explicitly initialized will be set to nil.

Instance variables may only be directly accessed and changed from within the class' own methods. Instance variables may be exposed to clients by adding getter and setter messages to the class. A **getter** message is a message with the same name as the variable which returns the value of that variable when sent to the receiver. A **setter** message is a message with the name of the variable with an underscore appended which sets the value of the instance variable to the value of the argument to the message. Getter and setter methods may be defined in the declaration of the instance variable. A getter message for an instance variable is created by typing a less than sign $<$ before the variable name. A setter message is created by typing a greater than $>$ sign before the variable name. If both occur then they must occur in the order $<>$.

```
var a, <b, >c, <>d;
```

a has no getter or setter.
b has a getter but not a setter.
c has only a setter.
d has both a getter and setter.

```
Point
 // x and y are instance variables which have both getter and setter methods
 // and default to zero.
var <>x = 0, <>y = 0;
...
}

p = Point.new;
          // send setter message to set x to 5
          // send setter message to set y to 7
          // send setter message using setter assignment (See [03 Assignment])
          // send setter message using setter assignment (See [03 Assignment])
          // send getter message for x
          // send getter message for y
```

Class Variables

Class variables are values that are shared by all objects in the class. Class variable declaration lists are preceeded by the reserved word **classvar** and may be interspersed with instance variable declaration lists. Class variables like instance variables may only be directly accessed by methods of the class. Class variables may also have getter and setter methods created for them using the less than $<$ and greater than $>$ symbols.

Instance Methods

The messages of a class' interface are implemented in the methods of the class. When an object is sent a message the method whose name matches the message selector in the receiver's class is executed.

Method definitions follow the class variable and instance variable declarations in the class.

Method definitions are similar to FunctionDefs in syntax. Method definitions begin with the message selector. The message selector must be an identifier or a binary operator. Methods have arguments and variable declarations with the same syntax as in FunctionDefs. Methods however have an implicit first argument named **this** which is the

receiver of the message. The variable 'this' may be referred to just like any of the other arguments and variables in the method. You may not assign a value to 'this'.

Class Methods

Class Methods are methods that implement messages sent to the class object. A common example is the message **new** which is sent to the class object to create a new instance of the class. Class method names have an asterisk preceeding the message selector.

ID: 279

Comments

Comments begin with `//` and go until the end of the line. Comments can also be delimited with `/*` and `*/`.
examples:

```
// single line comment


/*
multi
line
comment
*/

/* Unlike C, you can have /* nested */ comments */
```

ID: 280

Control Structures

Control structures in SuperCollider are implemented via message sends. Here are a few of those available.
See [Syntax-Shortcuts] for the various ways expressions can be written.

If

Conditional execution is implemented via the **if** message. The **if** message is sent to an expression which must return a Boolean value. In addition it takes two arguments: a function to execute if the expression is true and another optional function to execute if the expression is false. The **if** message returns the value of the function which is executed. If the falseFunc is not present and the expression is false then the result of the **if** message is nil.

syntax:

if (expr, trueFunc, falseFunc);
..or..
expr.if (trueFunc, falseFunc);

see also: the**[if] helpfile**

examples:

```
    false  true             // Boolean expression (chooses one at random)
 { "expression was true"          // true function
 { "expression was false"         // false function
)


(
var a = 1, z;
z = if (a < 5, { 100 },{ 200 });
z.postln;
)
```

```
(
var x;
if (x.isNil, { x = 99 });
x.postln;
)
```

›If' expressions are optimized (i.e. inlined) by the compiler if they do not contain variable declarations in the trueFunc and the falseFunc.

While

The **while** message implements conditional execution of a loop. If the testFunc answers true when evaluated, then the bodyFunc is evaluated and the process is repeated. Once the testFunc returns false, the loop terminates.

syntax:

while (testFunc, bodyFunc);
..or..
testFunc.while(bodyFunc);

example:

```
(
i = 0;
while ( { i < 5 }, { i = i + 1; "boing".postln });
)
```

›While' expressions are optimized by the compiler if they do not contain variable declarations in the testFunc and the bodyFunc.

For

The **for** message implements iteration over an integer series from a starting value to an end value stepping by one each time. A function is evaluated each iteration and is passed the iterated numeric value as an argument.

syntax:

for (startValue, endValue, function)
..or..
startValue.for (endValue, function)

example:

```
for (3, 7, { arg i; i.postln }); // prints values 3 through 7
```

ForBy

The **forBy** selector implements iteration over an integer series with a variable step size. A function is evaluated each iteration and is passed the iterated numeric value as an argument.

syntax:

forBy (startValue, endValue, stepValue, function);
..or..
startValue.forBy (endValue, stepValue, function);

example:

```
                arg             // prints values 0 through 8 by 2's
```

Do

Do is used to iterate over a collection. Positive Integers also respond to 'do' by iterating

from zero up to their value. Collections iterate, calling the function for each object they contain. Other kinds of Objects respond to do by passing themselves to the function one time. The function is called with two arguments, the item, and an iteration counter.

syntax:

do (collection, function)
..or..
collection.do(function)

example:

```
[ 1, 2, "abc", (3@4) ].do({ arg item, i; [i, item].postln; });

5.do({ arg item; item.postln }); // iterates from zero to four

"you"      arg                      // a String is a collection of characters

'they'      arg                      // a Symbol is a singular item

(8..20).do({ arg item; item.postln }); // iterates from eight to twenty

               arg                      // iterates from eight to twenty, with stepsize two

Routine({ var i=10; while { i > 0 } { i.yield; i = i - 5.0.rand } }).do({ arg item; item.postln });
```

Switch

Object implements a **switch** method which allows for conditional evaluation with multiple cases. These are implemented as pairs of test objects (tested using if this == test.value) and corresponding functions to be evaluated if true. The switch statement will be inlined if the test objects are all Floats, Integers, Symbols, Chars, nil, false, true and if the functions have no variable or argument declarations. The inlined switch uses a hash lookup (which is faster than nested if statements), so it should be very fast and scale to any number of clauses.

```
(
var x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
switch (z.choose.postln,
1,   { \no },
1.1, { \wrong },
1.3, { \wrong },
1.5, { \wrong },
2,   { \wrong },
0,   { \true }
).postln;
)
```

or:

```
(
var x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
x = switch (z.choose)
{1}   { \no }
{1.1} { \wrong }
{1.3} { \wrong }
{1.5} { \wrong }
{2}   { \wrong }
{0}   { \true };
x.postln;
)
```

Case

Function implements a **case** method which allows for conditional evaluation with multiple cases. Since the receiver represents the first case this can be simply written as pairs of test functions and corresponding functions to be evaluated if true. Case is inlined and is therefore just as efficient as nested if statements.

```
(
var i, x, z;
z = [0, 1, 1.1, 1.3, 1.5, 2];
i = z.choose;
x = case
```

```
{ i == 1 }   { \no }
{ i == 1.1 } { \wrong }
{ i == 1.3 } { \wrong }
{ i == 1.5 } { \wrong }
{ i == 2 }   { \wrong }
{ i == 0 }   { \true };
x.postln;
)
```

Other Control Structures

Using Functions, many control structures can be defined like the ones above. In the class **Collection** there are many more messages defined for iterating over Collections.

ID: 281

Debugging tips

Debugging synthdefs
Debugging client-to-server communication
Debugging client code

Debugging synthdefs

The challenge in debugging synthdefs is the invisibility of the server's operations. There are a handful of techniques to expose the output of various UGens.

SendTrig / OSCresponderNode

SendTrig is originally intended to send a trigger message back to the client, so the client can take further action on the server. However, it can be used to send any numeric value back to the client, which can then be printed out.

To print out the values, you need to create an OSCresponderNode as follows:

```
o = OSCresponderNode(myServer.addr, '/tr', { | time, resp, msg|  msg.postln }).add;
```

Each line of output is an array with four values: ['/tr', defNode, id (from SendTrig), value (from SendTrig)].

```
{ var freq;
freq = LFNoise1.kr(2, 600, 800);
  // Impulse is needed to trigger the /tr message to be sent
SendTrig.kr(Impulse.kr(4), 0, freq);
SinOsc.ar(freq, 0, 0.3) !  2
}.play;
[ /tr, 1000, 0, 1340.8098144531 ]
[ /tr, 1000, 0, 1153.9201660156 ]
[ /tr, 1000, 0, 966.35247802734 ]
[ /tr, 1000, 0, 629.31628417969 ]

          // when done, you need to clean up the OSCresponderNode
```

If you need to track multiple values, you can store them in a collection of arrays and

differentiate them by assigning different IDs in the SendTrig UGen.

```
l = { List.new } ! 2;
    OSCresponderNode                '/tr'    | time, resp, msg|
  // msg[2] is the index
l[msg[2]].add(msg[3]);
}).add;

{ var freq, amp;
freq = LFNoise0.kr(8, 600, 800);
amp = LFNoise1.kr(10, 0.5, 0.5);
  // Impulse is needed to trigger the /tr message to be sent
SendTrig.kr(Impulse.kr(4), 0, freq);
SendTrig.kr(Impulse.kr(4), 1, amp);
SinOsc.ar(freq, 0, 0.3) ! 2
}.play;

          // when done, you need to clean up the OSCresponderNode

               // view frequencies
l[1].array.plot // view amps
```

Shared controls (Internal server only, control rate only)

The internal server allocates a number of control buses whose memory addresses are shared with the client. The client can poll these buses without using OSC messages.

Insert a SharedOut.kr UGen into your synthdef. Then, on the client side, use s.getSharedControl(num) to read the value. If you want to track the value over time, use a routine to poll repeatedly.

```
{ var freq;
freq = LFNoise1.kr(2, 600, 800);
 SharedOut              // no need for Impulse here
SinOsc.ar(freq, 0, 0.3) ! 2
}.play;

l = List.new;
r = fork { loop { l.add(s.getSharedControl(0)); 0.1.wait } };
r.stop;
```

```
// to view the results graphically
```

Server-side trace

The /n_trace message causes the server to print a list of all the UGens in the node as well as their input and output values.

It takes some practice to read a synthdef trace, but it's the ultimate source of information when a synthdef is not behaving as expected. Signal flow can be identified by looking at the numbers at inputs and outputs. When a UGen's output feeds into another's input, the values will be the same at both ends.

For a concrete example, let's look at a synthdef that doesn't work. The intent is to generate a detuned sawtooth wave and run it through a set of parallel resonant filters whose cut-off frequencies are modulating randomly. We run the synth and generate the trace (reproduced below). The trace comes out in monochrome; colors are used here to highlight signal flow.

```
SynthDef \resonz    | freq = 440|
var sig, ffreq;
sig = Saw.ar([freq, freq+1], 0.2);
ffreq = LFNoise1.kr(2, 1, 0.5);
Out.ar(0, Resonz.ar(sig, (800, 1000..1800) * ffreq, 0.1))
}).send(s);

a = Synth(\resonz);
a.trace;
a.free;
```

```
TRACE 1005  resonz    #units: 21
unit 0 Control
in
out 440
unit 1 BinaryOpUGen
in  440 1
out 441
unit 2 Saw
in  441
out 0.451348
unit 3 BinaryOpUGen
```

in 0.451348 0.2
out 0.0902696
unit 4 Saw
in 440
out -0.367307
unit 5 BinaryOpUGen
in -0.367307 0.2
out -0.0734615
unit 6 LFNoise1
in 2
out -0.836168
unit 7 BinaryOpUGen
in -0.836168 0.5
out #ff0000-*0.336168*
unit 8 BinaryOpUGen
in 800 #ff0000-*0.336168*
out #ff0000-*268.934*
unit 9 Resonz
in -0.0734615 #ff0000-*268.934* 0.1
out **843934**
unit 10 BinaryOpUGen
in 1000 -0.336168
out -336.168
unit 11 Resonz
in 0.0902696 *-336.168* 0.1
#0000ff out **3.02999e+08**
unit 12 BinaryOpUGen
in 1200 -0.336168
out -403.402
unit 13 Resonz
in -0.0734615 *-403.402* 0.1
#996633 out **9.14995e+10**
unit 14 BinaryOpUGen
in 1400 -0.336168
out -470.635
unit 15 Resonz
in 0.0902696 *-470.635* 0.1
out #00ff00-**5.42883**
unit 16 BinaryOpUGen
in 1600 -0.336168

out -537.869
unit 17 Resonz
in -0.0734615 *-537.869* 0.1
out #ff00ff**515.506**
unit 18 BinaryOpUGen
in 1800 -0.336168
out -605.102
unit 19 Resonz
in 0.0902696 *-605.102* 0.1
out #ff8000**32785.2**
unit 20 Out
in 0 **843934** #0000ff**3.02999e+08**#996633**9.14995e+10**#00ff00-**5.42883**#ff00ff**515.506**#ff8000**3278**

out

Two problems leap out from the trace: first, there are six channels of the output (there should be 1), and second, all the outputs are well outside the audio range -1..1. The first is because we use multichannel expansion to produce an array of Resonz filters, but we don't mix them down into a single channel.

The above trace uses colors to track the source of each output signal. Note that there are no out of range signals prior to each Resonz. Looking at the Resonz inputs, we see that the frequency input is negative, which will blow up most digital filters.

The resonance frequency derives from multiplying an array by a LFNoise1. Tracing back (the red, italicized numbers), the LFNoise1 is outputting a negative number, where we expected it to be 0.5..1.5. But, the mul and add inputs are reversed!

If you look very carefully at the trace, you will see another problem relating to multichannel expansion. The two components of the detuned sawtooth go into alternate Resonz'es, where we expected both to go, combined, into every Resonz. To fix it, the sawtooths need to be mixed as well.

```
SynthDef \resonz    | freq = 440|
var sig, ffreq;
sig = Mix.ar(Saw.ar([freq, freq+1], 0.2));
ffreq = LFNoise1.kr(2, 0.5, 1);
Out.ar(0, Mix.ar(Resonz.ar(sig, (800, 1000..1800) * ffreq, 0.1)))
}).send(s);
```

```
a = Synth(\resonz);
a.trace;
a.free;
```

Debugging client-to-server communication

Some bugs result from OSC messages to the server being constructed incorrectly. Julian Rohrhuber's DebugNetAddr is a convenient way to capture messages. The class may be downloaded from:

http://swiki.hfbk-hamburg.de:8888/MusicTechnology/710

To use it, you need to quit the currently running local server, then create a new server using a DebugNetAddr instead of a regular NetAddr. Messages will be dumped into a new document window.

```
s.quit;

Server.default = s = Server.new('local-debug', DebugNetAddr("localhost", 57110));
s.boot;
s.makeWindow;  // optional

latency nil  // these messages get sent on bootup
[ "/notify", 1 ]

latency nil
[ "/g_new", 1 ]

a = { SinOsc.ar(440, 0, 0.4) !  2 }.play;

latency nil
[ "/d_recv", "data[ 290 ]", [ 9, "-1589009783", 1001, 0, 1, 'i_out', 0, 'out', 0 ] ]

a.free;

latency nil
[ 11, 1001 ]
```

Debugging client code

SuperCollider does not have a step trace function, which makes debugging on the client side tougher, but not impossible.

Errors

Learning how to read SuperCollider error output is absolutely essential. Error dumps often (though not always) contain a great deal of information: what the action was, which objects are being acted upon, and how the flow of execution reached that point.

See the [Understanding-Errors] help file for a tutorial.

There's also a graphic Inspector for error dumps, which is enabled with the following command:

```
Exception        true   // enable
Exception        false  // disable
```

In most cases, this will give you more information than a regular error dump. Usually the regular error dump is sufficient. If you are using Environments or prototype-style programming, the graphic inspector is indispensable.

Debug output using post statements

The most common approach is to insert statements to print the values of variables and expressions. Since the normal printing methods don't change the value of an expression, they can be placed in the middle of the statement without altering the processing flow. There's no significant difference between:

```
if(a > 0) { positive.value(a) };
```

and

```
if((a > 0).postln) { positive.value(a) };
```

Common methods to use are:

.postln
// post the object as a compile string
// post the object along with a tag identifying the caller

.debug is defined in the crucial library, so Linux and Windows users may not have access to it. It's used like this:

```
(
var positiveFunc;
positiveFunc = { | a|
 a.debug('positiveFunc-arg a'
a*10
};
a = 5;
if (a > 0) { positiveFunc.value(a) };
)

// output:
positiveFunc-arg a:  5
50
```

The caller argument is optional; however, it's very helpful for tracing the origin of erroneous values.

Another advantage of .debug is that it's easier to search for debug calls and differentiate them from legitimate postln and postcs calls.

To print multiple values at one time, wrap them in an array before using .debug or .postcs. Note that if any of the array members are collections, postln will hide them behind the class name: "an Array, a Dictionary" etc. Use postcs if you expect to be posting collections.

```
                        \myMethod   // or, for a non-Crucial way:
[\callerTag, val1, val2, val3].postcs;
```

By sprinkling these throughout your code, especially at the beginnings of functions or methods, the debugging output can give you a partial trace of which code blocks get visited in what order.

dumpBackTrace

If you discover that a particular method or function is being entered but you don't know how it got there, you can use the .dumpBackTrace method on any object. You'll get

what looks like an error dump, but without the error. Execution continues normally after the stack dump.

```
(
var positiveFunc;
positiveFunc = { | a|
 a.debug('positiveFunc-arg a'
a.dumpBackTrace;
a*10
};
a = 5;
if (a > 0) { positiveFunc.value(a) };
)

// output:
positiveFunc-arg a:  5
CALL STACK:
< FunctionDef in closed FunctionDef >
arg a = 5
< closed FunctionDef >
var positiveFunc = <instance of Function>
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
Process-interpretPrintCmdLine
arg this = <instance of Main>
50
```

This tells you that the function came from interpreting a closed FunctionDef (automatically created when evaluating a block of code).

In a method definition, it's recommended to use "this.dumpBackTrace"; in a free-standing function, there is no "this" so you should pick some arbitrary object.

Tracing streams

To see the results of a pattern, use the .trace method. Each output value from the pattern gets posted to the main output.

```
s.boot;
SynthDescLib.global.read;

p = Pbind(\degree, Pwalk((0..14), Pstutter(Pwhite(1, 4, inf), Prand(#[-2, -1, 1, 2], inf)), Pseq(#[-1,
1], inf), 0), \delta, 0.25, \sustain, 0.2, \instrument, \default).trace.play;

p.stop;
```

Debugging infinite loops or recursion

```
while(true);
```

This is a bad idea. It will lock up SuperCollider and you will have to force quit. Sometimes this happens in your code and the reason isn't obvious. Debugging these situations is very painful because you might have to force quit, relaunch SuperCollider, and reload your code just to try again.

```
f = { | func|  func.value(func) };
f.value(f);
```

Infinite recursion, on the other hand, is more likely to cause SuperCollider to quit unexpectedly when the execution stack runs out of space.

In Mac OS X, inserting "post" or "debug" calls will not help with infinite loops or recursion, because posted output is held in a buffer until execution is complete. If execution never completes, you never see the output.

One useful approach is to insert statements that will cause execution to halt. The easiest is .halt, but it provides you with no information about where or how it stopped, or how it got there. If you want a more descriptive message, make up an error and throw it:

```
Error "myFunction-halt"
```

When debugging code that crashes, place a line like this somewhere in the code. If you get the error output, you know that the infinite loop is happening after the error–so move the error.throw later and try again. If it crashes, you know the infinite loop is earlier. Eventually, after a lot of heartache, you can zero in on the location.

Here is a rogues' gallery of infinite loop gotchas–things that don't look like infinite loops, but they will kill your code quicker than you can wish you hadn't just pushed the enter

key:

```
i = 0;
while (i < 10) { i.postln; i = i+1 };  // crash
```

While loop syntax is different in SuperCollider from C. The above loop means to check whether i $<$ 10 once, at the beginning of the loop, then loop if the value is true. Since the loop condition is evaluated only once, it never changes, so the loop never stops. The loop condition should be written inside a function, to wit:

```
i = 0;
while { i < 10 } { i.postln; i = i+1 };
```

Routines and empty arrays:

```
a = Array.new;
    Routine
loop {
a.do({ | item|  item.yield });
}
});
r.next; // crash
```

This looks pretty innocent: iterate repeatedly over an array and yield each item successively. But, if the array is empty, the do loop never executes and yield never gets called. So, the outer loop{} runs forever, doing nothing.

Recursion is often used to walk through a tree structure. Tree structures are usually finite–no matter which branch you go down, eventually you will reach the end. If you have a data structure that is self-referential, you can easily get infinite recursion:

```
a = (1..10);
             // now one of the items of a is a itself
          // crash--postcs has to walk through the entire collection, which loops on itself
```

Self-referential data structures are sometimes an indication of poor design. If this is the case, avoid them.

```
a = 0;
SystemClock                         // crashes when scheduler fires the function
```

When a scheduled function executes, if it returns a number, the function will be rescheduled for now + the number. If the number is 0, it is effectively the same as an infinite loop.

To fix it, make sure the function returns a non-number.

```
a = 0;
SystemClock.sched(2, { a.postln; nil });
```

Removing debugging statements

Use formatting to help your eye locate debugging statements when it's time to remove them. SuperCollider code is usually indented. If you write your debugging statements fully left-justified, they're much easier to see.

```
a = Array.new;
    Routine
loop {
  "debugging"         // looks like regular code, doesn't stand out
a.do({ | item|  item.yield });
}
});
r.next; // crash

// vs:

a = Array.new;
    Routine
loop {
"debugging"         // this obviously sticks out
a.do({ | item|  item.yield });
}
});
r.next; // crash
```

ID: 282

Expression Sequence

A sequence of expressions separated by semicolons and optionally terminated by a semicolon are a single expression whose value is the value of the last expression. Such a sequence may be used anywhere that a normal expression may be used.

```
// computes the maximum of b+5 and 10
```

In the above example, the sequence: `b = a * 2; b + 5` acts as a single expression for the first argument to`max()`.

ID: 283

Functions

A [Function] is an expression which defines operations to be performed when it is sent the 'value' message. In functional languages, a function would be known as a lambda expression. Function definitions are enclosed in curly brackets {}. Argument declarations, if any, follow the open bracket. Variable declarations follow argument declarations. An expression follows the declarations.

```
{ arg a, b, c;  var d;   d = a * b; c + d }
```

Functions are not evaluated immediately when they occur in code, but are passed as values just like integers or strings.

A function may be evaluated by passing it the value message and a list of arguments.

When evaluated, the function returns the value of its expression.

```
f = { arg a, b; a + b };
f.value(4, 5).postln;
f.value(10, 200).postln;
```

An empty function returns the value nil when evaluated.

```
{}.value.postln;
```

Arguments

An argument list immediately follows the open curly bracket of a function definition. An argument list either begins with the reserved word **arg**, or is contained between two vertical bars. If a function takes no arguments, then the argument list may be omitted.

Names of arguments in the list may be initialized to a default value by using an equals sign. Arguments which are not explicitly initialized will be set to nil if no value is passed for them.

If the last argument in the list is preceeded by three dots (an ellipsis), then all the remaining arguments that were passed will be assigned to that variable as an Array. Arguments must be separated by commas.

examples:

```
arg           // is equivalent to:

| a, b, c=3|

arg  'stop'          // these args are initialised

arg               // any arguments after the first 3 will be assigned to d as an Array
```

If you want all the arguments put in an Array

```
arg ...  z;
```

In general arguments may be initialized to literals or expressions, but in the case of Function-play or SynthDef-play, they may only be initialized to literals.

```
// this is okay:

{arg a = Array.geom(4, 100, 3); a * 4 }.value;

// this is not:

{arg freq = Array.geom(4, 100, 3); Mix(SinOsc.ar(freq, 0, 0.1)) }.play; // silence

// but this is:
(
SynthDef(\freqs, { arg freq = #[ 100, 300, 900, 2700 ];
Out.ar(0, Mix(SinOsc.ar(freq, 0, 0.1)));
}).play;
)
```

See [Literals] for more information.

Variables

Following the argument declarations are the variable declarations. These may be declared in any order. Variable lists are preceeded by the reserved word **var**. There can be

multiple var declaration lists if necessary. Variables may be initialized to default values in the same way as arguments. Variable declarations lists may not contain an ellipsis.

examples:

```
var level=0, slope=1, curve=1;
```

See also [Function], [AbstractFunction], and [FunctionDef].

ID: 284

Objects, Messages

The SuperCollider language is an object oriented language. All entities in the language are objects. An **object** is something that has data, representing the object's state, and a set of operations that can be performed on the object. All objects are **instances** of some **class** which describes the structure of the object and its operations. Objects in SuperCollider include numbers, character strings, object collections, unit generators, wave samples, points, rectangles, graphical windows, graphical buttons, sliders and much more.

Operations upon objects are invoked by messages. A **message** is a request for an object, called the **receiver**, to perform one of its operations. The means by which the operation is performed is determined
by the object's class. Objects of different classes may implement the same message in different ways, each appropriate to the class of the object. For example all objects understand the 'value' message. Many objects simply return themselves in response to 'value', but other objects such as functions and streams first evaluate themselves and return the result of that evaluation. The ability for different objects to react differently to the same message is known as **polymorphism** and is perhaps the most important concept in object oriented programming since it allows the object's behaviour to be abstract from the point of view of the user of the object (the client).

The set of messages to which an object responds to is known as its **interface**. A set of messages that
implement a specific kind behaviour is known as a **protocol**. An object's interface may include several protocols which allow the object to interact in several different contexts. For example all objects implement the 'dependancy' protocol which allow the object to notify other dependant objects that the object has changed and that the dependant should do any necessary action to update itself.

An object's internal state may only be changed by sending it messages. This allows the implementation of the object to be hidden from the client. The advantage to this is that the client does not depend on the object's implementation and that that implementation can be changed without having to change the client.

Classes, Instance Variables, Methods

An object's **class** contains the description of the object's data and operations. A

class also describes
how to create an object which is a instance of that class.

An object's data is contained in its **instance variables**. These are named variables that describe
the object's state. The values of the instance variables are themselves objects. For example, instances of class Point have instance variables named 'x' and 'y' which contain the coordinate values of the Point.

An instance variable is only directly accessible from within the class itself. The author of a class may decide to expose instance variable access to clients by adding **getter** and/or **setter** messages to the class.

A **method** is a description of the operations necessary to implement a message for a particular class. The methods in a class tell how to implement messages sent to its instances. A class contains a method definition for each message to which its instances respond. Methods generally fall into several categories. Some methods inquire about some property of the receiver. Others ask the receiver to make some change to its internal state. Still others may ask the receiver to return some computed value.

Summary of Terminology

object something that has data, representing the object's state, and a set of operations that can be performed on the object.

message a request for an object to perform an operation.

receiver the object to which a message is sent.

class a description of the state and behaviour of a set of objects.

interface the set of messages to which an object responds.

protocol a set of messages that implement a specific kind of behaviour.

polymorphism the ability for different kinds of objects to respond differently to the same message.

method a description of the operations necessary to implement a message for a particular class.

instance one of the objects described by a class.

instance variable a part of an object's internal state

ID: 285

List Comprehensions

List comprehensions are a syntactic feature of functional programming languages like Miranda, Haskell, and Erlang which were later copied into Python.
You can search the web for "list comprehensions" or "generator expressions" to learn more.
Basically list comprehensions are for getting a series of solutions to a problem.

in SC these are just a syntax macro for a longer expression.

```
// read this as "all [x,y] for x in 1..5, y in 1..x, such that x+y is prime.
all {:[x,y], x <- (1..5), y <- (1..x), (x+y).isPrime }
```

```
[ [ 1, 1 ], [ 2, 1 ], [ 3, 2 ], [ 4, 1 ], [ 4, 3 ], [ 5, 2 ] ]
```

the list comprehension above is equivalent to the following code:

```
all(Routine.new({ (1..5).do {| x|  (1..x).do {| y|  if ((x+y).isPrime) {[x,y].yield} }}}));
```

..but much more concise and much easier to keep in your head than writing it out.

In the list comprehension compiler, simple series like (1..5) and (1..x) are treated as special cases and implemented as loops rather than making a collection.

A list comprehension in SC is really a Routine. You can use the 'all' message to collect all of the Routine's results into a list.

A few examples

```
all {:  x/(x+1), x <- (1..5) }
```

```
[ 0.5, 0.66666666666667, 0.75, 0.8, 0.83333333333333 ]
```

```
all {:[x,y], x <- (1..3), y <- [\a,\b,\c] }
```

```
[ [ 1, a ], [ 1, b ], [ 1, c ], [ 2, a ], [ 2, b ], [ 2, c ], [ 3, a ], [ 3, b ], [ 3, c ] ]
```

```
all {:[x,y], x <- (0..3), y <- (x..0) }
```

```
[ [ 0, 0 ], [ 1, 1 ], [ 1, 0 ], [ 2, 2 ], [ 2, 1 ], [ 2, 0 ], [ 3, 3 ], [ 3, 2 ], [ 3, 1 ], [ 3, 0 ] ]
```

```
all {:y, x <- (1..4), y <- (x..1) }
```

```
[ 1, 2, 1, 3, 2, 1, 4, 3, 2, 1 ]
```

```
(
var intervals;
// a function to generate intervals between all pairs of notes in a chord voicing
intervals = {| chord|
all {:  chord[i+gap] - chord[i],
gap <- (1 ..  chord.lastIndex),
i <- (0 ..  chord.lastIndex - gap)
}
};

intervals.([0,4,7,10]).postln;
intervals.([0,1,3,7]).postln;
)
```

```
[ 4, 3, 3, 7, 6, 10 ]
[ 1, 2, 4, 3, 6, 7 ]
```

```
all {:[y, z], x<-(0..30), var y = x.nthPrime, var z = 2 ** y - 1, z.asInteger.isPrime.not  }
[ [ 11, 2047 ], [ 23, 8388607 ], [ 29, 536870911 ] ] // mersenne numbers which are no primes
```

Qualifier Clauses

A list comprehension begins with {: and contains a body followed by several qualifier clauses separated by commas.

{: *body* , *qualifiers* }

There are several types of qualifier clauses that can appear after the body.

generator clause

The basic clause is the generator clause. Its syntax is

name <- *expr*

The expression should be something that can respond meaningfully to 'do' such as a collection or a stream.
The name takes on each value of the expression.
The name is a local variable whose scope extends to all clauses to the right. The name is also in scope in the body.

```
all {:  x, x <- (1..3) }
```

```
[ 1, 2, 3 ]
```

```
all {:  x, x <- [\a, \b, \c] }
```

```
[ a, b, c ]
```

```
all {:  x, x <- (1!3)++(2!2)++3 }
```

```
[ 1, 1, 1, 2, 2, 3 ]
```

multiple generators act like nested loops.

```
all {:  [x,y], x <- (1..2), y <- (10,20..30) }
```

```
[ [ 1, 10 ], [ 1, 20 ], [ 1, 30 ], [ 2, 10 ], [ 2, 20 ], [ 2, 30 ] ]
```

generators can depend on previous values.

```
all {:  x, x <- (1..3), y <- (1..x) }
```

```
[ 1, 2, 2, 3, 3, 3 ]
```

```
all {:  x, x <- (1..3), y <- (1..4-x) }
```

```
[ 1, 1, 1, 2, 2, 3 ]
```

guard clause

A guard clause is simply an expression. It should return a boolean value.

expr

The guard acts as a filter on the results and constrains the search.

```
all {:  x, x <- (0..10), x.odd }
```

```
[ 1, 3, 5, 7, 9 ]
```

x.odd is the guard and causes all even numbers to be skipped.

```
all {:  x, x <- (0..30), (x % 5 == 0) | | x.isPowerOfTwo }
```

```
[ 0, 1, 2, 4, 5, 8, 10, 15, 16, 20, 25, 30 ]
```

you can have multiple guards.

```
all {:  [x,y], x <- (0..10), (x % 5 == 0) | | x.isPowerOfTwo, y <- (1..2), (x+y).even }
```

```
[ [ 0, 2 ], [ 1, 1 ], [ 2, 2 ], [ 4, 2 ], [ 5, 1 ], [ 8, 2 ], [ 10, 2 ] ]
```

var clause

A var clause lets you create a new variable binding that you can use in your expressions. The scope of the name extends to all clauses to the right and in the body.

var *name* = *expr*

Unlike the generator clause, the name is bound to a single value, it doesn't iterate.

```
all {:  z, x <- (1..20), var z = (x*x-x) div:  2, z.odd }
```

```
[ 1, 3, 15, 21, 45, 55, 91, 105, 153, 171 ]
```

side effect clause

This clause lets you insert code to do some side effect like printing.

:: *expr*

```
all {:  z, x <- (1..20), var z = (x*x-x) div:  2, ::  [x,z].postln, z.even }
```

termination clause

The termination clause is for stopping further searching for results. Once the expression becomes false,
the routine halts.

:while *expr*

```
// using a guard
all {:  z, x <- (1..20), var z = (x*x-x) div:  2,  ::  [x,z].postln, z < 50 }

// using a termination clause
// this one stops searching, so does less work than the above.
all {:  z, x <- (1..20), var z = (x*x-x) div:  2,  ::  [x,z].postln, :while z < 50 }
```

Constrained Search

list comprehensions can solve constrained combinatorial problems like this one:

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors.
Baker does not live on the top floor. Cooper does not live on the bottom floor.
Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper.
Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's.
Where does everyone live?

```
(
z = {:  [baker, cooper, fletcher, miller, smith] ,
var floors = (1..5),
 baker <- floors,  baker != 5,  // Baker does not live on the top floor.
  // remove baker's floor from the list.  var creates a new scope, so the 'floors' on the left is a new
```

```
binding.
var floors = floors.removing(baker),
 cooper <- floors, cooper != 1, // Cooper does not live on the bottom floor.
  var                                  // remove cooper's floor from the list.
  fletcher <- floors, (fletcher != 5) &amp;amp;&amp;amp; (fletcher != 1) // Fletcher does not live on either the
top or the bottom floor.
          &amp;amp;&amp;amp; (absdif(fletcher, cooper) > 1), // Fletcher does not live on a floor adjacent to
Cooper's.
var floors = floors.removing(fletcher), // remove fletcher's floor
 miller <- floors, miller > cooper, // Miller lives on a higher floor than does Cooper.
var floors = floors.removing(miller), // remove miller's floor
 smith <- floors, absdif(fletcher, smith) > 1  // Smith does not live on a floor adjacent to Fletcher's.

};
)

        // [3, 2, 4, 5, 1 ]
        // nil.   only one solution
```

combinatorial problems can take a lot of time to run.
you can reorder the above tests to make it run faster. generally you want to search the most constrained variables first.
the most constrained person above is fletcher, so he should be searched first, then cooper, etc.

Grammar:

Here is the BNF grammar for list comprehensions in SC.

```
[ ] - optional
{ } - zero or more

<list_compre> ::= "{:" <body> ',' <qualifiers> "}"

<body> ::= <exprseq>

<exprseq> ::= <expr> { ";" <expr> }
```

```
<qualifiers> ::= <qualifier> { ’,’ <qualifiers> }

<qualifier> ::= <generator> | <guard> | <binding> | <side_effect> | <termination>

<generator> ::= <name> "<-" <exprseq>

<guard> ::= <exprseq>

<binding> ::= "var" <name> "=" <exprseq>

<side_effect> ::= "::" <exprseq>

<termination> ::= ":while" <exprseq>
```

Code Generation:

For each of the above clauses, here is how the code is generated. The body acts as the innermost qualifier.
By understanding these translations, you can better understand how scoping and control flow work in list comprehensions.

generator:

```
expr.do {| name| ..next qualifier..  }
```

guard:

```
if (expr) { ..next qualifier..  }
```

binding:

```
{| name| ..next qualifier..  }.value(expr)
```

side effect:

```
expr ; ..next qualifier..
```

termination:

```
if (expr) { ..next qualifier..  }{ nil.alwaysYield }
```

ID: 286

Literals

Literals are values which have a direct syntactic representation.
The following sections describe the types of literals that can be represented.

Numbers

An integer is any series of digits optionally preceeded by a minus sign.
examples of integers :

```
-13
666
2112
96
```

A float is one or more decimal digits followed by a decimal point followed by one or more decimal digits.
You must have digits on both sides of the decimal point. This distinguishes floating point numbers from
integer expressions like:

```
8.rand
```

examples of floats :

```
0.39
98.6
1.0
-0.5
```

Exponential notation is also supported.

```
1.2e4
1e-4
```

The constant pi can be appended to a number to create floating point constant:

```
2pi
0.5pi
-0.25pi
```

Numbers can also be written in radices other than base 10 up to base 36.
The radix is specified in base 10 followed by the letter 'r' followed by the value written in
that radix using characters 0-9,A-Z, or a-z, for digit values from 0 to 35.
For example you can write hexidecimal numbers as follows:

```
16rF0
16rA9FF
```

Binary numbers can be written as follows:

```
2r01101011
```

Floating point values may also be specified in any base:

```
12r4A.A
```

Characters

Characters are preceeded by a dollar sign:
$A
$B
$C

Tab, linefeed, carriage return, and backslash are preceeded by a backslash:

```
$\t
$\n
$\r
$\\
```

Symbols

A symbol is written as a string enclosed in single quotes.
examples of symbols:

```
'x'
'aiff'
```

```
'BigSwiftyAndAssoc'
'nowhere here'
'somewhere there'
'.+o*o+.'
```

A symbol consisting of a single word can be written with a preceeding backslash.

```
\x
\aiff
\BigSwiftyAndAssoc
```

Strings

Strings are written in double quotes:

```
"This is a string."
```

If two or more strings are lexically adjacent, then they combine into a larger string.

example:

```
"This" " is " "also a " "string."
```

Strings may span more than one line. If so, then the new line characters become part of the string.
example:

```
"This
is
also a
string.
"
```

Identifiers

Names of methods and variables begin with a lower case alphabetic character, followed by zero or more
alphanumeric characters.

```
var abc, z123, func;
```

Class Names

Class names always begin with a capital letter followed by zero or more alphanumeric characters.

```
Object
Point
Synth
```

Special Values

The singular instances of the classes True, False and Nil are written as the words true, false, nil and inf.

```
x = true;
y = false;
z = nil;
```

Literal Arrays

Arrays of literals are created at compile time and are written with a # preceeding the array as follows:

```
#[1, 2, 'abc', "def", 4]
```

Literal Arrays must be used as is and may not be altered at run time.

In literal Arrays names are interpreted as symbols. This is not the case in regular Arrays, where they are interpreted as variable names:

```
#[foo, bar] // this is legal; an Array of Symbols
[foo, bar] // this is only legal if foo and bar have been declared as variables
```

Arrays and other collections may also be created dynamically which is explained in Collections.help.

Using a literal Array is faster than building an array dynamically every time you need it.

When nesting literal arrays, only the outermost literal array needs the '#' character.

```
#[[1, 2, 3], [4, 5, 6]]
```

Literal Arrays can be useful for things such as tables of constants, for example note names:

```
(
// build a table of note names
var table = ();
value {
var semitones = [0, 2, 4, 5, 7, 9, 11];
var naturalNoteNames = ["c", "d", "e", "f", "g", "a", "b"];

(0..9).do {| o|
naturalNoteNames.do {| c, i|
var n = (o + 1) * 12 + semitones[i];
table[(c ++ o).asSymbol] = n;
table[(c ++ "s"  ++ o).asSymbol] = n + 1;
table[(c ++ "ss" ++ o).asSymbol] = n + 2;
table[(c ++ "b"  ++ o).asSymbol] = n - 1;
table[(c ++ "bb" ++ o).asSymbol] = n - 2;
};
};
};

// translate note names to midi keys
table.atAll(#[c4, e4, gs4, c5, e5, gs5, c6])
)
```

ID: 287

Messages

Sending messages is the way things get done in an object oriented language. A message consists of a message **selector** which names the type of operation, a **receiver** to which the message is sent and in some cases a list of **arguments** which give additional information pertaining to the operation. A message always
returns a result. The kind of result depends on the kind of message. In the default case, the return value is the receiver itself.

Messages may be written using binary operators, functional notation or receiver notation.

Binary operator messages

A binary operator selector is any string of characters from the list of legal binary operator characters:

```
!  @ % &amp; * - + = | < > ?  /
```

An exception is that no operator may begin with `//` or `/*` which are comment delimiters.

A binary operator expression consists of two expressions with a binary operator between them.

```
 1 + 2  // sum of one and two

a - b  // difference of a and b

x < 0.0  // answer whether x is less than zero
```

A binary operator can also be an identifier followed by a colon.

```
10 rrand:  100
```

Operator Precedence

There is none. All binary operators have the same level of precedence and associate from left to right.
For example, the expression:

```
a * b + c * d
```

is equivalent to:

```
((a * b) + c) * d
```

and not:

```
(a * b) + (c * d)
```

Therefore it is usually better style to fully parenthesize your expressions.

Functional notation messages

The message selector preceeds the parenthesized argument list. The first argument in the list is actually
the receiver.

```
sin(x)  // sine of x

max(a, b)  // maximum of a and b
```

Receiver notation messages

A method call in functional notation may be converted to receiver notation by putting the receiver before the method name followed by a dot as shown below.

`max(a, b)` is equivalent to : `a.max(b)`
`sin(x)` is equivalent to : `x.sin`

another example:

```
g(f(a, b), c)
```

is equivalent to :

```
g(a.f(b), c)
```

is equivalent to :

```
f(a, b).g(c)
```

is equivalent to :

```
a.f(b).g(c)
```

Default Argument Values

You may call a function or method with more or fewer arguments than it was declared to accept. If fewer arguments are passed, those arguments not passed are set to a default value if one is given in the method or function definition, or otherwise to nil. If too many arguments are passed, the excess arguments are either collected into an Array or ignored depending on whether or not the function or method has an ellipsis argument (explained in **Functions**). When calling a method or function with zero arguments you can omit the parentheses:

```
// x is declared to take two arguments a and b which default to 1 and 2 respectively.
// It returns their sum.  This syntax will be explained in the section on Functions.
x = { arg a=1, b=2; a + b };

                // z is set to 3.  (a defaults to 1, b defaults to 2)

                // z is set to 12.  (a is 10, b defaults to 2)

                  // z is set to 15.  (a is 10, b is 5)

                     // z is set to 15.  (a is 10, b is 5, 9 is ignored)
```

Keyword Arguments

Arguments to Methods may be specified by the name by which they are declared in a method's definition. Such arguments are called keyword arguments. Any argument may be passed as a keyword argument except for the receiver 'this'. Keyword arguments must come after any normal (aka 'positional') arguments, and may be specified in any order. If a keyword is specified and there is no matching argument then it is ignored and a warning will be printed. This warning may be turned off globally by making the

following call:

```
keywordWarnings(false)
```

If a keyword argument and a positional argument specify the same argument, then the keyword argument value overrides the positional argument value.

For example the 'ar' class method of the SinOsc class takes arguments named **freq**, **phase**, **mul**, and **add** in that order. All of the following are legal calls to that method.

```
SinOsc                    // all normal arguments:  freq, phase, mul, add

// freq = 800, mul = 0.2, others get default values.
SinOsc.ar(800, mul:  0.2);

// freq = 800, phase = pi, mul = 0.2, add gets its default value of zero.
SinOsc.ar(phase:  pi, mul:  0.2, freq:  800);

// keyword value of 1200 overrides the positional arg value of 800
SinOsc.ar(800, freq:  1200);

SinOsc                // invalid keyword prints warning
```

The arguments to a Function may also be specified by keyword arguments when using the 'value' message.

```
// function args may be specified by keyword.
{ arg a=1, b=2, c=3; [a, b, c].postln }.value(b:  7, c:  8);
```

You may also use keyword arguments when using the 'perform' method.

```
SinOsc.perform('ar', phase:  pi, mul:  0.2, freq:  800);
```

Cost of using keyword arguments

When using keyword arguments there is a runtime cost to do the matching that you should be aware of. The cost can be worth the convenience when speed is not critical.

ID: 288

Function Creation via Partial Application

Partial application is a way to create a function by passing only some arguments to a method. The _ character stands in for missing arguments and becomes an argument to the created function. It only applies to a single method, list, or dictionary call, not to a more complex nested expression.

for example:

```
f = _ + 2;
```

f is now a function of one argument.

```
f.value(7);
```

it is equivalent to having written:

```
f = {| x|  x + 2 };
```

(except that there is no name 'x' declared)

```
g = Point(_, _);
```

g is a function of two arguments.

```
g.value(3, 4);
```

Here are some example usages of this in a collect message. Below each is written the equivalent function.

```
(1..8).collect(_.isPrime);
(1..8).collect {| x|  x.isPrime };

(1..8).collect(_.hash);
(1..8).collect {| x|  x.hash };
```

```
(1..8).collect([\a, \b, _]);
(1..8).collect {| x|  [\a, \b, x] };

(1..8).collect((a:_));
(1..8).collect {| x|  (a:x) };

(1..8).collect(Polar(_, pi));
(1..8).collect {| x|  Polar(x, pi) };

(1..8).collect((1.._));
(1..8).collect {| x|  (1..x) };

       _    _   // f is a two argument function
       _        // g is a partial application of f
                // get the answer

// equivalent to this:
f = {| x, y|  (a:x, b:y) }
g = {| z|  f.(z, 5) };
g.value(7);
```

An example of what you can't do:

```
(1..8).collect( Point(100 * _, 50) ); // nested expression won't work.
// only the * gets partially applied, not the surrounding expression.

                | x| Point                // need to use a function for this.
```

ID: 289

Polymorphism

Polymorphism is the ability of different classes to respond to a message in different ways. A message generally has some underlying meaning and it is the responsibility of each class to respond in a way appropriate to that meaning.

For example, the '**value**' message means "give me the effective value of this object".

The value method is implemented by these classes (among others):

```
Function :     this.value(args)
Object :     this.value()
Ref :     this.value
```

Let's look at how these classes implement the value message.

Here's the value method in class **Object**:

```
value { ^this }
```

It simply returns itself. Since all classes inherit from class Object this means that unless a class overrides 'value', the object will respond to 'value' by returning itself.

```
            // posts itself
                 // value returns itself
'a symbol'
'a symbol'.value.postln;
[1,2,3].value.postln;
//etc...
```

In class **Function** the value method is a primitive.

```
value { arg ...  args;
  _FunctionValue
  // evaluate a function with args
^this.primitiveFailed
}
```

_FunctionValue is a C code primitive, so it is not possible to know just by looking at it what it does. However what it does is to evaluate the function and return the result.

```
                        // posts Instance of Function
{ 5.squared }.value.postln;  // posts 25
```

The **Ref** class provides a way to create an indirect reference to an object. It can be used to pass a value by reference. Ref objects have a single instance variable called '**value**'. The '**value**' method returns the value of the instance variable '**value**'. Here is the class definition for **Ref**.

```
Ref   AbstractFunction
{
var <>value;
*new { arg thing; ^super.new.value_(thing) }
set { arg thing; value = thing }
get { ^value }
dereference { ^value }
asRef { ^this }

 //behave like a stream
next { ^value }
embedInStream { arg inval;
^this.value.embedInStream(inval)
}

printOn { arg stream;
stream << "‘(" << value << ")";
}
storeOn { arg stream;
stream << "‘(" <<< value << ")";
}
}
```

Here is how it responds :

```
Ref.new(123).postln;
Ref.new(123).value.postln;
```

Ref also implements a message called '**dereference**' which is another good example of polymorphism. As implemented in Ref, dereference just returns the value instance variable which is no different than what the value method does. So what is the need for it? That is explained by how other classes respond to dereference. The dereference message means "remove any Ref that contains you". In class Object dereference returns the object itself, again just like the value message. The difference is that no other classes override this method. So that dereference of a Function is still the Function.

```
Object :     this.dereference()
Ref :     this.dereference()

5.value.postln;
{ 5.squared }.value.postln;
Ref.new(123).value.postln;

5.dereference.postln;
{ 5.squared }.dereference.postln;
Ref.new(123).dereference.postln;
```

Yet another example of polymorphism is **play**. Many different kinds of objects know how to play themselves.

```
{ PinkNoise.ar(0.1) }.play;      // Function

(            // AppClock
var w, r;
w = SCWindow("trem", Rect(512, 256, 360, 130));
w.front;
r = Routine({ arg appClockTime;
["AppClock has been playing for secs:",appClockTime].postln;
60.do({ arg i;
0.05.yield;
w.bounds = w.bounds.moveBy(10.rand2, 10.rand2);
w.alpha = cos(i*0.1pi)*0.5+0.5;
});
1.yield;
w.close;
});
AppClock.play(r);
```

```
)

(            // SynthDef
    SynthDef "Help-SynthDef"
{ arg out=0;
Out.ar(out, PinkNoise.ar(0.1))
}).play;
)

Pbind(\degree, Pseq([0, 1, 2, 3],inf)).play; // Pattern
```

Polymorphism allows you to write code that does not assume anything about the implementation of an object, but rather asks the object to "do what I mean" and have the object respond appropriately.

ID: 290

Scoping and Closure

SuperCollider has nested scoping of variables. A function can refer not only to its own arguments and variables, but also to those declared in any enclosing (defining) contexts.

For example :

The function defined below within `makeCounter` can access all of the arguments and variables declared in `makeCounter`. Other code can call the returned function at some later time and it can access and update the values contained in `makeCounter` at the time when the inner function was instantiated.

```
(
var makeCounter;
makeCounter = { arg curVal=0, stepVal=1;
  // return a function :
{
var temp;
  // temp is local to this function, curVal &amp;amp; stepVal in the
  // enclosing function are referred to here within.
temp = curVal;
curVal = curVal + stepVal;
  temp     // return result
}
};

// each invocation of makeCounter creates a new set of variables curVal and stepVal

x = makeCounter.value(10, 1);
z = makeCounter.value(99, 100);

// x and z are functions which refer to different instances of the variables curVal and stepVal

x.value.postln; // posts 10
x.value.postln; // posts 11
z.value.postln; // posts 99
z.value.postln; // posts 199
x.value.postln; // posts 12
```

```
x.value.postln; // posts 13
z.value.postln; // posts 299
z.value.postln; // posts 399

)
```

Note that even though the function which defines curVal and stepVal has completed execution, its variables are still accessible to those functions that were defined within its context. This is known as **lexical closure**, the capturing and availability of variables defined in outer contexts by inner contexts even when the outer contexts may have completed execution.

ID: 291

Catalog of symbolic notations in SuperCollider

Arithmetic operators

Math operators apply to many classes, including arrays and other collections.

Using a basic math operator on a Symbol swallows the operation (returns the symbol)

```
\symbol
symbol
```

```
number + number       addition
number - number    subtraction
number * number    multiplication
number / number       division
number % number       modulo
number ** number    exponentiation
```

Bitwise arithmetic

```
number &amp;amp; number    bitwise and
number | number       bitwise or
number << number    bitwise left shift
number >> number    bitwise right shift
number +>> number    unsigned bitwise right shift
```

Logical operators

```
object == object       equivalence
object === object       identity
object != object    not equal to
object !== object    not identical to
```

Objects may be equivalent but not identical.

```
[1, 2, 3] == [1, 2, 3]
```

```
true
[1, 2, 3] === [1, 2, 3]
false // a and b are two different array instances with the same contents

a = b = [1, 2, 3];
a === b;
true  // a and b are the same array instance
```

`number < number` comparison (less than)
`number <= number` comparison (less than or equal to)
`number > number` comparison (greater than)
`number >= number` comparison (greater than or equal to)

`Boolean &amp;&amp; Boolean` logical And
`Boolean | | Boolean` logical Or

When a function is the second operand, these operators perform short-circuiting (i.e., the function is executed only when its result would influence the result of the operation). This is recommended for speed.

With and: and or: second-argument functions will be inlined. If you use && or | | , no inlining will be done and performance will be slower.

```
a = 1;

a == 1 and:  { "second condition".postln; [true, false].choose }
second condition
true

a == 1 or:  { "second condition".postln; [true, false].choose }
true

a != 1 and:  { "second condition".postln; [true, false].choose }
false

a != 1 or:  { "second condition".postln; [true, false].choose }
second condition
true
```

In this case, the second condition will cause an error if a is nil, because nil does not understand addition. a.notNil is a safeguard to ensure the second condition makes sense.

```
a = nil;
a.notNil and:  { "second condition".postln; (a = a+1) < 5 }
false

a = 10;
a.notNil and:  { "second condition".postln; (a = a+1) < 5 }
second condition
false
```

Array and Collection operators

`object ++ object` concatenation
`collection +++ collection` lamination (see **[J_concepts_in_SC]**)

`collection @ index` collection/array indexing: .at(index) or [index]
`collection @@ integer` collection/array indexing: .wrapAt(int)
`collection @| @ integer` collection/array indexing: .foldAt(int)
`collection | @| integer` collection/array indexing: .clipAt(int)

Set operators

`set &amp; set` intersection of two sets
`set | set` union of two sets
`setA - setB` difference of sets (elements of setA not found in setB)

`set -- set` symmetric difference

(setA – setB) == ((setA - setB) | (setB - setA))

```
a = Set[2, 3, 4, 5, 6, 7];
b = Set[5, 6, 7, 8, 9];

a - b
Set[ 2, 4, 3 ]
```

```
b - a
Set[ 8, 9 ]

((a-b) | (b-a))
Set[ 2, 9, 3, 4, 8 ]

a -- b
Set[ 2, 9, 3, 4, 8 ]
```

Geometry operators

`number @ number` x @ y returns Point(x, y)
`point @ point` Point(left, top) @ Point(right, bottom) returns Rect(left, top, right-left, bottom-top)
`ugen @ ugen` create a Point with 2 UGens

`rect &amp; rect` intersection of two rectangles
`rect | rect` union of two rectangles (returns a Rect whose boundaries exactly encompass both Rects)

IOStream operators

`stream << object` represent the object as a string and add to the stream

A common usage is with the Post class, to write output to the post window.

```
Post   "Here is a random number:  "              ".\n"
Here is a random number:  13.
```

`stream <<* collection` add each item of the collection to the stream

```
Post << [0, 1, 2, 3]
[ 0, 1, 2, 3 ]

Post <<* [0, 1, 2, 3]
0, 1, 2, 3
```

`stream <<< object` add the object's compile string to the stream

```
Post    "a string"
"a string"
```

`stream <<<* collection` add each item's compile string to the stream

Conditional execution operators

`object ? object` nil check (no .value)
`object ?? function` nil check (.value, function is inlined)

If the object is nil, the second expression's value will be used; otherwise, it will be the first object.

```
a = [nil, 5];

10.do({ (a.choose ? 20.rand).postln });
10.do({ (a.choose ?? { 20.rand }).postln });
```

?? { } is generally recommended. ? always evaluates the second expression, even if its value will not be used. ?? evaluates the function conditionally (only when needed). If the function defines no variables, the function will be inlined for speed.

Especially useful when the absence of an object requires a new object to be created. In this example, it's critical that a new SCSlider not be created if the object was already passed in.

```
    | slider, parent|
slider = slider ?? { SCSlider.new(parent, Rect(0, 0, 100, 20)) };
slider.value_(0);
};
```

If the first line inside the function instead read `slider = slider ? SCSlider.new(parent, Rect(0, 0, 100, 20));`, a new slider would be created even if it is not needed, or used.

`object !? function` execute function if object is not nil

```
a = [10, nil].choose;
```

```
a !?  { "ran func".postln };
// equivalent of:
if (a.notNil) { "ran func".postln };
```

Used when an operation requires a variable not to be empty.

```
f = { | a|  a + 5 };
f.value
// error:  nil does not understand +

f = { | a|  a !?  { a+5 } };
f.value
nil // no error
f.value(2)
7
```

Miscellaneous operators

`object !  number` object.dup(number)

```
15 !  5
[ 15, 15, 15, 15, 15 ]
```

If the object is a function, it behaves like Array.fill(number, function).

```
{ 10.rand } !  5
[ 8, 9, 3, 8, 0 ]
```

`object -> object` creates an Association, used in dictionaries

`expression <!  expression` bypass value of second expression

This operator evaluates both expressions, and returns the value of the first.

```
a = 0;
0

// a is incremented twice, but the return value (1)
// comes from the first increment (0 + 1)
```

```
(a = a + 1) <!  (a = a + 1)
1


  // a's value reflects both increments
2
```

`function <> function` function composition operator

This operator returns a new function, which evaluates the second function and passes the result to the first function.

```
f = { | a|  a * 5 } <> {| a|  a + 2 };
f.(10);
   // == (10+2) * 5
```

An array as argument is passed through the chain:

```
f.([10, 75, 512]);
                 // == ([10, 75, 512]+2) * 5
```

Symbolic notations to define literals/other objects

`$` character prefix: "ABC".at(0) == $A
`''` or `\` define a literal Symbol: 'abc' === \abc
`""` define a literal String
`[item, item...]` define an Array containing given items
`Set[item, item...]` define a Set – any Collection class name can be used other than Set
`#[item, item...]` define a literal Array
`(a:1, b:2)` define an Event (same as Event[\a -> 1, \b -> 2])
`` ` `` (backtick or backquote) define a Ref: `1 == Ref(1), `(a+1) == Ref(a+1)

`\` inside a string or symbol, escapes the next character

```
"abc\"def\"ghi"
abc"def"ghi


'abc\'def\'ghi'
abc'def'ghi
```

\t tab character
\n newline character
\l linefeed character
\r carriage return character
\\ \character

`{ }` define an open function
`#{ }` define a closed function
`(_ * 2)` define a function { | a| a * 2 } (see **[Partial-Application]**)

Argument definition

`| a, b, c|` define function/method arguments
`| a, b ... c|` define function/method arguments;
arguments after a and b will be placed into c as an array

`#a, b, c = myArray` assign consecutive elements of myArray to multiple variables
`#a, b ... c = myArray` assign first two elements to a and b; the rest as an array into c

Where f is a function

`f.( )` evaluate the function with the arguments in parentheses
`f.(*argList)` evaluate the function with the arguments in an array
`f.(anArgName: value)` keyword addressing of function or method arguments

`SomeClass.[index]` Equivalent to SomeClass.at(index) – Instr.at is a good example

`myObject.method(*array)` call the method with the arguments in an array
`obj1 method: obj2` same as obj.method(obj2) or method(obj1, obj2)
This works only with single-argument methods.

Class and instance variable access

Inside a class definition (see **[Writing-Classes]**):

`classvar<a,` Define a class variable with a getter method (for outside access)
`>b,` Define a class variable with a setter method
`<>c;` Define a class variable with both a getter and setter method

`var <a,` Define an instance variable with a getter method (for outside access)

`>b,` Define an instance variable with a setter method
`<>c;` Define an instance variable with both a getter and setter method

These notations do not apply to variables defined within methods.

`^someExpression` Inside a method definition: return the expression's value to the caller

`instVar_ { }` define a setter for an instance variable
`myObject.instVar = x;` invoke the setter: (myObject.instVar_(x); x)

Array series and indexing

`(a..b)` produces an array consisting of consecutive integers from a to b
`(a, b..c)` e.g.: (1, 3..9) produces [1, 3, 5, 7, 9]
`(..b)` produces an array 0 through b
`(a..)` not legal (no endpoint given)

`a[i..j]` a.copyRange(i, j)
`a[i, j..k]` e.g.: a[1, 3..9] retrieves array elements 1, 3, 5, 7, 9
`a[..j]` a.copyRange(0, j)
`a[j..]` a.copyRange(i, a.size-1) (this is OK–Array is finite)

access an environment variable
`abc` compiles to \abc.envirGet
`abc = value` compiles to \abc.envirPut(value)

Adverbs to math operators (see **[Adverbs]**)

e.g.:

```
[1, 2, 3] * [2, 3, 4]
[ 2, 6, 12 ]

[1, 2, 3] *.t [2, 3, 4]
[ [ 2, 3, 4 ], [ 4, 6, 8 ], [ 6, 9, 12 ] ]
```

`.s` output length is the shorter of the two arrays
`.f` use folded indexing instead of wrapped indexing
`.t` table-style
`.x` cross (like table, except that the results of each operation are concatenated, not added as another dimension)

.0 operator depth (see **[J_concepts_in_SC]**)
.1 etc.

ID: 292

Syntax Shortcuts

This file shows a number of syntax equivalences in the compiler.

Example: multiple ways to write the same thing.

Because of the multiple syntax equivalences, some expressions can be written in many different ways. All of the following do the same thing and compile to the same code.

```
// new argument syntax

(1..10).collect({| n|  n.squared }); // receiver syntax

collect((1..10), {| n|  n.squared }); // function call syntax

                | n|              // receiver syntax with trailing function arg

                  | n|              // function call syntax with trailing function arg

(1..10) collect:  {| n|  n.squared }; // binary operator syntax


// old argument syntax

(1..10).collect({ arg n; n.squared }); // receiver syntax

collect((1..10), { arg n; n.squared }); // function call syntax

                 arg               // receiver syntax with trailing function arg

                   arg               // function call syntax with trailing function arg

(1..10) collect:  { arg n; n.squared }; // binary operator syntax


// partial application syntax
```

```
(1..10).collect( _.squared ); // receiver syntax

collect((1..10), _.squared ); // function call syntax

(1..10) collect:  _.squared ; // binary operator syntax
```

You could even start expanding out the equivalent of `(1..10)` which is really a shortcut for `series(1, nil, 10)`. This could also be written `1.series(nil,10)`. This adds another 26 variations to the 13 variations above.

---

functional and receiver notation

instead of writing: you can write:

```
f(x, y)        x.f(y)
f(g(x))        x.g.f
```

defining instance variable accessor methods

instead of writing: you can write:

```
Thing { var x;       Thing { var <>x; }
x { ^x }
x_ { arg z; x = z; }
}
```

calling an instance variable setter method

instead of writing: you can write:

```
p.x_(y)        p.x = y;
```

use a selector as binary operator

instead of writing: you can write:

```
min(x, y)       x min:  y
```

multiple assignment

instead of writing: you can write:

```
x = z.at(0); y = z.at(1);   # x, y = z;
```

get environment variable

instead of writing: you can write:

```
'myName'.envirGet      myName
```

set environment variable

instead of writing: you can write:

```
'myName'.envirSet(9);     myName = 9;
```

instantiate object

instead of writing: you can write:

```
Point.new(3, 4);     Point(3, 4)
```

create a collection

instead of writing: you can write:

```
Set.new.add(3).add(4).add(5);  Set[3, 4, 5]
```

moving blocks out of argument lists

instead of writing: you can write:

```
if (x<3, {\abc}, {\def});   if (x<3) {\abc} {\def}

z.do({| x| x.play });    z.do {| x| x.play };

while({ a < b },{ a = a * 2 });  while { a < b } { a = a * 2 };
```

shorter argument lists

instead of writing: **you can write:**

```
{ arg x; x < 2 }     {| x| x < 2 }
```

shorthand for Symbols

instead of writing: **you can write:**

```
'mySymbol'       \mySymbol
```

creating a Ref

instead of writing: **you can write:**

```
Ref.new(thing)      `thing
```

calling the 'value' method

instead of writing: **you can write:**

```
f.value(x)       f.(x)
```

indexing with 'at'

instead of writing: **you can write:**

```
z.at(i)        z[i]
```

indexing with 'put'

instead of writing: **you can write:**

```
z.put(i, y);      z[i] = y;
```

creating IdentityDictionaries

instead of writing: **you can write:**

```
IdentityDictionary['a'->1,'b'->2]  (a:  1, b:  2)
```

creating arithmetic series

instead of writing: you can write:

```
Array.series(16,1,1)  , or  series(1,nil,16)  (1..16)

Array.series(6,1,2)  , or  series(1,3,11)  (1,3..11)
```

accessing subranges of Arrays

instead of writing: you can write:

```
a.copyRange(4,8)      a[4..8]

a.copyToEnd(4)       a[4..]

a.copyFromStart(4)      a[..4]
```

calling performList

instead of writing: you can write:

```
object.performList(\method, a, b, array)  object.method(a, b, *array)
```

partial application

instead of writing: you can write:

```
{| x| object.msg(a, x, b) }      object.msg(a, _, b)

{| x,y| object.msg(a, x, y) }     object.msg(a, _, _)

{| x| a + x }          a + _

{| x| [a, b, x] }        [a, b, _]
```

```
{| x| (a:  x) }          (a:  _)
```

ID: 293

Understanding errors

1. **Reading error dumps**
2. **Error objects and error handling**
3. **Common primitive errors**
4. **A common network error**
5. **A common warning**

1. Reading error dumps

When sc3 reports an error to the user, there are usually three parts:

- the error text
- a dump of the receiver of the method that caused the error, and/or any arguments of the method call
- a dump of the call stack to the point of the error

For example:

```
        // no class implements this method; therefore you'll get an error

// error text
ERROR: Message 'blech' not understood.
// receiver and args
RECEIVER:
Integer 1
ARGS:
Instance of Array {    (02207560, gc=01, fmt=01, flg=11, set=00)
indexed slots [0]
}
// call stack
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of DoesNotUnderstandError>
Nil-handleError
arg this = nil
arg error = <instance of DoesNotUnderstandError>
Object-throw
```

```
arg this = <instance of DoesNotUnderstandError>
Object-doesNotUnderstand
arg this = 1
arg selector = 'blech'
arg args = [*0]
< closed FunctionDef >  (no arguments or variables)
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
Process-interpretPrintCmdLine
arg this = <instance of Main>
```

Each of these parts provides valuable information about the cause of the error. Debugging is much easier if you understand what the error output means.

Error text: A string describing the error. In this case, "Message 'xxx' not understood" means that you attempted to use the method xxx on a class that does not implement it.

Receiver and arguments: The method was applied to an integer (1), with no arguments (the size of the arguments array is 0).

Call stack: Order of execution in the call stack is in reverse: the top of the stack shows the most recent calls.

Most call stacks for errors will show the same top three calls as shown here (calling the method reportError on an error class, calling handleError on Nil, and calling throw on the error object). You can ignore these three calls.

Following is the meat: the error happened when an object was not understood. Continuing to read down, it happened inside a function definition. (Every time you highlight a block of code and press the enter key, the code is compiled into a function definition and executed. So, this function definition simply refers to the text submitted to the interpreter.) And, it all began with the instruction to interpret and print a command line.

Here is a slightly more complex example, showing how you can use the variables listed for each call in the call stack to help locate the error.

```
Routine
var a;
a = 5;
loop {
var b;
b = 20.rand;
b.postln.ecky_ecky_phtang;   // "NI!!!!"
a.wait;
}
}).play;

ERROR: Message 'ecky_ecky_phtang' not understood.
RECEIVER:
Integer 6
ARGS:
Instance of Array {    (02207560, gc=01, fmt=01, flg=11, set=00)
indexed slots [0]
}
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of DoesNotUnderstandError>
Nil-handleError
arg this = nil
arg error = <instance of DoesNotUnderstandError>
Object-throw
arg this = <instance of DoesNotUnderstandError>
Object-doesNotUnderstand
arg this = 6
arg selector = 'ecky_ecky_phtang'
arg args = [*0]
< FunctionDef in closed FunctionDef >
var b = 6
Function-loop
arg this = <instance of Function>
< FunctionDef in closed FunctionDef >
var a = 5
Routine-prStart
arg this = <instance of Routine>
arg inval = 1542.075067
```

Reading from the bottom this time, to trace the flow in chronological order: this time, execution did not begin with the command line, but with a routine commencing within the scheduler (Routine({...}).play). Note that there are two calls identified as "Function-Def in closed FunctionDef" and that they can be distinguished by the variables contained within. The earlier call (second from the bottom) defines the variable "a" while the other defines "b." To locate the error in the code, then, you should look for a function defining the variable "b" that is *called* within another function defining "a" inside a routine.

What if the error occurred not inside a function definition that you wrote, but inside a method in the class library? There may be a bug in the method, or you may have thought the method took a certain kind of argument when in fact it expects something else.

If you double click on the construction "ClassName-methodName" in the call stack, the whole thing is selected. Then you can press cmd-J to open the method definition and look at the source code.

2. Error objects and error handling

sc3 implements error reporting using Error objects, which are instances of the class Error or one of its subclasses. Any code (whether in the class library or any user application) can throw an error any time as follows:

```
Error "This is a basic error."
```

You can also catch exceptions that occur within functions by executing the function with "try" or "protect" instead of "value."

try - execute the first function. On an error, execute the second function and suppress the error. The second function can rethrow the error if desired, allowing you to decide which errors will be reported and which suppressed. In this example, we do not rethrow the error, so the error is swallowed and execution continues to the end.

```
try { 1.blech } { | error|  "oops".postln };
"next line"


oops
next line
```

protect - executes the first function. On an error, execute the second function before

reporting the error. This is useful when the steps before the protect make some changes that need to be undone if an error occurs. See the method Environment-use for an example.

```
protect { 1.blech } { | error|  "oops".postln };
"next line"

      // without protect, this would not be posted
ERROR: Message 'blech' not understood.
RECEIVER:
Integer 1
ARGS:
Instance of Array {    (02207560, gc=01, fmt=01, flg=11, set=00)
indexed slots [0]
}
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of DoesNotUnderstandError>
```

Prior to August 2004, try and protect do not return the value of the function to the caller if there is no error.

```
try { 1+1 }

a Function
```

More recent builds (since early August 2004) do return the function's value. Non-error objects can be thrown using the class **Exception**.

```
try { 1+1 }
2

// can't add a Point to an integer - binary op failed error
// result of catch func is returned instead
try { 1+Point(0, 0) } { 2*5 }
10
```

3. Common primitive errors

- operation cannot be called from this Process.

This is usually the results of performing a GUI operation within a routine or scheduled function that is executing on some clock other than AppClock. AppClock is the only clock that can execute GUI manipulation because it is a lower priority thread. If the CPU is busy with audio synthesis or maintaining accurate scheduling for musical events, AppClock events will be delayed until the CPU is free enough.

Solution: write your GUI updates as follows. **defer** schedules the function on AppClock.

```
{ myGUIObject.value_(newValue) }.defer;
```

- Attempted write to immutable object.

```
#[0, 1, 2].put(1, 3)
```

```
ERROR: Primitive '_BasicPut' failed.
Attempted write to immutable object.
```

#[0, 1, 2] is a literal array. Literal arrays cannot be manipulated–they can only be indexed. They cannot be changed internally.

Solution: copy the array first.

```
#[0, 1, 2].copy.put(1, 3)
```

```
[ 0, 3, 2 ]
```

- Index not an Integer.

```
#[0, 1, 2].at(\1)
```

```
ERROR: Primitive '_BasicAt' failed.
Index not an Integer
```

Arrays can be indexed only with integers (or, in builds since August 2004, floats).

Solution: use .asInteger—note that if the object cannot be converted into an integer, you'll get a "Does not understand" error!

```
#[0, 1, 2].at(\1.asInteger)
1
```

- Index out of range.

```
[0, 1, 2].put(5, 5)

ERROR: Primitive '_BasicPut' failed.
Index out of range.
```

Arrays have a finite size. If you try to put an object into an array slot but the slot does not exist because the array is too small, you'll get this error.

Solution: extend the array.

```
[0, 1, 2].extend(6).put(5, 5)

[ 0, 1, 2, nil, nil, 5 ]
```

Note that if the argument to extend() is smaller than the array, then the array will be truncated. If you're not sure, use max:

```
i = rrand(5, 10);
a = [0, 1, 2];
a.extend(max(i+1, a.size)).put(i, 100);
```

Why i+1? An array with size 4 allows 0, 1, 2 and 3 as indexes (4 elements starting with 0).

If it's a new array, use .newClear instead of .new.

```
a = Array.new(4);
a.put(3, 1);
ERROR: Primitive '_BasicPut' failed.
Index out of range.

a = Array.newClear(4);
a.put(3, 1);
[ nil, nil, nil, 1 ]
```

4. A common network error

```
Exception in World_OpenUDP: unable to bind udp socket
```

This is because you have multiple servers running, left over from crashes, unexpected quits etc.
One can't cause them to quit via OSC (the boot button).

```
// use this to remove them:
Server.killAll
```

5. A common warning

```
WARNING: FunctionDef contains variable declarations and so will not be inlined.
```

This warning can be safely ignored. Your code will still run, even if you get this warning.

Inlining is a compiler optimization that takes the operations inside a function and places them in the main line of the containing function. For instance,

```
// inlined
{ while { 0.9.coin } { 10.rand.postln }
}.def.dumpByteCodes;

BYTECODES: (16)
0   40       PushLiteral Float 0.9   3FECCCCC CCCCCCCD  // { 0.9.coin }
1   0D 2C    SendSpecialUnaryArithMsgX 'coin'
3   F9 00 09 JumpIfFalsePushNil 9  (15)
6   2C 0A    PushInt 10         // { 10.rand.postln }
8   0D 25    SendSpecialUnaryArithMsgX 'rand'
10   C1 38    SendSpecialMsg 'postln'
12   FD 00 0D JumpBak 13  (0)
15   F2       BlockReturn
a FunctionDef in closed FunctionDef
```

This function contains two other functions. One is the condition for the while loop; the other is the while loop's action. The compiler renders this into a single code block, using jump instructions to handle the looping and exit.

If, however, one of the functions defines a variable, then that function requires a separate execution frame. In this case, it's necessary for the compiler to push function definition objects onto the stack.

```
// not inlined
{ while { 0.9.coin } {
    var   // variable here prevents optimization
a = 10.rand;
a.postln
}
}.def.dumpByteCodes;

BYTECODES: (7)
0   04 00    PushLiteralX instance of FunctionDef in closed FunctionDef
2   04 01    PushLiteralX instance of FunctionDef in closed FunctionDef
4   C2 0C    SendSpecialMsg 'while'
6   F2       BlockReturn
a FunctionDef in closed FunctionDef
```

Inlined code will run faster, because pushing and using different execution frames is extra work for the virtual machine. If you're very concerned about speed, you can use this warning as an indicator that you might be able to optimize something in your code further.

Sometimes, there's no way around un-optimized code. To wit,

```
// inlined, optimized, but you'll get stuck notes
Routine
var synth;
{ synth = Synth("someSynth", [...args...]);
thisThread.clock.sched(10, {
synth.free;
});
2.wait;
}.loop;
```

```
}).play;

// not inlined, but no stuck notes
Routine
{ var synth;
synth = Synth("someSynth", [...args...]);
thisThread.clock.sched(10, {
synth.free;
});
2.wait;
}.loop;
}).play;
```

The first routine can be optimized because there is no variable declaration inside the loop. But, the synth variable changes on each iteration, meaning that by the time the first release happens, you don't have access anymore to the first note. Thus the first note will never terminate.

In the second case, each note has its own synth variable, so the notes will be terminated as expected. You get a warning, but it's better because the results are correct.

A solution to the above problem is to use a function with local variables.

```
(
Routine
var func;
func = {
  var       // this variable is local to the function
synth = Synth("default");
[\play, synth].postln;
thisThread.clock.sched(4.5, {
synth.free;
[\free, synth].postln;
});
};
{ func.value; 1.wait; }.loop
}).play;
)
```

14 Linux

ID: 294

```
// ============================================================

// LID – Linux Input Device
// ============================================================

//
// This class provides a way to access devices in the linux input
// layer, which supports many input devices (mouse, keyboard,
// joystick, gamepad, tablet) and busses (serial, PS/2, USB).

// ============================================================

// Opening a device
// ============================================================

//
// Input devices are accessed through device nodes, typically
// /dev/input/event[0-9]. When using a userspace daemon like udev,
// meaningful names can be assigned to devices.

// raw device name
d = LID("/dev/input/event4");

// symbolic device name
d = LID("/dev/input/trackball");

// device name relative to LID.deviceRoot
d = LID("gamepad");

// build a table of the available devices:

LID.buildDeviceTable

// buildDeviceTable builds a table of the devices found in LID.deviceRoot+"/event",
// trying to open all that it finds, looking up its name and closing them again.
// the results is returned and can later be accessed by LID.deviceTable.
// you can query another name than "/event" by passing an argument.
// (the search will be: LID.deviceRoot++"/"++name++"*")
```

```
LID.buildDeviceTable( "mouse" );

// is likely to give the info that the devices could not be opened, as "mouse"
// uses another interface (you can of course access mice via the "event" interface)

//=============================================================

// Querying device information
//=============================================================


d.info;
d.info.name;
d.info.vendor.asHexString(4);
d.info.product.asHexString(4);

//=============================================================

// Querying device capabilities
//=============================================================

//
// Device capabilities are reported as event type and event code
// mappings. Event type and event code constants can be found in
// /usr/include/linux/input.h

d.caps;
d.dumpCaps;

//=============================================================

// Event actions (raw events)
//=============================================================

//
// The device's 'action' instance variable can be used for event
// notifications. it is passed the event type, code and current value.

(
```

```
d.action = { | evtType, evtCode, evtValue |
[evtType.asHexString(4), evtCode.asHexString(4), evtValue].postln
}
)

d.action = nil;

//=====================================================================

// Event actions (raw slot events)
//=====================================================================

//
// When 'action' is nil, actions can be bound to specific events.

(
d.slot(0x0001, 0x0120).action = { | slot |
[slot.type.asHexString(4), slot.code.asHexString(4), slot.rawValue].postln;
}
)

//=====================================================================

// Device specs
//=====================================================================

//
// Device specs are mappings between event codes and symbolic control
// names. New specs can be added to LID.specs via LID»*register.

// Add a mouse device spec for a Logitech trackball
LID.register('Logitech Trackball', LID.mouseDeviceSpec);

// Add a custom device spec for a Logitech gamepad
(
LID.register('Logitech WingMan RumblePad', (
// key
rumble: #[0x0001, 0x0102], // rumble (toggles ff)
mode: #[0x0001, 0x0103], // mode (switches h and l)
a: #[0x0001, 0x0120],  // button a
```

```
b: #[0x0001, 0x0121],  // button b
c: #[0x0001, 0x0122],  // button c
x: #[0x0001, 0x0123],  // button x
y: #[0x0001, 0x0124],  // button y
z: #[0x0001, 0x0125],  // button z
l: #[0x0001, 0x0126],  // left front button
r: #[0x0001, 0x0127],  // right front button
s: #[0x0001, 0x0128],  // button s
// abs
lx: #[0x0003, 0x0000],  // left joystick x
ly: #[0x0003, 0x0001],  // left joystick y
rx: #[0x0003, 0x0005],  // right joystick x
ry: #[0x0003, 0x0006],  // right joystick y
hx: #[0x0003, 0x0010],  // hat x
hy: #[0x0003, 0x0011],  // hat y
slider: #[0x0003, 0x0002] // slider
));
)

//=============================================================

// Event actions (symbolic slot events)
//=============================================================

//
// When a device spec was registered for a given device name, slot
// actions can be assigned by using the symbolic control name.

d[\a].action = { | slot | [\a, slot.value].postln };

// There is also a default keyboard device spec.

(
LID.keyboardDeviceSpec.keys.do { | key |
d[key].action = { | slot | [key, slot.value].postln }
}
)

//=============================================================
```

```
// LED's
//=============================================================


// some devices have LEDs which can be turned on and off. These show up
// with d.caps as events of type 0x0011

d = LID("/dev/input/event0");
// LED's can be turned on:
d.setLEDState( 0x0, 1 )
// (LED 0x0 should be available on any keyboard)
// and off:
d.setLEDState( 0x0, 0 )
d.close;

// setLEDState( evtCode, evtValue ): value should be 1 or 0


//=============================================================

// Grabbing devices
//=============================================================

//
// Given proper permissions, devices can be grabbed to prevent use in
// other applications (including X). Be careful when grabbing mouse or
// keyboard!

d[\b].action = { d.ungrab };
d.grab;

d.isGrabbed;

//=============================================================

// Closing devices
//=============================================================


d.close;
```

```
LID.closeAll;

// ================================================================
```

15 Mark_Polishook_tutorial

15.1 Debugging

ID: 295

My code doesn't work!

Code doesn't always run as one might hope. In such cases, SuperCollider sometimes tells you why and sometimes it doesn't. When SuperCollider does supply information, it's usually to describe either a syntax or a runtime error.

When SuperCollider doesn't give information, it's often because the code works but not as expected. Example of this are synths (nodes) that execute in the wrong order (a source placed after, instead of before, an effect) and adding instead of multiplying (biasing an amplitude instead of scaling it).

For context, here are links that describe debugging (fixing errors in code) in languages other than SuperCollider.

http://www.elanus.net/book/debugging.html
http://www.javaworld.com/javaworld/jw-07-1996/jw-07-javascript.html
http://heather.cs.ucdavis.edu/ matloff/UnixAndC/CLanguage/Debug.html

go to **2_Syntax_errors**

ID: 296

Syntax and grammar

Before it actually runs a program, SuperCollider examines the code to ensure that syntax and grammar are correct. For example, are all variable names and/or keywords spelled correctly in a program? Are statements terminated by semi-colons?

If syntax or grammar errors are found, SuperCollider writes a notification to the post window. Such messages are descriptive but terse.

```
• ERROR: Parse error
in file 'selected text'
line 1 char 2 :
4,•
-----------------------------------
• ERROR: Command line parse failed
nil
```

//

Common errors

1. the name of a class or a variable is mispelled
2. a variable is used before being declared.
3. a parenthesis or a square or curly brace is missing or used in the wrong context
4. a required comma or semicolon is missing or used improperly

//

Two helpful commands in the SuperCollider Edit menu:

1. "Go to Line ..." transports you to the line number of your choice. Use this when an error message identifies the line number on which a problem occured.

2. "Find" searches for words or phrases. Use "Find" to locate code that has been identified in error messages or to replace all instances of an improperly spelled word.

//

go to **3_Runtime_errors**

ID: 297

Runtime errors

Runtime errors occur while a program is executing.

//

Common errors

1. an object receives a message which it doesn't understand
2. a binary operation (addition, subtraction, multiplication, etc.) can't be performed
3. a value other than true or false appears in a conditional (boolean) test
4. a file can't be opened (a primitive fails)

//

Object doesn't understand

In the case of

3.createRuntimeError

SuperCollider prints a four-part error notification to the post window. The parts of the notification are ERROR, RECEIVER, ARGS, and CALL STACK, as in

```
ERROR: Message 'createRuntimeError' not understood.
RECEIVER:
Integer 3
ARGS:
Instance of Array {    (057E7560, gc=01, fmt=01, flg=11, set=00)
indexed slots [0]
}
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of DoesNotUnderstandError>
Nil-handleError
arg this = nil
arg error = <instance of DoesNotUnderstandError>
Object-throw
```

```
arg this = <instance of DoesNotUnderstandError>
Object-doesNotUnderstand
arg this = 3
arg selector = 'createRuntimeError'
arg args = [*0]
< closed FunctionDef >  (no arguments or variables)
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
Process-interpretPrintCmdLine
arg this = <instance of Main>
```

//

The ERROR section explains what went wrong. The RECEIVER section names the the class of the object to which the message was sent. The ARGS section says how many arguments were included in the message. Read the CALL STACK from the bottom to the top to see where the error happened. Reading from bottom to top means going from

Process-interpretPrintCmdLine

to

Interpreter-interpretPrintCmdLine

to

Object-doesNotUnderstand

to

Object-throw

to

Nil-handleError

to

DoesNotUnderstandError

which is the first line in the stack.

//

DoesNotUnderstandError

is the mechanism that prints the error notification to the post window. Select it and press cmd-j to see how it works (how it prints the notification).

//

Execute

$a * $b

to create another runtime error message.

//

The ERROR, RECEIVER, ARGS, and CALL STACK headers in the post window explain the problem: Instances of class Char have no knowledge of multiplication.

ERROR: Message '*' not understood.
RECEIVER:
Character 97 'a'
ARGS:
Instance of Array { (067F5470, gc=C4, fmt=01, flg=00, set=01)
indexed slots [1]
0 : Character 98 'b'
}
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of DoesNotUnderstandError>
Nil-handleError
arg this = nil
arg error = <instance of DoesNotUnderstandError>
Object-throw
arg this = <instance of DoesNotUnderstandError>

```
Object-doesNotUnderstand
arg this = $a
arg selector = '*'
arg args = [*1]
< closed FunctionDef >  (no arguments or variables)
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
Process-interpretPrintCmdLine
arg this = <instance of Main>
```

//

Unitialized variable (binary operation fails)

Here, the variable a is initialized to an integer and the variable b isn't initialized. Multiplying a (the integer 10) by b (nil, the value that SuperCollider uses for unitialized data) will create a run-time error.

```
(
var        // a is declared and initialized
var    // b declared but not initialized, so it defaults to nil

t = Task({

4.do({ arg item, i;

if(i != 3)
   { i.postln }    // print the value of i if it doesn't equal 3
   { (a * b).postln }; // when i equals 3, do a * b
        // ...  which is a problem if b is nil
1.wait;

})

});
t.start;
)
```

//

The printout shows the code ran successfully until the index, i, reached 3, which is when a * b happened.
The ERROR, RECEIVER, ARGS, and CALL STACK headers describe the problem.

//

```
a Task
0
1
2
ERROR: binary operator '*' failed.
RECEIVER:
nil
ARGS:
Instance of Array {    (067D92B0, gc=CC, fmt=01, flg=00, set=01)
indexed slots [2]
0 :  Integer 10
1 :  nil
}
CALL STACK:
DoesNotUnderstandError-reportError
arg this = <instance of BinaryOpFailureError>
Nil-handleError
arg this = nil
arg error = <instance of BinaryOpFailureError>
Object-throw
arg this = <instance of BinaryOpFailureError>
Object-performBinaryOpOnSomething
arg this = nil
arg aSelector = '*'
arg thing = 10
arg adverb = nil
Integer-*
arg this = 10
arg aNumber = nil
arg adverb = nil
< FunctionDef in closed FunctionDef >
arg item = 3
arg i = 3
```

```
Integer-do
arg this = 4
arg function = <instance of Function>
var i = 3
< FunctionDef in closed FunctionDef >  (no arguments or variables)
Routine-prStart
arg this = <instance of Routine>
arg inval = 758.000000
```

//

True, false, or other

A value other than true or false in a boolean test, as in

```
if(x=4) { "this is ok"};
```

produces

```
ERROR: Non Boolean in test.
RECEIVER:
Integer 4
CALL STACK:
MethodError-reportError
arg this = <instance of MustBeBooleanError>
Nil-handleError
arg this = nil
arg error = <instance of MustBeBooleanError>
Object-throw
arg this = <instance of MustBeBooleanError>
Object-mustBeBoolean
arg this = 4
< closed FunctionDef >  (no arguments or variables)
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
Process-interpretPrintCmdLine
arg this = <instance of Main>
```

//

Correcting the test clause fixes the problem.

```
if(x==4) { "this is ok"};
```

//

Primitive fails

Asking for the length of a non-existent file creates a runtime error. The notification shows what went wrong (a C code primitive failed).

```
    File "i_don't_exist"  "r"
f.length;

ERROR: Primitive '_FileLength' failed.
Failed.
RECEIVER:
Instance of File {    (067D9970, gc=C4, fmt=00, flg=00, set=01)
instance variables [1]
fileptr :  nil
}
CALL STACK:
MethodError-reportError
arg this = <instance of PrimitiveFailedError>
Nil-handleError
arg this = nil
arg error = <instance of PrimitiveFailedError>
Object-throw
arg this = <instance of PrimitiveFailedError>
Object-primitiveFailed
arg this = <instance of File>
File-length
arg this = <instance of File>
< closed FunctionDef >  (no arguments or variables)
Interpreter-interpretPrintCmdLine
arg this = <instance of Interpreter>
var res = nil
var func = <instance of Function>
```

```
Process-interpretPrintCmdLine
arg this = <instance of Main>
```

//

15.2 First_steps

ID: 298

To begin

Navigate to the folder (the directory) in which SuperCollider resides and double-click on it (the red and white balloon icon). An untitled document with text such as

```
init_OSC
compiling class library..
NumPrimitives = 548
compiling dir:  '/Applications/SuperCollider3/SCClassLibrary'
pass 1 done
Method Table Size 3091264 bytes
Number of Method Selectors 2880
Number of Classes 1744
Number of Symbols 6926
Byte Code Size 129989
compiled 299 files in 1.61 seconds
compile done
prGetHostByName hostname 127.0.0.1 addr 2130706433
RESULT = 256
Class tree inited in 0.09 seconds
```

appears in the top left of the screen. The document functions as a "Post Window," so-called because SuperCollider uses it to "post" notifications.

//

Two more windows

On the bottom of the screen are two more windows. One is called "localhost server" and the other is "internal server." Click on the "boot" button on the localhost server. The words "localhost" in the black box of the button turn red and the word "Boot" on the button changes to "Quit." More text, such as

```
booting 57110
SC_AudioDriver:  numSamples=512, sampleRate=44100.000000
start   UseSeparateIO?:  0
PublishPortToRendezvous 0 57110
SuperCollider 3 server ready..
notification is on
```

will print to the post window. The localhost server is now ready to be used. Activate the internal server, if you wish, in the same way.

//

Workspace windows

Open existing SC documents with File->Open... or cmd-o. Use File->New or cmd-n to create new documents.

SuperCollider documents generally have .sc appended to their file names; however, SuperCollider can read and write documents in Rich Text Format (.rtf) and several other formats, as well.

//

go to **2_Evaluating_code**

ID: 299

Synthesizing sound

To run (evaluate) one line of code, such as

```
{ SinOsc.ar([400, 401], 0, 0.1) * Saw.ar([11, 33], 1) * EnvGen.kr(Env.sine(10)) }.play
```

first make sure that the localhost server is booted. Then put the cursor anywhere on the line (shown above) and press <enter>. The server will synthesize audio and text that looks something like

```
Synth("-613784702" :  1000)
```

will appear in the post window.

Press command-period (cmd-.) to stop synthesis.

//

To run more than one line of code, select all the lines and press <enter>.

To help with the selection process, examples with more than one line often are placed in enclosing parentheses. In such cases, select the text by clicking immediately to the right of the top parenthesis or to the left of the bottom parenthesis. Or, with the cursor to the right of the top parenthesis or the left of the bottom one, press cmd-shift-b.

Then press enter (to run the example).

```
(
{
RLPF.ar(
in:  Saw.ar([100, 102], 0.15),
freq:  Lag.kr(LFNoise0.kr(4, 700, 1100), 0.1),
rq:  0.05
)
}.play
)
```

The server will synthesize audio and text that looks something like

```
Synth("-393573063" :  1000)
```

will appear in the post window.

Press command-period (cmd-.) to stop synthesis.

//

Scoping sound

To scope whatever it is you're synthesizing (create a graphical display of the waveform):

1. make sure the internal server is running (press its boot button)
2. press the default button on the internal server window.
3. evaluate your code as described above.

For example, run

```
{ SinOsc.ar([400, 401], 0, 0.5) * Saw.ar([11, 33], 0.5) }.play
```

4. then evaluate

```
s.scope(2)
```

which will produce a window with the title of "stethoscope."

As a shortcut to steps 2 through 4, send the scope message directly to the example.

```
{ SinOsc.ar([400, 401], 0, 0.5) * Saw.ar([11, 33], 0.5) }.scope(2)
```

Press cmd-. to stop sound synthesis.

//

Recording sound

The localhost and the internal server windows have buttons, on the far right, to activate recording. To record, choose the a default server. The button on the default server of your choice intially will say "prepare rec." Press it once and it will say record >. Press it again to start recording.

//

go to **3_Comments**

ID: 300

Comments

Comments are descriptive remarks that are meant to be read by humans but ignored by computers. Programmers use comments to annotate how code works or what it does. It's also the case that some find it helpful to write programs by first notating comments and then filling in matching code.

//

To write a comment in SuperCollider, either precede text with

```
//
```

as in

```
// Everything up to the end of the line is a comment
```

or place text on one or more lines between

```
/* and */
```

as in

```
/*

This
is
a
comment

 */
```

If (when) evaluated, a comment will return nil, which is the value SuperCollider uses for unitialized data.

//

Use Format->Syntax Colorize (or cmd-') to syntax-colorize comments.

//

go to **4_Help**

ID: 301

Help

SuperCollider has a built-in help system. To see the main help page, press cmd-shift-? (without first selecting anything). From that page, double-click on topics which you'd like to see a help file and press cmd-shift-?. Another useful document is More-On-Getting-Help.

In general, there are help files for classes (capitalized words, such as SinOsc, Array, Nil, etc.). Select the name of a class and press cmd-shift-?. A help file, if one exists, will open.

//

To see every SuperCollider helpfile

evaluate

```
Help.all
```

//

To see all unit generators helpfiles

evaluate

```
Help "Help/UGens"            "SuperCollider UGens (Unit Generators)"
```

Each line of text in the document that appears contains a word to which is appended the suffix ".help". Double-click any of the words to select them and press cmd-shift-? to open the corresponding help file. Omit the ".help" appended to the word when double-clicking

//

Class definitions, message implementations, and the Find command

To see source code for class definitions, select the name of a class and type cmd-j

To see how a class or classes implement a particular message, select the message name and press cmd-y.

Use the Find and Find Next commands, available through the Edit menu, to search for text in the front-most document

//

grep

Use grep in the Terminal (in the Applications->Utilities folder) to search for all occurences of a given word or phrase. For example, to see all documents that use the LFSaw class, evaluate (in the Terminal application)

```
grep -r LFSaw /Applications/SuperCollider_f
```

Because lines in the terminal application break according to the size of the window and not through schemes that enhance readability, it may be easier to write grep results to a file, as in

```
// change the name of the path (the argument after the '>' sign, as appropriate
grep -r LFSaw /Applications/SuperCollider_f/ > /Users/yourHomeDirectory/Desktop/grep_results
```

//

Additional sources

The SuperCollider wiki: http://swiki.hfbk-hamburg.de:8888/MusicTechnology/6

The SuperCollider users mailing list archive: http://swiki.hfbk-hamburg.de:8888/MusicTechnology/437

The SuperCollider user or developer lists (or both).

http://www.create.ucsb.edu/mailman/listinfo/sc-users
http://www.create.ucsb.edu/mailman/listinfo/sc-dev

David Cottle wrote an extensive SC2 tutorial which he is now updating for SC3.

An introductory course by Nick Collins and Fredrik Olofsson:

http://www.sicklincoln.org/code/sc3tutorial.tar.gz

The pseudonym tutorial: http://www.psi-o.net/pseudonym/

The MAT tutorial (UC-Santa Barbara) tutorial: http://www.mat.ucsb.edu/ sc/

//

15.3 Miscellanea

ID: 302

For SuperCollider 3

//

First_steps

1_Startup
2_Evaluating_code
3_Comments
4_Help

//

Synthesis

1_The_network
2_Prerequisites
3_SynthDefs
4_Rates
5_Buses
6_Controls
7_Test_functions
8_UnaryOp_synthesis
9_BinaryOp_synthesis
10_Subtractive_synthesis
11_Groups
12_Playbuf
13_Delays_reverbs
14_Frequency_modulation
15_Scheduling

Please see the Japanese translation by Shigeru Kobayashi

//

Debugging

1_Debugging

2_Syntax_errors
3_Runtime_errors

//

last revised: August 2, 2004

//

Mark Polishook
polishoo@cwu.edu

15.4 Synthesis

ID: 303

Filtering

The basic idea of subtractive synthesis is similar to making coffee: something goes through a filter to remove unwanted components from the final product.

//

The .dumpClassSubtree message

Get a list of ugen filters in SuperCollider 3, by sending the .dumpClassSubtree message to the Filter class, as in

```
Filter.dumpClassSubtree;

(Object.dumpClassSubtree prints all SuperCollider classes)
```

//

The list of Filters, as of 19.5.04, includes

```
[
DetectSilence
Formlet
Ringz
SOS
FOS
Slew
Median
LPZ2
[ BRZ2 BPZ2 HPZ2 ]
Slope
LPZ1
[ HPZ1 ]
MidEQ
BPF
[ BRF ]
LPF
[ HPF ]
```

```
RLPF
[ RHPF ]
LeakDC
Lag
[ Ramp Lag3 Lag2 ]
Decay2
Decay
Integrator
TwoPole
[ APF TwoZero ]
OnePole
[ OneZero ]
Resonz
]
```

Look in Help/UGens/Filters in the SuperCollider help system to see filter help files and numerous examples.

//

Use LPF, a low-pass filter to subtract high-frequency content from an input source.

```
(
SynthDef "subtractive"
Out.ar(
0,
LPF.ar(
   Pulse                    // the source to be filtered
   Line                   // control the filter frequency with a line
)
)
}).load(s);
)

Synth("subtractive")
```

//

RLPF, a resonant low-pass filter, removes high-frequency content and emphasizes the cutoff frequency.

```
(
SynthDef "passLowFreqs2"
Out.ar(
0,
RLPF.ar(
Saw.ar([220, 221] + LFNoise0.kr(1, 100, 200), 0.2),
[LFNoise0.kr(4, 600, 2400), LFNoise0.kr(3, 600, 2400)],
0.1
)
)
}).load(s);
)

Synth "passLowFreqs2"

////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Resonz is a very, very, very strong filter. Use it to emphasize a frequency band.

Transform noise into pitch with a sharp cutoff.

```
(
SynthDef("noiseToPitch", { arg out = 0, mul = 1;
Out.ar(
out,
Resonz.ar(
WhiteNoise.ar(mul),
LFNoise0.kr(4, 110, 660),
[0.005, 0.005]
)
)
}).load(s);
)

(
// activate left and right channels
Synth("noiseToPitch", [\out, 0, \mul, 1]);
Synth("noiseToPitch", [\out, 1, \mul, 1]);
)
```

//

go to **11_Compound_synthesis**

ID: 304

The simplest synthesis processes use a single ugen.

```
{ Saw.ar(500, 0.1) }.scope;
```

or

```
{ Formlet.ar(Saw.ar(22), 400, 0.01, 0.11, 0.022) }.scope
```

Most of the SuperCollider help documents for the UGens show other such examples. Evaluate the following line to see a list of all UGen help files.

```
Help "Help/UGens"            "SuperCollider UGens (Unit Generators)"
```

//

Many synthesis processes, because they use more than a few ugens, are often best divided into component parts. This can make code modular, resusable, and easier to read.

The Group class, which is the means to specify a collection of nodes, provides a mechanism through which to control several synths at once.

//

Groups are linked lists

The important technical feature of groups is that the nodes they contain are items in a linked list. A linked list is a data structure that makes it easy to order and reorder nodes. The first item in a linked list is the "head" and the last item is the "tail."

Groups, through their head and tail mechanisms, allow synths to be placed in order so one synth verifiably executes before another, eg, the head synth runs before the tail synth. The ability to order synths is essential when sending source audio through an effect, such as a reverb or a filter.

Another feature of groups is they allow synths to receive messages from a single point of control, eg, one message to the group goes to all of nodes that belong to the group.

//

Nodes, linked lists, trees

See the Server-Architecture document for a definition of a node in SuperCollider or look to the Wikipedia for a general discussion of nodes, linked lists, and trees.

http://en.wikipedia.org/wiki/Node
http://en.wikipedia.org/wiki/Linked_list
http://en.wikipedia.org/wiki/Tree_data_structure

//

RootNode and default_group

By default, the localhost and internal servers each boot with two predefined groups: the RootNode and the default_group (see their help files). To see this, start the localhost server and then evaluate

```
s.queryAllNodes;
```

The next two lines

```
Group(0)
Group(1)
```

will appear in the transcript window.

Group(0) is the rootnode group and Group(1) is the default_group. Group(1) is indented to show that it branches from Group(0).

//

New synths are attached by default to the head of the default_group.

```
// 1st, evaluate a synthdef
(
SynthDef "ringModulation"
Out.ar(
0,
Mix.ar(
SinOsc.ar([440.067, 441.013], 0, 1)
```

```
*
SinOsc.ar([111, 109], 0, 0.2)
)
)
}).load(s);
)

// 2nd, make a synth
(
Synth "ringModulation"
)

// 3rd, tell the server to list its nodes
(
s.queryAllNodes;
)

Group(0)
Group(1)
Synth 1003
```

will appear in the transcript window. It shows Group(0) as the rootnode, Group(1) as the branching default_node and Synth 1003 (or some such number) as a leaf attached to the default_node.

```
Rootnode - Group(0)
|
|
default_node - Group(1)
/
/
Synth 1003

////////////////////////////////////////////////////////////////////////////////////////////////////
```

An example with two synths.

```
// 1st, evaluate a synthdef
(
SynthDef "pitchFromNoise"    arg
Out.ar(
```

```
out,
Resonz.ar(
WhiteNoise.ar(15),
LFNoise0.kr(2, 110, 660),
0.005
)
)
}).load(s);
)

// 2nd, make 2 synths
(
Synth "ringModulation"
Synth "pitchFromNoise"   \out
)

// 3rd, tell the server to list its nodes
(
s.queryAllNodes;
)
```

The printout in the transcript window

```
Group(0)
Group(1)
Synth 1005
Synth 1004
```

shows that Group(0) is the rootnode and Group(1) is the default_node.

Synth 1005 and 1004 (or similar such numbers) are leaves attached to the default_node. Synth 1005 is first in the list because of the way nodes are attached, by default, to the head of a list: Synth 1004, the "ringModulation" synth, was evaluated first and attached to the head of Group(1). Then Synth 1005, the "pitchFromNoise"s synth, was evaluated and placed at the head of the list (in front of Synth 1004).

```
Rootnode - Group(0)
|
|
default_node - Group(1)
```

```
/     \
/       \
Synth 1005  Synth 1004
(head)    (tail)
```

//

It's the responsibility of the user to make sure that nodes on the server are ordered properly. For this reason, the two synths below must be evaluated in the order in which they're given - because the first synth is source material for the second synth, a filter that processes its input.

```
(
SynthDef "firstNode-source"
Out.ar(
0,
Saw.ar([200, 201], 0.05)
)
}).load(s);

SynthDef "secondNode-filter"
 ReplaceOut
0,
LPF.ar(
In.ar(0, 2),
Lag.kr(
LFNoise0.kr([4, 4.001], 500, 1000),
0.1
)
)
)
}).load(s);
)

// evaluate "secondNode-filter" first
// "firstNode-source" will go at the head of default_node
(
Synth "secondNode-filter"
Synth "firstNode-source"
)
```

```
(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Or, use .head and .tail messages to attach the the nodes to the default_group).

```
(
Synth.head    "firstNode-source"
Synth.tail    "secondNode-filter"
)

(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Or, assign the synths to groups.

```
(
          Group          // attach the group to the head of the default_node
          Group          // attach the group to the tail of the default_node
)

(
// add the synths to the appropriate groups
Synth               "secondNode-filter"
Synth               "firstNode-source"
)
```

The idea is that the groups are attached first to the default_group in the desired order. The synths can then be evaluated in any order as long as they're attached to the appropriate group.

```
// run the code to see a diagram of the nodes
(
s.queryAllNodes;
)
```

```
Rootnode
|
|
default_node
/\
/  \
Group    Group
|         |
|         |
Synth     Synth
```

//

Set a control for all of the synths in a group.

```
// each of the synthdefs below has a control for amplitude (mul)
(
// build 3 synthdefs and a group
SynthDef("synthNumber1", { arg mul = 0.2;
Out.ar(
0,
BrownNoise.ar(mul) * LFNoise0.kr([1, 1.01])
)
}, [0.1]).load(s);
SynthDef("synthNumber2", { arg mul = 0.2;
Out.ar(
0,
WhiteNoise.ar(mul) * LFNoise1.kr([2.99, 3])
)
}, [0.1]).load(s);
SynthDef("synthNumber3", { arg mul = 0.2;
Out.ar(
0,
PinkNoise.ar(mul) * LFNoise2.kr([0.79, 0.67])
)
}, [0.1]).load(s);
)

(
// make a group
```

```
myGroup = Group.new;
)

(
// attach 3 synths
Synth.head( myGroup, "synthNumber1");
Synth.head( myGroup, "synthNumber2");
Synth.head( myGroup, "synthNumber3");
)

// set the \mul control of each of the 3 synths in the group
myGroup.set(\mul, 0.01.rrand(0.2))

// execute to see a diagram of the nodes
(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

go to **12_Playbuf**

ID: 305

Breaking synthesis processes into parts that accomplish small well-defined tasks encourages modular design and component reuse (the oop mantra).

```
(
// read a soundfile from disk
    Buffer          "sounds/a11wlk01.wav"

// a samplePlayer in mono ...  one channel only
SynthDef("aMonoSamplePlayer", { arg bus = 0, bufnum = 0, rateScale = 1;
Out.ar(
bus,
  PlayBuf
1,
bufnum,
BufRateScale.kr(bufnum) * rateScale
)
*
EnvGen.kr(Env.sine(BufDur.kr(bufnum)))
)
}).load(s);
)

(
// test the synthdef ...  does it work?  (yes, it's fine.  it plays on the left channel)
Synth("aMonoSamplePlayer", [\bus, 0, \bufNum, b.bufnum]);
)


(
// a simple example of component reuse ...  use the \bus argument to assign synths built from
// the same synthdef to different channels
// in this case, play a 1-channel soundfile on 2 channels
// a different playback rate for each channel makes the effect more obvious
Synth("aMonoSamplePlayer", [\bus, 0, \bufNum, b.bufnum, \rateScale, 0.99]);
Synth("aMonoSamplePlayer", [\bus, 1, \bufNum, b.bufnum, \rateScale, 1.01])
)

////////////////////////////////////////////////////////////////////////////////////////////////
```

Information

The BufRateScale and the BufDur ugens, as shown in the previous example, control the rate at which PlayBuf plays the soundfile and the length of the envelope applied to the playbuf.

BufRateScale and BufDur are of a family of ugens that inherit from InfoUGenBase or BufInfoUGenBase.

To see the complete list of such ugens, evaluate

```
InfoUGenBase.dumpClassSubtree
```

It returns

```
InfoUGenBase
[
NumRunningSynths
NumBuffers
NumControlBuses
NumAudioBuses
NumInputBuses
NumOutputBuses
ControlRate
RadiansPerSample
SampleDur
SampleRate
]
InfoUGenBase
```

Evaluate

```
BufInfoUGenBase.dumpClassSubtree
```

and it returns

```
BufInfoUGenBase
[
BufChannels
BufDur
BufSamples
```

```
BufFrames
BufRateScale
BufSampleRate
]
BufInfoUGenBase
```

//

Loop a sample

The next example uses three synthsdefs to make a chain. The first synthdef is a sample player that loops through a buffer. The second synthdef ring modulates its input. The third synthdef applies a lowpass filter.

```
(
// read a soundfile
    Buffer          "sounds/a11wlk01.wav"

// define a sample player that will loop over a soundfile
SynthDef("aLoopingSamplePlayer", { arg outBus = 0, bufnum = 0, rateScale = 1, mul = 1;
Out.ar(
outBus,
  PlayBuf
1,
bufnum,
BufRateScale.kr(bufnum) * rateScale + LFNoise1.kr(2.reciprocal, 0.05),
   loop:  1    // play the soundfile over and over without stopping
)
*
mul
)
}).load(s);

// apply amplitude modulation to an audio source
SynthDef("ampMod", { arg inBus = 0, outBus = 0, modFreq = 1;
Out.ar(
outBus,
  [ // In.ar ugen reads from an audio bus
In.ar(inBus, 1) * SinOsc.kr(modFreq),
In.ar(inBus, 1) * SinOsc.kr(modFreq - 0.02)
```

```
]
)
}).load(s);

// apply a low pass filter to an audio source
SynthDef("aLowPassFilter", { arg inBus = 0, outBus = 0, freq = 300, freqDev = 50, boost = 1;
Out.ar(
outBus,
RLPF.ar(
In.ar(inBus, 2),
Lag.kr(LFNoise0.kr(1, freqDev, freq), 1),
0.2
)
*
boost
*
LFPulse.kr(1, [0.25, 0.75], [0.5, 0.45])
+
In.ar(inBus, 2)
)
}).load(s);
)

// define 2 groups, 1 for source material and the other for effects
(
source = Group.head(s);
effect = Group.tail( s);
)

(
// add the samplePlayer to the source group
theSource = Synth.head(
source,
"aLoopingSamplePlayer", [\outBus, 3, \bufNum, b.bufnum, \rateScale, 1, \mul, 0.051]);
// add an amplitude modulation synth to the head of the effects group
fx1 = Synth.head(
effect,
 "ampMod"   \inBus     \outBus     \modFreq
// add filtering to the tail of the effects group
fx2 = Synth.tail(
```

```
effect,
"aLowPassFilter", [\inBus, 5, \outBus, 0, \boost, 5])
)

// examine the nodes
(
s.queryAllNodes;
)

// a diagram

RootNode
|
default_node
/\
/  \
                    //  source and  effects are groups
|         |       \
|         |       \
synth    synth    synth


// Changing argument (control) values effects timbre
(
theSource.set(\rateScale, 0.95.rrand(1.05), \mul, 0.051.rrand(0.07));
fx1.set(\modFreq, 800.0.rrand(1200));
fx2.set(\freq, 500.rrand(700), \freqDev, 180.rrand(210), \boost, 7);
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

go to **13_Delays_reverbs**

ID: 306

Time-based filters

The Delay, Comb, and Allpass family of ugens create time-based effects to give a sense of location and space.

```
////////////////////////////////////////////////////////////////////////////////////////////////

// 2 synthdefs - the 1st to make grains and the 2nd to delay them

// the synthdef that makes the grains is on the left channel
// the synthdef that delays the grains is on the right channel
(
SynthDef("someGrains", { arg centerFreq = 777, freqDev = 200, grainFreq = 2;
var gate;
gate = Impulse.kr(grainFreq);
Out.ar(
0,
SinOsc.ar(
LFNoise0.kr(4, freqDev, centerFreq),
0,
EnvGen.kr(Env.sine(0.1), gate, 0.1)
)
)
}).load(s);

SynthDef("aDelay", { arg delay = 0.25;
Out.ar(
1,
DelayN.ar(
In.ar(0, 1),
delay,
delay
)
)
}).load(s);
)

////////////////////////////////////////////////
```

```
// test the grains ...  and then turn them off
// ...  they're all on the left channel ...  good!
Synth "someGrains"
////////////////////////////////////////////////

// make 2 groups, the 1st for sources and the 2nd for effects
(
source = Group.head(s);
effects = Group.tail(s);
)

// place grains into the delay ...  source is on the left and delayed source is on the right
(
Synth.head( source, "someGrains");
Synth.head( effects, "aDelay");
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Feedback filters

Comb and Allpass filters are examples of ugens that feed some of their output back into their input. Allpass filters change the phase of signals passed through them. For this reason, they're useful even though don't seeem to differ much from comb filters.

```
////////////////////////////////////////////////////////////////////////////////
// TURN ON THE INTERNAL SERVER!!
// first a comb filter and then an allpass with (with the same parameters) - compare them
////////////////////////////////////////////////////////////////////////////////

// comb example
(
{
CombN.ar(
SinOsc.ar(500.rrand(1000), 0, 0.2) * Line.kr(1, 0, 0.1),
0.3,
0.25,
6
)
```

```
}.scope;
)

// allpass example - not much difference from the comb example
(
{
 AllpassN
SinOsc.ar(500.rrand(1000), 0, 0.2) * Line.kr(1, 0, 0.1),
0.3,
0.25,
6
)
}.scope;
)

////////////////////////////////////////////////////////////////////////////////
//
// first a comb example and then an allpass
// both examples have the same parameters
// the 2 examples have relatively short delay times ...  0.1 seconds
//
////////////////////////////////////////////////////////////////////////////////


// comb
(
{
CombN.ar(
SinOsc.ar(500.rrand(1000), 0, 0.2) * Line.kr(1, 0, 0.1),
0.1,
0.025,
6
)
}.scope;
)

// allpass ...  what's the difference between this example and the comb filter?
(
{
 AllpassN
```

```
SinOsc.ar(500.rrand(1000), 0, 0.2) * Line.kr(1, 0, 0.1),
0.1,
0.025,
6
)
}.scope
)
```

//

Reverberation

The next example is by James McCartney. It comes from the 01 Why SuperCollider document that was part of the SuperCollider2 distribution.

The example is more or less a Schroeder reverb - a signal passed through a parallel bank of comb filters which then pass through a series of allpass filters.

```
(
{
var s, z, y;
 // 10 voices of a random sine percussion sound :
s = Mix.ar(Array.fill(10, { Resonz.ar(Dust.ar(0.2, 50), 200 + 3000.0.rand, 0.003)}) );
 // reverb predelay time :
z = DelayN.ar(s, 0.048);
 // 7 length modulated comb delays in parallel :
y = Mix.ar(Array.fill(7,{ CombL.ar(z, 0.1, LFNoise1.kr(0.1.rand, 0.04, 0.05), 15) }));
 // two parallel chains of 4 allpass delays (8 total) :
4.do({ y = AllpassN.ar(y, 0.050, [0.050.rand, 0.050.rand], 1) });
 // add original sound to reverb and play it :
s+(0.2*y)
}.scope
)
```

//

Components

The following shows one way to divide the JMC example into components.

```
(
SynthDef "filteredDust"
Out.ar(
2,
Mix.arFill(10, { Resonz.ar(Dust.ar(0.2, 50), Rand(200, 3200), 0.003) })
)
}).load(s);

SynthDef "preDelay"
 ReplaceOut
4,
DelayN.ar(In.ar(2, 1), 0.048, 0.048)
)
}).load(s);

SynthDef "combs"
 ReplaceOut
6,
Mix.arFill(7, { CombL.ar(In.ar(4, 1), 0.1, LFNoise1.kr(Rand(0, 0.1), 0.04, 0.05), 15) })
)
}).load(s);

SynthDef("allpass", { arg gain = 0.2;
var source;
source = In.ar(6, 1);
4.do({ source = AllpassN.ar(source, 0.050, [Rand(0, 0.05), Rand(0, 0.05)], 1) });
 ReplaceOut
8,
source * gain
)
}).load(s);

SynthDef("theMixer", { arg gain = 1;
 ReplaceOut
0,
Mix.ar([In.ar(2, 1), In.ar(8, 2)]) * gain
)
}).load(s);
)
```

```
// as each line is executed, it becomes the tail node.  the result is that
// "filteredDust" is the first node and  "theMixer" is the last node ...
// ...  exactly what we need
(
Synth         "filteredDust"
Synth.tail(s, "preDelay");
Synth.tail(s, "combs");
Synth.tail(s, "allpass");
Synth.tail(s, "theMixer");
)

(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////

Or, use groups to control the order of execution.

(
source = Group.tail(s);
proc1 = Group.tail(s);
proc2 = Group.tail(s);
proc3 = Group.tail(s);
final = Group.tail(s);
)

// the nodes, below, are assigned to the groups, as ordered above,
(
Synth.head( final, "theMixer");
Synth.head( proc3, "allpass");
Synth.head( proc2, "combs");
Synth.head( proc1, "preDelay");
Synth.head( source, "filteredDust");
)

(
s.queryAllNodes;
)
```

//

For context, here, below, is the complete text of the 01 Why SuperCollider document (by James McCartney) from the SuperCollider 2 distribution.

//

´ SuperCollider 2.0

Why SuperCollider 2.0 ?

SuperCollider version 2.0 is a new programming language. **Why invent a new language**
and not use an existing language? Computer music composition is a specification problem.
Both sound synthesis and the composition of sounds are complex problems and demand a
language which is highly expressive in order to deal with that complexity. Real time signal
processing is a problem demanding an efficient implementation with bounded time operations.
There was no language combining the features I wanted and needed for doing digital music
synthesis. The SuperCollider language is most like Smalltalk. Everything is an object. It has
class objects, methods, dynamic typing, full closures, default arguments, variable length argument lists, multiple assignment, etc. The implementation provides fast, constant time method lookup, real time garbage collection, and stack allocation of most function contexts while maintaining full closure semantics.
The SuperCollider virtual machine is designed so that it can be run at interrupt level. There was no other language readily available that was high level, real time and capable of running at interrupt level.

SuperCollider version 1.0 was completely rewritten to make it both more expressive and more efficient. This required rethinking the implementation in light of the experience of the first version. It is my opinion that the new version has benefitted significantly from this rethink. It is not simply version 1.0 with more features.

Why use a text based language rather than a graphical language?

There are at least two answers to this. **Dynamism:** Most graphical synthesis environments
use statically allocated unit generators. In SuperCollider, the user can create structures which
spawn events dynamically and in a nested fashion. Patches can be built dynamically and parameterized not just by floating point numbers from a static score, but by other graphs of unit generators as well. Or you can construct patches algorithmically on the fly.
This kind of fluidity is not possible in a language with statically allocated unit generators.
Brevity: In SuperCollider, symmetries in a patch can be exploited by either multichannel expansion or programmatic patch building. For example, the following short program generates a patch of 49 unit generators. In a graphical program this might require a significant
amount of time and space to wire up. Another advantage is that the size of the patch below can
be easily expanded or contracted just by changing a few constants.

```
(
{
 // 10 voices of a random sine percussion sound :
s = Mix.ar(Array.fill(10, { Resonz.ar(Dust.ar(0.2, 50), 200 + 3000.0.rand, 0.003)}) );
 // reverb predelay time :
z = DelayN.ar(s, 0.048);
 // 7 length modulated comb delays in parallel :
y = Mix.ar(Array.fill(7,{ CombL.ar(z, 0.1, LFNoise1.kr(0.1.rand, 0.04, 0.05), 15) }));
 // two parallel chains of 4 allpass delays (8 total) :
4.do({ y = AllpassN.ar(y, 0.050, [0.050.rand, 0.050.rand], 1) });
 // add original sound to reverb and play it :
s+(0.2*y)
}.play )
```

Graphical synthesis environments are becoming a dime a dozen. It seems like a new one is announced every month. None of them have the dynamic flexibility of SuperCollider's complete programming environment. Look through the SuperCollider help files and examples
and see for yourself.

//

go to **14_Frequency_modulation**

ID: 307

Carriers and modulators

In its simplest form, frequency modulation (FM) synthesis - famous since the Yamaha DX7 of the 1980's - uses one oscillator to modulate the frequency of another. The modulating oscillator in FM synthesis usually runs at the audio rate and its amplitude often is shaped by an envelope or other controller, such as a low frequency oscillator.

```
(
SynthDef("fm1", { arg bus = 0, freq = 440, carPartial = 1, modPartial = 1, index = 3, mul = 0.05;

 // index values usually are between 0 and 24
 // carPartial ::  modPartial => car/mod ratio

var mod;
var car;

mod = SinOsc.ar(
freq * modPartial,
0,
freq * index * LFNoise1.kr(5.reciprocal).abs
);

car = SinOsc.ar(
(freq * carPartial) + mod,
0,
mul
);

Out.ar(
bus,
car
)
}).load(s);
)

(
Synth "fm1"   \bus     \freq       \carPartial     \modPartial
Synth("fm1", [\bus, 1, \freq, 442, \carPartial, 1, \modPartial, 2.401]);
```

```
)

(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

FM synthesis and reverb

```
// ...  a reverb adapted from the "01 Why SuperCollider document" in the SC2 distribution
(
SynthDef("preDelay", { arg inbus = 2;
 ReplaceOut
4,
DelayN.ar(In.ar(inbus, 1), 0.048, 0.048)
)
}).load(s);

SynthDef "combs"
 ReplaceOut
6,
Mix.arFill(7, { CombL.ar(In.ar(4, 1), 0.1, LFNoise1.kr(Rand(0, 0.1), 0.04, 0.05), 15) })
)
}).load(s);

SynthDef("allpass", { arg gain = 0.2;
var source;
source = In.ar(6, 1);
4.do({ source = AllpassN.ar(source, 0.050, [Rand(0, 0.05), Rand(0, 0.05)], 1) });
 ReplaceOut
8,
source * gain
)
}).load(s);

SynthDef("theMixer", { arg gain = 1;
 ReplaceOut
0,
```

```
Mix.ar([In.ar(2, 1), In.ar(8, 2)]) * gain
)
}).load(s);
)

(
Synth("fm1", [\bus, 2, \freq, 440, \carPartial, 1, \modPartial, 1.99, \mul, 0.071]);
Synth("fm1", [\bus, 2, \freq, 442, \carPartial, 1, \modPartial, 2.401, \mul, 0.071]);
Synth.tail(s, "preDelay");
Synth.tail(s, "combs");
Synth.tail(s, "allpass");
Synth.tail(s, "theMixer", [\gain, 0.64]);
)

(
s.queryAllNodes;
)
```

//

Components

Dividing the "fm" synth def into two pieces, a synthdef for a modulator and a synthdef for the carrier, gives more functionality - carrier signals can shaped by two or more modulators.

```
(
SynthDef("carrier", { arg inbus = 2, outbus = 0, freq = 440, carPartial = 1, index = 3, mul = 0.2;

 // index values usually are between 0 and 24
 // carPartial ::  modPartial => car/mod ratio

var mod;
var car;

mod = In.ar(inbus, 1);

Out.ar(
outbus,
SinOsc.ar((freq * carPartial) + mod, 0, mul);
)
```

```
}).load(s);

SynthDef("modulator", { arg outbus = 2, freq, modPartial = 1, index = 3;
Out.ar(
outbus,
SinOsc.ar(freq * modPartial, 0, freq)
*
LFNoise1.kr(Rand(3, 6).reciprocal).abs
*
index
)
}).load(s);
)

(
var freq = 440;
// modulators for the left channel
Synth.head(s, "modulator", [\outbus, 2, \freq, freq, \modPartial, 0.649, \index, 2]);
Synth.head(s, "modulator", [\outbus, 2, \freq, freq, \modPartial, 1.683, \index, 2.31]);

// modulators for the right channel
Synth.head(s, "modulator", [\outbus, 4, \freq, freq, \modPartial, 0.729, \index, 1.43]);
Synth.head(s, "modulator", [\outbus, 4, \freq, freq, \modPartial, 2.19, \index, 1.76]);

// left and right channel carriers
Synth.tail(s, "carrier", [\inbus, 2, \outbus, 0, \freq, freq, \carPartial, 1]);
Synth.tail(s, "carrier", [\inbus, 4, \outbus, 1, \freq, freq, \carPartial, 0.97]);
)

(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Reverberation and frequency modulation

```
(
var freq;
// generate a random base frequency for the carriers and the modulators
```

```
freq = 330.0.rrand(500);

// modulators for the left channel
Synth.head(s, "modulator", [\outbus, 60, \freq, freq, \modPartial, 0.649, \index, 2]);
Synth.head(s, "modulator", [\outbus, 60, \freq, freq, \modPartial, 1.683, \index, 2.31]);

// modulators for the right channel
Synth.head(s, "modulator", [\outbus, 62, \freq, freq, \modPartial, 1.11, \index, 1.43]);
Synth.head(s, "modulator", [\outbus, 62, \freq, freq, \modPartial, 0.729, \index, 1.76]);

// left and right channel carriers
Synth.tail(s, "carrier", [\inbus, 60, \outbus, 100, \freq, freq, \carPartial, 1]);
Synth.tail(s, "carrier", [\inbus, 62, \outbus, 100, \freq, freq+1, \carPartial, 2.91]);

Synth.tail(s, "preDelay", [\inbus, 100]);
Synth.tail(s, "combs");
Synth.tail(s, "allpass");
Synth.tail(s, "theMixer", [\gain, 0.2]);
)

(
s.queryAllNodes;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

go to **15_Scheduling**

ID: 308

Routines and clocks

Use clocks to create automated, algorithmic scheduling. Among the things that clocks "play" are routines, tasks, and patterns.

To see how a clock "plays" a routine, first examine how a function works in a routine.

The first argument (and usually the only argument) to a routine is a function.

```
// template for a routine
Routine   "....  code within curly braces is a function ....  "
```

A .yield message to an expression in a function (in a routine) returns a value.

```
r = Routine({ "hello, world".yield.postln });

// to evaluate a routine, send a .next message
// it will "hand over" the value of the expression to which the .yield message is attached
r.next;
```

Evaluate (again)

```
r.next;
```

The routine above returns nil when its evaluated a second time. This is because once a routine "yields" and if there's no additional code after the .yield message, the routine is finished, over, and done - unless it receives a reset message. Then it can start over again.

```
        // returns nil
         // reset the routine
        // it works!

////////////////////////////////////////////////////////////////////////////////////////////////////

(
r = Routine({
 "hello, world"
 "what a world"
```

```
 "i am a world"
});
)
```

The first three .next messages return a string. The fourth .next message returns nil.

```
        // returns a string
        // returns a string
        // returns a string
        // returns nil
```

Reset the routine.

```
r.reset;

r.next;
r.next;
r.next;
r.next;
```

//

Use a .do message in a routine to make a loop.

```
(
r = Routine({

 // setup code
var array;
 array = [ "hello, world"  "what a world"  "i am a world"

 // the loop
3.do({ array.choose.yield })

});
)
```

Evaluate the routine one more time than the loop in the routine allows.

```
4.do({ r.next.postln });
```

The routine returned three strings followed by nil.

//

Scheduling routines

Rewrite the routine so that it includes a .wait message.

```
(
r = Routine({

var array;
 array = [ "hello, world"  "what a world"  "i am a world"

3.do({
  1.wait;     // pause for 1 second
array.choose.postln;
})

});
)
```

Then "play" the routine, eg, send it a .play message.

```
r.play
```

Append a .reset message to the routine so that it can start over.

```
r.reset.play;
```

//

Clocks and the convenience of .play

When a routine receives a .play message, control (of the routine) is redirected to a clock. The clock uses the receiver of the .wait message as a unit of time to schedule ("play") the routine.

SuperCollider has three clocks, each of which has a help file.

```
SystemClock     // the most accurate
AppClock      // for use with GUIs
TempoClock     // to schedule in beats
```

The .play message is a convenience that allows one to write

```
                // reset the routine before playing it
```

instead of

```
SystemClock
```

//

Scheduling synths with routines

Enclose synths within routines. It's often the case that the synthdef used by the synth in routines should have an envelope with a doneAction parameter set to 2 (to deallocate the memory needed for the synth after its envelope has finished playing).

```
(
// DEFINE A SYNTHDEF
SynthDef "fm2"
arg bus = 0, freq = 440, carPartial = 1, modPartial = 1, index = 3, mul = 0.2, ts = 1;

 // index values usually are between 0 and 24
 // carPartial ::  modPartial => car/mod ratio

var mod;
var car;

mod = SinOsc.ar(
freq * modPartial,
0,
freq * index * LFNoise1.kr(5.reciprocal).abs
);

car = SinOsc.ar(
(freq * carPartial) + mod,
0,
```

```
mul
);

Out.ar(
bus,
car * EnvGen.kr(Env.sine(1), doneAction:  2, timeScale:  ts)
)
}).load(s);
)

(
// DEFINE A ROUTINE
    Routine

12.do({
  Synth
"fm2",
[
\bus, 2.rand, \freq, 400.0.rrand(1200),
\carPartial, 0.5.rrand(2), \ts, 0.5.rrand(11)
]
);
s.queryAllNodes;
"".postln.postln.postln.postln.postln;
2.wait;
})
});
)

// PLAY THE ROUTINE
r.reset.play;

////////////////////////////////////////////////////////////////////////////////////////////////////

Process synths spawned in a routine through effects that run outside of the routine.

(
// DEFINE A SYNTHDEF
SynthDef "echoplex"
 ReplaceOut
```

```
0,
CombN.ar(
In.ar(0, 1),
0.35,
[Rand(0.05, 0.3), Rand(0.05, 0.3)],
   // generate random values every time a synth is created
7,
0.5
)
)
}).load(s);

// DEFINE GROUPS TO CONTROL ORDER-OF-EXECUTION
// attach a  source group to the head of the rootnode and
// an  effects group to the tail of the rootenode
source = Group.head(s);
effect = Group.tail(s);

// DEFINE A ROUTINE
    Routine

 // loop is the same as inf.do, eg, create an infinite loop that runs forever
loop({
  Synth       // attach the synth to the head of the  source group
source,
"fm2",
[
\outbus, 0, \freq, 400.0.rrand(1200), \modPartial, 0.3.rrand(2.0),
\carPartial, 0.5.rrand(11), \ts, 0.1.rrand(0.2)]
);
s.queryAllNodes;
2.wait;
})
});

// TURN ON EFFECTS
Synth.head( effect, "echoplex");
Synth.tail( effect, "echoplex");
)
// PLAY THE ROUTINE
```

```
r.reset.play;

////////////////////////////////////////////////////////////////////////////////////////////////////
```

ID: 309

Networks and client/server

SuperCollider 3 uses a client/server model to operate across a network. What this means is that users write client programs that ask a server to do something, that is, they request service. Such requests can occur locally on one computer or they can be distributed remotely among two or more computers. Whether the computers are in the same room or separated across the world makes no difference as long as they're connected on a network.

Client programs in SuperCollider typically specify synthesis definition (how a particular sound will be made) and synthesis scheduling (when a particular sound will be made). In turn, a SuperCollider server (or servers) synthesizes audio according to client instructions.

To summarize, clients request; servers respond.

//

Client/server examples

```
// EX. 1 - execute each line, one at a time
// define a synthesis process and make a client request to a server
////////////////////////////////////////////////////////////////////////////////////////////////////

// define a server with a name and an address
    Server "aServer"  NetAddr "127.0.0.1"            // "localhost" is a synonym for an ip of             //
"127.0.0.1"
// start the server
s.boot;

// define a synthesis engine
SynthDef("sine", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).send(s);

// schedule (run) synthesis
s.sendMsg("s_new", "sine", n = s.nextNodeID, 0, 1);

// stop the synth (delete it)
s.sendMsg("/n_free", n);

// (optionally) stop the server
```

```
s.quit;

////////////////////////////////////////////////////////////////////////////////////////////////////
// EX. 2
// the same as in above, except on 2 computers across a network
////////////////////////////////////////////////////////////////////////////////////////////////////

// define a (remote) server; it represents a computer "somewhere" on the internet"
// the ip number has to be valid and the server, wherever it is, has to be running
// servers cannot be booted remotely, eg, a program on one machine can't boot a server on another
// this example assumes the server on the remote machine was booted from within
// supercollider and not from the terminal
    Server "aServer"  NetAddr "...  an ip number ..."

// define a synthesis engine ...  exactly as in the previous example
SynthDef("sine", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).send(s);

// schedule synthesis ...  exactly as in the previous example
s.sendMsg("s_new", "sine", n = s.nextNodeID, 0, 1);

// stop the synth (delete it)
s.sendMsg("/n_free", n);

////////////////////////////////////////////////////////////////////////////////////////////////////
// EX. 3
// client/server on one computer vs.  client server on two computers
// the previous examples without comments
// they're identical except that
// the example that runs on one computer explicitly boots the server
// the example on 2 computers _assumes the server "somewhere" on the internet is booted
////////////////////////////////////////////////////////////////////////////////////////////////////

// on one computer
    Server "aServer"  NetAddr "localhost"
s.boot;
SynthDef("sine", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).send(s);
s.sendMsg("s_new", "sine", n = s.nextNodeID, 0, 1);
s.sendMsg("/n_free", n);

vs.
```

```
// on two computers ...  the server has to have a valid ip address
    Server "aServer"  NetAddr "...  an ip number ..."
SynthDef("sine", { Out.ar(0, SinOsc.ar(440, 0, 0.2)) }).send(s);
s.sendMsg("s_new", "sine", n = s.nextNodeID, 0, 1);
s.sendMsg("/n_free", n);
```

//

Localhost and internal servers

The previous examples define server objects. But, for the most part, this isn't necessary as SuperCollider defines two such objects, the localhost and internal servers, at startup. They're represented by windows at the bottom of the screen. Each of the windows has a boot button to start its respective server.

See the ClientVsServer, Server, and ServerOptions and Tutorial documents in the SuperCollider help system for further information.

//

Go to **2_Prerequisites**

ID: 310

We know that SuperCollider applies a client/server model to audio synthesis and processing. Let's focus on synthesis definition. Some things to consider:

DSP

First, we require knowledge of digital signal processing. A reference, such as the "Computer Music Tutorial," (MIT Press) can be helpful. A source on the internet is "The Scientist and Engineer's Guide to Digital Signal Processing" at

http://www.dspguide.com/

OOP

Second, we need to know how to use the SuperCollider language to express synthesis algorithms. This means learning about object-oriented programming in general and about the grammar and syntax of the SuperCollider language in particular. A book about Smalltalk, the object-oriented computer language that SuperCollider closely resembles, can be helpful. Two books about Smalltalk on the www are

"The Art and Science of Smalltalk"
(http://www.iam.unibe.ch/ ducasse/FreeBooks/Art/artMissing186187Fix1.pdf)

and

Smalltalk by Example
(http://www.iam.unibe.ch/ ducasse/FreeBooks/ByExample/).

The SuperCollider documentation and numerous sites across the internet, such as the swiki at

http://swiki.hfbk-hamburg.de:8888/MusicTechnology/6

explain and show how the SuperCollider language works.

//

go to **3_SynthDefs**

ID: 311

SynthDefs

Use the SynthDef class to build the engines for synths that will run on the server. The engines, which can be saved to disk and reused, are analogous to presets on commercial hardware and software synthesizers.

When notated as code in client programs, the engines have two essential parts: a name and a function. In the jargon of SuperCollider, the function is called a ugenGraphFunc.

The term ugenGraphFunc derives from the notion of a graph, which is the data structure that SuperCollider uses to organize unit generators. SuperCollider constructs the graph for you from the code it finds in your function which means that don't have to know how a graph works or that it even exists.

If you wish to know more about graphs, visit the Wikipedia at

http://en.wikipedia.org/wiki/Graph_(data_structure).

Or, go to

http://www.nist.gov/dads/HTML/graph.html

//

Template

Here's a template for a synthdef showing that it consists of a name and a ugenGraphFunc

```
SynthDef
 "aSynthDef"       // the 1st argument is the name
 { ....  i am a ugenGraphFunc ...  } // the 2nd argument is the ugenGraphFunc
)
```

To make the template functional

1. put code into the ugenGraphFunc
2. send a .load message to the synthdef

```
(
```

```
SynthDef
 "aSynthDef"        // the name of the synthdef
 {         // the ugenGraphFunc with actual code
arg freq;
Out.ar(
0,
SinOsc.ar(freq, 0, 0.1)
)
}
).load(s);
)
```

//

The .load message and the variable 's'

The .load message writes synthdefs to disk and also sends them to the default server. The default server is defined by SuperCollider at startup time (as the localhost server) at which point it's also assigned to the variable 's'.

//

The .send message and a remote server

On the other hand, .send message,

```
SynthDef( ....  ).send(s);
```

instead of a .load message

```
SynthDef( ....  ).load(s);
```

is another way to get a synthdef to a server. The .send message, unlike the .load message, doesn't first write the synthdef to disk; instead it just transmits the synthdef directly to the server. This is therefore the message to use to define a synthdef on one computer but send it to another.

```
(
var aServer;
aServer =
 Server
```

```
  "aRemoteServerOnAnotherMachine"
  NetAddr "...  an ip # ..."          // a server on another computer
);
SynthDef( ....  ).send(aServer);
)
```

//

SynthDef browsers

Use the synthdef browser to examine synthdefs that have been written to disk.

```
(
// a synthdef browswer
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

The middle box (in the top row) shows the names of synthsdefs. Each name, when selected, causes the other boxes to display the ugens, controls, and inputs and outputs for a particular synthdef.

The box labeled "SynthDef controls" is useful because it shows the arguments that can be passed to a given synthdef.

The browser shows that the synthdef defined above - "aSynthDef" - is composed from four ugens, one control, no inputs, and one output. The four ugens include instances of Control, SinOsc, BinaryOpUGen, and Out classes.

The one control is "freq". A control is an argument that a synth can use when it is created or at any time while it (the synth) exists on the server. The browser also shows that "aSynth" has no inputs (which means that it doesn't use data from audio or control buses) and that it sends one channel of audio out to an audio Bus.

//

For further context, see the SynthDef, In, Out, SinOsc, Control, BinaryOpUGen files in the SuperCollider help system.

//

go to **4_Rates**

ID: 312

Audio rate

Ugens to which an .ar message is sent run at the audio rate, by default, at 44,100 samples per second. Send the .ar message to unit generators when they're part of the audio chain that will be heard.

```
SinOsc.ar(440, 0, 1);
```

//

Control rate

Ugens to which a .kr message is appended run at the control rate.

```
SinOsc.kr(440, 0, 1);
```

By default, control rate ugens generate one sample value for every sixty-four sample values made by an audio rate ugen. Control rate ugens thus use fewer resources and are less computationally expensive than their audio rate counterparts.

Use control rate ugens as modulators, that is, as signals that shape an audio signal.

//

Here, a control rate SinOsc modulates the frequency of the audio rate Pulse wave.

```
(
SynthDef "anExample"
Out.ar(
0,
Pulse.ar(
   [220, 221.5] + SinOsc                    // the control rate conserves CPU cycles
0.35,
0.02
)
)
}).load(s);
)
```

```
Synth "anExample"

Type command-period (cmd-.)  to stop synthesis.
```

//

Go to **5_Buses**

ID: 313

By default, SuperCollider has 128 buses for audio signals and 4096 for control signals. The buses, which are items in an array, are what SuperCollider uses to represent audio and control rate data.

```
////////////////////////////////////////////////////////////////////////////////////////////////////

// the array of audio buses (channels)
[ channel0, channel1, channel2, channel3, channel4, ...  , ..., ..., etc., ...  channel127 ]

// the array of control buses (channels)
[ channel0, channel1, channel2, channel3, channel4, ...  , ..., ..., etc., ...  channel4095 ]

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Placing audio into a bus

Use anOut ugen at the audio rate to put data into an audio bus.

```
(
SynthDef "dataForABus"
Out.ar(
  0,     // write 1 channel of audio into bus 0
Saw.ar(100, 0.1)
)
}).load(s);
)

Synth "dataForABus"
```

A SynthDef browser

```
(
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

shows 1 channel of output on channel 0.

//

Getting audio from a bus

Send an .ar message to an In ugen to get data from an audio bus.

```
(
SynthDef "dataFromABus"
Out.ar(
0,
  [ // the left channel gets input from an audio bus
In.ar(0, 1),
SinOsc.ar(440, 0.2),
]
)
}).load(s);
)

(
Synth "dataForABus"   // synthesize a sawtooth wave on channel 0
Synth "dataFromABus"   // pair it with a sine wave on channel 1
)
```

//

Control rate buses

Use In.kr and Out.kr to read from or write to control buses.

//

For additional information, see the Out, In, and Bus files in the SuperCollider help system.

//

go to **6_Controls**

ID: 314

Evaluate

```
(
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

and examine the box that lists the controls for each synth.

//

Controls (usually) are arguments

Use controls, which most often are defined as arguments in a ugenGraphFunc, to give information to a synth, either when it is created and/or after it is running. Supply default values to the arguments to make code more readable and to protect against user error (such as forgetting to supply a value to an argument).

```
(
// 3 arguments (controls) with default values
SynthDef
 "withControls"
{ arg freq = 440, beatFreq = 1, mul = 0.22;
Out.ar(
0,
SinOsc.ar([freq, freq+beatFreq], 0, mul)
)
}).load(s);
)

// items in the array are passed to the controls in Synth when it's created
z = Synth("withControls", [\freq, 440, \beatFreq, 1, \mul, 0.1]);

// evaluate this line after the synth is running to reset its controls
z.set(\freq, 700, \beatFreq, 2, \mul, 0.2);
```

//

Write controls names and appropriate values in the array given as an argument to a synth. Control names can be given as symbols (a unique name within the SuperCollider system).

```
Synth("withControls", [\freq, 440, \beatFreq, 0.5, \mul, 0.1]);
```

or as as strings (an array of characters)

```
Synth("withControls", ["freq", 440, "beatFreq", 0.5, "mul", 0.1]);
```

Either way, the pattern is

[controlName, value, controlName, value, controlName, value].

See the Symbol and String files in the SuperCollider help system.

//

A third way to pass controls to a synth is as

```
Synth("withControls", [0, 440, 1, 1, 2, 0.1]);
```

In this case, the pattern is

[controlIndex, value, controlIndex, value, controlIndex].

//

To adjust a control

Use the .set message to change the value of a control while a synth is running.

```
(
SynthDef("resetMyControls", { arg freq = 440, mul = 0.22;
Out.ar(
0,
SinOsc.ar([freq, freq+1], 0, mul)
)
}).load(s);
)
```

```
 aSynth = Synth("resetMyControls", [\freq, 440, \mul, 0.06]);
aSynth.set(\freq, 600, \mul, 0.25);
```

//

Global variables

The ' ' character before aSynth in the previous example defines a global variable. An advantage to using a global variable is that it doesn't have to be declared explicitly, as in

```
var        // variables without the ' ' MUST first be declared!!
```

More precisely, the character puts a variable named 'aSynth' into an instance of an object known as the currentEnvironment. For further information, see the Environment document in the SuperCollider help system.

//

Lag times

Use an array of lag times to state how long it takes to glide smoothly from one control value to another. Write the lag times in an array and place it in the synthdef after the ugenGraphFunc, as in

```
(
SynthDef("controlsWithLags", { arg freq1 = 440, freq2 = 443, mul = 0.12;
Out.ar(
0,
SinOsc.ar([freq1, freq2], 0, mul)
)
}, [4, 5]).load(s);
)

 aSynth = Synth("controlsWithLags", [\freq1, 550, \freq2, 344, \mul, 0.1]);
aSynth.set(\freq1, 600, \freq2, 701, \mul, 0.05);
```

//

SynthDef templates

The array of lagtimes means that the synthdef template with two components (discussed in 3_SynthDefs)

```
// a template for a synthdef with two components
SynthDef
 aSynth"        // 1st arsgument is a name
{ ....  i am a ugenGraphFunc ...  } // 2nd argument is a ugenGraphFunc
)
```

can be revised to include three components.

```
// a re-defined template for a synthdef _with an array of lagtimes
// the class definition for the lagtime array calls it 'rates'
SynthDef
 "aSynth"      // name
{ ....  i am a ugenGraphFunc ...  }, // ugenGraphFunc
[ ...  lagTimes ...  ]        // rates
)
```

//

go to **7_Test_functions**

ID: 315

Functions and .scope messages

An easy way to audition synthesis processes is to test them within a function. To do this, append a .scope or a .play message to a function. The .scope message, which works only with the internal server, displays a visual representation of a sound wave.

//

Boot (turn on) the internal server

```
Server.internal.boot;
```

Run this example, and look at the scope window.

```
// test a synthesis process in a function
(
{
 SinOsc.ar([440.067, 441.013], 0, 1)
*
SinOsc.ar([111, 109], 0, 0.2)
}.scope;
)
```

//

Code can be transfered from a test function into a synthdef. In the following example, the code from the function (above) is the second argument to the Out ugen.

```
(
SynthDef "ringMod"
Out.ar(
0,
SinOsc.ar([440.067, 441.013], 0, 1)
*
SinOsc.ar([111, 109], 0, 0.2)
)
}).load(s);
)
```

```
Synth "ringMod"
```

//

Multi-channel expansion

Expand a ugen to two channels with an array in any of the argument (control) slots.

```
{ Saw.ar([500, 933], 0.1) }.scope;
```

Another (longer) way to write the same thing is

```
{ [ Saw.ar(500, 0.1), Saw.ar(933, 0.1)] }.scope;
```

Expand a ugen to three channels by adding values to the array.

```
{ Saw.ar([500, 933, 2033], 0.1) }.scope;

// 4 channels

{ Saw.ar([500, 933, 2033, 895], 0.1) }.scope;
```

//

go to **8_UnaryOp_synthesis**

ID: 316

Unary messages

Some synthesis processes can be initiated with a unary message (a message with no arguments).

//

For example, compare

```
{ SinOsc.ar(500, 0, 0.5) }.scope;
```

to

```
{ SinOsc.ar(500, 0, 0.5).distort }.scope;
```

The .distort message modulates the SinOsc to create more partials.

//

Q: Where does the .distort message come from?

A: It's defined in the AbstractFunction class. The UGen class is a subclass of the AbstractFunction class. The idea is that all classes inherit methods defined in their superclasses; all ugens thus inherit from AbstractFunction).

Compare

```
{ SinOsc.ar(500, 0, 0.5) }.scope;
```

to

```
// .cubed is a unary operation
{ SinOsc.ar(500, 0, 0.5).cubed }.scope;
```

//

See the files in the UnaryOps folder in the SuperCollider help system

//

go to **9_BinaryOpSynthesis**

ID: 317

Binary messages

The pattern for a binary message is

RECEIVER OPERATOR OPERAND

For example

2 * 3

is a receiver (the object to which a message is sent), a binary operator, and an operand.

//

Mixing = addition

Use addition (a binary operation) to mix two or more ugens.

```
(
 // mix 2 sawtooth waves
{
Saw.ar(500, 0.05)    // receiver
  +       // operator
Saw.ar(600, 0.06)    // operand
}.scope;
)



(
// mix 3 unit generators.
{
Saw.ar(500, 0.05)    // receiver
  +       // operator
Saw.ar(600, 0.06)    // operand
          // when evaluated produce
          // a BinaryOpUGen
          // this BinaryOpUGen is then a receiver for an
  +        // addition operator followed by
```

```
Saw.ar(700, 0.07)     // an operand
}.scope;
)

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Rewrite the previous example with the Mix ugen.

```
(
{
Mix.ar(
  // the ugens that will be mixed go into an array
[
Saw.ar(500, 0.05),
Saw.ar(600, 0.06),
Saw.ar(700, 0.06)
]
)
}.scope
)
```

Or use Mix.arFill to create the same result.

```
{ Mix.arFill(3, { arg i; Saw.ar(500 + (i * 100), 0.05) }) }.scope;
```

Every time the function is evaluated, the argument i is incremented. So i equals 0 the first time the function is evaluated, i equals 1 the second time, i equals 2, the third time, and so on.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
```

Scaling = multiplication

Apply an envelope, in the form of a low-frequency sine wave, to a WhiteNoise generator.

```
{ WhiteNoise.ar(0.1) * SinOsc.kr(1, 1) }.scope;

(
 // scaling and mixing
 // ...  imitates a train?
{
```

```
(WhiteNoise.ar(0.1) * SinOsc.kr(1, 1))
+
(BrownNoise.ar(0.1) * SinOsc.kr(2, 1))

}.scope;
)
```

//

Envelopes

Dynamically modulate any parameter in a ugen (such as frequency, phase, or amplitude) with an envelope.

```
// modulate amplitude
{ SinOsc.ar(440, 0, 0.1) * EnvGen.kr(Env.sine(1), doneAction:  2) }.scope;
```

Setting the doneAction argument (control) to 2 insures that after the envelope reaches its endpoint, SuperCollider will release the memory it used for the instances of the SinOsc and the EnvGen.

//

Keyword arguments

Keywords make code easier to read and they allow arguments to be presented in any order. Here, the doneAction and the timeScale arguments are expressed in keyword style.

```
(
SynthDef("timeScale", { arg ts = 1;
Out.ar(
0,
SinOsc.ar(440, 0, 0.4)
*
EnvGen.kr(
Env.sine(1),
doneAction:  2,
   timeScale:  ts  // scale the duration of an envelope
)
)
}).load(s);
```

```
)

Synth "timeScale"   \ts         // timeScale controls the duration of the envelope

////////////////////////////////////////////////////////////////////////////////////////////////////

// scale the duration of the envelope for every new synth
(
    Routine
loop({
Synth("timeScale", [\ts, 0.01.rrand(0.3)]);
0.5.wait;
})
});
)
r.play

////////////////////////////////////////////////////////////////////////////////////////////////////
```

Additive Synthesis

```
Additive synthesis is as its name says.  Components are added (mixed) together.

(
  // evaluate the function 12 times
var n = 12;
Mix.arFill(
n,
{
SinOsc.ar(
[67.0.rrand(2000), 67.0.rrand(2000)],
0,
n.reciprocal * 0.75
)
}
)
*
EnvGen.kr(Env.perc(11, 6), doneAction:  2)
}.scope
)
```

//

Envelopes

The promise of additive synthesis is that one can add sine waves to create any sound that can be imagined.

The problem of additive synthesis is that each and every sine wave and their envelopes have to be specified explicitly.

Create nuanced textures by scaling sine waves with envelopes and then mixing the result.

```
(
{ var n = 12;

Mix.arFill(
   n,         // generate n sine waves
{
   SinOsc           // each with a possible frequency between
    [67.0.rrand(2000), 67.0.rrand(2000)], // low.rrand(high) ...  floating point values
0,
    n.reciprocal      // scale the amplitude of each sine wave
            // according to the value of n
)
*
   EnvGen           // put an envelope on each of the sine waves
Env.sine(2.0.rrand(17)),
    doneAction:  0      // deallocate envelopes only when the
            // entire sound is complete (why?)
)
}
)
 *           // put an envelope over the whole patch
 EnvGen
Env.perc(11, 6),
doneAction:  2,
levelScale:  0.75
)
```

```
}.scope
)
```

(Or use the Klang ugen to produce a similar effect).

//

Ring modulation

Multiply two UGens.

```
  SinOsc.ar(440, 0, 0.571)  SinOsc

// use an lfo to modulate the amplitude of the modulator
(
{
SinOsc.ar(440, 0, 0.571)
*
    (SinOsc             // wrap the modulator and the lfo in parenthese
*         // why ...  ?
SinOsc.kr([6.99, 8.01].reciprocal)
)
}.scope
)
```

//

Amplitude modulation

Multiply two UGens and restrict the value of the modulator to positive values (use the .abs message to calculate 'absolute' value) to create what Charles Dodge calls "classic" amplitude modulation.

```
// use an lfo to modulate the amplitude of the modulator
(
{
SinOsc.ar(440, 0, 0.571)
*
    (SinOsc                // wrap the modulator and the lfo in parenthese
*         // why ...  ?
SinOsc.kr([6.99, 8.01].reciprocal)
```

```
)
}.scope
)

////////////////////////////////////////////////////////////////////////////////////////////////////

Compare "classic" amplitude modulation and ring modulation

// "classic"
  SinOsc.ar(440, 0, 0.571)  SinOsc

// "ring"
// ...  what's the difference?
  SinOsc.ar(440, 0, 0.571)  SinOsc

////////////////////////////////////////////////////////////////////////////////////////////////////
```

go to **10_Subtractive_synthesis**

16 Math

ID: 318

AbstractFunction

An AbstractFunction is an object which responds to a set of messages that represent mathematical functions. Subclasses override a smaller set of messages to respond to the mathematical functions. The intent is to provide a mechanism for functions that do not calculate values directly but instead compose structures for calculating.

Function, Pattern, Stream and UGen are subclasses of AbstractFunction.
For example, if you multiply two UGens together the receiver responds by answering a new
instance of class BinaryOpUGen which has the two operands as inputs.

Unary Messages:

All of the following messages send the message composeUnaryOp to the receiver with the
unary message selector as an argument.

neg, reciprocal, bitNot, abs, asFloat, asInt, ceil, floor, frac, sign, squared, cubed, sqrt
exp, midicps, cpsmidi, midiratio, ratiomidi, ampdb, dbamp, octcps, cpsoct, log, log2,
log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, rand, rand2, linrand, bilinrand,
sum3rand, distort, softclip, nyqring, coin, even, odd, isPositive, isNegative, isStrictlyPositive

Binary Messages:

All of the following messages send the message composeBinaryOp to the receiver with the
binary message selector and the second operand as arguments.

+, -, *, /, div, %, **, min, max, <, <=, >, >=, &, | , bitXor, lcm, gcd,

round, trunc, atan2,
hypot, », +», fill, ring1, ring2, ring3, ring4, difsqr, sumsqr, sqrdif, absdif, amclip,
scaleneg, clip2, excess, <!, rrand, exprand

Messages with more arguments:

All of the following messages send the message composeNAryOp to the receiver with the
binary message selector and the other operands as arguments.

clip, wrap, fold, blend, linlin, linexp, explin, expexp

Function Composition:

when unary, binary or n-ary operators are appied to an abstract function, it returns an object that represents
this operation, without evaluating the function: UnaryOpFunction, BinaryOpFunction, NAryOpFunction.
Note that different subclasses like Pattern or UGen have their own composition scheme analogous to the one of AbstractFunction itself. More about functions, see [Function]

```
// examples

a = { 1.0.rand } + 8;
a.value;


y = { 8 } + { 1.0.rand };
y.value;

// arguments are passed into both functions

y = { | x=0|  x } + { 1.0.rand };
```

```
y.value(10);


y = { | x=0|  x * 3 } + { | x=0| x + 1.0.rand };
y.value(10);

y.postcs;

y = { | x=0|  x * 3 } + { | x=0| x + 1.0.rand } * { | x=0|  [50, 100].choose + x } + 1.0;
y.value(10);

// environments can be used as a lookup with valueEnvir:

(
Environment
y = 10;
x = 2;
z = { | x=8|  x } + { | y=0|  y + 1.0.rand };
z.valueEnvir;
}
)

// n-ary operators:

a = blend({ 3.0.rand }, { 1000.rand }, { | frac|  frac });
a.value(0.5);

                        // creates a range of values..
```

ID: 319

Adverbs for Binary Operators

Adverbs are a third argument passed to binary operators that modifies how they iterate over SequenceableCollections or Streams. The idea for adverbs is taken from the J programming language which is a successor of APL.

Adverbs and SequenceableCollections

Normally when you add two arrays like this:

```
[10, 20, 30, 40, 50] + [1, 2, 3]
```

You get this result:

```
[ 11, 22, 33, 41, 52 ]
```

A new array is created which is the length of the longer array and items from each array are added together by wrapped indexing.

Using adverbs can change this behavior. Adverbs are symbols and they follow a '.' after the binary operator. Adverbs can be applied to all binary operators.

adverb 's'

The first adverb is 's' which means 'short'. The add operation now returns the shorter of the two arrays.

```
[10, 20, 30, 40, 50] +.s [1, 2, 3]
```

returns:

```
[ 11, 22, 33 ]
```

adverb 'f'

Another adverb is 'f' which uses folded indexing instead of wrapped:

```
[10, 20, 30, 40, 50] +.f [1, 2, 3]
```

returns:

```
[ 11, 22, 33, 42, 51 ]
```

adverb 't'

The table adverb 't' makes an array of arrays where each item in the first array is added to the whole second array and the resulting arrays are collected.

```
[10, 20, 30, 40, 50] +.t [1, 2, 3]
```

returns:

```
[ [ 11, 12, 13 ], [ 21, 22, 23 ], [ 31, 32, 33 ], [ 41, 42, 43 ], [ 51, 52, 53 ] ]
```

adverb 'x'

The cross adverb 'x' is like table, except that the result is a flat array:

```
[10, 20, 30, 40, 50] +.x [1, 2, 3]
```

```
[ 11, 12, 13, 21, 22, 23, 31, 32, 33, 41, 42, 43, 51, 52, 53 ]
```

Adverbs and Streams

There is currently one adverb defined for streams. This is the cross adverb, 'x'.

Normally when you add two streams like this:

```
p = (Pseq([10, 20]) + Pseq([1, 2, 3])).asStream;
Array.fill(3, { p.next });
```

you get this:

```
[ 11, 22, nil ]
```

The items are paired sequentially and the stream ends when the earliest stream ends.

The cross adverb allows you to pair each item in the first stream with every item in the second stream.

```
p = (Pseq([10, 20]) +.x Pseq([1, 2, 3])).asStream;
Array.fill(7, { p.next });
```

the result is:

```
[ 11, 12, 13, 21, 22, 23, nil ]
```

You can string these together. Every item in the left stream operand is "ornamented" by the right stream operand.

```
p = (Pseq([100, 200]) +.x Pseq([10, 20, 30]) +.x Pseq([1, 2, 3, 4])).asStream;
Array.fill(25, { p.next });

[ 111, 112, 113, 114, 121, 122, 123, 124, 131, 132, 133, 134,
211, 212, 213, 214, 221, 222, 223, 224, 231, 232, 233, 234, nil ]
```

Sound example:

```
s.boot;
SynthDescLib.global.read;

Pbind(\dur, 0.17, \degree, Pwhite(0,6) +.x Pseq([[0, 2, 4],1,[0,2],3])).play
```

ID: 320

Complex

superclass: Number

A class representing complex numbers.

Creation

new(real, imag)

Create a new complex number with the given real and imaginary parts.

Accessing

<>real

The real part of the number.

<>imag

The imaginary part of the number.

Math

+ aNumber

Complex addition.

- aNumber

Complex subtraction.

*** aNumber**

Complex multiplication.

/ aNumber

Complex division.

< aNumber

Answer the comparision of just the real parts.

neg

Negation of both parts.

conjugate

Answer the complex conjugate.

Conversion

magnitude

Answer the distance to the origin.

rho

Answer the distance to the origin.

angle

Answer the angle in radians.

phase

Answer the angle in radians.

theta

Answer the angle in radians.

asPoint

Convert to a Point.

asPolar

Convert to a Polar

asInteger

Answer real part as Integer.

asFloat

Answer real part as Float.

ID: 321

Float

superclass: SimpleNumber

A 64 bit double precision floating point number. Float inherits most of its behaviour from its superclass.

Random Numbers

coin

Answers a Boolean which is the result of a random test whose probability of success in a range from
zero to one is this.

Testing

isFloat

Answer true since this is a Float.

Converting

asFloat

Answer this since this is a Float.

ID: 322

Infinitum

class Infinitum is removed.

inf is now a floating point infinity.

ID: 323

Integer

superclass: SimpleNumber

A 32 bit integer. Integer inherits most of its behaviour from its superclass.

Iteration

do(function)

Executes function for all integers from zero to this minus one.
function - a Function which is passed two arguments, both of which are the same integer from zero to this minus one. The reason two arguments are passed is for symmetry with the implementations of do in Collection.

for(endval, function)

Executes function for all integers from this to endval, inclusive.
endval - an Integer.
function - a Function which is passed two arguments, the first which is an integer from this to
endval, and the second which is a number from zero to the number of iterations minus one.

forBy(endval, step, function)

Executes function for all integers from this to endval, inclusive, stepping each time by step.
endval - an Integer.
step - an Integer.
function - a Function which is passed two arguments, the first which is an integer from this to
endval, and the second which is a number from zero to the number of iterations minus one.

Conversion

asAscii

Answer a Char which has the ASCII value of the receiver.

asDigit

Answer a Char which represents the receiver as an ASCII digit. For example 5.asDigit returns $5.

Random Numbers

xrand(exclude)

Answer a random value from zero to this, excluding the value exclude.
exclude - an Integer.

xrand2(exclude)

Answer a random value from **this**.neg to this, excluding the value exclude.
exclude - an Integer.

Powers Of Two

nextPowerOfTwo

Answer the next power of two greater than or equal to the receiver.

```
13.nextPowerOfTwo.postln;
64.nextPowerOfTwo.postln;
```

isPowerOfTwo

Answer the whether the receiver is a power of two.

```
13.isPowerOfTwo.postln;
64.isPowerOfTwo.postln;
```

Prime Numbers

nthPrime

Answer the nth prime number. The receiver must be from 0 to 6541.

```
[0,1,2,3,4,5].collect({ arg i; i.nthPrime; }).postln;
```

prevPrime

Answer the next prime less than or equal to the receiver up to 65521.

```
25.prevPrime.postln;
```

nextPrime

Answer the next prime less than or equal to the receiver up to 65521.

```
25.nextPrime.postln;
```

isPrime

Answer whether the receiver is prime.

```
25.isPrime.postln;
13.isPrime.postln;
```

Misc

getKeys

Returns the bits from the Macintosh GetKeys() Toolbox call. Receiver should be 0 to 3.

```
[0.getKeys, 1.getKeys, 2.getKeys, 3.getKeys].postln;
```

ID: 324

```
// The J programming language is a successor of APL. <http://www.jsoftware.com>
// These languages are made for processing arrays of data and are able to express
// complex notions of iteration implicitly.
// The following are some concepts borrowed from or inspired by J.
// Thinking about multidimensional arrays can be both mind bending and mind expanding.
// It may take some effort to grasp what is happening in these examples.

// iota fills an array with a counter
z = Array.iota(2, 3, 3);
        // 3 dimensions
         // gives the sizes of the dimensions
                        // reshape changes the dimensions of an array
        // 3 dimensions
z.shape;

// fill a 2D array
Array.fill2D(3,3,{1.0.rand.round(0.01)});

Array.fill2D(2,3,{| i,j|  i@j});

// fill a 3D array
Array.fill3D(2,2,2,{1.0.rand.round(0.01)});

Array.fill3D(2,2,2,{| i,j,k|  `[i,j,k]});


// using dup to create arrays
(1..4) dup:  3;
100.rand dup:  10;
{100.rand} dup:  10;
{100.rand} dup:  3 dup:  4;
{{100.rand} dup:  3} dup:  4;
{| i|  i.squared} dup:  10;
{| i|  i.nthPrime} dup:  10;

// !  is an abbreviation of dup
(1..4) !  3;
100.rand !  10;
```

```
{100.rand} !  10;
{100.rand} !  3 !  4;
{{100.rand} !  3} !  4;
{| i|  i.squared} !  10;
{| i|  i.nthPrime} !  10;

// other ways to do the same thing:
  // partial application
_.squared !  10;
_.nthPrime !  10;

 // operating on a list
(0..9).squared;
(0..9).nthPrime;

// operator adverbs
// Adverbs are a third argument passed to binary operators that modifies how they iterate over
// SequenceableCollections or Streams.
// see the Adverbs help file
[10, 20, 30, 40, 50] + [1, 2, 3]; // normal
[10, 20, 30, 40, 50] +.f [1, 2, 3]; // folded
[10, 20, 30, 40, 50] +.s [1, 2, 3]; // shorter
[10, 20, 30, 40, 50] +.x [1, 2, 3]; // cross
[10, 20, 30, 40, 50] +.t [1, 2, 3]; // table

// operator depth.
// J has a concept called verb rank, which is probably too complex to understand and implement
// in SC, but operator depth is similar and simpler.
// A binary operator can be given a depth at which to operate
// negative depths iterate the opposite operand.
// These are better understood by example.
// It is not currently possible to combine adverb and depth.
z = Array.iota(3,3);
y = [100, 200, 300];
z + y;
         // same as the above.  y added to each row of z
         // y added to each column of z
         // y added to each element of z
          // z added to each element of y
```

```
// deepCollect operates a function at different dimensions or depths in an array.
z = Array.iota(3, 2, 3);
f = {| item|  item.reverse };
z.deepCollect(0, f);
z.deepCollect(1, f);
z.deepCollect(2, f);

f = {| item|  item.stutter };
z.deepCollect(0, f);
z.deepCollect(1, f);
z.deepCollect(2, f);

// slice can get sections of multidimensional arrays.
// nil gets all the indices of a dimension
z = Array.iota(4,5);
z.slice(nil, (1..3));
z.slice(2, (1..3));
z.slice((2..3), (0..2));
z.slice((1..3), 3);
z.slice(2, 3);

z = Array.iota(3,3,3);
z.slice([0,1],[1,2],[0,2]);
z.slice(nil,nil,[0,2]);
z.slice(1);
z.slice(nil,1);
z.slice(nil,nil,1);
z.slice(nil,2,1);
z.slice(nil,1,(1..2));
z.slice(1,[0,1]);
z.flop;

// sorting order

                        // generate a random array;
// order returns an array of indices representing what would be the sorted order of the array.
o = z.order;
          // using the order as an index returns the sorted array

// calling order on the order returns an array of indices that returns the sorted array to the
```

```
// original scrambled order
p = o.order;
x = y[p];

// bubbling wraps an item in an array of one element.  it takes the depth and levels as arguments.
z = Array.iota(4,4);
z.bubble;
z.bubble(1);
z.bubble(2);
z.bubble(0,2);
z.bubble(1,2);
z.bubble(2,2);
// similarly, unbubble unwraps an Array if it contains a single element.
5.unbubble;
[5].unbubble;
[[5]].unbubble;
[[5]].unbubble(0,2);
[4,5].unbubble;
[[4],[5]].unbubble;
[[4],[5]].unbubble(1);
z.bubble.unbubble;
z.bubble(1).unbubble(1);
z.bubble(2).unbubble(2);

// laminating with the +++ operator
// the +++ operator takes each item from the second list and appends it to the corresponding item
// in the first list.  If the second list is shorter, it wraps.
z = Array.iota(5,2);
z +++ [77,88,99];
z +++ [[77,88,99]];
z +++ [[[77,88,99]]];
z +++ [ [[77]],[[88]],[[99]] ];
// same as:
z +++ [77,88,99].bubble;
z +++ [77,88,99].bubble(0,2);
z +++ [77,88,99].bubble(1,2);

z +++ [11,22,33].pyramidg;
z +++ [11,22,33].pyramidg.bubble;
z +++ [[11,22,33].pyramidg];
```

```
z +++ [[[11,22,33].pyramidg]];


(
z = (1..4);
10.do {| i|
z.pyramid(i+1).postln;
z.pyramidg(i+1).postln;
"".postln;
};
)

// reshapeLike allows you to make one nested array be restructured in the same manner as another.

a = [[10,20],[30, 40, 50], 60, 70, [80, 90]];
b = [[1, 2, [3, 4], [[5], 6], 7], 8, [[9]]];
a.reshapeLike(b);
b.reshapeLike(a);

// If the lengths are different, the default behaviour is to wrap:

a = [[10,20],[30, 40, 50]];
b = [[1, 2, [3, 4], [[5], 6], 7], 8, [[9]]];
a.reshapeLike(b);

// but you can specify other index operators:

a.reshapeLike(b, \foldAt);

a.reshapeLike(b, \clipAt);

a.reshapeLike(b, \at);

// allTuples will generate all combinations of the sub arrays
[[1, 2, 3], [4, 5], 6].allTuples;
[[1, 2, 3], [4, 5, 6, 7], [8, 9]].allTuples;
```

ID: 325

Magnitude

superclass: Object

Magnitudes represent values along a linear continuum which can be compared against each other.

< aMagnitude

Answer a Boolean whether the receiver is less than aMagnitude.

<= aMagnitude

Answer a Boolean whether the receiver is less than or equal to aMagnitude.

> aMagnitude

Answer a Boolean whether the receiver is greater than aMagnitude.

>= aMagnitude

Answer a Boolean whether the receiver is greater than or equal to aMagnitude.

min(aMagnitude)

Answer the minimum of the receiver and aMagnitude.

max(aMagnitude)

Answer the maximum of the receiver and aMagnitude.

clip(minVal, maxVal)

If the receiver is less than minVal then answer minVal, else if the receiver is greater than maxVal then answer maxVal, else answer the receiver.

inclusivelyBetween(minVal, maxVal)

Answer whether the receiver is greater than or equal to minVal and less than or equal to maxVal.

exclusivelyBetween(minVal, maxVal)

Answer whether the receiver is greater than minVal and less than maxVal.

ID: 326

Number

Superclass: Magnitude

Number represents a mathematical quantity.

Math

+ aNumber

Addition.

- aNumber

Subtraction.

*** aNumber**

Multiplication.

/ aNumber

Division.

div(aNumber)

Integer division.

% aNumber

Modulo.

**** aNumber**

Exponentiation.

squared

The square of the number.

cubed

The cube of the number.

Polar Coordinate Support

rho

Answer the polar radius of the number.

theta

Answer the polar angle of the number.

Complex Number Support

real

Answer the real part of the number.

imag

Answer the imaginary part of the number.

Conversion

@ aNumber

Create a new Point whose x coordinate is the receiver and whose y coordinate is aNumber.

complex(imaginaryPart)

Create a new Complex number whose real part is the receiver with the given imaginary

part.

polar(angle)

Create a new Polar number whose radius is the receiver with the given angle.

Iteration

for(endval, function)

Executes function for numbers from this up to endval, inclusive, stepping each time by 1.
endval - a Number.
function - a Function which is passed two arguments, the first which is an number from this to
endval, and the second which is a number from zero to the number of iterations minus one.

forBy(endval, step, function)

Executes function for numbers from this up to endval, stepping each time by step.
endval - a Number.
step - a Number.
function - a Function which is passed two arguments, the first which is an number from this to
endval, and the second which is a number from zero to the number of iterations minus one.

ID: 327

Polar

superclass: Number

Represents polar coordinates.

Creation

new(rho, theta)

Create a new polar coordinate with the given radius, rho, and angle in radians, theta.

Math

\+ - * /

The math operations of addition, subtraction, multiplication and division are accomplished by
first converting to complex numbers.

scale(aNumber)

Scale the radius by some value.

rotate(aNumber)

Rotate the angle by some value.

neg

Rotate by pi.

Conversion

magnitude

Answer the radius.

angle

Answer the angle in radians

phase

Answer the angle in radians

real

Answer the real part.

imag

Answer the imaginary part.

asComplex

Convert to Complex

asPoint

Convert to Point

ID: 328

SimpleNumber

superclass: Number

Represents numbers which can be represented by a single one dimensional value. Most of the Unary and Binary operations are also implemented by UnaryOpUGen and BinaryOpUGen, so you can get more examples by looking at the help for those.

Unary Operations

neg

negation

bitNot

ones complement

abs

absolute value.

ceil

next larger integer.

floor

next smaller integer

frac

fractional part.

sign

Answer -1 if negative, +1 if positive or 0 if zero.

squared

The square of the number.

cubed

The cube of the number.

sqrt

The square root of the number.

exp

e to the power of the receiver.

reciprocal

1 / this

midicps

Convert MIDI note to cycles per second

cpsmidi

Convert cycles per second to MIDI note.

midiratio

Convert an interval in semitones to a ratio.

ratiomidi

Convert a ratio to an interval in semitones.

ampdb

Convert a linear amplitude to decibels.

dbamp

Convert a decibels to a linear amplitude.

octcps

Convert decimal octaves to cycles per second.

cpsoct

Convert cycles per second to decimal octaves.

log

Base e logarithm.

log2

Base 2 logarithm.

log10

Base 10 logarithm.

sin

Sine.

cos

Cosine.

tan

Tangent.

asin

Arcsine.

acos

Arccosine.

atan

Arctangent.

sinh

Hyperbolic sine.

cosh

Hyperbolic cosine.

tanh

Hyperbolic tangent.

rand

Random number from zero up to the receiver, exclusive.

rand2

Random number from -this to +this.

linrand

Linearly distributed random number from zero to this.

bilinrand

Bilateral linearly distributed random number from -this to +this.

sum3rand

A random number from -this to +this that is the result of summing three uniform random generators

to yield a bell-like distribution. This was suggested by Larry Polansky as a poor man's gaussian.

distort

a nonlinear distortion function.

softclip

Distortion with a perfectly linear region from -0.5 to +0.5

coin

Answers a Boolean which is the result of a random test whose probability of success in a range from
zero to one is this.

even

Answer if the number is even.

odd

Answer if the number is odd.

isPositive

Answer if the number is >= 0.

isNegative

Answer if the number is < 0.

isStrictlyPositive

Answer if the number is > 0.

Binary Operations

+ aNumber

Addition

- aNumber

Subtraction

*** aNumber**

Multiplication

/ aNumber

Division

% aNumber

Modulo

div(aNumber)

Integer Division

**** aNumber**

Exponentiation

min(aNumber)

Minimum

max(aNumber)

Maximum

& aNumber

Bitwise And

| aNumber

Bitwise Or

bitXor(aNumber)

Bitwise Exclusive Or

lcm(aNumber)

Least common multiple

gcd(aNumber)

Greatest common divisor

round(aNumber)

Round to multiple of aNumber

trunc(aNumber)

Truncate to multiple of aNumber

atan2(aNumber)

Arctangent of (this/aNumber)

hypot(aNumber)

Square root of the sum of the squares.

« aNumber

Binary shift left.

» aNumber

Binary shift right.

+» aNumber

Unsigned binary shift right.

fill(aNumber)

ring1(aNumber)

(a * b) + a

ring2(aNumber)

((a*b) + a + b)

ring3(aNumber)

(a*a *b)

ring4(aNumber)

((a*a *b) - (a*b*b))

difsqr(aNumber)

(a*a) - (b*b)

sumsqr(aNumber)

(a*a) + (b*b)

sqrdif(aNumber)

(a - b)**2

sqrsum(aNumber)

(a + b)**2

absdif(aNumber)

(a - b).abs

amclip(aNumber)

0 when b <= 0, a*b when b > 0

scaleneg(aNumber)

a*b when a < 0, otherwise a.

clip2(aNumber)

clips receiver to +/- aNumber

excess(aNumber)

Returns the difference of the receiver and its clipped form: (a - clip2(a,b)).

<! aNumber

Return the receiver. aNumber is ignored.

asFraction(denominator, fasterBetter)

Return an array of denominator and divisor of the nearest and smallest fraction

rrand(aNumber)

Returns a random number in the interval [a, b). If both a and b are Integer then the result will be an Integer.

exprand(aNumber)

Returns an exponentially distributed random number in the interval [a, b). Always returns a Float.

degreeToKey(scale, stepsPerOctave)

the value is truncated to an integer and used as an index into an octave repeating table

of note values.
Indices wrap around the table and shift octaves as they do
stepsPerOctave is 12 by default

```
(
l = [0, 1, 5, 9, 11]; // pentatonic scale
(1, 2..15).collect { | i|  i.degreeToKey(l, 12) }
)
```

keyToDegree(scale, stepsPerOctave)

inverse of degreeToKey.
stepsPerOctave is 12 by default

```
(
l = [0, 1, 5, 9, 11]; // pentatonic scale
(60, 61..75).collect { | i|  i.keyToDegree(l, 12) }
)
(
l = [0, 1, 5, 9, 11]; // pentatonic scale
(60, 61..75).postln.collect { | i|  i.keyToDegree(l, 12).degreeToKey(l) }
)
```

nearestInList(list)

returns the value in the collection closest to this

```
(
l = [0, 0.5, 0.9, 1];
(0, 0.05..1).collect { | i| i.nearestInList(l) }
)
```

nearestInScale(scale, stepsPerOctave)

returns the value in the collection closest to this, assuming an octave repeating table of note values.
stepsPerOctave is 12 by default

```
(
l = [0, 1, 5, 9, 11]; // pentatonic scale
(60, 61..76).collect { | i| i.nearestInScale(l, 12) }
)
```

asTimeString(precision)

returns a string corresponding to the hours:minutes:seconds based on the receiver as number of seconds
precision is 0.1 by default

```
(
var start;
start = Main.elapsedTime;
{ loop({(Main.elapsedTime - start).asTimeString.postln; 0.05.wait}) }.fork;
)
```

17 Miscellanea

ID: 329

a list of changes

a list of changes to SuperCollider.

Note: this list does not guarantee to include every change.
To look up detailed changes check the CVS, or the sc-dev mailing list archives.

ID: 330

SuperCollider Help

Select any of the items listed below by clicking on it to open the corresponding helpfile. See **[More-On-Getting-Help]** for further information.

Essential Topics Language Tutorials

[More-On-Getting-Help] [Intro-to-Objects] [How-to-Use-the-Interpreter]
[Server-Architecture] [Literals] [Getting Started With SC] -Scott Wilson
[Server-Command-Reference] [Method-Calls] [Introductory_tutorial] -Mark Polishook
[Tour_of_UGens] [Assignment] [Tutorial] - foundational SC tutorial
[UGens-and-Synths] [Comments]
[ClientVsServer] [Expression-Sequence]
[NodeMessaging] [Functions]
[Order-of-execution] [Scope]
[ServerTiming] [Control-Structures]
[Internal-Snooping] [Classes]
[MultiChannel] [Polymorphism]
[SC3vsSC2] [Syntax-Shortcuts]
[Backwards-Compatibility] [SymbolicNotations]
[DocumentAutoCompletion] [Adverbs]
[Partial-Application]
[J_concepts_in_SC]
[ListComprehensions]

Extending SC Miscellaneous Topics

[Using-the-Startup-File] [UsingMIDI]
[Writing-Classes] [Understanding-Errors]
[Writing_Unit_Generators] [Debugging-tips]
[Using-Extensions] [Randomness]
[Creating-Standalone-Applications]

SCLang Classes (incomplete list)

[Undocumented-Classes]

Overviews Core Scheduling Control

[UGens] [Object] [AppClock] [Spec]
[Streams] [Class] [SystemClock] [ControlSpec]
[Operators] [Frame] [TempoClock] [HIDDeviceService]
[Collections] [Function] [Task] [MIDIIn]
[GUIClasses] [FunctionDef] [Scheduler] [MIDIOut]
[FFT_Overview] [Method] [Condition] [Env]
[RawPointer] [if]
[Ref] [CmdPeriod]
[Routine]
[AbstractFunction]
[Nil]
[Interpreter]
[Process]
[Boolean]
[False]
[True]
[Char]
[Symbol]
[Thread]

Math Geometry Files Server Control

[Integer] [Point] [UnixFILE] [RootNode]
[Float] [Rect] [File] [Node]
[Complex] [Pipe] [Group]
[Polar] [SoundFile] [default_group]
[Magnitude] [CocoaDialog] [Bus]
[Number] [Buffer]
[SimpleNumber] [OSCresponder]
[OSCresponderNode]
[Synth]
[Server]
[ServerOptions]
[NodeWatcher]
[NodeControl]
[bundledCommands]

[NetAddr]
[Non-Realtime-Synthesis]

Audio Misc GUI

[SynthDef] [initClass] [SCWindow]
[UGens] [writeAsPlist] [SCButton]
[randomSeed] [Stethoscope]
[play]
[asTarget]

Help Scripts (experimental)

*Show All Documented Classes *Show All Documented Extension Classes

*Show All Undocumented Classes

Extension Libraries

[CRUCIAL-LIBRARY]
[JITLib]

Publishing Code
[publishing_code]

Changes
[changes] an incomplete list of changes to the class library / sc app.

to be continued...

ID: 331

How to use the Interpreter

This document is OSX specific. For the linux emacs sclang interface please see linux specific documentation.

You can execute any single line expression by clicking anywhere in that line and pressing the **'Enter'** key.
Note that the 'Enter' key is not the same key as 'Return'.

You will need to start the default server before you can hear any examples. By convention the default server is assigned to the interpreter variable 's'. (At startup the default will be the localhost server.) You can start the server app by pressing the 'Boot' button on the localhost server window, or you can do it in code:

```
// execute these lines one at a time by placing the cursor on the line and then pressing 'enter'

        // this boots the default Server.  Watch the post window and server window for the result

// once that's done execute this to make a sound
{ FSinOsc.ar(800, 0, 0.1) }.play;
```

(Press and hold Cmd (the Apple key) and then press period to stop the sound started above.)

In the help files all executable fragments are written in the `Monaco` font.

If an expression has multiple lines you can select all of the lines before typing 'Enter'.

```
// Select all three of the following lines and press 'Enter':
w = SCWindow.new("Fading").front;
r = Routine({ 200.do({| i|  w.alpha = 1 - (i * 0.005); 0.005.wait;}); w.close; });
AppClock.play(r);
```

Some examples do require lines to be executed one at a time, or certain lines to be executed first. By far the most common case of this is booting the server app, as we did at the top of the page. Until the server has completed booting, no sound producing code will work.

However, most of the examples included with the app have parentheses around lines of code which should be executed at the same time. (This is a convention which should be followed in your own code.) This allows you to double click to the right of the open paren and select the entire expression. Then press 'enter'.

```
(
// ^^^^^^^^ double click above this line ^^^^^^^^
play({
// Three patches in one...
       // number of strings
      // array of possible impulse excitation behaviours
{ Impulse.ar(2 + 0.2.rand, 0.3) }, // slow phasing
{ Dust.ar(0.5, 0.3) }, // "wind chimes"
{ Impulse.ar(SinOsc.kr(0.05+0.1.rand, 2pi.rand, 5, 5.2), 0.3) } // races
 ].choose;  // choose one at random to use for all voices
Mix.new(
 Array           // n strings tuned randomly to MIDI keys 60-90
var delayTime;
  // calculate delay based on a random note
delayTime = 1 / (60 + 30.rand).midicps;
Pan2.ar(
   LeakDC      // removes DC buildup
    CombL      // used as a string resonator
     Decay      // decaying envelope for noise
      b.value,    // instantiate an exciter
      0.04,    // decay time of excitation
      PinkNoise           // multiply noise by envelope
     delayTime,   // max delay time
     delayTime,  // actual delay time
     4)),     // decay time of string
   1.0.rand2 // random pan position
)
}))
})
)
```

Again, press Cmd-. to stop the sound. This will stop all audio (and free all nodes on the server) at any time.

When you're done you can quit the server app by pressing the 'Quit' button on the

localhost server window, or do it by executing the following code:

```
s.quit;
```

ID: 332

More on Getting Help

SuperCollider 3 is a work in progress. As such much is undocumented. Many of the helpfiles have been copied over from SC2, and have unworking examples, etc. These are in the process of being updated.

Below are listed a few techniques for tracking down documentation and functionality. **Note:** If some of the terms used below (e.g. class, method, inheritance, etc.) are unclear to you, you may wish to read the **Language** helpfiles which are listed in the main help window for detail on some of these concepts. Reading a general tutorial on Object Oriented Programming at some point could also be useful, as could reading a FAQ, etc. about Smalltalk. Smalltalk is the general purpose OOP language upon which the the design of the SuperCollider language is based. It's syntax is different than SC's, but conceptually it has much in common.

NB: Be sure to check out the Further Info section at the bottom of this page.

Basics

As you've probably already learned selecting any text and pressing Cmd-Shift-? will open the corresponding helpfile. Usually helpfiles are either concept related, or document particular classes. In the SC language classes begin with capital letters. Try Cmd-Shift-? on the following (double click on the first word; the stuff after the two slashes is a comment):

```
Class// this is a class
```

A few methods also have helpfiles. Methods begin with lower-case letters, as do many other things in the language.

```
play // Cmd-shift-?  on this will open a helpfile detailing different implementations of this method
```

In addition there are a many helpfiles which explain important concepts. Most of these are listed in the main help window (Help.help.rtf, which will open if you press Cmd-Shift-? with no text selected), or in the overviews listed there. Cmd-Shift-? on this text for an example: **Tutorial**

A convention of the documentation is that text in bold face or between brackets **[]** refers

to other helpfiles. Double-click to select that text and press Cmd-Shift-? to open the corresponding helpfile. An example is the text **Undocumented-Classes** in the next section.

Note that many helpful methods print information to the 'post window'. Unless you have explicitly changed it (see the Window menu) this is the window which opened when you launched. Pressing Cmd-\will bring the current post window to the front.

Tracking Down Information

Executing the following

```
Help.all;
```

will open a new window which lists all helpfiles by directory. Similarly the helpfile **Undocumented-Classes** contains a (possibly out of date) list of all classes which have no helpfiles. This can be a good place to start looking for functionality which may already be implemented.

Looking in class definitions (select any class and press Cmd-j to open its class definition file) can help you to figure out what undocumented methods do.

```
Array// Try Cmd-j on this
```

Since many methods use other classes, you may need to continue your search in other class definitions or helpfiles.

Executing the method dumpInterface on any class will list its class and instance methods and their arguments (if any).

```
Array               // Look at the post window (the one that opened when you started SC)
```

Note that since the SuperCollider language is object-oriented many classes inherit methods from farther up the class hierarchy. (The many subclasses of Collection are a good example of this. See the **Collections** overview for more detail.) It would be impractical and redundant to document every inherited method that a class responds to, so it is important to be able to track down documentation and method definitions.

The method dumpFullInterface applied to any Class will list all class and instance methods that a class responds to, sorted by the class in which they are implemented. This

will include inherited methods. Methods overidden in a subclass are listed under the subclass.

```
Array.dumpFullInterface;
```

This can be a lot of information, so dumpAllMethods or class.dumpAllMethods will show only instance and class methods respectively.

```
Array                       // Only class methods that this responds to (including inherited ones)
Array                  // Only instance methods (including inherited ones)
```

There is also a graphical Class browser which will show all methods, arguments, subclasses, instance variables and class variables. (Currently this is only OSX.) Using the browser's buttons you can easily navigate to the class' superclass, subclasses, class source, method source, helpfile (if there is one), check references or implementation of methods, or even open a web browser to view the corresponding entry in the online CVS repository. (Note that the web repository is a backup often a day or two behind the status of what is available to developers.)

```
SequenceableCollection
```

Selecting any method and pressing Cmd-y will open a window with a list of all the classes that *implement* that method. (See the **Polymorphism** helpfile for detail on why different classes might implement methods with the same name.)

```
    // try it on this method
```

Similarly, selecting any text and typing shift-cmd-y will open a window showing all references to the selected text, i.e. each place it is used within the class library. (This will not find methods calls compiled with special byte codes like 'value'.)

```
// try it on these
asStream
SCWindow
```

In the resulting window selecting any class and method and pressing Cmd-j will take you to that method definition in that class definition. For example try selecting `Pattern-select` in the window resulting from the previous example. Note that SC supports defining methods in separate files, so a class' methods may be defined in more than one place. If

you try Cmd-j on the following you will see that it will open a file called dumpFullInterface.sc rather than one called Class.sc (its main class definition file). The `+ Class{...` syntax indicates that these are additional methods.

```
Class-dumpFullInterface
```

If you know that a class responds to a particular message, you can use findRespondingMethod to find out which class it inherits the corresponding method from.

```
Array                          'select'   // you can Cmd-J on the result in the post window
```

Similarly, helpFileForMethod will open the helpfile of the class in which the responding method is defined (if the helpfile exists). Note that this does not guarantee that the method is documented therein. As noted above, some documentation is incomplete, and some methods are 'private' or not intended for general use.

```
Array                    'select'   // this will open the Collection helpfile; scroll down for select
```

In general poking around can be a good way to learn about how things work. See **Internal-Snooping** for more advanced information about how to look 'under the hood.'

For Further Info

A good starting place for figuring out how to do something are the numerous files in the Examples folder. The SuperCollider swiki is another good source of tips, examples, and information:

#2121ffhttp://swiki.hfbk-hamburg.de:8888/MusicTechnology/6

To edit or add pages on the swiki use: username: sc password: sc

Further help can be obtained by subscribing and sending questions to the sc-users mailing list:
#1a1aff
#2121ffhttp://www.create.ucsb.edu/mailman/listinfo/sc-users

An archive of the list can be searched from this page:

#2121ffhttp://swiki.hfbk-hamburg.de:8888/MusicTechnology/437

Requests for documentation of undocumented methods or classes, as well as reports of errata, omissions, etc. in helpfiles can be sent to: #2121ffmullmusik@users.sourceforge.net or to the user's list above.

ID: 333

Non-Realtime Synthesis

This documentation is initial.

SuperCollider 3 supports non-realtime synthesis through the use of binary files of OSC commands.

First create an OSC command file (i.e. a score)

```
    File "Cmds.osc" "w"

// start a sine oscillator at 0.2 seconds.
c = [ 0.2, [\s_new, \NRTsine, 1001, 0, 0]].asRawOSC;
                // each bundle is preceeded by a 32 bit size.
            // write the bundle data.

// stop sine oscillator at 3.0 seconds.
c = [ 3.0, [\n_free, 1001]].asRawOSC;
f.write(c.size);
f.write(c);

// scsynth stops processing immediately after the last command, so here is
// a do-nothing command to mark the end of the command stream.
c = [ 3.2, [0]].asRawOSC;
f.write(c.size);
f.write(c);

f.close;

// the 'NRTsine' SynthDef
(
SynthDef("NRTsine",{ arg freq = 440;
Out.ar(0,
SinOsc.ar(freq, 0, 0.2)
)
}).writeDefFile;
)
```

then on the command line (i.e. in Terminal):

./scsynth -N Cmds.osc _ NRTout.aiff 44100 AIFF int16

The command line arguments are:

-N <cmd-filename> <input-filename> <output-filename> <sample-rate> <header-format> <sample-format> <...other scsynth arguments>

If you do not need an input sound file, then put "_" for the file name as in the example above.

For details on other valid arguments to the scsynth app see **[Server-Architecture]**.

This could be executed in SC as:

```
"./scsynth -N Cmds.osc _ NRTout.aiff 44100 AIFF int16 -o 1"          // -o 1 is mono output
```

A more powerful option is to use the **Score** object, which has convenience methods to create OSC command files and do nrt synthesis. See the **[Score]** helpfile for more details.

```
(
x = [

[0.0, [ \s_new, \NRTsine, 1000, 0, 0,  \freq, 1413 ]],
[0.1, [ \s_new, \NRTsine, 1001, 0, 0,  \freq, 712 ]],
[0.2, [ \s_new, \NRTsine, 1002, 0, 0,  \freq, 417 ]],
[0.3, [ \s_new, \NRTsine, 1003, 0, 0,  \freq, 1238 ]],
[0.4, [ \s_new, \NRTsine, 1004, 0, 0,  \freq, 996 ]],
[0.5, [ \s_new, \NRTsine, 1005, 0, 0,  \freq, 1320 ]],
[0.6, [ \s_new, \NRTsine, 1006, 0, 0,  \freq, 864 ]],
[0.7, [ \s_new, \NRTsine, 1007, 0, 0,  \freq, 1033 ]],
[0.8, [ \s_new, \NRTsine, 1008, 0, 0,  \freq, 1693 ]],
[0.9, [ \s_new, \NRTsine, 1009, 0, 0,  \freq, 410 ]],
[1.0, [ \s_new, \NRTsine, 1010, 0, 0,  \freq, 1349 ]],
[1.1, [ \s_new, \NRTsine, 1011, 0, 0,  \freq, 1449 ]],
[1.2, [ \s_new, \NRTsine, 1012, 0, 0,  \freq, 1603 ]],
[1.3, [ \s_new, \NRTsine, 1013, 0, 0,  \freq, 333 ]],
```

```
[1.4, [ \s_new, \NRTsine, 1014, 0, 0,  \freq, 678 ]],
[1.5, [ \s_new, \NRTsine, 1015, 0, 0,  \freq, 503 ]],
[1.6, [ \s_new, \NRTsine, 1016, 0, 0,  \freq, 820 ]],
[1.7, [ \s_new, \NRTsine, 1017, 0, 0,  \freq, 1599 ]],
[1.8, [ \s_new, \NRTsine, 1018, 0, 0,  \freq, 968 ]],
[1.9, [ \s_new, \NRTsine, 1019, 0, 0,  \freq, 1347 ]],

[3.0, [\c_set, 0, 0]]
];
)
```

You can then use Score.write to convert the above to the OSC command file as follows:

```
Score          "score-test.osc"
"./scsynth -N score-test.osc _ score-test.aiff 44100 AIFF int16 -o 1"
```

Score also provides methods to do nrt synthesis directly:

```
(
var f, o;
g = [
[0.1, [\s_new, \NRTsine, 1000, 0, 0, \freq, 440]],
[0.2, [\s_new, \NRTsine, 1001, 0, 0, \freq, 660]],
[0.3, [\s_new, \NRTsine, 1002, 0, 0, \freq, 220]],
[1, [\c_set, 0, 0]]
];
o = ServerOptions.new.numOutputBusChannels = 1; // mono output
Score              "help-oscFile.osc"  "helpNRT.aiff"               // synthesize
)
```

ID: 334

Operators

SuperCollider supports operator overloading. Operators can thus be applied to a variety of different objects; Numbers, Ugens, Collections, and so on. When operators are applied to ugens they result in BinaryOpUGens or UnaryOpUGens. See the **BinaryOpUGen** overviewhelpfile for details.

Unary Operators

neg .. inversion
reciprocal .. reciprocal
abs .. absolute value
floor .. next lower integer
ceil .. next higher integer
frac .. fractional part
sign .. -1 when a $<$ 0, $+$1 when a $>$ 0, 0 when a is 0
squared .. a*a
cubed .. a*a*a
sqrt .. square root
exp .. exponential
midicps .. MIDI note number to cycles per second
cpsmidi .. cycles per second to MIDI note number
midiratio .. convert an interval in MIDI notes into a frequency ratio
ratiomidi .. convert a frequency ratio to an interval in MIDI notes
dbamp .. decibels to linear amplitude
ampdb .. linear amplitude to decibels
octcps .. decimal octaves to cycles per second
cpsoct .. cycles per second to decimal octaves
log .. natural logarithm
log2 .. base 2 logarithm
log10 .. base 10 logarithm
sin .. sine
cos .. cosine
tan .. tangent
asin .. arcsine
acos .. arccosine
atan .. arctangent
sinh .. hyperbolic sine

cosh .. hyperbolic cosine
tanh .. hyperbolic tangent
distort .. distortion
softclip .. distortion
isPositive .. 1 when a >= 0, else 0
isNegative .. 1 when a < 0, else 0
isStrictlyPositive .. 1 when a > 0, else 0
(36)

Binary Operators

+ .. addition
- .. subtraction
* .. multiplication
/ .. division
% .. float modulo
** .. exponentiation
< .. less than
<= .. less than or equal
> .. greater than
>= .. greater than or equal
== .. equal
!= .. not equal
<! .. return first argument
min .. minimum of two
max .. maximum of two
round .. quantization by rounding
trunc .. quantization by truncation
atan2 .. arctangent
hypot .. hypotenuse sqrt(a*a + b*b)
hypotApx .. hypotenuse approximation
ring1 .. a*b + a or equivalently: a*(b + 1)
ring2 .. a*b + a + b
ring3 .. a*a*b
ring4 .. a*a*b - a*b*b
sumsqr .. a*a + b*b
difsqr .. a*a - b*b
sqrsum .. (a + b)**2
sqrdif .. (a - b)**2
absdif .. fabs(a - b)

thresh .. threshholding { 0 when a < b, a when a >= b }
amclip .. two quadrant multiply { 0 when b <= 0, a*b when b > 0 }
scaleneg .. nonlinear amplification { a when a >= 0, a*b when a < 0 }
clip2 .. bilateral clipping { b when a > b, -b when a < -b, else a }
wrap2 .. bilateral wrapping
fold2 .. bilateral folding
excess .. residual of clipping a - clip2(a,b)
(36)

ID: 335

SuperCollider 3 Synth Server Architecture

Introduction

The SuperCollider 3 Synth Server is a simple but powerful synthesis engine. While synthesis is running, new modules can be created, destroyed and repatched, sample buffers can be created and reallocated. Effects processes can be created and patched into a signal flow dynamically at scheduled times. All running modules are ordered in a tree of nodes that define an order of execution. Patching between modules is done through global audio and control buses.

All commands are received via TCP or UDP using a simplified version of Open Sound Control (OSC). The synth server and its client(s) may be on the same machine or across a network. The synth server does not send or receive MIDI. It is expected that the client will send all control commands. If MIDI is desired, it is up to the client to receive it and convert it to appropriate OSC commands for the synth engine.

Synth definitions are stored in files generated by the SuperCollider language application. Unit generator definitions are Mach-O bundles (not to be confused with CFBundles). The Unit generator API is a simple C interface.

Main Design Concepts

Node
A Node is an addressable node in a tree of nodes run by the synth engine. There are two types, Synths and Groups. The tree defines the order of execution of all Synths. All nodes have an integer ID.

Group
A Group is a collection of Nodes represented as a linked list. A new Node may be added to the head or tail of the group. The Nodes within a Group may be controlled together. The Nodes in a Group may be both Synths and other Groups. At startup there is a top

level group with an ID of zero that defines the root of the tree. If the server was booted from within SCLang (as opposed to from the command line) there will also be a 'default group' with an ID of 1 which is the default target for all new Nodes. See **RootNode** and **default_group** for more info.

Synth
A Synth is a collection of unit generators that run together. They can be addressed and controlled by commands to the synthesis engine. They read input and write output to global audio and control buses. Synths can have their own local controls that are set via commands to the server.

Synth Definition
Synths are created from Synth Definitions. Synth Definition files are created by the SuperCollider language application and are loaded into the synth server. Synth Definitions are referred to by name.

Audio Buses
Synths send audio signals to each other via a single global array of audio buses. Audio buses are indexed by integers beginning with zero. Using buses rather than connecting synths to each other directly allows synths to connect themselves to the community of other synths without having to know anything about them specifically. The lowest numbered buses get written to the audio hardware outputs. Immediately following the output buses are the input buses, read from the audio hardware inputs. The number of bus channels defined as inputs and outputs do not have to match that of the hardware.

Control Buses
Synths can send control signals to each other via a single global array of control buses. Buses are indexed by integers beginning with zero.

Shared Control Buses
The internal server (which runs within the same address space as the client app) also has a number of shared control buses to which the client app has synchronous read/write access. These buses are indexed by integers beginning with zero.

Buffers
Buffers are arrays of 32 bit floating point values with a small descriptive header. Buffers are stored in a single global array indexed by integers beginning with zero. Buffers may be safely allocated, loaded and freed while synthesis is running, even while unit generators are using them. Buffers are used for wave tables, sample buffers, delay lines, envelopes, or for any other need which can use an array of floating point values. Sound files may

be loaded into or written from buffers.

Unit Generator Definitions
Unit Generator Definitions are plug-ins loaded automatically when the program starts. They are binary code libraries that are used as building blocks by Synths to build synthesis algorithms. Unit Generator Definitions have names that match the names of SuperCollider language classes used in building Synth Definitions.

Command Line Arguments

One of -u or -t must be supplied. Both may be supplied.
-u *udp-port-number*
a port number 0-65535.
-t *tcp-port-number*
a port number 0-65535

-v *device-name*
Name of a sound i/o device to use. If not specified, the default device is used.
Currently this is ignored and the default device is used.

-a *num-audio-bus-channels*
number of audio bus channels (default = 128).
The space allocated for audio buses is: (numchannels * (blocksize + 1) * 4)
-i *num-input-bus-channels*
number of audio input bus channels (default = 8)
-o *num-output-bus-channels*
number of audio output bus channels (default = 8)
-c *num-control-bus-channels*
number of control bus channels (default = 4096)
The space allocated for control buses is: (numchannels * 8)
-b *num-buffers*
number of sample buffers (default = 1024)
-n *max-nodes*
maximum number of nodes (default = 1024)
-d *max-synth-defs*
maximum number of synth definitions (default = 1024)
-D *1 or 0*
if zero, then synth definitions will not be loaded on start up. (default = 1)

-z *block-size*
The number of samples in one control period. (default = 64)
-Z *preferred-hardware-buffer-size*
If non-zero, it will attempt to set the hardware buffer frame size. (default = 0)
-S *preferred-sample-rate*
If non-zero, it will attempt to set the hardware sample rate. (default = 0)
-m *real-time-memory-size*
The number of kilobytes of real time memory.
This memory is used to allocate synths and any memory that
unit generators themselves allocate. (default = 8192)
-r *random-number-generators*
The number of seedable random number generators. (default = 64)
-w *max-interconnect-buffers*
The maximum number of buffers that are allocated for buffers to interconnect unit generators. Sets the limit of complexity of synth defs that can be loaded at runtime. This value will be increased if a more complex synth-def is loaded at start up time, but it cannot be increased once synthesis has begun. (default = 64)

-l *max-logins*
maximum number of named return addresses stored (default = 64)
also maximum number of tcp connections accepted
-p *session-password*
When using TCP, the session password must be the first command sent.
The default is no password.
UDP ports never require passwords, so if password protection is desired,
use TCP.

-H *device-name*
name of the hardware I/O device. If not provided, the default device is used.

-I *input-streams-enable-string*
Allows turning off input streams that you are not interested in on the device.
If the string is 01100, for example, then only the second and third input streams
on the device will be enabled. Turning off streams can reduce CPU load.

-O *output-streams-enable-string*
Allows turning off output streams that you are not interested in on the device.
If the string is 11000, for example, then only the first two output streams
on the device will be enabled. Turning off streams can reduce CPU load.

-N *cmd-filename input-filename output-filename sample-rate header-format sample-format*

Run in non-real-time mode.
The *cmd-filename* should be a file that contains OSC bundles sorted in ascending time order. If *cmd-filename* is the underscore character _, then OSC will be streamed from standard input.
The audio input will taken from *input-filename*.
If *input-filename* is the underscore character _, then no input file will be read.
Output will be written to *output-filename*.
The output file's sample rate is specified by *sample-rate*.
The output file *header-format* should be one of: AIFF, WAVE, NeXT.
The output file *sample-format* should be one of: int16, int24, int32, float, double.
The number of channels in the output file is specified with the -o argument.

example:

scscynth -u 57117 >synth_log &

Accept commands via UDP on port 57117.
Send output to file "synth_log"
Run asynchronously: &.

scsynth -N score.osc _ out.aiff 48000 AIFF int24

Run in non real time mode with command file score.osc, no input file, and output file named out.aiff. Sample rate is 48000. Output file header format is aiff, sample format is 24 bit integer.

Binary Format of Messages

Messages are similar in format to Open Sound Control messages, except that OSC #bundles may not be nested, and pattern matching of the command name is not performed. When streamed via TCP, Messages are each preceeded by a 32 bit integer giving the length in bytes of the message. UDP datagrams contain this length information already.

Types:
All values are in network byte order.
long - a 64 bit integer. Used for time stamps only.

int - a 32 bit integer.
float - a 32 bit single precision floating point number.
double - a 64 bit double precision floating point number.
string - a string of 8 bit ASCII characters, zero padded to a multiple of 4 bytes.
bytes - a buffer of data preceeded by a 32 bit length field and padded to a multiple of 4 bytes.

Tags:
Command arguments have single character tags which occur in a tag string to identify their types.
'i' - an int.
'f' - a float
's' - a string
'b' - bytes

a Command consists of:
string - the command name. See the Command Reference below.
string - a string with tags defined by the types of the arguments to follow.
The tag string begins with a comma ',' character.
...any combination of arguments of types: int, float, string or bytes.

a Bundle consists of:
time stamp - long. Time stamps are in the same format as defined by Open Sound Control : The top 32 bits are seconds since 1900 and the lower 32 bits represent the 32 bit fraction of one second.
...a series of Commands each preceded by a 32-bit integer byte length.

a Message consists of:
using UDP :
one Bundle or one Command.

using TCP :
int - the length in bytes of the following message.
one Bundle or one Command.

Glossary

buffer - a header and array of floating point sample data. Buffers are used for sound files, delay lines, arrays of global controls, and arrays of inter-synth patch points.

group - a linked list of nodes. groups provide ways to control execution of many nodes at once. a group is a kind of node.

MIDI - a protocol for sending music control data between synthesizers.

node - an object in a tree of objects executed in a depth first traversal order by the synth engine. There are two types of nodes, synths and groups.

Open Sound Control - a protocol defined by CNMAT at UCBerkeley for controlling synthesizers. See http://cnmat.cnmat.berkeley.edu/OSC/ .

OSC - see Open Sound Control.

synth - a sound processing module. Similar to "voice " in other systems. Synths are referred to by a number.

synth definition - a definition for creating new synths. similar to "instrument" in other systems.

TCP - a protocol for streaming data over a network.

UDP - a protocol for sending datagrams over a network.

ID: 336

SuperCollider Server Synth Engine Command Reference

The following is a list of all server commands and their arguments.

Each command has a command number which can be sent to the server as a 32 bit integer instead of an OSC style string. Command numbers are listed at the end of this document.

If a command's description contains the word **Asynchronous**, then that command will be passed to a background thread to complete so as not to steal CPU time from the audio synthesis thread. All asynchronous commands send a reply to the client when they are completed. Many asynchronous commands can contain an OSC message or bundle to be executed upon completion.

eg.

```
["/d_load", "synthdefs/void.scsyndef",
["/s_new", "void", 1001, 1, 0] // completion message
]
```

Master Controls

/quit quit program
no arguments.

Exits the synthesis server.
Asynchronous. Replies to sender with **/done** just before completion.

/notify register to receive notifications from server
int - one to receive notifications, zero to stop receiving them.

If argument is one, server will remember your return address and send you notifications. if argument is zero, server will stop sending you notifications.
Asynchronous. Replies to sender with **/done** when complete.

/status query the status

no arguments.

Replies to sender with the following message.
/status.reply
int - 1. unused.
int - number of unit generators.
int - number of synths.
int - number of groups.
int - number of loaded synth definitions.
float - average percent CPU usage for signal processing
float - peak percent CPU usage for signal processing
double - nominal sample rate
double - actual sample rate

/cmd plug-in defined command
string - command name
...any arguments

Commands are defined by plug-ins.

/dumpOSC display incoming OSC messages
int - code

Turns on and off printing of the contents of incoming Open Sound Control messages. This is useful when debugging your command stream.
The values for the code are as follows:
0 - turn dumping OFF.
1 - print the parsed contents of the message.
2 - print the contents in hexadecimal.
3 - print both the parsed and hexadecimal representations of the contents.

/sync notify when async commands have completed.
int - a unique number identifying this command.

Replies with a /synced message when all asynchronous commands received before this one have completed. The reply will contain the sent unique ID.
Asynchronous. Replies to sender with **/synced, ID** when complete.

/clearSched clear all scheduled bundles.

Removes all bundles from the scheduling queue.

Synth Definition Commands

/d_recv receive a synth definition file
bytes - buffer of data.
bytes - an OSC message to execute upon completion. (optional)

Loads a file of synth definitions from a buffer in the message. Resident definitions with the same names are overwritten.
Asynchronous. Replies to sender with **/done** when complete.

/d_load load synth definition
string - pathname of file. Can be a pattern like "synthdefs/perc-*"
bytes - an OSC message to execute upon completion. (optional)

Loads a file of synth definitions. Resident definitions with the same names are overwritten.
Asynchronous. Replies to sender with **/done** when complete.

/d_loadDir load a directory of synth definitions
string - pathname of directory.
bytes - an OSC message to execute upon completion. (optional)

Loads a directory of synth definitions files. Resident definitions with the same names are overwritten.
Asynchronous. Replies to sender with **/done** when complete.

/d_free delete synth definition
[
string - synth def name
] * N

Removes a synth definition once all synths using it have ended.

Node Commands

/n_free delete a node.
[
int - node ID
] * N

Stops a node abruptly, removes it from its group, and frees its memory. A list of node IDs may be specified. Using this method can cause a click if the node is not silent at the time it is freed.

/n_run turn node on or off
[
int - node ID
int - run flag
] * N

If the run flag set to zero then the node will not be executed.
If the run flag is set back to one, then it will be executed.
Using this method to start and stop nodes can cause a click if the node is not silent at the time run flag is toggled.

/n_set set a node's control value(s)
int - node ID
[
int or string - a control index or name
float - a control value
] * N

Takes a list of pairs of control indices and values and sets the controls to those values. If the node is a group, then it sets the controls of every node in the group.

/n_setn set ranges of a node's control value(s)
int - node ID
[
int or string - a control index or name

int - number of sequential controls to change (M)
[
float - a control value
] * M
] * N

Set contiguous ranges of control indices to sets of values. For each range, the starting control index is given followed by the number of controls to change, followed by the values. If the node is a group, then it sets the controls of every node in the group.

/n_fill fill ranges of a node's control value(s)
int - node ID
[
int or string - a control index or name
int - number of values to fill (M)
float - value
] * N

Set contiguous ranges of control indices to single values. For each range, the starting control index is given followed by the number of controls to change, followed by the value to fill. If the node is a group, then it sets the controls of every node in the group.

/n_map map a node's controls to read from a bus
int - node ID
[
int or string - a control index or name
int - control bus index
] * N

Takes a list of pairs of control names or indices and bus indices and causes those controls to be read continuously from a global control bus. If the node is a group, then it maps the controls of every node in the group. If the control bus index is -1 then any current mapping is undone. Any **n_set**, **n_setn** and **n_fill** command will also unmap the control.

/n_mapn map a node's controls to read from buses
int - node ID
[
int or string - a control index or name
int - control bus index

int - number of controls to map
] * N

Takes a list of triplets of control names or indices, bus indices, and number of controls to map and causes those controls to be mapped sequentially to buses. If the node is a group, then it maps the controls of every node in the group. If the control bus index is -1 then any current mapping is undone. Any **n_set**, **n_setn** and **n_fill** command will also unmap the control.

/n_before place a node before another
[
int - the ID of the node to place (A)
int - the ID of the node before which the above is placed (B)
] * N

Places node A in the same group as node B, to execute immediately before node B.

/n_after place a node after another
[
int - the ID of the node to place (A)
int - the ID of the node after which the above is placed (B)
] * N

Places node A in the same group as node B, to execute immediately after node B.

/n_query get info about a node
[
int - node ID
] * N

The server sends an **/n_info** message for each node to registered clients.
See **Node Notifications** below for the format of the **/n_info** message.

/n_trace trace a node
[
int - node ID
] * N

Causes a synth to print out the values of the inputs and outputs of its unit generators for one control period. Causes a group to print the node IDs and names of each node

in the group for one control period.

Synth Commands

/s_new create a new synth
string - synth definition name
int - synth ID
int - add action (0,1,2, 3 or 4 see below)
int - add target ID
[
int or string - a control index or name
float - a control value
] * N

Create a new synth from a synth definition, give it an ID, and add it to the tree of nodes. There are four ways to add the node to the tree as determined by the add action argument which is defined as follows:
add actions:
0 - add the new node to the the head of the group specified by the add target ID.
1 - add the new node to the the tail of the group specified by the add target ID.
2 - add the new node just before the node specified by the add target ID.
3 - add the new node just after the node specified by the add target ID.
4 - the new node replaces the node specified by the add target ID. The target node is freed.
Controls may be set when creating the synth. The control arguments are the same as for the **n_set** command.

If you send **/s_new** with a synth ID of -1, then the server will generate an ID for you. The server reserves all negative IDs. Since you don't know what the ID is, you cannot talk to this node directly later. So this is useful for nodes that are of finite duration and that get the control information they need from arguments and buses or messages directed to their group. In addition no notifications are sent when there are changes of state for this node, such as /go, /end, /on, /off.

If you use a node ID of -1 for any other command, such as **/n_map**, then it refers to the most recently created node by **/s_new** (auto generated ID or not). This is how you can map the controls of a node with an auto generated ID. In a multi-client situation, the only way you can be sure what node -1 refers to is to put the messages in a bundle.

/s_get get control value(s)
int - synth ID
[
int or string - a control index or name
] * N

Replies to sender with the corresponding **/n_set** command.

/s_getn get ranges of control value(s)
int - synth ID
[
int or string - a control index or name
int - number of sequential controls to get (M)
] * N

Get contiguous ranges of controls. Replies to sender with the corresponding **/n_setn** command.

/s_noid auto-reassign synth's ID to a reserved value
[
int - synth ID
] * N

This command is used when the client no longer needs to communicate with the synth and wants to have the freedom to reuse the ID. The server will reassign this synth to a reserved negative number. This command is purely for bookkeeping convenience of the client. No notification is sent when this occurs.

Group Commands

/g_new create a new group
[
int - new group ID
int - add action (0,1,2, 3 or 4 see below)
int - add target ID
] * N

Create a new group and add it to the tree of nodes.

There are four ways to add the group to the tree as determined by the add action argument which is defined as follows (the same as for "/s_new"):
add actions:
0 - add the new group to the the head of the group specified by the add target ID.
1 - add the new group to the the tail of the group specified by the add target ID.
2 - add the new group just before the node specified by the add target ID.
3 - add the new group just after the node specified by the add target ID.
4 - the new node replaces the node specified by the add target ID. The target node is freed.
Multiple groups may be created in one command by adding arguments.

/g_head add node to head of group
[
int - group ID
int - node ID
] * N

Adds the node to the head (first to be executed) of the group.

/g_tail add node to tail of group
[
int - group ID
int - node ID
] * N

Adds the node to the tail (last to be executed) of the group.

/g_freeAll delete all nodes in a group.
[
int - group ID
] * N

Frees all nodes in the group. A list of groups may be specified.

/g_deepFree free all synths in this group and all its sub-groups.
[
int - group ID
] * N

Traverses all groups below this group and frees all the synths. Sub-groups are not freed.

A list of groups may be specified.

Unit Generator Commands

/u_cmd send a command to a unit generator
int - node ID
int - unit generator index
string - command name
...any arguments

Sends all arguments following the command name to the unit generator to be performed. Commands are defined by unit generator plug ins.

Buffer Commands

Buffers are stored in a global array, indexed by integers starting at zero.

/b_alloc allocate buffer space.
int - buffer number
int - number of frames
int - number of channels (optional. default = 1 channel)
bytes - an OSC message to execute upon completion. (optional)

Allocates zero filled buffer to number of channels and samples.
Asynchronous. Replies to sender with **/done** when complete.

/b_allocRead allocate buffer space and read a sound file.
int - buffer number
string - path name of a sound file.
int - starting frame in file (optional. default = 0)
int - number of frames to read (optional. default = 0, see below)
bytes - an OSC message to execute upon completion. (optional)

Allocates buffer to number of channels of file and number of samples requested, or fewer if sound file is smaller than requested. Reads sound file data from the given starting frame in the file. If the number of frames argument is less than or equal to zero, the entire file is read.
Asynchronous. Replies to sender with **/done** when complete.

/b_allocReadChannel allocate buffer space and read channels from a sound file.
int - buffer number
string - path name of a sound file
int - starting frame in file
int - number of frames to read
[
int - source file channel index
] * N N >= 0
bytes - an OSC message to execute upon completion. (optional)

As b_allocRead, but reads individual channels into the allocated buffer in the order specified.
Asynchronous. Replies to sender with **/done** when complete.

/b_read read sound file data into an existing buffer.
int - buffer number
string - path name of a sound file.
int - starting frame in file (optional. default = 0)
int - number of frames to read (optional. default = -1, see below)
int - starting frame in buffer (optional. default = 0)
int - leave file open (optional. default = 0)
bytes - an OSC message to execute upon completion. (optional)

Reads sound file data from the given starting frame in the file and writes it to the given starting frame in the buffer. If number of frames is less than zero, the entire file is read. If reading a file to be used by DiskIn ugen then you will want to set "leave file open" to one, otherwise set it to zero.
Asynchronous. Replies to sender with **/done** when complete.

/b_readChannel read sound file channel data into an existing buffer
int - buffer number
string - path name of a sound file
int - starting frame in file
int - number of frames to read
int - starting frame in buffer
int - leave file open
[
int - source file channel index

] * N N >= 0
bytes - completion message

As b_read, but reads individual channels in the order specified. The number of channels requested must match the number of channels in the buffer.
Asynchronous. Replies to sender with **/done** when complete.

/b_write write sound file data.
int - buffer number
string - path name of a sound file.
string - header format.
string - sample format.
int - number of frames to write (optional. default = -1, see below)
int - starting frame in buffer (optional. default = 0)
int - leave file open (optional. default = 0)
bytes - an OSC message to execute upon completion. (optional)

Write a buffer as a sound file.
Header format is one of:
"aiff", "next", "wav", "ircam"", "raw"
Sample format is one of:
"int8", "int16", "int24", "int32", "float", "double", "mulaw", "alaw"
Not all combinations of header format and sample format are possible.
If number of frames is less than zero, all samples from the starting frame to the end of the buffer are written.
If opening a file to be used by DiskOut ugen then you will want to set "leave file open" to one, otherwise set it to zero. If "leave file open" is set to one then the file is created, but no frames are written until the DiskOut ugen does so.
Asynchronous. Replies to sender with **/done** when complete.

/b_free free buffer data.
int - buffer number
bytes - an OSC message to execute upon completion. (optional)

Frees buffer space allocated for this buffer.
Asynchronous. Replies to sender with **/done** when complete.

/b_zero zero sample data
int - buffer number
bytes - an OSC message to execute upon completion. (optional)

Sets all samples in the buffer to zero.
Asynchronous. Replies to sender with **/done** when complete.

/b_set set sample value(s)
int - buffer number
[
int - a sample index
float - a sample value
] * N

Takes a list of pairs of sample indices and values and sets the samples to those values.

/b_setn set ranges of sample value(s)
int - buffer number
[
int - sample starting index
int - number of sequential samples to change (M)
[
float - a sample value
] * M
] * N

Set contiguous ranges of sample indices to sets of values. For each range, the starting sample index is given followed by the number of samples to change, followed by the values.

/b_fill fill ranges of sample value(s)
int - buffer number
[
int - sample starting index
int - number of samples to fill (M)
float - value
] * N

Set contiguous ranges of sample indices to single values. For each range, the starting sample index is given followed by the number of samples to change, followed by the value to fill. This is only meant for setting a few samples, not whole buffers or large sections.

/b_gen call a command to fill a buffer
int - buffer number
string - command name
.. command arguments

Plug-ins can define commands that operate on buffers. The arguments after the command name are defined by the command. The currently defined buffer fill commands are listed below in a separate section.

/b_close
int - buffer number

After using a buffer with DiskOut, close the soundfile and write header information.

/b_query
[
int - buffer number
] * N

Responds to the sender with a **/b_info** message. The arguments to **/b_info** are as follows:
[
int - buffer number
int - number of frames
int - number of channels
float - sample rate
] * N

/b_get get sample value(s)
int - buffer number
[
int - a sample index
] * N

Replies to sender with the corresponding **/b_set** command.

/b_getn get ranges of sample value(s)
int - buffer number
[
int - starting sample index

int - number of sequential samples to get (M)
] * N

Get contiguous ranges of samples. Replies to sender with the corresponding **/b_setn** command. This is only meant for getting a few samples, not whole buffers or large sections.

Control Bus Commands

/c_set set bus value(s)
[
int - a bus index
float - a control value
] * N

Takes a list of pairs of bus indices and values and sets the buses to those values.

/c_setn set ranges of bus value(s)
[
int - starting bus index
int - number of sequential buses to change (M)
[
float - a control value
] * M
] * N

Set contiguous ranges of buses to sets of values. For each range, the starting bus index is given followed by the number of channels to change, followed by the values.

/c_fill fill ranges of bus value(s)
[
int - starting bus index
int - number of buses to fill (M)
float - value
] * N

Set contiguous ranges of buses to single values. For each range, the starting sample index is given followed by the number of buses to change, followed by the value to fill.

/c_get get bus value(s)
[
int - a bus index
] * N

Takes a list of buses and replies to sender with the corresponding **/c_set** command.

/c_getn get ranges of bus value(s)
[
int - starting bus index
int - number of sequential buses to get (M)
] * N

Get contiguous ranges of buses. Replies to sender with the corresponding **/c_setn** command.

Non Real Time Mode Commands

/nrt_end end real time mode, close file
#ff0303**not yet implemented**
no arguments.

This message should be sent in a bundle in non real time mode.
The bundle timestamp will establish the ending time of the file.
This command will end non real time mode and close the sound file.
Replies to sender with **/done** when complete.

Replies to Commands

These messages are sent by the server in reponse to some commands.

/done an asynchronous message has completed.
string - the name of the command

Sent in response to all asynchronous commands. Sent only to the sender of the original

message.

/fail an error occurred.
string - the name of the command
string - the error message.

There was a problem. Sent only to the sender of the original message.

/late a command was received too late.
#ff0303**not yet implemented**
int - the high 32 bits of the original time stamp.
int - the low 32 bits of the original time stamp.
int - the high 32 bits of the time it was executed.
int - the low 32 bits of the time it was executed.

The command was received too late to be executed on time. Sent only to the sender of the original message.

Notifications from Server

These messages are sent as notification of some event to all clients who have registered via the /notify command .

Node Notifications

All of these have the same arguments:
int - node ID
int - the node's parent group ID
int - previous node ID, -1 if no previous node.
int - next node ID, -1 if no next node.
int - 1 if the node is a group, 0 if it is a synth

The following two arguments are only sent if the node is a group.
int - the ID of the head node, -1 if there is no head node.
int - the ID of the tail node, -1 if there is no tail node.

/n_go a node was started

This command is sent to all registered clients when a node is created.

/n_end a node ended

This command is sent to all registered clients when a node ends and is deallocated.

/n_off a node was turned off

This command is sent to all registered clients when a node is turned off.

/n_on a node was turned on

This command is sent to all registered clients when a node is turned on.

/n_move a node was moved

This command is sent to all registered clients when a node is moved.

/n_info reply to /n_query

This command is sent to all registered clients in response to an **/n_query** command.

Trigger Notification

This command is sent to all registered clients when a node is moved from one group to another.

/tr a trigger message
int - node ID
int - trigger ID
float - trigger value

This command is the mechanism that synths can use to trigger events in clients.
The node ID is the node that is sending the trigger. The trigger ID and value are determined by inputs to the SendTrig unit generator which is the originator of this message.

Buffer Fill Commands

These are the currently defined fill routines for use with the **/b_gen** command.

Wave Fill Commands

There are three defined fill routines for sine waves.

The flags are defined as follows:
1 - normalize - Normalize peak amplitude of wave to 1.0.
2 - wavetable - If set, then the buffer is written in wavetable format so that it can be read by interpolating oscillators.
4 - clear - if set then the buffer is cleared before new partials are written into it. Otherwise the new partials are summed with the existing contents of the buffer.

sine1
int - flags, see above
[
float - partial amplitude
] * N

Fills a buffer with a series of sine wave partials. The first float value specifies the amplitude of the first partial, the second float value specifies the amplitude of the second partial, and so on.

sine2
int - flags, see above
[
float - partial frequency (in cycles per buffer)
float - partial amplitude
] * N

Similar to sine1 except that each partial frequency is specified explicitly instead of being an integer series of partials. Non-integer partial frequencies are possible.

sine3
int - flags, see above
[
float - partial frequency (in cycles per buffer)
float - partial amplitude

float - partial phase
] * N

Similar to sine2 except that each partial may have a nonzero starting phase.

cheby
int - flags, see above
[
float - amplitude
] * N

Fills a buffer with a series of chebyshev polynomials, which can be defined as:
cheby(n) = amplitude * cos(n * acos(x))
The first float value specifies the amplitude for n = 1, the second float value specifies the amplitude for n = 2, and so on. To eliminate a DC offset when used as a waveshaper, the wavetable is offset so that the center value is zero.

Other Commands

copy
int - sample position in destination
int - source buffer number
int - sample position in source
int - number of samples to copy

Copy samples from the source buffer to the destination buffer specified in the **b_gen** command. If the number of samples to copy is negative, the maximum number of samples possible is copied.
Asynchronous. Replies to sender with **/done** when complete.

Command Numbers

These are the currently defined command numbers. More may be added to the end of the list in the future.

```
#760f50enum
cmd_none = #0000ff0,
```

```
cmd_notify = #0000ff1,
cmd_status = #0000ff2,
cmd_quit = #0000ff3,
cmd_cmd = #0000ff4,

cmd_d_recv = #0000ff5,
cmd_d_load = #0000ff6,
cmd_d_loadDir = #0000ff7,
cmd_d_freeAll = #0000ff8,

cmd_s_new = #0000ff9,

cmd_n_trace = #0000ff10,
cmd_n_free = #0000ff11,
cmd_n_run = #0000ff12,
cmd_n_cmd = #0000ff13,
cmd_n_map = #0000ff14,
cmd_n_set = #0000ff15,
cmd_n_setn = #0000ff16,
cmd_n_fill = #0000ff17,
cmd_n_before = #0000ff18,
cmd_n_after = #0000ff19,

cmd_u_cmd = #0000ff20,

cmd_g_new = #0000ff21,
cmd_g_head = #0000ff22,
cmd_g_tail = #0000ff23,
cmd_g_freeAll = #0000ff24,

cmd_c_set = #0000ff25,
cmd_c_setn = #0000ff26,
cmd_c_fill = #0000ff27,

cmd_b_alloc = #0000ff28,
cmd_b_allocRead = #0000ff29,
cmd_b_read = #0000ff30,
cmd_b_write = #0000ff31,
cmd_b_free = #0000ff32,
```

```
cmd_b_close = #0000ff33,
cmd_b_zero = #0000ff34,
cmd_b_set = #0000ff35,
cmd_b_setn = #0000ff36,
cmd_b_fill = #0000ff37,
cmd_b_gen = #0000ff38,

cmd_dumpOSC = #0000ff39,

cmd_c_get = #0000ff40,
cmd_c_getn = #0000ff41,
cmd_b_get = #0000ff42,
cmd_b_getn = #0000ff43,
cmd_s_get = #0000ff44,
cmd_s_getn = #0000ff45,

cmd_n_query = #0000ff46,
cmd_b_query = #0000ff47,

cmd_n_mapn = #0000ff48,
cmd_s_noid = #0000ff49,

cmd_g_deepFree = #0000ff50,
cmd_clearSched = #0000ff51,

cmd_sync = #0000ff52,

cmd_d_free = #0000ff53,

NUMBER_OF_COMMANDS = #0000ff54
#0000ff}
```

ID: 337

SuperCollider 3 Synth Definition File Format

Synth definition files are read by the synth server and define collections of unit generators and their connections. These files are currently written by the SuperCollider language application, but theoretically could be written by any program. Such a program would need knowledge of the SC unit generators and their characteristics, such as number of inputs and outputs and available calculation rates. The code to write these files is open and available in the SuperCollider language app.

Basic types

All data is stored big endian. All data is packed, not padded or aligned.
an **int32** is a 32 bit integer.
an **int16** is a 16 bit integer.
an **int8** is an 8 bit integer.
a **float32** is a 32 bit IEEE floating point number.
a **pstring** is a pascal format string: a byte giving the length followed by that many bytes.

File Format

a **synth-definition-file** is :
int32 - four byte file type id containing the ASCII characters: "SCgf"
int32 - file version, currently zero.
int16 - number of synth definitions in this file (D).
[**synth-definition**] * D
end

a **synth-definition** is :
pstring - the name of the synth definition

int16 - number of constants (K)
[**float32**] * K - constant values

int16 - number of parameters (P)
[**float32**] * P - initial parameter values

int16 - number of parameter names (N)
[**param-name**] * N

int16 - number of unit generators (U)
[**ugen-spec**] * U
end

a **param-name** is :
pstring - the name of the parameter
int16 - its index in the parameter array
end

a **ugen-spec** is :
pstring - the name of the SC unit generator class
int8 - calculation rate
int16 - number of inputs (I)
int16 - number of outputs (O)
int16 - special index
[**input-spec**] * I
[**output-spec**] * O
end

an **input-spec** is :
int16 - index of unit generator or -1 for a constant
if (unit generator index == -1) {
int16 - index of constant
} else {
int16 - index of unit generator output
}
end

an **output-spec** is :
int8 - calculation rate
end

Glossary

calculation rate - the rate that a computation is done. There are three rates numbered 0, 1, 2 as follows:
0 = scalar rate - one sample is computed at initialization time only. **1 = control rate** - one sample is computed each control period.
2 = audio rate - one sample is computed for each sample of audio output.
Outputs have their own calculation rate. This allows MultiOutUGens to have outputs at different rates. A one output unit generator's calculation rate should match that of its output.

constant - a single floating point value that is used as a unit generator input.

parameter - a value that can be externally controlled using server commands /s.new, /n.set, /n.setn, /n.fill, /n.map .

parameter name - a string naming an index in the the parameter array. This allows one to refer to the same semantic value such as 'freq' or 'pan' in different synths even though they exist at different offsets in their respective parameter arrays.

special index - this value is used by some unit generators for a special purpose. For example, UnaryOpUGen and BinaryOpUGen use it to indicate which operator to perform. If not used it should be set to zero.

synth - a collection of unit generators that execute together. A synth is a type of node.

synth definition - a specification for creating synths.

unit generator - a basic signal processing module with inputs and outputs. unit generators are connected together to form synths.

Notes

Unit generators are listed in the order they will be executed. Inputs must refer to constants or previous unit generators. No feedback loops are allowed. Feedback must be accomplished via delay lines or through buses.

For greatest efficiency:

Unit generators should be listed in an order that permits efficient reuse of connection buffers, which means that a depth first topological sort of the graph is preferable to

breadth first.

There should be no duplicate values in the constants table.

ID: 338

SuperCollider 3 Server Tutorial

To follow this tutorial you should read

Server-Architecture
and
Server-Command-Reference

This tutorial also assumes that you are familiar with SuperCollider version 2 since the creating a SynthDef in SC3 is very similar to creating a Synth in SC2.

There are two parts to SuperCollider. One part is the language application and another is a synthesis server that can run either inside the language application, or as a separate program on the same machine, or run on a different computer across a network connection. The language application sends command messages to the server using a subset of the Open Sound Control protocol.

Booting a Server

In order to run sound we need to start a server running. The easiest way to start a server is to click on one of the "Start Server" buttons in the server windows. Sometimes though it is useful to start a server programmatically. To do this we need to get or create a server object and tell it to "boot". Two servers, internal and local, are predefined.

The internal server runs in the same process as the SuperCollider application. It is internal to the program itself.

```
// set the interpreter variable s to the internal server object.
s = Server.internal;
```

VERY IMPORTANT: This line must be executed for the variable 's' to be set. The mechanics are different depending on your platform. The MacOSX standard is to place the cursor anywhere on this line and press the "Enter" key on the numeric keypad. Pressing the main return key does not execute code! This allows you to write code fragments of multiple lines. To execute a multi-line block of code, select the block and press "Enter." For convenience, a code block can be enclosed in parentheses, and the entire block selected by double-clicking just inside either parenthesis. (For linux or Windows instructions, consult the documentation specific to that platform.)

The local server runs on the same machine as the SuperCollider application, but is a separate program, 'scsynth'. **Note:** By default the interpreter variable s is set to the local server at startup. For further information see the **Server** helpfile.

```
// set the interpreter variable s to the local server object.
s = Server.local;
```

To boot the server you send it the boot message.

```
s.boot;
```

To quit the server send it the quit message.

```
s.quit;
```

We can also create a server to run. To create a server object we need to provide the IP address or the server and a port number. Port numbers are somewhat arbitrary but they should not conflict with common protocols like telnet, ftp http, etc. The IP address 127.0.0.1 is defined to mean the local host. This is the IP address to use for running a server on your own machine.

```
// create a server object that will run on the local host using port #58009
s = Server(\myServer, NetAddr("127.0.0.1", 58009));

        //start the server

        // quit the server
```

It is not possible to boot a server on a remote machine, but if you have one running already or you know of one running, you can send messages to it. You create the server object using the IP address of the machine running the server and the port it is using.

```
// create a server object for talking to the server running on a machine having
// IP address 192.168.0.47 using port #57110
s = Server(\myServer, NetAddr("192.168.0.47", 57110));
```

Making Sound

(note: This tutorial uses raw OSC commands as described in **Server-Command-Reference,** rather than the classes **Synth** and **Group**. See those helpfiles also for some simpler ways of working with Synths. This tutorial explains the basic underlying design of Synths and SynthDefs).

Now lets make some audio.

```
    Server        // assign it to interpreter variable 's'
```

Boot it.

```
s.boot;
```

Create a SynthDef. A SynthDef is a description of a processing module that you want to run on the server. It can read audio from the server's audio buses, read control from the control buses and write control or audio back to buses. Here we will create a sine oscillator and send it to audio bus zero.

```
(
SynthDef("sine", { arg freq=800;
var osc;
osc = SinOsc.ar(freq, 0, 0.1); // 800 Hz sine oscillator
 Out             // send output to audio bus zero.
                 // write the def to disk in the default directory synthdefs/
)
```

Send the SynthDef to the server.

```
s.sendSynthDef("sine");
```

Start the sound. The **/s_new** command creates a new Synth which is an instance of the "sine" SynthDef. Each synth running on the server needs to have a unique ID. The simplest and safest way to do this is to get an ID from the server's NodeIDAllocator. This will automatically allow IDs to be reused, and will prevent conflicts both with your own nodes, and with nodes created automatically for purposes such as visual scoping and recording. Each synth needs to be installed in a Group. We install it in group one which is the default group. There is a group zero, called the RootNode, which contains the default group, but it is generally best not to use it as doing so can result in order of execution issues with automatically created nodes such as those mentioned above. (For more detail see the **default_group**, **RootNode**, and **Order-of-execution** helpfiles.)

```
s.sendMsg("/s_new", "sine", x = s.nextNodeID, 1, 1);
```

Stop the sound.

```
s.sendMsg("/n_free", x);
```

Stop the server.

```
s.quit;
```

SynthDef has two methods which send the def automatically, load which writes it to disk, and send which sends it without writing it to disk. The latter can be useful to avoid clutter on your drive.

```
(
SynthDef("sine", { arg freq=800;
var osc;
osc = SinOsc.ar(freq, 0, 0.1); // 800 Hz sine oscillator
 Out             // send output to audio bus zero.
            // write to disk and send
)

(
SynthDef("sine", { arg freq=800;
var osc;
osc = SinOsc.ar(freq, 0, 0.1); // 800 Hz sine oscillator
 Out             // send output to audio bus zero.
            // send without writing
)
```

Using Arguments

It is useful to be able to specify parameters of a synth when it is created. Here a frequency argument is added to the sine SynthDef so that we can create it

```
    Server        // assign it to interpreter variable 's'

s.boot;
```

```
(
SynthDef("sine", { arg freq;
var osc;
osc = SinOsc.ar(freq, 0, 0.1); // 800 Hz sine oscillator
 Out             // send output to audio bus zero.
}).send(s);
)
```

Play a 900 Hz sine wave.

```
s.sendMsg("/s_new", "sine", x = s.nextNodeID, 1, 1, "freq", 900);

s.sendMsg("/n_free", x);
```

Play a 1000 Hz sine wave.

```
s.sendMsg("/s_new", "sine", y = s.nextNodeID, 1, 1, "freq", 1000);

s.sendMsg("/n_free", y);
```

Playing three voices at once

```
(
s.sendMsg("/s_new", "sine", x = s.nextNodeID, 1, 1, "freq", 800);
s.sendMsg("/s_new", "sine", y = s.nextNodeID, 1, 1, "freq", 1001);
s.sendMsg("/s_new", "sine", z = s.nextNodeID, 1, 1, "freq", 1202);
)

(
s.sendMsg("/n_free", x);
s.sendMsg("/n_free", y);
s.sendMsg("/n_free", z);
)
```

Playing three voices at once using bundles. Bundles allow you to send multiple messages

with a time stamp. The messages in the bundle will be scheduled to be performed together. The time argument to sendBundle is an offset into the future from the current thread's logical time.

```
(
s.sendBundle(0.2,
["/s_new", "sine", x = s.nextNodeID, 1, 1, "freq", 800],
["/s_new", "sine", y = s.nextNodeID, 1, 1, "freq", 1001],
["/s_new", "sine", z = s.nextNodeID, 1, 1, "freq", 1202]);
s.sendBundle(1.2, ["/n_free", x],["/n_free", y],["/n_free", z]);
)
```

Controlling a Synth

You can send messages to update the values of a Synth's arguments.

Play a 900 Hz sine wave.

```
s.sendMsg("/s_new", "sine", x = s.nextNodeID, 1, 1, "freq", 900);
```

Change the frequency using the **/n_set** command. You send the node ID, the parameter name and the value.

```
s.sendMsg("/n_set", x, "freq", 800);

s.sendMsg("/n_set", x, "freq", 700);

s.sendMsg("/n_free", x);
```

Adding an Effect Dynamically

You can dynamically add and remove an effect to process another synth. In order to do this, the effect has to be added after the node to be processed.

```
(
// define a noise pulse
SynthDef("tish", { arg freq = 1200, rate = 2;
var osc, trg;
trg = Decay2.ar(Impulse.ar(rate,0,0.3), 0.01, 0.3);
osc = {WhiteNoise.ar(trg)}.dup;
```

```
 Out             // send output to audio bus zero.
}).send(s);
)

(
// define an echo effect
SynthDef("echo", { arg delay = 0.2, decay = 4;
var in;
in = In.ar(0,2);
 // use ReplaceOut to overwrite the previous contents of the bus.
ReplaceOut.ar(0, CombN.ar(in, 0.5, delay, decay, 1, in));
}).send(s);
)

// start the pulse
s.sendMsg("/s_new", "tish", x = s.nextNodeID, 1, 1, \freq, 200, \rate, 1.2);

// add an effect
s.sendMsg("/s_new", "echo", y = s.nextNodeID, 1, 1);

// stop the effect
s.sendMsg("/n_free", y);

// add an effect (time has come today..  hey!)
s.sendMsg("/s_new", "echo", z = s.nextNodeID, 1, 1, \delay, 0.1, \decay, 4);

// stop the effect
s.sendMsg("/n_free", z);

// stop the pulse
s.sendMsg("/n_free", x);
```

This works because we added the effect after the other node. Sometimes you will need to use groups or /n_after to insure that an effect gets added after what it is supposed to process.

Chaining Effects

Using Control Buses

Mapping an Argument to a Control Bus

```
(
// define a control
SynthDef("line", { arg i_bus=10, i_start=1000, i_end=500, i_time=1;
ReplaceOut.kr(i_bus, Line.kr(i_start, i_end, i_time, doneAction:  2));
}).send(s)
)
```

Play a 900 Hz sine wave.

```
s.sendMsg("/s_new", "sine", x = s.nextNodeID, 1, 1, "freq", 900);
```

Put a frequency value on the control bus.

```
s.sendMsg("/c_set", 10, x);
```

Map the node's freq argument to read from control bus #10.

```
s.sendMsg("/n_map", x, \freq, 10);
```

Change the value on the control bus.

```
s.sendMsg("/c_set", 10, 1200);
```

Start a control process that writes to bus #10.
The EnvGen doneAction will free this node automatically when it finishes.

```
s.sendMsg("/s_new", "line", s.nextNodeID, 0, 1);
```

Free the node.

```
s.sendMsg("/n_free", x);
```

Gating Envelopes

Adding a GUI

Using Buffers

Filling Wavetables

Frequency Domain Processing

Sequencing with Routines

```
(
var space,offset,timer, saw, envsaw, sampler, delay;

SynthDef("saw",{ arg out=100, pan=0, trig=0.0, freq=500, amp=1, cutoff=10000, rezz=1;
freq = Lag.kr(freq,0.1);
Out.ar(out,Pan2.ar(RLPF.ar(Saw.ar([freq,freq*2],amp),cutoff,rezz),
pan));
}).load(s);

SynthDef("envsaw",{ arg out=100, pan=0, dur=0.5, freq=500, amp=1, cutoff=10000, rezz=1;
var env;
env = EnvGen.kr(Env.perc(0.01, dur, 0.2), doneAction:0, gate:amp);
Out.ar(out,Pan2.ar(RLPF.ar(Saw.ar(Lag.kr(freq,0.1),env),cutoff,rezz)*amp,
pan));
}).load(s);

SynthDef("delay", { arg out=0, delay = 0.4, decay = 14;
var in;
in = In.ar(out,2);
Out.ar(out, CombN.ar(in, 0.5, delay, decay, 1, in));
}).load(s);

SynthDef("sampler",{ arg sample, trig=1,rate=1.0,out=0,bufnum=0,pan=0,amp=1, dur=0.25;
var env;
env = EnvGen.kr(Env.perc(0.001, dur, 0.001), doneAction:2);
Out.ar(out,
Pan2.ar(
PlayBuf.ar(1,bufnum,rate,InTrig.kr(trig),0,0)*amp,
pan);
)
}).load(s);
```

```
Tempo.bpm = 120;
timer=BeatSched.new;
offset = Tempo.tempo.reciprocal;

        Buffer        "sounds/a11wlk01.wav"

saw=Synth("saw");
      Synth.after     "delay", [\decay, 20]

timer.sched(0,{
 var r;
 r=Routine({ var wait, freq, cutoff,rezz;
wait = Pseq([2],inf).asStream;
freq = Pseq([30,40,42,40],inf).asStream;
cutoff = Pfunc({500.rand2+1000}).asStream;
rezz = 0.5;
inf.do({saw.set("freq", freq.next.midicps, "cutoff", cutoff.next, "rezz", rezz, "amp", 0.1, "out", 0);

(wait.next*offset).wait});});
timer.sched(0,r);
});

timer.sched(0,{
 var r;
 r=Routine({ var wait, rate;
wait = Pseq([0.25],inf).asStream;
rate = Pfunc({0.5.rand}).asStream;
inf.do({Synth.before(delay, "sampler", [\bufnum, space.bufnum, \trig, 1, \amp,0.1, \rate, rate.next, \dur,
wait.next]);
(wait.next*offset).wait});});
timer.sched(0,r);
});

)
```

Sequencing with Patterns

```
(
//sappy emo electronica example...
Tempo.bpm = 120;
```

```
SynthDef("patternefx_Ex", { arg out, in;
var audio, efx;
audio = In.ar([20,21],2);
efx=CombN.ar(audio, 0.5, [0.24,0.4], 2, 1);
Out.ar([0,1], audio+efx);
}).load(s);

Synth     "patternefx_Ex"

SynthDef("pattern_Ex", { arg out, freq = 1000, gate = 1, pan = 0, cut = 4000, rez = 0.8, amp = 1;
Out.ar(out,
Pan2.ar(
RLPF.ar(
Pulse.ar(freq,0.05),
cut, rez),
pan) * EnvGen.kr(Env.linen(0.01, 1, 0.3), gate, amp, doneAction:2);
)
}).load(s);

SynthDef("bass_Ex", { arg out, freq = 1000, gate = 1, pan = 0, cut = 4000, rez = 0.8, amp = 1;
Out.ar(out,
Pan2.ar(
RLPF.ar(
SinOsc.ar(freq,0.05),
cut, rez),
pan) * EnvGen.kr(Env.linen(0.01, 1, 0.3), gate, amp, doneAction:2);
)
}).load(s);

SynthDescLib.global.read;


Pseq

Ptpar
0,Pbind(\instrument,\pattern_Ex, \out, 20, \dur,Pseq([2],16), \root,[-24,-17], \degree,Pseq([0,3,5,7,9,11,5,1],2),
\pan,1,\cut,Pxrand([1000,500,2000,300],16), \rez,Pfunc({0.7.rand +0.3}), \amp,0.12),

0.5,Pbind(\instrument,\pattern_Ex, \out, 20, \dur,Pseq([Pseq([2],15),1.5],1),  \root,-12, \degree,Pseq([0,3,5,7,9,11,5,1],2)
\pan,-1,\cut,2000, \rez,0.6, \amp,0.1);
```

```
]),

Ptpar
0,Pbind(\instrument,\pattern_Ex, \out, 20, \dur,2, \root,[-24,-17], \degree,Pseq([0,3,5,7,9,11,5,1],inf),
\pan,1,\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}), \amp,0.12),

  Pbind \instrument \bass_Ex  \dur    \root      \degree Pseq     inf   \pan    \cut      \rez      \amp


0.5,Pbind(\instrument,\pattern_Ex, \out, 20, \dur,2, \root,-12, \degree,Pseq([0,3,5,7,9,11,5,1],inf),
\pan,-1,\cut,2000, \rez,0.6, \amp,0.1);
]);


]).play;

)
```

ID: 339

Unit Generator Plug-In Example

Unit generator plug-ins will be described in another document. But for an example of what one looks like, here is the complete source to a plug-in for a sample-and-hold unit generator called Latch.

```
////////////////////////////////////////////////////////////////////////

        "SC_PlugIn.h"

      InterfaceTable

////////////////////////////////////////////////////////////////////////

struct Latch :  public Unit
{
float mLevel, m_prevtrig;
};


extern "C"
{
void start();
void load(InterfaceTable *inTable);

void Latch_Ctor(Latch *unit);
void Latch_next_ak(Latch *unit, int inNumSamples);
void Latch_next_aa(Latch *unit, int inNumSamples);
}

// Codewarrior's linker has a bug that demands this function be defined...
void start() {}

////////////////////////////////////////////////////////////////////////

    Latch_Ctor Latch
{
```

```
if (INRATE(1) == calc_FullRate) {
  SETCALC Latch_next_aa
} else {
  SETCALC Latch_next_ak
}

unit->m_prevtrig = 0.f;
unit->mLevel = 0.f;

ZOUT0(0) = 0.f;
}


void Latch_next_ak(Latch *unit, int inNumSamples)
{
float *out = ZOUT(0);
float level = unit->mLevel;

float curtrig = ZIN0(1);
if (unit->m_prevtrig <= 0.f &amp;&amp; curtrig > 0.f) level = ZIN0(0);

LOOP(inNumSamples, *++out = level; );

unit->m_prevtrig = curtrig;
unit->mLevel = level;
}


void Latch_next_aa(Latch *unit, int inNumSamples)
{
float *out = ZOUT(0);
float *in = ZIN(0);
float *trig = ZIN(1);
float prevtrig = unit->m_prevtrig;
float level = unit->mLevel;

LOOP(inNumSamples,
float curtrig = *++trig;
if (prevtrig <= 0.f &amp;&amp; curtrig > 0.f) level = *++in;
else { ++in; }
```

```
*++out = level;
prevtrig = curtrig;
);
unit->m_prevtrig = prevtrig;
unit->mLevel = level;
}

////////////////////////////////////////////////////////////////////////////

void load(InterfaceTable *inTable)
{
ft = inTable;

 DefineSimpleUnit Latch

}

////////////////////////////////////////////////////////////////////////////
```


Adding a Target to ProjectBuilder

Each group of plugins shares a target in ProjectBuilder. They create a combined file: eg. YourGroup.scx which is then copied into the plugins folder via a little shell script. The aggregate target builds all of the targets, each of which looks in the build directory to see if an up to date 'YourGroup.scx' is there. So you should not delete or move (rather than copy) the .scx file, or it will rebuild it each time.

Create a new target, select type 'Library'
Select the target and add your file 'YourGroup.cpp' to it by checking the box beside it in Files.
Go to expert view and set Library_Style to BUNDLE.
set OTHER_CFLAGS to -DSC_DARWIN
Set the product name to be YourGroup.scx.
Add a build phase (control or right click on build phases): a shell script build phase.
copy the one line script from one of the other targets, changing the filename to match yours.

Uncheck "run only when installing".
Open the "disclosure triangle" for the All Plugins target and drag your target into that list.

Your target will be built along with the others when the aggregate target is selected and built.

ID: 340

Undocumented Classes

Below is an alphabetical list of all classes which have no help files, as of October 17, 2004. This includes classes from the **CRUCIAL-LIBRARY** and **JITLib**. Note that many of these are either private classes not intended for direct use, abstract superclasses (such as Clock), or currently non-functioning or vestigal classes (such as the image synthesis classes from SC3d5). Nevertheless this is a good place to look for undocumented functionality. Note that some of these classes are covered in overviews, tutorials, etc.

```
AbstractConsole
AbstractIn
AbstractNodeWatcher
AbstractOpPlug
AbstractOut
AbstractPlayControl
AbstractPlayerEffect
AbstractSample
AbstractSFP
AbstractSFPGui
AbstractSinglePlayerEffect
AnnotatedDebugNodeWatcher
Any
APF
Archive
ArgNameLabel
Array2D
ArraySpec
AudioPatchIn
AudioPatchOut
AudioSpec
AutoCompClassBrowser
AutoCompClassSearch
AutoCompMethodBrowser
Balance2
BasicNodeWatcher
BasicOpUGen
BeatClockPlayerGui
BinaryOpFailureError
```

BinaryOpPlug

BinaryOpXStream

BooleanEditorGui

BroadcastServer

BufAllpassC

BufAllpassL

BufAllpassN

BufCombC

BufCombL

BufCombN

BufDelayC

BufDelayL

BufDelayN

BufferProxySpec

BufInfoUGenBase

BufSamples

BundleNetAddr

BusDriver

BusSynthDefs

CCPlayer

CCResponder

ClassBrowser

ClassGui

ClassInspector

ClassNameLabel

CleanupStream

ClientFunc

Clip

Clock

CmdPeriod

CollStream

CompanderD

Condition

ControlName

ControlPatchIn

ControlPatchOut

ControlRate

CosineWarp

CurveWarp

CXAbstractLabel

CXBundle

CXLabel

CXObjectInspector

CXPlayerControl

CXSynthPlayerControl

Date

DbFaderWarp

Dbrown

DebugFrame

DebugNodeWatcher

Def

Dgeom

Dibrown

Diwhite

Do

DoesNotUnderstandError

Done

Drand

Dseq

Dser

Dseries

Dswitch1

DualSeriesEfxGui

Dwhite

Dxrand

Editor

EditorGui

EnvEditor

EnvEditorGui

EnvGate

EnvirDocument

EnvironmentRedirect

EnvSpec

Error

Every

Exception

ExponentialWarp

EZNumber

FaderWarp

Filter

FilterPattern

Finalizer

FlowLayout

Fold

FrameInspector

FreeSelfWhenDone

FuncFilterPattern

FuncStream

FuncStreamAsRoutine

FunctionDefInspector

GetFileDialog

GetStringDialog

Gradient

GraphBuilder

Harmonics

HasItemSpec

HasSubject

HasSubjectGui

HIDDevice

HIDDeviceElement

HiliteGradient

IdentityBag

ImageWarp

ImmutableError

Impulsar

InBus

InfoUGenBase

InRange

InRect

Insp

Inspector

InspectorLink

InspManager

InstrSpawnerGui

Instrument

IntegerEditor

InterfaceDef

InterfaceGui

IODesc

IOStream

IsIn

IsNil

IsOdd

KDRMaskTester

KeyCodeResponderStack

KrNumberEditorGui

KrPlayer

KrPlayerGui

LagControl

LagIn

LeastChange

LibraryBase

LimitedWriteStream

LinearWarp

Lines

ListDUGen

ListPattern

LocalClient

Logistic

LRUNumberAllocator

Message

MethodError

MethodGui

MethodInspector

MethodLabel

MethodQuote

MidEQ

Midi2Freq

Midi2FreqGui

Midi2FreqUGen

MIDIClient

MIDIEndPoint

MIDIFreqPlayer

MIDIGatePlayer

MIDIHoldsNotes

MixedBundle

ModalFreqGui

ModalFreqUGen

Model

ModelImplementsGuiBody

Module

MulAdd

MultiplePlayers

MultiTrackAudioSpec

MultiTrackPlayer

MustBeBooleanError

NAryOpStream

NetAddr

NodeIDAllocator

NodeMapSetting

NoLagControlSpec

Not

NoteOffResponder

NoteOnResponder

NotificationRegistration

NotNil

NotYetImplementedError

NumAudioBuses

NumberEditorGui

NumBuffers

NumControlBuses

NumInputBuses

NumOutputBuses

ObjectInspector

ObjectNotFound

OneShotStream

OSCBundle

OSCMultiResponder

OSCpathDispatcher

OSCResponderQueue

OSCService

OutOfContextReturnError

Paddp

Paddpre

Pair

Panner

PatchGui

PatchIn

PatchOut

PatternControl

PauseSelfWhenDone

PauseStream

Pbindf

Pbinop

Pbrown

Pconst

PeakFollower

Peep

Pen

Pevent

PfadeIn

PfadeOut

Pfindur

Pfset

Pfunc

Pfuncn

Pfx

Pgeom

Pindex

Plag

PlayButton

PlayerAmpGui

PlayerBinop

PlayerBinopGui

PlayerEffectGui

PlayerEfxFuncGui

PlayerMixerGui

PlayerPoolGui

PlayerSpec

PlayerUnopGui

Ploop

Pmono

PMOsc

Pmulp

Pmulpre

Pnaryop

PointArray

Polygon

PopUp

PopUpEditor

PopUpEditorGui

Position

Post

PowerOfTwoAllocator

PowerOfTwoBlock

Ppar

Pplayer

Pretty

PrettyEat

PrettyEcho

PrettyPrintStream

PrettyState

PrimitiveFailedError

Prout

Proutine

ProxyNodeMap

ProxyNodeMapSetting

ProxySynthDef

Pseries

Psetp

Psetpre

PSinGrain

Pstep2add

Pstep3add

Pstretch

Pstretchp

Ptpar

Ptrace

Punop

PV_Add

PV_BinScramble

PV_BinShift

PV_BinWipe

PV_BrickWall

PV_CopyPhase

PV_Diffuser

PV_LocalMax

PV_MagAbove

PV_MagBelow

PV_MagClip

PV_MagFreeze

PV_MagMul

PV_MagNoise

PV_MagShift

PV_MagSmear

PV_MagSquared

PV_Max

PV_Min

PV_Mul

PV_PhaseShift

PV_PhaseShift270

PV_PhaseShift90

PV_RandComb

PV_RandWipe

PV_RectComb

PV_RectComb2

Pwhile

Pwhite

RadiansPerSample

Range

RefCopy

ResponderArray

ResponderClientFunc

RingNumberAllocator

Router

SampleGui

SampleSpec

SC2compat

ScalarPatchIn

ScalarPatchOut

ScalarSpec

SCButtonAdapter

SCContainerView

SCControlView

SCDragBoth

SCDragSink

SCDragSource

SCDragView

Scheduler

Schmidt

SCKnob

SCLayoutView

SCListView

ScopeOut

SCScope

SCSlider

SCSliderBase

SCStaticText

SCStaticTextBase

SCTopView

ScurryableInstrGateSpawner

SCUserView

SCViewAdapter

SelectorLabel

ServerGui

SFPGui

SharedNodeProxy

ShouldNotImplementError

Silent

SimpleController

SimpleKDRUnit

SimpleTrigger

SimpleTriggerGui

SineWarp

SinOscFB

SlotInspector

SoundFileFormats

SplayZ

StackNumberAllocator

StartRow

StaticIntegerSpec

StaticSpec

StreamControl

StreamKrDurGui

StringInspector

SubclassResponsibilityError

SymbolArray

SynthControl

SynthDefControl

SynthDescLib

SynthlessPlayer

TabFileReader

Tap

TChoose

TempoBusClock

TempoGui

TempoSpec

TestDependant

Tile

TPulse

Trapezoid

TrigControl

TrigSpec

TwoWayIdentityDictionary

UGenInstr

UI

UnaryOpPlug

UnicodeResponder

UniqueID

Unix

Updater

UpdatingScalarPatchOut

VariableNameLabel

Vibrato

Warp

WavetableSampleGui

Wrap

XFade

XFader

XFader4

XFaderN

XFaderPlayerGui

XIn

XInFeedback

Xor

XPlayPathButton

XY

ZigZag

ID: 341

Writing SuperCollider Classes

For a basic tutorial on how standard object-orientated classes are composed, look elsewhere
#0000ffhttp://www.google.com/search?q=oop+class+tutorial

Inheriting

```
NewClass   SomeSuperclass

}
```

Without specifying a superclass, Object is assumed as the default superclass.

```
NewClass  // :  Object is implied

}
```

Methods

class methods are specified with the asterix

```
*classMethod { arg argument;

}
```

within the class method, the keyword

```
this
```

refers to the class.

A class in smalltalk is itself an object. It is an instance of Class.

instance methods are specified :

```
instanceMethod { arg argument;

}
```

within the instance method, the keyword

```
   this
```

refers to the instance.

to **return** from the method use ^ (caret)

```
someMethod {
^returnObject
}
```

multiple exit points also possible :

```
someMethod { arg aBoolean;
if(aBoolean,{
^someObject
},{
^someOtherObject
})
}
```

if no ^ is specified, the method will return the **instance**.
(and in the case of Class methods, will return the class)

There is no such thing as returning void in Smalltalk.

New Instance creation

Object.new will return to you a new object.
when overiding the class method .new you must call the superclass, which in turn calls its superclass, up until Object.new is called and an object is actually created and its memory allocated.

```
  // this example adds no new functionality
*new {
^super.new
}

 // this is a normal constructor method
*new { arg arga,argb,argc;
^super.new.init(arga,argb,argc)
}
init { arg arga,argb,argc;
      // do initiation here
}
```

In this case note that super.new called the method new on the superclass and returned a new object. subsequently we are calling the .init method on that object, which is an instance method.

Warning: if the superclass also happened to call super.new.init it will have expected to call the .init method defined in that class (the superclass), but instead the message .init will find the implementation of the class that the object actually is, which is our new subclass. So you should use a unique methodname like myclassinit if this is likely to be a problem.

Over reliance on inheritance is usually a design flaw. Explore "object composition" rather than trying to obtain all your powers through inheritance. Is your "subclass" really some kind of "superclass" or are you just trying to swipe all of daddy's methods ? Do a websearch for Design Patterns.

Class variables are accessible within class methods and in any instance methods.

```
classvar myClassvar;
```

```
var myInstanceVar;
```

Overriding Methods (Overloading)

in order to change the behaviour of the superclass, often methods are overridden.
note that an object looks always for the method it has defined first and then looks in the superclass.
here `NewClass.value(2)` will return6, not4:

```
      Superclass

calculate { arg in; in * 2 }
value { arg in; ^this.calculate(in) }
}

   NewClass   Superclass

calculate { arg in; in * 3 }
}
```

if the method of the superclass is needed, it can be called by **super**.

```
      Superclass
var x;

init {
x = 5;
}
}

   NewClass   Superclass
var y;
init {
super.init;
y = 6;
}
```

```
}
```

Getter Setter

Classic Smalltalk demands that variables are not accessible outside of the class or instance. A method must be added to explicitly give access:

```
      NewClass   Superclass

var myVariable;

variable {
^variable
}

variable_ { arg newValue;
variable = newValue;
}
}
```

These are referred to as getter and setter methods.

SC allows these methods to be easily added by adding < or >

```
var <getMe, >setMe, <>getMeOrSetMe;
```

you now have the methods:

```
someObject.getMe;

someObject.setMe_(value);
```

this also allows us to say:

```
someObject.setMe = value;
```

```
someObject.getMeOrSetMe_(5);
someObject.getMeOrSetMe.postln;
```

a getter or setter method created in this fashion may be overriden in a subclass by manually writing the method
setter methods should **take only one argument** to support both ways of expression consistantly.
eg.

```
variable_ { arg newValue;
variable = newValue.clip(minval,maxval);
}
```

External Method Files

Methods may be added to Classes in separate files. This is equivalent to Protocols in Objective-C. By convention, the file name starts with a lower case letter: the name of the method or feature that the methods are supporting.

Syntax:

```
+ Class {
newMethod {

}
*newClassMethod {

}
}
```

Tricks and Traps

"Superclass not found..."

In one given code file, you can only put classes that inherit from each Object, each other, and one external class. In other words, you can't inherit from two separate classes that are defined in separate files.

If you should happen to declare a variable in a subclass and use the same name as a variable declared in a superclass, you will find that both variables exist, but only the one in the object's actual class is accessible. You should not do that. This will at some point become an error worthy of compilation failure.

-felix, jrh

ID: 342

How Unit Generator plug-ins work.

The server loads unit generator plug-ins when it starts up.
Unit Generator plug-ins are dynamically loaded libraries (DLLs) written in C++.
Each library may contain one or multiple unit generator definitions.
The server looks in the "plugins" directory for all files ending in .scx and calls the load() function in each one.

The load() function

When the library is loaded the server calls the load() function in the library.

The load function has two responsibilities:
• It needs to store the passed in pointer to the InterfaceTable in a global variable.
• It defines the unit generators.

```
// InterfaceTable contains pointers to functions in the host (server).
       InterfaceTable
...


// the load function is called by the host when the plug-in is loaded
void load(InterfaceTable *inTable)
{
 ft = inTable; // store pointer to InterfaceTable


 DefineSimpleUnit MySaw
}
```

Unit Generators are defined by calling a function in the InterfaceTable and passing it the name of the unit generator, the size of its C data struct, and pointers to functions for constructing and destructing it. The macro `DefineSimpleUnit` makes this more brief.

```
#define DefineSimpleUnit(name) \
(*ft->fDefineUnit)(#name, sizeof(name), (UnitCtorFunc)&amp;name##_Ctor, 0);
```

`ft->fDefineUnit` is a function pointer in the InterfaceTable to the server function that defines a new unit generator.

`#name` creates a string C from the name. In this case, "MySaw".

`sizeof(name)` will be the size of the struct MySaw.

`name##_Ctor` will macro-expand to `MySaw_Ctor`. There will need to be a function defined with this name.

`0` is the argument for the Dtor, or destructor function, which is not needed for this unit generator.

So the macro:

```
DefineSimpleUnit(MySaw);
```

expands to this:

```
(*ft->fDefineUnit)("MySaw", sizeof(MySaw), (UnitCtorFunc)&amp;MySaw_Ctor, 0);
```

A plug-in can also define things other than unit generators such as buffer fill ("/b_gen") commands.

Adding a Target to Xcode

You will need to have the Developer Tools installed to do this.
Each group of plugins shares a target in Xcode. They create a file: eg. MyUGens.scx which is then copied into the plugins folder via a shell script.

Create a new target, select type 'Legacy"Library'.
(In the future using 'Legacy' may not be necessary, but currently Xcode doesn't build BUNDLE library style correctly with non-legacy targets.)

<img src="Pasted%20Graphic.tiff" alt="Pasted Graphic.tiff">

Go to the Target Inspector.
Set the product name to be MyUGens.scx.

<img src="Pasted%20Graphic%201.tiff" alt="Pasted Graphic 1.tiff">

Click on "GCC CompilerSettings".
Set "Other C Flags" to -DSC_DARWIN

<img src="Pasted%20Graphic%209.tiff" alt="Pasted Graphic 9.tiff">

Click on "Expert View" and change LIBRARY_STYLE to BUNDLE
<img src="Pasted%20Graphic%206.tiff" alt="Pasted Graphic 6.tiff">

Add a build phase (control or right click on build phases): a shell script build phase.
Type this script (with the name of your build product in place of MyUGens.scx) :

cp build/MyUGens.scx build/plugins

<img src="Pasted%20Graphic%204.tiff" alt="Pasted Graphic 4.tiff">

Open the inspector for the All Plugins target and click on the plus button <img src="Pasted%20Graphic%203.tiff" alt="Pasted Graphic 3.tiff">.
A list of targets will open. Add your target.
<img src="Pasted%20Graphic%205.tiff" alt="Pasted Graphic 5.tiff">

Your target will be built along with the others when the aggregate target "All Plugins" is selected.

Create a new .cpp file and add it to the project.
<img src="Pasted%20Graphic%207.tiff" alt="Pasted Graphic 7.tiff">
Set the name to MyUGens.cpp
Uncheck "Also create 'MyUGens.h' "
Location should be in SuperCollider3/source/plugins .

Check your target's name in the targets list.
Click Finish.

<img src="Pasted%20Graphic%208.tiff" alt="Pasted Graphic 8.tiff">

Copy this code into the MyUGens.cpp file.

```
        "SC_PlugIn.h"

// InterfaceTable contains pointers to functions in the host (server).
      InterfaceTable

// declare struct to hold unit generator state
struct MySaw :  public Unit
{
 double mPhase; // phase of the oscillator, from -1 to 1.
  float mFreqMul; // a constant for multiplying frequency
};

// declare unit generator functions
extern "C"
{
void load(InterfaceTable *inTable);
void MySaw_next_a(MySaw *unit, int inNumSamples);
void MySaw_next_k(MySaw *unit, int inNumSamples);
void MySaw_Ctor(MySaw* unit);
};

//////////////////////////////////////////////////////////////////

// Ctor is called to initialize the unit generator.
// It only executes once.

// A Ctor usually does 3 things.
// 1.  set the calculation function.
// 2.  initialize the unit generator state variables.
```

```
// 3.  calculate one sample of output.
     MySaw_Ctor MySaw
{

 // 1.  set the calculation function.
if (INRATE(0) == calc_FullRate) {
  // if the frequency argument is audio rate
  SETCALC MySaw_next_a
} else {
  // if the frequency argument is control rate (or a scalar).
    SETCALC MySaw_next_k
}

 // 2.  initialize the unit generator state variables.
  // initialize a constant for multiplying the frequency
unit->mFreqMul = 2.0 * SAMPLEDUR;
 // get initial phase of oscillator
unit->mPhase = IN0(1);

 // 3.  calculate one sample of output.
  MySaw_next_k
}


//////////////////////////////////////////////////////////////////

// The calculation function executes once per control period
// which is typically 64 samples.

// calculation function for an audio rate frequency argument
void MySaw_next_a(MySaw *unit, int inNumSamples)
{
 // get the pointer to the output buffer
float *out = OUT(0);

 // get the pointer to the input buffer
float *freq = IN(0);

 // get phase and freqmul constant from struct and store it in a
 // local variable.
```

```
  // The optimizer will cause them to be loaded it into a register.
float freqmul = unit->mFreqMul;
double phase = unit->mPhase;

 // perform a loop for the number of samples in the control period.
  // If this unit is audio rate then inNumSamples will be 64 or whatever
 // the block size is.  If this unit is control rate then inNumSamples will
 // be 1.
for (int i=0; i < inNumSamples; ++i)
{
  // out must be written last for in place operation
float z = phase;
phase += freq[i] * freqmul;

  // these if statements wrap the phase a +1 or -1.
if (phase >= 1.f) phase -= 2.f;
else if (phase <= -1.f) phase += 2.f;

  // write the output
out[i] = z;
}

 // store the phase back to the struct
unit->mPhase = phase;
}

//////////////////////////////////////////////////////////////////

// calculation function for a control rate frequency argument
void MySaw_next_k(MySaw *unit, int inNumSamples)
{
 // get the pointer to the output buffer
float *out = OUT(0);

 // freq is control rate, so calculate it once.
float freq = IN0(0) * unit->mFreqMul;

 // get phase from struct and store it in a local variable.
  // The optimizer will cause it to be loaded it into a register.
double phase = unit->mPhase;
```

```
 // since the frequency is not changing then we can simplify the loops
 // by separating the cases of positive or negative frequencies.
  // This will make them run faster because there is less code inside the loop.
if (freq >= 0.f) {
  // positive frequencies
for (int i=0; i < inNumSamples; ++i)
{
out[i] = phase;
phase += freq;
if (phase >= 1.f) phase -= 2.f;
}
} else {
  // negative frequencies
for (int i=0; i < inNumSamples; ++i)
{
out[i] = phase;
phase += freq;
if (phase <= -1.f) phase += 2.f;
}
}

 // store the phase back to the struct
unit->mPhase = phase;
}

////////////////////////////////////////////////////////////////////

// the load function is called by the host when the plug-in is loaded
void load(InterfaceTable *inTable)
{
ft = inTable;

 DefineSimpleUnit MySaw
}

////////////////////////////////////////////////////////////////////
```

```
// In the MyUGens.sc file:

MySaw   UGen
*ar { arg freq = 440.0, iphase = 0.0, mul = 1.0, add = 0.0;
^this.multiNew('audio', freq, iphase).madd(mul, add)
}
*kr { arg freq = 440.0, iphase = 0.0, mul = 1.0, add = 0.0;
^this.multiNew('control', freq, iphase).madd(mul, add)
}
}
```

The SuperCollider class for your UGen allows the SuperCollider application to be able to write a SynthDef file.

The arguments to the MySaw UGen are freq and iphase.
The `multiNew` method handles multi channel expansion.
The `madd` method provides support for the mul and add arguments. It will create a MulAdd UGen if necessary. You could write the class without mul and add arguments, but providing them makes it more convenient for the user.

```
// without mul and add.
MySaw   UGen
*ar { arg freq = 440.0, iphase = 0.0;
^this.multiNew('audio', freq, iphase)
}
*kr { arg freq = 440.0, iphase = 0.0;
^this.multiNew('control', freq, iphase)
}
}
```

```
// test it:

{ MySaw.ar(200,0,0.1) }.play
```

Useful macros

These are defined in SC_Unit.h.

```
// These return float* pointers to input and output buffers.
#define IN(index)  (unit->mInBuf[index])
#define OUT(index) (unit->mOutBuf[index])


// These return a float value.  Used for control rate inputs and outputs.
#define INO(index)  (IN(index)[0])
#define OUTO(index) (OUT(index)[0])


// get the rate of the input.
#define INRATE(index) (unit->mInput[index]->mCalcRate)
```

The possible rates are:

```
calc_ScalarRate
calc_BufRate    "control rate"
calc_FullRate    "audio rate"


// set the calculation function
#define SETCALC(func) (unit->mCalcFunc = (UnitCalcFunc)&amp;func)
```

SETCALC must be called in the constructor. It may also be called from a calculation function to change to a different calculation function.

```
// calculate a slope for control rate interpolation to audio rate.
#define CALCSLOPE(next,prev) ((next - prev) * unit->mRate->mSlopeFactor)
```

CALCSLOPE returns (next - prev) / blocksize which is useful for calculating slopes for linear interpolation.

```
#define SAMPLERATE (unit->mRate->mSampleRate)
```

SAMPLERATE returns the sample rate for the unit generator. If it is audio rate then it will be the audio sample rate. If the ugen is control rate, then it will be the control rate. For example, if the ugen is control rate and the auio sample rate is 44100 and the block size is 64, then this will return 44100/64 or 689.0625.

```
#define SAMPLEDUR (unit->mRate->mSampleDur)
```

SAMPLEDUR is simply the reciprocal of the sample rate. It is the seconds per sample.

```
#define BUFLENGTH (unit->mBufLength)
```

BUFLENGTH is equal to the block size if the unit is audio rate and is equal to 1 if the unit is control rate.

```
#define BUFRATE (unit->mRate->mBufRate)
```

BUFRATE always returns the control rate.

```
#define BUFDUR (unit->mRate->mBufDuration)
```

BUFDUR is the reciprocal of the control rate.

Pointer aliasing

The server uses a "buffer coloring" algorithm to minimize use of buffers to optimize cache performance. This means that any of the output buffers may be the same as one of the input buffers. This allows for in-place operation which is very efficient. You must be careful however not to write any output sample before you have read all of the input samples. If you did, then the input will be overwritten with output.

```
// This code is correct.  It reads the freq input before writing to out.
for (int i=0; i < inNumSamples; ++i)
{
float z = phase; // store phase in z
phase += freq[i] * freqmul; // read freq
  out[i] = z; // write the output

  // these if statements wrap the phase a +1 or -1.
if (phase >= 1.f) phase -= 2.f;
else if (phase <= -1.f) phase += 2.f;
}


// If out and freq are the same, then the code below will fail.
for (int i=0; i < inNumSamples; ++i)
{
  // write the output
out[i] = phase;
phase += freq[i] * freqmul;

  // these if statements wrap the phase a +1 or -1.
if (phase >= 1.f) phase -= 2.f;
else if (phase <= -1.f) phase += 2.f;
}
```

If your unit generator cannot be written efficiently when pointers are aliased, then you can tell the server this by using one of the following macros when defininig it.

```
DefineSimpleCantAliasUnit(MyUGen);

DefineDtorCantAliasUnit(MyUGen);
```

The server will then ensure that no output buffers are the same as any input buffers.

A Unit Generator that needs a Dtor

This is code for a simple fixed delay line.

```
        "SC_PlugIn.h"

// InterfaceTable contains pointers to functions in the host (server).
      InterfaceTable

// declare struct to hold unit generator state
struct MyDelay :  public Unit
{
uint32 mDelayLength;
uint32 mPosition;
float *mData; // delay buffer
};

// declare unit generator functions
extern "C"
{
void load(InterfaceTable *inTable);
void MyDelay_next_notfull(MyDelay *unit, int inNumSamples);
void MyDelay_next_full(MyDelay *unit, int inNumSamples);
 void MyDelay_Ctor MyDelay
 void MyDelay_Dtor MyDelay
};

////////////////////////////////////////////////////////////////

// Ctor is called to initialize the unit generator.
// It only executes once.

// A Ctor usually does 3 things.
// 1.  set the calculation function.
// 2.  initialize the unit generator state variables.
// 3.  calculate one sample of output.
     MyDelay_Ctor MyDelay
{
```

```
 // 1.  set the calculation function.
  SETCALC MyDelay_next_notfull

 // 2.  initialize the unit generator state variables.
  // get the delay length
unit->mDelayLength = (uint32)(IN0(1) * SAMPLERATE);

 // allocate the buffer
unit->mData = (float*)RTAlloc(unit->mWorld, unit->mDelayLength * sizeof(float));
 // RTAlloc allocates out of the real time memory pool of the server
 // which is finite.  The size of the real time memory pool is set using the
 // -m command line argument of the server.

  // initialize the position
unit->mPosition = 0;

 // 3.  calculate one sample of output.
  MyDelay_next_notfull
}

//////////////////////////////////////////////////////////////////

// Dtor is called to perform any clean up for the unit generator.

     MyDelay_Dtor MyDelay
{
 // free the buffer
RTFree(unit->mWorld, unit->mData);
}


//////////////////////////////////////////////////////////////////

// The calculation function executes once per control period
// which is typically 64 samples.

// calculation function when the buffer has not yet been filled
void MyDelay_next_notfull(MyDelay *unit, int inNumSamples)
{
 // get the pointer to the output buffer
```

```
float *out = OUT(0);

 // get the pointer to the input buffer
float *in = IN(0);

 // get values from struct and store them in local variables.
  // The optimizer will cause them to be loaded it into a register.
float *data = unit->mData;
uint32 length = unit->mDelayLength;
uint32 position = unit->mPosition;
bool wrapped = false;

 // perform a loop for the number of samples in the control period.
  // If this unit is audio rate then inNumSamples will be 64 or whatever
 // the block size is.  If this unit is control rate then inNumSamples will
 // be 1.
for (int i=0; i < inNumSamples; ++i)
{
  // get old value in delay line
float z = data[position];
  // store new value in delay line
data[position] = in[i];

  // see if the position went to the end of the buffer
if (++position >= length) {
   position = 0; // go back to beginning
   wrapped = true  // indicate we have wrapped.

      // change the calculation function
   // next time, the MyDelay_next_full function will be called
   SETCALC MyDelay_next_full
}
  // if we have not yet wrapped, then z is garbage from the uninitialized
  // buffer, so output zero.  If we have wrapped, then z is a good value.
out[i] = wrapped ?  z :  0.f;
}

 // store the position back to the struct
unit->mPosition = position;
}
```

```
//////////////////////////////////////////////////////////////////

// calculation function when the buffer has been filled
void MyDelay_next_full(MyDelay *unit, int inNumSamples)
{
 // get the pointer to the output buffer
float *out = OUT(0);

 // get the pointer to the input buffer
float *in = IN(0);

 // get values from struct and store them in local variables.
  // The optimizer will cause them to be loaded it into a register.
float *data = unit->mData;
uint32 length = unit->mDelayLength;
uint32 position = unit->mPosition;

 // perform a loop for the number of samples in the control period.
  // If this unit is audio rate then inNumSamples will be 64 or whatever
 // the block size is.  If this unit is control rate then inNumSamples will
 // be 1.
for (int i=0; i < inNumSamples; ++i)
{
  // get old value in delay line
float z = data[position];
  // store new value in delay line
data[position] = in[i];
  // see if the position went to the end of the buffer
if (++position >= length) {
   position = 0; // go back to beginning
}
out[i] = z;
}

 // store the position back to the struct
unit->mPosition = position;
}

//////////////////////////////////////////////////////////////////
```

```
// the load function is called by the host when the plug-in is loaded
void load(InterfaceTable *inTable)
{
ft = inTable;

 DefineDtorUnit MyDelay
}

//////////////////////////////////////////////////////////////////
```

```
// In the MyUGens.sc file:

MyDelay   UGen
*ar { arg in, delaytime=0.4;
^this.multiNew('audio', in, delaytime)
}
*kr { arg in, delaytime=0.4;
^this.multiNew('control', in, delaytime)
}
}
```

```
// test it
(
{
var z;
z = SinOsc.ar * Decay.kr(Impulse.kr(1,0,0.2), 0.1);
[z, MyDelay.ar(z, 0.3)]
}.play;
)
```

TO DO:
UGens which access buffers.
UGens which use the built in random number generators.

18 Networking

ID: 343

NetAddr network address

superclass: Objects

***new(hostname, port)** create new net address.
Hostname is a string, either an ip number (e.g. *"192.168.34.56"*)
Port is a port number, like *57110*.
Note: to send messages internally, loopback ip is used: *"127.0.0.1"*

***fromIP(ip, port)** create new net address using an integer ip number.

sendMsg(args...) send a message without timestamp to the addr.

sendBundle(timestamp, args...) send a bundle with timestamp to the addr.

sendRaw(rawArray) send a raw message without timestamp to the addr.

connect(disconnectHandler) open TCP connection. **disconnectHandler** is called when
the connection is closed (either by the client or by the server)

disconnect close TCP connection

ip returns the ip number (as string)
example:

```
n = NetAddr("localhost", 57110);
n.ip;
```

***disconnectAll** close all TCP connections

```
// example
```

```
    NetAddr "127.0.0.1"          // 57120 is sclang default port
r = OSCresponder(n, '/good/news', { arg time, resp, msg; [time, msg].postln }).add;

          "/good/news"  "you"  "not you"
n.sendMsg("/good/news", 1, 1.3, 77);


n.sendBundle(0.2, ["/good/news", 1, 1.3, 77]);

r.remove;
```

19 OSX

19.1 Miscellanea

ID: 344

Cocoa

see also: [CocoaDialog]

Cocoa.getPathsInDirectory(path)

```
// example:
Cocoa.getPathsInDirectory("plugins")


// note:  it is better to now use pathMatch (unix compatible).  Wild cards like * can be used.

"plugins/*".pathMatch;
"plugins/D*"
"plugins/[D,T]*"


/*

 This is a temporary implementation before I (felix) gets around to doing the proper
 Directory implementation.
  It gets all paths in that directory and subdirectories.
  maxItems is the size of the array to use, and should be larger than the number of items
 you might return, else a primitive index error.


  all paths are standardized

*/
```

ID: 345

CocoaDialog file dialogs that utilize OS X Cocoa services.

see also: **[Cocoa]**

OSX only.

*getPaths(okFunc, cancelFunc, maxSize)

Displays an Open File Dialog. If ok is pressed then **okFunc** is evaluated with the selected paths passed as an Array of Strings as the first argument. If cancel is pressed then **cancelFunc** is evaluated. **maxSize** is the maximum number of files which may be selected. The default is 20.

```
(
CocoaDialog.getPaths({ arg paths;
paths.do({ arg p;
p.postln;
})
},{
 "cancelled"
});
)
```

*savePanel(okFunc, cancelFunc)

Displays a Save File Dialog. If ok is pressed then **okFunc** is evaluated with the selected path passed as a Strings as the first argument. If cancel is pressed then **cancelFunc** is evaluated.

```
(
CocoaDialog.savePanel({ arg path;
path.postln;
},{
"cancelled".postln;
});
)
```

ID: 346

MIDI

MIDIClient
MIDIEndpoint
MIDIIn
MIDIOut
superclass: Object

See the **[UsingMIDI]** helpfile for practical considerations and techniques for using MIDI in SC.

MIDIClient is a static class that starts up the MIDI service:
It initializes with a number of virtual inports and outports.
The default is 1. and usually not more a needed.
The information about the hardware is stored in MIDIClient.sources and MIDIClient.destinations as MIDIEndpoints.

MIDIIn represents a connection between a inport and a source of the MIDIClient.
There are three possibilities to connect them:
To do something with the incoming MIDI data set the actions.

example:

```
(
MIDIClient.init;
//There are three possibilities to connect for example to the first device:
//MIDIIn.connect(0, MIDIClient.sources.at(0));
//MIDIIn.connect(0, MIDIClient.sources.at(0).uid);
MIDIIn.connect(0, 0);
//set the action:
MIDIIn.control = {arg src, chan, num, val;
val.postln;
};
)
```

MIDIOut

example:

```
(
```

```
MIDIClient.init;
m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
m.noteOn(0, 60, 60);
)


)
```

ID: 347

NSObjectHolder

```
    NSObjectHolder       "NSWindow"

o.do("makeKeyAndOrderFront:",[0]);

o.do("close");

o.dealloc;
```

ID: 348

Speech

Speech lets you use the cocoa speech synthesizer.

```
"hi i'm talking with the default voice now, i guess"
```

First argument is always the voice channel number, second the value

```
 Speech.setSpeechVoice(0,14);
Speech.setSpeechPitch(0, 40); //pitch in MIDI Num
Speech.setSpeechRate(0, 10);
Speech.setSpeechVolume(0,0.8);
Speech.setSpeechPitchMod(0, 200);
```

Two actions can be applied:

```
Speech.wordAction = {arg voiceNum;
 //i.postln;
 // the currently speaking text may not be changed
//Speech.setSpeechPitch(voiceNum,[41,60].choose);
//Speech.setSpeechRate(voiceNum,[60,80, 10].choose);
};
Speech.doneAction_({arg voiceNum;
Speech.setSpeechPitch(voiceNum,[41,48,40,43,30,60].choose);
});
```

Pause the speech while speaking: 1=pause, 0= start

```
Speech.pause(0,1);
```

Initialization happens automatically, by default with one voice channel.
You may explicitly initalize with more channels, up to 128:

```
(
Speech.init(64);
```

```
Task
64.do ({arg i;
[0.1, 0.18, 0.2].choose.wait;
Speech.setSpeechRate(i,[90, 30, 60].choose);
Speech.setSpeechVolume(i,0.07);
  "no this is private.  float .  boolean me.  char[8] "
});
}).play;
)

//jan.t@kandos.de 04/2003
```

ID: 349

standardizePath

String Method

Returns a string with ~ replaced by the current user's home directory, and all symbolic links resolved.

```
" ~/Documents".standardizePath
```

```
/Volumes/Macintosh HD/Users/cruxxial/Documents
```

Note that my Documents folder is on a different partition than the boot,
so the full path uses /Volumes

Resolves symbolic links, but does not resolve aliases.

```
" /Library/Favorites/SuperCollider3"
```

```
/Volumes/Macintosh HD/Users/cruxxial/Library/Favorites/SuperCollider3
```

Removes extraneous . and .. and / but does not otherwise expand them.

```
"./Help/".standardizePath
```

```
Help
```

From the Cocoa documentation:

- Expand an initial tilde expression using #1a1affstringByExpandingTildeInPath.
- Reduce empty components and references to the current directory (that is, the sequences "//" and "/./") to single path separators.
- In absolute paths only, resolve references to the parent directory (that is, the component "..") to the real parent directory if possible using #1a1affstringByResolvingSymlinksInPath, which consults the file system to resolve each potential symbolic link.

In relative paths, because symbolic links can't be resolved, references to the parent directory are left in place.

• Remove an initial component of " /private" from the path if the result still indicates an existing file or directory (checked by consulting the file system).

ID: 350

writeAsPlist

Object method

```
object.writeAsPlist(path);
```

Write the object to disk as a PropertyList. It is used commonly in OS X to store data.

This is an XML format that may be read from easily from Objective-C, Java or any language that uses the CoreFoundation(Cocoa) framework.

Objective-C:

```
id rootObject = [NSKeyedUnarchiver  unarchiveObjectWithFile:  path ];
```

The file may be double-clicked to open it in Property List Editor.

When opening the PropertyList:

your SC objects are converted to Foundation objects in this fashion:

SimpleNumber -> NSNumber.
SequenceableCollections -> NSArray with each item converted
Strings -> NSString
Symbols -> NSString
Char -> NSString
Nil -> NSNull
Boolean ->NSNumber (numberWithBool:) equivalent to a CFBoolean
Dictionary ->NSDictionary (each item converted)

Other objects -> NSNull

Most commonly the root object is an NSDictionary or NSArray.

```
"testWritePlist.plist"
```

```
               "testWritePlist.plist"

"string"              "testWritePlist.plist"

'symbol'              "testWritePlist.plist"

                "testWritePlist.plist"

nil              "testWritePlist.plist"

true              "testWritePlist.plist"

false              "testWritePlist.plist"

   false 'symbol'               "testWritePlist.plist"

List[1,false,\symbol].writeAsPlist("testWritePlist.plist");

// cannot convert these objects
[Ref(pi),Pbind.new].writeAsPlist("testWritePlist.plist");

Dictionary
\a -> "a",
"b" -> \b,
3 -> 3.0
 ].writeAsPlist("testWritePlist.plist"


IdentityDictionary
\a -> "a",
"b" -> \b,
3 -> 3.0
 ].writeAsPlist("testWritePlist.plist"
```

19.2 Objc

ID: 351

SCNSObject

note: this is experimental (03/2006) things might change and be careful wrong or unsupported Cocoa-calls can crash this Application!

SCNSObject creates a bridge between SuperCollider and Objective-C / Cocoa.
It holds an NSObject and sends messages to it.
The class and messages are passed as Strings. Arguments must be in an Array.
On creation only the init message is passed, alloc is called internally. So all constructor messages other then alloc are not supported yet.

Example:
The Cocoa synthax:

```
NSNumber *n = [[NSNumber alloc] initWithFloat:  1.1];
[n floatValue];
```

turns into:

```
  n = SCNSObject "NSNumber"  "initWithFloat:"
n.invoke("floatValue");
```

Multiple messages are put together in one STring and their arguments in one Array.
Example:
Cocoa:

```
NSWindow *c = [[NSWindow alloc] initWithContentRect:  rect styleMask:  10 backing:  2 defer:YES];
```

SC:

```
 c = SCNSObject "NSWindow"  "initWithContentRect:styleMask:backing:defer:"  Rect
```

Defer:

Some methods need to be defered. If you want to defer ust call invoke with defer:true. Watch out there is no smart protection for methods that need defer until now! In general you should defer graphic operations.
So calling this might crash sc-lang: c.invoke("makeKeyAndOrderFront:", [nil]);
but this line is fine:

```
  c.invoke("makeKeyAndOrderFront:"   nil   true
```

Types:

SCNSObjects are converted to NSObjects.

Some types are converted directly:
Rect -> NSRect
Point -> NSPoint

Many obj-c types are not supported yet (NSRange, nib-files, ...).

A String in SC is different than the cString used in Cocoa. So you might get some strange artefacts.

```
SCNSObject "NSString" "initWithCString:"   "x 3456512"
```

Actions:

.initAction is a convenience method to add an action to a gui element.
Depending on the type there are different actions to be set: "doFloatAction:"
"doIntAction:"
"doStateAction:"
"doAction:"

Examples:

```
//create a window and add a Slider that posts its value.

(
var winname = "cocoa test", win, nsname, slider;
nsname = SCNSObject("NSString","initWithCString:length:", [winname, winname.size], false);

     SCNSObject "NSWindow"  "initWithContentRect:styleMask:backing:defer:"
[Rect(100,140,400,30), 10, 2, 1]);
win.setDelegate.action_({
 "closing window, releasing objects"
[winname,nsname,slider,e].do{| it|  it.release};
});
slider = SCNSObject("NSSlider", "initWithFrame:", [Rect(0,0,390,20)]);
    SCNSObject "SCGraphView"  "initWithFrame:"   Rect
win.invoke("setContentView:", [e], true);
e.invoke("addSubview:", [slider], true);
slider.invoke("setFloatValue:", [0.5]);
```

```
            "makeKeyAndOrderFront:"   nil   true
win.invoke("setTitle:", [nsname]);

{a = slider.initAction;
a.action_({| v,val|  val.postln});}.defer(0.1);
win = win;
)
win.className
win.invoke("close", defer:true);



(
     SCNSObject "NSString" "initWithCString:"   "x 3456512"

    SCNSObject "NSWindow"  "initWithContentRect:styleMask:backing:defer:"  Rect

c.setDelegate.action_({
"closing window, releasing objects".postln;
[z,c,d,e].do{| it| it.release};
});
    SCNSObject "NSTextField"  "initWithFrame:"   Rect
e = SCNSObject("NSView", "initWithFrame:", [Rect(0,0,400,100)]);
c.invoke("setContentView:", [e], true);
e.invoke("addSubview:", [d], true);
         "makeKeyAndOrderFront:"   nil   true

)
(
     SCNSObject "NSString" "initWithCString:"   "x 3456512"

    SCNSObject "NSWindow"  "initWithContentRect:styleMask:backing:defer:"  Rect

c.setDelegate.action_({
"closing window, releasing objects".postln;
[z,c,d,e].do{| it| it.release};
});
d = SCNSObject("NSButton", "initWithFrame:", [Rect(0,0,100,20)]);
e = SCNSObject("NSView", "initWithFrame:", [Rect(0,0,400,100)]);
c.invoke("setContentView:", [e], true);
```

```
e.invoke("addSubview:", [d], true);
         "makeKeyAndOrderFront:"   nil   true
d.invoke("setButtonType:", [3]);
{
             "doStateAction:"
d.nsAction.action_({| it,val|  val.postln;});
}.defer(0.1);
)
```

20 Other_Topics

ID: 352

asTarget

Convert to a valid Node Target

The classes listed below implement the method asTarget. This is used widely in the **Node** classes (**Group** and **Synth**) to convert non-Node objects to an appropriate target. This allows nil and instances of **Server** to be used as targets. This can be useful when writing classes which create nodes internally, but in most cases there should be little need to call asTarget in normal use.

Node - Returns the instance of Node itself. The subclasses of **Node** (**Synth** and **Group**) are valid targets and require no conversion.

Server - Returns a **Group** object representing the **default_group** of this instance of **Server**. Note that this object may not be identical with other objects representing the default group, but will be equivalent.

```
s = Server.default;
 g = s.asTarget; // the default group of s
h = s.defaultGroup; // and again
g == h;  // true
g === h; // false
```

Nil - Returns a **Group** object representing the **default_group** of the current default **Server**.

```
s = Server.default;
g = nil.asTarget;
g == s.defaultGroup; // true
```

Integer - Returns a **Group** object representing a group node on the current default **Server** with this **Integer** as its node ID number. **Note:** Although this can be convenient in some cases, it does not create the corresponding node on the default server, nor does it check to make sure that it exists. As well it does not directly access the server's

NodeIDAllocator, so duplication of node IDs is possible. For these reasons this method should be used with care. When not dealing with the default Server, Group-basicNew is safer and simpler, as otherwise one needs to set the server instance variable to ensure correct targeting.

```
/////// Showing the problems

s = Server.default;
s.boot;
g = s.nextNodeID.asTarget;
 x = Synth         "default"    // but g doesn't exist on the server
 s.sendMsg(*g.addToHeadMsg);  // now it's sent to the default server, in the default group
x = Synth.head(g, "default");  // now this works
x.free; g.free;

 // if not using the default Server Integer-asTarget can be problematic

Server.default = Server.local;
 Server                  // quit the default server
i = Server.internal; i.boot;
g = i.nextNodeID.asTarget;
i.sendMsg(*g.addToHeadMsg);  // seems to work, but...
  x = Synth         "default"    // oops, this goes to the default server, so Group not Found
 g.server == Server           // true, so that's the problem
g.server = i;
 x = Synth         "default"    // now to the right place
x.free; g.free;

/////// A more practical example

s = Server.default;
s.boot;
s.sendMsg(\g_new, x = s.nextNodeID);
 // ...

  // now if we need to use Node objects for some reason
y = Synth.head(x.asTarget, "default");

 // this is simpler than Group.basicNew(s, x);, providing you're using the default server:
z = Synth.head(Group.basicNew(s, x), "default");
```

```
y.free; z.free; x.asTarget.free;
```

ID: 353

Creating Stand-Alone Applications

Contents

Introduction

On OS X, applications are special directories known as "bundles." This allows you to create stand-alone applications running SuperCollider code that are opaque, in the sense that the user does not need to install SuperCollider, run SuperCollider code, or even know that SuperCollider is involved. (Of course, your application must be open-source and comply with the GPL.) This is useful for distributing applications to the general public, or for creating special-purpose applications for your own use.

Creating a stand-alone application using the Finder

Step 1: Make a copy of the SuperCollider application, and name it whatever you'd like your application to be called. Then control-click on the copy and select "Show Package Contents" from the contextual menu. You'll get a new Finder window that shows you the inside of the application bundle. Navigate to Contents/Resources. This is where the folders that normally reside outside the SuperCollider application will go.

Step 2: Option-drag the following items into the Resources folder as needed:

• **SCClassLibrary**–this is absolutely necessary, and of course can contain your own classes.
• **plugins**–your application will launch without this, but you won't get very far making sound.
• **recordings**–not necessary if your application does not allow recording, or if you have the user select the recording directory.
• **sounds**–if your application requires any preexisting soundfiles, put them here.
• **synthdefs**–not absolutely necessary, but very handy.
• **Help**–if your users aren't dealing with SuperCollider code, they don't need the SuperCollider help files, but you should create a **Help.help.rtf** file for your own application

and put it in here.
• If you want your application to be able to use the local server, drag in **scsynth**. This is not recommended for most applications, though; see **Adding your own behavior**, below.

Step 3: Edit **MainMenu.nib** by double-clicking on it. It will in open in InterfaceBuilder, presenting you with a "virtual" menubar that you can modify as you wish (deleting the **Help** menu, for example). The exception is the name of the application menu ("SuperCollider")–you can change it here, but the change will not be picked up by the application. See step 4.

Step 4: Edit **English.lproj/InfoPlist.strings**, replacing "SuperCollider" with the name of your application in the **CFBundleName** line. This will show up in your application's main menu.

Now you have an application that behaves exactly like SuperCollider, but is entirely self-contained (can be dragged anywhere with no accompanying files) and has its own name. To make it behave like an ordinary application (with its own main window, etc.), see **Adding your own behavior** below.

Now when you launch the copy, it will use the items you dragged in (and any modifications of the class library will be stored there instead of the original). You can drag it anywhere, give it to other people, etc.

Creating a stand-alone application using Xcode

In the SuperCollider source, there's an Xcode project called **xSC_StandAlone**. This is an Xcode 2.1 project, and will not open in earlier versions. It has two targets:

• **SC_StandAlone**. This creates a simple but complete application in its final form.
• **SC_StandAlone_Devel**. This creates a development version of the same application, the differences being that **Main-startup** is not modified, so that the server and post windows are created, and that an alias of its class library is used, so that any modifications are made to the original in the code folder, rather than in the bundle.

To create your own application:

Step 1: Make a copy of the **SCSA_Aux** folder (inside the **SuperCollider3** folder), and name it after your application (e.g. **MyApp_Aux**). Change the names of all its subfolders similarly. Edit the **MainMenu.nib** and **English.lproj/InfoPlist.strings** files

in both **MyApp_Resources** and **MyApp_Devel_Resources** as described in the previous section.

Step 2: Make a copy of the **xSC_StandAlone** project, name it after your application, and open it in Xcode. Delete the **SCSA_Resources** and **SCSA_Devel_Resources** groups (choosing **Delete References Only** at the dialog), and add **MyApp_Resources** and **MyApp_Devel_Resources** to the SC_StandAlone and SC_StandAlone_Devel targets, respectively (selecting the **Recursively Add Groups** option in the dialog). Make sure you're adding them to the right targets.

Delete the **SCSA_Library** folder (choosing **Delete References Only** at the dialog), and add **MyApp_Library** in its place, selecting the **Create Folder** option (it should show up as a blue folder instead of a yellow one). Drag it to the **Copy Files** build phase of the **SC_StandAlone** target. You don't need to add it to the **SC_StandAlone_Devel** target.

Step 3: Rename the the **SC_StandAlone_Devel** target to **MyApp_Devel** (by control-clicking on the target icon and selecting "Rename" from the contextual menu). Double-click on the **MyApp_Devel** target. In the Base Product Name field, change "SC_StandAlone_Devel" to "MyApp_Devel". Do the same thing in the **Info.Plist Entries** section. Click the **Run Script** box. In the script that appears, replace **SCSA** with **MyApp** wherever it appears (three times).

Step 4: Repeat step 3 for the SC_StandAlone target (leaving off _Devel, of course).

Step 5: Build away!

Adding your own behavior

Using either of the above methods, you've created an application that behaves exactly like SuperCollider. To run your own code on launch and simulate an ordinary application rather than a development environment, you'll need to modify Main-startup. Here's an example (the same code used by the **SC_StandAlone** target):

```
startup {
super.startup;

Document.startup;

  // set the 's' interpreter variable to the internal server.
```

```
    // You should use the internal server for standalone applications--
  // otherwise, if your application has a problem, the user will
  // be stuck with a process, possibly making sound, that he won't know
  // how to kill.
interpreter.s = Server.internal;

  // server windows turned off for stand-alone application
//  Server.internal.makeWindow;
//  Server.local.makeWindow;

  // Start the application using internal server
interpreter.s.waitForBoot({
var sb, demo;
sb = SCWindow.screenBounds;

demo = SCSA_Demo.new(
    "the cheese stands alone"
Rect(
(sb.width - SCSA_Demo.width) * 0.5,
(sb.height - SCSA_Demo.height) * 0.5,
SCSA_Demo.width,
SCSA_Demo.height
),
interpreter.s
);
demo.front;
   // Close post window after application launches.  If you want
   // to hide it completely, put this line after Document.startup instead.
Document.closeAll(false);
}, 25);

  // You probably don't want to include this, since the user won't have it
//  " /scwork/startup.rtf".loadPaths;
```

The class **SCSA_Demo** contains the entire application, including the main window. This is the tidiest way to work, and requires the least modification to SuperCollider. If you don't want to write a class, you can execute an .rtf file instead:

```
interpreter.executeFile(String.scDir ++ "/myapp.rtf");
```

However, any sizable application will benefit from encapsulation in classes.

Note that the example uses the internal server. This is part and parcel of keeping the application stand-alone; it shouldn't call extraneous processes behind the user's back that will persist if the application fails. If you need to use the local server for some reason, make sure **scsynth** is in the application's bundle.

ID: 354

Snooping around SuperCollider

You can inspect much of the internal structure of the class library and other data structures.
This can often be useful for research and debugging purposes.

Class Definitions, Implementations, and References

Selecting the name of any Class (e.g. **Object**) and typing cmd-j will open its class definition file.

Selecting the name of any method (e.g. **play**) and typing cmd-y will open a window showing all implementations of that method and their arguments. Selecting one of those classes and methods (e.g. **Sample-play**) and typing cmd-j will open the class definition at that method. (Note that cmd-y only shows implementations, and does not indicate inheritance).

Selecting any text (e.g. **SCWindow** or **asStream**) and typing shift-cmd-y will open a window showing all references to the selected text, i.e. each place it is used within the class library. (This will not find methods calls compiled with special byte codes like 'value'.)

SC has a graphical Class browser which will show all methods, arguments, subclasses, instance variables and class variables. (Currently this is only OSX.) Using the browser's buttons you can easily navigate to the class' superclass, subclasses, class source, method source, helpfile (if there is one), check references or implementation of methods, or even open a web browser to view the corresponding entry in the online CVS repository. (Note that the web repository is a backup often a day or two behind the status of what is available to developers.)

SequenceableCollection

Snooping in Classes

The **Class** help file documents some of these snooping methods.
Even though you may access these data structures, if you store things into them, you may break something.

```
Collection                // print all instance methods defined for this class

Collection                      // print all class methods defined for this class


// The following three include inherited methods

Collection.methods.collect(_.name);
      // print all instance methods that instances of this class respond to

Collection.class.methods.collect(_.name);
      // print all class methods that this class responds to

Collection                    // print all instance and class methods that this class responds to

Collection                 // print instance methods of this class and superclasses, in alpha order
      // also shows from which class the method is inherited
      // does not include Object or Class methods
      // for class methods, do Meta_Collection.dumpMethodList

Collection                   // dump all subclasses of this class

Collection                   // dump all subclasses, in alphabetical order

SCWindow                    // dump all instance variable names of this class

SCWindow                     // dump all class variable names of this class

SCWindow                        // the path to the file that defined this class

(
// print all classes whose names start with 'F'
Class.allClasses.do({ arg class;
if (class.name.asString.at(0) == $F, { class.name.postln; });
})
)

(
// find and print all class variable names defined in the system
Class.allClasses.do({ arg class;
```

```
if (class.classVarNames.notNil, {
  // classVarNames is an Array of Symbols
class.classVarNames.do({ arg varname;
(class.name.asString ++ " " ++ varname.asString).postln;
})
});
});
)

(
// find and print all methods that contain "ascii"
Class.allClasses.do({ arg class;
class.methods.do({ arg sel;
if(sel.name.asString.find("ascii").notNil) {
(class.name.asString + "-" + sel.name).postln;
}
});
}); ""
)
```

Snooping in Methods

Same thing goes here, if you store things into Methods, you may break something.

```
Collection            'select'   // does it have this method?

Array            'select'   // this class doesn't

Array                        'select'   // climb the class tree to find the method

Collection            'select'        // find a method object

Collection.findMethod('select').argNames.dump; // dump its argument names

Collection            'select'                // dump its local variable names

// dump its code.  mostly for debugging the compiler.
Collection.findMethod('select').dumpByteCodes;

Collection               'select'   // a shorter version of the above
```

```
                // this is a Function

                    // get its FunctionDef

{ 1 + 2 }.def.dumpByteCodes; // dump its code.
```

Snooping in Windows

```
(
// create some windows to snoop in
5.do({ arg i;
var w, b;
w = SCWindow.new("snoop " ++ i.asString,
Rect.new( 200 + 400.rand, 69 + 300.rand, 172, 90 ));
w.front;
b = SCButton.new( w, Rect.new( 23, 28, 127, 25 ));
b.states = [["BLAM-O", Color.red]];
}))

SCWindow                    // dump a list of all open SCWindows

// a little more helpful, dump their names
SCWindow.allWindows.collect({ arg w; w.name }).postln;

(
// change background colors of all open windows
SCWindow.allWindows.do({ arg window;
window.view.background = Color.new(0.5 + 0.5.rand, 0.5 + 0.5.rand, 0.5 + 0.5.rand);
}))

SCWindow            // close all the windows (This will close the server windows)
```

Snooping in SynthDefs

```
// First execute this:
(
    SynthDef "Help-SnoopSynthDef"
{ arg out=0;
```

```
Out.ar(out, PinkNoise.ar(0.1))
});
)

f.dumpUGens; // get the ugens, listed in order of execution, with rate, index and
    // inputs
```

Snooping in the Interpreter

When evaluating text in the interpreter, the variable 'this' always refers to the interpreter.

```
this       // display the values of all the interpreter variables a-z

this           // set all variables a-z to nil

    this         "(1 + 2).postln"   // compile some text into a Function

           // see, g is a Function

         // evaluate g

this           "(1 + 2).postln"   // interpret some text

this               "1 + 2"   // interpret some text and print the result
```

ID: 355

Multichannel Expansion

Multiple channels of audio are represented as Arrays.

```
s.boot;
// one channel
{ Blip.ar(800,4,0.1) }.play;

// two channels
{ [ Blip.ar(800,4,0.1), WhiteNoise.ar(0.1) ] }.play;
```

Each channel of output will go out a different speaker, so your limit here is two for a stereo output. If you have a supported multi channel audio interface or card then you can output as many channels as the card supports.

All UGens have only a single output. This uniformity facilitates the use of array operations to perform manipulation of multi channel structures.

In order to implement multichannel output, UGens create a separate UGen known as an **OutputProxy** for each output. An OutputProxy is just a place holder for the output of a multichannel UGen. OutputProxies are created internally, you never need to create them yourself, but it is good to be aware that they exist so you'll know what they are when you run across them.

```
// look at the outputs of Pan2:
Pan2.ar(PinkNoise.ar(0.1), FSinOsc.kr(3)).dump;

play({ Pan2.ar(PinkNoise.ar(0.1), FSinOsc.kr(1)); });
```

When an **Array** is given as an input to a unit generator it causes an array of multiple copies of that unit generator to be made, each with a different value from the input array. This is called multichannel expansion. All but a few special unit generators perform multichannel expansion. Only Arrays are expanded, no other type of Collection, not even subclasses of Array.

```
{ Blip.ar(500,8,0.1) }.play // one channel

// the array in the freq input causes an Array of 2 Blips to be created :
```

```
{ Blip.ar([499,600],8,0.1) }.play // two channels

Blip                    // one unit generator created.

Blip                        // two unit generators created.
```

Multichannel expansion will propagate through the expression graph. When a unit generator constructor is called with an array of inputs, it returns an array of instances. If that array is the input to another constructor, then another array is created, and so on.

```
{ RLPF.ar(Saw.ar([100,250],0.05), XLine.kr(8000,400,5), 0.05) }.play;

// the [100,250] array of frequency inputs to Saw causes Saw.ar to return
// an array of two Saws, that array causes RLPF.ar to create two RLPFs.
// Both RLPFs share a single instance of XLine.
```

When a constructor is parameterized by two or more arrays, then the number of channels created is equal to the longest array, with parameters being pulled from each array in parallel. The shorter arrays will wrap.

for example, the following:

```
Pulse.ar([400, 500, 600],[0.5, 0.1], 0.2)
```

is equivalent to:

```
[ Pulse.ar(400,0.5,0.2), Pulse.ar(500,0.1,0.2), Pulse.ar(600,0.5,0.2) ]
```

A more complex example based on the Saw example above is given below. In this example, the XLine is expanded to two instances, one going from 8000 Hz to 400 Hz and the other going in the opposite direction from 500 Hz to 7000 Hz. These two XLines are 'married' to the two Saw oscillators and used to parameterize two copies of RLPF. So on the left channel a 100 Hz Saw is filtered from 8000 Hz to 400 Hz and on the right channel a 250 Hz Saw is filtered from 500 Hz to 7000 Hz.

```
{ RLPF.ar(Saw.ar([100,250],0.05), XLine.kr([8000,500],[400,7000],5), 0.05) }.play;
```

Protecting arrays against expansion

Some unit generators such as **Klank** require arrays of values as inputs. Since all arrays

are expanded, you need to protect some arrays by a **Ref** object. A Ref instance is an object with a single slot named 'value' that serves as a holder of an object. Ref.new(object) one way to create a Ref, but there is a syntactic shortcut. The backquote ‘ is a unary operator that is equivalent to calling Ref.new(something). So to protect arrays that are inputs to a Klank or similar UGens you write:

```
Klank.ar(‘[[400,500,600],[1,2,1]], z)
```

You can still create multiple Klanks by giving it an array of Ref'ed arrays.

```
Klank.ar([ ‘[[400,500,600],[1,2,1]],  ‘[[700,800,900],[1,2,1]] ], z)
```

is equivalent to:

```
[ Klank.ar(‘[[400,500,600],[1,2,1]], z),  Klank.ar(‘[[700,800,900],[1,2,1]], z)]
```

Reducing channel expansion with Mix

The **Mix** object provides the means for reducing multichannel arrays to a single channel.

```
Mix               // array of channels
```

is equivalent to:

```
          // mixed to one
```

Mix is more efficient than using + since it can perform multiple additions at a time. But the main advantage is that it can deal with situations where the number of channels is arbitrary or determined at runtime.

```
// three channels of Pulse are mixed to one channel
{ Mix.new(  Pulse.ar([400, 501, 600], [0.5, 0.1], 0.1) ) }.play
```

Multi channel expansion works differently for Mix. Mix takes one input which is an array (one not protected by a Ref). That array does not cause copies of Mix to be made. All elements of the array are mixed together in a single Mix object. On the other hand if the array contains one or more arrays then multi channel expansion is performed one level down. This allows you to mix an array of stereo (two element) arrays resulting in one two channel array. For example:

```
Mix                                 // input is an array of stereo pairs
```

is equivalent to:

```
// mixed to a single stereo pair
[ Mix.new( [a, c, e] ), Mix.new( [b, d, f] ) ]
```

Currently it is not recursive. You cannot use Mix on arrays of arrays of arrays.

Here's a final example illustrating multi channel expansion and Mix. By changing the variable 'n' you can change the number of voices in the patch. How many voices can your machine handle?

```
(
{
var n;
 n = 8; // number of 'voices'
 Mix      // mix all stereo pairs down.
    Pan2     // pan the voice to a stereo position
   CombL     // a comb filter used as a string resonator
    Dust     // random impulses as an excitation function
     // an array to cause expansion of Dust to n channels
     // 1 means one impulse per second on average
Array.fill(n, 1),
     0.3 // amplitude
),
    0.01, // max delay time in seconds
    // array of different random lengths for each 'string'
Array.fill(n, {0.004.rand+0.0003}),
    4 // decay time in seconds
),
   Array                    // give each voice a different pan position
)
)
}.play;
)
```

Using flop for multichannel expansion

The method *flop* swaps columns and rows, allowing to derive series of argument sets:

```
(
SynthDef "help_multichannel"    | out=0, freq=440, mod=0.1, modrange=20|
Out.ar(out,
SinOsc.ar(
LFPar.kr(mod, 0, modrange) + freq
) * EnvGate(0.1)
)
}).send(s);
)

(
var freq, mod, modrange;

freq = Array.exprand(8, 400, 5000);
mod = Array.exprand(8, 0.1, 2);
modrange = Array.rand(8, 0.1, 40);

fork {
[\freq, freq, \mod, mod, \modrange, modrange].flop.do { | args|
args.postln;
  Synth "help_multichannel"
0.3.wait;
}
};
)
```

Similarly, **Function-flop** returns an unevaluated function that will expand to its arguments when evaluated:

```
(
SynthDef("blip", { | freq|  Out.ar(0, Line.ar(0.1, 0, 0.05, 1, 0, 2)
* Pulse.ar(freq * [1, 1.02])) }).send(s);

      | dur=1, x=1, n=10, freq=400|
fork { n.do {
if(x.coin) { Synth("blip", [\freq, freq]) };
(dur / n).wait;
} }
```

```
}.flop;
)

a.value(5, [0.3, 0.3, 0.2], [12, 32, 64], [1000, 710, 700]);
```

ID: 356

play

start a process

this message is of common use in sc. Different objects respond to it in various ways, but the simple meaning is: start a process.
It is usually implemented by objects in contributed libraries as well.

play usually returns the playing object which might not be the same as the one the message was sent to.

opposite: **stop**

clock.play(stream)
returns: the clock

```
(
r = Routine.new({ "...playing".postln; 1.wait; "ok, that was it".postln });
 SystemClock
)
```

routine.play(clock)
returns: the routine

```
Routine.new({ "...playing".postln; 1.wait; "ok, that was it".postln }).play;
```

stream.play(clock)
returns the stream
the stream will loop until it returns nil

```
FuncStream({ "ok, that was it".postln; 1 }).play;
```

pausestream.play(clock) / task.play(clock)
returns the stream

```
a = PauseStream.new(FuncStream.new({ "ok, that was it".postln; 1 }));
a.play;
a.stop;
a.play;
a.stop;

a = Task.new({ loop({ "ok, that was it".postln; 1.wait; }) });
a.play;
a.stop;
```

pattern.play(clock, protoEvent)
returns: an **EventStreamPlayer**

```
(
 Pseq
Pbind(\freq, Pn(500, 1)),
Pbind(\dur, Pn(0.1, 1))
], 2).play;
)
```

The following play messages both cause a **SynthDef** to be written, send it to the server and start a synth with it there.

they should not be used in quickly running automated processes,
as there are more efficient alternatives (see **SynthDefsVsSynths)**

synthDef.play(target, args, addAction)
returns: a **Synth**

note that you need an **out ugen** to hear the result.

in sc3 synths can write to busses using an out ugen or they can also just run without any writing activity.
one example of a synth without an out ugen is **SendTrig**, whereas you find different examples
of how to write to the busses in the helpfiles: **Out / ReplaceOut / XOut / OffsetOut**
as what is audible through the hardware busses must be written on them, one way or another
an out ugen is always needed.
some operations provide an out ugen **internally**: see for example function.play, which plays out
to a bus number provided in the argument passed to .play

```
(
x = SynthDef("test", { arg out, amp=0.1;
var sound;
sound = PinkNoise.ar(amp * [1,1]);
Out.ar(out, sound);
}).play;
)

//set the synth
x.set(\amp, 0.2);
//free the synth
x.free;
```

note: Synth.play(function), is synonymous. for backwards compatibility with sc2

function.play(target, outbus, fadeTime)
returns: a **Synth**
adds to tail by default
soft fade in and out.

```
//note that you can use Out ugens but you do not need to
{ PinkNoise.ar(0.1*[1,1]) }.play;

//mouse x controls level
```

```
{ XOut.ar(0, MouseX.kr(0,1), PinkNoise.ar(0.1*[1,1])) }.play;
```

the arguments of the function **are the same as in SynthDef.new**, which means you cannot pass
in any value - they are used to build Controls for the synth.

you can set the controls in the running synth:

```
x = { arg freq=900; Resonz.ar(PinkNoise.ar([1,1]), freq, 0.01) }.play(s, 0);
x.set(\freq, 1400);
x.free;
```

which this is equivalent to:

```
(
x = SynthDef("nextrandomname", { arg freq=900;
Out.ar(0, Resonz.ar(PinkNoise.ar([1,1]), freq, 0.01))
}).play(s);
)
x.set(\freq, 1400);
x.free;
```

more modularity can be achieved by using **[Instr]** from the **CRUCIAL-LIBRARY**.

ID: 357

publishing code

A computer language like sc makes it easy to share code with others. Apart from sending examples and pieces by email, you can use public repositories to make them available:

If you like to add some code, or even create your own little webspace on the supercollider wiki:

http://swiki.hfbk-hamburg.de:8888/MusicTechnology/6
(passwd: sc, user: sc)

Or you can join the electronic-life forum:

http://www.electroniclife.co.uk/scforum

You may think of an artistic **licencing scheme**, such as *creative commons* or *gpl*:

http://www.opensource.org
http://creativecommons.org

Generally, making software public that is written in sclang or software that includes changes to scserver
requires the sourcecode to be published together with it.
For more information, see http://www.gnu.org/copyleft/gpl.html

Quite often, code written by others is copied, modified and used in pieces or software. If the code was published without any specific licence, it is always good to at least **mention the**
original authors in your work in order to avoid later conflicts.

ID: 358

Randomness in SC

As in any computer program, there are no "truly random" number generators in SC. They are pseudo-random, meaning they use very complex, but deterministic algorithms to generate sequences of numbers that are long enough and complicated enough
to seem "random" for human beings. (i.e. the patterns are too complex for us to detect.)

If you start a random number generator algorithm with the same "seed" number several times, you get the same sequence of random numbers.
(See example below, *randomSeed*)

Create single random numbers:

1. Between zero and <number>:

```
// evenly distributed.

// probability decreases linearly from 0 to <number>.
```

2. Between -<number> and <number>:

```
// evenly distributed.

// probability is highest around 0,
// decreases linearly toward +-<number>.

// quasi-gaussian, bell-shaped distribution.
```

3. Within a given range:

```
// linear distribution in the given range.

// exponential distribution;
```

```
    // both numbers must have the same sign.
```

Test them multiple times with a do loop:

```
20.do({ 5.rand.postln; });   // evenly distributed

                                // probability decreases linearly from 0 to 1.0


20.do({ 5.0.rand2.postln; });  // even

                                 // probability is highest around 0,
      // decreases linearly toward +-<number>.

20.do({ 1.0.sum3rand.postln; }); // quasi-gaussian, bell-shaped.
```

Collect the results in an array:

```
Array.fill(10, { 1000.linrand }).postln;

// or more compact:

{ 1.0.sum3rand }.dup(100)

// or:

({ 1.0.sum3rand } !  100)
```

You can seed a random generator in order to repeat the same sequence of random numbers:

```
(
5.do({
thisThread.randSeed = 4;
```

```
Array.fill(10, { 1000.linrand}).postln;
});
)

// Just to check, no seeding:

(
5.do({ Array.fill(10, { 1000.linrand}).postln; });
)

// see also [randomSeed].
```

Demonstrate the various statistical distributions visually, with histograms:

[plot may not work in non-Mac SC3 versions.]

```
Array.fill(500, {  1.0.rand }).plot("Sequence of 500x 1.0.rand");

Array.fill(500, {  1.0.linrand }).plot("Sequence of 500x 1.0.linrand");

Array.fill(500, {  1.0.sum3rand }).plot("Sequence of 500x 1.0.sum3rand");

// Use a histogram to display how often each (integer)
// occurs in a collection of random numbers, :
(
var randomNumbers, histogram, maxValue = 500, numVals = 10000;

randomNumbers = Array.fill(numVals, { maxValue.rand; });

histogram = Signal.newClear(maxValue);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = each.asInteger;
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1)
});

histogram.plot("histogram for rand 0 - " ++ maxValue);
```

```
)
```

A histogram for linrand:

```
(
var randomNumbers, histogram, maxValue = 500.0, numVals = 10000;

randomNumbers = Array.fill(numVals, { maxValue.linrand; });

histogram = Signal.newClear(maxValue);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = each.asInteger;
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1)
});

histogram.plot("histogram for linrand 0 - " ++ maxValue);
)
```

A histogram for bilinrand:

```
(
var randomNumbers, histogram, minValue = -250, maxValue = 250, numVals = 10000, numBins = 500;

randomNumbers = Array.fill(numVals, { maxValue.bilinrand; });
histogram = Signal.newClear(numBins);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = ( each - minValue );
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1)
});

histogram.plot("histogram for bilinrand" + minValue + "to" + maxValue);
)
```

A histogram for exprand:

```
(
var randomNumbers, histogram, minValue = 5.0, maxValue = 500, numVals = 10000, numBins = 500;

randomNumbers = Array.fill(numVals, { exprand(minValue, maxValue); });
histogram = Signal.newClear(numBins);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = ( each - minValue ).round(1).asInteger;
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1);
});

histogram.plot("histogram for exprand:  " ++ minValue ++ " to " ++ maxValue);
)
```

And for sum3rand (cheap quasi-gaussian):

```
(
var randomNumbers, histogram, minValue = -250, maxValue = 250, numVals = 10000, numBins = 500;

randomNumbers = Array.fill(numVals, { maxValue.sum3rand; });
histogram = Signal.newClear(numBins);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = ( each - minValue ).round(1).asInteger; // catch float indices.
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1)
});

histogram.plot("histogram for sum3rand " ++ minValue ++ " to " ++ maxValue);
)
```

All of the single-number methods also work for (Sequenceable)Collections,

simply by applying the given random message to each element of the collection:

```
                \aSymbol                // note:  Symbols are left as they are.
List[ 10, -3.0, \aSymbol ].sum3rand.postln;
```

Arbitrary random distributions

An *integral table* can be used to create an arbitrary random distribution quite efficiently. The table
building is expensive though. The more points there are in the randomTable the more accurate the
distribution is, of course

```
(
var randomNumbers, histogram, distribution, randomTable, randTableSize=200;
var minValue = -250, maxValue = 250, numVals = 10000, numBins = 500;

// create some random distribution with values between 0 and 1
distribution = Array.fill(randTableSize,
{ arg i; (i/ randTableSize * 35).sin.max(0) * (i / randTableSize) }
);

// render a randomTable
randomTable = distribution.asRandomTable;

// get random numbers, scale them

randomNumbers = Array.fill(numVals, { randomTable.tableRand * (maxValue - minValue) + minValue; });

histogram = Signal.newClear(numBins);

randomNumbers.do({ arg each; var count, histoIndex;
histoIndex = ( each - minValue ).round(1).asInteger; // catch float indices.
count = histogram.at(histoIndex);
histogram.put(histoIndex, count + 1)
```

```
});

                "this is the histogram we got"
                   "this was the histogram we wanted"
)
```

Random decisions:

coin simulates a coin toss and results in true or false.
1.0 is always true, 0.0 is always false, 0.5 is 50:50 chance.

```
20.do({ 0.5.coin.postln });
```

biased random decision can be simulated bygenerating a single value and check against a threshhold:

```
20.do({ (1.0.linrand > 0.5).postln });
20.do({ (exprand(0.05, 1.0) > 0.5).postln });
```

Generating Collections of random numbers:

```
    // size, minVal, maxVal
Array.rand(7, 0.0, 1.0).postln;

// is short for:

Array.fill(7, { rrand(0.0, 1.0) }).postln;


  // size, minVal, maxVal
List.linrand(7, 10.0, 15.0).postln;

// is short for:
```

```
List.fill(7, { 10 + 5.0.linrand }).postln;

Signal.exprand(10, 0.1, 1);

Signal.rand2(10, 1.0);
```

Random choice from Collections

choose equal chance for each element.

```
10.do({ [ 1, 2, 3 ].choose.postln });
```

Weighted choice:

wchoose(weights) An array of weights sets the chance for each element.

```
10.do({ [ 1, 2, 3 ].wchoose([0.1, 0.2, 0.7]).postln });
```

Randomize the order of a Collection:

scramble

```
List[ 1, 2, 3, 4, 5 ].scramble.postln;
```

Randomly group a Collection:

curdle(probability)

The probability argument sets the chance that two adjacent elements will be separated.

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ].curdle(0.2).postln; // big groups
```

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ].curdle(0.75).postln; // small groups
```

Random signal generators, i.e. UGens:

PinkNoise
WhiteNoise
GrayNoise
BrownNoise
PinkerNoise
ClipNoise
LFNoise0
LFNoise1
LFNoise2
LFClipNoise
LFDNoise0
LFDNoise1
LFDNoise3
LFDClipNoise
Dust
Dust2
Crackle
LinCong
Latoocarfian
Rossler [not installed yet]
NoahNoise [not installed yet]

UGens that generate random numbers once, or on trigger:

Rand uniform distribution of float between (lo, hi), as for numbers.
IRand uniform distribution of integer numbers.

TRand uniform distribution of float numbers, triggered
TIRand uniform distribution of integer numbers, triggered
LinRand skewed distribution of float numbers, triggered
NRand sum of n uniform distributions, approximates gaussian distr. with higher n.
ExpRand exponential distribution
TExpRand exponential distribution, triggered
CoinGate statistical gate for a trigger
TWindex triggered weighted choice between a list

Like using randSeed to set the random generatorsfor each thread in sclang,
you can choose which of several random generators on the server to use,
and you can reset (seed) these random generators:
RandID
RandSeed

UGens that generate random numbers on demand ("Demand UGens"):

Dwhite
Dbrown
Diwhite
Dibrown
Drand
Dxrand

see random patterns with analogous names

Random Patterns:

```
Prand              inf    // choose randomly one from a list ( list, numRepeats)
Pxrand                 // choose one element from a list, no repeat of previous choice
Pwhite             // within range [<hi>, <lo>], choose a random value.
Pbrown               // within range [<hi>, <lo>], do a random walk
       // with a maximum <step> to the next value.

Pwrand                                  // choose from a list, probabilities by weights
```

```
Pshuf                       // scramble the list, then repeat that order <repeats> times.

Pwalk( (0 ..  10), Prand([ -2,-1, 1, 2], inf));  // random walk.

Pfsm      // random finite state machine pattern, see its help file.
              // see also MarkovSet on sc-swiki

Pseed                   // sets the random seed for that stream.


// some basic examples

(
Pbind(\note, Prand([ 0, 2, 4 ], inf),
\dur, 0.2
).play;
)

(
Pbind
\note, Pxrand([ 0, 2, 4 ], inf),
\dur, 0.2
).play;
)

(
Pbind
\note, Pwrand([ 0, 2, 4 ], [0.1, 0.3, 0.6], inf),
\dur, 0.2
).play;
)

(
Pbind
\midinote, Pwhite(48, 72, inf),
\dur, 0.2
).play;
)

(
```

```
Pbind
\midinote, Pbrown(48, 72, 5, inf),
\dur, 0.2
).play;
)
```

ID: 359

Unit Generators and Synths

A unit generator is an object that processes or generates sound. There are many classes of unit generators, all of which derive from the class **UGen**.

Unit generators in SuperCollider can have many inputs, but always have a single output. Unit generator classes which would naturally have several outputs such as a panner, return an array of unit generators when instantiated. The convention of having only a single output per unit generator allows the implementation of multiple channels by using arrays of unit generators. (See **MultiChannel** for more details.)

Instantiation. Audio Rate, Control Rate

A unit generator is created by sending the 'ar' or 'kr' message to the unit generator's class object. The 'ar' message creates a unit generator that runs at audio rate. The 'kr' message creates a unit generator that runs
at control rate. Control rate unit generators are used for low frequency or slowly changing control signals. Control rate unit generators produce only a single sample per control cycle and therefore use less processing power than audio rate unit generators.

The input parameters for a unit generator are given in the documentation for that class.

```
FSinOsc                    // create a sine oscillator at 800 Hz, phase 0.0, amplitude 0.2
```

A unit generator's signal inputs can be other unit generators, scalars, or arrays of unit generators and scalars.

SynthDefs and Synths

In order to play a unit generator one needs to compile it in a **SynthDef** and play it on the server in a **Synth**. A synth node is a container for one or more unit generators that execute together. A SynthDef is like a kind of pattern for creating synth nodes on the server.

```
        // boot the local server

// compile and send this def
SynthDef.new("FSinOsc-test", { Out.ar(0, FSinOsc.ar(800, 0, 0.2)) }).send(s); // out channel 0
```

```
// now create a Synth object which represents a synth node on the server
    Synth     "FSinOsc-test"

// free the synth
x.free;
```

The synth node created above could also be created using 'messaging style', thus saving the overhead of a clientside Synth object:

```
n = s.nextNodeID;
          "/s_new"  "FSinOsc-test"
s.sendMsg("/n_free", n);
```

Because any expression returns its value, we can nest the first two lines above for convenience. (See **Expression-Sequence** for more detail.)

```
s.sendMsg("/s_new", "FSinOsc-test", n = s.nextNodeID;);
s.sendMsg("/n_free", n);
```

It is VERY important and useful to understand the messaging structure which underlies the clientside Synth, Group, Buffer, and Bus objects. See **NodeMessaging**,**Tutorial**, and **ClientVsServer** for more detail.

As a convenience the 'play' method of class **Function** will compile a SynthDef and create and play a synth using the function for you. With this method an **Out** ugen will be created for you if you do not do so explicitly.

```
    FSinOsc                     // create and play a sine oscillator at 800 Hz
```

Building Patches

You can do math operations on unit generators and the result will be another unit generator. Doing math on unit generators is not doing any signal calculation itself - it is building the network of unit generators that will execute once they are played in a Synth. This is the essential thing to understand: Synthesis networks, or in other words signal flow graphs are created by executing expressions of unit generators. The following expression creates a flow graph whose root is an instance of **BinaryOpUGen** which performs the '+' operation. Its inputs are the **FSinOsc** and **BrownNoise** unit generators.

```
FSinOsc                    BrownNoise          // press enter and look at the post window

{FSinOsc.ar(800, 0.0, 0.2) + BrownNoise.ar(0.2)}.play; // play it
```

ID: 360

Using Extensions

See also: **[Writing-Classes] [Writing_Unit_Generators]**

SC supports extensions to its class library, documentation, and server UGen plugins. Extensions should be packaged as a single folder containing all three (for convenient addition or removal), or any combination, which can then be placed in platform-specific extension directories in order to be included.

How Extensions Folders Should be Organised

Class files and UGen plugins are recognised by their file extensions (.sc for the former and .scx or .sco for the latter). Anything placed within a folder named help/ or test/ (case insensitive) will be ignored when compiling the class library or loading plugins, but anything in the former will be added to the search path for help files.

Here is an example folder layout:

MyExtension/

classes/
myClass.sc myUGens.sc
plugins/
myUGenPlugins.scx
help/
myClass.rtf myUGen1.rtf myUGen2.rtf

Platform Specific Directories

User-specific

OSX /Library/Application Support/SuperCollider/Extensions/
Linux /share/SuperCollider/Extensions/

```
Platform.userExtensionDir
```

System-wide (apply to all users)

OSX /Library/Application Support/SuperCollider/Extensions/
Linux /usr/local/share/SuperCollider/Extensions/

```
Platform.systemExtensionDir
```

ID: 361

Using the Startup File

Once the class library is finished compiling the interpreter looks for a file at the path " /scwork/startup.rtf" and if such a file exists, executes any code within it. (This happens within Main-startup.) By creating a file in this location you can make user specific customizations. A common example would be to alter the options of the local and internal Servers:

```
// placing the following code in " /scwork/startup.rtf" will cause these modifications to be made
// at startup

Server                                          // change number of input and output channels
Server.local.options.numInputBusChannels = 4;
Server.internal.options.numOutputBusChannels = 4;
Server.internal.options.numInputBusChannels = 4;

Server.local.options.blockSize = 128; // increase block size
Server.internal.options.blockSize = 128;

Server                                      // increase sampling rate (if your hardware supports it)
Server.internal.options.sampleRate = 96000;

// change the standard synthDef directory to a custom one:
SynthDef.synthDefDir = " /scwork/synthdefs".standardizePath;

// change the standard archive path to a custom one:
Archive.archiveDir = " /scwork".standardizePath;
```

Naturally the file must contain only valid SC expressions.

21 Quarks

ID: 362

Quarks

A Quark is a package of SC classes, helpfiles, C++ source for UGens and other SC code

The Quarks subversion repository is hosted at

```
https://svn.sourceforge.net/svnroot/quarks
Quarks.repos.url
```

Quarks are downloaded onto your local machine in this folder:

```
{Application Support Directory}/quarks
Quarks.local.path
```

They are installed into your Extensions folder via the Quarks class which simply adds/removes symlinks.
see [Using-Extensions]

You may use standard SVN tools within Quarks.local.path to work with Quarks packages.

The Quarks class interface mirrors the svn commands with the correct paths inserted.

It is easiest to just check out all available quarks:

```
Quarks.checkoutAll

// post those check out
Quarks.checkedOut


Quarks         "testquark"

// recompile your library now
```

```
TestQuark.sayHello

Quarks.listInstalled

Quarks.uninstall("testquark")

// recompile your library now

// this returns quark objects for those installed
Quarks.installed

// quark objects for each one in repository that you could checkout
Quarks.repos.quarks

// quark objects that you could install
Quarks.local.quarks
```

To add your own quarks simply place the folder into local quarks/
and
svn add quarkname
place a quarkname.quark file into the DIRECTORY
svn add DIRECTORY/quarkname.quark

and

svn commit

Check status of your working copy:

```
Quarks.status

// update all
Quarks.update

// update specific
Quarks        "testquark"

// this would add the folder but would not add the DIRECTORY entry
```

```
/*
  Quarks     "somethingnew"
*/
```

Not checking out all Quarks

At a future time when the universe is populated by billions of quarks you may wish to not clutter your local.

In the {Application Support Directory}/quarks/DIRECTORY folder there is a quark specification file for each Quark named quarkname.quark

This is an sc code file containing a dictionary specifying name, version number and relative path.

You may check out only quarks and quarks/DIRECTORY

```
Quarks.repos.checkDir
```

This will post the command to checkout quarks non-recursively and then to checkout quarks/DIRECTORY

```
You want to do:
svn co -N url path

to non-recursively checkout only the top level quarks folder, and then check out individual



// update DIRECTORY
Quarks.updateDirectory

// list all quarks in the repository
Quarks.repos.quarks

// check one just out
Quarks.checkout("test")
```

```
Quarks.update("test")

Quarks.status("test")

Quarks.checkedOut

Quarks.installed.postln
```

22 Scheduling

ID: 363

AppClock

superclass: Clock

SystemClock is more accurate, but cannot call Cocoa primitives.
AppClock is less accurate (uses NSTimers) but can call Cocoa primitives.

You will need to use the SystemClock to get accurate/musical scheduling.

*sched(delta,task)

the float you return specifies the delta to resched the function for

```
  AppClock.sched(0.0,{ arg time;
    ["AppClock has been playing for ",time]
rrand(0.1,0.9)
});
```

returning nil will stop the task from being rescheduled

```
  AppClock.sched(2.0,{
   "2.0 seconds later"
nil
});
```

*clear

clear the AppClock's scheduler to stop it

```
  AppClock
```

*play(task)

The task/Routine yields a float value indicating the delta (secs) for the AppClock to wait
until resuming the Routine.

```
(
var w, r;
```

```
w = SCWindow("trem", Rect(512, 256, 360, 130));
w.front;
r = Routine({ arg appClockTime;
["AppClock has been playing for secs:",appClockTime].postln;
60.do({ arg i;
0.05.yield;
w.bounds = w.bounds.moveBy(10.rand2, 10.rand2);
w.alpha = cos(i*0.1pi)*0.5+0.5;
});
1.yield;
w.close;
});
AppClock.play(r);
)
```

*tick

AppClock.tick is called periodically by the SuperCollider application itself. This updates the Scheduler and causes any scheduled tasks to be executed. You should never call this method yourself.

ID: 364

Condition block execution of a thread

superclass: Object

***new(test)** create a new instance, set the **test** variable.

test return the test variable (boolean)

wait wait until the condition is true and signalled

hang(val) wait for **val** time, regardless of test

signal if **test** is true, reschedule blocked threads

unhang resume threads

```
// example


(
    Condition     false

Routine
1.wait;
 "waited for 1 second"
1.wait;
 "waited for another second, now waiting for you ...  "
c.wait;
 "the condition has stopped waiting."
1.wait;
 "waited for another second"
 "waiting for you ...  "
c.test = false;
```

```
c.wait;
 "the condition has stopped waiting."
1.wait;
"the end".postln;
}.play;
)

// continue
(
c.test = true;
c.signal;
)

// a typical use is a routine that can pause under certin conditions:
(
c = Condition.new;
fork { loop { 1.wait; "going".postln; c.wait } };
)
c.test = true; c.signal;
c.test = false;



// the same, using hang

(
c = Condition.new;

Routine
1.wait;
 "waited for 1 second"
1.wait;
 "waited for another second, now waiting for you ...  "
c.hang;
 "the condition has stopped waiting."
1.wait;
 "waited for another second"
 "waiting for you ...  "
c.hang;
 "the condition has stopped waiting."
```

```
}.play;
)

// continue
c.unhang;
```

ID: 365

Scheduler schedules functions to be evaluated in the future

superclass: Object

***new(clock, drift)**
clock: a clock, like SystemClock.
drift: if true, it will schedule the events relative to Main.elapsedTime, otherwise to the current seconds of the scheduler.

play(aTask)
schedules the task to play, with the delta time returned from it.

sched(delta, aTask)
schedule the task

advance(bySeconds)
advance time by n seconds. Any task that is scheduled within the new time, is evaluated, and, if it returns a new time, rescheduled.

seconds_(newSeconds)
set new time. Any task that is scheduled within the new time, is evaluated, and, if it returns a new time, rescheduled.

isEmpty
returns whether the scheduling queue is empty.

clear
clear the scheduliong queue

```
// example:

    Scheduler SystemClock
```

```
a.sched(3, { "now it is 3 seconds.".postln; nil });
a.sched(5, { "now it is 5 seconds.".postln; nil });
a.sched(1, { "now it is 1 second.".postln; nil });

a.advance(0.5);
a.advance(0.5);
a.advance(2);
a.advance(2);

// the beats, seconds and clock are passed into the task function:
a.sched(1, { arg beats, secs, clock; [beats, secs, clock].postln });
a.advance(1);

// the scheduling is relative to "now":
a.sched(3, { "now it was 3 seconds.".postln });
a.sched(5, { "now it was 5 seconds.".postln });
a.sched(1, { "now it was 1 second.".postln });

a.advance(0.5);
a.advance(0.5);
a.advance(2);
a.advance(2);

// play a Routine or a task:
a.play(Routine { 5.do { arg i; i.postln; 1.yield } });
a.advance(0.9);



// scheduling tasks


(
    Scheduler TempoClock

Task
inf.do { | i|
("next " ++ i ++  " in task." + Main.elapsedTime).postln;
0.5.wait;
```

```
}
}.play(x)
)

x.advance(0.1);
x.seconds;
x.advance(5);
x.seconds;

(
Routine
loop { x.advance(0.1); 0.1.wait }
}.play;
)

(
Task { 5.do {
x.advance(1);
2.0.rand.wait;
}
}.play;
)

x.advance(8.1);

Pbind(\degree, Pseries(0, 2, 8), \dur, 0.25).play(x);

(
Task { 5.do {
x.advance(0.20);
1.0.wait;
}
}.play;
)
```

ID: 366

SystemClock

superclass: Clock

SystemClock is more accurate, but cannot call Cocoa primitives.
AppClock is less accurate (uses NSTimers) but can call Cocoa primitives.

*sched(delta,task)

the float you return specifies the delta to resched the function for

```
  SystemClock.sched(0.0,{ arg time;
time.postln;
rrand(0.1,0.9)
});
```

returning nil will stop the task from being rescheduled

```
  SystemClock.sched(2.0,{
   "2.0 seconds later"
nil
});
```

*clear

clear the SystemClock's scheduler to stop it

```
  SystemClock
```

*schedAbs(time,task)

```
SystemClock.schedAbs( (thisThread.seconds + 4.0).round(1.0),{ arg time;
("the time is exactly " ++ time.asString
    ++ " seconds since starting SuperCollider")
});
```

*play(task)

Calls to the cocoa framework (including all GUI) may not be made directly from actions triggered by SystemClock or incoming socket messages

(OSCresponder).

To get around this, use `{ }.defer`
This will execute the function using the AppClock and is equivalent to `AppClock.sched(0, function):`

```
(
var w, r;
w = SCWindow("trem", Rect(512, 256, 360, 130));
w.front;
r = Routine({ arg time;
60.do({ arg i;
0.05.yield;
{
w.bounds = w.bounds.moveBy(10.rand2, 10.rand2);
w.alpha = cos(i*0.1pi)*0.5+0.5;
}.defer;
});
1.yield;
w.close;
});
SystemClock.play(r);
)
```

This example is only to show how to make calls to Cocoa/GUI when scheduling with the SystemClock.
If you only wish to control the GUI, use AppClock.

ID: 367

Task a pauseable process

superclass: PauseStream

Task is a pauseable process. It is implemented by wrapping a PauseStream around a Routine. Most of it's methods (start, stop, reset) are inherited from PauseStream.

Task.new(func, clock)
func - A **Function** to be evaluated.
clock - A Clock in which to play the **Routine**. If you do not provide a Clock the default is an instance of TempoClock. Remember that methods which call Cocoa primitives (i.e. GUI functions) must be played in AppClock.

```
t = Task({
50.do({ arg i;
i.squared.postln;
0.5.wait
});
});

t.start;
t.stop;
t.start;
t.reset;
t.stop;
```

ID: 368

TempoClock tempo based scheduler

TempoClock is a scheduler like SystemClock, but it schedules relative to a tempo in beats per second.

***new(tempo, beats, seconds)**
Create a new TempoClock scheduler with the given tempo and starting times.

If not given, **tempo** defaults to one, **beats** defaults to zero and **seconds** defaults to the current elapsed time.

stop
destroy the scheduler. releases the OS thread running the scheduler.

tempo
get the current tempo in beats per second.

tempo_(beatsPerSecond)
set the current tempo.
t.tempo = 2.0;
or
t.tempo_(2.0);

beatDur
get the current beat duration in seconds.

elapsedBeats
get the current elapsed time in beats.
This is equivalent to: tempoClock.secs2beats(Main.elapsedTime).
It is often preferrable to use **beats** instead of **elapsedBeats** because **beats** uses a thread's logical time.

beats
Returns the appropriate beat time of the clock from any thread. If the receiver is the clock of the current thread, this returns the current logical time: thisThread.beats. If

the receiver is not the current thread's clock then this translates the current thread's logical time in seconds to this clock's logical time in beats.

schedAbs(beat,function)
Schedule a function to be evaluated at a particular beat.

sched(delta,function)
Schedule a function to be evaluated delta beats from the current logical time in this clock. If the receiver is the clock of the current thread, the delta is applied to the current logical time. If the receiver is not the current thread's clock then the delta is applied to the clock's elapsed time.

clear
remove all tasks from the scheduling queue.

permanent_(bool)
if false (default) the clock is stopped (and thus removed) on cmd-period.
If set to true it persists, just like TempoClock.default does.

beats2secs(beats)
convert absolute beats to absolute seconds. Only works for times in the current tempo. If the tempo changes any computed time in future will be wrong.

secs2beats(seconds)
convert absolute seconds to absolute beats. Only works for times in the current tempo. If the tempo changes any computed time in future will be wrong.

Example:

```
//////////////////////

    TempoClock     // create a TempoClock

// schedule an event at next whole beat
t.schedAbs(t.beats.ceil, { arg beat, sec; [beat, sec].postln; 1 });


t.tempo = 2;
t.tempo = 4;
```

```
t.tempo = 0.5;
t.tempo = 1;

t.clear;

t.schedAbs(t.beats.ceil, { arg beat, sec; [beat, sec].postln; 1 });

t.stop;

////////////////////////

(
// get elapsed time, round up to next second
v = Main.elapsedTime.ceil;

// create two clocks in a 5:2 relation, starting at time v.
t = TempoClock(1, 0, v);
u = TempoClock(0.4, 0, v);

// start two functions at beat zero in each clock.
t.schedAbs(0, { arg beat, sec; [\t, beat, sec].postln; 1 });
u.schedAbs(0, { arg beat, sec; [\u, beat, sec].postln; 1 });
)



(
u.tempo = u.tempo * 3;
t.tempo = t.tempo * 3;
)

(
u.tempo = u.tempo * 1/4;
t.tempo = t.tempo * 1/4;
)


(
t.stop;
u.stop;
)
```

```
//////////////////////

(
// get elapsed time, round up to next second
v = Main.elapsedTime.ceil;

// create two clocks, starting at time v.
t = TempoClock(1, 0, v);
u = TempoClock(1, 0, v);

// start two functions at beat zero in each clock.
// t controls u's tempo.  They should stay in sync.
t.schedAbs(0, { arg beat, sec; u.tempo = t.tempo * [1,2,3,4,5].choose; [\t, beat, sec].postln; 1 });
u.schedAbs(0, { arg beat, sec; [\u, beat, sec].postln; 1 });
)



(
u.tempo = u.tempo * 3;
t.tempo = t.tempo * 3;
)

(
u.tempo = u.tempo * 1/4;
t.tempo = t.tempo * 1/4;
)



(
t.stop;
u.stop;
)
```

23 ServerArchitecture

ID: 369

Buffer client-side representation of a buffer on a server

superclass: Object

Buffer encapsulates a number of common tasks, OSC messages, and capabilities related to server-side buffers, which are globally available arrays of floats. These are commonly used to hold sampled audio, such as a soundfile loaded into memory, but can be used to hold other types of data as well. They can be freed or altered even while being accessed. Buffers are commonly used with **PlayBuf**, **RecordBuf**, **DiskIn**, **DiskOut**, **BufWr**, **BufRd**, and other UGens. (See their individual help files for more examples.) See **Server-Architecture** for more details.

Buffer Numbers and Allocation

Although the number of buffers on a server is set at the time it is booted, memory must still be allocated within the server app before they can hold values. (At boot time all buffers have a size of 0.)

Server-side buffers are identified by number, starting from 0. When using Buffer objects, buffer numbers are automatically allocated from the Server's bufferAllocator, unless you explicitly supply one. When you call **.free** on a Buffer object it will release the buffer's memory on the server, and free the buffer number for future reallocation. See **ServerOptions** for details on setting the number of available buffers.

Normally you should not need to supply a buffer number. You should only do so if you are sure you know what you are doing.

You can control which allocator determines the buffer index numbers by setting the server options **blockAllocClass** variable prior to booting the server. Two allocators are available to support different kinds of applications. See the **[ServerOptions]** help file for details.

Multichannel Buffers

Multichannel buffers interleave their data. Thus the actual number of available values when requesting or setting values by index using methods such as set, setn, get, getn, etc., is equal to numFrames * numChannels. Indices start at 0 and go up to (numFrames * numChannels) - 1. In a two channel buffer for instance, index 0 will be the first value

of the first channel, index 1 will be the first value of the second channel, index 2 will be the second value of the first channel, and so on.

In some cases it is simpler to use multiple single channel buffers instead of a single multichannel one.

Completion Messages and Action Functions

Many buffer operations (such as reading and writing files) are asynchronous, meaning that they will take an arbitrary amount of time to complete. Asynchronous commands are passed to a background thread on the server so as not to steal CPU time from the audio synthesis thread. Since they can last an aribitrary amount of time it is convenient to be able to specify something else that can be done immediately on completion. The ability to do this is implemented in two ways in Buffer's various methods: completion messages and action functions.

A completion message is a second OSC command which is included in the message which is sent to the server. (See **NodeMessaging** for a discussion of OSC messages.) The server will execute this immediately upon completing the first command. An action function is a **Function** which will be evaluated when the client receives the appropriate reply from the server, indicating that the previous command is done. Action functions are therefore inherently more flexible than completion messages, but slightly less efficient due to the small amount of added latency involved in message traffic. Action functions are passed the Buffer object as an argument when they are evaluated.

With Buffer methods that take a completion message, it is also possible to pass in a function that returns an OSC message. As in action functions this will be passed the Buffer as an argument. It is important to understand however that this function will be evaluated after the Buffer object has been created (so that its bufnum and other details are accessible), but *before* the corresponding message is sent to the server.

Bundling

Many of the methods below have two versions: a regular one which sends its corresponding message to the server immediately, and one which returns the message in an **Array** so that it can be added to a bundle. It is also possible to capture the messages generated by the regular methods using Server's automated bundling capabilities. See **Server** and **bundledCommands** for more details.

Accessing Instance Variables

The following variables have getter methods.

server - Returns the Buffer's **Server** object.

bufnum - Returns the buffer number of the corresponding server-side buffer.

numFrames - Returns the number of sample frames in the corresponding server-side buffer. Note that multichannel buffers interleave their samples, so when dealing with indices in methods like get and getn, the actual number of available values is numFrames * numChannels.

numChannels - Returns the number of channels in the corresponding server-side buffer.

sampleRate - Returns the sample rate of the corresponding server-side buffer.

path - Returns a string containing the path of a soundfile that has been loaded into the corresponding server-side buffer.

```
s.boot;
b = Buffer.alloc(s,44100 * 8.0,2);
b.bufnum.postln;
b.free;
```

Creation with Immediate Memory Allocation

*alloc(server, numFrames, numChannels, completionMessage, bufnum)

Create and return a Buffer and immediately allocate the required memory on the server. The buffer's values will be initialised to 0.0.

server - The server on which to allocate the buffer. The default is the default **Server**.
numFrames - The number of frames to allocate. Actual memory use will correspond to numFrames * numChannels.
numChannels - The number of channels for the Buffer. The default is 1.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
    // Allocate 8 second stereo buffer
s.boot;
b = Buffer.alloc(s, s.sampleRate * 8.0, 2);
b.free;
```

*allocConsecutive(numBufs, server, numFrames, numChannels, completionMessage, bufnum)

Allocates a range of consecutively-numbered buffers, for use with UGens like **[VOsc]** and **[VOsc3]** that require a contiguous block of buffers, and returns an array of corresponding Buffer objects.

numBufs - The number of consecutively indexed buffers to allocate.
server - The server on which to allocate the buffers. The default is the default **Server**.
numFrames - The number of frames to allocate in each buffer. Actual memory use will correspond to numFrames * numChannels.
numChannels - The number of channels for each buffer. The default is 1.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed each Buffer and its index in the array as arguments when evaluated.
bufnum - An explicitly specified buffer number for the initial buffer. Generally this is not needed.

N.B. You must treat the array of Buffers as a group. Freeing them individually or resuing them can result in allocation errors. You should free all Buffers in the array at the same time by iterating over it with **do**.

```
s.boot;
  // allocate an array of Buffers and fill them with different harmonics
(
b = Buffer.allocConsecutive(8, s, 4096, 1, { | buf, i|
buf.sine1Msg((1..((i+1)*6)).reciprocal) // completion Messages
});
)
a = { VOsc.ar(SinOsc.kr(0.5, 0).range(b.first.bufnum + 0.1, b.last.bufnum - 0.1)
[440, 441], 0, 0.2) }.play;


a.free;


  // iterate over the array and free it
```

```
b.do(_.free);
```

*read(server, path, startFrame, numFrames, action, bufnum)

Allocate a buffer and immediately read a soundfile into it. This method sends a query message as a completion message so that the Buffer's instance variables will be updated automatically. **N.B.** You cannot rely on the buffer's instance variables being instantly updated, as there is a small amount of latency involved. **action** will be evaluated upon receipt of the reply to the query, so use this in cases where access to instance variables is needed.

server - The server on which to allocate the buffer.
path - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
    // read a soundfile
s.boot;
  b = Buffer          "sounds/a11wlk01.wav"


  // now play it
(
x = SynthDef("help-Buffer",{ arg out = 0, bufnum;
Out.ar( out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum))
)
}).play(s,[\bufnum, b.bufnum ]);
)
x.free; b.free;


  // with an action function
  // note that the vars are not immediately up-to-date
(
b = Buffer.read(s, "sounds/a11wlk01.wav", action:  { arg buffer;
```

```
("After update:" + buffer.numFrames).postln;
x = { PlayBuf.ar(1, buffer.bufnum, BufRateScale.kr(buffer.bufnum)) }.play;
});
("Before update:" + b.numFrames).postln;
)
x.free; b.free;
```

*readChannel(server, path, startFrame, numFrames, channels, action, bufnum)

As ***read** above, but takes an Array of channel indices to read in, allowing one to read only the selected channels.

server - The server on which to allocate the buffer.
path - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
channels - An **Array** of channels to be read from the soundfile. Indices start from zero. These will be read in the order provided.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
s.boot;
  // first a standard read so we can see what's in the file
  b = Buffer         "sounds/SinedPink.aiff"
  // "sounds/SinedPink.aiff" contains SinOsc on left, PinkNoise on right
b.plot;
b.free;

  // Now just the sine
b = Buffer.readChannel(s, "sounds/SinedPink.aiff", channels:  [0]);
b.plot;
b.free;

  // Now just the pink noise
b = Buffer.readChannel(s, "sounds/SinedPink.aiff", channels:  [1]);
```

```
b.plot;
b.free;

  // Now reverse channel order
b = Buffer.readChannel(s, "sounds/SinedPink.aiff", channels:  [1, 0]);
b.plot;
b.free;
```

*readNoUpdate(server, path, startFrame, numFrames, completionMessage, bufnum)

As ***read** above, but without the automatic update of instance variables. Call **updateInfo** (see below) to update the vars.

server - The server on which to allocate the buffer.
path - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
        // with a completion message
s.boot;
(
SynthDef("help-Buffer",{ arg out=0,bufnum;
Out.ar( out,
PlayBuf.ar(1,bufnum,BufRateScale.kr(bufnum))
)
}).send(s);

y = Synth.basicNew("help-Buffer"); // not sent yet
b = Buffer.readNoUpdate(s,"sounds/a11wlk01.wav",
completionMessage:  { arg buffer;
    // synth add its s_new msg to follow
    // after the buffer read completes
y.newMsg(s,[\bufnum,buffer.bufnum],\addToTail)
});
```

```
)
  // note vars not accurate
b.numFrames; // nil
b.updateInfo;
b.numFrames; // 26977
  // when done...
y.free;
b.free;
```

*cueSoundFile(server, path, startFrame, numChannels, bufferSize, completionMessage)

Allocate a buffer and preload a soundfile for streaming in using **DiskIn**.

server - The server on which to allocate the buffer.
path - A **String** representing the path of the soundfile to be read.
startFrame - The frame of the soundfile that **DiskIn** will start playing at.
numChannels - The number of channels in the soundfile.
bufferSize - This must be a multiple of (2 * the server's block size). 32768 is the default and is suitable for most cases.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

```
s.boot;
(
SynthDef("help-Buffer-cue",{ arg out=0,bufnum;
Out.ar(out,
DiskIn.ar( 1, bufnum )
)
}).send(s);
)

(
s.makeBundle(nil, {
b = Buffer.cueSoundFile(s,"sounds/a11wlk01-44_1.aiff", 0, 1);
y = Synth.new("help-Buffer-cue", [\bufnum,b.bufnum], s);
});
)
b.free; y.free;
```

*loadCollection(server, collection, numChannels, action)

Load a large collection into a buffer on the server. Returns a Buffer object. This is accomplished through writing the collection to a **SoundFile** and loading it from there. For this reason this method will only work with a server on your local machine. For a remote server use ***sendCollection**, below. The file is automatically deleted after loading. This allows for larger collections than **setn**, below, and is in general the safest way to get a large collection into a buffer. The sample rate of the buffer will be the sample rate of the server on which it is created.

server - The server on which to create the buffer.
collection - A subclass of **Collection** (i.e. an **Array**) containing *only* floats and integers. Multi-dimensional arrays will not work.
numChannels - The number of channels that the buffer should have. Note that buffers interleave multichannel data. You are responsible for providing an interleaved collection if needed. Multi-dimensional arrays will not work.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.

```
s.boot;
(
a = FloatArray.fill(44100 * 5.0, {1.0.rand2}); // 5 seconds of noise
b = Buffer.loadCollection(s, a);
)

  // test it
b.get(20000,{| msg|  (msg == a[20000]).postln});
  // play it
x = { PlayBuf.ar(1, b.bufnum, BufRateScale.kr(b.bufnum), loop:  0) * 0.5 }.play;
b.free; x.free;

  // interleave a multi-dimensional array
(
l = Signal.sineFill(16384, Array.fill(200, {0}).add(1));
r = Array.fill(16384, {1.0.rand2});
  m = [Array                // a multi-dimensional array
  m = m.lace(32768); // interleave the two collections
b = Buffer.loadCollection(s, m, 2, {| buf|
x = { PlayBuf.ar(2, buf.bufnum, BufRateScale.kr(buf.bufnum), loop:  1) * 0.5 }.play;
```

```
});
)
b.plot;
x.free; b.free;
```

*sendCollection(server, collection, numChannels, wait, action)

Stream a large collection into a buffer on the server using multiple **setn** messages. Returns a Buffer object. This allows for larger collections than **setn**, below. This is not as safe as ***loadCollection**, above, but will work with servers on remote machines. The sample rate of the buffer will be the sample rate of the server on which it is created.

server - The server on which to create the buffer.
collection - A subclass of **Collection** (i.e. an **Array**) containing *only* floats and integers. Multi-dimensional arrays will not work.
numChannels - The number of channels that the buffer should have. Note that buffers interleave multichannel data. You are responsible for providing an interleaved collection if needed. Multi-dimensional arrays will not work. See the example in ***loadCollection**, above, to see how to do this.
wait - An optional wait time between sending **setn** messages. In a high traffic situation it may be safer to set this to something above zero, which is the default.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.

```
s.boot;
(
a = Array.fill(2000000,{ rrand(0.0,1.0) }); // a LARGE collection
b = Buffer.sendCollection(s, a, 1, 0, {arg buf; "finished".postln;});
)
b.get(1999999, {| msg|  (msg == a[1999999]).postln});
b.free;
```

*loadDialog(server, path, startFrame, numFrames, bufnum)

As ***read** above, but gives you a load dialog window to browse for a file. OSX only.

server - The server on which to allocate the buffer.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.

numFrames - The number of frames to read. The default is -1, which will read the whole file.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
s.boot;
(
b = Buffer.loadDialog(s, action:  { arg buffer;
x = { PlayBuf.ar(b.numChannels, buffer.bufnum, BufRateScale.kr(buffer.bufnum)) }.play;
});
)
x.free; b.free;
```

Creation without Immediate Memory Allocation

*new(server, numFrames, numChannels, bufnum)

Create and return a new Buffer object, without immediately allocating the corresponding memory on the server. This combined with 'message' methods can be flexible with bundles.

server - The server on which to allocate the buffer. The default is the default **Server**.
numFrames - The number of frames to allocate. Actual memory use will correspond to numFrames * numChannels.
numChannels - The number of channels for the Buffer. The default is 1.
bufnum - An explicitly specified buffer number. Generally this is not needed.

```
s.boot;
b = Buffer.new(s, 44100 * 8.0, 2);
c = Buffer.new(s, 44100 * 4.0, 2);
  b.query; // numFrames = 0
s.sendBundle(nil, b.allocMsg, c.allocMsg); // sent both at the same time
  b.query; // now it's right
c.query;
b.free; c.free;
```

alloc(completionMessage)
allocMsg(completionMessage)

Allocate the necessary memory on the server for a Buffer previously created with ***new**, above.

completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

```
s.boot;
b = Buffer.new(s, 44100 * 8.0, 2);
  b.query; // numFrames = 0
b.alloc;
  b.query; // numFrames = 352800
b.free;
```

allocRead(argpath, startFrame, numFrames, completionMessage)
allocReadMsg(argpath, startFrame, numFrames, completionMessage)

Read a soundfile into a buffer on the server for a Buffer previously created with ***new**, above. Note that this will not autoupdate instance variables. Call **updateInfo** in order to do this.

argpath - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

```
s.boot;
b = Buffer.new(s);
  b.allocRead("sounds/a11wlk01.wav"
x = { PlayBuf.ar(1, b.bufnum, BufRateScale.kr(b.bufnum), loop:  1) * 0.5 }.play;
x.free; b.free;
```

allocReadChannel(argpath, startFrame, numFrames, channels, completionMessage)
allocReadChannelMsg(argpath, startFrame, numFrames, channels, completionMessage)

As **allocRead** above, but allows you to specify which channels to read.

argpath - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
channels - An **Array** of channels to be read from the soundfile. Indices start from zero. These will be read in the order provided.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

```
s.boot;
b = Buffer.new(s);
  // read only the first channel (a Sine wave) of a stereo file
b.allocReadChannel("sounds/SinedPink.aiff", channels:  [0]);
x = { PlayBuf.ar(1, b.bufnum, BufRateScale.kr(b.bufnum), loop:  1) * 0.5 }.play;
x.free; b.free;
```

Instance Methods

read(path, fileStartFrame, numFrames, bufStartFrame, leaveOpen, action); readMsg(path, fileStartFrame, numFrames, bufStartFrame, leaveOpen, completionMessage);

Read a soundfile into an already allocated buffer. Note that if the number of frames in the file is greater than the number of frames in the buffer, it will be truncated. Note that **readMsg** will not auto-update instance variables. Call **updateInfo** in order to do this.

path - A **String** representing the path of the soundfile to be read.
fileStartFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
bufStartFrame - The index of the frame in the buffer at which to start reading. The default is 0, which is the beginning of the buffer.
leaveOpen - A boolean indicating whether or not the Buffer should be left 'open'. For use with **DiskIn** you will want this to be true, as the buffer will be used for streaming the soundfile in from disk. (For this the buffer must have been allocated with a multiple

of (2 * synth block size). A common number is 32768 frames. **cueSoundFile** below, provides a simpler way of doing this.) The default is false which is the correct value for all other cases.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

readChannel(path, fileStartFrame, numFrames, bufStartFrame, leaveOpen, channels, action);
readChannelMsg(path, fileStartFrame, numFrames, bufStartFrame, leaveOpen, channels, completionMessage);

As **read** above, but allows you to specify which channels to read.

path - A **String** representing the path of the soundfile to be read.
fileStartFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
numFrames - The number of frames to read. The default is -1, which will read the whole file.
bufStartFrame - The index of the frame in the buffer at which to start reading. The default is 0, which is the beginning of the buffer.
leaveOpen - A boolean indicating whether or not the Buffer should be left 'open'. For use with **DiskIn** you will want this to be true, as the buffer will be used for streaming the soundfile in from disk. (For this the buffer must have been allocated with a multiple of (2 * synth block size). A common number is 32768 frames. **cueSoundFile** below, provides a simpler way of doing this.) The default is false which is the correct value for all other cases.
channels - An **Array** of channels to be read from the soundfile. Indices start from zero. These will be read in the order provided. The number of channels requested must match this Buffer's **numChannels**.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

cueSoundFile(path, startFrame, completionMessage)

cueSoundFileMsg(path, startFrame, completionMessage)

A convenience method to cue a soundfile into the buffer for use with a **DiskIn**. The buffer must have been allocated with a multiple of (2 * the server's block size) frames. A common size is 32768 frames.

path - A **String** representing the path of the soundfile to be read.
startFrame - The first frame of the soundfile to read. The default is 0, which is the beginning of the file.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

```
s.boot;
    //create with cueSoundFile class method
b = Buffer.cueSoundFile(s, "sounds/a11wlk01-44_1.aiff", 0, 1);
x = { DiskIn.ar(1, b.bufnum) }.play;
  b.close; // must call close in between cueing
  // now use like named instance method, but different arguments
  b.cueSoundFile("sounds/a11wlk01-44_1.aiff"
  // have to do all this to clean up properly!
x.free; b.close; b.free;
```

write(path, headerFormat, sampleFormat, numFrames, startFrame, leaveOpen, completionMessage)
writeMsg(path, headerFormat, sampleFormat, numFrames, startFrame, leaveOpen, completionMessage)

Write the contents of the buffer to a file. See **SoundFile** for information on valid values for **headerFormat** and **sampleFormat**.

path - A **String** representing the path of the soundfile to be written.
numFrames - The number of frames to write. The default is -1, which will write the whole buffer.
startFrame - The index of the frame in the buffer from which to start writing. The default is 0, which is the beginning of the buffer.
leaveOpen - A boolean indicating whether or not the Buffer should be left 'open'. For use with **DiskOut** you will want this to be true. The default is false which is the correct value for all other cases.
completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

free(completionMessage)
freeMsg(completionMessage)

Release the buffer's memory on the server and return the bufferID back to the server's buffer number allocator for future reuse.

completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

zero(completionMessage)
zeroMsg(completionMessage)

Sets all values in this buffer to 0.0.

completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

set(index,float ... morePairs)
setMsg(index,float ... morePairs)

Set the value in the buffer at **index** to be equal to **float**. Additional pairs of indices and floats may be included in the same message. Note that multichannel buffers interleave their sample data, therefore the actual number of available values is equal to numFrames * numChannels. Indices start at 0.

```
s.boot;
b = Buffer.alloc(s, 4, 2);
  b.set(0, 0.2, 1, 0.3, 7, 0.4); // set the values at indices 0, 1, and 7.
b.getn(0, 8, {| msg|  msg.postln});
b.free;
```

setn(startAt,values ... morePairs)
setnMsg(startAt,values ... morePairs)

Set a contiguous range of values in the buffer starting at the index **startAt** to be equal to the **Array** of floats or integers, **values**. The number of values set corresponds to the size of **values**. Additional pairs of starting indices and arrays of values may be included in the same message. Note that multichannel buffers interleave their sample data, therefore the actual number of available values is equal to numFrames * numChannels. You

are responsible for interleaving the data in **values** if needed. Multi-dimensional arrays will not work. Indices start at 0.

N.B. The maximum number of values that you can set with a single setn message is 1633 when the server is using UDP as its communication protocol. Use **loadCollection** and **sendCollection** to set larger ranges of values.

```
s.boot;
b = Buffer.alloc(s,16);
b.setn(0, Array.fill(16, { rrand(0,1) }));
b.getn(0, b.numFrames, {| msg|  msg.postln});
b.setn(0, [1, 2, 3], 4, [1, 2, 3]);
b.getn(0, b.numFrames, {| msg|  msg.postln});
b.free;
```

loadCollection(collection, startFrame, action)

Load a large collection into this buffer. This is accomplished through writing the collection to a **SoundFile** and loading it from there. For this reason this method will only work with a server on your local machine. For a remote server use **sendCollection**, below. The file is automatically deleted after loading. This allows for larger collections than **setn**, above, and is in general the safest way to get a large collection into a buffer. The sample rate of the buffer will be the sample rate of the server on which it was created. The number of channels and frames will have been determined when the buffer was allocated. You are responsible for making sure that the size of **collection** is not greater than **numFrames**, and for interleaving any data if needed.

collection - A subclass of **Collection** (i.e. an **Array**) containing *only* floats and integers. Multi-dimensional arrays will not work.
startFrame - The index of the frame at which to start loading the collection. The default is 0, which is the start of the buffer.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.

```
s.boot;
(
v = Signal.sineFill(128, 1.0/[1,2,3,4,5,6]);
b = Buffer.alloc(s, 128);
)
```

```
(
b.loadCollection(v, action:  {| buf|
x = { PlayBuf.ar(buf.numChannels, buf.bufnum, BufRateScale.kr(buf.bufnum), loop:  1)
* 0.2 }.play;
});
)
x.free; b.free;

  // interleave a multi-dimensional array
(
l = Signal.sineFill(16384, Array.fill(200, {0}).add(1));
r = Array.fill(16384, {1.0.rand2});
  m = [Array                // a multi-dimensional array
  m = m.lace(32768); // interleave the two collections
  b = Buffer.alloc(s, 16384, 2);
)
(
b.loadCollection(m, 0, {| buf|
x = { PlayBuf.ar(2, buf.bufnum, BufRateScale.kr(buf.bufnum), loop:  1) * 0.5 }.play;
});
)
b.plot;
x.free; b.free;
```

sendCollection(collection, startFrame, wait, action)

Stream a large collection into this buffer using multiple **setn** messages. This allows for larger collections than **setn**. This is not as safe as **loadCollection**, above, but will work with servers on remote machines. The sample rate of the buffer will be the sample rate of the server on which it is created.

collection - A subclass of **Collection** (i.e. an **Array**) containing *only* floats and integers. Multi-dimensional arrays will not work.
startFrame - The index of the frame at which to start streaming in the collection. The default is 0, which is the start of the buffer.
wait - An optional wait time between sending **setn** messages. In a high traffic situation it may be safer to set this to something above zero, which is the default.
action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.

```
s.boot;
(
a = Array.fill(2000000,{ rrand(0.0,1.0) });
b = Buffer.alloc(s, 2000000);
)
b = b.sendCollection(a, action:  {arg buf; "finished".postln;});
b.get(1999999, {| msg|  (msg == a[1999999]).postln});
b.free;
```

get(index, action)
getMsg(index)

Send a message requesting the value in the buffer at **index**. **action** is a **Function** which will be passed the value as an argument and evaluated when a reply is received.

```
s.boot;
b = Buffer.alloc(s,16);
b.setn(0, Array.fill(16, { rrand(0.0, 1.0) }));
b.get(0, {| msg|  msg.postln});
b.free;
```

getn(index, count, action)
getMsg(index, count)

Send a message requesting the a contiguous range of values of size **count** starting from **index**. **action** is a **Function** which will be passed the values in an **Array** as an argument and evaluated when a reply is received. See **setn** above for an example.

N.B. The maximum number of values that you can get with a single getn message is 1633 when the server is using UDP as its communication protocol. Use **loadToFloatArray** and **getToFloatArray** to get larger ranges of values.

loadToFloatArray(index, count, action)

Write the buffer to a file and then load it into a **FloatArray**. This is safer than **getToFloatArray** but only works with a server on your local machine. In general this is the safest way to get a large range of values from a server buffer into the client app.

index - The index in the buffer to begin writing from. The default is 0.

count - The number of values to write. The default is -1, which writes from index until the end of the buffer.
action - A **Function** which will be passed the resulting **FloatArray** as an argument and evaluated when loading is finished.

```
s.boot;
  b = Buffer        "sounds/a11wlk01.wav"
  // same as Buffer.plot
b.loadToFloatArray(action:  { arg array; a = array; {a.plot;}.defer; "done".postln;});
b.free;
```

getToFloatArray(index, count, wait, timeout, action)

Stream the buffer to the client using a series of **getn** messages and put the results into a **FloatArray**. This is more risky than **loadToFloatArray** but does works with servers on remote machines. In high traffic situations it is possible for data to be lost. If this method has not received all its replies by **timeout** it will post a warning saying that the method has failed. In general use **loadToFloatArray** instead wherever possible.

index - The index in the buffer to begin writing from. The default is 0.
count - The number of values to write. The default is -1, which writes from index until the end of the buffer.
wait - The amount of time in seconds to wait between sending getn messages. Longer times are safer. The default is 0.01 seconds which seems reliable under normal circumstances. A setting of 0 is *not*recommended.
timeout - The amount of time in seconds after which to post a warning if all replies have not yet been received. the default is 3.
action - A **Function** which will be passed the resulting **FloatArray** as an argument and evaluated when all replies have been received.

```
s.boot;
  b = Buffer        "sounds/a11wlk01.wav"
  // like Buffer.plot
b.getToFloatArray(wait:0.01,action:{arg array; a = array; {a.plot;}.defer;"done".postln;});
b.free;
```

fill(startAt, numFrames, value ... more)
fillMsg(startAt, numFrames, value ... more)

Starting at the index **startAt**, set the next **numFrames** to value. Additional ranges

may be included in the same message.

copy(buf, dstStartAt, srcStartAt, numSamples)
copyMsg(buf, dstStartAt, srcStartAt, numSamples)

Starting at the index **srcSamplePos**, copy **numSamples** samples from this to the destination buffer **buf** starting at **dstSamplePos**. If numSamples is negative, the maximum number of samples possible is copied. The default is start from 0 in the source and copy the maximum number possible starting at 0 in the destination.

```
s.boot;
(
SynthDef("help-Buffer-copy", { arg out=0, buf;
Line.ar(0, 0, dur:  BufDur.kr(buf), doneAction:  2); // frees itself
Out.ar(out, PlayBuf.ar(1, buf, 0.25));
}).send(s);
)

(
  b = Buffer          "sounds/a11wlk01.wav"
c = Buffer.alloc(s, 120000);
)

Synth("help-Buffer-copy", [\buf, b.bufnum]);

  // copy the whole buffer
b.copy(c);
Synth("help-Buffer-copy", [\buf, c.bufnum]);

  // copy some samples
c.zero;
b.copy(c, numSamples:  4410);
Synth("help-Buffer-copy", [\buf, c.bufnum]);

  // buffer "compositing"
c.zero;
b.copy(c, numSamples:  4410);
b.copy(c, dstStartAt:  4410, numSamples:  15500);
Synth("help-Buffer-copy", [\buf, c.bufnum]);
```

```
b.free;
c.free;
```

close(completionMessage)
closeMsg(completionMessage)

After using a Buffer with a **DiskOut** or **DiskIn**, it is necessary to close the soundfile. Failure to do so can cause problems.

completionMessage - A valid OSC message or a **Function** which will return one. A Function will be passed this Buffer as an argument when evaluated.

plot(name, bounds)

Plot the contents of the Buffer in a GUI window. OSX only.

name - The name of the resulting window.
bounds - An instance of **Rect** determining the size of the resulting view.

```
s.boot;
  b = Buffer        "sounds/a11wlk01.wav"
b.plot;
b.free;
```

play(loop, mul)

Plays the contents of the buffer on the server and returns a corresponding **Synth**.

loop - A Boolean indicating whether or not to loop playback. If false the synth will automatically be freed when done. The default is false.
mul - A value by which to scale the output. The default is 1.

```
s.boot;
  b = Buffer        "sounds/a11wlk01.wav"
  b.play; // frees itself
x = b.play(true);
x.free; b.free;
```

query

Sends a b_query message to the server, asking for a description of this buffer. The results are posted to the post window. Does not update instance vars.

updateInfo(action)

Sends a b_query message to the server, asking for a description of this buffer. Upon reply this Buffer's instance variables are automatically updated.

action - A **Function** to be evaluated once the file has been read and this Buffer's instance variables have been updated. The function will be passed this Buffer as an argument.

```
s.boot;
b = Buffer.readNoUpdate(s, "sounds/a11wlk01.wav");
  b.numFrames; // incorrect, shows nil
b.updateInfo({| buf|  buf.numFrames.postln; }); // now it's right
b.free;
```

Buffer Fill Commands

These correspond to the various b_gen OSC Commands, which fill the buffer with values for use. See **Server-Command-Reference** for more details.

gen(genCommand, genArgs, normalize, asWaveTable, clearFirst)
genMsg(genCommand, genArgs, normalize, asWaveTable, clearFirst)

This is a generalized version of the commands below.

genCommand - A **String** indicating the name of the command to use. See **Server-Command-Reference** for a list of valid command names.
genArgs - An **Array** containing the corresponding arguments to the command.
normalize - A **Boolean** indicating whether or not to normalize the buffer to a peak value of 1.0. The default is true.
asWaveTable - A **Boolean** indicating whether or not to write to the buffer in wavetable format so that it can be read by interpolating oscillators. The default is true.
clearFirst - A **Boolean** indicating whether or not to clear the buffer before writing. The default is true.

sine1(amps, normalize, asWaveTable, clearFirst)
sine1Msg(amps, normalize, asWaveTable, clearFirst)

Fill this buffer with a series of sine wave harmonics using specified amplitudes.

amps - An **Array** containing amplitudes for the harmonics. The first float value specifies the amplitude of the first partial, the second float value specifies the amplitude of the second partial, and so on.
normalize - A **Boolean** indicating whether or not to normalize the buffer to a peak value of 1.0. The default is true.
asWaveTable - A **Boolean** indicating whether or not to write to the buffer in wavetable format so that it can be read by interpolating oscillators. The default is true.
clearFirst - A **Boolean** indicating whether or not to clear the buffer before writing. The default is true.

```
s.boot;
(
b = Buffer.alloc(s, 512, 1);
b.sine1(1.0/[1,2,3,4], true, true, true);

x = SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum, 200, 0, 0.5)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
x.free; b.free;
```

sine2(freqs, amps, normalize, asWaveTable, clearFirst)
sine2Msg(freqs, amps, normalize, asWaveTable, clearFirst)

Fill this buffer with a series of sine wave partials using specified frequencies and amplitudes.

freqs - An **Array** containing frequencies (in cycles per buffer) for the partials.
amps - An **Array** containing amplitudes for the partials. This should contain the same number of items as freqs.
normalize - A **Boolean** indicating whether or not to normalize the buffer to a peak value of 1.0. The default is true.
asWaveTable - A **Boolean** indicating whether or not to write to the buffer in wavetable format so that it can be read by interpolating oscillators. The default is true.
clearFirst - A **Boolean** indicating whether or not to clear the buffer before writing.

The default is true.

```
s.boot;
(
b = Buffer.alloc(s, 512, 1);
b.sine2([1.0, 3], [1, 0.5]);

x = SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum, 440, 0, 0.5)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
x.free; b.free;
```

sine3(freqs, amps, phases, normalize, asWaveTable, clearFirst)
sine3Msg(freqs, amps, phases, normalize, asWaveTable, clearFirst)

Fill this buffer with a series of sine wave partials using specified frequencies, amplitudes, and initial phases.

freqs - An **Array** containing frequencies (in cycles per buffer) for the partials.
amps - An **Array** containing amplitudes for the partials. This should contain the same number of items as freqs.
phases - An **Array** containing initial phase for the partials (in radians). This should contain the same number of items as freqs.
normalize - A **Boolean** indicating whether or not to normalize the buffer to a peak value of 1.0. The default is true.
asWaveTable - A **Boolean** indicating whether or not to write to the buffer in wavetable format so that it can be read by interpolating oscillators. The default is true.
clearFirst - A **Boolean** indicating whether or not to clear the buffer before writing. The default is true.

cheby(amplitudes, normalize, asWaveTable, clearFirst)
chebyMsg(amplitudes, normalize, asWaveTable, clearFirst)

Fill this buffer with a series of chebyshev polynomials, which can be defined as: cheby(n) = amplitude * cos(n * acos(x)). To eliminate a DC offset when used as a waveshaper, the wavetable is offset so that the center value is zero.

amplitudes - An **Array** containing amplitudes for the harmonics. The first float value specifies the amplitude for n = 1, the second float value specifies the amplitude for n = 2, and so on.
normalize - A **Boolean** indicating whether or not to normalize the buffer to a peak value of 1.0. The default is true.
asWaveTable - A **Boolean** indicating whether or not to write to the buffer in wavetable format so that it can be read by interpolating oscillators. The default is true.
clearFirst - A **Boolean** indicating whether or not to clear the buffer before writing. The default is true.

```
s.boot;
b = Buffer.alloc(s, 512, 1, {arg buf; buf.chebyMsg([1,0,1,1,0,1])});
(
x = play({
Shaper.ar(
b.bufnum,
SinOsc.ar(300, 0, Line.kr(0,1,6)),
0.5
)
});
)
x.free; b.free;
```

ID: 370

Bundled Server Messages

When using the **Synth**/**Node**/**Group** sclang objects there is often a need to construct bundles to send messages together. For example when you want to start a synth that should be mapped instantly to certain buses, or need to ensure that two synths start with precise synchronisation.

The simplest way to deal with this is through Server's automated bundling support. This allows you to open a bundle into which all osc messages will be collected until it is sent. See **Server** for details of makeBundle's arguments.

```
s.boot;
(
 // send a synth def to server
SynthDef("tpulse", { arg out=0,freq=700,sawFreq=440.0;
Out.ar(out, SyncSaw.ar(freq,  sawFreq,0.1) )
}).send(s);
)

 // all OSC commands generated in the function contained below will be added to a bundle
 // and executed simultaneously after 2 seconds.
(
s.makeBundle(2.0, {
x = Synth.new("tpulse");
a = Bus.control.set(440);
x.busMap(\freq, a);
});
)
x.free;

 // don't send
(
b = s.makeBundle(false, {
x = { PinkNoise.ar(0.1) * In.kr(0, 1); }.play;
});
)
 // now pass b as a pre-existing bundle, and start both synths synchronously
(
```

```
 s.makeBundle(nil    // nil executes ASAP
y = { SinOsc.kr(0.2).abs }.play(x, 0, 0, \addBefore); // sine envelope
}, b);
)
x.free; y.free;
```

In addition to this there are a number of methods which return OSC messages which can be added to a bundle. These are detailed in the helpfiles for **Node**, **Synth**, and **Group**.

```
s.boot;
b = List.new;
c = Bus.control(s, 1).set(660);
 x = Synth          "default"      // Create without sending
b.add(x.newMsg);
b.add(x.busMapMsg(\freq, c));
 b.postln; // here's what it looks like
s.listSendBundle(1.0, b); // Execute after 1 second
c.set(440);
s.queryAllNodes;
x.free;
```

ID: 371

Bus

The clientside representation of an audio or control bus on a server. Encapsulates all the OSC messages a Bus can receive. Manages allocation and deallocation of bus indices so that you don't need to worry about conflicts. The number of control busses, audio busses, and input and output busses is fixed and cannot be changed after the server has been booted. For more information see **ClientVsServer** and **Server-Architecture**.

Note that using the **Bus** class to allocate a multichannel bus does not 'create' a multichannel bus, but rather simply reserves a series of adjacent bus indices with the bus' **Server** object's bus allocators. abus.index simply returns the first of those indices. When using a Bus with an **In** or **Out** ugen there is nothing to stop you from reading to or writing from a larger range, or from hardcoding to a bus that has been allocated. You are responsible for making sure that the number of channels match and that there are no conflicts.

You can control which allocator determines the bus index numbers by setting the server options **blockAllocClass** variable prior to booting the server. Two allocators are available to support different kinds of applications. See the **[ServerOptions]** help file for details.

Class Methods

Bus.control(server, numChannels);
Allocate a control bus on the server.
Defaults: default server, 1 channel.

Bus.audio(server, numChannels);
Allocate an audio bus on the server.
Defaults: default server, 1 channel.

Bus.alloc(rate, server, numChannels);
Allocate a bus of either rate as specified by the symbols:
\control or \audio

Bus.new(rate, index, numChannels);
This method does not allocate a bus index, but assumes that you already have allocated the appropriate bus index and can supply it

yourself.

Instance Methods

index -Get the Bus' index.

free - Return the bus' indices to the server's bus allocator so they can be reallocated.

rate - Get the Bus' rate. This is a symbol, either \control or \audio.

numChannels - Get the Bus' number of channels.

server - Get the Bus' server object.

asMapArg - Returns a symbol consisting of the letter 'c' followed by the bus's index. This may be used when setting a synth node's control inputs to maps the input to the control bus. See the **[Node]** help file for more information on mapping controls to buses.

Note: It is impossible to map a synth control to an audio rate bus. Calling this method on an audio bus will throw an error.

```
(
var ffreqbus = Bus.control(s, 1),
rqbus = Bus.control(s, 1);

SynthDef \rlpf    | bus, ffreq, rq|
ReplaceOut.ar(bus, RLPF.ar(In.ar(bus, 1), ffreq, rq))
}).play(s, [\ffreq, ffreqbus.asMapArg, \rq, rqbus.asMapArg]);
)
```

Asynchronous Control Bus Methods

The following commands apply only to control buses and are asynchronous. For synchronous access to control buses one should use the internal server's shared control buses and **SharedIn** / **SharedOut**.

value_(aFloat) -Set all channels to this float value. This command is asynchronous.

set(...values) - A list of values for each channel of the control bus. The list of values supplied should not be greater than the number of channels. This command is asynchronous.

setn(values) - As **set** but takes an array as an argument.

get(action) - Get the current value of this control bus. This command is asynchronous. **action** is a function that will be evaluated when the server responds, with the current value of the bus passed as an argument. For multichannel buses use getN.

getn(count, action) - Get the current values of this control bus. This command is asynchronous. **count** is the number of channels to read, starting from this bus' first channel. **action** is a function that will be evaluated when the server responds, with the current values of the bus in an array passed as an argument.

OSC Bundle Methods

getMsg - Returns a msg of the type /c_get for use in osc bundles.

getnMsg(count) - Returns a msg of the type /c_getn for use in osc bundles. **count** is the number of channels to read, starting from this bus' first channel. The default is this bus' numChannels.

setMsg(... values) - Returns a msg of the type /c_set for use in osc bundles.

setnMsg(values) - Returns a msg of the type /c_setn for use in osc bundles. **values** is a an array of values to which adjacent channels should be set, starting at this bus' first channel.

fillMsg(value) - Returns a msg of the type /c_fill for use in osc bundles. **value** is the value to which this bus' channels will be set.

```
s = Server.local;
s.boot;

(
// something to play with
SynthDef("help-Bus", { arg out=0,ffreq=100;
```

```
var x;
x = RLPF.ar(LFPulse.ar(SinOsc.kr(0.2, 0, 10, 21), [0,0.1], 0.1),
ffreq, 0.1)
.clip2(0.4);
Out.ar(out, x);
}).send(s);

)

    Synth "help-Bus"

// get a bus
b = Bus.control(s);

// map the synth's second input (ffreq) to read
// from the bus' output index
x.map(1,b.index);

// By setting the bus' value you send a /c_fill message
// to each channel of the bus setting it to supplied float value
b.value = 100;
b.value = 1000;
b.value = 30;

// Since this is a single channel bus this has the same effect
b.set(300);
b.numChannels.postln;

// multi-channel:   b.set(300,350);
// Get the current value.  This is asynchronous so you can't rely on it happening immediately.
(
a = "waiting";
b.get({arg value; a = value; ("after the server responds a is set to:" + a).postln;});
("a is now:" + a).postln;
)

x.free;

        // release it so it may be reallocated!
```

ID: 372

Short Overview of Server Commands

see also [Server-Command-Reference]

Server Commands

```
/quit
/notify  flag
/status
/cmd name args ...
/dumpOSC [0:  off 1:  on 2:  hex 3:  both]

/d_recv bytes [complBytes]
/d_load path [complBytes]
/d_loadDir path [complBytes]
/d_free defName ...
```

Node:

```
/n_free nodeID ...
/n_run | nodeID flag | ...
/n_set nodeID  | control value | ...
/n_setn nodeID | control numCtl values..  | ...
/n_fill nodeID | control numCtl value | ...
/n_map  nodeID | control busIndex | ...
/n_mapn nodeID | control busIndex numCtl | ...

/n_before | movedNodeID targetNodeID | ...
/n_after | movedNodeID targetNodeID | ...
/n_query nodeID ...
/n_trace nodeID ...

addAction:
0 add to head
1 add to tail
2 add before
3 add after
4 replace
```

alternative syntax for "nodeID"

positionArg | nodeID

"h" - head of the current group

"t" - tail of the current group

"u" - up. the parent of the current node.

"p" - the previous node.

"n" - the next node.

Synth:

/s_new defName nodeID addAction targetNodeID | control value | ...

/s_get nodeID control ...

/s_getn nodeID | control numControls | ...

/s_noid nodeID ...

Group:

/g_new nodeID addAction targetNodeID

/g_head | groupID nodeID | ...

/g_tail | groupID nodeID | ...

/g_freeAll groupID ...

/g_deepFree groupID ...

UGen:

/u_cmd nodeID ugenIndex string arg ...

Buffer:

/b_alloc bufnum numFrames numChannels [complBytes]

/b_allocRead bufnum path startFrame numFrames [complBytes]

/b_read bufnum path startFrameFile numFrames startFrameBuf numChannels leaveOpen [complBytes]

/b_write bufnum path headerFormat sampleFormat numFrames startFrameBuf leaveOpen [complBytes]

/b_free bufnum [complBytes]

/b_zero bufnum [complBytes]

/b_set bufnum | index value | ...

/b_setn bufnum | startIndex numSamples values .. | ...

/b_fill bufnum | startIndex numSamples value | ...

/b_gen bufnum command args ...

```
/b_close  bufnum
/b_query  bufnum ...   (returns /b_info message:  /b_info bufnum numFrames numChannels sampleRate)
/b_get bufnum sampleIndex ...  (returns corresponding b_set message)
/b_getn bufnum startIndex numFrames (returns corresponding b_setn message)
```

Control Bus:

```
/c_set | index value | ...
/c_setn | startIndex num values ..  | ...
/c_fill | startIndex num value | ...
/c_get  index ...  (returns corresponding c_set message)
/c_getn | startIndex num | ...  (returns corresponding c_setn message)
```

Replies:

```
/done commandName
/fail commandName errorMsg
/late timeStamp-hiBits timeStamp-loBits execTime-hiBits execTime-loBits
```

Notifications:

```
all notifications have the same format:
cmd nodeID parentNodeID prevNodeID nextNodeID synthFlag (-1:group 0 synth) headNodeID tailNodeID
/n_go /n_end /n_on /n_off /n_move /n_info
```

Trigger Notifications:

```
/tr nodeID triggerID value
```

Buffer Fill Commands:

```
flag:
1:  normalize
2:  wavetable
4:  clear and then write

sine1 flag partialAmp ...
sine2 flag | partialFreq partialAmp |
sine3 flag | partialFreq partialAmp partialPhase |
```

```
cheby flag | amp |

copy posDest bufNumSrc posSrc numFrames
```

Glossary:

```
flag:
0 (false)
1 (true)
complBytes:
an osc message to evaluate after completion (array):  this also means command is asynchronous
control:
index or name

-1 is the equivalent of nil
'nothing' is replaced by 0
```

CommandNumbers:

```
cmd_none = #0000ff0,

cmd_notify = #0000ff1,
cmd_status = #0000ff2,
cmd_quit = #0000ff3,
cmd_cmd = #0000ff4,

cmd_d_recv = #0000ff5,
cmd_d_load = #0000ff6,
cmd_d_loadDir = #0000ff7,
cmd_d_freeAll = #0000ff8,

cmd_s_new = #0000ff9,

cmd_n_trace = #0000ff10,
cmd_n_free = #0000ff11,
cmd_n_run = #0000ff12,
cmd_n_cmd = #0000ff13,
```

```
cmd_n_map = #0000ff14,
cmd_n_set = #0000ff15,
cmd_n_setn = #0000ff16,
cmd_n_fill = #0000ff17,
cmd_n_before = #0000ff18,
cmd_n_after = #0000ff19,

cmd_u_cmd = #0000ff20,

cmd_g_new = #0000ff21,
cmd_g_head = #0000ff22,
cmd_g_tail = #0000ff23,
cmd_g_freeAll = #0000ff24,

cmd_c_set = #0000ff25,
cmd_c_setn = #0000ff26,
cmd_c_fill = #0000ff27,

cmd_b_alloc = #0000ff28,
cmd_b_allocRead = #0000ff29,
cmd_b_read = #0000ff30,
cmd_b_write = #0000ff31,
cmd_b_free = #0000ff32,
cmd_b_close = #0000ff33,
cmd_b_zero = #0000ff34,
cmd_b_set = #0000ff35,
cmd_b_setn = #0000ff36,
cmd_b_fill = #0000ff37,
cmd_b_gen = #0000ff38,

cmd_dumpOSC = #0000ff39,

cmd_c_get = #0000ff40,
cmd_c_getn = #0000ff41,
cmd_b_get = #0000ff42,
cmd_b_getn = #0000ff43,
cmd_s_get = #0000ff44,
cmd_s_getn = #0000ff45,

cmd_n_query = #0000ff46,
```

```
cmd_b_query = #0000ff47,

cmd_n_mapn = #0000ff48,
cmd_s_noid = #0000ff49,

cmd_g_deepFree = #0000ff50,
cmd_clearSched = #0000ff51,

cmd_sync = #0000ff52,
cmd_d_free = #0000ff53,

NUMBER_OF_COMMANDS = #0000ff54
```

ID: 373

The default Group id:1

root node (id:0) [default group (id:1)]

When a **Server** is booted there is a top level group with an ID of 0 that defines the root of the node tree. (This is represented by a subclass of Group: **RootNode**.) If the Server was booted from within SCLang (as opposed to from the command line) a default group with an ID of 1 will be automatically created. This is the default target for all Nodes when using object style and is what you will get if you supply a Server as a target. If you don't specify a target or pass in nil, you will get the default group of the default Server.

```
Server.default.boot;
    Synth     \default   // creates a synth in the default group of the default Server
         // note the Group ID
```

The default group serves an important purpose: It provides a predictable basic Node tree so that methods such as Server-scope, Server-record, etc. can function without running into order of execution problems. In the example below the scoping node will come after the default group.

```
Server.internal.boot;

{ SinOsc.ar(mul:  0.2) }.scope(1);

// watch the post window;
Server.internal.queryAllNodes;

// our SinOsc synth is within the default group (ID 1)
// the scope node comes after the default group, so no problems

Server.internal.quit;
```

Note that the default group is persistent: It is created in Server's initTree method (executed along with any code stored in its tree instance variable; see **Server** for more

detail) and is recreated upon reboot, after pressing Cmd-., and after all nodes are freed. When using osc messages this means that a target node of id 1 can always be used:

```
s.sendMsg("/s_new", "snd", 1832,0,1); // add to head of group 1
```

The default group can be specifically freed, but there is generally no reason that one would want to do so.

In general one should add nodes to the default group rather than the RootNode unless one has a specific reason to do so (i.e. adding some new functionality such as recording and scoping which is dependent upon a predictable basic node order). When using object style the default group is the default target for all new nodes, so nodes will normally not be added to the RootNode unless that is explicitly specified. It is of course possible to do so, but it should only be done with a proper understanding of the issues involved.

For instance, if one were to place an 'effects' synth after the default group and call Server-record or Server-scope, the resulting recording or scope synths may not be correctly ordered:

default group [
source synth1
source synth2
]
recording synth
effects synth

In the above example the recording synth might not capture the output of the effects synth since it comes later in the node order.

A better method in this case is to create a group within the default group and place the effects synth after that.

default group [
source group [
source synth1
source synth2
]
effects synth
]
recording synth

See also: **RootNode, NodeMessaging** and **Order-of-execution**

ID: 374

Group client-side representation of a group node on the server

superclass: Node

A Group is the client-side representation of a group node on the server, which is a collection of other nodes organized as a linked list. The Nodes within a Group may be controlled together, and may be both Synths and other Groups. Groups are thus useful for controlling a number of nodes at once, and when used as targets can be very helpful in controlling order of execution. (See **Order-of-execution** for more details on this topic).

For more on the crucial distinction between client objects and server nodes, see **ClientVsServer**. For information on creating nodes without using objects, see **NodeMessaging**.

N.B. Group is a subclass of **Node**, and thus many of its most useful and important methods are documented in the **Node** help file. Please refer to it for more information.

RootNode and the default group

When a **Server** is booted there is a top level group with an ID of zero that defines the root of the tree. This is represented by a subclass of Group: **RootNode**. If the Server was booted from within SCLang (as opposed to from the command line) then a **default_group** with an ID of 1 will be automatically created. This group is the default enclosing group for all Nodes, i.e. it's what you get if you don't specify. In general you should create new Nodes within the default group of a Server rather than in its RootNode. See **Server**, **default_group** and **RootNode** for more detail.

Bundling

Some of the methods below have two versions: a regular one which sends its corresponding message to the server immediately, and one which returns the message in an **Array** so that it can be added to a bundle. It is also possible to capture the messages generated by the regular methods using Server's automated bundling capabilities. See **Server** and **bundledCommands** for more detail.

Creation with Immediate Instantiation on the Server

***new(target, addAction)**

Create and return a Group.

target - A target for this Group. If target is not a Group or **Synth**, it will be converted as follows: If it is a **Server**, it will be converted to the **default_group** of that server. If it is nil, to the **default_group** of the default **Server**. **Note:** A Synth is not a valid target for \addToHead and \addToTail.

addAction - one of the following Symbols:
\addToHead - (the default) add at the head of the group specified by **target**
\addToTail - add at the tail of the group specified by **target**
\addAfter - add immediately after **target** in its server's node order
\addBefore - add immediately before **target** in its server's node order
\addReplace - replace **target** and take its place in its server's node order

```
s.boot;
  g = Group      // add a Group at the head of the default Server's default group
h = Group.new(g, \addAfter);
  s.queryAllNodes; // note the Group within the default group (ID 1)
g.free; h.free;
```

The following convenience methods correspond to the add actions above:

*after(aNode)

Create and return a Group and add it immediately after **aNode**.

*before(aNode)

Create and return a Group and add it immediately before **aNode**.

*head(aGroup)

Create and return a Group. If **aGroup** is a Group add it at the head of that group. If it is a **Server**, add it at the head of the **default_group** of that server. If it is nil, add it at the head of the **default_group** of the default **Server.**

*tail(aGroup)

Create and return a Group. If **aGroup** is a Group add it at the tail of that group. If it

is a **Server**, add it at the tail of the **default_group** of that server. If it is nil, add it at the tail of the **default_group** of the default **Server.**

***replace(nodeToReplace)**

Create and return a Group and use it to replace **nodeToReplace**, taking its place in its server's node order.

Creation without Instantiation on the Server

For use in message bundles it is also possible to create a Group object in the client app without immediately creating a group node on the server. Once done one can call methods which create messages to add to a bundle, which when sent to the server will instantiate the group or perform other operations. (See Control, below.)

***basicNew(server, nodeID)**

Create and return a Group object without creating a group node on the server. (This method is inherited from **Node** and is documented here only for convenience.)

server - An optional instance of **Server**. If nil this will default to the default **Server**.

nodeID - An optional node ID number. If not supplied one will be generated by the Server's NodeIDAllocator. Normally you should not need to supply an ID.

```
s.boot;
  g = Group              // Create without sending
  s.sendBundle(nil              // Now send a message; create at the head of s' default group
s.queryAllNodes;
g.free;
```

newMsg(target, addAction)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group. If target is nil, it will default to the **default_group** of the Server specified in ***basicNew** when this Group was created. The default addAction is `\addToHead`. (See ***new** above for details of addActions.

addToHeadMsg(aGroup)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group. If **aGroup** is a Group it will be added at the head of that group. If it is nil, it will be added at the head of the **default_group** of this Group's server (as specified when ***basicNew** was called).

addToTailMsg(target)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group. If **aGroup** is a Group it will be added at the tail of that group. If it is nil, it will be added at the tail of the **default_group** of this Group's server (as specified when ***basicNew** was called).

addBeforeMsg(aNode)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group, immediately before **aNode**.

addAfterMsg(aNode)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group, immediately after **aNode**.

addReplaceMsg(nodeToReplace)

Returns a message of the type **g_new** which can be bundled. When sent to the server this message will instantiate this group, replacing **nodeToReplace** in the server's node order.

Control

For further methods of controlling Groups (set, map, busMap, etc.), see the **Node** help file.

moveNodeToHead(aNode)
moveNodeToHeadMsg(aNode)

Move **aNode** to the head of this group

moveNodeToTail(aNode)

moveNodeToTailMsg(aNode)

Move **aNode** to the tail of this group

freeAll
freeAllMsg

Free all the nodes in this group, but do not free this group itself.

deepFree
deepFreeMsg

Free all Synths in the group, and all Synths in any enclosed groups, but do not free this group or any of its enclosed groups.

Examples

```
(
    Server          // just to be sure
s.boot;
)

(
SynthDef("help-Group-moto-rev", { arg out=0,freq=100,ffreq=120;
var x;
x = RLPF.ar(LFPulse.ar(SinOsc.kr(0.2, 0, 10, freq), [0,0.1], 0.1),
ffreq, 0.1).clip2(0.4);
Out.ar(out, x);
}).send(s);


SynthDef("help-Group-wah", { arg out, rate = 1.5, cfreq = 1400, mfreq = 1200, rq=0.1;
var zin, zout, q;

zin = In.ar(out, 2);
cfreq = Lag3.kr(cfreq, 0.1);
mfreq = Lag3.kr(mfreq, 0.1);
q   = Ramp.kr(rq, 0.1);
zout = RLPF.ar(zin, LFNoise1.kr(rate, mfreq, cfreq), q, 10).distort
```

```
* 0.15;

 // replace the incoming bus with the effected version
ReplaceOut.ar( out , zout );

}).send(s);
)

g = Group.new;

(
l = Array.fill(3,{
 // random freq for each synth, added to g at the head
 Synth "help-Group-moto-rev"  "out"   "freq"                  \addToHead
});
)

// set all controls that match "ffreq" in all nodes in g to 90
g.set("ffreq",300);

g.set("freq",80);

// since we stored the Synths in an Array, we can also control them individually
(
    Routine
inf.do({
l.do({ arg node;
node.set("freq",rrand(10,120));
1.0.wait;
});
})
});

r.play;
)

// g is in a group too.  Since we didn't specify it's the default group (ID 1) of the default Server
g.group.inspect;

// asking a wah to go order-of-execution after g, in the same group as g.
```

```
x = Synth.after(g,"help-Group-wah",["out",0]);

x.free;

// free all nodes in g, but not g itself
g.freeAll;

// don't forget the Routine is still running...
r.stop;

// oh, and set l to nil so the Synths and Array can be garbage collected
l = nil;

// and i'm still on the server, its just my children that were freed
g.query;

// don't need the individual synth objects this time
(
3.do({
 // random freq for each synth, added to g at the head
 Synth "help-Group-moto-rev"  "out"   "freq"                    \addToHead
});
)

// kill me and my children
g.free;

// see, I'm gone
g.query;
```

ID: 375

Node abstract superclass of Synth and Group

superclass: Object

See **Server-Architecture** for the definition of a node.

This class is the abstract super class of **Synth** and **Group**, which represent synth and group nodes on the server. Node objects are not made explicitly, but Synth and Group are subclasses, and inherit the methods documented below.

Freed Nodes and Node Status

Nodes which you explicitly free using the methods free or release will have their group instance variable set to nil. However Nodes which are automatically freed after a certain time (for instance by an **EnvGen** with a doneAction of 2) will not. This keeps the implementation of the classes simple and lightweight. To have the current state of a Node tracked you can register it with an instance of **NodeWatcher**. This will enable two variables, isPlaying and isRunning, which you can use for checking purposes.

Bundling

Many of the methods below have two versions: a regular one which sends its corresponding message to the server immediately, and one which returns the message in an Array so that it can be added to a bundle. It is also possible to capture the messages generated by the regular methods using Server's automated bundling capabilities. See **Server** and **bundledCommands** for more details.

Accessing Instance Variables

The following getter methods also have corresponding setters, but they should be used with extreme care and only if you are *sure* you know what you're doing.

nodeID - Returns the Node's node ID number.

group - Returns an instance of **Group** or **RootNode** corresponding to this Node's group on the server.

server - Returns an instance of **Server** corresponding to this Node's server app.

isPlaying - Returns a boolean indicating if this node is currently on the server, providing this Node has been registered with a **NodeWatcher**. **N.B.** If this Node has not been registered this will likely be false in any case.

isRunning - Returns a boolean indicating if this node is currently on the server, providing this Node has been registered with a **NodeWatcher**. **N.B.** If this Node has not been registered this will likely be false in any case.

Node Commands

See the Node Commands section in **Server-Command-Reference** for the OSC equivalents of the methods outlined below**.**

free(sendFlag)
freeMsg

Stop this Node and free it from it's parent group on the server.Once a Node has been freed, you cannot restart it. **sendFlag** is a boolean indicating whether the free message should be sent. If false an **n_free** message will not be sent to this Node's server, but its isPlaying and isRunning variables will be set to false. The default for **sendFlag** is true. If this Node is a **Group** this will free all Nodes within the Group.

```
s.boot;
x = Synth("default");
x.free;
```

run(boolean)
runMsg(boolean)

Set the running state of this Node according to a boolean. False will pause the node without freeing it. The default is true. If this Node is a **Group** this will set the running state of all Nodes within the Group.

```
s.boot;
(
x = SynthDef("help-node-set", {arg freq = 440, out = 0;
Out.ar(out, SinOsc.ar(freq, 0, 0.1));}).play(s);
```

```
)
x.run(false);
  x.run; // default is true
x.free;
```

set(controlName, value ... moreArgs)
setMsg(controlName, value ... moreArgs)

Set controls in this Node to values. Controls are defined in a **SynthDef** as args or instances of **Control**. They are specified here using symbols, strings, or indices, and are listed in pairs with values. If this Node is a **Group** this will set all Nodes within the Group.

```
s.boot;
(
x = SynthDef("help-node-set", {arg freq = 440, out = 0;
Out.ar(out, SinOsc.ar(freq, 0, 0.1));}).play(s);
)
x.set(\freq, 880, \out, 1); // two pairs
  x.set(0, 660, 1, 0); // freq is the first argument, so it's index is 0.  out is index 1.
x.free;
```

setn(controlName, values ... moreArgs)
setnMsg(controlName, values ... moreArgs)

Set sequential ranges of controls in this Node to values. Controls are defined in a **SynthDef** as args or instances of **Control**. They are specified here using symbols, strings, or indices, and are listed in pairs with arrays of values. If this Node is a **Group** this will setn all Nodes within the Group.

```
s.boot;
(
  x = SynthDef "help-node-setn"
arg freq1 = 440, freq2 = 440, freq3 = 440, amp1 = 0.05, amp2 = 0.05, amp3 = 0.05;
Out.ar(0, Mix(SinOsc.ar([freq1, freq2, freq3], 0, [amp1, amp2, amp3])));}).play(s);
)
  // set 3 controls starting from \freq1, and 3 controls starting from \amp1
x.setn(\freq1, [440, 880, 441], \amp1, [0.3, 0.1, 0.3]);
x.free;
```

fill(controlName, numControls, value ... moreArgs)
fillMsg(controlName, numControls, value ... moreArgs)

Set sequential ranges of controls in this Node to a single value. Controls are defined in a **SynthDef** as args or instances of **Control**. They are specified here using symbols, strings, or indices, and are listed in groups of three along with an integer indicating the number of controls to set, and the value to set them to. If this Node is a **Group** this will fill all Nodes within the Group.

busMap(firstControl, aBus ... moreArgs)
busMapMsg(firstControl, aBus ... moreArgs)

Map sequential ranges of controls in this Node to read from control rate Buses. Controls are defined in a **SynthDef** as args or instances of **Control**. They are specified here using symbols, strings, or indices, and are listed in pairs with Bus objects. The number of sequential controls mapped corresponds to the Bus' number of channels. If this Node is a **Group** this will busMap all Nodes within the Group.

```
s.boot;
(
b = Bus.control(s, 2); b.set(440, 660); // a two channel bus
c = Bus.control(s, 1); c.set(441);  // a one channel bus
x = SynthDef("help-Node-busMap", { arg freq1 = 440, freq2 = 440, freq3 = 440;
Out.ar(0, Mix(SinOsc.ar([freq1, freq2, freq3], 0, 0.1)));
}).play(s);)

  // b has two channels, so both freq2 and freq3 are mapped to its first and second channels
  // respectively; c has one channel, so it maps only freq1
x.busMap(\freq1, c, \freq2, b);
b.set(440, 880);
c.set(1200);
x.free; b.free; c.free;
```

map(controlName, index ... moreArgs)
mapMsg(controlName, index ... moreArgs)

Map controls in this Node to read from control rate Buses. Controls are defined in a **SynthDef** as args or instances of **Control**. They are specified here using symbols, strings, or indices, and are listed in pairs with bus indices. If this Node is a **Group** this will map all Nodes within the Group.

```
s.boot;
(
b = Bus.control(s, 1); b.set(880);
c = Bus.control(s, 1); c.set(884);
x = SynthDef("help-Node-busMap", { arg freq1 = 440, freq2 = 440;
Out.ar(0, SinOsc.ar([freq1, freq2], 0, 0.1));
}).play(s);)
x.map(\freq1, b.index, \freq2, c.index);
x.free; b.free; c.free;
```

mapn(controlName, index, numControls ... moreArgs)
mapnMsg(controlName, index, numControls ... moreArgs)

Map sequential ranges of controls in this Node to read from control rate Buses. This is similar to busMap above, but instead of passing in **Bus** objects you specify the index, and the number of sequential Controls to map. If this Node is a **Group** this will mapn all Nodes within the Group.

release(releaseTime)
releaseMsg(releaseTime)

This is a convenience method which assumes that the synth contains an envelope generator (an **EnvGen**, **Linen**, or similar **UGen**) running a sustaining envelope (see **Env**) and that it's gate argument is set to a control called \gate. This method will cause the envelope to release. If **releaseTime** is not nil, it will override the envelope's decay or release time. If this Node is a **Group** this will release all Nodes within the Group.

```
x = { arg gate=1; BrownNoise.ar(0.5) * EnvGen.kr(Env.cutoff(1), gate, doneAction:2) }.play;
  x.release(5); // overide the Env's specified 1 second release time
```

query

Sends an **n_query** message to the server, which will reply with a message containing information about this node and its place in the server's node tree. This information will be printed to the post window. (See also the queryAllNodes method of **Server**.) "parent" indicates the Node's enclosing group. If "prev" or "next" are equal to -1 that indicates that there are no other nodes in the enclosing group before or after this one, respectively.

```
g = Group.new;
x = Synth.head(g, "default");
x.query;
g.query;
  s.queryAllNodes; // Note the RootNode (ID 0) and the default Group (ID 1)
x.free; g.free;
```

trace

Causes a synth to print out the values of the inputs and outputs of its unit generators for one control period to the post window. Causes a group to print the node IDs and names of each node in the group for one control period.

```
g = Group.new;
x = Synth.head(g, "default");
x.trace;
g.trace;
x.free; g.free;
```

Changing the order of execution

The following methods can be used to change the Node's place in the order of execution. See the **Order-of-execution** help file for more information on this important topic. See **Server-Command-Reference** for the OSC equivalents of these methods.

moveAfter(aNode)
moveAfterMsg(aNode)

Move this Node to be directly after aNode. **N.B. n_after**, the OSC message which this method encapsulates, allows already freed nodes as targets. This is so that one may attempt a series of moves, with the last successful one taking effect. For this reason this method will fail silently if either the target or this node have already been freed. If you will need to check, you may register the relevant nodes with a **NodeWatcher**.

moveBefore(aNode)
moveBeforeMsg(aNode)

Move this Node to be directly before aNode. **N.B. n_before**, the OSC message which

this method encapsulates, allows already freed nodes as targets. This is so that one may attempt a series of moves, with the last successful one taking effect. For this reason this method will fail silently if either the target or this node have already been freed. If you will need to check, you may register the relevant nodes with a **NodeWatcher**.

moveToHead(aGroup)
moveToHeadMsg(aGroup)

If **aGroup** is a **Group** then this method will move this Node to the head of that Group. If it is nil this will move this Node to the head of the **default_group** of this Node's **Server**.

moveToTail(aGroup)
moveToTailMsg(aGroup)

If **aGroup** is a **Group** then this method will move this Node to the tail of that Group. If it is nil this will move this Node to the tail of the **default_group** of this Node's **Server**.

Other Methods

asTarget - Returns this Node. See the **asTarget** help file for more details.

printOn(stream) - Prints this Node's Class (Synth or Group) and nodeID on **stream**.

hash - Returns server.hash bitXor: nodeID.hash

== aNode - Returns true if this Node and aNode have the same nodeID and the same Server object, otherwise returns false. Under certain circumstances this Node and aNode might not be the same object, even though this returns true.

```
Group                  // the default group of s
h = Group.basicNew(s, 1); // and again
g == h;  // true
g === h; // false
```

ID: 376

NodeControl

encapsulates in an object a node and an index. this object can be held by a client and have its value set without otherwise having to store the details about where the node's input is.

```
d = SynthDef("help-NodeControl",{ arg out=0,freq=400;
Out.ar(out,
SinOsc.ar(freq, 0, 0.5)
)
});
y = d.play; // the synth

c = NodeControl(y,1);

c.value = 500;

c.value = 300;
```

ID: 377

Node Messaging

The most direct and fast way to send commands to the server is to send messages to the Server object, if you are within sc-lang. If you are in a shell you can use **sendOSC** (available from CNMAT).

this messaging scheme is explained in detail in

Server-Architecture
Server-Command-Reference
Tutorial

When creating nodes on the server (synths and groups) the only things we need to know are the nodeID and the server (its address to be precise).

In order to communicate with a synth, one sends messages with its nodeID. If you do not intend to communicate with the node after its creation (and the node will cause itself to end without external messaging), the node id can be set to -1, which is the server's equivalent to nil.

As soon as you want to pass around the reference to a certain node, assuming that you might not have only one server, it can be useful to create a Synth or Group object. These objects also respond to messages, and when needed can be used to obtain the state of the server side node.

see **Node**, **Synth**, and **Group** help for more detailed helpfiles on node objects.

```
// the equivalent of
n = s.nextNodeID;
s.sendMsg("/s_new", "default", n);
s.sendMsg("/n_free", n);

// is
 n = Synth "default"
n.free;
```

```
// when passing arguments:
n = s.nextNodeID;
s.sendMsg("/s_new", "default", n, 0, 0, \freq, 850);
s.sendMsg("/n_set", n, \freq, 500);
s.sendMsg("/n_free", n);

// it is
n = Synth("default", [\freq, 850]);
n.set(\freq, 500)
n.free;
```

The answer to the question of whether one should work with **node objects** or **directly with messages** depends to some extent on context, and to some extent is a matter of personal taste.

The encapsulation of node objects results in a certain generalization, meaning that other compound objects can respond to the same messages and thus exploit polymorphism. They also provide a certain level of convienence, keeping track of indexes and IDs, etc.

In certain cases, such as for granular synthesis it is recommended to use messages directly, because there is no benefit to be gained from the node objects (i.e. no need to message them) and they add cpu load to the client side.

```
(
 SynthDef "grain"
Out.ar(0, Line.kr(0.1, 0, 0.01, doneAction:2) * FSinOsc.ar(12000))
}).send(s);
)

(
 Routine
20.do({
s.sendMsg("/s_new", "grain", -1);
0.01.wait;
})
}).play;
)
```

In cases where you need to keep track of the synth's state, it is advisable to use node objects and register them with a **NodeWatcher**. (see helpfile)

Apart from such cases it is a matter of taste whether you want to use the combination of message and a numerical global representation or an object representation. The two can be mixed, and certain advantages of the object style can be accessed when using messaging style. For instance Server.nextNodeID allows one to use dynamically assigned IDs in messaging style. As a gross generalization, it is probably fair to say that object style is more convienent, but messaging style is more efficient, due to reduce client-side CPU load.

IMPORTANT NOTE: If you wish to have the functionality of the **default_group** (e.g. problem free use of Server's record and scope functionality) you should treat ID 1 (the default_group) as the root of your node tree rather than ID 0 (the **RootNode**). See **default_group** for more details.

Note that **Function-play** and **SynthDef-play** return a synth object that can be used to send messages to.

```
x = { arg freq=1000; Ringz.ar(Crackle.ar(1.95, 0.1), freq, 0.05) }.play(s);
x.set(\freq, 1500);
x.free;
```

ID: 378

Order of execution

Order of execution is one of the most critical and seemingly difficult aspects of using SuperCollider, but in reality it only takes a little thought in the early planning stages to make it work for you.

Order of execution in this context doesn't mean the order in which statements are executed in the language (the client). It refers to the ordering of synth nodes on the server, which corresponds to the order in which their output is calculated each control cycle (blockSize). Whether or not you specify the order of execution, each synth and each group goes into a specific place in the chain of execution.

If you have on the server:

synth 1 —> synth 2

... all the unit generators associated with synth 1 will execute before those in synth 2 during each control cycle.

If you don't have any synths that use In.ar, you don't have to worry about order of execution. It only matters when one synth is reading the output of another.

The rule is simple: if you have a synth on the server (i.e. an "effect") that depends on the output from another synth (the "source"), the effect must appear later in the chain of nodes on the server than the source.

source —> effect

If you have:

effect —> source

The effect synth will not hear the source synth, and you won't get the results you want.

Some Notes about Servers and Targets

There is always a default **Server**, which can be accessed or set through the class method Server.default. At startup this is set to be the local Server, and is also assigned to the

interpreter variable s.

```
// execute the following and watch the post window
s === Server.default;
s === Server.local;
Server.default = Server.internal; s === Server.default;
Server           Server        // return it to the local server
```

When a **Server** is booted there is a top level group with an ID of 0 that defines the root of the node tree. This is represented by a subclass of Group: **RootNode**. There is also a **default_group** with an ID of 1. This group is the default group for all Nodes. This is what you will get if you supply a Server as a target. If you don't specify a target or pass in nil, you will get the default group of the default Server.

The default group serves an important purpose: It provides a predictable basic Node tree so that methods such as Server-scope and Server-record can function without running into order of execution problems. Thus in general one should create new Nodes within the default group rather than in the RootNode. See **default_group** and **RootNode** for more detail.

Controlling order of execution

There are three ways to control the order of execution: using addAction in your synth creation messages, moving nodes, and placing your synths in groups. Using groups is optional, but they are the most effective in helping you organize the order of execution.

Add actions:

By specifying an addAction argument for **Synth.new** (or SynthDef.play, Function.play, etc.) one can specify the node's placement relative to a target. The target might be a group node, another synth node, or a server.

As noted above, the default target is the **default_group** (the group with **nodeID 1**) of the default Server.

The following **Symbols** are valid addActions for Synth.new: \addToHead, \addToTail, \addBefore, \addAfter, \addReplace.

Synth.new(defName, args, target, addAction)

if target is a Synth the \addToHead, and \addToTail methods will apply to that Synths group
if target is a Server it will resolve to that Server's default group
if target is nil it will resolve to the default group of the default Server

For each addAction there is also a corresponding convenience method of class **Synth**:

Synth.head(aGroup, defName, args)
add the new synth to the the head of the group specified by aGroup
if aGroup is a synth node, the new synth will be added to the head of that node's group
if target is a Server it will resolve to that Server's default group
if target is nil it will resolve to the default group of the default Server

Synth.tail(aGroup, defName, args)
add the new synth to the the tail of the group specified by aGroup
if aGroup is a synth node, the new synth will be added to the tail of that node's group
if target is a Server it will resolve to that Server's default group
if target is nil it will resolve to the default group of the default Server

Synth.before(aNode, defName, args)
add the new node just before the node specified by aNode.

Synth.after(aNode, defName, args)
add the new node just after the node specified by aNode.

Synth.replace(synthToReplace, defName, args)
the new node replaces the node specified by synthToReplace. The target node is freed.

Using Synth.new without an addAction will result in the default addAction. (You can check the default values for the arguments of any method by looking at a class' source code. See **Internal-Snooping** for more details.) Where order of execution matters, it is important that you specify an addAction, or use one of the convenience methods shown above.

Moving nodes:

.moveBefore
.moveAfter
.moveToHead

.moveToTail

If you need to change the order of execution after synths and groups have been created, you can do this using move messages.

```
fx = Synth.tail(s, "fx");
  src = Synth          "src"    // effect will not be heard b/c it's earlier
  src.moveBefore( fx);   // place the source before the effect
```

Groups

Groups can be moved in the same way as synths. When you move a group, all the synths in that group move with it. This is why groups are such an important tool for managing order of execution. (See the **Group** helpfile for details on this and other convenient aspects of Groups.)

Group 1 —> Group 2

In the above configuration, all of the synths in group 1 will execute before all of the synths in group 2. This is an easy, easy way to make the order of execution happen the way you want it to.

Determine your architecture, then make groups to support the architecture.

Using order of execution to your advantage

Before you start coding, plan out what you want and decide where the synths need to go.

A common configuration is to have a routine playing nodes, all of which need to be processed by a single effect. Plus, you want this effect to be separate from other things running at the same time. To be sure, you should place the synth -> effect chain on a private audio bus, then transfer it to the main output.

[Lots of synths] —-> effect —-> transfer

This is a perfect place to use a group:

Group ([lots of synths]) —-> effect —-> transfer

To make the structure clearer in the code, one can also make a group for the effect (even if there's only one synth in it):

Group ([lots of synths]) —-> Group ([effect]) —-> transfer

I'm going to throw a further wrench into the example by modulating a parameter (note length) using a control rate synth.

So, at the beginning of your program:

```
s.boot;

(
    Bus                 // get a bus for the LFO--not relevant to order-of-exec
    Bus              // assuming stereo--this is to keep the src->fx chain separate from
    // other similar chains
synthgroup = Group.tail(s);
fxgroup = Group.tail(s);

// now you have synthgroup --> fxgroup within the default group of s

// make some synthdefs to play with
SynthDef("order-of-ex-dist", { arg bus, preGain, postGain;
var sig;
sig = In.ar(bus, 2);
sig = (sig * preGain).distort;
ReplaceOut.ar(bus, sig * postGain);
}).send(s);

SynthDef("order-of-ex-pulse", { arg freq, bus, ffreq, pan, lfobus;
var sig, noteLen;
noteLen = In.kr(lfobus, 1);
sig = RLPF.ar(Pulse.ar(freq, 0.2, 0.5), ffreq, 0.3);
Out.ar(bus, Pan2.ar(sig, pan)
* EnvGen.kr(Env.perc(0.1, 1), timeScale:  noteLen, doneAction:  2));
}).send(s);

SynthDef("LFNoise1", { arg freq, mul, add, bus;
```

```
Out.kr(bus, LFNoise1.kr(freq, mul:mul, add:add));
}).send(s);
)

// Place LFO:

lfo = Synth.head(s, "LFNoise1", [\freq, 0.3, \mul, 0.68, \add, 0.7, \bus, l.index]);

// Then place your effect:

dist = Synth.tail( fxgroup, "order-of-ex-dist", [\bus, b.index, \preGain, 8, \postGain, 0.6]);

// transfer the results to main out, with level scaling
// play at tail of s's default group (note that Function-play also takes addActions!

xfer = { Out.ar(0, 0.25 * In.ar(b.index, 2)) }.play(s, addAction:  \addToTail);

// And start your routine:

(
    Routine
{
Synth.tail( synthgroup, "order-of-ex-pulse",
[\freq, rrand(200, 800), \ffreq, rrand(1000, 15000), \pan, 1.0.rand2,
\bus, b.index, \lfobus, l.index]);
0.07.wait;
}.loop;
        SystemClock
)

          false   // proves that the distortion effect is doing something
dist.run(true);

// to clean up:
(
r.stop;
[ synthgroup,  fxgroup, b, l,  lfo,  xfer].do({ arg x; x.free });
                          // clear all environment variables
)
```

Note that in the routine, using a Group for the source synths allows their order to easily be specified relative to each other (they are added with the .tail method), without worrying about their order relative to the effect synth.

Note that this arrangement prevents errors in order of execution, through the use of a small amount of organizational code. Although straightforward here, this arrangement could easily be scaled to a larger project.

Messaging Style

The above examples are in 'object style'. Should you prefer to work in 'messaging style' there are corresponding messages to all of the methods shown above. See **NodeMessaging**, and **Server-Command-Reference** for more details.

Feedback

When the various output ugens (**Out**, **OffsetOut**, **XOut**) write data to a bus, they *mix* it with any data from the current cycle, but *overwrite* any data from the previous cycle. (**ReplaceOut** overwrites all data regardless.) Thus depending on node order, the data on a given bus may be from the current cycle or be one cycle old. **In.ar** checks the timestamp of any data it reads in and zeros any data from the previous cycle (for use within that synth; the data remains on the bus). This is fine for audio data, as it avoids feedback, but for control data it is useful to be able to read data from any place in the node order. For this reason **In.kr** also reads data that is older than the current cycle.

In some cases we might also want to read audio from a node later in the current node order. This is the purpose of **InFeedback**. The delay introduced by this is at maximum one block size, which equals about 0.0014 sec at the default block size and sample rate.

The variably mixing and overwriting behaviour of the output ugens can make order of execution crucial when using **In.kr** or **InFeedback.ar**. (No pun intended.) For example with a node order like the following the InFeedback ugen in Synth 2 will only receive data from Synth 1 (-> = write out; <- = read in):

Synth 1 -> busA this synth overwrites the output of Synth3 before it reaches Synth 2

Synth 2 (with InFeedback) <- busA
Synth 3 -> busA

If Synth 1 were moved after Synth 2 then Synth 2's InFeedback would receive a mix of the output from Synth 1 and Synth 3. This would also be true if Synth 2 came after Synth1 and Synth 3. In both cases data from Synth 1 and Synth 3 would have the same time stamp (either current or from the previous cycle), so nothing would be overwritten.

(As well, if any In.ar wrote to busA earlier in the node order than Synth 2, it would zero the bus before Synth 3's data reached Synth 2. This is true even it there were no node before Synth 2 writing to busA.)

Because of this it is often useful to allocate a separate bus for feedback. With the following arrangement Synth 2 will receive data from Synth3 regardless of Synth 1's position in the node order.

Synth 1 -> busA
Synth 2 (with InFeedback) <- busB
Synth 3 -> busB + busA

The following example demonstrates this issue with In.kr:

```
(
SynthDef "help-Infreq"    arg
Out.ar(0, FSinOsc.ar(In.kr(bus), 0, 0.5));
}).send(s);

SynthDef("help-Outfreq", { arg freq = 400, bus;
Out.kr(bus, SinOsc.kr(1, 0, freq/40, freq));
}).send(s);

b = Bus.control(s,1);
)

// add the first control Synth at the tail of the default server; no audio yet
x = Synth.tail(s, "help-Outfreq", [\bus, b.index]);

// add the sound producing Synth BEFORE it; It receives x's data from the previous cycle
y = Synth.before(x, "help-Infreq", [\bus, b.index]);
```

```
// add another control Synth before y, at the head of the server
// It now overwrites x's cycle old data before y receives it
z = Synth.head(s, "help-Outfreq", [\bus, b.index, \freq, 800]);

// get another bus
c = Bus.control(s, 1);

// now y receives x's data even though z is still there
y.set(\bus, c.index); x.set(\bus, c.index);

x.free; y.free; z.free;
```

Helpful 'Third Party' Classes

James Harkins' MixerChannel class can help you with order of execution. Each channel includes a synth group and an effect group, as well as level and panning controls and pre- and post-fader sends. Using this class, it helps if you understand audio buses, but it does include methods .play and .playfx to place synths in the correct group and assign the bus automatically. You can download this class from http://www.duke.edu/ jharkins/sc3

If you run into trouble, the crucial library includes a utility to query all nodes on the server so that you can see the order in which they're executing. Type **Crucial.menu** to access this command.

ID: 379

RootNode

superclass: Group

A RootNode is the Group with the **nodeID of 0** which is always present on each Server and represents the root of that server's node tree.

It is always playing, and always running, cannot be freed, or moved anywhere.

Cacheing is used so that there is always one RootNode per Server.

```
s = Server.local;


a = RootNode(s);
b = RootNode(s);

        // identical object
```

sending "/s_new" messages to the server, the target 0 is what is represented by this object.

```
        "/s_new"  "default"           //the last argument is the target id
```

IMPORTANT: In general one should NOT add nodes to the RootNode unless one has a specific reason to do so. Instead one should add nodes to the **default_group**. This provides a known basic node order and protects functionality like Server.record, Server.scope, etc. The default group is the default target for all new nodes, so when using object style nodes will normally not be added to the RootNode unless that is explicitly specified. See **default_group** for more information.

ID: 380

Server object representing an sc-server application

superclass: Model

A Server object is the client-side representation of a server app and is used to control the app from the SuperCollider language application. (See **[ClientVsServer]** for more details on the distinction.) It forwards osc-messages and has a number of allocators that keep track of IDs for nodes, buses and buffers. The server application is a commandline program, so all commands apart from osc-messages are unix commands. The server application represented by a Server object might be running on the same machine as the client (in the same address space as the language application or separately; see below), or it may be running on a remote machine.

Most of a Server's options are contolled through its instance of **ServerOptions**. See the **[ServerOptions]** helpfile for more detail.

Paths

Server apps running on the local machine have two unix environment variables: SC_SYNTHDEF_PATH and SC_PLUGIN_PATH. These indicate directories of synthdefs and ugen plugins that will be loaded at startup. These are in addition to the default synthdef/ and plugin/ directories which are hard-coded. These can be set within SC using the getenv and setenv methods of class **[String]**.

```
// all defs in this directory will be loaded when a local server boots
"SC_SYNTHDEF_PATH"        " /scwork/"
"echo $SC_SYNTHDEF_PATH"
```

The default group

When a Server is booted there is a top level group with an ID of 0 that defines the root of the node tree. (This is represented by a subclass of Group: **RootNode**.) If the server app was booted from within SCLang (as opposed to from the command line) the

method initTree will be called automatically after booting. This will also create a **default_group** with an ID of 1, which is the default group for all Nodes when using object style. This provides a predictable basic node tree so that methods such as Server-scope, Server-record, etc. can function without running into order of execution problems. The default group is persistent, i.e. it is recreated after a reboot, pressing cmd-., etc. See **[RootNode]** and **[default_group]** for more information. Note that if a Server has been booted from the command line you must call initTree manually in order to initialize the default group, if you want it. See **initTree** below.

Local vs. Internal

In general, when working with a single machine one will probably be using one of two Server objects which are created at startup and stored in the class variables **local** and **internal**. By default two GUI windows are created to control these. The difference between the two is that the local server runs as a separate application with its own address space, and the internal server runs within the same space as the language/client app. The internal server has the advantage of being able to access shared memory, thus allowing for things like scope windows (see below) and **[SharedIn]**/[**SharedOut**]. It also minimizes messaging latency. The local server, and any other server apps running on your local machine, have the advantage that if the language app crashes, it (and thus possibly your piece) will continue to run. It is thus an inherently more robust arrangement.

The default Server

There is always a default Server, which is stored in the class variable **default**. Any Synths or Groups created without a target will be created on the default server. At startup this is set to be the **local** server (see above), but can be set to be any Server.

Class Methods

***new(name, addr, options, clientID)**

name - a symbol; each Server object is stored in one global classvariable under its name.
addr - an optional instance of **[NetAddr]**, providing host and port. The default is the localhost address using port 57110; the same as the local server.
options - an optional instance of **ServerOptions**. If nil, an instance of ServerOptions will be created, using the default values.
clientID - an integer. In multi client situations, every client can be given a separate nodeID range.
The default is 0.

***local** - returns the local server, stored in classvar local (created already on initClass)

***internal** - returns the internal server, stored in classvar local (created already on initClass)

***default** - returns the default server. By default this is the local server (see above)

***default(aServer)** - sets the default Server to be aServer

```
Server          Server          // set the internal Server to be the default Server
```

***quitAll** - quit all registered servers

***killAll** - query the system for any sc-server apps and hard quit them

***freeAll** - free all nodes in all registered servers

Instance Methods

sendMsg(arg1, arg2, arg3, ... argN) - send an osc message to the server.

```
s.sendMsg("/s_new", "default", s.nextNodeID, 0, 1);
```

sendBundle(time, array1, array1, array1, ... arrayN) - send an osc bundle to the server. Since the network may have irregular performance, **time** allows for the bundle to be evaluated at a specified point in the future. Thus all messages are synchronous relative to each other, but delayed by a constant offset. If such a bundle arrives late, the server replies with a late message but still evaluates it.

```
s.sendBundle(0.2, ["/s_new", "default", x = s.nextNodeID, 0, 1], ["/n_set", x, "freq", 500]);
```

sendRaw(aRawArray)

listSendMsg([arg1, arg2, arg3, ... argN]) - as sendMsg, but takes an array as argument.

listSendBundle(time, [array1, array1, array1, ... arrayN]) - as sendBundle, but takes an array as argument. This allows you to collect messages in an array and then send them.

```
s.listSendBundle(0.2, [["/s_new", "default", x = s.nextNodeID, 0, 1],
["/n_set", x, "freq", 600]]);
```

sendSynthDef(name, dir) - send a synthDef to the server that was written in a local directory

loadSynthDef(name, completionMsg, dir) - load a synthDef that resides in the remote directory

loadDirectory(dir, completionMsg) - load all the SynthDefs in the directory **dir**. **dir** is a String which is a valid path.

nextNodeID - get a unique nodeID.

wait(responseName) - this can be used within a Routine to wait for a server reply

waitForBoot(func, limit) - evaluate the function **func** as soon as the server has booted. If it is running, it is evaluated immediately. If it is not running, boot the server and evaluate the function. **limit** indicates the maximum times to try. (5 times/sec)

doWhenBooted(func, limit) - evaluate the function as soon as the server has booted. If it is running, it is evaluated immediately. **limit** is the maximum number of times to try. (5 times/sec)

boot(startAliveThread) - boot the remote server, create new allocators. **startAliveThread**: if set to false, the server is not queried to give information for the window. **N.B.** You cannot locally boot a server app on a remote machine.

quit - quit the server application

reboot - quit and restart the server application

freeAll - free all nodes in this server

status - query the server status

notify(flag) - server sends notifications, for example if a node was created, a 'tr' message from a SendTrig, or a /done action. if flag is set to false, these messages are not sent. The default is true.

dumpOSC(code)
code:
0 - turn dumping OFF.
1 - print the parsed contents of the message.
2 - print the contents in hexadecimal.
3 - print both the parsed and hexadecimal representations of the contents.

queryAllNodes - Post a representation of this Server's current node tree to the post window. Very helpful for debugging.

```
s.boot;
 s.queryAllNodes; // note the root node (ID 0) and the default group (ID 1)
s.quit;
```

ping(numberOfTimes, waitBewteen, completionFunction)
measure the time between server and client, which may vary. the completionFunction is evaluated after numberOfTimes and is passed the resulting maximum.

options - returns this Server's **[ServerOptions]** object. Changes take effect when the server is rebooted.

options_(aServerOptions) - sets this Server's **[ServerOptions]** object. Changes take effect when the server is rebooted.

defaultGroup - returns this Server's default group.

Automatic Message Bundling

Server provides support for automatically bundling messages. This is quite convenient in object style, and ensures synchronous execution. See also **bundledCommands**.

makeBundle(time, func, bundle) - The Function **func** is evaluated, and all OSC messages generated by it are deferred and added to a bundle. This method returns the bundle so that it can be further used if needed. If **time** is set to nil or a number the bundle will be automatically sent and executed after the corresponding delay in seconds. If **time** is set to false the bundle will not be sent. **bundle** allows you to pass in a preexisting bundle and continue adding to it. If an error is encountered while evaluating **func** this method will throw an **Error** and stop message deferral.

```
s.boot;
(
 // send a synth def to server
SynthDef("tpulse", { arg out=0,freq=700,sawFreq=440.0;
Out.ar(out, SyncSaw.ar(freq,  sawFreq,0.1) )
```

```
}).send(s);
)

 // all OSC commands generated in the function contained below will be added to a bundle
 // and executed simultaneously after 2 seconds.
(
s.makeBundle(2.0, {
x = Synth.new("tpulse");
a = Bus.control.set(440);
x.busMap(\freq, a);
});
)
x.free;

 // don't send
(
b = s.makeBundle(false, {
x = { PinkNoise.ar(0.1) * In.kr(0, 1); }.play;
});
)
 // now pass b as a pre-existing bundle, and start both synths synchronously
(
 s.makeBundle(nil    // nil executes ASAP
y = { SinOsc.kr(0.2).abs }.play(x, 0, 0, \addBefore); // sine envelope
}, b);
)
x.free; y.free;

 // Throw an Error
(
try {
s.makeBundle(nil, {
s.farkermartin;
});
 } { | error|
  ("Look Ma, normal operations resume even though:\n"
x = { FSinOsc.ar(440, 0, 0.2) }.play; // This works fine
}
)
x.free;
```

Shared Controls

The internal server has a number of shared control buses. Their values can be set or polled using the methods below.

getSharedControl(num) - get the current value of a shared control bus. **num** is the index of the bus to poll. This command is synchronous and only works with the internal server.

setSharedControl(num, value) - set the current value of a shared control bus to **value**. **num** is the index of the bus to set. This command is synchronous and only works with the internal server.

allocSharedControls(numControls) - set the number of shared control buses. Must be done before the internal server is booted. The default is 1024.

Persistent Node Trees

The instance variable **tree** can be used to store a function which will be evaluated after the server is booted, after all nodes are freed, and after cmd-. is pressed. This allows, for example, for one to create a persistent basic node structure. **tree** is evaluated in the method **initTree** after the default group is created, so its existence can be relied upon.

initTree - This method initializes the **[default_group]** and evaluates the **tree** function. This method is called automatically when you boot a Server from the language. **N.B.** If you started a server app from the command line you will have to call initTree manually if you need this functionality.

tree_(aFunction) - sets the function to be evaluated

```
s.quit;
s.tree = {Group.new(s.defaultGroup); "Other code can be evaluated too".postln;};
s.boot;
 s.queryAllNodes; // note the group within the default group
s.tree = nil; s.quit; // reset to default
```

tree - returns the contents of this Server's tree instance variable (most likely a Function).

Keyboard Shortcuts

when a server window is in focus, these shortcuts can be used:

space: start the server
d toggle dumpOSC
n node query
s scope (internal server only)

Scope Support

This only works with the internal server, and currently only on OSX.
see**[Stethoscope]** for further details.

scope(numChannels, index, bufsize, zoom, rate) - Open a scope window showing the output of the Server.

numChannels - the number of channels to be scoped out. The default is this server's options' numOutputBusChannels.
index - the first channel to be output. The default is 0.
bufsize - the size of the buffer for the ScopeView. The default is 4096.
zoom - a zoom value for the scope's X axis. Larger values show more. The default is 1.
rate - whether to display audio or control rate buses (either \audio or \control)

Recording Support

The following methods are for convenience use. For recording with sample accurate start and stop times you should make your own nodes. See the **[DiskOut]** helpfile for more info. For non-realtime recording, see the [**Non-Realtime-Synthesis**] helpfile.

This functionality is also available through the recording button on the server windows. Pressing it once calls **prepareForRecord**, pressing it again calls **record**, and pressing it a third time calls **stopRecording** (see below). When doing so the file created will be in the recordings/ folder and be named for the current date and time.

NOTE: record creates the recording synth after the Server's default group and uses **In.ar**. Thus if you add nodes after the recording synth their output will not be captured. To avoid this, either use **Node** objects (which use the default node as their target) or (when using messaging style) use a target nodeID of 1 .

```
s.sendMsg("/s_new", "default", s.nextNodeID, 1,1);
```

For more detail on this subject see **[Order-of-execution]**, [**default_group**], and **[NodeMessaging]**.

prepareForRecord(path) - Allocates the necessary buffer, etc. for recording the server's output. (See **record** below.) **path** is a String representing the path and name of the output file. If you do not specify a path than a file will be created in the folder recordings/ called SC_thisDateAndTime. Changes to the header or sample format, or to the number of channels must be made BEFORE calling this.

record - Starts or resumes recording the output. You must have called prepareForRecord first (see above).

pauseRecording - Pauses recording. Can be resumed by executing record again.

stopRecording - Stops recording, closes the file, and frees the associated resources. You must call this when finished recording or the output file will be unusable. Cmd-. while recording has the same effect.

recordNode - Returns the current recording synth so that it can be used as a target. This should only be necessary for nodes which are not created in the default group.

The following setter methods have corresponding getters. See **[SoundFile]** for information on the various sample and header formats. Not all sample and header formats are

compatible.

recChannels_(anInteger) - Sets the number of channels to record. The default is two. Must be called BEFORE prepareForRecord.

recHeaderFormat_(aString) - Sets the header format of the output file. The default is "aiff". Must be called BEFORE prepareForRecord.

recSampleFormat_(aString) - Sets the sample format of the output file. The default is "float". Must be called BEFORE prepareForRecord.

Note that the sampling rate of the ouput file will be the same as that of the server app. This can be set using the Server's **[ServerOptions]**.

```
        // start the server

// something to record
(
SynthDef "bubbles"
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps; // glissando function
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4); // echoing sine wave
Out.ar(0, zout);
}).send(s);
SynthDef("tpulse", { arg out=0,freq=700,sawFreq=440.0;
Out.ar(out, SyncSaw.ar(freq,  sawFreq,0.1) )
}).send(s);

)

x = Synth.new("bubbles");

                    // you have to call this first

s.record;

s.pauseRecording; // pausable

         // start again
```

```
                // this closes the file and deallocates the buffer recording node, etc.

        // stop the synths

// look in the recordings/ folder and you'll find a file named for this date and time
```

Asynchronous Commands

Server provides support for waiting on the completion of asynchronous OSC commands such as reading or writing soundfiles. **N.B.** The following methods must be called from within a running **[Routine]**. Explicitly passing in a **[Condition]** allows multiple elements to depend on different conditions. The examples below should make clear how all this works.

bootSync(condition) - Boot the Server and wait until it has completed before resuming the thread. **condition** is an optional instance of **Condition** used for evaluating this.

sendMsgSync(condition, args) - Send the following message to the wait until it has completed before resuming the thread. **condition** is an optional instance of **[Condition]** used for evaluating this. **args** should be one or more valid OSC messages.

sync(condition, bundles, latency) - Send a /sync message to the server, which will replie with the message /synced when all pending asynchronous commands have been completed. **condition** is an optional instance of **[Condition]** used for evaluating this. This may be slightly less safe then **sendMsgSync** under UDP on a wide area network, as packets may arrive out of order, but on a local network should be okay. Under TCP this should always be safe. **bundles** is one or more OSC messages which will be bundled before the sync message (thus ensuring that they will arrive before the /sync message). **latency** allows for the message to be evaluated at a specific point in the future.

```
(
Routine
var c;

 // create a condition variable to control execution of the Routine
c = Condition.new;
```

```
s.bootSync(c);
\BOOTED.postln;

s.sendMsgSync(c, "/b_alloc", 0, 44100, 2);
s.sendMsgSync(c, "/b_alloc", 1, 44100, 2);
s.sendMsgSync(c, "/b_alloc", 2, 44100, 2);
 \b_alloc_DONE
};
)

(
Routine
var c;

 // create a condition variable to control execution of the Routine
c = Condition.new;

s.bootSync(c);
\BOOTED.postln;

s.sendMsg("/b_alloc", 0, 44100, 2);
s.sendMsg("/b_alloc", 1, 44100, 2);
s.sendMsg("/b_alloc", 2, 44100, 2);
s.sync(c);
 \b_alloc_DONE
};
)
```

ID: 381

ServerOptions encapsulates the commandline options for a Server

ServerOptions encapsulates the commandline options for a server app within an object. This makes it convienent to launch multiple servers with the same options, or to archive different sets of options, etc. Every **Server** has an instance of ServerOptions created for it if one is not passed as the options argument when the Server object is created. (This is the case for example with the local and internal Servers which are created at startup.)

A Server's instance of ServerOptions is stored in its **options** instance variable, which can be accessed through corresponding getter and setter methods.

Note: A ServerOptions' instance variables are translated into commandline arguments when a server app is booted. Thus a running Server must be rebooted before changes will take effect. There are also a few commandline options which are not currently encapsulated in ServerOptions. See **Server-Architecture** for more details.

Class Methods

***new**

Create and return a new instance of ServerOptions.

Instance Variables (The Options)

The following instance variables can be changed through getter and setter methods. Note that the defaults listed below only apply to newly created instances of ServerOptions. The options for the local and internal Servers may have been changed at startup in Main-startup or in /scwork/startup.rtf.

numAudioBusChannels - The number of internal audio rate busses. The default is 128.

numControlBusChannels - The number of internal control rate busses. The default is 4096.

The following two options need not correspond to the available number of hardware inputs and outputs.

numInputBusChannels - The number of audio input bus channels. The default is 8.

numOutputBusChannels - The number of audio output bus channels. The default is 8.

numBuffers - The number of global sample buffers available. (See **Buffer**.) The default is 1024.

maxNodes - The maximum number of Nodes. The default is 1024.

maxSynthDefs - The maximum number of SynthDefs. The default is 1024.

protocol - A symbol representing the communications protocol. Either `\udp` or `\tcp`. The default is udp.

blockSize - The number of samples in one control period. The default is 64.

hardwareBufferSize - The preferred hardware buffer size. If non-nil the server app will attempt to set the hardware buffer frame size. Not all sizes are valid. See the documentation of your audio hardware for details.

memSize - The number of kilobytes of real time memory allocated to the server. This memory is used to allocate synths and any memory that unit generators themselves allocate (for instance in the case of delay ugens which do not use buffers, such as **CombN**), and is separate from the memory used for buffers. Setting this too low is a common cause of 'exception in real time: alloc failed' errors. The default is 8192.

numRGens - The number of seedable random number generators. The default is 64.

numWireBufs - The maximum number of buffers that are allocated to interconnect unit generators. (Not to be confused with the global sample buffers represented by **Buffer**.) This sets the limit of complexity of SynthDefs that can be loaded at runtime. This value will be automatically increased if a more complex def is loaded at startup, but it cannot be increased thereafter without rebooting. The default is 64.

sampleRate - The preferred sample rate. If non-nil the server app will attempt to set the hardware sample rate.

loadDefs - A **Boolean** indicating whether or not to load the synth definitions in synthdefs/ (or anywhere set in the environment variable SC_SYNTHDEF_PATH) at startup. The default is `true`.

inputStreamsEnabled - A **String** which allows turning off input streams that you are not interested in on the audio device. If the string is "01100", for example, then only the second and third input streams on the device will be enabled. Turning off streams can reduce CPU load.

outputStreamsEnabled - A **String** which allows turning off output streams that you are not interested in on the audio device. If the string is "11000", for example, then only the first two output streams on the device will be enabled. Turning off streams can reduce CPU load.

blockAllocClass - Specifies the class the server will use to allocate index numbers for buffers and audio and control buses. Should be given as a **class name**, not a symbol. Currently implemented choices are:

PowerOfTwoAllocator: The original allocator. Intended for allocating these resources very quickly for relatively stable configurations. Not ideal for situations where buses or buffers will be allocated and deallocated frequently.

ContiguousBlockAllocator: Designed for allocations that need to change frequently. Sacrifices a small amount of speed for reliability. See the **[ContiguousBlockAllocator]** helpfile.

PowerOfTwoAllocator is the default.

Instance Methods

firstPrivateBus - Returns the index of the first audio bus on this server which is not used by the input and output hardware.

asOptionsString - Returns a **String** specifying the options in the format required by the command-line scsynth app.

For further information see **Server**, **Server-Architecture**, and **Server-Command-Reference**.

Examples

```
// Get the local server's options

o = Server.local.options;

// Post the number of output channels

o.numOutputBusChannels.postln;

// Set them to a new number

                            // The next time it boots, this will take effect

// Create a new instance of ServerOptions

    ServerOptions

// Set the memory size to twice the default

o.memSize = 4096;

// Create a new Server on the local machine using o for its options

t = Server(\Local2, NetAddr("127.0.0.1", 57111), o);
t.makeWindow;
t.boot;
t.quit;
```

ID: 382

Bundling latency

To ensure correct timing of events on the server, OSC messages may be sent with a time stamp, indicating the precise time the sound is expected to hit the hardware output.

In the SuperCollider language, the time stamp is generated behind the scenes based on a parameter called "**latency**."

To understand how latency works, we need to understand the concepts of **logical time** and **physical time**.

Every clock in SuperCollider has both a logical time and physical time.

Physical time: always advances, represents real time.
Logical time: advances only when a scheduling thread wakes up.

While a scheduled function or event is executing, logical time holds steady at the "expected" value. That is, if the event is scheduled for 60 seconds exactly, throughout the event's execution, the logical time will be 60 seconds. If the event takes 2 seconds to execute (very rare), at the end of the event, the logical time will still be 60 seconds but the physical time will be 62 seconds. If the next event is to happen 3 seconds after the first began, its logical time will be 63 seconds. Logical time is not affected by fluctuations in system performance.

This sequencing example illustrates the difference. It's written deliberately inefficiently to expose the problem more clearly. Two copies of the same routine get started at the same time. On a theoretically perfect machine, in which operations take no time, we would hear both channels in perfect sync. No such machine exists, and this is obviously not the case when you listen. The routines also print out the logical time (clock.beats) and physical time (clock.elapsedBeats) just before playing a grain.

```
s.boot;

SynthDef \sinGrain    | out = 0, freq = 440, amp = 0.5, dur = 1|
Out.ar(out, SinOsc.ar(freq, 0, amp) * EnvGen.kr(Env.sine(1), timeScale:dur, doneAction:2));
}).send(s);

2.do({ | chan|
```

```
var rout;
rout = Routine({
var freq;
{ freq = 0;
rrand(400, 1000).do({ freq = freq + 1 });
[thisThread.clock.beats, thisThread.clock.elapsedBeats].postln;
Synth(\sinGrain, [\out, chan, \freq, freq, \dur, 0.005]);
0.1.wait;
}.loop;
});
TempoClock.default.schedAbs(TempoClock.default.elapsedBeats.roundUp(1), rout);
});
```

```
Left channel                    Right channel
Logical vs Physical time        Logical vs Physical time
95          95.001466112        95          95.002988196
95.1        95.101427968        95.1        95.103152311
95.2        95.201250057        95.2        95.202905826
95.3        95.301592755        95.3        95.303724638
95.4        95.401475486        95.4        95.403289141

Average physical latency:
0.00144247559999830             .0032120224000039
```

Notice that even though the left and right channel patterns were scheduled for exactly the same time, the events don't complete executing at the same time. Further, the Synth(...) call instructs the server to play the synth immediately on receipt, so the right channel will be lagging behind the left by about 2 ms each event–and not by the same amount each time. Timing, then, is always slightly imprecise.

This version is the same, but it generates each synth with a 1/4 second latency parameter:

```
2.do({ | chan|
var rout;
rout = Routine({
var freq;
{ freq = 0;
rrand(400, 1000).do({ freq = freq + 1 });
```

```
[thisThread.clock.beats, thisThread.clock.elapsedBeats].postln;
s.makeBundle(0.25, { Synth(\sinGrain, [\out, chan, \freq, freq, \dur, 0.005]); });
0.1.wait;
}.loop;
});
TempoClock.default.schedAbs(TempoClock.default.elapsedBeats.roundUp(1), rout);
});
```

By using makeBundle with a time argument of 0.25, the \s_new messages for the left and right channel are sent with the same timestamp: the clock's current logical time plus the time argument. Note in the table that both channels have the same logical time throughout, so the two channels are in perfect sync.

These routines are written deliberately badly. If they're made maximally efficient, the synchronization will be tighter even without the latency factor, but it can never be perfect. You'll also see this issue, however, if you have several routines executing and several of them are supposed to execute at the same time. Some will execute sooner than others, but *their logical time will all be the same*. If they're all using the same amount of latency, you will still hear them at the same time.

In general, all synths that are triggered by live input (MIDI, GUI, HID) should specify no latency so that they execute as soon as possible. All sequencing routines should use latency to ensure perfect timing.

The latency value should allow enough time for the event to execute and generate the OSC bundle, and for the server to interpret the message and render the audio in time to reach the hardware output on time. If the client and server are on the same machine, this value can be quite low. Running over a network, you must allow more time. (Latency compensates for network timing jitter also.)

Pbind automatically imports a latency parameter from the server's latency variable. You can set the default latency for event patterns like this:

```
myServer.latency = 0.2;  // 0.2 is the default
```

Here are three ways to play a synth with the latency parameter:

```
// messaging style
// s.nextNodeID is how to get the next unused node ID from the server
s.sendBundle(latency, [\s_new, defName, s.nextNodeID, targetID, addAction, arguments]);
```

```
// object style, asking the object for the message
synth = Synth.basicNew(defName, s);
s.sendBundle(latency, synth.newMsg(target, arguments, addAction));

// object style, using automatic bundling
// like the previous example, when this finishes you'll have the Synth object in the synth variable
s.makeBundle(latency, { synth = Synth(defName, arguments, target, addAction); });
```

ID: 383

Synth client-side representation of a synth node on the server

superclass: Node

A Synth is the client-side representation of a synth node on the server. A Synth represents a single sound producing unit. What it does is defined in a **SynthDef**, which specifies what UGens are used and how they are patched together. It also specifies what inputs and outputs the Synth will have. A SynthDef is thus a kind of fixed pattern, upon which Synths are be based. (Despite this, a given SynthDef can provide a surprising amount of variation.) For more detail on SynthDefs, their construction, and how to send them to a server, see the **SynthDef** help file.

For more on the important distinction between client objects and server nodes, see **ClientVsServer**. For information on creating nodes without using objects, see **NodeMessaging**.

N.B. Synth is a subclass of **Node**, and thus many of its most useful and important methods are documented in the **Node** help file.

Order of Execution

Order of execution is a crucial issue when creating Synths which interact with each other.

sound -> filter

If a sound is to be passed through a filter, the synth that does the filtering must be later in the order of execution than the synth which is its input. The computer must calculate a buffer's worth of sound, and then the computer moves on to calculate a buffer's worth of the filtered version of that sound.

The actual interconnection between synth nodes is accomplished with buses. See **Bus** and **Server-Architecture** for details.

See the **Order-of-execution** help file for a more detailed discussion of this important topic.

Bundling

Some of the methods below have two versions: a regular one which sends its corresponding message to the server immediately, and one which returns the message in an **Array** so that it can be added to a bundle. It is also possible to capture the messages generated by the regular methods using Server's automated bundling capabilities. See **Server** and **bundledCommands** for more details.

Accessing Instance Variables

defName - Returns the name of this Synth's SynthDef.

For other instance variables see **Node**.

Creation with Immediate Instantiation on the Server

*new(defName, args: [arg1, value1, ... argN, valueN], target, addAction)

Create and return a new Synth object, and immediately start the corresponding synth node on the server.

defName - A **String** or **Symbol** specifying the name of the **SynthDef** to use in creating the Synth.

args - An optional array specifying initial values for the SynthDef's arguments (controls). These are specified in pairs of control name or index and value. If names are used they can be specified with either Strings or Symbols. e.g. `[\frequency, 440, \amplitude, 1, ...]`

target - A target for this Synth. If target is not a **Group** or Synth, it will be converted as follows: If it is a **Server**, it will be converted to the **default_group** of that server. If it is nil, to the **default_group** of the default Server. If it is an integer, it is created relative to a group with that id.
Note: A Synth is not a valid target for `\addToHead` and `\addToTail`.

addAction - one of the following Symbols:
`\addToHead` - (the default) add at the head of the group specified by **target**
`\addToTail` - add at the tail of the group specified by **target**
`\addAfter` - add immediately after **target** in its server's node order
`\addBefore` - add immediately before **target** in its server's node order
`\addReplace` - replace **target** and take its place in its server's node order

```
s.boot;
  // create a Synth at the head of the default Server's default group
  // based on the SynthDef "default"
x = Synth.new("default");
  s.queryAllNodes; // note the default group (ID 1)
x.free;
```

*newPaused(defName, args: [arg1, value1,... argN, valueN], target, addAction)

As ***new** above, but creates a node which is paused. This can be started by calling run on it.

```
s.boot;
x = Synth.newPaused("default");
s.queryAllNodes; // see I'm here
  x.run; // true is the default
  x.run(false   // pause me again
x.free;
```

*grain(defName, args: [arg1, value1, ... argN, valueN], target, addAction)

A convenience method which will create a synth node with an node ID of -1. Such a node cannot be messaged after creation. As such this method does not create an object, and returns nil. For details of its arguments see ***new** above.

The following convenience methods correspond to the add actions of **Synth.new**:

*after(aNode, defName, args)

Create and return a Synth and add it immediately after **aNode.**

*before(aNode, defName, args)

Create and return a Synth and add it immediately before **aNode.**

*head(aGroup, defName, args)

Create and return a Synth. If **aGroup** is a **Group** add it at the head of that group. If it is a **Server**, add it at the head of the **default_group** of that server. If it is nil, add it at the head of the **default_group** of the default server**.** If it is an integer, it is created relative to a group with that id.

*tail(aGroup, defName, args)

Create and return a Synth. If **aGroup** is a **Group** add it at the tail of that group. If it is a **Server**, add it at the tail of the **default_group** of that server. If it is nil, add it at the tail of the **default_group** of the the default server**.** If it is an integer, it is created relative to a group with that id.

*replace(nodeToReplace, defName, args)

Create and return a Synth and use it to replace **nodeToReplace**, taking its place in its server's node order.

Creation without Instantiation on the Server

For use in message bundles it is also possible to create a Synth object in the client app without immediately creating a synth node on the server. Once done one can call methods which create messages to add to a bundle, which when sent to the server will instantiate the synth.

*basicNew(defName, server, nodeID)

Create and return a Synth object without creating a synth node on the server.

defName - A **String** or **Symbol** specifying the name of the **SynthDef** to use in creating the Synth.

server - An optional instance of **Server**. If nil this will default to the default **Server**.

nodeID - An optional node ID number. If not supplied one will be generated by the Server's NodeIDAllocator. Normally you should not need to supply an ID.

```
s.boot;
x = Synth.basicNew("default", s); // Create without sending
  s.sendBundle(nil              // Now send a message; create at the head of s' default group
```

```
s.queryAllNodes;
x.free;
```

newMsg(target, args, addAction)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth. If target is nil, it will default to the **default_group** of the Server specified in ***basicNew** when this Synth was created. The default addAction is `\addToHead`. (See ***new** above for details of addActions and args.)

addToHeadMsg(aGroup, args)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth. If **aGroup** is a **Group** it will be added at the head of that group. If it is nil, it will be added at the head of the **default_group** of this Synth's server (as specified when ***basicNew** was called). See ***new** above for details on **args**.

addToTailMsg(aGroup, args)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth. If **aGroup** is a **Group** it will be added at the tail of that group. If it is nil, it will be added at the tail of the **default_group** of this Synth's server (as specified when ***basicNew** was called). See ***new** above for details on **args**.

addBeforeMsg(aNode, args)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth, immediately before **aNode**. See ***new** above for details on **args**.

addAfterMsg(aNode, args)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth, immediately after **aNode**. See ***new** above for details on **args**.

addReplaceMsg(nodeToReplace, args)

Returns a message of the type **s_new** which can be bundled. When sent to the server this message will instantiate this synth, replacing **nodeToReplace** in the server's node order. See ***new** above for details on **args**.

Control

For further methods of controlling Synths (set, map, busMap, etc.), see the **Node** helpfile.

get(index, action)
getMsg(index)

Query the server for the current value of a **Control** (argument). **index** is a control name or index. **action** is a **Function** which will be evaluated with the value passed as an argument when the reply is received.

```
s.boot;
(
SynthDef("help-Synth-get", { arg freq = 440;
Out.ar(0, SinOsc.ar(freq, 0, 0.1));
}).send(s);
)
  x = Synth "help-Synth-get"
x.set(\freq, 220 + 440.rand);
x.get(\freq, { arg value; ("freq is now:" + value + "Hz").postln; });
x.free;
```

getn(index, count, action)
getnMsg(index, count)

Query the server for the current values of a sequential range of Controls (arguments). **index** is a control name or index. **count** is the number of sequential controls to query, starting at **index**. **action** is a **Function** which will be evaluated with an **Array** containing the values passed as an argument when the reply is received.

Examples

```
// boot the default server
```

```
    Server          // just to be sure
s.boot;


(
// send a synth def to server
SynthDef("tpulse", { arg out = 0,freq = 700, sawFreq = 440.0;
Out.ar(out, SyncSaw.ar(freq, sawFreq, 0.1));
}).send(s);
)

// Here the defaults for *new will result in a Synth at the head of the default group
// of the default Server.  This will use the SynthDef's default arguments;
y = Synth.new("tpulse");
y.free;

// The same done explicitly
y = Synth.new("tpulse", nil, s, \addToHead);
y.free;

// With some arguments
y = Synth.new("tpulse", [\freq, 350, \sawFreq, 220]);
y.free;

// make a new synth
    Synth "tpulse"

// pause
y.run(false);

y.run(true);

// set a control by argument name
y.set("freq", 200);

// or by index
y.set(2, 100.0);

// modulate out to bus number 1 (the right speaker)
y.set(0, 1);
```

```
//  multiple set commands in one message
y.set("out", 0, "freq",300);

// free the synth from the server
y.free;



////////////// Filtering

(
// first collect some things to play with
SynthDef("moto-rev", { arg out=0;
var x;
x = RLPF.ar(LFPulse.ar(SinOsc.kr(0.2, 0, 10, 21), [0,0.1], 0.1),
100, 0.1).clip2(0.4);
Out.ar(out, x);
}).send(s);

SynthDef("bubbles", { arg out=0;
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps;
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4); // echoing sine wave
Out.ar(out, zout);
}).send(s);


SynthDef("rlpf",{ arg out=0,ffreq=600,rq=0.1;
ReplaceOut.ar( out, RLPF.ar( In.ar(out), ffreq,rq) )
}).send(s);


SynthDef("wah", { arg out, rate = 1.5, cfreq = 1400, mfreq = 1200, rq=0.1;
var zin, zout;

zin = In.ar(out, 2);
cfreq = Lag3.kr(cfreq, 0.1);
mfreq = Lag3.kr(mfreq, 0.1);
rq    = Ramp.kr(rq, 0.1);
```

```
zout = RLPF.ar(zin, LFNoise1.kr(rate, mfreq, cfreq), rq, 10).distort
* 0.15;

 // replace the incoming bus with the effected version
ReplaceOut.ar( out , zout );

}).send(s);

SynthDef("modulate",{ arg out = 0, freq = 1, center = 440, plusMinus = 110;
Out.kr(out, SinOsc.kr(freq, 0, plusMinus, center));
}).send(s);
)

// execute these one at a time

// y is playing on bus 0
    Synth "moto-rev"  "out"

// z is reading from bus 0 and replacing that; It must be *after* y
z = Synth.after(y,"wah",["out",0]);

// stop the wah-ing
z.run(false);

// resume the wah-ing
z.run(true);

// add a rlpf after that, reading and writing to the same buss
x = Synth.after(z,"rlpf",["out",0]);

// create another rlpf after x
t = Synth.after(x,"rlpf",["out",0]);

x.set("ffreq", 400);

      \ffreq        // Symbols work for control names too

// Now let's modulate x's ffreq arg
// First get a control Bus
b = Bus.control(s, 1);
```

```
// now the modulator, *before* x
m = Synth.before(x, "modulate", [\out, b.index]);

// now map x's ffreq to b
x.busMap("ffreq", b);

m.set("freq", 4, "plusMinus", 20);

x.free;
z.free;
m.free;

// now place another synth after y, on the same bus
// they both write to the buss, adding their outputs
r = Synth.after(y,"bubbles",["out",0]);

y.free;

r.free;

// look at the Server window
// still see 4 Ugens and 1 synth?
// you can't hear me, but don't forget to free me
t.free;
```

ID: 384

SynthDef client-side representation of a synth definition

superclass: Object

The server application uses synth definitions as templates for creating **[Synth]** nodes. (Methods such as **Function-play**, etc. are simply conveniences which automatically create a def for you.) The SynthDef class encapsulates the client-side representation of a given def, and provides methods for creating new defs, writing them to disk, and streaming them to a server.

SynthDef is one of the more complicated classes in SC and an exhaustive explanation of it is beyond the scope of this document. As such, the examples at the bottom of this document and those found in the various tutorials accessible from **[Help]** may be necessary to make some aspects of its use clear.

UGen Graph Functions and Special Argument Forms

The core of a def is its unit generator graph function. This is an instance of **[Function]** which details how the def's unit generators are interconnected, its inputs and outputs, and what parameters are available for external control. In a synth based on the def, arguments to the function will become instances of **[Control]**. These can have default values, or can be set at the time the synth is created. After creation they will be controllable through Node's **set** and **setn** methods, or the n_set and n_setn OSC messages.

There are three special types of arguments, which are treated differently:

initial rate -Arguments that begin with "i_" (e.g. i_freq), or that are specified as **ir** in the def's rates argument (see below), will be static and non-modulatable. They will not respond to /n_set or /n_map. This is slightly more efficient in terms of CPU than a regular arg.

trigger rate -Arguments that begin with "t_" (e.g. t_trig), or that are specified as **tr** in the def's rates argument (see below), will be made as a **TrigControl**. Setting the argument will create a control-rate impulse at the set value. This is useful for triggers.

literal arrays -Arguments which have literal arrays as default values (see **[Literals]**) result in multichannel controls, which can be set as a group with **Node-setn** or n_setn.

When setting such controls *no bounds checking is done*, so you are responsible for making sure that you set the correct number of arguments.

See the examples below for more detail on how this works.

Certain argument names (such as 'out' to specify an out bus) are in such common use that adopting them might be said to constitute 'good style'. One of these, '**gate**' when used to control the gate input of an **[EnvGen]**, deserves special mention, as it allows one to use Node's **release** method. See **[Node]** for an example and more detail.

Static versus Dynamic Elements

It is important to understand that although a single def can provide a great deal of flexibility through its arguments, etc., it is nevertheless a static entity. A def's UGen graph function (and the SC code within it) is evaluated *only* when the def is created. Thus 'if' statements, etc. will have no further effect at the time the def is used to create a Synth, and it is important to understand that a UGen graph function should not be designed in the same way as functions in the language, where multiple evaluations can yield different results. It will be evaluated *once and only once*.

There are other ways of achieving similar results, however, often using UGens such as **[Rand]**. For example, the following def will have a single randomly generated frequency, which will be the same for every Synth based on it:

```
(
SynthDef("help-notRand", { Out.ar(0, SinOsc.ar(rrand(400, 800), 0, 0.2)
* Line.kr(1, 0, 1, doneAction:  2)); }).send(s);
)
    Synth "help-notRand"
    Synth "help-notRand"   // the same freq as a
```

This one on the other hand will have a different random freq for each Synth created:

```
(
SynthDef("help-isRand", { Out.ar(0, SinOsc.ar(Rand(400, 800), 0, 0.2)
* Line.kr(1, 0, 1, doneAction:  2)); }).send(s);
)
    Synth "help-isRand"
    Synth "help-isRand"   // a different randomly selected freq
```

Class Methods

*new(name, ugenGraphFunc, rates, prependArgs, variants)

Create a SynthDef instance, evaluate the ugenGraphFunc and build the ugenGraph.

name - A **[String]** or **[Symbol]** (i.e. "name" or \name). This name will be used to refer to the SynthDef when creating a **[Synth]** based upon it, and should be unique.

ugenGraphFunc - An instance of **[Function]** specifying how the def's UGens are interconnected. See the discussion above for information on how the Function's arguments are specified.

rates - An optional Array of specifications for the ugenGraphFunc's arguments. The order corresponds to the order of arguments. See the examples below to see how these are used.
A specification can be:
nil/zero A standard control rate **[Control]** is created.
a float the Control will have a lag of the specified time. This can be used to create smooth transitions between different values. t_ and i_ args cannot be lagged.
ir The Control can be set only at creation ('initial rate'). See discussion above.
tr The Control is used as a trigger. See discussion above.

prependArgs - An optional Array of objects which will be passed as the first arguments to the **ugenGraphFunc** when it is evaluated. Arguments which receive values in this way will not be converted to instances of **[Control]**. See the ***wrap** example below for an example of how this can be used.

variants - An optional **[Event]** containing default argument settings. These can override the defaults specified in the **ugenGraphFunc**. When creating a Synth a variant can be requested by appending the defName argument in the form "name.variant". See example below.

*writeOnce(name, ugenGraphFunc, rates, prependArgs, dir)

Create a new SynthDef and write it to disk, providing a def file with this name does not already exist. This is useful in class definitions so that the def is not written every time the library is compiled. Note that this will not check for differences, so you will need to delete the defFile to get it to rebuild. Default for **dir** is synthdefs/.

***wrap(ugenGraphFunc, rates, prependArgs)**

Wraps a def within an enclosing synthdef. Can be useful for mass-producing defs. See example below.

***synthDefDir**
***synthDefDir_(dir)**

Get or set the default directory to which defs are written.

Instance Methods

writeDefFile(dir)

Writes the def as a file called name.scsyndef in a form readable by a server. Default for **dir** is synthdefs/. Defs stored in the default directory will be automatically loaded by the local and internal Servers when they are booted.

load(server, completionMessage, dir)

Write the defFile and send a message to **server** to load this file. When this asynchronous command is completed, the **completionMessage** (a valid OSC message) is immediately executed by the server. Default for **dir** is synthdefs/.

send(server, completionMessage)

Compile the def and send it to **server** without writing to disk (thus avoiding that annoying SynthDef buildup). When this asynchronous command is completed, the **completionMessage** (a valid OSC message) is immediately executed by the server.

store(libname, dir, completionMessage)

Write the defFile and store it in the **SynthDescLib** specified by **libname**, and send a message to the library's server to load this file. When this asynchronous command is completed, the **completionMessage** (a valid OSC message) is immediately executed by the server. Default for **libname** is \global, for **dir** is synthdefs/. This is needed to use defs with the event stream system. See **[Streams]** and **[Pattern]**.

play(target, args, addAction)

A convenience method which compiles the def and send it to **target's** server. When this asynchronous command is completed, create one synth from this definition, using the argument values specified in the Array **args**. Returns a corresponding **Synth** object. For a list of valid **addActions** see **[Synth]**. The default is \addToHead.

name

Return this def's name.

variants

Return an **[Event]** containing this def's variants.

Examples

Basic

```
// Note that constructions like SynthDef(...)  and Synth(...)  are short for SynthDef.new(...), etc.
// With SynthDef it is common to chain this with calls on the resulting instance,
// e.g.  SynthDef(...).send(s) or SynthDef(...).play

// make a simple def and send it to the server

s.boot;
SynthDef(\SimpleSine, { arg freq = 440; Out.ar(0, SinOsc.ar(freq, 0, 0.2)) }).send(s);

// the above is essentially the same as the following:
d = SynthDef.new(\SimpleSine, { arg freq = 440; Out.ar(0, SinOsc.ar(freq, 0, 0.2)) });
d.send(s);

// now make a synth from it, using the default value for freq, then another with a different value
    Synth \SimpleSine
y = Synth(\SimpleSine, [\freq, 660]);

// now change the freq value for x
x.set(\freq, 880);
```

```
x.free; y.free;

// using the play convenience method
x = SynthDef(\SimpleSine, { arg freq = 440; Out.ar(0, SinOsc.ar(freq, 0, 0.2)) }).play
x.free;
```

Argument Rates

```
// the following two defs are equivalent.  The first uses a 't_' arg:
(
SynthDef "trigTest"    arg                    // t_trig creates a TrigControl
Out.ar(0, SinOsc.ar(freq+[0,1], 0, Decay2.kr(t_trig, 0.005, 1.0)));
           // lag the freq by 4 seconds (the second arg), but not t_trig (won't work anyway)
);
)

// This second version makes trig a \tr arg by specifying it in the rates array.  Send this one.
(
SynthDef("trigTest2", { arg trig=0, freq=440;
Out.ar(0, SinOsc.ar(freq+[0,1], 0, Decay2.kr(trig, 0.005, 1.0)));
 }, [\tr      // lag the freq (lagtime:  4s), \tr creates a TrigControl for trig
).send(s);
)

// Using the second version create a synth
z = Synth.head(s, \trigTest2);

// now trigger the decay envelope
      \trig          // you can do this multiple times
      \trig     \freq         // hear how the freq lags
z.set(\trig, 1, \freq, 880);

        //free the synth
```

Variants

```
// create a def with some variants
(
```

```
SynthDef "vartest"   | out=0, freq=440, amp=0.2, a = 0.01, r = 1|
 // the EnvGen with doneAction:  2 frees the synth automatically when done
Out.ar(out, SinOsc.ar(freq, 0, EnvGen.kr(Env.perc(a, r, amp), doneAction:  2)));
}, variants:  (alpha:  [a:  0.5, r:  0.5], beta:  [a:  3, r:  0.01], gamma:  [a:  0.01, r:  4])
).send(s);
)

// now make some synths.  First using the arg defaults
Synth "vartest"

// now the variant defaults
Synth "vartest.alpha"
Synth "vartest.beta"
Synth "vartest.gamma"

// override a variant
Synth("vartest.alpha", [\release, 3, \freq, 660]);
```

Literal Array Arguments

```
// freqs has a literal array of defaults.  This makes a multichannel Control of the same size.
(
SynthDef("arrayarg", { arg amp = 0.1, freqs = #[300, 400, 500, 600], gate = 1;
var env, sines;
env = Linen.kr(gate, 0.1, 1, 1, 2) * amp;
sines = SinOsc.ar(freqs +.t [0,0.5]).cubed.sum; // A mix of 4 oscillators
Out.ar(0, sines * env);
}, [0, 0.1, 0]).send(s);
)

    Synth "arrayarg"
x.setn("freqs", [440, 441, 442, 443]);

// Don't accidentally set too many values, or you may have unexpected side effects
// The code below inadvertantly sets the gate arg, and frees the synth
x.setn("freqs", [300, 400, 500, 600, 0]);

// Mr.  McCartney's more complex example
(
```

```
fork {
 z = Synth "arrayarg"

2.wait;
10.do {
z.setn(\freqs, {exprand(200,800.0)} !  4);
(2 ** (0..3).choose * 0.2).wait;
};

z.set(\amp, -40.dbamp);

10.do {
z.setn(\freqs, {exprand(200,800.0)} !  4);
(2 ** (0..3).choose * 0.2).wait;
};
2.wait;

z.release;
};
)
```

Wrapping Example: 'factory' production of effects defs

```
// The makeEffect function below wraps a simpler function within itself and provides
// a crossfade into the effect (so you can add it without clicks), control over wet
// and dry mix, etc.
// Such functionality is useful for a variety of effects, and SynthDef-wrap
// lets you reuse the common code.
(
// the basic wrapper
makeEffect = { arg name, func, lags, numChannels = 2;

SynthDef(name, { arg i_bus = 0, gate = 1, wet = 1;
var in, out, env, lfo;
in = In.ar(i_bus, numChannels);
env = Linen.kr(gate, 2, 1, 2, 2); // fade in the effect

  // call the wrapped function.  The in and env arguments are passed to the function
  // as the first two arguments (prependArgs).
```

```
    // Any other arguments of the wrapped function will be Controls.
out = SynthDef.wrap(func, lags, [in, env]);

XOut.ar(i_bus, wet * env, out);
}, [0, 0, 0.1] ).send(s);

};
)

// now make a wah
(
makeEffect.value(\wah, { arg in, env, rate = 0.7, ffreq = 1200, depth = 0.8, rq = 0.1;
 // in and env come from the wrapper.  The rest are controls
var lfo;
lfo = LFNoise1.kr(rate, depth * ffreq, ffreq);
RLPF.ar(in, lfo, rq, 10).distort * 0.15; },
 [0.1, 0.1, 0.1, 0.1],  // lags for rate ffreq, depth and rq
 2 // numChannels
);
)

// now make a simple reverb
(
makeEffect.value(\reverb, { arg in, env;
 // in and env come from the wrapper.
var input;
input = in;
16.do({ input = AllpassC.ar(input, 0.04, Rand(0.001,0.04), 3)});
input; },
 nil   // no lags
 2 // numChannels
);
)

// something to process
x = { {Decay2.ar(Dust2.ar(3), mul:  PinkNoise.ar(0.2))} !  2}.play;

y = Synth.tail(s, \wah);
z = Synth.tail(s, \reverb, [\wet, 0.5]);
```

```
// we used an arg named gate, so Node-release can crossfade out the effects
y.release;

// setting gate to zero has the same result
z.set(\gate, 0);

x.free;
```

common argument names: out and gate

```
// arguments named 'out' and 'gate' are commonly used to specify an output bus and
// EnvGen gate respectively.  Although not required, using them can help with consistency
// and interchangeability.  'gate' is particularly useful, as it allows for Node's release
// method.
(
SynthDef(\synthDefTest, { arg out, gate=1, freq=440;
 // doneAction:  2 frees the synth when EnvGen is done
Out.ar(out, SinOsc.ar(freq) * EnvGen.kr(Env.asr(0.1, 0.3, 1.3), gate, doneAction:2));
          // use store for compatibility with pattern example below
)

    Synth \synthDefTest   \out       // play out through hardware output bus 0 (see Out.help)
           // releases and frees the synth (if doneAction is > 2; see EnvGen)

//equivalent:

    Synth \synthDefTest   // out defaults to zero, if no default arg is given.
x.set(\gate, 0);

// if value is negative, it overrides the release time, to -1 - gate
    Synth \synthDefTest
      \gate       // 4 second release

//equivalent:
    Synth \synthDefTest
x.release(4);

// if the out arg is used in a standard way, it can always be changed without knowing the synth def
    Synth \synthDefTest   \out
```

```
      \out      //play through channel 1
x.release;

// Another good example of this is with patterns, which can use gate to release notes
(
Pbind
 \instrument  \synthDefTest
\freq, Pseq([500, 600, Prand([200, 456, 345],1)], inf),
\legato, Pseq([1.5, 0.2], inf),
\dur, 0.4,
\out, Pseq([0, 1], inf)
).play;
)
```

ID: 385

SynthDesc description of a synth definition

contains information about a SynthDef, such as its name, its control names and default values.
also information is provided of its outputs and inputs and whether it has a gate control.

***read(path, keepDefs, dict)**
adds all synthDescs in a path to a dict

SynthDescs are created by **SynthDescLib**, by reading a compiled synth def file.

```
SynthDescLib              "synthdefs/default.scsyndef"
SynthDescLib.global.synthDescs.at(\default)
```

name returns the name of the SynthDef
controlNames returns an array of instances of ControlName, each of which have the following fields: name, index, rate, defaultValue

```
SynthDescLib.global.synthDescs.at(\default).controlNames.postln;
```

hasGate is true if the Synthdef has a gate input
msgFunc the function which is used to create an array of arguments for playing a synth def in patterns

```
SynthDescLib.global.synthDescs.at(\default).msgFunc.postcs;
```

SynthDescs are needed by the event stream system, so when using **Pbind** and **NotePlayer**,

the instruments' default parameters are derived from the SynthDesc.

aSynthDef.store also creates a synthDesc in the global library:

```
(
```

```
 SynthDef("test", { arg out, freq, xFade;
XOut.ar(out, xFade, SinOsc.ar(freq))
}).store
);

 SynthDescLib                // browse the properties of SynthDescs
```

24 Streams

ID: 386

BinaryOpStream

Superclass: Stream

A BinaryOpStream is created as a result of a binary math operation on a pair of Streams. It is defined to respond to **next** by returning the result of the math operation on the **next** value from both streams. It responds to **reset** by resetting both Streams.

```
(FuncStream.new({ 9.rand }) + 100).dump
```

```
(
x = (FuncStream.new({ 9.rand }) + 100);
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
)
```

ID: 387

CSVFileReader

reads comma-separated text files into 2D arrays line by line.

**Use TabFileReader for tab delimited files,
and FileReader for space-delimited files, or custom delimiters.**

*read(path, skipEmptyLines, skipBlanks, func)

```
(
// write a test file:
    File "CSVReadTest.sc"  "w"
f.write(
"Some,comma,delimited,items, in line 1


and then, some more, with several commas,,,, in line 3
"
);
f.close;
)
 // open file, read and put strings into array, close file.
x = CSVFileReader.read("CSVReadTest.sc").postcs;

 // can skip empty lines:
x = CSVFileReader.read("CSVReadTest.sc", true).postcs;

 // can skip blank entries caused by multiple commas :
x = CSVFileReader.read("CSVReadTest.sc", true, true).postcs;

 // do file open/close by hand if you prefer:
f = File("CSVReadTest.sc", "r"); f.isOpen;
    CSVFileReader
t.read(true, true).postcs;
f.close;

(
// write a test file with numbers:
    File "CSVReadTestNum.sc"  "w"
```

```
(1..10).do { | n|  f.write(n.asString ++ ","); };
f.close;
)

x = CSVFileReader.read("CSVReadTestNum.sc", true, true).postcs;
x.collect(_.collect(_.interpret)); // convert to numbers.

      // or do it immediately:
x = CSVFileReader.readInterpret("CSVReadTestNum.sc").postcs;

(
// write a test file with several lines of numbers:
    File "CSVReadTestNum.sc"  "w"

(1..100).do { | n|
f.write(n.asString ++ if (n % 10 != 0, ",", Char.nl)); };
f.close;
)


x = CSVFileReader.readInterpret("CSVReadTestNum.sc", true, true).postln;
```

ID: 388

EventStream

superclass: Object

An event stream is a normal [**Stream**], that returns events when called. (see class [**Event**])
Usually, an event stream requires an event to be passed in, often the default event is used:

```
t = Pbind(\x, Pseq([1,2,3])).asStream; // pbind, e.g. creates an event stream
t.next(Event.default);
t.next(Event.default);
```

An event stream cannot be played directly with a clock, as it returns events and not time values.
Therefore, an [**EventStreamPlayer**] is used, which replaces the event by according time value.

ID: 389

EventStreamPlayer

superclass: PauseStream

An **EventStreamPlayer** is used by Event based Patterns.
You do not explictly create an EventStreamPlayers, they are created for you
when you call Pattern-play.

The **EventStreamPlayer** holds a stream which returns a series of Events, and a protoEvent. At each call to next, it copies the protoEvent, passes that to the stream, and calls **play** on the **Event** returned.

for more on **EventStreamPlayer** see [Streams-Patterns-Events4]

ID: 390

FileReader

reads space-delimited text files into 2D arrays line by line.

**Use TabFileReader for tab delimited files,
and CSVFileReader for comma-separated files.**

*read(path, skipEmptyLines, skipBlanks, func, delimiter)

```
(
// write a test file:
    File "FileReaderTest.sc"  "w"
f.write(
"Some space delimited items in line 1

and then some more with several blanks     in line 3
"
);
f.close;
)
 // open file, read and put strings into array, close file.
x = FileReader.read("FileReaderTest.sc").postcs;

 // can skip empty lines:
x = FileReader.read("FileReaderTest.sc", true).postcs;

 // can skip blank entries caused by multiple spaces :
x = FileReader.read("FileReaderTest.sc", true, true).postcs;

 // do file open/close by hand if you prefer:
    File "FileReaderTest.sc"  "r"
    FileReader    true  true
t.read;
f.close;

// take letter "a" as delimiter:
    FileReader      "FileReaderTest.sc"  true  true, delimiter:  $a

(
```

```
// write a test file with numbers:
    File "FileReadTestNum.sc"  "w"

(1..10).do { | n|  f.write(n.asString ++ " "); };
f.close;
)

    FileReader      "FileReadTestNum.sc"
x.collect(_.collect(_.interpret)); // convert to numbers.

      // or do it immediately:
x = FileReader.readInterpret("FileReadTestNum.sc").postcs;

(
// write a test file with several lines of numbers:
    File "FileReadTestNum.sc"  "w"

(1..100).do { | n|
f.write(n.asString ++ if (n % 10 != 0, " ", Char.nl)); };
f.close;
)


x = FileReader.readInterpret("FileReadTestNum.sc", true, true).postln;
```

ID: 391

NodeEvent

The methods **Event-synth** and **Event-group** set the parent event of the receiver to specialized events that duplicate the functionality of Synth and Group objects. These objects follow the naming conventions of patterns (i.e., groups and addActions are integer ID's) and have the same stop/play/pause/resume interface as the EventStreamPlayers created by Pattern-play. So, they can be used interchangeably with patterns in control schemes and GUI's.

The following example creates a group with nodeID = 2 and plays a synth within it.

```
g = (id: 2).group;
g.play;
a = (group: 2).synth
a.play;
g.release;
g.stop;
```

Caution: the play method returns a time value (Event-delta), so an expression of the form
a = (type: \Group, id: 1).play
will assign the default duration of 1 to the variable a, not the group event!

interface:

play starts synth or group, returns this.delta
stop if ev[\hasGate] == true set gate to 0, otherwise frees the node
pause disables the node
resume reenables the node
set(key, value) sets control identified by key to value
split returns an array of events, one for each synth or group specified by the receiver

map(key, busID) maps control to control bus
before(nodeID) moves to immediately before nodeID
after(nodeID)
headOf(nodeID)

tailOf(nodeID)

With the exception of server, latency, and instrument any key in the event can have an array as a
value and the standard rules of multi-channel expansion will be followed.

Here is a simple example of its use:

```
(
g = (id: [2,3,4,5,6], group: 0, addAction: 1).group ;  // define a multiple Group event
g.play;                // play it

b = ( freq: [500,510], group: [2,3]).synth;            // make a Synth event
b.play;

b.set(\freq,[1000,1006])

g.release

b.play;
h = g.split;                                           // split into individual group events
c = b.split;      // and synth events
c[0].set(\freq,700);
c[1].set(\freq,400);

h[0].stop;
h[1].stop;

g.stop;
)
```

ID: 392

Padd event pattern that adds to existing value of one key

superclass: Pset

Padd(name, value, pattern)

```
(
var a, b;
a = Padd(\freq, 801, Pbind(\freq, 100));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Padd(\freq, Pseq([401, 801], 2), Pbind(\freq, 100));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, gate=1;
var env;
env = EnvGen.kr(Env.asr(0.001, 1, 0.2), gate, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)
```

```
(
    Pbind \dur       \instrument  \sinegrain, \freq, 440
b = Padd(\freq, Pseq([10, 30, 100], inf), a);
b.play;
)
```

ID: 393

Paddp event pattern that adds with existing value of one key

superclass: Psetp

Paddp(name, value, pattern)

multiplies a value in an event stream until it ends, repeats this with new values until the value stream ends.

value can be a pattern, a stream or an array. the resulting stream ends when that incoming stream ends.

```
(
var a, b;
a = Paddp(\freq, Pseq([2, 3, pi],inf), Pbind(\freq, Pseq([100, 200, 300])));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)

(
a = Pbind(\freq, Pseq([500, 600, 700]), \instrument, \sinegrain);
a = Paddp(\freq, Pseq([30, 90, -100], inf), a);
a.play;
```

)

ID: 394

Paddpre event pattern that adds to existing value of one key

superclass: FilterPattern

Pset(name, value, pattern)

adds a value in an event, before it is passed up the stream.
to add to the value after it has been passed to the stream, use **Padd**

```
(
var a, b;
a = Paddpre(\x, 8, Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next((\x:4)).postln; });
)
```

Paddpre does not override incoming values:

```
(
var a, b;
a = Paddpre(\freq, 302, Pset(\freq, 500, Pbind(\dur, 0.3)));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Paddpre(\legato, Pseq([0.2, 0.4], 2), Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)



(
a = Pbind(\dur, 0.5, \instrument, \sinegrain);
b = Paddpre(\freq, Pseq([10, 30, 100], inf), a);
b.play;
)
```

ID: 395

Pattern

superclass: AbstractFunction

Patterns versus Streams

Pattern is an abstract class that is the base for the Patterns library. These classes form a
rich and concise score language for music. The series of help files entitled **Streams-Patterns-Events**
gives a detailed introduction. This attemps a briefer characterization.

A **Stream** is an object that responds to `next`, `reset`, and `embedInStream`. Streams
represent sequences of values that are obtained one at a time by with message `next`. A `reset`
message will cause the stream to restart (many but not all streams actually repeat themselves.)
If a stream runs out of values it returns `nil` in response to `next`. The message `embedInStream`
allows a stream definition to allow another stream to "take over control" of the stream.
All objects respond to `next` and `reset`,most by returning themselves in response to next.
Thus, the number 7 defines a Stream that produces an infinite sequence of 7's. Most objects
respond to `embedInStream` with a singleton Stream that returns the object once.

A **Pattern** is an object that responds to `asStream` and `embedInStream`. `A` Pattern
defines the behavior of a Stream and creates such streams in response to the messages `asStream`.
The difference between a Pattern and a Stream is similar to the difference between a score and a
performance of that score or a class and an instance of that class. All objects respond to this interface,
most by returning themselves. So most objects are patterns that define streams that are an infinite
sequence of the object and embed as singleton streams of that object returned once.

Patterns are defined in terms of other Patterns rather than in terms of specific
values. This allows a Pattern of arbitrary complexity to be substituted for a
single value anywhere within a Pattern definition. A comparison between a Stream

definition and a Pattern will help illustrate the usefulness of Patterns.

example 1 - Pseq vs. Routine

The Pattern class **Pseq(array, repetitions)** defines a Pattern that will create a Stream that iterates
an array. The class **Routine(func, stackSize)** defines a single Stream, the function that runs within
that stream is defined to perform the array iteration.

Below a stream is created with **Pseq** and an `asStream`message and an identical stream is created directly using **Routine**.

```
// a Routine vs a Pattern
(

a = [-100,00,300,400];     // the array to iterate

p = Pseq(a);       // make the Pattern
q = p.asStream;      // have the Pattern make a Stream
r = Routine({ a.do({ arg v; v.yield}) }) ;  // make the Stream directly

5.do({ Post « Char.tab « r.next « " " « q.next « Char.nl; });
)
```

example 2 - Nesting patterns

In example 1, there is little difference between using Pseq and Routine. But Pseq actually
iterates its array as a collection of *patterns to be embedded*, allowing another Pseq to replace any
of the values in the array. The Routine, on the other hand, needs to be completely redefined.

```
(
var routinesA;
a = [3, Pseq([-100,00,300,400]), Pseq([-100,00,300,400].reverse) ];
routinesA = [[3], [-100,00,300,400], [-100,00,300,400].reverse];
p = Pseq(a);
q = p.asStream;
```

```
r = Routine({
routinesA.do({ arg v;
v.do({ arg i; i.yield})
}) ;
});
10.do({ Post « Char.tab « r.next « " " « q.next « Char.nl; });
)
```

example 3 - Stream-embedInStream

The message `embedInStream` is what allows Patterns to do this kind of nesting. Most objects
(such as the number 3 below) respond to `embedInStream`by yielding themselves once and returning.
Streams respond to embedInStream by iterating themselves to completion, effectively "taking over" the
calling stream for that time.

A Routine can perform a pattern simply by replacing calls to `yield` with calls to `embedInStream`.

```
(
a = [3, Pseq([-100,00,300,400]), Pseq([-100,00,300,400].reverse) ];

r = Routine({ a.do({ arg v; v.embedInStream}) }) ;
p = Pseq(a);
q = p.asStream;
10.do({ Post « Char.tab « r.next « " " « q.next « Char.nl; });
)
```

Of course, there is no concise way to *define* this stream without using Pseq.

note: For reasons of efficiency, the implementation of embedInStream assumes that it is called from
within a Routine. Consequently, embedInStream should never be called from within the function that
defines a FuncStream or a Pfunc (the pattern that creates FuncStreams).

Event Patterns

An Event is a Environment with a 'play' method. Typically, an Event consists of a collection of key/value pairs that determine what the play method actually does. The values may be any object including functions defined in terms of other named attributes.
Changing those values can generate a succession of sounds sometimes called 'music'...
The pattern **Pbind** connects specific patterns with specific names. Consult its help page for
details.

.....

A Summary of Pattern classes

Below are brief examples for most of the classes derived from Pattern. These examples all rely on the patterns assigned to the Interpreter variable p, q, and r in the first block of code.

```
(
SynthDef(\cfstring1.postln, { arg i_out, freq = 360, gate = 1, pan, amp=0.1;
var out, eg, fc, osc, a, b, w;
fc = LinExp.kr(LFNoise1.kr(Rand(0.25,0.4)), -1,1,500,2000);
osc = Mix.fill(8, {LFSaw.ar(freq * [Rand(0.99,1.01),Rand(0.99,1.01)], 0, amp) }).distort * 0.2;
eg = EnvGen.kr(Env.asr(1,1,1), gate, doneAction:2);
out = eg * RLPF.ar(osc, fc, 0.1);
#a, b = out;
Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+0.3])));
}).store;

SynthDef "sinegrain2"
{ arg out=0, freq=440, sustain=0.05, pan;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.3), doneAction:2);
Out.ar(out, Pan2.ar(SinOsc.ar(freq, 0, env), pan))
}).store;

p = Pbind(
[\degree, \dur], Pseq([[0,0.1],[2,0.1],[3,0.1],[4,0.1],[5,0.8]],1),
 \amp        \octave     \instrument  \cfstring1  \mtranspose
```

```
    Pbindf    \instrument  \default

r = Pset(\freq, Pseq([500, 600, 700], 2), q);

)
```

// EVENT PATTERNS - patterns that generate or require event streams

// Pbind(ArrayOfPatternPairs)

```
   p = Pbind
[\degree, \dur], Pseq([[0,0.1],[2,0.1],[3,0.1],[4,0.1],[5,0.8]],1),
  \amp        \octave     \instrument  \cfstring1  \mtranspose

p.play;
```

//Ppar(arrayOfPatterns, repeats) - play in parallel

```
Ppar([Pseq([p],4),Pseq([Pbindf(q,\ctranspose, -24)],5)]).play
```

//Ptpar(arrayOfTimePatternPairs, repeats) - play in parallel at different times

```
Ptpar([1,Pseq([p],4),0, Pseq([Pbindf(q,\ctranspose, -24)],5)]).play
```

// Pbindf(pattern, ArrayOfNamePatternPairs)

```
q = Pbindf(p, \instrument, \default );
q.play;
```

//Pfset(function, pattern)

// function constructs an event that is passed to the pattern.asStream

```
Pfset({  freq = Pseq([500, 600, 700], 2).asStream }, q).play;
```

//Pset(name, valPattern, pattern)

// set one field of the event on an event by event basis (**Pmul**, **Padd** are similar)

```
Pset(\freq, Pseq([500, 600, 700], 2), q).play;
```

//Psetp(name, valPattern, pattern)

// set once for each iteration of the pattern (**Pmulp**, **Paddp** are similar)

```
r = Pset(\freq, Pseq([500, 600, 700], 2), q);

Psetp(\legato, Pseq([0.01, 1.1],inf), r).play;
```

//Psetpre(name, valPattern, pattern)

// set before passing the event to the pattern (**Pmulpre**, **Paddpre** are similar)

```
r = Psetpre(\freq, Pseq([500, 600, 700], 2), q);

Psetp(\legato, Pseq([0.01, 1.1],inf), r).play;
```

//Pstretch(valPattern, pattern)

// stretches durations after

```
r = Psetpre(\freq, Pseq([500, 600, 700], 2), q);

 Pstretch(Pn(Env([0.5,2,0.5], [10,10]))

 Pset(\stretch, Pn(Env([0.5,2,0.5], [10,10]) ), Pn(r)).play
```

//Pstretchp(valPattern, pattern)

// stretches durations after

```
r = Psetpre(\freq, Pseq([500, 600, 700], 2), q);

 Pstretchp(Pn(Env([0.5,2,0.5], [10,10]))
```

// Pfindur(duration, pattern) - play pattern for duration

```
Pfindur(2,Pn(q,inf)).play;
```

// PfadeIn(pattern, fadeTime, holdTime, tolerance)

```
PfadeIn(Pn(q), 3, 0).play(quant:  0);
```

// PfadeOut(pattern, fadeTime, holdTime, tolerance)

```
PfadeOut(Pn(q), 3, 0).play(quant:  0);
```

// Psync(pattern, quantization, dur, tolerance)
// pattern is played for dur seconds (within tolerance), then a rest is played so the next pattern

```
Pn(Psync(
Pbind( \dur, Pwhite(0.2,0.5).round(0.2),
\db, Pseq([-10,-15,-15,-15,-15,-15,-30])
), 2,3
)).play
```

//Plag(duration, pattern)

```
Ppar([Plag(1.2,Pn(p,4)),Pn(Pbindf(q,\ctranspose, -24),5)]).play
```

// GENERAL PATTERNS that work with both event and value streams

//Ptrace(pattern, key, printStream) - print the contents of a pattern

```
r = Psetpre(\freq, Pseq([500, 600, 700], 2), q);

Ptrace(r).play;
 Ptrace(r, \freq

(
{ var printStream;
printStream = CollStream.new;
 Pseq([Ptrace(r, \freq, printStream), Pfunc({printStream.collection.dump; nil }) ]).play;

}.value;
)
```

//Pseed(seed, pattern) - set the seed of the random number generator
// to force repetion of pseudo-random patterns

```
Pn(Pseed(44, Pbindf(q,\ctranspose,Pbrown(-3.0,3.0, 10) ) ) ).play;
```

//Proutine(function) - on exit, the function must return the last value returned by yield
// (otherwise, the pattern cannot be reliably manipulated by other patterns)

```
Proutine({ arg inval;
inval = p.embedInStream(inval);
inval = Event.silent(4).yield;
inval = q.embedInStream(inval);
inval = r.embedInStream(inval);
inval;
}).play
```

//Pfunc(function) - the function should not have calls to embedInStream, use Proutine instead.

```
Pn(Pbindf(q,\legato, Pfunc({ arg inval; if (inval.at(\degree)== 5)  {4} {0.2}; })) ).play
```

// the following patterns control the sequencing and repetition of other patterns

//Pseq(arrayOfPatterns, repeats) - play as a sequence

```
Pseq([Pseq([p],4),Pseq([Pbindf(q,\ctranspose, -24)],5)]).play
```

//Pser(arrayOfPatterns, num) - play num patterns from the arrayOfPatterns

```
Pser([p,q,r],5).play
```

//Place(arrayOfPatterns, repeats) - similar to Pseq
// but array elements that are themselves arrays are iterated
// embedding the first element on the first repetition, second on the second, etc

```
Place([[p,q,r],q,r],5).play
```

// Pn(pattern, patternRepetitions) - repeat the pattern n times

```
Pn(p,2).play;
```

// Pfin(eventcount, pattern) - play n events from the pattern

```
Pfin(12,Pn(p,inf)).play;
```

// Pstutter(eventRepetitions, pattern) - repeat each event from the pattern n times

```
Pstutter(4,q).play
```

//Pwhile(function, pattern)

```
Pwhile({coin(0.5).postln;}, q).play
```

// Pswitch(patternList, selectPattern) - when a pattern ends, switch according to select

```
Pswitch([p,q,r], Pwhite(0,100)).play
```

// Pswitch1(patternList, selectPattern) - on each event switch according to select

```
Pn(Pswitch1([p,q,r], Pwhite(0,2))).play
```

// Prand(patternList, repeats) - random selection from list

```
Prand([p,q,r],  inf).play
```

// Pxrand(patternList, repeats) - random selection from list without repeats

```
Pxrand([p,q,r],  inf).play
```

// Pwrand(patternList, weights, repeats) - weighted random selection from list

```
Pwrand([p,q,r], #[1, 3, 5].normalizeSum, inf).play
```

// Pwalk(patternList, stepPattern, directionPattern) - walk through a list of patterns

```
Pwalk([p,q,r], 1, Pseq([-1,1], inf)).play
```

// Pslide(list, repeats, length, step, start)

```
Pbind(\degree, Pslide(#[1, 2, 3, 4, 5], inf, 3, 1, 0), \dur,0.2).play
```

// Pshuf(patternList, repeats) - random selection from list

```
Pn(Pshuf([p,q,r,r,p])).play
```

// Ptuple(list, repeats)

```
Pbind(\degree,Ptuple([Pwhite(1,-6), Pbrown(8,15,2)]),
\dur, Pfunc({ arg ev; ev.at(\degree).last/80 round:  0.1}),
\db, Pfunc({ if (coin(0.8)) {-25} {-20} })
).play
```

// the following patterns can alter the values returned by other patterns

//Pcollect(function, pattern)

```
Pcollect({ arg inval; inval.use({  freq = 1000.rand }); inval}, q).play
```

//Pselect(function, pattern)

```
Pselect({ arg inval; inval.at(\degree) != 0 }, q).play(quant:  0)
```

//Preject(function, pattern)

```
Preject({ arg inval; inval.at(\degree) != 0 }, q).play(quant:  0)
```

//Ppatmod(pattern, function, repeats) -
// function receives the current pattern as an argument and returns the next pattern to be played

```
Ppatmod(p, { arg oldPat; [p,q,r].choose }, inf).play
```

// VALUE PATTERNS: these patterns define or act on streams of numbers

// Env as a pattern

```
Pbindf(Pn(q,inf),\ctranspose, Pn(Env.linen(3,0,0.3,20),inf) ).play;
```

// Pwhite(lo, hi, length)

```
Pbindf(Pn(q,inf),\ctranspose,Pwhite(-3.0,3.0) ).play;
```

// Pbrown(lo, hi, step, length)

```
Pbindf(Pn(q,inf),\ctranspose,Pbrown(-3.0,3.0, 2) ).play;
```

// Pseries(start,step, length)

```
Pbindf(Pn(q,inf),\ctranspose,Pseries(0,0.1,10) ).play;
```

// Pgeom(start,step, length)

```
Pbindf(Pn(q,inf),\ctranspose,Pgeom(1,1.2,20) ).play;
```

// Pwrap(pattern,lo, hi)

```
Pbind(\note, Pwrap(Pwhite(0, 128), 10, 20).round(2), \dur, 0.05).play;
```

// PdegreeToKey(pattern, scale, stepsPerOctave)
// this reimplements part of pitchEvent (see Event)

```
Pbindf(Pn(q,inf),\note,PdegreeToKey(Pbrown(-8, 8, 2), [0,2,4,5,7,9,11]) ).play;
```

// Prewrite(pattern, dict, levels) - see help page for details.
// (notice use of Env to define a chord progression of sorts...

```
Pbind(\degree,
Prewrite(0, ( 0:  #[2,0],
1:  #[0,0,1],
2:  #[1,0,1]
), 4
) + Pn(Env([4,0,1,4,3,4], [6.4,6.4,6.4,6.4,6.4],'step')),
\dur, 0.2).play
```

// PdurStutter(repetitionPattern, patternOfDurations) -

```
Pbindf(Pn(q), \dur, PdurStutter(
Pseq(#[1,1,1,1,1,2,2,2,2,2,3,4,5,7,15],inf),
Pseq(#[0.5],inf)
)
).play;
```

// Pstep2add(pat1, pat2)
// Pstep3add(pat1, pat2, pat3)
// PstepNadd(pat1,pat2,...)
// PstepNfunc(function, patternArray)
// combine multiple patterns with depth first traversal

```
   Pbind
  \octave
  \degree  PstepNadd
Pseq([1, 2, 3]),
Pseq([0, -2, [1, 3], -5]),
Pshuf([1, 0, 3, 0], 2)
),
  \dur  PstepNadd
Pseq([1, 0, 0, 1], 2),
Pshuf([1, 1, 2, 1], 2)
).loop * (1/8),
\legato, Pn(Pshuf([0.2, 0.2, 0.2, 0.5, 0.5, 1.6, 1.4], 4), inf),
\scale, #[0, 1, 3, 4, 5, 7, 8]
).play;
```

ID: 396

PatternConductor

superclass: Object

PatternConductor provides a simple interactive control (supporting play, pause, resume, stop)
for playing pattern, much like Pattern-play. However, PatternConductor creates its own clock and
directly controls the release of sounding notes as well as their initiation by the pattern.

Class Methods

***new(pattern, event, quant)**

Instance Methods

tempo_(tempo)
Sets the tempo of the PatternConductor
play
Play the pattern. A TempoClock is created, its tempo is set to the PatternConductor tempo,
and the pattern is played using that clock.if quant is non-zero, this is synchronized with TempoClock.default at the specified quantization.
pause(pauseTempo = 0.000001)
Pause the pattern, sustaining notes indefinitely.
a subsequent **resume** will return to the original tempo (so the notes will end as scheduled).
a subsequent **play** will cut-off any sounding notes and resume play at the original tempo.
stop(stopTempo) can cut-off or shorten sounding notes, depending on the value of tempo.
If stopTempo is nil, all notes are cut-off immediately. Otherwise, notes end at the specified tempo.

example:

```
(
// a pattern with long notes
p = Pbind(
\freq, Pwhite(0,log(32)).exp.round(1) * 36.midicps,
```

```
\detune, Pfunc({ | ev | ev[\freq]  * rand(0.01) }),
\sustain, Pwhite(log(0.1), log(20)).exp,
\dur, Prand([0.1,0.1,0.1,0.1,0.2,1,2],inf),
\db, Pstep(Pseq([-20,-30,-25,-30], inf),0.2)
);


// unrelated cluster pattern running on TempoClock.default
Pbind(\dur,2, \midinote, Pseq([(48..60)],20), \db, -30).play;

// make a conductor
a = PatternConductor(p, quant: 2);
a.play;

)
(
// now try some interactive control options line by line:
a.quant = 0;
a.pause
a.resume
a.stop
a.play
a.pause
a.play
a.stop(1000)
)
```

ID: 397

Patterns in bold are included in Pattern tests
Patterns underlined have help pages
Patterns italicized have been tested but not included in the Pattern tests page.

[
FilterPattern
[
PfadeIn
[**PfadeOut**]
Pavaroh
PdegreeToKey
Pseed
Prewrite
Ptrace
Pwrap
Pstutter
[**PdurStutter**]
Pbindf
Plag
Pconst
Psync
Pfindur
Pfin
Pplayer
Pstretch
[**Pstretchp**]
Pset
[
Psetp
[Pmulp Paddp]
Pmul
Padd
]
Psetpre
[Pmulpre Paddpre]
FuncFilterPattern
[**Pwhile Pfset PrejectPselectPcollect**]
Pn

[Ploop]
]
Phid
Pdict
Pdefn
[
Tdef
[**Pdef**]
]
Pswitch
[**Pswitch1**]
ListPattern
[
Pwalk
Pslide
Ptuple
Ppar
[**Ptpar**]
Pdfsm
Pfsm
Pwrand
Pxrand
Prand
Pshuf
Pseq
[*Ppatlace* **Place** **Pser**]
]
Pindex
Ppatmod
Plazy
Pstatbalnorm
Pstatbal
PstepNfunc
[*PstepNadd*]
Pstep3add
Pstep2add
Pwhite
Pbrown
Pgeom
Pseries

Pbind

Pevent

_Pnaryop

Pbinop

Punop

_Pfuncn

Prout

Proutine

Pfunc

```
The following lists all the Pattern classes where
embedInStream uses the default Pattern-embedInStream.
These create a new stream whenever embedded.

(
var look;

look = { arg class;
class.subclasses.do({arg class;
if (class.findMethod(\embedInStream).isNil) {
class.postln;
look.value(class);
};
});
};

look.value(Pattern);
)



// base classes support both approaches:
FilterPattern
FuncFilterPattern
ListPattern

// classes not reimplemented
Ptrace
Pconst
```

Pdict

Pstatbalnorm

Pstatbal

ID: 398

Pavaroh

superclass: FilterPattern

basic classical indian scale pattern
allowing to apply an ascending scale (**aroh**) and a descending scale (**avaroh**)

Pavaroh(keypattern, aroh, avaroh, stepsPerOctave)

note that no special pakads (movements) or vakras (twists) are applied.

the pakad is often a natural consequence of the notes of arohana / avarohana (ascending and descending structures). This is the purpose of this pattern

```
(
Pbind \note  Pavaroh
Pseq([1, 2, 3, 2, 5, 4, 3, 4, 2, 1], 2),
#[0, 2, 3, 6, 7, 9],
#[0, 1, 3, 7, 8, 11]
),
\dur, 0.25
).play;
)


//___indian video game (1)
(
SynthDef("ivg", { arg out, freq=900, pan;
var trig, snd;
trig = Impulse.kr(LFClipNoise.kr(4, 3, LFClipNoise.kr(0.2, 2, 7)));
snd = RLPF.ar(
SinOsc.ar(freq, 0, Decay.kr(trig, 1.8)).distort
, 554, 0.3)
* 0.1;
Out.ar(out, Pan2.ar(snd, pan))
```

```
}).send(s);
)

(
var aroh, avaroh, synth, str, pat;

        //gandhari raga.   vadi:  dha (7) samvadi:  ga (3)

aroh = #[0, 2, 5, 7, 10];
avaroh =  #[0, 1, 3, 5, 7, 9, 10];

synth = Synth.head(s, \ivg);
pat = Prand([0, 2, 3, 4, 2, 1, 0, -1, -2], inf);
str = Pavaroh(pat, aroh, avaroh).asStream;
Routine({

loop({
synth.set(\freq, midicps(str.next + 60) );
rrand([0.1,1.0].choose, 0.5).wait;
});

}).play;

)
```

ID: 399

Pbind

superclass: Pattern

Pbind(pattern pairs)

The class Pbind provides a bridge between value patterns and event patterns. It binds symbols in each event to values obtained from a pattern. Pbind takes arguments in pairs, the first of a pair being a Symbol and the second being a value Pattern. Any object can act as a Pattern, so constants can be used as values.

The Pbind stream returns nil whenever the first one of its streams ends or if nil is passed in.

```
// example:

a = Pbind(\x, 77, \y, Pseq([1, 2, 3]));
x = a.asStream;
4.do { x.next(Event.new).postln };

a = Pbind(\x, 77, \y, Pseq([1, 2, 3]));
x = a.asStream;
        // this returns nil.
```

An **event stream** is created for a Pattern by sending it the *asEventStream* message. The *asEventStream* message takes an Event as an argument. This event is copied for each call to next to pass down and back up the tree of patterns so that each pattern can modify the event. What Pbind does is put the values for its symbols into the event, possibly overwriting previous bindings to those symbols.

This uses the default event. In the next example we will supply our own event (synth function).

```
// example:
(
Pbind
```

```
\degree, Pseq([1,3,5,7], inf),
\dur, 0.125,
 \octave
 \root
).play
)
```

To use another than the default SynthDef, we need to read the synth description library so that event know's what kind of arguments there are in each SynthDef. Use **.store** (instead of load or send) to create a Synth Description (SynthDesc).

Special **control name conventions** which should be used so that the SynthDef works with the default system:

out output bus index
gate envelope gate (not level!) - should default to 1.0
amp synth amplitude - should default to 0.1

The control names of the synth definition should match the keys. The default event scheme omplements a couple of useful parameter transformations in addition to that.

sustain envelope duration (not dur!) - should default to 1.0. **legato** and**dur** values are translated to **sustain**
freq some frequency input - often defaults to 440. **degree, note** and**midinote** values are translated to **freq**.
bufnum buffer number
pan panning position

```
(
SynthDef(\cfstring1, { arg i_out, freq = 360, gate = 1, pan, amp=0.1;
var out, eg, fc, osc, a, b, w;
fc = LinExp.kr(LFNoise1.kr(Rand(0.25,0.4)), -1,1,500,2000);
osc = Mix.fill(8, {LFSaw.ar(freq * [Rand(0.99,1.01),Rand(0.99,1.01)], 0, amp) }).distort * 0.2;
eg = EnvGen.kr(Env.asr(1,1,1), gate, doneAction:2);
out = eg * RLPF.ar(osc, fc, 0.1);
#a, b = out;
```

```
Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+0.3])));
}).store;
)



(
e = Pbind(
\degree, Pwhite(0,12),
\dur, 0.2,
 \instrument  \cfstring1
        // returns an EventStream
)

// the event stream's stream can be changed while it is running:

(
e.stream = Pbind(
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, Prand([0.2,0.4,0.8],inf),
\amp, 0.05, \octave, 5,
   \instrument  \cfstring1  \ctranspose
).asStream;
)

(
e.stream = Pbind(
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, Prand([0.2,0.4,0.8],inf),
\amp, 0.05, \octave, 5,
   \instrument  \cfstring1  \ctranspose
).asStream;
)

(
e.stream = Pbind(
\degree, Pxrand([0,1,2,4,6,3,5,7,8],inf),
\dur, Prand([0.2,0.4,0.8],inf), \amp, 0.05,
   \octave     \instrument  \cfstring1
).asStream;
)
```

```
// pairs of names can be used to group several parameters

(
e.stream = Pbind(
 [\degree  \dur   Pseq
Pseq([[0,0.1],[2,0.1],[3,0.1],[4,0.1],[5,0.8]],2),
Ptuple([Pxrand([6,7,8,9],4), 0.4]),
Ptuple([Pseq([9,8,7,6,5,4,3,2]), 0.2])
],inf),
 \amp        \octave     \instrument  \cfstring1  \mtranspose
)

(
e.stream = Pbind(
 [\degree  \dur   Pseq
Pseq([[0,0.1],[2,0.1],[3,0.1],[4,0.1],[5,0.8]],2),
Ptuple([Pxrand([6,7,8,9],4), 0.4]),
Ptuple([Pseq([9,8,7,6,5,4,3,2]), 0.2])
],inf),
 \amp        \octave     \instrument  \cfstring1  \mtranspose
)




// play control:

        // keeps playing, but replaces notes with rests

e.unmute;

          // reset the stream.
          // reset the stream.
          // reset the stream.
          // reset the stream.

          // will resume where paused.

e.play;
```

```
        // will reset before resume.

e.play;
```

Another example with a different SynthDef:

```
(
SynthDef(\berlinb, { arg out=0, freq = 80, amp = 0.01, pan=0, gate=1;
var synth, env;
env = Decay2.kr(gate, 0.05, 8, 0.0003);
synth = RLPF.ar(
LFPulse.ar(freq, 0, SinOsc.kr(0.12,[0,0.5pi],0.48,0.5)),
freq * SinOsc.kr(0.21,0,18,20),
0.07
);
#a, b = synth*env;
DetectSilence.ar(a, 0.1, doneAction:  2);
Out.ar(out, Mix.ar(PanAz.ar(4, [a,b], [pan, pan+1])));
}).store;
)

(
f = Pbind(
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
  \dur       \octave     \instrument  \berlinb
).play;
)

(
f.stream = Pbind(
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, 0.5, \octave, [2,1],
  \instrument  \berlinb
\pan, Pfunc({1.0.rand2})
).asStream;
)
```

Additional arguments

Here is an example with more bindings; Here we have added a filter with cutoff and resonance arguments.
You will need to hit command '.' before executing the next few pbind ex. without having them stack up.
also, due to the synthdef's and synthdeclib, if the server is shut down you will have to reload the
synthdef and re-read the synthdesclib.

```
(
SynthDef("acid", { arg out, freq = 1000, gate = 1, pan = 1, cut = 4000, rez = 0.8, amp = 1;
Out.ar(out,
Pan2.ar(
RLPF.ar(
Pulse.ar(freq,0.05),
cut, rez),
pan) * EnvGen.kr(Env.linen(0.01, 1, 0.3), gate, amp, doneAction:2);
)
}).store;
)

(
Pbind(\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],inf), \root,-12,
\degree,Pseq([0,3,5,7,9,11,5,1],inf), \pan,Pfunc({1.0.rand2}),
\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}), \amp,0.2).play;
)
```

The **ListPatterns** can be put around Event Streams to create sequences of Event Streams.

```
(
Pseq
Pbind(\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],4), \root,-24,
\degree,Pseq([0,3,5,7,9,11,5,1],inf), \pan,Pfunc({1.0.rand2}),
\cut,Pxrand([1000,500,2000,300],inf),\rez,Pfunc({0.7.rand +0.3}), \amp,0.2),

Pbind(\instrument,\acid, \dur,Pseq([0.25],6), \root,-24, \degree,Pseq([18,17,11,9],inf),
```

```
\pan,Pfunc({1.0.rand2}),\cut,1500, \rez,Pfunc({0.7.rand +0.3}), \amp,0.16)

],inf).play;
)
```

'**Pseq**' in the above ex. can be any pattern object:

```
(
Prand
Pbind(\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],4), \root,-24,
\degree,Pseq([0,3,5,7,9,11,5,1],inf),\pan,Pfunc({1.0.rand2}),
\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}),
\amp,0.2),

Pbind(\instrument,\acid, \dur,Pseq([0.25],6), \root,-24, \degree,Pseq([18,17,11,9],inf),   \pan,Pfunc({1.0.rand2}),\cut,1500
\rez,Pfunc({0.7.rand +0.3}), \amp,0.16)

],inf).play;
)
```

Multichannel Expansion.
If we supply an array for any argument, the synth node will automatically replicate to handle the additional arguments.
When we give the 'root' argument an array, we should hear a chord....

```
(
Pbind
\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],inf),
\root,[-24,-17],
\degree,Pseq([0,3,5,7,9,11,5,1],inf),
\pan,Pfunc({1.0.rand2}),\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}),
\amp,0.2).play;

)
```

Using [**Pdef**] (JITLib) makes it easy to replace patterns on the fly:

```
(
Pdef \buckyball
)

(
Pdef(\buckyball, Pbind(\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],inf), \root,[-24,-17],
\degree,Pseq([0,3,5,7,9,11,[5,17],1],inf), \pan,Pfunc({[1.0.rand2,1.0.rand2]}),
\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}), \amp,[0.15,0.22]));
)
(
Pdef(\buckyball, Pbind(\instrument,\acid, \dur,Pseq([0.25,0.5,0.25],inf), \root,[-24,-17],
\degree,Pseq([0b,3b,5b,7b,9b,11b,5b,0b],inf), \pan,Pfunc({1.0.rand2}),  //notice the flats
\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}), \amp,0.2));
)

//stop the Pdef
Pdef \buckyball

//start the Pdef
Pdef(\buckyball).resume;

//removing the Pdef
Pdef.remove(\buckyball);
```

Sending to effects.

Assignment to effect processors can be achieved by setting the 'out' argument to the desired
efx's input bus. The effect Synth must also be created. Synth.new is one way of doing this.

```
(
//efx synthdef- dig the timing on the delay and the pbind.   :-P
SynthDef("pbindefx", { arg out, in, time1=0.25, time2=0.5;   var audio, efx;
audio = In.ar([20,21],2);
efx=CombN.ar(audio, 0.5, [time1,time2], 10, 1, audio);   Out.ar(out, efx);
}).load(s);

//create efx synth
a = Synth.after(1, "pbindefx");
```

```
//if you don't like the beats change to 0.4,0.24
//a.set(\time1,0.4, \time2,0.24);

SynthDef("acid", { arg out, freq = 1000, gate = 1, pan = 0, cut = 4000, rez = 0.8, amp = 1;
Out.ar(out,
Pan2.ar(
RLPF.ar(
Pulse.ar(freq,0.05),
cut, rez),
pan) * EnvGen.kr(Env.linen(0.02, 1, 0.3), gate, amp, doneAction:2);
)
}).load(s);

SynthDescLib.global.read;

)


(
Pbind(\instrument,\acid, \out, 20, \dur,Pseq([0.25,0.5,0.25],inf), \root,[-24,-17],
\degree,Pseq([0,3,5,7,9,11,5,1],inf), \pan,Pfunc({1.0.rand2}),
\cut,Pxrand([1000,500,2000,300],inf), \rez,Pfunc({0.7.rand +0.3}), \amp,0.12).play;
)
```

//UGens as Event values.

```
//The following example creates unit generators instead of scalar values for
//the values bound to the arguments. This shows that you can use patterns
//to dynamically build your patch. Score data is not limited to scalar values.
//This example can generate 36 different patches: 3 instruments * 3 freqs
//* 2 amps * 2 pans
//
//
//I don't know if this is possible in sc3.
////(
//SynthDef(\cfstring1.postln, { arg i_out, freq = 360, gate = 1, pan, amp=0.1;
// var out, eg, fc, osc, a, b, w;
```

```
// fc = LinExp.kr(LFNoise1.kr(Rand(0.25,0.4)), -1,1,500,2000);
// osc = Mix.fill(8, { LFSaw.ar(freq * [Rand(0.99,1.01),Rand(0.99,1.01)], 0, amp) }).distort * 0.2;
// eg = EnvGen.kr(Env.asr(0.1,1,1), gate, doneAction:2);
// out = eg * RLPF.ar(osc, fc, 0.1);
// #a, b = out;
// Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+0.3])));
//}).load(s);
//
//SynthDef(\berlinb, { arg out=0, freq = 80, amp = 0.01, pan=0, gate=1;
// var synth, env;
// env = Decay2.kr(gate, 0.05, 8, 0.0003);
// synth = RLPF.ar(
//  LFPulse.ar(freq, 0, SinOsc.kr(0.12,[0,0.5pi],0.48,0.5)),
//  freq * SinOsc.kr(0.21,0,18,20),
//  0.07
// );
// #a, b = synth*env;
// DetectSilence.ar(a, 0.1, doneAction:  2);
// Out.ar(out, Mix.ar(PanAz.ar(4, [a,b], [pan, pan+1])));
//}).load(s);
//
//SynthDef("acid", { arg out, freq = 1000, gate = 1, pan = 0,  amp = 0.3;
// Out.ar(out,
//  Pan2.ar(
//    Pulse.ar(freq*0.125,0.05),
//  pan) * EnvGen.kr(Env.linen(0.01, 1, 0.3), gate, amp, doneAction:2);
// )
// }).load(s);
//
//SynthDescLib.global.read;
//)
//
//(
//var a, b, c, pattern, stream;
//
//pattern = Pbind(
// \freq, Pfunc({Line.kr(40, 2000, 0.2)}),
//
// \amp, Pfunc({
//   [
```

```
//    { SinOsc.kr(20.0.rand, 0, 0.1, 0.1) },
//    { XLine.kr(exprand(0.002, 0.2), exprand(0.002, 0.2), 2.2) }
//   ].choose.value;
//  }),
// \pan, Pfunc({
//   [
//    { Line.kr(1.0.rand2, 1.0.rand2, 2.2)  },
//    { SinOsc.kr(4.0.rand) }
//   ].choose.value;
//  }),
// \instrument, Prand([ \cfstring1, \acid, \berlinb ], inf)
//);
//
//)
//
```

ID: 400

Pbus

superclass: FilterPattern

Pbus(pattern, dur, fadeTime)

Starts a new group and plays the pattern in this group, on a private bus.
The group and the bus is released when the stream has ended.

This is useful in order to isolate a Pfx.

dur delay to allow inner patterns to decay.
fadeTime fading out the inner pattern after dur over this time
numChannels number of channels of the bus (should match the synthdef) **default: 2**
rate bus rate (default: 'audio')

Example:

```
(
SynthDef(\echo, { arg out=0, maxdtime=0.2, dtime=0.2, decay=2, gate=1;
var env, in;
env = Linen.kr(gate, 0.05, 1, 0.1, 2);
in = In.ar(out, 2);
XOut.ar(out, env, CombL.ar(in * env, maxdtime, dtime, decay, 1, in));
}, [\ir, \ir, 0.1, 0.1, 0]).store;

SynthDef(\distort, { arg out=0, pregain=40, amp=0.2, gate=1;
var env;
env = Linen.kr(gate, 0.05, 1, 0.1, 2);
XOut.ar(out, env, (In.ar(out, 2) * pregain).distort * amp);
}, [\ir, 0.1, 0.1, 0]).store;
```

```
SynthDef(\wah, { arg out=0, gate=1;
var env, in;
env = Linen.kr(gate, 0.05, 1, 0.4, 2);
in = In.ar(out, 2);
XOut.ar(out, env, RLPF.ar(in, LinExp.kr(LFNoise1.kr(0.3), -1, 1, 200, 8000), 0.1).softclip * 0.8);
}, [\ir, 0]).store;
)

(
var p, q, r, o;
p = Pbind(\degree, Prand((0..7),12), \dur, 0.3, \legato, 0.2);

q = Pfx(p, \echo, \dtime, 0.2, \decay, 3);

r = Pfx(q, \distort, \pregain, 20, \amp, 0.25);

o = Pfx(r, \wah);

Ppar
 [p, q, r, o].collect(Pbus _    // play each in a different bus.
).play;
)



// compare to playing them together on one bus.
(
var p, q, r, o;
p = Pbind(\degree, Prand((0..7),12), \dur, 0.3, \legato, 0.2);

q = Pfx(p, \echo, \dtime, 0.2, \decay, 3);

r = Pfx(q, \distort, \pregain, 20, \amp, 0.25);

o = Pfx(r, \wah);

Ppar([p, q, r, o]).play;
)
```

ID: 401

Pchain pass values from stream to stream

superclass: Pattern

Pchain(pattern1, pattern2, ... patternN) *pattern1 <- pattern2 <- ...patternN*

pattern1, pattern2.. the patterns to be chained up.

values that the stream of **pattern2** produces are used as inval to the stream of **pattern1**.
Therefore pattern1 overrides (or filters) the output of pattern2, and so forth.
This is an equivalent to the composite pattern: *pattern1 <> pattern2 <> ... patternN*

<> pattern add another pattern to the chain

```
// examples

(
Pchain
Pbind(\detune, Pseq([-30, 0, 40], inf), \dur, Prand([0.2, 0.4], inf)),
Pbind(\degree, Pseq([1, 2, 3], inf), \dur, 1)
).trace.play;
)


// also events can be used directly:
(
Pchain
Pbind(\degree, Pseq([1, 2, 3], inf)),
(detune:  [0, 4])
```

```
).trace.play;
)

// compose some more complicated patterns:
(
var a, b;
a = Prand([
Pbind(\degree, Pseq([0, 1, 3, 5, 6])),
Pbind(\dur, Pshuf([0.4, 0.3, 0.3]), \degree, Pseq([3, -1]))
], inf);
b = Prand([
Pbind(\ctranspose, Pn(1, 4)),
Pbind(\mtranspose, Pn(2, 7))
], inf);
c = Prand([
Pbind(\detune, Pfuncn( { [0, 10.0].rand }, 5), \legato, 0.2, \dur, 0.2),
Pbind(\legato, Pseq([0.2, 0.5, 1.5], 2), \dur, 0.3)
], inf);
Pchain(a, b, c).trace.play;
)
```

pattern composition: pattern <> pattern <> pattern

```
// implicitly, the composition operator <> returns a Pchain when applied to a pattern.
// so that a <> b creates a Pchain (a, b).
// as seen above, in Pchain(a, b), a specifies (and overrides) b:  b is the input to a.

// the above example is equivalent to:

(Pbind(\degree, Pseq([1, 2, 3], inf)) <> (detune:  [0, 4])).trace.play;

(
a = Pbind(\degree, Pseq([1, 2, 3], inf), \dur, Prand([0.2, 0.4], inf));
b = Pbind(\detune, Pseq([-30, 0, [0, 40]], inf), \dur, 0.1);
c = b <> a;
```

```
        // see that the \dur key of a is overridden by b
)

// also value streams can be composed
(
a = Pfunc { | x|  x + 1.33 };
b = Pfunc { | x|  x * 3 };
c = Pseries(1, 2, inf);
)

// post some values from the composite streams:

t = (a <> b).asStream;
10.do { t.value(10).postln };

t = (a <> b <> c).asStream;
10.do { t.value(10).postln };

t = (b <> c <> a).asStream;
10.do { t.value(10).postln };
```

ID: 402

Pcollect

superclass: FuncFilterPattern

Pcollect(func, pattern)

modifies each value by passing it to the function

```
(
var a, b;
a = Pcollect({ arg item; item * 3 }, Pseq(#[1, 2, 3],inf));
x = a.asStream;
9.do({ x.next.postln; });
)
```

the message collect returns a Pcollect when passed to a pattern

```
(
var a, b;
a =  Pseq(#[1, 2, 3],inf).collect({ arg item; item * 3 });
a.postln;
x = a.asStream;
9.do({ x.next.postln; });
)
```

ID: 403

Pconst constrain the sum of a value pattern

superclass: FilterPattern

Pconst(sum, pattern, tolerance)

embeds elements of the **pattern** into the stream until the sum comes close enough to **sum**.
similar to **Pfindur**, but works with the value directly.

```
(
var a, b;
a = Pconst(5, Prand([1, 2, 0.5, 0.1], inf));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

Pconst used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)

(
Pn
 Pbind
\dur, Pconst(1, Prand([1, 0.02, 0.2], inf)),
  \instrument  \help_sinegrain
\degree, Pseries(0, 1, inf),
  \octave
```

```
)
).play;
)
```

ID: 404

PdegreeToKey

superclass: FilterPattern

PdegreeToKey(keypattern, scale, stepsPerOctave)

returns a series of notes derived from an index into a scale.
if the scale is a pattern, it streams the scales accordingly

keypattern integer index into the scale
scale an array
stepsPerOctave the number of steps per octave in the scale. The default is 12.

```
(
Pbind \note  PdegreeToKey
Pseq([1, 2, 3, 2, 5, 4, 3, 4, 2, 1], 2),
#[0, 2, 3, 6, 7, 9],
12
),
\dur, 0.25
).play;
)
```

```
(
var scales;
scales = #[[0, 2, 3, 6, 7, 9], [0, 1, 5, 6, 7, 9, 11], [0, 2, 3]];
Pbind \note  PdegreeToKey
Pseq([1, 2, 3, 2, 5, 4, 3, 4, 2, 1], 4),
Pstutter(3, Prand(scales, inf)),
12
),
\dur, 0.25
).play;
```

)

ID: 405

Pdfsm deterministic finite state machine

by ccos

superclass: ListPattern

deterministic finite state machine with signal input.

list - a list consisting of the stream which gives input signals to determine state transitions, and then dictionary entries, one for each state, mapping the destinattion state and yield streams to those input signals.
startState - an integer index for the state to start with. defaults to 0.
repeats - an integer giving the number of times the pattern should cycle.
a cycle ends when the **signal stream** ends or nil is given for the destination state to a signal value, see below. defaults to 1

more on the list -

[
signal stream - can be a stream of anythingwhich can serve as a key for an associative collection. integers, symbols, etc...
asStream is called on this for each repeat.
states - states should be IdentityDictionaries or some other associative collection
]

list syntax -

[
signal stream,
(// state 0,
signal value : [**destination state, return stream or pattern**],
signal value : [**destination state, return stream or pattern**]
),
... // state 1 ... N
]

any number of states can be given, and are indexed by the order in which they are given.
if the fsm is in state x and it receives a **signal value** y it looks up y in the state dictionary supplied for x, if there is no y entry, it looks for a \default entry and uses that.
the next state is then set to **destination state**, and the stream yielded is given by **return stream or pattern.**
that is unless the **destination state** is given as nil, or if a **destination state** is given for which you havenot
supplied a dictionary - in both cases the current cycle ends and any remaining repeats are executed.
if there is no **signal value** given for a particular signal, and no \default is supplied then upi will get a runtime error.

```
(
p = Pdfsm(
[
   Pseq   \foo \bar        // foobar signals
   ( // state 0
\foo :  [ 1, Pseq([ 0, 1 ], 2 ) ]
),
   ( // state 1
\bar :  [ 0, 3 ]
)
],
0,
2
).asStream;

11.do({ p.next.postln });
)

(
SynthDef 'Help-Pdfsm1'
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar([freq, freq + 0.1.rand2], 0, env))
}).send(s);
)
```

```
(
var p;
p = Pdfsm(
[
Prand([0,1,2],inf),  // signalStream

  IdentityDictionary   // state 0
0 -> [ 2, Pseq([67,68,69], 2) ],
1 -> [ 0, 66 ],
2 -> [ 1, 65 ]
],
  IdentityDictionary   // state 1
1 -> [ 1, Pseq([69,68,67],2) ],
\default -> [ 0, 70 ]
],
  IdentityDictionary
0 -> [ 0, 71 ],
1 -> [ 0, 72 ],
   2 -> [ nil   // signalStream is infinitely long,
      // so the fsm only ends because of this nil
      // 2 -> [nil, nil] is also fine
]
],
 1,      // startState
 1      // repeats
).asStream;

Routine
var freq;
while({ (freq = p.next.postln).notNil },{
Synth('Help-Pdfsm1', [ \freq, freq.midicps ]);
0.1.wait;
})
}).play;
)

(
SynthDef 'Help-Pdfsm2'
{ arg freq, gate=1;
var n=8, env, osc;
```

```
env = Linen.kr( gate, 0.01, 1, 0.03, 2 );
osc = {Mix.fill( n, { arg i;
FSinOsc.ar(freq + Rand(-2.0,2.0), Rand(0, 0.05pi)) ring4:
FSinOsc.ar(freq * (i+1));
})}.dup * FSinOsc.kr(Rand(1.5,4.5),{Rand(-0.1pi,0.1pi)}.dup,0.6,env*0.4);
Out.ar(0, env * osc / (n*4)  )
}).load(s);
 SynthDescLib              // needed for the Pbinds below
)

(
var n=3, base, penult;

base = [3,4,4,0];

for( 1, n, { arg i;
penult = Pbind( \degree, Pshuf(base - (i*5), 2), \dur, Pseq([0.2],2) );
 Pset
  \instrument  'Help-Pdfsm2'
  Pdfsm
[
    Pseq   // signalStream
Pn(1,22 + i),
Pn(0,4),
Pn(1,8),
Pseq([
Pn(0,3),
Prand([0,1],8),
Pn(1,8)
], 3 ),
Pn(2,2)
], 1 ),
    (  // state 0
0 :  [ 0, Pbind( \degree, Pseq(base - i, 1), \dur, Pxrand([0.2,0.3],4) ) ],
1 :  [ 1, Pbind( \degree, Pseq(base.reverse - (i*2), 2), \dur, Pseq([0.2,0.21],1) ) ],
2 :  [ 2, penult ]
),
    (  // state 1
0 :  [ 0, Pbind( \degree, Pshuf(base * i.neg, 8), \dur, Pseq([0.08],8) ) ],
1 :  [ 0, Pbind( \degree, Pseq(base - (i*3),3+i), \dur, Pseq([0.11],3+i) ) ],
```

```
2 :  [ 2, penult ]
),
    (  // state 2
\default :  [ 2, Pbind( \degree, Prand(base - (i*7), 5), \dur, Prand([0.6,0.8],5) ) ]
)
],
   i%2   // startState
)
).play;
})
)
```

ID: 406

PdurStutter

PdurStutter(stutterPattern,floatPattern)

a filter pattern designed for a stream of float durations.
subdivides each duration by each stutter and yields that value stutter times.
a stutter of 0 will skip the duration value, a stutter of 1 yields the duration value unaffected.

```
(
    PdurStutter
Pseq(#[1,1,1,1,1,2,2,2,2,2,0,1,3,4,0],inf),
Pseq(#[0.5, 1, 2, 0.25,0.25],inf)
);
x = a.asStream;
100.do({ x.next.postln; });
)
```

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
(

    PdurStutter
Pseq(#[1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4],inf),
Pseq(#[0.5, 1, 2, 0.25,0.25],inf)
);
x = a.asStream;
Routine
loop({
   Synth.grain "help-sinegrain"   \freq
```

```
x.next.wait;
})
}).play(TempoClock.default);
)



(

    PdurStutter
Pseq(#[1,1,1,1,1,2,2,2,2,2,3,3,3,3,4,4,0,4,4],inf),
Pseq(#[0.5, 1, 2, 0.25,0.25],inf)
);
x = a.asStream;
Routine
loop({
   Synth.grain "help-sinegrain"   \freq
x.next.wait;
})
}).play(TempoClock.default);
)
```

Frequencies like being divided too.

```
(
    PdurStutter
Pseq(#[1,1,1,1,1,2,2,2,2,2,3,3,3,3,4,4,0,4,4],inf),
Pseq((80 + [ 0, 2, 3, 5, 7, 9, 10 ]).midicps ,inf)
);
x = a.asStream;
Routine
loop({
Synth.grain("help-sinegrain", [\freq, x.next.postln]);
0.25.wait
})
}).play(TempoClock.default);
)
```

-felix

ID: 407

Penvir use an environment when embedding the pattern in a stream

superclass: Pattern

Penvir(envir, pattern, independent)

envir an environment with objects to embed

pattern pattern or stream, ususally a Pfunc, Prout.

independent if true (default) streams can write to the environment without influencing other
streams created from this pattern.
if false, the streams write to a common namespace.

```
// examples:

(
x = (a:8);
    Penvir
x,
Pfunc {  a * 2 }
);

t = y.asStream;
)

t.next;



(
```

```
x = (a:8);
    Penvir
x,
Pfunc {  a =  a * 2 }
);

t = y.asStream;
z = y.asStream;
)

t.next;
t.next;
          // x stays unchanged
```

ID: 408

Pfin

superclass: FilterPattern

Pfin(count, pattern)

embeds count elements of the pattern into the stream

```
(
var a, b;
a = Pfin(5, Pseq(#[1, 2, 3],inf));
x = a.asStream;
9.do({ x.next.postln; });
)
```

Pfin used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)

(
c = Pn(Pfin({ rrand(3, 5)}, Pseq([1, 2, 3, 4, 5, 6],inf)*4+65),inf);
x = c.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
0.12.wait;
})
}).play;
```

)

ID: 409

Pfindur

superclass: FilterPattern

Pfindur(dur, pattern, tolerance)

embeds elements of the **pattern** into the stream until the duration comes close enough to **dur**.

```
(
var a, b;
a = Pfindur(5, Pbind(\dur, Prand([1, 2, 0.5, 0.1], inf)));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

Pfindur used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
```

```
(
var c;
c = Pbind(
\dur, Prand([1, 0.02, 0.2], inf),
 \instrument  \help_sinegrain
\degree, Pseries(0, 1, inf),
```

```
\octave, 6
);

Pn
Pfindur(1, c)
).play;
)
```

ID: 410

Pflow

superclass: FilterPattern

advances the substream according to a time pattern from moment of embedding in stream

Pflow(timepattern, pattern)

replaced by **Pstep(pattern, durpattern)**

ID: 411

Pfset create an environment to modify values in the incoming stream

superclass: FuncFilterPattern

Pfset(name, value, pattern)

```
(
var a, b;
a = Pfset({
legato = 0.3;
detune = rrand(0, 30);
}, Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

Pfset does not override incoming values:

```
(
var a, b;
a = Pfset({
dur = 0.3;
}, Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
```

```
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)


(
a = Pbind(\dur, 0.5, \instrument, \sinegrain, \x, Pfunc { rrand(500, 600) });
a = Pfset({  freq = {  x.postln * 2 };  legato = 3; },  a);
a.play;
)
```

ID: 412

Pfsm

superclass: ListPatterns

finite state machine: every state links to possible **next states (indices)**.
starting from one of the **entry states** one of these links is randomly
chosen used to get the next state. When an end state is reached, the stream ends.

Pfsm(list,repeats)

list:
[
[entry states],
item, [next states],
item, [next states],
...
end item (or nil), nil
]

next states: nil is terminal

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
```

```
(
a = Pfsm([
#[0,1],
67, #[0, 0, 3],
```

```
72, #[2],
73, #[0, 2],
Pseq([74, 75, 76, 77]), #[2, 3, 3],
  nil, nil
], inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.1.wait;
})
}).play;
)

(
Pfsm
#[5, 6, 7], // entry states

    //e1 (== state 0)
Pbind( \dur, Pseq([ 1/8, 3/8 ]), \midinote, Pseq([ 86, 75 ]) ),
     //#[1],  // as given in CMJ
     // my de-boredom-ated version..
#[1, 1, 1, 1, 1, 1, 1, 8],
    //e2 (== state 1)
Pbind( \dur, 1/2, \midinote, Pseq([ 69 ]) ),
#[0, 1],
    //e3 (== state 2)
Pbind( \dur, 1/3, \midinote, Pseq([ 55, 60, 66 ]) ),
#[0, 1, 2, 2, 2, 2, 3, 3, 3, 3],
    //e4 (== state 3)
Pbind( \dur, 1/4, \midinote, Pseq([ 81, 80, 77, 76 ]) ),
#[1, 4, 4, 4, 4],
    //e5 (== state 4)
Pbind( \dur, Pseq([1, 2/3, 2/3, 2/3, 1]), \midinote, Pseq([ \, 70, 70, 70, \]) ),
#[2, 3],
    //e6 (== state 5)
Pbind( \dur, 1/4, \midinote, Pseq([ 59, 61 ]) ),
#[0, 2, 4, 5, 5, 5, 5, 5, 5, 5],
    //e7 (== state 6)
Pbind( \dur, 1/4, \midinote, Pseq([ 87, 88 ], 2) ),
#[4, 4, 4, 4, 6, 6, 6, 7, 7, 7],
```

```
    //e8 (== state 7)
Pbind( \dur, 1, \midinote, Pseq([ 56 ]) ),
#[1, 3, 6, 6, 6],
    // terminal state
nil, nil
]).play;
)
```

ID: 413

Pfx

superclass: FilterPattern

Pfx(pattern, fxname, name, value, name, value, ...)

Puts an effect node on the tail of the current group and releases it when the contained pattern finishes. If a bus is given, it is used as an effect bus. Name value pairs are inserted into the event for starting the effect node. The effect parameters are set from the event.

Example:

```
(
SynthDef(\echo, { arg out=0, maxdtime=0.2, dtime=0.2, decay=2, gate=1;
var env, in;
env = Linen.kr(gate, 0.05, 1, 0.1, 2);
in = In.ar(out, 2);
XOut.ar(out, env, CombL.ar(in * env, maxdtime, dtime, decay, 1, in));
}, [\ir, \ir, 0.1, 0.1, 0]).store;

SynthDef(\distort, { arg out=0, pregain=40, amp=0.2, gate=1;
var env;
env = Linen.kr(gate, 0.05, 1, 0.1, 2);
XOut.ar(out, env, (In.ar(out, 2) * pregain).distort * amp);
}, [\ir, 0.1, 0.1, 0]).store;

SynthDef(\wah, { arg out=0, gate=1;
var env, in;
env = Linen.kr(gate, 0.05, 1, 0.4, 2);
in = In.ar(out, 2);
XOut.ar(out, env, RLPF.ar(in, LinExp.kr(LFNoise1.kr(0.3), -1, 1, 200, 8000), 0.1).softclip * 0.8);
}, [\ir, 0]).store;
)

(
```

```
var p, q, r, o;
p = Pbind(\degree, Prand((0..7),12), \dur, 0.3, \legato, 0.2);

q = Pfx(p, \echo, \dtime, 0.2, \decay, 3);

r = Pfx(q, \distort, \pregain, 20, \amp, 0.25);

o = Pfx(r, \wah);

Pseq([p, q, r, o], 2).play;
)
```

ID: 414

Pgroup

superclass: FilterPattern

Pgroup(pattern)

Starts a new group and plays the pattern in this group.
The group is released when the stream has ended.

Example:

```
(
var p, q, r, o;
p = Pbind(\degree, Prand((0..7),12), \dur, 0.3, \legato, 0.2);

Pgroup(p).play;

// post the node structure:
fork {
s.queryAllNodes;
3.wait;
s.queryAllNodes;
2.wait;
s.queryAllNodes;
}
)
```

ID: 415

Phid pattern that polls values from a human device interface

superclass: Pattern

*new(element, locID, repeats)

element one element of the interface of the device, like a button or an axis can be an index or, if the devicespec is present, also a symbol

locID the index of the device, defaults to 0 (first device)

repeats number of values to return (defaults to inf)

```
//example

//while this is running, test your device
(
a = Phid(0,0);
x = a.asStream;

Routine({ loop({
x.next.postln;
0.2.wait;
}) }).play;
)

// using devicespecs:
// for example wingman.  for other specs see [HIDDeviceService]
(
HIDDeviceService                'WingMan Action Pad'
   IdentityDictionary
    // buttons
\a -> 0, \b-> 1, \c-> 2,
\x-> 3, \y-> 4, \z-> 5,
```

```
    \l      //front left
    \r      //front right
\s-> 8,
\mode-> 9,
    \xx         //analog controller x axis
    \yy         //analog controller y axis
\slider-> 12,
\hat-> 13
])
)


//then you can use the named key:
(
a = Phid(\x, 0, inf);
x = a.asStream;

Routine({ loop({
x.next.postln;
0.2.wait;
}) }).play;
)



//'musical' example:
(
 Pbind
\freq, Pseq([Phid(\x,0,8),Phid(\y,0,8)],inf) * 500 + 200,
\dur, Phid(\slider) + 0.02,
\pan, Phid(\hat) * 2 - 1
).play;
)
```

ID: 416

Place interlaced embedding of subarrays

superclass: ListPatterns

Place(list, repeats)

returns elements in the list :
if an element is an array itself,
it embeds the first element when it comes by first time,
the second element when it comes by the second time
...
the nth when it comes by the nth time.

see also: **[Ppatlace]**

```
(
var a, b;
a = Place(#[1, [2,5], [3, 6]], inf);
x = a.asStream;
8.do({ x.next.postln; });
)

1
2
3
1
5
6
1
2
```

Place used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)



(
c = Place(#[0, 0, [0, 4, 7], [1, 5, 8], [2, 6, 9]], inf) + 67;
x = c.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
0.17.wait;
})
}).play;
)
```

ID: 417

Plazy

superclass: Pattern

evaluates a function that returns a pattern and embeds it in a stream.

Plazy(func)

```
(
a = Plazy({
var x, y;
x = Array.series(rrand(2, 4), [1, 100].choose, 1);
Pshuf(x,1);
});
x = Pn(a, inf).asStream;
30.do({ x.next.postln });
)
```

Plazy used to produce a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).store;
)
```

```
(
a = Plazy({
var x, y;
```

```
x = Array.series(rrand(2, 4), [1, 5].choose, 1);
x.put(x.size.rand, 8+0.1.rand2);
Pseq(x,1);
});
x = Pn(a, inf).asStream;

Routine
loop({
Synth("help-sinegrain", [\freq, (x.next*5+70).midicps]);
0.13.wait;
})
}).play;
)

// using event streams

(
a = Plazy({
var x, y;
x = Array.series(rrand(2, 4), [1, 5].choose, 1);
x.put(x.size.rand, 8+0.1.rand2);
 Pbind
  \instrument  'help-sinegrain'
\dur, 0.12,
\degree, Pseq(x, 2)
)
});

Pn(a, inf).play;
)
```

ID: 418

PlazyEnvir

superclass: Plazy

Evaluates a function that returns a pattern and embeds it in a stream.
In difference to [Plazy], the function is evaluated using the environment passed in by the stream

PlazyEnvir(func)

```
(
a = PlazyEnvir({ arg a=0, b=1; Pshuf([a, a, b], 2) }); // a, b default to 0,1
x = Pn(a, inf).asStream;

10.do { x.next.postln }; Post.nl;
e = (a:100);
10.do { x.next(e).postln }; Post.nl;
e = (a:100, b:200);
10.do { x.next(e).postln };
)
```

PlazyEnvir used to produce a Pbind:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05, pan=0;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, Pan2.ar(SinOsc.ar(freq, 0, env), pan))
}).store;

a = PlazyEnvir({ arg g=0, h=0, dur=1;
 postf("g:  %, h:  %, dur:  %\n"
```

```
 Pbind
  \instrument  'help-sinegrain'
\dur, dur,
\degree, Pseq([g, g, h, g, h], 2)
)
});

)


// different variants
(a <> (g:  0, h:  3, dur:0.2)).play; // single stream
(a <> (g:  [0, 4], h:  [3, -1], dur:0.2)).play; // same durations, two streams

// for more about the composition operator <> see:  Pchain
```

Some parameters, like duration, cannot be used in the form of an array in the Pbind. For full parallel expansion see [**PlazyEnvirN**]

ID: 419

PlazyEnvirN

superclass: PlazyEnvir

Evaluates a function that returns a pattern and embeds it in a stream.
In difference to [Plazy], the function is evaluated using the environment passed in by the stream.
In difference to [PlazyEnvir], PlayzEnvirN expands to **multiple parallel patterns** if the function arguments
receive multiple channels. In difference to PlazyEnvir, this works only with event streams.

PlazyEnvirN(func)

```
// example

(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05, pan=0;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, Pan2.ar(SinOsc.ar(freq, 0, env), pan))
}).store;

a = PlazyEnvirN({ arg g=0, h=0, dur=1;
 postf("g:  %, h:  %, dur:  %\n"

 Pbind
  \instrument  'help-sinegrain'
\dur, dur,
\degree, Pseq([g, g, h, g, h], 2)
)
});
);
```

```
// different variants
(a <> (g:  0, h:  3, dur:0.2)).play; // single stream
(a <> (g:  [0, 4], h:  [3, -1], dur:0.2)).play; // same durations, two streams
(a <> (g:  [0, 4], h:  [3, -1], dur:  [0.2, 0.3])).play; // different durations, two streams

// for more about the composition operator <> see:  Pchain
```

ID: 420

Pmono

superclass: Pattern

Pmono(synthDefName, patternPairs)

Plays one instance of a synth. The pattern pairs define changes in that one synth's controls.
If event[\id] is not nil, Pmono simply directs its pattern changes to that node and does not create an extra synth.

examples:

```
p = Pmono("default", \dur, 0.2, \freq, Pwhite(1,8) * 100 ).play

p.stop

// multi channel expansion is supported:

p = Pmono("default", \dur, 0.2, \freq, Pwhite(1,8) * 100, \detune, [0,2,5,1]).play

p.stop


// the following example will end after 5 seconds
// or you can stop it sooner with a stop message

(
p = Pfindur(5,
Pset(\detune,Pwhite(0,1.0) * [0,1,3,7],
Ppar([
Pmono("default", \dur, 0.2, \freq, Pwhite(1,8) * 100 ),
Pmono("default", \dur, 0.1, \freq, Pwhite(1,8) * 300)
])
)
).play;
)
```

```
p.stop;
```

ID: 421

Pmul event pattern that multiplies with existing value of one key

superclass: Pset

Pmul(name, value, pattern)

```
(
var a, b;
a = Pmul(\freq, 801, Pbind(\freq, 100));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Pmul(\freq, Pseq([3, 4, 6], 2), Pbind(\freq, 100));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, gate=1;
var env;
env = EnvGen.kr(Env.asr(0.001, 1, 0.2), gate, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)
```

```
(
    Pbind \dur       \instrument  \sinegrain, \freq, 440
b = Pmul(\freq, Pseq([1, 2, 3, 4, 5, 6, 7], inf), a);
b.play;
)
```

ID: 422

Pmulp event pattern that multiplies with existing value of one key

superclass: Psetp

Pmulp(name, value, pattern)

multiplies a value in an event stream until it ends, repeats this with new values until the value stream ends.

value can be a pattern, a stream or an array. the resulting stream ends when that incoming stream ends.

```
(
var a, b;
a = Pmulp(\freq, Pseq([2, 3, pi],inf), Pbind(\freq, Pseq([100, 200, 300])));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)

(
a = Pbind(\freq, Pseq([500, 600, 700]), \instrument, \sinegrain);
a = Pmulp(\freq, Pseq([0.5, 0.9, 0.8], inf), a);
a.play;
```

)

ID: 423

Pmulpre event pattern that multiplies with existing value of one key

superclass: FilterPattern

Pset(name, value, pattern)

multiplies with a value in an event, before it is passed up the stream.
to multiply with the value after it has been passed to the stream, use **Pmul**

```
(
var a, b;
a = Pmulpre(\note, 2, Pbind(\note, Pseq([1, 2, 3])));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

Pmulpre does not override incoming values:

```
(
var a, b;
a = Pmulpre(\freq, 801, Pset(\freq, 500, Pbind(\dur, 0.2)));
x = a.asStream;
9.do({ x.next(Event.default).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Pmulpre(\freq, Pseq([401, 801], 2), Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)



(
a = Pbind(\dur, 0.5, \instrument, \sinegrain);
b = Pmulpre(\freq, Pseq([1, 2, 3], inf), a);
b.play;
)
```

ID: 424

Pn

superclass: FilterPatterns

repeats the enclosed pattern a number of times

Pn(pattern, repeats)

```
(
var a, b;
	Pseq                    // repeat pattern four times
b = a.asStream;
16.do({ b.next.postln; });
)
```

ID: 425

Ppar embed event streams in parallel

superclass: ListPatterns

Embeds several event streams so that they form a single output stream with all their events in temporal order.
When one stream ends, the other streams are further embedded until all have ended.

Ppar(list, repeats)

list: list of patterns or streams
repeats: repeat the whole pattern n times (default: 1)

```
// see the delta values in the resulting events
(
var a, b, c, t;
a = Pbind(\x, Pseq([1, 2, 3, 4]), \dur, 1);
b = Pbind(\x, Pseq([10, 20, 30, 40]), \dur, 0.4);
c = Ppar([a, b]);
t = c.asStream;
20.do({ t.next(Event.default).postln; });
)

// sound example
(
var a, b;
a = Pbind(\note, Pseq([7, 4, 0], 4), \dur, Pseq([1, 0.5, 1.5], inf));
b = Pbind(\note, Pseq([5, 10, 12], 4), \dur, 1);
Ppar([ a, b ]).play;
)
```

ID: 426

Ppatlace interlaced embedding of streams

superclass: Pseq

Similar to **[Place]**, but the list is an array of streams or patterns. The results of each stream will be output in turn.

Ppatlace(list, repeats)

```
// example

p = Ppatlace([Pwhite(1, 5, 5), Pgeom(10, 1.01, 10)], inf);
x = p.asStream;
x.all;



   // from Pwhite
    // from Pgeom
   // from Pwhite
      // etc....
5
10.201
4
10.30301
2
10.4060401
10.510100501
10.61520150601
10.72135352107
10.828567056281
10.936852726844
nil
```

Note that the Ppatlace has an infinite number of repeats, but the resulting stream is

finite because the member streams are all finite. When the first stream (Pwhite) comes to an end, it is skipped and you see only the second stream until it stops.

If even one member stream is infinite and Ppatlace has infinite repeats, the Ppatlace stream will also be infinite.

Ppatlace as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)



// interlace two streams
(
var    Ppatlace
Pseq([0, 0, 0, 0, 8, 0, 8], inf),
Pseries(1, 1, 32)
], inf) + 67;
x = c.asStream;

Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps, \dur, 0.2]);
0.17.wait;
})
}).play;
)




// a more complicated example:
```

```
(
    Ppatlace
 Pxrand
Pseq(#[0, -2, -3, -5, -7], 1), Pwhite(-12, 4, 3), Pshuf(#[0, -2, -3, -5, -7], 1)
], inf),
 Pxrand
Pseq(#[0, 2, 4, 5, 7], 1), Pwhite(-4, 12, 3), Pshuf(#[0, 2, 4, 5, 7], 1)
], inf)
], inf) + 67;
x = c.asStream;

Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps, \dur, 0.2]);
0.17.wait;
})
}).play;
)
```

ID: 427

Ppatmod

superclass: Pattern

the function that modifies the enclosed pattern and embeds it in the stream.

Ppatmod(pattern, func, repeats)

```
(
    Ppatmod
Pseq([0, 0, 0, 0],1),
{ arg pat, i;
var list;
list = pat.list;
pat.list = list.put(list.size.rand, 2);
}, inf);

x = a.asStream;
30.do({ x.next.postln });
)
```

Ppatmod used to modify a pattern that produces a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)


(
```

```
a = Pn(
  Ppatmod
Pseq([0, 0, 0, 0],1),
{ arg pat, i;
var list;
list = pat.list;
pat.list = list.put(list.size.rand, 2);
}, 15),
inf).asStream;

Routine
loop({
Synth("help-sinegrain", [\freq, (a.next*5+77).midicps]);
0.13.wait;
})
}).play;
)
```

ID: 428

Pprob random values with arbitrary probability distribution

superclass: Patterns

creates an integral table on instantiation (cpu intensive) which is then used by the streams
to generate random values efficiently.

Pprob(distribution, lo, hi, length, tableSize)

distribution
desired probability distribution (histogram)
lo, hi
lower and upper bounds of the resulting values
length
number of values to repeat
tableSize
resample table to this size. If the size of the distribution is
smaller than 64, it is (linearly) resampled to this minimum size

distribution_(list)
set the distribution, the table is recalculated
tableSize_(n)
set the resample size, the table is recalculated

```
// a consistency test
(
var a = Pprob([0,0,0,0,1,1,1,1,3,3,6,6,9].scramble);
var b = a.asStream;
b.nextN(800).sort.plot("sorted distribution");
                        "sorted distribution, again"
)


// comparison:  emulate a linrand
```

```
(
var a, b;
a = Pprob([1, 0]);
x = Pfunc({ 1.0.linrand });

b = a.asStream;
y = x.asStream;

postf("Pprob mean:  % linrand mean:  % \n", b.nextN(800).mean, y.nextN(800).mean);

b.nextN(800).sort.plot("this is Pprob");
y.nextN(800).sort.plot("this is linrand");
)


// compare efficiency

        Pprob           // this is fairly expensive
bench { 16.do { Pseq([0, 1] !  32) } }

x = Pprob([0, 1]).asStream;
y = Pseq([0, 1], inf).asStream;

bench { 100.do { x.next } }; // this very efficient
bench { 100.do { y.next } };



// sound example
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)


(
```

```
a = Pprob([0, 0, 1, 0, 1, 1, 0, 0], 60, 80);
t = a.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, t.next.midicps]);
0.01.wait;
})
}).play;
)

a.distribution = [0, 1];
a.distribution = [1, 0];
a.distribution = [0, 0, 0, 0, 1, 0];
a.distribution = [0, 1, 0, 0, 0, 0];

// higher resolution results in a more accurate distribution:
a.tableSize = 512;
a.tableSize = 2048;
```

ID: 429

Prand

superclass: ListPatterns

returns one item from the list at random for each repeat.

```
(
var a, b;
a = Prand.new(#[1, 2, 3, 4, 5], 6); // return 6 items
b = a.asStream;
7.do({ b.next.postln; });
)
```

Prand used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)

(
a = Prand(#[60, 61, 63, 65, 72], inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.2.wait;
})
}).play;
)
```

ID: 430

Preject

superclass: FuncFilterPattern

Preject(func, pattern)

rejects values for which the function returns true. the value is passed to the function

```
(
var a, b;
a = Preject({ arg item; item == 1 }, Pseq(#[1, 2, 3],inf));
x = a.asStream;
9.do({ x.next.postln; });
)
```

the message reject returns a Preject when passed to a pattern

```
(
var a, b;
a =  Pseq(#[1, 2, 3],inf).reject({ arg item; item == 1 });
a.postln;
x = a.asStream;
9.do({ x.next.postln; });
)
```

ID: 431

Prewrite

superclass: FilterPattern

Lindenmayer system pattern for selfsimilar structures.
Its **dictionary** maps one element to an array of child elements.
The algorithm replaces iteratively (**levels** deep) elements by
arrays of elements starting with the values in the **pattern.**

Prewrite(pattern, dictionary, levels)

dict:
IdentityDictionary[
elem1 -> [otherElements],
elem2 -> [otherElements],
elem2 -> [otherElements]
]
the examples use the shortcut for IdentityDictionary.

```
(
    Prewrite    //start with 0
( 0:  #[2,0],
1:  #[0,0,1],
2:  #[1,0,1]
), 4);
x = a.asStream;
30.do({ x.next.postln });
)
```

Prewrite used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
```

```
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)



(
a = Prewrite(0, ( 0:  #[2,0],
1:  #[0,0,1],
2:  #[1,0,1]
),
4).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, (a.next * 5 + 70).midicps]);
0.1.wait;
})
}).play;
)
```

ID: 432

Prorate divide stream proportionally

superclass: FilterPattern

*new(proportions, pattern)

proportions: a pattern that returns either numbers (divides the pattern into pairs) or arrays of size n which are used to split up the input into n parts.
pattern: a numerical pattern

```
// examples:


// divide 1 into various proportions
(
a = Prorate(Pseq([0.35, 0.5, 0.8]), 1);

x = a.asStream;

x.nextN(8)
)

// divide a pattern into various proportions
(
a = Prorate(Pseq([0.35, 0.5, 0.8]), Prand([20, 1], inf));

x = a.asStream;

x.nextN(8)
)


// divide 1 into several parts
```

```
(
a = Prorate(Pseq([[1, 2], [5, 7], [4, 8, 9]]).collect(_.normalizeSum), 1);

x = a.asStream;

x.nextN(8)
)
```

ID: 433

Pseed set the random seed in subpattern

superclass: FilterPattern

set the random generator seed of the resulting stream.
see [**randomSeed**] helpfile

Pseed(seed, pattern)

seed: integer number, pattern or stream that return an integer number

```
a = Pseed(1972, Prand([1,2,3], inf));

b = a.asStream;
10.do({ b.next.post });

c = a.asStream;
10.do({ c.next.post });


//using a seed pattern as input:

a = Pseed(Pseq([1812, 1912], inf), Prand([1,2,3], 5));

b = a.asStream;
2.do({ 5.do({ b.next.post });"".postln;  });

c = a.asStream;
2.do({ 5.do({ c.next.post });"".postln;  });
```

```
//outer thread is independant:

a = Pseed(Prand([1534, 1600, 1798, 1986, 2005], inf), Pshuf([1, Prand([7, 9], 2), 1, 2, 3], 1));

//returns random streams
b = a.asStream;
2.do({ 5.do({ b.next.post });"".postln;  });

c = a.asStream;
2.do({ 5.do({ c.next.post });"".postln;  });
```

ID: 434

Pseg

superclass: Pstep

Pseg(levelpattern, durpattern, curvepattern)

Pseg defines a function of time as a breakpoint envelope using the same parameters as **Env**. These patterns
can be used to describe tempo or dynamic variations independent of the rhythmic patterns that express them.

levelpattern - The first level is the initial value of the envelope, all subsequent values are interpolated
If durpattern is nil, then levelpattern specifies the entire envelope by returning arrays of the form:
[level, dur, curve]
durpattern - duration of segments in seconds.
curvepattern - this parameter determines the shape of the envelope segments.
The possible values are:
'step' - flat segments
'linear' - linear segments, the default
'exponential' - natural exponential growth and decay. In this case, the levels must all be nonzero
and the have the same sign.
'sine' - sinusoidal S shaped segments.
'welch' - sinusoidal segments shaped like the sides of a Welch window.
a Float - a curvature value for all segments.
An Array of Floats - curvature values for each segments.

```
s.boot;

// change a parameter
(
Pbind
\note,  Pseg( Pseq([1, 5],inf), Pseq([4,1],inf), 'linear'),
 \dur  0.1
).play;
```

```
)

(
Pbind
\freq,  Pseg( Pseq([400, 1500],inf), Pseq([4,4],inf), Pseq(['linear','exp'],inf)),
 \dur  0.1
).play;
)
```

ID: 435

Pselect

superclass: FuncFilterPattern

Pselect(func, pattern)

returns values for which the function returns true. the value is passed to the function

```
(
var a, b;
a = Pselect({ arg item; item != 2 }, Pseq(#[1, 2, 3],inf));
x = a.asStream;
9.do({ x.next.postln; });
)
```

the message select returns a Pselect when passed to a pattern

```
(
var a, b;
a =  Pseq(#[1, 2, 3],inf).select({ arg item; item != 2 });
a.postln;
x = a.asStream;
9.do({ x.next.postln; });
)
```

ID: 436

Pseq

superclass: ListPatterns

cycles over a list of values. The repeats variable gives
the number of times to repeat the entire list.

Pseq(list, repeats, offset)

```
(
var a, b;
a = Pseq.new(#[1, 2, 3], 2); // repeat twice
b = a.asStream;
7.do({ b.next.postln; });
)
```

Pseq also has an offset argument which gives a starting offset into the list.

```
(
var a, b;
a = Pseq.new(#[1, 2, 3, 4], 3, 2); // repeat 3, offset 2
b = a.asStream;
13.do({ b.next.postln; });
)
```

You can pass a function for the repeats variable that gets evaluated when the stream is created.

```
(
var a, b;
a = Pseq.new(#[1, 2], { rrand(1, 3) }); // repeat 1,2, or 3 times
b = a.asStream;
7.do({ b.next.postln; });
)
```

If you specify the value **inf** for the repeats variable, then it will repeat indefinitely.

```
(
```

```
var a, b;
a = Pseq.new(#[1, 2, 3], inf); // infinite repeat
b = a.asStream;
10.do({ b.next.postln; });
)
```

Pseq used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
```

```
(
a = Pseq(#[60, 61, 63, 65, 72], inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.2.wait;
})
}).play;
)
```

ID: 437

Pser

superclass: ListPatterns

is like Pseq, however the repeats variable gives the number of items returned instead of the number of complete cycles.

```
(
var a, b;
a = Pser.new(#[1, 2, 3], 5); // return 5 items
b = a.asStream;
6.do({ b.next.postln; });
)
```

Pser used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)



(
a = Pser([Pser(#[60, 61, 63, 65, 72], 3)], inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.2.wait;
})
}).play;
)
```

ID: 438

Pset event pattern that sets values of one key

superclass: FilterPattern

Pset(name, value, pattern)

sets a value in an event stream. acts like one key in a **Pbindf**

```
(
var a, b;
a = Pset(\freq, 801, Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

Pset overrides incoming values:

```
(
var a, b;
a = Pset(\freq, 801, Pbind(\freq, 108));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Pset(\freq, Pseq([401, 801], 2), Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)


(
a = Pbind(\dur, 0.5, \instrument, \sinegrain);
a = Pset(\freq, Pseq([500, 600, 700], inf), a);
a = Pset(\legato, Pseq([0.01, 1],inf), a);
a.play;
)
```

ID: 439

Psetp event pattern that sets values of one key

superclass: Pset

Psetp(name, value, pattern)

sets a value in an event stream until it ends, repeats this with new values until the value stream ends.

value can be a pattern, a stream or an array. the resulting stream ends when that incoming stream ends.

```
(
var a, b;
a = Psetp(\freq, Pseq([801, 1008],inf), Pbind(\dur, Pseq([0.5, 0.111, 0.22])));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

Psetp overrides incoming values:

```
(
var a, b;
a = Psetp(\freq, 801, Pbind(\freq, 108));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
```

```
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)

(
a = Pbind(\dur, Pseq([0.5, 0.3, 0.1]), \instrument, \sinegrain);
a = Psetp(\freq, Pseq([500, 600, 700], inf), a);
a.play;
)
```

ID: 440

Psetpre event pattern that sets values of one key

superclass: FilterPattern

Pset(name, value, pattern)

sets a value in an event, before it is passed up the stream.
to set the value after it has been passed to the stream, use **Pset**

```
(
var a, b;
a = Psetpre(\freq, 801, Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

Psetpre does not override incoming values:

```
(
var a, b;
a = Psetpre(\freq, 801, Pbind(\freq, 108));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

value can be a pattern or a stream. the resulting stream ends when that incoming stream ends

```
(
var a, b;
a = Psetpre(\freq, Pseq([401, 801], 2), Pbind(\dur, 0.5));
x = a.asStream;
9.do({ x.next(Event.new).postln; });
)
```

sound example

```
(
SynthDef \sinegrain
{ arg out=0, freq=440, sustain=0.02;
var env;
env = EnvGen.kr(Env.perc(0.001, sustain), 1, doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env * 0.1))
}).store;
)



(
a = Pbind(\dur, 0.5, \instrument, \sinegrain);
a = Psetpre(\freq, Pseq([500, 600, 700], inf), a);
a = Psetpre(\legato, Pseq([0.01, 1],inf), a);
a.play;
)
```

ID: 441

Pshuf

superclass: ListPatterns

returns a shuffled version of the list item by item, with n repeats.

```
(
var a, b;
a = Pshuf(#[1, 2, 3, 4, 5], 3); // repeat 3 times
b = a.asStream;
16.do({ b.next.postln; });
)
```

Pshuf used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)



(
a = Pn(Pshuf(#[60, 60, 60, 61, 63, 65, 72], 4), inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.15.wait;
})
}).play;
)
```

ID: 442

Pslide

superclass: ListPatterns

Pslide(list, repeats, length, step, start, wrapAtEnd)

repeats: number of segments
length: length of each segment
step: is how far to step the start of each segment from previous.
start: what index to start at.
wrapAtEnd: if true (default), indexing wraps around if goes past beginning or end. If false, the pattern stops if it hits a nil element or goes outside the list bounds.

step can be negative.

```
(
var a, b;
a = Pslide(#[1, 2, 3, 4, 5], inf, 3, 1, 0);
x = a.asStream;
13.do({ x.next.postln; });
)
```

Pslide used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
```

```
(
c = Pslide(#[1, 2, 3, 4, 5], inf, 3, 1, 0) * 3 + 67;
x = c.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
0.17.wait;
})
}).play;
)
```

ID: 443

Pstep

superclass: Pattern
related: Pseg

Pstep(levelpattern, durpattern)

Levelpattern can return either individual values or arrays. The value returned by levelpattern is
returned for the duration returned by durpattern.

Pstep is good for representing chord progressions, scale progressions, accent patterns, etc.

```
s.boot;


// change degree independant of number of events that have been playing

(
Pbindf
Ppar([
Pbind(
\degree, Pbrown(0,12,1),
   \dur              0.1,0.2,0.4,0.8,1.6],inf),3.2)
  ),
Pbind(
\degree, Pbrown(0,20,1),
   \dur              0.1,0.2,0.4,0.8,1.6],inf),4.5)
)
]),
\scale, Pstep(Pseq([ [0,2,4,5,7,9,11], [0,1,2,3,4,5,6]], inf), 5),
\db,Pstep(Pseq([4,-4,0,-4],inf),0.25) + Pwhite(-20, -15)
).play;
)
```

```
// change one parameter
(
Pbind
\degree, Pstep(Pseq([1, 2, 3, 4, 5]), 1.0).trace,
\dur, Pseries(0.1, 0.1, 15)
).play;
)


// change degree independant of number of events that have been playing

(
var a, b;
a = Pbind(
\degree, Pstep(Pseq([0, 2b, 3],1), 1.0),
\dur, Prand([0.2, 0.5, 1.1, 0.25, 0.15], inf)
);
b = Pbind(
\degree, Pseq([0, 2b, 3], 1),
\dur, 2,
 \ctranspose
);
Pseq([Event.silent(1.25), Ppar([a, b])], inf).play;
)



// test tempo changes

(
var a, b;
a = Pbind(
\degree, Pstep(Pseq([0, 2b, 3],1), 1.0),
\dur, Prand([0.2, 0.5, 1.1, 0.25, 0.15], 9)
);
b = Pbind(
```

```
\degree, Pseq([0, 2b, 3], 1),
\dur, 2,
 \ctranspose
);

Ppar([a, b], inf).play;
)


SystemClock.sched(0, { TempoClock.default.tempo = [1, 2, 3, 5].choose.postln; 2 });

TempoClock.default.tempo = 1.0;



// timing test:
// parallel streams


(

var times, levels;


SynthDef "pgrain"
{ arg out = 0, freq=800, sustain=0.001, amp=0.5, pan = 0;
var window;
window = Env.sine(sustain, amp);
Out.ar(out,
Pan2.ar(
SinOsc.ar(freq) * EnvGen.ar(window, doneAction:2),
pan
)
)
}
).store;


times = Pseq([3.4, 1, 0.2, 0.2, 0.2], inf);
levels = Pseq([0, 1, 2, 3, 4], inf);
```

```
a = Pstep(levels, times);
    Pbind \instrument  \pgrain  \octave     \dur        \degree
x = times;

Ppar([b, Pset(\mtranspose, 2, b) ]).play;



b.play;
r {
var z = x.asStream; // direct times
loop {
z.next.wait;
s.makeBundle(0.2, {
Synth(\pgrain, [\freq, 3000, \sustain, 0.01]); // signal tone
})

}
}.play(quant:1)
)
```

ID: 444

PstepNadd pattern that returns combinatoric sums

superclass: PstepNfunc

combines an arbitrary number of patterns by summing (depth first traversal).
when a stream ends it is recreated from its pattern until the top stream ends.

see also: Pstep3add

*new(pattern1, pattern2, ... patternN);

```
//examples


// comparing PstepNadd and Pstep3add (test)
(
x = PstepNadd(Pseq([1, 2, 3]), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300])).asStream;
y = Pstep3add(Pseq([1, 2, 3]), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300])).asStream;

50.do({ [x.next, y.next].postln });
)

// pattern return stream until the longest stream ended
(
    PstepNadd
Plazy({ "pattern1.asStream".postln; Pseq([1, 2, 3], 2) }),
Plazy({ "pattern2.asStream".postln; Pshuf([10, 20, 30, 40]) }),
Plazy({ "pattern3.asStream".postln; Pseq([100, 200, 300]) }),
Plazy({  Pseries(1, 1, 4) * 0.01 })
).asStream;
150.do({ x.next.postln });
)

// if the last pattern loops it the combinatorics loop there:
x = PstepNadd(Pseq([1, 2, 3]), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300], inf)).asStream;
```

```
50.do({ x.next.postln });

// if the first pattern loops, the whole iteration loops as if it was used in a Pn(.., inf):
x = PstepNadd(Pseq([1, 2, 3], inf), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300])).asStream;
y = Pn(PstepNadd(Pseq([1, 2, 3]), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300])), inf).asStream;
150.do({ [x.next, y.next].postln });

// sound example
(
Pbind
 \octave
 \degree  PstepNadd
Pseq([1, 2, 3]),
Pseq([0, -2, [1, 3], -5]),
Pshuf([1, 0, 3, 0], 2),
Pseq([1, -1], 5)
),
 \dur  PstepNadd
Pseq([1, 0, 0, 1], 2),
Pshuf([1, 1, 2, 1], 2)
).loop * (1/8),
\legato, Pn(Pshuf([0.2, 0.2, 0.2, 0.5, 0.5, 1.6, 1.4], 4), inf),
\scale, #[0, 1, 3, 4, 5, 7, 8]
).play;
)
```

ID: 445

PstepNfunc combinatoric pattern

superclass: Pattern

combines an arbitrary number of patterns by evaluating a function (depth first traversal). when a stream ends it is recreated from its pattern until the top stream ends.

see also: **PstepNadd**

*new(func, patternList);

```
//examples
(
f = { arg vals;
vals.postln;
};
    PstepNfunc
Pseq([1, 2, 3]), Pseq([4, 5, 6]), Pseq([7, 8, 9])
]).asStream;
50.do({ x.next });
)

(
f = { arg vals;
var r;
r = vals.copy.removeAt(0);
vals.do({ arg item;  r = item / r.squared * 10 });
r
};
    PstepNfunc
[
Pseq([1, 2, 3], inf),
Pseq([2, pi, 1]),
```

```
Pseq([0.1, 3, 0.2, 3])
]
).asStream;

50.do({ x.next.postln });
)

// note that if the last pattern loops it will stick to that one:
(
f = { arg vals;
vals.postln;
};
x = PstepNfunc(f, [Pseq([1, 2, 3]), Pseq([10, 20, 30, 40]), Pseq([100, 200, 300], inf)]).asStream;
50.do({ x.next });
)



(
f = { arg vals;
 vals.inject(1, { arg x, y; x * y })
};
    PstepNfunc
[
Pseq([1, 2, 3], inf),
Pseq([2, pi, 1]),
Pseq([0.1, 3, 0.2, 3])
]
).asStream;

50.do({ x.next.postln });
)
```

ID: 446

Pstutter

superclass: FilterPatterns

Pstutter(n, pattern)

repeat each element n times

n may be a pattern, so the number of times can vary each iteration

```
(
var a, b;
a = Pstutter(2, Pseq(#[1, 2, 3],inf));
x = a.asStream;
13.do({ x.next.postln; });
)
```

Pstutter used as a sequence of pitches

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)


(
c = Pstutter(3, Prand([1, 2, 3],inf)*4+65);
x = c.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
```

```
0.12.wait;
})
}).play;
)
```

ID: 447

Pswitch

Pswitch(list, which)

chooses elements from the **list** by a stream of indices (**which**).
the elements are embedded in the stream, so if an element is a pattern, it will play until it is finished. if it is a simple number it will just yield itself.
play then resumes embedding the next element in the **list**...

```
(
var a, b;
a = Pseq(#[1, 2, 3], 2);
b = Pseq(#[65, 76]);
c = Pswitch([a, b, 800], Pseq([2, 2, 0, 1], inf));
x = c.asStream;
24.do({ x.next.postln; });
)
```

Pswitch used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)

(
a = Pseq(#[73, 71, 69], 2);
b = Pseq(#[0, 0, 0, 4, 0]+64);
c = Pswitch([a, b, 75], Pseq([2, 2, 0, 1], inf));
x = c.asStream;
```

```
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
0.18.wait;
})
}).play;
)
```

ID: 448

Pswitch1

Pswitch1(list, which)

the elements in the list are collected as streams,
the stream of indices (**which**) is used to in turn select
one of the elements. one value only is yielded from that stream.

this is different than Pswitch which embeds the element in the stream,
allowing it to play out until it is finished. Pswitch1 switches every event.

```
(
var a, b;
a = Pseq(#[1, 2, 3], inf);
b = Pseq(#[65, 76], inf);
c = Pswitch1([a, b, 800], Pseq([2, 2, 0, 1], inf));
x = c.asStream;
24.do({ x.next.postln; });
)
```

Pswitch used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
```

```
(
a = Pseq(#[73, 71, 69], inf);
```

```
b = Pseq(#[0, 0, 0, 4, 0]+64, inf);
c = Pswitch1([a, b, 75], Pseq([2, 2, 0, 1], inf));
x = c.asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, x.next.midicps]);
0.18.wait;
})
}).play;
)
```

ID: 449

Psync synchronise and limit pattern duration

superclass: FilterPattern

***new(pattern, min, max, tolerance)**
pattern: a pattern that returns events
min: beat duration for ending patterns
max: maximum length of pattern
tolerance: difference threshhold that a pattern must exceed max to be ended

```
(
SynthDef "sinegrain2"
{ arg out=0, freq=440, sustain=0.05, pan;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.3), doneAction:2);
Out.ar(out, Pan2.ar(SinOsc.ar(freq, 0, env), pan))
}).store;
)

s.boot;


// example:


(
// a fixed duration pattern:

f = Pbind(
\dur, 0.5,
\degree, Pn(4,1),
    \instrument  \sinegrain2
);
```

```
// this pattern has indetermined length:
a = Prand([
   Pbind
\dur, Pseq([0.02, 0.002, 0.1, 0.1],2),
\degree, Pseq([9, 7, 5],inf),
    \instrument  \sinegrain2
),
   Pbind
\dur, Pseq([1, 0.35],2),
\degree, Pseq([0, [2b,5b]],inf),
    \instrument  \sinegrain2
),
   Pbind
\dur, Pseq([0.15, 0.25, 1.3],2),
\degree, Pseq([2b,4,5b],inf),
    \instrument  \sinegrain2
)
]);
)

Pseq([f, f, a, a], inf).play; // play a sequence


// Psync allows to limit the duration of a stream relative to a beat grid

    Psync           // create a sequence of exactly 1 beat elements
Pseq([f, f, b, b], inf).play;


    Psync           // create a sequence of elements of either 1 or 2 beats length
Pseq([f, f, b, b], inf).play;

(
    Psync        // create a sequence of elements with a minimum of 2 beats,
    // but with undetermined upper limit
Ppar
Pseq([f, f, b, b], inf), // sequence
 Pbind \instrument  \sinegrain2  \freq        \sustain        \dur     // metronome
]).play;
)
```

ID: 450

Ptime

superclass: Pattern

returns time in beats from moment of embedding in stream

Ptime(repeats)

```
s.boot;

// post time
(
Pbind
\pfunc, Ptime.new.trace,
\dur, Pseries(0.5, 0.5, 5)
).play;
)


// change degree independant of number of events that have been playing

(
var a, b;
a = Pbind(
\degree, Pswitch(#[0, 2b, 3], Ptime(8).round(2) / 2),
\dur, Prand(#[0.2, 0.5, 1.1, 0.25, 0.15], inf)
);
b = Pbind(
\degree, Pseq(#[0, 2b, 3], 1),
\dur, 2,
 \ctranspose
);
Pseq([Event.silent(1.25), Ppar([a, b])], inf).play;
)
```

```
// test tempo changes

(
var a, b;
a = Pbind(
\degree, Pswitch(#[0, 2b, 3], Ptime(8).round(2) / 2),
\dur, Prand(#[0.2, 0.5, 1.1, 0.25, 0.15], 9)
);
b = Pbind(
\degree, Pseq(#[0, 2b, 3], 1),
\dur, 2,
 \ctranspose
);
Pn
Pfset({  tempo = #[1, 2, 4].choose.postln },
Pseq([
Event.silent(1.25),
Ppar([a, b])
])
)
).play
)
```

ID: 451

Ptpar embed event streams in parallel, with time offset

superclass: ListPatterns

Embeds several event streams so that they form a single output stream with
all their events in temporal order, providing a global **offset** for each.
When one stream ends, the other streams are further embedded until all have ended.

Ptpar(list, repeats)

list: list of pairs of times and patterns: [time, pat, time, pat ..]
repeats: repeat the whole pattern n times (default: 1)

```
// see the delta values in the resulting events
(
var a, b, c, t;
a = Pbind(\x, Pseq([1, 2, 3, 4]), \dur, 1);
b = Pbind(\x, Pseq([10, 20, 30, 40]), \dur, 0.4);
c = Ptpar([0.0, a, 1.3, b]);
t = c.asStream;
20.do({ t.next(Event.default).postln; });
)

// sound example
(
var a, b;
a = Pbind(\note, Pseq([7, 4, 0], 4), \dur, Pseq([1, 0.5, 1.5], inf));
b = Pbind(\note, Pseq([5, 10, 12], 4), \dur, 1);
Ptpar([ 0.0, a, 1.3, b ]).play;
)
```

ID: 452

Ptuple

superclass: ListPatterns

Ptuple(list, repeats)
list: an Array of Patterns
repeats: an Integer or inf

At each iteration returns a tuple (array) combining the output of each of the patterns in the list. When any of the patterns returns a nil, Ptuple ends that 'repeat' and restarts all of the streams.

```
(
var a, b;
a = Pseq(#[1, 2, 3], inf);
b = Pseq(#[65, 76], inf);
c = Ptuple([a, a, b], inf);
x = c.asStream;
8.do({ x.next.postln; });
)
```

```
(
var a, b;
a = Pseq(#[1, 2, 3], inf);
    Pseq                // stops after 3 cycles
    Ptuple                // stops after 4 cycles
x = c.asStream;
8.do({ x.next.postln; });
)
```

Ptuple used as a sequence of pitches (chords)

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
```

```
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)

(
a = Pseq(#[73, 71, 69, 69, 65, 64], inf);
b = Pseq(#[0, 0, 0, 4, 0, 3, 2]+64, inf);
c = Ptuple([a, b], inf);
x = c.asStream;
Routine
var chord;
loop({
chord = x.next.midicps;
chord.do({ arg freq;
Synth("help-sinegrain", [\freq, freq]);
});
0.1.wait;
})
}).play;
)
```

ID: 453

Pwalk : ListPattern

A one-dimensional random walk.

*new(list, stepPattern, directionPattern, startPos)

list: The items to be walked over.
stepPattern: Returns integers that will be used to increment the index into list.
directionPattern: Used to determine the behavior at boundaries. When the index crosses a boundary, the next direction is drawn from this stream: 1 means use stepPattern as is, -1 means go in the reverse direction.

Common patterns: **1** – always wrap around to the other boundary.
Pseq([1, -1], inf) – go forward first, then backward, then forward again

startPos: Where to start in the list.

Example:

```
p = Pwalk(
Array.series(20, 0, 1),  // integers, 0-19
  // steps up to 2 in either direction, weighted toward positive
Pwrand([-2, -1, 0, 1, 2], [0.05, 0.1, 0.15, 1, 0.1].normalizeSum, inf),
  // reverse direction at boundaries
Pseq([1, -1], inf),
 10); // start in the middle
a = p.asStream;

200.do({ a.next.post; ", ".post });

                                  // this one will always wrap around
b = q.asStream;

200.do({ b.next.post; ", ".post });

// non-random walk:  easy way to do up-and-down arpeggiation
s.boot;
(
```

```
p = Pwalk(
[60, 64, 67, 72, 76, 79, 84].midicps,  // C major
Pseq([1], inf),
 Pseq          inf   // turn around at either end
0);
f = p.asStream;

SynthDef "help-Pwalk"    arg
Out.ar(0, Saw.ar([freq, freq+1], 0.5) * EnvGen.kr(Env.perc(0.01, 0.1), doneAction:2))
}).send(s);
)

(
r = Task({
{
Synth.new("help-Pwalk", [\freq, f.next]);
0.1.wait;
}.loop;
        SystemClock
)

r.stop;
```

ID: 454

Pwhile while a condition holds, repeatedly embed stream

superclass: FuncFilterPattern

Pwhile(func, pattern)

```
(
var a, b, z = true;
a = Pwhile({ z }, Pseq(#[1, 2, 3]));
x = a.asStream;
9.do({ x.next.postln; });
z = false;
x.next.postln;
)
```

ID: 455

Pwrand

superclass: ListPatterns

returns one item from the list at random for each repeat, the probability for each item is determined by a list of weights which should sum to 1.0.

```
(
var a, b;
a = Pwrand.new(#[1, 2, 3], #[1, 3, 5].normalizeSum, 6); // return 6 items
b = a.asStream;
7.do({ b.next.postln; });
)
```

Prand used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
```

```
(
a = Pwrand(#[60, 61, 63, 65, 72], #[10, 2, 3, 1, 3].normalizeSum, inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.1.wait;
})
}).play;
)
```

ID: 456

Pwrap

superclass: FilterPattern

Pwrap(pattern,lo,hi)

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)

(
a = Pn(
 Pwrap
Pgeom(200,1.07,96),
200,
1000.0
),
 inf
);
x = a.asStream;

Routine
loop({
Synth("help-sinegrain", [\freq, x.next.debug,\dur,0.3]);
0.12.wait;
})
}).play;

)
```

```
-felix
```

ID: 457

Pxrand

like Prand, returns one item from the list at random for each repeat, but Pxrand never repeats the same element twice in a row.

```
(
var a, b;
a = Pxrand.new(#[1, 2, 3], 10); // return 10 items
b = a.asStream;
11.do({ b.next.postln; });
)
```

Pxrand used as a sequence of pitches:

```
(
SynthDef "help-sinegrain"
{ arg out=0, freq=440, sustain=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction:2);
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)
```

```
(
a = Pxrand(#[60, 61, 63, 65, 72], inf).asStream;
Routine
loop({
Synth("help-sinegrain", [\freq, a.next.midicps]);
0.1.wait;
})
}).play;
)
```

ID: 458

Stream

superclass: AbstractFunction

Stream is an abstract class that is not used directly. The following attempts to document some
aspects of the use of Streams for music generation.

Overview

A Stream represents a sequence of values that are obtained incrementally by repeated **next** messages. A Stream can be restarted with a **reset** message. (Not all streams actually implement reset semantics.)

The class Object defines **next** to return the object itself. Thus every object can be viewed
as a stream and most simply stream themselves.

Stream is the base class for classes that define streams.

In SuperCollider, Streams are primarily used for handling text and for generating music.

Two Stream classes: FuncStream and Routine

FuncStream(nextFunction, resetFunction)

A Function defines a stream consisting of the Function itself, a FuncStream defines a stream
that consists of *evaluations* ofits nextFunction.

```
  // Example 1:  a Function vs.  a FuncStream
(
f = { 33.rand };
x = FuncStream(f);
10.do({ [f.next, x.next].postln });
```

```
)

 // Example 2:  the reset function
(

f = { 33.rand };
x = FuncStream(f, {thisThread.randSeed_(345)});
x.reset;
10.do({ [f.next, x.next].postln });
x.reset;
10.do({ [f.next, x.next].postln });
)
```

Routine(nextFunction, stacksize)

In a FuncStream, the nextFunction runs through to completion for each element of the stream.
In a Routine, the nextFunction returns values with **yield** and resumes execution (when it receives
a **next** message) at the expression folowing the yield. This allows a sequence of expressions in
the function definition to represent a sequence of distinct events, like a musical score.

```
  // example
(
x = Routine({
1.yield;
2.yield;
3.yield;
});
4.do({ x.next.postln });
)
```

Once the nextFunction completes execution, the Routine simply yields nil repeatedly.
Control structures (such as **do** or **while**) can be used within the nextFunction in a manner analogous
to repeat marks in a score

```
 // example
```

```
(
x = Routine({
4.do({
[1,2,3,4].do({ arg i; i.yield; });
})
});
17.do({ x.next.postln });
)
```

Playing streams

Because streams respond like functions to the value message, they can be used as a scheduling task.

```
  // compare:
  // a function, returning 0.5
(
SystemClock.sched(0.0,
{ "***".postln; 0.5 }
);
)

 // a stream, returning 0.5 and 0.1
(
SystemClock.sched(0.0,
Routine({ loop {
"***".postln; 0.5.yield;
"_*_".postln; 0.1.yield;
} })
);
)

 // this is the reason why 'wait' works the same (for numbers) like 'yield'
(
SystemClock.sched(0.0,
Routine({ loop {
"***".postln; 0.5.wait;
"_*_".postln; 0.1.wait;
```

```
} })
);
)
```

Streams that return **numbers** can be played directly with the **play** message:

play(clock, quant)
clock: a Clock, TempoClock by default
quant: either a number**n** (quantize to **n** beats)
or an array**[n, m]** (quantize to **n** beats, with offset **m**)

```
 // play at the next beat, with offset 0.4
 (
 Routine({ loop {
"***".postln; 0.5.wait;
"_*_".postln; 0.1.wait;
} }).play(quant:[1, 0.4]);
)
```

Streams that return **Events** need to be wrapped in an **EventStreamPlayer**.
The Event's **delta** (can also be set by **dur**) is used as a scheduling beats value:

```
     // play at the next beat, with offset 0.4
 (
 Routine({ loop {
"///".postln; (delta:0.5).yield;
"_/_".postln; (delta:  0.1).wait;
} }).asEventStreamPlayer.play;
)
```

Iteration

do (function)

iterate until a nil is encountered
beware: applying do to an endless stream will lock up the interpreter!

Where **do** effectively 'plays' a stream by iterating all of its contects, the following messages create a stream by filtering another stream in some way.

collect (function)
iterate indefinitely

reject (function)
return only those elements for which function.value(element) is false

select (function)
return only those elements for which function.value(element) is true

dot(function, stream)
return **function.value(this.next, stream.next)** for each element

interlace(function, stream)
iterate all of stream for each element of this. Combine the values using function.

appendStream(stream)
append stream after thisreturns nil. The same like ++

embedInStream(inval)
iterate all of this from within whatever Stream definition it is called.

trace(key, printStream, prefix)
print out the results of a stream while returning the original values

key: when streaming events, post only this key.
printStream: printOn this stream (default: Post)
prefix: string added to the printout to separate different streams

Composite Streams

Routines can be embedded in each other, using **embedInStream**:

```
  // example
(
x = Routine({
2.do({
[1,2,3,4].do({ arg i; i.yield; });
})
});
y = Routine({
100.yield;
30.yield;
x.embedInStream;
440.yield;
1910.yield
});
17.do({ y.next.postln });
)
```

Routines can be **concatenated** just like Streams:

```
(
x = Routine({
2.do({
[1,2,3,4].do({ arg i; i.yield; });
})
});
y = Routine({
100.yield;
30.yield;
});
z = x ++ y;
17.do({ z.next.postln });
)
```

Routines can be combined with the **composition** operator <>

```
(
```

```
x = Routine({ arg inval;
2.do({

[1,2,3,4].do({ arg i;
if(inval.isNil) { nil.alwaysYield };
inval = (i * inval).yield;
});
})
});

y = Routine({
100.yield;
30.yield;
4.do { 1.0.rand.yield };
});

z = x <> y;
 17.do({ z.value.postln }); // call .value here, as this is a function.
)
```

Composite Streams can be defined as combinations of Streams using the unary and binary
messages.

Unary messages

Streams support most of the unary messages defined in AbstractFunction:

```
(
a = Routine({ 20.do({ 33.rand.yield }) });
b = Routine({ [-100,00,300,400].do({ arg v; v.yield}) });

 c = b.neg; // define a composite stream

 // enumerate and perform all of the unary messages :
[
  \neg  \reciprocal  \bitNot  \abs  \asFloat  \asInteger  \ceil
```

```
  \floor  \frac  \sign  \squared  \cubed  \sqrt  \exp  \midicps
  \cpsmidi  \midiratio  \ratiomidi  \ampdb  \dbamp  \octcps
  \cpsoct  \log  \log2  \log10  \sin  \cos  \tan  \asin  \acos  \atan
  \sinh  \cosh  \tanh  \rand  \rand2  \linrand  \bilinrand  \sum3rand
  \distort  \softclip  \coin  \even  \odd  \isPositive  \isNegative
  \isStrictlyPositive
]
.do({ arg msg;
postf("\n msg:  % \n", msg);
b.reset.perform(msg).do({arg v; v.post; " ".post;})
});
nil;

)
```

Binary messages

Streams support the following binary messages defined in AbstractFunction:

```
(
a = Routine({ 20.do({ 33.rand.yield }) });
b = Routine({ [-100,00,300,400].do({ arg v; v.yield}) });
[
 '+'  '-'  '*'  '/'  \div  '%'  '**'  \min  \max  '<'  '<='  '>'  '>='  '&amp;'  '| '
 \bitXor  \lcm  \gcd  \round  \trunc  \atan2
 \hypot  '>>'  '+>>'  \ring1  \ring2  \ring3  \ring4
 \difsqr  \sumsqr  \sqrdif  \absdif  \amclip
 \scaleneg  \clip2  \excess  '<!'  \rrand  \exprand
]
.do({ arg msg;
postf("\n msg:  % \n", msg);
b.reset.perform(msg).do({ arg v; v.post; " ".post; })
});
nil;
)
```

ID: 459

Understanding Streams, Patterns and Events - Part 1

The SuperCollider Pattern library provides a means of specifying dynamic structural transformations of musical processes. It provides similar capabilities as one finds in Nyquist, Elody, Siren, Kyma, HMSL, DMix, and Patchwork.

By using coroutines and streams rather than eager functional methods it is able to work in a lazy event by event method instead of the all-at-once method of Elody and Siren. It provides the kind of dynamic live control found in HMSL but with the more general event models of the others. In Nyquist and Siren certain transformation like Stretch and Transpose are specially coded into the framework. In SuperCollider Patterns, any parameter
may have transformations applied to it. The only one treated specially is time, so that parallel
streams can be merged.

In order to understand the framework, a number of concepts must be covered.
These concepts are embodied in the classes for Streams, Patterns, and Events.
You should learn these concepts in the order presented. The framework is built
up in layers. If you skip ahead to get to the cool stuff first, you will have missed
some important points.

Streams

A stream represents a lazy sequence of values. The next value in the sequence is obtained by
sending the message next to the stream object. The sequence can be restarted from the beginning by sending the message reset to the stream object. A stream can be of finite or infinite length. When a finite length stream has reached the end, it returns nil.

A stream can be any object that responds to the next and reset messages.
Any object that responds to these messages can act as a stream.
It happens that the class Object defines next and reset for all objects.
In Object, both next and reset are defined to return 'this'.
Thus any object is by default a stream that represents an infinite sequence of itself.

```
// 7 responds to next by returning itself
```

Stream and its subclasses

In addition to the default streams implemented by Object, there is a class Stream that provides more functionality such as math operations on streams and filtering of streams.

A generally useful subclass of Stream is the class FuncStream which allows the user to provide functions to execute in response to next and reset.
Here is a FuncStream that represents an infinite random sequence:

```
(
var a;
a = FuncStream.new({ #[1, 2, 3, 4].choose });
                          // print 5 values from the stream
)
```

Another useful subclass of Stream is Routine which is a special kind of function that can act like a Stream.
Routines are functions that can return a value from the middle and then be resumed from that
point when called again. The yield message returns a value from the Routine. The next time theRoutine
is called it begins by returning from the yield and continues from that point.
See the Routine help file.

Here is a Routine that represents a finite sequence of values:

```
(
var a;
a = Routine.new({
3.do({ arg i; i.yield; })
});
                         // print 4 values from stream
)
```

and another:

```
(
var a;
a = Routine.new({
```

```
3.do({ arg i;
(i+1).do({ arg j; j.yield; })
})
});
                        // print 8 values from stream
)
```

Math operations on Streams

Stream is a subclass of AbstractFunction which means that one can do math operations on streams to produce other streams.

Applying a unary operator to a stream:

```
(
var a, b;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
            // stream b is a square of the stream a
12.do({ b.next.postln; });
)
```

Using a binary operator on a stream:

```
(
var a, b;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
          // add a constant value to stream a
12.do({ b.next.postln; });
)
```

Using a binary operator on two streams:

```
(
```

```
var a, b, c;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
// b is a stream that counts from 100 to 280 by 20
b = Routine.new({
forBy (100,280,20, { arg i; i.yield })
});
           // add streams a and b
12.do({ c.next.postln; });
)
```

Filtering operations on streams

Streams respond to the messages collect, select, and reject by returning a new Stream.

The collect message returns a stream that is modified by a function in the same way as the collect message sent to a Collection returns a modified Collection.

```
(
var a, b;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
// b is a stream that adds 100 to even values
b = a.collect({ arg item; if (item.even, { item + 100 },{ item }); });
6.do({ b.next.postln; });
)
```

The select message creates a stream that passes only items that return true from a user supplied function.

```
(
var a, b;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
```

```
// b is a stream that only returns the odd values from stream a
b = a.select({ arg item; item.odd; });
6.do({ b.next.postln; });
)
```

The reject message creates a stream that passes only items that return false from a user supplied function.

```
(
var a, b;
// a is a stream that counts from 0 to 9
a = Routine.new({
10.do({ arg i; i.yield; })
});
// b is a stream that only returns the non-odd values from stream a
b = a.reject({ arg item; item.odd; });
6.do({ b.next.postln; });
)
```

Making Music with Streams

Here is a sound example to show how you might use Streams to generate musical material.

```
(
s = Server.local;
SynthDef( "Help-SPE1", { arg i_out=0, freq;
var out;
out = RLPF.ar(
LFSaw.ar( freq, mul:  EnvGen.kr( Env.perc, levelScale:  0.3, doneAction:  2 )),
LFNoise1.kr(1, 36, 110).midicps,
0.1
);
  // out = [out, DelayN.ar(out, 0.04, 0.04) ];
4.do({ out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4) });
Out.ar( i_out, out );
}).send(s);
)
```

```
(
// streams as a sequence of pitches
var stream, dur;
dur = 1/8;
stream = Routine.new({
loop({
if (0.5.coin, {
    // run of fifths:
24.yield;
31.yield;
36.yield;
43.yield;
48.yield;
55.yield;
});
rrand(2,5).do({
    // varying arpeggio
60.yield;
#[63,65].choose.yield;
67.yield;
#[70,72,74].choose.yield;
});
   // random high melody
rrand(3,9).do({  #[74,75,77,79,81].choose.yield });
});
});
 Routine
loop({
Synth( "Help-SPE1", [ \freq, stream.next.midicps ] );
   dur.wait; // synonym for yield, used by .play to schedule next occurence
})
}).play
)
```

Optional:
More about Streams can be learned from the book A Little Smalltalk by Timothy Budd. He calls them Generators and shows how they can be used to solve problems like the "eight queens" problem etc.

To go to the next file, double click on the] character to select the filename and type cmd-H:
[Streams-Patterns-Events2]

ID: 460

Understanding Streams, Patterns and Events - Part 2

Patterns

Often one wants to be able to create multiple streams from a single stream specification.
Patterns are just a way to make multiple Streams from a single specification, like a cookie cutter.
A pattern can be any object that responds to the asStream message by creating a Stream.
Once again there is a default implementation in class Object of asStream that simply returns the receiver as its own stream. Thus any object is by default a pattern that returns itself as a stream when sent the asStream message.

```
(
a = 7.asStream;
a.postln;
a.next.postln;
)
```

Pattern and its subclasses

There is a class named Pattern that provides more functionality for the concept of a pattern.

Pfunc is a Pattern that returns a FuncStream.
The same function arguments are supplied as are supplied to FuncStream.

```
(
var a, b;
a = Pfunc.new({ #[1, 2, 3, 4].choose });
                // make a stream from the pattern
                          // print 5 values from the stream

)
```

Prout is a Pattern that returns a Routine.

The same function argument is supplied as is supplied to Routine.

```
(
var a, b, c;
a = Prout.new({
3.do({ arg i; 3.rand.yield; })
});
// make two streams from the pattern
b = a.asStream;
c = a.asStream;
                        // print 4 values from first stream
                        // print 4 values from second stream
)
```

Pseries is a Pattern that generates an arithmetic series.

```
(
var a, b;
    Pseries              // stream starts at 10, steps by 3 and has length 8
b = a.asStream;
                         // print 9 values from stream
)
```

Pgeom is a Pattern that generates a geometric series.

```
(
var a, b;
// stream starts at 10, steps by factor of 3 and has length 8
a = Pgeom.new(10, 3, 8);
b = a.asStream;
                        // print 9 values from stream
)
```

Math operations on Patterns

Patterns also respond to math operators by returning patterns that respond to asStream with appropriately modified streams.

Applying a unary operator to a pattern

```
(
var a, b, c;
// a is a pattern whose stream counts from 0 to 9
a = Pseries.new(0,1,10);
                // pattern b is a square of the pattern a
c = b.asStream;
12.do({ c.next.postln; });
)
```

Using a binary operator on a pattern

```
(
var a, b, c;
// a is a pattern whose stream counts from 0 to 9
a = Pseries.new(0,1,10);
              // add a constant value to pattern a
c = b.asStream;
12.do({ c.next.postln; });
)
```

Filtering operations on patterns

Patterns also respond to the messages collect, select, and reject by returning a new Pattern.

The collect message returns a Pattern whose Stream is modified by a function in the same way
as the collect message sent to a Collection returns a modified Collection.

```
(
var a, b, c;
// a is a pattern whose stream counts from 0 to 9
a = Pseries.new(0,1,10);
// b is a pattern whose stream adds 100 to even values
b = a.collect({ arg item; if (item.even, { item + 100 },{ item }); });
c = b.asStream;
6.do({ c.next.postln; });
)
```

The select message creates a pattern whose stream passes only items that return true

from a
user supplied function.

```
(
var a, b, c;
// a is a pattern whose stream counts from 0 to 9
a = Pseries.new(0,1,10);
// b is a pattern whose stream only returns the odd values
b = a.select({ arg item; item.odd; });
c = b.asStream;
6.do({ c.next.postln; });
)
```

The reject message creates a pattern whose stream passes only items that return false from a
user supplied function.

```
(
var a, b, c;
// a is a pattern whose stream counts from 0 to 9
a = Pseries.new(0,1,10);
// b is a pattern whose stream that only returns the non-odd values
b = a.reject({ arg item; item.odd; });
c = b.asStream;
6.do({ c.next.postln; });
)
```

Making Music with Patterns

Here is a variation of the example given in part 1 that uses a Pattern to create two instances of
the random melody stream.

```
(
s = Server.local;
SynthDef( "Help-SPE2", { arg i_out=0, i_dur=1, freq;
var out;
out = RLPF.ar(
LFSaw.ar( freq ),
```

```
LFNoise1.kr(1, 36, 110).midicps,
0.1
) * EnvGen.kr( Env.perc, levelScale:  0.3,
timeScale:  i_dur, doneAction:  2 );
   //out = [out, DelayN.ar(out, 0.04, 0.04) ];
4.do({ out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4) });
Out.ar( i_out, out );
}).send(s);
)
(
// streams as a sequence of pitches
var pattern, streams, dur, durDiff;
dur = 1/7;
durDiff = 3;
pattern = Prout.new({
loop({
if (0.5.coin, {
#[ 24,31,36,43,48,55 ].do({ arg fifth; fifth.yield });
});
rrand(2,5).do({
    // varying arpeggio
60.yield;
#[63,65].choose.yield;
67.yield;
#[70,72,74].choose.yield;
});
   // random high melody
rrand(3,9).do({  #[74,75,77,79,81].choose.yield });
});
});
streams = [
(pattern - Pfunc.new({ #[12, 7, 7, 0].choose })).midicps.asStream,
pattern.midicps.asStream
];
 Routine
loop({
Synth( "Help-SPE2", [ \freq, streams.at(0).next, \i_dur, dur * durDiff  ] );
durDiff.do({
Synth( "Help-SPE2", [ \freq, streams.at(1).next, \i_dur, dur  ] );
dur.wait;
```

```
});
})
}).play
)
```

To go to the next file, double click on the] to select the filename and type cmd-H: [Streams-Patterns-Events3]

ID: 461

Understanding Streams, Patterns and Events - Part 3

ListPatterns

ListPatterns are Patterns that iterate over arrays of objects in some fashion. All ListPatterns have in common the instance variables list and repeats. The list variable is some Array to be iterated over. The repeats variable is some measure of the number of times to do something, whose meaning varies from subclass to subclass. The default value for repeats is 1.

Pseq is a Pattern that cycles over a list of values. The repeats variable gives the number of times to repeat the entire list.

```
////////////////////////////////////////////////////////////
// Note:  This SynthDef used throughout this document
(
s = Server.local;
SynthDef  "Help-SPE3-SimpleSine"
arg freq, dur=1.0;
var osc;
osc = SinOsc.ar( [freq,freq+0.05.rand] ) * EnvGen.ar(
Env.perc, doneAction:  2, levelScale:  0.3, timeScale:  dur
);
Out.ar(0,osc);
}).send(s);
)
////////////////////////////////////////////////////////////

(
var a, b;
a = Pseq.new(#[1, 2, 3], 2); // repeat twice
b = a.asStream;
7.do({ b.next.postln; });
)
```

Pseq also has an offset argument which gives a starting offset into the list.

```
(
var a, b;
a = Pseq.new(#[1, 2, 3, 4], 3, 2); // repeat 3, offset 2
b = a.asStream;
13.do({ b.next.postln; });
)
```

You can pass a function for the repeats variable that gets evaluated when the stream is created.

```
(
var a, b;
a = Pseq.new(#[1, 2], { rrand(1, 3) }); // repeat 1,2, or 3 times
b = a.asStream;
7.do({ b.next.postln; });
)
```

If you specify the value inf for the repeats variable, then it will repeat indefinitely.

```
(
var a, b;
a = Pseq.new(#[1, 2, 3], inf); // infinite repeat
b = a.asStream;
10.do({ b.next.postln; });
)
```

Pseq used as a sequence of pitches:

Remember that math operations like midicps can be used on streams.
The alternative Pseq(...).midicps.asStream is also possible because
both pattern and stream inherit from AbstractFunction for which
midicps is a method. (midicps converts a midi value to cycles per second or Hz)

```
(
var a, d;
a = Pseq(#[60, 61, 63, 65, 67, 63], inf ).asStream.midicps;
d = 0.3;
Task
12.do({
```

```
Synth("Help-SPE3-SimpleSine", [ \freq, a.next, \dur, d ]);
d.wait;
});
}).play
)
```

Pser is like Pseq, however the repeats variable gives the number of items returned instead of
the number of complete cycles.

```
(
var a, b;
a = Pser.new(#[1, 2, 3], 5); // return 5 items
b = a.asStream;
6.do({ b.next.postln; });
)
```

Prand returns one item from the list at random for each repeat.

```
(
var a, b;
a = Prand.new(#[1, 2, 3, 4, 5], 6); // return 6 items
b = a.asStream;
7.do({ b.next.postln; });
)
```

Prand used as a sequence of pitches:

```
(
var a;
a = Prand(#[60, 61, 63, 65], inf).midicps.asStream;
Task
12.do({
  Synth "Help-SPE3-SimpleSine"  \freq
d.wait;
});
}).play;
)
```

Pxrand, like Prand, returns one item from the list at random for each repeat, but Pxrand

never
repeats the same element twice in a row.

```
(
var a, b;
a = Pxrand.new(#[1, 2, 3], 10); // return 10 items
b = a.asStream;
11.do({ b.next.postln; });
)
```

Pxrand used as a sequence of pitches:

```
(
var a;
a = Pxrand(#[60, 61, 63, 65], inf).midicps.asStream;
Task
12.do({
  Synth "Help-SPE3-SimpleSine"  \freq
0.8.wait;
});
}).play;
)
```

Pshuf iterates over the list in scrambled order. The entire scrambled list is repeated in the same order the number of times given by the repeats variable.

```
(
var a, b;
a = Pshuf.new(#[1, 2, 3, 4], 3);
b = a.asStream;
13.do({ b.next.postln; });
)
```

Pshuf used as a sequence of pitches:

```
(
var a, b;
a = Pshuf(#[60, 61, 65, 67], inf).midicps.asStream;
Task
12.do({
```

```
  Synth "Help-SPE3-SimpleSine"  \freq
0.5.wait;
});
}).play;
)
```

Nesting Patterns

If a Pattern encounters another Pattern in its list, it embeds that pattern in its output. That is, it creates a stream on that pattern and iterates that pattern until it ends before moving on.
For example here is one pattern nested in another.

```
(
var a, b;
a = Pseq.new([1, Pseq.new([100,200], 2), 3], 3);
b = a.asStream;
19.do({ b.next.postln; });
)
```

Pseqs nested in a Prand:

```
(
var a, b;
a = Prand.new([
Pseq.new([1, 2], 2),
Pseq.new([3, 4], 2),
Pseq.new([5, 6], 2)
], 3);
b = a.asStream;
13.do({ b.next.postln; });
)
```

Nested sequences of pitches:

```
(
var a;
a = Prand([
Pseq(#[60, 61, 63, 65, 67, 63]),
Prand(#[72, 73, 75, 77, 79], 6),
```

```
Pshuf(#[48, 53, 55, 58], 2)
], inf
).midicps.asStream;
Task
loop({
  Synth  "Help-SPE3-SimpleSine"   \freq
0.3.wait;
});
}).play;
)
```

Math operations on ListPatterns

Pattern b plays pattern a once normally, once transposed up a fifth and once transposed up a fourth.

```
(
var a, b;
a = Pseq(#[60, 62, 63, 65, 67, 63]);
b = Pseq([ a, a + 7, a + 5], inf).asStream;
Task
24.do({
Synth("Help-SPE3-SimpleSine", [ \freq, b.next.midicps ]);
0.3.wait;
});
}).play;
)
```

Adding two patterns together. The second pattern transposes each fifth note of the first pattern
down an octave.

```
(
var a;
a = Pseq(#[60, 62, 63, 65, 67, 63], inf) + Pseq(#[0, 0, 0, 0, -12], inf);
a = a.asStream.midicps;
Task
25.do({
  Synth "Help-SPE3-SimpleSine"  \freq
0.3.wait;
```

```
});
}).play;
)
```

Making Music with ListPatterns

Here is the same example given in part 2 rewritten to use ListPatterns.
It uses nested patterns and results in much more concise code.
SuperCollider allows you to write SomeClass.new(params) as SomeClass(params) eliminating the ".new" . This can make code like the pattern examples below, which create
a lot of objects, more readable.

```
(
SynthDef  "Help-SPE3-Allpass6"    arg
var out, env;
out = RLPF.ar(
LFSaw.ar( freq, mul:  EnvGen.kr( Env.perc, levelScale:  0.3, doneAction:  2 ) ),
LFNoise1.kr(1, 36, 110).midicps,
0.1
);
6.do({ out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4) });
Out.ar( 0, out  );
}).send(s)
)

(

var freqStream;

freqStream = Pseq([
 Prand
  nil   // a nil item reached in a pattern causes it to end
Pseq(#[24, 31, 36, 43, 48, 55]);
]),
Pseq([ 60, Prand(#[63, 65]), 67, Prand(#[70, 72, 74]) ], { rrand(2, 5) }),
Prand(#[74, 75, 77, 79, 81], { rrand(3, 9) })
], inf).asStream.midicps;

Task
```

```
loop({
Synth( "Help-SPE3-Allpass6", [\freq, freqStream.next ]);
0.13.wait;
});
}).play;

)
```

Here is an example that uses a Pattern to create a rhythmic solo. The values in the pattern
specify the amplitudes of impulses fed to the Decay2 generator.

```
(
SynthDef  "Help-SPE3-Mridangam"    arg
var out;

out = Resonz.ar(
WhiteNoise.ar(70) * Decay2.kr( t_amp, 0.002, 0.1 ),
60.midicps,
0.02,
4
).distort * 0.4;

Out.ar( 0, out  );
DetectSilence.ar( out,  doneAction:  2 );
}).send(s);
SynthDef  "Help-SPE3-Drone"
var out;
out = LPF.ar(
Saw.ar([60, 60.04].midicps)
+
Saw.ar([67, 67.04].midicps),
108.midicps,
0.007
);
Out.ar( 0, out  );
}).send(s);
)
```

```
(
// percussion solo in 10/8

var stream, pat, amp;

pat = Pseq([
Pseq(#[0.0], 10),

 // intro
Pseq(#[0.9, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 2),
Pseq(#[0.9, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.2, 0.0, 0.0], 2),
Pseq(#[0.9, 0.0, 0.0, 0.2, 0.0, 0.2, 0.0, 0.2, 0.0, 0.0], 2),
Pseq(#[0.9, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.2, 0.0, 0.2], 2),

 // solo
 Prand
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.0, 0.2, 0.0, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.2, 0.0, 0.7, 0.0, 0.2, 0.0, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.0, 0.2, 0.0, 0.7, 0.0, 0.2]),
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.2, 0.2, 0.0, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.0, 0.2, 0.2, 0.7, 0.2, 0.0]),
Pseq(#[0.9, 0.2, 0.2, 0.7, 0.2, 0.2, 0.2, 0.7, 0.2, 0.2]),
Pseq(#[0.9, 0.2, 0.2, 0.7, 0.2, 0.2, 0.2, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.2, 0.2, 0.2, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.4, 0.0, 0.4, 0.0, 0.4, 0.0, 0.4, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.4, 0.0, 0.0, 0.4, 0.2, 0.4, 0.2]),
Pseq(#[0.9, 0.0, 0.2, 0.7, 0.0, 0.2, 0.0, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.7, 0.0, 0.0, 0.0, 0.7, 0.0, 0.0]),
Pseq(#[0.9, 0.7, 0.7, 0.0, 0.0, 0.2, 0.2, 0.2, 0.0, 0.0]),
Pseq(#[0.9, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
], 30),

 // tehai :  7 beat motif 3 times sharing 1st beat with next 7x3
 // and again the third time:
  //   123456712345671234567                    123456712345671234567
 //                        123456712345671234567
 //   !                              !                              !                              !
  //   12345678901234567890123456789012345678901234567890123456789012345678901
Pseq(#[2.0, 0.0, 0.2, 0.5, 0.0, 0.2, 0.9,
```

```
1.5, 0.0, 0.2, 0.5, 0.0, 0.2, 0.9,
1.5, 0.0, 0.2, 0.5, 0.0, 0.2], 3),
Pseq(#[5], 1), // sam

Pseq(#[0.0], inf)
]);

stream = pat.asStream;

Task
 Synth "Help-SPE3-Drone"
loop({
if( ( amp = stream.next ) > 0,
   { Synth "Help-SPE3-Mridangam"    \t_amp
);
(1/8).wait;
})
}).play

)
```

To go to the next file, double click on the] to select the filename and type cmd-H: [Streams-Patterns-Events4]

ID: 462

Understanding Streams, Patterns and Events - Part 4

The preceeding sections showed how to use Streams and Patterns to generate complex sequences of values for a single parameter at a time.
This section covers Environments and Events, which are used to build a symbolic event framework for patterns, allowing you to control all aspects of a composition using patterns.

Environment

An Environment is an IdentityDictionary mapping Symbols to values.
There is always one current Environment which is stored in the currentEnvironment class variable of class Object.

Symbol and value pairs may be put into the current Environment as follows:

```
currentEnvironment.put(\myvariable, 999);
```

and retrieved from the current Environment as follows:

```
currentEnvironment.at(\myvariable).postln;
```

The compiler provides a shorthand for the two constructs above .

```
myvariable = 888;
```

is equivalent to:

```
currentEnvironment.put(\myvariable, 888);
```

and:

```
myvariable.postln;
```

is equivalent to:

```
currentEnvironment.at(\myvariable).postln;
```

Making an Environment

Environment has a class method **make** which can be used to create an Environment and fill it with values. What **make** does is temporarily replace the current Environment with a new one, call your function where you fill the Environment with values, then it replaces the previous current Environment and returns you the new one.

```
(
var a;
a = Environment.make({
a = 100;
b = 200;
c = 300;
});
a.postln;
)
```

Using an Environment

The instance method **use** lets you temporarily replace the current Environment with one you have made.
The **use** method returns the result of your function instead of the Environment like **make** does.

```
(
var a;
a = Environment.make({
a = 10;
b = 200;
c = 3000;
});
a.use({
a +  b +  c
}).postln;
)
```

There is also a **use** class method for when you want to make and use the result from an Environment directly.

```
(
var a;
    Environment
a = 10;
b = 200;
c = 3000;
a +  b +  c
}).postln;
)
```

Calling Functions with arguments from the current Environment

It is possible to call a Function and have it look up any unspecified argument values from the
current Environment. This is done with the **valueEnvir** and **valueArrayEnvir** methods. These methods will, for any unspecified argument value, look in the current Environment for
a symbol with the same name as the argument. If the argument is not found then whatever the
function defines as the default value for that argument is used.

```
(
var f;

// define a function
f = { arg x, y, z; [x, y, z].postln; };

Environment
x = 7;
y = 8;
z = 9;

f.valueEnvir(1, 2, 3); // all values supplied
 f.valueEnvir(1, 2); // z is looked up in the current Environment
 f.valueEnvir(1);  // y and z are looked up in the current Environment
 f.valueEnvir;   // all arguments are looked up in the current Environment
 f.valueEnvir(z:  1); // x and y are looked up in the current Environment
});
)
```

Here is a somewhat contrived example of how the Environment might be used to manufacture SynthDefs.
Even though the three functions below have the freq, amp and pan args declared in different orders it does not matter, because valueEnvir looks them up in the environment.

```
(
var a, b, c, n;

n = 40;
a = { arg freq, amp, pan;
Pan2.ar(SinOsc.ar(freq), pan, amp);
};
b =  { arg amp, pan, freq;
Pan2.ar(RLPF.ar(Saw.ar(freq), freq * 6, 0.1), pan, amp);
};
c =  { arg pan, freq, amp;
Pan2.ar(Resonz.ar(GrayNoise.ar, freq * 2, 0.1), pan, amp * 2);
};

Task
n.do({ arg i;
  SynthDef "Help-SPE4-EnvirDef-"
var out;
   Environment
     // set values in the environment
freq = exprand(80, 600);
amp = 0.1 + 0.3.rand;
pan = 1.0.rand2;

     // call a randomly chosen instrument function
     // with values from the environment
out = [a,b,c].choose.valueEnvir;
});
out = CombC.ar(out, 0.2, 0.2, 3, 1, out);
out = out * EnvGen.kr(
Env.sine, doneAction:  2, timeScale:  1.0 + 6.0.rand, levelScale:  0.3
);
Out.ar( 0, out );
```

```
}).send(s);
0.02.wait;
});
loop({
Synth( "Help-SPE4-EnvirDef-" ++ n.rand.asString );
(0.5 + 2.0.rand).wait;
});
}).play;
)
```

Event

The class Event is a subclass of Environment. Events are mappings of Symbols representing
names of parameters for a musical event to their value. This lets you put any information you
want into an event.

The class getter method **default** retrieves the default prototype event which has been initialized with
values for many useful parameters. It represents only one possible event model. You are free to create
your own, however it would be good to understand the one provided first so that you can see what can be done.

A prototype event is a default event which will be transformed by the streams returned by patterns.
Compositions produced by event patterns are created entirely from transformations of copies of a single protoEvent.
It's all a part of the Big Note, but don't tell the pigs and ponies.

Value Patterns, Event Patterns and Pbind

The patterns discussed in parts 2 and 3 are known as "value patterns" because their streams
return a single value for each call to **next**. Here we introduce "event patterns" which once turned
into streams, return an Event for each call to **next**.

The class **Pbind** provides a bridge between value patterns and event patterns. It binds symbols in
each event to values obtained from a pattern. Pbind takes arguments in pairs, the first of a pair
being a Symbol and the second being a value Pattern. Any object can act as a Pattern, so you can use constants as the pattern (see \amp in the example below).

The Pbind stream returns nil whenever the first one of its streams ends.

```
Pbind( \freq, Pseq([440,880]) ).play
```

An event stream is created for a Pattern by sending it the **asStream** message. What Pbind does is to produce a stream which puts the values for its symbols into the event, possibly overwriting previous bindings to those symbols:

```
t  =  Pbind( \freq, Pseq([440,880]) ).asStream;
t.next(Event.default);
t.next(Event.default);
t.next(Event.default);
```

When calling Pattern-play an EventStreamPlayer is automatically generated which handles scheduling as well as passing the protoEvent into the event stream.

EventStreamPlayer is a subclass of **PauseStream**. A PauseStream is just a wrapper for a stream allowing to play, stop, start it, etc...
EventStreamPlayers are initialized using the event stream returned by Pbind-asStream, as well as with a protoEvent. The EventStreamPlayer passes in a **protoEvent**, at each call to **next** on the Pbind stream. The Pbind stream copies the event to pass down and back up the tree of pattern streams so that each stream can modify it.
An EventStreamPlayer is itself a stream which returns scalars which are used by the clock to schedule its next invocation. At every call to EventStreamPlayer-next by the clock, the player gets its delta values by querying the Event after it has been returned by the Pbind stream traversal.

Changes in SC3

In SC2 you called asEventStream on an Pattern you'd get a stream which actually returned events.
Now if you want an event stream proper you call asStream on the Event Pattern.
This will give you a stream of events which you can then use to initialize an EventStreamPlayer object. You don't however need to worry about that because it is usually done for you. Also changed is that you do not pass
in your protoEvent through the asStream method. It is passed in for you by the EventStreamPlayer at each call
to next on the stream.

Here you can see what the stream returned from a Pbind looks like.

```
(
var pattern, stream;

 // bind Symbol xyz to values obtained from a pattern
pattern = Pbind(
\xyz, Pseq([1, 2, 3])
);

 // create a stream of events for the Pbind pattern.
stream = pattern.asStream;

 // event Streams require a prototype event as input.
  // this example uses an empty Event as a prototype
4.do({  stream.next(Event.new).postln; });
)
```

Here is an example with more bindings.

```
(
var pattern, stream;

pattern = Pbind(
\abc, Prand([6, 7, 8, 9], inf ),
\xyz, Pseq([1, 2, 3], 2 ),
 \uuu      // a constant represents an infinite sequence of itself
);

stream = pattern.asStream;
```

```
7.do({  stream.next(Event.new).postln; });
)
```

The ListPatterns discussed in part 3 can be put around Event Streams to create sequences of Event Streams.

```
(
var pattern, stream;
pattern =
 Pseq
Pbind( \abc, Pseq([1, 2, 3])),
Pbind( \def, Pseq([4, 5, 6])),
Pbind( \xyz, Pseq([7, 8, 9]))
]);
stream = pattern.asStream;
10.do({  stream.next(Event.new).postln; });
)

(
var pattern, stream;
pattern =
 Prand
Pbind( \abc, Pseq([1, 2, 3])),
Pbind( \def, Pseq([4, 5, 6])),
Pbind( \xyz, Pseq([7, 8, 9]))
], 3);
stream = pattern.asStream;
10.do({  stream.next(Event.new).postln; });
)
```

To go to the next file, double click on the] to select the filename and type cmd-H:
[Streams-Patterns-Events5]

ID: 463

Understanding Streams, Patterns and Events - Part 5

More about the default Event

protoEvents

The protoEvent contains default values for many useful parameters.
The default protoEvent is Event.default. It provides default bindings for duration, envelope,
instrument, making a very simple Pattern directly playable:

```
(
// an endless sequence of middle Cs
Pbind.new.play
)
```

By adding other bindings, you can override the defaults in the protoEvent.

```
(
// an endless sequence of middle Cs
Pbind( \dur, 0.25 ).play
)

(
Pbind
\dur, 0.125,
\legato, 0.2,
\midinote, Pseq(#[60, 62, 64, 65, 67, 69, 71, 72], inf)
).play
)
```

finish

Event.default contains a function bound to the Symbol 'finish' which is called for each new event generated in order to complete any computations that

depend on the other values in the event.

The pitch model

Event.default implements a multi level pitch model which allows composition using modal scale degrees, equal division note values, midi note values, or frequencies in Hertz. These different ways of specifying the pitch can all be used interchangably.

The way this works is due to the default values bound to the Symbols of the pitch model.

The lowest level Symbol in the pitch model is 'freq'. The default binding for 'freq' is a Function
which calculates the frequency by getting the value of 'midinote', adding a transpose value
and converting it to Hertz using midicps.

```
freq = {
( midinote.value +  ctranspose).midicps;
};
```

If you compose with 'freq' directly then this default function is overridden.

```
(
Pbind
\dur, 0.25,
\freq, Pseq(#[300, 400, 500, 700, 900], inf)
).play;
)
```

Event.default's 'finish' function sends the value message to the current binding of 'freq' in order to get the value for the frequency and adds a detune value to it which transposes the
frequency in Hertz.

```
(
Pbind
\dur, 0.25,
\detune, -20,
\freq, Pseq(#[300, 400, 500, 700, 900], inf)
).play
```

```
)
```

The next level is 'midinote' which is by default bound to this function:

```
midinote = {
( note.value +  gtranspose + ( octave * divs) +  root)
* 12.0 /  stepsPerOctave;
};
```

This function gets the value bound to 'note' which is a value expressed in some equal temperament,
not necessarily 12. It adds a gamut transpose value 'gtranspose', and scales from the number of
notes per octave being used into 12 notes per octave MIDI key values. If you compose with 'midinote'
directly then that will override this function.

```
(
Pbind
\dur, 0.2,
\midinote, Pseq([ Pshuf(#[60, 61, 62, 63, 64, 65, 66, 67], 3) ], inf)
).play
)
```

Another level higher is 'note' which is defined by default by this function:

```
note = {
var divs;
divs =  stepsPerOctave;
( degree +  mtranspose).degreeToKey( scale, divs);
};
```

This function derives the note value from the next higher level variables which specify a pitch from a scale. These variables are defined as follows:

```
stepsPerOctave = 12.0;
```

The number of equal divisions of an octave for this tuning. The equal temperament defined by this
variable is known as the gamut.

If you wanted to work in cents for example you could set this to 1200.0.

```
octave = 5.0;
```

The current octave. Middle C is the lowest note in octave 5.

```
root = 0.0;
```

The root of the scale given in equal divisions defined by stepsPerOctave.

```
scale = #[0, 2, 4, 5, 7, 9, 11]; // diatonic major scale
```

A set of scale pitches given in equal divisions defined by stepsPerOctave.

```
degree = 0;
```

A scale degree index into the scale. 0 is the root and the scale wraps in the manner defined by degreeToKey.

```
mtranspose = 0;
```

A modal transposition value that is added to the scale degree.

```
gtranspose = 0;
```

A gamut transposition value that is added to the gamut pitch.

```
ctranspose = 0;
```

A chromatic transposition value expressed in semitones.

Pitch model Examples:

```
(
// a simple scale degree sequence
Pbind
  // -7 is 8ve below, -3 is a 4th below,
  // 0 is root, 2 is 3rd above, 4 is 5th above, 7 is 8ve above.
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], inf),
\dur, 0.15
```

```
).play
)


(
// change the octave
Pbind
\dur, 0.15,
 \octave
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], inf)
).play
)


(
// change  the scale
Pbind
\dur, 0.15,
\scale, [0, 2, 3, 5, 7, 8, 10],
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], inf)
).play
)


(
// modal transposition
var notes;
notes = Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1);
Pseq
 Pbind
\dur, 0.15,
  \mtranspose
\degree, notes
),
 Pbind
\dur, 0.15,
  \mtranspose
\degree, notes
),
 Pbind
```

```
\dur, 0.15,
  \mtranspose
\degree, notes
)
], inf).play
)


(
// chromatic transposition
var notes;
notes = Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1);
Pseq
 Pbind
\dur, 0.15,
  \ctranspose
\degree, notes
),
 Pbind
\dur, 0.15,
  \ctranspose
\degree, notes
),
 Pbind
\dur, 0.15,
  \ctranspose
\degree, notes
)
], inf).play
)


(
// frequency detuning
var notes;
notes = Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1);
Pseq
 Pbind
\dur, 0.15,
  \detune
```

```
\degree, notes
),
 Pbind
\dur, 0.15,
\detune, 20,
\degree, notes
),
 Pbind
\dur, 0.15,
\detune, 40,
\degree, notes
)
], inf).play
)


(
// chords.  If an Array of pitches is returned by a Stream for pitch, then a chord
// will be played.
Pbind
\dur, 0.15,
 \degree  Pseq
Pshuf(#[-7,-3,0,2,4,7], 4)+[0,4],
Pseq( [0,1,2,3,4,5,6,7] )+[0,2]
], inf)
).play
)


(
// composing in non 12 equal temperaments.   72 tone equal temp.
Pbind
 \stepsPerOctave
\note, Pseq([
    //  1/1, 7/6, 3/2, 7/4, 9/8
Pseq([ [0,16,42,58,84], Pseq([ 0, 16, 42, 58, 72, 84 ], 2), [0,16,42,58,84] ], 1),
    //  1/1, 6/5, 3/2, 9/5, 9/8
Pseq([ [0,19,42,61,84], Pseq([ 0, 19, 42, 61, 72, 84 ], 2), [0,19,42,61,84] ], 1),
    //  1/1, 5/4, 3/2, 15/8, 9/8
Pseq([ [0,23,42,65,84], Pseq([ 0, 23, 42, 65, 72, 84 ], 2), [0,23,42,65,84] ], 1),
```

```
    //  1/1, 9/7, 3/2, 27/14, 9/8
Pseq([ [0,26,42,68,84], Pseq([ 0, 26, 42, 68, 72, 84 ], 2), [0,26,42,68,84] ], 1)
], inf),
\dur, Pseq([ 1.2, Pseq([0.15], 12), 1.2], inf)
).play
)
```

The duration model

Duration is expressed in beats and is bound to the 'dur' symbol.
The sustain time of a note can be expressed directly in beats or by
using a legato value which is multiplied by the note duration to get the sustain time.

```
(
// changing duration
Pbind
\dur, Pseq([ Pgeom(0.05, 1.1, 24), Pgeom(0.5, 0.909, 24) ], inf),
\midinote, Pseq(#[60, 58], inf)
).play
)
```

```
(
// changing legato value
Pbind
\dur, 0.2,
\legato, Pseq([ Pseries(0.05, 0.05, 40), Pseries( 2.05, -0.05, 40) ], inf),
\midinote, Pseq(#[48, 51, 55, 58, 60, 58, 55, 51], inf)
).play
)
```

To go to the next file, double click on the] to select the filename and type cmd-H:
[Streams-Patterns-Events6]

ID: 464

Understanding Streams, Patterns and Events - Part 6

Parallel Patterns

Ppar

The **Ppar** pattern allows you to merge multiple event streams to play in parallel.
Ppar is a ListPattern and so like most ListPatterns it takes two arguments, a **list** of event
patterns to play in parallel and a **repeats** count.
Ppar's child patterns must be event patterns. Using value patterns in a Ppar is an error because value patterns contain no duration data.
A Ppar is done when all of its subpatterns are done.

```
(
Ppar
Pbind(\dur, 0.2, \midinote, Pseq([62, 65, 69, 72], inf)),
Pbind(\dur, 0.4, \midinote, Pseq([50, 45], inf))
]).play
)



(
// Ppars can be nested
Ppar
 Pbind
\dur, Prand([0.2, 0.4, 0.6], inf),
\midinote, Prand([72, 74, 76, 77, 79, 81], inf),
\db, -26,
\legato, 1.1
),
 Pseq
Pbind(\dur, 3.2, \freq, Pseq([\rest]) ),
  Prand
Ppar([
```

```
Pbind(\dur, 0.2, \pan,  0.5, \midinote, Pseq([60, 64, 67, 64])),
Pbind(\dur, 0.4, \pan, -0.5, \midinote, Pseq([48, 43]))
]),
Ppar([
Pbind(\dur, 0.2, \pan,  0.5, \midinote, Pseq([62, 65, 69, 65])),
Pbind(\dur, 0.4, \pan, -0.5, \midinote, Pseq([50, 45]))
]),
Ppar([
Pbind(\dur, 0.2, \pan,  0.5, \midinote, Pseq([64, 67, 71, 67])),
Pbind(\dur, 0.4, \pan, -0.5, \midinote, Pseq([52, 47]))
])
], 12)
], inf)
], inf).play;
)
```

Ptpar

The **Ppar** pattern starts all of its subpatterns at the same time.
Ptpar pattern includes a start time parameter before each subpattern which allow the subpatterns to
be started at some time delay within the pattern.
The start time is given in beats.

```
(
var makePattern, durpat;

durpat = Pseq([ Pgeom(0.05, 1.1, 24), Pgeom(0.5, 0.909, 24) ], 2);

makePattern = { arg note, db, pan;
Pbind( \dur, durpat, \db, db, \pan, pan, \midinote, Pseq([note, note-4], inf) );
};

Ptpar
0.0, makePattern.value(53, -20, -0.9),
2.0, makePattern.value(60, -23, -0.3),
4.0, makePattern.value(67, -26,  0.3),
6.0, makePattern.value(74, -29,  0.9)
], inf).play;
```

```
)
```

The time argmuents are sent the 'value' message when the Ptpar pattern is started, so you may use functions to specify the times.

```
(
var makePattern, durpat;

durpat = Pseq([ Pgeom(0.05, 1.1, 24), Pgeom(0.5, 0.909, 24) ], 2);

makePattern = { arg note, db, pan;
Pbind( \dur, durpat, \db, db, \pan, pan, \midinote, Pseq([note, note-4], inf) );
};

Ptpar
{ 0.0 }, makePattern.value(53, -20, -0.9),
{ 8.0.rand }, makePattern.value(60, -23, -0.3),
{ 8.0.rand }, makePattern.value(67, -26,  0.3),
{ 8.0.rand }, makePattern.value(74, -29,  0.9)
], inf).play;

)
```

FilterPatterns and transformation

FilterPatterns take an existing pattern and apply some modification to its properties.

Padd, Pmul, Pset, Pstretch

There is a simpler way to write the modal transposition example given in part 5.
In fact the earlier examples are setting the values of mtranspose and ctranspose which is not
the best way to change those variables, because it wipes out any modifications to them by parent
patterns. It is better to take the current value of those properties and add a value to them.
The **Padd** filter takes the current value of a property and adds a value to it.

```
(
// modal transposition
var pattern;

// define the basic pattern
pattern = Pbind(
\dur, 0.15,
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1)
);

Pseq
 pattern,  // untransposed
 Padd \mtranspose               // modal transpose up 1 degree
 Padd \mtranspose              // modal transpose up 2 degrees
], inf).play
)
```

Similarly, **Pmul** multiplies the current value of a property by a value.
Pset sets the property to a value.
Pnot does a logical negation of a property with a Boolean value.

In order to process duration correctly **Pstretch** should be used.

```
(
// beat stretching using Pstretch
var pattern;

// define the basic pattern
pattern = Pbind(
\dur, 0.15,
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1)
);

Pseq
pattern,  // normal
 Pstretch                // stretch durations by a factor of 1/2
 Pstretch               // stretch durations by a factor of 2
], inf).play
)
```

Paddp, Pmulp, Psetp, Pstretchp

In fact there is an even shorter version of the modal transposition example.
Paddp reads one pattern to get values for adding to a property and plays the second pattern
once through modified with each new value.

```
(
// modal transposition
var pattern;

// define the basic pattern
pattern = Pbind(
\dur, 0.15,
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1)
);

Paddp
 \mtranspose     // property to be modified
 Pseq          inf    // a value pattern as a source of values for adding to mtranspose
 pattern    // the pattern to be modified
).play
)
```

Nested modifications:

```
(
// modal transposition
var pat1, pat2;

// define the basic pattern
pat1 = Pbind(
\dur, 0.15,
\degree, Pseq([ Pshuf(#[-7,-3,0,2,4,7], 4), Pseq([0,1,2,3,4,5,6,7]) ], 1)
);

pat2 = Paddp(
```

```
 \mtranspose    // property to be modified
 Pseq            // a value pattern as a source of values for adding to mtranspose
Ppar([
pat1,
Padd(\mtranspose, -3, pat1), // down a 4th
Padd(\mtranspose, 2, pat1)  // up a 3rd
])
);

Pseq
 pat1, // unmodified pattern
 pat2,  // parallel sequence
 Pstretch             // parallel sequence stretched by 3/2
], inf).play
)
```

Another example using Paddp:

```
(
var chord;
chord = Prand([[53, 58, 64],[53, 60, 64],[57,60,65]]);
Paddp(\ctranspose, Prand([-1,0,2,4,5], inf),
 Ppar
  Pbind  // melody part
\dur, Prand([0.2, 0.4, 0.6], inf),
\midinote, Pxrand([71, 72, 74, 76, 77, 79], 10),
\db, -26,
\legato, 1.1
),
  Pbind  // harmony part
\pan, 0.4,
\dur, Pseq([0.1, 0.5, 0.4, 0.6], 4),
\midinote, Pseq([chord,\rest,chord,\rest], 4)
),
  Pbind  // bass part
\pan, -0.4,
\dur, 0.4,
\midinote, Pseq([38, 45, 38, 36], 4)
)
```

```
])
).play
)


(
// chromatic transposition
var pattern;

// define the basic pattern
pattern = Pbind(
\dur, 0.1,
\degree, Pseq([0,1,2,3,4,5,6,7])
);

Paddp
 \ctranspose    // property to be modified
 Pseries           // a value pattern as a source of values for multiplying with ctranspose
 pattern   // the pattern to be modified
).play
)


(
// beat time stretching
var pattern;

// define the basic pattern
pattern = Pbind(
\dur, 0.1,
\degree, Pseq([0,1,2,3,4,5,6,7])
);

Pstretchp
 Pseq          inf    // a value pattern as a source of values for multiplying with stretch
 pattern    // the pattern to be modified
).play
)
```

Pbindf

Pbindf is like **Pbind** except that it merges all the bound symbols into events that it gets from a subpattern. It takes the same initial arguments in pairs as Pbind does, with an additional
pattern to be modified as the last argument.

```
(
var pattern;
pattern = Pbind( \midinote, Pseq(#[60, 62, 64, 65, 67, 69, 71, 72]) );

Pseq
Pbindf(pattern, \legato, 0.1, \dur, 0.2),
Pbindf(pattern, \legato, 1.0, \dur, 0.125),
Pbindf(pattern, \legato, 2.0, \dur, 0.3)
], inf).play
)
```

Patterns can be used as the arguments to Pbindf.

```
(
var pattern;
pattern = Pbind( \midinote, Pseq(#[60, 62, 64, 65, 67, 69, 71, 72, 74, 76, 77, 79]) );

Pseq
Pbindf(pattern,\legato, 0.1, \dur, Pgeom(0.3, 0.85, inf)),
Pbindf(pattern,\legato, 1.0, \dur, Pseq([0.3, 0.15], inf)),
Pbindf(pattern,\legato, 2.0, \dur, Pseq([0.2, 0.2, 0.4], inf))
], inf).play
)
```

To go to the next file, double click on the] to select the filename and type cmd-H:
[Streams-Patterns-Events7]

ID: 465

Understanding Streams, Patterns and Events - Part 7

Practical Considerations

Using your own instrument

```
(
SynthDef("Help-SPE7-BerlinB", { arg i_out=0, freq = 80, amp = 0.2, pan=0;
var out, a, b;
amp = Decay2.kr(Impulse.kr(0), 0.05, 8, amp);
out = RLPF.ar(
LFPulse.ar(freq, 0, SinOsc.kr(0.12,[0,0.5pi],0.48,0.5), amp),
freq * SinOsc.kr(0.21,0,4,8),
0.07
);
#a, b = out;
DetectSilence.ar(a, 0.0001, doneAction:  2);
Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+1])));
}).store;

SynthDef("Help-SPE7-CFString1", { arg i_out, freq = 360, gate = 1, pan, amp=0.1;
var out, eg, fc, osc, a, b, w;
fc = LinExp.kr(LFNoise1.kr(Rand(0.25,0.4)), -1,1,500,2000);
osc = Mix.fill(8, { LFSaw.ar(freq * [Rand(0.99,1.01),Rand(0.99,1.01)], 0, amp) }).distort * 0.2;
eg = EnvGen.kr(Env.asr(1,1,1), gate, doneAction:2);
out = eg * RLPF.ar(osc, fc, 0.1);
#a, b = out;
Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+0.3])));
}).store;
)

Pattern-play creates an EventStreamPlayer for you and also supplies a default
protoEvent.  If you were using your own event model you would just pass in your own
protoEvent to the play method.
```

```
(
Pbind
 \instrument  Prand  'Help-SPE7-BerlinB' 'Help-SPE7-CFString1'  inf
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, 0.8,
\octave, 3,
\amp, 0.03
        // this returns an EventStreamPlayer
)
```

Defining your own message bindings

NotePlayer uses a message function to compile it's message for the server, and no longer does a valueEnvir like in SC2, but instead calls 'use' on the event, and then fills a message with bindings which you need to specify. You can't just automatically add your own bindings to a Pbind and expect them to be passed on to the server. Here's an example:

```
(
SynthDef("Help-SPE4-CFString2", { arg i_out, freq = 360, gate = 1, pan, amp=0.1, dorkarg=1;
var out, eg, fc, osc, a, b, w;
fc = LinExp.kr(LFNoise1.kr(Rand(0.25,0.4)), -1,1,500,2000);
osc = Mix.fill(8, { LFSaw.ar(freq * [Rand(0.99,1.01),Rand(0.99,1.01)], 0, amp * dorkarg ) }).distort *
0.2;
eg = EnvGen.kr(Env.asr(1,1,1), gate, doneAction:2);
out = eg * RLPF.ar(osc, fc, 0.1);
#a, b = out;
Out.ar(i_out, Mix.ar(PanAz.ar(4, [a, b], [pan, pan+0.3])));
}).send(s);
)
```

As you can see I have added dorkarg to the arglist of the SynthDef from earlier.

```
(
Pbind
 \instrument  "Help-SPE4-CFString2"
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, 0.4,
\octave, 3,
\amp, 0.03,
```

```
 \dorkarg  Pseq         inf  // silence every second note - doesn't work
).play;
)
```

Surprisingly \dorkarg has not been defined by the default \msgFunc, so we have to supply a \msgFunc which does.

```
(
Pbind
 \instrument  "Help-SPE4-CFString2"
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, 0.4,
\octave, 3,
\amp, 0.03,
 \dorkarg  Pseq         inf   // silence every second note - now works
\msgFunc, { arg id, freq;
[[
9,  instrument, id, 0,  group,
\out,  out, \freq, freq, \amp,  amp, \pan,  pan, \vol,  vol, \dorkarg,  dorkarg
]];
}
).play;
)
```

This is quite clumsy and with some luck (read: work) will not always be the case so keep your eyes open for changes.

The other option you have if you will be using unspecified bindings, is of course to define an event with the appropriate msgFunc as default. Have a look at Event's source, it's easy, and it's cleaner than passing in the msgFunc every time.

Manipulating an EventStreamPlayer in Realtime

```
(
p = Pbind(
 \degree, Pwhite(0,12),
\dur, 0.2,
 \instrument  "Help-SPE4-CFString1"
);
```

```
// e is an EventStreamPlayer
e = p.play;
)

(
// you can change the stream at any point in time
e.stream = Pbind(
\degree, Pseq([0,1,2,4,6,3,4,8],inf),
\dur, Prand([0.2,0.4,0.8],inf),
\amp, 0.05,
 \octave
 \instrument  'Help-SPE4-BerlinB'  // you can also use a symbol
 \ctranspose
).asStream;
)

(
e.stream = Pbind(
 [\degree  \dur   Pseq
[
Pseq([[0,0.1],[2,0.1],[3,0.1],[4,0.1],[5,0.8]],2),
Ptuple([Pxrand([6,7,8,9],4), 0.4]),
Ptuple([Pseq([9,8,7,6,5,4,3,2]), 0.2])
], inf
 ),
\amp, 0.05,
 \octave
 \instrument  "Help-SPE4-CFString1"
).asStream;
)
```

The following methods are possible because an EventStreamPlayer is a PauseStream:

```
        // keeps playing, but replaces notes with rests

e.unmute;

          // reset the stream.
```

```
          // will resume where paused.

e.resume;

         // will reset before resume.

e.resume;
```

ID: 466

Patterns/Streams Help

For an overview click on the right bracket and Command-Shift-?

[Streams-Patterns-Events1] - Streams & Routines
[Streams-Patterns-Events2] - Patterns Introduction
[Streams-Patterns-Events3] - ListPatterns
[Streams-Patterns-Events4] - Environment & Event
[Streams-Patterns-Events5] - Event.default
[Streams-Patterns-Events6] - Parallel Patterns
[Streams-Patterns-Events7] - Practical Considerations

ListPatterns

Pseq
Pser
Prand
Pwrand
Pxrand
Pshuf
Place
Ptuple
Pslide
Pfsm
Place
...

FilterPatterns

Pseed
Prewrite
Pswitch
Pswitch1
Pn
Pstutter
Pfin

Psync
Pcollect
Pselect
Preject

PdurStutter
Pconst
Pwrap
PdegreeToKey
Pavaroh

event stream specific filter patterns

Pset
Pfset
Pmul
Padd
Psetp
Pmulp
Paddp
Pfindur

other Patterns

Ppatmod
Plazy
Pbind
Phid
PstepNadd
PstepNfunc

Streams

BinaryOpStream
UnaryOpStream
EventStream
EventStreamPlayer

to be continued...

ID: 467

TabFileReader

reads tab/return delimited files into 2D arrays.

*read(path, skipEmptyLines)

```
(
// write a test file:
    File "TabDelTest.sc"  "w"
f.write(
"Some tab- delimited items in line 1


and then some more in line 3
"
);
f.close;
)
// open file, read and put strings into array, close file.
x = TabFileReader      "TabDelTest.sc"

// can skip empty lines:
x = TabFileReader      "TabDelTest.sc", true

// do file open/close by hand if you prefer:
f = File("TabDelTest.sc", "r"); f.isOpen;
    TabFileReader
t.read;
f.close;

(
// write a test file with numbers:
    File "TabDelTestNum.sc"  "w"

(1..10).do { | n| f.write(n.asString ++ Char.tab); };
f.close;
)

x = TabFileReader      "TabDelTestNum.sc"
x.collect(_.collect(_.interpret)); // convert to numbers.
```

```
// or you can do it immediately:
x = TabFileReader.readInterpret("TabDelTestNum.sc").postcs;

(
// write a test file with numbers:
    File "TabDelTestNum.sc"  "w"

(1..100).do { | n|
f.write(n.asString ++ if (n % 10 != 0, Char.tab, Char.nl)); };
f.close;
)


x = TabFileReader.readInterpret("TabDelTestNum.sc").postln;
```

ID: 468

UnaryOpStream

Superclass: Stream

A UnaryOpStream is created as a result of a unary math operation on a Stream. It is defined to respond to **next** by returning the result of the math operation on the **next** value from the stream. It responds to **reset** by resetting the Stream.

```
(Routine.new({ 6.do({ arg i; i.yield; })}).squared).dump
```

```
(
x = (Routine.new({ 6.do({ arg i; i.yield; })}).squared);
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
x.next.postln;
)
```

25 UGens

25.1 Analysis

ID: 469

Amplitude amplitude follower

Amplitude.kr(input, attackTime, releaseTime, mul, add)

Tracks the peak amplitude of a signal.
input - input signal.
attackTime - 60dB convergence time for following attacks.
releaseTime - 60dB convergence time for following decays.

```
(
// use input amplitude to control Pulse amplitude - use headphones to prevent feedback.
SynthDef "help-Amplitude"   arg
Out.ar(out,
Pulse.ar(90, 0.3, Amplitude.kr(AudioIn.ar(1)))
)
}).play;

)



(
// use input amplitude to control SinOsc frequency - use headphones to prevent feedback.
SynthDef "help-Amplitude"   arg
Out.ar(out,
SinOsc.ar(
   Amplitude
AudioIn.ar(1),
0.01,
0.01,
1200,
400)
, 0, 0.3)
)
}).play;

)
```

ID: 470

Compander compressor, expander, limiter, gate, ducker

Compander.ar(input, control, threshold, slopeBelow, slopeAbove, clampTime, relaxTime, mul, add)

General purpose dynamics processor.

```
(
// example signal to process
play({
var z;
z = Decay2.ar(
Impulse.ar(8, 0,LFSaw.kr(0.3, 0, -0.3, 0.3)),
0.001, 0.3, Mix.ar(Pulse.ar([80,81], 0.3)))
})
)

(
// noise gate
play({
var z;
z = Decay2.ar(
Impulse.ar(8, 0,LFSaw.kr(0.3, 0, -0.3, 0.3)),
0.001, 0.3, Mix.ar(Pulse.ar([80,81], 0.3)));
Compander.ar(z, z, MouseX.kr(0.1, 1), 10, 1, 0.01, 0.01);
})
)

(
// compressor
play({
var z;
z = Decay2.ar(
Impulse.ar(8, 0,LFSaw.kr(0.3, 0, -0.3, 0.3)),
0.001, 0.3, Mix.ar(Pulse.ar([80,81], 0.3)));
Compander.ar(z, z, MouseX.kr(0.1, 1), 1, 0.5, 0.01, 0.01);
```

```
})
)

(
// limiter
play({
var z;
z = Decay2.ar(
Impulse.ar(8, 0,LFSaw.kr(0.3, 0, -0.3, 0.3)),
0.001, 0.3, Mix.ar(Pulse.ar([80,81], 0.3)));
Compander.ar(z, z, MouseX.kr(0.1, 1), 1, 0.1, 0.01, 0.01);
})
)

(
// sustainer
play({
var z;
z = Decay2.ar(
Impulse.ar(8, 0,LFSaw.kr(0.3, 0, -0.3, 0.3)),
0.001, 0.3, Mix.ar(Pulse.ar([80,81], 0.3)));
Compander.ar(z, z, MouseX.kr(0.1, 1), 0.1, 1, 0.01, 0.01);
})
)
```

ID: 471

Pitch autocorrelation pitch follower

#freq, hasFreq = Pitch.kr(in, initFreq, minFreq, maxFreq, execFreq, maxBinsPerOctave, median, ampThreshold, peakThreshold, downSample)

This is a better pitch follower than **ZeroCrossing**, but more costly of CPU. For most purposes the default settings can be used and only **in** needs to be supplied. Pitch returns two values (via an Array of OutputProxys, see the OutputProxy help file), a **freq** which is the pitch estimate and **hasFreq**, which tells whether a pitch was found. Some vowels are still problematic, for instance a wide open mouth sound somewhere between a low pitched short 'a' sound as in 'sat', and long 'i' sound as in 'fire', contains enough overtone energy to confuse the algorithm.

Examples: (use headphones!)

```
s = Server.local;

(
SynthDef "pitchFollow1"
var in, amp, freq, hasFreq, out;
in = Mix.new(AudioIn.ar([1,2]));
amp = Amplitude.kr(in, 0.05, 0.05);
# freq, hasFreq = Pitch.kr(in, ampThreshold:  0.02, median:  7);
 //freq = Lag.kr(freq.cpsmidi.round(1).midicps, 0.05);
out = Mix.new(VarSaw.ar(freq * [0.5,1,2], 0, LFNoise1.kr(0.3,0.1,0.1), amp));
6.do({
out = AllpassN.ar(out, 0.040, [0.040.rand,0.040.rand], 2)
});
Out.ar(0,out)
}).play(s);
)

(
SynthDef "pitchFollow2"
var in, amp, freq, hasFreq, out;
in = Mix.new(AudioIn.ar([1,2]));
amp = Amplitude.kr(in, 0.05, 0.05);
```

```
# freq, hasFreq = Pitch.kr(in, ampThreshold:  0.02, median:  7);
out = CombC.ar(LPF.ar(in, 1000), 0.1, (2 * freq).reciprocal, -6).distort * 0.05;
6.do({
out = AllpassN.ar(out, 0.040, [0.040.rand,0.040.rand], 2)
});
Out.ar(0,out);
}).play(s);
)
```

How it works:
The pitch follower executes periodically at the rate specified by **execFreq** in cps. **execFreq** is clipped to be between **minFreq** and **maxFreq** . First it detects whether the input peak to peak amplitude is above the **ampThreshold**. If it is not then no pitch estimation is performed, **hasFreq** is set to zero and **freq** is held at its previous value. It performs an autocorrelation on the input and looks for the first peak after the peak around the lag of zero that is above **peakThreshold** times the amplitude of the peak at lag zero.

Using a **peakThreshold** of one half does a pretty good job of eliminating overtones, and finding the first peak above that threshold rather than the absolute maximum peak does a good job of eliminating estimates that are actually multiple periods of the wave.

The autocorrelation is done coarsely at first using a maximum of **maxBinsPerOctave** lags until the peak is located. Then a fine resolution search is performed until the peak is found. (Note that maxBinsPerOctave does NOT affect the final pitch resolution; a fine resolution search is always performed. Setting maxBinsPerOctave larger will cause the coarse search to take longer, and setting it smaller will cause the fine search to take longer.)

The three values around the peak are used to find a fractional lag value for the pitch. If the pitch frequency is higher than **maxFreq**, or if no peak is found above **minFreq**, then **hasFreq** is set to zero and **freq** is held at its previous value.

It is possible to put a median filter of length **median** on the output estimation so that outliers and jitter can be eliminated. This will however add latency to the pitch estimation for new pitches, because the median filter will have to become half filled with new values before the new one becomes the median value. If median is set to one then that is equivalent to no filter, which is the default.

When an in range peak is found, it is inserted into the median filter, a new pitch is read out of the median filter and output as **freq**, and **hasFreq** is set to one.

It is possible to down sample the input signal by an integer factor **downSample** in order to reduce CPU overhead. This will also reduce the pitch resolution.

Until Pitch finds a pitch for the first time, it will output **initFreq**.

None of these settings are time variable.

Default Argument values:
initFreq = 440.0
minFreq = 60.0
maxFreq = 4000.0
execFreq = 100.0
maxBinsPerOctave = 16
median = 1
ampThreshold = 0.01
peakThreshold = 0.5
downSample = 1

ID: 472

RunningSum

A running sum over a user specified number of samples, useful for running RMS power windowing.

Class Methods

*ar(in, numsamp=40)

in- Input signal

numsamp- How many samples to take the running sum over (initialisation time only, not modulatable)

Examples

```
//overloads of course- would need scaling
{RunningSum.ar(AudioIn.ar)}.play

//Running Average over x samples
(
{
var x =100;

RunningSum.ar(LFSaw.ar,x)*(x.reciprocal)
}.play
)

//RMS Power
(
{
var input, numsamp;

input= LFSaw.ar;
numsamp=30;

(RunningSum.ar(input.squared,numsamp)/numsamp).sqrt
```

```
}.play
)


//shortcut in class
{RunningSum.rms(AudioIn.ar)}.play


//play around
(
{
var input, numsamp, power;

input= AudioIn.ar;
numsamp=500;
power= MouseX.kr(0.1,4);

(RunningSum.ar(input**power,numsamp)/numsamp)**(power.reciprocal)
}.play
)
```

ID: 473

Slope slope of signal

Slope.ar(in, mul, add)

Measures the rate of change per second of a signal.
Formula implemented is:

out[i] = (in[i] - in[i-1]) * sampling_rate

in - input signal to measure.

```
(
{
var a, b, c, scale;
 a = LFNoise2           // quadratic noise
 b = Slope         // first derivative produces line segments
 c = Slope         // second derivative produces constant segments
 scale = 0.0002; // needed to scale back to +/- 1.0
[a, b * scale, c * scale.squared]
}.plot
)
```

For another example of Slope see**[hypot]**.

ID: 474

ZeroCrossing zero crossing frequency follower

ZeroCrossing.ar(in)

Outputs a frequency based upon the distance between interceptions of the X axis. The X intercepts are determined via linear interpolation so this gives better than just integer wavelength resolution. This is a very crude pitch follower, but can be useful in some situations.

in - input signal.

```
Server.internal.boot;
(
{
var a;
a = SinOsc.ar(SinOsc.kr(1, 0, 600,700), 0, 0.1);
[a, ZeroCrossing.ar(a) * 0.0005]
}.scope;
)
```

25.2 Chaos

ID: 475

CuspL cusp map chaotic generator

CuspL.ar(freq, a, b, xi, mul, add)

freq - iteration frequency in Hertz
a, b - equation variables
xi - initial value of x

A linear-interpolating sound generator based on the difference equation:

x _{n+1} = a - b*sqrt(| x _n|)

```
// vary frequency
{ CuspL.ar(MouseX.kr(20, SampleRate.ir), 1.0, 1.99) * 0.3 }.play(s);

// mouse-controlled params
{ CuspL.ar(SampleRate.ir/4, MouseX.kr(0.9,1.1,1), MouseY.kr(1.8,2,1)) * 0.3 }.play(s);

// as a frequency control
{ SinOsc.ar(CuspL.ar(40, MouseX.kr(0.9,1.1,1), MouseY.kr(1.8,2,1))*800+900)*0.4 }.play(s);
```

ID: 476

CuspN cusp map chaotic generator

CuspN.ar(freq, a, b, xi, mul, add)

freq - iteration frequency in Hertz
a, b - equation variables
xi - initial value of x

A non-interpolating sound generator based on the difference equation:

x _{n+1} = a - b*sqrt(| x _n|)

```
// vary frequency
{ CuspN.ar(MouseX.kr(20, SampleRate.ir), 1.0, 1.99) * 0.3 }.play(s);

// mouse-controlled params
{ CuspN.ar(SampleRate.ir/4, MouseX.kr(0.9,1.1,1), MouseY.kr(1.8,2,1)) * 0.3 }.play(s);

// as a frequency control
{ SinOsc.ar(CuspN.ar(40, MouseX.kr(0.9,1.1,1), MouseY.kr(1.8,2,1))*800+900)*0.4 }.play(s);
```

ID: 477

FBSineC feedback sine with chaotic phase indexing

FBSineC.ar(freq, im, fb, a, c, xi, yi, mul, add)

freq - iteration frequency in Hertz
im - index multiplier amount
fb - feedback amount
a - phase multiplier amount
c - phase increment amount
xi - initial value of x
yi - initial value of y

A cubic-interpolating sound generator based on the difference equations:

x _{n+1} = sin(im*y _n + fb*x _n)
y _{n+1} = (ay _n + c) % 2pi

This uses a linear congruential function to drive the phase indexing of a sine wave. For **im** = 1, **fb** = 0, and **a** = 1 a normal sinewave results.

```
// default initial params
{ FBSineC.ar(SampleRate.ir/4) * 0.2 }.play(s);

// increase feedback
{ FBSineC.ar(SampleRate.ir, 1, Line.kr(0.01, 4, 10), 1, 0.1) * 0.2 }.play(s);

// increase phase multiplier
{ FBSineC.ar(SampleRate.ir, 1, 0, XLine.kr(1, 2, 10), 0.1) * 0.2 }.play(s);

// modulate frequency and index multiplier
{ FBSineC.ar(LFNoise2.kr(1, 1e4, 1e4), LFNoise2.kr(1,16,17), 1, 1.005, 0.7) * 0.2 }.play(s);

// randomly modulate params
(
  FBSineC
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(1, 32, 33),
LFNoise2.kr(1, 0.5),
```

```
LFNoise2.kr(1, 0.05, 1.05),
LFNoise2.kr(1, 0.3, 0.3)
) * 0.2 }.play(s);
)
```

ID: 478

FBSineL feedback sine with chaotic phase indexing

FBSineL.ar(freq, im, fb, a, c, xi, yi, mul, add)

freq - iteration frequency in Hertz
im - index multiplier amount
fb - feedback amount
a - phase multiplier amount
c - phase increment amount
xi - initial value of x
yi - initial value of y

A linear-interpolating sound generator based on the difference equations:

x _{n+1} = sin(im*y _n + fb*x _n)
y _{n+1} = (ay _n + c) % 2pi

This uses a linear congruential function to drive the phase indexing of a sine wave. For **im** = 1, **fb** = 0, and **a** = 1 a normal sinewave results.

```
// default initial params
{ FBSineL.ar(SampleRate.ir/4) * 0.2 }.play(s);

// increase feedback
{ FBSineL.ar(SampleRate.ir, 1, Line.kr(0.01, 4, 10), 1, 0.1) * 0.2 }.play(s);

// increase phase multiplier
{ FBSineL.ar(SampleRate.ir, 1, 0, XLine.kr(1, 2, 10), 0.1) * 0.2 }.play(s);

// modulate frequency and index multiplier
{ FBSineL.ar(LFNoise2.kr(1, 1e4, 1e4), LFNoise2.kr(1,16,17), 1, 1.005, 0.7) * 0.2 }.play(s);

// randomly modulate params
(
  FBSineL
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(1, 32, 33),
LFNoise2.kr(1, 0.5),
```

```
LFNoise2.kr(1, 0.05, 1.05),
LFNoise2.kr(1, 0.3, 0.3)
) * 0.2 }.play(s);
)
```

ID: 479

FBSineN feedback sine with chaotic phase indexing

FBSineN.ar(freq, im, fb, a, c, xi, yi, mul, add)

freq - iteration frequency in Hertz
im - index multiplier amount
fb - feedback amount
a - phase multiplier amount
c - phase increment amount
xi - initial value of x
yi - initial value of y

A non-interpolating sound generator based on the difference equations:

x _{n+1} = sin(im*y _n + fb*x _n)
y _{n+1} = (ay _n + c) % 2pi

This uses a linear congruential function to drive the phase indexing of a sine wave. For **im** = 1, **fb** = 0, and **a** = 1 a normal sinewave results.

```
// default initial params
{ FBSineN.ar(SampleRate.ir/4) * 0.2 }.play(s);

// increase feedback
{ FBSineN.ar(SampleRate.ir, 1, Line.kr(0.01, 4, 10), 1, 0.1) * 0.2 }.play(s);

// increase phase multiplier
{ FBSineN.ar(SampleRate.ir, 1, 0, XLine.kr(1, 2, 10), 0.1) * 0.2 }.play(s);

// modulate frequency and index multiplier
{ FBSineN.ar(LFNoise2.kr(1, 1e4, 1e4), LFNoise2.kr(1,16,17), 1, 1.005, 0.7) * 0.2 }.play(s);

// randomly modulate params
(
  FBSineN
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(1, 32, 33),
LFNoise2.kr(1, 0.5),
```

```
LFNoise2.kr(1, 0.05, 1.05),
LFNoise2.kr(1, 0.3, 0.3)
) * 0.2 }.play(s);
)
```

ID: 480

GbmanL gingerbreadman map chaotic generator

GbmanL.ar(freq, xi, yi, mul, add)

freq - iteration frequency in Hertz
xi - initial value of x
yi - initial value of y

A linear-interpolating sound generator based on the difference equations:

x _{n+1} = 1 - y _n + | x _n|
y _{n+1} = x _n

The behavior of the system is dependent only on its initial conditions and cannot be changed once it's started.

Reference:
Devaney, R. L. "The Gingerbreadman." Algorithm 3, 15-16, Jan. 1992.

```
// default initial params
{ GbmanL.ar(MouseX.kr(20, SampleRate.ir)) * 0.1 }.play(s);

// different initial params
{ GbmanL.ar(MouseX.kr(20, SampleRate.ir), -0.7, -2.7) * 0.1 }.play(s);

// wait for it...
{ GbmanL.ar(MouseX.kr(20, SampleRate.ir), 1.2, 2.0002) * 0.1 }.play(s);

// as a frequency control
{ SinOsc.ar(GbmanL.ar(40)*400+500)*0.4 }.play(s);
```

ID: 481

GbmanN gingerbreadman map chaotic generator

GbmanN.ar(freq, xi, yi, mul, add)

freq - iteration frequency in Hertz
xi - initial value of x
yi - initial value of y

A non-interpolating sound generator based on the difference equations:

x _{n+1} = 1 - y _n + | x _n|
y _{n+1} = x _n

The behavior of the system is only dependent on its initial conditions.

Reference:
Devaney, R. L. "The Gingerbreadman." Algorithm 3, 15-16, Jan. 1992.

```
// default initial params
{ GbmanN.ar(MouseX.kr(20, SampleRate.ir)) * 0.1 }.play(s);

// change initial params
{ GbmanN.ar(MouseX.kr(20, SampleRate.ir), -0.7, -2.7) * 0.1 }.play(s);

// wait for it...
{ GbmanN.ar(MouseX.kr(20, SampleRate.ir), 1.2, 2.0002) * 0.1 }.play(s);

// as a frequency control
{ SinOsc.ar(GbmanN.ar(40)*400+500)*0.4 }.play(s);
```

ID: 482

HenonC hénon map chaotic generator

HenonC.ar(freq, a, b, x0, x1, mul, add)

freq - iteration frequency in Hertz
a, b - equation variables
x0 - initial value of x
x1 - second value of x

A cubic-interpolating sound generator based on the difference equation:

x _{n+2} = 1 - ax _{n+1}² + bx _n

This equation was discovered by French astronomer Michel Hénon while studying the orbits of stars
in globular clusters.

```
// default initial params
{ HenonC.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// mouse-control of params
{ HenonC.ar(SampleRate.ir/4, MouseX.kr(1,1.4), MouseY.kr(0,0.3)) * 0.2 }.play(s);

// randomly modulate params
(
{ HenonC.ar(
 SampleRate
LFNoise2.kr(1, 0.2, 1.2),
LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 }.play(s);
)

// as a frequency control
{ SinOsc.ar(HenonC.ar(40, MouseX.kr(1,1.4), MouseY.kr(0,0.3))*800+900)*0.4 }.play(s);
```

ID: 483

HenonL hénon map chaotic generator

HenonL.ar(freq, a, b, x0, x1, mul, add)

freq - iteration frequency in Hertz
a, b - equation variables
x0 - initial value of x
x1 - second value of x

A linear-interpolating sound generator based on the difference equation:

x _{n+2} = 1 - ax _{n+1}² + bx _n

This equation was discovered by French astronomer Michel Hénon while studying the orbits of stars
in globular clusters.

```
// default initial params
{ HenonL.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// mouse-control of params
{ HenonL.ar(SampleRate.ir/4, MouseX.kr(1,1.4), MouseY.kr(0,0.3)) * 0.2 }.play(s);

// randomly modulate params
(
{ HenonL.ar(
 SampleRate
LFNoise2.kr(1, 0.2, 1.2),
LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 }.play(s);
)

// as a frequency control
{ SinOsc.ar(HenonL.ar(40, MouseX.kr(1,1.4), MouseY.kr(0,0.3))*800+900)*0.4 }.play(s);
```

ID: 484

HenonN hénon map chaotic generator

HenonN.ar(freq, a, b, x0, x1, mul, add)

freq - iteration frequency in Hertz
a, b - equation variables
x0 - initial value of x
x1 - second value of x

A non-interpolating sound generator based on the difference equation:

x _{n+2} = 1 - ax _{n+1}² + bx _n

This equation was discovered by French astronomer Michel Hénon while studying the orbits of stars
in globular clusters.

```
// default initial params
{ HenonN.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// mouse-control of params
{ HenonN.ar(SampleRate.ir/4, MouseX.kr(1,1.4), MouseY.kr(0,0.3)) * 0.2 }.play(s);

// randomly modulate params
(
{ HenonN.ar(
 SampleRate
LFNoise2.kr(1, 0.2, 1.2),
LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 }.play(s);
)

// as a frequency control
{ SinOsc.ar(HenonN.ar(40, MouseX.kr(1,1.4), MouseY.kr(0,0.3))*800+900)*0.4 }.play(s);
```

ID: 485

LatoocarfianC latoocarfian chaotic generator

LatoocarfianC.ar(freq, a, b, c, d, xi, yi, mul, add)

freq - iteration frequency in Hertz
a, b, c, d - equation variables
xi - initial value of x
yi - initial value of y

A cubic-interpolating sound generator based on a function given in Clifford Pickover's book Chaos In Wonderland, pg 26.
The function is:

x _{n+1} = sin(by _n) + c*sin(bx _n)
y _{n+1} = sin(ay _n) + d*sin(ax _n)

According to Pickover, parameters **a** and **b** should be in the range from -3 to +3, and parameters **c** and **d** should be in the range from 0.5 to 1.5.
The function can, depending on the parameters given, give continuous chaotic output, converge to a single value (silence) or oscillate in a cycle (tone).
This UGen is experimental and not optimized currently, so is rather hoggish of CPU.

```
// default initial params
{ LatoocarfianC.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate all params
(
  LatoocarfianC
 SampleRate
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,0.5,1.5),
LFNoise2.kr(2,0.5,1.5)
) * 0.2 }.play(s);
)
```

ID: 486

LatoocarfianL latoocarfian chaotic generator

LatoocarfianL.ar(freq, a, b, c, d, xi, yi, mul, add)

freq - iteration frequency in Hertz
a, b, c, d - equation variables
xi - initial value of x
yi - initial value of y

A linear-interpolating sound generator based on a function given in Clifford Pickover's book Chaos In Wonderland, pg 26.
The function is:

x _{n+1} = sin(by _n) + c*sin(bx _n)
y _{n+1} = sin(ay _n) + d*sin(ax _n)

According to Pickover, parameters **a** and **b** should be in the range from -3 to +3, and parameters **c** and **d** should be in the range from 0.5 to 1.5.
The function can, depending on the parameters given, give continuous chaotic output, converge to a single value (silence) or oscillate in a cycle (tone).
This UGen is experimental and not optimized currently, so is rather hoggish of CPU.

```
// default initial params
{ LatoocarfianL.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate all params
(
  LatoocarfianL
 SampleRate
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,0.5,1.5),
LFNoise2.kr(2,0.5,1.5)
) * 0.2 }.play(s);
)
```

ID: 487

LatoocarfianN latoocarfian chaotic generator

LatoocarfianN.ar(freq, a, b, c, d, xi, yi, mul, add)

freq - iteration frequency in Hertz
a, b, c, d - equation variables
xi - initial value of x
yi - initial value of y

A non-interpolating sound generator based on a function given in Clifford Pickover's book Chaos In Wonderland, pg 26.
The function is:

x _{n+1} = sin(by _n) + c*sin(bx _n)
y _{n+1} = sin(ay _n) + d*sin(ax _n)

According to Pickover, parameters **a** and **b** should be in the range from -3 to +3, and parameters **c** and **d** should be in the range from 0.5 to 1.5.
The function can, depending on the parameters given, give continuous chaotic output, converge to a single value (silence) or oscillate in a cycle (tone).
This UGen is experimental and not optimized currently, so is rather hoggish of CPU.

```
// default initial params
{ LatoocarfianN.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate all params
(
  LatoocarfianN
 SampleRate
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,1.5,1.5),
LFNoise2.kr(2,0.5,1.5),
LFNoise2.kr(2,0.5,1.5)
) * 0.2 }.play(s);
)
```

ID: 488

LinCongC linear congruential chaotic generator

LinCongC.ar(freq, a, c, m, xi, mul, add)

freq - iteration frequency in Hertz
a - multiplier amount
c - increment amount
m - modulus amount
xi - initial value of x

A cubic-interpolating sound generator based on the difference equation:

x _{n+1} = (ax _n + c) % m

The output signal is automatically scaled to a range of [-1, 1].

```
// default initial params
{ LinCongC.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate params
(
  LinCongC
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(0.1, 0.5, 1.4),
LFNoise2.kr(0.1, 0.1, 0.1),
LFNoise2.kr(0.1)
) * 0.2 }.play(s);
)

// as frequency control...
(
{
SinOsc
 LinCongC
40,
LFNoise2.kr(0.1, 0.1, 1),
LFNoise2.kr(0.1, 0.1, 0.1),
```

```
LFNoise2.kr(0.1),
0, 500, 600
)
) * 0.4 }.play(s);
)
```

ID: 489

LinCongL linear congruential chaotic generator

LinCongL.ar(freq, a, c, m, xi, mul, add)

freq - iteration frequency in Hertz
a - multiplier amount
c - increment amount
m - modulus amount
xi - initial value of x

A linear-interpolating sound generator based on the difference equation:

x _{n+1} = (ax _n + c) % m

The output signal is automatically scaled to a range of [-1, 1].

```
// default initial params
{ LinCongL.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate params
(
  LinCongL
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(0.1, 0.5, 1.4),
LFNoise2.kr(0.1, 0.1, 0.1),
LFNoise2.kr(0.1)
) * 0.2 }.play(s);
)

// as frequency control...
(
{
SinOsc
 LinCongL
40,
LFNoise2.kr(0.1, 0.1, 1),
LFNoise2.kr(0.1, 0.1, 0.1),
```

```
LFNoise2.kr(0.1),
0, 500, 600
)
) * 0.4 }.play(s);
)
```

ID: 490

LinCongN linear congruential chaotic generator

LinCongN.ar(freq, a, c, m, xi, mul, add)

freq - iteration frequency in Hertz
a - multiplier amount
c - increment amount
m - modulus amount
xi - initial value of x

A non-interpolating sound generator based on the difference equation:

x _{n+1} = (ax _n + c) % m

The output signal is automatically scaled to a range of [-1, 1].

```
// default initial params
{ LinCongN.ar(MouseX.kr(20, SampleRate.ir)) * 0.2 }.play(s);

// randomly modulate params
(
  LinCongN
LFNoise2.kr(1, 1e4, 1e4),
LFNoise2.kr(0.1, 0.5, 1.4),
LFNoise2.kr(0.1, 0.1, 0.1),
LFNoise2.kr(0.1)
) * 0.2 }.play(s);
)

// as frequency control...
(
{
SinOsc
 LinCongN
40,
LFNoise2.kr(0.1, 0.1, 1),
LFNoise2.kr(0.1, 0.1, 0.1),
```

```
LFNoise2.kr(0.1),
0, 500, 600
)
) * 0.4 }.play(s);
)
```

ID: 491

LorenzL lorenz chaotic generator

LorenzL.ar(freq, s, r, b, h, xi, yi, zi, mul, add)

freq - iteration frequency in Hertz
s, r, b - equation variables
h - integration time step
xi - initial value of x
yi - initial value of y
zi - initial value of z

A strange attractor discovered by Edward N. Lorenz while studying mathematical models of the atmosphere.
The system is composed of three ordinary differential equations:

x' = s(y - x)
y' = x(r - z) - y
z' = xy - bz

The time step amount **h** determines the rate at which the ODE is evaluated. Higher values will increase the
rate, but cause more instability. A safe choice is the default amount of 0.05.

```
// vary frequency
{ LorenzL.ar(MouseX.kr(20, SampleRate.ir)) * 0.3 }.play(s);

// randomly modulate params
(
  LorenzL
 SampleRate
LFNoise0.kr(1, 2, 10),
LFNoise0.kr(1, 20, 38),
LFNoise0.kr(1, 1.5, 2)
) * 0.2 }.play(s);
)

// as a frequency control
```

```
{ SinOsc.ar(Lag.ar(LorenzL.ar(MouseX.kr(1, 200)),3e-3)*800+900)*0.4 }.play(s);
```

ID: 492

QuadC general quadratic map chaotic generator

QuadC.ar(freq, a, b, c, xi, mul, add)

freq - iteration frequency in Hertz
a, b, c - equation variables
xi - initial value of x

A cubic-interpolating sound generator based on the difference equation:

x _{n+1} = ax _n² + bx _n + c

```
// default params
{ QuadC.ar(SampleRate.ir/4) * 0.2 }.play(s);

// logistic map
// equation:  x1 = -r*x0^2 + r*x0
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
QuadC.ar(SampleRate.ir/4, r.neg, r, 0, 0.1) * 0.4;
}.play(s);
)

// logistic map as frequency control
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
SinOsc.ar(QuadC.ar(40, r.neg, r, 0, 0.1, 800, 900)) * 0.4;
}.play(s);
)
```

ID: 493

QuadL general quadratic map chaotic generator

QuadL.ar(freq, a, b, c, xi, mul, add)

freq - iteration frequency in Hertz
a, b, c - equation variables
xi - initial value of x

A linear-interpolating sound generator based on the difference equation:

x _{n+1} = ax _n² + bx _n + c

```
// default params
{ QuadL.ar(SampleRate.ir/4) * 0.2 }.play(s);

// logistic map
// equation:  x1 = -r*x0^2 + r*x0
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
QuadL.ar(SampleRate.ir/2, r.neg, r, 0, 0.1) * 0.4;
}.play(s);
)

// logistic map as frequency control
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
SinOsc.ar(QuadL.ar(40, r.neg, r, 0, 0.1, 800, 900)) * 0.4;
}.play(s);
)
```

ID: 494

QuadN general quadratic map chaotic generator

QuadN.ar(freq, a, b, c, xi, mul, add)

freq - iteration frequency in Hertz
a, b, c - equation variables
xi - initial value of x

A non-interpolating sound generator based on the difference equation:

x _{n+1} = ax _n² + bx _n + c

```
// default params
{ QuadN.ar(SampleRate.ir/4) * 0.2 }.play(s);

// logistic map
// equation:  x1 = -r*x0^2 + r*x0
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
QuadN.ar(SampleRate.ir/2, r.neg, r, 0, 0.1) * 0.4;
}.play(s);
)

// logistic map as frequency control
(
{ var r;
r = MouseX.kr(3.5441, 4); // stable range
SinOsc.ar(QuadN.ar(40, r.neg, r, 0, 0.1, 800, 900)) * 0.4;
}.play(s);
)
```

ID: 495

StandardL standard map chaotic generator

StandardL.ar(freq, k, xi, yi, mul, add)

freq - iteration frequency in Hertz
k - perturbation amount
xi - initial value of x
yi - initial value of y

A linear-interpolating sound generator based onthe difference equations:

x _{n+1} = (x _n + y _{n+1}) % 2pi
y _{n+1} = (y _n + ksin(x _n)) % 2pi

The standard map is an area preserving map of a cylinder discovered by the plasma physicist Boris Chirikov.

.

```
// vary frequency
{ StandardL.ar(MouseX.kr(20, SampleRate.ir)) * 0.3 }.play(s);


// mouse-controlled param
{ StandardL.ar(SampleRate.ir/2, MouseX.kr(0.9,4)) * 0.3 }.play(s);


// as a frequency control
{ SinOsc.ar(StandardL.ar(40, MouseX.kr(0.9,4))*800+900)*0.4 }.play(s);
```

ID: 496

StandardN standard map chaotic generator

StandardN.ar(freq, k, xi, yi, mul, add)

freq - iteration frequency in Hertz
k - perturbation amount
xi - initial value of x
yi - initial value of y

A non-interpolating sound generator based onthe difference equations:

x _{n+1} = (x _n + y _{n+1}) % 2pi
y _{n+1} = (y _n + ksin(x _n)) % 2pi

The standard map is an area preserving map of a cylinder discovered by the plasma physicist Boris Chirikov.

.

```
// vary frequency
{ StandardN.ar(MouseX.kr(20, SampleRate.ir)) * 0.3 }.play(s);

// mouse-controlled param
{ StandardN.ar(SampleRate.ir/2, MouseX.kr(0.9,4)) * 0.3 }.play(s);

// as a frequency control
{ SinOsc.ar(StandardN.ar(40, MouseX.kr(0.9,4))*800+900)*0.4 }.play(s);
```

25.3 Control

ID: 497

Dbrown, Dibrown demand rate brownian movement generators

superclass: UGen

*new(lo, hi, step, length)

lo minimum value
hi maximum value
step maximum step for each new value
length number of values to create

Dbrown returns numbers in the continuous range between lo and hi, Dibrown returns integer values
The arguments can be a number or any other ugen

structurally related: **Pbrown, BrownNoise**
see also: **Demand**

```
   // example

 // Dbrown
(
{
var a, freq, trig;
a = Dbrown(0, 15, 1, inf);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)



 // Dibrown
```

```
(
{
var a, freq, trig;
a = Dibrown(0, 15, 1, inf);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 498

Dbufrd buffer demand ugen

superclass: UGen

*new(bufnum, phase, loop)

bufnum buffer number to read from
phase index into the buffer (demand ugen or any other ugen)
loop when phase exceeds number of frames in buffer, loops when set to 1 (default :1)

```
// example

b = Buffer.alloc(s, 24, 1);
b.setn(0, { exprand(200, 500) } !  b.numFrames);
b.getn(0, b.numFrames, {| x|  x.postln })

(
{ var indexPattern;
indexPattern = Dseq([Dseq([0, 3, 5, 0, 3, 7, 0, 5, 9], 3), Dbrown(0, 23, 1, 5)], inf);
 SinOsc
Demand.kr(Dust.kr(10), 0, Dbufrd(b.bufnum, indexPattern))
) * 0.1
}.play;
)

// buffer as a time pattern

c = Buffer.alloc(s, 24, 1);
c.setn(0, { [1, 0.5, 0.25].choose } !  c.numFrames);
c.getn(0, c.numFrames, {| x|  x.postln })

(
{ var indexPattern;
indexPattern = Dseq([Dseq([0, 3, 5, 0, 3, 7, 0, 5, 9], 3), Dbrown(0, 23, 1, 5)], inf);
 SinOsc
```

```
Duty.kr(
Dbufrd(c.bufnum, Dseries(0, 1, inf)) * 0.5,
0,
Dbufrd(b.bufnum, indexPattern)
)
) * 0.1
}.play;
)

// free buffers

b.free; c.free;
```

ID: 499

Demand demand results from demand rate ugens

superclass: MultiOutUGen

Demand.ar(trig, reset, [..ugens..])

When there is a trigger at the trig input, a value is demanded each ugen in the list and output. The unit generators in the list should be 'demand' rate.
When there is a trigger at the reset input, the demand rate ugens in the list are reset.

trig - trigger. Can be any signal. A trigger happens when the signal changes from non-positive to positive.
reset - trigger. Resets the list of ugens when triggered.

```
// examples

(
{
var trig, seq, freq;
trig = Impulse.kr(24);
seq = Drand([Dseq((1..5).mirror1, 1), Drand((4..10), 8)], 2000);
freq = Demand.kr(trig, 0, seq * 100);
SinOsc.ar(freq + [0,0.7]).cubed.cubed.scaleneg(MouseX.kr(-1,1)) * 0.1;
}.play;
)

(
{
var trig, seq, freq;
trig = Impulse.kr(12);
seq = Drand([Dseq((1..5).mirror1, 1), Drand((4..10), 8)], 2000) * Drand([1,2,4,8],2000);
freq = Demand.kr(trig, 0, seq * 100);
SinOsc.ar(freq + [0,0.7]).cubed.cubed.scaleneg(MouseX.kr(-1,1)) * 0.1;
}.play;
```

```
)
```

```
(
{
var freq, trig, reset, seq;
trig = Impulse.kr(10);
seq = Diwhite(60, 72, inf).midicps;
freq = Demand.kr(trig, 0, seq);
SinOsc.ar(freq + [0,0.7]).cubed.cubed * 0.1;
}.play;
)

(
{
var freq, trig, reset, seq;
trig = Impulse.kr(10);
seq = Dseq([72, 75, 79, Drand([82,84,86])], inf).midicps;
freq = Demand.kr(trig, 0, seq);
SinOsc.ar(freq + [0,0.7]).cubed.cubed * 0.1;
}.play;
)


(
{
var freq, trig, reset, seq;
trig = Impulse.kr(10);
seq = Dswitch1(
[
Diwhite(60, 72, inf),
Dseq([72, 75, 79, Drand([82,84,86])], inf)
],
LFPulse.kr(0.2)
);
freq = Demand.kr(trig, 0, seq.midicps);
SinOsc.ar(freq + [0,0.7]).cubed.cubed * 0.1;
}.play;
```

```
)


(
{
var freq, trig, reset, seq1, seq2;
trig = Impulse.kr(10);
seq1 = Drand([72, 75, 79, 82] - 12, inf).midicps;
seq2 = Dseq([72, 75, 79, Drand([82,84,86])], inf).midicps;
freq = Demand.kr(trig, 0, [seq1, seq2]);
SinOsc.ar(freq + [0,0.7]).cubed.cubed * 0.1;
}.play;
)

(
{
var trig, seq;
trig = Impulse.kr(8);
seq = Drand([
Dseq([4,0,0,1,2,1,0,1]),
Dseq([4,0,2,0,1,0,1,1]),
Dseq([4,0,0,2,0,0,1,1]),
Dseq([4,0,1,2,0,1,2,0]),
Dseq([4,1,1,1,2,2,3,3]),
Dseq([4,1,0,1,0,1,0,1])
], inf);
trig = Demand.kr(trig, 0, seq * 0.4) * trig;
{LPF.ar(PinkNoise.ar, 5000)}.dup * Decay.kr(trig, 0.5);
}.play;
)
```

ID: 500

```
// experimental, might change!  //

.ar(levels, times, shapes, curves, gate, reset, levelScale, levelOffset, timeScale, doneAction)
.kr(levels, times, shapes, curves, gate, reset, levelScale, levelOffset, timeScale, doneAction)

levels:  a demand ugen or any other ugen
times:  a demand ugen or any other ugen
if one of these ends, the doneAction is evaluated

shapes:   a demand ugen or any other ugen:  the number given is the shape number according to Env
curves:  a demand ugen or any other ugen:  if shape is 5, this is the curve factor


some curves/shapes don't work if the duration is too short.  have to see how to improve this.
also some depend on the levels obviously, like exponential cannot cross zero.

gate:  if gate is x >= 1, the ugen runs
if gate is 0 > x > 1, the ugen is released at the next level (doneAction)
if gate is x < 0, the ugen is sampled end held

reset:  if reset crosses from nonpositive to positive, the ugen is reset at the next level
if it is > 1, it is reset immediately.


these parameters may change.




s.reboot;


// frequency envelope with random times
(
{
var freq;
```

```
freq = DemandEnvGen.ar(
Dseq([204, 400, 201, 502, 300, 200], inf),
Drand([1.01, 0.2, 0.1, 2], inf) * MouseY.kr(0.01, 3, 1),
    7 // cubic interpolation
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)

// frequency modulation
(
{
var freq, list;
list = { exprand(200, 1000.0) } !  32;
freq = DemandEnvGen.ar(
{ Dseq(list.scramble, inf) } !  2,
SampleDur.ir * MouseY.kr(1, 3000, 1),
    5, // curve interpoaltion
    MouseX               // curve must be negative for fast interpol.
);
SinOsc.ar(freq) * 0.1

}.play;
)


//  gate
// mouse x on right side of screen toggles gate
(
{
var freq;
freq = DemandEnvGen.kr(
Dwhite(300, 1000, inf).round(100),
0.1,
    5, 0.3, // curve:  0.3
MouseX.kr > 0.5,
1
);
SinOsc.ar(freq * [1, 1.21]) * 0.1
```

```
}.play;
)

// sample and hold (0.5 > gate > 0)
// mouse x on right side of screen toggles gate
// mouse y scales frequency
(
{
var freq;
freq = DemandEnvGen.kr(
Dwhite(300, 1000, inf).round(100),
0.1,
5, 0.3,
MouseX.kr > 0.5 + 0.1
);
SinOsc.ar(freq * [1, 1.21]) * 0.1

}.play;
)

// gate
// mouse x on right side of screen toggles gate
// mouse button does soft reset
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([Dseries(400, 200, 5), 500, 800, 530, 4000, 900], 2),
Dseq([0.2, 0.1, 0.2, 0.3, 0.1], inf),
Dseq([1, 0, 0, 6, 1, 1, 0, 2], inf), // shapes
0,
MouseX.kr > 0.5, // gate
MouseButton.kr > 0.5, // reset
doneAction:0
);
SinOsc.ar(freq * [1, 1.001]) * 0.1

}.play;
)
```

```
//  gate
// mouse x on right side of screen toggles sample and hold
// mouse button does hard reset
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([Dseries(400, 200, 5), 500, 800, 530, 4000, 900], 2),
0.1,
3, 0,
    MouseX                 // gate:  sample and hold
MouseButton.kr > 0.5 * 2, // hard reset
doneAction:  0
);
SinOsc.ar(freq * [1, 1.001]) * 0.1

}.play;
)




// short sequence with doneAction, linear
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([1300, 500, 800, 300, 400], 1),
0.2,
1,
doneAction:2
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)
```

```
// short sequence with doneAction, step
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([1300, 500, 800, 300, 400], 1),
0.2,
0,
doneAction:2
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)



// a linear ramp
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([300, 800], 1),
1,
1
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)



// random gate:   release.  gate low level > 0.
// only end points are kept as release levels
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([500, 800], inf),
```

```
0.03,
1,0, // linear
ToggleFF.kr(Dust.kr(5)) + 0.1 // gate


);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)

// random gate:  sample and hold.  gate low level = 0.
(
{
var freq;
freq = DemandEnvGen.kr(
Dseq([500, 800, 600], inf),
0.03,
1,0, // linear
ToggleFF.kr(Dust.kr(5)), // gate
    0 // reset

);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)




// lfnoise1
(
{
   DemandEnvGen
Dwhite(-0.1, 0.1, inf),
SampleDur.ir * MouseY.kr(0.5, 20),
5,
-4
);
```

```
}.play;
)

// lfbrownnoise
(
{
   DemandEnvGen
Dbrown(-0.1, 0.1, 0.1, inf),
SampleDur.ir * MouseY.kr(1, 100, 1)
);


}.play;
)




Server.internal.boot;

// hardsyncing a saw
(
{

   DemandEnvGen
Dseq([Dseries(-0.1, 0.01, 20)], inf),
SampleDur.ir * MouseY.kr(1, 100, 1),
1, 0,
K2A.ar(1),
Impulse.ar(MouseX.kr(1, SampleRate.ir * MouseX.kr(0.002, 1, 1), 1), 0, 1.5)

)


}.scope;
)
```

```
// softsyncing a saw
(
{

   DemandEnvGen
Dseq([Dseries(-0.1, 0.01, 20)], inf),
SampleDur.ir * MouseY.kr(1, 100, 1),
1, 0,
K2A.ar(1),
Impulse.ar(MouseX.kr(1, SampleRate.ir * MouseX.kr(0.002, 1, 1), 1)) + [0, 0.3]

)


}.scope;
)



// hardsyncing a saw, som random elements
(
{

   DemandEnvGen
Dseq([Dseries(-0.1, 0.01, 20), Dseries(-0.1, 0.01, 20), Dwhite(-0.1, 0.1, 5)], inf),
SampleDur.ir * MouseY.kr(1, 100, 1),
3, 0,
1,
Impulse.ar(MouseX.kr(1, SampleRate.ir * MouseX.kr(0.002, 1, 1), 1), 0, 1.5)

)


}.scope;
)

// softsyncing a saw, som random elements
(
```

```
{

   DemandEnvGen
Dseq([Dseries(-0.1, 0.01, 20), Dseries(-0.1, 0.01, 20), Dwhite(-0.1, 0.1, 5)], inf),
SampleDur.ir * MouseY.kr(1, 100, 1),
    1, 0, // linear interpolation
1,
Impulse.ar(MouseX.kr(1, SampleRate.ir * MouseX.kr(0.002, 1, 1), 1))

)



}.scope;
)



// multichannel expansion
// mouse x on right side of screen toggles gate
// mouse y controls speed

(
{
var freq;
freq = DemandEnvGen.kr(
{ Dseq([300, 800, Drand([1000, 460, 300], 1), 400], inf) + 3.0.rand } !  2,
MouseY.kr(0.001, 2, 1),
5, -4,
MouseX.kr > 0.5
);
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 501

Dgeom demand rate geometric series ugen

superclass: UGen

*new(start, grow, length)

start start value
grow value by which to grow (x = x[-1] * grow)
length number of values to create

structurally related: **Pgeom**
The arguments can be a number or any other ugen

```
// example

(
{
var a, freq, trig;
a = Dgeom(1, 1.2, 15);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)

(
{
var a, freq, trig;
a = Dgeom(1, 1.2, inf);
trig = Dust.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 502

Drand, Dxrand demand rate random sequence generators

superclass: ListDUGen

*new(array, length)

array array of values or other ugens
length number of values to return

structurally related: **Prand**
see also: **Demand**

Dxrand never plays the same value twice, whereas Drand chooses any value in the list

```
// example

(
{
var a, freq, trig;
a = Drand([1, 3, 2, 7, 8], inf);
trig = Impulse.kr(MouseX.kr(1, 400, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)

(
{
var a, freq, trig;
a = Dxrand([1, 3, 2, 7, 8], inf);
trig = Impulse.kr(MouseX.kr(1, 400, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 503

Dseq demand rate sequence generator

superclass: ListDUGen

*new(array, length)

array array of values or other ugens
length number of repeats

structurally related: **Pseq**
see also: **Demand**

```
   // example

(
{
var a, freq, trig;
a = Dseq([1, 3, 2, 7, 8], 3);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)

 // audio rate
(
{
var a, freq, trig;
a = Dseq({ 10.rand } !  32, inf);
trig = Impulse.ar(MouseX.kr(1, 10000, 1));
freq = Demand.ar(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 504

Dser demand rate sequence generator

superclass: ListDUGen

*new(array, length)

array array of values or other ugens
length number of values to return

structurally related: **Pser**
see also: **Demand**

```
   // example

(
{
var a, freq, trig;
a = Dser([1, 3, 2, 7, 8], 8);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 505

Dseries demand rate arithmetic series ugen

superclass: UGen

*new(start, step, length)

start start value
step step value
length number of values to create
The arguments can be a number or any other ugen

structurally related: **Pseries**
see also: **Demand**

```
   // example

(
{
var a, freq, trig;
a = Dseries(0, 1, 15);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)

(
{
var a, freq, trig;
a = Dseries(0, 1, inf);
trig = Dust.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) % 15 * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
```

)

ID: 506

Dswitch1 demand rate generator for switching between inputs

superclass: UGen

*new(array, index)

array array of values or other ugens
index which of the inputs to return

structurally related: **Pswitch1**
see also: **Demand**

```
   // example

(
{
var a, freq, trig;
a = Dswitch1([1, 3, MouseY.kr(1, 15), 2, Dwhite(0, 3, 2)], MouseX.kr(0, 4));
trig = Impulse.kr(3);
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)

(
{
var a, freq, trig;
a = Dswitch1({ | i|  Dseq((0..i*3), inf) } !  5, MouseX.kr(0, 4));
trig = Impulse.kr(6);
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 507

Duty demand results from demand rate ugens

superclass: UGen

Duty.ar(duration, reset, level, doneAction)

A value is demanded each ugen in the list and output according to a stream of duration values.
The unit generators in the list should be 'demand' rate.
When there is a trigger at the reset input, the demand rate ugens in the list and the duration are reset.
The reset input may also be a demand ugen, providing a stream of reset times.

duration time values. Can be a demand ugen or any signal.
The next level is acquired after duration.

reset trigger or reset time values. Resets the list of ugens and the duration ugen when triggered.
The reset input may also be a demand ugen, providing a stream of reset times.

level demand ugen providing the output values.

doneAction a doneAction that is evaluated when the duration stream ends.
See **[UGen-doneActions]** for more detail.

```
// examples

s.boot;
```

```
(
{
var freq;
freq = Duty.kr(
    Drand                   inf   // demand ugen as durations
0,
Dseq([204, 400, 201, 502, 300, 200], inf)
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)

(
{
var freq;
freq = Duty.kr(
    MouseX                  // control rate ugen as durations
0,
Dseq([204, 400, 201, 502, 300, 200], inf)
);
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)



// resetting the demand ugens

(
{
var freq;
freq = Duty.kr(
Dseq([0.2, 0.3, 0.4, Dseq([1, 1, 1, 2, 1, 2], inf)]) / 2,
    Dust        // control rate reset
Dseq([0, 1, 2, Dseq([1, 2, 3, 4, 5], inf)])
) * 30 + 250;
SinOsc.ar(freq * [1, 1.01]) * 0.1
```

```
}.play;
)

(
{
var freq;
freq = Duty.kr(
Dseq([0.2, 0.3, 0.4, Dseq([1, 1, 1, 2, 1, 2], inf)]) / 2,
Dseq([1, 2, 4, 5], inf), // demand rate reset
Dseq([0, 1, 2, Dseq([1, 2, 3, 4, 5], inf)])
) * 30 + 250;
SinOsc.ar(freq * [1, 1.01]) * 0.1

}.play;
)


// demand ugen as audio oscillator

(
{
var a, n=5, m=64;
a = {
var x;
x = { 0.2.rand2 } !  m;
x = x ++ ({  Drand({ 0.2.rand2 } !  n) } !  m.rand);
Dseq(x.scramble, inf)
} !  n;
Duty.ar(
MouseX.kr(1, 125, 1) * SampleDur.ir * [1, 1.02],
0,
Dswitch1(a, MouseY.kr(0, n-1))
)

}.play;
)
```

ID: 508

Dwhite, Diwhite demand rate white noise random generators

superclass: UGen

*new(lo, hi, length)

lo minimum value
hi maximum value
length number of values to create

Dwhite returns numbers in the continuous range between lo and hi, Diwhite returns integer values
The arguments can be a number or any other ugen

structurally related: **Pwhite, WhiteNoise**
see also: **Demand**

```
   // example

 // Dwhite
(
{
var a, freq, trig;
a = Dwhite(0, 15, inf);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)



 // Diwhite

(
```

```
{
var a, freq, trig;
a = Diwhite(0, 15, inf);
trig = Impulse.kr(MouseX.kr(1, 40, 1));
freq = Demand.kr(trig, 0, a) * 30 + 340;
SinOsc.ar(freq) * 0.1

}.play;
)
```

ID: 509

Integrator leaky integrator

Integrator.ar(in, coef, mul, add)

Integrates an input signal with a leak. The formula implemented is:

out(0) = in(0) + (coef * out(-1))

in - input signal
coef - leak coefficient.

```
{ Integrator.ar(LFPulse.ar(300, 0.2, 0.1), MouseX.kr(0.001, 0.999, 1)) }.play

// used as an envelope
{ Integrator.ar(LFPulse.ar(3, 0.2, 0.0004), 0.999, FSinOsc.ar(700), 0) }.play


// scope, using the internal server:

{ Integrator.ar(LFPulse.ar(1500 / 4, 0.2, 0.1), MouseX.kr(0.01, 0.999, 1)) }.scope
```

ID: 510

Latch sample and hold

Latch.ar(in, trig)

Holds input signal value when triggered.
in - input signal.
trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from
non-positive to positive.

```
{ Blip.ar(Latch.ar(WhiteNoise.ar, Impulse.ar(9)) * 400 + 500, 4, 0.2) }.play;
```

The above is just meant as example. LFNoise0 is a faster way to generate random steps :

```
{ Blip.ar(LFNoise0.kr(9, 400 ,500), 4, 0.2) }.play;
```

ID: 511

TDuty demand results as trigger from demand rate ugens

superclass: Duty

TDuty.ar(duration, reset, level, doneAction)

A value is demanded each ugen in the list and output as a trigger according to a stream of duration values.
The unit generators in the list should be 'demand' rate.
When there is a trigger at the reset input, the demand rate ugens in the list and the duration are reset.
The reset input may also be a demand ugen, providing a stream of reset times.

duration time values. Can be a demand ugen or any signal.
The next trigger value is acquired after the duration provided by the last time value.

reset trigger or reset time values. Resets the list of ugens and the duration ugen when triggered.
The reset input may also be a demand ugen, providing a stream of reset times.

level demand ugen providing the output values.

doneAction a doneAction that is evaluated when the duration stream ends.
For the various doneActions, see: [Synth-Controlling-UGens]

```
// examples

s.boot;
```

```
// play a little rhythm

{ TDuty.ar(Dseq([0.1, 0.2, 0.4, 0.3], inf)) }.play; // demand ugen as durations



// amplitude changes
(
{
var trig;
trig = TDuty.ar(
Dseq([0.1, 0.2, 0.4, 0.3], inf), // demand ugen as durations
0,
Dseq([0.1, 0.4, 0.01, 0.5, 1.0], inf) // demand ugen as amplitude
);
Ringz.ar(trig, 1000, 0.1)

}.play;
)

(
{
var trig;
trig = TDuty.ar(
    MouseX                  // control rate ugen as durations
0,
Dseq([0.1, 0.4, 0.01, 0.5, 1.0], inf)
);
Ringz.ar(trig, 1000, 0.1)

}.play;
)



// demand ugen as audio oscillator

(
```

```
{
var a, trig, n=5, m=64;
a = {
var x;
x = { 0.2.rand2 } !  m;
x = x ++ ({  Drand({ 0.2.rand2 } !  n) } !  m.rand);
Dseq(x.scramble, inf)
} !  n;
trig = TDuty.ar(
MouseX.kr(1, 2048, 1) * SampleDur.ir * [1, 1.02],
0,
Dswitch1(a, MouseY.kr(0, n-1))
);
Ringz.ar(trig, 1000, 0.01)

}.play;
)


// single impulses

(
SynthDef("delta_demand", { arg amp=0.5, out;
OffsetOut.ar(out,
TDuty.ar(Dseq([0]), 0, amp, 2)
)
}).send(s);
)

fork { 10.do { s.sendBundle(0.2, ["/s_new", "delta_demand", -1]); 1.0.rand.wait } };


// chain of impulses
(
SynthDef "delta_demand2"
 OffsetOut
TDuty.ar(Dgeom(0.05, 0.9, 20), 0, 0.5, 2)
)
}).send(s);
)
```

```
fork { 10.do { s.sendBundle(0.2, ["/s_new", "delta_demand2", -1]); 1.0.rand.wait } };



// multichannel expansion

(
{
var t;
t = TDuty.ar(
Drand([Dgeom(0.1, 0.8, 20), 1, 2], inf) !  2,
0,
[Drand({ 1.0.rand } !  8, inf), Dseq({ 1.0.rand } !  8, inf)] * 2
);
x = Ringz.ar(t, [400, 700], 0.1) * 0.1;

}.play;
)
```

25.4 Controls

ID: 512

Decay exponential decay

Decay.ar(in, decayTime, mul, add)

This is essentially the same as **Integrator** except that instead of supplying the coefficient directly, it is caculated from a 60 dB decay time. This is the time required for the integrator to lose 99.9 % of its value or -60dB. This is useful for exponential decaying envelopes triggered by impulses.

in - input signal
decayTime - 60 dB decay time in seconds.

```
// plot({ Decay.ar(Impulse.ar(1), 0.01) });

// used as an envelope
play({ Decay.ar(Impulse.ar(XLine.kr(1,50,20), 0.25), 0.2, PinkNoise.ar, 0) });
```

ID: 513

Decay2 exponential decay

Decay2.ar(in, attackTime, decayTime, mul, add)

Decay has a very sharp attack and can produce clicks. Decay2 rounds off the attack by subtracting one Decay from another. Decay2.ar(in, attackTime, decayTime) is equivalent to:

Decay.ar(in, decayTime) - Decay.ar(in, attackTime)

in - input signal
attackTime - 60 dB attack time in seconds.
decayTime - 60 dB decay time in seconds.

```
//plot({ Decay2.ar(Impulse.ar(1), 0.001, 0.01) })

// since attack and decay are a difference of two Decays, if you swap the values,
// then the envelope turns upside down
//plot({ Decay2.ar(Impulse.ar(1), 0.01, 0.001) })

// used as an envelope
{ Decay2.ar(Impulse.ar(XLine.kr(1,50,20), 0.25), 0.01, 0.2, FSinOsc.ar(600)) }.play;

// compare the above with Decay used as the envelope
{ Decay.ar(Impulse.ar(XLine.kr(1,50,20), 0.25), 0.2, FSinOsc.ar(600), 0)  }.play;
```

ID: 514

DegreeToKey convert signal to modal pitch

DegreeToKey.ar(bufnum, in, octave, mul, add)

The input signal value is truncated to an integer value and used as an index into an octave repeating table of note values. Indices wrap around the table and shift octaves as they do.

bufnum - index of the buffer which contains the steps for each scale degree.
in - the input signal.
octave - the number of steps per octave in the scale. The default is 12.

```
(
// modal space
// mouse x controls discrete pitch in dorian mode
var scale, buffer;
scale = FloatArray[0, 2, 3.2, 5, 7, 9, 10]; // dorian scale
buffer = Buffer.alloc(s, scale.size,1, {| b| b.setnMsg(0, scale) });

play({
var mix;

mix =

 // lead tone
 SinOsc
(
   DegreeToKey
buffer.bufnum,
    MouseX            // mouse indexes into scale
    12,     // 12 notes per octave
1,     // mul = 1
    72     // offset by 72 notes
)
   + LFNoise1                 // add some low freq stereo detuning
  ).midicps,      // convert midi notes to hertz
```

```
0,
0.1)

 // drone 5ths
+ RLPF.ar(LFPulse.ar([48,55].midicps, 0.15),
SinOsc.kr(0.1, 0, 10, 72).midicps, 0.1, 0.1);

 // add some 70's euro-space-rock echo
CombN.ar(mix, 0.31, 0.31, 2, 1, mix)
})
)
```

ID: 515

K2A control rate to audio rate converter

K2A.ar(in)

To be able to play a control rate UGen into an audio rate UGen, sometimes the rate must be converted.
K2A converts via linear interpolation.

in - input signal

```
{ K2A.ar(WhiteNoise.kr(0.3)) }.scope;


// compare:

(
{
[
K2A.ar(WhiteNoise.kr(0.3)),
WhiteNoise.ar(0.3)
]
}.scope;
)


(
{
var freq, blockSize, sampleRate;
blockSize = Server.internal.options.blockSize; // usually 64
sampleRate = Server.internal.sampleRate;
freq = MouseX.kr(0.1, 40, 1) / blockSize * sampleRate;
[
K2A.ar(LFNoise0.kr(freq)),
```

```
LFNoise0.ar(freq)
] * 0.3
}.scope;
)
```

ID: 516

KeyState respond to the state of a key

superclass: UGen

***kr(keycode, minval, maxval, lag)**

keycode - The keycode value of the key to check. This corresponds to the keycode values passed into the keyDownActions of SCViews. See example below.
minval - Thevalue to output when the key is not pressed.
maxval - Thevalue to output when the key is pressed.
lag - A lag factor.

See also **MouseButton**, **MouseX**, **MouseY**

Note that this UGen does not prevent normal typing. It therefore may be helpful to select a GUI window rather than an SC document when using KeyState, as the latter will be altered by any keystrokes.

```
s.boot;

// execute the code below to find out a key's keycode
// the char and keycode of any key you press will be printed in the post window
(
    SCWindow     "I catch keystrokes"
w.view.keyDownAction = { arg view, char, modifiers, unicode, keycode;  [char, keycode].postln; };
w.front;
)

// then execute this and then press the 'j' key
(
        // something safe to type on
{ SinOsc.ar(800, 0, KeyState.kr(38, 0, 0.1)) }.play;
)
```

ID: 517

MouseButton mouse button ugen

superclass: UGen

*kr(minval, maxval, lag)

minval value when the key is not pressed
maxval value when the key is pressed

lag lag factor

see also **MouseX**, **MouseY**

```
//example

{ SinOsc.ar(MouseButton.kr(400, 440, 0.1), 0, 0.1) }.play;
{ SinOsc.ar(MouseButton.kr(400, 740, 2), 0, 0.1) }.play;
```

ID: 518

MouseX cursor ugen

superclass: UGen

*kr(minval, maxval, warp, lag)

minval, maxval range between left and right end of screen

warp mapping curve. 0 is linear, 1 is exponential (for freq or times e.g)
alternative: 'linear', 'exponential'

lag lag factor to dezpipper cursor movement

see also **MouseX**, **MouseButton**

```
//example

{ SinOsc.ar(MouseX.kr(40, 10000, 1), 0, 0.1) }.play;
```

ID: 519

MouseY cursor ugen

superclass: UGen

*kr(minval, maxval, warp, lag)

minval, maxval range between top and low end of screen

warp mapping curve. 0 is linear, 1 is exponential (for freq or times e.g)
alternative: 'linear', 'exponential'

lag lag factor to dezpipper cursor movement

see also **MouseY**, **MouseButton**

```
//example

{ SinOsc.ar(MouseY.kr(40, 10000, 1), 0, 0.1) }.play;
```

ID: 520

Slew slew rate limiter

Slew.ar(in, upSlope, downSlope, mul, add)

Limits the slope of an input signal. The slope is expressed in units per second.
in - input signal.
upSlope - maximum upward slope.
downSlope - maximum downward slope.

```
(
{
z = LFPulse.ar(800);
[z, Slew.ar(z, 4000, 4000)]
}.plot)
```

```
Has the effect of removing transients and higher frequencies.
(
{

z = Saw.ar(800,mul:0.2);
Slew.ar(z,400,400)

}.play
)
```

ID: 521

WrapIndex index into a table with a signal

WrapIndex.ar(bufnum, in, mul, add)
WrapIndex.kr(bufnum, in, mul, add)

The input signal value is truncated to an integer value and used as an index into the table.
Out of range index values are wrapped cyclically to the valid range.
bufnum - index of the buffer
in - the input signal.

```
(
// indexing into a table
s = Server.local;
t = [ 200, 300, 400, 500, 600, 800 ];
b = Buffer(s,t.size,1);

// alloc and set the values
s.listSendMsg( b.allocMsg( b.setnMsg(0, t) ).postln );

SynthDef("help-Index",{ arg out=0,i_bufnum=0;
Out.ar(0,
SinOsc.ar(
   WrapIndex
i_bufnum,
MouseX.kr(0, t.size * 3)
),
0,
0.5
)
)
}).play(s,[\i_bufnum,b.bufnum]);

)
```

25.5 Delays

ID: 522

AllpassC all pass delay line with cubic interpolation

AllpassC.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
AllpassC.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

All pass delay line with cubic interpolation. See also **[AllpassN]** which uses no interpolation, and **[AllpassL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufAllpassC]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ AllpassC.ar(WhiteNoise.ar(0.1), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + AllpassN.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + AllpassL.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)
```

```
(
{
z = WhiteNoise.ar(0.2);
z + AllpassC.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)


// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ AllpassC.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 523

AllpassL all pass delay line with linear interpolation

AllpassL.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
AllpassL.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

All pass delay line with linear interpolation. See also **[AllpassN]** which uses no interpolation, and **[AllpassC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufAllpassL]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ AllpassC.ar(WhiteNoise.ar(0.1), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + AllpassN.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + AllpassL.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)
```

```
(
{
z = WhiteNoise.ar(0.2);
z + AllpassC.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ AllpassL.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 524

AllpassN all pass delay line with no interpolation

AllpassN.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
AllpassN.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

All pass delay line with no interpolation. See also **[AllpassC]** which uses cubic interpolation, and **[AllpassL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufAllpassN]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ AllpassC.ar(WhiteNoise.ar(0.1), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + AllpassN.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + AllpassL.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)
```

```
(
{
z = WhiteNoise.ar(0.2);
z + AllpassC.ar(z, 0.01, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ AllpassN.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 525

BufAllpassC buffer based all pass delay line with cubic interpolation

BufAllpassC.ar(buf, in, delaytime, decaytime, mul, add)
BufAllpassC.kr(buf, in, delaytime, decaytime, mul, add)

All pass delay line with cubic interpolation which uses a buffer for its internal memory. See also **[BufAllpassN]** which uses no interpolation, and **[BufAllpassL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[AllpassC]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ BufAllpassC.ar(b.bufnum, WhiteNoise.ar(0.1), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassN.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
```

```
{
z = WhiteNoise.ar(0.2);
z + BufAllpassL.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassC.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ BufAllpassN.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 526

BufAllpassL buffer based all pass delay line with linear interpolation

BufAllpassL.ar(buf, in, delaytime, decaytime, mul, add)
BufAllpassL.kr(buf, in, delaytime, decaytime, mul, add)

All pass delay line with linear interpolation which uses a buffer for its internal memory. See also **[BufAllpassN]** which uses no interpolation, and **[BufAllpassC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[AllpassL]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ BufAllpassC.ar(b.bufnum, WhiteNoise.ar(0.1), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassN.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)
```

```
(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassL.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassC.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ BufAllpassL.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 527

BufAllpassN buffer based all pass delay line with no interpolation

BufAllpassN.ar(buf, in, delaytime, decaytime, mul, add)
BufAllpassN.kr(buf, in, delaytime, decaytime, mul, add)

All pass delay line with no interpolation which uses a buffer for its internal memory. See also **[BufAllpassC]** which uses cubic interpolation, and **[BufAllpassL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[AllpassN]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

// Since the allpass delay has no audible effect as a resonator on
// steady state sound ...

{ BufAllpassC.ar(b.bufnum, WhiteNoise.ar(0.1), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// ...these examples add the input to the effected sound and compare variants so that you can hear
// the effect of the phase comb:

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassN.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
```

```
{
z = WhiteNoise.ar(0.2);
z + BufAllpassL.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

(
{
z = WhiteNoise.ar(0.2);
z + BufAllpassC.ar(b.bufnum, z, XLine.kr(0.0001, 0.01, 20), 0.2)
}.play)

// used as an echo - doesn't really sound different than Comb,
// but it outputs the input signal immediately (inverted) and the echoes
// are lower in amplitude.
{ BufAllpassN.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 528

BufCombC buffer based comb delay line with cubic interpolation

BufCombCar(buf, in, delaytime, decaytime, mul, add)
BufCombC.kr(buf, in, delaytime, decaytime, mul, add)

Comb delay line with cubic interpolation which uses a buffer for its internal memory. See also **[BufCombN]** which uses no interpolation, and **[BufCombL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[CombC]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// allocate buffer
b = Buffer.alloc(s,44100,1);

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;
```

```
{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
{ BufCombC.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 529

BufCombL buffer based comb delay line with linear interpolation

BufCombLar(buf, in, delaytime, decaytime, mul, add)
BufCombL.kr(buf, in, delaytime, decaytime, mul, add)

Comb delay line with linear interpolation which uses a buffer for its internal memory. See also **[BufCombN]** which uses no interpolation, and **[BufCombC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[CombL]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// allocate buffer
b = Buffer.alloc(s,44100,1);

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;
```

```
{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
{ BufCombL.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 530

BufCombN buffer based comb delay line with no interpolation

BufCombNar(buf, in, delaytime, decaytime, mul, add)
BufCombN.kr(buf, in, delaytime, decaytime, mul, add)

Comb delay line with no interpolation which uses a buffer for its internal memory. See also **[BufCombL]** which uses linear interpolation, and **[BufCombC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[CombN]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// allocate buffer
b = Buffer.alloc(s,44100,1);

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ BufCombN.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ BufCombL.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;
```

```
{ BufCombC.ar(b.bufnum, WhiteNoise.ar(0.01), XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
{ BufCombN.ar(b.bufnum, Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 3) }.play;
```

ID: 531

BufDelayC buffer based simple delay line with cubic interpolation

BufDelayC.ar(buf, in, delaytime, mul, add)
BufDelayC.kr(buf, in, delaytime, mul, add)

Simple delay line with cubic interpolation which uses a buffer for its internal memory. See also **[BufDelayN]** which uses no interpolation, and **[BufDelayL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[DelayC]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
BufDelayC                           // input is mixed with delay via the add input
}.play
)
```

ID: 532

BufDelayL buffer based simple delay line with linear interpolation

BufDelayL.ar(buf, in, delaytime, mul, add)
BufDelayL.kr(buf, in, delaytime, mul, add)

Simple delay line with linear interpolation which uses a buffer for its internal memory. See also **[BufDelayN]** which uses no interpolation, and **[BufDelayC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[DelayL]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
BufDelayL                          // input is mixed with delay via the add input
}.play
)
```

ID: 533

BufDelayN buffer based simple delay line with no interpolation

BufDelayN.ar(buf, in, delaytime, mul, add)
BufDelayN.kr(buf, in, delaytime, mul, add)

Simple delay line with no interpolation which uses a buffer for its internal memory. See also **[BufDelayL]** which uses linear interpolation, and **[BufDelayC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[DelayN]**.

buf - buffer number.
in - the input signal.
delaytime - delay time in seconds.

```
// allocate buffer
b = Buffer.alloc(s,44100,1);

(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
BufDelayN                          // input is mixed with delay via the add input
}.play
)
```

ID: 534

CombC comb delay line with cubic interpolation

CombC.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
CombC.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

Comb delay line with cubic interpolation. See also **[CombN]** which uses no interpolation, and **[CombL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufCombC]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
```

```
{ CombC.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 535

CombL comb delay line with linear interpolation

CombL.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
CombL.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

Comb delay line with linear interpolation. See also **[CombN]** which uses no interpolation, and **[CombC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufCombL]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
```

```
{ CombL.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 536

CombN comb delay line with no interpolation

CombN.ar(in, maxdelaytime, delaytime, decaytime, mul, add)
CombN.kr(in, maxdelaytime, delaytime, decaytime, mul, add)

Comb delay line with no interpolation. See also **[CombL]** which uses linear interpolation, and **[CombC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufCombN]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.
decaytime - time for the echoes to decay by 60 decibels. If this time is negative then the feedback
coefficient will be negative, thus emphasizing only odd harmonics at an octave lower.

```
// These examples compare the variants, so that you can hear the difference in interpolation

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.play;

// with negative feedback:
{ CombN.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombL.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

{ CombC.ar(WhiteNoise.ar(0.01), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.play;

// used as an echo.
```

```
{ CombN.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.play;
```

ID: 537

Delay1 single sample delay

Delays the input by 1 sample.
Delay1.ar(in, mul, add)

in - input to be delayed.

```
(
plot({
var z;
z = Dust.ar(1000);
 [z, z - Delay1         // [ original, subtract delayed from original ]
}))
```

ID: 538

Delay2 two sample delay

Delays the input by 2 samples.
Delay2.ar(in, mul, add)

in - input to be delayed.

```
(
plot({
var z;
z = Dust.ar(1000);
 [z, z - Delay2         // [ original, subtract delayed from original ]
}))
```

ID: 539

DelayC simple delay line with cubic interpolation

DelayC.ar(in, maxdelaytime, delaytime, mul, add)
DelayC.kr(in, maxdelaytime, delaytime, mul, add)

Simple delay line with cubic interpolation. See also **[DelayN]** which uses no interpolation, and **[DelayL]** which uses linear interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufDelayC]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.

```
(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
DelayC                        // input is mixed with delay via the add input
}.play
)
```

ID: 540

DelayL simple delay line with linear interpolation

DelayL.ar(in, maxdelaytime, delaytime, mul, add)
DelayL.kr(in, maxdelaytime, delaytime, mul, add)

Simple delay line with linear interpolation. See also **[DelayN]** which uses no interpolation, and **[DelayC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufDelayL]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.

```
(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
DelayL                        // input is mixed with delay via the add input
}.play
)
```

ID: 541

DelayN simple delay line with no interpolation

DelayN.ar(in, maxdelaytime, delaytime, mul, add)
DelayN.kr(in, maxdelaytime, delaytime, mul, add)

Simple delay line with no interpolation. See also **[DelayL]** which uses linear interpolation, and **[DelayC]** which uses cubic interpolation. Cubic interpolation is more computationally expensive than linear, but more accurate.

See also **[BufDelayN]**.

in - the input signal.
maxdelaytime - the maximum delay time in seconds. used to initialize the delay buffer size.
delaytime - delay time in seconds.

```
(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
DelayN                        // input is mixed with delay via the add input
}.play
)
```

ID: 542

MultiTap multiple tap delay

MultiTap.ar(timesArray, levelsArray, in, mul, add, bufnum)

This is a wrapper which creates a multiple tap delay line using **RecordBuf** and **PlayBuf**.
timesArray - a Ref to an Array of delay times in seconds.
levelsArray - a Ref to an Array of amplitudes.
in - the input signal.
bufnum - the number of the buffer to use for the delay. This must be at least as long as the longest tap time.

```
s.boot;
b = Buffer.alloc(s, s.sampleRate);
(
{
MultiTap.ar(‘[0.1, 0.2, 0.3, 0.4], ‘[0.1, 0.2, 0.4, 0.8],
Decay.ar(Dust.ar(2), 0.1, PinkNoise.ar), bufnum:  b.bufnum)
}.play
)
```

ID: 543

PingPong stereo ping pong delay

PingPong.ar(bufnum, inputArray, delayTime, feedback, rotate)

Bounces sound between two outputs ... like a ping-pong ball.
PingPong is actually a compound built upon **RecordBuf** and **PlayBuf**.

bufnum - first index of a multi channel buffer .
inputArray - an array of audio inputs, the same size as your buffer.
delaytime - delay time in seconds..
feedback - feedback coefficient.
rotate - default 1: which rotates the inputArray by one step. (left -> right, right -> left)
rotation of 0 (or 2) would result in no rotation to the inputArray

```
(
s = Server.local;
s.waitForBoot({

b = Buffer.alloc(s,44100 * 2, 2);

SynthDef("help-PingPong",{ arg out=0,bufnum=0,feedback=0.5,delayTime=0.2;
var left, right;
left = Decay2.ar(Impulse.ar(0.7, 0.25), 0.01, 0.25,
SinOsc.ar(SinOsc.kr(3.7,0,200,500)));
right = Decay2.ar(Impulse.ar(0.5, 0.25), 0.01, 0.25,
Resonz.ar(PinkNoise.ar(4), SinOsc.kr(2.7,0,1000,2500), 0.2));

Out.ar(0,
PingPong.ar(bufnum, [left,right], delayTime, feedback, 1)
)
}).play(s,[\out, 0, \bufnum, b.bufnum,\feedback,0.5,\delayTime,0.1]);
});
)


(
```

```
s = Server.local;
s.waitForBoot({

b = Buffer.alloc(s,44100 * 2, 2);

SynthDef("help-PingPong",{ arg out=0,bufnum=0;
var left, right;
left = Decay2.ar(Impulse.ar(0.7, 0.25), 0.01, 0.25,
SinOsc.ar(SinOsc.kr(3.7,0,200,500)));
right = Decay2.ar(Impulse.ar(0.5, 0.25), 0.01, 0.25,
Resonz.ar(PinkNoise.ar(4), SinOsc.kr(2.7,0,1000,2500),
0.2));

Out.ar(0,
PingPong.ar(bufnum, [left,right] *  EnvGen.kr(Env([1, 1, 0], [2, 0.1])),
0.1, 0.8, 1)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
});
)




(

Patch({ arg buffer,feedback=0.5,delayTime=0.2;
var left, right;
left = Decay2.ar(Impulse.ar(0.7, 0.25), 0.01, 0.25,
SinOsc.ar(SinOsc.kr(3.7,0,200,500)));
right = Decay2.ar(Impulse.ar(0.5, 0.25), 0.01, 0.25,
Resonz.ar(PinkNoise.ar(4), SinOsc.kr(2.7,0,1000,2500), 0.2));

PingPong.ar(buffer.bufnumIr, [left,right], delayTime, feedback, 1)

}).gui

)
```

ID: 544

PitchShift

PitchShift.ar(in, windowSize, pitchRatio, pitchDispersion, timeDispersion, mul, add)

A time domain granular pitch shifter.
Grains have a triangular amplitude envelope and an overlap of 4:1.
in - the input signal.
windowSize - the size of the grain window in seconds. This value cannot be modulated.
pitchRatio - the ratio of the pitch shift. Must be from 0.0 to 4.0.
pitchDispersion - the maximum random deviation of the pitch from the pitchRatio.
timeDispersion - a random offset of from zero to timeDispersion seconds is added to the delay
of each grain. Use of some dispersion can alleviate a hard comb filter effect due to uniform
grain placement. It can also be an effect in itself. timeDispersion can be no larger than windowSize.

```
(
play({
z = Blip.ar(800, 6, 0.1);
PitchShift.ar(z, 0.02, Line.kr(0.1,4,20), 0, 0.0001)
}))

(
// pitch shift input - USE HEADPHONES to prevent feedback.
play({
 PitchShift
  AudioIn            // stereo audio input
  0.1,    // grain size
  MouseX          // mouse x controls pitch shift ratio
  0,     // pitch dispersion
  0.004   // time dispersion
)
}))

(
// use PitchShift to granulate input - USE HEADPHONES to prevent feedback.
```

```
// upper left corner is normal playback.  x = pitch dispersion, y = time dispersion
var grainSize;
grainSize = 0.5;
play({
 PitchShift
AudioIn.ar([1,2]),
grainSize,
  1,      // nominal pitch rate = 1
  MouseX             // pitch dispersion
MouseY.kr(0, grainSize) // time dispersion
)
}))
```

25.6 Envelopes

ID: 545

DetectSilence when input falls below a threshhold, evaluate doneAction

superclass: UGen

***ar(input, thresh, time, doneAction)**
***kr(input, thresh, time, doneAction)**

input - any source
thresh - when input falls below this, evaluate doneAction
time - the minimum duration of the for which input must fall below thresh before this triggers. The default is 0.1 seconds.
doneAction - an integer representing a done action. See **[UGen-doneActions]** for more detail.

If the signal input starts with silence at the beginning of the synth's duration, then DetectSilence will wait indefinitely until the first sound before starting to monitor for silence.

```
  //example

(
 SynthDef "detectSilence-help"    arg
var z;
z = SinOsc.ar(Rand(400, 700), 0, LFNoise2.kr(8, 0.2).max(0));
DetectSilence.ar(z, doneAction:2);
Out.ar(out, z);
}).send(s);
)

 s.sendMsg("/s_new"  "detectSilence-help"
 s.sendMsg("/s_new"  "detectSilence-help"
 s.sendMsg("/s_new"  "detectSilence-help"

(
 Task
```

```
loop({
   s.sendMsg("/s_new"  "detectSilence-help"
[0.5, 1].choose.wait;
})
}).play;
)
```

ID: 546

EnvGen envelope generator

superclass: UGen

Plays back break point envelopes. The envelopes are instances of the Env class. See the **[Env]** for more info. The arguments for **levelScale**, **levelBias**, and **timeScale** are polled when the EnvGen is triggered and remain constant for the duration of the envelope.

***ar(envelope, gate, levelScale, levelBias, timeScale, doneAction)**
***kr(envelope, gate, levelScale, levelBias, timeScale, doneAction)**

envelope - an instance of **Env**, or an Array of Controls. (See **[Control]** and the example below for how to use this.)
gate -this triggers the envelope and holds it open while > 0. If the Env is fixed-length (e.g. Env.linen, Env.perc), the gate argument is used as a simple trigger. If it is an sustaining envelope (e.g. Env.adsr, Env.asr), the envelope is held open until the gate becomes 0, at which point is released.
levelScale - scales the levels of the breakpoints.
levelBias - offsets the levels of the breakpoints.
timeScale - scales the durations of the segments.
doneAction - an integer representing an action to be executed when the env is finished playing. This can be used to free the enclosing synth, etc. See **[UGen-doneActions]** for more detail.

```
{ EnvGen.kr(Env.perc, 1.0, doneAction:  2) * SinOsc.ar(440,0,0.1) }.play;



// example

(
SynthDef("env-help", { arg out, gate;
var z;
z =  EnvGen.kr(Env.adsr,gate) * SinOsc.ar(440,0,0.1);
Out.ar(out, z)
}).send(s);
```

```
)

 s.sendMsg("/s_new"  "env-help"          // start a synth (silently, as gate defaults to 0)


 // turn on
s.sendMsg("/n_set", 1980, \gate, 1);

 // turn off
s.sendMsg("/n_set", 1980, \gate, 0);

 // it does not matter to what value the gate is set, as long as it is > 0
s.sendMsg("/n_set", 1980, \gate, 2);

s.sendMsg("/n_free", 1980);
```

Changing an Env while playing

```
(
SynthDef("env", { arg i_outbus=0;
var env, envctl;

  // make a dummy 8 segment envelope
env = Env.newClear(8);

  // create a control argument array
envctl = Control.names([\env]).kr( env.asArray );

ReplaceOut.kr(i_outbus, EnvGen.kr(envctl, doneAction:  2));
}).send(s);
)

(
 SynthDef "sine"
Out.ar(0, SinOsc.ar(In.kr(0), 0, 0.2));
```

```
}).send(s);
)

s.sendMsg(\c_set, 0, 800);

s.sendMsg(\s_new, \sine, 1001, 1, 0);

e = Env([700,900,900,800], [1,1,1]*0.4, \exp).asArray;
s.sendBundle(nil,[\s_new, \env, 1002, 2, 1001],[\n_setn, 1002, \env, e.size] ++ e);

f = Env([1000,1000,800,1000,900,1000], [1,1,1,1,1]*0.3, \step).asArray;
s.sendBundle(nil,[\s_new, \env, 1003, 2, 1001],[\n_setn, 1003, \env, f.size] ++ f);

s.sendMsg(\n_free, 1001);
```

Forced release of the EnvGen

If the gate of an EnvGen is set to -1 or below, then the envelope will cutoff immediately. The time for it to cutoff is the amount less than -1, with -1 being as fast as possible, -1.5 being a cutoff in 0.5 seconds, etc. The cutoff shape is linear.

```
  (
 SynthDef("stealMe", { arg gate = 1;
Out.ar(0, {BrownNoise.ar}.dup * EnvGen.kr(Env.asr, gate, doneAction:2))
}).send(s);
)

s.sendMsg(\s_new, \stealMe, 1001, 1, 0);

s.sendMsg(\n_set, 1001, \gate, -1.1); // cutoff in 0.1 seconds
```

If the synthDef has an arg named "gate", the convienience method of Node can be used:

```
Node-release(releaseTime)

d = { arg gate=1; {BrownNoise.ar}.dup * EnvGen.kr(Env.asr, gate, doneAction:2) }.play;
```

```
d.release(3);
```

Fast triggering tests

```
(
{
   EnvGen
Env.new([ 0.001, 1, 0.5, 0 ], [ 0.01, 0.3, 1 ], -4, 2, nil),
Impulse.kr(10)
) * SinOsc.ar(440,0,0.1)
}.play;
)

(
  {
   EnvGen
Env.perc( 0.1, 0.0, 0.5, 1, \welch ),
Impulse.kr(100),
timeScale:  0.1
) * SinOsc.ar(440,0,0.3)
}.play;
)
```

Modulating the levelScale

no,it doesn't take a ugen in

```
(
 {
  EnvGen
Env.asr( 0.1, 1.0, 0.5, \welch ),
1.0,
FSinOsc.ar(1.0).range(0.0,1.0),
timeScale:  0.1
) * SinOsc.ar(440,0,0.3)
}.play;

)
```

an .ir rate input, a float or an ir rate ugen like Rand would work

```
(
 {
  EnvGen
Env.asr( 0.1, 1.0, 0.5, \welch ),
1.0,
Rand(0.1,1.0),
timeScale:  0.1
) * SinOsc.ar(440,0,0.3)
}.play;
)
```

ID: 547

Free when triggered frees a node

superclass: UGen

*kr(trig, nodeID)

trig when triggered, frees node

nodeID node to be freed

```
 //example

s.boot;

SynthDef("a", { Out.ar(0, SinOsc.ar(800, 0, 0.2)) }).send(s);

SynthDef("b", { arg t_t=0; Out.ar(1, PinkNoise.ar(0.3)); Free.kr(t_t, 1001); }).send(s);

s.sendMsg(\s_new, \a, 1001, 0, 0);

s.sendMsg(\s_new, \b, 1002, 0, 0);

s.sendMsg(\n_set, 1002, \t_t, 1);

s.sendMsg(\s_new, \a, 1001, 0, 0);

s.sendMsg(\n_set, 1002, \t_t, 1);

s.sendMsg(\s_new, \a, 1001, 0, 0);

s.sendMsg(\n_set, 1002, \t_t, 1);
```

ID: 548

FreeSelf when triggered, free enclosing synth

superclass: UGen

free enclosing synth when input signal crosses from non-positive to positive

***kr(src)** **src** - input signal

```
 //example

(
SynthDef("freeSelf-help", { arg out, t_trig;
FreeSelf.kr(t_trig);
Out.ar(out, SinOsc.ar(400,0,0.2));
}).send(s);
)

 s.sendMsg("/s_new"  "freeSelf-help"
s.sendMsg("/n_set", 1731, \t_trig, 1);

 // a single impulse SynthDef:

(
SynthDef("dirac", { arg out, amp=0.1;
var u;
u = Impulse.ar(1);
FreeSelf.kr(u);
Out.ar(out, u * amp);
   // multiply by amp after using for release, so amp = 0
   // doesn't cause synth buildup.
}).send(s);
)

(
 Task
loop({
fork {
```

```
exprand(34, 156).do {| i|
i = i + 1;
s.sendMsg("/s_new", "dirac", -1,0,0, \amp, 1 / i);
(0.006 * i).wait;
};
};
1.wait;
})
} .play;
)
```

ID: 549

Line line generator

Line.ar(start, end, dur, mul, add, doneAction)
Line.kr(start, end, dur, mul, add, doneAction)

Generates a line from the start value to the end value.
start - starting value
end - ending value
dur - duration in seconds
doneAction - a doneAction to be evaluated when the Line is completed. See **[UGen-doneActions]** for more detail.

```
// XLine is usually better than Line for frequency
play({ SinOsc.ar(Line.kr(200,17000,10),0,0.1) });
```

ID: 550

Linen simple linear envelope generator

Linen.kr(gate = 1.0, attackTime = 0.01, susLevel = 1.0, releaseTime = 1.0, doneAction = 0)

See **[UGen-doneActions]** for more detail.

```
// trigged
(
SynthDef("help-Linen",{ arg out = 0;
Out.ar(out,
Linen.kr(Impulse.kr(2), 0.01, 0.6, 1.0, doneAction:  0) * SinOsc.ar(440, 0, 0.1)
)
}).play;
)

// play once and end the synth
(
SynthDef "help-Linen"   arg
Out.ar(out,
Linen.kr(Impulse.kr(0), 0.01, 0.6, 1.0, doneAction:  2) * SinOsc.ar(440, 0, 0.1)
)
}).play;
)

// play once and sustain
(
x = SynthDef("help-Linen",{ arg gate = 1, out = 0; // use gate arg for release
Out.ar(out,
Linen.kr(gate, 0.01, 0.6, 1.0, doneAction:  2) * SinOsc.ar(440, 0, 0.1)
)
}).play;
)
              // change the release time

// longer gate, can pass in duration
(
SynthDef("help-Linen",{ arg out = 0, dur = 0.1;
```

```
var gate;
gate = Trig.kr(1.0, dur);
Out.ar(out,
Linen.kr(gate, 0.01, 0.6, 1.0, doneAction:  2) * SinOsc.ar(440, 0, 0.1)
)
}).play(nil, [\out, 0, \dur, 2.0]);
)



// used below in a Routine varying the releaseTime
(
SynthDef("help-Linen",{ arg out=0,freq=440,attackTime=0.01,susLevel=0.6,releaseTime=0.1;
Out.ar(out,
Linen.kr(Impulse.kr(0), attackTime, susLevel, releaseTime, doneAction:  2)
* SinOsc.ar(freq, 0, 0.1)
)
}).send(s);
)

(
// debussey sleeping through math class
x = Pbrown(0.01, 2.0, 0.2, inf).asStream;
Routine
loop({
Synth.grain("help-Linen",[\freq, (rrand(20, 50) * 2).midicps, \releaseTime, x.next]);
0.25.wait;
})
}).play(TempoClock.default)
)




(
SynthDef("help-Linen",{ arg out = 0;
Out.ar(out,

Linen.kr(Impulse.kr(2),
```

```
0.01,
   // sustain level is polled at time of trigger
FSinOsc.kr(0.1).range(0, 1),
1.0,
doneAction:  0)

* SinOsc.ar(440, 0, 0.1)
)
}).play;
)
```

ID: 551

Pause when triggered pauses a node

superclass: UGen

*kr(gate, nodeID)

gate when gate is 0, node is paused, when 1 it runs

nodeID node to be paused

```
 //example

s.boot;

SynthDef("a", { Out.ar(0, SinOsc.ar(800, 0, 0.2)) }).send(s);

SynthDef("b", { arg gate=1; Out.ar(1, PinkNoise.ar(0.3)); Pause.kr(gate, 1001); }).send(s);

s.sendMsg(\s_new, \a, 1001, 0, 0);

s.sendMsg(\s_new, \b, 1002, 0, 0);

s.sendMsg(\n_set, 1002, \gate, 0);

s.sendMsg(\n_set, 1002, \gate, 1);
```

ID: 552

PauseSelf when triggered, pause enclosing synth

superclass: UGen

pause enclosing synth when input signal crosses from non-positive to positive

***kr(src)** **src** - input signal

```
 //example

(
SynthDef("pauseSelf-help", { arg out, t_trig;
PauseSelf.kr(t_trig);
Out.ar(out, SinOsc.ar(400,0,0.2));
}).send(s);
)

 s.sendMsg("/s_new"  "pauseSelf-help"
s.sendMsg("/n_set", 1731, \t_trig, 1);
s.sendMsg("/n_run", 1731, 1);
s.sendMsg("/n_set", 1731, \t_trig, 1);
s.sendMsg("/n_free", 1731);
```

ID: 553

UGen Done Actions

A number of UGens implement doneActions. These allow one to optionally free or pause the enclosing synth and other related nodes when the UGen is finished. These include **[EnvGen]**, **[Line]**,**[XLine]**,**[Linen]**,**[DetectSilence]** and some **[Demand]** ugens.

The available done actions are as follows:

0 do nothing when the UGen is finished
1 pause the enclosing synth, but do not free it
2 free the enclosing synth
3 free both this synth and the preceding node
4 free both this synth and the following node
5 free this synth; if the preceding node is a group then do g_freeAll on it, else free it
6 free this synth; if the following node is a group then do g_freeAll on it, else free it
7 free this synth and all preceding nodes in this group
8 free this synth and all following nodes in this group
9 free this synth and pause the preceding node
10 free this synth and pause the following node
11 free this synth and if the preceding node is a group then do g_deepFree on it, else free it
12 free this synth and if the following node is a group then do g_deepFree on it, else free it
13 free this synth and all other nodes in this group (before and after)
14 free the enclosing group and all nodes within it (including this synth)

For information on freeAll and deepFree, see **[Group]** and **[Server-Command-Reference]**.

ID: 554

XLine exponential line generator

XLine.ar(start, end, dur, mul, add, doneAction)
XLine.kr(start, end, dur, mul, add, doneAction)

Generates an exponential curve from the start value to the end value. Both the start and end values
must be non-zero and have the same sign.
start - starting value
end - ending value
dur - duration in seconds
doneAction - a doneAction to be evaluated when the Line is completed. See **[UGen-doneActions]** for more detail.

```
play({ SinOsc.ar(XLine.kr(200,17000,10),0,0.1) });
```

25.7 FFT

ID: 555

Convolution real-time convolver

Convolution.ar(in, kernel, framesize, mul, add)

Strict convolution of two continuously changing inputs. Also see [Convolution2] for a cheaper CPU cost alternative for the case of a fixed kernel which can be changed with a trigger message.

#1a1aff#236e25//see ch18 #1a1affhttp://www.dspguide.com/ch18.htm
Steven W Smith

in - processing target
kernel - processing kernel.
framesize- size of FFT frame, must be a power of two

```
(

{ var input, kernel;

input=AudioIn.ar(1);
kernel= Mix.ar(LFSaw.ar([300,500,800,1000]*MouseX.kr(1.0,2.0),0,1.0));

 //must have power of two framesize
Out.ar(0,Convolution.ar(input,kernel, 1024, 0.5));
}.play;

)

(
//must have power of two framesize- FFT size will be sorted by Convolution to be double this
//maximum is currently a=8192 for FFT of size 16384
a=2048;
s = Server.local;
//kernel buffer
g = Buffer.alloc(s,a,1);
)
```

```
(
//random impulse response
g.set(0,1.0);
100.do({arg i; g.set(a.rand, 1.0.rand)});


{ var input, kernel;

input=AudioIn.ar(1);
kernel= PlayBuf.ar(1,g.bufnum,BufRateScale.kr(g.bufnum),1,0,1);

Out.ar(0,Convolution.ar(input,kernel, 2*a, 0.5));
}.play;

)
```

ID: 556

Convolution2 real-time convolver

Convolution2.ar(in, bufnum, trigger, framesize, mul, add)

Strict convolution with fixed kernel which can be updated using a trigger signal.

#1a1aff#236e25//see ch18 #1a1affhttp://www.dspguide.com/ch18.htm
Steven W Smith

in - processing target
bufnum - buffer index for the fixed kernel, may be modulated in combination with the trigger
trigger - update the kernel on a change from <=0 to >0
framesize - size of FFT frame, must be a power of two. Convolution uses twice this number internally, maximum value you can give this argument is 2^16=65536. Note that it gets progressively more expensive to run for higher powers! 512, 1024, 2048, 4096 standard.

```
 //allocate three buffers
b = Buffer.alloc(s,2048);
c = Buffer.alloc(s,2048);
d = Buffer.alloc(s,2048);

b.zero;
c.zero;
d.zero;
)

(
50.do({ | it|  c.set(20*it+10, 1.0.rand); });
3.do({ | it|  b.set(400*it+100, 1); });
20.do({ | it|  d.set(40*it+20, 1); });
)


(
```

```
SynthDef( "conv-test", { arg kernel, trig=0;
var input;

input=Impulse.ar(1);

 //must have power of two framesize
Out.ar(0,Convolution2.ar(input,kernel,trig,2048, 0.5));
}).send(s)

)


x = Synth.new("conv-test",[\kernel,b.bufnum]);

// changing the buffer number:
x.set(\kernel,c.bufnum);
x.set(\trig,0);
      \trig     // after this trigger, the change will take effect.
x.set(\kernel,d.bufnum);
x.set(\trig,0);
      \trig     // after this trigger, the change will take effect.

d.zero;
        | it|                        // changing the buffers' contents
x.set(\trig,0);
      \trig     // after this trigger, the change will take effect.

x.set(\kernel,b.bufnum);
x.set(\trig,0);
      \trig     // after this trigger, the change will take effect.



////next example

    Buffer        "sounds/a11wlk01.wav"

(
{ var input, kernel;
```

```
input=AudioIn.ar(1);

 //must have power of two framesize
Out.ar(0,Convolution2.ar(input,b.bufnum,0,512, 0.5));
}.play;

)


//another example

(
//must have power of two framesize- FFT size will be sorted by Convolution2 to be double this
//maximum is currently a=8192 for FFT of size 16384
a=2048;
s = Server.local;
//kernel buffer
g = Buffer.alloc(s,a,1);
)

(
g.set(0,1.0);
100.do({arg i; g.set(a.rand, (i+1).reciprocal)});
)

(
//random impulse response

{
var input,inputAmp,threshhold,gate;

input = AudioIn.ar(1);
inputAmp = Amplitude.kr(input);
                    // noise gating threshold
gate = Lag.kr(inputAmp > threshhold, 0.01);

Out.ar(0,Convolution2.ar(input*gate,g.bufnum,0, a, 0.5));
}.play;

)
```

```
//one last example
(
b = Buffer.alloc(s, 512, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);
)

(
{ var input, kernel;

input=AudioIn.ar(1);

 //must have power of two framesize
Out.ar(0,Convolution2.ar(input,b.bufnum,0, 512, 0.5));
}.play;

)
```

ID: 557

FFT Fast Fourier Transform

The fast fourier transform analyzes the frequency content of a signal. See also **[FFT Overview]**.

FFT(buffer, input)

FFT uses a local buffer for holding the buffered audio. The window size corresponds to the buffer size. The overlap is 2.

```
s = Server.local.boot;

b = Buffer.alloc(s,2048,1);

(
SynthDef("help-noopFFT", { arg out=0,bufnum=0;
var in, chain;
in = WhiteNoise.ar(0.01);
chain = FFT(bufnum, in);
chain.inspect; // its an FFT
Out.ar(out,
  IFFT        // inverse FFT
);
}).play(s,[\out,0,\bufnum,b.bufnum]);
)

(
SynthDef("help-sineFFT", { arg out=0,bufnum=0;
var in, chain;
in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08,0,6,6.2).squared, 0, 100,800));
chain = FFT(bufnum, in);
Out.ar(out, IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)

(
SynthDef("help-magAbove", { arg out=0,bufnum=0;
```

```
var in, chain;
in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08,0,6,6.2).squared, 0, 100,800));
 //in = WhiteNoise.ar(0.2);
chain = FFT(bufnum, in);
chain = PV_MagAbove(chain, 310);
Out.ar(out, 0.5 * IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)

(
SynthDef("help-brick", { arg out=0,bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.2)}.dup;
chain = FFT(bufnum, in);
chain = PV_BrickWall(chain, SinOsc.kr(0.1));
Out.ar(out, IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)

(
SynthDef("help-randcomb", { arg out=0,bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.8)}.dup;
chain = FFT(bufnum, in);
chain = PV_RandComb(chain, 0.95, Impulse.kr(0.4));
Out.ar(out, IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)

(
SynthDef("help-rectcomb", { arg out=0,bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.2)}.dup;
chain = FFT(bufnum, in);
chain = PV_RectComb(chain, 8, LFTri.kr(0.097,0,0.4,0.5),
LFTri.kr(0.24,0,-0.5,0.5));
Out.ar(out, IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)
```

```
(
SynthDef("help-magFreeze", { arg out=0,bufnum=0;
var in, chain;
 in = SinOsc    LFNoise1    5.2,250,400
chain = FFT(bufnum, in);
// moves in and out of freeze
chain = PV_MagFreeze(chain, SinOsc.kr(0.2) );
Out.ar(out, 0.5 * IFFT(chain));
}).play(s,[\out,0,\bufnum,b.bufnum]);
)
```

ID: 558

FFT Overview

FFT and IFFT

SuperCollider implements a number of UGens supporting FFT based processing. The most basic of these are **[FFT]** and **[IFFT]** which convert data between the time and frequency domains:

FFT(buffer, input)
IFFT(buffer)

FFT stores spectral data in a local buffer (see **[Buffer]**) in the following order: DC, nyquist, real 1f, imag 1f, real 2f, imag 2f, ... real (N-1)f, imag (N-1)f, where f is the frequency corresponding to the window size, and N is the window size / 2.

The buffer's size must correspond to a power of 2. The window size is equivalent to the buffer size, and the window overlap is fixed at 2. Both FFT and IFFT use a Welch window, the combination of which (i.e. Welch ²) is a Hanning window.

Phase Vocoder UGens and Spectral Processing

In between an FFT and an IFFT one can chain together a number of Phase Vocoder UGens (i.e. 'PV_...') to manipulate blocks of spectral data before reconversion. The process of buffering the appropriate amount of audio, windowing, conversion, overlap-add, etc. is handled for you automatically.

```
s = Server.local.boot;
b = Buffer.alloc(s,2048,1);

(
{ var in, chain;
in = {WhiteNoise.ar(0.8)}.dup;
chain = FFT(b.bufnum, in);
chain = PV_RandComb(chain, 0.95, Impulse.kr(0.4));
IFFT(chain);
}.play(s);
)
b.free;
```

PV Ugens write their output data *in place*, i.e. back into the same buffer from which they read. PV UGens which require two buffers write their data into the first buffer, usually called 'bufferA'.

```
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
 d = Buffer        "sounds/a11wlk01.wav"
)

(
{ var inA, chainA, inB, chainB, chain;
inA = LFSaw.ar([100, 150], 0, 0.2);
inB = PlayBuf.ar(1, d.bufnum, BufRateScale.kr(d.bufnum), loop:  1);
chainA = FFT(b.bufnum, inA);
chainB = FFT(c.bufnum, inB);
chain = PV_MagMul(chainA, chainB); // writes into bufferA
0.1 * IFFT(chain);
}.play(s);
)
[b, c, d].do(_.free);
```

Because each PV UGen overwrites the output of the previous one, it is necessary to copy the data to an additional buffer at the desired point in the chain in order to do parallel processing of input without using multiple FFT UGens. **[PV_Copy]** allows for this.

```
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
)

 //// proof of concept
(
x = { var inA, chainA, inB, chainB, chain;
inA = LFClipNoise.ar(100);
chainA = FFT(b.bufnum, inA);
chainB = PV_Copy(chainA, c.bufnum);
  IFFT          IFFT          // cancels to zero so silent!
}.play(s);
```

```
)
x.free;
 // IFFTed frames contain the same windowed output data
b.plot(\b, Rect(200, 430, 700, 300)); c.plot(\c, Rect(200, 100, 700, 300));
[b, c].do(_.free);
```

Note that PV UGens convert as needed between cartesian (complex) and polar representations, therefore when using multiple PV UGens it may be impossible to know in which form the values will be at any given time. FFT produces complex output (see above), so while the following produces a reliable magnitude plot:

```
b = Buffer.alloc(s,1024);
a = { FFT(b.bufnum, LFSaw.ar(4000)); 0.0 }.play;
(
b.getn(0, 1024, { arg buf;
var z, x;
z = buf.clump(2).flop;
z = [Signal.newFrom(z[0]), Signal.newFrom(z[1])];
x = Complex(z[0], z[1]);
{x.magnitude.plot}.defer
})
)
a.free; b.free;
```

any Synth using PV UGens might not.

PV and FFT UGens in the Standard Library

The following PV UGens are included in the standard SC distribution:

[FFT] Fast Fourier Transform
[IFFT] Inverse Fast Fourier Transform
[PV_Add] complex addition
[PV_BinScramble] scramble bins
[PV_BinShift] shift and stretch bin position
[PV_BinWipe] combine low and high bins from two inputs
[PV_BrickWall] zero bins
[PV_ConformalMap] complex plane attack
[PV_Copy] copy an FFT buffer
[PV_CopyPhase] copy magnitudes and phases

[PV_Diffuser] random phase shifting
[PV_HainsworthFoote]
[PV_JensenAndersen]
[PV_LocalMax] pass bins which are a local maximum
[PV_MagAbove] pass bins above a threshold
[PV_MagBelow] pass bins below a threshold
[PV_MagClip] clip bins to a threshold
[PV_MagFreeze] freeze magnitudes
[PV_MagMul] multiply magnitudes
[PV_MagNoise] multiply magnitudes by noise
[PV_MagShift] shift and stretch magnitude bin position
[PV_MagSmear] average magnitudes across bins
[PV_MagSquared] square magnitudes
[PV_Max] maximum magnitude
[PV_Min] minimum magnitude
[PV_Mul] complex multiply
[PV_PhaseShift]
[PV_PhaseShift270] shift phase by 270 degrees
[PV_PhaseShift90] shift phase by 90 degrees
[PV_RandComb] pass random bins
[PV_RandWipe] crossfade in random bin order
[PV_RectComb] make gaps in spectrum
[PV_RectComb2] make gaps in spectrum

ID: 559

IFFT Inverse Fast Fourier Transform

The inverse fast fourier transform converts from frequency content to a signal. See also **[FFT Overview]**.

IFFT(buffer)

```
s = Server.local;


b = Buffer.alloc(s,2048,1);



SynthDef("help-noopFFT", { arg out=0,bufnum=0;
var in, chain;
in = WhiteNoise.ar(0.01);
chain = FFT(bufnum, in);
chain.inspect; // its an FFT
Out.ar(out,
  IFFT        // inverse FFT
);
}).play(s,[\out,0,\bufnum,b.bufnum]);
```

See **FFT** for more examples.

ID: 560

PV_Add complex addition

PV_Add.ar(bufferA, bufferB)

Complex Addition: RealA + RealB, ImagA + ImagB
bufferA - fft buffer A.
bufferB - fft buffer B.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-add", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum;
var inA, chainA, inB, chainB, chain ;
inA = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
inB =  PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum) * 0.5, loop:  1);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_Add(chainA, chainB);
Out.ar(out,  0.1 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum, d.bufnum]);
)
```

ID: 561

PV_BinScramble scramble bins

PV_BinScramble.ar(buffer, wipe, width, trig)

Randomizes the order of the bins.
The trigger will select a new random ordering.
buffer - fft buffer.
wipe - scrambles more bins as wipe moves from zero to one.
width - a value from zero to one, indicating the maximum randomized distance of a bin from its
original location in the spectrum.
trig - a trigger selects a new random ordering.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)


(
//trig with MouseY
SynthDef("help-binScramble", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_BinScramble(chain, MouseX.kr , 0.1, MouseY.kr > 0.5 );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 562

PV_BinShift shift and stretch bin position

PV_BinShift.ar(buffer, stretch, shift)

Shift and scale the positions of the bins.
Can be used as a very crude frequency shifter/scaler.
buffer - fft buffer.
stretch - scale bin location by factor.
shift - add an offset to bin position.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-binStretch", { arg out=0, bufnum=0;
var in, chain;
in = LFSaw.ar(200, 0, 0.2);
chain = FFT(bufnum, in);
chain = PV_BinShift(chain, MouseX.kr(0.25, 4, \exponential) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-binStretch2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_BinShift(chain, MouseX.kr(0.25, 4, \exponential) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)

(
```

```
SynthDef("help-binShift", { arg out=0, bufnum=0;
var in, chain;
in = LFSaw.ar(200, 0, 0.2);
chain = FFT(bufnum, in);
chain = PV_BinShift(chain, 1, MouseX.kr(-128, 128) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-binShift2", {  arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_BinShift(chain, 1, MouseX.kr(-128, 128) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 563

PV_BinWipe combine low and high bins from two inputs

PV_BinWipe.ar(bufferA, bufferB, wipe)

Copies low bins from one input and the high bins of the other.
bufferA - fft buffer A.
bufferB - fft buffer B.
wipe - can range between -1 and +1.
if wipe == 0 then the output is the same as inA.
if wipe > 0 then it begins replacing with bins from inB from the bottom up.
if wipe < 0 then it begins replacing with bins from inB from the top down.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-binWipe", { arg out=0,bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain;
inA = WhiteNoise.ar(0.2);
inB = LFSaw.ar(100, 0, 0.2);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_BinWipe(chainA, chainB, MouseX.kr(-1, 1));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum ]);
)

(
SynthDef("help-binWipe2", { arg out=0,bufnumA=0, bufnumB=1, soundBufnum=2;
var inA, chainA, inB, chainB, chain;
inA = WhiteNoise.ar(0.2);
inB = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chainA = FFT(bufnumA, inA);
```

```
chainB = FFT(bufnumB, inB);
chain = PV_BinWipe(chainA, chainB, MouseX.kr(-1, 1));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum, d.bufnum]);
)
```

ID: 564

PV_BrickWall zero bins

PV_BrickWall.ar(buffer, wipe)

Clears bins above or below a cutoff point.
buffer - fft buffer.
wipe - can range between -1 and +1.
if wipe == 0 then there is no effect.
if wipe > 0 then it acts like a high pass filter, clearing bins from the bottom up.
if wipe < 0 then it acts like a low pass filter, clearing bins from the top down.

```
s.boot;


b = Buffer.alloc(s,2048,1);


(
SynthDef("help-brick", { arg out=0, bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.2)}.dup;
chain = FFT(bufnum, in);
chain = PV_BrickWall(chain, SinOsc.kr(0.1));
Out.ar(out, IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
```

ID: 565

PV_ConformalMap complex plane attack

PV_ConformalMap.ar(buffer, real, imag)

Applies the conformal mapping z -> (z-a)/(1-za*) to the phase vocoder bins z with a given by the real and imag imputs to the UGen.

ie, makes a transformation of the complex plane so the output is full of phase vocoder artifacts but may be musically fun. Usually keep | a| <1 but you can of course try bigger values to make it really noisy. a=0 should give back the input mostly unperturbed.

See http://mathworld.wolfram.com/ConformalMapping.html

buffer - buffer number of buffer to act on, passed in through a chain (see examples below).
real - real part of a.
imag - imaginary part of a.

```
//explore the effect
(
SynthDef "conformer1"
var in, chain;
in = AudioIn.ar(1,0.5);
chain = FFT(0, in);
chain=PV_ConformalMap(chain, MouseX.kr(-1.0,1.0), MouseY.kr(-1.0,1.0));
Out.ar(0, Pan2.ar(IFFT(chain),0));
}).load(s);
)

s.sendMsg("/b_alloc", 0, 1024, 1);
s.sendMsg("/s_new", "conformer1", 2002, 1, 0);
s.sendMsg("/n_free", 2002);



(
SynthDef "conformer2"
```

```
var in, chain, out;
in = Mix.ar(LFSaw.ar(SinOsc.kr(Array.rand(3,0.1,0.5),0,10,[1,1.1,1.5,1.78,2.45,6.7]*220),0,0.3));
chain = FFT(0, in);
chain=PV_ConformalMap(chain, MouseX.kr(0.01,2.0, 'exponential'), MouseY.kr(0.01,10.0, 'exponential'));

out=IFFT(chain);

Out.ar(0, Pan2.ar(CombN.ar(out,0.1,0.1,10,0.5,out),0));
}).load(s);
)



s.sendMsg("/b_alloc", 0, 2048, 1);
s.sendMsg("/s_new", "conformer2", 2002, 1, 0);
s.sendMsg("/n_free", 2002);
```

ID: 566

PV_Copy copy an FFT buffer

PV_Copy.ar(bufferA, bufferB)

Copies the spectral frame in bufferA to bufferB at that point in the chain of PV UGens. This allows for parallel processing of spectral data without the need for multiple FFT UGens, and to copy out data at that point in the chain for other purposes. bufferA and bufferB must be the same size.
bufferA - source buffer.
bufferB - destination buffer.

See also **[FFT Overview]**.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
e = Buffer.alloc(s,2048,1);
f = Buffer.alloc(s,2048,1);
)

//// proof of concept
(
x = { var inA, chainA, inB, chainB, chain;
inA = LFClipNoise.ar(100);
chainA = FFT(b.bufnum, inA);
chainB = PV_Copy(chainA, c.bufnum);
 IFFT           IFFT          // cancels to zero so silent!
}.play(s);
)
x.free;
// IFFTed frames contain the same windowed output data
b.plot(\b, Rect(200, 430, 700, 300)); c.plot(\c, Rect(200, 100, 700, 300));


//// crossfade between original and magmul-ed whitenoise
(
```

```
x = { var in, in2, chain, chainB, chainC;
in = PlayBuf.ar(1, d.bufnum, BufRateScale.kr(d.bufnum), loop:  1);
in2 = WhiteNoise.ar;
chain = FFT(b.bufnum, in);
chainB = FFT(c.bufnum, in2);
chainC = PV_Copy(chain, e.bufnum);
chainB = PV_MagMul(chainB, chainC);
XFade2.ar(IFFT(chain), IFFT(chainB) * 0.1, SinOsc.kr(0.1, 1.5pi), 0.25);
}.play(s);
)
x.free;


//// as previous but with Blip for 'vocoder' cross synthesis effect
(
x = { var in, in2, chain, chainB, chainC;
in = PlayBuf.ar(1, d.bufnum, BufRateScale.kr(d.bufnum), loop:  1);
in2 = Blip.ar(100, 50);
chain = FFT(b.bufnum, in);
chainB = FFT(c.bufnum, in2);
chainC = PV_Copy(chain, e.bufnum);
chainB = PV_MagMul(chainB, chainC);
XFade2.ar(IFFT(chain), IFFT(chainB) * 0.1, SinOsc.ar(0.1), 0.25);
}.play(s);
)
x.free;


//// Spectral 'pan'
(
x = { var in, chain, chainB, pan;
in = PlayBuf.ar(1, d.bufnum, BufRateScale.kr(d.bufnum), loop:  1);
chain = FFT(b.bufnum, in);
chainB = PV_Copy(chain, c.bufnum);
pan = MouseX.kr(0.001, 1.001, 'exponential') - 0.001;
chain = PV_BrickWall(chain, pan);
chainB = PV_BrickWall(chainB, -1 + pan);
0.5 * IFFT([chain, chainB]);
}.play(s);
)
```

```
x.free;


//// Multiple Magnitude plots
(
x = { var in, chain, chainB, chainC;
in = WhiteNoise.ar;
chain = FFT(b.bufnum, in);
PV_Copy(chain, c.bufnum); // initial spectrum
chain = PV_RectComb(chain, 20, 0, 0.2);
PV_Copy(chain, e.bufnum); // after comb
2.do({chain = PV_MagSquared(chain)});
PV_Copy(chain, f.bufnum); // after magsquared
0.00001 * Pan2.ar(IFFT(chain));
}.play(s);
)
x.free;

(
c.getToFloatArray(action:  { arg array;
var z, x;
z = array.clump(2).flop;
 // Initially data is in complex form
z = [Signal.newFrom(z[0]), Signal.newFrom(z[1])];
x = Complex(z[0], z[1]);
{x.magnitude.plot('Initial', Rect(200, 560, 700, 200))}.defer
});
e.getToFloatArray(action:  { arg array;
var z, x;
z = array.clump(2).flop;
 // RectComb doesn't convert, so it's still complex
z = [Signal.newFrom(z[0]), Signal.newFrom(z[1])];
x = Complex(z[0], z[1]);
{x.magnitude.plot('After RectComb', Rect(200, 330, 700, 200))}.defer
});
f.getToFloatArray(action:  { arg array;
var z, x;
z = array.clump(2).flop;
 // MagSquared converts to Polar
x = Signal.newFrom(z[0]); // magnitude first
```

```
{x.plot('After MagSquared', Rect(200, 100, 700, 200))}.defer
})
)

[b, c, d, e, f].do(_.free); // free the buffers
```

ID: 567

PV_CopyPhase copy magnitudes and phases

PV_CopyPhase.ar(bufferA, bufferB)

Combines magnitudes of first input and phases of the second input.

bufferA - fft buffer A.
bufferB - fft buffer B.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-copyPhase", { arg out=0, bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain;
inA = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800)); inB = WhiteNoise.ar(0.2);

chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_CopyPhase(chainA, chainB);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum ]);
)

(
SynthDef("help-copyPhase2", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum=2;
var inA, chainA, inB, chainB, chain;
inA = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
inB =  SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_CopyPhase(chainA, chainB);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum, d.bufnum]);
```

)

ID: 568

PV_Diffuser random phase shifting

PV_Diffuser.ar(buffer, trig)

Adds a different constant random phase shift to each bin.
The trigger will select a new set of random phases.
buffer - fft buffer.
trig - a trigger selects a new set of random values.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
//trig with MouseY
SynthDef("help-diffuser", { arg out=0, bufnum=0 ;
var in, chain;
in = Mix.ar(SinOsc.ar(200 * (1..10), 0, Array.fill(10, {rrand(0.1, 0.2)}) ));
chain = FFT(bufnum, in);
chain = PV_Diffuser(chain, MouseY.kr > 0.5 );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum ]);
)

(
//trig with MouseY
SynthDef("help-diffuser2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_Diffuser(chain, MouseY.kr > 0.5 );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 569

PV_HainsworthFoote

FFT onset detector based on work described in

Hainsworth, S. (2003) Techniques for the Automated Analysis of Musical Audio. PhD, University of Cambridge engineering dept. See especially p128. The Hainsworth metric is a modification of the Kullback Liebler distance.

The onset detector has general ability to spot spectral change, so may have some ability to track chord changes aside from obvious transient jolts, but there's no guarantee it won't be confused by frequency modulation artifacts.

Hainsworth metric on it's own gives good results but Foote might be useful in some situations: experimental.

Class Methods

*ar(buffer, proph=0.0, propf=0.0, threshold=1.0, waittime=0.04)

buffer- FFT buffer to read from

proph- What strength of detection signal from Hainsworth metric to use.

propf- What strength of detection signal from Foote metric to use. The Foote metric is normalised to [0.0,1.0]

threshold- Threshold hold level for allowing a detection

waittime- If triggered, minimum wait until a further frame can cause another spot (useful to stop multiple detects on heavy signals)

Examples

```
//just Hainsworth metric with low threshold
(
b=Buffer.alloc(s,2048,1);
```

```
SynthDef \fftod
{
var source1, detect;

source1= AudioIn.ar(1);

detect= PV_HainsworthFoote.ar(FFT(b.bufnum,source1), 1.0, 0.0);

Out.ar(0,SinOsc.ar([440,445],0,Decay.ar(0.1*detect,0.1)));
}).play(s);
)



//spot note transitions
(
b=Buffer.alloc(s,2048,1);

SynthDef \fftod
{
var source1, detect;

source1= LFSaw.ar(LFNoise0.kr(1,90,400),0,0.5);

detect= PV_HainsworthFoote.ar(FFT(b.bufnum,source1), 1.0, 0.0, 0.9, 0.5);

Out.ar(0,Pan2.ar(source1,-1.0)+ Pan2.ar(SinOsc.ar(440,0,Decay.ar(0.1*detect,0.1)),1.0));
}).play(s);
)



//Foote solo- never triggers with threshold over 1.0, threshold under mouse control
(
b=Buffer.alloc(s,2048,1);

SynthDef \fftod
{
var source1, detect;

source1= AudioIn.ar(1);
```

```
detect= PV_HainsworthFoote.ar(FFT(b.bufnum,source1), 0.0, 1.0, MouseX.kr(0.0,1.1), 0.02);

Out.ar(0,Pan2.ar(source1,-1.0)+ Pan2.ar(SinOsc.ar(440,0,Decay.ar(0.1*detect,0.1)),1.0));
}).play(s);
)



//compare to Amplitude UGen
(
b=Buffer.alloc(s,2048,1);

SynthDef \fftod
{
var source1, detect;

source1= AudioIn.ar(1);

detect= (Amplitude.ar(source1)) > (MouseX.kr(0.0,1.1));

Out.ar(0,Pan2.ar(source1,-1.0)+ Pan2.ar(SinOsc.ar(440,0,Decay.ar(0.1*detect,0.1)),1.0));
}).play(s);
)
```

ID: 570

PV_JensenAndersen

FFT feature detector for onset detection based on work described in

#236e25Jensen,K. & Andersen, T. H. (2003). Real-time Beat Estimation Using Feature Extraction. #236e25In Proceedings of the Computer Music Modeling and Retrieval Symposium, Lecture Notes in Computer Science. Springer Verlag.

First order derivatives of the features are taken. Threshold may need to be set low to pick up on changes.

Class Methods

*ar(buffer, propsc=0.25, prophfe=0.25, prophfc=0.25, propsf=0.25, threshold=1.0, waittime=0.04)

buffer- FFT buffer to read from.

propsc- Proportion of spectral centroid feature.

prophfe- Proportion of high frequency energy feature.

prophfc- Proportion of high frequency content feature.

propsf- Proportion of spectral flux feature.

threshold- Threshold level for allowing a detection

waittime- If triggered, minimum wait until a further frame can cause another spot (useful to stop multiple detects on heavy signals)

Examples

```
(
b=Buffer.alloc(s,2048,1);

SynthDef \fftod
```

```
{
var source1, detect;

source1= AudioIn.ar(1);

detect= PV_JensenAndersen.ar(FFT(b.bufnum,source1), threshold:MouseX.kr(0.1,1.0));

Out.ar(0,SinOsc.ar([440,445],0,Decay.ar(0.1*detect,0.1)));
}).play(s);
)
```

ID: 571

PV_LocalMax pass bins which are a local maximum

PV_LocalMax.ar(buffer, threshold)

Passes only bins whose magnitude is above a threshold and above their nearest neighbors.
buffer - fft buffer.
threshold - magnitude threshold.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-localMax", { arg out=0, bufnum=0;
var in, chain;
in = Mix.arFill(3, { LFSaw.ar(exprand(100, 500), 0, 0.1); });
chain = FFT(bufnum, in);
chain = PV_LocalMax(chain, MouseX.kr(0, 50));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-localMax2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_LocalMax(chain, MouseX.kr(0, 100));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 572

PV_MagAbove pass bins above a threshold

PV_MagAbove.ar(buffer, threshold)

Passes only bins whose magnitude is above a threshold.
buffer - fft buffer.
threshold - magnitude threshold.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magAbove", { arg out=0, bufnum=0;
var in, chain;
in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
 //in = WhiteNoise.ar(0.2);
chain = FFT(bufnum, in);
chain = PV_MagAbove(chain, 310);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magAbove2", { arg out=0, bufnum=0;
var in, chain;
in = WhiteNoise.ar(0.2);
chain = FFT(bufnum, in);
chain = PV_MagAbove(chain, MouseX.kr(0, 10));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magAbove3", { arg out=0, bufnum=0, soundBufnum=2;
```

```
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagAbove(chain, MouseX.kr(0, 310));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 573

PV_MagBelow pass bins below a threshold

PV_MagBelow.ar(buffer, threshold)

Passes only bins whose magnitude is below a threshold.
buffer - fft buffer.
threshold - magnitude threshold.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magBelow", { arg out=0, bufnum=0;
var in, chain;
in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
chain = FFT(bufnum, in);
chain = PV_MagBelow(chain, 10);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magBelow2", { arg out=0, bufnum=0;
var in, chain;
in = WhiteNoise.ar(0.2);
chain = FFT(bufnum, in);
chain = PV_MagBelow(chain, MouseX.kr(0, 7));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magBelow3", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
```

```
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagBelow(chain, MouseX.kr(0, 310));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 574

PV_MagClip clip bins to a threshold

PV_MagClip.ar(buffer, threshold)

Clips bin magnitudes to a maximum threshold.
buffer - fft buffer.
threshold - magnitude threshold.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magClip", { arg out=0, bufnum=0;
var in, chain;
in = Mix.arFill(3, { LFSaw.ar(exprand(100, 500), 0, 0.1); });
chain = FFT(bufnum, in);
chain = PV_MagClip(chain, MouseX.kr(0, 15));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magClip2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagClip(chain, MouseX.kr(0, 50));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 575

PV_MagFreeze freeze magnitudes

PV_MagFreeze.ar(buffer, freeze)

Freezes magnitudes at current levels when freeze > 0.
buffer - fft buffer.
freeze - if freeze > 0 then magnitudes are frozen at current levels.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magFreeze", { arg out=0, bufnum=0;
var in, chain;
in = SinOsc.ar(LFNoise1.kr(5.2,250,400));
chain = FFT(bufnum, in);
 // moves in and out of freeze
chain = PV_MagFreeze(chain, SinOsc.kr(0.2) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);

)

(
//trig with MouseY
SynthDef("help-magFreeze2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagFreeze(chain, MouseY.kr > 0.5 );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 576

PV_MagMul multiply magnitudes

PV_MagMul.ar(bufferA, bufferB)

Multiplies magnitudes of two inputs and keeps the phases of the first input.
bufferA - fft buffer A.
bufferB - fft buffer B.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magMul", { arg out=0, bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain;
inA = WhiteNoise.ar(0.2);
inB = LFSaw.ar(100, 0, 0.2);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_MagMul(chainA, chainB);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum ]);
)

(
SynthDef("help-magMul2", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum=2;
var inA, chainA, inB, chainB, chain;
inA = LFSaw.ar([100, 150], 0, 0.2);
inB = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_MagMul(chainA, chainB);
Out.ar(out,  0.1 * IFFT(chain));
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum, d.bufnum]);
```

)

ID: 577

PV_MagNoise multiply magnitudes by noise

PV_MagNoise.ar(buffer)

Magnitudes are multiplied with noise.
buffer - fft buffer.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magNoise", { arg out=0, bufnum=0;
var in, chain;
in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
chain = FFT(bufnum, in);
chain = PV_MagNoise(chain);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
(
SynthDef("help-magNoise2", {  arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagNoise(chain);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 578

PV_MagShift shift and stretch magnitude bin position

PV_MagShift.ar(buffer, stretch, shift)

Shift and stretch the positions of only the magnitude of the bins.
Can be used as a very crude frequency shifter/scaler.
buffer - fft buffer.
stretch - scale bin location by factor.
shift - add an offset to bin position.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magStretch", { arg out=0, bufnum=0;
var in, chain;
in = LFSaw.ar(200, 0, 0.2);
chain = FFT(bufnum, in);
chain = PV_MagShift(chain, MouseX.kr(0.25, 4, \exponential) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magStretch2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagShift(chain, MouseX.kr(0.25, 4, \exponential) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

```
(
SynthDef("help-magShift", { arg out=0, bufnum=0;
var in, chain;
in = LFSaw.ar(200, 0, 0.2);
chain = FFT(bufnum, in);
chain = PV_MagShift(chain, 1, MouseX.kr(-128, 128) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magShift2", {  arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagShift(chain, 1, MouseX.kr(-128, 128) );
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 579

PV_MagSmear average magnitudes across bins

PV_MagSmear.ar(buffer, bins)

Average a bin's magnitude with its neighbors.
buffer - fft buffer.
bins - number of bins to average on each side of bin. As this number rises, so will CPU usage.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-magSmear", { arg out=0, bufnum=0;
var in, chain;
in = LFSaw.ar(500, 0, Decay2.ar(Impulse.ar(2,0,0.2), 0.01, 2));
chain = FFT(bufnum, in);
chain = PV_MagSmear(chain, MouseX.kr(0, 100));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-magSmear2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagSmear(chain, MouseX.kr(0, 100));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 580

PV_MagSquared square magnitudes

PV_MagSquared.ar(buffer)

Squares the magnitudes and renormalizes to previous peak. This makes weak bins weaker.
buffer - fft buffer.

```
s.boot;


(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)


(
SynthDef("help-magSquared", {  arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_MagSquared(chain);
Out.ar(out, 0.003 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 581

PV_Max maximum magnitude

PV_Max.ar(bufferA, bufferB)

Output copies bins with the maximum magnitude of the two inputs.
bufferA - fft buffer A.
bufferB - fft buffer B.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
var exBuf;
 CocoaDialog            arg       //get a second soundfile;
paths.do({ arg p; exBuf = Buffer.read(s, p);

SynthDef("help-max", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum1=2, soundBufnum2 = 3;
var inA, chainA, inB, chainB, chain ;
inA = PlayBuf.ar(1, soundBufnum1, BufRateScale.kr(soundBufnum1), loop:  1);
inB =  PlayBuf.ar(1, soundBufnum2, BufRateScale.kr(soundBufnum2), loop:  1);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_Max(chainA, chainB);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum1, d.bufnum, \soundBufnum2, exBuf.bufnum]);

})
},{
 "cancelled"
});
)
```

ID: 582

PV_Min minimum magnitude

PV_Max.ar(bufferA, bufferB)

Output copies bins with the minimum magnitude of the two inputs.
bufferA - fft buffer A.
bufferB - fft buffer B.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
var exBuf;
 CocoaDialog            arg       //get a second soundfile;
paths.do({ arg p; exBuf = Buffer.read(s, p);

SynthDef("help-min", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum1=2, soundBufnum2 = 3;
var inA, chainA, inB, chainB, chain ;
inA = PlayBuf.ar(1, soundBufnum1, BufRateScale.kr(soundBufnum1), loop:  1);
inB =  PlayBuf.ar(1, soundBufnum2, BufRateScale.kr(soundBufnum2), loop:  1);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_Min(chainA, chainB);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum1, d.bufnum, \soundBufnum2, exBuf.bufnum]);

})
},{
 "cancelled"
});
)
```

ID: 583

PV_Mul complex multiply

PV_Mul.ar(bufferA, bufferB)

Complex Multiplication: (RealA * RealB) - (ImagA * ImagB), (ImagA * RealB) + (RealA * ImagB)
bufferA - fft buffer A.
bufferB - fft buffer B.

```
s = Server.internal.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
)

(
SynthDef("help-mul", { arg out=0, bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain ;
inA = SinOsc.ar(500, 0, 0.5);
inB =  SinOsc.ar(Line.kr(100, 400, 5), 0, 0.5);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_Mul(chainA, chainB);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum]);
s.scope;
)

(
SynthDef("help-mul2", { arg out=0, bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain ;
inA = SinOsc.ar(500, 0, 0.5) * Line.kr;
inB = LFNoise1.ar(20);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_Mul(chainA, chainB);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum]);
```

```
s.scope;
)
```

ID: 584

PV_PhaseShift

PV_PhaseShift.ar(buffer, shift)

buffer - fft buffer.
shift - phase shift in degrees.

```
s.boot;


b = Buffer.alloc(s,2048,1);


(
SynthDef("help-phaseShift", {  arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = SinOsc.ar(500);
chain = FFT(bufnum, in);
chain = PV_PhaseShift(chain, LFNoise2.kr(1, 180, 180));
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 585

PV_PhaseShift270 shift phase by 270 degrees

PV_PhaseShift270.ar(buffer)

Shift phase of all bins by 270 degrees.
buffer - fft buffer

```
Server.internal.boot;


b = Buffer.alloc(Server.internal,2048,1);
c = Buffer.alloc(Server.internal,2048,1);


(
{  arg out=0, bufnum=0;
var in, fft, fft2, shifted;
in = SinOsc.ar(500, 0, 0.4);
fft = FFT(b.bufnum, in);
fft2 = FFT(c.bufnum, in);
shifted = PV_PhaseShift270(fft);
Out.ar(0,  [IFFT(fft2), IFFT(shifted)]);
}.scope
)
```

ID: 586

PV_PhaseShift90 shift phase by 90 degrees

PV_PhaseShift90.ar(buffer)

Shift phase of all bins by 90 degrees.
buffer - fft buffer

```
Server.internal.boot;


b = Buffer.alloc(Server.internal,2048,1);
c = Buffer.alloc(Server.internal,2048,1);


(
{  arg out=0, bufnum=0;
var in, fft, fft2, shifted;
in = SinOsc.ar(500, 0, 0.4);
fft = FFT(b.bufnum, in);
fft2 = FFT(c.bufnum, in);
shifted = PV_PhaseShift90(fft);
Out.ar(0,  [IFFT(fft2),IFFT(shifted)]);
}.scope
)
```

ID: 587

PV_RandComb pass random bins

PV_RandComb.ar(buffer, wipe, trig)

Randomly clear bins.
buffer - fft buffer.
wipe - clears bins from input in a random order as wipe goes from 0 to 1.
trig - a trigger selects a new random ordering.

```
s.boot;

(
b = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
SynthDef("help-randcomb", { arg out=0, bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.8)}.dup;
chain = FFT(bufnum, in);
chain = PV_RandComb(chain, 0.95, Impulse.kr(0.4));
Out.ar(out, IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
//trig with MouseY
SynthDef("help-randcomb2", { arg out=0, bufnum=0, soundBufnum=2;
var in, chain;
in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop:  1);
chain = FFT(bufnum, in);
chain = PV_RandComb(chain, MouseY.kr, Impulse.kr(0.4));
Out.ar(out, IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum, \soundBufnum, c.bufnum]);
)
```

ID: 588

PV_RandWipe crossfade in random bin order

PV_RandWipe.ar(bufferA, bufferB, wipe, trig)

Cross fades between two sounds by copying bins in a random order.
bufferA - fft buffer A.
bufferB - fft buffer B.
wipe - copies bins from bufferB in a random order as wipe goes from 0 to 1.
trig - a trigger selects a new random ordering.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
)

(
//trig with MouseY
SynthDef("help-randWipe", { arg out=0, bufnumA=0, bufnumB=1;
var inA, chainA, inB, chainB, chain;
inA = Mix.arFill(6, { LFSaw.ar(exprand(400, 1000), 0, 0.1) }); inB = Mix.arFill(6, { LFPulse.ar(exprand(80,
400), 0, 0.2, SinOsc.kr(8.0.rand, 0, 0.2).max(0)) });
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_RandWipe(chainA, chainB, MouseX.kr, MouseY.kr > 0.5);
Out.ar(out, 0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum ]);
)
```

ID: 589

PV_RectComb make gaps in spectrum

PV_RectComb.ar(buffer, numTeeth, phase, width)

Makes a series of gaps in a spectrum.
buffer - fft buffer.
numTeeth - number of teeth in the comb.
phase - starting phase of comb pulse.
width - pulse width of comb.

```
s.boot;

b = Buffer.alloc(s,2048,1);

(
SynthDef("help-rectcomb", { arg out=0, bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.2)}.dup;
chain = FFT(bufnum, in);
chain = PV_RectComb(chain, 8, LFTri.kr(0.097, 0, 0.4, 0.5),
LFTri.kr(0.24, 0, -0.5, 0.5));
Out.ar(out, IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
SynthDef("help-rectcomb2", { arg out=0, bufnum=0;
var in, chain;
in = {WhiteNoise.ar(0.2)}.dup;
chain = FFT(bufnum, in);
chain = PV_RectComb(chain,  MouseX.kr(0, 32), MouseY.kr, 0.2);
Out.ar(out, IFFT(chain).dup);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
```

ID: 590

PV_RectComb2 make gaps in spectrum

PV_RectComb2.ar(bufferA, bufferB, numTeeth, phase, width)

Alternates blocks of bins between the two inputs.
bufferA - fft buffer A.
bufferB - fft buffer B.
numTeeth - number of teeth in the comb.
phase - starting phase of comb pulse.
width - pulse width of comb.

```
s.boot;
(
b = Buffer.alloc(s,2048,1);
c = Buffer.alloc(s,2048,1);
    Buffer        "sounds/a11wlk01.wav"
)

(
var exBuf;
 CocoaDialog            arg       //get a second soundfile;
paths.do({ arg p; exBuf = Buffer.read(s, p);

SynthDef("help-max", { arg out=0, bufnumA=0, bufnumB=1, soundBufnum1=2, soundBufnum2 = 3;
var inA, chainA, inB, chainB, chain ;
inA = PlayBuf.ar(1, soundBufnum1, BufRateScale.kr(soundBufnum1), loop:  1);
inB =  PlayBuf.ar(1, soundBufnum2, BufRateScale.kr(soundBufnum2), loop:  1);
chainA = FFT(bufnumA, inA);
chainB = FFT(bufnumB, inB);
chain = PV_RectComb2(chainA, chainB,  MouseX.kr(0, 32), MouseY.kr, 0.3);
Out.ar(out,  0.5 * IFFT(chain).dup);
}).play(s,[\out, 0, \bufnumA, b.bufnum, \bufnumB, c.bufnum, \soundBufnum1, d.bufnum, \soundBufnum2, exBuf.bufnum]);

})
},{
 "cancelled"
});
```

)

25.8 Filters

ID: 591

AmpComp basic psychoacoustic amplitude compensation

superclass: UGen

`implements the (optimized) formula:` `compensationFactor = (root / freq) ** exp`

Higher frequencies are normally perceived as louder, which AmpComp compensates.

***ar(freq, root, exp)**
***kr(freq, root, exp)**
***ir(freq, root, exp)**

freq input frequency value. For freq == root, the output is 1.0.
root root freq relative to which the curve is calculated (usually lowest freq)
default value: C (60.midicps)
exp exponent: how steep the curve decreases for increasing freq (see plots below)
default value 0.3333

see also [AmpCompA]

```
// compare a sine without compensation

{ SinOsc.ar(MouseX.kr(300, 15000, 1)) * 0.1 }.play;

// with one that uses amplitude compensation
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
SinOsc.ar(freq) * 0.1 * AmpComp.kr(freq, 300)
}.play;
)
```

```
// different sounds cause quite different loudness perception,
// and the desired musical behavior can vary, so the exponent can be tuned:
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
Pulse.ar(freq) * 0.1 * AmpComp.kr(freq, 300, 1.3)
}.play;
)

// the curves:

// exp = 0.3333
(200,210..10000).collect {| freq|  (200/freq) ** 0.3333 }.plot;

// nearly linear for semitone steps:

(48..72).midicps.collect {| freq|  (48.midicps/freq) ** 0.3333 }.plot;
{ AmpComp.ar(Line.ar(48, 72, 1).midicps, 48.midicps) }.plot(1.0);

// exp = 1.2
(200,210..10000).collect {| freq|  (200/freq) ** 1.2 }.plot;
(48..72).midicps.collect {| freq|  (200/freq) ** 1.2 }.plot;
{ AmpComp.ar(Line.ar(48, 72, 1).midicps, 48.midicps, 1.2) }.plot(1.0);


// amplitude compensation in frequency modulation
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
freq = freq * SinOsc.ar(MouseY.kr(3, 200, 1), 0, 0.5, 1);
SinOsc.ar(freq) * 0.1 * AmpComp.ar(freq, 300)
}.play;
)

// without amplitude compensation
(
{
```

```
var freq;
freq = MouseX.kr(300, 15000, 1);
freq = freq * SinOsc.ar(MouseY.kr(3, 200, 1), 0, 0.5, 1);
SinOsc.ar(freq) * 0.1
}.play;
)

// in granular synthesis:
(
SynthDef "pgrain"
{ arg out = 0, sustain=0.01, amp=0.5, pan = 0;
var freq = MouseX.kr(300, 7000, 1);
var window = Env.sine(sustain, amp * AmpComp.ir(freq));
Out.ar(out,
Pan2.ar(
SinOsc.ar(freq),
pan
) * EnvGen.ar(window, doneAction:2)
)
}
).send(s);
)

// send grains
(
fork {
loop {
s.sendBundle(0.1, [\s_new, \pgrain, -1,1,1]);
0.02.wait;
};
}
)


// try different synth defs:


// without AmpComp:

(
```

```
SynthDef "pgrain"
{ arg out = 0, sustain=0.01, amp=0.5, pan = 0;
var freq = MouseX.kr(300, 7000, 1);
var window = Env.sine(sustain, amp);
Out.ar(out,
Pan2.ar(
SinOsc.ar(freq),
pan
) * EnvGen.ar(window, doneAction:2)
)
}
).send(s);
)

// with AmpCompA
(
SynthDef "pgrain"
{ arg out = 0, sustain=0.01, amp=0.5, pan = 0;
var freq = MouseX.kr(300, 7000, 1);
var window = Env.sine(sustain, amp * AmpCompA.ir(freq));
Out.ar(out,
Pan2.ar(
SinOsc.ar(freq),
pan
) * EnvGen.ar(window, doneAction:2)
)
}
).send(s);
)
```

ID: 592

AmpCompA ANSI A-weighting curve
basic psychoacoustic amplitude compensation

superclass: UGen

Higher frequencies are normally perceived as louder, which AmpCompA compensates.
Following the measurings by Fletcher and Munson, the ANSI standard describes
a function for loudness vs. frequency.
Note that this curve is only valid for standardized amplitude. [1]

For a simpler but more flexible curve, see [AmpComp]

***ar(freq, root, minAmp, rootAmp)**
***kr(freq, root, minAmp, rootAmp)**
***ir(freq, root, minAmp, rootAmp)**

freq input frequency value. For freq == root, the output is rootAmp. (default freq 0 Hz)
root root freq relative to which the curve is calculated (usually lowest freq) (default 0 Hz)
default value: C (60.midicps)

minAmp amplitude at the minimum point of the curve (around 2512 Hz) (default -10dB)
rootAmp amplitude at the root frequency. (default 1)

apart from *freq*, the values are not modulatable

```
// compare a sine without compensation

{ SinOsc.ar(MouseX.kr(300, 15000, 1)) * 0.1 }.play;
```

```
// with one that uses amplitude compensation
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
SinOsc.ar(freq) * 0.3 * AmpCompA.kr(freq)
}.play;
)


// adjust the minimum and root amp
// (in this way one can flatten out the curve for higher amplitudes)

(
{
var freq;
freq = MouseX.kr(300, 18000, 1);
Formant.ar(300, freq, 20, 0.1) * AmpCompA.kr(freq, 300, 0.6, 0.3)
}.play;
)

// the curve:

{ AmpCompA.ar(Line.ar(48, 120, 1).midicps, 48.midicps) }.plot(1.0);

// freqs:

{ AmpCompA.ar(Line.ar(0, 20000, 1)) }.plot(1.0);

// compare with AmpComp (exponential decay)

{ AmpComp.ar(Line.ar(48, 120, 1).midicps, 48.midicps) }.plot(1.0);

// freqs:

{ AmpComp.ar(Line.ar(40, 20000, 1), 40) }.plot(1.0);



// amplitude compensation in frequency modulation (using Fletscher-Munson curve)
```

```
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
freq = freq * SinOsc.ar(MouseY.kr(3, 200, 1), 0, 0.5, 1);
SinOsc.ar(freq) * 0.1 * AmpCompA.ar(freq, 300)
}.play;
)

// amplitude compensation in frequency modulation (using AmpComp exponential decay)
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
freq = freq * SinOsc.ar(MouseY.kr(3, 200, 1), 0, 0.5, 1);
SinOsc.ar(freq) * 0.1 * AmpComp.ar(freq, 300)
}.play;
)


// without amplitude compensation
(
{
var freq;
freq = MouseX.kr(300, 15000, 1);
freq = freq * SinOsc.ar(MouseY.kr(3, 200, 1), 0, 0.5, 1);
SinOsc.ar(freq) * 0.1
}.play;
)
```

[1] Function freq -> dB,
derived from http://www.beis.de/Elektronik/AudioMeasure/WeightingFilters.html
and modified to map freq -> amp

```
(
var k =  3.5041384e16;
```

```
var c1 = 424.31867740601;
var c2 = 11589.093052022;
var c3 = 544440.67046057;
var c4 = 148698928.24309;
f = {| f|
var r = squared(f);
var m1 = pow(r,4);
var n1 = squared(c1 + r);
var n2 = c2 + r;
var n3 = c3 + r;
var n4 = squared(c4 + r);
var level = k * m1 / (n1 * n2 * n3 * n4);
sqrt(level)
};
)
```

ID: 593

BPF 2nd order Butterworth bandpass filter

BPF.ar(in, freq, rq, mul, add)

A second order low pass filter.
in - input signal to be processed
freq - cutoff frequency in Hertz.
rq - the reciprocal of Q. bandwidth / cutoffFreq.

```
{ BPF.ar(Saw.ar(200,0.5), FSinOsc.kr(XLine.kr(0.7,300,20),0,3600,4000), 0.3) }.play;


{ BPF.ar(Saw.ar(200,0.5), MouseX.kr(100, 10000, 1), 0.3) }.play;


 // BPF on control signals:
(
{  var vib = BPF.kr(PinkNoise.kr, MouseX.kr(1, 100, 1), 0.3) * 10;
SinOsc.ar(vib * 200 + 600) * 0.2 }.play;
)
```

ID: 594

BPZ2 two zero fixed midpass

BPZ2.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.5 * (in(i) - in(i-2))

This filter cuts out 0 Hz and the Nyquist frequency.
Compare:

```
{ WhiteNoise.ar(0.25) }.play;

{ BPZ2.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 595

BRF 2nd order Butterworth band reject filter

BRF.ar(in, freq, rq, mul, add)

A second order low pass filter.
in - input signal to be processed
freq - cutoff frequency in Hertz.
rq - the reciprocal of Q. bandwidth / cutoffFreq.

```
{ BRF.ar(Saw.ar(200,0.1), FSinOsc.kr(XLine.kr(0.7,300,20),0,3800,4000), 0.3) }.play;


{ BRF.ar(Saw.ar(200,0.5), MouseX.kr(100, 10000, 1), 0.3) }.play;



// BRF on control signals:
(
   var      BRF    SinOsc.kr([1, 3, 10], 0, [1, 0.5, 0.25]).sum
SinOsc.ar(vib * 200 + 600) * 0.2 }.play;
)
```

ID: 596

BRZ2 two zero fixed midcut

BRZ2.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.5 * (in(i) + in(i-2))

This filter cuts out frequencies around 1/2 of the Nyquist frequency.
Compare:

```
{ WhiteNoise.ar(0.25) }.play;

{ BRZ2.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 597

Clip clip a signal outside given thresholds

Clip.ar(in, lo, hi)
Clip.kr(in, lo, hi)

This differs from the **BinaryOpUGen clip2** in that it allows one to set both low and high thresholds.
in - signal to be clipped
lo - low threshold of clipping
hi - high threshold of clipping

```
Server.internal.boot;

{ Clip.ar(SinOsc.ar(440, 0, 0.2), -0.07, 0.07) }.scope;
```

ID: 598

DynKlank bank of resonators

DynKlank.ar(specificationsArrayRef, input, freqscale, freqoffset, decayscale)

DynKlank is a bank of frequency resonators which can be used to simulate the resonant modes of an object. Each mode is given a ring time, which is the time for the mode to decay by 60 dB.

Unlike Klank, the frequencies in DynKlank can be changed after it has been started.

specificationsArrayRef - a Ref to an Array of three Arrays :
frequencies - an Array of filter frequencies.
amplitudes - an Array of filter amplitudes, or nil. If nil, then amplitudes default to 1.0
ring times - an Array of 60 dB decay times for the filters.
All subarrays, if not nil, should have the same length.
input - the excitation input to the resonant filter bank.
freqscale - a scale factor multiplied by all frequencies at initialization time.
freqoffset - an offset added to all frequencies at initialization time.
decayscale - a scale factor multiplied by all ring times at initialization time.

```
s.boot;

{ DynKlank.ar(‘[[800, 1071, 1153, 1723], nil, [1, 1, 1, 1]], Impulse.ar(2, 0, 0.1)) }.play;

{ DynKlank.ar(‘[[800, 1071, 1353, 1723], nil, [1, 1, 1, 1]], Dust.ar(8, 0.1)) }.play;

{ DynKlank.ar(‘[[800, 1071, 1353, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar(0.007)) }.play;

{ DynKlank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar([0.007,0.007])) }.play;


(
// change freqs and ringtimes with mouse
{ var freqs, ringtimes;
freqs = [800, 1071, 1153, 1723] * MouseX.kr(0.5, 2, 1);
ringtimes = [1, 1, 1, 1] * MouseY.kr(0.1, 10, 1);
DynKlank.ar(‘[freqs, nil, ringtimes ], Impulse.ar(2, 0, 0.1))
}.play;
```

```
)

(
// set them from outside later:
SynthDef 'help-dynKlank'
var freqs, ringtimes, signal;
freqs = Control.names([\freqs]).kr([800, 1071, 1153, 1723]);
ringtimes = Control.names([\ringtimes]).kr([1, 1, 1, 1]);
signal = DynKlank.ar(`[freqs, nil, ringtimes ], Impulse.ar(2, 0, 0.1));
Out.ar(0, signal);
}).load(s);
)

    Synth 'help-dynKlank'

a.setn(\freqs, Array.rand(4, 500, 2000));
a.setn(\ringtimes, Array.rand(4, 0.2, 4) );

  // create multichannel controls directly with literal arrays:
SynthDef 'help-dynKlank'   | freqs (#[100, 200, 300, 400]),
 amps (#[1, 0.3, 0.2, 0.05]),
 rings (#[1, 1, 1, 2])|

Out.ar(0, DynKlank.ar(`[freqs, amps, rings], WhiteNoise.ar * 0.001))
}).send(s)
)

    Synth 'help-dynKlank'

a.setn(\freqs, Array.rand(4, 500, 2000));
a.setn(\amps, Array.exprand(4, 0.01, 1));
```

ID: 599

Fold fold a signal outside given thresholds

Fold.ar(in, lo, hi)
Fold.kr(in, lo, hi)

This differs from the **BinaryOpUGen fold2** in that it allows one to set both low and high thresholds.
in - signal to be folded
lo - low threshold of folding. Sample values < lo will be folded.
hi - high threshold of folding. Sample values > hi will be folded.

```
Server.internal.boot;

{ Fold.ar(SinOsc.ar(440, 0, 0.2), -0.1, 0.1) }.scope;
```

ID: 600

Formlet FOF-like filter

Formlet.ar(in, freq, attacktime, decaytime, mul, add)

This is a resonant filter whose impulse response is like that of a sine wave with a Decay2 envelope over it.
It is possible to control the attacktime and decaytime.
Formlet is equivalent to:

Ringz(in, freq, decaytime) - Ringz(in, freq, attacktime)

Note that if attacktime == decaytime then the signal cancels out and if attacktime > decaytime
then the impulse response is inverted.

The great advantage to this filter over FOF is that there is no limit to the number of overlapping
grains since the grain is just the impulse response of the filter.

in - input signal to be processed
freq - resonant frequency in Hertz
attackTime - 60 dB attack time in seconds.
decayTime - 60 dB decay time in seconds.

```
{ Formlet.ar(Impulse.ar(20, 0.5), 1000, 0.01, 0.1) }.play;

{ Formlet.ar(Blip.ar(XLine.kr(10,400,8), 1000, 0.1), 1000, 0.01, 0.1) }.play;

(
// modulating formant frequency
{
var in;
in = Blip.ar(SinOsc.kr(5,0,20,300), 1000, 0.1);
Formlet.ar(in, XLine.kr(1500,700,8), 0.005, 0.04);
}.play;
```

```
)

(
// mouse control of frequency and decay time.
{
var in;
in = Blip.ar(SinOsc.kr(5,0,20,300), 1000, 0.1);
Formlet.ar(in,
MouseY.kr(700,2000,1),
0.005, MouseX.kr(0.01,0.2,1));
}.play;
)

(
// mouse control of frequency and decay time.
{
var freq;
freq = Formlet.kr(
Dust.kr(10 !  2),
MouseY.kr(7,200,1),
0.005, MouseX.kr(0.1,2,1)
);
SinOsc.ar(freq * 200 + [500, 600] - 100) * 0.2
}.play;
)
```

ID: 601

FOS first order filter section

FOS.ar(in, a0, a1, b1, mul, add)

A standard first order filter section. Filter coefficients are given directly rather than calculated for you.
Formula is equivalent to:

out(i) = (a0 * in(i)) + (a1 * in(i-1)) + (b1 * out(i-1))

Examples:

```
(
// same as OnePole
{ var x;
x = LFTri.ar(0.4, 0, 0.99);
FOS.ar(LFSaw.ar(200, 0, 0.2), 1 - x.abs, 0.0, x)
}.play;
)

(
// same as OneZero
{ var x;
x = LFTri.ar(0.4, 0, 0.99);
FOS.ar(LFSaw.ar(200, 0, 0.2), 1 - x.abs, x, 0.0)
}.play;
)

(
// same as OnePole, kr
{ var x, ctl;
x = LFTri.kr(0.2, 0, 0.99);
ctl = FOS.kr(LFSaw.kr(8, 0, 0.2), 1 - x.abs, 0.0, x);
LFTri.ar(ctl * 200 + 500);
}.play;
)
```

ID: 602

FreqShift Frequency Shifter

FreqShift.ar(input, shift, phase, mul, add)

FreqShift implements single sideband amplitude modulation, also known as frequency shifting, but not to be confused with pitch shifting. Frequency shifting moves all the components of a signal by a fixed amount but does not preserve the original harmonic relationships.

input - audio input
shift - amount of shift in cycles per second
phase - phase of the frequency shift (0 - 2pi)

Examples

```
// shifting a 100Hz tone by 1 Hz rising to 500Hz
{FreqShift.ar(SinOsc.ar(100),XLine.kr(1,500,5),0,[0.5,0.5])}.play(s);

// shifting a complex tone by 1 Hz rising to 500Hz
{FreqShift.ar(Klang.ar(`[[101,303,606,808]]),XLine.kr(1,500,10),0,[0.25,0.25])}.play(s);

// modulating shift and phase
{FreqShift.ar(SinOsc.ar(10),LFNoise2.ar(0.3,1500),SinOsc.ar(500).range(0,2pi),[0.5,0.5])}.play(s);

// the ubiquitous houston example
(
    Buffer        "sounds/a11wlk01.wav"
{FreqShift.ar(PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum),loop:1),LFNoise0.kr(0.45,1000),0,[1,1])}.play(s);

)

// shifting bandpassed noise
{FreqShift.ar(BPF.ar(WhiteNoise.ar(1),1000,0.001),LFNoise0.kr(5.5,1000),0,[32,32])}.play(s);
```

More Examples (send a SynthDef, run the routine then send a different SynthDef)

```
 // simple detune &amp;amp; pitchmod via FreqShift
SynthDef("frqShift1",{arg frq,detune=1.5;
var e1,left,right;
e1 = EnvGen.ar(Env.new([0,1,0],[1,2.3]),1,doneAction:2);
left = SinOsc.ar(frq,0,e1); // original tone
left = left + FreqShift.ar(left,frq*detune); // shift and add back to original
right = FreqShift.ar(left,SinOsc.kr(3.23,0,5));
Out.ar(0, [left,right] * 0.25);
}).send(s);
)

 // the routine
    Routine
var table,pitch;
table = [0,2,4,5,7,9,11,12];
inf.do{
pitch = (48+(12*2.rand) + table.choose).midicps;
s.sendMsg("s_new","frqShift1",-1,1,1,"frq",pitch);
3.wait;
};
};
).play;
)

 // shift pulse wave in opposite directions
SynthDef("frqShift1",{arg frq,detune=0.15;
var e1,snd,left,right;
e1 = EnvGen.ar(Env.new([0,1,0],[0.02,3.2]),1,doneAction:2);
snd = Pulse.ar(frq,SinOsc.kr(2.3).range(0.2,0.8),e1); // original tone
left = FreqShift.ar(snd,XLine.kr(-0.1,-200,2)); // shift and add back to original
right = FreqShift.ar(snd,XLine.kr(0.1,200,2));
Out.ar(0, [left,right] * 0.25);
}).send(s)
)

 // FreqShift >> feedback >>> FreqShift
SynthDef "frqShift1"  arg
var e1,snd,snd2,in;
in = FreqShift.ar(InFeedback.ar(0,1)*3.2,XLine.ar(0.01,frq*1.5,1)); // shift the feedback
e1 = Env.new([0,1,0],[0.02,2.98]);
```

```
snd = SinOsc.ar(frq,0,EnvGen.ar(e1,1,doneAction:2));
snd2 = FreqShift.ar(snd+in,SinOsc.ar(4.24,0.5,3),0,0.5); // subtle modulating shift
OffsetOut.ar([0,1], Limiter.ar(snd2+snd * 0.5,1,0.005));
}).send(s);
)


 // sslloowww columbia tuned shift detune
        // stop old routine
Buffer.read(s,"sounds/a11wlk01.wav", bufnum:99);

SynthDef("frqShift1",{arg frq, bufnum;
var e1,snd,left,right;
e1 = Env.new([0,1,0],[3,1],-4);
snd = PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum) * 0.01, loop:1);
left = FreqShift.ar(snd,frq*2,0,EnvGen.ar(e1,1,doneAction:2)); // subtle shift of the output
right = FreqShift.ar(snd,frq*3,0,EnvGen.ar(e1,1,doneAction:2));
Out.ar(0, [left,right] * 3);
}).send(s);
 // the routine
    Routine
var table,pitch;
table = [0,2,4,5,7,9,11,12];
inf.do{
pitch = (48+(12*2.rand) + table.choose).midicps;
s.sendMsg("s_new","frqShift1",-1,1,1, "frq", pitch, "bufnum", 99);
3.wait;
};
};
).play;
)


)
```

ID: 603

HPF 2nd order Butterworth highpass filter

HPF.ar(in, freq, mul, add)

A second order high pass filter.
in - input signal to be processed
freq - cutoff frequency.

```
{ HPF.ar(Saw.ar(200,0.1), FSinOsc.kr(XLine.kr(0.7,300,20),0,3600,4000), 5) }.play;

(
{  var ctl = HPF.kr(LFSaw.kr(5), SinOsc.kr(XLine.kr(0.07,30,20), 0, 35, 40)) ;
SinOsc.ar(ctl * 200 + 500);
}.play;
)

(
{  var ctl = HPF.kr(LFSaw.kr(5,0.1), MouseX.kr(2, 200, 1));
SinOsc.ar(ctl * 200 + 400) * 0.1;
}.play;
)
```

ID: 604

HPZ1 two point difference filter

HPZ1.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.5 * (in(i) - in(i-1))

which is a two point differentiator.
Compare:

```
{ WhiteNoise.ar(0.25) }.play;
```

```
{ HPZ1.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 605

HPZ2 two zero fixed highpass

HPZ2.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.25 * (in(i) - (2*in(i-1)) + in(i-2))

Compare:

```
{ WhiteNoise.ar(0.25) }.play;

{ HPZ2.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 606

Klank bank of resonators

Klank.ar(specificationsArrayRef, input, freqscale, freqoffset, decayscale)

Klank is a bank of fixed frequency resonators which can be used to simulate the resonant modes of an object. Each mode is given a ring time, which is the time for the mode to decay by 60 dB.

specificationsArrayRef - a Ref to an Array of three Arrays :
frequencies - an Array of filter frequencies.
amplitudes - an Array of filter amplitudes, or nil. If nil, then amplitudes default to 1.0
ring times - an Array of 60 dB decay times for the filters.
All subarrays, if not nil, should have the same length.
input - the excitation input to the resonant filter bank.
freqscale - a scale factor multiplied by all frequencies at initialization time.
freqoffset - an offset added to all frequencies at initialization time.
decayscale - a scale factor multiplied by all ring times at initialization time.

```
{ Klank.ar(‘[[800, 1071, 1153, 1723], nil, [1, 1, 1, 1]], Impulse.ar(2, 0, 0.1)) }.play;

{ Klank.ar(‘[[800, 1071, 1353, 1723], nil, [1, 1, 1, 1]], Dust.ar(8, 0.1)) }.play;

{ Klank.ar(‘[[800, 1071, 1353, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar(0.007)) }.play;

{ Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar([0.007,0.007])) }.play;

(
play({
Klank.ar(‘[
Array.rand(12, 800.0, 4000.0),  // frequencies
  nil          // amplitudes (default to 1.0)
Array.rand(12, 0.1, 2)    // ring times
], Decay.ar(Impulse.ar(4), 0.03, ClipNoise.ar(0.01)))
})
)
```

```
// a synth def that has 4 partials
(
s = Server.local;
SynthDef("help-Control", { arg out=0,i_freq;
var klank, n, harm, amp, ring;
n = 9;

 // harmonics
harm = Control.names([\harm]).ir(Array.series(4,1,1).postln);
 // amplitudes
amp = Control.names([\amp]).ir(Array.fill(4,0.05));
 // ring times
ring = Control.names([\ring]).ir(Array.fill(4,1));

klank = Klank.ar(`[harm,amp,ring], {ClipNoise.ar(0.003)}.dup, i_freq);

Out.ar(out, klank);
}).send(s);
)

// nothing special yet, just using the default set of harmonics.
a = Synth("help-Control",[\i_freq, 300]);
b = Synth("help-Control",[\i_freq, 400]);
c = Synth("help-Control",[\i_freq, 533.33]);
d = Synth("help-Control",[\i_freq, 711.11]);

a.free;
b.free;
c.free;
d.free;

// in order to set the harmonics amps and ring times at
// initialization time we need to use an OSC bundle.
(
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7] // set odd harmonics
);
)
```

```
s.sendMsg("/n_free", 2000);

(
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
 // set geometric series harmonics
["/n_setn", 2000, "harm", 4] ++ Array.geom(4,1,1.61)
);
)

s.sendMsg("/n_free", 2000);

(
// set harmonics, ring times and amplitudes
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7], // set odd harmonics
["/n_setn", 2000, "ring", 4] ++ Array.fill(4,0.1), // set shorter ring time
["/n_setn", 2000, "amp", 4] ++ Array.fill(4,0.2) // set louder amps
);
)

s.sendMsg(\n_trace, 2000)

(
// same effect as above, but packed into one n_setn command
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7,
"ring", 4] ++ Array.fill(4,0.1)
++ ["amp", 4] ++ Array.fill(4,0.2)
);
)
```

```
//(
//play({
// OverlapTexture.ar({
//  Pan2.ar(
//   Klank.ar(`[
//    Array.rand(12, 200.0, 4000.0),  // frequencies
//    nil,         // amplitudes (default to 1.0)
//    Array.rand(12, 0.1, 2)    // ring times
//    ], Decay.ar(Impulse.ar(0.8+1.4.rand), 0.03, ClipNoise.ar(0.01))),
//   1.0.rand2
//  )
// }, 8, 3, 4, 2)
//}))
//
//
//(
//// frequency and decay scaling
//var env, specs, mode;
//env = Env.new([1,1,0],[0.4,0.01]);   // cutoff envelope
//specs = `[
//   Array.series(12, 1, 1),  // partials
//   nil,        // amplitudes (default to 1.0)
//   Array.rand(12, 0.1, 2)   // ring times
//  ];
//mode = #[0, 2, 4, 5, 7, 9, 11, 12, 14, 16, 17, 19, 21, 23, 24];
//play({
// Spawn.ar({ arg spawn, i;
//  Pan2.ar(
//   Klank.ar(specs,
//    Decay.ar(Impulse.ar(0), 0.03, ClipNoise.ar(0.01)),
//    (72 + (mode @@ i)).midicps,  // scale to this frequency
//    0,
//    MouseX.kr(0.2, 3),    // scale decay times
//    EnvGen.kr(env)
//   ),
//   1.0.rand2
//  )
// }, 2, 0.2)
//}))
//
```

```
//
//(
//play({
// OverlapTexture.ar({
//  Pan2.ar(
//   Klank.ar(`[
//    Array.linrand(12, 80.0, 6000.0), // frequencies
//    nil,         // amplitudes (default to 1.0)
//    Array.rand(12, 0.1, 3)    // ring times
//    ], BrownNoise.ar(0.0012)),
//   1.0.rand2
//  )
// }, 6, 4, 5, 2)
//}))
//
//
//(
//var e;
//e = Env.new([1,1,0],[18, 3]);
//play({
// Spawn.ar({
//  Pan2.ar(
//   EnvGen.kr(e) *
//   Klank.ar(`[
//    Array.linrand(12, 80.0, 12000.0),  // frequencies
//    nil,         // amplitudes (default to 1.0)
//    Array.rand(12, 3, 10)    // ring times
//    ], Decay.ar(Impulse.ar(0.2+0.4.rand), 0.8, ClipNoise.ar(0.001))),
//   1.0.rand2
//  )
// }, 2, 3)
//}))
//
```

ID: 607

Lag exponential lag

Lag.ar(in, lagTime, mul, add)
Lag.kr(in, lagTime, mul, add)

This is essentially the same as **OnePole** except that instead of supplying the coefficient directly, it is calculated from a 60 dB lag time. This is the time required for the filter to converge to within 0.01 % of a value. This is useful for smoothing out control signals.

in - input signal
lagTime - 60 dB lag time in seconds.

```
(
// used to lag pitch
{
 SinOsc      // sine wave
  Lag       // lag the modulator
LFPulse.kr(4, 0, 0.5, 50, 400), // frequency modulator
Line.kr(0, 1, 15)    // modulate lag time
),
  0, // phase
  0.3 // sine amplitude
)
}.play
)
```

ID: 608

Lag2 exponential lag

Lag2.ar(in, lagTime, mul, add)
Lag2.kr(in, lagTime, mul, add)

Lag2 is equivalent to Lag.kr(Lag.kr(in, time), time), thus resulting in a smoother transition. This saves on CPU as you only have to calculate the decay factor once instead of twice. See **Lag** for more details.

in - input signal
lagTime - 60 dB lag time in seconds.

```
(
// used to lag pitch
{
 SinOsc      // sine wave
  Lag2       // lag the modulator
LFPulse.kr(4, 0, 0.5, 50, 400), // frequency modulator
Line.kr(0, 1, 15)    // modulate lag time
),
  0, // phase
  0.3 // sine amplitude
)
}.play
)
```

ID: 609

Lag3 exponential lag

Lag3.ar(in, lagTime, mul, add)
Lag3.kr(in, lagTime, mul, add)

Lag3 is equivalent to Lag.kr(Lag.kr(Lag.kr(in, time), time), time), thus resulting in a smoother transition. This saves on CPU as you only have to calculate the decay factor once instead of three times. See **Lag** for more details.

in - input signal
lagTime - 60 dB lag time in seconds.

```
(
// used to lag pitch
{
 SinOsc      // sine wave
  Lag3       // lag the modulator
LFPulse.kr(4, 0, 0.5, 50, 400), // frequency modulator
Line.kr(0, 1, 15)    // modulate lag time
),
  0, // phase
  0.3 // sine amplitude
)
}.play
)
```

ID: 610

LeakDC remove DC

LeakDC.ar(in, coef, mul, add)

This filter removes a DC offset from a signal.
in - input signal.
coef - leak coefficient.

```
(
// this is really better with scope()
play({
var a;
a = LFPulse.ar(800, 0.5, 0.5, 0.5);
[a, LeakDC.ar(a, 0.995)]
}))
```

ID: 611

Limiter peak limiter

Limiter.ar(input, level, lookAheadTime)

Limits the input amplitude to the given level. Limiter will not overshoot like Compander will, but it needs to look ahead in the audio. Thus there is a delay equal to twice the lookAheadTime.
Limiter, unlike Compander, is completely transparent for an in range signal.

input - the signal to be processed.
level - the peak output amplitude level to which to normalize the input.
lookAheadTime - the buffer delay time. Shorter times will produce smaller delays and quicker transient response times, but may introduce amplitude modulation artifacts.

```
(
// example signal to process
Synth.play({
var z;
z = Decay2.ar(
Impulse.ar(8, LFSaw.kr(0.25, -0.6, 0.7)),
0.001, 0.3, FSinOsc.ar(500));
}, 0.8)
)

(
Synth.play({
var z;
z = Decay2.ar(
Impulse.ar(8, LFSaw.kr(0.25, -0.6, 0.7)),
0.001, 0.3, FSinOsc.ar(500));
[z, Limiter.ar(z, 0.4, 0.01)]
}, 0.5)
)
```

ID: 612

LinExp convert a linear range to an exponential range

LinExp.ar(in, srclo, srchi, dstlo, dsthi)
LinExp.kr(in, srclo, srchi, dstlo, dsthi)

Converts a linear range of values to an exponential range of values.
The dstlo and dsthi arguments must be nonzero and have the same sign.
in - input to convert.
srclo - lower limit of input range.
srchi - upper limit of input range.
dstlo - lower limit of output range.
dsthi - upper limit of output range.

ID: 613

LinLin map a linear range to another linear range

LinLin.ar(in, srclo, srchi, dstlo, dsthi)
LinLin.kr(in, srclo, srchi, dstlo, dsthi)

in - input to convert.
srclo - lower limit of input range.
srchi - upper limit of input range.
dstlo - lower limit of output range.
dsthi - upper limit of output range.

ID: 614

LPF 2nd order Butterworth lowpass filter

LPF.ar(in, freq, mul, add)

A second order low pass filter.
in - input signal to be processed
freq - cutoff frequency.

```
{ LPF.ar(Saw.ar(200,0.1), SinOsc.kr(XLine.kr(0.7,300,20),0,3600,4000)) }.play;

// kr:
(
{ var ctl = LPF.kr(LFPulse.kr(8), SinOsc.kr(XLine.kr(1, 30, 5)) + 2);
SinOsc.ar(ctl * 200 + 400)
}.play;
)

(
  var      LPF      Pulse        MouseX.kr(2, 50, 1)
SinOsc.ar(ctl * 200 + 400)
}.play;
)
```

ID: 615

LPZ1 two point average filter

LPZ1.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.5 * (in(i) + in(i-1))

which is a two point averager.
Compare:

```
{ WhiteNoise.ar(0.25) }.play;

{ LPZ1.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 616

LPZ2 two zero fixed lowpass

LPZ2.ar(in, mul, add)

A special case fixed filter. Implements the formula:

out(i) = 0.25 * (in(i) + (2*in(i-1)) + in(i-2))

Compare:

```
{ WhiteNoise.ar(0.25) }.play;

{ LPZ2.ar(WhiteNoise.ar(0.25)) }.play;
```

ID: 617

Median median filter

Median.ar(length, in, mul, add)

Returns the median of the last **length** input points.
This non linear filter is good at reducing impulse noise from a signal.
length - number of input points in which to find the median. Must be an odd number from 1 to 31.
If length is 1 then Median has no effect.
in - input signal to be processed

```
// a signal with impulse noise.
{ Saw.ar(500, 0.1) + Dust2.ar(100, 0.9) }.play;

// after applying median filter
{ Median.ar(3, Saw.ar(500, 0.1) + Dust2.ar(100, 0.9)) }.play;

// The median length can be increased for longer duration noise.

// a signal with longer impulse noise.
{ Saw.ar(500, 0.1) + LPZ1.ar(Dust2.ar(100, 0.9)) }.play;

// length 3 doesn't help here because the impulses are 2 samples long.
{ Median.ar(3, Saw.ar(500, 0.1) + LPZ1.ar(Dust2.ar(100, 0.9))) }.play;

// length 5 does better
{ Median.ar(5, Saw.ar(500, 0.1) + LPZ1.ar(Dust2.ar(100, 0.9))) }.play;

// long Median filters begin chopping off the peaks of the waveform
(
{
x = SinOsc.ar(1000, 0, 0.2);
[x, Median.ar(31, x)]
}.play;
)

// another noise reduction application:
```

```
Synth.play({ WhiteNoise.ar(0.1) + SinOsc.ar(800,0,0.1) });

// use Median filter for high frequency noise
Synth.play({ Median.ar(31, WhiteNoise.ar(0.1) + SinOsc.ar(800,0,0.1)) });

(
// use LeakDC for low frequency noise
Synth.play({
LeakDC.ar(Median.ar(31, WhiteNoise.ar(0.1) + SinOsc.ar(800,0,0.1)), 0.9)
});
)
```

ID: 618

MidEQ parametric filter

MidEQ.ar(in, freq, rq, db, mul, add)

Attenuates or boosts a frequency band.

in - input signal to be processed
freq - center frequency of the band in Hertz.
rq - the reciprocal of Q. bandwidth / cutoffFreq.
db - amount of boost (db $>$ 0) or attenuation (db $<$ 0) of the frequency band.

```
// mixer parametric eq as wahwah
{ MidEQ.ar(Saw.ar(200,0.2), FSinOsc.kr(1, 0, 24, 84).midicps, 0.3, 12) }.play


// notch filter
(
{ var in;
in = PinkNoise.ar(0.2) + SinOsc.ar(600, 0, 0.1);
MidEQ.ar(in, SinOsc.kr(0.2, 0.5pi) * 2 + 600, 0.01, -24)
}.play
)

/////
// first start the synth
(
     | freq=400, db=0, rq=0.1 |
var in;
in =  SinOsc.ar([ 400, 420 ], 0, 0.4);
MidEQ.ar(in, freq, Lag.kr(rq, 0.3), db)
}.play
)

// now play with its parameters to hear how the filter affects two frequencies
// that are very close to each other

x.set(\db, -12)
```

```
x.set(\rq, 0.1)
x.set(\rq, 0.03)
x.set(\freq, 410)
x.set(\freq, 420)
x.set(\freq, 400)
x.set(\freq, 500)
```

ID: 619

Normalizer flattens dynamics

Normalizer.ar(input, level, lookAheadTime)

Normalizes the input amplitude to the given level. Normalize will not overshoot like Compander will, but it needs to look ahead in the audio. Thus there is a delay equal to twice the lookAheadTime.

input - the signal to be processed.
level - the peak output amplitude level to which to normalize the input.
lookAheadTime - the buffer delay time. Shorter times will produce smaller delays and quicker transient response times, but may introduce amplitude modulation artifacts.

```
(
// example signal to process
Synth.play({
var z;
z = Decay2.ar(
Impulse.ar(8, LFSaw.kr(0.25, -0.6, 0.7)),
0.001, 0.3, FSinOsc.ar(500));
}, 0.8)
)

(
Synth.play({
var z;
z = Decay2.ar(
Impulse.ar(8, LFSaw.kr(0.25, -0.6, 0.7)),
0.001, 0.3, FSinOsc.ar(500));
[z, Normalizer.ar(z, 0.4, 0.01)]
}, 0.5)
)
```

ID: 620

OnePole one pole filter

OnePole.ar(in, coef, mul, add)

A one pole filter. Implements the formula :

out(i) = ((1 - abs(coef)) * in(i)) + (coef * out(i-1))

in - input signal to be processed
coef - feedback coefficient. Should be between -1 and +1

```
{ OnePole.ar(WhiteNoise.ar(0.5), 0.95) }.play
```

```
{ OnePole.ar(WhiteNoise.ar(0.5), -0.95) }.play
```

```
{ OnePole.ar(WhiteNoise.ar(0.5), Line.kr(-0.99, 0.99, 10)) }.play
```

ID: 621

OneZero one zero filter

OneZero.ar(in, coef, mul, add)

A one zero filter. Implements the formula :

out(i) = ((1 - abs(coef)) * in(i)) + (coef * in(i-1))

in - input signal to be processed
coef - feed forward coefficient. +0.5 makes a two point averaging filter (see also LPZ1), -0.5 makes a differentiator (see also HPZ1), +1 makes a single sample delay (see also Delay1),
-1 makes an inverted single sample delay.

```
{ OneZero.ar(WhiteNoise.ar(0.5), 0.5) }.play
```

```
{ OneZero.ar(WhiteNoise.ar(0.5), -0.5) }.play
```

```
{ OneZero.ar(WhiteNoise.ar(0.5), Line.kr(-0.5, 0.5, 10)) }.play
```

ID: 622

Ramp linear lag

Ramp.ar(in, lagTime, mul, add)
Ramp.kr(in, lagTime, mul, add)

This is similar to **Lag** but with a linear rather than exponential lag. This is useful for smoothing out control signals.

in - input signal
lagTime - 60 dB lag time in seconds.

```
Server.internal.boot;
(
// used to lag pitch
{
 SinOsc      // sine wave
  Ramp       // lag the modulator
LFPulse.kr(4, 0, 0.5, 50, 400), // frequency modulator
Line.kr(0, 1, 15)    // modulate lag time
),
  0, // phase
  0.3 // sine amplitude
)
}.scope;
)

// Compare
(
var pulse;
{
pulse = LFPulse.kr(8.772);
Out.kr(0,[Ramp.kr(pulse, 0.025), Lag.kr(pulse, 0.025), pulse]);
}.play(Server.internal);
Server.internal.scope(3, bufsize:  44100, rate:  \control, zoom:  40);
)
```

ID: 623

Resonz resonant filter

Resonz.ar(in, freq, rq, mul, add)

A two pole resonant filter with zeroes at z = +/- 1. Based on K. Steiglitz, "A Note on Constant-Gain
Digital Resonators," *Computer Music Journal*, vol 18, no. 4, pp. 8-10, Winter 1994.
in - input signal to be processed
freq - resonant frequency in Hertz
rq - bandwidth ratio (reciprocal of Q). rq = bandwidth / centerFreq
The reciprocal of Q is used rather than Q because it saves a divide operation inside the unit generator.

```
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, 0.1) }.play

// modulate frequency
{ Resonz.ar(WhiteNoise.ar(0.5), XLine.kr(1000,8000,10), 0.05) }.play

// modulate bandwidth
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(1, 0.001, 8)) }.play

// modulate bandwidth opposite direction
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(0.001, 1, 8)) }.play
```

ID: 624

RHPF

RHPF.ar(in, freq, q, mul, add)

A resonant high pass filter.
in - input signal to be processed
freq - cutoff frequency.
rq - the reciprocal of Q. bandwidth / cutoffFreq.

```
{ RHPF.ar(Saw.ar(200,0.1), FSinOsc.kr(XLine.kr(0.7,300,20), 0, 3600, 4000), 0.2) }.play;


(
{  var ctl = RHPF.kr(LFSaw.kr(2), SinOsc.kr(XLine.kr(0.07,30,20), 0, 35, 40), 0.05);
SinOsc.ar(ctl * 200 + 500);
}.play;
)
```

ID: 625

Ringz ringing filter

Ringz.ar(in, freq, decaytime, mul, add)

This is the same as Resonz, except that instead of a resonance parameter, the bandwidth is
specified in a 60dB ring decay time. One Ringz is equivalent to one component of the Klank UGen.
in - input signal to be processed
freq - resonant frequency in Hertz
decaytime - the 60 dB decay time of the filter

```
{ Ringz.ar(Dust.ar(3, 0.3), 2000, 2) }.play

{ Ringz.ar(WhiteNoise.ar(0.005), 2000, 0.5) }.play

// modulate frequency
{ Ringz.ar(WhiteNoise.ar(0.005), XLine.kr(100,3000,10), 0.5) }.play

{ Ringz.ar(Impulse.ar(6, 0, 0.3),  XLine.kr(100,3000,10), 0.5) }.play

// modulate ring time
{ Ringz.ar(Impulse.ar(6, 0, 0.3),  2000, XLine.kr(4, 0.04, 8)) }.play

// modulate ring time opposite direction
{ Ringz.ar(Impulse.ar(6, 0, 0.3), 2000, XLine.kr(0.04, 4, 8)) }.play


(
{
var exciter;
exciter = WhiteNoise.ar(0.001);
Mix.arFill(10, {
Ringz.ar(exciter,
XLine.kr(exprand(100.0,5000.0), exprand(100.0,5000.0), 20),
0.5)
})
```

```
}.play
)
```

ID: 626

RLPF

RLPF.ar(in, freq, rq, mul, add)

A resonant low pass filter.
in - input signal to be processed
freq - cutoff frequency.
rq - the reciprocal of Q. bandwidth / cutoffFreq.

```
{ RLPF.ar(Saw.ar(200,0.1), FSinOsc.kr(XLine.kr(0.7,300,20),3600,4000), 0.2) }.play;
```

```
(
{  var ctl = RLPF.ar(Saw.ar(5,0.1), 25, 0.03);
SinOsc.ar(ctl * 200 + 400) * 0.1;
}.play;
)
```

```
(
{  var ctl = RLPF.ar(Saw.ar(5,0.1), MouseX.kr(2, 200, 1), MouseY.kr(0.01, 1, 1));
SinOsc.ar(ctl * 200 + 400) * 0.1;
}.play;
)
```

ID: 627

SOS second order filter section (biquad)

SOS.ar(in, a0, a1, a2, b1, b2, mul, add)

A standard second order filter section. Filter coefficients are given directly rather than calculated for you.
Formula is equivalent to:

out(i) = (a0 * in(i)) + (a1 * in(i-1)) + (a2 * in(i-2))
+ (b1 * out(i-1)) + (b2 * out(i-2))

```
// example:  same as TwoPole
(
{
var rho, theta, b1, b2;
theta = MouseX.kr(0.2pi, pi);
rho = MouseY.kr(0.6, 0.99);
b1 = 2.0 * rho * cos(theta);
b2 = rho.squared.neg;
SOS.ar(LFSaw.ar(200, 0, 0.1), 1.0, 0.0, 0.0, b1, b2)
}.play
)


(
{
var rho, theta, b1, b2;
theta = MouseX.kr(0.2pi, pi);
rho = MouseY.kr(0.6, 0.99);
b1 = 2.0 * rho * cos(theta);
b2 = rho.squared.neg;
SOS.ar(WhiteNoise.ar(0.1 !  2), 1.0, 0.0, 0.0, b1, b2)
}.play
)

// example with SOS.kr kr as modulator
```

```
(
{
var rho, theta, b1, b2, vib;
theta = MouseX.kr(0.2pi, pi);
rho = MouseY.kr(0.6, 0.99);
b1 = 2.0 * rho * cos(theta);
b2 = rho.squared.neg;

vib = SOS.kr(LFSaw.kr(3.16), 1.0, 0.0, 0.0, b1, b2);
SinOsc.ar( vib * 200 + 600) * 0.2
}.play
)
```

ID: 628

TwoPole two pole filter

TwoPole.ar(in, freq, radius, mul, add)

A two pole filter. This provides lower level access to setting of pole location.
For general purposes Resonz is better.

in - input signal to be processed
freq - frequency of pole angle.
radius - radius of pole. Should be between 0 and +1

```
// examples

{ TwoPole.ar(WhiteNoise.ar(0.005), 2000, 0.95) }.play

{ TwoPole.ar(WhiteNoise.ar(0.005), XLine.kr(800,8000,8), 0.95) }.play

{ TwoPole.ar(WhiteNoise.ar(0.005), MouseX.kr(800,8000,1), 0.95) }.play
```

ID: 629

TwoZero two zero filter

TwoZero.ar(in, freq, radius, mul, add)

A two zero filter.
in - input signal to be processed
freq - frequency of zero angle.
radius - radius of zero.

```
{ TwoZero.ar(WhiteNoise.ar(0.125), XLine.kr(20,20000,8), 1) }.play
```

ID: 630

Wrap wrap a signal outside given thresholds

Wrap.ar(in, lo, hi)
Wrap.kr(in, lo, hi)

This differs from the **BinaryOpUGen wrap2** in that it allows one to set both low and high thresholds.
in - signal to be wrapped
lo - low threshold of wrapping
hi - high threshold of wrapping

```
Server.internal.boot;

{ Wrap.ar(SinOsc.ar(440, 0, 0.2), -0.15, 0.15) }.scope;
```

25.9 InfoUGens

ID: 631

BufChannels current number of channels of soundfile in buffer

superclass: BufInfoUGenBase

***kr(bufnum)**
***ir(bufnum)**
the .ir method is not the safest choice. Since a buffer can be reallocated at any time, using .ir will not track the changes.

ID: 632

BufDur current duration of soundfile in buffer

superclass: BufInfoUGenBase

returns the current duration of soundfile

***kr(bufnum)**
***ir(bufnum)**
the .ir method is not the safest choice. Since a buffer can be reallocated at any time, using .ir will not track the changes.

```
// example

          "/b_allocRead"      "sounds/a11wlk01.wav"


{ BufRd.ar(1, 0,  Sweep.ar(Impulse.ar(BufDur.kr(0).reciprocal), BufSampleRate.kr(0))) }.play;
```

ID: 633

BufFrames current number of frames allocated in the buffer

superclass: BufInfoUGenBase

returns the current number of allocated frames

***kr(bufnum)**
***ir(bufnum)**
the .ir method is not the safest choice. Since a buffer can be reallocated at any time, using .ir will not track the changes.

```
// example

        "/b_allocRead"     "sounds/a11wlk01.wav"

// indexing with a phasor
{ BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0), 0, BufFrames.kr(0))) }.play;


// indexing by hand
{ BufRd.ar(1, 0, K2A.ar(MouseX.kr(0, BufFrames.kr(0)))) }.play;
```

ID: 634

BufRateScale buffer rate scaling in respect to server samplerate

superclass: BufInfoUGenBase

returns a ratio by which the playback of a soundfile is to be scaled

***kr(bufnum)**
***ir(bufnum)**
the .ir method is not the safest choice. Since a buffer can be reallocated at any time, using .ir will not track the changes.

```
// example

          "/b_allocRead"     "sounds/a11wlk01.wav"

(
x = { arg rate=1;
BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)))
}.play;
)
```

ID: 635

BufSampleRate buffer sample rate

superclass: BufInfoUGenBase

returns the buffers current sample rate

***kr(bufnum)**
***ir(bufnum)**
the .ir method is not the safest choice. Since a buffer can be reallocated at any time, using .ir will not track the changes.

```
// example

          "/b_allocRead"     "sounds/a11wlk01.wav"

// compares a 1102.5 Hz sine tone (11025 * 0.1, left) with a 1100 Hz tone (right)
// the apollo sample has a sample rate of 11.025 kHz
(
{
var freq;
freq = [ BufSampleRate.kr(0) * 0.1, 1100];
SinOsc.ar(freq, 0, 0.1)
}.play;
)
```

ID: 636

NumRunningSynths number of currently running synths

superclass: InfoUGenBase

*ir

```
// example:  frequency is derived from the number of synths running
(
SynthDef "numRunning"    arg
Out.ar(out, SinOsc.ar(NumRunningSynths.ir * 200 + 400, 0, 0.1));
}).send(s);
)

s.sendMsg("/s_new", "numRunning", -1, 0, 0);
s.sendMsg("/s_new", "numRunning", -1, 0, 0);
s.sendMsg("/s_new", "numRunning", -1, 0, 0);
s.sendMsg("/s_new", "numRunning", -1, 0, 0);
```

ID: 637

SampleDur duration of one sample

superclass: InfoUGenBase

returns the current sample duration of the server

***ir**

equivalent to 1 / **SampleRate**

ID: 638

SampleRate server sample rate

superclass: InfoUGenBase

returns the current sample rate of the server

*ir

```
// example

          "/b_allocRead"     "sounds/a11wlk01.wav"

// compares a 441 Hz sine tone derived from sample rate (44100 * 0.01, left)
// with a 440 Hz tone (right), resultin in a 1 Hz beating
(
{
var freq;
freq = [ SampleRate.ir(0) * 0.01, 440];
SinOsc.ar(freq, 0, 0.1)
}.play;
)
```

ID: 639

SubsampleOffset offset from synth start within one sample

superclass: InfoUGenBase

SubsampleOffset.ir

When a synth is created from a time stamped osc-bundle, it starts calculation at the next possible block (normally 64 samples). Using an **OffsetOut** ugen, one can delay the audio so that it matches sample accurately.
For some synthesis methods, one needs subsample accuracy. **SubsampleOffset** provides the information where, within the current sample, the synth was scheduled. It can be used to offset envelopes or resample the audio output.

see also: [OffsetOut]

```
// example:  demonstrate cubic subsample interpolation


Server               Server           // switch servers for scope


// impulse train that can be moved between samples
(
SynthDef \Help_SubsampleOffset    | out, addOffset|
var in, dt, sampDur, extraSamples, sampleOffset, resampledSignal;
in = Impulse.ar(2000, 0, 0.3); // some input.
  sampDur = SampleDur      // duration of one sample
 extraSamples = 4;   // DelayC needs at least 4 samples buffer
 sampleOffset = 1 - SubsampleOffset     // balance out subsample offset
sampleOffset = sampleOffset + MouseX.kr(0, addOffset); // add a mouse dependent offset
 // cubic resampling:
resampledSignal = DelayC.ar(in,
maxdelaytime:  sampDur * (1 + extraSamples),
delaytime:  sampDur * (sampleOffset + extraSamples)
```

```
);
OffsetOut.ar(out, resampledSignal)
}).send(s);
)

// create 2 pulse trains 1 sample apart, move one relatively to the other.
// when cursor is at the left, the impulses are adjacent, on the right, they are
// exactly 1 sample apart.

(
var dt = s.sampleRate.reciprocal; // 1 sample delay
s.sendBundle(0.2, [9, \Help_SubsampleOffset, s.nextNodeID, 1, 1, \out, 40, \addOffset, 3]);
s.sendBundle(0.2 + dt, [9, \Help_SubsampleOffset, s.nextNodeID, 1, 1, \out, 40, \addOffset, 0]);
)

s.scope(1, 40, zoom:  0.2);




// example of a subsample accurate sine grain:
// (I don't hear a difference to normal sample accurate grains, but maybe
// someone could add an example that shows the effect)

(
SynthDef "Help_Subsample_Grain"
{ arg out=0, freq=440, sustain=0.005, attack=0.001;
var env, offset, sig, sd;

sd = SampleDur.ir;
offset = (1 - SubsampleOffset.ir) * sd;
  // free synth after delay:
Line.ar(1,0, attack + sustain + offset, doneAction:2);   env = EnvGen.kr(Env.perc(attack, sustain, 0.2));

sig = SinOsc.ar(freq, 0, env);

sig = DelayC.ar(sig, sd * 4, offset);
OffsetOut.ar(out, sig);
}, [\ir, \ir, \ir, \ir]).send(s);
```

```
)

(
 Routine
loop {
s.sendBundle(0.2, [9, \Help_Subsample_Grain, -1, 1, 1, \freq, 1000]);
rrand(0.001, 0.002).wait;
}
}.play;
)
```

25.10 InOut

ID: 640

AudioIn read audio input

AudioIn.ar(channel, mul, add)

Reads audio from the sound input hardware.
channel - input channel number to read.
Channel numbers begin at 1.

// watch the feedback

// patching input to output
(
```
SynthDef "help-AudioIn"   arg
Out.ar(out,
AudioIn.ar(1)
)
}).play;
```
)

// stereo through patching from input to output
```
(
SynthDef "help-AudioIn"   arg
Out.ar(out,
AudioIn.ar([1,2])
)
}).play;
)
```

ID: 641

In read a signal from a bus

superclass: AbstractIn

***ar(bus, numChannels)** - read a signal from an audio bus.
***kr(bus, numChannels)** -read a signal from a control bus.

bus - the index of the bus to read in from.
numChannels - the number of channels (i.e. adjacent buses) to read in. The default is 1. You cannot modulate this number by assigning it to an argument in a SynthDef.

In.kr is functionally similar to **[InFeedback].** That is it reads all data on the bus whether it is from the current cycle or not. This allows for it to receive data from later in the node order. In.ar reads only data from the current cycle, and will zero data from earlier cycles (for use within that synth; the data remains on the bus). Because of this and the fact that the various out ugens *mix* their output with data from the current cycle but *overwrite* data from an earlier cycle it may be necessary to use a private control bus when this type of feedback is desired. There is an example below which demonstrates the problem. See the **[InFeedback]** and **[Order-of-execution]** helpfiles for more details.

Note that using the **Bus** class to allocate a multichannel bus simply reserves a series of adjacent bus indices with the **[Server]** object's bus allocators. abus.index simply returns the first of those indices. When using a Bus with an **In** or **[Out]** ugen there is nothing to stop you from reading to or writing from a larger range, or from hardcoding to a bus that has been allocated. You are responsible for making sure that the number of channels match and that there are no conflicts. See the **[Server-Architecture]** and **[Bus]** helpfiles for more information on buses and how they are used.

The hardware input busses begin just after the hardwrae output busses and can be read from using In.ar. (See **[Server-Architecture]** for more details.) The number of hardware input and output busses can vary depending on your Server's options. For a convienent wrapper class which deals with this issue see **[AudioIn]**.

read from an audio bus

```
(
s = Server.local;
s.boot;
```

```
)

(
 SynthDef "help-PinkNoise"    arg
Out.ar(out, PinkNoise.ar(0.1))
}).send(s);

SynthDef("help-In", { arg out=0, in=0;
var input;
input = In.ar(in, 1);
Out.ar(out, input);

}).send(s);
)

 //play noise on the right channel
 x = Synth "help-PinkNoise"   \out

 //read the input and play it out on the left channel
Synth.after(x, "help-In", [\out, 0, \in, 1]);
```

read from a control bus

```
(
SynthDef("help-InKr",{ arg out=0, in=0;
Out.ar(out,
SinOsc.ar(In.kr(in, 1), 0, 0.1)
)
}).send(s);
SynthDef("help-lfo", { arg out=0;
  Out         LFNoise1.kr(0.3, 200, 800)
}).send(s);
)


b = Bus.control(s,1);
b.set(800);

Synth("help-InKr",[\in, b.index]);
```

```
b.set(400);
b.set(300);
Synth("help-lfo", [\out, b.index]);
```

read control data from a synth later in the node order

```
(
 SynthDef "help-Infreq"    arg
Out.ar(0, FSinOsc.ar(In.kr(bus), 0, 0.5));
}).send(s);

SynthDef("help-Outfreq", { arg freq = 400, bus;
Out.kr(bus, SinOsc.kr(1, 0, freq/40, freq));
}).send(s);

b = Bus.control(s,1);
)

 // add the first control Synth at the tail of the default server; no audio yet
x = Synth.tail(s, "help-Outfreq", [\bus, b.index]);

 // add the sound producing Synth BEFORE it; It receives x's data from the previous cycle
y = Synth.before(x, "help-Infreq", [\bus, b.index]);

 // add another control Synth before y, at the head of the server
 // It now overwrites x's cycle old data before y receives it
z = Synth.head(s, "help-Outfreq", [\bus, b.index, \freq, 800]);

 // get another bus
c = Bus.control(s, 1);

 // now y receives x's data even though z is still there
y.set(\bus, c.index); x.set(\bus, c.index);

x.free; y.free; z.free;
```

ID: 642

InFeedback read signal from a bus with a current or one cycle old timestamp

superclass: MultiOutUGen

***ar(bus, numChannels)**

bus - the index of the bus to read in from.
numChannels - the number of channels (i.e. adjacent buses) to read in. The default is 1. You cannot modulate this number by assigning it to an argument in a SynthDef.

When the various output ugens (**Out**, **OffsetOut**, **XOut**) write data to a bus, they *mix* it with any data from the current cycle, but *overwrite* any data from the previous cycle. (**ReplaceOut** overwrites all data regardless.) Thus depending on node order and what synths are writing to the bus, the data on a given bus may be from the current cycle or be one cycle old at the time of reading. **In.ar** checks the timestamp of any data it reads in and zeros any data from the previous cycle (for use within that node; the data remains on the bus). This is fine for audio data, as it avoids feedback, but for control data it is useful to be able to read data from any place in the node order. For this reason **In.kr** also reads data that is older than the current cycle.

In some cases we might also want to read audio from a node later in the current node order. This is the purpose of InFeedback. The delay introduced by this is one block size, which equals about 0.0014 sec at the default block size and sample rate. (See the resonator example below to see the implications of this.)

The variably mixing and overwriting behaviour of the output ugens can make order of execution crucial. (No pun intended.) For example with a node order like the following the InFeedback ugen in Synth 2 will only receive data from Synth 1 (-> = write out; <- = read in):

Synth 1 -> busA this synth overwrites the output of Synth3 before it reaches Synth 2
Synth 2 (with InFeedback) <- busA
Synth 3 -> busA

If Synth 1 were moved after Synth 2 then Synth 2's InFeedback would receive a mix of the output from Synth 1 and Synth 3. This would also be true if Synth 2 came after

Synth1 and Synth 3. In both cases data from Synth 1 and Synth 3 would have the same time stamp (either current or from the previous cycle), so nothing would be overwritten.

Because of this it is often useful to allocate a separate bus for feedback. With the following arrangement Synth 2 will receive data from Synth3 regardless of Synth 1's position in the node order.

Synth 1 -> busA
Synth 2 (with InFeedback) <- busB
Synth 3 -> busB + busA

The second example below demonstrates this issue.

See also **LocalIn** and **LocalOut**.

Examples

audio feedback modulation:

```
  (
 SynthDef("help-InFeedback", { arg out=0, in=0;
var input, sound;
input = InFeedback.ar(in, 1);
sound = SinOsc.ar(input * 1300 + 300, 0, 0.4);
Out.ar(out, sound);

}).play;
)
```

this shows how a node can read audio from a bus that is being written to by a synth following it:

```
(
SynthDef("help-InFeedback", { arg out=0, in=0;
Out.ar(out,
InFeedback.ar(in, 1)
);
}).send(s);
SynthDef("help-SinOsc", { arg out=0, freq=440;
```

```
Out.ar(out, SinOsc.ar(freq, 0, 0.1))
}).send(s);
)

x = Bus.audio(s, 1);

 // read from bus n play to bus 0 (silent)
a = Synth("help-InFeedback",[\in, x.index, \out, 0]);

 // now play a synth after this one, playing to bus x
b = Synth.after(a, "help-SinOsc", [\out, x.index]);

 // add another synth before a which also writes to bus x
 // now you can't hear b, as its data is one cycle old, and is overwritten by c
c = Synth.before(a, "help-SinOsc", [\out, x.index, \freq, 800]);

 // free c and you can hear b again
c.free;
x.free;

a.free; b.free;
```

The example below implements a resonator. Note that you must subtract the blockSize in order for the tuning to be correct. See **LocalIn** for an equivalent example.

```
(
var play, imp, initial;
 SynthDef "testRes"

 play = InFeedback            // 10 is feedback channel
imp = Impulse.ar(1);

 // feedback
OffsetOut.ar(10, DelayC.ar(imp + (play * 0.995), 1,
440.reciprocal - ControlRate.ir.reciprocal)); // subtract block size

OffsetOut.ar(0, play);

}).play(s);
```

```
 // Compare with this for tuning
{ SinOsc.ar(440, 0, 0.2) }.play(s, 1);
)
```

ID: 643

InTrig generate a trigger anytime a bus is set

superclass: MultiOutUGen

***kr(bus, numChannels)**

bus - the index of the bus to read in from.
numChannels - the number of channels (i.e. adjacent buses) to read in. The default is 1. You cannot modulate this number by assigning it to an argument in a SynthDef.

Any time the bus is "touched" ie. has its value set (using "/c_set" etc.), a single impulse trigger will be generated. Its amplitude is the value that the bus was set to.

```
s = Server.local;
b = Bus.control(s,1);

SynthDef("help-InTrig",{arg out=0,busnum=0;
var inTrig;
inTrig = InTrig.kr( busnum );
Out.ar(out,
EnvGen.kr(Env.perc,gate:  inTrig,levelScale:  inTrig ) * SinOsc.ar
)
}).play(s,[\out, 0, \busnum, b.index]);



b.set(1.0);

b.value = 1.0;

b.value = 0.2;

b.value = 0.1;
```

compare with [In] example.

ID: 644

LocalIn define and read from buses local to a synth

superclass: AbstractIn

***ar(numChannels)** - define and read from an audio bus local to the enclosing synth.
***kr(numChannels)** -define and read from a control bus local to the enclosing synth.

numChannels - the number of channels (i.e. adjacent buses) to read in. The default is 1. You cannot modulate this number by assigning it to an argument in a SynthDef.

LocalIn defines buses that are local to the enclosing synth. These are like the global buses, but are more convenient if you want to implement a self contained effect that uses a feedback processing loop.
There can only be one audio rate and one control rate **LocalIn** per SynthDef.
The audio can be written to the bus using **LocalOut**.

N.B. Audio written to a **LocalOut** will not be read by a corresponding LocalIn until the next cycle, i.e. one block size of samples later. Because of this it is important to take this additional delay into account when using LocalIn to create feedback delays with delay times shorter than the threshold of pitch (i.e. < 0.05 seconds or > 20Hz), or where sample accurate alignment is required. See the resonator example below.

```
(
{
var source, local;

source = Decay.ar(Impulse.ar(0.3), 0.1) * WhiteNoise.ar(0.2);

local = LocalIn.ar(2) + [source, 0]; // read feedback, add to source

local = DelayN.ar(local, 0.2, 0.2); // delay sound

 // reverse channels to give ping pong effect, apply decay factor
LocalOut.ar(local.reverse * 0.8);

Out.ar(0, local);
}.play;
)
```

Where: Help→UGens→InOut→LocalIn

```
(
z = SynthDef("tank", {
var local, in;

in = Mix.fill(12, {
Pan2.ar(
Decay2.ar(Dust.ar(0.05), 0.1, 0.5, 0.1)
* FSinOsc.ar(IRand(36,84).midicps).cubed.max(0),
Rand(-1,1))
});
in = in + Pan2.ar(Decay2.ar(Dust.ar(0.03), 0.04, 0.3) * BrownNoise.ar, 0);

4.do { in = AllpassN.ar(in, 0.03, {Rand(0.005,0.02)}.dup, 1); };

local = LocalIn.ar(2) * 0.98;
local = OnePole.ar(local, 0.5);

local = Rotate2.ar(local[0], local[1], 0.23);
local = AllpassN.ar(local, 0.05, {Rand(0.01,0.05)}.dup, 2);

local = DelayN.ar(local, 0.3, [0.19,0.26]);
local = AllpassN.ar(local, 0.05, {Rand(0.03,0.15)}.dup, 2);

local = LeakDC.ar(local);
local = local + in;

LocalOut.ar(local);

Out.ar(0, local);
}).play;
)
```

```
(
z = SynthDef("tape", {
var local, in, amp;
```

Where: Help→UGens→InOut→LocalIn

```
in = AudioIn.ar([1,2]);

amp = Amplitude.kr(Mix.ar(in));
in = in * (amp > 0.02); // noise gate

local = LocalIn.ar(2);
local = OnePole.ar(local, 0.4);
local = OnePole.ar(local, -0.08);

local = Rotate2.ar(local[0], local[1], 0.2);

local = DelayN.ar(local, 0.25, 0.25);

local = LeakDC.ar(local);
local = ((local + in) * 1.25).softclip;

LocalOut.ar(local);

Out.ar(0, local * 0.1);
}).play;
)

// Resonator, must subtract blockSize for correct tuning
(
var play, imp, initial;
SynthDef "testRes"

play = LocalIn.ar(1);
imp = Impulse.ar(1);

LocalOut.ar(DelayC.ar(imp + (play * 0.995), 1, 440.reciprocal - ControlRate.ir.reciprocal)); // for feed-
back

OffsetOut.ar(0, play);

}).play(s);

{SinOsc.ar(440, 0, 0.2) }.play(s, 1); // compare pitch
```

)

ID: 645

LocalOut write to buses local to a synth

superclass: AbstractOut

***ar(channelsArray)** - write a signal to an audio bus local to the enclosing synth.
***kr(channelsArray)** -write a signal to a control bus local to the enclosing synth.

channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

LocalOut writes to buses that are local to the enclosing synth. The buses should have been defined by a **LocalIn** ugen. The **channelsArray** must be the same number of channels as were declared in the LocalIn. These are like the global buses, but are more convenient if you want to implement a self contained effect that uses a feedback processing loop.
See **[LocalIn]**.

N.B. Audio written to a LocalOut will not be read by a corresponding **LocalIn** until the next cycle, i.e. one block size of samples later. Because of this it is important to take this additional delay into account when using LocalIn to create feedback delays with delay times shorter than the threshold of pitch (i.e. $<$ 0.05 seconds or $>$ 20Hz), or where sample accurate alignment is required. See the resonator example below.

```
(
{
var source, local;

source = Decay.ar(Impulse.ar(0.3), 0.1) * WhiteNoise.ar(0.2);

local = LocalIn.ar(2) + [source, 0]; // read feedback, add to source

local = DelayN.ar(local, 0.2, 0.2); // delay sound

 // reverse channels to give ping pong effect, apply decay factor
LocalOut.ar(local.reverse * 0.8);

Out.ar(0, local);
}.play;
```

```
)



(
z = SynthDef("tank", {
var local, in;

in = Mix.fill(12, {
Pan2.ar(
Decay2.ar(Dust.ar(0.05), 0.1, 0.5, 0.1)
* FSinOsc.ar(IRand(36,84).midicps).cubed.max(0),
Rand(-1,1))
});
in = in + Pan2.ar(Decay2.ar(Dust.ar(0.03), 0.04, 0.3) * BrownNoise.ar, 0);

4.do { in = AllpassN.ar(in, 0.03, {Rand(0.005,0.02)}.dup, 1); };

local = LocalIn.ar(2) * 0.98;
local = OnePole.ar(local, 0.5);

local = Rotate2.ar(local[0], local[1], 0.23);
local = AllpassN.ar(local, 0.05, {Rand(0.01,0.05)}.dup, 2);

local = DelayN.ar(local, 0.3, [0.19,0.26]);
local = AllpassN.ar(local, 0.05, {Rand(0.03,0.15)}.dup, 2);

local = LeakDC.ar(local);
local = local + in;

LocalOut.ar(local);

Out.ar(0, local);
}).play;
)



(
z = SynthDef("tape", {
```

```
var local, in, amp;

in = AudioIn.ar([1,2]);

amp = Amplitude.kr(Mix.ar(in));
in = in * (amp > 0.02); // noise gate

local = LocalIn.ar(2);
local = OnePole.ar(local, 0.4);
local = OnePole.ar(local, -0.08);

local = Rotate2.ar(local[0], local[1], 0.2);

local = DelayN.ar(local, 0.25, 0.25);

local = LeakDC.ar(local);
local = ((local + in) * 1.25).softclip;

LocalOut.ar(local);

Out.ar(0, local * 0.1);
}).play;
)

// Resonator, must subtract blockSize for correct tuning
(
var play, imp, initial;
SynthDef "testRes"

play = LocalIn.ar(1);
imp = Impulse.ar(1);

LocalOut.ar(DelayC.ar(imp + (play * 0.995), 1, 440.reciprocal - ControlRate.ir.reciprocal)); // for feed-
back

OffsetOut.ar(0, play);

}).play(s);

{SinOsc.ar(440, 0, 0.2) }.play(s, 1); // compare pitch
```

)

ID: 646

OffsetOut write a signal to a bus with sample accurate timing

superclass: Out

Output signal to a bus, the sample offset within the bus is kept exactly; i.e. if the synth is scheduled to be started part way through a control cycle, OffsetOut will maintain the correct offset by buffering the output and delaying it until the exact time that the synth was scheduled for.

This ugen is used where sample accurate output is needed.

***ar(bus, channelsArray)** - write a signal to an audio bus.
***kr(bus, channelsArray)** -write a signal to a control bus.

bus - the index of the bus to write out to. The lowest numbers are written to the audio hardware.
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

See the **[Server-Architecture]** and **[Bus]** helpfiles for more information on buses and how they are used.

for achieving subsample accuracy see: [SubsampleOffset]

```
// example

(
 SynthDef "help-OffsetOut"
{ arg out=0, freq=440, dur=0.05;
var env;
env = EnvGen.kr(Env.perc(0.01, dur, 0.2), doneAction:2);
OffsetOut.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);

 SynthDef "help-Out"
{ arg out=0, freq=440, dur=0.05;
var env;
```

```
env = EnvGen.kr(Env.perc(0.01, dur, 0.2), doneAction:2);
   //compare to Out:
Out.ar(out, SinOsc.ar(freq, 0, env))
}).send(s);
)


 // these are in sync
(
 Routine
loop {
s.sendBundle(0.2, ["/s_new", "help-OffsetOut", -1]);
0.01.wait;
}
}).play;
)

 // these are less reliably in sync and are placed at multiples of blocksize.
(
 Routine
loop {
s.sendBundle(0.2, ["/s_new", "help-Out", -1]);
0.01.wait;
}
}).play;
)
```

Note that if you have an input to the synth, it will be coming in and its normal time, then mixed in your synth, and then delayed with the output.
So you shouldn't use OffsetOut for effects or gating.

```
SynthDef "trig1"
var gate,tone;
gate = Trig1.ar(1.0,t);
 tone = In           // tone comes in normally
 // but is then delayed when by the OffsetOut
 OffsetOut
```

```
tone * EnvGen.ar(
Env([0,0.1,0.1,0],[0.01,1.0,0.01],[-4,4],2),
gate,doneAction:  2
)
)
})
```

ID: 647

Out write a signal to a bus

superclass: AbstractOut

***ar(bus, channelsArray)** - write a signal to an audio bus.
***kr(bus, channelsArray)** -write a signal to a control bus.

bus - the index of the bus to write out to. The lowest numbers are written to the audio hardware.
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

N.B. Out is subject to control rate jitter. Where sample accurate output is needed, use **OffsetOut**.
Note that using the **Bus** class to allocate a multichannel bus simply reserves a series of adjacent bus indices with the **Server** object's bus allocators. abus.index simply returns the first of those indices. When using a Bus with an **In** or **Out** ugen there is nothing to stop you from reading to or writing from a larger range, or from hardcoding to a bus that has been allocated. You are responsible for making sure that the number of channels match and that there are no conflicts.

See the **[Server-Architecture]** and **[Bus]** helpfiles for more information on buses and how they are used.

```
  (
 SynthDef("help-out", { arg out=0, freq=440;
var source;
source = SinOsc.ar(freq, 0, 0.1);

   // write to the bus, adding to previous contents
Out.ar(out, source);

}).send(s);
)



Synth("help-out", [\freq, 500]);
Synth("help-out", [\freq, 600]);
```

```
Synth("help-out", [\freq, 700]);
```

ID: 648

ReplaceOut send signal to a bus, overwriting previous contents

superclass: Out

***ar(bus, channelsArray)** - write a signal to an audio bus.
***kr(bus, channelsArray)** -write a signal to a control bus.

bus - the index of the bus to write out to. The lowest numbers are written to the audio hardware.
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

Out adds it's output to a given bus, making it available to all nodes later in the node tree. (See **Synth** and **Order-of-execution** for more information.) ReplaceOut overwrites those contents. This can make it useful for processing.

See the **Server-Architecture** and **Bus** helpfiles for more information on buses and how they are used.

```
(
SynthDef("ReplaceOutHelp", { arg out=0, freq=440;
var source;
source = SinOsc.ar(freq, 0, 0.1);

   // write to the bus, replacing previous contents
ReplaceOut.ar(out, source);

}).send(s);
)

 // each Synth replaces the output of the previous one
x = Synth.tail(s, "ReplaceOutHelp", [\freq, 500]);
y = Synth.tail(s, "ReplaceOutHelp", [\freq, 600]);
z = Synth.tail(s, "ReplaceOutHelp", [\freq, 700]);

 // release them in reverse order; the older Synths are still there.
z.free;
```

```
y.free;
x.free;
```

ID: 649

SharedIn read from a shared control bus

superclass: AbstractIn

SharedIn.kr(bus, numChannels)

Reads from a control bus shared between the internal server and the SC client. Control rate only. Writing to a shared control bus from the client is synchronous. When not using the internal server use node arguments or the set method of **Bus** (or /c_set in messaging style).
bus - the index of the shared control bus to read from
numChannels - the number of channels (i.e. adjacent buses) to read in. The default is 1. You cannot modulate this number by assigning it to an argument in a SynthDef.

```
(
// only works with the internal server
s = Server.internal;
s.boot;
)

(
SynthDef "help-SharedIn1"
Out.ar(0, SinOsc.ar(Lag.kr(SharedIn.kr(0, 1), 0.01), 0, 0.2));
}).send(s);
SynthDef "help-SharedIn2"
Out.ar(1, SinOsc.ar(Lag.kr(SharedIn.kr(0, 1), 0.01, 1.5), 0, 0.2));
}).send(s);
)

(
s.setSharedControl(0, 300); // an initial value
s.sendMsg(\s_new, "help-SharedIn1", x = s.nextNodeID, 0, 1);
s.sendMsg(\s_new, "help-SharedIn2", y = s.nextNodeID, 0, 1);

Routine
30.do({
s.setSharedControl(0, 300 * (10.rand + 1));
0.2.wait;
```

```
});
s.sendMsg(\n_free, x);
s.sendMsg(\n_free, y);
}).play;
)


s.quit;
```

ID: 650

SharedOut write to a shared control bus

superclass: AbstractOut

SharedOut.kr(bus, channelsArray)

Reads from a control bus shared between the internal server and the SC client. Control rate only. Reading from a shared control bus on the client is synchronous. When not using the internal server use the get method of **Bus** (or /c_get in messaging style) or **SendTrig** with an **OSCresponder** or **OSCresponderNode**.
bus - the index of the shared control bus to read from
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

```
(
// only works with the internal server
s = Server.internal;
s.boot;
)

(
SynthDef "help-SharedOut"
SharedOut.kr(0, SinOsc.kr(0.2));
}).send(s);
)

(
s.sendMsg(\s_new, "help-SharedOut", x = s.nextNodeID, 0, 1);
s.sendMsg(\n_trace, x);

// poll the shared control bus
Routine
30.do({
s.getSharedControl(0).postln;
0.2.wait;
});
}).play;
)
```

```
s.quit;
```

ID: 651

XOut send signal to a bus, crossfading with previous contents

superclass: AbstractOut

***ar(bus, xfade, channelsArray)** - crossfade an audio bus.
***kr(bus, xfade, channelsArray)** - crossfade an control bus.

bus - the index of the bus to write out to. The lowest numbers are written to the audio hardware.
xfade - crossfade level.
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

xfade is a level for the crossfade between what is on the bus and what you are sending. The algorithm is equivalent to this:

bus_signal = (input_signal * xfade) + (bus_signal * (1 - xfade));

See the **[Server-Architecture]** and **[Bus]** helpfiles for more information on buses and how they are used.

```
(
SynthDef("help-SinOsc", { arg freq=440, out;
Out.ar(out, SinOsc.ar(freq, 0, 0.1))
}).send(s);

SynthDef("help-XOut", { arg out=0, xFade=1;
var source;
source = PinkNoise.ar(0.05);

   // write to the bus, crossfading with previous contents
XOut.ar(out, xFade, source);

}).send(s);
)

Synth("help-SinOsc", [\freq, 500]);
a = Synth.tail(s, "help-XOut");
```

```
a.set(\xFade, 0.7);
a.set(\xFade, 0.4);
a.set(\xFade, 0.0);
```

25.11 Miscellanea

ID: 652

DiskIn stream in audio from a file

DiskIn.ar(numChannels, bufnum)

Continously play a longer soundfile from disk. This requires a buffer to be preloaded with one buffer size of sound.

```
        // start the server

(
SynthDef("help-Diskin", { arg bufnum = 0;
Out.ar(0, DiskIn.ar(1, bufnum));
}).send(s)
)
```

OSC Messaging Style

```
// allocate a disk i/o buffer
s.sendMsg("/b_alloc", 0, 65536, 1);

// open an input file for this buffer, leave it open
          "/b_read"     "sounds/a11wlk01-44_1.aiff"

// create a diskin node
s.sendMsg("/s_new", "help-Diskin", x = s.nextNodeID, 1, 1);

          "/b_close"      // close the file (very important!)


// again
// don't need to reallocate and Synth is still reading
          "/b_read"     "sounds/a11wlk01-44_1.aiff"

s.sendMsg("/n_free", x); // stop reading

          "/b_close"      // close the file.
```

```
        "/b_free"       // frees the buffer
```

Using Buffer (Object Style)

```
b = Buffer.cueSoundFile(s, "sounds/a11wlk01-44_1.aiff", 0, 1);

x = { DiskIn.ar(1, b.bufnum) }.play;

b.close;

// again
// note the like named instance method, but different arguments
              "sounds/a11wlk01-44_1.aiff"

x.free; b.close; b.free;



// loop it (for better looping use PlayBuf!)
(
    "sounds/a11wlk01-44_1.aiff"
a = SoundFile.new;
a.openRead(p);
d = a.numFrames/s.sampleRate; // get the duration
        // don't forget
b = Buffer.cueSoundFile(s, p, 0, 1);
f = { DiskIn.ar(1, b.bufnum) };
x = f.play;
    Routine
loop({ d.wait; x.free; x = f.play; b.close( b.cueSoundFileMsg(p, 0)) });
}).play; )
                                // you need to do all these to properly cleanup



// cue and play right away
(
```

```
SynthDef("help-Diskin", { arg bufnum = 0;
Out.ar(0, DiskIn.ar(1, bufnum));
}).send(s);
)
(
x = Synth.basicNew("help-Diskin");
m = { arg buf; x.addToHeadMsg(nil, [\bufnum,buf.bufnum])};

b = Buffer.cueSoundFile(s,"sounds/a11wlk01-44_1.aiff",0,1, completionMessage:  m);

)
```

See **PlayBuf** for playing a soundfile loaded into memory.

ID: 653

DiskOut

DiskOut.ar(bufnum, channelsArray)

Record to a soundfile to disk. Uses a **Buffer**.
bufnum - the number of the buffer to write to (prepared with /b-write or Buffer.write)
channelsArray - the Array of channels to write to the file.

```
        // start the server
(
// something to record
SynthDef "bubbles"
var f, zout;
f = LFSaw.kr(0.4, 0, 24, LFSaw.kr([8,7.23], 0, 3, 80)).midicps; // glissando function
zout = CombN.ar(SinOsc.ar(f, 0, 0.04), 0.2, 0.2, 4); // echoing sine wave
Out.ar(0, zout);
}).send(s);

// this will record to the disk
SynthDef "help-Diskout"   arg
DiskOut.ar(bufnum, In.ar(0,2));
}).send(s);

// this will play it back
SynthDef "help-Diskin-2chan"    arg
Out.ar(0, DiskIn.ar(2, bufnum));
}).send(s);
)
```

Object Style:

```
// start something to record
x = Synth.new("bubbles");

// allocate a disk i/o buffer
b= Buffer.alloc(s, 65536, 2);
```

```
// create an output file for this buffer, leave it open
        "recordings/diskouttest.aiff"  "aiff"  "int16"        true
// create the diskout node; making sure it comes after the source
d = Synth.tail(nil, "help-Diskout", ["bufnum", b.bufnum]);
// stop recording
d.free;
// stop the bubbles
x.free;
// close the buffer and the soundfile
b.close;
// free the buffer
b.free;

// play it back
(
    Synth          "help-Diskin-2chan"
m = { arg buf; x.addToHeadMsg(nil, [\bufnum,buf.bufnum])};

b = Buffer.cueSoundFile(s,"recordings/diskouttest.aiff", 0, 2, completionMessage:  m);
)
x.free; b.close; b.free; // cleanup
```

Messaging Style:

```
// The same thing done in Messaging Style (less overhead but without the convienence of objects)
// start something to record
s.sendMsg("/s_new", "bubbles", 2003, 1, 1);

// allocate a disk i/o buffer
s.sendMsg("/b_alloc", 0, 65536, 2); // Buffer number is 0

// create an output file for this buffer, leave it open
          "/b_write"     "recordings/diskouttest.aiff"  "aiff"  "int16"

// create the diskout node
s.sendMsg("/s_new", "help-Diskout", 2004, 3, 2003, "bufnum", 0);

s.sendMsg("/n_free", 2004); // stop recording
s.sendMsg("/n_free", 2003); // stop the bubbles
```

```
        "/b_close"      // close the file.
s.sendMsg("/b_free", 0);
```

See **RecordBuf** for recording into a buffer in memory.

ID: 654

Mix sum an array of channels

Mix.new(array)

Mix will mix an array of channels down to a single channel or an array of arrays of channels down to a single array of channels. More information can be found under**MultiChannel**.

```
s.boot;
{ Mix.new([ PinkNoise.ar(0.1), FSinOsc.ar(801, 0.1), LFSaw.ar(40, 0.1)]) }.play
```

*fill(n, function)

A common idiom using Mix is to fill an Array and then mix the results:

```
(
play({
Mix.new( Array.fill(8, { SinOsc.ar(500 + 500.0.rand, 0, 0.05) }) );
}))
```

The ***fill** methods allow this idiom to be written more concisely:

```
(
play({
Mix.fill(8, { SinOsc.ar(500 + 500.0.rand, 0, 0.05) });
}))
```

Note that Mix-ar and Mix-kr in SC2 are equivalent to Mix-new in SC3, and that Mix-arFill and Mix-krFill are equivalent to Mix-fill.

ID: 655

MultiOutUGen

superclass: UGen

A superclass for all UGens with multiple ouptuts.
MultiOutUGen creates the **OutputProxy** ugens needed for the multiple outputs.

initOutputs(argNumChannels)

Create an array of OutputProxies for the outputs.

ID: 656

OutputProxy output place holder

Superclass: UGen

OutputProxy is used by some UGens as a place holder for multiple outputs.
There is no reason for a user to create an OutputProxy directly.

```
(
var out;
// Pan2 uses an OutputProxy for each of its two outputs.
out = Pan2.ar(WhiteNoise.ar, 0.0);
out.postln;
)
```

Methods:

source

Get the UGen that is the source for this OutputProxy.

```
(
var left, right;
// Pan2 uses an OutputProxy for each of its two outputs.
# left, right = Pan2.ar(WhiteNoise.ar, 0.0);
left.source.postln;
)
```

The source method is also defined in Array, so that the source can be obtained this way as well:

```
(
var out;
// Pan2 uses an OutputProxy for each of its two outputs.
out = Pan2.ar(WhiteNoise.ar, 0.0);
out.postln;
out.source.postln;
)
```

ID: 657

PlayBuf sample playback oscillator

Plays back a sample resident in memory.

PlayBuf.ar(numChannels,bufnum,rate,trigger,startPos,loop)

numChannels - number of channels that the buffer will be.
this must be a fixed integer. The architechture of the SynthDef
cannot change after it is compiled.
warning: if you supply a bufnum of a buffer that has a different
numChannels then you have specified to the PlayBuf, it will
fail silently.
bufnum - the index of the buffer to use
rate - 1.0 is the server's sample rate, 2.0 is one octave up, 0.5 is one octave down
-1.0 is backwards normal rate ... etc. Interpolation is cubic.
Note: If the buffer's sample rate is different from the server's, you will need to
multiply the desired playback rate by (file's rate / server's rate). The UGen
BufRateScale.kr(bufnum) returns this factor. See examples below.
BufRateScale should be used in virtually every case.
trigger - a trigger causes a jump to the startPos.
A trigger occurs when a signal changes from $<= 0$ to > 0.
startPos - sample frame to start playback.
loop - 1 means true, 0 means false. this is modulate-able.

```
(
// read a whole sound into memory
s = Server.local;
// note:  not *that* columbia, the first one
    Buffer        "sounds/a11wlk01.wav"   // remember to free the buffer later.
)

SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum))
```

```
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

Note again that the number of channels must be fixed for the SynthDef. It cannot vary depending on which buffer you use.

```
// loop is true
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum), loop:  1.0)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

```
// trigger one shot on each pulse
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
var trig;
trig = Impulse.kr(2.0);
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum), trig, 0, 0)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

```
// trigger one shot on each pulse
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
var trig;
trig = Impulse.kr(XLine.kr(0.1, 100, 30));
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum), trig, 5000, 0)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

```
// mouse control of trigger rate and startpos
SynthDef("help_PlayBuf", { arg out=0, bufnum=0;
```

```
var trig;
trig = Impulse.kr(MouseY.kr(0.5,200,1));
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum), trig, MouseX.kr(0, BufFrames.kr(bufnum)), 1)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);



// accelerating pitch
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
var rate;
rate = XLine.kr(0.1, 100, 60);
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum)*rate, 1.0, 0.0, 1.0)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);



// sine wave control of playback rate.  negative rate plays backwards
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
var rate;
rate = FSinOsc.kr(XLine.kr(0.2, 8, 30), 0, 3, 0.6);
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum) * rate, 1, 0, 1)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);


// zig zag around sound
SynthDef("help_PlayBuf", { arg out=0,bufnum=0;
var rate;
rate = LFNoise2.kr(XLine.kr(1, 20, 60), 2);
Out.ar(out,
PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum) * rate, 1, 0, 1)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

```
b.free;
```

ID: 658

PV_ConformalMap complex plane attack

PV_ConformalMap.ar(buffer, real, imag)

Applies the conformal mapping z -> (z-a)/(1-za*) to the phase vocoder bins z with a given by the real and imag imputs to the UGen.

ie, makes a transformation of the complex plane so the output is full of phase vocoder artifacts but may be musically fun. Usually keep | a| <1 but you can of course try bigger values to make it really noisy. a=0 should give back the input mostly unperturbed.

See http://mathworld.wolfram.com/ConformalMapping.html

buffer - buffer number of buffer to act on, passed in through a chain (see examples below).
real - real part of a.
imag - imaginary part of a.

```
//explore the effect
(
SynthDef "conformer1"
var in, chain;
in = AudioIn.ar(1,0.5);
chain = FFT(0, in);
chain=PV_ConformalMap(chain, MouseX.kr(-1.0,1.0), MouseY.kr(-1.0,1.0));
Out.ar(0, Pan2.ar(IFFT(chain),0));
}).load(s);
)

s.sendMsg("/b_alloc", 0, 1024, 1);
s.sendMsg("/s_new", "conformer1", 2002, 1, 0);
s.sendMsg("/n_free", 2002);



(
SynthDef "conformer2"
```

```
var in, chain, out;
in = Mix.ar(LFSaw.ar(SinOsc.kr(Array.rand(3,0.1,0.5),0,10,[1,1.1,1.5,1.78,2.45,6.7]*220),0,0.3));
chain = FFT(0, in);
chain=PV_ConformalMap(chain, MouseX.kr(0.01,2.0, 'exponential'), MouseY.kr(0.01,10.0, 'exponential'));

out=IFFT(chain);

Out.ar(0, Pan2.ar(CombN.ar(out,0.1,0.1,10,0.5,out),0));
}).load(s);
)



s.sendMsg("/b_alloc", 0, 2048, 1);
s.sendMsg("/s_new", "conformer2", 2002, 1, 0);
s.sendMsg("/n_free", 2002);
```

ID: 659

RecordBuf record or overdub into a Buffer

Records input into a **Buffer**.

RecordBuf.ar(inputArray, bufnum, offset, recLevel, preLevel, run, loop, trigger)

If recLevel is 1.0 and preLevel is 0.0 then the new input overwrites the old data.
If they are both 1.0 then the new data is added to the existing data. (Any other settings are also valid.)

inputArray - an Array of input channels
bufnum - the index of the buffer to use
offset - an offset into the buffer in samples, the default is 0.0.
recLevel - value to multiply by input before mixing with existing data. Default is 1.0.
preLevel - value to multiply to existing data in buffer before mixing with input. Default is 0.0.
run - If zero, then recording stops, otherwise recording proceeds. Default is 1.
loop - If zero then don't loop, otherwise do. This is modulate-able. Default is 1.
trigger - a trigger causes a jump to the start of the Buffer.
A trigger occurs when a signal changes from $<= 0$ to > 0.

```
// Execute the following in order
(
// allocate a Buffer
s = Server.local;
    Buffer                          // a four second 1 channel Buffer
)

// record for four seconds
(
SynthDef("help-RecordBuf",{ arg out=0,bufnum=0;
var formant;
 // XLine will free the Synth when done
formant = Formant.ar(XLine.kr(400,1000, 4, doneAction:  2), 2000, 800, 0.125);
RecordBuf.ar(formant, bufnum);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
```

```
)

// play it back
(
SynthDef "help-RecordBuf play"   arg
var playbuf;
playbuf = PlayBuf.ar(1,bufnum);
 FreeSelfWhenDone              // frees the synth when the PlayBuf is finished
Out.ar(out, playbuf);
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

// overdub
(
SynthDef "help-RecordBuf overdub"   arg
var formant;
 // XLine will free the Synth when done
formant = Formant.ar(XLine.kr(200, 1000, 4, doneAction:  2), 2000, 800, 0.125);
RecordBuf.ar(formant, bufnum, 0, 0.5, 0.5); // mixes equally with existing data
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

// play back the overdubbed version
Synth.new("help-RecordBuf play", [\out, 0, \bufnum, b.bufnum], s);

// write the contents of the buffer to a file
        "recordings/RecordBuf-test.aiff"                'int16'

b.close; b.free; // cleanup
```

Note again that the number of channels must be fixed for the SynthDef, it cannot vary depending on which buffer you use.

ID: 660

Select

The output is selected from an array of inputs.

Select.ar(which,array)
Select.kr(which,array)

```
(

SynthDef "help-Select"   arg

var a,cycle;
a = [
SinOsc.ar,
Saw.ar,
Pulse.ar
];
cycle = a.size  * 0.5;
Out.ar(out,
Select.ar(LFSaw.kr(1.0,0.0,cycle,cycle),a) * 0.2
)
}).play;

)
```

Note: all the ugens are continously running. This may not be the most efficient way if each input is cpu-expensive.

Here used as a sequencer:

```
(
SynthDef "help-Select-2"   arg

var a,s,cycle;
a = Array.fill(32,{ rrand(30,80) }).midicps;
a.postln;
cycle = a.size  * 0.5;
```

```
s = Saw.ar(
Select.kr(
LFSaw.kr(1.0,0.0,cycle,cycle),
a
),
0.2
);
Out.ar(out,s )
}).play;
)
```

Note that the array is fixed at the time of writing the SynthDef, and the whole array is embedded in the SynthDef file itself. For small arrays this is more efficient than reading from a buffer.

ID: 661

SendTrig

On receiving a trigger (a non-positive to positive transition), send a trigger message from the server back to the client.

SendTrig.kr(input, id, value)
input - the trigger
id - an integer that will be passed with the trigger message.
this is useful if you have more than one SendTrig in a SynthDef
value - a UGen or float that will be polled at the time of trigger,
and its value passed with the trigger message

The trigger message sent back to the client is this:
/tr a trigger message
int - node ID
int - trigger ID
float - trigger value

This command is the mechanism that synths can use to trigger events in clients.
The node ID is the node that is sending the trigger. The trigger ID and value are determined by inputs to the SendTrig unit generator which is the originator of this message.

```
s = Server.local;
s.boot;

SynthDef("help-SendTrig",{
SendTrig.kr(Dust.kr(1.0),0,0.9);
}).send(s);

// register to receive this message
OSCresponder(s.addr,'/tr',{ arg time,responder,msg;
[time,responder,msg].postln;
}).add;

Synth("help-SendTrig");
```

ID: 662

Synth Controlling UGens

the UGens listed here can affect a node's state from within the synth, they can free or pause nodes, for example.

Done
FreeSelf
PauseSelf
FreeSelfWhenDone
PauseSelfWhenDone
Pause
Free

UGens with a doneAction

EnvGen
Linen
Line
XLine
DetectSilence
Duty
TDuty

See **[UGen-doneActions]** for more detail on done actions.

ID: 663

TGrains buffer granulator

Triggers generate grains from a buffer. Each grain has a Hanning envelope (sin^2(x) for x from 0 to pi) and is panned between two channels of multiple outputs.

TGrains.ar(numChannels, trigger, bufnum, rate, centerPos, dur, pan, amp, interp)

numChannels - number of output channels.
trigger - at each trigger, the following arguments are sampled and used as the arguments of a new grain.
A trigger occurs when a signal changes from <= 0 to > 0.
If the trigger is audio rate then the grains will start with sample accuracy.
bufnum - the index of the buffer to use. It must be a one channel (mono) buffer.
rate - 1.0 is normal, 2.0 is one octave up, 0.5 is one octave down -1.0 is backwards normal rate ... etc.
Unlike PlayBuf, the rate is multiplied by BufRate, so you needn't do that yourself.
centerPos - the position in the buffer in seconds at which the grain envelope will reach maximum amplitude.
dur - duration of the grain in seconds.
pan - a value from -1 to 1. Determines where to pan the output in the same manner as PanAz.
amp - amplitude of the grain.
interp - 1,2,or 4. Determines whether the grain uses (1) no interpolation, (2) linear interpolation,
or (4) cubic interpolation.

```
(
s = Server.internal;
Server.default = s;
s.boot;
)
```

```
        \b_allocRead      "sounds/a11wlk01.wav"
```

```
(
{
var b = 10, trate, dur;
trate = MouseY.kr(2,200,1);
dur = 4 / trate;
TGrains.ar(2, Impulse.ar(trate), b, 1, MouseX.kr(0,BufDur.kr(b)), dur, 0, 0.1, 2);
}.scope(zoom:  4);
)

(
{
var b = 10, trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 12 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b, 1, pos, dur, pan, 0.1);
}.scope(zoom:  4);
)

// 4 channels
(
{
var b = 10, trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 12 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(4, clk, b, 1, pos, dur, pan, 0.1);
}.scope(4, zoom:  4);
)

(
{
var b = 10, trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 4 / trate;
clk = Dust.kr(trate);
```

```
pos = MouseX.kr(0,BufDur.kr(b)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b, 1, pos, dur, pan, 0.1);
}.scope(zoom:  4);
)



(
{
var b = 10, trate, dur, clk, pos, pan;
trate = LinExp.kr(LFTri.kr(MouseY.kr(0.1,2,1)),-1,1,8,120);
dur = 12 / trate;
clk = Impulse.ar(trate);
pos = MouseX.kr(0,BufDur.kr(b));
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b, 1, pos, dur, pan, 0.1);
}.scope(zoom:  4);
)



(
{
var b = 10, trate, dur, clk, pos, pan;
trate = 12;
dur = MouseY.kr(0.2,24,1) / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b, 1, pos, dur, pan, 0.1);
}.scope(zoom:  4);
)

(
{
var b = 10, trate, dur, clk, pos, pan;
trate = 100;
dur = 8 / trate;
clk = Impulse.kr(trate);
pos = Integrator.kr(BrownNoise.kr(0.001));
```

```
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b, 1, pos, dur, pan, 0.1);
}.scope(zoom:  4);
)

(
{
var b = 10, trate, dur, clk, pos, pan;
trate = MouseY.kr(1,400,1);
dur = 8 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b));
pan = WhiteNoise.kr(0.8);
TGrains.ar(2, clk, b, 2 ** WhiteNoise.kr(2), pos, dur, pan, 0.1);
}.scope(zoom:  4);
)



(
{
var b = 10, trate, dur;
trate = MouseY.kr(2,120,1);
dur = 1.2 / trate;
TGrains.ar(2, Impulse.ar(trate), b, (1.2 ** WhiteNoise.kr(3).round(1)), MouseX.kr(0,BufDur.kr(b)), dur,
WhiteNoise.kr(0.6), 0.1);
}.scope(zoom:  4);
)
```

ID: 664

SIGNAL PROCESSING in SuperCollider

1. A Tour of available Unit Generators.

SuperCollider has over 250 unit generators.
If you count the unary and binary operators, there are over 300.
This tour covers many, but not all of them.

categories of unit generators:
sources: periodic, aperiodic
filters
distortion
panning
delays and buffer ugens
control: envelopes, triggers, counters, gates, lags, decays
spectral

2. Techniques

broadening a sound:
decorrelation, beat frequencies, delays.
series and parallel structures.

```
(
s = Server.internal;
Server.default = s;
s.boot;
)
```

Periodic Sources: Oscillators.

LF - "Low Frequency" Unit Generators.

LFPar, LFCub, LFTri, Impulse, LFSaw, LFPulse, VarSaw, SyncSaw

geometric waveforms, not band limited.

will cause aliasing at higher frequencies.

LFPar, LFCub, LFTri, LFSaw, Impulse
arguments: frequency, phase, mul, add

```
// parabolic approximation of sine
{ LFPar.ar(LFPar.kr(LFPar.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFPar.ar(LFPar.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFPar.ar(800,0,0.1) }.scope(1, zoom:  4);
// since it is not band limited, there are aliasing artifacts
{ LFPar.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);

// cubic approximation of sine
{ LFCub.ar(LFCub.kr(LFCub.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFCub.ar(LFCub.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFCub.ar(800,0,0.1) }.scope(1, zoom:  4);
{ LFCub.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);


{ LFTri.ar(LFTri.kr(LFTri.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFTri.ar(LFTri.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFTri.ar(800,0,0.1) }.scope(1, zoom:  4);
{ LFTri.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);


{ LFSaw.ar(LFSaw.kr(LFSaw.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFSaw.ar(LFSaw.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ LFSaw.ar(100,0,0.1) }.scope(1, zoom:  4);
{ LFSaw.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);


{ Impulse.ar(LFTri.kr(LFTri.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ Impulse.ar(LFTri.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ Impulse.ar(100,0,0.1) }.scope(1, zoom:  4);
{ Impulse.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);
```

LFPulse, VarSaw
arguments: frequency, phase, width, mul, add

```
{ LFPulse.ar(LFPulse.kr(LFPulse.kr(0.2,0,0.5,8,10),0,0.5, 400,800),0,0.5,0.1) }.scope(1, zoom:  4);
{ LFPulse.ar(LFPulse.kr(3, 0, 0.3, 200, 200), 0, 0.2, 0.1) }.scope(1, zoom:  4);
{ LFPulse.ar(XLine.kr(100,15000,6),0,0.5,0.1) }.scope(1, zoom:  4);

// pulse width modulation
{ LFPulse.ar(100,0,MouseY.kr(0,1),0.1) }.scope(1, zoom:  4);
{ LFPulse.ar(100,0,LFTri.kr(0.2,0,0.5,0.5),0.1) }.scope(1, zoom:  4);



{ VarSaw.ar(VarSaw.kr(VarSaw.kr(0.2,0,0.2,8,10),0,0.2, 400,800),0,0.2,0.1) }.scope(1, zoom:  4);
{ VarSaw.ar(VarSaw.kr(0.2, 0, 0.2, 400,800),0,0.2,0.1) }.scope(1, zoom:  4);
{ VarSaw.ar(XLine.kr(100,15000,6),0,0.2,0.1) }.scope(1, zoom:  4);

// pulse width modulation
{ VarSaw.ar(100,0,MouseY.kr(0,1),0.1) }.scope(1, zoom:  4);
{ VarSaw.ar(100,0,LFTri.kr(0.2,0,0.5,0.5),0.1) }.scope(1, zoom:  4);
```

SyncSaw
arguments: syncFreq, sawFreq, mul, add

```
{ SyncSaw.ar(100, MouseX.kr(100, 1000), 0.1) }.scope(1, zoom:  4);
{ SyncSaw.ar(100, Line.kr(100, 800, 12), 0.1) }.scope(1, zoom:  4);
```

Band Limited Oscillators

SinOsc, FSinOsc, Blip, Saw, Pulse
will not alias.

SinOsc, FSinOsc
arguments: frequency, phase, mul, add

```
{ SinOsc.ar(SinOsc.kr(SinOsc.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ SinOsc.ar(SinOsc.kr(0.2, 0, 400,800),0,0.1) }.scope(1, zoom:  4);
{ SinOsc.ar(800,0,0.1) }.scope(1, zoom:  4);
{ SinOsc.ar(XLine.kr(100,15000,6),0,0.1) }.scope(1, zoom:  4);

{ FSinOsc.ar(800,0,0.1) }.scope(1, zoom:  4);
```

```
// FSinOsc should not be frequency modulated.
// Since it is based on a filter at the edge of stability, it will blow up:
{ FSinOsc.ar(FSinOsc.kr(FSinOsc.kr(0.2,0,8,10),0, 400,800),0,0.1) }.scope(1, zoom:  4);
```

Blip
arguments: frequency, numHarmonics, mul, add

```
{ Blip.ar(XLine.kr(20000,200,6),100,0.2) }.scope(1);
{ Blip.ar(XLine.kr(100,15000,6),100,0.2) }.scope(1); // no aliasing
// modulate number of harmonics
{ Blip.ar(200,Line.kr(1,100,20),0.2) }.scope(1);
```

Saw
arguments: frequency, mul, add

```
{ Saw.ar(XLine.kr(20000,200,6),0.2) }.scope(1);
{ Saw.ar(XLine.kr(100,15000,6),0.2) }.scope(1); // no aliasing
```

Pulse
arguments: frequency, width, mul, add

```
{ Pulse.ar(XLine.kr(20000,200,6),0.3,0.2) }.scope(1);
{ Pulse.ar(XLine.kr(100,15000,6),0.3,0.2) }.scope(1); // no aliasing

// modulate pulse width
{ Pulse.ar(200, Line.kr(0.01,0.99,8), 0.2) }.scope(1);

// two band limited square waves thru a resonant low pass filter
{ RLPF.ar(Pulse.ar([100,250],0.5,0.1), XLine.kr(8000,400,5), 0.05) }.scope(1);
```

Klang - sine oscillator bank
arguments: ‘[frequencies, amplitudes, phases], mul, add

```
{ Klang.ar(‘[ [800, 1000, 1200],[0.3, 0.3, 0.3],[pi,pi,pi]], 1, 0) * 0.4}.scope(1);

{ Klang.ar(‘[ {exprand(400, 2000)}.dup(16), nil, nil ], 1, 0) * 0.04 }.scope(1);
```

Table Oscillators

Osc, COsc, VOsc, VOsc3
Use a buffer allocated on the server.

Osc
arguments: buffer number, frequency, phase, mul, add.

```
(
b = Buffer.alloc(s, 2048, 1, bufnum:  80);
b.sine1(1.0/(1..6), true, true, true);
)

{ Osc.ar(80, 100, 0, 0.1) }.scope(1, zoom:4);

b.sine1(1.0/(1..12));
b.sine1(1.0/(1..24));
b.sine1(1.0/(1..32));
b.sine1([1.0/(1,3..12), 0].flop.flat.postln);
b.sine1([1.0/(1,3..32).squared, 0].flop.flat.postln);
b.sine1((1.dup(4) ++ 0.dup(8)).scramble.postln);
b.sine1((1.dup(4) ++ 0.dup(8)).scramble.postln);
b.sine1((1.dup(4) ++ 0.dup(8)).scramble.postln);
b.sine1((1.dup(4) ++ 0.dup(8)).scramble.postln);
b.sine1({1.0.rand2.cubed}.dup(8).round(1e-3).postln);
b.sine1({1.0.rand2.cubed}.dup(12).round(1e-3).postln);
b.sine1({1.0.rand2.cubed}.dup(16).round(1e-3).postln);
b.sine1({1.0.rand2.cubed}.dup(24).round(1e-3).postln);
```

COsc - two oscillators, detuned
arguments: buffer number, frequency, beat frequency, mul, add.

```
b.sine1(1.0/(1..6), true, true, true);

{ COsc.ar(80, 100, 1, 0.1) }.scope(1, zoom:4);
// change buffer as above.
```

VOsc - multiple wave table crossfade oscillators
arguments: buffer number, frequency, phase, mul, add.

```
(
// allocate tables 80 to 87
8.do {| i|  s.sendMsg(\b_alloc, 80+i, 1024); };
)

(
// fill tables 80 to 87
8.do({| i|
var n, a;
 // generate array of harmonic amplitudes
 n = (i+1)**2; // num harmonics for each table:  [1,4,9,16,25,36,49,64]
a = {| j|  ((n-j)/n).squared }.dup(n);
 // fill table
s.listSendMsg([\b_gen, 80+i, \sine1, 7] ++ a);
});
)

{ VOsc.ar(MouseX.kr(80,87), 120, 0, 0.3) }.scope(1, zoom:4);

(
// allocate and fill tables 80 to 87
8.do({| i|
 // generate array of harmonic amplitudes
a = {1.0.rand2.cubed }.dup((i+1)*4);
 // fill table
s.listSendMsg([\b_gen, 80+i, \sine1, 7] ++ a);
});
)
```

VOsc3 - three VOscs summed.
arguments: buffer number, freq1, freq2, freq3, beat frequency, mul, add.

```
// chorusing
{ VOsc3.ar(MouseX.kr(80,87), 120, 121.04, 119.37, 0.2) }.scope(1, zoom:4);

// chords
{ VOsc3.ar(MouseX.kr(80,87), 120, 151.13, 179.42, 0.2) }.scope(1, zoom:4);
```

Aperiodic Sources: Noise.

LF "Low Frequency" Noise Generators.

LFNoise0, LFNoise1, LFNoise2, LFClipNoise
arguments: frequency, mul, add

```
{ LFClipNoise.ar(MouseX.kr(200, 10000, 1), 0.125) }.scope(1);
{ LFNoise0.ar(MouseX.kr(200, 10000, 1), 0.25) }.scope(1);
{ LFNoise1.ar(MouseX.kr(200, 10000, 1), 0.25) }.scope(1);
{ LFNoise2.ar(MouseX.kr(200, 10000, 1), 0.25) }.scope(1);

// used as controls
{ LFPar.ar(LFClipNoise.kr(MouseX.kr(0.5, 64, 1), 200, 400), 0, 0.2) }.scope(1, zoom:8);
{ LFPar.ar(LFNoise0.kr(MouseX.kr(0.5, 64, 1), 200, 400), 0, 0.2) }.scope(1, zoom:8);
{ LFPar.ar(LFNoise1.kr(MouseX.kr(0.5, 64, 1), 200, 400), 0, 0.2) }.scope(1, zoom:8);
{ LFPar.ar(LFNoise2.kr(MouseX.kr(0.5, 64, 1), 200, 400), 0, 0.2) }.scope(1, zoom:8);
```

Broad Spectrum Noise Generators

ClipNoise, WhiteNoise, PinkNoise, BrownNoise, GrayNoise
arguments: mul, add

```
{ ClipNoise.ar(0.2) }.scope(1);
{ WhiteNoise.ar(0.2) }.scope(1);
{ PinkNoise.ar(0.4) }.scope(1);
{ BrownNoise.ar(0.2) }.scope(1);
{ GrayNoise.ar(0.2) }.scope(1);
```

Impulse Noise Generators

Dust, Dust2
arguments: density, mul, add

```
{ Dust.ar(MouseX.kr(1,10000,1), 0.4) }.scope(1, zoom:4);
{ Dust2.ar(MouseX.kr(1,10000,1), 0.4) }.scope(1, zoom:4);
```

Chaotic Noise Generators

Crackle
arguments: chaosParam, mul, add

```
{ Crackle.ar(MouseX.kr(1,2), 0.5) }.scope(1);
```

Filters

Low Pass, High Pass

LPF, HPF - 12 dB / octave
arguments: in, freq, mul, add

```
{ LPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ HPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ LPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ HPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
```

Band Pass, Band Cut

BPF, BRF - 12 dB / octave
arguments: in, freq, rq, mul, add
rq is the reciprocal of the Q of the filter,
or in other words: the bandwidth in Hertz = rq * freq.

```
{ BPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.4, 0.4) }.scope(1);
{ BRF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.4, 0.2) }.scope(1);
{ BPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.4, 0.4) }.scope(1);
{ BRF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.4, 0.2) }.scope(1);

// modulating the bandwidth
{ BPF.ar(WhiteNoise.ar, 3000, MouseX.kr(0.01,0.7,1), 0.4) }.scope(1);
```

Resonant Low Pass, High Pass, Band Pass

RLPF, RHPF - 12 dB / octave

arguments: in, freq, rq, mul, add

```
{ RLPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RHPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RLPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RHPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
```

Resonz - resonant band pass filter with uniform amplitude
arguments: in, freq, rq, mul, add

```
// modulate frequency
{ Resonz.ar(WhiteNoise.ar(0.5), XLine.kr(1000,8000,10), 0.05) }.scope(1);

// modulate bandwidth
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(1, 0.001, 8)) }.scope(1);

// modulate bandwidth opposite direction
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(0.001, 1, 8)) }.scope(1);
```

Ringz - ringing filter.
Internally it is the same as Resonz but the bandwidth is expressed as a ring time.
arguments: in, frequency, ring time, mul, add

```
{ Ringz.ar(Dust.ar(3, 0.3), 2000, 2) }.scope(1, zoom:4);

{ Ringz.ar(WhiteNoise.ar(0.005), 2000, 0.5) }.scope(1);

// modulate frequency
{ Ringz.ar(WhiteNoise.ar(0.005), XLine.kr(100,3000,10), 0.5) }.scope(1, zoom:4);

{ Ringz.ar(Impulse.ar(6, 0, 0.3),  XLine.kr(100,3000,10), 0.5) }.scope(1, zoom:4);

// modulate ring time
{ Ringz.ar(Impulse.ar(6, 0, 0.3), 2000, XLine.kr(0.04, 4, 8)) }.scope(1, zoom:4);
```

Simpler Filters
6 dB / octave

```
{ OnePole.ar(WhiteNoise.ar(0.5), MouseX.kr(-0.99, 0.99)) }.scope(1);
{ OneZero.ar(WhiteNoise.ar(0.5), MouseX.kr(-0.49, 0.49)) }.scope(1);
```

NonLinear Filters
Median, Slew

```
// a signal with impulse noise.
{ Saw.ar(500, 0.1) + Dust2.ar(100, 0.9) }.scope(1);
// after applying median filter
{ Median.ar(3, Saw.ar(500, 0.1) + Dust2.ar(100, 0.9)) }.scope(1);

// a signal with impulse noise.
{ Saw.ar(500, 0.1) + Dust2.ar(100, 0.9) }.scope(1);
// after applying slew rate limiter
{ Slew.ar(Saw.ar(500, 0.1) + Dust2.ar(100, 0.9),1000,1000) }.scope(1);
```

Formant Filter
Formlet - A filter whose impulse response is similar to a FOF grain.

```
{ Formlet.ar(Impulse.ar(MouseX.kr(2,300,1), 0, 0.4), 800, 0.01, 0.1) }.scope(1, zoom:4);
```

Klank - resonant filter bank
arguments: ‘[frequencies, amplitudes, ring times], mul, add

```
{ Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], Impulse.ar(2, 0, 0.1)) }.play;

{ Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], Dust.ar(8, 0.1)) }.play;

{ Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar(0.007)) }.play;

{ Klank.ar(‘[ {exprand(200, 4000)}.dup(12), nil, nil ], PinkNoise.ar(0.007)) }.scope(1);

{ Klank.ar(‘[ (1..13)*200, 1/(1..13), nil ], PinkNoise.ar(0.01)) }.scope(1);

{ Klank.ar(‘[ (1,3..13)*200, 1/(1,3..13), nil ], PinkNoise.ar(0.01)) }.scope(1);
```

Distortion

abs, max, squared, cubed

```
{ SinOsc.ar(300, 0, 0.2) }.scope(1);
{ SinOsc.ar(300, 0, 0.2).abs }.scope(1);
{ SinOsc.ar(300, 0, 0.2).max(0) }.scope(1);
{ SinOsc.ar(300, 0).squared * 0.2 }.scope(1);
{ SinOsc.ar(300, 0).cubed * 0.2 }.scope(1);
```

distort, softclip, clip2, fold2, wrap2,

```
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).distort * 0.2 }.scope(1);
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).softclip * 0.2 }.scope(1);
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).clip2(1) * 0.2 }.scope(1);
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).fold2(1) * 0.2 }.scope(1);
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).wrap2(1) * 0.2 }.scope(1);
{ SinOsc.ar(300, 0, MouseX.kr(0.1,80,1)).wrap2(1) * 0.2 }.scope(1);
```

scaleneg

```
{ SinOsc.ar(200, 0, 0.2).scaleneg(MouseX.kr(-1,1)) }.scope(1);
```

waveshaping by phase modulating a 0 Hz sine oscillator (currently there is a limit of 8pi)

```
(
{
var in;
in = SinOsc.ar(300, 0, MouseX.kr(0.1,8pi,1));
 SinOsc                 // 0 Hz sine oscillator
}.scope(1);
)
```

Shaper
input is used to look up a value in a table.
Chebyshev polynomials are typically used to fill the table.

<img src="chebyshevpolynomials.gif" alt="chebyshevpolynomials.gif">

```
s.sendMsg(\b_alloc, 80, 1024); // allocate table
// fill with chebyshevs
s.listSendMsg([\b_gen, 80, \cheby, 7] ++ {1.0.rand2.squared}.dup(6));

{ Shaper.ar(80, SinOsc.ar(600, 0, MouseX.kr(0,1))) * 0.3; }.scope(1);

s.listSendMsg([\b_gen, 80, \cheby, 7] ++ {1.0.rand2.squared}.dup(6));
s.listSendMsg([\b_gen, 80, \cheby, 7] ++ {1.0.rand2.squared}.dup(6));
```

Panning

```
(
s = Server.internal;
Server.default = s;
s.quit;
s.options.numOutputBusChannels = 8;
s.options.numInputBusChannels = 8;
s.boot;
)
```

Pan2 - equal power stereo pan a mono source
arguments: in, pan position, level
pan controls typically range from -1 to +1

```
{ Pan2.ar(BrownNoise.ar, MouseX.kr(-1,1), 0.3) }.scope(2);
{ Pan2.ar(BrownNoise.ar, SinOsc.kr(0.2), 0.3) }.scope(2);
```

LinPan2 - linear pan a mono source (not equal power)
arguments: in, pan position, level

```
{ LinPan2.ar(BrownNoise.ar, MouseX.kr(-1,1), 0.3) }.scope(2);
{ LinPan2.ar(BrownNoise.ar, SinOsc.kr(0.2), 0.3) }.scope(2);
```

Balance2 - balance a stereo source
arguments: left in, right in, pan position, level

```
{ Balance2.ar(BrownNoise.ar, BrownNoise.ar, MouseX.kr(-1,1), 0.3) }.scope(2);
```

Pan4 - equal power quad panner

```
{ Pan4.ar(BrownNoise.ar, MouseX.kr(-1,1), MouseY.kr(1,-1), 0.3) }.scope(4);
```

PanAz - azimuth panner to any number of channels arguments: num channels, in, pan position, level, width

```
{ PanAz.ar(5, BrownNoise.ar, MouseX.kr(-1,1), 0.3, 2) }.scope(5);

// change width to 3
{ PanAz.ar(5, BrownNoise.ar, MouseX.kr(-1,1), 0.3, 3) }.scope(5);
```

XFade2 - equal power cross fade between two inputs arguments: in1, in2, crossfade, level

```
{ XFade2.ar(BrownNoise.ar, SinOsc.ar(500), MouseX.kr(-1,1), 0.3) }.scope(1);
```

PanB2 and DecodeB2 - 2D ambisonics panner and decoder

```
(
{
var w, x, y, p, lf, rf, rr, lr;

p = BrownNoise.ar; // source

 // B-format encode
#w, x, y = PanB2.ar(p, MouseX.kr(-1,1), 0.3);

 // B-format decode to quad.  outputs in clockwise order
#lf, rf, rr, lr = DecodeB2.ar(4, w, x, y);

 [lf, rf, lr, rr] // reorder to my speaker arrangement:  Lf Rf Lr Rr
}.scope(4);
)
```

Rotate2 - rotate a sound field of ambisonic or even stereo sound.

```
(
{
 // rotation of stereo sound via mouse
var x, y;
x = Mix.fill(4, { LFSaw.ar(200 + 2.0.rand2, 0, 0.1) }); // left in
y = WhiteNoise.ar * LFPulse.kr(3,0,0.7,0.2); // right in
#x, y = Rotate2.ar(x, y, MouseX.kr(0,2));
[x,y]
}.scope(2);
)
```

Delays and Buffer UGens

DelayN, DelayL, DelayC
simple delays
N - no interpolation
L - linear interpolation
C - cubic interpolation
arguments: in, maximum delay time, current delay time, mul, add

```
(
// Dust randomly triggers Decay to create an exponential
// decay envelope for the WhiteNoise input source
{
z = Decay.ar(Dust.ar(1,0.5), 0.3, WhiteNoise.ar);
DelayN                          // input is mixed with delay via the add input
}.scope(1, zoom:  4)
)

(
{
z = Decay.ar(Impulse.ar(2,0,0.4), 0.3, WhiteNoise.ar);
DelayL                                      // input is mixed with delay via the add input
}.scope(1, zoom:  4)
)
```

CombN, CombL, CombC

feedback delays
arguments: in, maximum delay time, current delay time, echo decay time, mul, add

```
// used as an echo.
{ CombN.ar(Decay.ar(Dust.ar(1,0.5), 0.2, WhiteNoise.ar), 0.2, 0.2, 3) }.scope(1, zoom:4);

// Comb used as a resonator.  The resonant fundamental is equal to
// reciprocal of the delay time.
{ CombN.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.scope(1);

{ CombL.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.scope(1);

{ CombC.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), 0.2) }.scope(1);

// with negative feedback:
{ CombN.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.scope(1);

{ CombL.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.scope(1);

{ CombC.ar(WhiteNoise.ar(0.02), 0.01, XLine.kr(0.0001, 0.01, 20), -0.2) }.scope(1);

{ CombC.ar(Decay.ar(Dust.ar(1,0.1), 0.2, WhiteNoise.ar), 1/100, 1/100, 3) }.play;
{ CombC.ar(Decay.ar(Dust.ar(1,0.1), 0.2, WhiteNoise.ar), 1/200, 1/200, 3) }.play;
{ CombC.ar(Decay.ar(Dust.ar(1,0.1), 0.2, WhiteNoise.ar), 1/300, 1/300, 3) }.play;
{ CombC.ar(Decay.ar(Dust.ar(1,0.1), 0.2, WhiteNoise.ar), 1/400, 1/400, 3) }.scope(1, zoom:4);
```

AllpassN, AllpassL, AllpassC
allpass delay
arguments: in, maximum delay time, current delay time, echo decay time, mul, add

```
(
{
var z;
z = Decay.ar(Dust.ar(1,0.5), 0.1, WhiteNoise.ar);
8.do { z = AllpassL.ar(z, 0.04, 0.04.rand, 2) };
```

```
z
}.scope(1);
)
```

PlayBuf

buffer playback

arguments: numChannels, buffer number, rate, trigger, start pos, loop

```
// read sound
    Buffer         "sounds/a11wlk01.wav"

{ SinOsc.ar(800 + (700 * PlayBuf.ar(1,b.bufnum, BufRateScale.kr(b.bufnum),  loop:1)),0,0.3) }.scope(1);


// loop is true
{ PlayBuf.ar(1,b.bufnum, BufRateScale.kr(b.bufnum), loop:1) }.scope(1);


// trigger one shot on each pulse
(
{
var trig;
trig = Impulse.kr(2.0);
PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum),trig,0,0);
}.scope(1);
)

// trigger one shot on each pulse
(
{
var trig;
trig = Impulse.kr(XLine.kr(0.1,100,30));
PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum),trig,5000,0);
}.scope(1);
)

// mouse control of trigger rate and startpos
(
```

```
{
var trig;
trig = Impulse.kr(MouseY.kr(0.5,200,1));
PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum),trig,MouseX.kr(0,BufFrames.kr(b.bufnum)),1)
}.scope(1);
)

// accelerating pitch
(
{
var rate;
rate = XLine.kr(0.1,100,60);
PlayBuf.ar(1, b.bufnum, rate, 1.0,0.0, 1.0)
}.scope(1);
)

// sine wave control of playback rate.  negative rate plays backwards
(
{
var rate;
rate = FSinOsc.kr(XLine.kr(0.2,8,30), 0, 3, 0.6);
PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum)*rate,1,0,1)
}.scope(1);
)

// zig zag around sound
(
{
var rate;
rate = LFNoise2.kr(XLine.kr(1,20,60), 2);
PlayBuf.ar(1,b.bufnum,BufRateScale.kr(b.bufnum) * rate,1,0,1)
}.scope(1);
)


// free sound
b.free;
```

TGrains

granulation of a buffer
arguments: numChannels, trigger, buffer number, rate, center pos, dur, pan, amp, interpolation

```
// read sound
    Buffer         "sounds/a11wlk01.wav"


(
{
var trate, dur;
trate = MouseY.kr(2,200,1);
dur = 4 / trate;
TGrains.ar(2, Impulse.ar(trate), b.bufnum, 1, MouseX.kr(0,BufDur.kr(b.bufnum)), dur, 0, 0.1, 2);
}.scope(2, zoom:  4);
)


(
{
var trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 12 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)

// 4 channels
(
{
var trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 12 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(4, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(4, zoom:  4);
)
```

```
(
{
var trate, dur, clk, pos, pan;
trate = MouseY.kr(8,120,1);
dur = 4 / trate;
clk = Dust.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)



(
{
var trate, dur, clk, pos, pan;
trate = LinExp.kr(LFTri.kr(MouseY.kr(0.1,2,1)),-1,1,8,120);
dur = 12 / trate;
clk = Impulse.ar(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum));
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)



(
{
var trate, dur, clk, pos, pan;
trate = 12;
dur = MouseY.kr(0.2,24,1) / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum)) + TRand.kr(0, 0.01, clk);
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)
```

```
(
{
var trate, dur, clk, pos, pan;
trate = 100;
dur = 8 / trate;
clk = Impulse.kr(trate);
pos = Integrator.kr(BrownNoise.kr(0.001));
pan = WhiteNoise.kr(0.6);
TGrains.ar(2, clk, b.bufnum, 1, pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)

(
{
var trate, dur, clk, pos, pan;
trate = MouseY.kr(1,400,1);
dur = 8 / trate;
clk = Impulse.kr(trate);
pos = MouseX.kr(0,BufDur.kr(b.bufnum));
pan = WhiteNoise.kr(0.8);
TGrains.ar(2, clk, b.bufnum, 2 ** WhiteNoise.kr(2), pos, dur, pan, 0.1);
}.scope(2, zoom:  4);
)




(
{
var trate, dur;
trate = MouseY.kr(2,120,1);
dur = 1.2 / trate;
TGrains.ar(2, Impulse.ar(trate), b.bufnum, (1.2 ** WhiteNoise.kr(3).round(1)), MouseX.kr(0,BufDur.kr(b.bufnum)),
dur, WhiteNoise.kr(0.6), 0.1);
}.scope(2, zoom:  4);
)

// free sound
b.free;
```

Control

Filters for Controls

Decay
triggered exponential decay
arguments: in, decay time, mul, add

```
{ WhiteNoise.ar * Decay.ar(Impulse.ar(1), 0.9, 0.2) }.scope(1, zoom:4);
{ WhiteNoise.ar * Decay.ar(Dust.ar(3), 0.9, 0.2) }.scope(1, zoom:4);
{ SinOsc.ar(Decay.ar(Dust.ar(4), 0.5, 1000, 400), 0, 0.2) }.scope(1, zoom:4);
```

Decay2
triggered exponential attack and exponential decay
arguments: trigger, attack time, decay time, mul, add

```
{ WhiteNoise.ar * Decay2.ar(Impulse.ar(1), 0.2, 0.9, 0.2) }.scope(1, zoom:4);
{ WhiteNoise.ar * Decay2.ar(Dust.ar(3), 0.2, 0.9, 0.2) }.scope(1, zoom:4);
```

Lag
arguments: trigger, duration

```
{ SinOsc.ar(Lag.ar(LFPulse.ar(2,0,0.5,800,400), MouseX.kr(0,0.5)), 0, 0.2) }.scope(1, zoom:4);
```

Integrator
leaky integrator

```
{ SinOsc.ar(Integrator.ar(Dust2.ar(8), 0.99999, 200, 800), 0, 0.2) }.scope(1)
```

Triggers

Trig, Trig1
timed duration gate
arguments: trigger, duration

```
// amplitude determined by amplitude of trigger
{ Trig.ar(Dust.ar(2), 0.2) * FSinOsc.ar(800, 0, 0.4) }.scope(1, zoom:4);
// amplitude always the same.
{ Trig1.ar(Dust.ar(2), 0.2) * FSinOsc.ar(800, 0, 0.4) }.scope(1, zoom:4)
```

TDelay
delays a trigger. only delays one pending trigger at a time.
arguments: trigger, delay time

```
(
{
var trig;
trig = Dust.ar(2);
[(Trig1.ar(trig, 0.05) * FSinOsc.ar(600, 0, 0.2)),
(Trig1.ar(TDelay.ar(trig, 0.1), 0.05) * FSinOsc.ar(800, 0, 0.2))]
}.scope(2, zoom:4);
)
```

Latch
sample and hold
arguments: in, trigger

```
{ Blip.ar(Latch.ar(WhiteNoise.ar, Impulse.ar(9)) * 400 + 500, 4, 0.2) }.play;
{ Blip.ar(Latch.ar(SinOsc.ar(0.3), Impulse.ar(9)) * 400 + 500, 4, 0.2) }.play;
```

Gate
pass or hold
arguments: in, trigger

```
{ Blip.ar(Gate.ar(LFNoise2.ar(40), LFPulse.ar(1)) * 400 + 500, 4, 0.2) }.scope(1, zoom:4);
```

PulseCount
count triggers
arguments: trigger, reset

```
(
{
SinOsc
```

```
PulseCount.ar(Impulse.ar(10), Impulse.ar(0.4)) * 200,
0, 0.05
)
}.scope(2, zoom:4);
)
```

PulseDivider
arguments: trigger, div, start

```
(
{
var p, a, b;
p = Impulse.ar(8);
a = SinOsc.ar(1200, 0, Decay2.ar(p, 0.005, 0.1));
b = SinOsc.ar(600,  0, Decay2.ar(PulseDivider.ar(p, MouseX.kr(1,8).round(1)), 0.005, 0.5));

[a, b] * 0.4
}.scope(2, zoom:4);
)
```

EnvGen
envelope generator
envelope is specified using an instance of the Env class.

```
{ EnvGen.kr(Env.perc, doneAction:2) * SinOsc.ar(880,0,0.2) }.play;
{ EnvGen.kr(Env.perc(1,0.005,1,4), doneAction:2) * SinOsc.ar(880,0,0.2) }.play;

{ EnvGen.kr(Env.perc, Impulse.kr(2)) * SinOsc.ar(880,0,0.2) }.play;
{ EnvGen.kr(Env.perc, Dust.kr(3)) * SinOsc.ar(880,0,0.2) }.play;

// for sustain envelopes a gate is required
z = { arg gate=1; EnvGen.kr(Env.adsr, gate, doneAction:2) * SinOsc.ar(880,0,0.2) }.play;
z.release;

(
// randomly generated envelope
z = { arg gate=1;
var env, n=32;
```

```
env = Env(
[0]++{1.0.rand.squared}.dup(n-1) ++ [0],
{rrand(0.005,0.2)}.dup(n),
\lin, n-8, 8 );
EnvGen.kr(env, gate, doneAction:  2) * LFTri.ar(220,0,0.4)
}.scope(1, zoom:4);
)
z.release;
```

Spectral

FFT, IFFT and the phase vocoder ugens.

FFT calculates the spectrum of a sound, puts it into a buffer, and outputs a trigger each time the
buffer is ready to process. The PV UGens process the spectrum when they receive the trigger.
IFFT converts the spectrum back into sound.

```
// alloc a buffer for the FFT
b = Buffer.alloc(s,2048,1);
// read a sound
    Buffer         "sounds/a11wlk01.wav"


(
// do nothing
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
0.5 * IFFT(chain);
}.scope(1);
)

(
// pass only magnitudes above a threshold
{
```

```
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_MagAbove(chain, MouseX.kr(0.1,512,1));
0.5 * IFFT(chain);
}.scope(1);
)

(
// pass only magnitudes below a threshold
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_MagBelow(chain, MouseX.kr(0.1,512,1));
0.5 * IFFT(chain);
}.scope(1);
)

(
// brick wall filter.
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_BrickWall(chain, MouseX.kr(-1,1));
0.5 * IFFT(chain);
}.scope(1);
)

(
// pass random frequencies.  Mouse controls how many to pass.
// trigger changes the frequencies periodically
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_RandComb(chain, MouseX.kr(0,1), Impulse.kr(0.4));
0.5 * IFFT(chain);
}.scope(1);
```

```
)

(
// rectangular comb filter
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_RectComb(chain, 8, MouseY.kr(0,1), MouseX.kr(0,1));
0.5 * IFFT(chain);
}.scope(1);
)

(
// freeze magnitudes
{
var in, chain;
in = PlayBuf.ar(1,c.bufnum, BufRateScale.kr(c.bufnum), loop:1);
chain = FFT(b.bufnum, in);
chain = PV_MagFreeze(chain, LFPulse.kr(1, 0.75));
0.5 * IFFT(chain);
}.scope(1);
)
```

2. Techniques

Artificial Space

Building a sense of space into a sound by setting up phase differences between the speakers.

```
{ var x; x = BrownNoise.ar(0.2); [x,x] }.scope(2); // correlated
{ {BrownNoise.ar(0.2)}.dup }.scope(2); // not correlated

// correlated
{ var x; x = LPF.ar(BrownNoise.ar(0.2), MouseX.kr(100,10000)); [x,x] }.scope(2);
// not correlated
{ LPF.ar({BrownNoise.ar(0.2)}.dup, MouseX.kr(100,10000)) }.scope(2);
```

```
// correlated
(
{ var x;
x = Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar(7e-3));
[x,x]
}.scope(2))
// not correlated
{ Klank.ar(‘[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar([7e-3,7e-3])) }.scope(2);

// two waves mixed together coming out both speakers
{ var x; x = Mix.ar(VarSaw.ar([100,101], 0, 0.1, 0.2)); [x,x] }.scope(2);
// two waves coming out each speaker independantly
{ VarSaw.ar([100,101], 0, 0.1, 0.2 * 1.414) }.scope(2); // * 1.414 to compensate for power

// delays as cues to direction
// mono
{ var x; x = LFTri.ar(1000,0,Decay2.ar(Impulse.ar(4,0,0.2),0.004,0.2)); [x,x]}.scope(2);

(
// inter-speaker delays
{ var x; x = LFTri.ar(1000,0,Decay2.ar(Impulse.ar(4,0,0.2),0.004,0.2));
[DelayC.ar(x,0.01,0.01),DelayC.ar(x,0.02,MouseX.kr(0.02, 0))]
}.scope(2);
)

(
// mixing two delays together
// you hear a phasing sound but the sound is still flat.
{ var x; x = BrownNoise.ar(0.2);
x = Mix.ar([DelayC.ar(x,0.01,0.01),DelayC.ar(x,0.02,MouseX.kr(0,0.02))]);
[x,x]
}.scope(2);
)

(
// more spatial sounding.  phasing causes you to perceive directionality
{ var x; x = BrownNoise.ar(0.2);
[DelayC.ar(x,0.01,0.01),DelayC.ar(x,0.02,MouseX.kr(0.02, 0))]
```

```
}.scope(2);
)
```

Parallel Structures

```
(
{
 // mixing sine oscillators in parallel
 var        // number of structures to make
 // mix together  parallel structures
Mix.fill(n,
   // this function creates an oscillator at a random frequency
{ FSinOsc.ar(200 + 1000.0.rand) }
 ) / (2*n)   // scale amplitude
}.scope(1);
)

(
{
 // mixing sine oscillators in parallel
 var        // number of structures to make
 // mix together  parallel structures
Mix.fill(n,
   // this function creates an oscillator at a random frequency
{ FSinOsc.ar(200 + 1000.0.rand + [0, 0.5]) }
 ) / (2*n)   // scale amplitude
}.scope(2);
)

(
{
 // mixing sine oscillators in parallel
 var        // number of structures to make
 // mix together  parallel structures
Mix.fill(n,
{
var amp;
amp = FSinOsc.kr(exprand(0.1,1),2pi.rand).max(0);
Pan2.ar(
```

```
FSinOsc.ar(exprand(100,1000.0), 0, amp),
1.0.rand2)
}
 ) / (2*n)   // scale amplitude
}.scope(2);
)



(
{
var n;
 n = 8; // number of 'voices'
 Mix     // mix all stereo pairs down.
    Pan2     // pan the voice to a stereo position
   CombL     // a comb filter used as a string resonator
    Dust     // random impulses as an excitation function
     // an array to cause expansion of Dust to n channels
     // 1 means one impulse per second on average
1.dup(n),
     0.3 // amplitude
),
    0.01, // max delay time in seconds
    // array of different random lengths for each 'string'
{0.004.rand+0.0003}.dup(n),
    4 // decay time in seconds
),
   {1.0.rand2}.dup(n) // give each voice a different pan position
)
)
}.scope(2, zoom:4);
)
```

ID: 665

UGen abstract superclass of all unit generators

superclass: AbstractFunction

Unit generators are the basic building blocks of synths on the server, and are used to generate or process audio or control signals. The many subclasses of UGen are the client-side representations of unit generators, and are used to specify their parameters when constructing synth definitions (see **[SynthDef]**).

See also **[UGens]**, **[Tour_of_UGens]**, and **[UGens-and-Synths]**.

Convenience Methods

scope(name, bufsize, zoom)

Displays the output of this UGen in an individual **[Stethoscope]** window. **name** is the name of the window.

```
Server.default = s = Server.internal.boot;
{ Ringz.ar(PinkNoise.ar([0.1, 0.2]).scope(\pink), 2000, 1, 0.25) }.play; // multichannel works
        // can still separately scope the output of the server
```

poll(interval, label)

Polls the output of this UGen every **interval** seconds, and posts the result. The default **interval** is 0.1 seconds.

```
{ SinOsc.ar(LFNoise0.ar(2).range(420, 460).poll(label:  \LFNoise), 0, 0.2) }.play;

// Multichannel is supported
{ SinOsc.ar(SinOsc.ar([0.2, 0.3]).range(420, 460).poll(label:  \SinOscs), 0, 0.2) }.play;
```

range(lo, hi)

Scales the output of this UGen to be within the range of **lo** and **hi**. **N.B.** 'range' expects the default output range, and thus should not be used in conjunction with **mul** and **add** arguments.

```
{ SinOsc.ar(SinOsc.ar(0.3).range(440, 660), 0, 0.5) * 0.1 }.play;
```

exprange(lo, hi)

Maps the output of this UGen exponentially to be within the range of **lo** and **hi** using a **[LinExp]** UGen. **lo** and **hi** should both be non-zero and have the same sign. **N.B.** 'exprange' expects the default output range, and thus should not be used in conjunction with **mul** and **add** arguments.

clip(lo, hi)

Wraps the receiver in a **[Clip]** UGen, clipping its output at **lo** and **hi**.

fold(lo, hi)

Wraps the receiver in a **[Fold]** UGen, folding its output at **lo** and **hi**.

wrap(lo, hi)

Wraps the receiver in a **[Wrap]** UGen, wrapping its output at **lo** and **hi**.

lag(lagTime)

Wraps the receiver in a **[Lag]** UGen, smoothing it's output by lagTime.

lag2(lagTime)

Wraps the receiver in a **[Lag2]** UGen, smoothing it's output by lagTime.

lag3(lagTime)

Wraps the receiver in a **[Lag3]** UGen, smoothing it's output by lagTime.

degreeToKey(scale, stepsPerOctave)

Wraps the receiver in a **[DegreeToKey]** UGen. The default **stepsPerOctave** is 12.

minNyquist

Wraps the receiver in a **[min]** UGen, such that the lesser of the receiver's output and the Nyquist frequency is output. This can be useful to prevent aliasing.

if(trueUGen, falseUGen)

Outputs **trueUGen** when the receiver outputs 1, **falseUGen** when the receiver outputs 0. If the receiver outputs a value between 0 and 1, a mixture of both will be played. (This is implemented as: ^(this * (trueUGen - falseUGen)) + falseUGen) Note that both **trueUGen** and **falseUGen** will be calculated regardless of whether they are output, so this may not be the most efficient approach.

```
// note different syntax in these two examples
{ if( LFNoise1.kr(1.0, 0.5, 0.5) , SinOsc.ar, Saw.ar ) * 0.1 }.play;


{ Trig1.ar(Dust.ar(3), 0.2).lag(0.1).if(FSinOsc.ar(440), FSinOsc.ar(880)) * 0.1 }.play;
```

@ y

Dynamic geometry support. Returns **Point(this, y)**.

asComplex

Complex math support. Returns **Complex(this, 0.0)**.

dumpArgs

Posts a list of the arguments for this UGen and their values.

Other Instance Methods

The following methods and instance variables are largely used in the construction of synth definitions, synth descriptions (see **[SynthDesc]**), UGen class definitions, etc., and are usually not needed for general use. Users should not attempt to set any of these values in general code.

synthDef

The SynthDef which contains the UGen.

inputs

The array of inputs to the UGen.

rate

The output rate of the UGen which is one of the Symbols 'audio', or 'control'.

signalRange

Returns a symbol indicating the signal range of the receiver. Either \bipolar or \unipolar.

numChannels

Returns the number of output Channels. For a UGen, this will always be 1, but **[Array]** also implements this method, so multichannel expansion is supported. See **[MultiChannel]**.

numInputs

Returns the number of inputs for this UGen.

numOutputs

Returns the number of outputs for this UGen.

name

Returns the **[Class]** name of the receiver as a **[String]**.

madd(mul, add)

Wraps the receiver in a **MulAdd** UGen. This is only used in UGen class definitions in order to allow efficient implementation of **mul** and **add** arguments.

isValidUGenInput

Returns true.

asUGenInput

Returns the receiver.

ID: 666

UGens Overview

See also: [UGen] and [Tour_of_UGens]

Below is a partial list of UGens sorted by type.

Analysis

Amplitude
Compander
Pitch
Slope
ZeroCrossing

Control

DegreeToKey
AmpComp
Slew
MouseX
MouseY
MouseButton

Demand UGens:

Demand
Duty
TDuty
Dseries
Dgeom
Dseq
Dser
Drand
Dxrand
Dswitch1
Dwhite
Diwhite

Dbrown
Dibrown

In/Out

AudioIn
In
InTrig
InFeedback
Out
ReplaceOut
XOut
OffsetOut
LocalIn
LocalOut
SharedIn
SharedOut

Oscillators

Blip
Formant
FSinOsc
Gendy1
Gendy2
Gendy3
Impulse
Klang
LFCub
LFPar
LFPulse
LFSaw
Osc
OscN
Pulse
Saw
SinOsc
SyncSaw
VarSaw
VOsc

VOsc3

Delays

AllpassC
AllpassL
AllpassN
BufAllpassC
BufAllpassL
BufAllpassN
BufDelayC
BufDelayL
BufDelayN
BufCombC
BufCombL
BufCombN
CombC
CombL
CombN
Delay1
Delay2
DelayC
DelayL
DelayN
PingPong
PitchShift

Multiple Channels

Mix
MultiOutUGen
OutputProxy

BiPanB2
DecodeB2
LinPan2
LinXFade2
Pan2
Pan4
PanAz

PanB
PanB2
Rotate2
XFade2

Physical Models

Ball
Spring
TBall

Filters

BPF
BPZ2
BRF
Formlet
FOS
HPF
HPZ1
HPZ2
Integrator
Klank
Lag
Lag2
Lag3
LeakDC
Limiter
LinExp
LinLin
LPF
LPZ1
LPZ2
Median
Normalizer
OnePole
OneZero
Resonz
RHPF
Ringz

RLPF
SOS
TwoPole
TwoZero

Noise

LFNoise1
LFNoise2
LFNoise0
LFClipNoise

NoahNoise
WhiteNoise
GrayNoise
Hasher
MantissaMask
PinkerNoise
PinkNoise
ClipNoise
Dust
Dust2
Latoocarfian
Rossler
Crackle

Triggers

MostChange
PulseCount
Stepper
Gate
CoinGate
Peak
PulseDivider
LastValue
Latch
Trig
Trig1
SendTrig

Sweep
Timer
Phasor

Frequency Domain

FFT
Convolution
PV_ConformalMap

Buffer Manipulation

BufRd
BufWr
DiskIn
DiskOut
Index
WrapIndex
PlayBuf
RecordBuf
PingPong
TGrains

Binary Operators

BinaryOpUGen (overview)
+
-
*
/
**
absdif
amclip
atan2
clip2
difsqr
excess
fold2
hypot
hypotApx

max
min
ring1
ring2
ring3
ring4
round
scaleneg
sqrdif
sqrsum
sumsqr
thresh
trunc
wrap2

InfoUGens

BufChannels
BufDur
BufFrames
BufRateScale
BufSampleRate
NumRunningSynths
SampleDur
SampleRate

Synth-Controlling-UGens

see overview: [Synth-Controlling-UGens]

Control
DetectSilence
EnvGen
Line
XLine
Linen
Free
FreeSelf
Pause

PauseSelf

Random Values

Rand
ExpRand
LinRand
NRand
IRand
TIRand
TRand
TExpRand
RandSeed
RandID

Array Usage

Select
TWindex

to be continued ....

25.12 Noise

ID: 667

BrownNoise

BrownNoise.ar(mul, add)

Generates noise whose spectrum falls off in power by 6 dB per octave.

```
// compare
{ BrownNoise.ar(0.5) }.play;
{ WhiteNoise.ar(0.5) }.play;
```

ID: 668

ClipNoise

ClipNoise.ar(mul, add)

Generates noise whose values are either -1 or 1.
This produces the maximum energy for the least peak to peak amplitude.

```
(
SynthDef "help-ClipNoise"    arg
Out.ar(out,
ClipNoise.ar(0.2)
)
}).play;
)
```

ID: 669

CoinGate statistical gate

***kr(prob, trig)**
***ar(prob, trig)**

When it receives a trigger, it tosses a coin, and either passes the trigger or doesn't.

prob value between 0 and 1 determines probability of either possibilities
trig input signal

```
//examples

(
a = SynthDef("help-TCoin", { arg out=0, prob=0.5;
var trig;
trig = CoinGate.kr(prob, Impulse.kr(10));
Out.ar(out,
SinOsc.ar(
TRand.kr(300.0, 400.0, trig),0,0.2
)
)
}).play;
)

a.set(\prob, 1.0);
a.set(\prob, 0.0);
a.set(\prob, 0.1);


(
a = SynthDef("help-TCoin", { arg out=0, prob=0.5;
var trig;
trig = Impulse.ar(20, 0, SinOsc.kr(0.5,0,1,1));
Out.ar(out,
Mix.fill(3, {Ringz.ar(CoinGate.ar(prob, trig*0.5), #[1,1.5]*Rand(1000, 9000), 0.01)})
)
```

```
}).play;
)

a.set(\prob, 1.0);
a.set(\prob, 0.0);
a.set(\prob, 0.1);
```

ID: 670

Crackle chaotic noise function

Crackle.ar(param, mul, add)

A noise generator based on a chaotic function.
param - a parameter of the chaotic function with useful values from just below 1.0 to just above 2.0. Towards 2.0 the sound crackles.

```
(
SynthDef "help-Crackle"    arg
Out.ar(out,
Crackle.ar(1.95, 0.5)
)
}).play;
)

//modulate chaos parameter
(
SynthDef "help-Crackle"    arg
Out.ar(out,
Crackle.ar(Line.kr(1.0, 2.0, 3), 0.5)
)
}).play;
)
```

ID: 671

Dust random impulses

Dust.ar(density, mul, add)

Generates random impulses from 0 to +1.
density - average number of impulses per second

```
(
SynthDef("help-Dust", { arg out=0;
Out.ar(out,
Dust.ar(200, 0.5)
)
}).play;
)

(
SynthDef("help-Dust", { arg out=0;
Out.ar(out,
Dust.ar(XLine.kr(20000, 2, 10), 0.5)
)
}).play;
)
```

ID: 672

Dust2 random impulses

Dust2.ar(density, mul, add)

Generates random impulses from -1 to +1.
density - average number of impulses per second

```
(
SynthDef("help-Dust2", { arg out=0;
Out.ar(out,
Dust2.ar(200, 0.5)
)
}).play;
)

(
SynthDef("help-Dust2", { arg out=0;
Out.ar(out,
Dust2.ar(XLine.kr(20000, 2, 10), 0.5)
)
}).play;
)
```

ID: 673

ExpRand

ExpRand(lo, hi)

Generates a single random float value in an exponential distributions from lo to hi.

```
(
SynthDef("help-ExpRand", { arg out=0, n=0;
Out.ar(out,
  FSinOsc
ExpRand(100.0, 8000.0, n),
0, Line.kr(0.2, 0, 0.01, doneAction:2))
)
}).send(s);
)

(
Routine({
inf.do({ arg i;
Synth.new("help-ExpRand"); 0.05.wait;
})
}).play;
)
```

ID: 674

GrayNoise

GrayNoise.ar(mul, add)

Generates noise which results from flipping random bits in a word.
This type of noise has a high RMS level relative to its peak to peak level.
The spectrum is emphasized towards lower frequencies.

```
(
SynthDef "help-GrayNoise"    arg
Out.ar(out,
GrayNoise.ar(0.1)
)
}).play;
)
```

ID: 675

Hasher randomized value

Hasher.ar(in, mul, add)

Returns a unique output value from zero to one for each input value according to a hash function. The same input value will always produce the same output value. The input need not be from zero to one.
in - input signal

```
{ Hasher.ar(Line.ar(0,1,1), 0.2) }.play;

(
{
 SinOsc
Hasher.kr(MouseX.kr(0,10).round(1), 300, 500)
) * 0.1
}.play;
)

(
{
 SinOsc
Hasher.kr(MouseX.kr(0,10).round(1) + 0.0001, 300, 500)
) * 0.1
}.play;
)

(
{
 SinOsc
Hasher.kr(MouseX.kr(0,10), 300, 500)
) * 0.1
}.play;
)
```

ID: 676

IRand

IRand(lo, hi)

Generates a single random integer value in uniform distribution from lo to hi

```
(
SynthDef "help-IRand"
Out.ar(
IRand(0, 1), //play on random channel between 0 and 1
FSinOsc.ar(500,
0, Line.kr(0.2, 0, 0.1, doneAction:2))
)
}).send(s);
)

(
Routine({
16.do({
Synth.new("help-IRand"); 0.5.wait;
})
}).play;
)
```

ID: 677

Latoocarfian chaotic function

Latoocarfian.ar(a, b, c, d, mul, add)

This is a function given inClifford Pickover's book Chaos In Wonderland, pg 26.
The function has four parameters a, b, c, and d.
The function is:

xnew = sin(y * b) + c * sin(x * b);
ynew = sin(x * a) + d * sin(y * a);
x = xnew;
y = ynew;
output = x;

According to Pickover, parameters **a** and **b** should be in the range from -3 to +3, and parameters **c** and **d** should be in the range from 0.5 to 1.5.
The function can, depending on the parameters given, give continuous chaotic output, converge to a single value (silence) or oscillate in a cycle (tone).
This UGen is experimental and not optimized currently, so is rather hoggish of CPU.

```
//not installed yet!
(
SynthDef("help-Latoocarfian", { arg out=0, a=1.0, b=1.0, c=0.7, d=0.7;
var env, a, b, c, d;
env = EnvGen.kr(Env.linen(0.1, 1, 0.1), doneAction:2);
Out.ar(out,
Latoocarfian.ar(a, b, c, d, 0.05)
)
}).send(s);
)

{
 Synth      "help-Latoocarfian", [
  \a, 3.0.rand, \b, 3.0.rand,
  \c, 0.5 + 1.5.rand, \d, 0.5 + 1.5.rand]
 );
1.0.wait;
```

```
}.play;


//todo:
(
// GUI version:
w = GUIWindow.new("Latoocarfian", Rect.newBy(40,40,200,300));
SliderView.new(w, Rect.newBy(8,8,20,280), nil, 0, -3, 3);
SliderView.new(w, Rect.newBy(32,8,20,280), nil, 0, -3, 3);
SliderView.new(w, Rect.newBy(56,8,20,280), nil, 1, 0.5, 1.5);
SliderView.new(w, Rect.newBy(80,8,20,280), nil, 1, 0.5, 1.5);
{ XFadeTexture.ar({
w.views.at(0).value = 3.0.rand2;
w.views.at(1).value = 3.0.rand2;
w.views.at(2).value = 0.5 + 1.0.rand;
w.views.at(3).value = 0.5 + 1.0.rand;
//[a, b, c, d].postln;
Latoocarfian.ar(w.views.at(0).value, w.views.at(1).value,
w.views.at(2).value, w.views.at(3).value, 0.05)
},  1, 0.1, 1) }.play;
)
```

ID: 678

LFClipNoise clipped noise

LFClipNoise.ar(freq, mul, add)

Randomly generates the values -1 or +1 at a rate given by the nearest integer division of the sample rate by the freq argument. It is probably pretty hard on your speakers!
freq - approximate rate at which to generate random values.

```
(
SynthDef "help-LFClipNoise"    arg
Out.ar(out,
LFClipNoise.ar(1000, 0.25)
)
}).play;
)

//modulate frequency
(
SynthDef "help-LFClipNoise"    arg
Out.ar(out,
LFClipNoise.ar(XLine.kr(1000, 10000, 10), 0.25)
)
}).play;
)

//use as frequency control
(
SynthDef "help-LFClipNoise"    arg
Out.ar(out,
SinOsc.ar(
LFClipNoise.ar(4, 200, 600),
0, 0.2
)
)
}).play;
```

)

ID: 679

LFDClipNoise dynamic clipped noise

LFClipNoise.ar(freq, mul, add)

Like **LFClipNoise**, it generates the values -1 or +1 at a rate given by the **freq** argument, with two differences:

-no time quantization
-fast recovery from low freq values.

(LFClipNoise, as well as LFNoise0,1,2 quantize to the nearest integer division of the samplerate, and they poll the freq argument only when scheduled; thus they often seem to hang when freqs get very low).

If you don't need very high or very low freqs, or use fixed freqs, LFNoise0 is more efficient.

freq - rate at which to generate random values.

```
// try wiggling the mouse quickly;
// LFNoise frequently seems stuck, LFDNoise changes smoothly.

{ LFClipNoise.ar(MouseX.kr(0.1, 1000, 1), 0.1)  }.play

{ LFDClipNoise.ar(MouseX.kr(0.1, 1000, 1), 0.1)  }.play

// silent for 2 secs before going up in freq

{ LFClipNoise.ar(XLine.kr(0.5, 10000, 3), 0.1) }.scope;

{ LFDClipNoise.ar(XLine.kr(0.5, 10000, 3), 0.1) }.scope;


// LFNoise quantizes time steps at high freqs, LFDNoise does not:

{ LFClipNoise.ar(XLine.kr(1000, 20000, 10), 0.1) }.scope;
```

```
{ LFDClipNoise.ar(XLine.kr(1000, 20000, 10), 0.1) }.scope;
```

ID: 680

LFDNoise0 dynamic step noise

LFDNoise0.ar(freq, mul, add)

Like **LFNoise0**, it generates random values at a rate given by the **freq** argument, with two differences:

-no time quantization
-fast recovery from low freq values.

(LFNoise0,1,2 quantize to the nearest integer division of the samplerate, and they poll the freq argument only when scheduled, and thus seem to hang when freqs get very low).

If you don't need very high or very low freqs, or use fixed freqs, LFNoise0 is more efficient.

freq - rate at which to generate random values.

```
// try wiggling mouse quickly;
// LFNoise frequently seems stuck, LFDNoise changes smoothly.

{ LFNoise0.ar(MouseX.kr(0.1, 1000, 1), 0.1)  }.play

{ LFDNoise0.ar(MouseX.kr(0.1, 1000, 1), 0.1)  }.play

// silent for 2 secs before going up in freq

{ LFNoise0.ar(XLine.kr(0.5, 10000, 3), 0.1) }.scope;

{ LFDNoise0.ar(XLine.kr(0.5, 10000, 3), 0.1) }.scope;


// LFNoise quantizes time steps at high freqs, LFDNoise does not:

{ LFNoise0.ar(XLine.kr(1000, 20000, 10), 0.1) }.scope;
```

```
{ LFDNoise0.ar(XLine.kr(1000, 20000, 10), 0.1) }.scope;

{ LFNoise2.ar(1000, 0.25) }.play;
```

ID: 681

LFDNoise1 dynamic ramp noise

LFDNoise1.ar(freq, mul, add)

Like **LFNoise1**, it generates linearly interpolated random values
at a rate given by the **freq** argument, with two differences:

-no time quantization
-fast recovery from low freq values.

(LFNoise0,1,2 quantize to the nearest integer division of the samplerate,
and they poll the freq argument only when scheduled, and thus seem
to hang when freqs get very low).

If you don't need very high or very low freqs, or use fixed freqs,
LFNoise1 is more efficient.

freq - rate at which to generate random values.

```
// try wiggling mouse quickly;
// LFNoise frequently seems stuck, LFDNoise changes smoothly.

{  SinOsc.ar(LFNoise1.ar(MouseX.kr(0.1, 1000, 1), 200, 500), 0, 0.2)  }.play

{  SinOsc.ar(LFDNoise1.ar(MouseX.kr(0.1, 1000, 1), 200, 500), 0, 0.2)  }.play


// LFNoise quantizes time steps at high freqs, LFDNoise does not:

{ LFNoise1.ar(XLine.kr(2000, 20000, 8), 0.1) }.scope;

{ LFDNoise1.ar(XLine.kr(2000, 20000, 8), 0.1) }.scope;
```

ID: 682

LFDNoise3 dynamic cubic noise

LFDNoise3.ar(freq, mul, add)

Similar to **LFNoise2**, it generates polynomially interpolated random values at a rate given by the **freq** argument, with 3 differences:

-no time quantization
-fast recovery from low freq values
-cubic instead of quadratic interpolation

(LFNoise0,1,2 quantize to the nearest integer division of the samplerate, and they poll the freq argument only when scheduled, and thus seem to hang when freqs get very low).
If you don't need very high or very low freqs, or use fixed freqs, LFNoise2 is more efficient.

freq - rate at which to generate random values.

```
// try wiggling mouse quickly:
// LFNoise2 overshoots when going from high to low freqs, LFDNoise changes smoothly.

{  SinOsc.ar(LFNoise2.ar(MouseX.kr(0.1, 1000, 1), 200, 500), 0, 0.2)  }.play

{  SinOsc.ar(LFDNoise3.ar(MouseX.kr(0.1, 1000, 1), 200, 500), 0, 0.2)  }.play


// LFNoise quantizes time steps at high freqs, LFDNoise does not:

{ LFNoise2.ar(XLine.kr(2000, 20000, 8), 0.1) }.scope;

{ LFDNoise3.ar(XLine.kr(2000, 20000, 8), 0.1) }.scope;

// use as frequency control
(
{
SinOsc.ar(
LFDNoise3.ar(4, 400, 450),
```

```
0, 0.2
)
}.play;
)
```

ID: 683

LFNoise0 step noise

LFNoise0.ar(freq, mul, add)

Generates random values at a rate given by the nearest integer division of the sample rate by the freq argument.
freq - approximate rate at which to generate random values.

```
(
SynthDef "help-LFNoise0"    arg
Out.ar(out,
LFNoise0.ar(1000, 0.25)
)
}).play;
)

//modulate frequency
(
SynthDef "help-LFNoise0"    arg
Out.ar(out,
LFNoise0.ar(XLine.kr(1000, 10000, 10), 0.25)
)
}).play;
)

//use as frequency control
(
SynthDef "help-LFNoise0"    arg
Out.ar(out,
SinOsc.ar(
LFNoise0.ar(4, 400, 450),
0, 0.2
)
)
}).play;
)
```

Where: Help→UGens→Noise→LFNoise0

ID: 684

LFNoise1 ramp noise

LFNoise1.ar(freq, mul, add)

Generates linearly interpolated random values at a rate given by the nearest integer division of the sample rate by the freq argument.
freq - approximate rate at which to generate random values.

```
(
SynthDef "help-LFNoise1"    arg
Out.ar(out,
LFNoise1.ar(1000, 0.25)
)
}).play;
)

//modulate frequency
(
SynthDef "help-LFNoise1"    arg
Out.ar(out,
LFNoise1.ar(XLine.kr(1000, 10000, 10), 0.25)
)
}).play;
)

//use as frequency control
(
SynthDef "help-LFNoise1"    arg
Out.ar(out,
SinOsc.ar(
LFNoise1.ar(4, 400, 450),
0, 0.2
)
)
}).play;
)
```

ID: 685

LFNoise2 quadratic noise

LFNoise2.ar(freq, mul, add)

Generates quadratically interpolated random values at a rate given by the nearest integer division of the sample rate by the freq argument.
freq - approximate rate at which to generate random values.

```
(
SynthDef "help-LFNoise2"    arg
Out.ar(out,
LFNoise2.ar(1000, 0.25)
)
}).play;
)

//modulate frequency
(
SynthDef "help-LFNoise2"    arg
Out.ar(out,
LFNoise2.ar(XLine.kr(1000, 10000, 10), 0.25)
)
}).play;
)

//use as frequency control
(
SynthDef "help-LFNoise2"    arg
Out.ar(out,
SinOsc.ar(
LFNoise2.ar(4, 400, 450),
0, 0.2
)
)
}).play;
)
```

ID: 686

LinRand

LinRand(lo, hi, minmax)

Generates a single random float value in linear distribution from lo to hi, skewed towards lo if minmax < 0, otherwise skewed towards hi.

```
(
SynthDef("help-LinRand", { arg out=0, minmax=1;
Out.ar(out,
  FSinOsc
LinRand(200.0, 10000.0, minmax),
0, Line.kr(0.2, 0, 0.01, doneAction:2))
)
}).send(s);
)

//towards hi
(
Routine
loop({
Synth.new("help-LinRand"); 0.04.wait;
})
}).play;
)

//towards lo (doesn't work like that yet)
(
Routine
loop({
Synth.new("help-LinRand", [\minmax, -1]); 0.04.wait;
})
}).play;
)
```

ID: 687

MantissaMask reduce precision

MantissaMask.ar(in, bits, mul, add)

Masks off bits in the mantissa of the floating point sample value. This introduces a quantization noise, but is less severe than linearly quantizing the signal.

in - input signal
bits - the number of mantissa bits to preserve. a number from 0 to 23.

```
// preserve only 3 bits of mantissa.
{ MantissaMask.ar(SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4), 3) }.play

// the original
{ SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4) }.play

// the difference.
(
{
var in;
in = SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4);
Out.ar(0, in - MantissaMask.ar(in, 3));
}.play
)


// preserve 7 bits of mantissa.
// This makes the lower 16 bits of the floating point number become zero.
{ MantissaMask.ar(SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4), 7) }.play

// the original
{ SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4) }.play

// the difference.
(
{
```

```
var in;
in = SinOsc.ar(SinOsc.kr(0.2,0,400,500), 0, 0.4);
Out.ar(0, in - MantissaMask.ar(in, 7));
}.play
)
```

ID: 688

NoahNoise

NoahNose.ar(mul, add)

//not installed yet

```
(
SynthDef "help-NoahNoise"    arg
Out.ar(out,
NoahNoise.ar(0.25)
)
}).play;
)
```

ID: 689

NRand

NRand(lo, hi, n)

Generates a single random float value in a sum of n uniform distributions from lo to hi.

n = 1 : uniform distribution - same as Rand
n = 2 : triangular distribution
n = 3 : smooth hump
as n increases, distribution converges towards gaussian

```
(
SynthDef("help-NRand", { arg out=0, n=0;
Out.ar(out,
  FSinOsc
 NRand(1200.0, 4000.0, n),
0, Line.kr(0.2, 0, 0.01, doneAction:2))
)
}).send(s);
)

(
n = 0;
Routine({
inf.do({ arg i;
  Synth     "help-NRand", [\n, n]
})
}).play;
)

n = 1;
n = 2;
n = 4;
```

ID: 690

PinkerNoise

PinkerNoise.ar(mul, add)

Generates noise whose spectrum falls off in power by 3 dB per octave.
This gives equal power over the span of each octave.
This version gives 16 octaves of pink noise, whereas PinkNoise only gives 8 octaves.

```
  not installed
(
SynthDef "help-PinkerNoise"    arg
Out.ar(out,
  PinkerNoise
)
}).play;
)
```

ID: 691

PinkNoise

PinkNoise.ar(mul, add)

Generates noise whose spectrum falls off in power by 3 dB per octave.
This gives equal power over the span of each octave.
This version gives 8 octaves of pink noise.

```
(
SynthDef "help-PinkNoise"    arg
Out.ar(out,
PinkNoise.ar(0.4)
)
}).play;
)
```

ID: 692

Rand

Rand(lo, hi)

Generates a single random float value in uniform distribution from lo to hi.
It generates this when the SynthDef first starts playing, and remains fixed for the duration of the synth's existence.

```
(
SynthDef("help-Rand", { arg out=0;
Out.ar(out,
  FSinOsc
Rand(200.0, 400.0),
0, Line.kr(0.2, 0, 1, doneAction:2))
)
}).send(s);
)

(
Routine({
8.do({
Synth.new("help-Rand"); 1.0.wait;
})
}).play;
)
```

ID: 693

RandID set the synth's random generator id

RandID.kr(seed)
RandID.ir(seed)

Choose which random number generator to use for this synth. All synths that use the same generator reproduce the same sequence of numbers when the same seed is set again

See also: [**RandSeed**][**randomSeed**]

```
//start a noise patch and set the id of the generator
(
SynthDef("help-RandID", { arg out=0, id=1;
RandID.ir(id);
Out.ar(out,
WhiteNoise.ar(0.05) + Dust2.ar(70)
)
}).send(s);
)

//reset the seed of my rgen at a variable rate
(
SynthDef("help-RandSeed", { arg seed=1910, id=1;
RandID.kr(id);
RandSeed.kr(Impulse.kr(FSinOsc.kr(0.2, 0, 10, 11)), seed);
}).send(s);

)

//start two noise synths on left and right channel with a different randgen id
a = Synth("help-RandID", [\out, 0, \id, 1]);
b = Synth("help-RandID", [\out, 1, \id, 2]);

//reset the seed of randgen 1
    Synth "help-RandSeed"   \id
```

```
//change the target randgen to 2 (affects right channel)
x.set(\id, 2);
```

ID: 694

RandSeed set the synth's random generator seed

RandSeed.kr(trig, seed)

When the trigger signal changes from nonpositive to positve, the synth's random generator seed is reset to the given value. All synths that use the same random number generator reproduce the same sequence of numbers again.

see [**RandID**] ugen for setting the randgen id,
also [**randomSeed**] for the client side equivalent

```
// start a noise patch

(
{
var noise, filterfreq;
noise = WhiteNoise.ar(0.05 !  2) + Dust2.ar(70 !  2);
filterfreq = LFNoise1.kr(3, 5500, 6000);
Resonz.ar(noise * 5, filterfreq, 0.5) + (noise * 0.5)
}.play;
)

// reset the seed at a variable rate
(
x = { arg seed=1956;
RandSeed.kr(Impulse.kr(MouseX.kr(0.1, 100)), seed);
}.play;
)



x.set(\seed, 2001);
x.set(\seed, 1798);
x.set(\seed, 1902);
```

```
// above you can see that the sound of the LFNoise1 is not exactly reproduced (filter frequency)
// this is due to interference between the internal phase of the noise ugen and the
// seed setting rate.

// a solution is to start a new synth:

(
SynthDef("pseudorandom", { arg out, sustain=1, seed=1967, id=0;
var noise, filterfreq;
RandID.ir(id);
RandSeed.ir(1, seed);


noise = WhiteNoise.ar(0.05 !  2) + Dust2.ar(70 !  2);
filterfreq = LFNoise1.kr(3, 5500, 6000);

Out.ar(out,
Resonz.ar(noise * 5, filterfreq, 0.5) + (noise * 0.5)
*
Line.kr(1, 0, sustain, doneAction:2)
)

}).send(s);
)

// the exact same sound is reproduced
(
fork {
loop {
  Synth "pseudorandom"
  1.1.wait; // wait a bit longer than sustain, so sounds don't overlap
}
}
)

// changing the rand seed changes the sound:

(
fork {
(1902..2005).do { | seed|
```

```
seed.postln;
3.do {
Synth("pseudorandom", [\seed, seed]);
1.1.wait;
}
}
}
)

// cd skipper
(
fork {
(1902..2005).do { | seed|
seed.postln;
rrand(4,10).do {
Synth("pseudorandom", [\seed, seed, \sustain, 0.05]);
0.06.wait;
}
}
}
)

// if the sounds overlap, this does not work as expected anymore
// sounds vary.

(
fork {
loop {
  Synth "pseudorandom"
  0.8.wait; // instead of 1.1
}
}
)

// rand id can be used to restrict the resetting of the seed to each voice:

(
fork {
var id=0;
(1902..2005).do { | seed|
```

```
seed.postln;
3.do {
Synth("pseudorandom", [\seed, seed, \id, id]);
   id = id + 1 % 16; // there is 16 different random generators
0.8.wait;
}
}
}
)
```

ID: 695

Rossler chaotic function

Rossler.ar(chaosParam, dt, mul, add)

The Rossler attractor is a well known chaotic function.
The **chaosParam** can be varied from 1.0 to 25.0 with a **dt** of 0.04.
Valid ranges for **chaosParam** vary depending on **dt**.
chaosParam - a Float.
dt - time step parameter. Default is 0.04.

```
//not defined yet!

(
SynthDef "help-Rossler"    arg
Out.ar(out,
Rossler.ar(4, 0.08)
)
}).play;
)

(
a = SynthDef("help-Rossler", { arg out=0, param=4, dt=0.04;
Out.ar(out,
Rossler.ar(param, dt)
)
}).play;
)

a.set(\param, 2.5);
a.set(\dt, 0.02);
```

ID: 696

TExpRand triggered exponential random number generator

TExpRand.ar(lo, hi, trig)
TExpRand.kr(lo, hi, trig)

Generates a random float value in exponential distribution from lo to hi each time the trig signal changes from nonpositive to positive values
lo and hi must both have the same sign and be non-zero.

```
(
{
var trig = Dust.kr(10);
 SinOsc
TExpRand.kr(300.0, 3000.0, trig)
) * 0.1
}.play;
)

(
{
var trig = Dust.ar(MouseX.kr(1, 8000, 1));
 SinOsc
TExpRand.ar(300.0, 3000.0, trig)
) * 0.1
}.play;
)
```

ID: 697

TIRand triggered integer random number generator

TIRand.kr(lo, hi, trig)
TIRand.ar(lo, hi, trig)

Generates a random integer value in uniform distribution from lo to hi each time the trig signal changes from nonpositive to positive values

```
(
SynthDef "help-TRand"
var trig, outBus;
trig = Dust.kr(10);
outBus = TIRand.kr(0, 1, trig); //play on random channel between 0 and 1
Out.ar(outBus, PinkNoise.ar(0.2))

}).play;
)

(
{
var trig = Dust.kr(10);
 SinOsc
TIRand.kr(4, 12, trig) * 100
) * 0.1
}.play;
)

(
{
var trig = Dust.ar(MouseX.kr(1, 8000, 1));
 SinOsc
TIRand.ar(4, 12, trig) * 100
) * 0.1
}.play;
)
```

ID: 698

TRand triggered random number generator

TRand.kr(lo, hi, trig)
TRand.ar(lo, hi, trig)

Generates a random float value in uniform distribution from lo to hi each time the trig signal changes from nonpositive to positive values

```
(
{
var trig = Dust.kr(10);
 SinOsc
TRand.kr(300, 3000, trig)
) * 0.1
}.play;
)

(
{
var trig = Dust.ar(MouseX.kr(1, 8000, 1));
 SinOsc
TRand.ar(300, 3000, trig)
) * 0.1
}.play;
)
```

ID: 699

WhiteNoise

WhiteNoise.ar(mul, add)

Generates noise whose spectrum has equal power at all frequencies.

```
(
SynthDef "help-WhiteNoise"    arg
Out.ar(out,
WhiteNoise.ar(0.25)
)
}).play;
)
```

25.13 Oscillators

ID: 700

Blip band limited impulse oscillator

Blip.ar(kfreq, knumharmonics, mul, add)

Band Limited ImPulse generator. All harmonics have equal amplitude.
This is the equivalent of 'buzz' in MusicN languages. WARNING: This waveform in its raw form could be damaging to your ears at high amplitudes or for long periods.

Implementation notes:
It is improved from other implementations in that it will crossfade in a control period when the number of harmonics changes, so that there are no audible pops. It also eliminates the divide in the formula by using a 1/sin table (with special precautions taken for 1/0). The lookup tables are linearly interpolated for better quality.

(Synth-O-Matic (1990) had an impulse generator called blip, hence that name here rather than 'buzz').

kfreq - frequency in Hertz
knumharmonics - number of harmonics. This may be lowered internally if it would cause aliasing.

```
// modulate frequency
{ Blip.ar(XLine.kr(20000,200,6),100,0.2) }.play;

// modulate numharmonics
{ Blip.ar(200,Line.kr(1,100,20),0.2) }.play;
```

ID: 701

BufRd buffer reading oscillator

read the content of a buffer at an index.
see also **BufWr**

BufRd.ar(numChannels, bufnum, phase, loop)

numChannels number of channels that the buffer will be.
this must be a fixed integer. The architechture of the SynthDef
cannot change after it is compiled.
warning: if you supply a bufnum of a buffer that has a different
numChannels then you have specified to the BufRd, it will
fail silently.

bufnum the index of the buffer to use
phase **audio** rate modulateable index into the buffer.
loop 1 means true, 0 means false. this is modulateable.
interpolation 1 means no interpolation, 2 is linear, 4 is cubic interpolation

in comparison to **PlayBuf**:
PlayBuf plays through the buffer by itself,
BufRd only moves its read point by the phase input
and therefore has no pitch input

BufRd has variable interpolation

```
(
// read a whole sound into memory
s = Server.local;
// note:  not *that* columbia, the first one
```

```
        "/b_allocRead"     "sounds/a11wlk01.wav"
)

//use any AUDIO rate ugen as an index generator

{ BufRd.ar(1, 0, SinOsc.ar(0.1) * BufFrames.ir(0)) }.play;
{ BufRd.ar(1, 0, LFNoise1.ar(1) * BufFrames.ir(0)) }.play;
{ BufRd.ar(1, 0, LFNoise1.ar(10) * BufFrames.ir(0)) }.play;
{ BufRd.ar(1, 0, LFTri.ar(0.1) + LFTri.ar(0.23) * BufFrames.ir(0)) }.play;
// original duration
{ BufRd.ar(1, 0, LFSaw.ar(BufDur.ir(0).reciprocal).range(0, BufFrames.ir(0)) ) }.play;


//use a phasor index into the file

{ BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0), 0, BufFrames.kr(0))) }.play;


//change rate and interpolation
(
x = { arg rate=1, inter=2;
BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)), 1, inter)
}.play;
)

x.set(\rate, 0.9);
x.set(\rate, 0.6);
x.set(\inter, 1);
x.set(\inter, 0);


//write into the buffer with a BufWr
(
y = { arg rate=1;
var in;
in = SinOsc.ar(LFNoise1.kr(2, 300, 400), 0, 0.1);
BufWr.ar(in, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)));
 0.0 //quiet
}.play;
)
```

```
//read it with a BufRd
(
x = { arg rate=1;
BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)))
}.play;
)



x.set(\rate, 5);
y.set(\rate, 2.0.rand);
x.set(\rate, 2);
```

ID: 702

BufWr buffer writing oscillator

write to a buffer at an index
see also **BufRd**

BufWr.ar(input, bufnum, phase, loop)

input input ugens (channelArray)

bufnum the index of the buffer to use
phase modulateable index into the buffer (has to be audio rate).
loop 1 means true, 0 means false. this is modulateable.

Note: BufWr (in difference to BufRd) does not do multichannel expansion, because input is an array.

```
(
// allocate a buffer for writinig into
s = Server.local;
s.sendMsg("/b_alloc", 0, 44100 * 2);
)


//write into the buffer with a BufWr
(
y = { arg rate=1;
var in;
in = SinOsc.ar(LFNoise1.kr(2, 300, 400), 0, 0.1);
BufWr.ar(in, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)));
```

```
 0.0 //quiet
}.play;
)

//read it with a BufRd
(
x = { arg rate=1;
BufRd.ar(1, 0, Phasor.ar(0, BufRateScale.kr(0) * rate, 0, BufFrames.kr(0)))
}.play(s);
)



x.set(\rate, 5);
y.set(\rate, 2.0.rand);
x.set(\rate, 2);
```

ID: 703

COsc chorusing wavetable oscillator

COsc.ar(bufnum, freq, beats, mul, add)

Chorusing wavetable lookup oscillator. Produces sum of two signals at (freq +/- (beats / 2)). Due to summing, the peak amplitude is twice that of the wavetable.
bufnum - the number of a buffer filled in wavetable format
freq - frequency in Hertz
beats - beat frequency in Hertz

```
(
b = Buffer.alloc(s, 512, 1, {arg buf; buf.sine1Msg(1.0/[1,2,3,4,5,6,7,8,9,10])});
{ COsc.ar(b.bufnum, 200, 0.7, 0.25) }.play;
)
```

ID: 704

Formant formant oscillator

Formant.ar(kfundfreq, kformfreq, kwidthfreq, mul, add)

Generates a set of harmonics around a formant frequency at a given fundamental frequency.
kfundfreq - fundamental frequency in Hertz.
kformfreq - formant frequency in Hertz.
kwidthfreq - pulse width frequency in Hertz. Controls the bandwidth of the formant. Widthfreq must be greater than or equal fundfreq.

```
// modulate fundamental frequency, formant freq stays constant
{ Formant.ar(XLine.kr(400,1000, 8), 2000, 800, 0.125) }.play

// modulate formant frequency, fundamental freq stays constant
{ Formant.ar(200, XLine.kr(400, 4000, 8), 200, 0.125) }.play

// modulate width frequency, other freqs stay constant
{ Formant.ar(400, 2000, XLine.kr(800, 8000, 8), 0.125) }.play
```

ID: 705

FSinOsc fast sine oscillator

FSinOsc.ar(freq, iphase,mul, add)

Very fast sine wave generator (2 PowerPC instructions per output sample!) implemented using a ringing
filter. This generates a much cleaner sine wave than a table lookup oscillator and is a lot faster.
However, the amplitude of the wave will vary with frequency. Generally the amplitude will go down as
you raise the frequency and go up as you lower the frequency.
WARNING: In the current implementation, the amplitude can blow up if the frequency is modulated
by certain alternating signals.
freq - frequency in Hertz

```
{ FSinOsc.ar(800, 0.0, 0.25) }.play;

{ FSinOsc.ar(XLine.kr(200,4000,1),0.0, 0.25) }.play;

// loses amplitude towards the end
{ FSinOsc.ar(FSinOsc.ar(XLine.kr(4,401,8),0.0, 200,800),0.0, 0.25) }.play;
```

ID: 706

Gendy1

An implementation of the dynamic stochastic synthesis generator conceived by Iannis Xenakis and described in Formalized Music (1992, Stuyvesant, NY: Pendragon Press) chapter 9 (pp 246-254) and chapters 13 and 14 (pp 289-322). The BASIC program in the book was written by Marie-Helene Serra so I think it helpful to credit her too.

The program code has been adapted to avoid infinities in the probability distribution functions.

The distributions are hard-coded in C but there is an option to have new amplitude or time breakpoints sampled from a continuous controller input.

Technical notes- X's plan as described in chapter 13 allows the 12 segments in the period to be successively modified with each new period. Yet the period is allowed to vary as the sum of the segment durations, as figure 1 demonstrates. We can setup some memory of n (conventionally 12) points, or even simply vary sucessively a single point's ordinate and duration. There are thus various schemes available to us. In one, fix period T and only move the (ti, Ei) within the period. In another, have a memory of 12 segments but allow continuous modification of the inter point intervals and the amplitudes. In yet another, just have one point and random walk its amplitude and duration based on the probability distribution. In this implementation I allow the user to initialise a certain number of memory points which is up to them. To restrict the period to be unchanging, you must set rate variation to zero (dscale=0).

Class Methods

***ar(ampdist=1, durdist=1, adparam=1.0, ddparam=1.0, minfreq=20, maxfreq=1000, ampscale= 0.5, durscale=0.5, initCPs=12, knum=12, mul=1.0, add=0.0)**

All parameters can be modulated at control rate except for initCPs which is used only at initialisation.

ampdist - Choice of probability distribution for the next perturbation of the amplitude of a control point.

The distributions are (adapted from the GENDYN program in Formalized Music):

0- LINEAR
1- CAUCHY
2- LOGIST
3- HYPERBCOS
4- ARCSINE
5- EXPON
6- SINUS
Where the sinus (Xenakis' name) is in this implementation taken as sampling from a third party oscillator. See example below.

durdist- Choice of distribution for the perturbation of the current inter control point duration.

adparam- A parameter for the shape of the amplitude probability distribution, requires values in the range 0.0001 to 1 (there are safety checks in the code so don't worry too much if you want to modulate!)

ddparam- A parameter for the shape of the duration probability distribution, requires values in the range 0.0001 to 1

minfreq- Minimum allowed frequency of oscillation for the Gendy1 oscillator, so gives the largest period the duration is allowed to take on.

maxfreq- Maximum allowed frequency of oscillation for the Gendy1 oscillator, so gives the smallest period the duration is allowed to take on.

ampscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for amplitude. An ampscale of 1.0 allows the full range of -1 to 1 for a change of amplitude.

durscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for duration. An ampscale of 1.0 allows the full range of -1 to 1 for a change of duration.

initCPs- Initialise the number of control points in the memory. Xenakis specifies 12. There would be this number of control points per cycle of the oscillator, though the oscillator's period will constantly change due to the duration distribution.

knum- Current number of utilised control points, allows modulation.

Examples

```
//warning- if you have lots of CPs and you have fast frequencies, the CPU cost goes up a lot because a
new CP move happens every sample!


//defaults
{Pan2.ar(Gendy1.ar)}.play

//wandering bass/ powerline
{Pan2.ar(Gendy1.ar(1,1,1.0,1.0,30,100,0.3,0.05,5))}.play

//play me
{Pan2.ar(RLPF.ar(Gendy1.ar(2,3,minfreq:20, maxfreq:MouseX.kr(100,1000), durscale:0.0, initCPs:40), 500,0.3,
0.2), 0.0)}.play

//scream!  - careful with your ears for this one!
(
{
var mx, my;

mx= MouseX.kr(220,440);
my= MouseY.kr(0.0,1.0);

Pan2.ar(Gendy1.ar(2,3,1,1,minfreq:mx, maxfreq:8*mx, ampscale:my, durscale:my, initCPs:7, mul:0.3), 0.0)}.play

)


//1 CP = random noise effect
{Pan2.ar(Gendy1.ar(initCPs:1))}.play

//2 CPs = suudenly an oscillator (though a fast modulating one here)
{Pan2.ar(Gendy1.ar(initCPs:2))}.play


//used as an LFO
(
{Pan2.ar(SinOsc.ar(Gendy1.kr(2,4,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51), 3.4,3.5, SinOsc.kr(0.17,0,0.49,0.51
SinOsc.kr(0.19,0,0.49,0.51),10,10,50, 350), 0, 0.3), 0.0)}.play
```

```
)

//wasp
{Pan2.ar(Gendy1.ar(0, 0, SinOsc.kr(0.1, 0, 0.1, 0.9),1.0, 50,1000, 1,0.005, 12, 12, 0.2), 0.0)}.play


//modulate distributions
//change of pitch as distributions change the duration structure and spectrum
{Pan2.ar(Gendy1.ar(MouseX.kr(0,7),MouseY.kr(0,7),mul:0.2), 0.0)}.play


//modulate num of CPs
{Pan2.ar(Gendy1.ar(knum:MouseX.kr(1,13),mul:0.2), 0.0)}.play


//Gendy into Gendy...with cartoon side effects
{Pan2.ar(Gendy1.ar(maxfreq:Gendy1.kr(5,4,0.3, 0.7, 0.1, MouseY.kr(0.1,10), 1.0, 1.0, 5,5, 500, 600), knum:MouseX.kr(1,13),mu
0.0)}.play


//use SINUS to track any oscillator and take CP positions from it, use adparam and ddparam as the in-
puts to sample
{Pan2.ar(Gendy1.ar(6,6,LFPulse.kr(100, 0, 0.4, 1.0), SinOsc.kr(30, 0, 0.5),mul:0.2), 0.0)}.play


//try out near the corners especially
{Pan2.ar(Gendy1.ar(6,6,LFPulse.kr(MouseX.kr(0,200), 0, 0.4, 1.0), SinOsc.kr(MouseY.kr(0,200), 0, 0.5),mul:0.2),
0.0)}.play


//texture
(
{
Mix.fill
var freq;

freq= rrand(130,160.3);
Pan2.ar(SinOsc.ar(Gendy1.ar(6.rand,6.rand,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51),freq
,freq, SinOsc.kr(0.17,0,0.49,0.51), SinOsc.kr(0.19,0,0.49,0.51), 12, 12, 200, 400), 0, 0.1), 1.0.rand2)
```

```
});
}.play
)

//wahhhhhhhh- try durscale 10.0 and 0.0 too
(
{Pan2.ar(
CombN
Resonz
Gendy1.ar(2,3,minfreq:1, maxfreq:MouseX.kr(10,700), durscale:0.1, initCPs:10),
MouseY.kr(50,1000), 0.1)
,0.1,0.1,5, 0.6
)
, 0.0)}.play
)


//overkill
(
{
var n;
n=10;

Mix.fill
var freq, numcps;

freq= rrand(130,160.3);
numcps= rrand(2,20);
Pan2.ar(Gendy1.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq ,freq, 1.0.rand, 1.0.rand, numcps, SinOsc.kr(exprand(0.02,0.2),
0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
});
}.play
)




//another traffic moment
(
```

```
{
var n;
n=10;

Resonz.ar(
Mix.fill
var freq, numcps;

freq= rrand(50,560.3);
numcps= rrand(2,20);
Pan2.ar(Gendy1.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq ,freq, 1.0.rand, 1.0.rand, numcps, SinOsc.kr(exprand(0.02,0.2),
0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
})
,MouseX.kr(100,2000), MouseY.kr(0.01,1.0))
;
}.play
)




(
{
var n;
n=15;

Out.ar(0,
Resonz
Mix.fill
var freq, numcps;

freq= rrand(330,460.3);
numcps= rrand(2,20);
Pan2.ar(Gendy1.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq,MouseX.kr(freq,2*freq), 1.0.rand, 1.0.rand, num-
cps, SinOsc.kr(exprand(0.02,0.2), 0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
})
,MouseX.kr(100,2000), MouseY.kr(0.01,1.0))
)

}.play;
```

)

```
//SuperCollider implementation by Nick Collins (sicklincoln.org)
```

ID: 707

Gendy2

See Gendy1 help file for background. This variant of GENDYN is closer to that presented in

Hoffmann, Peter. (2000) The New GENDYN Program. Computer Music Journal 24:2, pp 31-38.

Technical notes- random walk is of the amplitude and time delta, not the amp and time directly. The amplitude step random walk uses a lehmer style number generator whose parameters are accessible.

Class Methods

***ar(ampdist=1, durdist=1, adparam=1.0, ddparam=1.0, minfreq=20, maxfreq=1000, ampscale= 0.5, durscale=0.5, initCPs=12, knum=12, a= 1.17, c=0.31, mul=1.0, add=0.0)**

All parameters can be modulated at control rate except for initCPs which is used only at initialisation.

ampdist - Choice of probability distribution for the next perturbation of the amplitude of a control point.

The distributions are (adapted from the GENDYN program in Formalized Music):
0- LINEAR
1- CAUCHY
2- LOGIST
3- HYPERBCOS
4- ARCSINE
5- EXPON
6- SINUS
Where the sinus (Xenakis' name) is in this implementation taken as sampling from a third party oscillator. See example below.

durdist- Choice of distribution for the perturbation of the current inter control point duration.

adparam- A parameter for the shape of the amplitude probability distribution, requires values in the range 0.0001 to 1 (there are safety checks in the code so don't worry too much if you want to modulate!)

ddparam- A parameter for the shape of the duration probability distribution, requires values in the range 0.0001 to 1

minfreq- Minimum allowed frequency of oscillation for the Gendy1 oscillator, so gives the largest period the duration is allowed to take on.

maxfreq- Maximum allowed frequency of oscillation for the Gendy1 oscillator, so gives the smallest period the duration is allowed to take on.

ampscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for amplitude. An ampscale of 1.0 allows the full range of -1 to 1 for a change of amplitude.

durscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for duration. An ampscale of 1.0 allows the full range of -1 to 1 for a change of duration.

initCPs- Initialise the number of control points in the memory. Xenakis specifies 12. There would be this number of control points per cycle of the oscillator, though the oscillator's period will constantly change due to the duration distribution.

knum- Current number of utilised control points, allows modulation.

a- parameter for Lehmer random number generator perturbed by Xenakis as in ((old*a)+c)%1.0

c- parameter for Lehmer random number generator perturbed by Xenakis

Examples

```
//warning- if you have lots of CPs and you have fast frequencies, the CPU cost goes up a lot because a
new CP move happens every sample!


//LOUD! defaultslike a rougher Gendy1
{Pan2.ar(Gendy2.ar)}.play
```

```
//advantages of messing with the random number generation- causes periodicities
{Pan2.ar(Gendy2.ar(a:MouseX.kr(0.0,1.0),c:MouseY.kr(0.0,1.0)))}.play


(
{Pan2.ar(
Normalizer
RLPF.ar(
RLPF.ar(Gendy2.ar(a:SinOsc.kr(0.4,0,0.05,0.05),c:SinOsc.kr(0.3,0,0.1,0.5)),
MouseX.kr(10,10000,'exponential'),0.05),
MouseY.kr(10,10000,'exponential'),0.05)
,0.9)
,Lag.kr(LFNoise0.kr(1),0.5))}.play
)



{Pan2.ar(Gendy2.ar(3,5,1.0,1.0,50,1000,MouseX.kr(0.05,1),MouseY.kr(0.05,1),15, 0.05,0.51,mul:0.5))}.play



//play me
{Pan2.ar(RLPF.ar(Gendy2.ar(1,3,minfreq:20, maxfreq:MouseX.kr(100,1000), durscale:0.0, initCPs:4), 500,0.3,
0.2), 0.0)}.play


//1 CP = random noise effect
{Pan2.ar(Gendy2.ar(initCPs:1))}.play

//2 CPs = suudenly an oscillator (though a fast modulating one here)
{Pan2.ar(Gendy2.ar(initCPs:2))}.play


//used as an LFO
(
{Pan2.ar(SinOsc.ar(Gendy2.kr(2,1,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51), 3.4,3.5, SinOsc.kr(0.17,0,0.49,0.51
SinOsc.kr(0.19,0,0.49,0.51),10,10,mul:50, add:350), 0, 0.3), 0.0)}.play
)
```

```
//very angry wasp
{Pan2.ar(Gendy2.ar(0, 0, SinOsc.kr(0.1, 0, 0.1, 0.9),1.0, 50,1000, 1,0.005, 12, 12, 0.2,0.2,0.2), 0.0)}.play



//modulate distributions
//change of pitch as distributions change the duration structure and spectrum
{Pan2.ar(Gendy2.ar(MouseX.kr(0,7),MouseY.kr(0,7),mul:0.2), 0.0)}.play


//modulate num of CPs
{Pan2.ar(Gendy2.ar(knum:MouseX.kr(1,13),mul:0.2), 0.0)}.play


//Gendy1 into Gendy2...with cartoon side effects
{Pan2.ar(Gendy2.ar(maxfreq:Gendy1.kr(5,4,0.3, 0.7, 0.1, MouseY.kr(0.1,10), 1.0, 1.0, 5,5, 500, 600), knum:MouseX.kr(1,13),mu
0.0)}.play


//use SINUS to track any oscillator and take CP positions from it, use adparam and ddparam as the in-
puts to sample
{Pan2.ar(Gendy2.ar(6,6,LFPulse.kr(100, 0, 0.4, 1.0), SinOsc.kr(30, 0, 0.5),mul:0.2), 0.0)}.play


//try out near the corners especially
{Pan2.ar(Gendy2.ar(6,6,LFPulse.kr(MouseX.kr(0,200), 0, 0.4, 1.0), SinOsc.kr(MouseY.kr(0,200), 0, 0.5),mul:0.2),
0.0)}.play


//texture- the howling wind?
(
{
Mix.fill
var freq;

freq= rrand(130,160.3);
Pan2.ar(SinOsc.ar(Gendy2.ar(6.rand,6.rand,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51),freq
,freq, SinOsc.kr(0.17,0,0.49,0.51), SinOsc.kr(0.19,0,0.49,0.51), 12, 12, 0.4.rand, 0.4.rand, 200, 400),
0, 0.1), 1.0.rand2)
});
```

```
}.play
)

//CAREFUL! mouse to far right causes explosion of sound
(
{Pan2.ar(
CombN
Resonz
Gendy2.ar(2,3,minfreq:1, maxfreq:MouseX.kr(10,700), initCPs:100),
MouseY.kr(50,1000), 0.1)
,0.1,0.1,5, 0.16
)
, 0.0)}.play
)



//storm
(
{
var n;
n=15;

0.5*Mix.fill(n,{
var freq, numcps;

freq= rrand(130,160.3);
numcps= rrand(2,20);
Pan2.ar(Gendy2.ar(6.rand,6.rand,10.0.rand,10.0.rand,freq,freq*exprand(1.0,2.0), 10.0.rand, 10.0.rand,
numcps, SinOsc.kr(exprand(0.02,0.2), 0, numcps/2, numcps/2), 10.0.rand,10.0.rand,0.5/(n.sqrt)), 1.0.rand2)

});
}.play
)



//another traffic moment
(
{
var n;
```

```
n=10;

Resonz
Mix.fill
var freq, numcps;

freq= rrand(50,560.3);
numcps= rrand(2,20);
Pan2.ar(Gendy2.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq ,freq, 1.0.rand, 1.0.rand, numcps, SinOsc.kr(exprand(0.02,0.2),
0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
})
,MouseX.kr(100,2000), MouseY.kr(0.01,1.0), 0.3)
;
}.play
)


//SuperCollider implementation by Nick Collins (sicklincoln.org)
```

ID: 708

Gendy3

See Gendy1 help file for background. This variant of GENDYN normalises the durations in each period to force oscillation at the desired pitch. The breakpoints still get perturbed as in Gendy1.

There is some glitching in the oscillator caused by the stochastic effects- control points as they vary cause big local jumps of amplitude. Put ampscale and durscale low to minimise the rate of this.

Class Methods

*ar(ampdist=1, durdist=1, adparam=1.0, ddparam=1.0, freq=440, ampscale=0.5, durscale=0.5, initCPs=12, knum=12, mul=1.0, add=0.0)

All parameters can be modulated at control rate except for initCPs which is used only at initialisation.

ampdist - Choice of probability distribution for the next perturbation of the amplitude of a control point.

The distributions are (adapted from the GENDYN program in Formalized Music):
0- LINEAR
1- CAUCHY
2- LOGIST
3- HYPERBCOS
4- ARCSINE
5- EXPON
6- SINUS
Where the sinus (Xenakis' name) is in this implementation taken as sampling from a third party oscillator. See example below.

durdist- Choice of distribution for the perturbation of the current inter control point duration.

adparam- A parameter for the shape of the amplitude probability distribution, requires values in the range 0.0001 to 1 (there are safety checks in the code so don't worry too much if you want to modulate!)

ddparam- A parameter for the shape of the duration probability distribution, requires values in the range 0.0001 to 1

freq- Oscillation frquency.

ampscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for amplitude. An ampscale of 1.0 allows the full range of -1 to 1 for a change of amplitude.

durscale- Normally 0.0 to 1.0, multiplier for the distribution's delta value for duration. An ampscale of 1.0 allows the full range of -1 to 1 for a change of duration.

initCPs- Initialise the number of control points in the memory. Xenakis specifies 12. There would be this number of control points per cycle of the oscillator, though the oscillator's period will constantly change due to the duration distribution.

knum- Current number of utilised control points, allows modulation.

Examples

```
//warning- if you have lots of CPs and you have fast frequencies, the CPU cost goes up a lot because a new CP move happens every sample!


//LOUD! defaults like a rougher Gendy1
{Pan2.ar(Gendy3.ar(mul:0.5))}.play

{Pan2.ar(Gendy3.ar(freq:MouseX.kr(220,880,'exponential'), durscale:0.01, ampscale:0.02, mul:0.2))}.play


//stochastic waveform distortion- also play me at the same time as the previous example...
{Pan2.ar(Gendy3.ar(1,2,0.3,-0.7,MouseX.kr(55,110,'exponential'),0.03,0.1))}.play


(
{Pan2.ar(
Normalizer
RLPF.ar(
```

```
RLPF.ar(Mix.new(Gendy3.ar(freq:[230, 419, 546, 789])),
MouseX.kr(10,10000,'exponential'),0.05),
MouseY.kr(10,10000,'exponential'),0.05)
,0.9)
,Lag.kr(LFNoise0.kr(1),0.5))}.play
)


//concrete pH?
(
{Pan2.ar(
Mix.new(Gendy3.ar(freq:([1,1.2,1.3,1.76,2.3]*MouseX.kr(3,17,'exponential')),mul:0.2)))}.play
)

//glitch low, mountain high
(
{Pan2.ar(
Mix.new(Gendy3.ar(3,5,1.0,1.0,(Array.fill(5,{LFNoise0.kr(1.3.rand,1,2)})*MouseX.kr(100,378,'exponential')),MouseX.kr(0.01,0.

)

//play me
{Pan2.ar(RLPF.ar(Gendy3.ar(1,3,freq:MouseX.kr(100,1000), durscale:0.0, ampscale:MouseY.kr(0.0,0.1), initCPs:7,
knum:  MouseY.kr(7,2)), 500,0.3, 0.2), 0.0)}.play



//used as an LFO
(
{Pan2.ar(SinOsc.ar(Gendy3.kr(2,5,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51), 0.34, SinOsc.kr(0.17,0,0.49,0.51),
SinOsc.kr(0.19,0,0.49,0.51),10,10,mul:50, add:350), 0, 0.3), 0.0)}.play
)

//buzzpipes
{Pan2.ar(Mix.new(Gendy3.ar(0, 0, SinOsc.kr(0.1, 0, 0.1, 0.9),1.0, [100,205,410], 0.011,0.005, 12, 12,
0.12)), 0.0)}.play


//modulate distributions
//change of pitch as distributions change the duration structure and spectrum
```

```
{Pan2.ar(Gendy3.ar(MouseX.kr(0,7),MouseY.kr(0,7),mul:0.2), 0.0)}.play


//modulate num of CPs
{Pan2.ar(Gendy3.ar(knum:MouseX.kr(2,13),mul:0.2), 0.0)}.play


//Gendy1 into Gendy2 into Gendy3...with cartoon side effects
(
{Pan2.ar(Gendy3.ar(1,2,freq:Gendy2.ar(maxfreq:Gendy1.kr(5,4,0.3, 0.7, 0.1, MouseY.kr(0.1,10), 1.0, 1.0,
5,5, 25,26),minfreq:24, knum:MouseX.kr(1,13),mul:150, add:200), durscale:0.01, ampscale:0.01, mul:0.1),
0.0)}.play
)

//use SINUS to track any oscillator and take CP positions from it, use adparam and ddparam as the in-
puts to sample
{Pan2.ar(Gendy3.ar(6,6,LFPulse.kr(LFNoise0.kr(19.0,0.5,0.6), 0, 0.4, 0.5), Gendy1.kr(durscale:0.01,ampscale:0.01),
MouseX.kr(10,100),mul:0.2), 0.0)}.play



//wolf tones
(
{
Mix.fill
var freq;

freq= exprand(130,1160.3);
Pan2.ar(SinOsc.ar(Gendy3.ar(6.rand,6.rand,SinOsc.kr(0.1,0,0.49,0.51),SinOsc.kr(0.13,0,0.49,0.51),freq,
SinOsc.kr(0.17,0,0.0049,0.0051), SinOsc.kr(0.19,0,0.0049,0.0051), 12, 12, 200, 400), 0, 0.1), 1.0.rand2)

});
}.play
)

//CAREFUL! mouse to far right causes explosion of sound-
//notice how high frequency and num of CPs affects CPU cost
(
{Pan2.ar(
CombN
```

```
Resonz
Gendy3.ar(2,3,freq:MouseX.kr(10,700), initCPs:100),
MouseY.kr(50,1000), 0.1)
,0.1,0.1,5, 0.16
)
, 0.0)}.play
)


//storm
(
{
var n;
n=15;

0.5*Mix.fill(n,{
var freq, numcps;

freq= rrand(130,160.3);
numcps= rrand(2,20);
Pan2.ar(Gendy3.ar(6.rand,6.rand,10.0.rand,10.0.rand,freq*exprand(1.0,2.0), 10.0.rand, 10.0.rand, num-
cps, SinOsc.kr(exprand(0.02,0.2), 0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
});
}.play
)



//another glitchy moment
(
{
var n;
n=10;

Resonz
Mix.fill
var freq, numcps;

freq= rrand(50,560.3);
numcps= rrand(2,20);
```

```
Pan2.ar(Gendy3.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq, 1.0.rand, 1.0.rand, numcps, SinOsc.kr(exprand(0.02,0.2),
0, numcps/2, numcps/2), 0.5/(n.sqrt)), 1.0.rand2)
})
,MouseX.kr(100,2000), MouseY.kr(0.01,1.0), 0.3)
;
}.play
)


//SuperCollider implementation by Nick Collins (sicklincoln.org)
```

ID: 709

Impulse impulse oscillator

Impulse.ar(freq, phase, mul, add)

Outputs non band limited single sample impulses.
freq - frequency in Hertz
phase - phase offset in cycles (0..1)

```
{ Impulse.ar(800, 0.0, 0.5, 0) }.play
```

```
{ Impulse.ar(XLine.kr(800,100,5), 0.0,  0.5, 0) }.play
```

modulate phase:

```
{ Impulse.ar(4, [0, MouseX.kr(0, 1)], 0.2) }.play;
```

ID: 710

Index index into a table with a signal

Index.ar(bufnum, in, mul, add)
Index.kr(bufnum, in, mul, add)

The input signal value is truncated to an integer value and used as an index into the table.
Out of range index values are clipped to the valid range.
bufnum - index of the buffer
in - the input signal.

```
(
// indexing into a table
s = Server.local;
t = [ 200, 300, 400, 500, 600, 800 ];
b = Buffer(s,t.size,1);

// alloc and set the values
s.listSendMsg( b.allocMsg( b.setnMsg(0, t) ).postln );

SynthDef("help-Index",{ arg out=0,i_bufnum=0;
Out.ar(0,
SinOsc.ar(
Index.kr(
i_bufnum,
LFSaw.kr(2.0).range(0.0,7.0)
),
0,
0.5
)
)
}).play(s,[\i_bufnum,b.bufnum]);

)
```

```
/*

(
// indexing into a table
var table;
table = [ 200, 300, 400, 500, 600, 800 ];
b =
Synth.play({
SinOsc.ar(
Index.kr(
table,
MouseX.kr(0, table.size)
),
0,
0.1
)
});
)


*/
```

ID: 711

Klang sine oscillator bank

Klang.ar(specificationsArrayRef, freqScale, freqOffset)

Klang is a bank of fixed frequency sine oscillators. Klang is more efficient than creating individual oscillators but offers less flexibility.

specificationsArrayRef - a Ref to an Array of three Arrays :
frequencies - an Array of filter frequencies.
amplitudes - an Array of filter amplitudes, or nil. If nil, then amplitudes default to 1.0
phases - an Array of initial phases, or nil. If nil, then phases default to 0.0
freqScale - a scale factor multiplied by all frequencies at initialization time.
freqOffset - an offset added to all frequencies at initialization time.

```
play({ Klang.ar(`[ [800, 1000, 1200],[0.3, 0.3, 0.3],[pi,pi,pi]], 1, 0) * 0.4});

play({ Klang.ar(`[ [800, 1000, 1200], nil, nil], 1, 0) * 0.25});

play({ Klang.ar(`[ Array.rand(12, 600.0, 1000.0), nil, nil ], 1, 0) * 0.05 });


//////////////
s.boot;

(
Routine
loop({
play({
Pan2.ar(Klang.ar(`[ Array.rand(12, 200.0, 2000.0), nil, nil ], 1, 0), 1.0.rand)
* EnvGen.kr(Env.sine(4), 1, 0.02, doneAction:  2);
});
2.wait;
})
}).play;
)
```

ID: 712

LFCub

- a sine like shape made of two cubic pieces. smoother than LFPar.

```
freq = 440.0, iphase = 0.0, mul = 1.0, add = 0.0
```

```
{ LFCub.ar(LFCub.kr(LFCub.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFCub.ar(LFCub.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFCub.ar(800,0,0.1) }.play
{ LFCub.ar(XLine.kr(100,8000,30),0,0.1) }.play
```

```
compare:
```

```
{ LFPar.ar(LFPar.kr(LFPar.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFPar.ar(LFPar.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFPar.ar(800,0,0.1) }.play
{ LFPar.ar(XLine.kr(100,8000,30),0,0.1) }.play
```

```
{ SinOsc.ar(SinOsc.kr(SinOsc.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ SinOsc.ar(SinOsc.kr(0.2, 0, 400,800),0,0.1) }.play
{ SinOsc.ar(800,0,0.1) }.play
{ SinOsc.ar(XLine.kr(100,8000,30),0,0.1) }.play

{ LFTri.ar(LFTri.kr(LFTri.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFTri.ar(LFTri.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFTri.ar(800,0,0.1) }.play
{ LFTri.ar(XLine.kr(100,8000,30),0,0.1) }.play
```

ID: 713

LFPar

- a sine-like shape made of two parabolas. has audible odd harmonics.

```
{ LFPar.ar(LFPar.kr(LFPar.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFPar.ar(LFPar.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFPar.ar(800,0,0.1) }.play
{ LFPar.ar(XLine.kr(100,8000,30),0,0.1) }.play
```

compare:

```
{ LFCub.ar(LFCub.kr(LFCub.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFCub.ar(LFCub.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFCub.ar(800,0,0.1) }.play
{ LFCub.ar(XLine.kr(100,8000,30),0,0.1) }.play

{ SinOsc.ar(SinOsc.kr(SinOsc.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ SinOsc.ar(SinOsc.kr(0.2, 0, 400,800),0,0.1) }.play
{ SinOsc.ar(800,0,0.1) }.play
{ SinOsc.ar(XLine.kr(100,8000,30),0,0.1) }.play

{ LFTri.ar(LFTri.kr(LFTri.kr(0.2,0,8,10),0, 400,800),0,0.1) }.play
{ LFTri.ar(LFTri.kr(0.2, 0, 400,800),0,0.1) }.play
{ LFTri.ar(800,0,0.1) }.play
{ LFTri.ar(XLine.kr(100,8000,30),0,0.1) }.play
```

ID: 714

LFPulse pulse oscillator

LFPulse.ar(freq, iphase,width, mul, add)

A non-band-limited pulse oscillator. Outputs a high value of one and a low value of zero.
freq - frequency in Hertz
iphase - initial phase offset in cycles (0..1)
width - pulse width duty cycle from zero to one.

```
//Synth.plot({ LFPulse.ar(500, 0, 0.3, 1, 0) });


// used as both Oscillator and LFO:
play({ LFPulse.ar(LFPulse.kr(3, 0, 0.3, 200, 200), 0, 0.2, 0.1) });

compare:
//plot({
// [Pulse.ar(10.0, 0.3, 1, 0),
//   LFPulse.ar(10.0, 0.3, 1, 0)]
//},1.0);
```

ID: 715

LFSaw sawtooth oscillator

LFSaw.ar(freq, iphase, mul, add)

A non-band-limited sawtooth oscillator. Output ranges from -1 to +1.

freq - frequency in Hertz
iphase - initial phase offset. For efficiency reasons this is a value ranging from 0 to 2.

```
{ LFSaw.ar(500, 1, 0.1) }.play

// used as both Oscillator and LFO:
{ LFSaw.ar(LFSaw.kr(4, 0, 200, 400), 0, 0.1) }.play
```

ID: 716

LFTri triangle oscillator

LFTri.ar(freq, iphase, mul, add)

A non-band-limited triangle oscillator. Output ranges from -1 to +1.
freq - frequency in Hertz
iphase - initial phase offset. For efficiency reasons this is a value ranging from 0 to 4.

```
{ LFTri.ar(500, 0, 0.1) }.play

// used as both Oscillator and LFO:
{ LFTri.ar(LFTri.kr(4, 0, 200, 400), 0, 0.1) }.play
```

ID: 717

Osc interpolating wavetable oscillator

Osc.ar(table, freq, phase, mul, add)

Linear interpolating wavetable lookup oscillator with frequency and phase modulation inputs.

This oscillator requires a buffer to be filled with a wavetable format signal. This pre-processes the Signal into a form which can be used efficiently by the Oscillator. The buffer size must be a power of 2.

This can be acheived by creating a Buffer object and sending it one of the "b_gen" messages (sine1, sine2, sine3) with the wavetable flag set to true.

This can also be acheived by creating a Signal object and sending it the 'asWavetable' message, saving it to disk, and having the server load it from there.

table - buffer index
freq - frequency in Hertz
phase - phase offset or modulator in radians

note about wavetables:
OscN requires the b_gen sine1 wavetable flag to be OFF.
Osc requires the b_gen sine1 wavetable flag to be ON.

```
(
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);

SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum, 200, 0, 0.5)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
```

```
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);

SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum, XLine.kr(2000,200), 0, 0.5)// modulate freq
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)


(
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1([1.0], true, true, true);

SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum,
Osc.ar(bufnum,
XLine.kr(1,1000,9),
0,
200,
800),
0,
0.25)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)


(
// modulate phase
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1([1.0], true, true, true);

SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
```

```
Osc.ar(bufnum,
800,
Osc.ar(bufnum,
XLine.kr(20,8000,10),
0,
2pi),
0.25)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)



(
// change the buffer while its playing
s = Server.local;
b = Buffer.alloc(s, 4096, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);

SynthDef("help-Osc",{ arg out=0,bufnum=0;
Out.ar(out,
Osc.ar(bufnum, [80,80.2], 0, 0.2)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
fork {
var n = 32;
50.do {
b.sine1(Array.rand(n,0,1).cubed, true, true, true);
0.25.wait;
};
};
)
```

ID: 718

OscN noninterpolating wavetable oscillator

OscN.ar(table, freq, phase, mul, add)

Noninterpolating wavetable lookup oscillator with frequency and phase modulation inputs.
It is usually better to use the interpolating oscillator.
buffer - buffer index. the buffer size must be a power of 2. The buffer should NOT be filled using Wavetable format (b_gen commands should set wavetable flag to false. Raw signals (not converted with asWavetable) can be saved to disk and loaded into the buffer.
freq - frequency in Hertz
phase - phase offset or modulator in radians

```
// compare examples below with interpolating Osc examples.

(
s = Server.local;
b = Buffer.alloc(s,512,1);
b.sine1(1.0/[1,2,3,4,5,6],true,false,true);

SynthDef("help-OscN",{ arg out=0,bufnum=0;
Out.ar(out,
OscN.ar(bufnum, 500, 0, 0.5)
)
}).play(s,[0,0,1,b.bufnum]);

)
b.free;



(
// noninterpolating - there are noticeable artifacts
// modulate freq

s = Server.local;
b = Buffer.alloc(s,512,1);
b.sine1(1.0/[1,2,3,4,5,6].squared,true,false,true);
```

```
SynthDef("help-OscN",{ arg out=0,bufnum=0;
Out.ar(out,
OscN.ar(bufnum, XLine.kr(2000,200), 0, 0.5)
)
}).play(s,[\out,0,\bufnum,b.bufnum]);

)
b.free;

(
// sounds very different than the Osc example
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1([1.0], true, true, true);

SynthDef("help-OscN",{ arg out=0,bufnum=0;
Out.ar(out,
OscN.ar(bufnum,
OscN.ar(bufnum,
XLine.kr(1,1000,9),
0,
200,
800),
0,
0.25)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);

)
b.free;

(
// modulate phase
s = Server.local;
b = Buffer.alloc(s, 512, 1);
b.sine1([1.0], true, true, true);

SynthDef("help-OscN",{ arg out=0,bufnum=0;
Out.ar(out,
```

```
OscN.ar(bufnum,
800,
OscN.ar(bufnum,
XLine.kr(20,8000,10),
0,
2pi),
0.25)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)
b.free;


(
// change the buffer while its playing
s = Server.local;
b = Buffer.alloc(s, 4096, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);

SynthDef("help-OscN",{ arg out=0,bufnum=0;
Out.ar(out,
OscN.ar(bufnum, [80,80.2], 0, 0.2)
)
}).play(s,[\out, 0, \bufnum, b.bufnum]);
)

(
Routine
var n = 32;
50.do({
b.sine1(Array.rand(n,0,1).cubed, true, true, true);
0.25.wait;
});
}).play;
)
b.free;
```

ID: 719

PMOsc phase modulation oscillator pair

PMOsc.ar(carfreq, modfreq, index, modphase, mul, add)

Phase modulation sine oscillator pair.
carfreq - carrier frequency in cycles per second.
modfreq - modulator frequency in cycles per second.
index - modulation index in radians.
modphase - a modulation input for the modulator's phase in radians

```
play({ PMOsc.ar(Line.kr(600, 900, 5), 600, 3, 0, 0.1) }); // modulate carfreq

play({ PMOsc.ar(300, Line.kr(600, 900, 5), 3, 0, 0.1) }); // modulate modfreq

play({ PMOsc.ar(300, 550, Line.ar(0,20,8), 0, 0.1) }); // modulate index

(
e = Env.linen(2, 5, 2);
Routine.run({
loop({
play({
LinPan2.ar(EnvGen.ar(e) *
PMOsc.ar(2000.0.rand,800.0.rand, Line.kr(0.0, 12.0.rand,9),0,0.1), 1.0.rand2)});
2.wait;
})
}))
```

ID: 720

Pulse band limited pulse wave

Pulse.ar(kfreq, kwidth, mul, add)

Band limited pulse wave generator with pulse width modulation.
kfreq - frequency in Hertz
kwidth - pulse width ratio from zero to one. 0.5 makes a square wave.

```
// modulate frequency
{ Pulse.ar(XLine.kr(40,4000,6),0.1, 0.2) }.play;

// modulate pulse width
{ Pulse.ar(200, Line.kr(0.01,0.99,8), 0.2) }.play;

// two band limited square waves thru a resonant low pass filter
{ RLPF.ar(Pulse.ar([100,250],0.5,0.1), XLine.kr(8000,400,5), 0.05) }.play;
```

ID: 721

Saw band limited sawtooth

Saw.ar(kfreq, mul, add)

Band limited sawtooth wave generator.
kfreq - frequency in Hertz

```
// modulating the frequency

{ Saw.ar(XLine.kr(40,4000,6),0.2) }.play;

// two band limited sawtooth waves through a resonant low pass filter

{ RLPF.ar(Saw.ar([100,250],0.1), XLine.kr(8000,400,5), 0.05) }.play;
```

ID: 722

Shaper wave shaper

Shaper.ar(bufnum, in, mul, add)

Performs waveshaping on the input signal by indexing into the table.
bufnum - the number of a buffer filled in wavetable format containing the transfer function.
in - the input signal.

```
Server.default = s = Server.internal; s.boot;
b = Buffer.alloc(s, 512, 1, {arg buf; buf.chebyMsg([1,0,1,1,0,1])});
(
scope({
 Shaper
b.bufnum,
SinOsc.ar(300, 0, Line.kr(0,1,6)),
0.5
)
});
)
```

ID: 723

SinOsc interpolating sine wavetable oscillator

SinOsc.ar(freq, phase, mul, add)

This is the same as Osc except that the table is a sine table of 8192 entries.
freq - frequency in Hertz
phase - phase offset or modulator in radians

```
{ SinOsc.ar(200, 0, 0.5) }.play;


// modulate freq
{ SinOsc.ar(XLine.kr(2000, 200), 0, 0.5) }.play;

// modulate freq
{ SinOsc.ar(SinOsc.ar(XLine.kr(1, 1000, 9), 0, 200, 800), 0, 0.25) }.play;


// modulate phase
{ SinOsc.ar(800, SinOsc.ar(XLine.kr(1, 1000, 9), 0, 2pi), 0.25) }.play;
```

ID: 724

SyncSaw hard sync sawtooth wave

SyncSaw.ar(ksyncFreq, ksawFreq, mul, add)

A sawtooth wave that is hard synched to a fundamental pitch. This produces an effect similar to moving formants or pulse width modulation. The sawtooth oscillator has its phase reset when the sync oscillator completes a cycle. This is not a band limited waveform, so it may alias.

ksyncFreq - frequency of the fundamental.
ksawFreq - frequency of the slave synched sawtooth wave. sawFreq should always be greater than syncFreq.

```
{ SyncSaw.ar(100, Line.kr(100, 800, 12), 0.1) }.play;


(
plot { [
SyncSaw.ar(800, 1200),
 Impulse         // to show sync rate
] }
)

(
plot { [
SyncSaw.ar(800, Line.kr(800, 1600, 0.01)), // modulate saw freq
 Impulse         // to show sync rate
] }
)

// scoping the saw:  hit 's' when focused on the scope window to compare the channels
(
scope {
var freq = 400;
[
SyncSaw.ar(freq, freq * MouseX.kr(1, 3)), // modulate saw freq
```

```
 Impulse          // to show sync rate
] * 0.3 }
)
```

ID: 725

TWindex triggered windex

*kr(trig,array,normalize)

When triggered, returns a random index value based on array as a list of probabilities. by default the list of probabilities should sum to 1.0, when the normalize flag is set to 1, the values get normalized by the ugen (less efficient)

```
//assuming normalized values
(

a = SynthDef("help-TWindex",{ arg w1=0.0, w2=0.5, w3=0.5;

var trig, index;
trig = Impulse.kr(6);
index = TWindex.kr(trig, [w1, w2, w3]);

Out.ar(0,
SinOsc.ar(
Select.kr(index,[400, 500, 600]),
0, 0.2
)
)
}).play;

)

a.setn(0, [0,0,1].normalizeSum);
a.setn(0, [1,1,1].normalizeSum);
a.setn(0, [1,0,1].normalizeSum);


//modulating probability values
(

a = SynthDef("help-TWindex",{ arg w1=0.0, w2=0.5;
```

```
var trig, index;
trig = Impulse.kr(6);
index = TWindex.kr(
trig,
[w1, w2, SinOsc.kr(0.3, 0, 0.5, 0.5)],//modulate probability
     1 //do normalisation
);

Out.ar(0,
SinOsc.ar(
Select.kr(index,[400, 500, 600]),
0, 0.2
)
)
}).play;

)

a.setn(0, [0,0]);
a.setn(0, [1,1]);
a.setn(0, [1,0]);
a.setn(0, [0,1]);
```

ID: 726

VarSaw

variable duty saw

```
play({
 VarSaw
LFPulse.kr(3, 0, 0.3, 200, 200),
0,
LFTri.kr(1.0).range(0,1), //width
0.1)
});


play({ VarSaw.ar(LFPulse.kr(3, 0, 0.3, 200, 200), 0, 0.2, 0.1) });

compare:

play({ LFPulse.ar(LFPulse.kr(3, 0, 0.3, 200, 200), 0, 0.2, 0.1) });
```

ID: 727

VOsc variable wavetable oscillator

VOsc.ar(table, freq, phase, mul, add)

A wavetable lookup oscillator which can be swept smoothly across wavetables. All the wavetables must be allocated to the same size. Fractional values of table will interpolate between two adjacent tables.

This oscillator requires at least two buffers to be filled with a wavetable format signal. This preprocesses the Signal into a form which can be used efficiently by the Oscillator. The buffer size must be a power of 2.

This can be acheived by creating a Buffer object and sending it one of the "b_gen" messages (sine1, sine2, sine3) with the wavetable flag set to true.

This can also be acheived by creating a Signal object and sending it the 'asWavetable' message, saving it to disk, and having the server load it from there.

If you use Buffer objects to manage buffer numbers, you can use the *allocConsecutive method to allocate a continuous block of buffers. See the **[Buffer]** helpfile for details.

table - buffer index. Can be swept continuously among adjacent wavetable buffers of the same size.
freq - frequency in Hertz
phase - phase offset or modulator in radians

note about wavetables:
VOsc requires the b_gen sine1 wavetable flag to be ON.

```
(
s = Server.local;
// allocate and fill tables 0 to 7
8.do({ arg i;
var n, a;
 // allocate table
s.sendMsg(\b_alloc, i, 1024);
 // generate array of harmonic amplitudes
```

```
n = (i+1)**2;
a = Array.fill(n, { arg j; ((n-j)/n).squared.round(0.001) });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)

(
SynthDef("help-VOsc",{ arg out=0, bufoffset=0;
var x;
// mouse x controls the wavetable position
x = MouseX.kr(0,7);
Out.ar(out,
VOsc.ar(bufoffset+x, [120,121], 0, 0.3)
)
}).play(s,[\out, 0, \bufoffset, 0]);
)

(
8.do({ arg i;
var a;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
a = Array.fill(i, 0) ++ [0.5, 1, 0.5];
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)

(
8.do({ arg i;
var a;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
a = Array.fill(32,0);
12.do({ arg i; a.put(32.rand, 1) });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)
```

```
(
8.do({ arg i;
var a;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
n = (i+1)**2;
a = Array.fill(n, { arg j; 1.0.rand2 });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)
```

ID: 728

VOsc3 three variable wavetable oscillators

VOsc3.ar(table, freq1, freq2, freq3, mul, add)

A wavetable lookup oscillator which can be swept smoothly across wavetables. All the wavetables must be allocated to the same size. Fractional values of table will interpolate between two adjacent tables. This unit generator contains three oscillators at different frequencies, mixed together.

This oscillator requires at least two buffers to be filled with a wavetable format signal. This preprocesses the Signal into a form which can be used efficiently by the Oscillator. The buffer size must be a power of 2.

This can be achieved by creating a Buffer object and sending it one of the "b_gen" messages (sine1, sine2, sine3) with the wavetable flag set to true.

This can also be achieved by creating a Signal object and sending it the 'asWavetable' message, saving it to disk, and having the server load it from there.

If you use Buffer objects to manage buffer numbers, you can use the *allocConsecutive method to allocate a continuous block of buffers. See the **[Buffer]** helpfile for details.

table - buffer index. Can be swept continuously among adjacent wavetable buffers of the same size.
freq1, freq1, freq3 - frequencies in Hertz of the three oscillators
phase - phase offset or modulator in radians

note about wavetables:
VOsc3 requires the b_gen sine1 wavetable flag to be ON.

```
(
s = Server.local;
// allocate and fill tables 0 to 7
8.do({ arg i;
var n, a;
 // allocate table
s.sendMsg(\b_alloc, i, 1024);
```

```
 // generate array of harmonic amplitudes
n = (i+1)**2;
a = Array.fill(n, { arg j; ((n-j)/n).squared.round(0.001) });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)

(
SynthDef("help-VOsc",{ arg out=0, bufoffset=0, freq=240;
var x;
// mouse x controls the wavetable position
x = MouseX.kr(0,7);
Out.ar(out,
VOsc3.ar(bufoffset+x, freq+[0,1],freq+[0.37,1.1],freq+[0.43, -0.29], 0.3)
)
}).play(s,[\out, 0, \bufoffset, 0]);
)

(
8.do({ arg i;
var a;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
a = Array.fill(i, 0) ++ [0.5, 1, 0.5];
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)

(
8.do({ arg i;
var a, n;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
 n = (i+1)*8;
a = Array.fill(n,0);
(n>>1).do({ arg i; a.put(n.rand, 1) });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
```

```
});
)

(
8.do({ arg i;
var a;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
n = (i+1)**2;
a = Array.fill(n, { arg j; 1.0.rand2 });
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a);
});
)

(
var a;
a = Array.fill(64, { arg j; 1.0.rand2 });
8.do({ arg i;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
n = (i+1)**2;
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a.extend(n.asInteger));
});
)

(
var a;
a = Array.fill(64, { arg j; 1/(j+1) });
8.do({ arg i;
s.sendMsg(\b_alloc, i, 1024); // allocate table
 // generate array of harmonic amplitudes
n = (i+1)**2;
 // fill table
s.performList(\sendMsg, \b_gen, i, \sine1, 7, a.extend(n.asInteger));
});
)
```

25.14 Panners

ID: 729

BiPanB2 2D Ambisonic B-format panner

BiPanB2.kr(inA, inB, azimuth, gain)

Encode a two channel signal to two dimensional ambisonic B-format.
This puts two channels at opposite poles of a 2D ambisonic field.
This is one way to map a stereo sound onto a soundfield.
It is equivalent to:

```
PanB2(inA, azimuth, gain) + PanB2(inB, azimuth + 1, gain)
```

inA - input signal A
inB - input signal B
azimuth - position around the circle from -1 to +1.
-1 is behind, -0.5 is left, 0 is forward, +0.5 is right, +1 is behind.
gain - amplitude control

```
(
{
var w, x, y, p, q, a, b, c, d;

p = LFSaw.ar(200);
q = LFSaw.ar(301);

 // B-format encode
#w, x, y = BiPanB2.ar(p, q, MouseX.kr(-1,1), 0.1);

 // B-format decode to quad
#a, b, c, d = DecodeB2.ar(4, w, x, y);

 [a, b, d, c] // reorder to my speaker arrangement:  Lf Rf Lr Rr
}.play;
)
```

ID: 730

DecodeB2 2D Ambisonic B-format decoder

DecodeB2.kr(numChans, w, x, y, orientation)

Decode a two dimensional ambisonic B-format signal to a set of speakers in a regular polygon.
The outputs will be in clockwise order. The position of the first speaker is either center or left of center.
numChans - number of output speakers. Typically 4 to 8.
w, x, y - the B-format signals.
orientation - Should be zero if the front is a vertex of the polygon. The first speaker will be directly in front. Should be 0.5 if the front bisects a side of the polygon. Then the first speaker will be the one left of center. Default is 0.5.

```
(
{
var w, x, y, p, a, b, c, d;

p = PinkNoise.ar; // source

 // B-format encode
#w, x, y = PanB2.ar(p, MouseX.kr(-1,1), 0.1);

 // B-format decode to quad
#a, b, c, d = DecodeB2.ar(4, w, x, y);

 [a, b, d, c] // reorder to my speaker arrangement:  Lf Rf Lr Rr
}.play;
)
```

ID: 731

LinPan2 two channel linear pan

LinPan2.ar(in, pos, level)

Two channel linear panner. This one sounds more like the Rhodes tremolo than **Pan2**.
in - input signal
pos - pan position, -1 is left, +1 is right
level - a control rate level input

```
play({ LinPan2.ar(PinkNoise.ar(0.4), FSinOsc.kr(2)) });

SynthDef("help-LinPan2", {  Out.ar(0, LinPan2.ar(FSinOsc.ar(800, 0, 0.1), FSinOsc.kr(3))) }).play;
```

ID: 732

LinXFade2 two channel linear crossfade

LinXFade2.ar(inA, inB, pan, level)

Two channel linear crossfader.

inA - an input signal
inB - another input signal
pan - cross fade position from -1 to +1
level - a control rate level input

```
play({ LinXFade2.ar(FSinOsc.ar(800, 0, 0.2), PinkNoise.ar(0.2), FSinOsc.kr(1)) });
```

ID: 733

Pan2 two channel equal power pan

Pan2.ar(in, pos, level)

Two channel equal power panner.
in - input signal
pos - pan position, -1 is left, +1 is right
level - a control rate level input

```
SynthDef("help-Pan2", { Out.ar(0, Pan2.ar(PinkNoise.ar(0.4), FSinOsc.kr(2), 0.3)) }).play;
```

ID: 734

Pan4 four channel equal power pan

Pan4.ar(in, xpos, ypos, level)

Four channel equal power panner.
in - input signal
xpos - x pan position from -1 to +1(left to right)
ypos - y pan position from -1 to +1 (back to front)
level - a control rate level input.

Outputs are in order LeftFront, RightFront, LeftBack, RightBack.

```
// You'll only hear the front two channels on a stereo setup.
(
SynthDef("help-Pan4", {
 Out.ar(0, Pan4.ar(PinkNoise.ar, FSinOsc.kr(2), FSinOsc.kr(1.2), 0.3))
}).play;
)

play({ Pan4.ar(PinkNoise.ar, -1,  0, 0.3) }); // left pair
play({ Pan4.ar(PinkNoise.ar,  1,  0, 0.3) }); // right pair
play({ Pan4.ar(PinkNoise.ar,  0, -1, 0.3) }); // back pair
play({ Pan4.ar(PinkNoise.ar,  0,  1, 0.3) }); // front pair

play({ Pan4.ar(PinkNoise.ar,  0,  0, 0.3) }); // center
```

ID: 735

PanAz azimuth panner

PanAz.ar(numChans, in, pos, level, width, orientation)

Multichannel equal power panner.

numChans - number of output channels
in - input signal
pos - pan position. Channels are evenly spaced over a cyclic
period of 2.0 in pos with 0.0 equal to channel zero and `2.0/numChans` equal to channel 1, `4.0/numChans` equal to channel 2, etc.
Thus all channels will be cyclically panned through if a sawtooth wave from -1 to +1 is used to
modulate the pos.
level - a control rate level input.
width - The width of the panning envelope. Nominally this is 2.0 which pans between pairs
of adjacent speakers. Width values greater than two will spread the pan over greater numbers
of speakers. Width values less than one will leave silent gaps between speakers.
orientation - Should be zero if the front is a vertex of the polygon. The first speaker will be directly in front. Should be 0.5 if the front bisects a side of the polygon. Then the first speaker will be the one left of center. Default is 0.5.

```
// five channel circular panning
Server.internal.boot;
(
{
PanAz.ar(
  5,     // numChans
  ClipNoise      // in
LFSaw.kr(MouseX.kr(0.2, 8, 'exponential')), // pos
0.5,   // level
  3   // width
);
}.play(Server.internal);
Server.internal.scope;
```

)

ID: 736

PanB Ambisonic B format panner

PanB.ar(in, azimuth, elevation, gain)

in - input signal
azimuth - in radians, -pi to +pi
elevation - in radians, -0.5pi to +0.5pi
gain - a control rate level input.

Output channels are in order W,X,Y,Z.

```
// You'll only hear the first two channels on a stereo setup.
play({
#w, x, y, z = PanB.ar(WhiteNoise.ar, LFSaw.kr(0.5,pi), FSinOsc.kr(0.31, 0.5pi), 0.3);
//decode for 4 channels
 DecodeB2.ar(4, w, x, y, 0.5);
});
```

ID: 737

PanB2 2D Ambisonic B-format panner

PanB2.kr(in, azimuth, gain)

Encode a mono signal to two dimensional ambisonic B-format.
in - input signal
azimuth - position around the circle from -1 to +1.
-1 is behind, -0.5 is left, 0 is forward, +0.5 is right, +1 is behind.
gain - amplitude control

```
(
{
var w, x, y, p, a, b, c, d;

p = PinkNoise.ar; // source

 // B-format encode
#w, x, y = PanB2.ar(p, MouseX.kr(-1,1), 0.1);

 // B-format decode to quad
#a, b, c, d = DecodeB2.ar(4, w, x, y);

 [a, b, d, c] // reorder to my speaker arrangement:  Lf Rf Lr Rr
}.play;
)
```

ID: 738

Rotate2 Rotate a sound field

Rotate2.kr(x, y, pos)

Rotate2 can be used for rotating an ambisonic B-format sound field around an axis.
Rotate2 does an equal power rotation so it also works well on stereo sounds.
It takes two audio inputs (x, y) and an angle control (pos).
It outputs two channels (x, y).
It computes this:

xout = cos(angle) * xin + sin(angle) * yin;
yout = cos(angle) * yin - sin(angle) * xin;

where angle = pos * pi, so that -1 becomes -pi and +1 becomes +pi.
This allows you to use an LFSaw to do continuous rotation around a circle.

x, y - input signals
pos - angle to rotate around the circle from -1 to +1.
-1 is 180 degrees, -0.5 is left, 0 is forward, +0.5 is right, +1 is behind.

```
(
{
var w, x, y, p, q, a, b, c, d;

p = WhiteNoise.ar(0.05); // source
q = LFSaw.ar(200,0,0.03)+LFSaw.ar(200.37,0,0.03)+LFSaw.ar(201,0,0.03);

 // B-format encode 2 signals at opposite sides of the circle
#w, x, y = PanB2.ar(p, -0.5) + PanB2.ar(q, 0.5);

#x, y = Rotate2.ar(x, y, MouseX.kr(-1,1));

 // B-format decode to quad
#a, b, c, d = DecodeB2.ar(4, w, x, y);

 [a, b, d, c] // reorder to my speaker arrangement:  Lf Rf Lr Rr
}.play;
)
```

```
// Rotation of stereo sound:
(
{
 // rotation via lfo
var x, y;
x = PinkNoise.ar(0.4);
y = LFTri.ar(800) * LFPulse.kr(3,0,0.3,0.2);
#x, y = Rotate2.ar(x, y, LFSaw.kr(0.1));
[x,y]
}.play;
)

{
 // rotation via mouse
var x, y;
x = Mix.fill(4, { LFSaw.ar(200 + 2.0.rand2, 0, 0.1) });
y = SinOsc.ar(900) * LFPulse.kr(3,0,0.3,0.2);
#x, y = Rotate2.ar(x, y, MouseX.kr(0,2));
[x,y]
}.play;


// Rotate B-format about Z axis:

wout = win;
zout = zin;
#xout, yout = Rotate2.ar(xin, yin, pos);

// Rotate B-format about Y axis:

wout = win;
yout = yin;
#xout, zout = Rotate2.ar(xin, zin, pos);

// Rotate B-format about X axis:

wout = win;
```

```
xout = xin;
#yout, zout = Rotate2.ar(yin, zin, pos);
```

ID: 739

Splay

*ar(inArray, spread,level, center, levelComp)
*arFill(n, function, spread,level, center, levelComp)

Splay spreads an array of channels across the stereo field.
Optional spread and center controls, and levelComp(ensation) (equal power).

```
x = { arg spread=1, level=0.2, center=0.0;
Splay.ar(
SinOsc.ar( { | i| LFNoise2.kr( rrand(10, 20), 200, 400) } !  10),
spread,
level,
center
);
}.play;

x.set(\spread, 1,   \center, 0);  // full stereo
x.set(\spread, 0.5, \center, 0);  // less wide
x.set(\spread, 0,   \center, 0);  // mono center
      \spread       \center        // spread from center to right
x.set(\spread, 0,   \center, -1); // all left
x.set(\spread, 1,   \center, 0);  // full stereo



 // the same example written with arFill:
x = { arg spread=1, level=0.5, center=0.0;
Splay.arFill(10,
{ | i| SinOsc.ar( LFNoise2.kr( rrand(10, 20), 200, i + 3 * 100))  },
spread,
level,
center
);
}.play;

// with mouse control
```

```
x = { var src;
src = SinOsc.ar( { | i| LFNoise2.kr( rrand(10, 20), 200, i + 3 * 100) } !  10);
Splay.ar(src, MouseY.kr(1, 0), 0.5, MouseX.kr(-1, 1));
}.play;
```

ID: 740

SplayZ

*ar(inArray, spread,level, width, center, levelComp)
*arFill(n, function, spread,level, width, center, levelComp)

SplayZ spreads an array of channels across a ring of channels.
Optional spread and center controls, and levelComp(ensation) (equal power).

```
x = { arg spread=1, level=0.2, width=2, center=0.0;
 SplayZ
4,
SinOsc.ar( { | i|  LFNoise2.kr( rrand(10, 20), 200, i + 3 * 100) } !  10),
spread,
level,
width,
center
);
}.scope;

x.set(\spread, 1,   \center, 0);  // full n chans
x.set(\spread, 0.5, \center, -0.25); // less wide
      \spread     \center       // mono center (depends on orientation, see PanAz)
x.set(\spread, 0, \center, -0.25); //
      \spread       \center        // mono, but rotate 1 toward the higher channels
      \spread       \center        // spread over the higher channels
x.set(\spread, 0,   \center, -0.25); // all first
x.set(\spread, 1,   \center, 0);  // full n chans

x.free;

 // the same example written with arFill:
x = { arg spread=1, level=0.5, width=2, center=0.0;
SplayZ.arFill(
4,
10,
{ | i|  SinOsc.ar( LFNoise2.kr( rrand(10, 20), 200, i + 3 * 100) ) },
spread,
```

```
level,
width,
center
);
}.scope;

// or with mouse control
x = { var src;
src = SinOsc.ar( { | i| LFNoise2.kr( rrand(10, 20), 200, i * 100 + 400) } !  10);
SplayZ.ar(4, src, MouseY.kr(1, 0), 0.2, center:  MouseX.kr(-1, 1));
}.scope;
```

ID: 741

XFade2 equal power two channel cross fade

XFade2.ar(inA, inB, pan, level)

pan - -1 .. 1

```
(
SynthDef "help-XFade2"
Out.ar(0, XFade2.ar( Saw.ar, SinOsc.ar , LFTri.kr(0.1) ));
}).play
)
```

25.15 PhysicalModels

ID: 742

Ball physical model of bouncing object

superclass: UGen

models the path of a bouncing object that is reflected by a vibrating surface

*ar(in, g, damp, friction)

in modulated surface level
g gravity
damp damping on impact
friction proximity from which on attraction to surface starts

```
// examples

// mouse x controls switch of level
(
{
var f, sf;
sf = K2A.ar(MouseX.kr > 0.5) > 0;
f = Ball.ar(sf, MouseY.kr(0.01, 20, 1), 0.01);
f = f * 10 + 500;
SinOsc.ar(f, 0, 0.2)
}.play;
)


// mouse x controls modulation rate
// mouse y controls gravity
(
{
var f, sf, g;
sf = LFNoise0.ar(MouseX.kr(1, 100, 1));
g = MouseY.kr(0.1, 10, 1);
f = Ball.ar(sf, g, 0.01, 0.01);
f = f * 140 + 500;
```

```
SinOsc.ar(f, 0, 0.2)
}.play;
)

 // the general german police choir
 // mouse x controls damping
 // mouse y controls gravity
(
{
var f, sf, g;
sf = LFPulse.ar(0.6, 0.2, 0.5);
g = MouseY.kr(0.1, 10, 1);
d = MouseX.kr(0.0001, 1, 1);
f = Ball.ar(sf, g, d);
f = f * 140 + 400;
SinOsc.ar(f, 0, 0.2)
}.play;
)
```

ID: 743

Spring physical model of resonating spring

superclass: UGen

models the force of a resonating string

*ar(in, spring, damp)

in modulated input force
spring spring constant (incl. mass)
damp damping

```
 // examples
 // trigger gate is mouse button
 // spring constant is mouse x
 // mouse y controls damping
(
{
var inforce, outforce, freq, k, d;
inforce = K2A.ar(MouseButton.kr(0,1,0)) > 0;
k = MouseY.kr(0.1, 20, 1);
d = MouseX.kr(0.00001, 0.1, 1);
outforce = Spring.ar(inforce, k, d);
  freq = outforce * 400 + 500; // modulate frequency with the force
SinOsc.ar(freq, 0, 0.2)
}.play;
)
```

```
 // several springs in series.
  // trigger gate is mouse button
 // spring constant is mouse x
 // mouse y controls damping
(
{  var m0, m1, m2, m3, d, k, inforce;
```

```
d = MouseY.kr(0.00001, 0.01, 1);
k = MouseX.kr(0.1, 20, 1);
inforce = K2A.ar(MouseButton.kr(0,1,0)) > 0;
m0 = Spring.ar(inforce, k, 0.01);
m1 = Spring.ar(m0, 0.5 * k, d);
m2 = Spring.ar(m0, 0.6 * k + 0.2, d);
m3 = Spring.ar(m1 - m2, 0.4, d);
  SinOsc                          // modulate frequency with the force

}.play;
)

 // modulating a resonating string with the force
 // spring constant is mouse x
 // mouse y controls damping
(
{  var m0, m1, m2, m3, m4, d, k, t;
k = MouseX.kr(0.5, 100, 1);
d = MouseY.kr(0.0001, 0.01, 1);
t = Dust.ar(2);
m0 = Spring.ar(ToggleFF.ar(t), 1 * k, 0.01);
m1 = Spring.ar(m0, 0.5 * k, d);
m2 = Spring.ar(m0, 0.6 * k, d);
m3 = Spring.ar([m1,m2], 0.4 * k, d);
m4 = Spring.ar(m3 - m1 + m2, 0.1 * k, d);
CombL.ar(t, 0.1, LinLin.ar(m4, -10, 10, 1/8000, 1/100), 12)

}.play;
)
```

ID: 744

TBall physical model of bouncing object

superclass: UGen

models the impacts of a bouncing object that is reflected by a vibrating surface

*ar(in, g, damp, friction)

in modulated surface level
g gravity
damp damping on impact
friction proximity from which on attraction to surface starts

```
// examples


// mouse x controls switch of level
// mouse y controls gravity
(
{
var t, sf;
sf = K2A.ar(MouseX.kr > 0.5) > 0;
t = TBall.ar(sf, MouseY.kr(0.01, 1.0, 1), 0.01);
Pan2.ar(Ringz.ar(t * 10, 1200, 0.1), MouseX.kr(-1,1));
}.play;
)


// mouse x controls step noise modulation rate
// mouse y controls gravity
(
{
var t, sf, g;
sf = LFNoise0.ar(MouseX.kr(0.5, 100, 1));
g = MouseY.kr(0.01, 10, 1);
t = TBall.ar(sf, g, 0.01, 0.002);
```

```
Ringz.ar(t * 4, [600, 645], 0.3);
}.play;
)

 // mouse x controls sine modulation rate
 // mouse y controls friction
 // gravity changes slowly
(
{
var f, g, h, fr;
fr = MouseX.kr(1, 1000, 1);
h = MouseY.kr(0.0001, 0.001, 1);
g = LFNoise1.kr(0.1, 3, 5);
f = TBall.ar(SinOsc.ar(fr), g, 0.1, h);
Pan2.ar(Ringz.ar(f, 1400, 0.04),0,5)
}.play;
)

 // sine frequency rate is modulated with a slow sine
 // mouse y controls friction
 // mouse x controls gravity
(
{
var f, g, h, fr;
fr = LinExp.kr(SinOsc.kr(0.1), -1, 1, 1, 600);
h = MouseY.kr(0.0001, 0.001, 1);
g = MouseX.kr(1, 10);
f = TBall.ar(SinOsc.ar(fr), g, 0.1, h);
Pan2.ar(Ringz.ar(f, 1400, 0.04),0,5)
}.play;
)

 // this is no mbira:  vibrations of a bank of resonators that are
 // triggered by some bouncing things that bounce one on each resonator

 // mouse y controls friction
 // mouse x controls gravity
(
{
var sc, g, d, z, lfo, rate;
```

```
g = MouseX.kr(0.01, 100, 1);
d = MouseY.kr(0.00001, 0.2);
sc = #[451, 495.5, 595, 676, 734.5]; //azande harp tuning by B. Guinahui
lfo = LFNoise1.kr(1, 0.005, 1);
rate = 2.4;
rate = rate * sc.size.reciprocal;
z = sc.collect { | u,i|
var f, in;
in = Decay.ar(
Mix(Impulse.ar(rate, [1.0, LFNoise0.kr(rate / 12)].rand, 0.1)),      0.001
);
in = Ringz.ar(in,
Array.fill(4, { | i| (i+1) + 0.1.rand2 }) / 2
* Decay.ar(in,0.02,rand(0.5,1), lfo)      * u,
Array.exprand(4, 0.2, 1).sort
);
in = Mix(in);
f = TBall.ar(in * 10, g, d, 0.001);

in + Mix(Ringz.ar(f, u * Array.fill(4, { | i| (i+1) + 0.3.rand2 }) * 2, 0.1))
};
Splay.ar(z) * 0.8
}.play;
)
```

25.16 SynthControl

ID: 745

Control

superclass: MultiOutUGen

Used to bring signals and floats into the ugenGraph function of your SynthDef. This is the UGen that delivers the args into your function.

Generally you do not create Controls yourself. (See Arrays example below)

The rate may be either .kr (continous control rate signal) or .ir (a static value, set at the time the synth starts up, and subsequently unchangeable).

SynthDef creates these when compiling the ugenGraph function. They are created for you, you use them, and you don't really need to worry about them if you don't want to.

```
SynthDef("help-Control",{ arg freq=200;

 freq.inspect; // at the time of compiling the def

}).writeDefFile;
```

What is passed into the ugenGraph function is an OutputProxy, and its source is a Control.

The main explicit use of Control is to allow Arrays to be sent to running Synths:

```
// a synth def that has 4 partials
(
s = Server.local;
SynthDef("help-Control", { arg out=0,i_freq;
var klank, n, harm, amp, ring;
n = 9;

 // harmonics
harm = Control.names([\harm]).ir(Array.series(4,1,1).postln);
 // amplitudes
amp = Control.names([\amp]).ir(Array.fill(4,0.05));
```

```
 // ring times
ring = Control.names([\ring]).ir(Array.fill(4,1));

klank = Klank.ar(`[harm,amp,ring], {ClipNoise.ar(0.003)}.dup, i_freq);

Out.ar(out, klank);
}).send(s);
)

// nothing special yet, just using the default set of harmonics.
a = Synth("help-Control",[\i_freq, 300]);
b = Synth("help-Control",[\i_freq, 400]);
c = Synth("help-Control",[\i_freq, 533.33]);
d = Synth("help-Control",[\i_freq, 711.11]);

a.free;
b.free;
c.free;
d.free;

// in order to set the harmonics amps and ring times at
// initialization time we need to use an OSC bundle.
(
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7] // set odd harmonics
);
)

s.sendMsg("/n_free", 2000);

(
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
 // set geometric series harmonics
["/n_setn", 2000, "harm", 4] ++ Array.geom(4,1,1.61)
);
)

s.sendMsg("/n_free", 2000);
```

```
(
// set harmonics, ring times and amplitudes
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7], // set odd harmonics
["/n_setn", 2000, "ring", 4] ++ Array.fill(4,0.1), // set shorter ring time
["/n_setn", 2000, "amp", 4] ++ Array.fill(4,0.2) // set louder amps
);
)

s.sendMsg(\n_trace, 2000);
s.sendMsg(\n_free, 2000);

(
// same effect as above, but packed into one n_setn command
s.sendBundle(nil,
["/s_new", "help-Control", 2000, 1, 0, \i_freq, 500], // start note
["/n_setn", 2000, "harm", 4, 1, 3, 5, 7,
"ring", 4] ++ Array.fill(4,0.1)
++ ["amp", 4] ++ Array.fill(4,0.2)
);
)

s.sendMsg(\n_trace, 2000);
s.sendMsg(\n_free, 2000);
```

25.17 Triggers

ID: 746

Gate gate or hold

Gate.ar(in, gate)

Allows input signal value to pass when gate is positive, otherwise holds last value.

in - input signal.
gate - trigger. Trigger can be any signal. A trigger happens when the signal changes from non-positive to positive.

```
Server.internal.boot;
// Control rate so as not to whack your speakers with DC
{ Gate.kr(WhiteNoise.kr(1, 0), LFPulse.kr(1.333, 0.5))}.scope(zoom:  20);
```

ID: 747

InRange tests if a signal is within a given range

inRange.ar(in, lo, hi)
InRange.kr(in, lo, hi)

If in is >= lo and <= hi output 1.0, otherwise output 0.0. Output is initially zero.

in - signal to be tested
lo - low threshold
hi - high threshold

```
Server.internal.boot;

{ InRange.kr(SinOsc.kr(1, 0, 0.2), -0.15, 0.15)}.scope; // see the trigger

{ InRange.kr(SinOsc.kr(1, 0, 0.2), -0.15, 0.15) * BrownNoise.ar(0.1)}.scope; // trigger noise Burst
```

ID: 748

LastValue

LastValue.ar(in, diff)
LastValue.kr(in, diff)

output the last value before the input changed more than a threshhold

in input
diff difference threshhold

```
d = { arg freq=440; SinOsc.ar(LastValue.ar(freq, 20), 0, 0.2) }.play;

d.set(\freq, 400);
d.set(\freq, 200);
d.set(\freq, 670);
d.set(\freq, 680);
d.set(\freq, 695);
d.free;
```

return the difference between currrent and the last changed

```
(
d = { arg out=0, val=1;
SinOsc.ar(
abs(val - LastValue.kr(val)) * 400 + 200,
0, 0.2
)
}.play;
)

d.set(\val, 3);
d.set(\val, 2);
d.set(\val, 0.2);
d.set(\val, 1);
```

```
d.set(\val, 2);
d.free;
```

ID: 749

MostChange output most changed

MostChange.ar(in1, in2)
MostChange.kr(in1, in2)

output the input that changed most

in1, in2 - inputs

//doesn't work yet.

```
d = SynthDef("help-MostChange", { arg amp=1.0;
var out, in1, in2;
in1 = LFNoise1.ar(800, amp);
in2 = SinOsc.ar(800);
out = MostChange.ar(in1, in2) * 0.1;
Out.ar(0, out)
}).play;

d.set(\amp, 0.1);
d.set(\amp, 0);
d.set(\amp, 3);
d.free;
```

the control that changed most is used for output:

```
d = SynthDef("help-MostChange", { arg freq=440;
var out, internalFreq;
internalFreq = LFNoise0.ar(0.3, 300, 800);
out = SinOsc.ar(
MostChange.kr(freq, internalFreq)
);
Out.ar(0, out * 0.1)
}).play;

d.set(\freq, 800);
      \freq        //nothing changed
d.set(\freq, 900);
```

```
d.free;
```

ID: 750

Peak track peak signal amplitude

Peak.ar(in, trig)

Outputs the peak amplitude of the signal received at the input.
When triggered, the maximum output value is reset to the current value.
in - input signal.
trig - A trigger resets the output value to the current input value.
A trigger happens when the signal changes from non-positive to positive.

Internally, the absolute value of the signal is used, to prevent underreporting the peak value if there is a negative DC offset. To obtain the minimum and maximum values of the signal as is, use the **[RunningMin]** and **[RunningMax]** UGens.

```
(
{
 SinOsc
Peak.ar(Dust.ar(20), Impulse.ar(0.4)) * 500 + 200,
0, 0.2
)

}.play;
)

// follow a sine lfo, reset rate controlled by mouse x
(
{
 SinOsc
Peak.kr(SinOsc.kr(0.2), Impulse.kr(MouseX.kr(0.01, 2, 1))) * 500 + 200,
0, 0.2
)

}.play;
)
```

ID: 751

PeakFollower track peak signal amplitude

PeakFollower.ar(in, decay)

Outputs the peak amplitude of the signal received at the input.
if level is below maximum, the level decreases by the factor given in decay.

in - input signal.
decay - decay factor.

Internally, the absolute value of the signal is used, to prevent underreporting the peak value if there is a negative DC offset. To obtain the minimum and maximum values of the signal as is, use the **[RunningMin]** and **[RunningMax]** UGens.

```
s.boot;

// no decay
(
{
 SinOsc
PeakFollower.ar(Dust.ar(20, Line.kr(0, 1, 4)), 1.0) * 1500 + 200,
0, 0.2
)

}.play;
)

// a little decay
(
{
 SinOsc
PeakFollower.ar(Dust.ar(20, Line.kr(0, 1, 4)), 0.999) * 1500 + 200,
0, 0.2
)
```

```
}.play;
)

// mouse x controls decay, center of the
(
{
var decay;
decay = MouseX.kr(0.99, 1.00001).min(1.0);
 SinOsc
PeakFollower.ar(Dust.ar(20), decay) * 1500 + 200,
0, 0.2
);

}.play;
)




// follow a sine lfo, decay controlled by mouse x
(
{
var decay;
decay = MouseX.kr(0, 1.1).min(1.0);
 SinOsc
PeakFollower.kr(SinOsc.kr(0.2), decay) * 200 + 500,
0, 0.2
)

}.play;
)
```

ID: 752

Phasor a resettable linear ramp between two levels

superclass: UGen

Phasor is a linear ramp between start and end values. When its trigger input crosses from non-positive to positive, Phasor's output will jump to its reset position. Upon reaching the end of its ramp Phasor will wrap back to its start. **N.B.** Since **end** is defined as the wrap point, its value is never actually output.

Phasor is commonly used as an index control with **[BufRd]** and**[BufWr]**.

***ar(trig, rate, start, end, resetPos)**
***kr(trig, rate, start, end, resetPos)**

trig
whan resetPos (default: 0, equivalent to start)
rate
the amount of change per sample
i.e at a rate of 1 the value of each sample will be 1 greater than the preceding sample

start, end
start and end points of ramp.

resetPos
the value to jump to upon receiving a trigger.

```
// phasor controls sine frequency:  end frequency matches a second sine wave.

(
{ var trig, rate, x, sr;
rate = MouseX.kr(0.2, 2, 1);
trig = Impulse.ar(rate);
sr = SampleRate.ir;
x = Phasor.ar(trig, rate / sr);
 SinOsc
```

```
[
   LinLin                          // convert range from 0..1 to 600..1000
   1000 // constant second frequency
], 0, 0.2)

}.play;
)


// two phasors control two sine frequencies:  mouse y controls resetPos of the second
(
{ var trig, rate, x, sr;
rate = MouseX.kr(1, 200, 1);
trig = Impulse.ar(rate);
sr = SampleRate.ir;
x = Phasor.ar(trig, rate / sr, 0, 1, [0, MouseY.kr(0, 1)]);
SinOsc.ar(x * 500 + 500, 0, 0.2)
}.play;
)


// use phasor to index into a sound file

// allocate a buffer with a sound file
    Buffer         "sounds/a11wlk01.wav"

// simple playback (more examples:  see BufRd)
// Start and end here are defined as 0 and the number of frames in the buffer.
// This means that the Phasor will output values from 0 to numFrames - 1 before looping,
// which is perfect for driving BufRd.  (See note above)
{ BufRd.ar(1, b.bufnum, Phasor.ar(0, BufRateScale.kr(b.bufnum), 0, BufFrames.kr(b.bufnum))) }.play;


// two phasors control two sound file positions:  mouse y controls resetPos of the second
(
{ var trig, rate, framesInBuffer;
rate = MouseX.kr(0.1, 100, 1);
trig = Impulse.ar(rate);
framesInBuffer = BufFrames.kr(b.bufnum);
x = Phasor.ar(trig, BufRateScale.kr(b.bufnum), 0, framesInBuffer,
```

```
[0, MouseY.kr(0, framesInBuffer)]);
BufRd.ar(1, b.bufnum, x);
}.play;
)
```

ID: 753

PulseCount pulse counter

PulseCount.ar(trig, reset)

Each trigger increments a counter which is output as a signal.
trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from
non-positive to positive.
reset - resets the counter to zero when triggered.

```
SynthDef "help-PulseCount"   arg
Out.ar(out,
SinOsc.ar(
PulseCount.ar(Impulse.ar(10), Impulse.ar(0.4)) * 200,
0, 0.05
)
)
}).play;
```

ID: 754

PulseDivider pulse divider

PulseDivider.ar(trig, div, startCount)

Outputs one impulse each time it receives a certain number of triggers at its input.
trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from
non-positive to positive.
div - number of pulses to divide by.
startCount - starting value for the trigger count. This lets you start somewhere in the middle of a count,
or if startCount is negative it adds that many counts to the first time the output is triggers.

```
SynthDef "help-PulseDivider"   arg
var p, a, b;
p = Impulse.ar(8);
a = SinOsc.ar(1200, 0, Decay2.ar(p, 0.005, 0.1));
b = SinOsc.ar(600,  0, Decay2.ar(PulseDivider.ar(p, 4), 0.005, 0.5));

Out.ar(out,(a + b) * 0.4)
}).play;
```

ID: 755

RunningMax track maximum level

RunningMax.ar(in, trig)

Outputs the maximum value received at the input.
When triggered, the maximum output value is reset to the current value.
in - input signal.
trig - A trigger resets the output value to the current input value.
A trigger happens when the signal changes from non-positive to positive.

```
(
{
 SinOsc
RunningMax.ar(Dust.ar(20), Impulse.ar(0.4)) * 500 + 200,
0, 0.2
)

}.play;
)

// follow a sine lfo, reset rate controlled by mouse x
(
{
 SinOsc
RunningMax.kr(SinOsc.kr(0.2), Impulse.kr(MouseX.kr(0.01, 2, 1))) * 500 + 200,
0, 0.2
)

}.play;
)
```

ID: 756

RunningMin track minimum level

RunningMin.ar(in, trig)

Outputs the minimum value received at the input.
When triggered, the minimum output value is reset to the current value.
in - input signal.
trig - A trigger resets the output value to the current input value.
A trigger happens when the signal changes from non-positive to positive.

```
(
{
 SinOsc
RunningMin.ar(Dust.ar(20), Impulse.ar(0.4)) * 500 + 200,
0, 0.2
)

}.play;
)

// follow a sine lfo, reset rate controlled by mouse x
(
{
 SinOsc
RunningMin.kr(SinOsc.kr(0.2), Impulse.kr(MouseX.kr(0.01, 2, 1))) * 500 + 200,
0, 0.2
)

}.play;
)
```

ID: 757

Schmidt Schmidt trigger

Schmidt.ar(in, lo, hi)
Schmidt.kr(in, lo, hi)

When in crosses to greater than hi, output 1.0, then when signal crosses lower than lo output 0.0. Uses the formula if(out == 1, { if(in < lo, { out = 0.0 }) }, { if(in > hi, { out = 1.0 }) }). Output is initially zero.

in - signal to be tested
lo - low threshold
hi - high threshold

```
Server.internal.boot;

{ Schmidt.kr(SinOsc.kr(1, 0, 0.2), -0.15, 0.15)}.scope; // see the trigger

{ Schmidt.kr(MouseX.kr(0, 1), 0.2, 0.8)}.scope; // try it with the cursor

// threshold octave jumps
(
{
var in = LFNoise1.kr(3);
var octave = Schmidt.kr(in, -0.15, 0.15) + 1;
SinOsc.ar(in * 200 + 500 * octave, 0, 0.1)
}.scope;
)
```

ID: 758

SetResetFF set-reset flip flop

SetResetFF.ar(trig, reset)

Output is set to 1.0 upon receiving a trigger in the set input, and to 0.0 upon receiving a trigger
in the reset input. Once the flip flop is set to zero or one further triggers in the same input
are have no effect. One use of this is to have some precipitating event cause something to
happen until you reset it.
trig - trigger sets output to one
reset - trigger resets output to zero

```
(
play({
 a = Dust        // the set trigger
 b = Dust        // the reset trigger
SetResetFF.ar(a,b) * BrownNoise.ar(0.2);

})
```

ID: 759

Stepper pulse counter

Stepper.kr(trig, reset, min, max, step, resetval)

Each trigger increments a counter which is output as a signal. The counter wraps between min and max.
trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from
non-positive to positive.
reset - resets the counter to **resetval** when triggered.
min - minimum value of the counter.
max - maximum value of the counter.
step - step value each trigger. May be negative.
resetval - value to which the counter is reset when it receives a reset trigger. If nil, then this is patched to **min**.

```
SynthDef "help-Stepper"   arg
Out.ar(out,
SinOsc.ar(
Stepper.kr(Impulse.kr(10), 0, 4, 16, 1) * 100,
0, 0.05
)
)
}).play;

SynthDef "help-Stepper"   arg
Out.ar(out,
SinOsc.ar(
Stepper.kr(Impulse.kr(10), 0, 4, 16, -3) * 100,
0, 0.05
)
)
}).play;

SynthDef "help-Stepper"   arg
Out.ar(out,
SinOsc.ar(
```

```
Stepper.kr(Impulse.kr(10), 0, 4, 16, 4) * 100,
0, 0.05
)
)
}).play;


////////////////////////////////////////////////////////////////////////////
//
// Using Stepper and BufRd for sequencing
//

s.boot;

s.sendMsg(\b_alloc, 10, 128);

m = #[0,3,5,7,10];

a = ({rrand(0,15)}.dup(16).degreeToKey(m) + 36).midicps;
s.performList(\sendMsg, \b_setn, 10, 0, a.size, a);

(
SynthDef \stepper
var rate, clock, index, freq, ffreq, env, out, rev, lfo;

rate = MouseX.kr(1,5,1);
clock = Impulse.kr(rate);
env = Decay2.kr(clock, 0.002, 2.5);
index = Stepper.kr(clock, 0, 0, 15, 1, 0);
freq = BufRd.kr(1, 10, index, 1, 1);
freq = Lag2.kr(freq) + [0,0.3];
ffreq = MouseY.kr(80,1600,1) * (env * 4 + 2);
out = Mix.ar(LFPulse.ar(freq * [1, 3/2, 2], 0, 0.3));
out = RLPF.ar(out, ffreq, 0.3, env);
out = RLPF.ar(out, ffreq, 0.3, env);
out = out * 0.02;

 // echo
out = CombL.ar(out, 1, 0.66/rate, 2, 0.8, out);
```

```
 // reverb
rev = out;
5.do { rev = AllpassN.ar(rev, 0.05, {0.05.rand}.dup, rrand(1.5,2.0)) };
out = out + (0.3 * rev);

out = LeakDC.ar(out);

 // flanger
 lfo = SinOsc.kr(0.2, [0,0.5pi], 0.0024, 0.0025);
1.do { out = DelayL.ar(out, 0.1, lfo, 1, out) };

 // slight bass emphasis
out = OnePole.ar(out, 0.9);

Out.ar(0, out);

}).send(s);
)

s.sendMsg(\s_new, \stepper, 1000, 0, 0);

a = ({rrand(0,15)}.dup(16).degreeToKey(m) + 38).midicps;
s.performList(\sendMsg, \b_setn, 10, 0, a.size, a);

                    // transpose up 2 semitones
s.performList(\sendMsg, \b_setn, 10, 0, a.size, a);


(
a = [ 97.999, 195.998, 523.251, 466.164, 195.998, 233.082, 87.307, 391.995, 87.307, 261.626, 195.998,
77.782, 233.082, 195.998, 97.999, 155.563 ];
s.performList(\sendMsg, \b_setn, 10, 0, a.size, a);
)

s.sendMsg(\n_free, 1000);
```

ID: 760

Sweep triggered linear ramp

superclass: UGen

starts a linear raise by rate/sec from zero when trig input crosses from non-positive to positive

***ar(trig, rate)**
***kr(trig, rate)**

```
// using sweep to modulate sine frequency
(
{ var trig;
trig = Impulse.kr(MouseX.kr(0.5, 20, 1));
SinOsc.ar(Sweep.kr(trig, 700) + 500, 0, 0.2)
}.play;
)
```

```
// using sweep to index into a buffer

          "/b_allocRead"     "sounds/a11wlk01.wav"

(
{ var trig;
trig = Impulse.kr(MouseX.kr(0.5, 10, 1));
BufRd.ar(1, 0, Sweep.ar(trig, BufSampleRate.ir(0)))
}.play;
)

// backwards, variable offset
```

```
(
{ var trig, pos, rate;
trig = Impulse.kr(MouseX.kr(0.5, 10, 1));
rate = BufSampleRate.ir(0);
pos = Sweep.ar(trig, rate.neg) + (BufFrames.ir(0) * LFNoise0.kr(0.2));
BufRd.ar(1, 0, pos)
}.play;
)


// raising rate
(
{ var trig, rate;
trig = Impulse.kr(MouseX.kr(0.5, 10, 1));
rate = Sweep.kr(trig, 2) + 0.5;
BufRd.ar(1, 0, Sweep.ar(trig, BufSampleRate.ir(0) * rate))
}.play;
)
```

ID: 761

TDelay trigger delay

TDelay.ar(trigger, delayTime)

Delays a trigger by a given time. Any triggers which arrive in the time between an input trigger and its delayed output, are ignored.
trigger - input trigger signal.
delayTime - delay time in seconds.

```
(
{
z = Impulse.ar(2);
[z * 0.1, ToggleFF.ar(TDelay.ar(z, 0.5)) * SinOsc.ar(mul:  0.1)]
}.scope)
```

ID: 762

Timer returns time since last triggered

superclass: UGen

*ar(trig)
*kr(trig)

```
// using timer to modulate sine frequency:  the slower the trigger is the higher the frequency
(
{ var trig;
trig = Impulse.kr(MouseX.kr(0.5, 20, 1));
SinOsc.ar(Timer.kr(trig) * 500 + 500, 0, 0.2)
}.play;
)
```

ID: 763

ToggleFF toggle flip flop

ToggleFF.ar(trig)

Toggles between zero and one upon receiving a trigger.
trig - trigger input

```
(
play({
SinOsc.ar((ToggleFF.ar(Dust.ar(XLine.kr(1,1000,60))) * 400) + 800, 0, 0.1)
}))
```

ID: 764

Trig timed trigger

Trig.ar(trig, dur)

When a nonpositive to positive transition occurs at the input, Trig outputs the level of the triggering input for the specified duration, otherwise it outputs zero.

trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from non-positive to positive.
dur - duration of the trigger output.

```
{ Trig.ar(Dust.ar(1), 0.2) * FSinOsc.ar(800, 0.5) }.play
```

```
{ Trig.ar(Dust.ar(4), 0.1) }.play
```

ID: 765

Trig1 timed trigger

Trig1.ar(in, dur)

When a nonpositive to positive transition occurs at the input, Trig outputs 1.0 for the specified duration, otherwise it outputs zero.

trig - trigger. Trigger can be any signal. A trigger happens when the signal changes from non-positive to positive.
dur - duration of the trigger output.

```
{ Trig1.ar(Dust.ar(1), 0.2) * FSinOsc.ar(800, 0.5) }.play
```

To create a fixed duration gate

```
(

SynthDef("trig1",{ arg dur=0.125;
var gate;
gate = Trig1.kr(1.0,dur);
OffsetOut.ar(0,
SinOsc.ar(800, 0.3)
* EnvGen.kr(
Env([0,0.1,0.1,0],[0.01,1.0,0.01],[-4,4],2),
gate,
doneAction:  2)
)
}).send(s);

Routine
1.0.wait;
100.do({
s.sendBundle(0.05,["s_new", "trig1" ,-1,0,0,0,rrand(0.02,0.25)]);
0.25.wait
})
        SystemClock
```

```
)
```

This should sound like a continous sine wave, although it is actually a series of 0.25 second synths.

```
(
SynthDef "trig1"
var gate;
gate = Trig1.kr(1.0,0.25);
 OffsetOut
SinOsc.ar(800, 0.3)
* EnvGen.kr(
Env([0,0.1,0.1,0],[0.01,1.0,0.01],[-4,4],2),
gate,
doneAction:  2)
)
}).send(s);

Routine
1.0.wait;
100.do({
s.sendBundle(0.05,["s_new", "trig1" ,-1]);
0.25.wait
})
        SystemClock

)
```

26 UnaryOps

ID: 766

abs absolute value

a.abs
abs(a)

abs(-5).postln;

```
(
{
var a;
a = SyncSaw.ar(100, 440, 0.1);
 // Absolute value
a.abs
}.play
)
```

compared to

```
(
{
var a;
a = SyncSaw.ar(100, 440, 0.1);
a
}.play
)
```

ID: 767

acos arccosine

a.acos
acos(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
(a.acos / 0.5pi) - 1
}.plot)
```

ID: 768

ampdb convert linear amplitude to decibels

a.ampdb
ampdb(a)

```
0.1.ampdb.postln;
(ampdb(0.1).asString ++ " decibels").postln;
```

ID: 769

asin arcsine

a.asin
asin(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
a.asin / 0.5pi
}.plot)
```

ID: 770

atan arctangent

a.atan
atan(a)

```
(
{
var a;
a = Line.ar(-10, 10, 0.01);
a.atan / 0.5pi
}.plot)
```

ID: 771

ceil next higher integer

a.ceil
ceil(a)

```
(
{
var a;
a = Line.ar(1, -1, 0.01);
[ a, a.ceil ]
}.plot)
```

ID: 772

cos cosine

a.cos
cos(a)

```
(
{
var a;
a = Line.ar(0, 2pi, 0.01);
a.cos
}.plot)
```

ID: 773

cosh hyperbolic cosine

a.cosh
cosh(a)

```
(
{
var a;
a = Line.ar(-pi, pi, 0.01);
a.cosh * 0.1
}.plot)
```

ID: 774

cpsmidi convert cycles per second to MIDI note

a.cpsmidi
cpsmidi(a)

```
440.cpsmidi.postln
```

ID: 775

cpsoct convert cycles per second to decimal octaves

a.cpsoct
cpsoct(a)

440.cpsoct.postln

ID: 776

cubed cubed value

a.cubed
cubed(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.cubed ]
}.plot)
```

ID: 777

dbamp convert decibels to linear amplitude

a.dbamp
dbamp(a)

```
-20.dbamp.postln
```

```
{ FSinOsc.ar(800, 0.0, Line.kr(-3,-40,10).dbamp) }.play;
```

ID: 778

distort nonlinear distortion

a.distort
distort(a)

```
(
{
var a;
a = Line.ar(-4, 4, 0.01);
a.distort
}.plot)

{ FSinOsc.ar(500, 0.0, XLine.kr(0.1, 10, 10)).distort * 0.25 }.scope;
```

ID: 779

exp exponential

a.exp
exp(a)

```
(
{
var a;
a = Line.ar(0, -5, 0.01);
[ a, a.exp ]
}.plot)
```

ID: 780

floor next lower integer

a.floor
floor(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.floor ]
}.plot)
```

ID: 781

frac fractional part

a.frac
frac(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.frac ]
}.plot)
```

ID: 782

isNegative test if signal is < 0

a.isNegative
isNegative(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
a.isNegative
}.plot)
```

ID: 783

isPositive test if signal is >= 0

a.isPositive
isPositive(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
a.isPositive
}.plot)
```

ID: 784

isStrictlyPositive test if signal is > 0

a.isStrictlyPositive
isNegative(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
a.isStrictlyPositive
}.plot)
```

ID: 785

log natural logarithm

a.log
log(a)

```
(
{
var a, e;
e = exp(1);
a = Line.ar(e, 1/e, 0.01);
a.log
}.plot)
```

ID: 786

log10 base 10 logarithm

a.log10
log10(a)

```
(
{
var a;
a = Line.ar(10, 1/10, 0.01);
a.log10
}.plot)
```

ID: 787

log2 base 2 logarithm

a.log2
log2(a)

```
(
{
var a;
a = Line.ar(2, 1/2, 0.01);
a.log2
}.plot)
```

ID: 788

midicps convert MIDI note to cycles per second

a.midicps
midicps(a)

```
(
{
Saw.ar(Line.kr(24,108,10).midicps, 0.2)
}.play)
```

ID: 789

neg inversion

a.neg
neg(a)

```
(
{
var a;
a = FSinOsc.ar(300);
[ a, a.neg ]
}.plot)
```

ID: 790

octcps convert decimal octaves to cycles per second

a.octcps
octcps(a)

```
(
{
Saw.ar(Line.kr(2,9,10).octcps, 0.2)
}.play)
```

ID: 791

reciprocal reciprocal

a.reciprocal
reciprocal(a)

```
(
10.do({ arg a;
a = a + 1;
a.reciprocal.postln
});
)

9.reciprocal == (1 / 9);
10.reciprocal == (1 / 10);
```

ID: 792

sign sign function

a.sign
sign(a)

-1 when a $<$ 0, $+$1 when a $>$ 0, 0 when a is 0

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.sign ]
}.plot)
```

ID: 793

sin sine

a.sin
sin(a)

```
(
{
var a;
a = Line.ar(0, 2pi, 0.01);
a.sin
}.plot)
```

ID: 794

sinh hyperbolic sine

a.sinh
sinh(a)

```
(
{
var a;
a = Line.ar(-pi, pi, 0.01);
a.sinh * 0.1
}.plot)
```

ID: 795

softclip nonlinear distortion

a.softclip
softclip(a)

Distortion with a perfectly linear region from -0.5 to +0.5

```
Server.internal.boot;

{ FSinOsc.ar(500,0.0, XLine.kr(0.1, 10, 10)).softclip * 0.25 }.scope(2);
```

ID: 796

sqrt square root

a.sqrt
sqrt(a)

The definition of square root is extended for signals so that sqrt(a) when a<0 returns -sqrt(-a).

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.sqrt ]
}.plot)
```

ID: 797

squared squared value

a.squared
squared(a)

```
(
{
var a;
a = Line.ar(-1, 1, 0.01);
[ a, a.squared ]
}.plot)
```

ID: 798

tan tangent

a.tan
tan(a)

```
(
{
var a;
a = Line.ar(-pi/2, pi/2, 0.01);
a.tan / 10
}.plot)
```

ID: 799

tanh hyperbolic tangent

a.tanh
tanh(a)

```
(
{
var a;
a = Line.ar(-pi, pi, 0.01);
a.tanh
}.plot)
```

ID: 800

UnaryOpUGen

superclass: UGen

UnaryOpUGens are created as the result of a unary operator applied to a UGen.

Examples

```
SinOsc.ar(300).abs.dump;

{ SinOsc.ar(300).abs }.plot
```