The Manual

Writing SuperCollider Synths For Sonic Pi

View on GitHub

The Manual

// ## Writing Super Collider Synthesisers For Sonic Pi

// This manual is based on Sonic Pi 5 (Technical Preview).

// ## Gordon Guthrie

```
__          __        _      _____
\ \        / /       | |    |_   _|
 \ \  /\  / /__  _ __| | __   | |  _ __
  \ \/  \/ / _ \| '__| |/ /   | | | '_ \
   \  /\  / (_) | |  |   <   _| |_| | | |
 ___\/  \/ \___/|_|  |_|\_\ |_____|_| |_|
|  __ \
| |__) | __ ___   __ _ _ __ ___  ___ ___
|  ___/ '__/ _ \ / _` | '__/ _ \/ __/ __|
| |   | | | (_) | (_| | | |  __/\__ \__ \
|_|   |_|  \___/ \__, |_|  \___||___/___/
                  __/ |
                 |___/
```

Chapter 1 - First beeps

Chapter 2 - Existing synths in Sonic Pi

Chapter 3 - Deep dive

TheManual is maintained by gordonguthrie.
This page was generated by GitHub Pages.

The Manual

Writing SuperCollider Synths For Sonic Pi

View on GitHub

Chapter 1 - First beeps

SonicPi and SuperCollider

Sonic Pi is a user-friendly live-coding environment that lets people start making music very quickly and is focussed on ease of use and quick starts.

The heavy lifting of making actual noises in Sonic Pi is provided by SuperCollider which is a full-blown sonic engine for making and manipulating audio signals. SuperCollider is far from user-friendly requiring a high degree of both programming skill and a profound understanding of synthesiser design.

What is the purpose of this manual?

In scope - custom synths

Sonic Pi has a great range of synthesisers but there is always room for more. This manual will make it easier for a small number of synthesiser designers and developers to make a much large range of synthesisers and effects available to the very large community of normal users of Sonic Pi.

The goal of this project is to enable a small community of sound sculptors to build new synthesisers for the large community of live coders to incorporate by demystifying the messy business of making one coding paradigm available in a completely foreign one.

It is a step-by-step manual explaining in detail the intricacies of the relationship between the Sonic Pi front end and the SuperCollider back end.

It is not a substitute for Sonic Pi book or the SuperCollider book.

Out of scope - custom FXs

As we go through the book it will become clear that SuperCollider is used for all things that make sounds in Sonic Pi, synths, samples and effects (FXs). This book only covers synths.

Who is this for?

This manual is for people who can already use Sonic Pi and now want to learn how to build additional synthesisers for Sonic Pi in SuperCollider.

SuperCollider offers functionality that allows you build and play synthesisers.

This manual focusses on building them only: for people to play them using SonicPi.

SuperCollider is a major programming environment in its own right and to learn how to get the most out of it you will need to study it. There

Do learn how to do this, you will need to install SuperCollider.

What does it cover?

Chapter 1 - First Beeps

This chapter shows you how to make a beep in SuperCollider and then how to make that same beep from within Sonic Pi.

Chapter 2 - Existing synths in Sonic Pi

This chapter looks at the existing synths and the two different ways they are implemented - the old way (Overtone) and the new approved way (SuperCollider).

Chapter 3 - Deep dive

This OPTIONAL chapter does a deep dive into how synths are loaded and invoked - not strictly necessary for you to absorb all this to write your own synths.

Chapter 4 - How built-in synths behave

The chapter goes through how built in synths behave in terms of:

- paramaters, default values and validations
- error messages
- integration with the documentation/in Sonic Pi help
- in relation to other synths - to be well behaved members of the Sonic Pi synth family

Chapter 5 - Recreating the beep synth

In this chapter it all comes together and we rebuild our own version of the Beep synthesiser directly in SuperCollider and integrate it into our Sonic Pi instance.

Chapter 6 - Next steps

What the future holds for this manual and user-defined synths in Sonic Pi

---

TheManual is maintained by gordonguthrie.
This page was generated by GitHub Pages.

The Manual

Writing SuperCollider Synths For Sonic Pi

View on GitHub

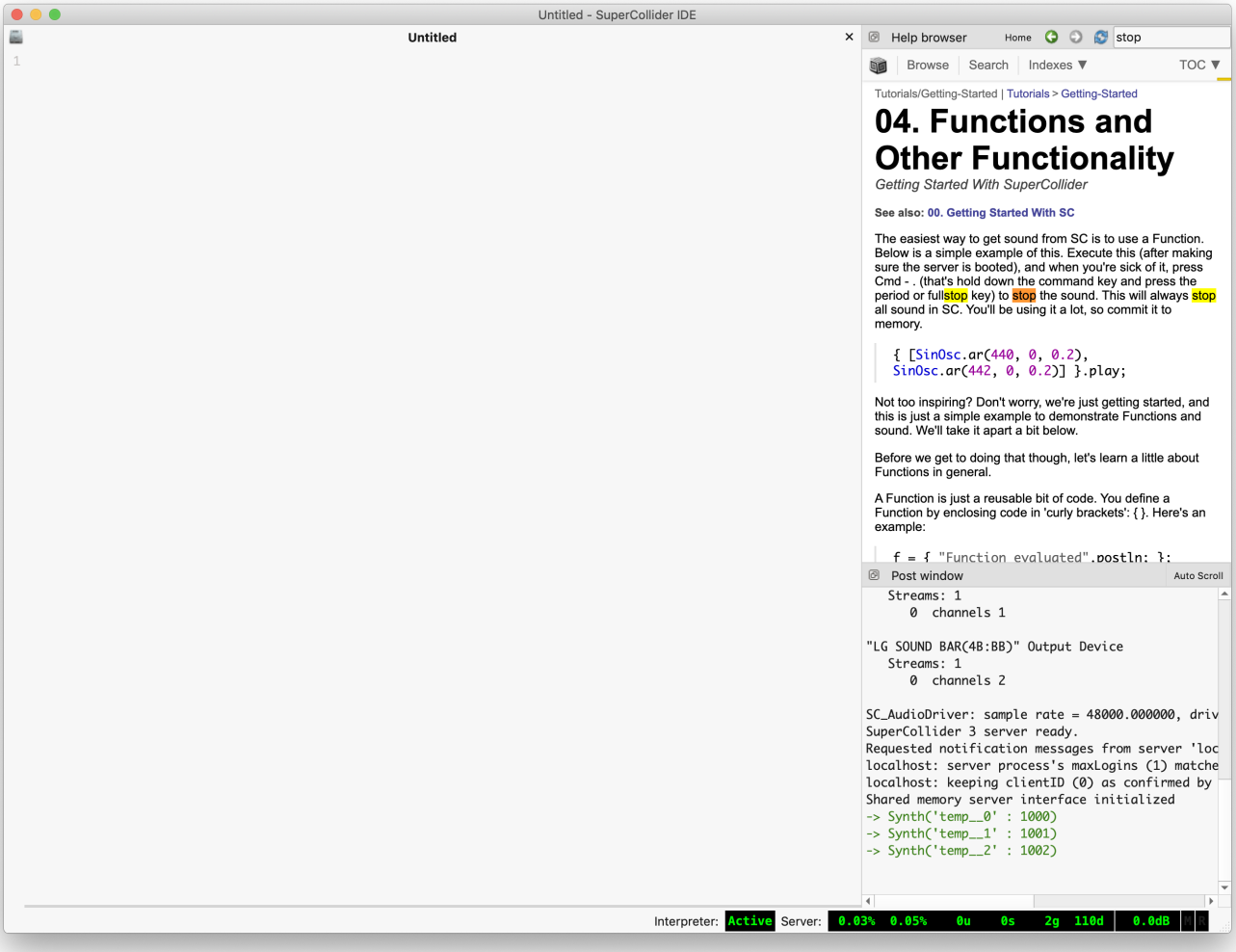
Chapter 1 - First Beeps

Installing SuperCollider

Download SuperCollider for your platform and install it.

(There is a great set of introductory examples on that page to play as well.)

On starting SuperCollider you will see three panels opened up:



The left hand panel `untitled` is the coding panel, the top right has the SuperCollider documentation browser and the bottom right is the logs panel.

Reference

There is a guide to using the code panel (also known as the interpreter).

TheManual is maintained by gordonguthrie.
This page was generated by GitHub Pages.

The Manual

Writing SuperCollider Synths For Sonic Pi

View on GitHub

Chapter 1 - First beeps

First beep in SuperCollider

Most of the synths you are used to calling in Sonic Pi are implemented in Super Collider.

This tutorial is to enable you to write your own synths and for you, or other people, to use them in Sonic Pi.

Obviously if you can write Super Collider code you could just programme Super Collider without Sonic Pi. And this manual will help you learn some, but not all, of how to do just that.

Like Sonic Pi this manual has an overriding design principle - to get from nothing to make a noise as quickly as possible.

With Sonic Pi that's just:

```
play :beep 69
```

With Super Collider it is nearly as fast.

Lets have a go.

To start with you need to tell the SuperCollider App to start the server engine.

This can be done with the menu item: `Server -> Boot Server`

You then need to check the log window to see if there are error messages. Sometimes you will need to edit audio settings on your machine to get SuperCollider to work.

The Super Collider version of `play :beep 69` is:

```
{SinOsc.ar(440, 0, 0.2)}.play;
```

Copy the snippet into the coding window on SuperCollider. Place the cursor in the code line and press `[SHIFT]` `[ENTER]` and you will hear an `A4` note (the Stuttgart pitch for standard tuning).

This will play *forever*.

To stop it you press `[COMMAND][.]` (the command key and the full stop at the same time.

One of the key differences between Super Collider and Sonic Pi is that Sonic Pi is sound-based (play this sound for this time period) and Super Collider is state-based - get this synthesiser into this state. By default sounds in Super Collider are of indefinite duration.

Lets break down that beep:

Its a function:

```
{ ... }
```

We are invoking the play method with the function:

```
{ ...}.play;
```

and the body of the function is a Sine Oscillator:

```
{SinOsc.ar(440, 0, 0.2)}.play;
```

`SinOsc` is an object, a sine wave oscillator.

`ar` is a method called on that object with the parameters `440`, `0` and `0.2`.

If we look up `SinOsc` in the documentation browser we will see that the object exposes two functions with the same signature:

```
SinOsc.ar(freq: 440.0, phase: 0.0, mul: 1.0, add: 0.0)
SinOsc.kr(freq: 440.0, phase: 0.0, mul: 1.0, add: 0.0)
```

We can see that the parameters in our invocation are:

- `440` - the frequency, corresponding to the note A4
- `0` - the phase
- `0.2` - the `mul` (or multiplier)

In this instance the `mul` is effectively the volume, make it higher and the note will sound lounder, lower it will be quieter.

Lets look again at the interface and the two methods `ar` and `kr`. The `ar`/`kr` pairing appears throughout SuperCollider so its important to understand what they mean.

- `ar` means audio rate
- `kr` means control rate

For the moment we will stick to `ar`. Later on when we start building a proper synthesiser for Sonic Pi we will look again at the meaning of these.

The Manual

Writing SuperCollider Synths For Sonic Pi

View on GitHub

Chapter 1 - First beeps

First synth in Sonic Pi

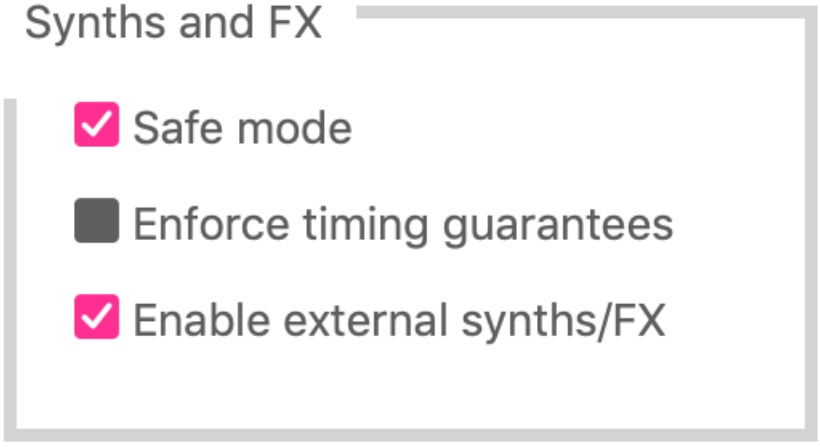
We will make this oscillator into a synthesiser that we can use in Sonic Pi:

```
{SinOsc.ar(440, 0, 0.2)}.play;
```

If we run this definition in Super Collider we can get a synth that we can play in Sonic Pi. Notice that it is writing the synthdef to a default location in my home directory (on a Mac) where Sonic Pi will find it - you will need to put your home directory in whatever operating system you use into that path.

```
(SynthDef("myfirstsynth", {arg out = 0;
     var note, envelope;
     envelope = Line.kr(0.1, 0.0, 1.0, doneAction: 2);
     note = SinOsc.ar(440, 0, envelope);
     Out.ar(out, note);
}).writeDefFile("/Users/gordonguthrie/.synthdefs"))
```

To use it in Sonic Pi we need to do two things. Firstly we need to enable it in the GUI:



Nota Bene/Pay Attention the GUI enables both external synths and external FXs. This manual only covers synths but in theory you could write your own effects too.

Then in Sonic Pi we have to load it and then just use it as normal

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play 69
```

Play about with this. There are a couple of things to notice - the sound will always be coming from one side of the speakers. That's a bit odd. And also as you change the note up and down it makes no difference. It just plays A4 that fades out in 1 second.

We will fix both of these things later on. But first lets breakdown the synth definition:

```
/*
Define the synth - give it a name "myfirstsynth"
let it take one argument: `out`
*/

(SynthDef("myfirstsynth", {arg out = 0;

     // define 2 variables
     var note, envelope;

     /*
     Create a sound envelope that goes from 0.1 to 0 in 1 second
     and when it has done that trigger an action
     that destroys the running instance of
     the synthesiser and frees all memory
     */
     envelope = Line.kr(0.1, 0.0, 1.0, doneAction: 2);

     // define the note we are going to play A4 at 440Hz and set the volume to be the envelope
     note = SinOsc.ar(440, 0, envelope);

     // send the new note to the output channel 0
     Out.ar(out, note);
}).writeDefFile("/Users/gordonguthrie/.synthdefs"))
```

We have had to do a bit more work to get it to play nice with Sonic Pi. It has a few problems:

- it only plays one note (A4) - it needs to play any note
- it only plays one note at a time - it needs to play chords
- it plays in the left speaker only - it needs to be in stereo
- each note is 1 second in duration - we need to be able to control how long a note lasts

But its not all bad - the line that determines the length of the note also calls a self-destruct function that cleans up and frees resources - without it your computer would gradually fill up with unused instances of synthesisers consuming both memory and CPU and eventually would just crash.

In Chapter 3 we will gradually build up this synthesiser until it is a clone of the `sine` synthesiser that is built into Sonic Pi.

We will add the following functions and defaults:

- `note` - default `52` - slideable
- `amp` - default `1` - slideable
- `pan` - default `0` - slideable
- `attack`
- `decay` - default `0`
- `sustain` - default `0`
- `release` - default `1`

- `attack_level`
- `decay_level`
- `sustain_level`
- `sustain_level` - default `1`
- `env_curve` - default `2`

and the following slide options:

- `_slide`
- `_slide_shape`
- `_slide_curve`

But before we can build a proper synth we need to understand a lot of stuff:

- how synths are currently defined
- how they are invoked
- why they are built like they are
- how SuperCollider works

Subsequent chapters of this book step through these in a systematic fashion.

Chapter 1 - First beeps

Setting up Sonic Pi from source

To investigate how Sonic Pi works and become familiar with the code we are going to have do a couple of things:

- install Sonic Pi from source so we can tinker with it
- install the SuperCollider GUI so we can build SuperCollider synths to load into Sonic Pi

The Sonic Pi github has a load of READMEs covering installing on different platforms.

There is a problem tho. When we install Sonic Pi we pull down a SuperCollider server to run but we don't bring the GUI components.

On the Mac OS X - the installed Sonic Pi instance has a pre-built SuperCollider server bundled with it in an executable package. We can just install SuperCollider the app with a GUI alongside it.

With Linux and Raspberry Pi we install the SuperCollider server (but not GUI or CLI) as a dependency alongside our compiled SonicPi.

To muck about with the synths we need to install the other components.

On the Pi we just run:

```
sudo apt-get -y install supercollider
```

for other distros you will need to find the appropriate incantation.

We can now start SuperCollider from the command line:

```
scide
```

Chapter 2 - Existing synths in Sonic Pi

The synths

These are all the synths in Sonic Pi:

Synth name	Where defined
Bass Foundation	SuperCollider
Bass Highend	SuperCollider
Beep	Clojure
Blade	Clojure
Bnoise	Clojure
Chipbass	Clojure
Chiplead	Clojure
Chipnoise	Clojure
Cnoise	Clojure
Dark Ambience	Clojure
Dpulse	Clojure
Dsaw	Clojure
Dtri	Clojure
Dull Bell	Clojure
Fm	Clojure
Gnoise	Clojure
Growl	Clojure
Hollow	Clojure
Hoover	Clojure
Kalimba	SuperCollider

Synth name	Where defined
Kalimba	SuperCollider
Mod Beep	Clojure
Mod Dsaw	Clojure
Mod Fm	Clojure
Mod Pulse	Clojure
Mod Saw	Clojure
Mod Sine	Clojure
Mod Tri	Clojure
Noise	Clojure
Organ Tonewheel	SuperCollider
Piano	SuperCollider
Pluck	Clojure
Pnoise	Clojure
Pretty Bell	Clojure
Prophet	Clojure
Pulse	Clojure
Rodeo	SuperCollider
Saw	Clojure
Sine	Clojure
Sound In	Special
Sound In Stereo	Special
Square	Clojure
Subpulse	Clojure
Supersaw	Clojure
Tb303	Clojure
Tech Saws	Clojure
Tri	Clojure
Winwood Lead	SuperCollider
Zawa	Clojure

There is a directory with all the Clojure synths synths defined in it, and another directory with all the SuperCollider synths.

The synths marked `special` are the ones that use the soundcard as a synthesiser and won't be discussed here.

When you look at the sources you will find lots of other stuff that doesn't appear as a `synth` in Sonic Pi. This is because all the effects are also created in SuperCollider. The sample handling code also uses it.

Later on in this chapter we will investigate how these synths are defined and invoked.

Chapter 2 - Existing synths in Sonic Pi

The Clojure library: Overtone

For earlier versions of Sonic Pi synths were written not in SuperCollider but with a Clojure libary called Overtone.

Remember that FX and sample handling and anything that also affects sound is handled in SuperCollider under the covers and the source code that you find in the Clojure synths directory will include code to do that too.

The reason for this is that it is easier to incorporate Clojure source code into a multi-server compile and tooling chain - but harder to do the same with SuperCollider source which is designed to be saved and compiled inside the SuperCollider built in IDE.

`Sonic PI V5.0.0 Tech Preview 2` has some newer, more sophisticated synths written directly in SuperCollider.

To see how `Sonic PI V5.0.0 Tech Preview 2` handles the built in synths written in `overtone` lets look the file basic.clj

If we scroll down to the bottom we can see the synths being compiled:

```
(comment
  (core/save-synthdef sonic-pi-beep)
  (core/save-synthdef sonic-pi-saw)
  (core/save-synthdef sonic-pi-tri)
  (core/save-synthdef sonic-pi-pulse)
  (core/save-synthdef sonic-pi-subpulse)
  (core/save-synthdef sonic-pi-square)
  (core/save-synthdef sonic-pi-dsaw)
  ...
```

If we scroll back up to the top we can find the definition of the `beep` synth in Clojure:

```
(without-namespace-in-synthdef
 (defsynth sonic-pi-beep [note 52
                          note_slide 0
                          note_slide_shape 1
                          note_slide_curve 0
                          amp 1
                          amp_slide 0
                          amp_slide_shape 1
                          amp_slide_curve 0
                          pan 0
                          pan_slide 0
                          pan_slide_shape 1
                          pan_slide_curve 0
                          attack 0
                          decay 0
                          sustain 0
                          release 1
                          attack_level 1
                          decay_level -1
                          sustain_level 1
                          env_curve 1
                          out_bus 0]
   (let [decay_level (select:kr (= -1 decay_level) [decay_level sustain_level])
         note        (varlag note note_slide note_slide_curve note_slide_shape)
         amp         (varlag amp amp_slide amp_slide_curve amp_slide_shape)
         amp-fudge   1
         pan         (varlag pan pan_slide pan_slide_curve pan_slide_shape)
         freq        (midicps note)
         snd         (sin-osc freq)
         env         (env-gen:kr (core/shaped-adsr attack decay sustain release attack_level decay_lev
     (out out_bus (pan2 (* amp-fudge env snd) pan amp))))
```

Lets have a look at this.

```
defsynth sonic-pi-beep
```

This is obviously the opening of the SuperCollider code:

```
SynthDef("myfirstsynth"
```

The next bit is all the arguments the synth takes (with their defaults).

Then we hit the meat of the beast:

```
   (let [decay_level (select:kr (= -1 decay_level) [decay_level sustain_level])
         note        (varlag note note_slide note_slide_curve
   ...
```

If we go through this code we can see a lot of things that are clearly UGens and methods:

- `select` - the supercollider Select UGen
- `varlag` - the class VarLag
- `midicps` - the class midicps
- `sin-osc` - the SinOsc UGen
- `env-gen` - the EnvGen UGen
- `core/shaped-adsr` - something that invokes asdr somehow
- `out` - the Out UGen

- `pan` - one of the Pan UGen family
- `:action` - related to doneAction somehow

We can also see that when a `UGen` offers a `.kr` and `.ar` option the clojure default is `.ar` and we have to specify `.kr` explicitly. This makes sense as we use `.ar` for sound signals (that we care a lot about) and `.kr` for control signals that we are a bit meh about.

This synth uses the UGen `SinOsc` to generate its output - just like the first synth in Chapter 1. This is the synth we will be recreating in Chapter 3.

If we look at our old first synth definition we can see some of these elements and how we have to compose them:

```
(SynthDef("myfirstsynth", {arg out = 0;

    // define 2 variables
    var note, envelope;

    /*
    Create a sound envelope that goes from 0.1 to 0 in 1 second
    and when it has done that trigger an action
    that destroys the running instance of
    the synthesiser and frees all memory
    */
    envelope = Line.kr(0.1, 0.0, 1.0, doneAction: 2);

    // define the note we are going to play A4 at 440Hz and set the volume to be the envelope
    note = SinOsc.ar(440, 0, envelope);

    // send the new note to the output channel 0
    Out.ar(out, note);
}).writeDefFile("/Users/gordonguthrie/.synthdefs"))
```

So using the Overtone library we can transcribe a SuperCollider definition of a synthesizer into a Lisp format and then use a compiler against that to emit the appropriate compiled `SuperCollider` bytecode for Sonic Pi to use.

We we develop our own version of beep we will reverse engineer this Overtone description into SuperCollider code.

Chapter 2 - Existing synths in Sonic Pi

SuperCollider synths in Sonic Pi

There is a directory SuperCollider with all the SuperCollider synths in it.

This manual will not spend a lot of time looking at these - but as we build out our version of `beep` directly in SuperCollider it will end up looking a lot like any of these ones.

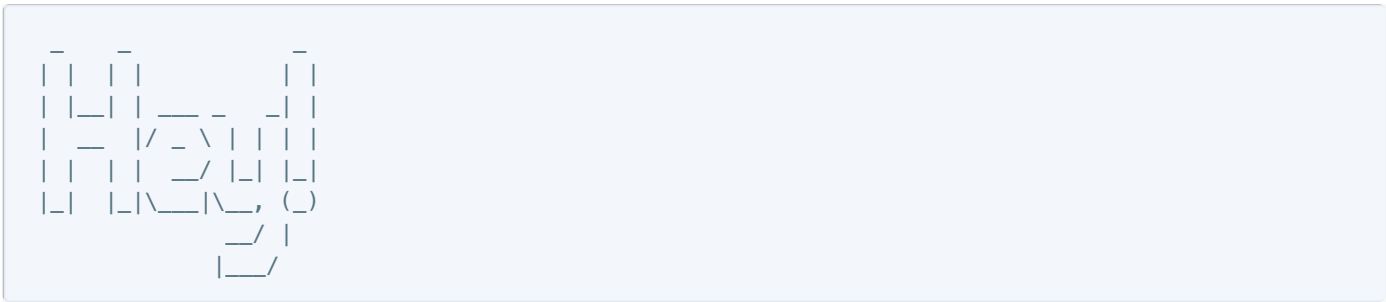
Remember that all sound manipulation in Sonic Pi is done in SuperCollider so don't be surprise when you find things in this directory that aren't synthesisers but FX etc (for instance `autotuner`).

Chapter 3 - Deep dive

How does it all work?

This chapter will help you understand how Sonic Pi plays synths - but it is not strictly necessary for you to understand that to just add your own synths written in SuperCollider.

Stop, look and listen!



Don't feel that you *MUST* read this chapter, you can skip it if you want.

If you want to add your own well-behaved user-defined synth in Sonic Pi and compile it in so it works as native you can just skip this chapter.

But if you want your synth to be a bit funky, to not work quite like the built-in ones then you will need to take a swatch at the source code to figure out why it works as it does and how your synth may or may nor work with all the features of Sonic Pi.

This section doesn't cover all the pathways by which your synth will be called, nor is it exhaustive in the one, main pathway it explores - its job is to cut a track through the jungle - its up to you to make the journey

Small apology

Its been 25 years since last I wrote Ruby in anger, I can sort-of read it still, but not write it, so*caveat lector/reader beware*.

Chapter 3 - Deep dive

Investigating Sonic Pi and error messages

We can learn a lot about how a system works by using it and also looking at error messages, so lets try and break Sonic Pi and see what we can learn.

Lets just muck about with this programme, first with a built-in Sonic Pi synth (by default `beep`) and then with our own one.

Built-in synths

```
play 60
```

We get the log:

```
{run: 8, time: 0.0}
 └─ synth :beep, {note: 60.0}
```

The integer note `60` has become a float `60.0` but we can use floats in our play command quite happily too:

```
play 60.0
```

gives:

```
{run: 42, time: 0.0}
 └─ synth :beep, {note: 60.0}
```

We haven't named the note argument but we can, and we get the same result:

```
play note: 60
```

We can swap the number of the note out for a symbol:

```
play :c4
```

Which resolves to the Midi note `60` as we expect:

```
{run: 24, time: 0.0}
 └─ synth :beep, {note: 60.0}
```

But if we play not a `note` but `notes` we get a different result:

```
play notes: [60, 62, 65]
```

```
{run: 37, time: 0.0}
 └─ synth :beep, {note: [60.0, 62.0, 65.0]}
```

So the `notes` parameter has magically been transformed into `note` with a list.

What happens if we use a `notes` parameter without a list?

```
play notes: 60
```

The sound has changed - the note played is different. What's that about?

```
{run: 38, time: 0.0}
 └─ synth :beep, {notes: 60, note: 52.0}
```

So whereas when we passed a list tagged `notes` SonicPi recognised it as a list and said, "ooh, notes is just the plural of note, lets move this value to the note slot".

We can go crazy and bung a nutty list in there:

```
play notes: [60, chord(:c3, :minor), [55, 57, 59]]
```

and Sonic Pi just goes "you are having a giraffe m8":

```
{run: 40, time: 0.0}
 └─ synth :beep, {note: [60.0, nil, nil]}
```

Sonic Pi just saying "I don't know what these things are but they are not numbers so into the bin with them".

We can change the note to a chord:

```
play chord(:e3, :minor)
```

Now the synthesiser is turning that chord into a list of notes:

```
{run: 18, time: 0.0}
 └─ synth :beep, {note: [52.0, 55.0, 59.0]}
```

But a chord isn't a list of notes, its a data structure called a ring:

```
 print(chord(:e3, :minor))
```

gives:

```
(ring <SonicPi::Chord :E :minor [52, 55, 59])
```

So something inside Sonic Pi is taking the ring data structure (which contains a list and information about the list like how long it is and code to enable indexes to be mapped to an element in the list) and pulling the list out.

What we type into Sonic Pi is not what is being played - the term of art for this is munging - the inputs are being munged into something else.

We can play a chord explicitly too:

```
play [52, 55, 59]
```

giving:

```
{run: 20, time: 0.0}
 └─ synth :beep, {note: [52.0, 55.0, 59.0]}
```

Obviously we can add other attributes:

```
play :c4, pan: 0.3
```

gives:

```
{run: 25, time: 0.0}
 └─ synth :beep, {note: 60.0, pan: 0.3}
```

Lets add a non-existent parameter to our call:

```
play :c4, pan: 0.3, gordon: 99
```

It seems to be kept:

```
{run: 28, time: 0.0}
 └─ synth :beep, {note: 60.0, pan: 0.3, gordon: 99}
```

Just for badness, trust me it will make sense later, lets go again with a non-option but make it `out`:

```
play 60, out: 3
```

and as you would expect the note plays:

```
{run: 55, time: 0.0}
 └─ synth :beep, {note: 60.0, out: 3}
```

What happens if we try and leave out the note, is there a default value here? The result from passing in a single value in `notes` would tend to suggest there is:

```
play pan: 0.3
```

as we suspected:

```
{run: 36, time: 0.0}
 └─ synth :beep, {pan: 0.3, note: 52.0}
```

But look what happens when we try and pass a duff value to a known parameter:

```
play :c4, pan: [3, 4, 5], gordon: 99
```

we get an error message:

```
Runtime Error: [buffer 5, line 1] - RuntimeError
Thread death!
Unable to normalise argument with key :pan and value [3, 4, 5]
```

And the same if we try and bust the bounds. The `pan` parameter places the sound on the left/right pan with -1.0 being hard left and 1.0 being hard right:

```
play 60, pan: 3
```

throws an error:

```
Runtime Error: [buffer 5, line 5] - RuntimeError
Thread death!
Value of opt :pan must be a value between -1 and 1.0 inclusively, got 3
```

And if we try and use `play` without a number, symbol or list we also get a crash:

```
play :juicyfruit
```

resulting in:

```
Runtime Error: [buffer 5, line 5] - SonicPi::Note::InvalidNoteError
Thread death!
Invalid note: :juicyfruit
```

Custom synths

If we switch to our `myfirstsynth` we get a slightly different story:

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play 60
```

This plays (in mono) the same sound as `beep`:

```
{run: 29, time: 0.0}
 └─ synth :myfirstsynth, {note: 60}
```

We can throw options (sensible or otherwise at it):

```
 load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play 60, pan: 3, gordon: 99
```

and it just plays:

```
{run: 30, time: 0.0}
 └─ synth :myfirstsynth, {note: 60, pan: 3, gordon: 99}
```

`pan` is sanity checked to be between `-1.0` and `1.0` when we use a built-it synth, that doesn't happen here.

When we read the code for our synthesizer we realise this is a bit odd. Our function only takes one argument `out` and yet we are calling it with 3, none of which is `out`.

```
(SynthDef("myfirstsynth", {arg out = 0;
     var note, envelope;
     envelope = Line.kr(0.1, 0.0, 1.0, doneAction: 2);
     note = SinOsc.ar(440, 0, envelope);
     Out.ar(out, note);
}).writeDefFile("/Users/gordonguthrie/.synthdefs"))
```

So whatever parameters we send are all being chucked away - apart from `out`. What happens if we actually pass in an `out` parameter:

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play 60, out: 3
```

Well the logs say all is well:

```
{run: 55, time: 0.0}
 └─ synth :beep, {note: 60.0, out: 3}
```

but actually there is no sound. To understand this we need to trace through where the `out` value is used in our code. We use it as the first parameter in the uGen `Out`. It determines the output channel. The way Sonic Pi is wired up the channel `0` makes our computer play noise, any other channel is not connected to something to turn signal into sound - hence the silence.

With a built-in synthesiser we can set `out` to `3` but when the synth is called the output plays - implying that the `out` parameter has been set to `0` in the munging.

Lets look at some other incantations - particularly around notes.

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play note: [60, 62, 66]
```

This no longer works:

```
Runtime Error: [buffer 5, line 5] - RuntimeError
Thread death!
Unable to normalise argument with key :note and value [60, 62, 66]
```

Switching to `notes` doesn't help either:

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play notes: [60, 63, 65]
```

giving (essentially) the same error:

```
Runtime Error: [buffer 5, line 5] - RuntimeError
Thread death!
Unable to normalise argument with key :notes and value [60, 62, 66]
```

You can't play chords either. But a note-free incantation still works:

```
load_synthdefs "/Users/gordonguthrie/.synthdefs"

use_synth(:myfirstsynth)

play pan: 44
```

Giving:

```
{run: 60, time: 0.0}
 └─ synth :myfirstsynth, {pan: 44}
```

So what have we learned?

We have learned that Sonic Pi monkey's about with the parameters you have passed in before it sends them on to SuperCollider.

If you are using a built-in synth, Sonic Pi checks your parameters systematically - but passes on additional parameters unchecked - this makes Sonic Pi work seemlessly, if you switch a built-in synth with additional parameters out for a simpler one the extended values are silently dropped.

By contrast with a synth that Sonic Pi doesn't recognise - it just sends all the parameters unchanged to the synth.

In the next section we will look at ways to find out what is happening in the code, and in the one after that we will peek inside Sonic Pi to figure out what's really going on.

Chapter 3 - Deep dive

Loggin, loggin, loggin

To mess around with the bit of SonicPi that handles synths you will need to download the source and compile your own instance of SonicPi.

The various markdown files at the root level in the source code have instructions on how to build for many platforms.

For this exploration we will be looking at the `ruby` part of SonicPi.

Luckily once SonicPi is built this is very straightforward. `ruby` is an interpreted and not a compiled language and by simply editing `ruby` source code and stopping and restarting SonicPi we can see the changes.

Once we have compiled a built SonicPi we can start and run it by invoking the binary `sonic-pi` which is created in the directory `app/build/gui/qt`.

Lets look at 4 techniques for understanding what is going on:

- existing log messages
- built in messaging inside the runtime
- logging during boot
- native Ruby Logging

but before we do, beware false friends!

False friends

The Sonic Pi language has a couple of *false friend* functions - things that look like they will be helpful in this context, but they mostly aren't.

They are the commands `use_debug` and `with_debug` in the language reference. They only affect logging of synth triggers to the front end.

If we run the following code in the SonicPI gui:

```
use_synth :bass_foundation

play 60
```

we see the following log message in the log window of the GUI:

```
=> Starting run 6

{run: 6, time: 0.0}
 └─ synth :bass_foundation, {note: 60.0}
```

If we now add the `use_debug` command the log message goes away:

```
use_debug false
use_synth :bass_foundation

play 60
```

This is just a convenience function for front end users and not a proper debugging tool.

Existing log messages

Sonic Pi writes its log to the directory `~/.sonic-pi/log`. If we pop in there we can see a useful set of logs:

```
gordon@raspberrypi:~/.sonic-pi/log $ ls
daemon.log  gui.log  jackd.log    spider.log    tau.log
debug.log   history  scsynth.log  tau_boot.log
```

You can get a lot more info if you go into theutil module and set the debug mode to `true` tho:

```
    def debug_mode
      false
    end
```

BEWARE: there is more than one module called `util.lib` you want the one in `/app/server/ruby/lib/sonicpi/`

Built in messaging inside the runtime

When we run code in the Sonic Pi like:

```
load_synthdefs "/home/gordon/.synthdefs"

use_synth :myfirstsynth

play 60
```

we see messages on the logging tab like this

```
=> Starting run 3

=> Loaded synthdefs in path: /home/gordon/.synthdefs
   - /home/gordon/.synthdefs/myfirstsynth.scsyndef

=> Completed run 3
```

If we grep the string `Loaded synthdefs` we can find the origin - in the module sound.rb:

```
      def load_synthdefs(path=Paths.synthdef_path)
        raise "load_synthdefs argument must be a valid path to a synth design. Got an empty string." 
        path = File.expand_path(path)
        raise "No directory or file exists called #{path.inspect}" unless File.exist? path
        if File.file?(path)
          load_synthdef(path)
        else
          @mod_sound_studio.load_synthdefs(path)
          sep = "   - "
          synthdefs = Dir.glob(path + "/*.scsyndef").join("#{sep}\n")
          __info "Loaded synthdefs in path: #{path}
#{sep}#{synthdefs}"
        end
      end
      doc name:          :load_synthdefs,
          introduced:    Version.new(2,0,0),
          summary:       "Load external synthdefs",
          doc:           "Load all pre-compiled synth designs in the specified directory. This is use

...
```

The function `__info` that is being called to write the msg to the front end is found in the module runtime.rb:

```
 def __info(s, style=0)
      __msg_queue.push({:type => :info, :style => style, :val => s.to_s}) unless __system_thread_loca
    end
```

We can prove it is this definition by adding another message push as so:

```
 def __info(s, style=0)
      __msg_queue.push({:type => :info, :style => style, :val => "banjo"}) unless __system_thread_loc
      __msg_queue.push({:type => :info, :style => style, :val => s.to_s}) unless __system_thread_loca
    end
```

So now when we stop and start Sonic Pi and run the same code we see that every msg we get in the front end, we get a `banjo` before it.

```
=> Starting run 3

=> banjo

=> Loaded synthdefs in path: /home/gordon/.synthdefs
   - /home/gordon/.synthdefs/myfirstsynth.scsyndef

=> banjo

=> Completed run 3
```

So by simply adding lines to our ruby that calls this `__info` function we can see what's going on when we do stuff.

Logging during boot

The runtime logging is great but what happens when you want to figure out what is happening during boot before the GUI is available to show your messages?

Well it turns out that Sonic Pi has that sorted too. You can write a message to a buffer and when the boot is

completed the buffer is dumped into the log window. Lets see that in action in the studio module which handles the boot process and the creation of the GUI.

If I invoke the function `message` with a string, as I do here, it will appear in the log screen on boot.

```
    def init_scsynth
      message "bingo bongo dandy dongo"
      @server = Server.new(@scsynth_port, @msg_queue, @state, @register_cue_event_lambda, @current_spi
      message "Initialised SuperCollider Audio Server #{@server.version}"
    end
```

and as expected printing my message in the GUI's log window:

```
ome to Sonic Pi v5.0.0-Tech Preview 2

=> Running on Ruby v2.7.4

=> Initialised Erlang OSC Scheduler

=> Initialised SuperCollider Audio Server v3.11.2

=> bingo bongo dandy dongo

=> Remember, when live coding music
   there are no mistakes
   only opportunities to learn
   and improve.

=> Let the Live Coding begin...

=> Has Sonic Pi made you smile?

   We need *your* help to fund further development!

   Sonic Pi is not financially supported by
   any organisation.

   We are therefore crowdsourcing funds from kind
   people like you using Patreon.

   We need at least 1000 supporters to continue.
   Currently we have 733 generous individuals.

   Please consider becoming a Patreon supporter too,
   and help us keep Sonic Pi alive:

   https://patreon.com/samaaron
```

Native ruby logging

Sometimes, maybe, the front end logging might not be enough.

In that case we can use the built in `ruby` Logger.

We can now sprinkle the code with log calls and try and figure out how the server works.

Using Logger is pretty straightforward, you need to load the library into the module, create a new logger with a fully qualified filename to log to and write a log statement:

```
require 'logger'
...
logger = Logger.new("/home/gordon/Dev/tmp/sonic_pi.log")
logger.debug("normalising synth args")
```

I am telling Logger to use a file in my home directory, you need to get it write it to wherever suits you. The file must already exist and the path must be fully qualified so no `../..` or `~`s.

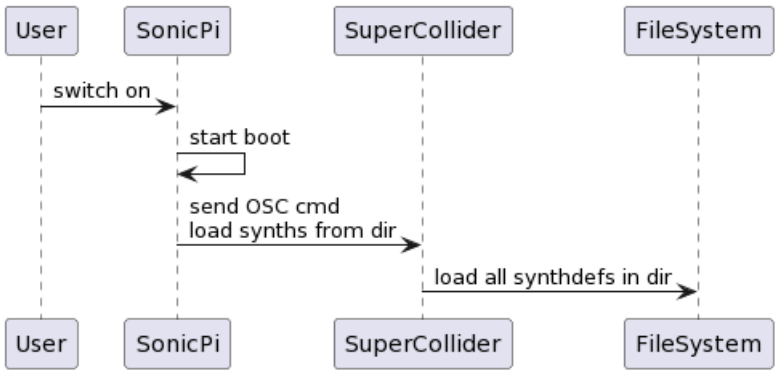
Chapter 3 - Deep dive

##

How built-in synthdefs are loaded

Lets go hunting for the code that loads synth definitions and see what it does.

This is the boot sequence:



Using a combination of the logging techiques in the previous section we can soon find out roughly how it works.

The very earliest message we see in the GUI is:

```
=> Welcome to Sonic Pi v5.0.0-Tech Preview 2
```

This message is sent from the runtime module:

```
  def __load_buffer(id)
      id = id.to_s
      raise "Aborting load: file name is blank" if  id.empty?
      path = File.expand_path("#{Paths.project_path}/#{id}.spi")
      s = "# Welcome to Sonic Pi\n\n"
      if File.exist? path
        s = IO.read(path)
      end
      __replace_buffer(id, s)
    end
```

That function `__load_buffer` is called in one place and one place only, by the spider server.

We know that the `spider-server` is special because it sits in `app/bin` and not `app/server/ruby/lib/sonicpi` like

the rest of the ruby code.

Spider has its own log in `~/.sonic-pi/logs` and if we look at them we can figure out what's going on:

```
Sonic Pi Spider Server booting...
The time is 2023-03-26 13:02:47 +0100
Using primary protocol: udp
Detecting port numbers...
Ports: {:server_port=>37064, :gui_port=>37065, :scsynth_port=>37066, :scsynth_send_port=>37066, :osc_
Token: 520542600
Opening UDP Server to listen to GUI on port: 37064
Spider - Pulling in modules...
Spider - Starting Runtime Server
TauComms - Sending /ping to tau: 127.0.0.1:37067
TauComms - Receiving ack from tau
TauComms - connection established
studio - init
scsynth boot - Waiting for the SuperCollider Server to have booted...
scsynth boot - Sending /status to server: 127.0.0.1:37066
scsynth boot - Receiving ack from scsynth
scsynth boot - Server connection established
scsynth - clear!
scsynth - clear schedule
scsynth - schedule cleared!
scsynth - group clear 0
scsynth - group clear 0 completed
Studio - Initialised SuperCollider Audio Server v3.11.2
Studio - Resetting server
Studio - Reset and setup groups and busses
Studio - Clearing scsynth
scsynth - clear schedule
scsynth - clear scsynth
scsynth - clear!
scsynth - clear schedule
scsynth - schedule cleared!
scsynth - group clear 0
scsynth - group clear 0 completed
scsynth - cleared scsynth
scsynth - bus allocators reset
Studio - Allocating audio bus
Studio - Create Base Synth Groups
Studio - Starting mixer
Studio - Starting scope
Spider - Runtime Server Initialised
Spider - Registering incoming Spider Server API endpoints
Spider - Booted Successfully.
Spider - v5.0.0-Tech Preview 2, OS raspberry, on Ruby  2.7.4 | 2.7.0.
Spider - ----------------------------------------
```

It co-ordinates the dance with the Tau server that handles timing and events and the SuperCollider server which actually makes the sounds.

Somewhere in here it starts up the sound module that actually starts the `studio`.

If we examine the `initialize` method of the class Studio and match what happens we can see how the synth definitions are loaded:

```
    def initialize(ports, msg_queue, state, register_cue_event_lambda, current_spider_time_lambda)

      STDOUT.puts "studio - init"
      STDOUT.flush

      @state = state
      @scsynth_port = ports[:scsynth_port]
      @scsynth_send_port = ports[:scsynth_send_port]
      @msg_queue = msg_queue
      @error_occured_mutex = Mutex.new
      @error_occurred_since_last_check = false
      @sample_sem = Mutex.new
      @reboot_mutex = Mutex.new
      @rebooting = false
      @cent_tuning = 0
      @sample_format = "int16"
      @paused = false
      @register_cue_event_lambda = register_cue_event_lambda
      @current_spider_time_lambda = current_spider_time_lambda
      @global_timewarp = 0
      init_scsynth
      reset_server
      init_studio
    end
```

If we trace down the last three function invocations and match their log messages to those in the `spider.log` we can see this function setting everything up for the user - after the SuperCollider and Tau engines have both started.

The last function to run on creating a new Studio is init_studio and it loads the synthdefs:

```
    def init_studio
      @server.load_synthdefs(Paths.synthdef_path)
      @amp = [0.0, 1.0]
      @server.add_event_handler("/sonic-pi/amp", "/sonic-pi/amp") do |payload|
        @amp = [payload[2], payload[3]]
      end
```

Actually it tells the server to load them: which it does by sending an OSC message to SuperCollider:

```
    def load_synthdefs(path)
      info "Loading synthdefs from path: #{path}" if @debug_mode
      with_done_sync [@osc_path_d_loaddir] do
        osc @osc_path_d_loaddir, path.to_s
      end
    end
```

The format of the message is an instruction to load code and a filepath - so at this stage Sonic Pi doesn't know anything more about the built in synthesisers other than their location `etc/synthdefs/compiled`.

If we pop a new compiled synthdef in here it will load on boot automatically.

Chapter 3 - Deep dive

How Sonic Pi plays synths

Lets put some logging in and see what happens when we write a simple synth command:

```
use_synth :beep
play chord(:E3, :minor) , pan: -0.3, george: 44
```

use_synth function in sound.rb

The function use_synth sets the name of the synth into a shared memory area where it can be used later.

play function in sound.rb

Then the function play is called - it does some preparatory work on setting up the call to the synthesisers - in particular taking an unnamed first value `n` passed in that isn't a hash of any sort and tagging it as `{note: n}`.

synth function in sound.rb

It then calls the function synth. If the option to use external synths isn't checked and the synth isn't a built-in one it will crash out with an error here. If no synthesiser is specified in this call (which in our example there won't be) then the synth name is taken from the thread-shared storage that `use_synth` popped it into.

`synth` does a call to `Synths::SynthInfo.get_info(sn_sym)` to pick up the information about the synth - this will be used later on.

This is the critical part for the difference between handling built-in synths and user-defined ones. If the call to `get_info` returns `nil` then SonicPi knows that its not a built-in synth and will simply not try and use the validation that comes with built-in synths.

In `Sonic PI V5.0.0 Tech Preview 2` code for built in synths is extended over a base class called BaseInfo in the file `synthinfo.rb`.

The class has a whole range of functions which must be overwritten in implementing a new synth. Some refer to the lifetime of the synth like `initialize`, `on_start` and `on_finish`, some are invoked at runtime like `munge_opts` and some relate to how the synth presents to Sonic Pi like `arg_doc` and `introduced`.

The functions in `synthinfo.rb` and its role in defining the behaviour of Sonic Pi will be covered extensively in *Chapter 4 - the world of built-in synths*.

`synth` takes the arguments passed in and call the external utility function resolve_synth_opts_hash_or_array which does the first munge - it looks at the data structure that is passed in and checks it is an object that it can use, or it needs to be sanitised elsewhere. If this function is called with an `SPVector` it is sent off to merge_synth_arg_maps_array to fix up.

Next `synth` checks if the note is a rest note - and if it does it returns nothing.

Now we start getting to where built-in and user-defined synths are treated differently.

`synth` checks if the synth info is `nil` - if it isn't it then knows that this is a built-in synth and is well behaved.

There are a number of global settings that can be applied to code blocks to change the notes being played:

- `use_cent_tuning` / `with_cent_tuning`
- `use_octave` / `with_octave`
- `use_transpose` / `with_transpose`

If the synth is well behaved these global settings will be applied in normalise_transpose_and_tune_note_from_args.

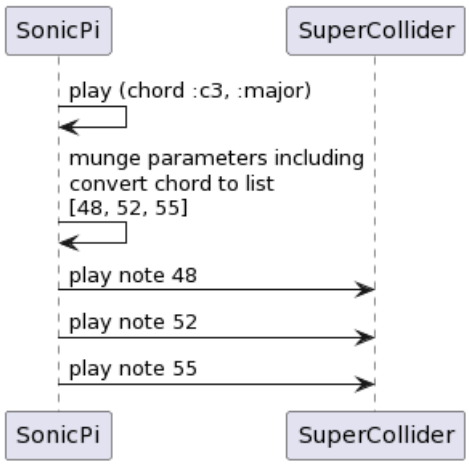
Earlier we looked at error messages and saw that you could play chords with built-in synths but not with user-defined ones.

Playing chords requires a transform which is only done for built-in functions.

A call to a built-in synth may (if it is a chord) be passed onto the function trigger_chord but if it is a user-defined one it will always be passed to trigger_inst.

trigger_chord function in sound.rb

Nota Bene/Take Note: the function `trigger_chord` *DOESN'T* call the synth in SuperCollider and pass it a chord - it asks SuperCollider to play each note separately.



IT does some housekeeping - including calling normalise_and_resolve_synth_args - to make sure that SuperCollider behaves well - the synth that plays each note is grouped, the volume of each note is normalised - the volume of each note is divided by the number of the notes so that the chord as a whole sounds as loud as the specified volume.

`trigger_chord` ends up calling trigger_synth.

trigger_inst function in sound.rb

`trigger_inst` does a little housekeeping - including calling normalise_and_resolve_synth_args and possibly tweaking the slide times in add_arg_slide_times - before moving on to calling trigger_synth.

normalise_and_resolve_synth_args function in sound.rb

Lets look at how Sonic Pi handles synth arguments in some more details. Here is the function:

```
      def normalise_and_resolve_synth_args(args_h, info, combine_tls=false)
        purge_nil_vals!(args_h)
        defaults = info ? info.arg_defaults : {}
        if combine_tls
          t_l_args = __thread_locals.get(:sonic_pi_mod_sound_synth_defaults) || {}
          t_l_args.each do |k, v|
            args_h[k] = v unless args_h.has_key? k || v.nil?
          end
        end
```

This block handles the use of synth defaults and we can see if it we run code like this in Sonic Pi:

```
use_synth_defaults amp: 0.5, pan: -1

play 50
```

In this case the synth defaults are stashed and retrieved by the call to `get` the `:sonic_pi_mod_sound_synth_defaults` setting `t_l_args` to `(map amp: 0.5, pan: -1)`.

The function `normalise_args!` later on turns options like `bpm_scale` which take `true` or `false` as options into numerical arguments - so `1.0` and `0.0`.

If we pass in a `duration` by calling this code:

```
play note: 44, duration: 0.3, pan: -0.3, george: 44
```

when we log the transform we see that a sustain has been added by the function `calculate_sustain!`:

```
synth :beep, {note: 44.0, pan: -0.3, george: 44, sustain: 0.3}
```

at the end of `normalise_and_resolve_synth_args` all the user supplied arguments have been tidied up and made coherent.

If we try the same thing with our custom synth we see that these transforms have also been made:

```
use_synth :myfirstsynth
play note: 44, duration: 0.3, pan: -0.3, george: 44
```

is transformed to:

```
synth :myfirstsynth, {note: 44.0, pan: -0.3, george: 44, sustain: 0.3}
```

trigger_synth function in sound.rb

This function actually makes the sound happen - but before it does that it does validation of the arguments in validate_if_necessary!

This call to `validate_if_necessary!` is the end of our deep dive. This function takes the current synth object from all the way back up in the call to `play` and asks it to validate itself.

If the synth is built-in, it calls its validator function and borks if the parameters are invalid. If the synth is user-defined there is no validator and the parameters are sent across to SuperCollider as-is.

What you need and don’t need to know to write your own synth

You don’t need to know anything of this chapter to write your own well-behaved synthesiser - to write a badly-behaved one, this spelunk should get you started.

Chapter 4 - How built-in synths are defined in Sonic Pi

Philosophy

One of the working philosophies of Sonic Pi is that the tech shouldn't get in the way of experimentation.

All the built-in synths share common parameters - some have additional parameters. The idea is that if you have some running code (or are live coding) and you swap out one synth for another then bad things SHOULDN'T happen - it should behave much as you expected and not in a surprising way.

This section is going to look at synthinfo.rb.

BaseInfo

All synthesiser objects inherit from the base class `BaseInfo` mostly by the chain of indirection: `SonicPiSynth < SynthInfo < BaseInfo`.

But looking at all the synthesiser classes we see a simple pattern synths come in families and often descend from a common base class.

Note that sometimes synth names are just aliases for each other `sine` / `beep` and `mod_sine` / `mod_beep`. This aliasing happens in the global variable @@synth_info.

To be recognised as a synth you need to be added to the global variable `@@synth_infos` in `synthinfo.rb`.

We can tell what functions are designed to be implemented in the sub-classes by looking for base class members that will blow up if they are not. Examining the code we see:

```
    def doc
      "Please write documentation!"
    end

    def arg_defaults
      raise "please implement arg_defaults for #{self.class}"
    end

    def name
      raise "please implement name for synth info: #{self.class}"
    end

    def category
      raise "please implement category for synth info: #{self.class}"
    end

    def prefix
      ""
    end

    def synth_name
      raise "Please implement synth_name for #{self.class}"
    end

    def introduced
      raise "please implement introduced version for synth info: #{self.class}"
    end

    def trigger_with_logical_clock?
      raise "please implement trigger_with_logical_clock? for synth info: #{self.class}"
    end
```

There is an extra function `specific_arg_info` which isn't in this list and by default which returns an empty hashes.

Lets go through them one by one.

function doc

This function is called when the displaying the gui. If we go to the `doc` function for the `beeb` synth and edit it to add the word *yowza*:

```
    def doc
      "A simple pure sine wave, yowza. The sine wave is the simplest, purest sound there is and is
    end
```

This function is called during the compile process (not at run time) and is used to generate the entry about the synth in the gui.

function arg_defaults

Here is the `arg_defaults` function of the `beep` synthesizer:

```
      {
         :note => 52,
         :note_slide => 0,
         :note_slide_shape => 1,
         :note_slide_curve => 0,
         :amp => 1,
         :amp_slide => 0,
         :amp_slide_shape => 1,
         :amp_slide_curve => 0,
         :pan => 0,
         :pan_slide => 0,
         :pan_slide_shape => 1,
         :pan_slide_curve => 0,

         :attack => 0,
         :decay => 0,
         :sustain => 0,
         :release => 1,
         :attack_level => 1,
         :decay_level => :sustain_level,
         :sustain_level => 1,
         :env_curve => 2
       }
     end
```

It is simply the list of all the arguments and their default values - note how the values are chained - the default value of `:decay_level` is defined as `:sustain_level`. (The chaining is only 1 level deep - you can chain a variable to the value of another one, but that one needs an actual value.)

Oftentimes this function is shared between multiple synths by use of an intermediate class. See later on where the following functions inherit their arguments from the `Noise` synth:

- BNoise

- ChipNoise
- CNoise
- GNoise
- PNoise

If you are writing a family of synths you should consider this strategy.

The *values* here show one side of the story - but the function `default_arg_info` contains another:

```
      def default_arg_info
        {
          :note =>
          {
            :doc => "Note to play. Either a MIDI number or a symbol representing a note. For example:
            :validations => [v_positive(:note)],
            :modulatable => true
          },

          :note_slide =>
          {
            :doc => "Amount of time (in beats) for the note to change. A long slide value means that
            :validations => [v_positive(:note_slide)],
            :modulatable => true,
            :bpm_scale => true
          },
          ...
```

This function contains a big set of standard, well-named common parameters. Different synthesisers support different subsets (and different families like the detuned ones or the mod ones or the pulse ones) support similar sets of parameters.

- `note`
- `note_slide`
- `note_slide_shape`
- `note_slide_curve`
- `amp`
- `amp_slide`
- `pan`
- `pan_slide`
- `attack`
- `decay`
- `sustain`
- `release`
- `attack_level`
- `decay_level`
- `sustain_level`
- `env_curve`
- `cuttoff`
- `cuttoff_slide`
- `detune`
- `detune_slide`
- `mod_phase`
- `mod_phase_offset`
- `mod_phase_slide`
- `mod_range`
- `mod_range_slide`

- `res`
- `res_slide`
- `pulse_width`
- `pulse_width_slide`
- `mod_pulse_width`
- `mod_pulse_width_slide`
- `mod_wave`
- `mod_invert_wave`

Using these parameters (where appropriate) with these names and the default validations in them will determine if your synthesiser *feels like* a well behaved SonicPi synthesiser.

To understand that better you will need to study the synthesiser definitions in SonicPi and figure out which synth uses which parameter and then dig in and see how it is defined in the synthdefs.

Some, of course, will be in `Overtone` and you will have to reverse engineer the underlying `SuperCollider` form.

function name

This is the name of the synth as it appears in the GUI - the name you use in code is defined in the function `synth_name`.

functions category and prefix

Both of these are preset for you during the class inheritance chain: `MySynth > SynthInfo > BaseInfo`

```
    class SynthInfo < BaseInfo
      def category
        :general
      end

      def prefix
        "sonic-pi-"
      end
    end
```

Remember that SonicPi uses SuperCollider to:

- define and play synths
- define and wire up FX
- play samples

This code base is used to support all three - but the dip into `SynthInfo` makes a synth a synth and your class invoked wherever synths are in play.

function synth_name

This is the name of the synth as used in SonicPi code - all lowercase and spaces replaced with `_`s.

These names are also aliased in the definition of `@@synth_infos`

function trigger_with_logical_clock

This function is used by FXs and not synths - so don't worry about it.

function specific_arg_info

The function `specific_arg_info` lets you do validation on arguements that you have added to your synth that aren't part of the set that was discussed in the section `arg_defaults` - they take the same format.

In addition synths commonly add the functions `arg_defaults` and `specific_arg_info`.

Synth name	Base Class	arg_defaults	specific_arg_info
Bass Foundation	SonicPiSynth	Yes	
Bass Highend	SonicPiSynth	Yes	Yes
Beep/SynthViolin	SonicPiSynth	Yes	
Blade	SonicPiSynth	Yes	Yes
Bnoise	Noise		
Chipbass	SonicPiSynth	Yes	Yes
Chiplead	SonicPiSynth	Yes	Yes
Chipnoise	Noise	Yes	Yes
Cnoise	Noise		
Dark Ambience	SonicPiSynth	Yes	Yes
Dpulse	Dsaw	Yes	Yes
Dsaw	SonicPiSynth	Yes	
Dtri	Dsaw		
Dull Bell	SonicPiSynth	Yes	
Fm	SonicPiSynth	Yes	Yes
Gnoise	Noise		
Growl	SonicPiSynth	Yes	
Hollow	SonicPiSynth	Yes	Yes
Hoover	SonicPiSynth	Yes	
(Synth) Kalimba	SonicPiSynth	Yes	Yes
Mod Beep	alias for ModSine	Yes	
Mod Dsaw	SonicPiSynth	Yes	
Mod Fm	FM	Yes	
Mod Pulse	SonicPiSynth	Yes	
Mod Saw	SonicPiSynth	Yes	
Mod Sine	SonicPiSynth	Yes	
Mod Tri	SonicPiSynth	Yes	
Noise	Pitchless	Yes	

Synth name	Base Class	arg_defaults	specific_arg_info
Organ Tonewheel	SonicPiSynth	Yes	Yes
(Synth) Piano	SonicPiSynth	Yes	Yes
(Synth) Pluck	SonicPiSynth	Yes	Yes
Pnoise	Noise		
Pretty Bell	DullBell		
Prophet	SonicPiSynth	Yes	
Pulse	Square	Yes	
(Synth) Rodeo	SonicPiSynth	Yes	Yes
Saw	Beep	Yes	
Sine	alias for Beep	Yes	
Square	SonicPiSynth	Yes	
Subpulse	Pulse	Yes	Yes
Supersaw	SonicPiSynth	Yes	
Tb303	SonicPiSynth	Yes	Yes
Tech Saws	SonicPiSynth	Yes	
Tri	Pulse		
Winwood Lead	SonicPiSynth	Yes	Yes
Zawa	SonicPiSynth	Yes	Yes

The base class broadly defines a well-behaved Sonic Pi synth, particularly in the function default_arg_info which defines a complete set of arguments most built-in synthesisers accept.

Chapter 4 - How built-in synths are defined in Sonic Pi

The synth beep

Lets look at this, as it is what we are reimplementing:

```
class Beep < SonicPiSynth
  def name
    "Sine Wave"
  end

  def introduced
    Version.new(2,0,0)
  end

  def synth_name
    "beep"
  end

  def doc
    "A simple pure sine wave. The sine wave is the simplest, purest sound there is and is the fun
  end

  def arg_defaults
    {
      :note => 52,
      :note_slide => 0,
      :note_slide_shape => 1,
      :note_slide_curve => 0,
      :amp => 1,
      :amp_slide => 0,
      :amp_slide_shape => 1,
      :amp_slide_curve => 0,
      :pan => 0,
      :pan_slide => 0,
      :pan_slide_shape => 1,
      :pan_slide_curve => 0,

      :attack => 0,
      :decay => 0,
      :sustain => 0,
      :release => 1,
      :attack_level => 1,
      :decay_level => :sustain_level,
      :sustain_level => 1,
      :env_curve => 2
    }
  end
end
```

To make our reimplementation a first-class Sonic Pi synth we will create a new Class for it, inheriting from `SonicPiSynth` and implement the following functions:

- `name`
- `arg_defaults`
- `introduced`
- `synth_name`
- `doc`

In addition we will need to add an entry into@@synth_info.

Chapter 5 - Recreating the beep synth

Planning our new synth

By inspecting the `Overtone` source in Chapter 2 we know what components we need to use to rebuild our own version of `beep`:

```
(without-namespace-in-synthdef
 (defsynth sonic-pi-beep [note 52
                          note_slide 0
                          note_slide_shape 1
                          note_slide_curve 0
                          amp 1
                          amp_slide 0
                          amp_slide_shape 1
                          amp_slide_curve 0
                          pan 0
                          pan_slide 0
                          pan_slide_shape 1
                          pan_slide_curve 0
                          attack 0
                          decay 0
                          sustain 0
                          release 1
                          attack_level 1
                          decay_level -1
                          sustain_level 1
                          env_curve 1
                          out_bus 0]
   (let [decay_level (select:kr (= -1 decay_level) [decay_level sustain_level])
         note        (varlag note note_slide note_slide_curve note_slide_shape)
         amp         (varlag amp amp_slide amp_slide_curve amp_slide_shape)
         amp-fudge   1
         pan         (varlag pan pan_slide pan_slide_curve pan_slide_shape)
         freq        (midicps note)
         snd         (sin-osc freq)
         env         (env-gen:kr (core/shaped-adsr attack decay sustain release attack_level decay_lev
     (out out_bus (pan2 (* amp-fudge env snd) pan amp))))
```

- `select` - the supercollider Select UGen
- `varlag` - the class VarLag
- `midicps` - the class midicps
- `sin-osc` - the SinOsc UGen
- `env-gen` - the EnvGen UGen

- `core/shaped-adsr` - something that invokes asdr somehow
- `out` - the Out UGen
- `pan` - one of the Pan UGen family
- `:action` - related to doneAction somehow

This was our first attempt in Chapter 1:

```
(SynthDef("myfirstsynth", {arg out = 0;
     var note, envelope;
     envelope = Line.kr(0.1, 0.0, 1.0, doneAction: 2);
     note = SinOsc.ar(440, 0, envelope);
     Out.ar(out, note);
}).writeDefFile("/Users/gordonguthrie/.synthdefs"))
```

So there are some things we can see that overlap with the `Overtone` description.

In both the source of sound is a Sine Oscillator the uGen `SinOsc`, and the sound is patched to the speakers using the `Out` uGen.

Our synth uses a `doneAction: 2` and `Overtone` has an `:action: FREE` to destroy the synth and free up its resource.

Our synth has use the `Line` uGen whereas the `Overtone` one uses `EnvGen` - the fact that our `Line` uGen is bound to a variable called `envelope` does give the game away a bit here - our synth plays a constant volume, but `beep` has an envelope with `attack`, `decay`, `sustain` and `release`.

There's some wierd stuff tho. The `Overtone` definition has all the default arguments from the function `arg_defaults` baked in too, along with an outbus set to `0` (which just means play the sound on the computer). But a couple are different. In `Overtone` the `env_curve` default is `1` and the `sustain_level` is - `whereas in` arg_defaults `both are set to` 1`.

This is just belt and braces tho, when the synth is written in SuperCollider it makes sense to bake in the default values as you develop it, you will be writing, running and debugging the code in SuperCollider and will want to know what it sounds like when its played in SonicPi. In theory you could then copy the defaults to your `arg_default` *function and delete them in your SuperCollider synth defintion. But why bother?*

So whats the plan?

- `mysecondsynth` will use the `note` parameter and the `midicps` uGen to turn SonicPi midi notes into frequenies and let our synth play plain notes without a bend
- `mythirdsynth` will use the `amp` and `pan` parameters with the `Pan` uGen to add volume and panning control
- `myfourthsynth` will switch out from a `Line` envelope to a proper one using `env-gen` and take the `attack`, `decay`, `sustain` and `release` parameters - it will also use the `env_curve` to alter the shape of the envelope
- `myfifthsynth` will use `varlag` to add the sliding behaviour to `note`, `pan` and `amp`

In between these synths we will look at elements of the SuperCollider language.

This manual will not teach you SuperCollider - just enough to get a Sonic Pi synth up from scratch.

Playing these synths

The synths in this manual are all written in literate SuperCollider - the code that generates the page you are reading is runnable in SuperCollider.

Chapter 5 - Recreating the beep synth

Improving our first synthesiser

There are somethings we will have to do to make it usable:

- it will need to turn off and not play endlessly. Sonic Pi expects notes to have a duration
- we will need to be able to pass the note in so it will play different notes
    - Sonic Pi expresses notes in terms of the midi note table, so `A4` not `440hz` - the synth will need to be able to accept Sonic Pi notes in and convert them into frequencies on the way out
- we will want to be able to make the note louder or quieter

Chapter 5 - Recreating the beep synth

UGens, channels, mixing and panning

Before we can start making our simple synthesizer fit for use with Sonic Pi we need to learn a little about SuperCollider, in particular:

- UGens - Unit Generators
- Channels - how SuperCollider outputs sound signals
- Mixing - how we merge sound signals
- Panning - how we place sounds in the stereo field

UGens

The code that we write for SuperCollider seems familiar, it seems like normal computer code, but its not.

Most computer programmes work on `values`, SuperCollider works on `signals`.

If we set a variable to be 3 for instance:

```
a=3;
```

We can ask what value does `a` have and the answer is three.

What about setting a variable to the result of evaluation a `SinOsc`?

```
b={SinOsc(440 0, 0.2)};
```

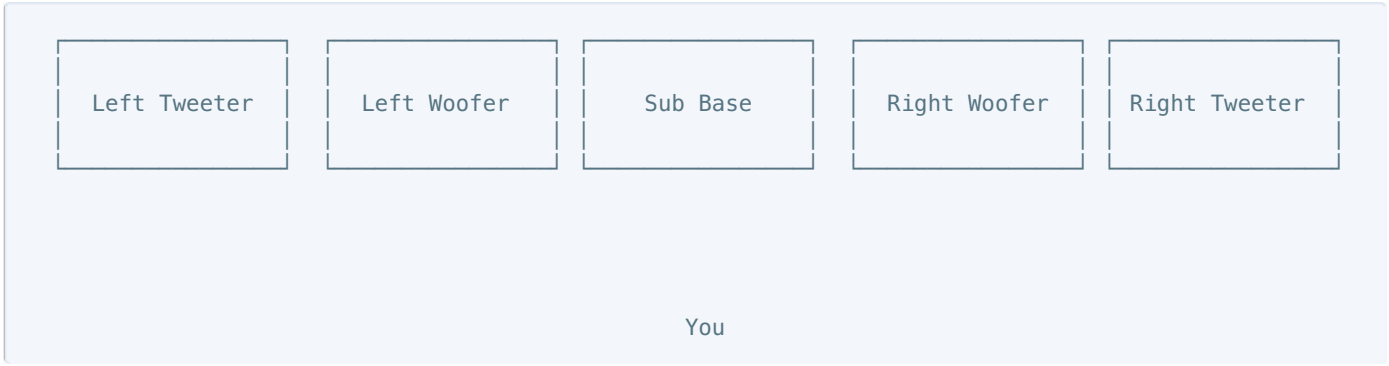
What value does `b` have? Well it fluctuates between -1 and 1 440 times a second. `b` is not a variable like in `javascript` or `c` which holds discrete values, it holds a signal.

`SinOsc` is a `UGen` - a unit generator - a bit of code that generates a signal.

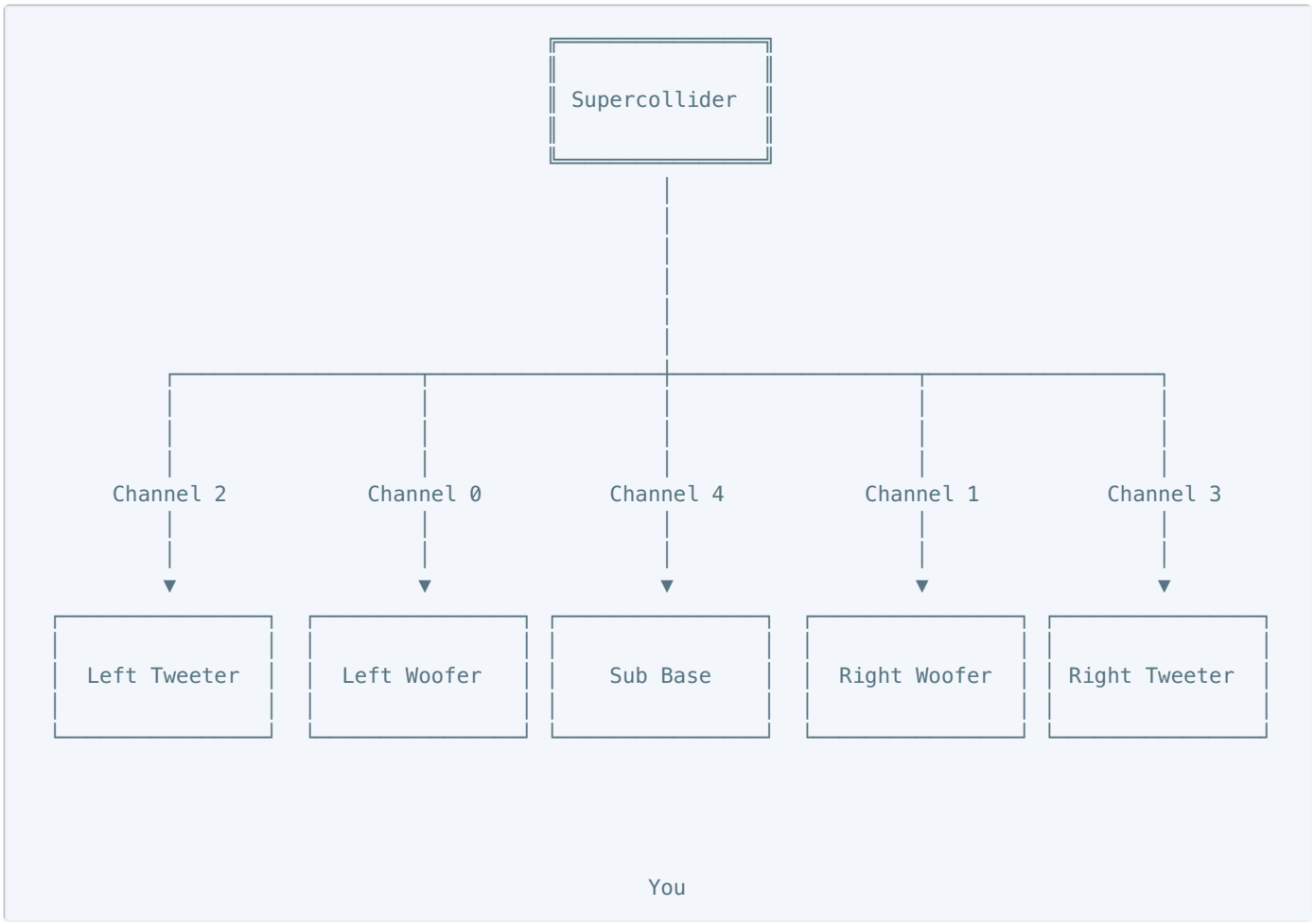
Channels

Channels are just the software equivalent of the cables that you wire your stereo up with.

Maybe you have a fancy hifi setup on your flat screen at home with 5 speakers:

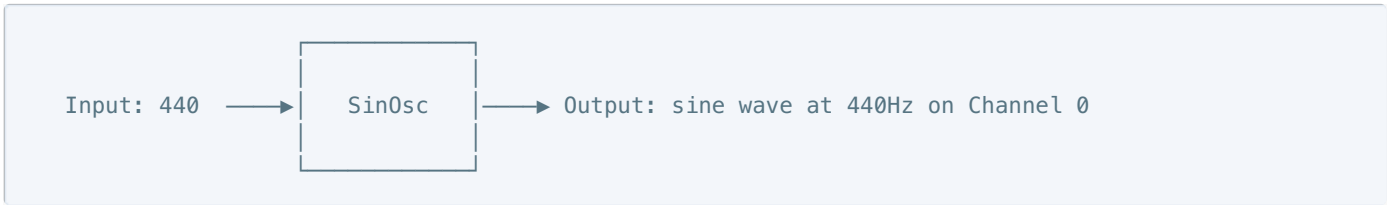


If you want SuperCollider to use this you would need to have it generate 5 Channels of output:

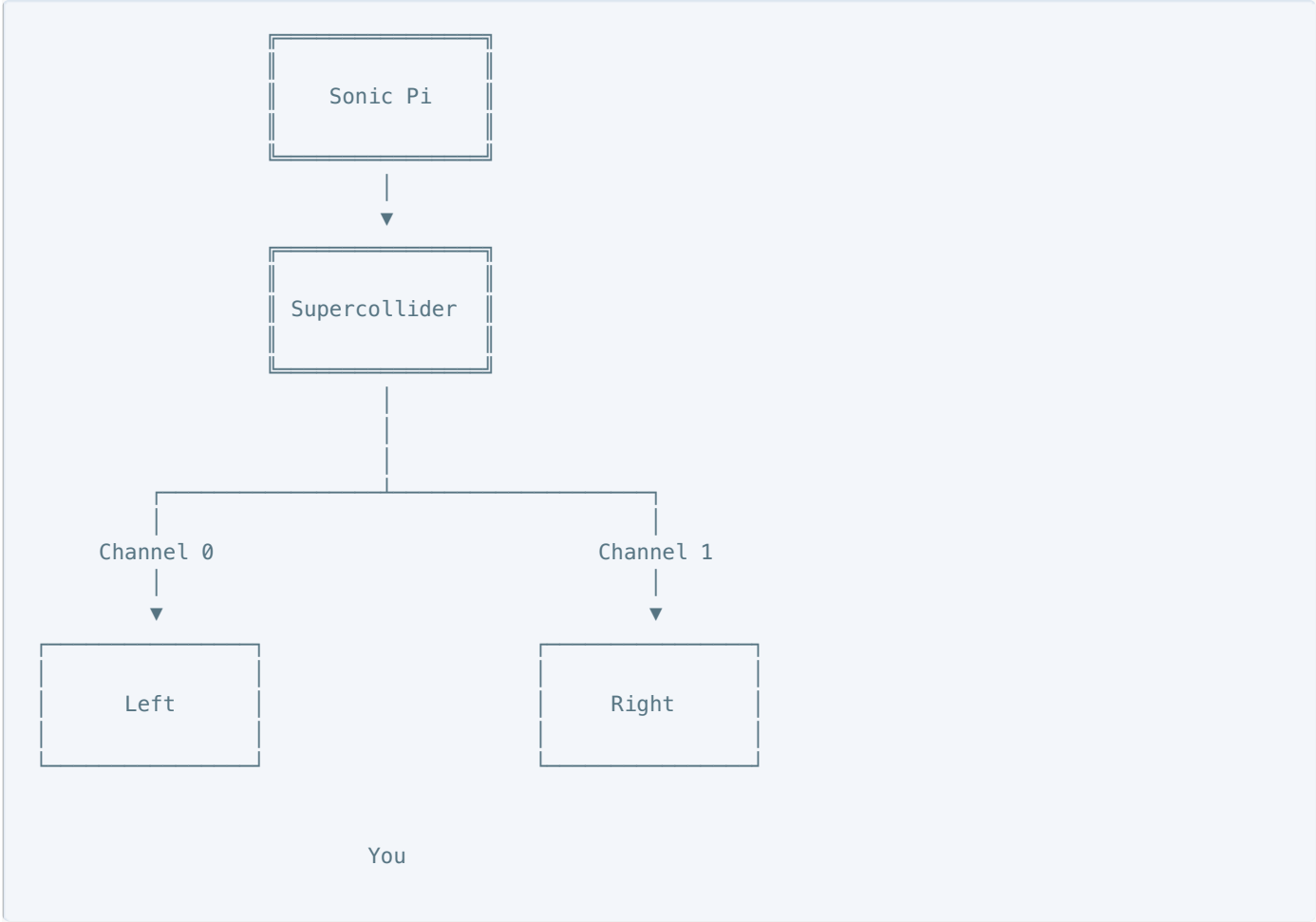


Obviously SuperCollider doesn't know your setup - you have to tell it, so it organises outputs using a simple approach and lets you handle how you want to wire up its outputs to a physical system.

If we look at our simple synth again, we can diagram its action:



So what happens when we call that from Sonic Pi? Sonic Pi assumes you are on a computer and you have 2 speakers only, a left and a right, either on your computer, on your desktop or in your headphones or earphones. So when we play our simple synth in Sonic Pi, we are making a noise in a setup like this:



This is why our basic synth makes a sound in the left hand speaker only.

We have seen that the `SinOsc` `UGen` takes a frequency parameter and outputs on a single channel. Well if instead of one value we accept an array of frequencies we will get an array of channels:

Because SuperCollider can't know your setup it just numbers the channels from 0. Its up to you to organise sending the signals to where they want to go.

So what happens if I have a 2 speaker setup, with Channel 0 on the left and Channel 1 on the right like in Sonic Pi? What happens to the noise on Channels 2 and up? They plop out of the back of your computer and make a little pile of discarded noise on the table.

The way we solve this problem is by `mixing`.

The Mix UGen

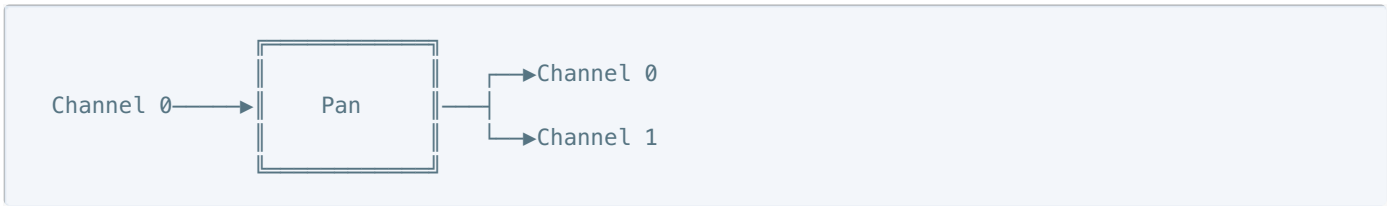
`Mix` is a `UGen` that takes an array of channels in, adds the signals together and outputs them on a single channel:

Great, so now if we get our `SinOsc` to play a chord, we can mix all the beeps together and actually hear them all, as a single thing. But its still in the left speaker only. Maybe we want it one the left, maybe on the right, maybe in the middle.

We can use a `UGen` called `Pan` to do this.

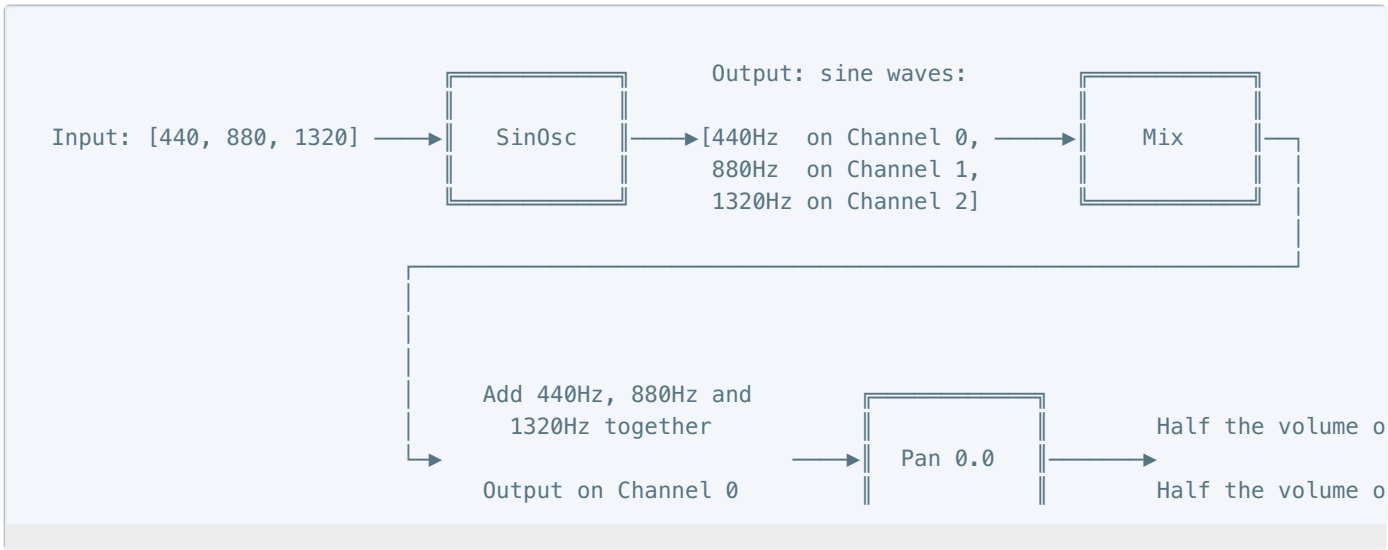
The Pan UGen

`Pan` is a `UGen` that takes a single channel input and splits it into two:



Put it all together

So the first thing we have to do to get our synth ready is put all these things together:



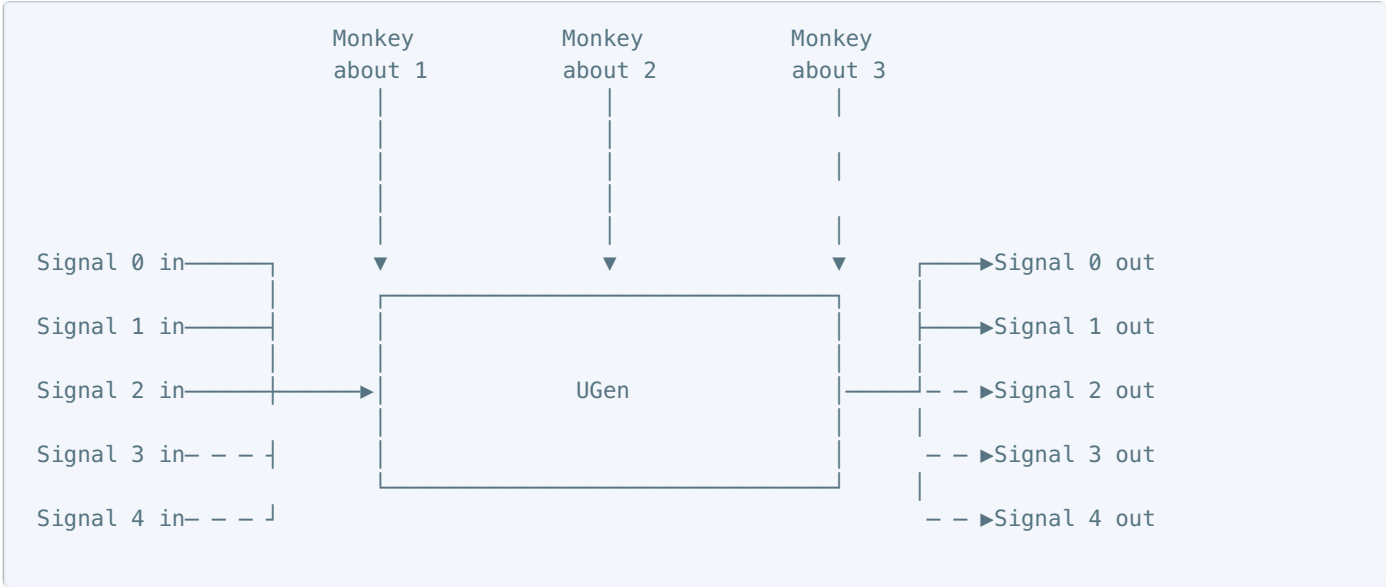
Chapter 5 - Recreating the beep synth

Why does this matter?

The code for synthesisers is pretty confusing. Good documentation is key to understanding how they are built and why. This chapter lays the ground work for understanding our documentation going forward.

Moar UGen Stuff

Generally we can diagram a `UGen` like this:



We take a set of audio signals in, we monkey with them based on parameters we pass in and we pass a set of signals out. Usually the signals are audio signals, but they can be controls.

Previously it seemed we had two different constructs:

```
a=3;
b={SinOsc(440, 0.1, 1)};
```

Where `a` is a constant and `b` is a signal. Reading SuperCollider code it often seems like these are different things, parameters and streams. In fact `a` is a constant signal, not a discrete constant. Anywhere you see numbers passed in, as volume parameters, as panning parameters to place sounds in the stereo field, you can also pass in variable streams.

Sonic Pi uses this to control synths:

```
s = play 60, release: 5
sleep 0.5
control s, note: 65
sleep 0.5
control s, note: 67
sleep 3
control s, note: 72
```

Here we have started a synth and passed in a constant stream of 60 as the note value. Then we change the value of the stream to 65, then 67 and then 72.

.kr and .ar

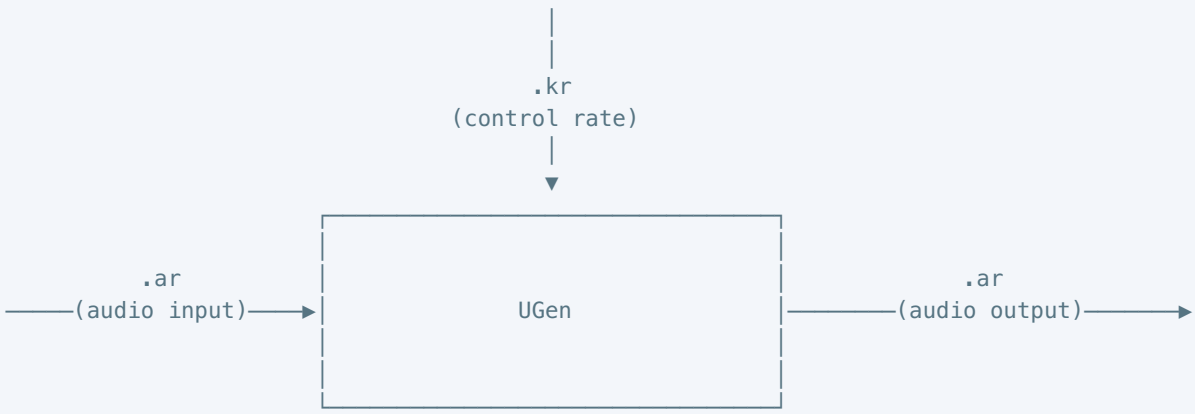
We saw earlier that `SinOsc` has two output modes:

- `.ar` or audio rate
- `.kr` or control rate

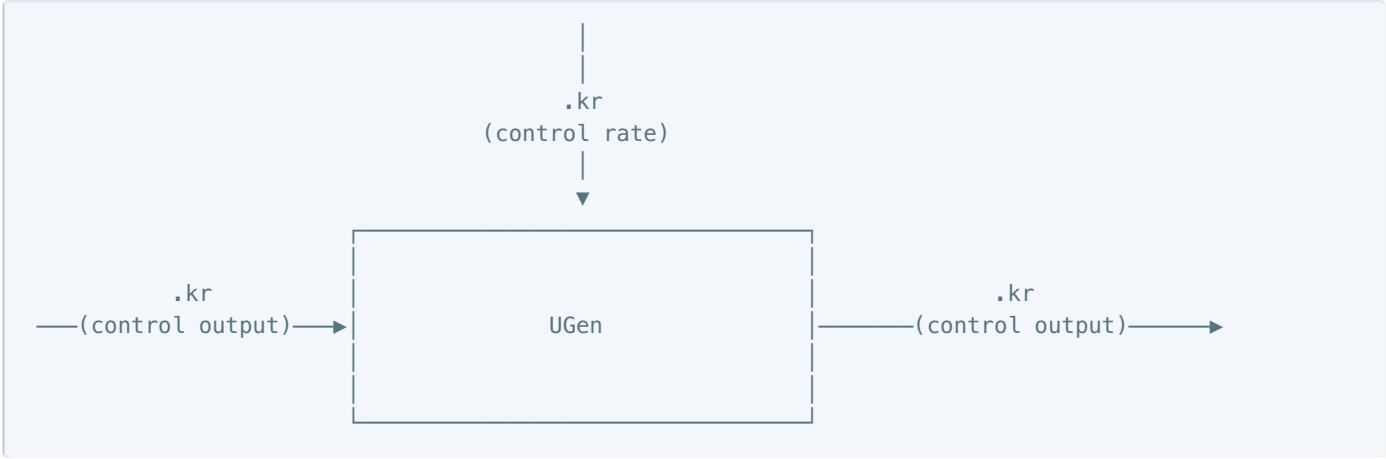
We know understand a bit better what that means. `audio rate` means the stream is fine grained and of high enough quality that it can be used to generate sounds (think representing an analog sound wave by a digital signal that goes up and down in tiny steps - the `.ar` is 44,100 steps per second.). `control rate` is a lower quality, less fine grained signal that is good enough for controls (think turning volumes up or down in big-ish steps - the `.kr` is 1/64th of the `.ar` or 690 steps per second).

`.ar` signals are 64 times as expensive to calculate than `.kr` ones. You can use `.ar` for everything (including controls) but best practice is not to.

With this understanding of `.kr` and `.ar` we see that there are two main `UGen` configurations - ones for manipulating sounds:



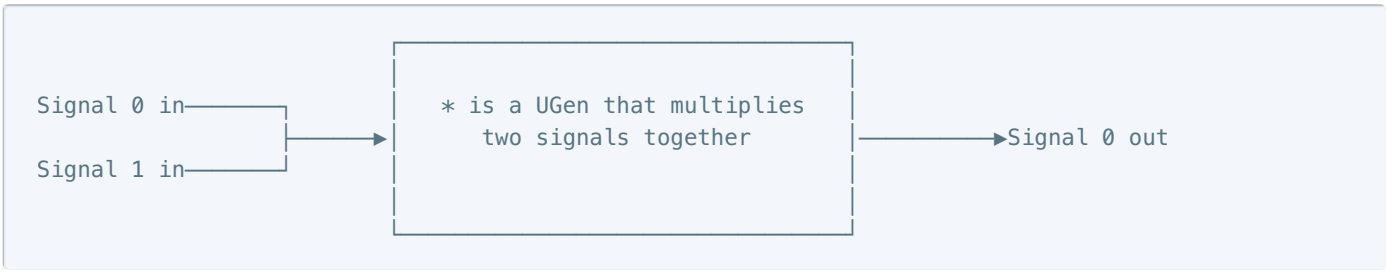
and ones for preparing control signals:

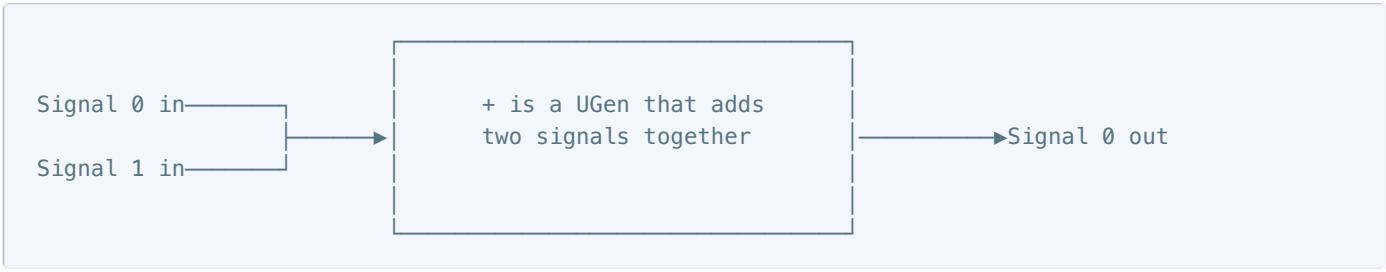


(Sometimes you might want to use an audio signal both in sound processing and in controlling another `UGen` so don't obsess about this.)

Hidden Ugens - + and *

There are a couple of hidden `Ugens` in Supercollider code - our old friends `*` and `+`.





Chapter 5 - Recreating the beep synth

Funky stuff part uno - Variables

In the last section we started by creating a named function and calling it:

```
f={SinOsc.ar(440, 0, 0.2)};

f.play;
```

The name we chose was `f`. Try and replace that with a better name like `my_synth`.

Oops it doesn't work.

SuperCollider has 3 types of variable name:

- one letter global variables like `f`
- long global variables that begin with a tilda - like `~my_synth`
- local variables that are multi-letter like `mysynth`

By convention the letter `s` is used to control the local server:

```
s=Server.local;

s.boot;

s.quit;
```

Chapter 6 - What's next

Integration

At the moment the loading of a synth and the presentation of it as built in are in two separate places.

The load command doens't know or check if the synth description, parameters and validations are loaded. The runtime looks and treats synths one way if they are built in and a different way if they are not recognised.

The strategic view is to merge these two things - to have a load process that loads both the synthesiser and the synth description, parameters and validation.

The load process should enable user-defined synths to be well-behaved (ie behave like and are subject to the same possible transforms as built-in ones) or badly-behaved.

The future of this manual

When there is a single integrated way of loading synths without recompiling the git repo that contains this manual will become a community library of different synths, that might be:

- well-behaved - have parameter sets aligned with the built in synths so that Sonic Pi is happy to do the standard transpositions and pitch shifts
- badly-behaved - have freaky parameters that might cause Sonic Pi to bork if the user mistypes or switches synths either live coding or in mucking about, or with odd ways of specifying notes that should not/cannot be subject to transpositions
- experimental

This manual is compiled with the Literate Code Reader escript which means that valid SuperCollider code will compile into a readable html page.

This means that simply by commiting a working SuperCollider synth for Sonic Pi it will be added to the book.

Custom effects (FXs)

In theory you should be able to write and load your own effects (FXs) in Sonic Pi. This is out of scope for this manual, but could be in scope for the 2nd edition.