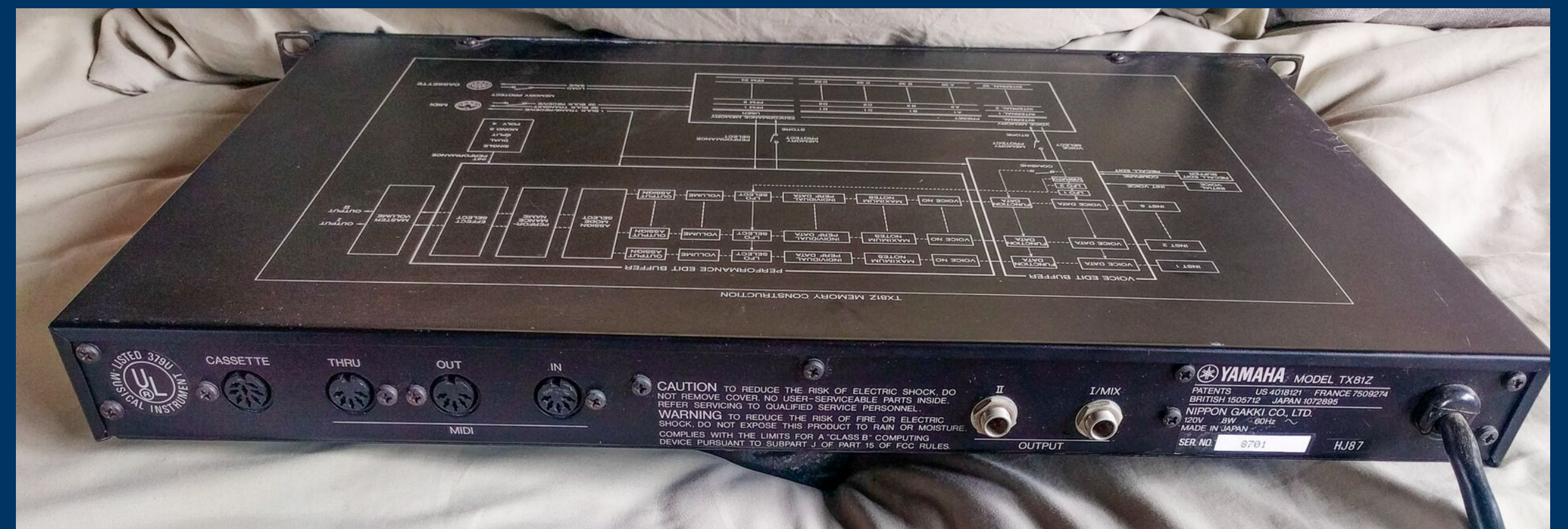
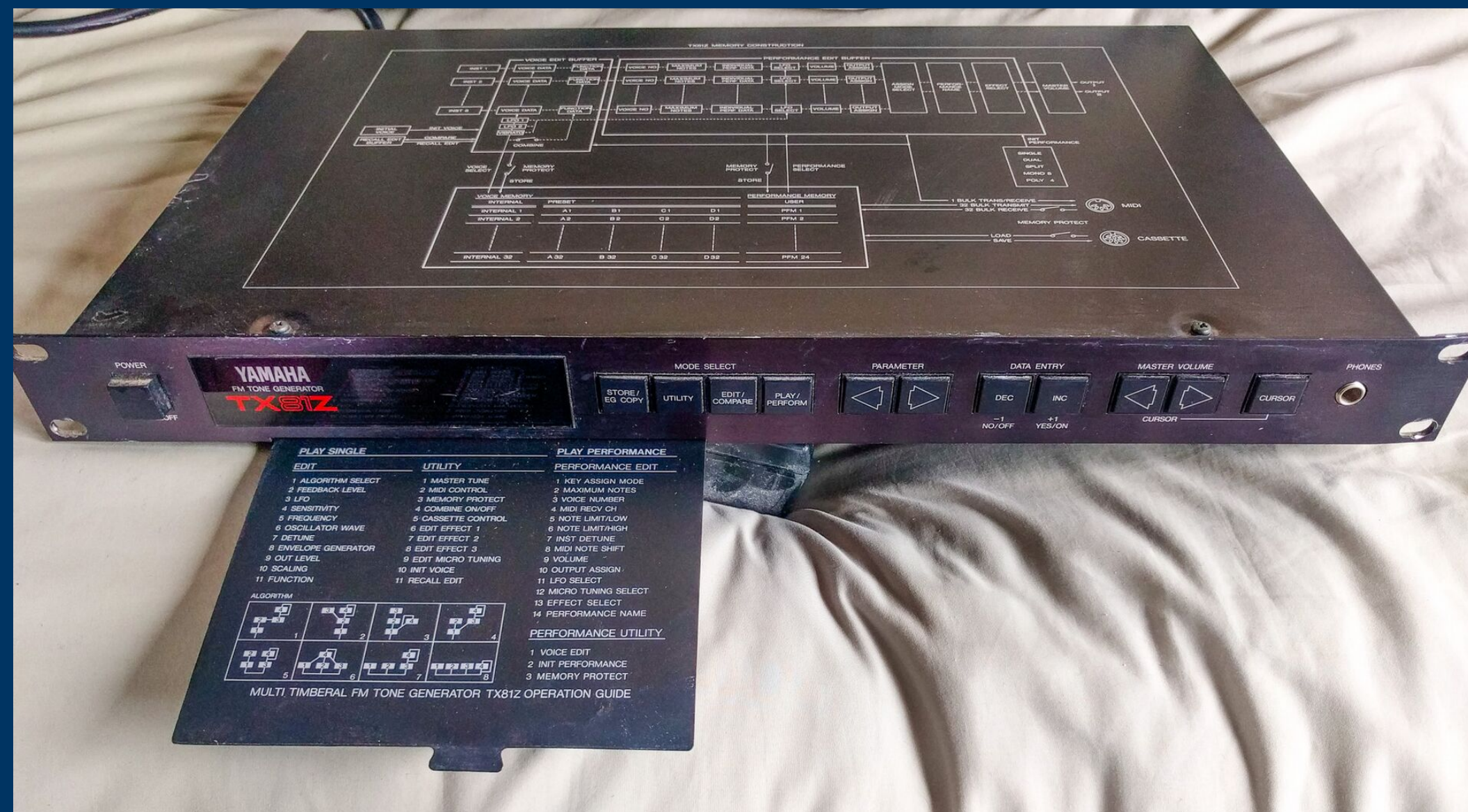


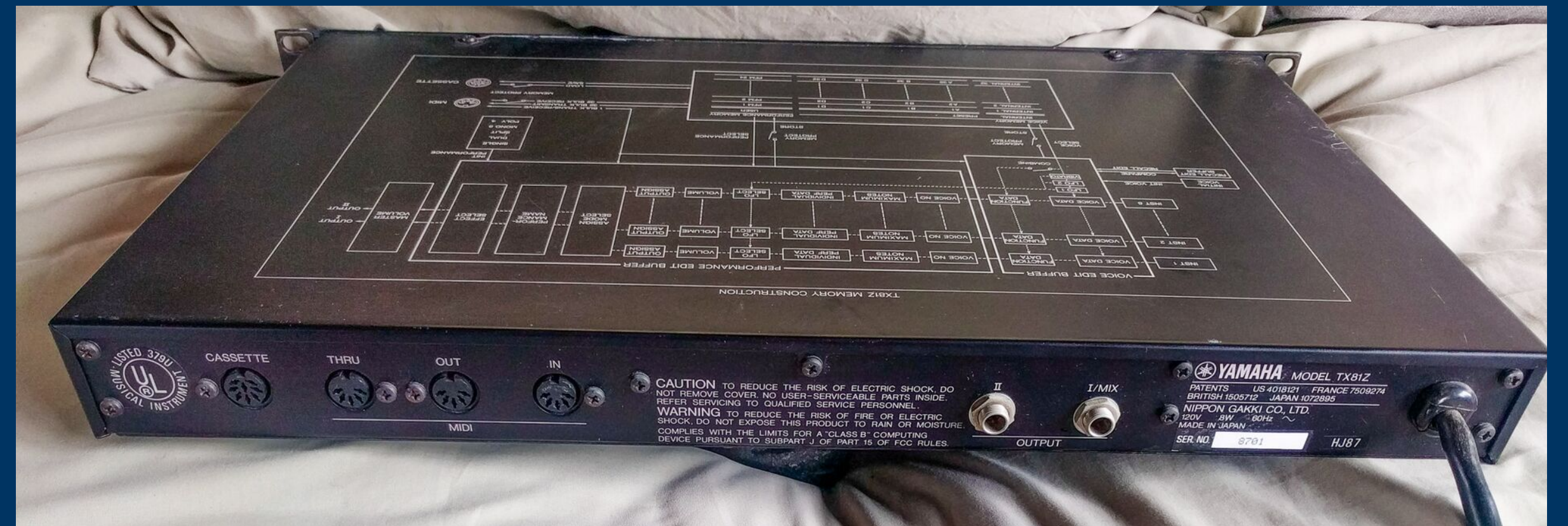
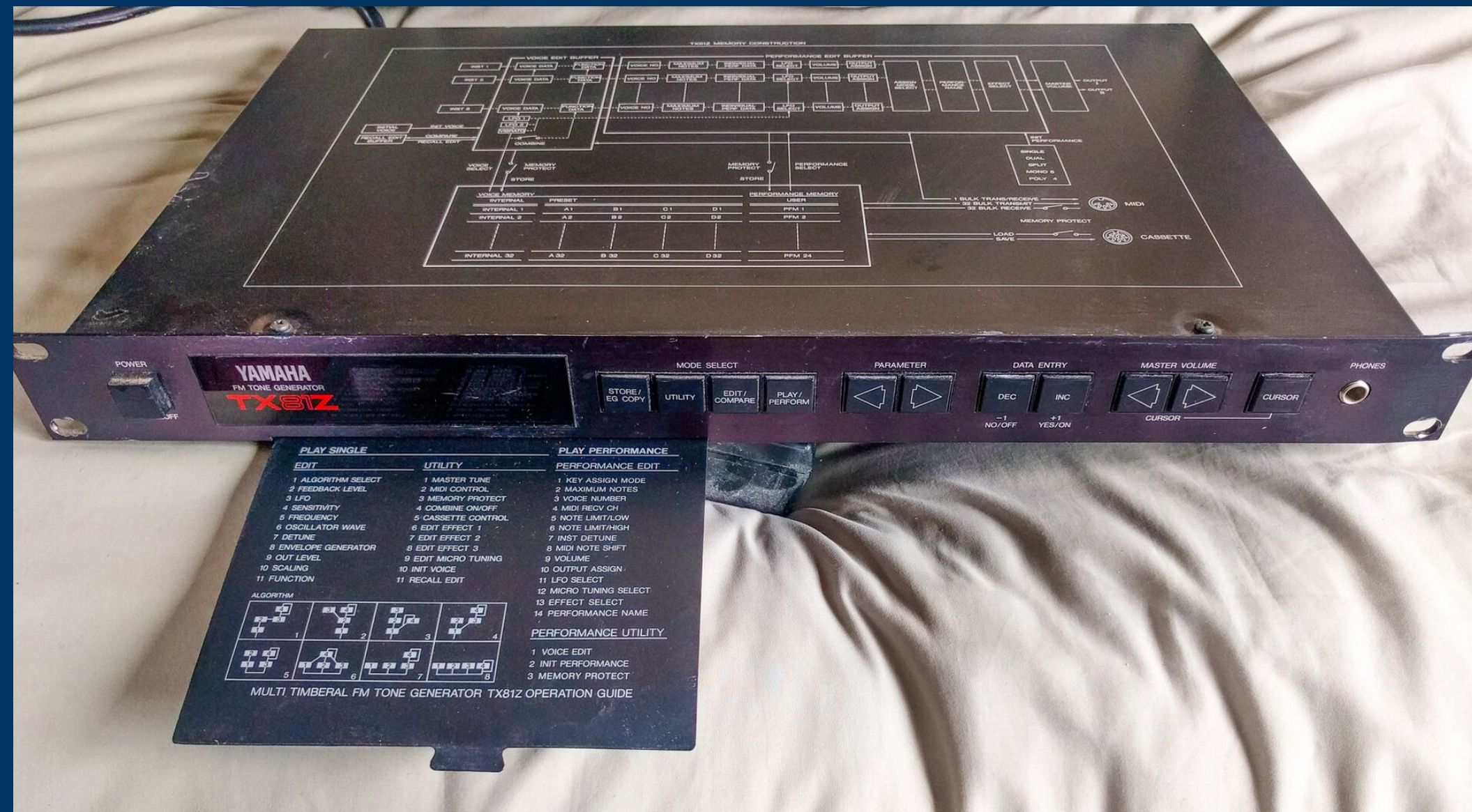
Introducing the Yamaha TX81Z

- Introduced in 1987
- A 4 operator FM synth, supporting multiple oscillator waveforms, with 8 algorithms
- 8 voices, 8 voice polyphony
- Stereo output



Introducing the Yamaha TX81Z

- Totally digital design - no analog filters
- Supports midi



Introducing the Yamaha TX81Z

Famous for the 'Lately Bass' preset which has appeared on countless recordings

The screenshot shows a YouTube playlist interface. On the left, a large video thumbnail displays the Yamaha TX81Z FM Tone Generator screen with the 'Lately Bass' preset selected. Below the thumbnail, the playlist title 'Lately Bass' is shown, along with the creator 'Tom Hall', 29 videos, and 2,612 views. A description states that the preset is used in each release in the playlist. A 'Play all' button and sharing options are visible.

The main content area lists seven tracks:

- 1** Toni Scott - That's How I'm Living (1988) (MTV)
90srave • 1.6K views • 7 years ago
- 2** The System - You're In My System (Atmospheric Dub) [Ibadan] (1998)
FAUX CODA • 205K views • 9 years ago
- 3** Haddaway - What Is Love [Official 4K]
CoconutMusicGermany • 344M views • 10 years ago
- 4** Making a 90s Banger with the Yamaha TX81Z!
Alex Ball • 192K views • 4 years ago
- 5** Satisfaxion. The volume. The Solid Collective. Can You Feel It
juandibreakbeat • 127K views • 12 years ago
- 6** Fluke - Atom Bomb
PostEchoCorp • 98K views • 14 years ago
- 7** Orbital - Halcyon On and On
RoundxSeal • 21M views • 15 years ago

Introducing the Yamaha TX81Z

When buying vintage gear, check the mains voltage...

[Home](#) > [Yamaha](#) > [Yamaha Keyboards and Synths](#) > [Yamaha TX81Z Rackmount FM Tone Generator 1987 - 1988](#)



Yamaha TX81Z Rackmount FM Tone Generator

Used - Excellent [Top product](#)

£200

Free delivery

Klarna

Make 3 payments of £66.66. [Learn more](#)
18+, T&C apply, Credit subject to status.

Free Shipping from Southport, United Kingdom

Buy it now

Add to Basket

Make an offer

Watch

Listed: 2 months ago Views: 104 Watchers: 6 Offers: 0



Dan's Gear

Southport, United Kingdom

Choosing an emulation strategy

So we want to emulate it, let's sample it - easy right?

For each preset we would require:

- Multiple samples per octave
- Loop points per sample to allow sounds to sustain
- Multiple velocity layers
- No support for aftertouch, modulation sources etc etc

Choosing an emulation strategy

So instead, how about dismantling the instrument, decapping the chips, taking photos of them, reverse engineering the processor gates, and building an FPGA model?

Choosing an emulation strategy

<https://www.righto.com/2021/11/reverse-engineering-yamaha-dx7.html>

Ken Shirriff's blog

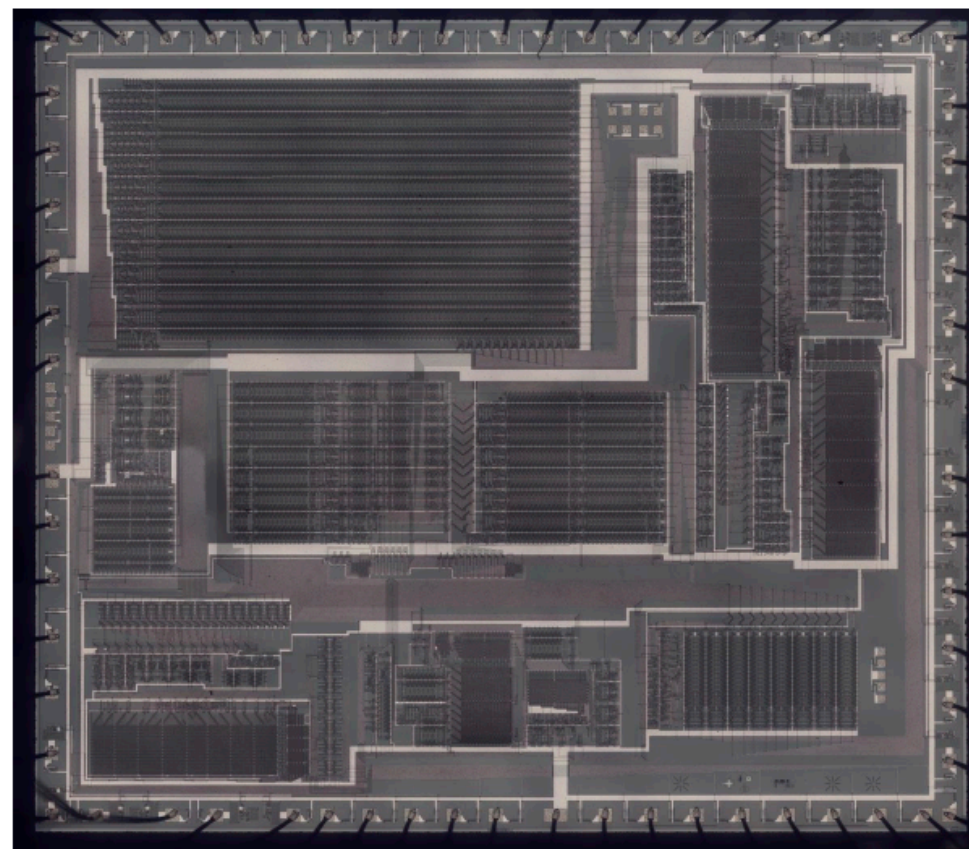
Computer history, restoring vintage computers, IC reverse engineering, and whatever

Reverse-engineering the Yamaha DX7 synthesizer's sound chip from die photos

The Yamaha DX7 digital synthesizer was released in 1983 and became "one of the most important advances in the history of modern popular music"¹. It defined the sound of 1980s pop music, used by bands from A-ha and Michael Jackson to Dolly Parton and Whitney Houston. The DX7's electric piano sound can be heard in over 40% of 1986's top hits.² Compared to earlier synthesizers, the DX7 was compact, inexpensive, easy to use, and provided a new soundscape.³

While digital synthesis is straightforward nowadays, microprocessors⁴ weren't fast enough to do this in the early 1980s. Instead, the DX7 used two custom chips: the YM21290 EGS "envelope" chip generated frequency and envelope data, which it fed to the YM21280⁵ OPS "operator" chip that generated the sound waveforms. In this blog post, I investigate the operator chip and how it digitally produced sounds using a technique called FM synthesis.^{6 21}

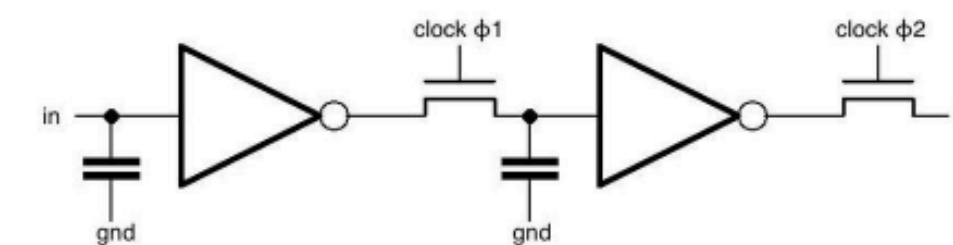
I created the high-resolution die photo below by compositing over a hundred microscope photos.⁶ Around the edges, you can see the 64 bond wires attached to pads; these connect the silicon die to the chip's 64 pins. The chip has one layer of metal, visible as the whitish lines on top. (Power and ground are the thick metal lines.) Underneath the metal, the polysilicon wiring layer appears reddish or greenish. Finally, the underlying silicon is grayish. The overall layout of the chip is dense rectangles of circuitry with the space between them used for signal routing. I will discuss these circuitry blocks in detail below.



Die photo of the DX7's YM21280 Operator chip. Click this photo (or any other) for a magnified version.

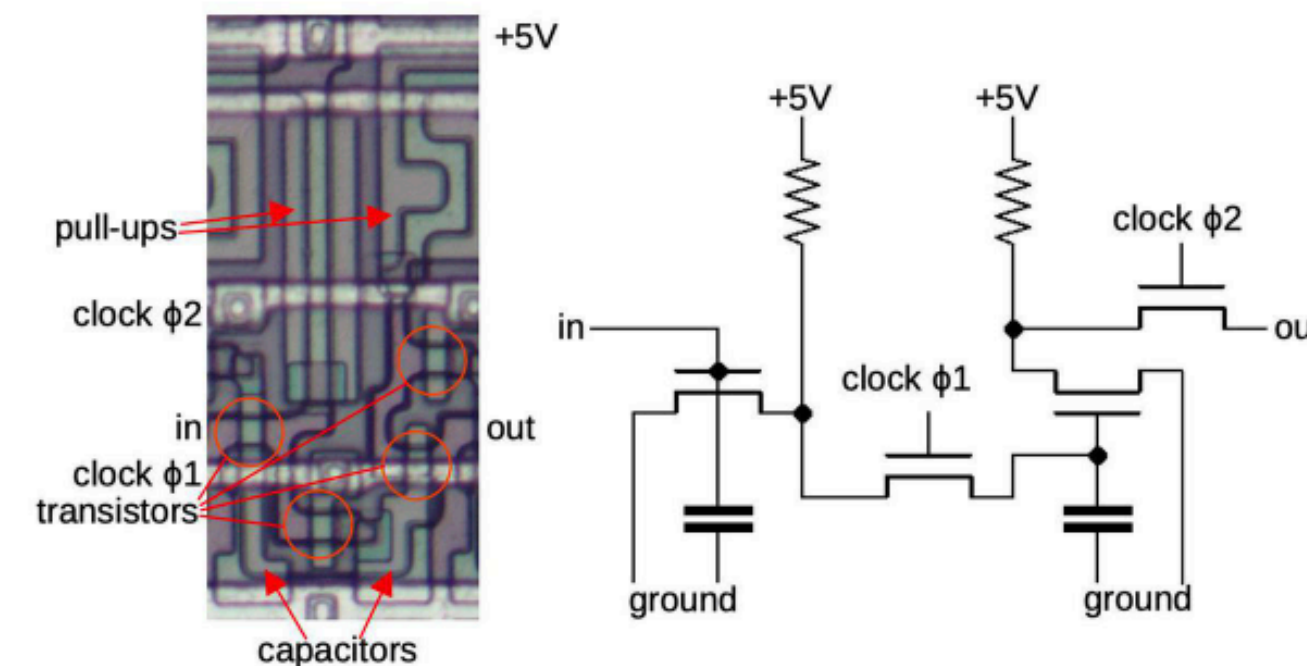
The photo below shows the integrated circuit with the metal lid removed, showing the silicon die inside. The pins have been flattened in the photo; they are normally bent downwards, but in a staggered pattern.⁷ The four rows of pins make this a **quad in-line package**, with twice the pin density as a regular DIP chip. As a result, this 64-pin chip has a smaller package than a standard 40-pin DIP chip.

The schematic below shows how one stage of the shift register is implemented. The chip uses a two-phase clock. In the first phase, clock $\phi 1$ goes high, turning on the first transistor. The input signal goes through the inverter, through the transistor, and the voltage is stored in the capacitor. In the second phase, clock $\phi 2$ goes high, turning on the second transistor. The value stored in the capacitor goes through the second inverter, through the second transistor, and to the output, where it enters the next shift register stage. Thus, in one clock cycle ($\phi 1$ and then $\phi 2$), the input bit is transferred to the output. (The circuit is similar to dynamic RAM in the sense that bits are stored in capacitors. The clock needs to cycle before the charge on the capacitor drains away and data is lost. The inverters amplify and regenerate the bit at each stage.)



Schematic of one stage of the shift register.

The diagram below shows the physical implementation of one shift register stage. It's a bit confusing because there are three layers: the whitish metal on top, doped silicon regions on the bottom (which appear outlined in black), and polysilicon lines in the middle (which appear reddish or greenish). Transistors are formed when a polysilicon line crosses doped silicon. A capacitor is created similarly, with a polysilicon line and doped silicon forming the two plates of the capacitor. An inverter is created from a transistor that pulls the output to ground, along with a pull-up resistor. (The pull-up resistor is actually another transistor, specially doped to make it a **depletion** transistor.)



Implementation of one bit of the shift register. This matches the earlier schematic, but shows the components of the inverters.

Choosing an emulation strategy

Ok, so instead, let's build a software emulation. What's the approach?

1. Build a simplified conceptual model of the architecture using resources such as manuals and tinkering with the instrument
2. Determine which parts are relevant for the sounds you want to recreate - there's no point finding out about bits you don't need!
3. For the relevant parts, work out how to exercise each to build an understanding of how they work and behave
4. Implement each of the model components - divide and conquer

TX81Z architecture

TL;DR The manual is aimed at musicians, not engineers, but does at least list all of the parameters, and what they are for

HOW DOES THE TX81Z WORK ?

The TX81Z has two main modes. Each main mode has three "sub-modes".

SINGLE

← →

Press twice, remains lit.

PERFORMANCE

PLAY (Single)

Select and play any voice using chords of up to 8 notes (p.11).

EDIT (Single)

Create your own voices or modify an existing voice (p.12).

UTILITY (Single)

*Save and load data (p.28).
 *Set microtone tables (p.31).
 *Set program change table (p.26).
 *Set pan, delay and chord effects (p.29).
 *And other useful functions.

PLAY (Performance)

The TX81Z acts as up to 8 independent instruments as specified in the Performance Memory that you select (p.35).

EDIT (Performance)

Change the settings of a Performance Memory (p.37).

UTILITY (Performance)

*Set a Performance to a basic setting (p.42).
 *And other useful functions.

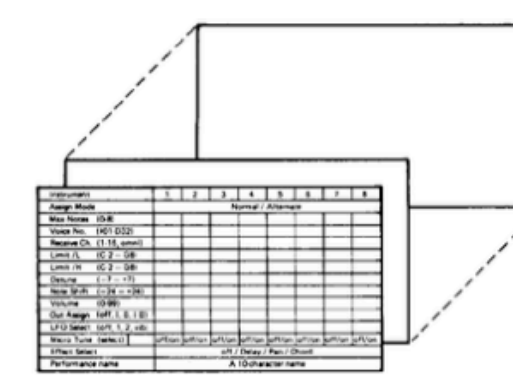
Here are the main memory areas inside the TX81Z.

Voice Memory (p.11)

There are 5 voice memory banks, each with 32 voices. Banks A-D are preset, and cannot be changed. Bank I is for you to store your own voices in.

Performance Memory (p.35)

Each performance memory can set the TX81Z to act as up to 8 independent instruments, each controlled on a different channel.



Effect Memory (p.29)

Each performance can use one of the three effects.

Effect 1 (Delay)

Delay Time

Push Shift

Feedback

Effect Level

Effect 2 (Pan)

Direction

Pan Speed

Effect 3 (Chord)

Key on note	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
Chord note												

Program Change Table (p.26)

Incoming program change messages can select anything you want; voices or performance memories.

Program Change Table	
Incoming	Selected
1	B19
2	PF24
3	I07
...	
127	A32
128	C14

Microtone Tables (p.31)

You can use non-standard scales. 11 scales are preset. The Octave and Full Settings are user programmable.

1/8 Tone

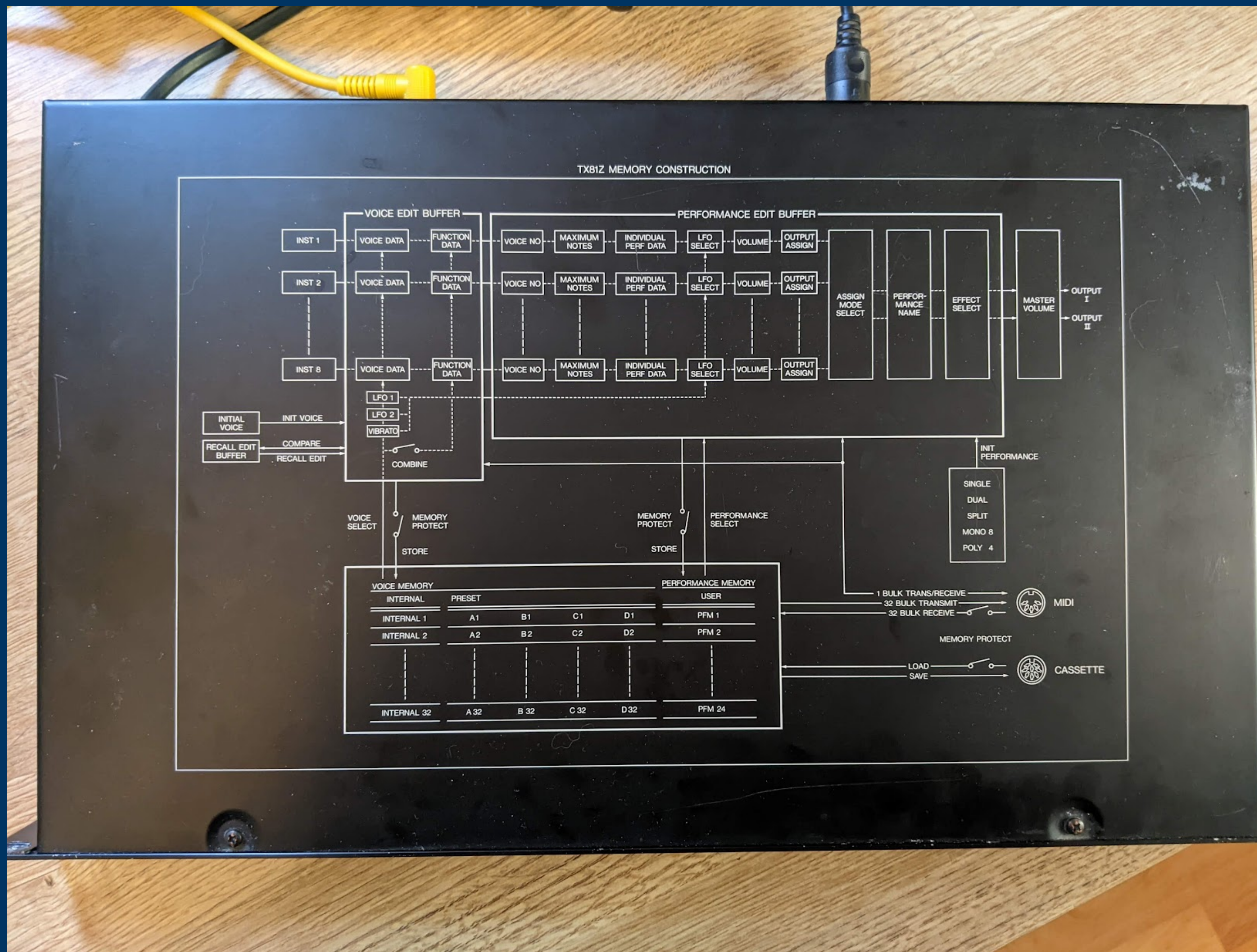
User Full	
C-2	
C#-2	
...	
G#	

1/4 Tone

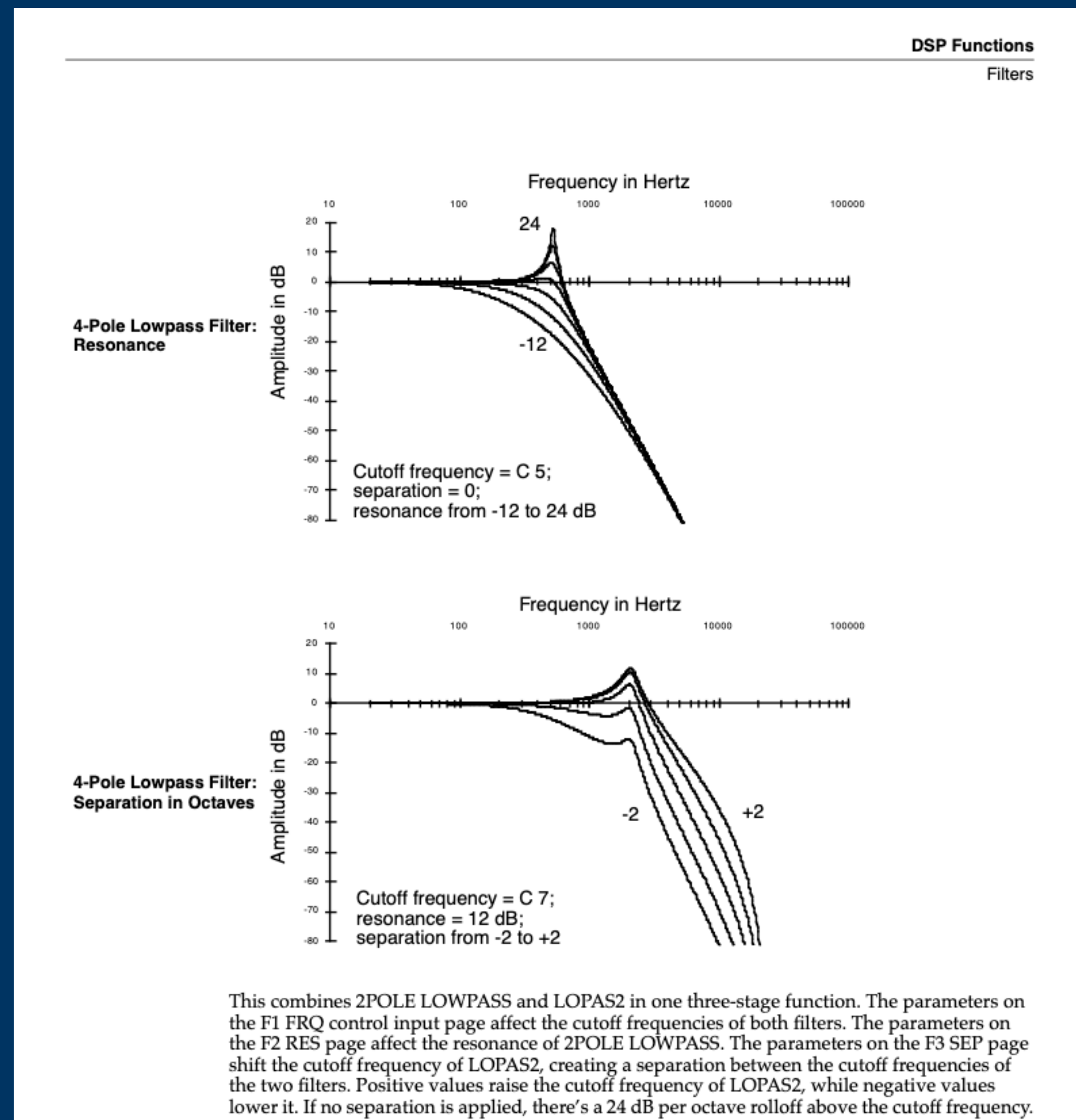
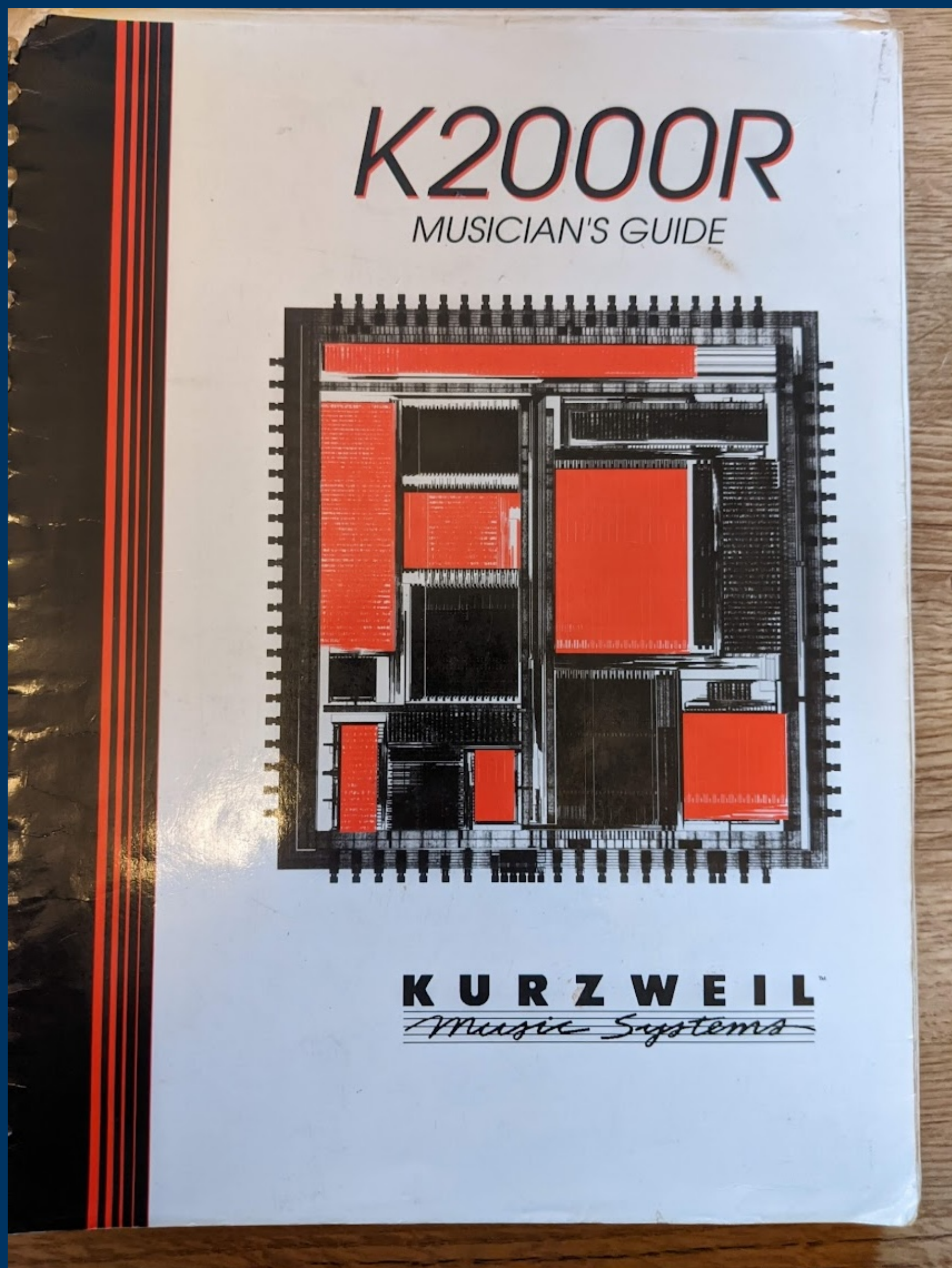
User Octave
C-3
C#3
D
D#
E
F
F#
G
G#
A
A#
B

Vallotti & Young

TX81Z architecture



TX81Z architecture



TX81Z architecture

So let's instead start playing around with the instrument, and find which features are and aren't important for the sounds we care about.

Look for parameters which are set to zero, routing which isn't used, and determine what controls are important to achieve the sound you want.

Take patches you like, fiddle with parameters, turn stuff off, see what is important.

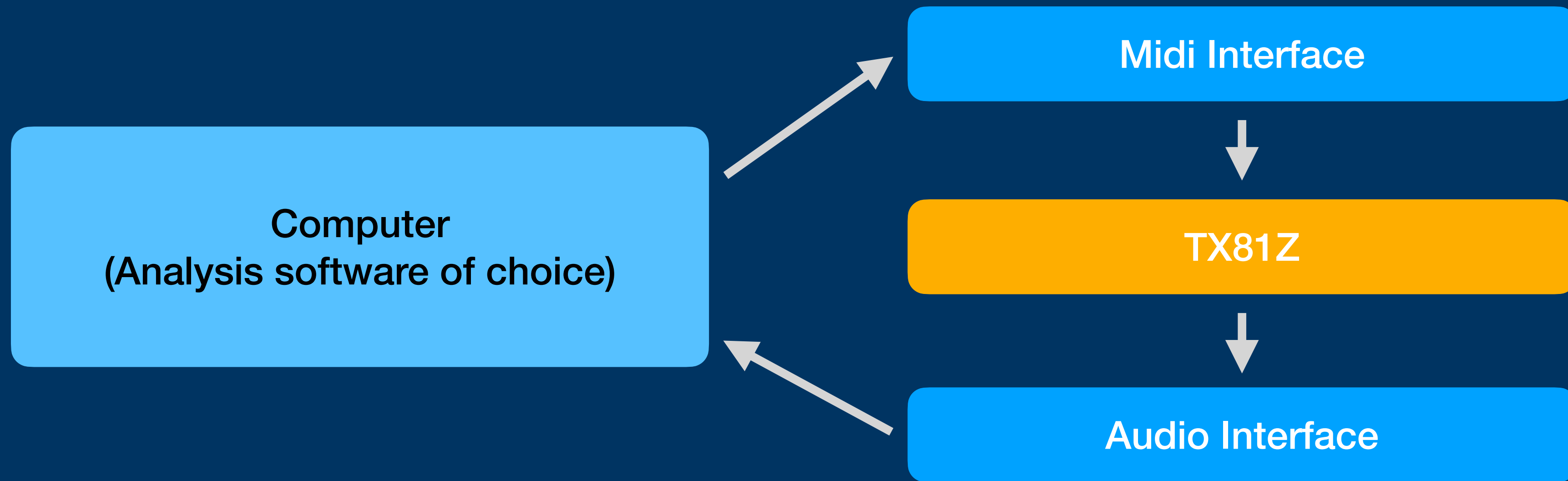
Do this for a range of useful sounds...

TX81Z architecture

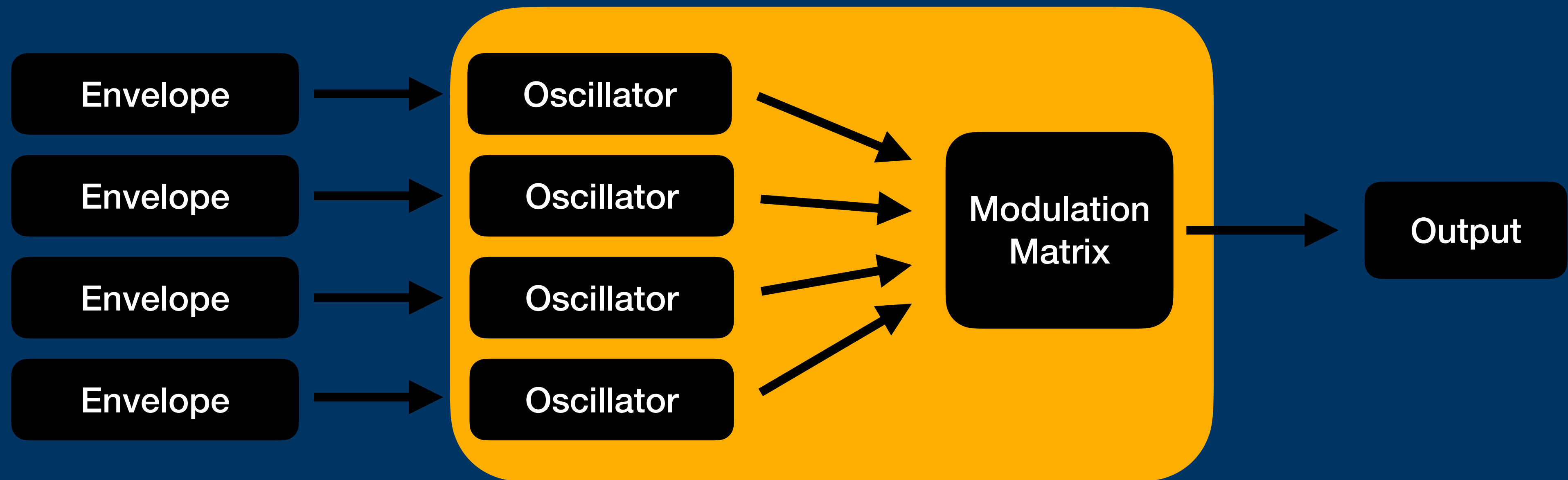
Things we don't care about:

- Performance Mode
- Effects ('pseudo' reverb, pan, delay, chorus)
- Microtonal scales
- LFO
- Breath controller
- Oscillators features Envelope Shift and Level Scaling

How do we analyse the instrument?



TX81Z architecture

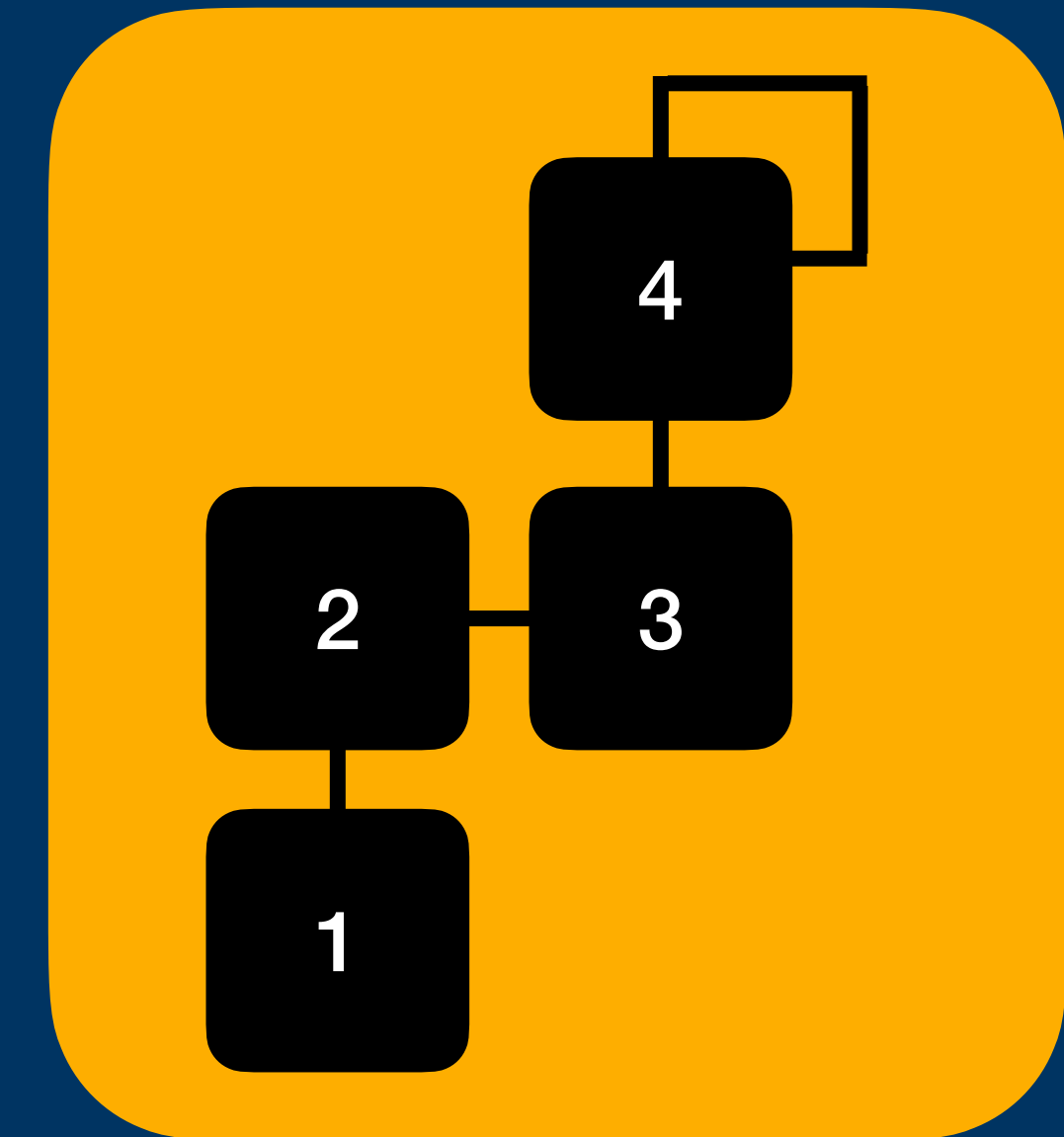


But what's going on in the modulation matrix?

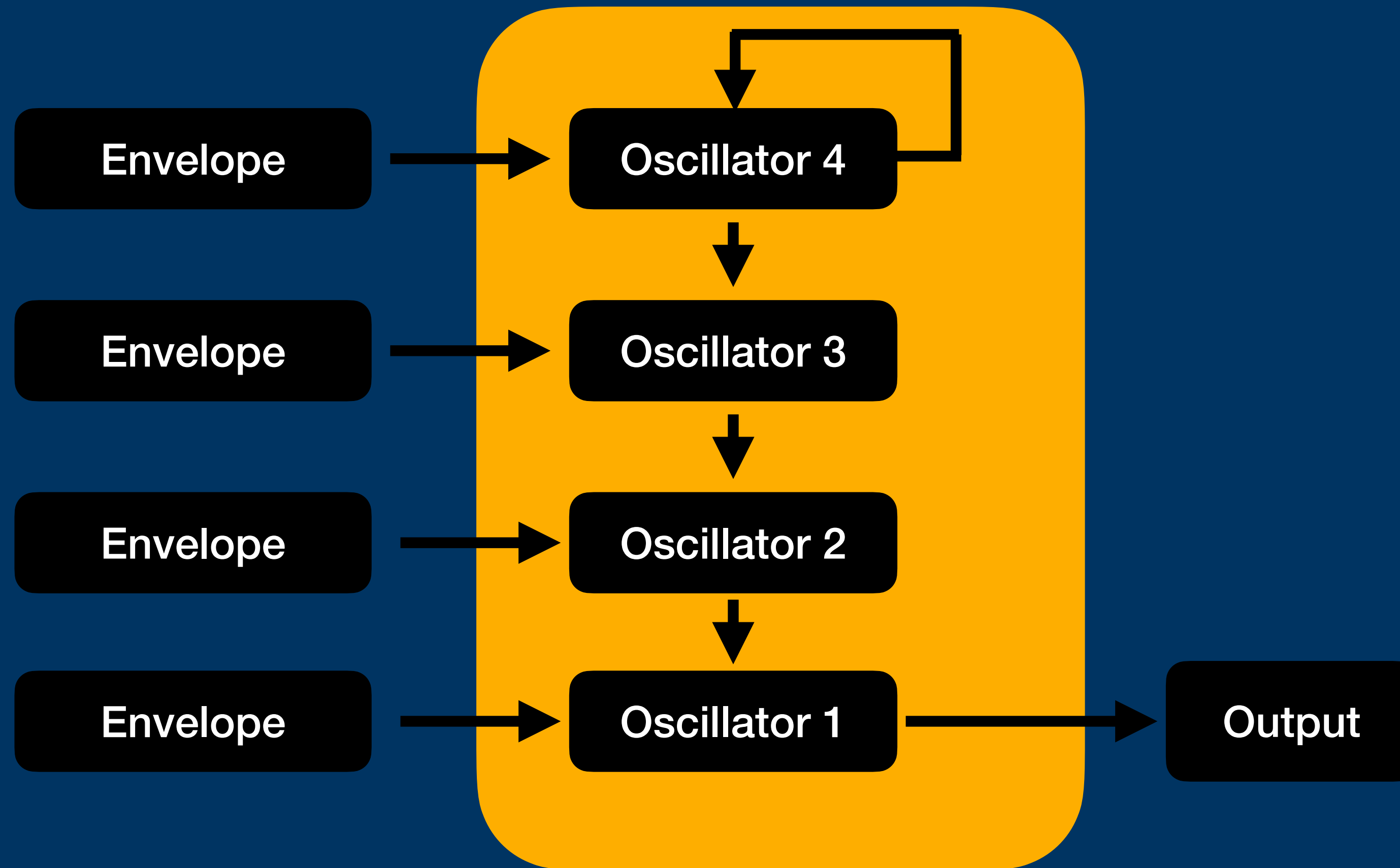
TX81Z architecture

8 Modulation Algorithms

- The algorithm determines which operator (oscillator) modulates which other oscillator(s)
- Operators can form modulation chains
- Operator 4 can modulate itself (feedback)



TX81Z architecture

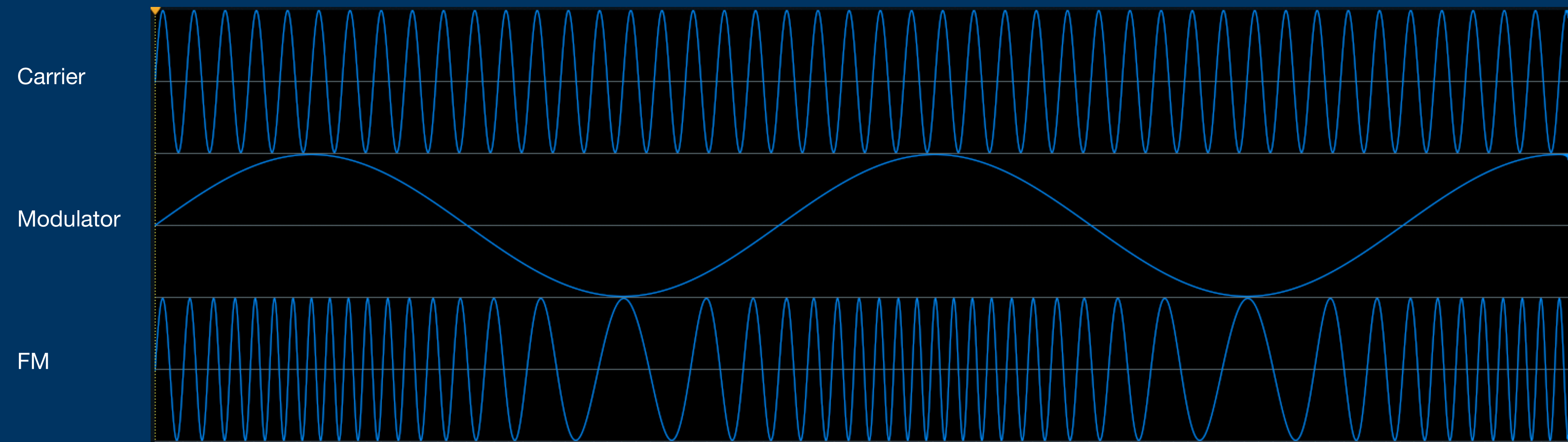


What is FM?

So what is FM anyway?

‘Frequency modulation (FM) is the encoding of information in a carrier wave by varying the instantaneous frequency of the wave.’

wikipedia



What is FM?

FM Modulation implementation in Cmajor

Sine oscillator at frequency $\text{freq} + \text{modulator} * \text{modulationAmount}$

```
processor OscillatorFM (float freq, float modulationAmount)
{
    input stream float modulator;
    output stream float out;

    void main()
    {
        var phaseInc = float (twoPi * freq / processor.frequency);
        float32 phase;

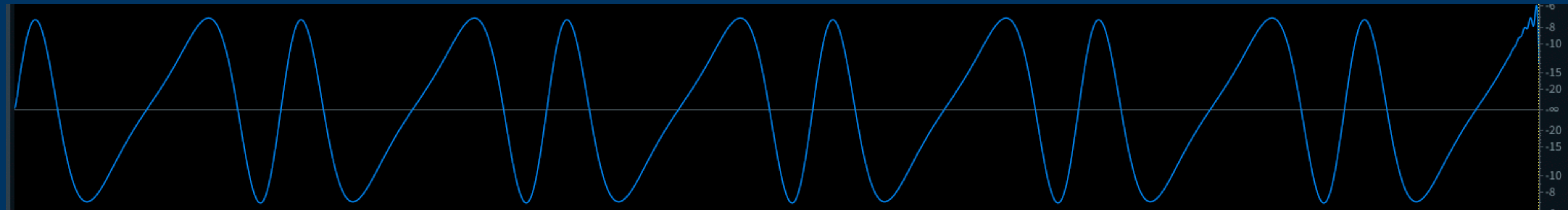
        loop
        {
            out <- sin (phase);
            phase = (phase + phaseInc + modulationAmount * modulator) % float (twoPi);
            advance();
        }
    }
}
```


TX81Z FM test

Create an 'init' patch, which is a simple sine oscillator, and add a modulator at a ratio of 0.5.

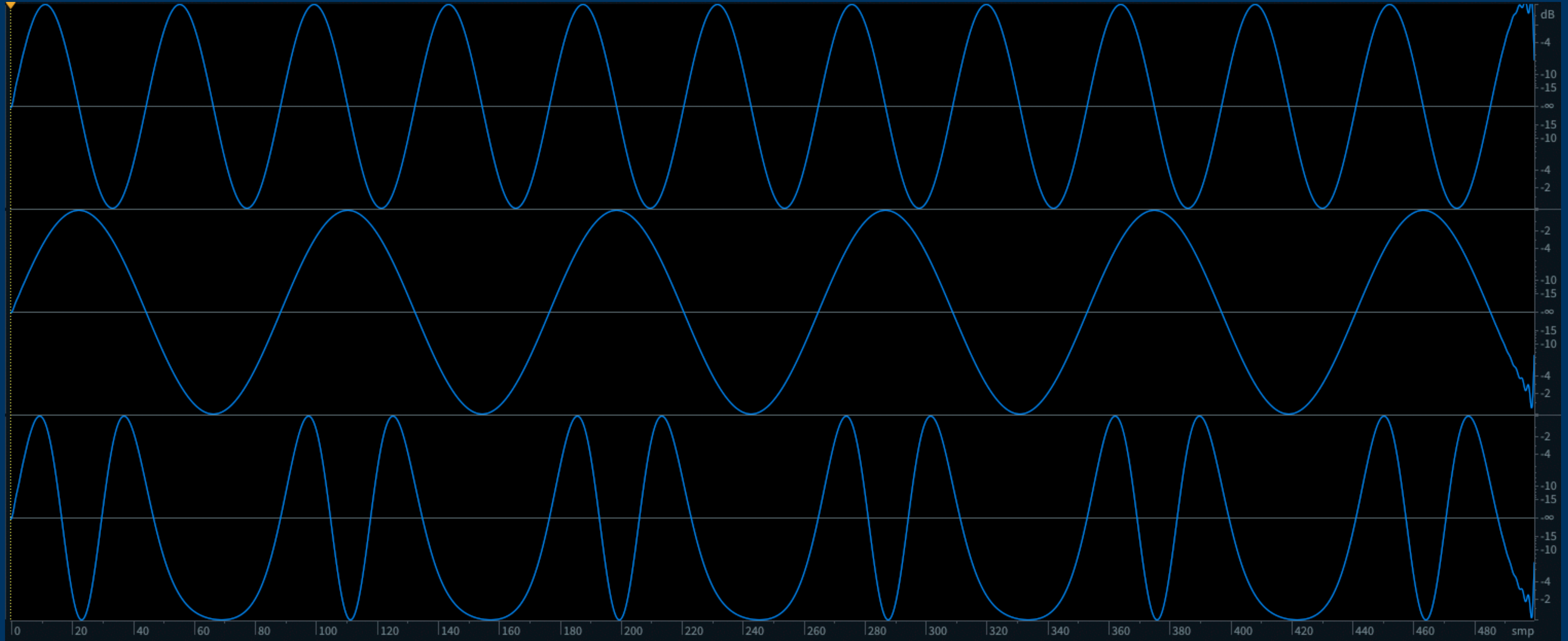
If the oscillator is at 1Khz, the modulator will be 500Hz

TX81Z

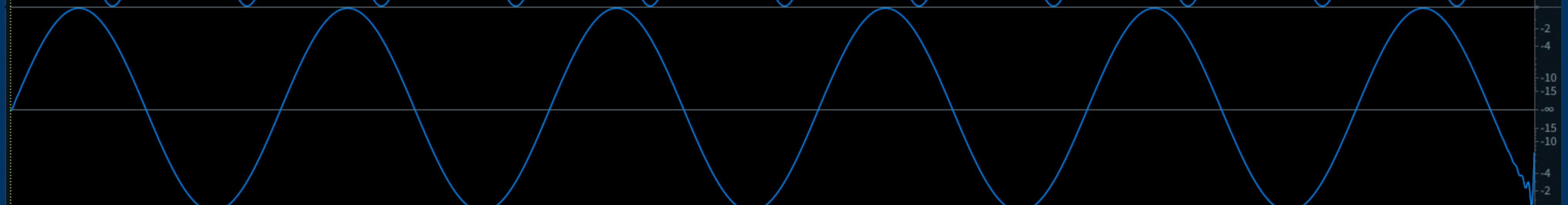


TX81Z FM test

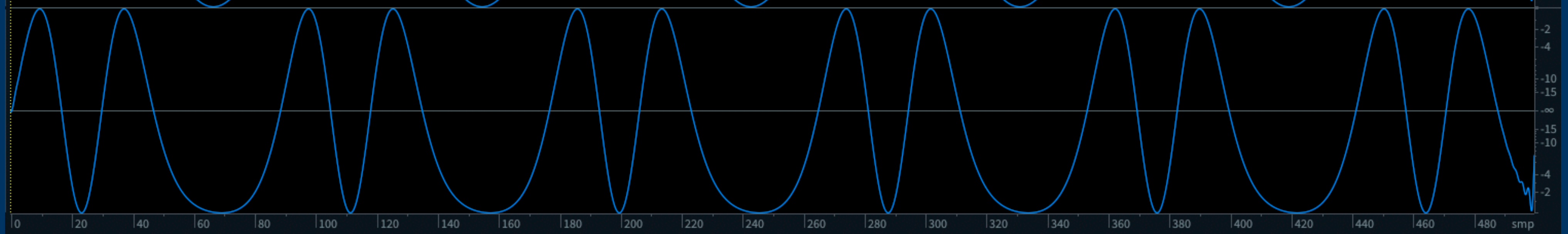
Carrier



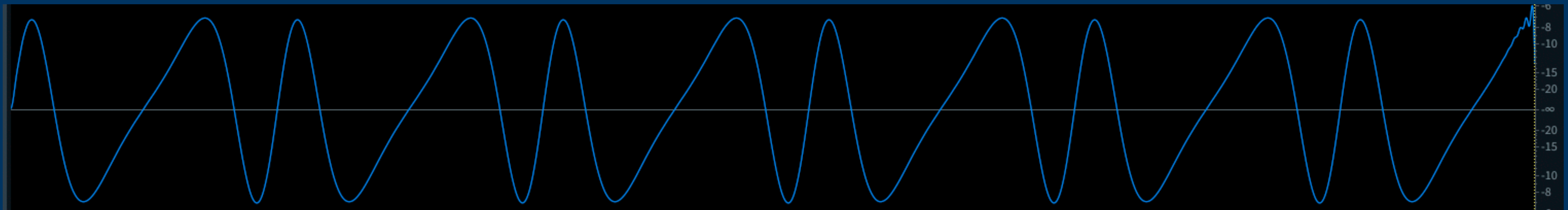
Mod



FM



TX81Z

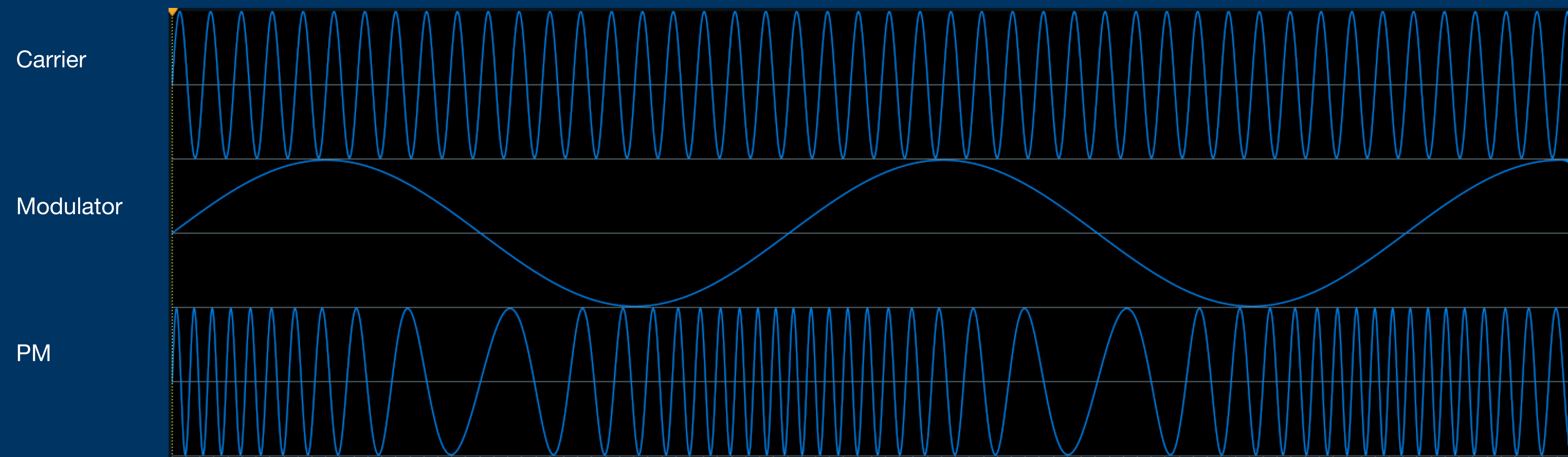


What is Phase Modulation?

So what is PM anyway?

‘It encodes a message signal as variations in the instantaneous phase of a carrier wave.’

wikipedia



PM Modulation vs FM Modulation

```
processor OscillatorFM (float freq, float modulationAmount)
{
  input stream float modulator;
  output stream float out;

  void main()
  {
    var phaseInc = float (twoPi * freq / processor.frequency);
    float32 phase;

    loop
    {
      out <- sin (phase);
      phase = (phase + phaseInc + modulationAmount * modulator) % float (twoPi);
      advance();
    }
  }
}
```

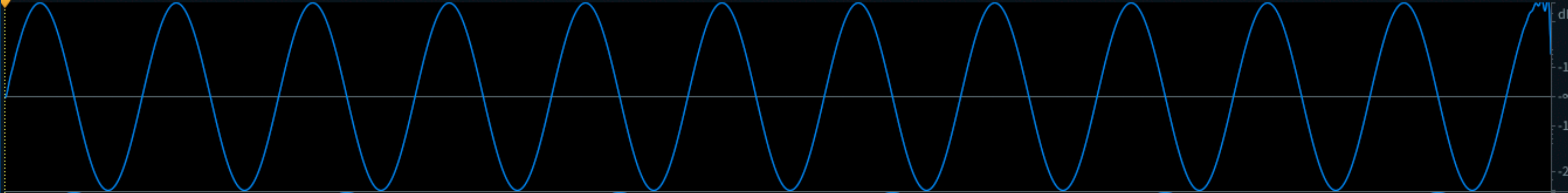
```
processor OscillatorPM (float freq, float modulationAmount)
{
  input stream float modulator;
  output stream float out;

  void main()
  {
    var phaseInc = float (twoPi * freq / processor.frequency);
    float32 phase;

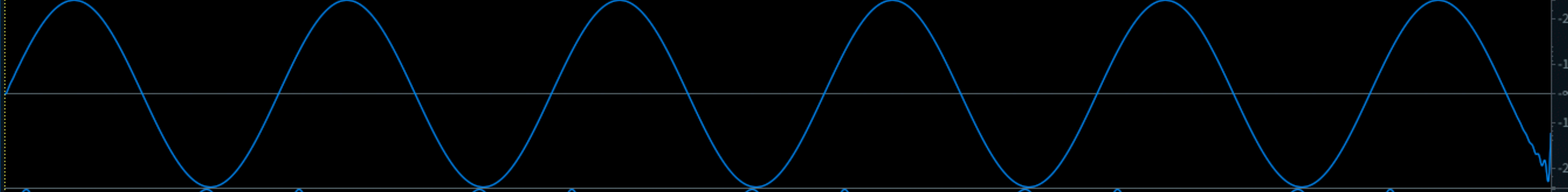
    loop
    {
      out <- sin (phase + modulationAmount * modulator);
      phase = (phase + phaseInc) % float (twoPi);
      advance();
    }
  }
}
```

TX vs Phase Modulation

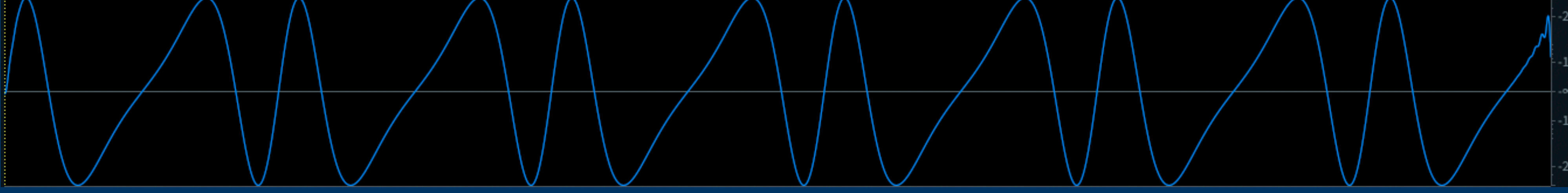
Carrier



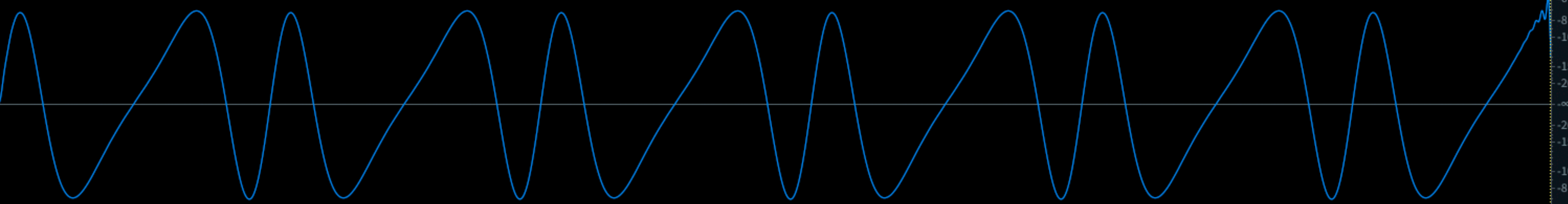
Mod



PM

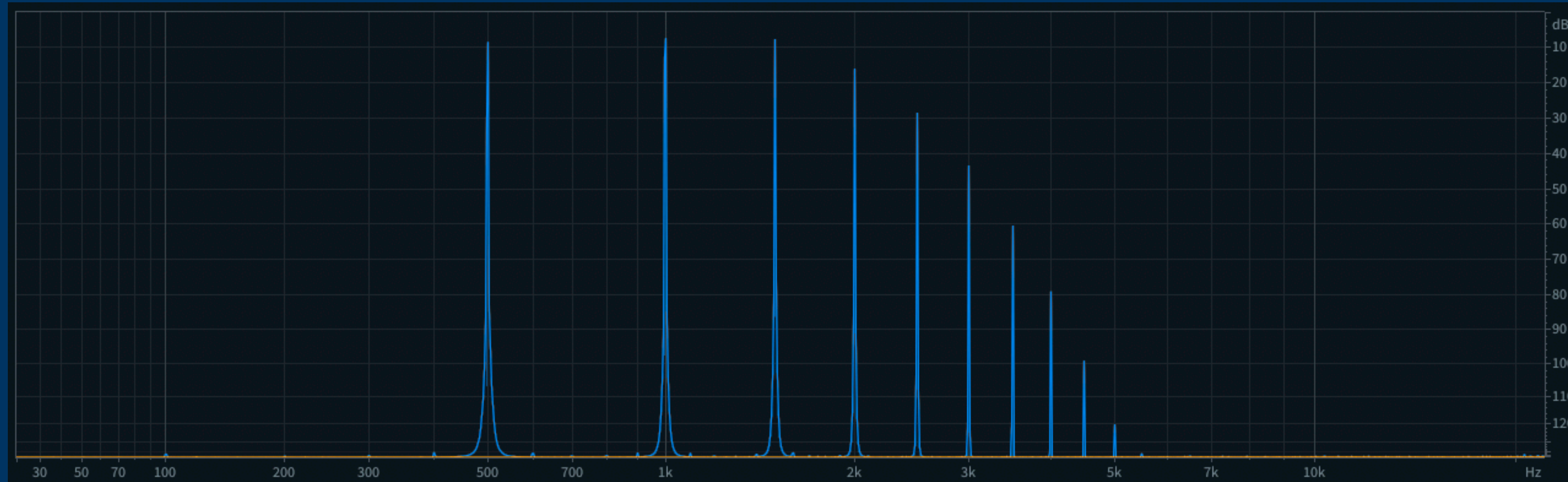


TX81Z

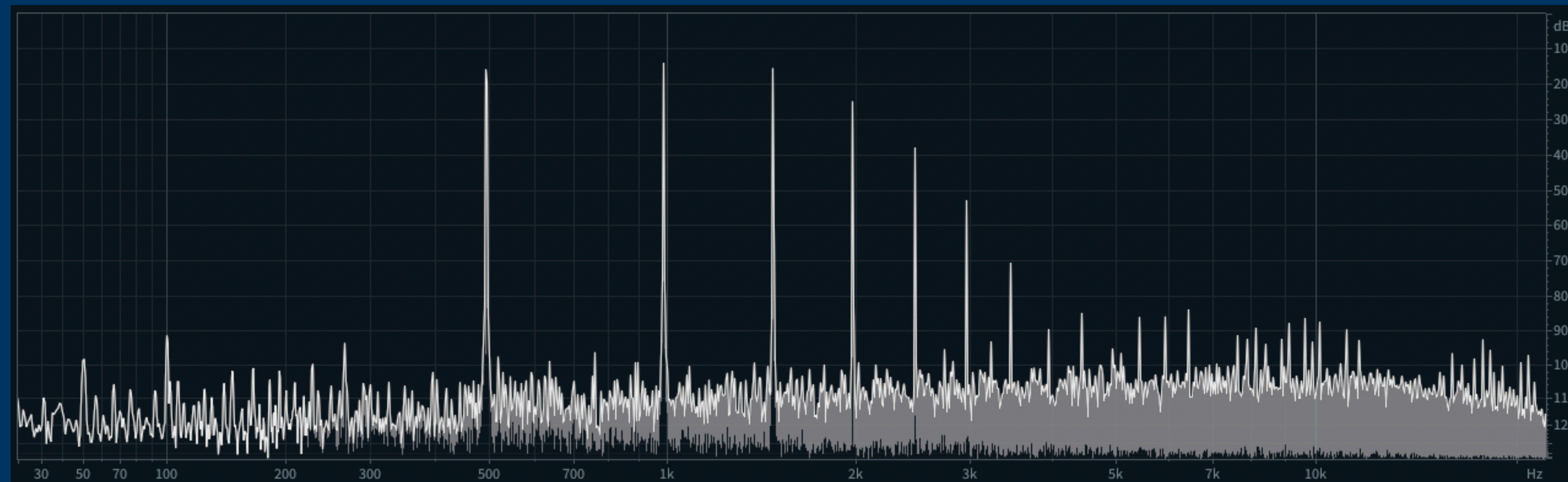


Frequency Plot of PM output vs TX

PM



TX81Z

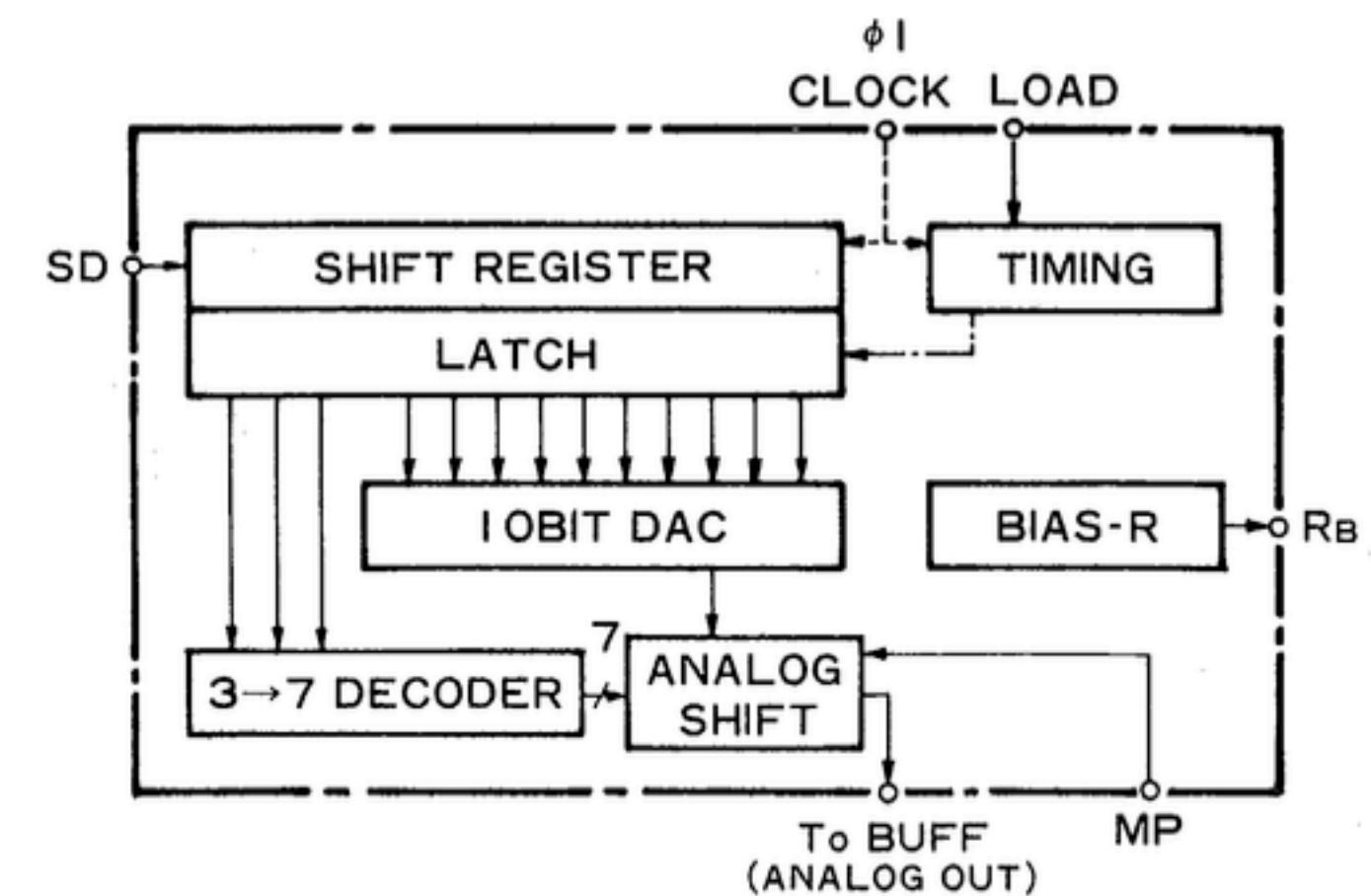


YM3012 DAC

- 10 bit resolution
- 16 bit dynamic range, using 3 bit exponent (analog shift)

■ OUTLINE

YM3012: DAC-MS (hereinafter referred to as DAC) is a floating D/A converter with serial input for two channels . It can generate analog output (dynamic range 16 bits) of 10-bit mantissa section and 3-bit exponent section on the basis of input digital signal.



Emulating the YM3012

- Quantise signal, add noise, add mains hum

```
// The TM3012 has a 16 bit dynamic range, but 10 bit resolution
graph Ym3012DacEmulation
{
    input stream float in;
    output stream float out;

    node quantiser = Quantiser;
    node noise = std::noise::White;
    node mainsHum1 = std::oscillators::Sine (float, 50.0f);
    node mainsHum2 = std::oscillators::Sine (float, 100.0f);
    node noiseFilter = filter::Processor (filter::Mode::lowPass, 12000.0f);

    namespace filter = std::filters::tpt::onepole;

    connection
    {
        in -> quantiser -> noiseFilter.in;
        noise.out * 0.001f -> noiseFilter.in;
        noiseFilter.out -> out;
        mainsHum1.out * 0.00002f -> out;
        mainsHum2.out * 0.00005f -> out;
    }
}
```

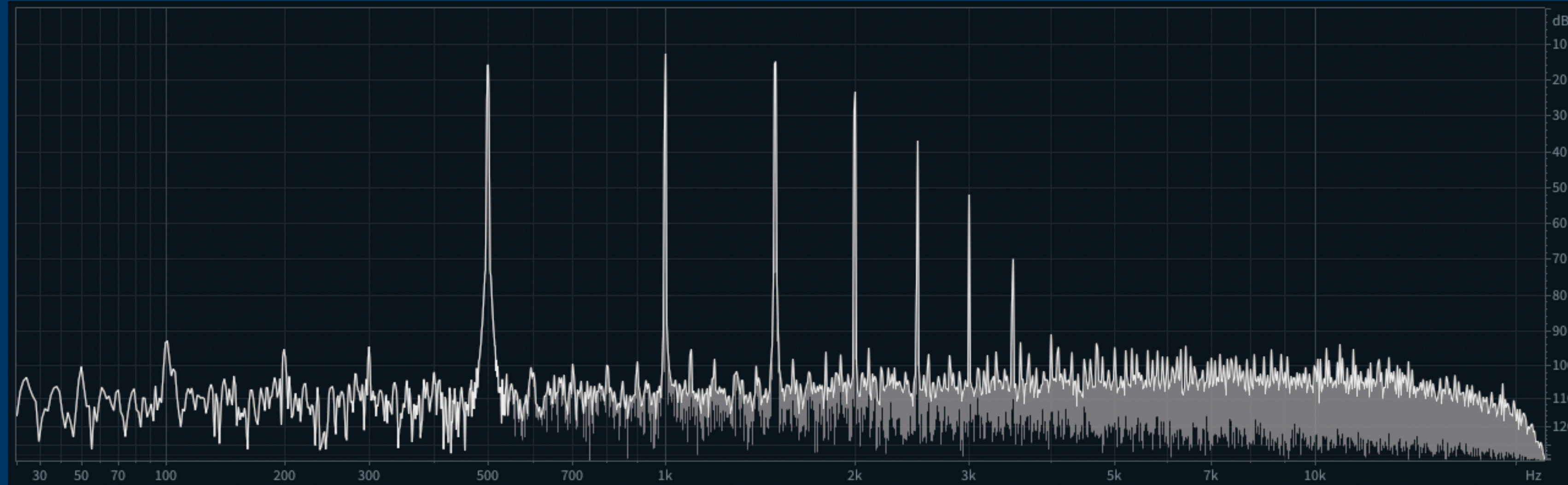
```
processor Quantiser
{
    input stream float in;
    output stream float out;

    float quantise (float f)
    {
        let i = 32768 + int (f * 32767.0f);
        let shift = min (clz (i) - 16, 7);
        let shifted = i << shift;
        let quantised = shifted & 0xffc0;
        return float ((quantised >> shift) - 32768) * (1.0f / 32767.0f);
    }

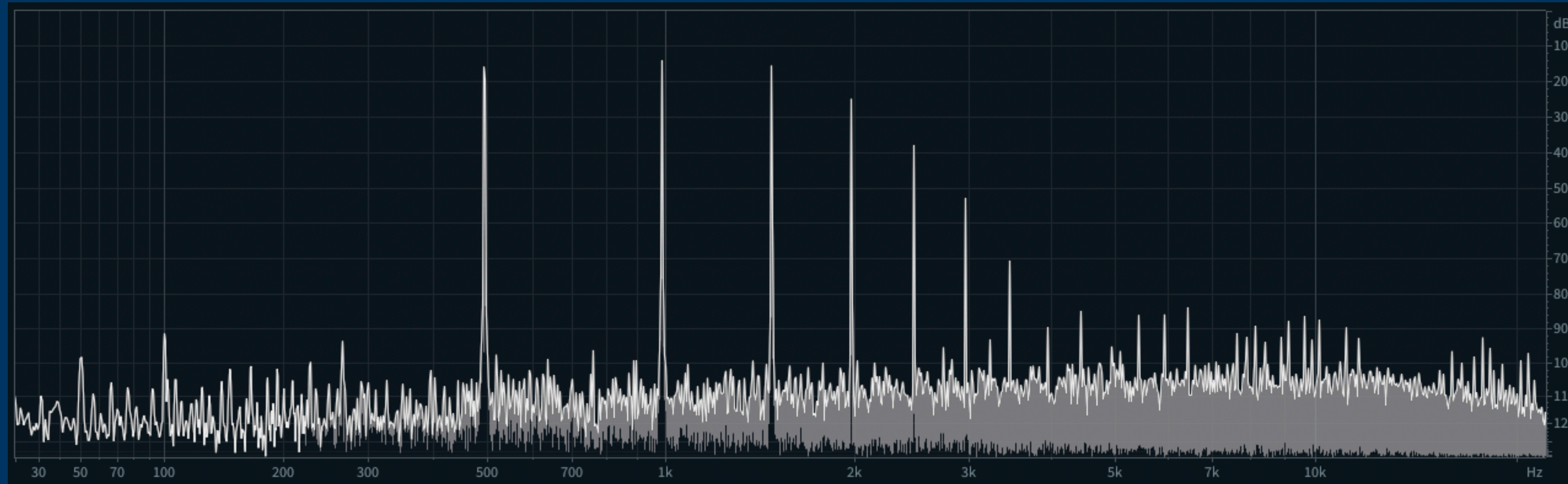
    void main()
    {
        loop
        {
            out <- quantise (in);
            advance();
        }
    }
}
```

Frequency Plot of PM output vs TX

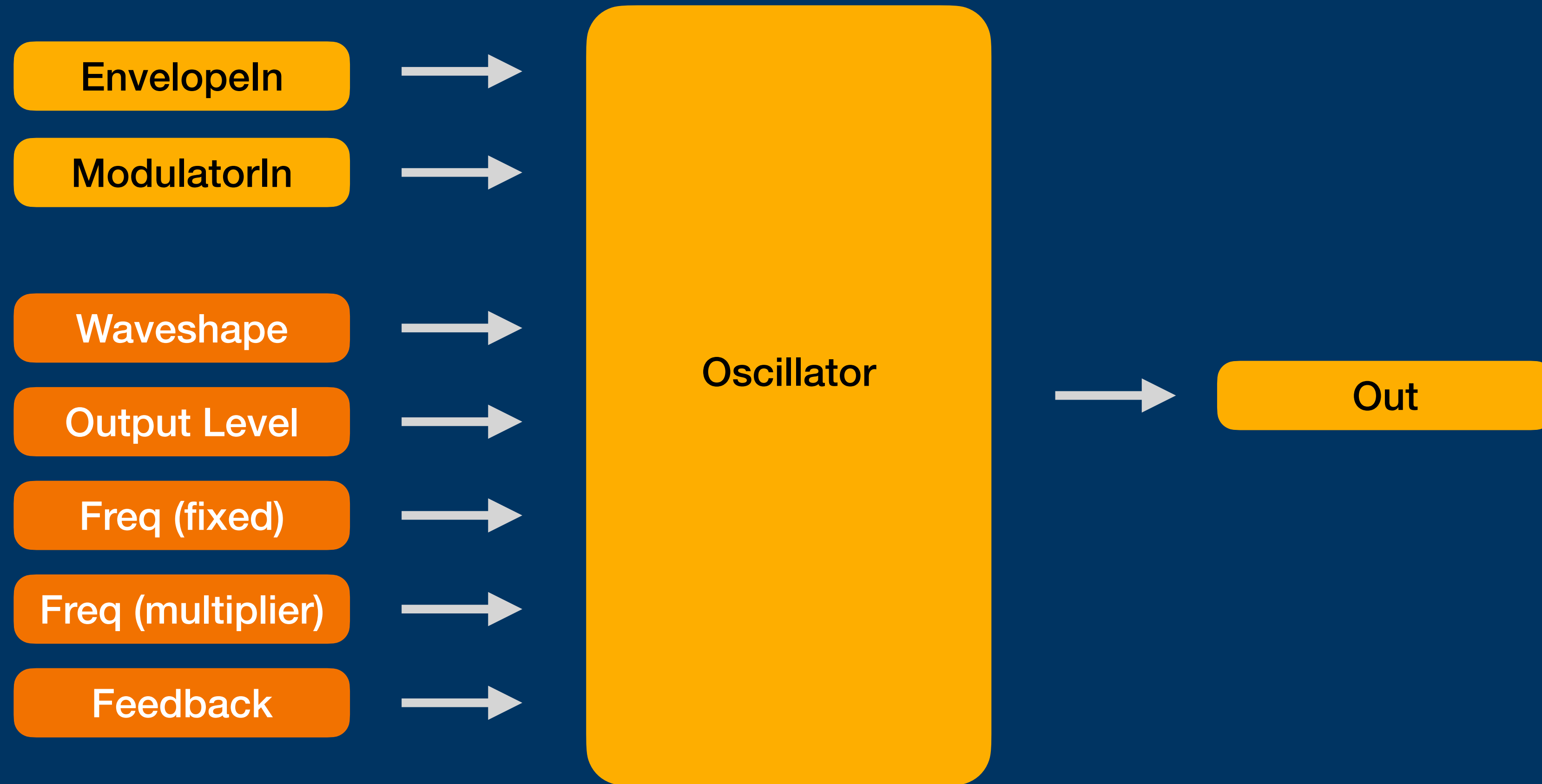
PM +
YM3012



TX81Z



Let's build the oscillator



Let's build the oscillator

Our oscillator will be a phase modulation based oscillator, with the following parameters:

- Waveform Shapes
- Frequency (fixed and multiplier)
- Output (modulation) levels
- Feedback

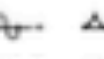
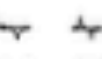


Let's build the oscillator - Waveform

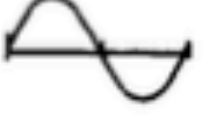
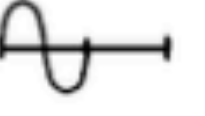
8 waveform shapes, listed in the manual

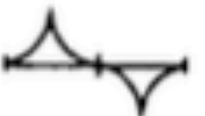
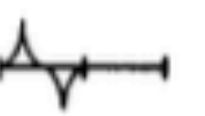
The shapes are all based on a sine oscillator, chopped into parts

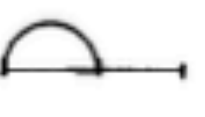
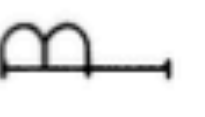
Oscillator Wave

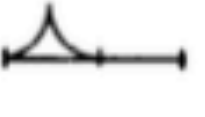

The TX81Z is the first Yamaha synthesizer to offer FM synthesis with 7 new waveforms in addition to sine waves. Each operator can be independently set to one of the following 8 waveforms. The selected waveform will be graphically indicated in the LCD.

• 1 1 1 1				
OSW	W5	W2	W6	W7

W1:  W5: 

W2:  W6: 

W3:  W7: 

W4:  W8: 

These new waveforms can be used as both carriers and modulators. Sine waves are pure tones. The seven additional waveforms have additional harmonics already in them. This allows for complex waveforms to be created from fewer operators. For ideas on how to use the new waveforms, examine the preset voices. The harmonic content of each waveform is described on p.50.

TX81Z Oscillator Waveforms

getWaveshape function using the diagrams in the manual to build the different oscillator patterns from an input phase (0 .. 1)

```
// 8 waveshapes defined within the instrument, all variations on the sin function
float getWaveshape (int waveshape, float phase)
{
    phase = fmod (phase, 1.0f);

    if (waveshape == 1)        return sinOfPhase (phase);

    if (waveshape == 2)
    {
        if (phase < 0.25f)    return sinOfPhase (phase - 0.25f) + 1.0f;
        if (phase < 0.5f)    return sinOfPhase (phase + 0.25f) + 1.0f;
        if (phase < 0.75f)    return sinOfPhase (phase - 0.25f) - 1.0f;
        return sinOfPhase (phase + 0.25f) - 1.0f;
    }

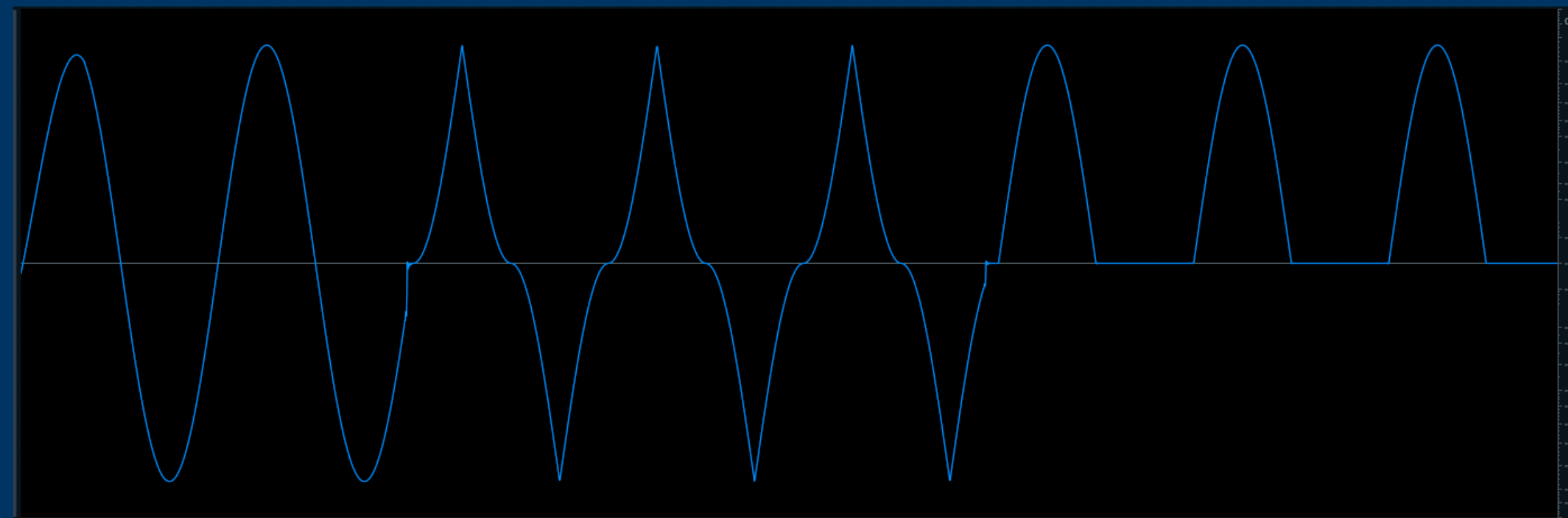
    if (waveshape == 3)        return phase < 0.5 ? sinOfPhase (phase) : 0.0f;

    if (waveshape == 4)
    {
        if (phase < 0.25f)    return sinOfPhase (phase - 0.25f) + 1.0f;
        if (phase < 0.5f)    return sinOfPhase (phase + 0.25f) + 1.0f;
        return 0.0f;
    }
}
```

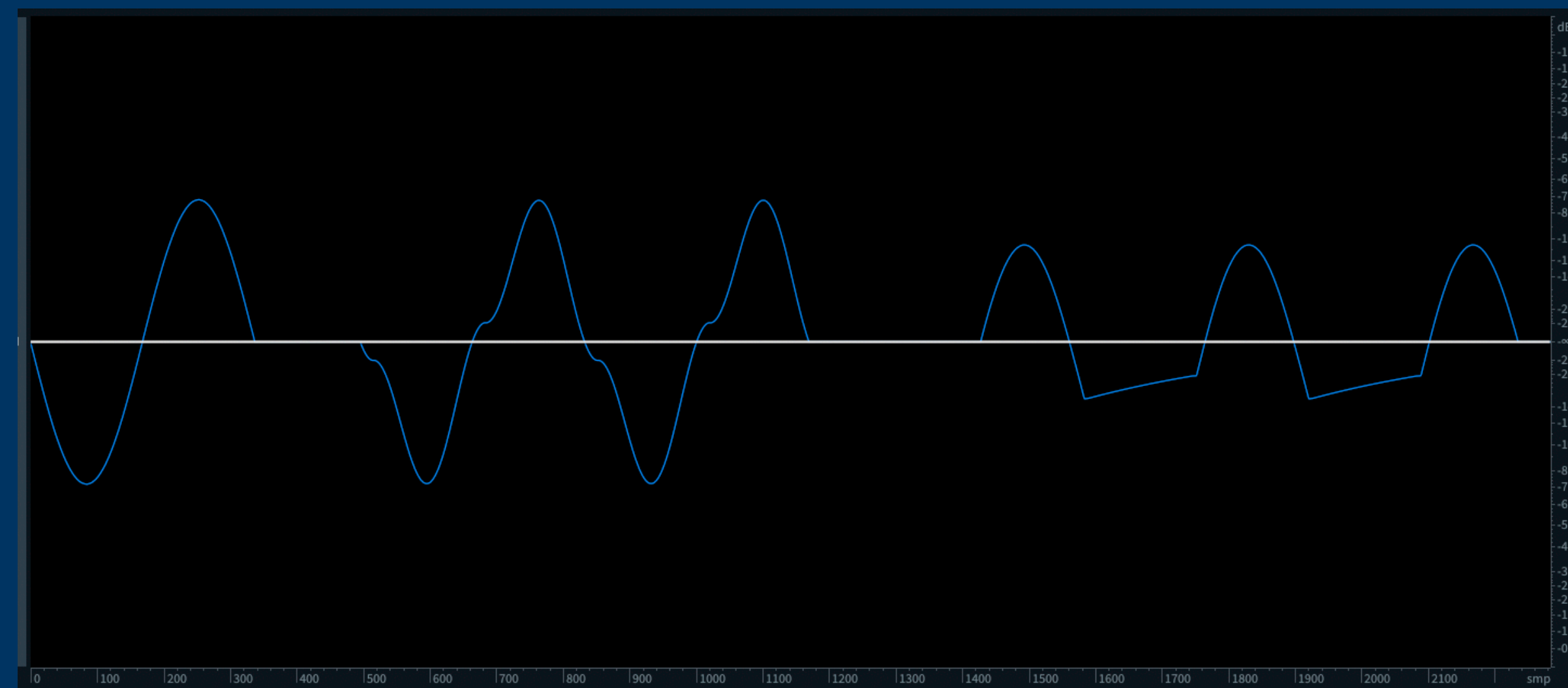

TX81Z Oscillator Waveforms

Comparing the first three waveforms with the manual shows roughly the same shape, but something else going on...

Oscillator



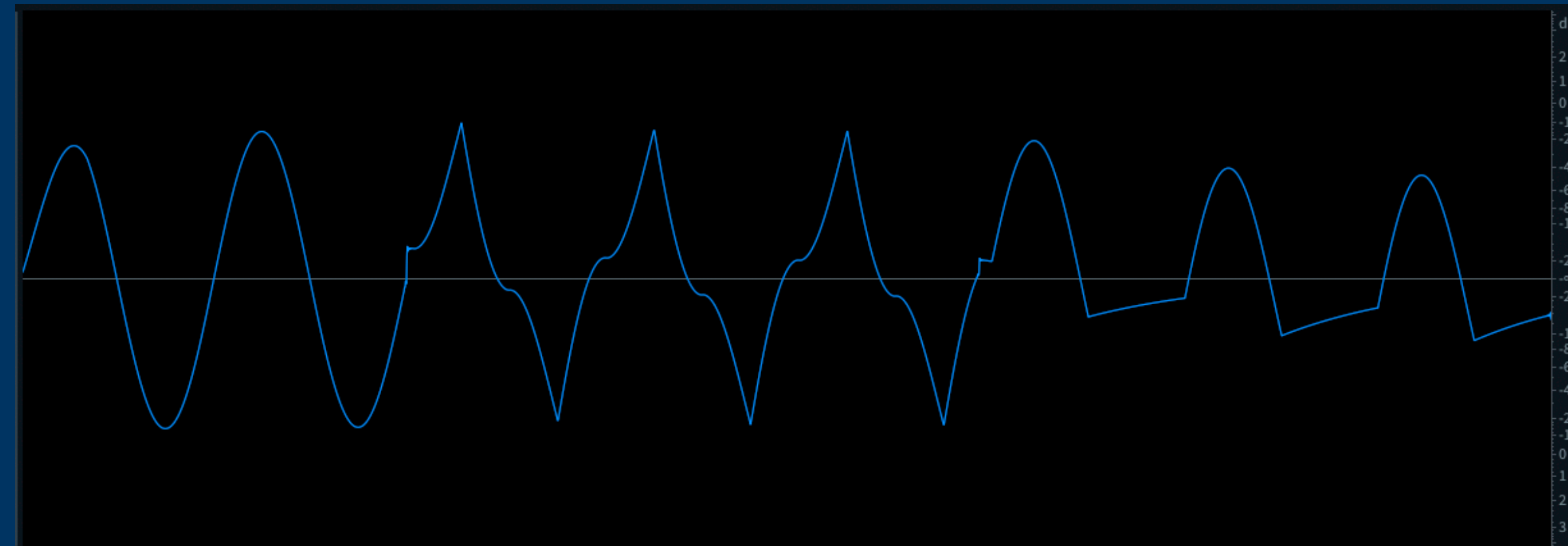
TX81Z



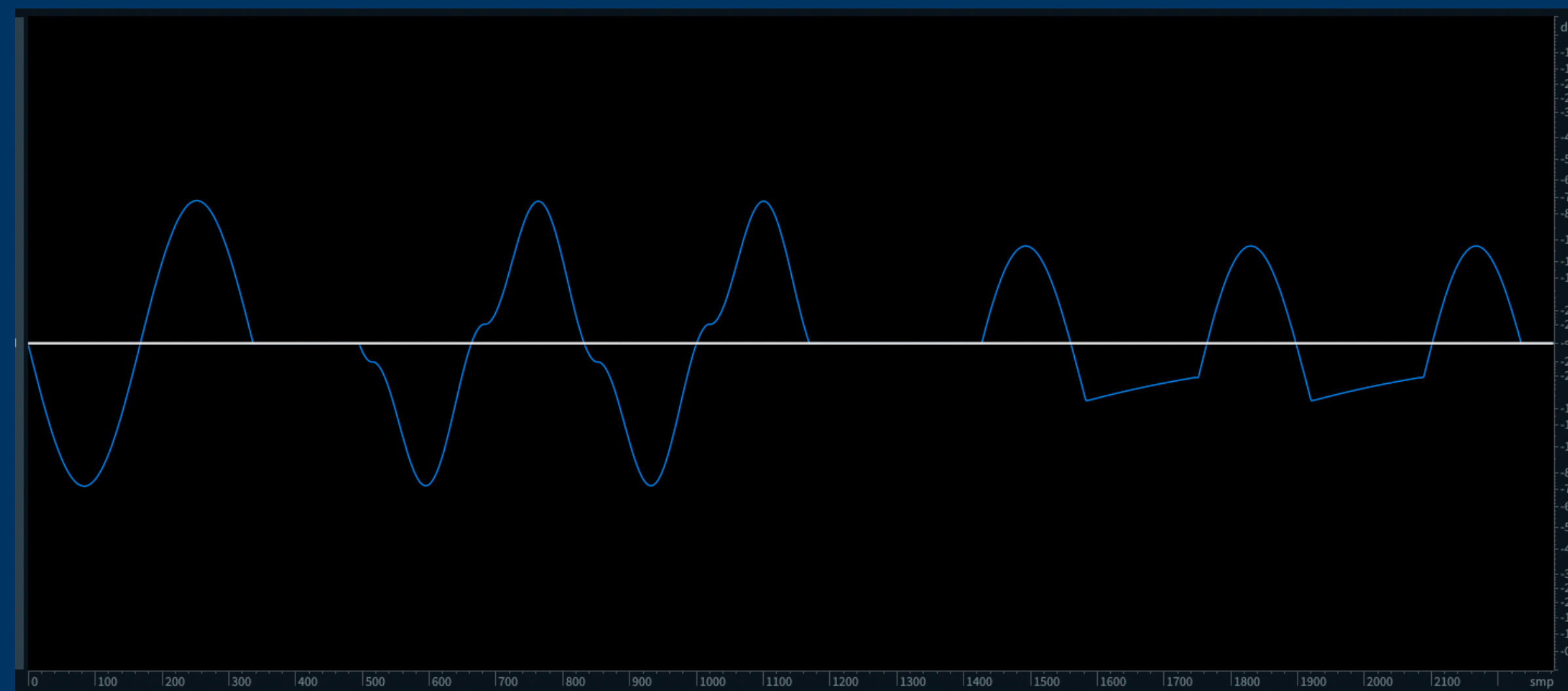
TX81Z Oscillator Waveforms

Just adding a DC blocking filter and the waveforms are more aligned

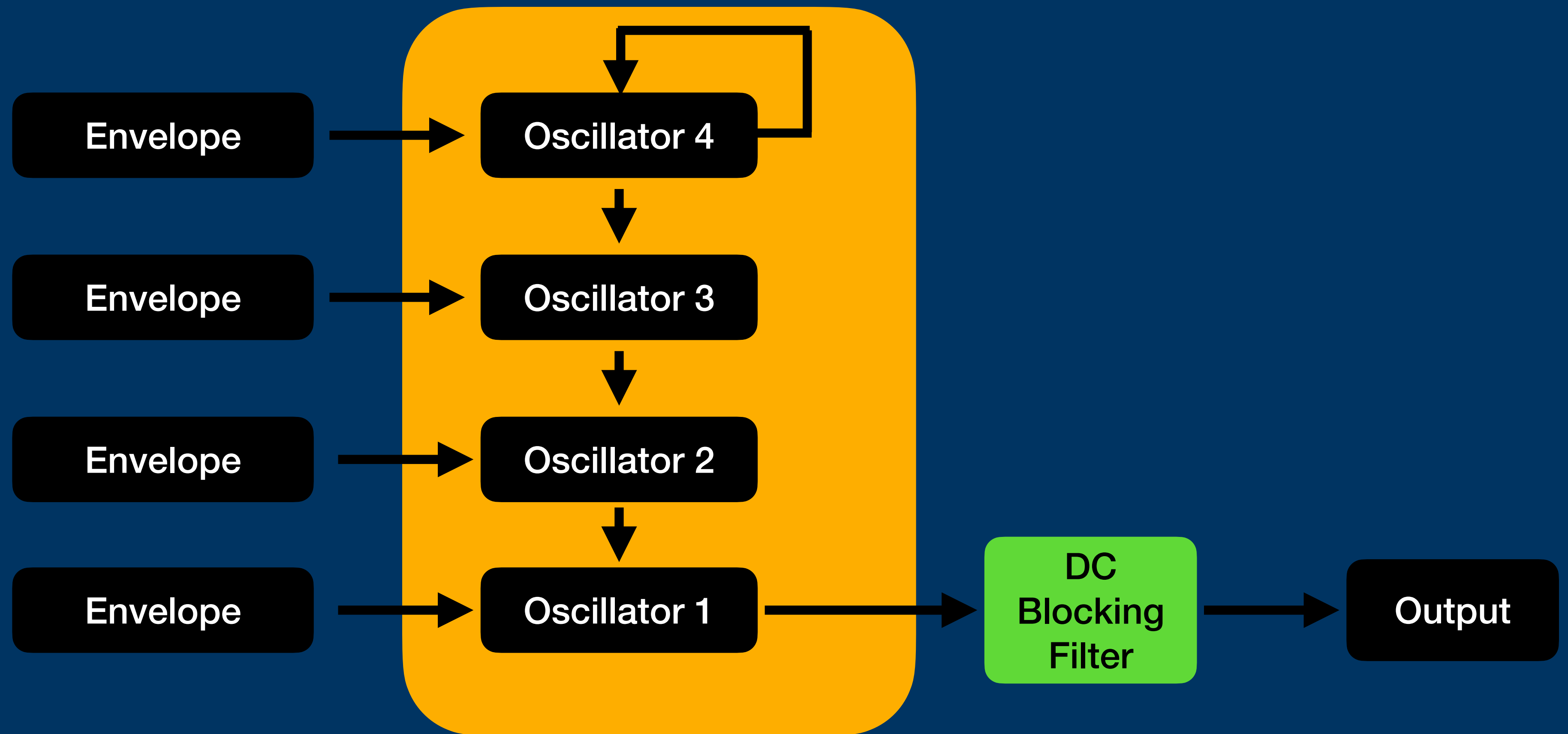
Oscillator



TX81Z



TX81Z architecture



TX81Z Oscillator Output Level

Working out the instrument scaling for output level

1. Build a simple patch with a single oscillator sine wave (no envelope)
2. Vary the output level, recording the peak value achieved at each level.
3. Plot on a graph, find a function that matches the curve

TX81Z Oscillator Output Level

```
processor Test
{
  input stream float in;
  output event (std::notes::NoteOn, std::notes::NoteOff) eventOut;
  output event float peakLevel;

  void main()
  {
    eventOut <- std::notes::NoteOff (0, 48.0f, 1.0f);

    loop
    {
      eventOut <- std::notes::NoteOn (0, 48.0f, 1.0f);

      float peak = 0.0f;

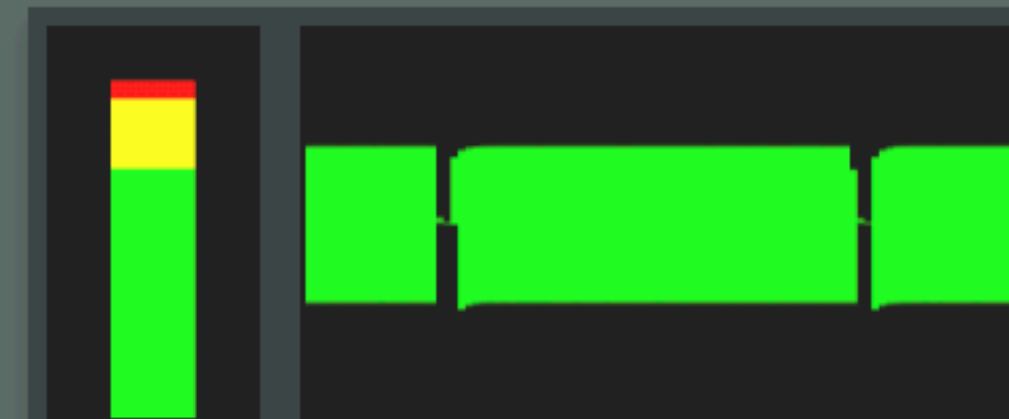
      loop (1000)
      {
        advance();
      }

      loop (int (processor.frequency / 2))
      {
        peak = max (peak, abs (in));
        advance();
      }

      peakLevel <- std::levels::gainTodB (peak);
      eventOut <- std::notes::NoteOff (0, 48.0f, 1.0f);

      loop (1000)
      {
        advance();
      }
    }
  }
}
```

Inputs



Live File Mute in

Outputs

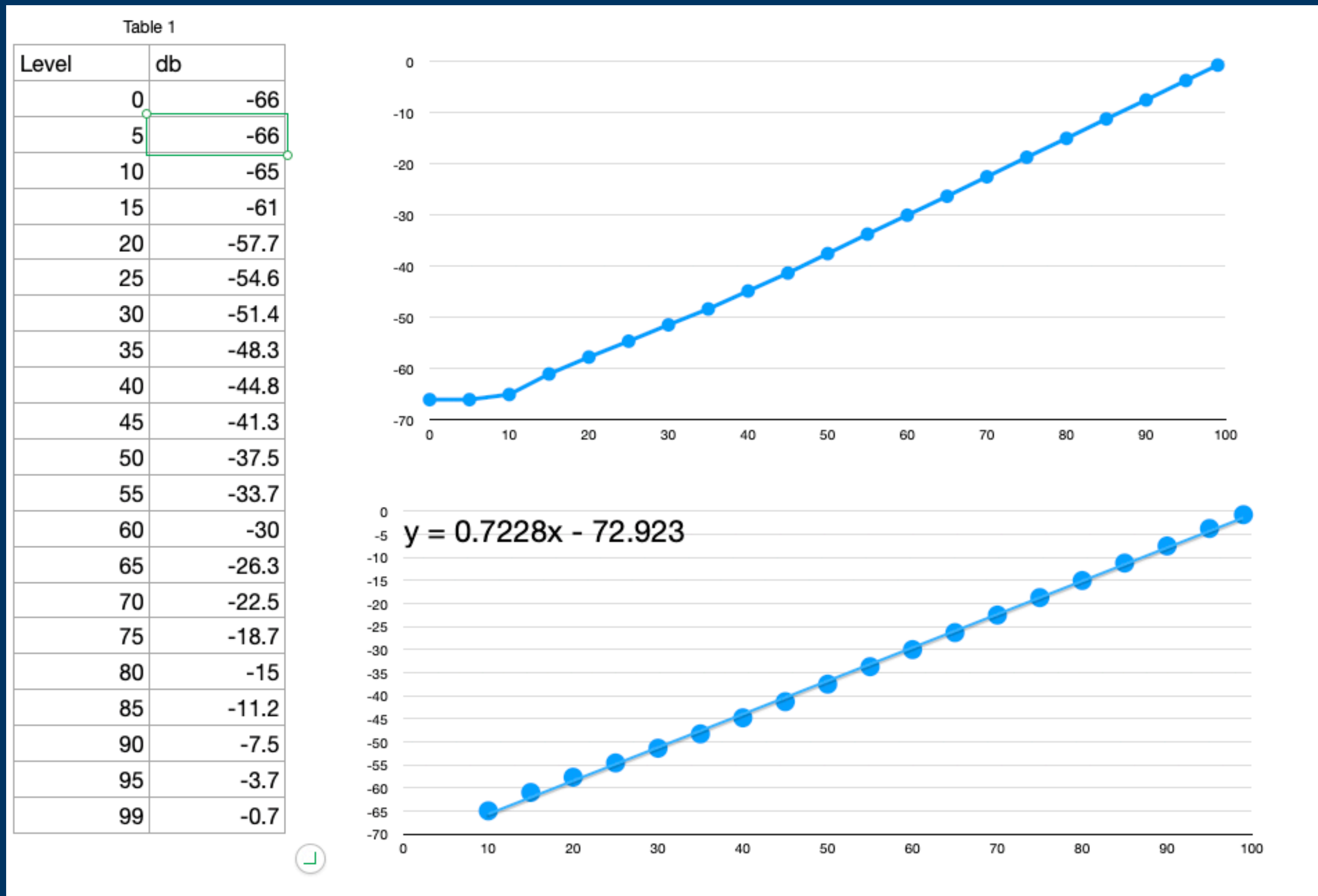
```
Note-Off: C2 Channel 1 Velocity 127
Note-On: C2 Channel 1 Velocity 127
Note-Off: C2 Channel 1 Velocity 127
Note-On: C2 Channel 1 Velocity 127
Note-Off: C2 Channel 1 Velocity 127
Note-On: C2 Channel 1 Velocity 127
Note-Off: C2 Channel 1 Velocity 127
Note-On: C2 Channel 1 Velocity 127
```

Clear midiOut

```
-7.001480579376221
-7.0146589279174805
-7.015223503112793
-7.025267601013184
-7.0233893394470215
-7.029960632324219
-7.039812088012695
-7.041423797607422
```

Clear peakLevel

TX81Z Oscillator Output Level



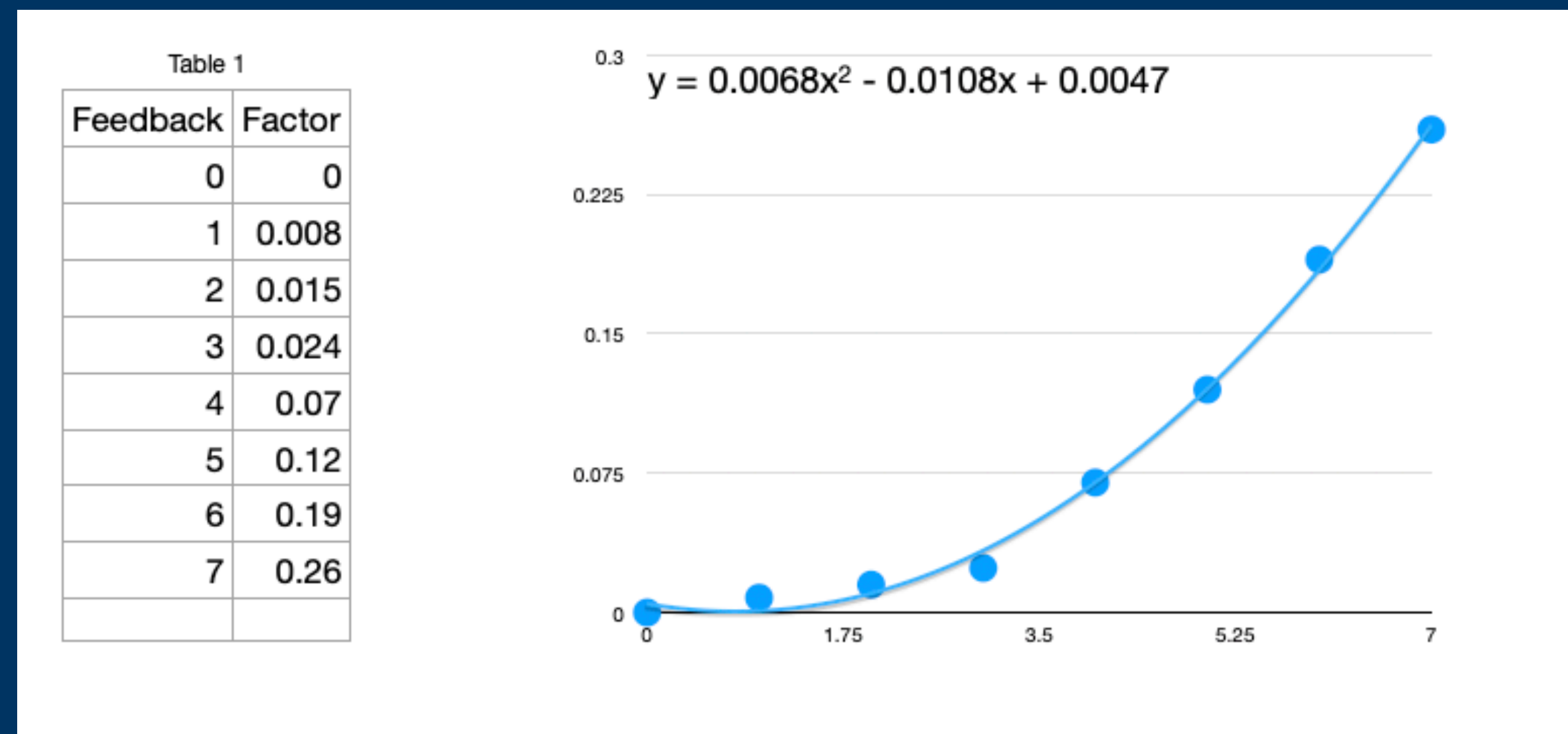
```
float levelToGain (float level)
{
    return std::levels::dBtoGain ((0.723f * level) - 72.9f);
}
```


TX81Z Oscillator Feedback

Oscillator feedback, only 8 levels supported (0 through 7).

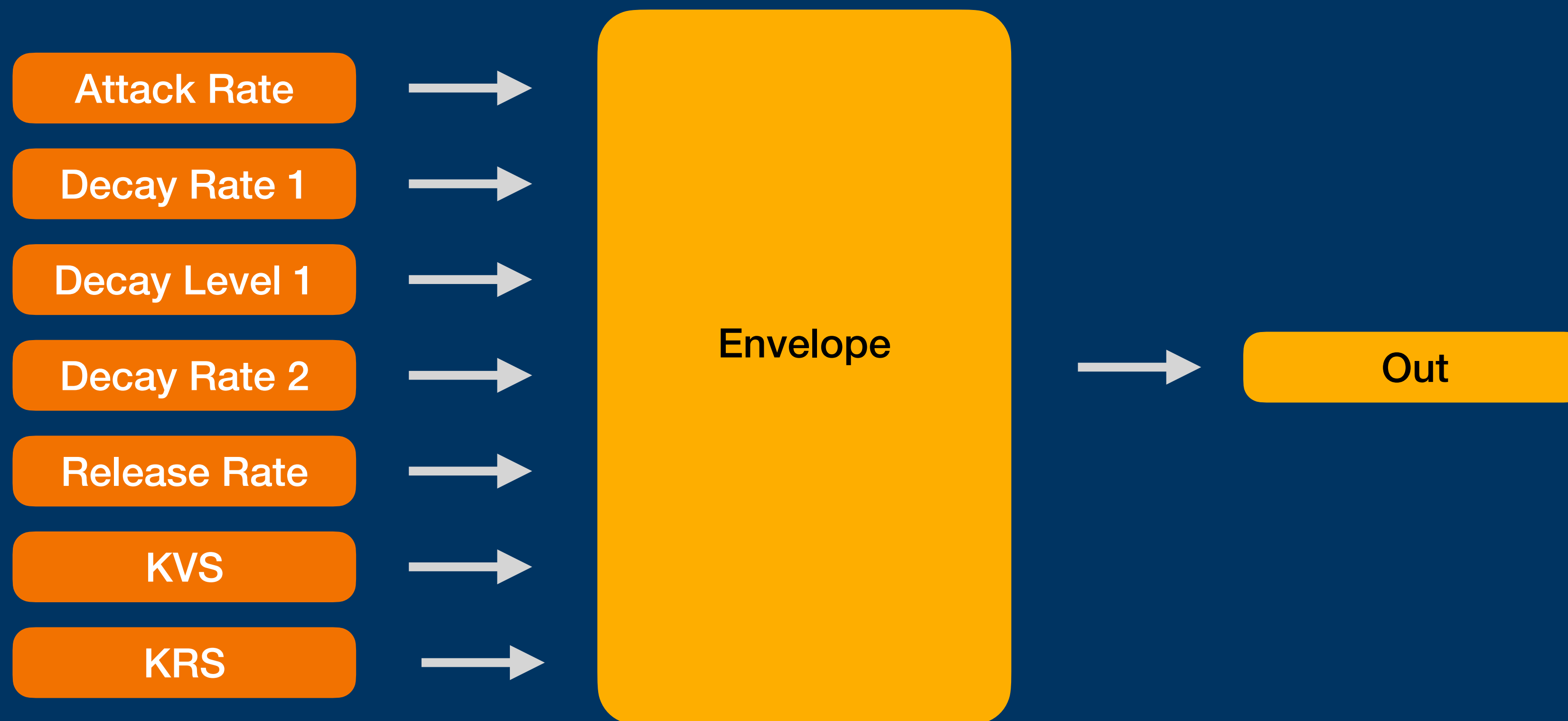
For each level, record a plain sine output with this level of feedback, and use spectrum analyser to tune equivalent within our oscillator.

The curve wasn't a good fit, so use a lookup table



Oscillator demo

Envelope



Envelope

The envelopes are multi-stage, with different shapes for attack and decay/release phases. How to determine the shape?

As before, build a test sound using a simple oscillator, but using the envelope features you want to understand.

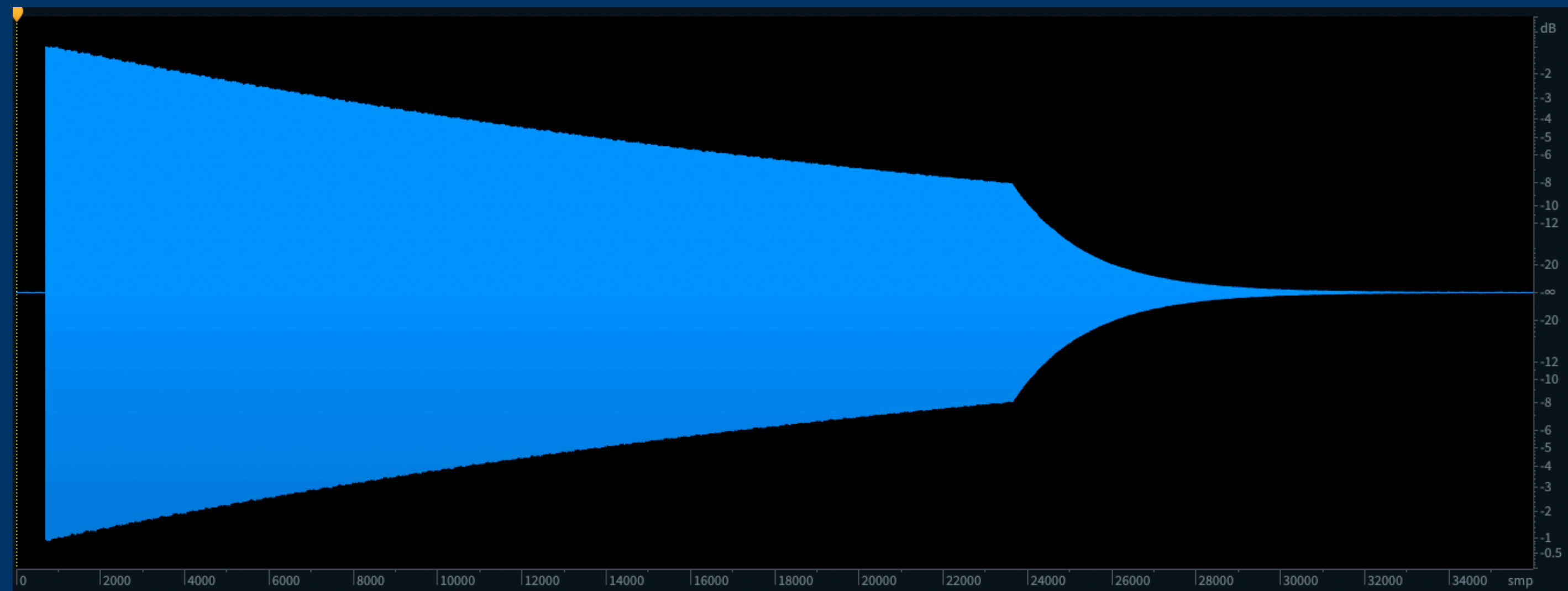
Generate test data to determine a strategy.

Envelope

Write a basic envelope follower:

Play a high pitched note, rectify, determine peak amplitude in a time window.

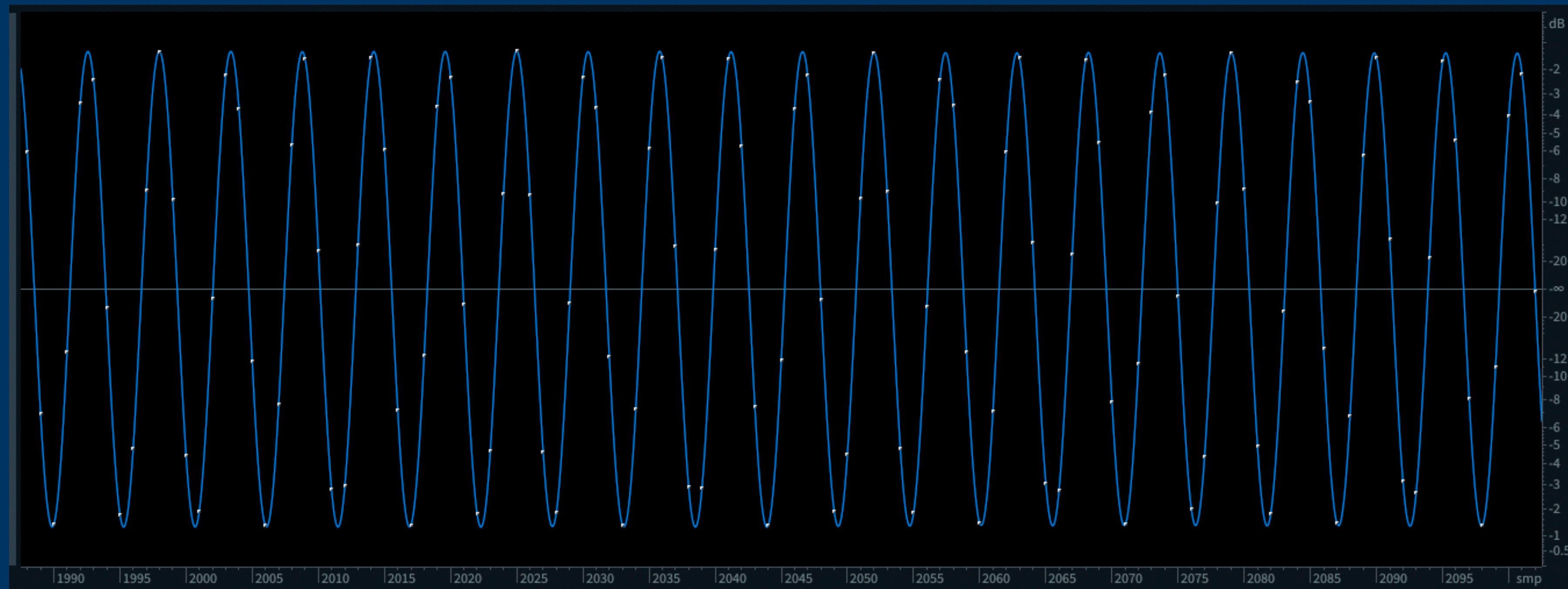
```
float peak = 0.0f;  
  
loop (int (processor.frequency / 2))  
{  
    peak = max (peak, abs (in));  
    advance();  
}
```



Envelope

Remember - the discrete sampled signal values do not correspond to signal amplitude.

Mitigate by averaging over multiple waveform cycles, choice of oscillator shape, more samples per cycle (e.g. higher sample rate, lower note pitch)



Envelope

Analysis of the sounds of interest showed attack was always 'as fast as possible' so a simple model using a linear attack was suitable

Decay Rates and Sustain Rates looked like exponential decay curves, so a 'time for -24db attenuation' was measured, and a curve fitted to this.

Envelope Key Velocity Scaling

For each KVS setting, generate an array of note amplitudes for each midi velocity.

```
void main()
{
  eventOut <- std::notes::NoteOff (0, midiNote, 0.0f);

  loop
  {
    float[128] velocityScaling;

    for (wrap<128> i)
      velocityScaling[i] = calculatePeakForVelocity (i);

    console <- velocityScaling;
  }
}
```

```
float calculatePeakForVelocity (int v)
{
  eventOut <- std::notes::NoteOn (0, midiNote, float (v) / 127.0f);

  float peak = 0.0f;

  // Skip the first part to allow the voice to settle
  loop (1000)
    advance();

  loop (int (processor.frequency / 32))
  {
    peak = max (peak, abs (in));
    advance();
  }

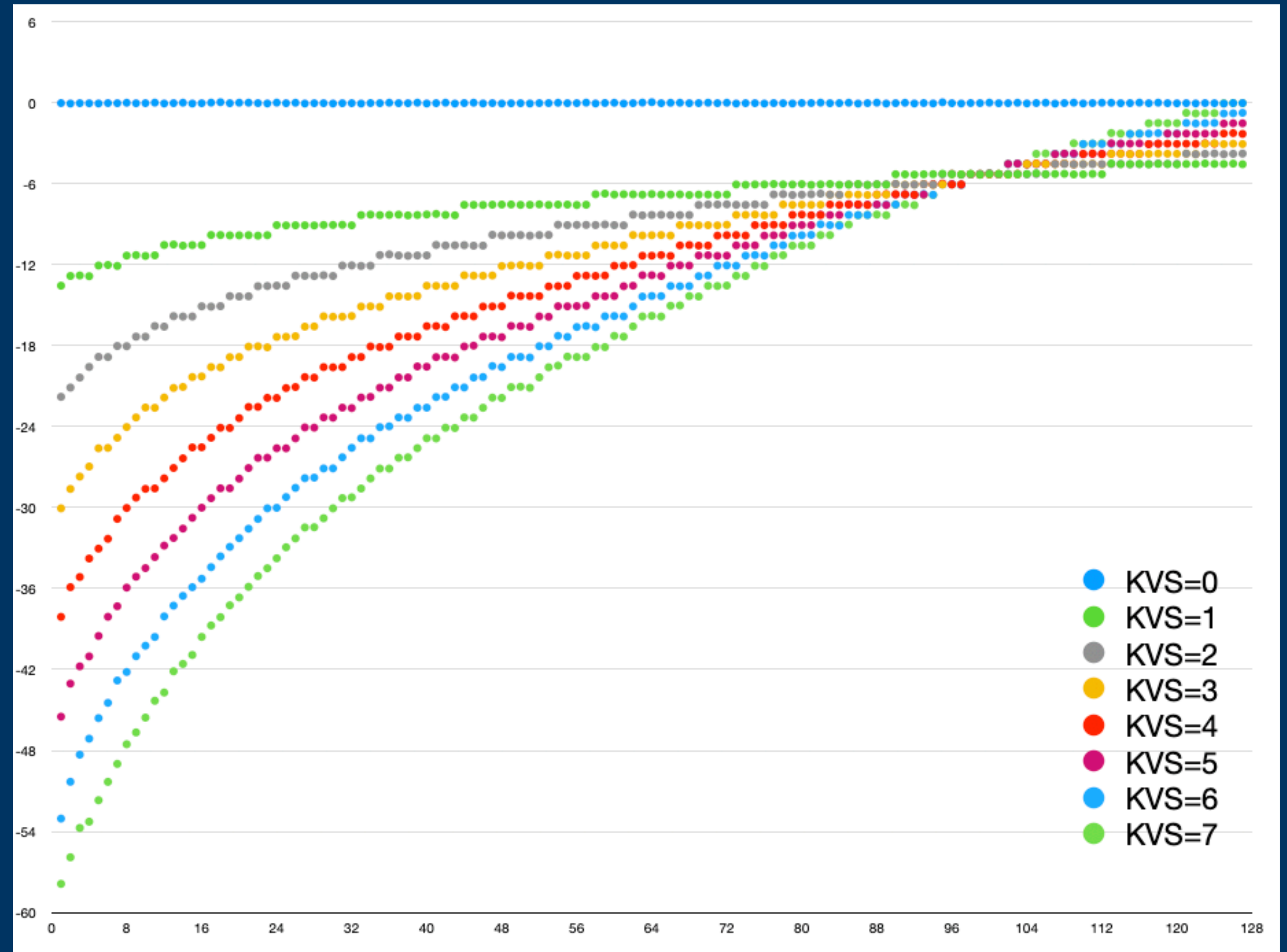
  eventOut <- std::notes::NoteOff (0, midiNote, 0.0f);

  loop (1000)
    advance();

  return std::levels::gainTodB (peak);
}
```


Envelope Key Velocity Scaling

- Discrete amplitude levels in output
- Could implement as LUT
- Instead, use fitted curves



Envelope

Envelope written as a cascade of loops, one per segment

Attack linear, other stages written as exponential decay curves

```
void main()
{
    let envelopeLimit = 0.0001f;

    loop
    {
        while (! active)
            advance();

        float value;

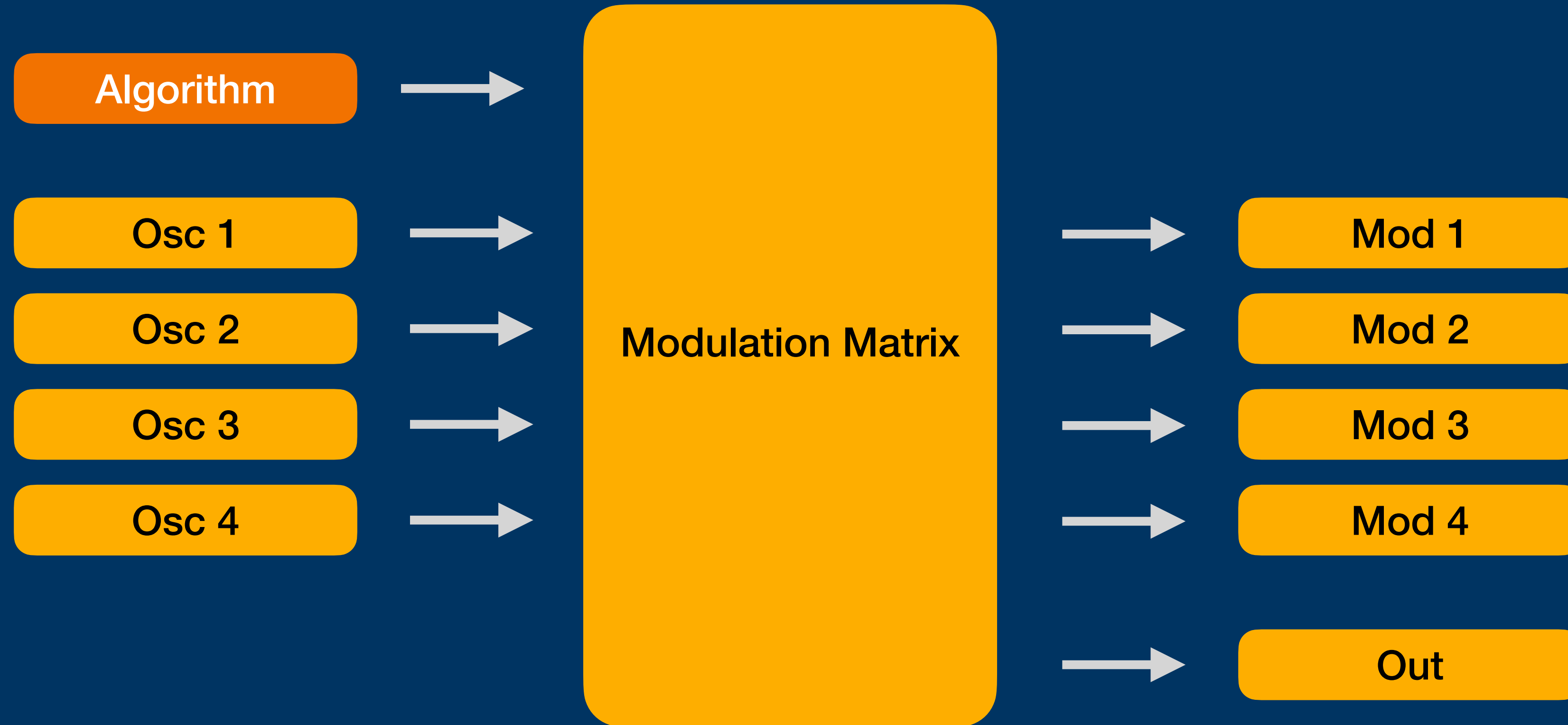
        // Attack
        while (active && value < 1.0f)
        {
            value += attackFactor;
            out <- keyScaling * value;
            advance();
        }

        if (active)
        {
            value = 1.0f;

            // Decay1
            while (active && value > decay1Target)
            {
                value *= decay1Factor;
                out <- keyScaling * value;
                advance();
            }
        }
    }
}
```

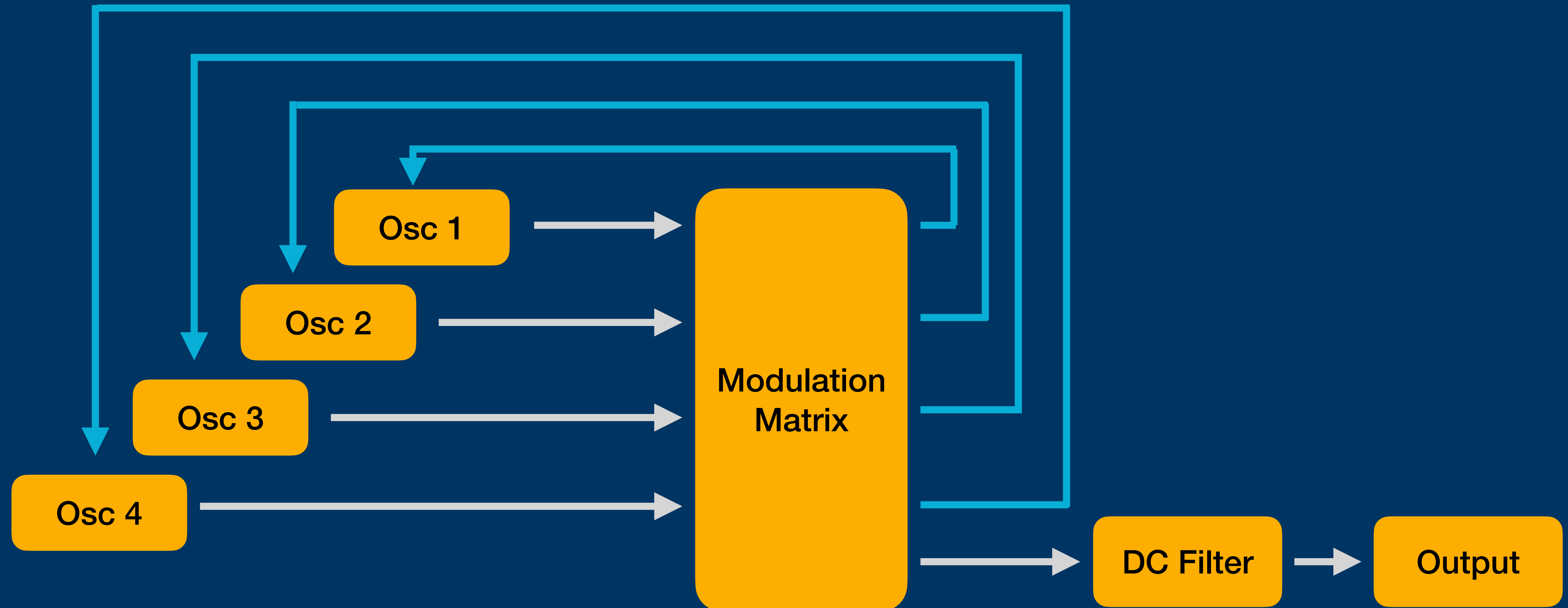
Envelope demo

Modulation Matrix



Modulation Matrix

Feedback requires delay



Modulation Matrix

Handle algorithm changes

Write a loop for each algorithm

Route the oscillators to the correct modulation destinations

Output the correct source

Could be implemented as a complete modulation matrix...

```
processor AlgorithmMatrix
{
  input event int algorithmIn;
  input stream float osc[4];
  output stream float mod[4];
  output stream float out;

  event algorithmIn (int i) { algorithm = i; }

  int algorithm = 1;

  void main()
  {
    loop
    {
      while (algorithm == 1)
      {
        mod[2] <- osc[3];
        mod[1] <- osc[2];
        mod[0] <- osc[1];
        out <- osc[0];
        advance();
      }

      while (algorithm == 2)
      {
        mod[1] <- osc[3];
        mod[1] <- osc[2];
        mod[0] <- osc[1];
        out <- osc[0];
        advance();
      }

      ....
    }
  }
}
```


Synth demo

Conclusions

- Decide what is important, and what is not. Spend your time analysing and implementing the important parts of the behaviour
- Use signal processing techniques to exercise and analyse the behaviour you care about. Write processing tools to help with the analysis.
- Divide and Conquer - work on the components in turn, refer back to the original instrument to validate your work
- Innovate. Use the components in unique ways to build new instruments.

Thank you!

Any questions ?

<https://cmajor.dev>

<https://cmajor.dev/docs/Examples/TX81Z>

<https://github.com/cesaref/TX81Z>