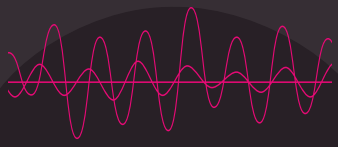
The MagPi ESSENTIALS

CODE MUSIC WITH Sonic Pi



LIVE CODE & CREATE AMAZING SOUNDS ON YOUR *Raspberry Pi*

Written by **Sam Aaron**

WELCOME TO CODE MUSIC WITH SONIC PI

Sonic Pi is a powerful new kind of musical instrument. Instead of strumming strings or whacking things with sticks, you write code… live. We call it live coding.

Sonic Pi comes pre-installed on your Raspberry Pi, but is also available to download for Mac and PC for free at **sonic-pi.net**.

I have written this book to complement Sonic Pi's built-in tutorial and to help you jump-start your live-coding career. It's packed full of fun, instantly rewarding examples to highlight many of the new things Sonic Pi will enable you to do. By the time you've finished it you'll be able to code up some phat bass, sparkling synth leads and start practicing for your first live-coding gig.

Remember, with live coding there are no mistakes - only opportunities!

Sam Aaron
Creator of Sonic Pi

FIND US ONLINE raspberrypi.org/magpi **GET IN TOUCH** magpi@raspberrypi.org

The MagPi

Available on the App Store

Get it on Google play

EDITORIAL
Managing Editor: **Russell Barnes**
russell@raspberrypi.org
Features Editor: **Rob Zwetsloot**
Sub Editors: **Laura Clay, Phil King, Lorna Lynch**

DESIGN
Critical Media: **criticalmedia.co.uk**
Head of Design: **Dougal Matthews**
Designers: **Lee Allen, Mike Kay**
Illustrator: **Sam Alder**

DISTRIBUTION
Seymour Distribution Ltd
2 East Poultry Ave,
London
EC1A 9PT | **+44 (0)207 429 4000**

SUBSCRIPTIONS
Select Publisher Services Ltd
PO Box 6337, Bournemouth
BH1 9EH | **+44 (0)1202 586 848**
magpi.cc/Subs1

In print, this product is made using paper sourced from sustainable forests and the printer operates an environmental management system which has been assessed as conforming to ISO 14001.

The MagPi ESSENTIALS

CONTENTS

[SAM AARON]

Sam Aaron is a live coder exploring the intersection between code, art, and education. He sees programming as performance, and builds systems to lower the barrier for people to have a creative experience with code.

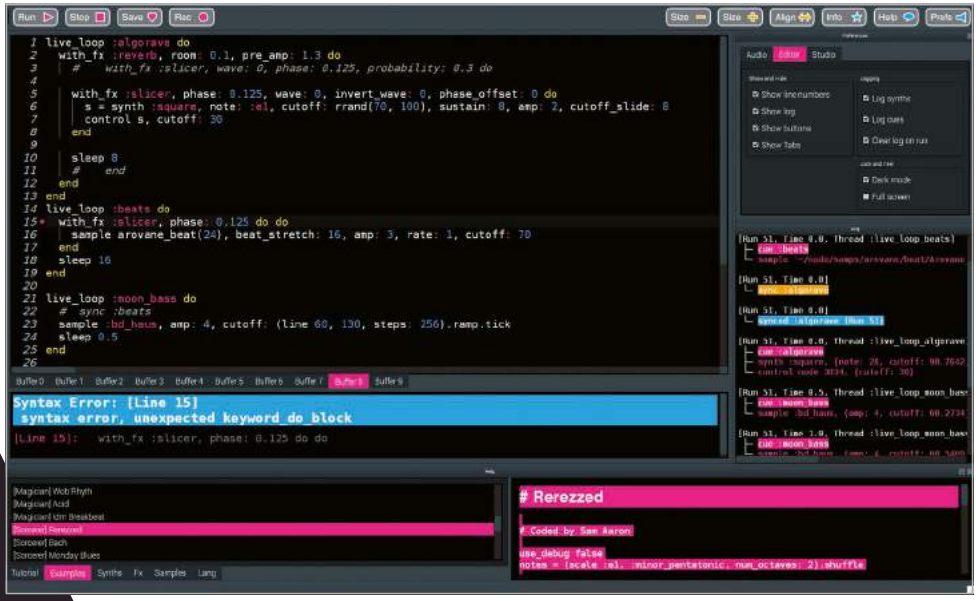
By day, Sam is a Research Associate at the University of Cambridge Computer Laboratory; by night, he makes people dance by live coding in clubs and events all over the world, using Sonic Pi running on a Raspberry Pi.

The MagPi ESSENTIALS

[CHAPTER ONE]
LIVE CODING

Digital musician and Cambridge Computer Lab researcher **Sam Aaron** starts off his Sonic Pi tutorial series by introducing the art of live coding

Below The new Dark theme is lovely!



The laser beams sliced through the wafts of smoke as the subwoofer pumped bass deep into the bodies of the crowd. The atmosphere was rife with a heady mix of synths and dancing. However, something wasn't quite right in this nightclub. Projected in bright colours above the DJ booth was futuristic text, moving, dancing, flashing. This wasn't fancy visuals; it was merely a projection of Sonic Pi running on a Raspberry Pi. The occupant of the DJ booth wasn't spinning discs; she was writing, editing, and evaluating code. Live. This is Live Coding.

This may sound like a far-fetched story from the future, but coding music like this is a growing trend and is often described as live coding (toplap.org). One of the recent directions this approach to music-making has taken is the Algorave (algorave.com) – events where artists like myself code music for people to dance to. However, you don't need to be in a nightclub to live-code; with Sonic Pi v2.9+, you can do it anywhere you can take your Raspberry Pi and a pair of headphones or some speakers. Once you reach the end of this article, you'll be programming your own beats and modifying them live. Where you go afterwards will only be constrained by your imagination.

Live loop

The key to live coding with Sonic Pi is mastering the **live_loop**. Let's look at one:

```
live_loop :beats do
  sample :bd_haus
  sleep 0.5
end
```

There are four core ingredients to a **live_loop**. The first is its name. Our **live_loop** above is called **:beats**. You're free to call yours anything you want. Go crazy. Be creative. I often use names that communicate something about the music they're making to the audience. The second ingredient is the **do** word, which marks where the **live_loop** starts. The third is the **end** word, which marks where the **live_loop** finishes. Finally, there is the body of the **live_loop**, which describes what the loop is going to repeat – that's the bit between the **do** and **end**. In this case, we're repeatedly playing a bass drum sample and waiting for half a beat. This produces a nice regular bass beat. Go ahead: copy it into an empty Sonic Pi buffer and hit **Run**. Boom, boom, boom!

Redefining on-the-fly

OK, so what's so special about the **live_loop**? So far it just seems like a glorified 'loop'! Well, the beauty of **live_loop**s is that you can redefine them on-the-fly. This means that while they're still running, you can change what they do. This is the secret to live coding with Sonic Pi. Let's have a play:

```
live_loop :choral_drone do
  sample :ambi_choir, rate: 0.4
  sleep 1
end
```

Now hit the **Run** button or press **ALT+R**. You're now listening to some gorgeous choir sounds. Now, while it's still playing, change the rate from **0.4** to **0.38**. Hit **Run** again. Whoa! Did you hear the choir change note? Change it back up to **0.4** to return to how it was.

Below Sam Aaron headlining an Algorave at the Glasgow School of Art

Now, drop it to **0.2**, down to **0.19**, and then back up to **0.4**. See how changing just one parameter on-the-fly can give you real control of the music? Now play around with the rate yourself – choose your own values. Try negative numbers, really small numbers, and large numbers. Have fun!

Sleeping is important

One of the most important lessons about **live_loop**s is that they need rest. Consider the following **live_loop**:

```
live_loop :infinite_impossibilities do
  sample :ambi_choir
end
```

If you try running this code, you'll immediately see Sonic Pi complaining that the **live_loop** did not sleep. This is a safety system kicking in! Take a moment to think about what this code is asking the computer to do. That's right, it's asking the computer to play an infinite amount of choir samples in zero time. Without the safety system, the poor computer will try to do this and crash and burn in the process. So remember- your **live_loop**s must contain a **sleep**.

Combining sounds

Music is full of things happening at the same time. Drums at the same time as bass at the same time as vocals at the same time as guitars… In computing we call this concurrency, and Sonic Pi provides us with an

amazingly simple way of playing things at the same time. Simply use more than one **live_loop**!

```
live_loop :beats do
  sample :bd_tek
  with_fx :echo, phase: 0.125, mix: 0.4 do
    sample  :drum_cymbal_soft, sustain: 0, release: 0.1
    sleep 0.5
  end
end

live_loop :bass do
  use_synth :tb303
  synth :tb303, note: :e1, release: 4, cutoff: 120,
    cutoff_attack: 1
  sleep 4
end
```

Here, we have two **live_loop**s: one looping quickly, making beats; another looping slowly, making a crazy bass sound. One of the interesting things about using multiple **live_loop**s is that they each manage their own time. This means it's really easy to create interesting polyrhythmical structures and even play with phasing, Steve Reich style. Check this out:

Steve Reich's piano phase

```
notes = (ring :E4, :Fs4, :B4, :Cs5, :D5, :Fs4,
         :E4, :Cs5, :B4, :Fs4, :D5, :Cs5)

live_loop :slow do
  play notes.tick, release: 0.1
  sleep 0.3
end

live_loop :faster do
  play notes.tick, release: 0.1
  sleep 0.295
end
```

Bringing it all together

In each of these tutorials, we'll end with a final example in the form of a new piece of music which draws from all of the ideas introduced. Read this code and see if you can imagine what it's doing. Then, copy it into a fresh Sonic Pi buffer and hit **Run** and actually hear what it sounds like. Finally, change one of the numbers, or comment and uncomment things out. See if you can use this as a starting point for a new performance – and most of all, have fun!

```
with_fx :reverb, room: 1 do
  live_loop :time do
    synth :prophet, release: 8, note: :e1, cutoff: 90, amp: 3
    sleep 8
  end
end

live_loop :machine do
  sample :loop_garzul, rate: 0.5, finish: 0.25
  sample :loop_industrial, beat_stretch: 4, amp: 1
  sleep 4
end

live_loop :kik do
  sample :bd_haus, amp: 2
  sleep 0.5
end

with_fx :echo do
  live_loop :vortex do
    # use_random_seed 800
    notes = (scale :e3, :minor_pentatonic, num_octaves: 3)
    16.times do
      play notes.choose, release: 0.1, amp: 1.5
      sleep 0.125
    end
  end
end
```

[CHAPTER TWO]

CODED BEATS

By playing sample loops, it's possible to recreate some of the most influential drum breaks in music history

Below The Akai MPC 2000, a classic early sampler

One of the most exciting and revolutionary technical developments in modern music was the invention of computer-based samplers in the late 1970s. These electronic boxes of tricks allowed you to record any sound into them and then manipulate and play back those sounds in many interesting ways. For example, you could take an old record, find a drum solo (or break), record it into your sampler, and then play it back on repeat at half-speed to provide the foundation for your latest beats. This is how early hip-hop music was born, and today it's almost impossible to find electronic music that doesn't incorporate samples of some kind. Using samples is a really great way of easily introducing new and interesting elements into your live-coded performances.

So where can you get a sampler? Well, you already have one: it's your Raspberry Pi! The built-in live-coding app Sonic Pi has an extremely powerful sampler built into its core. Let's play with it!

The Amen Break

One classic and immediately recognisable drum break sample is called the Amen Break. It was first performed in 1969 in the song *Amen Brother* by The Winstons, as part of a drum break. However, it was when it was discovered and sampled by early hip-hop musicians in the 1980s that it started being heavily used in a wide variety of other musical styles such as drum and bass, breakbeat, hardcore techno, and breakcore.

I'm sure you're excited to hear that it's also built right into Sonic Pi. Clear up a buffer and throw in the following code:

```
sample :loop_amen
```

Hit **Run** and boom! You're listening to one of the most influential drum breaks in the history of dance music. However, this sample wasn't famous for being played as a one-shot: it was built for being looped.

Beat stretching

Let's loop the Amen Break by using our old friend the **`live_loop`**, introduced in chapter 1:

```
live_loop :amen_break do
  sample :loop_amen
  sleep 2
end
```

OK, so it is looping, but there's an annoying pause every time round. That is because we asked it to sleep for **`2`** beats; however, with the default BPM of **`60`**, the **`:loop_amen`** sample only lasts for **`1.753`** beats. We therefore have a silence of **`2 - 1.753 = 0.247`** beats. Even though it's short, it's still noticeable.To fix this issue, we can use the **`beat_stretch:`** opt to ask Sonic Pi to stretch (or shrink) the sample to match the specified number of beats.

Sonic Pi's `sample` and `synth` functions give you a lot of control via optional parameters such as `amp:`, `cutoff:`, and `release:`. However, the term 'optional parameter' is a real mouthful, so we just call them opts to keep things nice and simple.

```
live_loop :amen_break do
  sample :loop_amen, beat_stretch: 2
  sleep 2
end
```

Now we're dancing! Although, perhaps we want speed it up or slow it down to suit the mood.

Playing with time

OK, so what if we want to change styles to old-school hip-hop or breakcore? One simple way of doing this is to play with time or, in other words, to mess with the tempo. This is super-easy in Sonic Pi: just throw a **use_bpm** into your live loop…

```
live_loop :amen_break do
  use_bpm 30
  sample :loop_amen, beat_stretch: 2
  sleep 2
end
```

Whilst you're rapping over those slow beats, notice that we're still sleeping for **2** and our BPM is **30**, yet everything is in time. This is because the **beat_stretch** opt uses the current BPM to make sure everything just works.

Now, here's the fun part. Whilst the loop is still live, change the **30** in the **use_bpm 30** line to **50**. Whoa, everything just got faster yet **kept in time**! Try going faster: up to **80**…to **120**…now go crazy and punch in **200**!

Filtering

Now we can live-loop samples, let's look at some of the most fun opts provided by the **sample** synth. First up is **cutoff:**, which controls the cutoff filter of the sampler. This is disabled by default, but you can easily turn it on:

```
live_loop :amen_break do
  use_bpm 50
  sample :loop_amen, beat_stretch: 2, cutoff: 70
  sleep 2

end
```

Go ahead and change the **cutoff:** opt. For example, increase it to **100**, hit **Run**, and wait for the loop to cycle round to hear the change in the sound. Notice that low values like **50** sound mellow and bassy, and high values like **100** and **120** are more full-sounding and raspy. This is because the **cutoff:** opt will chop out the high-frequency parts of the sound, just like a lawnmower chops off the top of the grass. The **cutoff:** opt is like the length setting, determining how much grass is left over.

Slicing

Another great tool to play with is the slicer FX. This will chop (slice) the sound up. Wrap the **sample** line with the FX code like this:

```
live_loop :amen_break do
  use_bpm 50
  with_fx :slicer, phase: 0.25, wave: 0, mix: 1 do
    sample :loop_amen, beat_stretch: 2, cutoff: 100
  end

  sleep 2

end
```

Notice how the sound bounces up and down a little more. (You can hear the original sound without the FX by changing the **mix:** opt to **0**). Now, try playing around with the **phase:** opt. This is the rate (in beats) of the slicing effect. A smaller value like **0.125** will slicer faster and larger values like **0.5** will slice more slowly. Notice that successively halving or doubling the **phase:** opt value tends to always sound good. Finally, change the **wave:** opt to one of **0**, **1**, or **2** and hear how it changes the sound. These are the various wave shapes. **0** is a

saw wave, (hard in, fade out), **1** is a square wave (hard in, hard out), and **2** is a triangle wave (fade in, fade out).

Bringing it all together

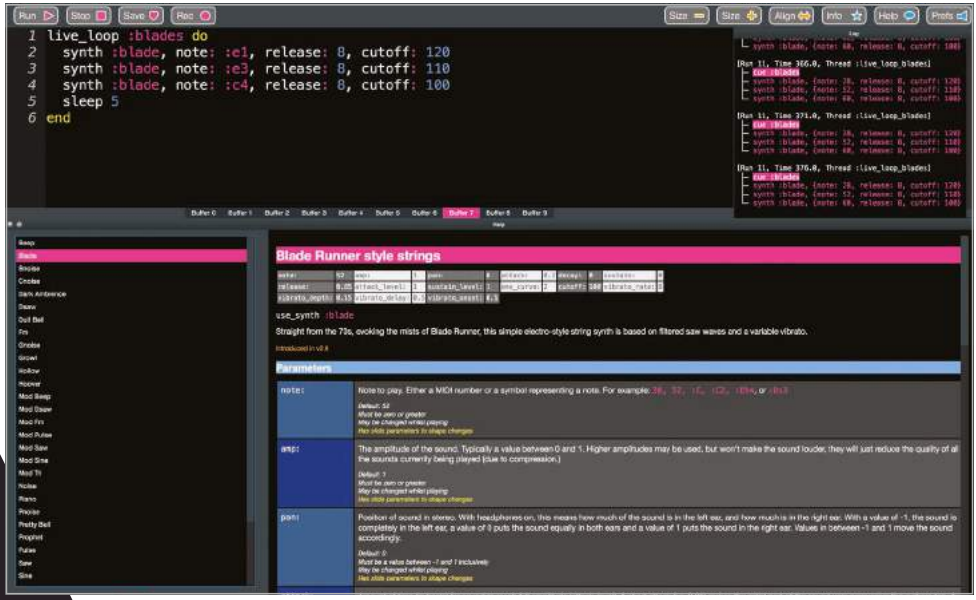
Finally, let's revisit the early Bristol drum and bass scene. Don't worry too much about what all this code means; just type it in, hit **Run**, then start live-coding it by changing opt numbers and see where you can take it. Please do share what you create!

```
use_bpm 90
use_debug false
live_loop :amen_break do
  p = [0.125, 0.25, 0.5].choose
  with_fx :slicer, phase: p, wave: 0, mix: rrand(0.7, 1),
    reps: 4 do
    r = [1, 1, 1, -1].choose
    sample :loop_amen, beat_stretch: 2, rate: r , amp: 2
    sleep 2
  end
end

live_loop :bass_drum do
  sample :bd_haus, cutoff: 70, amp: 1.5
  sleep 0.5
end

live_loop :landing do
  bass_line = (knit :e1, 3, [:c1, :c2].choose, 1)
  with_fx :slicer, phase: [0.25, 0.5].choose,
    invert_wave: 1, wave: 0 do
    s = synth :square, note: bass_line.tick, sustain: 4,
      cutoff: 60
    control s, cutoff_slide: 4, cutoff: 120
  end
  sleep 4

end
```

The MagPi ESSENTIALS

[CHAPTER THREE]
SYNTH RIFFS

Here we take a look at synth riffs, coding their timbre, melody, rhythm



Whether it's the haunting drift of rumbling oscillators or the detuned punch of saw waves piercing through the mix, the lead synth plays an essential role on any electronic track. In chapter 2, we covered how to code our beats. In this tutorial we'll cover how to code up the three core components of a synth riff – the timbre, melody, and rhythm.

OK, so power up your Raspberry Pi, crack open Sonic Pi v2.9+, and let's make some noise!

> We can control the timbre in Sonic Pi in two ways...

Timbral possibilities

An essential part of any synth riff is changing and playing with the timbre of the sounds. We can control the timbre in Sonic Pi in two ways – choosing different synths for a dramatic change, and setting

the various synth opts for more subtle modifications. We can also use FX, but that's for another tutorial…

Let's create a simple live loop where we continually change the current synth:

```
live_loop :timbre do
  use_synth (ring :tb303, :blade,
             :prophet, :saw, :beep, :tri).tick
  play :e2, attack: 0, release: 0.5, cutoff: 100
  sleep 0.5
end
```

Take a look at the code. We're simply ticking through a ring of synth names (this will cycle through each of these in turn, repeating the list over and over). We pass this synth name to the **use_synth** fn (function), which will change the **live_loop**'s current synth. We also play note **:e2** (E at the second octave), with a release time of **0.5** beats (half a second at the default BPM of **60**) and with the **cutoff:** opt set to **100**.

Hear how the different synths have very different sounds, even though they're all playing the same note. Now experiment and have a play. Change the release time to bigger and smaller values. For example, change the **attack:** and **release:** opts to see how different fade in/out times have a huge impact on the sound. Finally, change the **cutoff:** opt to see how different cutoff values also massively influence the timbre (values between **60** and **130** are good). See how many different sounds you can create by just changing a few values. Once you've mastered that, just head to the Synths tab in the Help system for a full list of all the synths and all the available opts each individual synth supports, to see just how much power you have under your coding fingertips.

Timbre is just a fancy word describing the sound of a sound. If you play a the same note with different instruments such as a violin, guitar, and piano, the pitch (how high or low it sounds) would be the same, but the sound quality would be different. That sound quality – the thing which allows you to tell the difference between a piano and a guitar – is the timbre.

Melodic composition

Another important aspect to our lead synth is the choice of notes we want to play. If you already have a good idea, then you can simply create a ring with your notes in and tick through them:

```
live_loop :riff do
  use_synth :prophet
  riff = (ring :e3, :e3, :r, :g3, :r, :r, :r, :a3)
  play riff.tick, release: 0.5, cutoff: 80
  sleep 0.25
end
```

In this example, we have defined our melody with a ring, which includes both notes such as **:e3** and rests, represented by **:r**. We are then using **.tick** to cycle through each note to give us a repeating riff.

Auto melody

It's not always easy to come up with a nice riff from scratch. Instead, it's often easier to ask Sonic Pi for a selection of random riffs and to choose the one you like the best. To do that, we need to combine three things: rings, randomisation, and random seeds. Let's look at an example:

```
live_loop :random_riff do
  use_synth :dsaw
  use_random_seed 3
  notes = (scale :e3, :minor_pentatonic).shuffle
  play notes.tick, release: 0.25, cutoff: 80
  sleep 0.25
end
```

There's a few things going on – let's look at them in turn. First, we specify that we're using random seed 3. What does this mean? Well, The useful thing is that when we set the seed, we can predict what the next random value is going to be – it's the same as it was last time we set the seed to **3** (see 'Pseudo Randomisation' box below). Another useful thing to know is that shuffling a ring of notes works in the same way. In the example above, we're essentially asking for the 'third

shuffle' in the standard list of shuffles – which is also the same every time, as we're always setting the random seed to the same value right before the shuffle. Finally, we're just ticking through our shuffled notes to play the riff.

Now, here's where the fun starts. If we change the random seed value to another number, say **3000**, we get an entirely different shuffling of the notes. So now it's very easy to explore new melodies. Simply choose the list of notes to shuffle (scales are a great starting point), then the seed to shuffle with. If you don't like the melody, just change one of those two things and try again. Repeat until you like what you hear!

Pseudo randomisation

Sonic Pi's randomisation is not actually random – it's what's called pseudorandom. Imagine if you were to roll a dice 100 times and write down the result of each roll onto a piece of paper. Sonic Pi has the equivalent of this list of results, which it uses when you ask for a random value. Instead of rolling an actual dice, it just picks the next value from the list. Setting the random seed is just jumping to a specific point in that list.

Finding your rhythm

Another important aspect to our riff is the rhythm – when to play a note and when not to. As we saw above, we can use **:r** in our rings to insert rests. Another very powerful way is to use spreads. Today, however, we'll use randomisation to help us find our rhythm. Instead of playing every note, we can use a conditional to play a note with a given probability. Let's take a look:

```
live_loop :random_riff do
  use_synth :dsaw
  use_random_seed 30
  notes = (scale :e3, :minor_pentatonic).shuffle
  16.times do
    play notes.tick, release: 0.2, cutoff: 90 if one_in(2)
    sleep 0.125
  end
end
```

A really useful fn to know is **one_in**, which will give us a **true** or **false** value with the specified probability. Here, we're using a value of **2**, so on average, one time every two calls to **one_in** it will return **true**. Using higher values will make it return **false** more often, introducing more space into the riff.

Notice that we've added some iteration in here with **16.times**. This is because we only want to reset our random seed value every 16 notes, so our rhythm repeats every 16 times. This doesn't affect the shuffling, as that is still done immediately after the seed is set. We can use the iteration size to alter the length of the riff. Try changing the **16** to **8** or even **4** or **3** and see how it affects the rhythm of the riff. Finally, experiment with different seed values!

Bringing it all together

OK, so let's combine everything we've learned together into one final example. See you in the next chapter!

```
use_debug false
live_loop :random_riff do
  #  uncomment and hit Run to bring in:
  #  synth :blade, note: :e4, release: 4, cutoff: 100, amp: 1.5
  use_synth :dsaw
  use_random_seed 30030
  notes = (scale :e3, :minor_pentatonic, num_octaves: 2).shuffle.take(8)
  8.times do
    play notes.tick, release: rand(0.5),
      cutoff: rrand(60, 130) if one_in(2)
    sleep 0.125
  end
end
live_loop :drums do
  use_random_seed 500
  16.times do
    sample :bd_haus, rate: 2, cutoff: 110 if rand < 0.35
    sleep 0.125
  end
end
live_loop :bd do
  sample :bd_haus, cutoff: 100, amp: 3
  sleep 0.5
end
```

The MagPi ESSENTIALS

[CHAPTER FOUR]
ACID BASS

Yes, you can turn your Raspberry Pi into a TB-303 for the infamous acid house bass sound

It's impossible to look through the history of electronic dance music without seeing the enormous impact of the tiny Roland TB-303 synthesiser. It's the secret sauce behind the original acid bass sound. Those classic squealing and squelching TB-303 bass riffs can be heard from the early Chicago House scene through to more recent electronic artists such as Plastikman, Squarepusher, and Aphex Twin.

Interestingly, Roland never intended for the TB-303 to be used in dance music. It was originally created as a practice aid for guitarists. Roland imagined that people would program the TB-303 to play basslines to jam along to. Unfortunately, there were a number of problems: it was a little fiddly to program, didn't sound particularly good as a bass-guitar replacement, and was pretty expensive to buy. Opting to cut its losses, Roland stopped making the TB-303 after 10,000 units were sold. After a number of years sitting on guitarists' shelves, many ended in the windows of second-hand shops. These discarded TB-303s were waiting to be discovered by a new generation, which started experimenting and using them to create new crazy sounds. Acid house was born.

While getting your hands on an original TB-303 isn't so easy, you'll be pleased to know that you can turn your Raspberry Pi into one using the power of Sonic Pi. Just put this code into an empty buffer and hit **Run**:

```
use_synth :tb303
play :e1
```

Instant acid bass! Let's play around…

Squelch that bass

First, let's build a live arpeggiator to make things fun. In chapter 3, we looked at how riffs can just be a ring of notes that we tick through one after another, repeating when we get to the end. Let's create a live loop that does exactly that:

```
use_synth :tb303
live_loop :squelch do
  n = (ring :e1, :e2, :e3).tick
  play n, release: 0.125, cutoff: 100, res: 0.8, wave: 0
  sleep 0.125
end
```

Take a look at each line…

1. On the first line, we set the default synth to be **tb303** with the **use_synth** function.

2. On line two, we create a live loop called **:squelch**, which will just loop round and round.

3. Line three is where we create our riff – a ring of notes (E in octaves 1, 2, and 3), which we simply tick through with **.tick**. We define **n** to represent the current note in the riff. The equals sign just means to assign the value on the right to the name on the left. This will be different every time round the loop. The first time round, **n** will be set to **:e1**. The second time round, it will be **:e2**, followed by **:e3**, and then back to **:e1**, cycling round forever.

4. Line four is where we actually trigger our **:tb303** synth. We're passing a few interesting opts here: **release:**, **cutoff:**, **res:**, and **wave:**, which we will discuss below.

5. Line five is our **sleep** – we're asking the live loop to loop round every **0.125** seconds, which works out at eight times a second at the default BPM of **60**.

6. Line six is the **end** to the live loop. This just tells Sonic Pi where the end of the live loop is.

Whilst you're still figuring out what's going on, type in the code above and hit the **Run** button. You should hear the **:tb303** kick into action. Now, this is where the action is: let's start live coding.

Whilst the loop is still live, change the **cutoff:** opt to **110**. Now hit the **Run** button again. You should hear the sound become a little harsher and more squelchy. Dial in **120** and hit **Run**. Now **130**. Listen how higher cutoff values make it sound more piercing and intense. Finally, drop it down to **80** when you feel like a rest. Then repeat as many times as you want. Don't worry, I'll still be here…

Another opt worth playing with is **res:**. This controls the level of resonance of the filter. A high resonance is characteristic of acid bass

sounds. We currently have our **res:** set to **0.8**. Try cranking it up to **0.85**, then **0.9**, and finally **0.95**. You might find that a cutoff such as **110** or higher will make the differences easier to hear. Now, go crazy and dial in **0.999** for some insane sounds. At a **res:** this high, you're hearing the cutoff filter resonate so much that it starts to make sounds of its own!

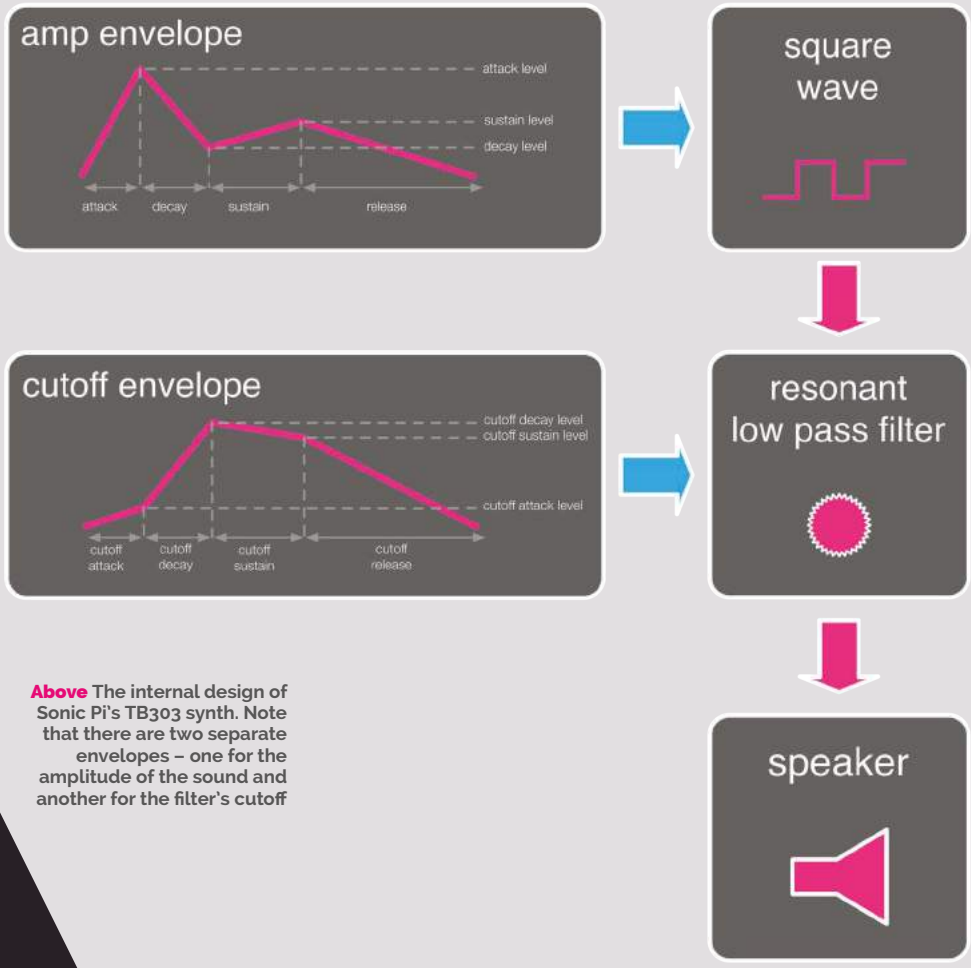
Finally, for a big impact on the timbre, try changing the **wave:** opt to **1**. This is the choice of source oscillator. The default is **0**, which is a sawtooth wave. **1** is a pulse wave and **2** is a triangle wave.

Of course, try different riffs by changing the notes in the ring or even picking notes from scales or chords. Have fun with your first acid bass synth.

Deconstructing the TB-303

The design of the original TB-303 is actually pretty simple. There are only four core parts. First is the oscillator wave – the raw ingredients of the sound. For instance, this could be a square wave. Next, there's the oscillator's amplitude envelope, which controls the amp of the square wave through time. These are accessed in Sonic Pi by the **attack:**, **decay:**, **sustain:**, and **release:** opts, along with their level counterparts. For more information, read Section 2.4, 'Duration with Envelopes', of the built-in tutorial. We then pass our enveloped square wave through a resonant low-pass filter. This chops off the higher frequencies, as well as having that nice resonance effect. Now this is where the fun starts. The cutoff value of this filter is also controlled by its own envelope! This means we have amazing control over the timbre of the sound by playing with both of these envelopes. Let's take a look:

```
use_synth :tb303

with_fx :reverb, room: 1 do
  live_loop :space_scanner do
    play :e1, cutoff: 100, release: 7, attack: 1,
      cutoff_attack: 4, cutoff_release: 4
    sleep 8
  end
end
```



Above The internal design of Sonic Pi's TB303 synth. Note that there are two separate envelopes – one for the amplitude of the sound and another for the filter's cutoff

For each standard envelope opt, there's a **`cutoff_`** equivalent opt in the **`:tb303`** synth. So, to change the cutoff attack time, we can use the **`cutoff_attack:`** opt. Copy the code above into an empty buffer and hit **Run**. You'll hear a crazy sound warble in and out. Now start to play around with it. Try changing the **`cutoff_attack:`** time to **`1`** and then **`0.5`**. Now try **`8`**.

Notice that I've passed everything through a **:reverb** FX for extra atmosphere – try other FX to see what works!

Bringing it all together

Finally, here's a piece I composed using the ideas in this tutorial. Copy it into an empty buffer, listen for a while, and then start live-coding your own changes. See what crazy sounds you can make with it! See you next time…

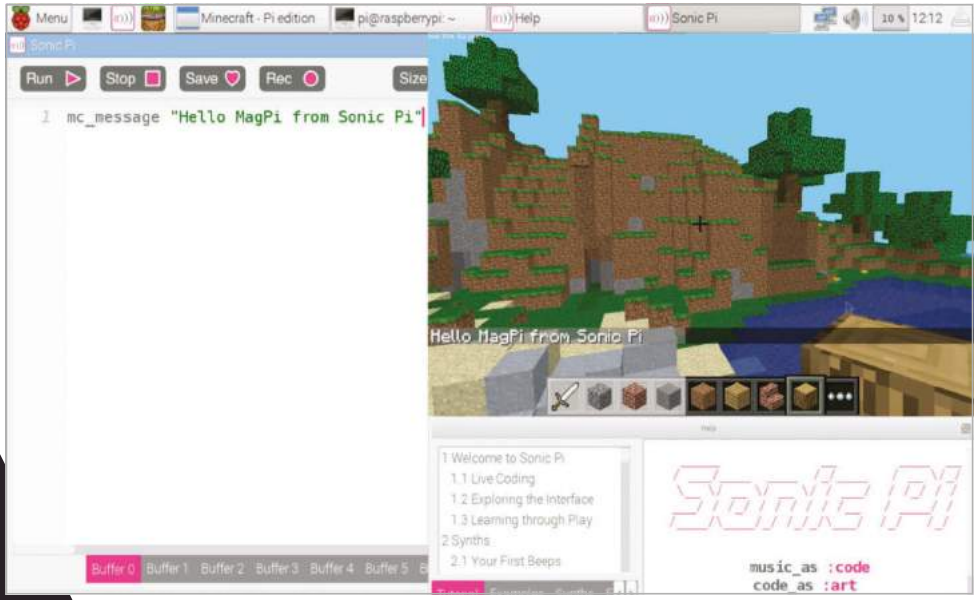
```
use_synth :tb303
use_debug false

with_fx :reverb, room: 0.8 do
  live_loop :space_scanner do
    with_fx :slicer, phase: 0.25, amp: 1.5 do
      co = (line 70, 130, steps: 8).tick
      play :e1, cutoff: co, release: 7, attack: 1,
        cutoff_attack: 4, cutoff_release: 4
      sleep 8
    end
  end

  live_loop :squelch do
    use_random_seed 3000
    16.times do
      n = (ring :e1, :e2, :e3).tick
      play n, release: 0.125, cutoff: rrand(70, 130),
        res: 0.9, wave: 1, amp: 0.8
      sleep 0.125
    end
  end
end
```

[CHAPTER FIVE]

MUSICAL MINECRAFT

Sonic Pi can be used to make much more than just music – you can even code it to control Minecraft!

Below Send messages to Minecraft!



In the previous tutorials, we've focused purely on the music possibilities of Sonic Pi, which can turn your Raspberry Pi into a performance-ready musical instrument. So far, we've learned how to:

- **Live-code, changing the sounds on-the-fly**
- **Code some huge beats**
- **Generate powerful synth leads**
- **Recreate the famous TB-303 acid-bass sound**

There's so much more to show you (which we will explore later. However, this time, let's look at something Sonic Pi can do that you probably didn't realise: control *Minecraft*.

Hello Minecraft World

OK, let's get started. Boot up your Raspberry Pi, fire up *Minecraft Pi*, and create a new world in it. Now start up Sonic Pi, and resize and move your windows so that you can see both Sonic Pi and *Minecraft Pi* at the same time.

In a fresh buffer, type the following:

```
mc_message "Hello Minecraft from Sonic Pi!"
```

Now, hit **Run**. Boom! Your message appeared in *Minecraft*! How easy was that? Now, stop reading this for a moment and play about with your own messages. Have fun!

Sonic teleporter

Now let's do some exploring. The standard option is to reach for the mouse and keyboard and start walking around. That works, but it's pretty slow and boring. It would be far better if we had some sort of teleport machine. Well, thanks to Sonic Pi, we have one. Try this:

```
mc_teleport 80, 40, 100
```

Crikey! That was a long way up. If you weren't in flying mode, then you would have fallen back down all the way to the ground. If you double-tap the space bar to enter flying mode and teleport again, you'll stay hovering at the location you zap to.

Now, what do those numbers mean? We have three numbers which describe the coordinates of where in the Minecraft world we want to go. We give each number a name: x, y, and z:

- **x – how far left and right (80 in our example)**
- **y – how high we want to be (40 in our example)**
- **z – how far forward and back (100 in our example)**

By choosing different values for x, y, and z, we can teleport *anywhere* in our world. Try it! Choose different numbers and see where you can end up. If the screen goes black, it's because you've teleported yourself under the ground or into a mountain. Just choose a higher y value to get back out above land. Keep on exploring until you find somewhere you like.

Using the ideas so far, let's build a Sonic teleporter that makes a fun teleport sound whilst it whizzes us across the *Minecraft* world:

```
mc_message "Preparing to teleport...."
sample :ambi_lunar_land, rate: -1
sleep 1
mc_message "3"
sleep 1
mc_message "2"
sleep 1
mc_message "1"
sleep 1
mc_teleport 90, 20, 10
mc_message "Whoooosh!"
```

Magic blocks

Now you've found a nice spot, let's start building. You could do what you're used to and start clicking the mouse furiously to place blocks under the cursor. Or you could use the magic of Sonic Pi. Try this:

```
x, y, z = mc_location
mc_set_block :melon, x, y + 5, z
```

Now look up: there's a melon in the sky! Take a moment to look at the code. What did we do? On line one we grabbed the current location of Steve as the variables x, y, and z. These correspond to our coordinates described above. We use these coordinates in the function **`mc_set_block`**, which will place the block of your choosing at the specified coordinates. In order to make something higher up in the sky, we just need to increase the y value, which is why we add 5 to it. Let's make a long trail of them:

```
live_loop :melon_trail do
  x, y, z = mc_location
  mc_set_block :melon, x, y-1, z
  sleep 0.125
end
```

Now, jump over to *Minecraft*, make sure you're in flying mode (double-tap the space bar if not) and fly all around the world. Look behind you to see a pretty trail of melon blocks! See what kind of twisty patterns you can make in the sky.

Right Live coding with Minecraft is easier than you might think

Live-coding Minecraft

Those of you that have been following the tutorials in the previous chapters will probably have your minds blown at this point. The trail of melons is pretty cool, but the most exciting part of the previous example is that you can use **`live_loop`** with *Minecraft*! For those who don't know, **`live_loop`** is Sonic Pi's special magic ability that no other programming language has. It lets you run multiple loops at the same time and allows you to change them whilst they run. They are incredibly powerful, and amazing fun. I use **`live_loop`** to perform music in nightclubs with Sonic Pi: DJs may use discs, but I use **`live_loop`** instead! However, today we're going to be live-coding both music and *Minecraft*.

Let's get started. **Run** the code above and start making your melon trail again. Now, without stopping the code, just simply change **`:melon`** to **`:brick`** and hit run. Hey presto, you're now making a brick trail. How simple was that! Fancy some music to go with it? Easy. Try this:

```
live_loop :bass_trail do
  tick
  x, y, z = mc_location
  b = (ring :melon, :brick, :glass).look
  mc_set_block b, x, y -1, z
  note = (ring :e1, :e2, :e3).look
  use_synth :tb303
  play note, release: 0.1, cutoff: 70
  sleep 0.125
end
```

Now, whilst that's playing, start changing the code. Change the block types: you could try **`:water`**, **`:grass`**, or your favourite block type. Also, try changing the cutoff value from **`70`** to **`80`** and then up to **`100`**. Isn't this fun?

Bringing it all together

Let's combine everything we've seen so far with a little extra magic. We'll combine our teleportation ability with block placing and music to make a *Minecraft* music video. Don't worry if you don't understand it all: just type it in and have a play by changing some of the values whilst it's running live. Have fun, and see you next time…

```
live_loop :note_blocks do
  mc_message "This is Sonic Minecraft"
  with_fx :reverb do
    with_fx :echo, phase: 0.125, reps: 32 do
      tick
      x = (range 30, 90, step: 0.1).look
      y = 20
      z = -10
      mc_teleport x, y, z
      ns = (scale :e3, :minor_pentatonic)
      n = ns.shuffle.choose
      bs = (knit :glass, 3, :sand, 1)
      b = bs.look
      synth :beep, note: n, release: 0.1
      mc_set_block b, x+20, n-60+y, z+10
      mc_set_block b, x+20, n-60+y, z-10
      sleep 0.25
    end
  end
end

live_loop :beats do
  sample :bd_haus, cutoff: 100
  sleep 0.5
end
```

The MagPi ESSENTIALS

[CHAPTER SIX]

BINARY BIZET

We're going to bring a classical operatic dance piece straight into the 21st century, using the awesome power of code

Let's jump into a time machine and head back to the year 1875. A composer called Bizet had just finished his latest opera, *Carmen*. Unfortunately, like many exciting and disruptive new pieces of music, people initially didn't like it at all because it was too outrageous and different. Sadly, Bizet died ten years before the opera gained huge international success and became one of the most famous and frequently performed operas of all time. In sympathy with this tragedy, let's take one of the main themes from *Carmen* and convert it to a modern format of music that is also too outrageous and different for most people in our time: live-coded music!

Decoding the Habanera

Trying to live-code the whole opera would be a bit of a challenge for this tutorial, so let's focus on one of the most famous parts: the bassline to the Habanera.

This may look extremely unreadable to you if you haven't yet studied music notation. However, as programmers we see music notation as just another form of code, only it represents instructions to a musician instead of a computer. We therefore need to figure out a way of decoding it.

Notes

The notes are arranged from left to right, like the words in this magazine, but also have different heights. *The height on the score represents the pitch of the note.* The higher the note on the score, the higher the pitch of the note.

In Sonic Pi, we already know how to change the pitch of a note: we either use high or low numbers such as **`play 75`** and **`play 80`**, or we use the note names such as **`play :E`** and **`play :F`**. Luckily, each of the vertical positions of the musical score represents a specific note name, as shown here...

Rests

Music scores are an extremely rich and expressive kind of code, capable of communicating many things. It therefore shouldn't come as much of a surprise that musical scores can not only tell you what notes to play, but also when *not* to play notes. In programming, this is pretty much equivalent to the idea of 'nil' or 'null' – the absence of a value. In other words, not playing a note is like the absence of a note.

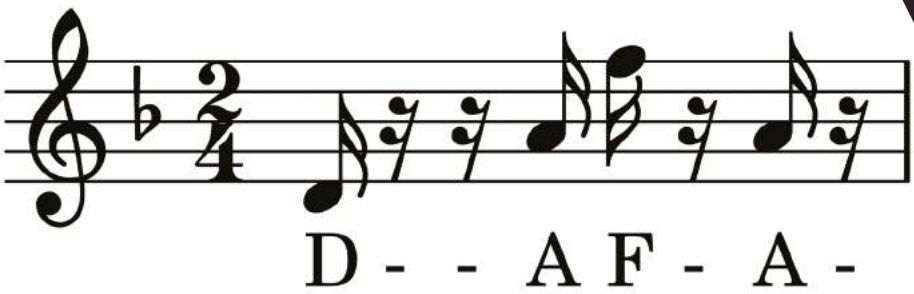
If you look closely at the score, you'll see that it's actually a combination of black dots with lines which represent notes to play, and squiggly things which represent the rests. Luckily, Sonic Pi (v2.7+) has a very handy representation for a rest – **`:r`**. So if we run **`play :r`**, it actually plays silence! We could also write **`play :rest`**, **`play nil`**, or **`play false`**, which are all equivalent ways of representing rests.

Rhythm

Finally, there's one last thing to learn how to decode in the notation: the timings of the notes. In the original notation, you'll see that the

notes are connected with thick lines called beams. The second note has two of these beams, which means it lasts for a 16th of a beat. The other notes have a single beam, which means they last for an 8th of a beat. The rest have two squiggly beams, which means they also represent a 16th of the beat.

When we decode and try to understand new things, a handy trick is to try to make everything as similar as possible to attempt to spot any relationships or patterns. When we rewrite our notation purely in 16ths, you can see that our notation just turns into a nice sequence of notes and rests...



Recoding the Habanera

We're now in a position to start translating this bassline to Sonic Pi. Let's encode these notes and rests in a ring:

```
(ring :d, :r, :r, :a, :f5, :r, :a, :r)
```

Let's see what this sounds like. Throw it in a live loop and tick through it:

```
live_loop :habanera do
   play (ring :d, :r, :r, :a, :f5, :r, :a, :r).tick
   sleep 0.25
end
```

Fabulous: that instantly recognisable riff springs to life through your speakers. It took a lot of effort to get here, but it was worth it. High-five!

Moody synths

Now we have the bassline, let's recreate some of the ambience of the operatic scene. One synth to try out is **:blade**, which is a moody 1980s-style synth lead. Let's try it with the starting note **:d**, passed through a slicer and reverb:

```
live_loop :habanera do
  use_synth :fm
  use_transpose -12
  play (ring :d, :r, :r, :a, :f5, :r, :a, :r).tick
  sleep 0.25
end

with_fx :reverb do
  live_loop :space_light do
    with_fx :slicer, phase: 0.25 do
      synth :blade, note: :d, release: 8, cutoff: 100, amp: 2
    end
    sleep 8
  end
end
```

Now, try the other notes in the bassline: **:a** and **:f5**. Remember, you don't need to hit Stop; just modify the code while the music is playing and hit **Run** again. Also, try different values for the slicer's **phase:**, such as **0.5**, **0.75**, and **1**.

Bringing it all together

Finally, let's combine all the ideas so far into a new remix of the Habanera. You might notice that I've included another part of the bassline as a comment. Once you've typed it all into a fresh buffer, hit **Run** to hear the composition. Now, without hitting Stop, *uncomment* the second line by removing the **#** and hit **Run** again; how marvellous is that? Now, start mashing it around yourself and have fun!

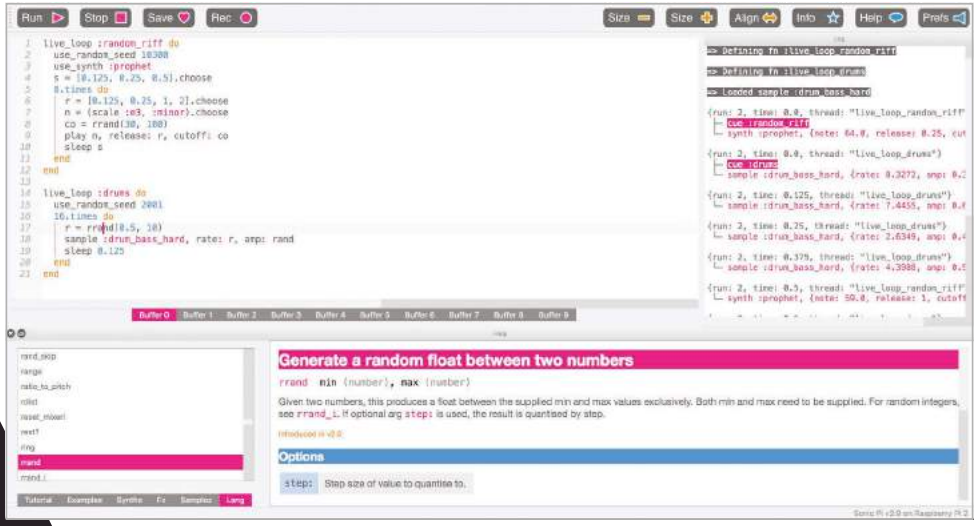
```
use_debug false
bizet_bass = (ring :d, :r, :r, :a, :f5, :r, :a, :r)
#bizet_bass = (ring :d, :r, :r, :Bb, :g5, :r, :Bb, :r)

with_fx :reverb, room: 1, mix: 0.3 do
  live_loop :bizet do
    with_fx :slicer, phase: 0.125 do
      synth :blade, note: :d4, release: 8,
        cutoff: 100, amp: 1.5
    end
    16.times do
      tick
      play bizet_bass.look, release: 0.1
      play bizet_bass.look - 12, release: 0.3
      sleep 0.125
    end
  end
end

live_loop :ind do
  sample :loop_industrial, beat_stretch: 1,
    cutoff: 100, rate: 1
  sleep 1
end

live_loop :drums do
  sample :bd_haus, cutoff: 110
  synth :beep, note: 49, attack: 0,
    release: 0.1
  sleep 0.5
end
```

The MagPi ESSENTIALS

[CHAPTER SEVEN]

SURFING RANDOM STREAMS

In this guide, we demonstrate the incredible power of randomisation in live-coded music



Back in chapter four, we looked at randomisation while coding some sizzling synth riffs. Since randomisation is an important part of my live-coding DJ sets, I thought it would be useful to cover the fundamentals in detail. So get your lucky hat on; let's surf some random streams!

There is no random

The first surprise when playing with the randomisation functions is that they're not random. What does this mean? Let's try some tests. Imagine a number in your head between 0 and 1, but don't tell me. Now let me guess… was it **`0.321567`**? No? I'm no good at this. Let me try again, but let's ask Sonic Pi to choose a number. Start Sonic Pi v2.9+ and ask it for a random number, but don't tell me:

```
print rand
```

Now for the reveal… was it **`0.75006103515625`**? Yes! I can see you're sceptical; it was a lucky guess. Let's try again. Press the **Run** button again and see what we get… The same again? This clearly can't be random! You're right, it's not.

What's going on? The fancy computer science term is determinism. This means that nothing is by chance and everything is destined to be. Sonic Pi is destined to always return **`0.75006103515625`** in the program above. It may sound useless, but in fact it's one of the most powerful parts of Sonic Pi. If you persevere, you'll learn how to rely on the deterministic nature of Sonic Pi's randomisation as a fundamental building block to your compositions and live-coded DJ sets.

A random tune

When Sonic Pi boots, it actually loads into memory a sequence of 441,000 pre-generated random values. When you call a random function such as **`rand`** or **`rrand`**, this random stream is used to generate your result. Each call to a random function

> "What's going on? The fancy computer science term is determinism"

uses a value from this stream, so the tenth call to a random function will use the tenth value from the stream. Also, every time you press **Run**, the stream is reset for that run. That's why I could predict the result to **`rand`** and why the 'random' tune was the same every time. Everybody's version of Sonic Pi uses the same random stream, which is important when we start sharing our pieces with each other.

Let's use this knowledge to generate a repeatable random tune:

```
8.times do
  play rrand_i(50, 95)
  sleep 0.125
end
```

Type this into a spare buffer and press **Run**. You'll hear a tune consisting of 'random' notes between **`50`** and **`95`**. When it's ended, press **Run** again to hear exactly the same tune again.

Resetting the stream

While repeating a sequence of notes is essential for replaying a riff on the dance floor, it might not be the one you want. Wouldn't it be great if we could try a number of different riffs and choose the one we liked best? This is where the real magic starts.

We can manually set the stream with the function **use_random_seed**. In computer science, a random seed is the starting point for a new stream of random values. Let's try it:

```
use_random_seed 0
3.times do
  play rrand_i(50, 95)
  sleep 0.125
end
```

Great, we get the first three notes of our tune above: **84**, **83**, and **71**. We can now change the seed to something else, like this:

```
use_random_seed 1
3.times do
  play rrand_i(50, 95)
  sleep 0.125
end
```

We get **83**, **71**, and **61**. You may spot that the first two numbers are the same as the last two numbers before – no coincidence.

The random stream is just a giant list of 'pre-rolled' values. Using a random seed simply jumps us to a point in that list. Imagine a big deck of pre-shuffled cards. Using a random seed is cutting the deck at a particular point. The great part of this is that it's this ability to jump around the random list which gives us power when making music.

Let's revisit our random eight-note tune with this new power, and also add a live loop so we can experiment live while it's playing:

Handy randomisation functions

Sonic Pi comes with a number of useful functions for working with the random stream. Here are some of the most useful:

- **rand** Simply returns the next value in the random stream
- **rrand** Returns a random value within a range
- **rrand_i** Returns a random whole number within a range
- **one_in** Returns true or false with the given probability
- **dice** Imitates rolling a dice and returns a value between 1 and 6
- **choose** Chooses a random value from a list

Check out their documentation in the Help system for detailed information and examples.

```
live_loop :random_riff do
  use_random_seed 0
  8.times do
    play rrand_i(50, 95), release: 0.1
    sleep 0.125
  end
end
```

While it's still playing, change the seed value from **0** to something else. Try **100**, or **999**. Try your own values and experiment – see which seed generates the riff you like best.

Bringing it all together

This tutorial has been quite a technical dive into the workings of Sonic Pi's randomisation functionality. Hopefully, it's explained how it works and how you can start using randomisation reliably to create repeatable patterns in your music. Crucially, you can use repeatable randomisation *anywhere*: the amplitude of notes, the timing of the rhythm, amount of reverb, current synth, the mix of an FX, etc. In the future we'll take a close look at these, but for now I'll end with a short example.

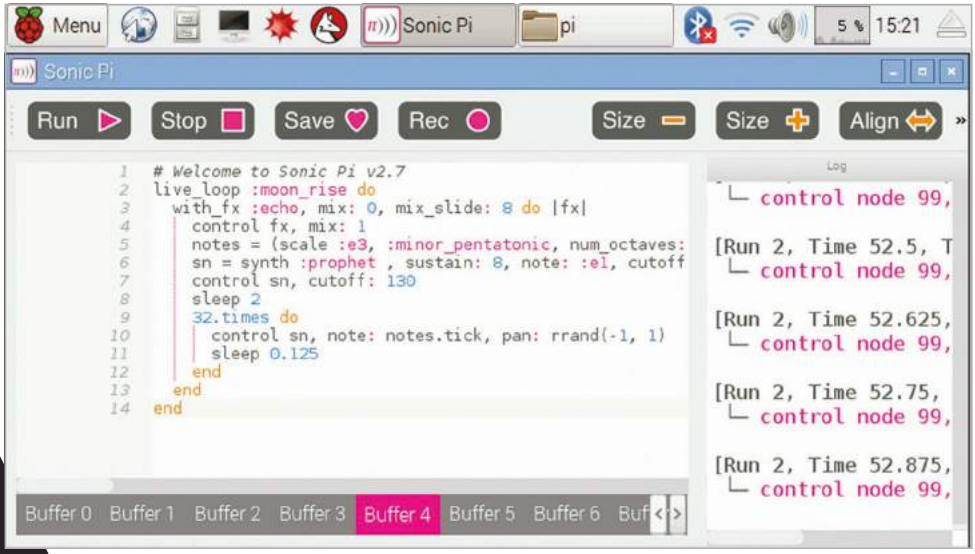
Type the code below into a spare buffer, press **Run**, then change the seeds. Press **Run** again while it's playing, and explore the different sounds, rhythms, and melodies you can make. When you find a nice one, note the seed number so you can return to it. Once you've found a few seeds you like, put on a live-coded performance by simply switching between your favourite seeds to make a full piece.

```
live_loop :random_riff do
  use_random_seed 10300
  use_synth :prophet
  s = [0.125, 0.25, 0.5].choose
  8.times do
    r = [0.125, 0.25, 1, 2].choose
    n = (scale :e3, :minor).choose
    co = rrand(30, 100)
    play n, release: r, cutoff: co
    sleep s
  end
end

live_loop :drums do
  use_random_seed 2001
  16.times do
    r = rrand(0.5, 10)
    sample :drum_bass_hard, rate: r, amp: rand
    sleep 0.125
  end
end
```

The MagPi ESSENTIALS

[CHAPTER EIGHT]
CONTROLLING YOUR SOUND

Shape and sculpt your sounds by automatically altering various parameters while they're playing



So far, we've focused on triggering sounds. We've discovered that we can trigger the many synths built into Sonic Pi with **play** or **synth**, and pre-recorded samples with **sample**. We've also looked at how we can wrap these triggered sounds within studio FX such as reverb and distortion, using **with_fx**. Combine this with Sonic Pi's incredibly accurate timing system and you can produce a vast array of sounds, beats, and riffs. However, once you've carefully selected a particular sound's options and triggered it, there's no ability to mess with it whilst it's playing, right? Wrong! Today you'll learn something very powerful: how to control running synths.

A basic sound

Let's create a nice simple sound. Fire up Sonic Pi and, in a fresh buffer, type the following:

```
synth :prophet, note: :e1, release: 8, cutoff: 100
```

Now press the **Run** button at the top left to hear a lovely rumbling synth sound. Go ahead, press it again a few times to get a feel for it. OK, done? Let's start controlling it!

Synth nodes

A little-known feature in Sonic Pi is that the fns **play**, **synth**, and **sample** return something called a 'SynthNode', which represents a running sound. You can capture one of these 'SynthNodes' using a standard variable and then *control* it at a later point in time. For example, let's change the value of the **cutoff:** opt after one beat:

```
sn = synth :prophet, note: :e1, release: 8, cutoff: 100
sleep 1
control sn, cutoff: 130
```

Let's look at each line in turn…

Firstly, we trigger the **:prophet** synth using the **synth** fn, as normal. However, we also capture the result in a variable called **sn**. We could have called this variable something completely different, such as 'synth_node' or 'jane' – the name doesn't matter. However, it's important to choose a name that's meaningful to you for your performances and for people reading your code. We chose **sn** as it's a nice short mnemonic for synth node.

On line 2 we have a standard **sleep** command. This does nothing special – it just asks the computer to wait for one beat before moving onto the next line.

Line 3 is where the control fun starts. Here, we use the **control** fn to tell our running 'SynthNode' to change the cutoff value to **130**. If you hit the **Run** button, you'll hear the **:prophet** synth start playing as before, but after one beat it will shift to sound a lot brighter.

Multiple changes

Whilst a synth is running, you're not limited to changing it only once – you're free to change it as many times as you like. For example, we can turn our **:prophet** into a mini arpeggiator with the following:

```
notes = (scale :e3, :minor_pentatonic)
sn = synth :prophet, note: :e1, release: 8, cutoff: 100
sleep 1
16.times do
  control sn, note: notes.tick
  sleep 0.125
end
```

In this snippet of code, we just added a couple of extra things. Firstly, we defined a new variable called **notes**, which contains the notes we'd like to cycle through (an arpeggiator is just a fancy name for something that cycles through a list of notes in order). Secondly, we replaced our single call to **control** with an iteration calling it 16 times. In each call to **control**, we **.tick** through our ring of **notes**, which will automatically repeat once we get to the end (thanks to the fabulous power of Sonic Pi's rings). For a bit of variety, try replacing **.tick** with **.choose** and see if you can hear the difference.

Note that we are able to change multiple opts simultaneously. Try changing the control line to the following and then listen for the difference:

```
control sn, note: notes.tick, cutoff: rrand(70, 130)
```

Modulatable options

Most of Sonic Pi's synths and FX opts may be changed after being triggered. However, this isn't the case for all of them. For example, the envelope opts **attack:**, **decay:**, **sustain:**, and **release:** can only be set when triggering the synth. Figuring out which opts can and can't be changed is simple – just head to the documentation for a given synth or FX, then scroll down to the individual option documentation and look for the phrases 'May be changed whilst playing' or 'Can not be changed once set'. For example, the documentation for the **:beep** synth's **attack:** opt makes it clear that it's not possible to change it:

- Default: 0
- Must be zero or greater
- Can not be changed once set
- Scaled with current BPM value

Sliding

When we control a 'SynthNode', it responds exactly on time and instantly changes the value of the opt to the new one, as if you'd pressed a button requesting the change. This can sound rhythmical and percussive – especially if the opt controls an aspect of the timbre, such as **`cutoff:`**. However, sometimes you don't want the change to happen instantaneously. Instead, you might want to smoothly move

> Sometimes you don't want the change to happen instantaneously

from the current value to the new one, as if you'd moved a slider or dial. Of course, Sonic Pi can also do this too using **`_slide:`** opts.

Each opt that can be modified also has a special corresponding **`_slide:`** opt that allows you to specify a slide time. For example, **`amp:`** has **`amp_slide:`**, and **`cutoff:`** has **`cutoff_slide:`**. These slide opts work slightly differently from all the other opts in that they tell the synth note how to behave *next time they are controlled*. Let's take a look:

```
sn = synth :prophet, note: :e1, release: 8, cutoff: 70,
  cutoff_slide: 2
sleep 1
control sn, cutoff: 130
```

Note how this example is the same as before, except with the addition of **`cutoff_slide:`**. This is saying that next time this synth has its **`cutoff:`** opt controlled, it will take two beats to slide from the current value to the new one. Therefore, when we use **`control`**, you can hear the cutoff slide from **`70`** to **`130`**. It creates an interesting

dynamic feel to the sound. Now, try reducing the **cutoff_slide:** time to **0.5**, or increasing it to **4**, to see how it changes the sound. Remember, you can slide any of the modifiable opts in this way, and each **_slide:** value can be totally different, so you can have the cutoff sliding slowly, the amp sliding fast, and the pan sliding somewhere in between if you like.

Bringing it all together

Let's look at a short example which demonstrates the power of controlling synths after they've been triggered. Note that you can also slide FX just like synths, but with a slightly different syntax. Check out section 7.2 of the built-in tutorial for more information on controlling FX.

Copy the code into a spare buffer and listen. Don't stop there, though – play around with the code. Change the slide times, the notes, the synth, the FX, and the sleep times and see if you can turn it into something completely different!

```
live_loop :moon_rise do
  with_fx :echo, mix: 0, mix_slide: 8 do |fx|
    control fx, mix: 1
    notes = (scale :e3, :minor_pentatonic,
             num_octaves: 2).shuffle
    sn = synth :prophet , sustain: 8, note: :e1,
      cutoff: 70, cutoff_slide: 8
    control sn, cutoff: 130
    sleep 2
    32.times do
      control sn, note: notes.tick, pan: rrand(-1, 1)
      sleep 0.125
    end
  end
end
```

The MagPi ESSENTIALS

[CHAPTER NINE]

BECOME A MINECRAFT VJ

Use Sonic Pi with Minecraft to create amazing visuals for your music as you perform it!

Everyone has built amazing structures, designed cunning traps, and even created elaborate cart tracks in *Minecraft*. How many of you have performed with *Minecraft*? We bet you didn't know that you could use *Minecraft* to create amazing visuals, just like a professional VJ.

As noted in the fifth chapter, you can program *Minecraft* with Sonic Pi as well as with Python, which makes the coding not only easy but also incredibly fun. In this chapter, we'll be showing you some of the tips and tricks that we've used to create performances in nightclubs and music venues around the world.

Enter a new world in *Minecraft* and open Sonic Pi. When we're using *Minecraft* to create visuals, we try to think about what will both look interesting and also be easy to generate from code. One nice trick is to create a sandstorm by dropping sand blocks from the sky. For that, all we need are a few basic functions:

- `sleep` – for inserting a delay between actions
- `mc_location` – to find our current location
- `mc_set_block` – to place sand blocks at a specific location
- `rrand` – to allow us to generate random values within a range
- `live_loop` – to allow us to continually make it rain sand

Let's make it rain a little first, before unleashing the full power of the storm. Grab your current location and use it to create a few sand blocks up in the sky nearby:

```
x, y, z = mc_location
mc_set_block :sand, x, y + 20, z + 5
sleep 2
mc_set_block :sand, x, y + 20, z + 6
sleep 2
mc_set_block :sand, x, y + 20, z + 7
sleep 2
mc_set_block :sand, x, y + 20, z + 8
```

When you press **Run**, you might have to look around a little, as the blocks may start falling down behind you depending on which direction you're currently facing. Don't worry: if you missed them, just press **Run** again for another batch of sand rain – just make sure you're looking the right way!

Let's quickly review what's going on here. On the first line, we grabbed Steve's location as coordinates with the fn **`mc_location`** and placed them into the vars **`x`**, **`y`**, and **`z`**. Then, on the next lines, we used

Below The rrand function is used to generate blocks in random positions

the **mc_set_block** fn to place some sand at the same coordinates as Steve, but with some modifications. We chose the same x coordinate, a y coordinate 20 blocks higher, and then successively larger z coordinates so the sand dropped in a line away from Steve.

> Try adding more lines, changing the sleep times

Why don't you take that code and start playing around with it yourself? Try adding more lines, changing the sleep times, try mixing **:sand** with **:gravel**, and choose different coordinates. Just experiment and have see what happens!

Live loops unleashed

OK, it's time to get the storm raging by unleashing the full power of Sonic Pi's magical ability, the **live_loop**, which unleashes the full power of live coding: changing code on the fly while it's running!

```
live_loop :sand_storm do
  x, y, z = mc_location
  xd = rrand(-10, 10)
  zd = rrand(-10, 10)
  co = rrand(70, 130)
  synth :cnoise, attack: 0, release: 0.125, cutoff: co
  mc_set_block :sand, x + xd, y+20, z+zd
  sleep 0.125
end
```

Now we're looping round pretty quickly (eight times a second), and during each loop we're finding Steve's location like before, but then generating three random values:

- **xd** – the difference for x, which will be between -10 and 10
- **zd** – the difference for z, also between -10 and 10
- **co** – a cutoff value for the low pass filter, between 70 and 130

Above Generate large patterned walls with Sonic Pi code

We use those random values in the fns **`synth`** and **`mc_set_block`**, giving us sand falling in random locations around Steve, along with a percussive rain-like sound from the **`:cnoise`** synth.

Now things can get really interesting, as we get stuck into live coding. While the code is running and the sand is pouring down, try changing one of the values, perhaps the **`sleep`** time to **`0.25`** or the **`:sand`** block type to **`:gravel`**. Now press the **Run** button again. Hey presto! Things have changed without the code even stopping. This is your gateway to performing like a real VJ. Keep practising and changing things around. How different can you make the visuals without stopping the code?

Epic block patterns

Finally, another great way of creating interesting visuals is to generate huge patterned walls to fly towards and get close to. For this effect, we'll need to move from placing the blocks randomly to placing them in an ordered manner. We can do this by nesting two sets of iteration; press the Help button and navigate to section 5.2 of the tutorial, 'Iteration and Loops', for more background on iteration. The funny **`|xd|`** after the **`do`** means that **`xd`** will be set for each value of the iteration. So, the first time it will be **`0`**,

then **1**, then **2** and so on. By nesting two lots of iteration together like this, we can generate all the coordinates for a square. We can then randomly choose block types from a ring of blocks for an interesting effect:

```
x, y, z = mc_location
bs = (ring :gold, :diamond, :glass)
10.times do |xd|
  10.times do |yd|
    mc_set_block bs.choose, x + xd, y + yd, z
  end
end
```

Pretty neat. Whilst we're having fun here, try changing **`bs.choose`** to **`bs.tick`** to move from a random pattern to a more regular one. Try changing the block types: the more adventurous among you might want to try sticking this within a **`live_loop`** so that the patterns keep changing automatically.

Now, for the VJ finale. Change the two **`10.times`** to **`100.times`** and press **Run**. Kaboom!... A gigantic wall of randomly placed bricks. Imagine how long it would take you to build that manually with your mouse! Double-tap **SPACE** to enter fly-mode and start swooping by for some great visual effects. Don't stop here, though – use your imagination to conjure up some cool ideas and then use the coding power of Sonic Pi to make it real. When you've practised enough, dim the lights and put on a VJ show for your friends!

Help with functions

If you're unfamiliar with any of the built-in fns such as **`rrand`**, just type the word into your buffer, click on it, and then press the keyboard combo **CTRL+I** to bring up the built-in documentation. Alternatively, you can navigate to the 'lang' tab in the Help system and then look up the fns directly, along with all the other exciting things you can do.

[CHAPTER TEN]
QUICK REFERENCE

You can find the full reference within Sonic Pi itself, but here are some of the most important for Synths, FX and Samples…

SECTION 01 - SYNTHS

BLADE RUNNER STYLE STRINGS

Straight from the 70s, evoking the mists of Blade Runner, this simple electro-style string synth is based on filtered saw waves and a variable vibraton.

```
use_synth :blade
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades

the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,
must be a value less than 131
May be changed whilst playing

vibrato_rate:

Number of wobbles per second. For realism this should be between 6 and 8, maybe even faster for really high notes.
Default: 6
Must be a value greater than or equal to 0.0,must be a value less than or equal to 20.0
May be changed whilst playing

vibrato_depth:

Amount of variation around the central note. 1 is the sensible maximum (but you can go up to 5 if you want a special effect), 0 would mean no vibrato. Works well around 0.15 but you can experiment.
Default: 0.15
Must be a value greater than or equal to 0.0,must be a value less than or equal to 5.0
May be changed whilst playing

vibrato_delay:

How long in seconds before the vibrato kicks in.
Default: 0.5
Must be zero or greater
Can not be changed once set

vibrato_onset:

How long in seconds before the vibrato reaches full power.
Default: 0.1
Must be zero or greater
Can not be changed once set

DETUNED SAW WAVE

A pair of detuned saw waves passed through a low pass filter. Two saw waves with slightly different frequencies generates a nice thick sound which is the basis for a lot of famous synth sounds. Thicken the sound by increasing the detune value, or create an octave-playing synth by choosing a detune of 12 (12 MIDI notes is an octave).

```
use_synth :dsaw
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater, must be a value less than 131
May be changed whilst playing

detune:

Distance (in MIDI notes) between components of sound. Affects thickness, sense of tuning and harmony. Tiny values such as 0.1 create a thick sound. Larger values such as 0.5 make the tuning sound strange. Even bigger values such as 5 create chord-like sounds.
Default: 0.1
May be changed whilst playing

BASIC FM SYNTHESIS

A sine wave with a fundamental frequency which is modulated at audio rate by another sine wave with a specific modulation, division and depth. Useful for generating a wide range of sounds by playing with the divisor and depth params. Great for deep powerful bass and crazy 70s sci-fi sounds.

```
use_synth :fm
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes

the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2

Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater, must be a value less than 131
May be changed whilst playing

divisor:

Modifies the frequency of the modulator oscillator relative to the carrier. Don't worry too much about what this means – just try different numbers out!
Default: 2
May be changed whilst playing

depth:

Modifies the depth of the carrier wave used to modify fundamental frequency. Don't worry too much about what this means – just try different numbers out!
Default: 1
May be changed whilst playing

HOOVER

Classic early 90's rave synth – 'a sort of slurry chorussy synth line like the classic Dominator by Human Resource'. Based on Dan Stowell's implementation in SuperCollider and Daniel Turczanski's port to Overtone. Works really well with portamento (see docs for the 'control' method).

```
use_synth :hoover
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically

a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0.05
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before

decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 130
Must be zero or greater, must be a value less than 131
May be changed whilst playing

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0.1
Must be zero or greater, must be a value less than 1
May be changed whilst playing

SYNTHPIANO

A basic piano synthesiser. Note that due to the plucked nature of this synth the envelope opts such as attack:, sustain: and release: do not work as expected. They can only shorten the natural length of the note, not prolong it. Also, the note: opt will only honour whole tones.

```
use_synth :piano
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3. Note that the piano synth can only play whole tones such as 60 and does not handle floats such as 60.3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

vel:

Velocity of keypress.
Default: 0.2
Must be a value between 0 and 1 inclusively
Can not be changed once set

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. With the piano synth, this opt can only have the effect of shortening the attack phase, not prolonging it.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level). With the piano synth, this opt can only have the effect of controlling the amp within the natural duration of the note and can not prolong the sound.

Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. With the piano synth, this opt can only have the effect of controlling the amp within the natural duration of the note and can not prolong the sound.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. With the piano synth, this opt can only have the effect of controlling the amp within the natural duration of the note and can not prolong the sound.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

hard:

Hardness of keypress.
Default: 0.5
Must be a value between 0 and 1 inclusively
Can not be changed once set

stereo_width:

Width of the stereo effect (which makes low notes sound towards the left, high notes towards the right). 0 to 1.
Default: 0
Must be a value between 0 and 1 inclusively
Can not be changed once set

THE PROPHET

Dark and swirly, this synth uses Pulse Width Modulation (PWM) to create a timbre which continually moves around. This effect is created using the pulse ugen which produces a variable width square wave. We then control the width of the pulses using a variety of LFOs – sin-osc and lf-tri in this case. We use a number of these LFO modulated pulse ugens with varying LFO type and rate (and phase in some cases) to provide the LFO with a different starting point. We then mix all these pulses together to create a thick sound and then feed it through a resonant low pass filter (rlpf). For extra bass, one of the pulses is an octave lower (half the frequency) and its LFO has a little bit of randomisation thrown into its frequency component for that extra bit of variety.

```
use_synth :prophet
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0

Must be a value between
-1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 110
Must be zero or greater,
must be a value less than 131
May be changed whilst playing

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0.7
Must be zero or greater,
must be a value less than 1
May be changed whilst playing

PULSE WAVE

A simple pulse wave with a low pass filter. This defaults to a square wave, but the timbre can be changed dramatically by adjusting the pulse_width arg between 0 and 1. The pulse wave is thick and heavy with lower notes and is a great ingredient for bass sounds.

```
use_synth :pulse
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played

(due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can ot be changed once set

cutoff

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,
must be a value less than 131
May be changed whilst playing

pulse_width:

The width of the pulse wave as a value between 0 and 1. A width of 0.5 will produce a square wave. Different values will change the timbre of the sound. Only valid if wave is type pulse.
Default: 0.5
Must be a value between
0 and 1 exclusively
May be changed whilst playing

PULSE WAVE WITH SUB

A pulse wave with a sub sine wave passed through a low pass filter. The pulse wave is thick and heavy with lower notes and is a great ingredient for bass sounds – especially with the sub wave.

```
use_synth :subpulse
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.

Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,must be a value less than 131
May be changed whilst playing

pulse_width:

The width of the pulse wave as a value between 0 and 1. A width of 0.5 will produce a square wave. Different values will change the timbre of the sound. Only valid if wave is type pulse.
Default: 0.5
Must be a value between 0 and 1 exclusively
May be changed whilst playing

sub_amp:

Amplitude for the additional sine wave.
Default: 1
May be changed whilst playing

sub_detune:

Amount of detune from the note for the additional sine wave. Default is -12
Default: -12
May be changed whilst playing

SUPERSAW

Thick swirly saw waves sparkling and moving about to create a rich trancy sound.

```
use_synth :supersaw
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a

sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 130
Must be zero or greater,
must be a value less than 131
May be changed whilst playing

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0.7
Must be zero or greater,
must be a value less than 1
May be changed whilst playing

TB-303 EMULATION

Emulation of the classic Roland TB-303 Bass Line synthesiser. Overdrive the res (i.e. use very large values) for that classic late 80s acid sound

```
use_synth :tb303
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.

Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
Must be one of the following values: [1, 2, 3, 4, 6, 7]
Can not be changed once set

cutoff:

The maximum cutoff value as a MIDI note.
Default: 120
Must be a value less than or equal to 130
May be changed whilst playing

cutoff_min:

The minimum cutoff value.
Default: 30
Must be a value less than or equal to 130
May be changed whilst playing

cutoff_attack:

Attack time for cutoff filter. Amount of time (in beats) for sound to reach full cutoff value. Default value is set to match amp envelope's attack value.
Default: attack
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

cutoff_decay:

Decay time for cutoff filter. Amount of time (in beats) for sound to move from full cutoff value (cutoff attack level) to the cutoff sustain level. Default value is set to match amp envelope's decay value.
Default: decay
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

cutoff_sustain:

Amount of time for cutoff value to remain at sustain level in beats. Default value is set to match amp envelope's sustain value.
Default: sustain
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

cutoff_release:

Amount of time (in beats) for sound to move from cutoff sustain value to cutoff min value. **Default** value is set to match amp envelope's release value.

Default: release
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

cutoff_attack_level:

The peak cutoff (value of cutoff at peak of attack) as a value between 0 and 1 where 0 is the :cutoff_min and 1 is the :cutoff value.
Default: 1
Must be a value between
0 and 1 inclusively
Can not be changed once set

cutoff_decay_level:

The level of cutoff after the decay phase as a value between 0 and 1 where 0 is the :cutoff_min and 1 is the :cutoff value.
Default: cutoff_sustain_level
Must be a value between
0 and 1 inclusively
Can not be changed once set

cutoff_sustain_level:

The sustain cutoff (value of cutoff at sustain time) as a value between 0 and 1 where 0 is the :cutoff_min and 1 is the :cutoff value.
Default: 1
Must be a value between
0 and 1 inclusively
Can not be changed once set

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0.9
Must be zero or greater,
must be a value less than 1
May be changed whilst playing

wave:

Wave type - 0 saw, 1 pulse, 2 triangle. Different waves will produce different sounds.
Default: 0
Must be one of the following values:
[0, 1, 2]
May be changed whilst playing

pulse_width:

The width of the pulse wave as a value between 0 and 1. A width of 0.5 will produce a square wave. Different values will change the timbre of the sound. Only valid if wave is type pulse.
Default: 0.5
Must be a value between
0 and 1 exclusively
May be changed whilst playing

ZAWA

Saw wave with oscillating timbre. Produces moving saw waves with a unique character controllable with the control oscillator (usage similar to mod synths).

```
use_synth :pulse
```

PARAMETERS

note:

Note to play. Either a MIDI number or a symbol representing a note. For example: 30, 52, :C, :C2, :Eb4, or :Ds3
Default: 52
Must be zero or greater
May be changed whilst playing

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Amount of time (in beats) for sound to reach full amplitude (attack_level). A short attack (i.e. 0.01) makes the initial part of the sound very percussive like a sharp tap. A longer attack (i.e 1) fades the sound in gently. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

decay:

Amount of time (in beats) for the sound to move from full amplitude (attack_level) to the sustain amplitude (sustain_level).
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

sustain:

Amount of time (in beats) for sound to remain at sustain level amplitude. Longer sustain values result in longer sounds. Full length of sound is attack + decay + sustain + release.
Default: 0
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

release:

Amount of time (in beats) for sound to move from sustain level amplitude to silent. A short release (i.e. 0.01) makes the final part of the sound very percussive (potentially resulting in a click). A longer release (i.e 1) fades the sound out gently. Full length of sound is attack + decay + sustain + release.
Default: 1
Must be zero or greater
Can not be changed once set
Scaled with current BPM value

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
Must be zero or greater
Can not be changed once set

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
Must be zero or greater
Can not be changed once set

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
Must be zero or greater
Can not be changed once set

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,
must be a value less than 131
May be changed whilst playing

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency.

Smaller values produce less resonance.
Default: 0.9
Must be zero or greater,
must be a value less than 1
May be changed whilst playing

phase:

Phase duration in beats
of timbre modulation.
Default: 1
Must be greater than zero
May be changed whilst playing
Scaled with current BPM value

phase_offset:

Initial phase offset of the sync wave
(a value between 0 and 1).
Default: 0
Must be a value between
0 and 1 inclusively
Can not be changed once set

wave:

Wave shape controlling freq sync saw
wave. 0=saw wave, 1=pulse, 2=triangle
wave and 3=sine wave.
Default: 3
Must be one of the following values:
[0, 1, 2, 3]
May be changed whilst playing

invert_wave:

Invert sync freq control waveform
(i.e. flip it on the y axis). 0=uninverted
wave, 1=inverted wave.
Default: 0
Must be one of the following values:
[0, 1]
May be changed whilst playing

range:

Range of the associated sync saw
in MIDI notes from the main note.
Modifies timbre.
Default: 24
Must be a value between
0 and 90 inclusively
May be changed whilst playing

disable_wave:

Enable and disable sync control
wave (setting to 1 will stop timbre
movement).
Default: 0
Must be one of the following values:
[0, 1]
May be changed whilst playing

pulse_width:

The width of the pulse wave
as a value between 0 and 1. A width
of 0.5 will produce a square wave.
Different values will change the
timbre of the sound. Only valid
if wave is type pulse.
Default: 0.5
Must be a value between
0 and 1 exclusively
May be changed whilst playing

SECTION 02 - FX

BITCRUSHER

Creates lo-fi output by decimating and deconstructing the incoming audio by lowering both the sample rate and bit depth. The default sample rate for CD audio is 44100, so use values less than that for that crunchy chip-tune sound full of artefacts and bitty distortion. Similarly, the default bit depth for CD audio is 16, so use values less than that for lo-fi sound.

```
with_fx :bitcrusher do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)

Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the default) and a mix of 0.5 means that half the original and half of the FX is heard.

Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.

Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

sample_rate:

The sample rate the audio will be resampled at. This represents the number of times per second the audio is sampled. The higher the sample rate, the closer to the original the sound will be, the lower the more low-fi it will sound. The highest sample rate is 44100 (full quality) and the lowest is ~100 (extremely low quality). Try values in between such as 1000, 3000, 8000…

Default: 10000
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes

bits:

The bit depth of the resampled audio. Lower bit depths make the audio sound grainy and less defined. The highest bit depth is 16 (full quality) and the lowest is 1 (lowest quality).

Default: 8
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.

Default: 0
Must be zero or greater,
must be a value less than 131
May be changed whilst playing
Has slide parameters to shape changes

COMPRESSOR

Compresses the dynamic range of the incoming signal. Equivalent to automatically turning the amp down when the signal gets too loud and then back up again when it's quiet. Useful for ensuring the containing signal doesn't overwhelm other aspects of the sound. Also a general purpose hard-knee dynamic range processor which can be tuned via the opts to both expand and compress the signal.

```
with_fx :compressor do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't

make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

threshold:

Threshold value determining the break point between slope_below and slope_above.
Default: 0.2
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

clamp_time:

Time taken for the amplitude adjustments to kick in fully (in seconds). This is usually pretty small (not much more than 10 milliseconds). Also known as the time of the attack phase
Default: 0.01
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

slope_above:

Slope of the amplitude curve above the threshold. A value of 1 means that the output of signals with amplitude above the threshold will be unaffected. Greater values will magnify and smaller values will attenuate the signal.
Default: 0.5
May be changed whilst playing
Has slide parameters to shape changes

slope_below:

Slope of the amplitude curve below the threshold. A value of 1 means that the output of signals with amplitude below the threshold will be unaffected. Greater values will magnify and smaller values will attenuate the signal.
Default: 1
May be changed whilst playing
Has slide parameters to shape changes

relax_time:

Time taken for the amplitude adjustments to be released. Usually a little longer than clamp_time. If both times are too short, you can get some (possibly unwanted) artefacts.

Also known as the time of the release phase.
Default: 0.01
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

ECHO

Standard echo with variable phase duration (time between echoes) and decay (length of echo fade out). If you wish to have a phase duration longer than 2s, you need to specify the longest phase duration you'd like with the arg max_phase. Be warned, echo FX with very long phases can consume a lot of memory and take longer to initialise.

```
with_fx :echo do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input

signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

phase:

The time between echoes in beats.
Default: 0.25
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

decay:

The time it takes for the echoes to fade away in beats.
Default: 2
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

max_phase:

The maximum phase duration in beats.
Default: 2
Must be greater than zero
Can not be changed once set

FLANGER

Mix the incoming signal with a copy of itself which has a rate modulating faster and slower than the original. Creates a swirling/whooshing effect.

```
with_fx :flanger do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the

default) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

phase:

Phase duration in beats of flanger modulation.
Default: 4
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

phase_offset:

Initial modulation phase offset (a value between 0 and 1).
Default: 0
Must be a value between 0 and 1 inclusively
Can not be changed once set

wave:

Wave type - 0 saw, 1 pulse, 2 triangle, 3 sine, 4 cubic. Different waves will produce different flanging modulation effects.
Default: 4
Must be one of the following values: [0, 1, 2, 3, 4]
May be changed whilst playing

invert_wave:

Invert flanger control waveform (i.e. flip it on the y axis). 0=uninverted wave, 1=inverted wave.
Default: 0
Must be one of the following values: [0, 1]
May be changed whilst playing

stereo_invert_wave:

Make the flanger control waveform in the left ear an inversion of the control waveform in the right ear. 0=uninverted wave, 1=inverted wave. This happens after the standard wave inversion with param :invert_wave.
Default: 0
Must be one of the following values: [0, 1]
May be changed whilst playing

delay:

Amount of delay time between original and flanged version of audio.
Default: 5
May be changed whilst playing
Has slide parameters to shape changes

max_delay:

Max delay time. Used to set internal buffer size.
Default: 20

Must be zero or greater
Can not be changed once set

depth:

Flange depth - greater depths produce a more prominent effect.
Default: 5
May be changed whilst playing
Has slide parameters to shape changes

decay:

Flange decay time in ms
Default: 2
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

feedback:

Amount of feedback.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

invert_flange:

Invert flanger signal. 0=no inversion, 1=inverted signal.
Default: 0
Must be one of the following values: [0, 1]
May be changed whilst playing

KRUSH

Krush that sound!

```
with_fx :krush do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically

the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

gain:

Amount of crushing to serve.
Default: 5
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,must be a value less than 131
May be changed whilst playing
Has slide parameters to shape changes

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0
Must be zero or greater, must be a value less than 1
May be changed whilst playing
Has slide parameters to shape changes

LOW PASS FILTER

Dampens the parts of the signal that are higher than the cutoff point (typically the crunchy fizzy harmonic overtones) and keeps the lower parts (typically the bass/mid of the sound). Choose a higher cutoff to keep more of the high frequences/treble of the sound and a lower cutoff to make the sound more dull and only keep the bass.

```
with_fx :lpf do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.
Default: 100
Must be zero or greater,
must be a value less than 131
May be changed whilst playing
Has slide parameters to shape changes

PAN

Specify where in the stereo field the sound should be heard. A value of −1 for pan will put the sound in the left speaker, a value of 1 will put the sound in the right speaker and values in between will shift the sound accordingly.

```
with_fx :pan do   play 50 end
```

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
Must be a value between -1 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

PAN SLICER

Slice the pan automatically from left to right. Behaves similarly to slicer and wobble FX but modifies stereo panning of sound in left and right speakers. Default slice wave form is square (hard slicing between left and right) however other wave forms can be set with the wave: opt.

```
with_fx :panslicer do   play 50 end
```

PARAMETERS

amp:

The amplitude of the resulting effect.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

phase:

The phase duration (in beats) of the slices.
Default: 0.25
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

pan_min:

Minimum pan value (-1 is the left speaker only.)
Default: -1
Must be a value between -1 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pan_max:

Maximum pan value (+1 is the right speaker only)
Default: 1
Must be a value between -1 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pulse_width:

The width of the pulse wave as a value between 0 and 1. A width of 0.5 will produce a square wave. Different values will change the timbre of the sound. Only valid if wave is type pulse.
Default: 0.5
Must be a value between 0 and 1 exclusively
May be changed whilst playing
Has slide parameters to shape changes

phase_offset:

Initial phase offset.

Default: 0
Must be a value between
0 and 1 inclusively
Can not be changed once set

wave:

Control waveform used to modulate the amplitude. 0=saw, 1=pulse, 2=tri, 3=sine
Default: 1
Must be one of the following values: [0, 1, 2, 3]
May be changed whilst playing

invert_wave:

Invert control waveform (i.e. flip it on the y axis). 0=uninverted wave, 1=inverted wave.
Default: 0
Must be one of the following values: [0, 1]
May be changed whilst playing

probability:

Probability (as a value between 0 and 1) that a given slice will be replaced by the value of the prob_pos opt (which defaults to 0, i.e. silence)
Default: 0
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

prob_pos:

Position of the slicer that will be jumped to when the probability test passes as a value between 0 and 1
Default: 0
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

seed:

Seed value for rand num generator used for probability test.
Default: 0
Can not be changed once set

smooth:

Amount of time in seconds to transition from the current value to the next. Allows you to round off harsh edges in the slicer wave which may cause clicks.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_up:

Amount of time in seconds to transition from the current value to the next only when the value is going up. This smoothing happens before the main smooth mechanism.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_down:

Amount of time in seconds to transition from the current value to the next only when the value is going

down. This smoothing happens before the main smooth mechanism.
Default: 0

Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

REVERB

Make the incoming signal sound more spacious or distant as if it were played in a large room or cave. Signal may also be dampened by reducing the amplitude of the higher frequencies.

```
with_fx :reverb do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the default) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 0.4
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

room:

The room size – a value between 0 (no reverb) and 1 (maximum reverb).
Default: 0.6
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

damp:

High frequency dampening – a value between 0 (no dampening) and 1 (maximum dampening).
Default: 0.5
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

SLICER

Modulates the amplitude of the input signal with a specific control wave and phase duration. With the default pulse wave, slices the signal in and out, with the triangle wave, fades the signal in and out and with the saw wave, phases the signal in and then dramatically out. Control wave may be inverted with the arg invert_wave for more variety.

```
awith_fx :slicer do   play 50 end
```

PARAMETERS

amp:

The amplitude of the resulting effect.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.
Default: 1
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

phase:

Phase duration (in beats) of the slices.
Default: 0.25
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

amp_min:

Minimum amplitude of the slicer.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

amp_max:

Maximum amplitude of the slicer.
Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

pulse_width:

The width of the pulse wave as a value between 0 and 1. A width of 0.5 will produce a square wave. Different values will change the timbre of the sound. Only valid if wave is type pulse.
Default: 0.5
Must be a value between
0 and 1 exclusively
May be changed whilst playing
Has slide parameters to shape changes

phase_offset:

Initial phase offset.
Default: 0
Must be a value between
0 and 1 inclusively
Can not be changed once set

wave:

Control waveform used to modulate the amplitude. 0=saw, 1=pulse, 2=tri, 3=sine
Default: 1
Must be one of the following values:
[0, 1, 2, 3]
May be changed whilst playing

invert_wave:

Invert control waveform (i.e. flip it on the y axis). 0=uninverted wave, 1=inverted wave.
Default: 0
Must be one of the following values:
[0, 1]
May be changed whilst playing

probability:

Probability (as a value between 0 and 1) that a given slice will be replaced by the value of the prob_pos opt (which defaults to 0, i.e. silence.)
Default: 0
Must be a value between
0 and 1 inclusively

May be changed whilst playing
Has slide parameters to shape changes

prob_pos:

Position of the slicer that will be jumped to when the probability test passes as a value between 0 and 1
Default: 0
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

seed:

Seed value for rand num generator used for probability test.
Default: 0
Can not be changed once set

smooth:

Amount of time in seconds to transition from the current value to the next. Allows you to round off harsh edges in the slicer wave which may cause clicks.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_up:

Amount of time in seconds to transition from the current value to the next only when the value is going up. This smoothing happens before the main smooth mechanism.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_down:

Amount of time in seconds to transition from the current value to the next only when the value is going down. This smoothing happens before the main smooth mechanism.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

WOBBLE

Versatile wobble FX. Will repeatedly modulate a range of filters (rlpf, rhpf) between two cutoff values using a range of control wave forms (saw, pulse, tri, sine). You may alter the phase duration of the wobble, and the resonance of the filter. Combines well with the dsaw synth for crazy dub wobbles. Cutoff value is at cutoff_min at the start of phase

```
with_fx :wobble do   play 50 end
```

PARAMETERS

amp:

The amplitude of the sound. Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)

Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

mix:

The amount (percentage) of FX present in the resulting sound represented as a value between 0 and 1. For example, a mix of 0 means that only the original sound is heard, a mix of 1 means that only the FX is heard (typically the **default**) and a mix of 0.5 means that half the original and half of the FX is heard.

Default: 1
Must be a value between 0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

pre_amp:

Amplification applied to the input signal immediately before it is passed to the FX.

Default: 1
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

phase:

The phase duration (in beats) for filter modulation cycles.

Default: 0.5
Must be greater than zero
May be changed whilst playing
Has slide parameters to shape changes
Scaled with current BPM value

cutoff_min:

Minimum (MIDI) note that filter will move to whilst wobbling. Choose a lower note for a higher range of movement. Full range of movement is the distance between cutoff_max and cutoff_min.

Default: 60
Must be zero or greater, must be a value less than 130
May be changed whilst playing
Has slide parameters to shape changes

cutoff_max:

Maximum (MIDI) note that filter will move to whilst wobbling. Choose a higher note for a higher range of movement. Full range of movement is the distance between cutoff_max and cutoff_min.

Default: 120
Must be zero or greater, must be a value less than 130
May be changed whilst playing
Has slide parameters to shape changes

res:

Filter resonance as a value between 0 and 1. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.
Default: 0.8
Must be zero or greater,
must be a value less than 1
May be changed whilst playing
Has slide parameters to shape changes

phase_offset:

Initial modulation phase offset (a value between 0 and 1).
Default: 0
Must be a value between
0 and 1 inclusively
Can not be changed once set

wave:

Wave shape of wobble. Use 0 for saw wave, 1 for pulse, 2 for triangle wave and 3 for a sine wave.
Default: 0
Must be one of the following values:
[0, 1, 2, 3]
May be changed whilst playing

invert_wave:

Invert control waveform (i.e. flip it on the y axis). 0=uninverted wave, 1=inverted wave.
Default: 0
Must be one of the following
values: [0, 1]
May be changed whilst playing

pulse_width:

Only valid if wave is type pulse.
Default: 0.5
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

filter:

Filter used for wobble effect. Use 0 for a resonant low pass filter or 1 for a resonant high pass filter.
Default: 0
Must be one of the following
values: [0, 1]
May be changed whilst playing

probability:

Probability (as a value between 0 and 1) that a given wobble will be replaced by the value of the prob_pos opt (which defaults to 0, i.e. min_cutoff)
Default: 0
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

prob_pos:

Position of the wobble that will be jumped to when the probability test passes as a value between 0 and 1
Default: 0
Must be a value between
0 and 1 inclusively
May be changed whilst playing
Has slide parameters to shape changes

seed:

Seed value for rand num generator
used for probability test
Default: 0
Can not be changed once set

smooth:

Amount of time in seconds to
transition from the current value to
the next. Allows you to round off harsh
edges in the slicer wave which may
cause clicks.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_up:

Amount of time in seconds to transition
from the current value to the next only
when the value is going up.
This smoothing happens before
the main smooth mechanism.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

smooth_down:

Amount of time in seconds to transition
from the current value to the next only
when the value is going down. This
smoothing happens before the main
smooth mechanism.
Default: 0
Must be zero or greater
May be changed whilst playing
Has slide parameters to shape changes

SECTION 03 - SAMPLES

amp:

The amplitude of the sound.
Typically a value between 0 and 1. Higher amplitudes may be used, but won't make the sound louder, they will just reduce the quality of all the sounds currently being played (due to compression.)
Default: 1
must be zero or greater
May be changed whilst playing

pan:

Position of sound in stereo. With headphones on, this means how much of the sound is in the left ear, and how much is in the right ear. With a value of -1, the sound is completely in the left ear, a value of 0 puts the sound equally in both ears and a value of 1 puts the sound in the right ear. Values in between -1 and 1 move the sound accordingly.
Default: 0
must be a value between -1 and 1 inclusively
May be changed whilst playing

attack:

Duration of the attack phase of the envelope.
Default: 0
must be zero or greater

decay:

Duration of the decay phase of the envelope.
Default: 0
must be zero or greater

sustain:

Duration of the sustain phase of the envelope. When -1 (the default) will auto-stretch.
Default: -1
must either be a positive value or -1

release:

Duration of the release phase of the envelope.
Default: 0
must be zero or greater

attack_level:

Amplitude level reached after attack phase and immediately before decay phase.
Default: 1
must be zero or greater

decay_level:

Amplitude level reached after decay phase and immediately before sustain phase. Defaults to sustain_level unless explicitly set.
Default: sustain_level
must be zero or greater

sustain_level:

Amplitude level reached after decay phase and immediately before release phase.
Default: 1
must be zero or greater

env_curve:

Select the shape of the curve between levels in the envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
must be one of the following values: [1, 2, 3, 4, 6, 7]

cutoff_attack:

Attack time for cutoff filter. Amount of time (in beats) for sound to reach full cutoff value. Default value is set to match amp envelope's attack value.
Default: attack
must be zero or greater

cutoff_decay:

Decay time for cutoff filter. Amount of time (in beats) for sound to move from full cutoff value (cutoff attack level) to the cutoff sustain level. Default value is set to match amp envelope's decay value.
Default: decay
must be zero or greater

cutoff_sustain:

Amount of time for cutoff value to remain at sustain level in beats. When -1 (the default) will auto-stretch.
Default: sustain
must either be a positive value or -1

cutoff_release:

Amount of time (in beats) for sound to move from cutoff sustain value to cutoff min value. Default value is set to match amp envelope's release value.
Default: release
must be zero or greater

cutoff_attack_level:

The peak cutoff (value of cutoff at peak of attack) as a MIDI note.
Default: cutoff
must be a value between 0 and 130 inclusively

cutoff_decay_level:

The level of cutoff after the decay phase as a MIDI note.
Default: cutoff
must be a value between 0 and 130 inclusively

cutoff_sustain_level:

The sustain cutoff (value of cutoff at sustain time) as a MIDI note.
Default: cutoff
must be a value between 0 and 130 inclusively

cutoff_env_curve:

Select the shape of the curve between levels in the cutoff envelope. 1=linear, 2=exponential, 3=sine, 4=welch, 6=squared, 7=cubed
Default: 2
must be one of the following values: [1, 2, 3, 4, 6, 7]

cutoff_min:

The minimum cutoff value.
Default: 30
must be a value less than or equal to 130 May be changed whilst playing

rate:

Rate with which to play back - default is 1. Playing the sample at rate 2 will play it back at double the normal speed. This will have the effect of doubling the frequencies in the sample and halving the playback time. Use rates lower than 1 to slow the sample down. Negative rates will play the sample in reverse.

Default: 1

must not be zero

start:

A fraction (between 0 and 1) representing where in the sample to start playback. 1 represents the end of the sample, 0.5 half-way through etc.

Default: 0

must be a value between 0 and 1 inclusively

finish:

A fraction (between 0 and 1) representing where in the sample to finish playback. 1 represents the end of the sample, 0.5 half-way through etc.

Default: 1

must be a value between 0 and 1 inclusively

res:

Filter resonance as a value between 0 and 1. Only functional if a cutoff value is specified. Large amounts of resonance (a res: near 1) can create a whistling sound around the cutoff frequency. Smaller values produce less resonance.

Default: 0

must be zero or greater,

must be a value less than 1

May be changed whilst playing

cutoff:

MIDI note representing the highest frequencies allowed to be present in the sound. A low value like 30 makes the sound round and dull, a high value like 100 makes the sound buzzy and crispy.

Default: 0

must be zero or greater,

must be a value less than 131

May be changed whilst playing

norm:

Normalise the audio (make quieter parts of the sample louder and louder parts quieter) - this is similar to the normaliser FX. This may emphasise any clicks caused by clipping.

Default: 0

must be one of the following values: [0, 1]

window_size:

Pitch shift works by chopping the input into tiny slices, then playing these slices at a higher or lower rate. If we make the slices small enough and overlap them, it sounds like the original sound with the pitch changed. The window_size is the length of the slices and is measured in seconds. It needs to be around 0.2 (200ms) or greater for pitched sounds like guitar or bass, and needs

to be around 0.02 (20ms) or lower for percussive sounds like drum loops. You can experiment with this to get the best sound for your input.
Default: 0.2
must be a value greater than 5.0e-05
May be changed whilst playing

pitch_dis:

Pitch dispersion - how much random variation in pitch to add. Using a low value like 0.001 can help to "soften up" the metallic sounds, especially on drum loops. To be really technical, pitch_dispersion is the maximum random deviation of the pitch from the pitch ratio (which is set by the pitch param.)
Default: 0.0
must be a value greater than
or equal to 0
May be changed whilst playing

time_dis:

Time dispersion - how much random delay before playing each grain (measured in seconds). Again, low values here like 0.001 can help to soften up metallic sounds introduced by the effect. Large values are also fun as they can make soundscapes and textures from the input, although you will most likely lose the rhythm of the original. NB - This won't have an effect if it's larger than window_size.
Default: 0.0
must be a value greater than
or equal to 0
May be changed whilst playing.

The MagPi ESSENTIALS

raspberrypi.org/**magpi**