

# Computation as material: (meta) live coding experiences

## Abstract

What does computation sound like, and how can process be integrated into livecoding practices along with code? This paper gives insights into three years of artistic research and performance practice with Betablocker, an imaginary CPU architecture, specifically designed and implemented for livecoding purposes. It covers the themes of algorithmic composition, sound generation, genetic programming and autonomous coding in the light of self-manipulating code and artistic research practice.

Computation is odd. In fact in many ways it's one of the strangest things we have discovered, and 77 years on we continue to fail to fully grasp it's behaviour. And so, we struggle to keep unseen ramifications of our decisions under control. Faced with such a strange and yet essential beast, the only sane strategy is to try and tame it. The field of software engineering is littered with concepts such as regression testing, type checking, white rooms, sandboxing, contracts, and encapsulation – all attempts to cope with the fact that we fail to naturally understand how a mundane machine of sufficient complexity will act when it is programmed by a human.

Livecoders look this beast straight in the eyes. When a livecoding performer takes to the stage and projects her screen, she invites us to join her in attempting to understand this intricate dance between human and machine.

Beyond performance practice, livecoding is a broad field that can be viewed from many different angles. From rehearsed public performances, collaborative improvisations and individual sound explorations, to it's utilisation as a method in science, it has been

employed in a manifold of applications. All these derivations of the original theme share elements of live coding as they are featured in the Toplap manifesto draft.<sup>1</sup> (Ward et al. 2004) Livecoding attempts to treat computation as a material on it's own terms, to pick it up and try to gain understanding into what it looks(McLean et al. 2010) and sounds like.

Based on this ground, we introduce Betablocker, our contribution to the field of artistic understanding of low level computation(Howse and Kemp 2006)<sup>2</sup> which intersects livecoding practise, and brings to light many different strategies for presenting computation as tangible material – by making *process* as much part of livecoding as the code that describes it.

Built around this element is a variety of applications for testing, exploring and performing scientific as well as artistic investigations.

This paper collects experiences, challenges and insights gathered by the two authors over the course of 3 years while they designed imaginary CPU architectures and investigated the combination of established as well as emerging live coding techniques. Central to the work presented here is the use of livecoding, performance contexts and exploration strategies in order to gain understandings of behaviour emerging from simple instructions sets when given self-modification abilities. This leads us towards a kind of second-order livecoding where, beginning with a blank slate, programs written a few minutes ago interact and interfere with the livecoding of subsequent programs, or even write programs all of their own.

The paper starts with an introduction to the Betablocker core technology, followed by a background section featuring a historical overview. After that, we describe our findings and explorations with regard to Betablocker and give insights on how its behaviour

---

<sup>1</sup><http://toplap.org/wiki/ManifestoDraft>

<sup>2</sup>Viznut "Algorithmic symphonies from one line of code" <http://countercomplex.blogspot.co.uk/2011/10/algorithmic-symphonies-from-one-line-of.html>

influenced our livecoding sessions. This section is followed by a report on more than three years of performance practice in Betablocker livecoding. The paper concludes with a discussion and outlook on future work.

## BetaBlocker core technology

The core of all investigations described in this paper is the Betablocker engine, is a highly simplified multithreaded virtual machine.

The BetaBlocker engine consists of (a) a *heap* with  $2^8 = 256$  addressable slots to store an 8bit value (a byte), (b) an *instruction table* linking a byte to an atomic instruction, and (c) a number of independent *threads*, each with a (ca) an 8bit *program counter* (pC) pointing to an address in the heap, and (cb) a *stack* of eight bytes. The state vector consisting of a heap, a stack and a program counter sufficiently represents the whole Betablocker engine.

The heap is where all data is stored. Following *von Neumann architecture* (von Neumann 1993), Betablocker computation does not make any distinction between a program (i.e., values representing instructions) and data (i.e., values on which the program operates). The notions of code, data and program are therefore synonyms, however they represent semantical meaning rather than systematical/structural meaning. Particular attention has to be given to the fact that, although a specific part of data stored on Betablocker's heap is meant to be a program, by definition the possibility exists that it is altered (e.g., by a separate process running independently on the same heap or even by itself). In reverse, a critical differentiation with other virtual machines (eg. Java VM, the .NET CLR) is therefore that every possible combination of values is a valid, executable program and that each of these values is interpreted either as an instruction, an address, or a data item.

Regarding the instruction table, we differentiate between the core set which consists of 22 distinct instructions and various extensions to this set, which, depending on the actual application, introduce features such as playing a note (`NOTE`) or copying information to the output pins of an AVR microprocessor (`OUT`). An overview of the instructions is given in Figure 1. As there are no instructions, interrupts or error states telling the engine to halt, programs will continue to run until externally terminated. However, they will eventually converge into a (possibly complex) stable state, including infinite loops.

A thread executes code stored on the heap. Its program counter  $pC$  stores the current position (i.e., the instruction it is currently evaluating). The stack serves as a temporary private storage for a thread, not addressable by other threads. It is accessed by either *popping* an item from the stack to an address or the indirect location specified by an address (e.g. `POP POPI`), or pushing a value to the stack in immediate/literal, address or indirect variants (e.g. `PSHL`, `PSH`, `PSHI`). All mathematical processes operate using the stack rather than file registers, so the resulting language is close to the operation of the Forth programming language (Rather et al. 1993).

## Background and historical context

Betablocker first came to life as an attempt at a game pad programmable live coding system in fluxus.<sup>3</sup> It was inspired by a discussion on the TOPLAP mailing list about virtual machines, and by games such as *Mr Driller* in terms of colourful blocks interacting with each other and setting up chain reactions as a game mechanic.<sup>4</sup> While *Core Wars*,<sup>5</sup> – a game where contestants write simple programs that battle to gain memory footprint over

---

<sup>3</sup><http://www.pawfal.org/fluxus/>

<sup>4</sup><http://uk.ign.com/games/mr-driller/dc-14308>

<sup>5</sup><http://www.corewars.org/>

























	<b>ORG</b> 0x01	Set address start (beginning of program)		<b>POP</b> 0x09	Pop stack byte to address at location specified by argument (indirect).		<b>NOT</b> 0x11	Bitwise NOT's the top stack byte.
	<b>EQU</b> 0x02	Push 1 if top two stack items are equal, 0 otherwise. Pops 2.		<b>ADD</b> 0x0a	Pops the top 2 stack items, adds them and pushes the result.		<b>ROR</b> 0x12	Bitwise rotates the top stack byte by bits specified by the argument, (right)
	<b>JMP</b> 0x03	Jump program counter to location specified by the argument		<b>SUB</b> 0x0b	Pops the top 2 stack items, subtracts them and pushes the result		<b>ROL</b> 0x13	Bitwise rotates the top stack byte by bits specified by the argument, (left)
	<b>JMPZ</b> 0x04	Jumps if top of stack is zero. Pops 1 item.		<b>INC</b> 0x0c	Increments the top stack byte by one.		<b>PIP</b> 0x14	Increments byte specified by argument.
	<b>PSHL</b> 0x05	Push argument on to stack (push literal).		<b>DEC</b> 0x0d	Decrements the top stack byte by one.		<b>PDP</b> 0x15	Decrements byte specified by argument.
	<b>PSH</b> 0x06	Push byte at address given by the argument..		<b>AND</b> 0x0e	Pops the top 2 stack items, AND's them and pushes the result.		<b>DUP</b> 0x16	Pushes a duplicate of the top stack byte.
	<b>PSHI</b> 0x07	Push byte pointed to by byte at address given by argument (indirect)		<b>OR</b> 0x0f	Pops the top 2 stack items, OR's them and pushes the result		<b>NOTE</b> 0x17	Pops the stack and plays the current sample at a frequency specified by the byte.
	<b>POP</b> 0x08	Pop stack byte to address given by argument.		<b>XOR</b> 0x10	Pops the top 2 stack items, XOR's them and pushes the result		<b>VOX</b> 0x18	Pops the stack and uses the byte to choose the sample to play at the current frequency.

Figure 1. Betablocker instructions.

each other was also an important inspiration, both in terms of domain specific low level languages, and imaginary hardware.

From a musical point of view, Betablocker can be viewed as a tool for algorithmic composition (Maurer 1999) and simultaneous sonification and visualisation of process. Also, it is inspired by nonlinear and stochastic synthesis techniques as described by Xenakis (Xenakis 1971; Luque 2009).

Betablocker was driven by a need to escape conventions of office-based programming in livecoding performances, removing the reliance on the keyboard/mouse combination. An unintended side effect was the ability to remove the laptop entirely from focus – with a long usb cable, at times being able to join the audience with the gamepad and use the projection screen for livecoding. The language and concept of bytecode level visualisation and icon based assembler programming was initially motivated primarily for these reasons. Later, after moving to dedicated games hardware it became more obvious that the musical and artistic scope of the technique were also worth pursuing for their own reasons, leading to its later use in embedded hardware and robotics.

## Livecoding philosophy and aesthetic findings

BetaBlocker is distinct to other approaches of computational machines in several aspects: In the design of Betablocker, the ability to observe computational behaviour was an essential goal, whereas the result of the computation (and therefore process optimisation) played a minor role. Furthermore, Betablocker is implemented as a software emulation and can be modified easily. This allows for its alteration regarding introspection (by adding probes) and customisation (by altering the instruction set). Last not least, compared to common microprocessor architectures such as the x86 (Alpert and Avnon 1993) or even RISC microcontrollers (Kane 1988) such as AVR, Betablocker's architecture is extremely simplistic. Operational complexity arises from the induced data and its processing rather than from the system's architectural features.

The character of Betablocker influences not only the sonic results of a livecoding session but also its overall structuring. Due to its potential process complexity, it is e.g. possible to start with a blank page<sup>6</sup> and quickly build interacting software agents which gain sufficient complexity to result in an interesting output while remaining controllable.

We describe our findings and explorations with regard to Betablocker in the next paragraphs and give insights on how its behaviour influenced our livecoding sessions. As we used Betablocker mainly in two perceptual contrasting settings, *rhythmic*, which slowed down processing in order to perceive it and *sonic*, speeding up processing in order to hear its sonic properties directly, we differentiate in the following between Betablocker/Rhythm and Betablocker/Audio.

---

<sup>6</sup>As proposed in the TOPLAP Manifesto draft published at <http://toplap.org/wiki/ManifestoDraft>.

## Very slow computing – Betablocker/Rhythm

Betablocker/Rhythm (the Fluxus and Gameboy DS version) makes computation tangible by slowing down processes by many orders of magnitude compared to typical computing applications. The smallest musical loop in Betablocker/Rhythm is 3 instructions long – push a value onto the stack [`PSH, 123`], play it as a note [`NOTE`], jump back to start [`JMP, 0`].

In order to play this at 120 BPM it will need to run at 6 cycles per second, the machine I am typing on peaks at  $2.8 \times 10^9$  cycles per second, which is 466 million times faster than Betablocker/Rhythm. This makes computation audible as poly-rhythmic techno music, and simultaneously visible as the thread's positions are displayed on the livecoding interface synced with the audio along with the entire heap's memory contents. The memory is displayed on a  $16 \times 16$  grid, in order to maximise the visibility of the state and align with hexadecimal notation.

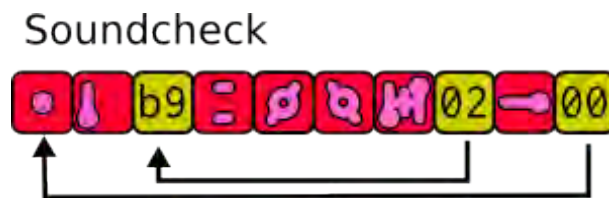
Performing with Betablocker/Rhythm consists of starting with empty memory and manually entering programs byte by byte that trigger synthesis patches while modifying themselves and each other. Bytes used as address locations are visualised literally as pointers - arrows indicating the address in memory they are referencing. Eventually the quantity of self replicating fork bombs, sequencers interpreting memory as percussion triggers and programs busily reversing the contents of memory reaches a level where the performer is confronted with a kind of 'Turing soup' which is writing it's own code. At this point the decision is whether to take advantage of this situation or attempt to gain control over it. Taking advantage of it involves looking for interesting sections of code that emerge and manually modifying them, while the best strategy for taking back control is attempting to find a space to write a program that gradually clears memory so you can start again. In most successful performances both options were repeatedly taken over the

course of the concert.

It is this gradual cessation of control, with immediacy and totality of the understanding (or at very least display) of the machine that sets this apart from traditional livecoding performance, where the high level description of a process is the only connection between all, audience, performer and the process creating the music they hear.

### *Code patterns for livecoding*

In order to further illustrate the kinds of programs and livecoding strategies, we list concrete 'code patterns' which are used in common live coding situations. Firstly, a program which will play all the sounds in a sound bank at all frequencies available. Multiple threads can be issued to play this program at the same time to provide more complexity. Values interpreted as pointers are visualised as arrows:



*Figure 2. Sound check*

In Betablocker/Rhythm the beat is always locked to the instruction cycle, so faster melodies or beats are a matter of optimisation. Fast inner loop programs can be used for playing sounds while another slower program modifies the first by overwriting parts of it while it's running, as in this example.



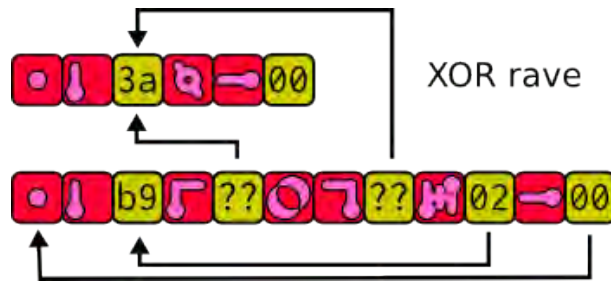


Figure 3. XOR rave

The following programs are both 16 bytes long, and require 3 concurrent threads to be running over different parts of the program. From left to right, the first thread loads an immediate byte and plays it as a sound repeatedly, the second one increments the byte that the first is loading, and the third periodically resets the byte to a constant value.

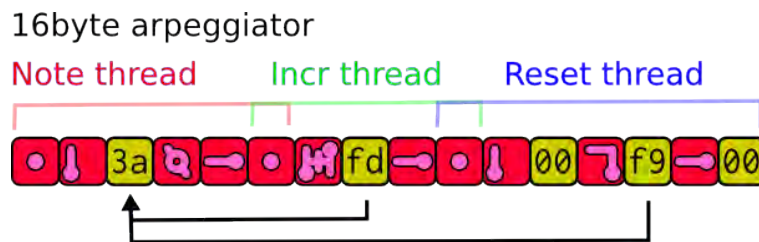


Figure 4. A simple arpeggiator for Betablocker/Rhythm.

The first example simply plays the byte as a note - resulting in an arpeggiator style melody. After writing the code, the performer can change the speed of each of the three threads to get more variety. For example, the slower the resetting thread runs, the longer the pattern will be before it repeats.

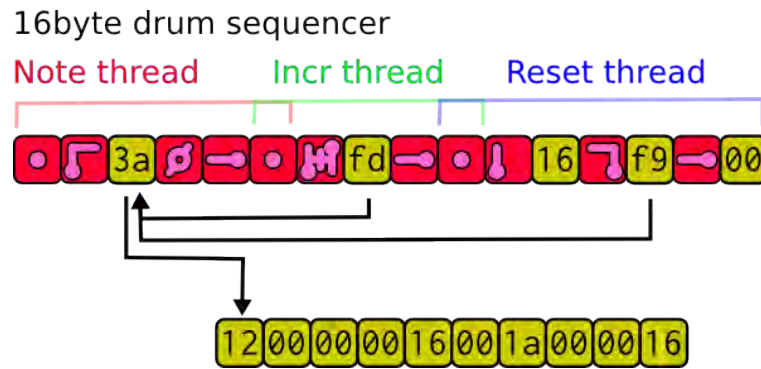


Figure 5. A sequencer for Betablocker/Rhythm.

The second example is very similar, but is slightly modified to play a sequence of different sounds rather than notes, in order to become a percussion sequencer. The second instruction is changed from a "push literal" to a "push contents of address" which results in the program playing a part of memory as a sequence. We can control the start address location with the reset byte, write directly to this part of memory, and modify the other thread's speed to get even more complexity.

Other types of program alter the memory rather than focus on sound. For example given two addresses, a program may increment one and decrement the other, copying bytes from the source to the destination, which results in the entire CPU memory to be reversed. Another type of program is a self replicating fork bomb, which copies its code to other locations and gradually captures other threads into it's replication.

### *Use of Genetic Algorithms as creative augmentation in livecoding*

The same features that makes the Betablocker approach suitable for exploratory artistic and musical applications (lack of error states and code/data isomorphism) also make it a highly convenient environment for employing evolutionary strategies.

The use of Genetic Algorithms (Barricelli 1963) in this livecoding case is different

from their traditional use. In the case of Betablocker/Rhythm the desire to use genetic algorithms arose from the need to discover and learn new, optimal coding patterns for use in livecoding performances, as a way of escaping a feeling of being 'stuck in a creative rut'. The use of a genetic algorithm in this case is to provide *new ideas for artistic use* - not final programs for their own sake.

The success of an algorithm used for livecoding is to some extent how easy it is to change, how malleable it is, and one of the features of evolutionary strategy is that programs are automatically converged on that may present extremely unorthodox approaches - especially when optimisation is linked to fitness criteria and architectures are used where self modification is possible. In this case we want to unpick successful individual programs in order to employ them musically in hand written code.

In a genetic algorithm, in order to select promising programs from a population, we need a way to judge fitness. In this application we are creating lists of note numbers, and in order to create a fitness metric we favour patterns that have unique notes (to avoid repetition) but also look for a rhythmic nature by measuring them with this function:

```
(defn freq [l n]
  (defn _ [c]
    (cond
      (>= (+ c n) (count l)) 0
      (= (nth l c) (nth l (+ c n))) (+ 1 (_ (+ c 1)))
      :else (_ (+ c 1))))
    (_ 0))
```

Which looks for repeating values spaced apart by "n" positions, so:

```
(freq '(100 1 2 3 100 4 5 6 100 7 8 9 100) 4)
```

Returns 3, as it finds three pairs of equal values (100) distanced by the specified 4 positions.

With this fitness function:

```
(+ (* 50 (count (num-unique res))) ; unique notes are very good
  (freq res 4) ; equal notes every 4 beats are good
  (freq res 6)) ; equal notes every 6 beats are good
```

we favour rhythmic components of 4 or 6 beats and boost the uniqueness score by multiplying it by 50. Here is a resulting evolved individual program found after several thousand generations:



Figure 6. A program evolved by genetic programming for Betablocker/Rhythm.

It creates a complex pattern efficiently in four different ways: (a) a descending scale using the stack to store the current position, (b) a rising scale by self modifying address 12 (the first 00), (c) playing *it's own code* as a pattern of notes by using address 12 as a pointer, and (d) playing the bitwise NOT of it's own code at the same time. The resulting note numbers: 0 0 232 255 23 1 232 254 13 2 242 253 22 3 233 252 7 4 248 251 12 5 243 250 17 6 238 249 20 7 235 248 12 8 score poorly for rhythmic patterns, but it is probable that this surprising local maxima was found by the early influence of the rhythmic fitness criteria on its ancestor populations.

The evolved programs provided many new ideas used in subsequent livecoding performances, including self playing programs like the one above, greater use of stack (evolved programs showed less reliance on storing values on the heap, leading to more robust programs) and more importantly, a general reappraisal of the potential scope of programs possible with this restricted instruction set.

Following this work, Betablocker was ported as *Spork Factory* to the ATtiny85 processor as part of an ongoing investigation of physical, livecodable objects and robots. The instruction set was extended by IN and OUT instructions for accessing the pins of the microcontroller. This enables audio programs to be evolved for the processor, using FFT analysis on the resulting sounds as the data to provide fitness metrics for.

## **Sound computing – Betablocker/Audio**

One goal of the investigation into Betablocker was to gain new insights into the aesthetics of typical data processing. How dynamic is the behaviour of a chip when it runs an algorithm? What does computation sound like and is the variety of programs reflected in a diversity of sounds? To get answers to these questions, we facilitated the method of introducing *pointsOfTouch* research supported by sonification (Bovermann et al. 2011). Just as it is possible to probe hardware circuitry with e.g. the *Audio Sniffer*<sup>7</sup>, we intended to probe Betablocker engines while executing their heaps.

To be able to capture signals in the audible range, the Betablocker/Audio variation runs at rates around typical audio sampling rates. Because of the intention to probe an otherwise independent system, no separate instructions or other structural enhancements that would influence its computational behaviour were introduced to the Betablocker engine. Instead, we added *pointsOfTouch* to meaningful parts like the stack and the

---

<sup>7</sup><http://www.openmusiclabs.com/projects/audio-sniffer/>

program counter.

The implementation of Betablocker/Audio was done in the SuperCollider programming environment.<sup>8</sup> To facilitate typical usage, we decided to provide two different interfaces: a demand-rate UGen (`DetaBlockerBuf`) exposing the topmost value of the stack by means of SuperCollider's demand chain implementation, and a multi-out version (`BBlockerBuf`) that exposes the position of the program counter as well as the complete stack. A basic example on how they can be accessed is given in the following listings.<sup>9</sup> By applying a `LeakDC` operation to the stack values, DC offsets that could harm the speaker system are removed.

```
x = { // BBlockerBuf example
    var signal = Demand.ar(
        Impulse.ar('bbRate'.kr(22100)),
        'reset'.tr(0),
        DetaBlockerBuf('heapBuf'.kr(0))
    );
    LeakDC.ar(signal);
}.play;
```

```
x = { // BBlockerBuf example
    var pC, stack;
    #pC ... stack = BBlockerBuf.ar('bbRate'.kr(22100),
        'heapBuf'.kr(0), 'startpoint'.kr(0));
    stack = LeakDC.ar(stack);
}
```

---

<sup>8</sup>The source code is available as a plugin to Supercollider from <https://github.com/supercollider/sc3-plugins>.

<sup>9</sup>It is assumed that the heap to be operated on is pre-loaded into a Buffer object.

```

        // use information on program counter to place sound output
        Splay.ar(stack, spread: 0.1, center: pC);
    }.play;

```

### *Noise aesthetics – on texture and rhythm*

Although the system is deterministic, i.e., its outcome can be calculated given that its state is known. Due to its complexity a valid and, as it turned out, fruitful approach is to work with randomness and empirical observations rather than with analytical approaches.

To investigate into this, we implemented a cluster generator for synthesising Betablocker operations of randomly generated heaps:

```

{var srcs = ({BBlockerBuf.ar(
    s.sampleRate * 0.5,
    BBlockerProgram({rrand(0, 24)}!256).makeLocalBuf
) [1]}!20);
    Splay.ar(srcs.collect{|src| LeakDC.ar(src)}, 1, 0.4);
}.play

```

After extensive listening, we found out that they often sound like airily high-pitched and infinitely sustained sound clouds, broken up occasionally by a rhythmical pattern.

Typically, they settle either to a constant pitch or to silence after about 0.1 seconds. In the long run (range of 4 minutes), such clusters settle into a local stability with much less high frequencies than at the beginning. The first 1000 computation steps could therefore be called transients, whereas the longterm variation can be considered a timbral decay.

## *Lessons learned from manually programming sonic structures*

When interacting with a low-level computational system, it is crucial to understand it's inner structure and get an idea on how it operates. It is possible to gain such knowledge by manually programming (simple) algorithms in the system's native language and observe its resulting behaviour. In order to understand the Betablocker engines specifics regarding it's sonic characteristic, we hand-crafted algorithms using the SuperCollider implementation to render given waveforms as close as possible.

We found that a sawtooth wave can be modelled by setting the initial 4 bytes of the heap to `[ORG, INC, JMP, 1]`. When computation starts with `pC = 0`, the remaining 252 bytes can be set arbitrarily as they will never be reached by the engine. Rendering an impulse at nearly Nyquist frequency can be achieved by executing the program `[ORG, NOT, JMP, 1]` whereas a pulse wave can be implemented with `[ORG, NOP, ... , NOT, JMP, 1]`. The number of `NOP` here determines the pulse width. Similar to the sawtooth example, the values of the remaining bytes in the heap do not affect the program execution.

We developed all the example codes for classic waveforms by exploratory livecoding using the helper class `BBlockerProgram`, which implements (among other functionality) a `play` and a `plot` method to play, respectively display, a given Betablocker heap. An overview of the rendered signals can be found in Figure 7.

While investigating possibilities to implement a sawtooth with adjustable slope, we came across a more complex structure that resembles such a sawtooth combined with a pulse: `[ORG, PSHL, 6, ADD, JMP, 1, <slope>]`. Here, the decision to add a probe to the stack rather than implementing additional side-effect instructions came into play: the pulsewave in the program's output (i.e., its stack's top value) is caused by an



operation that seems necessary to implement the adjustable slope with a given "data" item. The shape of the waveform is therefore limited due to the way the sound signal is derived from the Betablocker engine, namely probing the items on one of its functional parts. Note that the slot in which the slope parameter is stored will never be reached by the program counter and can therefore be used as a "safe data storage" for the slope parameter.

Another finding of the livecoding-supported Betablocker exploration linking the results of the noise exploration described in the previous paragraph with explicitly writing code was the sonic outcome of a combination of random "data" and a specifically designed algorithm that interprets it. Given the program `[NOP, ORG, NOP, NOP, PSH, 245, PSH, -1, SUB, DUP, POP, -1, POP, 1, JMP, 1]` followed by random numbers, the topmost element in the stack varies according to values addressed in previously pushed values. Every set of "data" then sounds different, however, the sounds share a "root note" that depends on the execution rate. Note that the "program" itself is as well part of the data and can be addressed, too. As the point, to which the stack pops its topmost values, is depending on its state, it might as well destroy the original program, leading to unstable (yet maybe refreshingly new) behaviour.

To explore such variations in SuperCollider, we introduced the flag `fillUpRandom` to the helperclass `BBlockerProgram`. When set, the heap's remaining slots are filled with random values whenever a Betablocker engine is initialised with the given program:

```
a = BBlockerProgram([\NOP, \ORG, \NOP, \NOP, \PSH, 245,  
  \PSH, -1, \SUB, \DUP, \POP, -1, \POP, 1, \JMP, 1]);  
a.fillUpRandom = true;  
x = a.play(1000, leak: true);
```

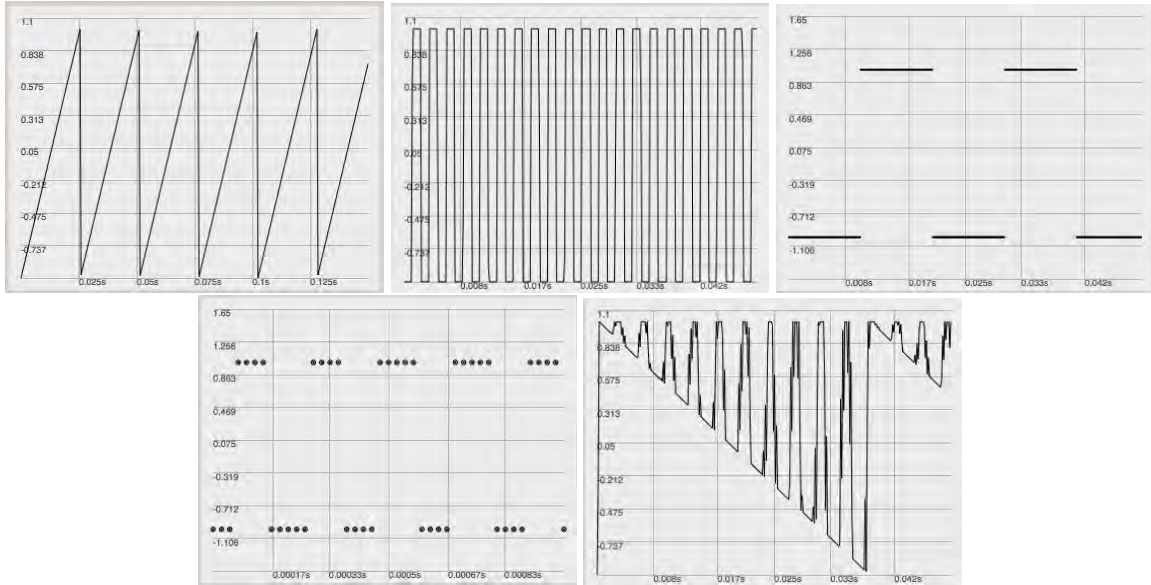


Figure 7. Manually generated waveforms: sawtooth, impulse, long pulse, short pulse, saw impulse.

### Classic synthesis techniques adapted to Betablocker/Audio

Classic signal synthesis techniques such as additive synthesis, frequency modulation and amplitude modulation combine several sound generating modules in order to create new sounds. The result depends highly on factors like the relation between the components, the signal characteristics of the base oscillators and the parameter range in which the oscillators control each other.

Next to being able to modulate its output amplitude, one feature of the Betablocker/Audio implementation in SuperCollider is that it allows to alter its calculation rate on a per-cycle basis. This makes it possible to add dynamics not only in amplitude but also in pitch. In several research-oriented livecoding sessions, we explored the sonic character of assembling Betablocker engines according to the mentioned classic synthesis techniques.

Figure 8 shows block diagrams for additive synthesis, frequency modulation and amplitude modulation of two Betablocker/Audio modules. While the implementation of

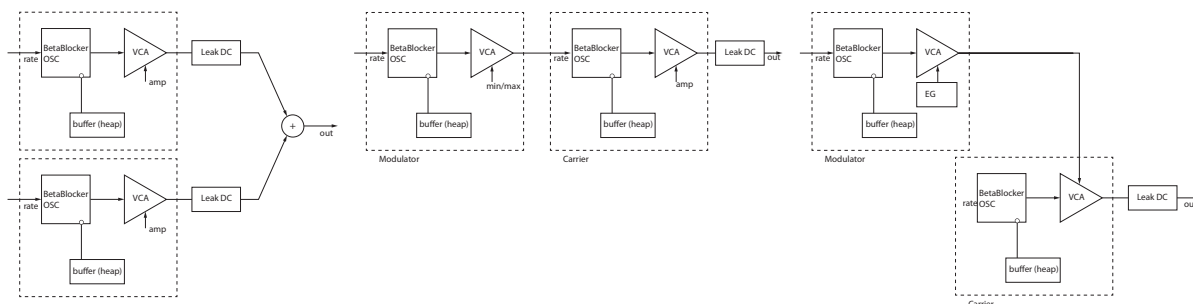


Figure 8. Block diagrams of classic synthesis techniques adapted to Betablocker/Audio.

these synthesis techniques is straight forward, the livecoding experimentation revealed several aspects worth mentioning:

Adjusting the rate of the modulator in FM/AM synthesis to be below audio range results in sequencer-like monophonic synth patterns that are mostly pseudo-repetitive. I.e., they start with a chaotic movement that is followed by repetitions of a possibly slowly changing pattern. This behaviour extends significantly the sonic qualities of a Betablocker/Audio engine, a feature Till used extensively in the livecoding performance at the SuperCollider symposium in London (see below report on Betablocker livecoding practice for details).

As known from classical FM synthesis, modulation rates in the audio range result in a broader frequency spectrum. Linking the rates of all used Betablocker engines results in harmonic additions to the (noisy) spectra. The sonic relation between different Betablocker programs seems to follow a specific rule, however, it does not seem to be a straight forward. Another way to generate aesthetically interesting patterns with complex rhythmical interrelations can be achieved by letting both engines run on the same heap.

To summarise, the application of classic synthesis techniques allows for additional control and variation of the sonic output and, depending on the actual code on which the engines are running on, may even induce musical structures.

## **Betablocker and livecoding performance practice**

Over the course of the last 3 years, we used Betablocker in a wide variety of livecoding practice, including both public performances and research-oriented investigations. The result of the research process is reported in the above section. Subsequent, we describe relevant Betablocker-based public performances in their order of appearance.

### **Piksel Festival Bergen [10.2006]**

The very first performance of Betablocker in the Fluxus environment took place in 2006 at the Piksel open source media arts fesival in Norway. This was a 20 minute solo livecoding performance using a gamepad as the only input mechanism. Subsequently Betablocker was used in a series of performaces in London, both solo and in collaborative musical live sessions as part of the group 'slub'<sup>10</sup>.

### **4for8, Aalto Media Lab, Helsinki [4.2011]**

After some time experimenting with Betablocker using Fluxus, during 2010 the engine was ported to the Gameboy DS system via the homebrew development toolchain. This gaming platform was chosen for it's stylus input mechanism, sound co-processor and extremely fast dedicated graphics hardware. The touch screen provided improved input, over the previous gamepad mechanic, with new features such as drag-drop local frame address lookup. It was first performed in Helsinki in 2010, using a web camera to project the screen, stylus and fingers of the livecoder. This system and software is now used as a way of explaining and demonstrating livecoding, particually to children, who are familiar with the Gameboy DS.

---

<sup>10</sup><http://slub.org>

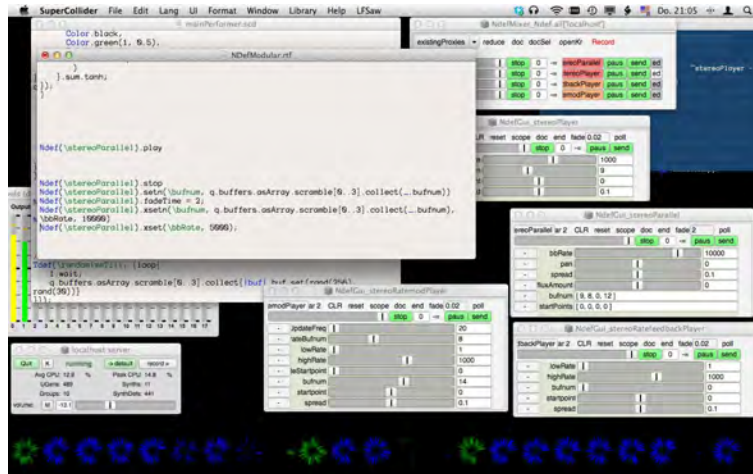


Figure 9. Till's livecoding environment as used at the SuperCollider symposium 2012, London. In the bottom part, the heaps are visualised.

## SuperCollider symposium London – livecoding and visuals [4.2012]

For the livecoding gig at the SuperCollider Symposium 2012, London, the two authors performed together for the first time. As they are not located at the same place, they developed a remote rehearsal routine in which each performer prepared his part as a sound recording and the other performed on top of it. At the same time, both authors prepared their performance setups.

Till's aesthetic goal for the performance was to play poly-rhythmical layers of coloured, noisy sounds with a setup purely based around his SuperCollider Betablocker implementation. At the same time, he aimed for a certain amount of transparency, i.e., it should be possible to not only see the changes he is making to the code but also how Betablocker engines alter the heaps' content. This meant (a) to develop a visual representation of the heaps he was playing, (b) to design a textual helper environment for his livecoding, mainly providing text blocks, and (c) to pre-select heaps that he used as "raw materials" for his part of the performance. Figure 9 shows a screenshot of Till's setup.

This improvisation with Till allowed David some freedom from the synchronised



*Figure 10. Performing at the SuperCollider symposium 2012, London.*

lockstep of his solo performances, and this concert was the first to experiment with independant speed settings for the different threads. Having control over this meant greater breadth of possible higher level patterns, being able to speed up or slow down different parts while listening to Till's music provided greater control needed in collaborative performance.

This performance using Betablocker DS was also the first to use a new synthesis engine written specifically for the ARM7 sound streaming co-processor on the DS, parts of which were written in hand optimised assembly allowing for greater sonic possibilities with an unfortunate corresponding increased battery consumption. Figure 10 gives an overview of the performance situation.

## **Oulipop – translation of codes [5.2012, 4.2013]**

This piece for two performers and one laptop was conceptualised and realised by Till Bovermann and Sara Hildebrand Marques Lopes. By manipulating texts according to Oulipo rules (Mathews, Brotchie, and Queneau 2005), it deals with the relation between writing and manipulating text, code and sound.

While editing text in one of two text panes, each keystroke also triggers a process that



Figure 11. The Oulipop performance environment consists of a laptop with two typing areas.

translates the current text's characters into a set of Betablocker/audio instructions. The correspondence of characters to instructions is defined by connecting the letter frequency for the performance language (here German) to values representing the influence of each instruction onto the rendered signal (e.g., NOT has a high value as it inverts the current output whereas NOP has a small value as it does not change the output). Each line in the editor is treated as input to one Betablocker module. The output of these Betablocker engines is organised in an FM-like structure. The resulting sound, an ever-changing blend of noisy texture and rhythmical patterns, influences the performers' decisions on how to further alter the text. Over the course of the performance, an initially meaningful text turns into nonsense before it eventually settles into a new and often surprising semantical meaning.<sup>11</sup>

## Meta-Discussion

In this paper, we presented our work on and with Betablocker, an imaginary CPU architecture, specifically designed and implemented for livecoding purposes. Livecoding with Betablocker can be undertaken in various forms and on different levels; ranging from assembler-level programming to recursive livecoding based on the engine's capability of

<sup>11</sup>For more information and a documentation video, see <http://www.tai-studio.org/oulipop/>.

self-modification up to genetic algorithms and structural programming. We shared our insights on the system-inherent aesthetic possibilities and presented our performance practice around livecoding with Betablocker in its various flavours.

Over the course of the project, our preconception was that the beast of computation with all its complexity was not to be tamed. So is it with Betablocker: Although the output of a Betablocker engine is deterministic, it still remains a mystery for the human mind, it is *pseudo-indeterministic*. However, human desire for understanding leads us to suspend our disbelief and attach higher level meaning to a systems reaction as a way to gain tangible understanding. In this light, we recognise that, with its ability of self-modification and inherent complexity, Betablocker can be viewed as a companion for livecoding that one has the opportunity to get to know, collaborate with and, sometimes, work against. In our endeavour to find out specifics of digital processing and using Betablocker in both performances and the object of investigation itself, we saw it unfolding as a digital object with an intrinsic character.

In the future, we imagine Betablocker being implemented as a part of a wide range of mostly artistic or educative systems, and being at the heart of an extended set of interpretations of (pseudo-)historical breakthroughs in computation such as mastering the P-NP problem, artificial intelligence, neuroinformatics or theoretical swarm robotics.

## References

- Alpert, D., and D. Avnon. 1993. "Architecture of the Pentium microprocessor." *Micro, IEEE* 13(3):11–21.
- Barricelli, N. 1963. "Numerical testing of evolution theories." *Acta Biotheoretica* 16(3):99–126.



- Bovermann, T., J. Rohrerhuber, and A. de Campo. 2011. *The Sonification Handbook*, chapter Laboratory Methods for Experimental Sonification. Berlin, Germany: Logos Publishing House, pp. 237–272. URL <http://sonification.de/handbook>.
- Howse, M., and J. Kemp, (editors) . 2006. *[the] xxxxx [reader]*. Mute.
- Kane, G. 1988. *mips RISC Architecture*. Prentice-Hall, Inc.
- Luque, S. 2009. “The Stochastic Synthesis of Iannis Xenakis.” *Leonardo Music Journal* :77–84.
- Mathews, H., A. Brotchie, and R. Queneau. 2005. *Oulipo compendium*. Atlas Press.
- Maurer, J. 1999. “A Brief History of Algorithmic Composition.” URL <http://ccrma.stanford.edu/~blackrse/algorithm.html>. [Accessed: Jan. 19, 2013].
- McLean, A., D. Griffiths, N. Collins, and G. Wiggins. 2010. “Visualisation of live code.” *Proceedings of Electronic Visualisation and the Arts 2010* .
- Rather, E., D. Colburn, and C. Moore. 1993. “The evolution of Forth.” In *ACM Sigplan Notices*, volume 28. ACM, pp. 177–199.
- von Neumann, J. 1993. “First Draft of a Report on the EDVAC.” *Annals of the History of Computing, IEEE* 15(4):27–75. Originally published in 1945.
- Ward, A., J. Rohrerhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander. 2004. “Live algorithm programming and a temporary organisation for its promotion.” *Proceedings of the README Software Art Conference* .
- Xenakis, I. 1971. *Formalized Music*. Bloomington: Indiana University Press.