```
(
	Server.default = Server.internal;
	s = Server.default;
	s.boot;
)
```

Server Concepts

The SC Synth Server is a dynamic synthesis engine. While synthesis is running, new modules can be created, destroyed and repatched. Effects processes can be created and patched into a signal flow dynamically at scheduled times.

Patching between modules is done through global **audio buses and control buses**.
All running modules are ordered in a **tree of nodes** that define an order of execution.

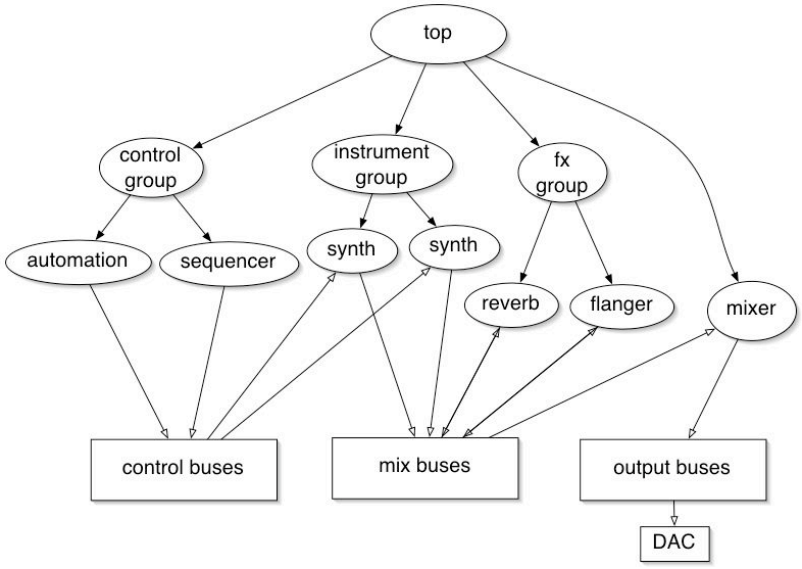
Global Buses

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. The lowest numbered buses get written to the audio hardware outputs. Stereo output uses buses 0 and 1. 8-channel output uses 0 through 7. 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.

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 biggest advantage of the audio buses is that they provide a mechanism for applying "effects" to the output of synths. Specifying the output bus of one or more audio-generating synths enables them to be routed to an "effects" synths.

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.



Concerning Ins and Outs and Busses

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.

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.

ReplaceOut **send signal to a bus, overwriting previous contents**

superclass: Out

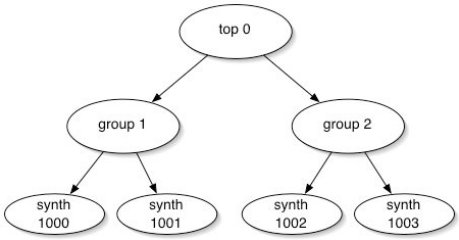
***ar(bus, channelsArray)** - write a signal to an audio bus.
***kr(bus, channelsArray)** - write a signal to a control bus.

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.

Tree of synthesis nodes:

What happens is that SC3 takes the **SynthDef** code and compiles it into bytecode that the server app can understand. SynthDef.play takes care of starting the Synth for you, but more typically you will precompile the SynthDef and use Synth to trigger audio events. The server reads the SynthDef and creates a **synth node** based upon it. The **synth node** begins execution as quickly as it can. If this is the only Synth performing, it will have ID# 1000. If we start it up again so that there are additional Synths executing, each new version will have the next ID# in sequence---1001, 1002, etc.

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 to the server, and thus when it will be available for creating new Synths. One simple way around this is to execute SynthDef code in blocks before anything else happens.



Groups are internal nodes, Synths are leaf nodes. Every node has an ID. Top group is always ID zero. The tree defines the order of execution: 0(1(1000, 1001), 2(1002, 1003))

Terms:

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, and a 'default group' with an ID of 1 which is the default group for all new Nodes.

Order of execution

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. 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 read the output of another, you don't have to worry about order of execution. 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.

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 Synth'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

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.
Where order of execution matters, it is important that you specify an addAction, or use one of the convenience methods shown above.

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.ar(0, osc); // 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);
)

// execute these one at a time:
x = Synth("tish", [\freq, 200, \rate, 1.2]);
y = Synth.after(x,"echo");
y.free;
y = Synth.after(x,"echo",[ \delay, 0.1, \decay, 4]);
y.free;
x.free;
```

This works because we added the effect after the other node.

Filtering Example

```
(
	// first collect some things to play with
	SynthDef("moto-rev", { arg outChan=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(outChan, Mix.ar(x));
    }).send(s);

    SynthDef("bubbles", { arg outChan=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(outChan, zout);
    }).send(s);

    SynthDef("rlpf",{ arg inChan=0, outChan=0,ffreq=600,rq=0.1;
        Out.ar( outChan, RLPF.ar( In.ar(inChan), ffreq,rq) )
    }).send(s);

    SynthDef("wah", { arg inChan=0, outChan=0, rate = 1.5, cfreq = 1400, mfreq = 1200, rq=0.1;
        var zin, zout;

        zin = In.ar(inChan);
        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;
        Out.ar( outChan , zout );
    }).send(s);

)

    // execute these one at a time

    // y is playing on bus 0
    y = Synth("moto-rev",[\outChan,1]);

    // change to bus 20
    y.set(\outChan, 20);

    // z is reading from bus 20 and writing to 0; It must be *after* y
    z = Synth.after(y,"wah",[\inChan, 20, \outChan,0]);

    // change 'wha' to bus 20
    z.set(\outChan, 20);

    // add a rlpf after that, reading from 20 and writing to 0
    x = Synth.after(z,"rlpf",[\inChan, 20, \outChan,0]);

    // now place another synth on buss 30
    r = Synth("bubbles",["outChan",30]);

    // add a rlpf after that, reading from 30 and writing to 1
    t = Synth.after(r,"rlpf",[\inChan, 30, \outChan,1]);
```

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