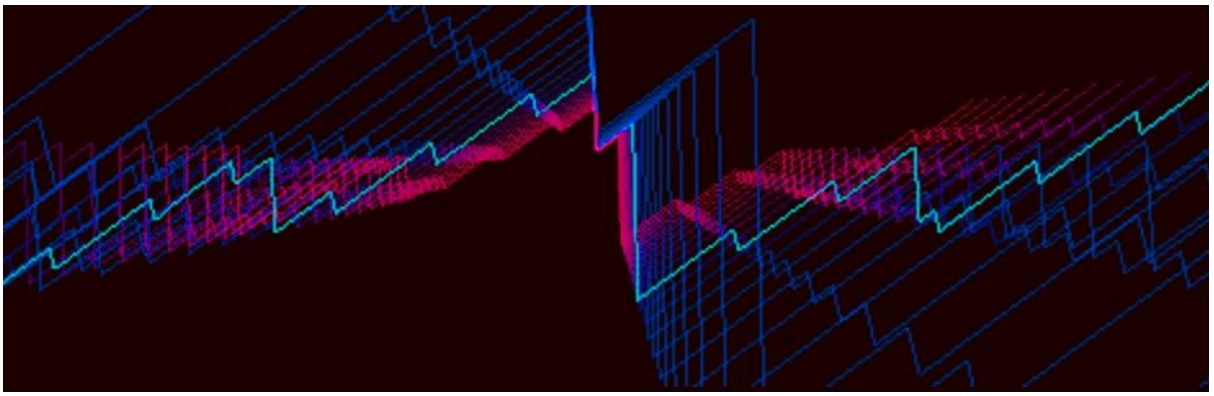


# *The absolute beginner's guide to coding Bytebeats!*

v1.4, 2019-10-19

written by
The Tuesday Night Machines
hello@nightmachines.tv
http://nightmachines.tv/youtube

Good morning!

Thank you for checking out my introduction to Bytebeat coding! When I discovered Bytebeats, I was amazed how a few characters of text could generate such a vast variety of awesome lo-fi sounds.

I'm not a good mathematician or programmer though, so at first I believed that I would never be able to code a Beat without resorting entirely to trial and error. Of course randomly hacking Bytebeats was still fun, but after a while, I really wanted to know why things sounded the way they did and have the the ability to deterministically add elements to my Beats. Unfortunately, specific information for absolute beginners was surprisingly hard to find online, so after a bunch of long train-rides and revisiting old math and programming lessons, I finally had some of the awaited epiphanies, resulting in a compilation of notes, which were the basis for the document at hand.

This guide is for electronic musicians with no, or barely any, programming experience, who enjoy experimenting with weird audio. You should know a little bit about waveforms, like sawtooth, square and triangle waves, frequencies and amplitudes in an electronic music context, as well as basic arithmetic: addition, subtraction, multiplication and division.

Anything else, I try to explain in an easy to follow manner, with fun exercises in between. Let me know how this guide worked for you.

Happy coding :-)

Felix / The Tuesday Night Machines

Get the latest version of this PDF here:
https://github.com/TuesdayNightMachines/Bytebeats

*Table of contents:*

## *What are Bytebeats?*

Bytebeats are musical creations from only a few lines of code, typically directed to an unsigned 8 bit, 8 kHz audio stream. This means that a mathematical expression is processed by the computer 8,000 times per second, resulting in an audible waveform with a 256-step resolution from silence (0) to full amplitude or "volume" (255).

For example:
`(t*4|t|t>>3&t+t/4&t*12|t*8>>10|t/20&t+140)&t>>4`
(click here to listen to it in your web browser)

Oh man … that looks super abstract and complicated! How is that fun?!

Honestly, when I wrote it, I had almost no clue what was going on. The above example was the result of pure trial and error and I assume that many people code Bytebeats purely in this fashion. It is still lots of fun, because it's quick and easy to do and happy accidents are plenty. It's the ultimate weird sample factory for your daily commute :D

Let's try it out right now!

**Exercise 01: Experimentation**

Open Greggman's HTML 5 Bytebeat Player using this link and press play. The variable `t` generates a sawtooth wave. Leave this as it is. Now copy and paste some of the following lines of code after `t` and see what happens.

```
|t*4
|t%128*10
|t*5^t/30
|t/20
&120
^t*7
```

Feel free to rewrite the code too, by changing values and operators! Notice how the lines above all start with a logical operator like `OR |`, `AND &` and `XOR ^`. This is required to combine several statements after the initial `t`.

If you encounter interesting sounds or melodies, save the code in a text file, Google Doc or write them in your diary.

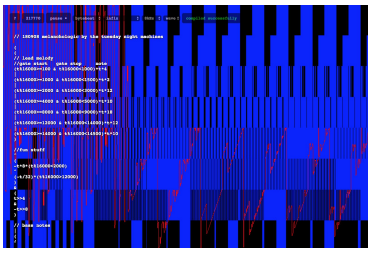
I hope that was somewhat enjoyable! Who needs programming knowledge anyway?

However, in this introduction I would still like to teach you some of the basics of Bytebeat coding, so that you can combine the trial and error results with more predictable elements.

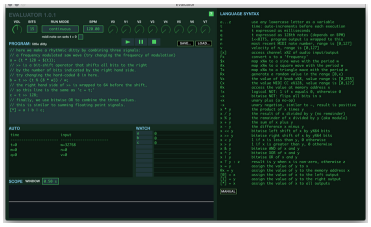
## *How to code Bytebeats?*

Various applications exist to make coding, playing and recording Bytebeats more accessible, compared to doing it manually in the Linux Terminal. Here are four examples.

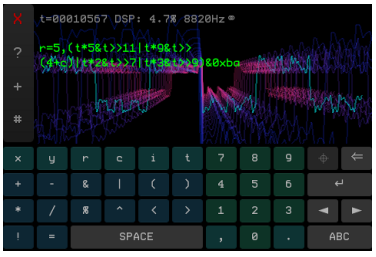
**Greggman's HTML 5 Bytebeat Player** lets you create Bytebeats in your web browser:
https://greggman.com/downloads/examples/html5bytebeat/html5bytebeat.html



**Damien Quartz' Evaluator** is a full-featured Bytebeat player with MIDI support for macOS and Windows. It's available as a standalone app or VST plugin:
https://damikyu.itch.io/evaluator



**Kymatica's BitWiz iOS App** offers a clean, multi-line coding interface, an oscilloscope, performance controls and inter-app audio:
http://kymatica.com/Software/BitWiz




**Single Cell's Caustic 3** for Android, iOS, Windows and macOS is a full-featured DAW including a Bytebeat synthesizer, called "8BitSynth":
http://www.singlecellsoftware.com/caustic

## *What makes an expression a "Bytebeat"?*

Earlier, I wrote that a Bytebeat is a mathematical expression, executed through an 8 bit, 8 kHz audio output, so one might think that one can just write anything imaginable with enough knowledge of math and programming. But what makes the result a "Bytebeat" then exactly?

Bytebeats can be considered a form of low-complexity art, with ties to the Demoscene, where the aim is to create art in often incredibly limited environments.

For Bytebeats, these limitations have typically been:

- One expression only
- Only one pre-defined variable: `t`
- Mathematical operators: `( ) + - * / %`
- Bitwise operators: `& | ^ << >>`
- Relational operators: `< > <= >= == !=`

Of course, those are just arbitrary limitations and you may do whatever you like. If you don't understand any of the above things yet, don't worry. We'll start learning about them right away.

But first, try out the above mathematical, bitwise and relational operators in combination with `t` and randomly selected numbers again. Just like we did earlier. See what you come up with and save your cool code snippets!

**Exercise 02: Operators**

Here is a new link to the HTML 5 Bytebeat Player with `t/20` as a starting point. Now, add one or more of the following lines of code and experiment with replacing the operators with the ones above.

```
|t*4*(t%10000>2000)&t>>4
|t&64
|t*t%128*(t%15000<2000)
|t
```

**Tip: Comments**

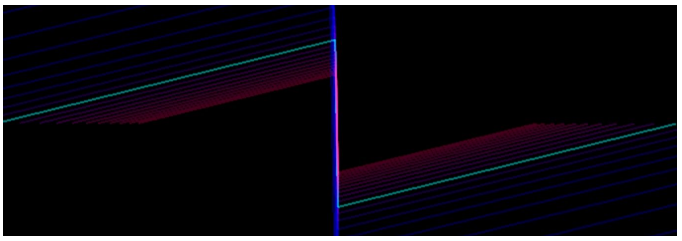
After clicking the previous exercises' links to the HTML 5 Player, notice how the first lines start with `//`. This double slash denotes a "comment", which is ignored by the compiler. So any characters in a line of code after `//` won't affect the result of our Bytebeat, which means you can use `//` to take out code snippets, without actually having to delete them. Of course, it also makes sense to add written comments or notes to your Bytebeats, explaining what a piece of code does, so that you keep an overview if things get more complex.

## *Understanding your first Bytebeat*

Alright, after messing around, let's see if we can actually figure out why a certain piece of code behaves like it does. Open one of the above-mentioned editors and let's start coding! The simplest, audible Bytebeat, as you already know, is the following:

```
t
```

(click here to listen to it in your web browser)



It's a sawtooth wave! Yay! … but why?!

The pre-defined variable `t` returns the time our Bytebeat is running. It is a timer, or counter, which starts at zero on initial playback and increases by 1 on every new audio frame. So at an 8 kHz sample rate, `t` increases by 8,000 every second and it just keeps going and growing. This alone doesn't create a sawtooth wave though, only a constantly rising number. To make it a sawtooth wave, we need to have a look at our 8 bit audio output and some binary basics.

"8 bit binary" means that there are eight digits, or bits, which can either be 0 or 1 and which together can represent numbers from 0 to 255. Each of the eight bits represents one specific number, from left to right:

```
128, 64, 32, 16, 8, 4, 2, 1
```

The numbers corresponding to the bits which are 1, are added together to create any number from 0 to 255.

Here are some examples:

```
00000000 = 0    (all bits 0)
00000001 = 1    (first bit = 1)
00000010 = 2    (second bit = 2)
00000011 = 3    (second and first bits = 2 + 1)
...
00010000 = 16   (fifth bit = 16)
00010001 = 17   (fifth and first bit = 16 + 1)
...
10000000 = 128  (eighth bit = 128)
10000001 = 129  (eighth and first bit = 128 + 1)
...
10001011 = 139  (eighth, fourth, second and first bit = 128 + 8 + 2 + 1)
...
11111111 = 255  (all eight bits = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1)
```

Remember that `t` is growing by 8,000 each second? That's way beyond our 0 to 255 range! So what happens after the first split second when `t` becomes greater than 255? Obviously, we can use greater numbers than 255 **inside** of our code, otherwise the whole `t` counter thing wouldn't work. It is only the **output** of our expression which gets truncated after 8 bits. Looking at this process in binary again should make it clear how the sawtooth wave is created.

Let's say, for example, we have 10 bits available, letting us count from 0 to 1023. So we have 10 digits, each 0 or 1, representing the following numbers:
`512, 256, 128, 64, 32, 16, 8, 4, 2, 1`

If we count up from zero, the 10 bits will fill up with ones from 0000000000 to 1111111111, from right to left:

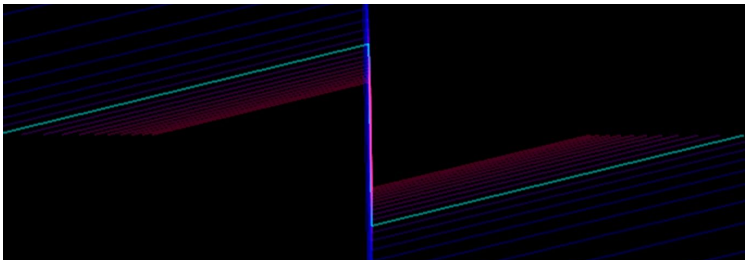
```
0000000000 = 0
0000000001 = 1
0000000010 = 2
0000000011 = 3
0000000100 = 4
0000000101 = 5
0000000110 = 6
0000000111 = 7
0000001000 = 8
0000001001 = 9
0000001010 = 10
0000001011 = 11
0000001100 = 12
0000001101 = 13
0000001110 = 14
0000001111 = 15
0000010000 = 16
0000010001 = 17
0000010010 = 18
0000010011 = 19
0000010100 = 20
...
0011111110 = 244
0011111111 = 255
```

This is also visualized in [this video].

As we said, we are using 10 bits for this example, so we have 10 digits available and can count further up to 1023. When our **output** is truncated after 8 bits though, see what happens when we count past 255, which is the 8 bit maximum value.

```
0011111111 = 255  (255 in 10 bit)
~~00~~11111111 = 255  (255 truncated after 8th bit, nothing changes)

0100000000 = 256  (256 in 10 bit)
~~01~~00000000 = 0    (256 truncated after 8th bit)

0100000001 = 257  (257 in 10 bit)
~~01~~00000001 = 1    (257 truncated after 8th bit)

0100000010 = 258  (258 in 10 bit)
~~01~~00000010 = 2    (258 truncated after 8th bit)
```

So our counter `t` can grow way beyond 255 internally in our expression, but the expression's result, processed through the 8 bit output, wraps around to 0 after 255, counts up to 255, then falls down to 0 again, counts up to 255, back to 0 and so on. It's a rising sawtooth!



Let's look at the chain of events inside our Bytebeat player once more, using the simple expression `t*50`:

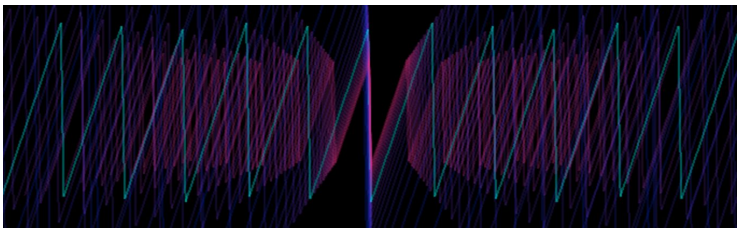
| INPUT | -> EXPRESSION | -> RESULT | -> OUTPUT TO 8 BIT |
|---|---|---|---|
| t=0 | t*50 | 0 | 0 (~~00~~00000000) |
| t=1 | t*50 | 50 | 50 (~~00~~00110010) |
| t=2 | t*50 | 100 | 100 (~~00~~01100100) |
| t=3 | t*50 | 150 | 150 (~~00~~10010110) |
| t=4 | t*50 | 200 | 200 (~~00~~11001000) |
| t=5 | t*50 | 250 | 250 (~~00~~11111010) |
| t=6 | t*50 | 300 | 44 (~~01~~00101100) |
| t=7 | t*50 | 350 | 94 (~~01~~01011110) |
| t=8 | t*50 | 400 | 144 (~~01~~10010000) |

How far can `t` grow? Up to 2,147,483,647, the maximum number of 32 bits, which seems to be the norm for Bytebeat players. At 8 kHz, i.e. 8,000 samples per second, this means that it can count up for about 74 hours. What happens then? I don't know … it either starts back from 0, crashes, or it opens up a portal to the Chiptune Dimension. Try it out :-P

## *Coding with mathematical operators*

Now, let's modify our sawtooth wave with the available mathematical operators:
`* / + - %`

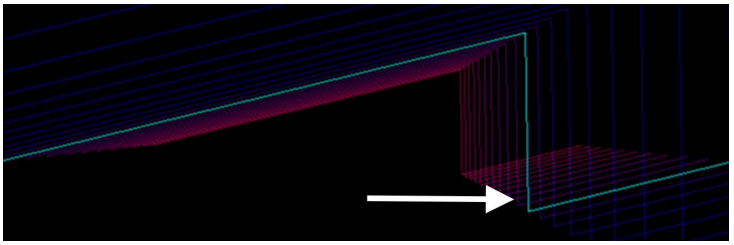
**Multiplying** `t` with a number makes `t` grow faster, increasing the sawtooth's frequency.
For example:
`t*12`



**Dividing** `t` by a number makes `t` grow slower, decreasing the sawtooth's frequency. This can go way below audio rate, into super slow LFO territory.
For example:
`t/30`

**Adding or subtracting** numbers will apply an offset to `t`, which by itself won't do anything sonically, because we're just constantly adding/subtracting **before** the 8 bit output, which wraps everything to a range of 0 to 255.

For example:
`t+200`



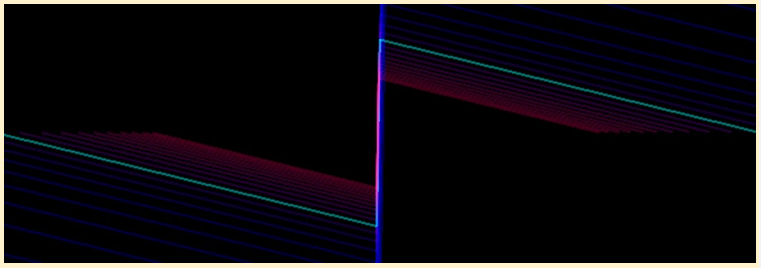
This does not have an audible effect by itself, but it does shift the phase, or position in time, of the waveform `t`, which becomes apparent when combining waves of different phases. We will try this out shortly!

Another use-case of the minus operator is to **invert** a value, the same as if you'd multiply it by -1.

**Exercise 03: Multiplication and Division**

Open your Bytebeat editor of choice and enter any of the code snippets above, playing with the multiplication and division of `t`.

Also try out:
`-t`
… the inversion of our rising sawtooth, i.e. a falling sawtooth.



**Modulo**. Ah! Now things get interesting! The modulo `%` returns the remainder after a division. What's a remainder? Let's do some simple math (that's all I can do, in fact):

```
15 / 4 = 3.75
```

This means that 4 fits into 15 three whole times, plus a little bit. That little bit is the remainder:

```
0.75 * 4 = 3
```

Or you could think of it this way, which I actually found easier to grasp at the beginning:

```
15 / 4 = 3.75
```

Leaving away the 0.75, let's only do:

```
3 * 4 = 12
```

… because we know that 4 fits into 15 three whole times. That makes 12, plus the remainder, which is the difference between 15 and 12, i.e. 3.

To get only the remainder 3, you would write the following:

```
15 % 4 = 3
```

That's … uhm … cool, I guess?! But what does it mean for my sick Bytebeat coding? I'm glad you asked! Let's look at some more modulo calculations and see if we can spot a pattern:

```
 4 % 4 = 0
 5 % 4 = 1
 6 % 4 = 2
 7 % 4 = 3
 8 % 4 = 0
 9 % 4 = 1
10 % 4 = 2
11 % 4 = 3
12 % 4 = 0
```

Notice something? That's like a tiny sawtooth wave! The modulo of *anything* divided by 4, is always 0, 1, 2 or 3! Or more generally speaking:

`x % y` is always between `0` and `y-1`

So let's code a tiny sawtooth Bytebeat and see what happens to our audio!

**Exercise 04: Modulo %**

Open your Bytebeat editor of choice and enter:
`t%128`

This will make the expression's result something between 0 and 127, which means we only use half of our 8 bit range from 0 to 255, reducing our sawtooth amplitude (or "volume") by half. Now try out different numbers instead of 128 and see what happens!

Well done! Have you noticed how the wave not only changes in amplitude, but also in pitch? That means that you can also use the modulo to change the frequency of a waveform by small amounts, not just large ones, like when you multiply by 2 or divide by 2 or more.

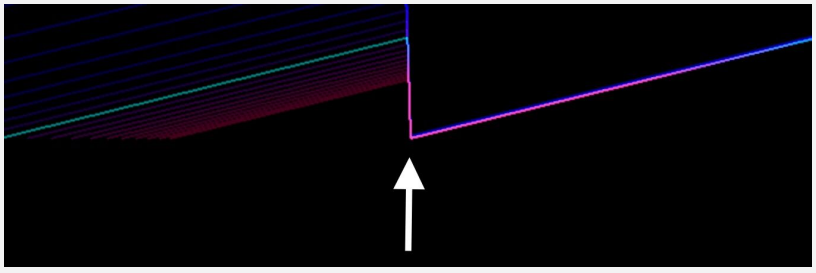
The modulo also acts just like our 8 bit output, i.e. it wraps the incoming values to a specific range (like 0 to 127 above), only that we can now decide that range for ourselves inside our expression. Of course our expression's result is still truncated to 8 bits at the end.

A while back, I told you we'd revisit **addition and subtraction**. Now is that time!

**Exercise 05: Addition and Subtraction**

Open your Bytebeat editor of choice and enter:
`t%128`

Yawn! Okay, okay! Enter something new:
`t%128+128`

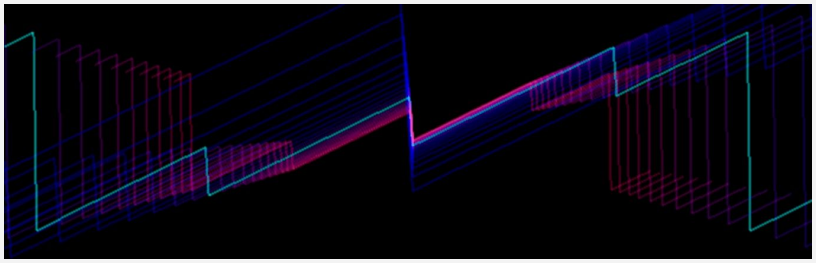


Now our sawtooth isn't going from 0 to 127 anymore, but from 128 to 255! This doesn't change its amplitude (still 127 steps), but it changes the values we're working with.

Try adding or subtracting other values and see what happens, especially when you offset the sawtooth to go partly below 0 or above 255.

Time for some phase shifting!
`t%255+t%64`

This adds two `t` waves of different frequencies and amplitudes together, resulting in a sound similar to the well known "Phaser" effect.



## *Coding with bitwise operators*

Working with mathematical operators was probably not much news to you, so let's try out our available bitwise operators, which perform tasks based on a number's binary representation.
`& | ^ << >>`

**AND &** compares the bit values of two binary numbers and returns 1 only if **both** bits are 1.

```
  010 1 = 5
& 001 1 = 3
= 000 1 = 1 = 5&3

  10 11 = 11
& 00 11 = 3
= 00 11 = 3 = 11&3

  0011 = 3
& 0100 = 4
= 0000 = 0 = 3&4
```

Yes, `AND &` only outputs 1 if both bits are 1. It's like **multiplying** both bits:

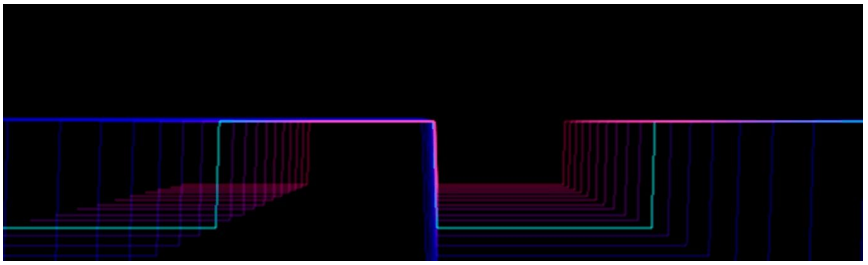
```
0 * 0 = 0
1 * 0 = 0
1 * 1 = 1
```

Now spot the pattern in those examples:

```
  11 1 1111100 = 1020
& 00 1 0000000 = 128
= 00 1 0000000 = 128

  1100001110 = 782
& 0010000000 = 128
= 0000000000 = 0

  10 1 1011101 = 733
& 00 1 0000000 = 128
= 00 1 0000000 = 128

  1001011010 = 602
& 0010000000 = 128
= 0000000000 = 0
```

So if we write something like `t&128`, we'll only get two alternating values: 128 and 0. What does that look and sound like?

Correct! A square wave. In fact, a square wave using only half of our available 8 bit range.



**Exercise 06: AND &**

Open your Bytebeat editor and enter:
`t&128`
It's a square wave! Replace the 128 with other values and see what happens.

Next, combine our sawtooth `t` with a much slower version of itself, enter:
`t&t/30`
That's some sweet ducking!

What happens when you invert that slow sawtooth wave using the minus operator in front of it?
`t&-t/30`

A curious property of an `AND &` operation is that the result cannot be larger than the smallest operand. For example:

`t&140`
… can never grow past 140, no matter how high `t` gets.

**OR |** compares the bit values of two binary numbers and returns 1 if **one or both** bits are 1. This operator is thusly called "inclusive OR" or sometimes "and/or".

```
  0101 = 5
| 0011 = 3
= 0111 = 8 = 5|3

  1011 = 11
| 0011 = 3
= 1011 = 11 = 11|3

  0011 = 3
| 0100 = 4
= 0111 = 8 = 3|4
```

So `OR |` just takes any 1 it gets and keeps it. It's like **adding** both bits together:

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = uhm ...
```

Okay, sorry, not entirely like **adding** the bits together ... I just wanted to make the point that you can use `OR |` to **add** waveforms together, almost like with a super lo-fi audio mixer.

**Exercise 07: OR |**

Open your Bytebeat editor of choice and enter:
`t`
then add:
`|t*4`
and then add:
`|t*12`

Use `OR |` to add more sawtooth waves of different frequencies (`t` multiplied or divided by a number).

A property of `OR |` that makes it behave so similar to an audio mixer, is that the results can never be smaller than the smallest operand (or "audio input" in our mixer analogy). So:

`t|140`
… will always result in a number of 140 or greater, no matter how low `t` is.

**XOR ^** is an "exclusive OR" which returns 1 on non-matching bits, i.e. if they are 1 and 0:

```
  0101 = 5
^ 0011 = 3
= 0110 = 6 = 5^3

  1011 = 11
^ 0011 = 3
= 1000 = 8 = 10^3

  0011 = 3
^ 0100 = 4
= 0110 = 6 = 3^4
```

Unlike our other two operators above, the result of an `XOR ^` operation is not limited to a minimum or maximum number.

**Exercise 08: AND &, OR |, XOR ^**

If you haven't already tried it out by yourself, go ahead and mix some waveforms using all of the above logical operators: `AND &`, `OR |` and `XOR ^`.

Here's a fun example to get you started. First, try to understand what kind of waveform each line of code generates (slow or fast? sawtooth or square?) and then listen to the output when you combine them using different logical operators.

```
t&128
| t&64
& t/20
^ t/15
| t*2
```

As already stated, the `OR |` operator provides the most similar results to a normal audio mixer. Although, since we're talking about Bytebeats, "similar" is still quite far off. Audibly, `AND &` works too in many cases, but not all, while `XOR ^` often breaks up the signal significantly. It really depends on your expression though, so never stop experimenting.

What?! You're still here? Okay, let's check out the remaining two bitwise operators then!

**Bitshift left << and Bitshift right >>** work a little different than the rest, as they don't compare two binary numbers. Instead the only look at the first operand in binary and shift its bits left or right, by the second operand's value.

```
12 << 3
0000 1100 = 12
  <---
0 1100 000 = 96
```

All bits are simply shifted 3 places to the left; new bits are 0. This equals the following:

$12 * 2^3 = 96$

or

$12 * 2 * 2 * 2 = 96$

Now the other way around, bitshift right:

```
12 >> 3
00001100 = 12
  --->
00000001 = 1
```

Again, all bits are shifted 3 places, this time to the right. This equals the following:

$12 / 2^3 = 1$
or
$12 / (2 * 2 * 2) = 1$

So why not simply use regular, mathematical multiplication and division operators? Bitshifts can be processed faster by the computer, which might create different results, depending on the complexity of your code … however, that fact probably never actually applies to the rather simple Bytebeats. But there's another neat thing about using bitshifts in our Bytebeats, as we will find out now.

**Exercise 09: Bitshifts << >>**

Shift those Bitz and pay attention to the audio. How does `t`'s frequency change, when shifting it by 1, 2, 3, etc.?

```
t<<1
t<<2
t<<3

t>>1
t>>2
t>>3
```

Shifting `t` by 1, makes its frequency go up or down one octave! Good to know, when you want to code traditionally western, musical Bytebeats!

## *Coding with relational operators*

One more category to go! Relational operators find out whether a certain relation between two numbers is "true" or "false" and output 1 or 0 accordingly:

`2 > 1 = 1` ("two is **greater than** one is true")
`2 < 1 = 0` ("two is **less great than** one is false")
`2 == 1 = 0` ("two is **equal to** one is false")
`2 != 1 = 0` ("two is **not equal to** one is true")
`2 >= 2 = 1` ("two is **greater than or equal to** two is true")
`2 <= 2 = 0` ("two is **greater than or equal to** two is true")

We can use relational operators to create on/off switches for parts of our code, which takes our Bytebeats to another level!

**Exercise 10: Relational Operators**

Stop your editor's playback and make sure that `t` is reset to zero (in Greggman's HTML 5 Bytebeat Player, click on the time value between the ? and the play button). Enter all of the following code and press play.

```
(t>8000)*t
|(t>16000)*t*2
|(t>24000)*t*6
|(t>32000)*t*12
|(t>40000)*t*40
|(t>48000)*t/20
```

After six seconds you can press stop, reset `t` to zero, and start again, if you like. Remember that the output of a relational operation is 1 (true) or 0 (false)? By multiplying this output with part of our expression, we can switch that part on or off, because:

```
0 * x = 0
1 * x = x
```

In the above example, we get 8,000 samples of silence, which at 8 kHz is one second. Then, when `t > 8000` is true, the result of the first line of code reads:

```
(1)*t
```

Our sawtooth wave starts playing! One second later, at `t > 16000`, we get:

```
(1)*t
|(1)*t*2
```

… a higher pitched sawtooth wave is multiplied by 1, i.e. it is added to our output via the `OR |` operator. Now go ahead and experiment with the other relational operators and as usual, save fun code snippets to your Bytebeat diary!

## *Putting it all together*

In this chapter, I would like to show you step-by-step examples of applying the various concepts we've learned in one Bytebeat.

**Coding a Step Sequencer**
Let's build a traditional 16-step sequencer, which loops sixteen sounds indefinitely. For this, we need a looping counter, indicating the different step positions in our sequence. We have already worked with such a revolving counter. Can you remember?

Look at this piece of code:
`t%16`

The **modulo %** operator! In this case, *anything* % 16 will result in values between 0 and 15, i.e. 16 steps. How cool is that? Of course, at the 8 kHz sample rate at which `t` is counting upwards, those 16 steps will be very short, so let's make them longer:
`t%16000`

Now every one of our 16 steps can be 1,000 audio frames long, or 1/8th of a second, at 8 kHz. That should be enough.

Next, we need to define the actual steps:
*Step 1 should last from 0 to 999*
*Step 2 from 1,000 to 1,999*
*Step 3 from 2,000 to 2,999*
*…*
*Step 15 from 14,000 to 14,999*
*Step 16 from 15,000 to 15,999*

How could we possibly implement this? Exactly: relational operators.
`(t%16000 >= 0  &  t%16000 < 1000)`

The above code will output 1 (true) as long as …
*… the step counter* `t%16000` *is* ***greater than or equal to*** *0*
***AND***
*… the step counter is* ***less than*** *1000*

So from 0 to 999, this will output 1 (true). Of course, this by itself won't make a sound, but we can multiply the result (true or false, 1 or 0) with trusty old `t`:
`(t%16000 >= 0  &  t%16000 < 1000) * t`

Try the above line in your Bytebeat player and then, add a second line with `OR |` for step 2, from 1000 to 1999, which could play a higher note, like `t*3`:
`| (t%16000 >= 1000  &  t%16000 < 2000) * t*3`

… and then add more lines, for the remaining 14 steps in our sequence.

Look at that! You turn the page and new code appears! Isn't that great? Here's the complete sequencer code for all 16 steps, playing a simple sawtooth melody:

```
(t%16000  >= 0     &   t%16000 < 1000)  * t
|(t%16000 >= 1000  &   t%16000 < 2000)  * t*3
|(t%16000 >= 2000  &   t%16000 < 3000)  * t*5
|(t%16000 >= 3000  &   t%16000 < 4000)  * t*8
|(t%16000 >= 4000  &   t%16000 < 5000)  * t*2
|(t%16000 >= 5000  &   t%16000 < 6000)  * t
|(t%16000 >= 6000  &   t%16000 < 7000)  * t*12
|(t%16000 >= 7000  &   t%16000 < 8000)  * t*8
|(t%16000 >= 8000  &   t%16000 < 9000)  * t*3
|(t%16000 >= 9000  &   t%16000 < 10000) * t
|(t%16000 >= 10000 &   t%16000 < 11000) * t*4
|(t%16000 >= 11000 &   t%16000 < 12000) * t*10
|(t%16000 >= 12000 &   t%16000 < 13000) * t*7
|(t%16000 >= 13000 &   t%16000 < 14000) * t*6
|(t%16000 >= 14000 &   t%16000 < 15000) * t*5
|(t%16000 >= 15000 &   t%16000 < 16000) * t*6
```

The `t` sawtooth sound is seriously getting on my nerves by now, so please, let's change it into a square wave! Lucky for us, that's easily done. Put the sequencer code into parenthesis and add `&128` afterwards:

```
(
sequencer code
)
&128
```

Revisit the chapter on bitwise operators, if you don't remember why `&128` creates a square wave.

You know, on second thought, it's not just the sawtooth wave that I got tired of. That repeating melody could use some variation too. How about we transpose it up one octave every other run-through?

What we need for this:

- code for the transposition
- a counter and relational operation that switches the transposition on and off

Can you come up with the required expression? A possible solution is on the next page.

Here's the new line of code we need:
`<< (t%32000 <= 16000)`

To transpose our sequencer output up by one full octave, we can simply bitshift it to the left by 1. To switch the bitshifting on and off every other playthrough, we add a relational operation, with a counter twice a high as our sequencer's, i.e. `t%32000`. Then, we ask if the counter is equal to or greater than 16,000 and if true, we know we're on a second run-through and get 1 from the operation. In this case the result of our above expression would read:

`<< (1)`
… bitshift left by 1 (true), which is a transposition up one octave.

To make it work properly, we have to insert that line of code between the sequencer and our square wave conversion:

```
(
sequencer code
)
<< (t%32000 <= 16000)
&128
```

Alright, this has been a very deterministic approach to building our step sequencer program, without any trial and error or randomly written elements. Go ahead and tweak the code to your liking now. For example, you could change each of the 16 steps' sounds from a sawtooth wave to something completely wild and different, you could also change the length for which each sound plays (right now it's 1,000 audio frames), or you could add some code around the sequencer to completely warp its output.

**Tip: Line Breaks**

While one-liners look impressive, they often don't provide a good overview of what's actually going on in the code. Using line breaks and `//` comments helps to find your way easily and change parts quickly.

Cool, but inefficient:
`t*30&(t%3000<1500)*t/10|-t>>3*t/100`

Uncool, but splendidly efficient (it's a joy!):

```
t*30                //super high pitched sawtooth
&
(t%3000<1500)*t/10 // rhythmically breaking up the above sawtooth
|
-t>>3*t/100         // bitshifting everything to hell
```

**Coding a Square Wave with PWM**

Square waves sound even nicer with pulse-width modulation applied, a very common effect found on most regular synthesizers. If you don't know about PWM yet, check out this video.

Let's see how we can implement this effect in our Bytebeat. First, we need to write a square wave expression that actually gives us control over the time that the wave is high and low.

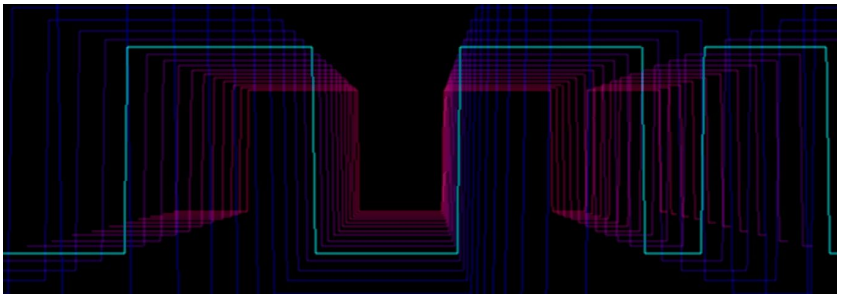
Here's a possible solution:

```
(t%100 >= 50)*255
```

There is a counter `t%100`, counting from 0-99. We then check if its output is equal to or greater than 50. If this is true, we get 1, which we multiply by the amplitude (or "volume" level) we want our square wave to have, something between 0 (silence) and 255 (full volume). The above code gives us a 50% pulse-width, full-volume square wave. To alter the pulse-width, we have to change the `50` to something else, between 0 and 100. Try it!

To get the pulse-width moving, we can write the following:

```
(t%100 >= t/200%100)*255
```



Here, the `50` was exchanged for `t/200%100`, another counter, counting from 0 to 99 again. However this one is not counting at an audio-rate speed of `t`, but way slower at `t/200`. Play around with this speed value and also decrease the counter's maximum value below 100, to only apply the pulse-width modulation across part of the wave's duty cycle.

**Coding a Triangle Wave with Wave Folding**

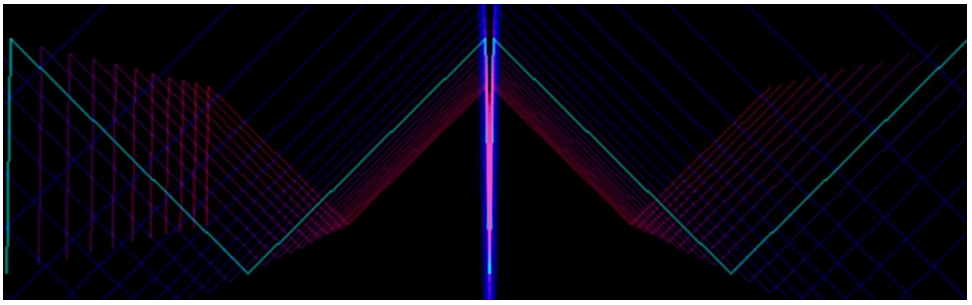
So far, we've only worked with very "harsh" sounding waves. Let us now code a triangle wave.

Think about how we could approach this. We know that `t` provides a rising sawtooth and `-t` an inversion of that, i.e. a falling sawtooth. If we switched between `t` and `-t` after each cycle, we'd get a triangle wave. How long is one cycle of our sawtooth waves? 256, because after that, the output wraps around and starts a new sawtooth. So for starters, we need to switch between the two waveforms every 256 audio frames.

An expression for this could be:

```
t   * (t%512 >= 256)
|
-t * (t%512 <  256)
```

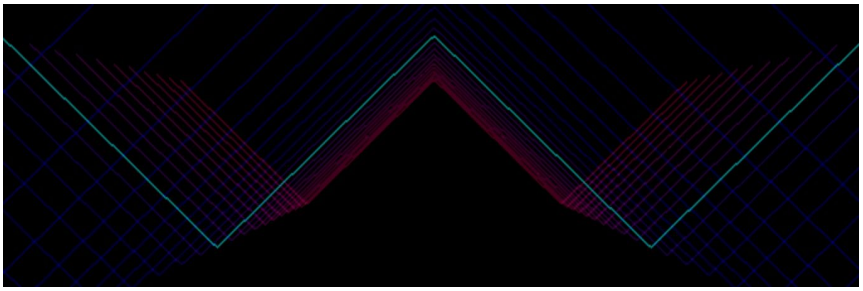
As long as `t%512` is equal to or greater than 256, play `t` and while `t%512` is less than 256, play `-t`. This will alternate between the rising and falling sawtooths. Try it out and see what happens!



It almost looks correct, but there is one thing that messes up our smooth triangle wave. What is it and how can we fix it? Try to find the solution yourself, before swiping to the next page.

The solution:

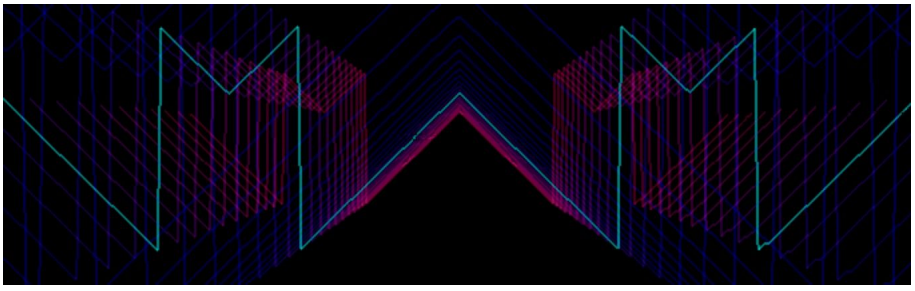
```
t       * (t%512 >= 256)
|
(-t-1)  * (t%512 <  256)
```



Before, on the previous page, both sawtooth waves would always start their cycle at 0. This meant that the rising `t` wave would start 0, 1, 2, 3, 4 and so on, until 255, after which the falling `-t` wave would take over, starting 0, 255, 254, 253, etc. This resulted in the super short, but sonically very significant, drop down to 0, when both waves met at the top. Subtracting 1 from `-t` achieves a tiny offset, or phase shift, between the waves, so that `-t` takes over when it is at 255 and not at 0.

Next, let's try some wave folding, by putting parentheses around the triangle wave code and then moving the whole thing upwards continuously.

```
(
t*4*(t*4%512 >= 256)
|
(-t*4-1)*(t*4%512 < 256)
)
+t/60
```



Outside of the parentheses, we added `+t/60` which shifts up the triangle wave very slowly and continuously. As soon as the triangle wave wraps around from 255 back to 0, it is cut up, resulting in harsher, pulsating sounds.

Also note how we multiplied all of the `t` waves inside the parentheses by 4, to increase the triangle wave's pitch to something more audible.

## *Breaking the chains*

Until now, we've stayed well inside the commonly accepted, yet arbitrarily defined, constraints of Bytebeat coding. Let's take a quick peek over the fence!

Sometimes you might want to alter parts of your Bytebeat code while it is being executed without typing on your keyboard, recompiling the code and restarting playback; for example during a live performance. A simple way to do this, is by using other variables apart from `t`. Kymatica's BitWiz iOS App, for example, offers a multi-touch XY performance pad, as well as the ability to route external MIDI signals to variables in your code. So you could set it up that the variable `a` receives CC data from a connected MIDI keyboard's modulation wheel.

```
t*a
```

... would then allow you to change the pitch of the `t` sawtooth wave by turning the wheel, while your Bytebeat is running. Of course that is just the tip of the iceberg and you can create much more complex, yet still playable, Beats using variables (think back to the relational operators or "on/off switches").

Another limitation we could ignore is "one expression only". In Single Cell's Caustic 8BitSynth, you can write two separate expressions, which run at the same time, letting you blend between them and even automate this. Other players will allow you to write several expressions into a single text field, separated by commas, with only the last one being sent to the audio output. This allows you to declare variables and do other things outside of your sound expression.

```
s=6,
(t*s)*(t*s%512>256)
|
(-t*s-1)*(t*s%512<256)
```

That's the triangle wave from the previous page, but instead of the value 4, we have the variable `s`, for "speed" or the wave's frequency. First, `s` is declared as the number 6, after which we have a comma and then the second and last expression which will be played through the speakers. To change the pitch of the wave now, you only need to alter `s` and not type the new value four times in various places in your code.

And finally, certain Bytebeat programs, like Greggman's HTML 5 Player, might accept more operators than the basic ones we discussed in this guide, for example `sin`, `cos`, `tan`, `sqrt`, etc.

## *Sources & further reading*

That's it for this guide. Thank you very much for checking it out! I hope you enjoyed learning about Bytebeat coding. If you have comments or suggestions regarding this PDF, please let me know :-)

Of course there is more on this topic out there, so here's a collection of websites you might find interesting, some of which have also been linked earlier.

Content by Ville-Matias "viznut" Heikkila, who started the whole Bytebeat thing in 2011:
http://viznut.fi/texts-en/bytebeat_exploring_space.pdf
http://viznut.fi/texts-en/bytebeat_algorithmic_symphonies.html
http://viznut.fi/texts-en/bytebeat_deep_analysis.html
https://countercomplex.blogspot.com/2011/10/some-deep-analysis-of-one-line-music.html

Article with a bunch of other links:
http://canonical.org/~kragen/bytebeat/

Reddit post with lots of cool Bytebeat code snippets:
https://www.reddit.com/r/bytebeat/comments/20km9l/cool_equations/?st=jm4me4ki&sh=d5e70bef

Bytebeat software:
http://coleingraham.com/2013/04/28/bytebeat-shell-script/
https://github.com/greggman/html5bytebeat
https://damikyu.itch.io/evaluator
http://kymatica.com/Software/BitWiz
http://www.singlecellsoftware.com/caustic

Various Wikipedia articles:
https://en.wikipedia.org/wiki/Low-complexity_art
https://en.wikipedia.org/wiki/Demoscene
https://en.wikipedia.org/wiki/Binary_number
https://en.wikipedia.org/wiki/Bitwise_operations_in_C

More from The Tuesday Night Machines:
YouTube:
https://nightmachines.tv/youtube
Modular Synth Basics:
https://nightmachines.tv/modularbasics
My Synthesizer Wardrobe:
https://nightmachines.tv/ikea-pax
Like my free content? Here are a few ways to support TTNM:
https://nightmachines.tv/support

## *Version history*

2018-09-17 v1.0:
- initial release for comments on the AE Modular forum: http://forum.aemodular.com

2018-09-18 v1.1:
- spelling and mark-up corrections
- additional content and editing (thanks to forum users *thetechnobear* and *careck*)

2018-09-22 v1.2:
- minor editing
- added screenshots from BitWiz to illustrate certain waveforms

2018-10-05 v1.3:
- added QR Code
- minor spelling corrections

2019-10-19 v1.4:
- moved PDF hosting to GitHub
- added new QR Code

Get the latest version of this PDF here:
https://github.com/TuesdayNightMachines/Bytebeats