SUPER MUSCLE

Summoning Percussive Music Using Live Electronics

An Embedded Platform for

Real-Time Composition and Improvisation

using a Drum Set and Electronics

Description, Setup and Manual

Version 0.2.0

github.com/dunland/muscle

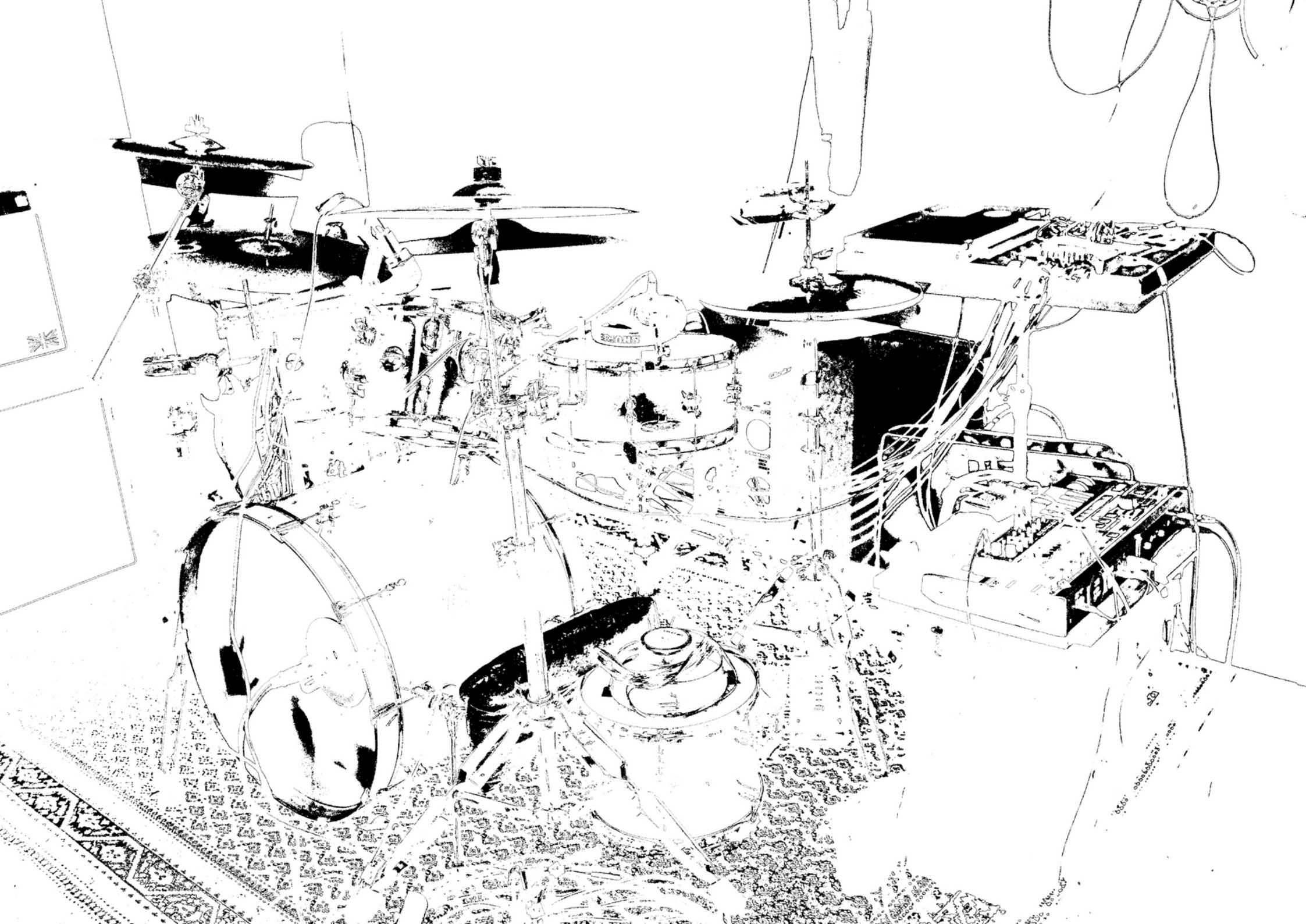

Table of Contents

1 Introduction

1.1 Preface

As an experimental musician, I love weaving different sonic textures together. The more tightly they interlink, the greater the tickling in the ears. As a drummer, I like sophisticated post rock music, beat-tight funk genres and experimental polyrhythmics. In this project, I decided to combine the sounds of an acoustic drum set with electronic sounds of physical Synthesizers. The aim was to create a system, sophisticated enough to detect rhythms and trigger sounds, while the drummer does not have to care about neither turning knobs on external devices nor a backtrack. The only human-machine interfaces should be the sticks and the drums. Plug&Play is an essential requirement that I have towards the system I built, and it should prevail over all extensions I will probably add to it.

The guitar world is dominated by legions of effect pedals, bringing us mostly single-function-effects like delays, reverbs, tremolos, choruses and distortions. I would like to extend this idea to the world of percussion. SUPER MUSCLE is a first approach to explore what an equivalent drum effects box would require. All interfaces should be usable by the performer in an intuitive manner.

Another reason for the creation of this platform is that humans sometimes are relatively slack companions, when it comes to joint music making. Talking about music is enormously hard (unless you're a professional, which I'm not), and sometimes humans don't show up for rehearsals. This is a try to play solely with machines – hoping that, in the end, I'll be able to perform with other experimental musicians, again.

As for this text, it is meant to be a manual for the setup I developed. If you are interested in certain conceptions I included in my work, you can directly go there and start reading, and I hope it will make sense. Likewise, it is alright to skip certain passages if you know the basics or if they bore you.

Before I dive into the tech-y aspects of explaining what I built, there is a short chapter (1.2 Topographies of Time) on some music-philosophical ideas that inspire my working with sound as a student of *Digital Media* and that also defined my way of programming. In short, rhythm, as in *re-occuring events in time*, is the most fundamental element in sound and in music. Taking place at different time levels, it constitutes everything we perceive as sound, from high-pitched noise at the upper limit of hearing, over tonal frequencies, to the inaudible time spans of composition. Above all, it was Curtis Roads' *Microsound* that has shaped my understanding of the different *Time Scales of Music* enormously – and that has shown me, which role rhythm – or rather: periodicity – plays to compose on different levels, what we conceive of as Music.

1.2 Topographies of Time

The general compositional and technical ideas of the code are inspired by the idea of *Time Topographies*, i.e. characteristics and intensities of events happening over certain time intervals. While programming the code for MUSCLE, I realized that, when dealing with rhythms, the same mechanisms of event detection apply to all time domains. Rhythm has the implicit quality of presenting discrete patterns on many time layers; it does not merge into continuous experience, like repeating oscillation with a wavelength of greater than about 50 ms (20 Hz) does by transforming to *tones.* (Maybe there is the exceptional case of a very continuous, *topographically flat,* composition, inducing a feel of *suspension* or *time-loss* of several minutes).

The time intervals defining the range of a Time Topography may be of any length; time frames applying for the MUSCLE platform extend from a few milliseconds (layer of stroke sampling) over several seconds (rhythmic layer and beat layer) to minutes (compositional layer). Beyond that, there are many more time domains shown in Figure 1, taken from Curtis Roads' book *Microsound* (MIT Press, 2001).

In a drum beat, the topographies could be defined by the intensity of a drum hit, leading to the drum head oscillating with higher amplitudes. On the next level above, the strokes would shape the topography over the course of a bar with their occurrence in the rhythm. Looking at several bars, then, the number of strokes (maybe even of all instruments combined) regularly occurring at the same beat positions would describe the elevation of the single peaks in an overall beat topography.

(Learn more on the implementations of these ideas in chapter 2.5 Topography class.)

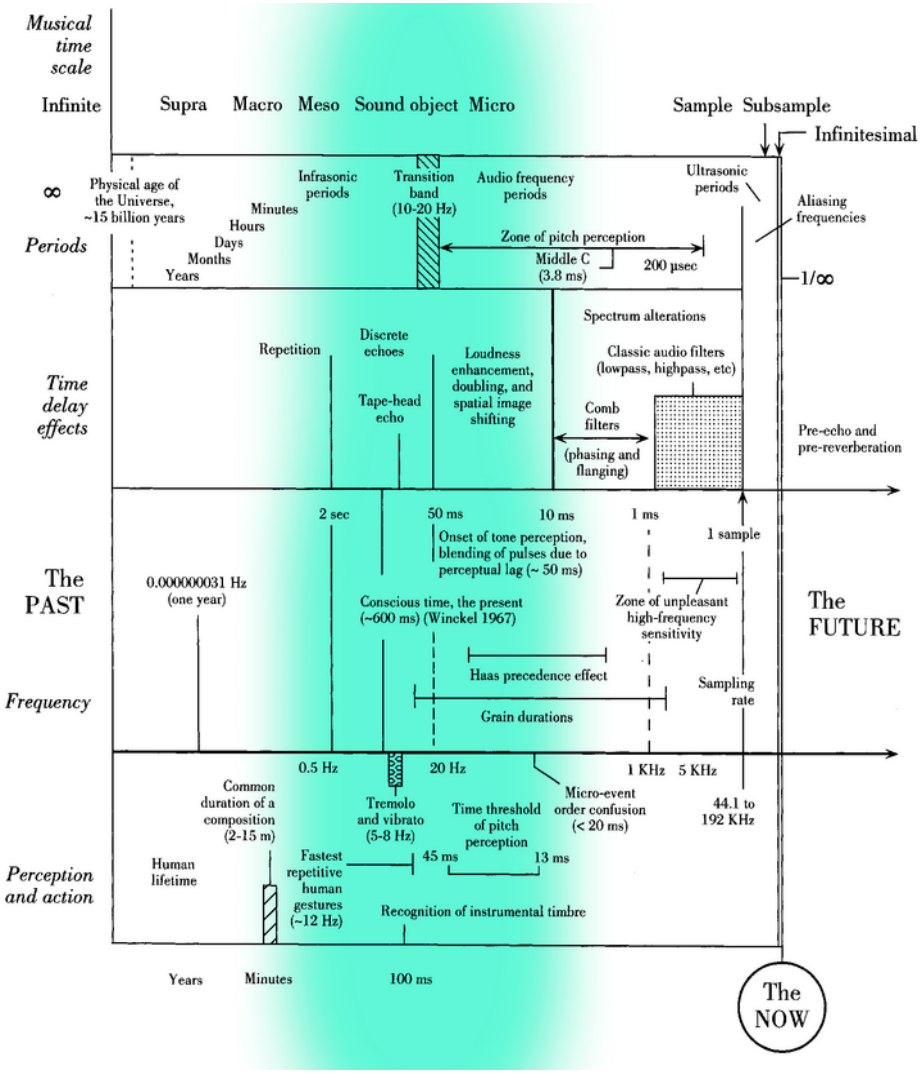

Figure 1. The Time Scales of Music, taken from Curtis Roads Microsound (2001), p. 5. The colored area shows the time domains over which the MUSCLE platform extends.

1.3 Used Hardware and Software

The core device I'm using to run my program is a Teensy 3.2 microcontroller (as of today, Teensy 4.1 unfortunately does not support the *AltSoftSerial* library, which is crucial for the use of the *sparkfun Tsunami* device I am using, too). That's where all the wires eventually go. The contact microphones each are connected via a little resistor circuit that produces a nice digital signal when pressure is applied to the piezo elements. All of the circuits are hidden in a little box with MIDI contacts attached for the connection of external MIDI devices.

Furthermore, I installed a foot switch to have some additional control over the code, while it is running. The foot switch basically separates a digital input pin (used as a pull-up pin) from ground and is read once each program cycle.

The *sparkfun Tsunami Super Wav Trigger* is an embedded audio player that can play up to 32 mono tracks (18 stereo tracks) simultaneously. I use this device to enable the integration of rhythmic field recordings to my performances. This device is not crucial for the setup, but can produce some more experimental Audio experiences.

The synthesizer I am using for prototyping, testing and performing, is my good old friend, the *microKORG*. A synthesizer is required to make any sense of my Code as is, because I did not implement any sound generation (yet).

The Code provided in the *git* Repository (p. 28) in the folder **`Code/teensy`** is organized using *platform.io*, a tool for embedded systems programming. I used this platform together with the programming IDEs *Visual Studio Code* and *Atom*, because they enable a better workflow than the standard *Arduino IDE*, and they support the use of project-associated C++ header files, which is convienent for a project of this size.

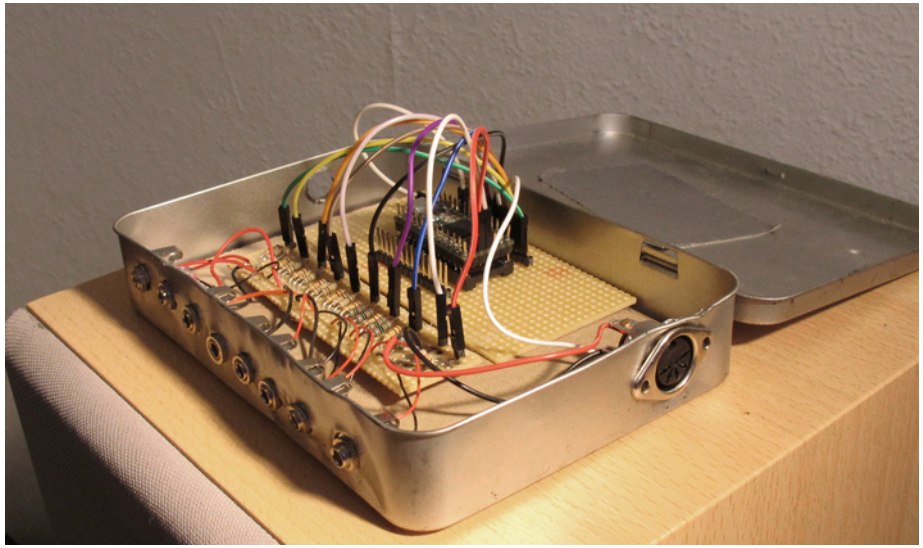

Figure 2. All electronics are placed in a little tin box.

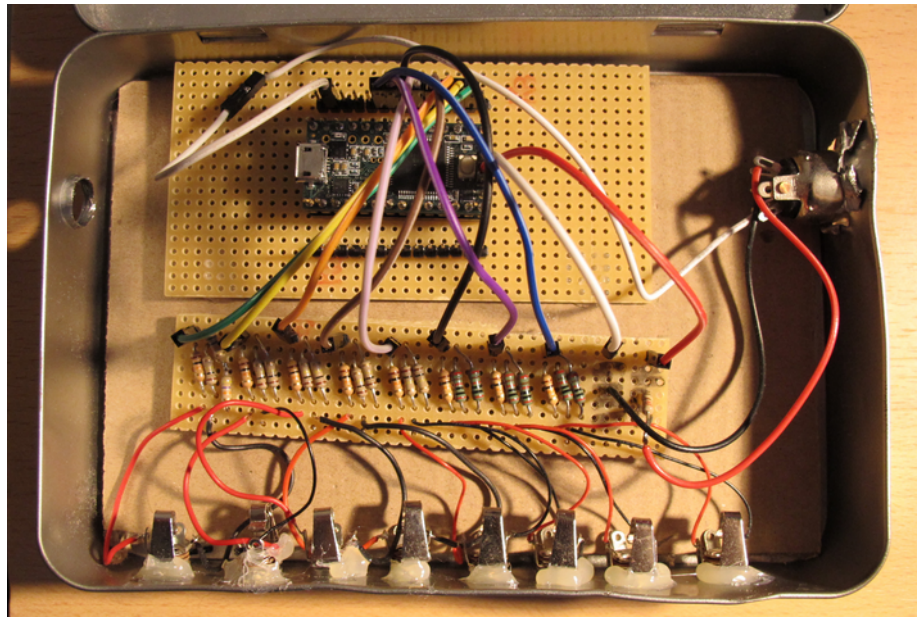

Figure 3. Connection of the piezo elements to the microcontroller via resistors.

2 Manual

2.1 Platform Overview

This is the manual for Version 0.2.0 of the platform. The program itself is currently in an intermediate state, with some functions working fairly well and others that still need a push to create interesting outputs. All of the functions can be used at this point, but there is still a lot of experimenting to be done with it in order to organize the sounds it can (make MIDI instruments) make (for future implementation ideas see p. 33). Consider it a work in progress and as such, the released version is but an alpha version. There are many more features to be designed and implemented.
If you want to get started immediately, read section 3 Installation & Usage).

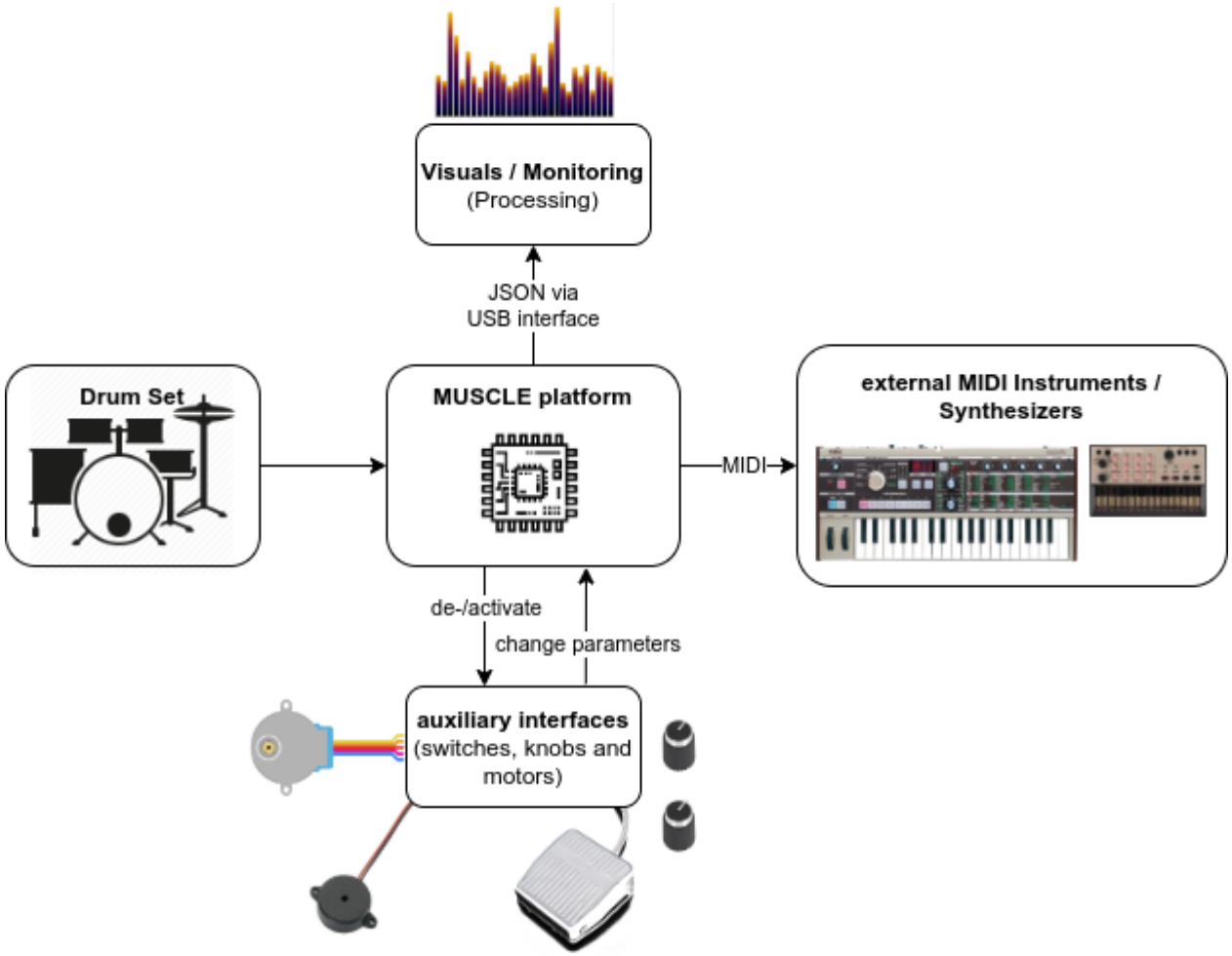

Figure 4 shows an overview of the MUSCLE platform with all possible extensions:
The play of the Drum Set will be recognized by the microcontroller (further explained in chapter 2.3 Stroke Detection).
Incoming signals can be processed and used to trigger MIDI signals on external devices (see chapter 2.7 Instrument Effects and 2.11 Synthesizer class).
Auxiliary interfaces such as foot switches and knobs can be implemented. Additionally, actuators like transducers or motors could be attached to create a more physical experience (chapter 2.10 Hardware class).
The capacities of the used processor allow the additional communication with a computer to generate real-time visuals to the music (a prospect to this can be found in chapter 5 Prospect)

setup function

```
// INSTRUMENT SETUP:
// instantiate external MIDI devices:
mKorg = new Synthesizer(2);
volca = new Synthesizer(1);
// instantiate instruments:
snare = new Instrument(A5, Snare);
hihat = new Instrument(A6, Hihat);
kick = new Instrument(A1, Kick);
tom2 = new Instrument(A7, Tom2);
standtom = new Instrument(A2, Standtom1);
cowbell = new Instrument(A3, Cowbell);
crash1 = new Instrument(A0, Crash1);
ride = new Instrument(A4, Ride);

instruments = {snare, hihat, kick, tom2,
standtom, cowbell, crash1, ride};

// instrument calibration:
hihat->setup_sensitivity(80, 15, 10, false);
standtom->setup_sensitivity(200, 10, 10,
false);
tom2->setup_sensitivity(100, 9, 10, false);
kick->setup_sensitivity(200, 12, 10, false);
cowbell->setup_sensitivity(80, 15, 10, false);
crash1->setup_sensitivity(300, 2, 5, true);
ride->setup_sensitivity(400, 2, 5, true);
snare->setup_sensitivity(120, 10, 10, false);
```

*Codeblock 1. From **`main.cpp`**: Instantiation of Synthesizers (p. 26) with a parameter for the MIDI channel to use and instantiation of Instruments informing the used input pin and the Drum Type (p 16).*
The instrument calibration values have to be provided manually (see chapter 2.3 Stroke Detection).

As you know it from standard Arduino projects, there is a setup section that initializes all needed variables. First, Global and Debug variables can be set to declare whether to use USB communication or not, select a responsive calibration method and decide what to print to the console.
There is an initial time window of five seconds to connect the device to a computer via USB. If there is no connection found, the microcontroller will into non-console-mode, where it will not print anything serially.

Some variables have to be declared in this section:
As presented in Codeblock 1, instruments and synthesizers have to be instantiated, because they belong to non-static classes. The instruments must be stored in a list called `instruments`, that is used for iteration throughout all of the code.

Besides that, it is necessary to manually provide sensitivity values for the calibration of the instruments. (See chapter 2.3 Stroke Detection on how to acquire threshold values for stroke detection.)

Eventually, the two interrupt timers are started.

```
void setup_sensitivity(int threshold_, int
crossings_, int delayAfterStroke_, boolean
firstStroke_);
```

*Codeblock 2. From **`Instruments.cpp`**: The `setup_sensitivity` function must be used with the parameters for a pin-value `threshold_` that needs to be crossed for oscillations on the drum head to be counted, which will eventually lead to the detection of a stroke. `crossings_` defines the minimum oscillations needed, and `delayAfterStroke_` defines the interval during which oscillations are counted. `firstStroke_` controls whether to start counting after the first or last threshold transgression.*

main loop

The loop section has four main stages it passes repeatedly:

1. Each instrument is asked whether it was just hit. (The boolean `stroke_detected` is polled once each program cycle, but this is more than sufficient to not miss any of the strokes). When an instrument was hit, a trigger event set for the instrument will be executed.
2. According to the current beat position (that is signaled by the `masterClock` interrupt timer), timed effects of each instrument are evoked.

```
// START TIMERS :
pinMonitor.begin(samplePins, 1000); // sample
pin every 1 millisecond

Globals::masterClock.begin(Globals::masterClock
Timer, Globals::tapInterval * 1000 * 4 / 128);
```

*Codeblock 3. From **`main.cpp`**: The last action evoked in the setup section of the code is to start the interrupt functions.*

3. The third stage executes all Score commands. These can be resetting the instrument effects, play MIDI notes on a synthesizer, water the flowers or other things that do not need to be implemented in instrument effect functions.
4. Lastly, the code finishes all the tasks that are not timed to the `masterClock` interrupt timer. For example, MIDI notes can be turned off after a fixed time interval (in milliseconds, instead of beat timing). Furthermore, external hardware is addressed to check their states and perform actions.

The main loop procedure is depicted in Figure 5, sandwiched by the two interrupt timers.

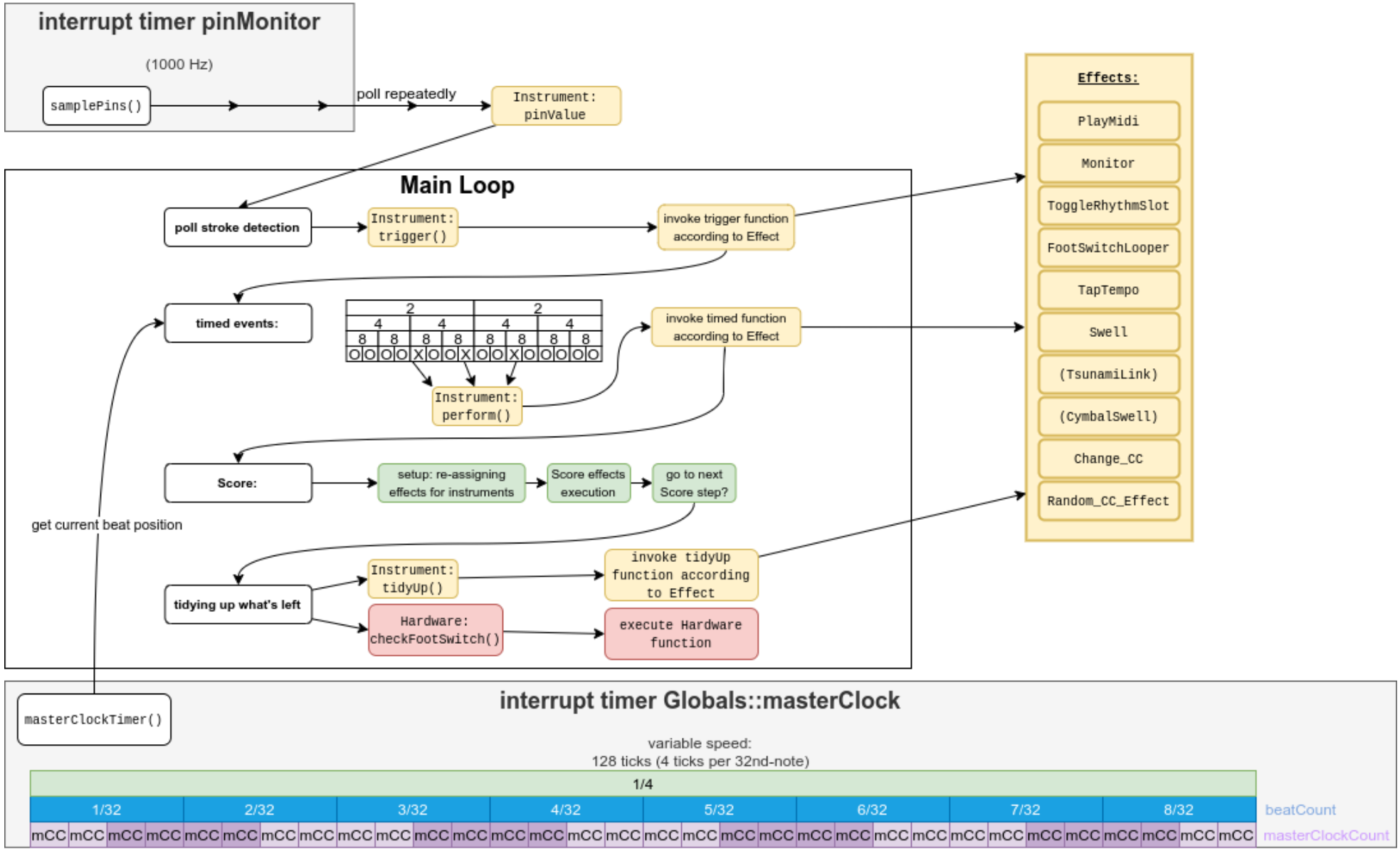

Figure 5. Flowchart of the program. The two interrupt timers pinMonitor and masterClock are the core features for stroke detection (see p. 14) and quantization. The main loop basically puts the Instrument functions `trigger`, `perform` and `tidyUp` (p.18) into order. The right-hand block shows the possible functions, with experimental/unfinished ones in brackets. Eventually, external hardware is addressed.

2.2 Timing / Quantization

Because of the program's requirement of handling beat-tight Effects, a reliable quantization system is provided. Almost all of the implemented effects require the use of this system the in some way or another. Some effects re-play the played instrument strokes or perform actions linked to the occurrence in the beat, more sophisticated algorithms for the evaluation of higher-level parameters like the accuracy of the playing strongly rely on the quantization.

There are two interrupt timers running on the *microcontroller*: The `pinMonitor` takes samples of the all the piezo elements connected to the drums at a rate of 1000 Hz: Once each millisecond, a sample is read from the instruments' pins to evaluate whether the instrument was hit or not. (see chapter 2.3 Stroke Detection). A global `masterClock` takes care of all necessary quantization. This interrupt runs on a speed determined by the tap tempo (p. 22) and basically overflows after 32 cycles. This way, a global variable `beatCount` can be informed to always point at the current 32nd-note in the beat. Since the `masterClock` overflows at the integer 31, meaning it supplies 32 steps, a precision of 32nd-notes can be acquired (actually, the 32nd-steps are subdivided into four steps each, to determine whether a stroke occurred more on the previous or the following step – leading to a precision of 128th-notes).

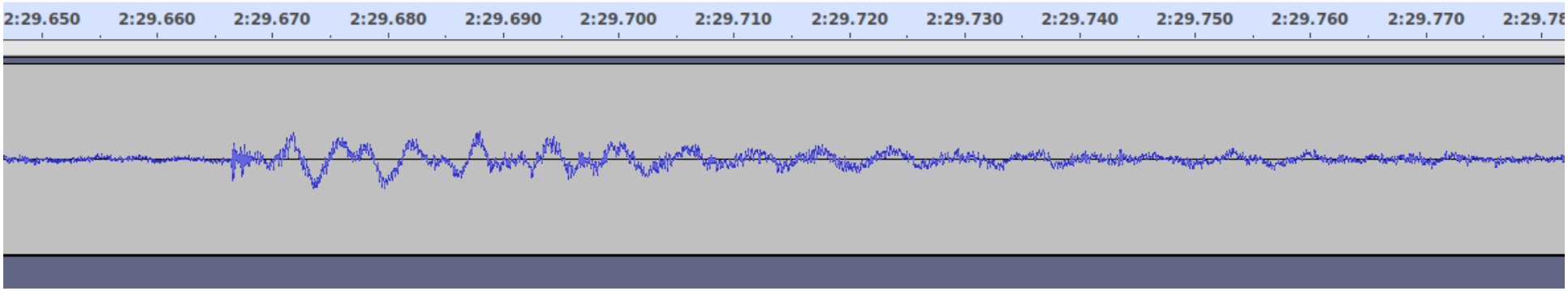

Figure 6. The sound of a hit snare drum extends over about 20 ms. This sound wave was recorded using a regular sound microphone; the oscillation on the drum head varies accordingly, maybe a bit longer. The measure is in the format [minutes:seconds:milliseconds].

2.3 Stroke Detection

The first step in the course of this project was to digitize the play on the acoustic drum set. Because a drum set is a strongly vibrating instrument plain threshold transgression cannot be used. The snare drum rings with bigger drums being played, and cymbals resonate with all types of surrounding noises. I am using contact microphones for the pickup of the strokes to reduce leaking from one instrument to another to a minimum. The microphones produce highly oscillating signals that have to be digitized so to have only on/off signals left. For this, each piezo element is connected to the microcontroller via a resistor circuit, producing pure digital values.

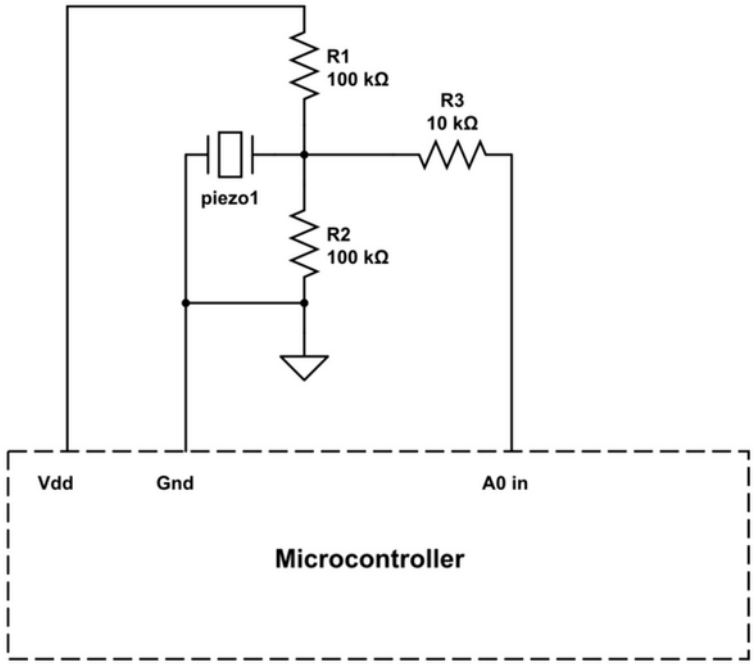

Figure 7. Circuit to connect the piezoelements (contact microphones) to the microcontroller. The resistors are used to reduce the signal's oscillations and make it digital. Taken from https://arduino.stackexchange.com/questions/32793/several-piezo-contact-mics-with-arduino (2020-11-30)

After that, there will still be oscillations left in the signal, but we will just take advantage of these: By counting how often they pass the center value around which they oscillate over a certain time span, the microcontroller can evaluate whether there was a stroke on the instrument or not.
Hitting the snare drum, for example, will induce a sound lasting for about 100 ms (until it is no longer distinguishable from the noise). It suffices to read the first 10 ms after the initial crossing of a threshold and count all following transgressions to acquire a valid stroke recognition (see Figure 6).

The threshold by which the stroke detection should be initiated/counted, has to be defined manually. For this, I provide an additional program sketch in **`Code/debug/contactMic_evaluate_threshold/`**. The noise floor of the piezo elements is calculated automatically during each setup of the Code.

Once the threshold is crossed, the microcontroller starts counting the oscillations using an interrupt function. The minimum number of crossings must be provided for each individual instrument and can be estimated using another sketch: **`Code/debug/evaluate_threshold_crossings_sensitivity/`**

The values for threshold and counts (representing the minimum amount of crossings needed to detect a stroke) have to be manually set and are stored in the instruments sensitivity structs (see p. 18).

2.4 Globals class

All classes have to be able to see certain variables throughout the entirety of the code. Therefore, there is is a class called `Globals` that supplies all these variables. Like the `Score` class, this one is a static class, too, meaning it does not have to have any instances. It is just *there*, invisibly, yet omnipresent. All its' variables can be addressed from anywhere. Like prayers to a God, with the difference, that you will feel the effect immediately.

There are some debug parameters that can be used to determine which way of communication to other machines is wanted (like sending JSON files to be deciphered by a Processing sketch (see p. 31), or printing strokes to the Arduino console).
Then, there is all the important data on the beat timing stored in here, since all of the time-performing Instrument Effects (p. 20) depend on these values.

Another divine feature of the class: It can print all sorts of data types to the console and, it can translate some of them into human-readable form. See Codeblock 4 for comparison.

Some enumerators, not exactly part of the `Globals` class, are supplied in the **`Globals.h`** header file, because they have to be globally accessible by all other classes.

The Drum- and Effect Types are explained in the following section.

```
// ---------------------------- Auxiliary -------------------------

static String DrumtypeToHumanreadable(DrumType type); // casts the input
DrumType to human-readable form

static String EffectstypeToHumanReadable(EffectsType type); // casts the
input EffectsType to human-readable form

static CC_Type int_to_cc_type(int); // CC_Type to human-readable form

// DEBUG FUNCTIONS: ------------------------------------------------
// print the play log to Serial monitor:
static void print_to_console(String message_to_print);
static void print_to_console(int int_to_print);
static void print_to_console(float float_to_print);

static void println_to_console(String message_to_print);
static void println_to_console(int int_to_print);
static void println_to_console(float float_to_print);

static void printTopoArray(TOPOGRAPHY *topography);
```

*Codeblock 4. Auxiliary data type conversions and debug functions in **`Globals.h`***

Enumerators

To understand the language of the SUPER MUSCLE platform, a little bit of vocabulary has to be learned beforehand. There are three "Enumerators", which are basically lists translating numbers into words. This way, a more human-friendly code appearance can be achieved. For example, one can address the a synth's Cutoff Filter using the word Cutoff instead of 44 (for MIDI CC-channel 44).

As explained in Codeblock 5, there are three of these Enumerators, which do not only help you to understand the code, but also depict which CC-Types are implemented so far, how many different Instrument Effects exist and which drum types can be used until now.
The lists are still to be completed and extended.

```
// instrument naming for human-readable
console outputs:
enum DrumType
{
  Snare,
  Hihat,
  Kick,
  Tom1,
  Tom2,
  Standtom1,
  Cowbell,
  Standtom2,
  Ride,
  Crash1,
  Crash2
};

enum EffectsType
{
  PlayMidi = 0,
  Monitor = 1,
  ToggleRhythmSlot = 2,
  FootSwitchLooper = 3,
  TapTempo = 4,
  Swell = 5,
  TsunamiLink = 6,
  CymbalSwell = 7,
  TopographyMidiEffect = 8,
  Change_CC = 9,
  Random_CC_Effect = 10
};

enum CC_Type // channels on
mKORG:
{
  None = -1,
  Osc2_semitone = 18,
  Osc2_tune = 19,
  Mix_Level_1 = 20,
  Mix_Level_2 = 21,
  Patch_1_Depth = 28,
  Patch_3_Depth = 30,
  Cutoff = 44,
  Resonance = 71,
  Amplevel = 50,
  Attack = 23,
  Sustain = 25,
  Release = 26,
  DelayTime = 51,
  DelayDepth = 94
};
```

Codeblock 5. The three Enumerators ***`Globals.h`*** *can be used globally to read and write the code in a human-understandable manner. All of these lists are yet to be completed.*

2.5 Topography class

With the possibility to quantize every stroke on the instruments, the playing of the drummer can easily be logged. Even though an accuracy of 32nd-notes can be achieved, it is more realistic to log 16th-notes only, since the interplay with the machine will lack precision due to human failure here (that's at least what I experienced – maybe there are more machine drummers out there).

The topographies are used to realize the two-dimensional outline of a rhythm, described in chapter 1.2 Topographies of Time. Topographies must be globally accessible, since every instrument has its own topography to log their rhythm, and the `Score` class provides more than one topography for the logging of an overall beat instance and its regularity.

The most obvious feature of the class is an array of length 16, providing a slot for each 16th-note in the beat. Additional parameters describing the average values of the array and threshold values for this average sum are supplied. They can be used for the changing of effects or progression to the next part in the score, for example (see Codeblock 6).

In order to get rid of the noise in the arrays, they can be smoothened, using the `smoothen_dataArray()` function, which will erase all values that are smaller than 1:3 (a threshold that can be adjusted by changing `snr_thresh`) to the highest peak.
Topography arrays can be added to one another via `add()` to summarize multiple instrument's arrays into one overall beat topography, for example.
They can be reset to zero using `reset()`.
A very important feature of the topographies is that they can be derived from other instances, meaning that they can observe the difference of an array to a former state, and from that derive a regularity in their changes. Applying this to the overall beat structure, the regularity of the drummer's playing can be obtained. The `Score::beat_regularity` is made for this and will trigger the score to proceed to the next step once it has reached a certain threshold.

```
std::vector<int> a_16 = {0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //
size-16 array for abstractions like beat
regularity etc

int snr_thresh = 3; // threshold for
signal-to-noise-ratio to be smoothened
int activation_thresh = 10; // threshold
in average_smooth to activate next
action
int average_smooth = 0;
int regular_sum = 0;

bool ready(); // returns true when
average_smooth has reached
activation_thresh

boolean flag_entry_dismissed = false; //
indicates that an entry has been dropped
due to too high topography difference
(for console prints)

// regularity:
int regularity = 0; // the regularity of
a played instrument boiled down to one
value
```

*Codeblock 6. Excerpt from the `TOPOGRAPHY` class in **`Globals.h`**: the array of size 16 is the core element of the class, followed by thresholds and variables as handles for instrument effects and score procedures.*

2.6 Instrument class

The `Instrument` class is the key element of the software: An instrument object must be created for every physical instrument that is hooked up.

Generally, each instrument has a `drumtype` and an input `pin` that should be self-explanatory.
The structs `SENSITIVITY` and `TIMING` hold necessary threshold values for the stroke detection (p. 14). These values are calculated with the calculateNoiseFloor() function during setup; some of them have to be provided manually (see sections 2.3 Stroke Detection and 3.1 Calibration and Initiation).
The structs MIDI and SCORE are for parameters used in different effect situations.

Effects can be assigned to the instruments, from simple ones (like playing a note via an auxiliary synthesizer, when hit) to more sophisticated ones. The assigning is done via the `set_effect` function. To get an overview of all possibilities implemented so far, see chapter 2.7 Instrument Effects.

The instruments have three basic methods that make up the behavior of the program:

- When an instrument is hit, the `trigger` function executes whichever function is needed according to the stored effects. This could be, for example, changing a MIDI CC value.
- The `perform` function is called once every 32nd-note. This way, tempo-linked effects can be invoked. Think for example of a delay effect, that re-triggers what was triggered before, every 8th-note.
- Eventually, at the end of each main loop (way more often than once a beat-step), the `tidyUp` function takes care of things like de- or increasing MIDI values, turning off motors or LEDs, or what more (might be there to come)..

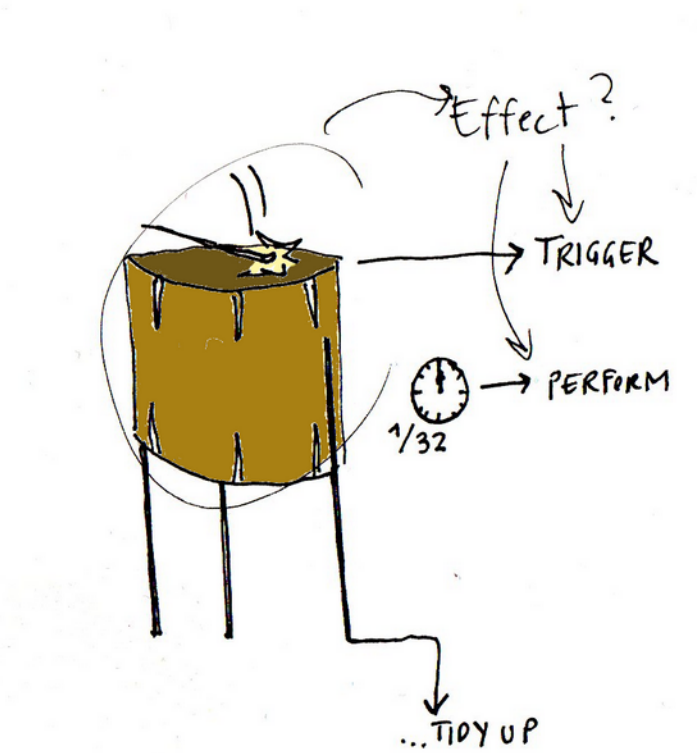

Figure 8. When a drum is hit, a trigger function will be invoked right away. According to the drum's effect setting, a performance method will be triggered in a timed many (linked to the quantized beat). Eventually, a function called tidyUp takes care of parameters that have to be reset.

```
void trigger(midi::MidiInterface<HardwareSerial>);

void perform(std::vector<Instrument *> instruments,
  midi::MidiInterface<HardwareSerial>);

void tidyUp(midi::MidiInterface<HardwareSerial>); // turn of MIDI notes etc

bool stroke_detected();

////////////////////// SETUP FUNCTIONS ////////////////////////////

void setup_notes(std::vector<int> list);

void setup_midi(CC_Type cc_type, Synthesizer *synth, int cc_max, int cc_min,
  float cc_increase_factor, float cc_decay_factor);

void setup_sensitivity(int threshold_, int crossings_, int delayAfterStroke_,
  boolean firstStroke_);

void calculateNoiseFloor();

void set_effect(EffectsType effect_); // just set effect
```

*Codeblock 7. From **`Instruments.h`**: The standard routine methods for each Instrument are: **`trigger`** (at instrument stroke), **`perform`** (beat-linked, timed execution) and **`tidyUp`** (adjust variables in the loop).*

2.7 Instrument Effects

So far, there are a dozen of effects implemented an instrument (stroke) can evoke. I played around with many of them to find out which ones sound interesting and which are helpful in playing freely. Clearly, there is still a lot to be experimented with, and many more functions could be thought up. Luckily, my setup facilitates the quick prototyping of new ideas. In the following section, I will describe the algorithms behind the implemented effects.

```
// trigger effects:
  -----------------------------------------------
void playMidi(midi::MidiInterface<HardwareSerial>);

void monitor(); // just prints what is being played.

void toggleRhythmSlot();

void footswitch_recordSlots();

void getTapTempo();

void change_cc_in(midi::MidiInterface<HardwareSerial>
  MIDI); // instead of stroke detection, MIDI CC val is
  altered when sensitivity threshold is crossed.

void swell_rec(midi::MidiInterface<HardwareSerial>);

void countup_topography();

void tsunamiLink(); // Tsunami beat-linked pattern
```

```
// timed events:
-------------------------------------------------
  void swell_perform(midi::MidiInterface<HardwareSerial>
    MIDI);

  void
    sendMidiNotes_timed(midi::MidiInterface<HardwareSerial>
    MIDI);

  void setInstrumentSlots(); // for Footswitchlooper

  void tsunami_beat_playback();

  void topography_midi_effects(std::vector<Instrument *>
    instruments, midi::MidiInterface<HardwareSerial>); //
    MIDI playback according to beat_topography

  // final tidy up functions:
    ---------------------------------------
  void turnMidiNoteOff(midi::MidiInterface<HardwareSerial>);

  void change_cc_out(midi::MidiInterface<HardwareSerial>);
```

*Codeblock 8. From **`Instruments.h`**: The list of effects that can be triggered either by the stroke of an instrument, or timed, according to the instrument's perform() method. Functions of the category "tidy up" usually don't do anything exciting, but are necessary to take care of some parameters.*

The implemented effects are listed in a globally accessible enumerator called `EffectsType` (see Codeblock 5 on page 10).

PlayMidi

- **trigger:** `playMidi`() simply commands the synthesizer, that is linked to the instrument, to play a Midi note. (see p.27)
- **tidy up:** `turnMidiNoteOff`() turns the MIDI note off after 200 ms

Monitor

- **trigger:** `monitor`() prints the instrument's tag to the console when it is played. The tag is printed at the right position on the quantized beat, so the played Groove can be reconstructed graphically.

ToggleRhtythmSlot

- **trigger:** `toggleRhythmSlot`() records the position of an instrument stroke in the beat (and prints it to the console)
- **perform:** `sendMidiNotes_timed`() commands the linked synthesizer to play a note at the recorded positions each bar. It can be used to perform any action (like replaying notes or changing MIDI CC values).

FootSwitchLooper

- **trigger:** `footswitch_recordSlots`() works like the `ToggleRhythmSlot` function: the beat strokes are written into an array to be repeated in every bar, but the program will only log as long as the auxiliary foot switch is pressed.
- **perform:** `sendMidiNotes_timed`() commands the linked synthesizer to play a note at the recorded positions each bar.

```
11356   23      ※                                      xx
11419   24                                              xx
11481   25                                              xx
11544   26              x                               xx
11606   27                                              xx
11669   28                      ■                       xx
11731   29                                              xx
11794   30      ※                                      xx
11856   31                                              xx
11919   0                                               xx
11981   1                                               xx
12044   2               x                               xx
12106   3                                               xx
12169   4                                               xx
12231   5
12294   6                                               xx
12356   7
12419   8                               o
12481   9
12544   10              x
12606   11      ※
12669   12
12731   13
12794   14
12856   15              x
12919   16
12981   17                      ■
13044   18
13106   19      ※               ■               □
13169   20
13231   21
13294   22
13356   23
13419   24                      ■                       -X-
13481   25                      ■                       -X-
13544   26                                              -X-
13606   27                                              -X-
13669   28                                              -X-
13731   29                                              -X-
13794   30              x                               -X-
13856   31                                              -X-
13919   0       ※                                      -X-
13981   1                                               -X-
14044   2                                               -X-
14106   3               x                               -X-
14169   4                                               -X-
14231   5                                               -X-
14294   6                       ■                       -X-
14356   7                                               -X-
14419   8       ※                                      -X-
14481   9                                               -X-
14544   10                                              -X-
14606   11                                              -X-
14669   12                                              -X-
14731   13                                              -X-
14794   14                                              -X-
14856   15              x
```

Figure 9: The console output of the `Monitor` Effect: prints symbols for each instrument at a dedicated position.

TapTempo

- **trigger:** `getTapTempo()` is probably the most important feature implemented so far, facilitating a stable synchronized play between the drummer and the machine. The function counts the interval between the strokes (within a certain threshold) and divides them by their number, to calculate a global `tapInterval` and the `current_BPM`. It eventually restarts the microcontroller's timer, which has effect on the overall tempo of the play.

Swell

- **trigger:** `swell_rec()` records multiple strokes with the intervals between them on the instrument and repeats them in the same interval (Figure 10). Each hit increases a counter.
- **perform:** in `swell_perform()`, the counter is being decreased at every step in the beat the instrument was not played. The counter value can be used to command the synthesizer to execute MIDI commands, like playing a note at every of these intervals, or change an effect value.

TsunamiLink

- **perform:** in `swell_perform()`, matches the instrument's rhythmic pattern (using its topography array) to a rhythmic field recording sample stored in the *Tsunami Super Wav trigger*'s database. The sample is then played back repeatedly and pitched to the right tempo of the rhythm. This way, some interesting sound effects can be linked tightly to the played beats.

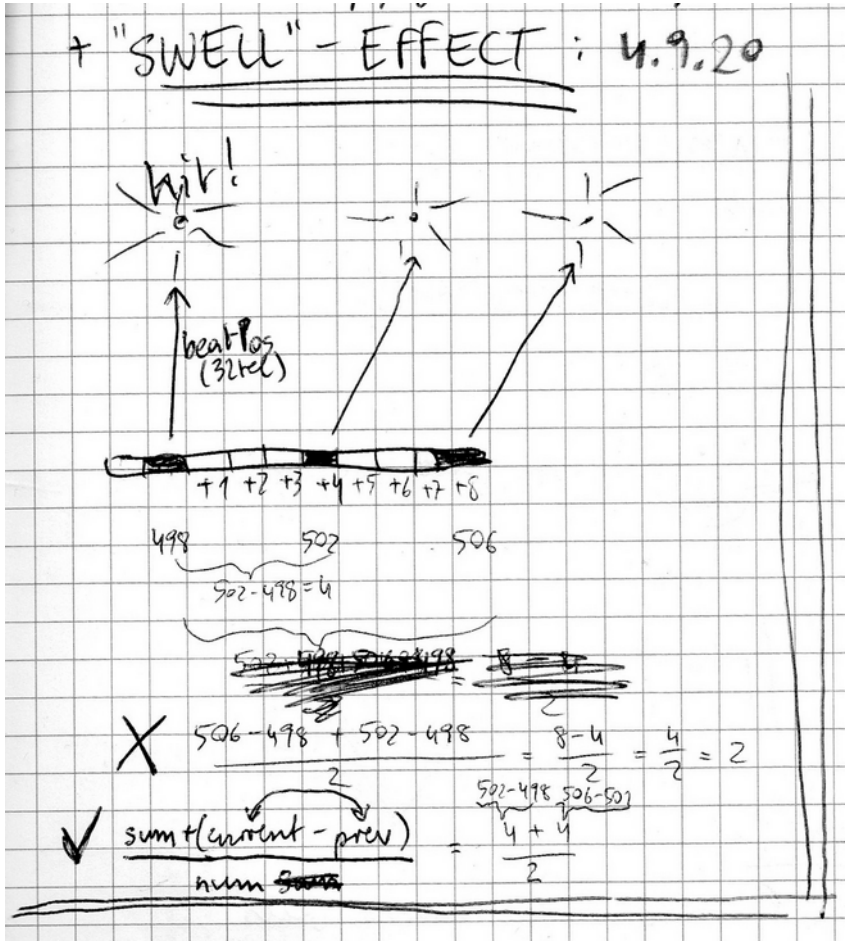

Figure 10. Swell Effect: Re-triggering a synthesizer in the same interval as drum was hit.

TopographyMidiEffect

- **perform:** `topography_midi_effects()` copies the instruments beat-topography (see chapter 2.5 Topography class) and adds it to the static `topo_midi_effect` topography of the Score class. The `topo_midi_effect` is then used as an envelope filter to send commands to the synthesizer according to the topography. This way, synth effects can be tightly linked to the played beats.

Change_CC

Like the PlayMidi effect, Change_CC is quite a simple, yet very effective method:

- **trigger:** `change_cc_in()` increases a value counter whenever the instrument is hit. A MIDI-Control-Change command is sent to the instrument right away, ordering it to change the instrument's pre-programmed midi-effect to the given value.
- **tidyUp:** `change_cc_out()` decreases the value again, until it reaches the pre-defined minimum value. All relevant parameters (minimum and maximum value, increase- and decrease factor) can be set via the `setup_midi()` function of the Instrument class.

2.8 Score class

The `Score` class is to the Code what a Score is to Music: A static manifestation, not sounding itself, but still transcending the composition by its presence. It is not manifested in any musical object (like the played notes) but living outside of the music. As scores serve the musician to perform the music as instructed, the Score class will tell the machine what to do. More specifically, it organizes which effects are being used at what point over the course of a piece.

Generally, the `Score` class provides variables and functions that can be accessed from anywhere in the Code, and they do not necessarily have to be linked to an instrument's action. One of these elements is a list of `notes` that can be added and accessed to let the connected synthesizers play MIDI notes, that should not be linked to an instrument.
Three `Topography class` are contained in the class:

- `beat_sum` is the sum of all instrument topographies. This element can be used to trail the overall topography of the played beat. The metadata of this struct are important to acquire an understanding of the most common timings of all instruments combined, and with that, of the character of the played beat.
- `beat_regularity` is an abstraction of `beat_sum`. It observes the positions in `beat_sum` where a stroke is expected to occur and from that, derive an accuracy of the drumming.
- `topo_midi_effect` is reserved for the `TopographyMidiEffect` (p.22) Using this topography's array as an envelope filter can lead to interesting effects that appear very beat-tight.

In order to organize the score layout for a composition, a `step` integer is provided, representing the current state of the machine. The code evaluates the Score step and behaves according to what is programmed into each of the steps. Using a switch case evaluation, the musical score can be programmed linearly.

(A typical score notation is presented in Codeblock 9.)

The machine state resides on one score step until a certain criterion is reached, that allows the advance to the next step. When the criterion is fulfilled, the auxiliary foot switch can be pressed to proceed. Normally, this criterion is linked to the net accuracy of the played beat over a certain time, manifested by the `beat_rgularity.average_smooth` (p. 24, p.17), i.e. the variable representing the (smoothened) average value of the regular timing accuracy calculated over all 16th-notes.

```
switch (Score::step)
{
  case 1: // fade in the synth's amplitude
  if (Score::setup) // score setup is run once and reset
when next score step is activated.
  {
  // assign effects to instruments:
  // the hihat will change the allocated (AmpLevel) value
on the synth, whenever hit:
  hihat->effect = Change_CC;
  hihat->setup_midi(Amplevel, mKorg, 127, 0, 0.65, 0);

  // these instruments do not play a role here. just print
what they do:
  snare->effect = Monitor;
  kick->effect = Monitor;
  tom2->effect = Monitor;
  ride->effect = Monitor;
  crash1->effect = Monitor;
  standtom->effect = Monitor;
Score::continuousBassNote(mKorg, MIDI); // start playing a
bass note on synth
  Score::setup = false; // leave setup section
  }

  Globals::print_to_console("amplevel_val = ");
  Globals::println_to_console(hihat->midi_settings.cc_val);

  if (hihat->midi_settings.cc_val >= 127)
  {
  Score::beat_sum.activation_thresh = 0; // score step is
ready
  }
  break;
}
```

*Codeblock 9. The score is programmed in the **`main.cpp`** file. This examples shows how the hihat is programmed to steadily increase the amplitude of a synthesizer until the value of 127 (maximum volume) is reached*

2.9 Score Effects

The Score class offers a good possibility to implement effect functions that do not have to be linked to an instrument's activity. They can be triggered globally and independently. There are not many Score functions implemented yet, but a simple, yet important feature is the `playSingleNote` function: It commands the selected synthesizer to start playing a MIDI note. This note can be a first starting point for MIDI effects to be applied to it.

`playRhythmicNotes` goes one step further and makes the synth play a note at a certain interval.

Other Score effects can be thought of to create envelope filters or similar effects that reference to elements not provided by any instrument alone – like the `beat_regularity` (p.24) or the score's state `step`.

2.10 Hardware class

The `Hardware` class designed for the use of external hardware like LEDs for the supervision of the internal machine states or visual special effects, motors for more physical extensions of the drum set or switches and knobs to get more control over the machine.

Functions and Parameters can be set here, that are globally applicable and visible for the instruments.

2.11 Synthesizer class

Just like the `Hardware` class, the `Synthesizer` class offers possibilities to address external devices. Being designed specifically for MIDI instruments, it supplies a number of variables representing the MIDI settings of the connected device. These are known parameters like Cutoff, Resonance, Attack, Release, Amplitude, Attack, Sustain, Delay-Time and -Depth, and so on..

This class is not static, meaning an instance for each external physical device has to be created. Upon construction, the object has to be supplied with the MIDI channel the instrument shall be addressed by.
The class also includes some regular MIDI communication methods: `sendControlChange`, `sendNoteOn` and `sendNoteOff` do exactly as they promise and send the respective commands to the synthesizer.

Foto von benutzten Synths?

After creating instances of physical instruments in the code, these MIDI Synthesizers can be linked to the different drums, so that they can be controlled via the instrument's effects (see Codeblock 10).

```
Instrument *snare; // create a snare drum object
Instrument *hihat; // create a hihat object

Synthesizer *mKorg; // create a KORG microKorg instrument called mKorg
Synthesizer *volca; // create a KORG Volca Keys instrument called volca

void setup()
{
  snare = new Instrument(A5, Snare); // instantiate snare drum using pin
A5
  hihat = new Instrument(A6, Hihat); // instantiate hihat using pin A6

  snare->midi_settings.synth = mKorg; // allocating mKorg to snare
  hihat->midi_settings.synth = volca; // allocating volca to hihat

  hihat->setup_midi(Amplevel, mKorg, 127, 0, 0.65, 0); // alternatively,
   allocate volca to hihat via setup_midi and prepare the values for
   Change_CC effect simultaneously
}
```

Codeblock 10. Exemplary linking of instruments to MIDI synthesizers. (Caution: This is not a complete setup function. Find all necessary setup elements on page 10.)

3 Installation & Usage

```
project
└──Code
|     └──teensy
|     |     holding the main code
|     └──debug
|     |     for Arduino-sketches used to
|     |     evaluate the contact mic
|     |     sensitivities
|     └──processing
|     |     with an exemplary sketch for the
|     |     real-time generation of visuals
|     └──python
|           with a script supporting the
|           calibration process graphically
└──Doc
|     Documentation folder with images and
|     progress logs and other
|     media files
└──hardware
|     CAD and ECAD files
└──literature
|     some interesting material on using piezo
|     microphones, interrupts on teensy, etc
└──logs
|     this folder holds logs that are recorded
|     from contact microphone streams
|     while playing the instruments
└──misc
      miscellaneous.
```

Figure 11. Overview of the project repository folder on GitHub.

To get started with the SUPER MUSCLE platform, go to **`https://github.com/dunland/muscle`** and download or clone the repository. You can also download the stable release accompanying this manual Version 0.2.0.

The main code is found in the **`Code/teensy`** subfolder and must be run using an IDE that supports the *platform.io* plugin.

Connect your teensy 3.2 to the computer, and upload the sketch using *platform.io*. (Another teensy version > 3.2 can be used, but unfortunately, the teensy 4.1 cannot be used together with the *sparkfun Tsunami Super Wav trigger*.)

The piezo elements (contact mics), hooked up to the teensy via some resistors (as explained in section 1.3 Used Hardware and Software) are connected to the analog inputs of the microcontroller.

First of all, calibration values the contact microphones have to be obtained.

3.1 Calibration and Initiation

Part of the calibration process has to be done manually; for this, I provide additional Arduino sketches that have to be executed and evaluated. Because of the way, the strokes are detected (see chapter 2.3 Stroke Detection), the relevant amount of threshold-crossings for a detected stroke, must be provided. I usually use the file **`Code/debug/contactMic_evaluate_threshold`** to examine the threshold of the mics and **`Code/debug/evaluate_threshold_crossings_sensitivity`** to find out about the right number of needed crossings per stroke:

1. Run the Code in **`Code/debug/contactMic_evaluate_threshold`** and print the output to the Arduino Serial Plotter. Look at the output to see where the noise level is, without hitting any of the mics, and define the **`noiseFloor`** variable accordingly (in the main code, this will be done automatically using the `calculateNoiseFloor()` function).
2. You should now hit the instrument selected via its input_pin and note down the maximum peaks. Also, hit some other instruments, to see how much they leak to your selected mic.
   Decide for a good value to use as a trigger threshold for this instrument. The leaking peaks from other instruments should not be able to reach this level (see Figure 12).
3. Run **`Code/debug/evaluate_threshold_crossings_sensitivity`** inserting the values for the noiseFloor and threshold for the instrument and open the Arduino Serial Monitor. Take a look at how many zero-crossings are produced using this threshold value, and decide for yourself how many of the transgressions should suffice for a stroke to be detected. Note down these values.
4. You now have to create instances for your instrument and add them to the `instruments` list. Also instantiate any external MIDI Instruments you plan to use (see page 10 for more on this).
5. Insert the calculated sensitivity values into the main Code using the instruments' `setup_sensitivity()` function described on page 10).
6. You can now start assigning effects to instruments (→ Codeblock 7, section 2.7 Instrument Effects) and write a score re-assigning effects after intervals or events you can decide (→ chapter 2.8 Score class).

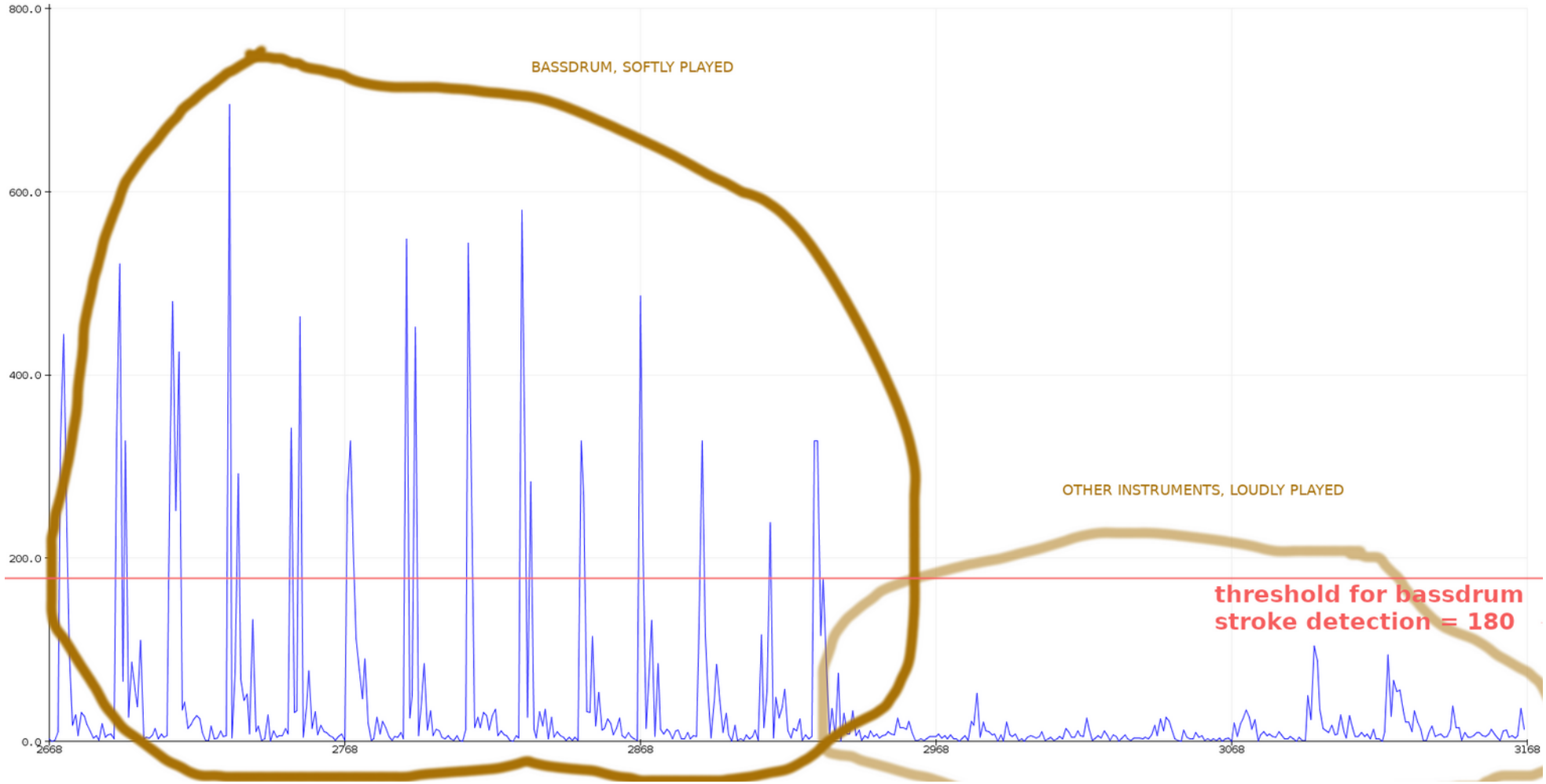

Figure 12. Threshold evaluation for the bass drum: a good threshold value for stroke detection would be about 180: other instruments do not reach this level, while each kick would be recognized. Values are normalized to the piezo element's noise floor.

4 Extra: Serial Communication + Visuals

The teensy microcontroller has a high capacity and runs on such fast rates, that it is not a big deal for it to serialize loads of variables into JSON format (using the brilliant *ArduinoJSON* library) and feed this into a computer. (For the structure of the JSON, see Codeblock 11.) If the microcontroller is connected via USB to a computer within the first seconds of program booting, it will switch to a Serial Communication Mode to send these variables on to the road to be received and deserialzed by a program for the generation of visuals. I wrote a test version of this in *Processing*, to be found in **`Code/processing/processing_visuals`**. It is conceivable using this communication method to create extra graphics and visuals for a more sophisticated live performance using other programs like *vvvv*, *MAX/MSP*, *Unreal Engine*, ...

```
{
 "score":
 {
  "millis": 300,
  "current_beat_pos": 32,
  "score_step": 1,
  "notes": [1,2,3,4,5],
  "note": 5,
  "topo": [1,7,0,0,1,0,0,0,1,0,0,1,0,0,2,0],
  "topo_ready": false,
  "average_smooth":0,
  "activation_thresh": 15
 },
 "Snare": {
  "topo": [1,3,3,2,0,0,4,0,9,8,11,0,0,0,5,0],
  "average_smooth":3,
  "activation_thresh": 15,
  "cc_val": 5,
  "cc_increase": 1,
  "cc_decay": 0.1,
  "effect": "PlayMidi",
  "wasHit": false
 },
 ...
}
```

Codeblock 11. JSON structure for communication to computer generating visuals. The section "Snare" repeats for each instrument, with their respective names. The list is to be extended..

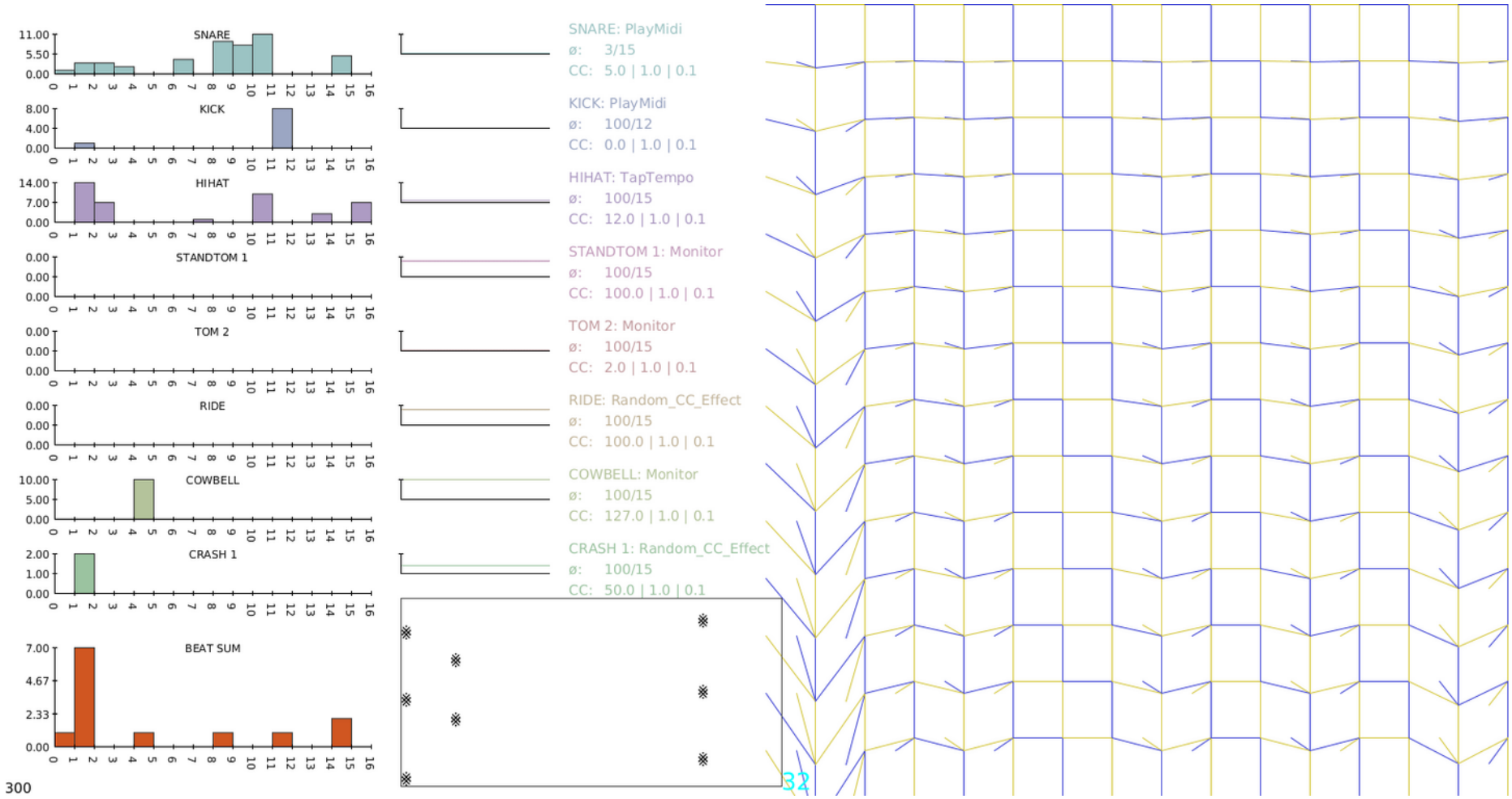

Figure 13. Screenshot of possible additional visual graphics. On the left, parameters of the played rhythms are shown in graphs. The right side shows a more aesthetic/experimental representation of the same values. Real-time strokes on the drums are shown in the rectangle in the lower center of the image.

5 Prospect

SUPER MUSCLE has quite a modular character and can be extended in many directions, as the graphical overview on page 9 shows. I developed the whole platform from scratch, and on the path of conceptualizing to programming, I experimented with APIs to many different areas.

- The *sparkfun Tsunami Super Wav Trigger*, for example, is an entry point to exploring **new sound scapes** integrated into drum beats. The core processor can detect played beats and translate them into 8-bit-arrays (so far) that can select from a database of field recordings according to their topographical foot print (p. 22: *TsunamiLink Effect*). This database still has to be filled with lots of samples, that will be fun to go after.
- Another area that is up for exploration is the **visual domain**, manifested in the JSON interface to a computer. The possibilities in this form of communication can be enhanced much more and more complicated visual patterns could be generated including more variables from the code. Furthermore, the visuals could be used to get much deeper insights into what's going on in the machine. This way, a more sophisticated score compilation could be achieved which could even be rendered readable by the generation of graphical music scores. The derivation of beat regularities still has to be improved a lot for this.
- The installation of **LEDs** / **light systems** would help to get a deeper understanding for the machine states, too. All instruments could be blinking according to their rhythm patterns and glow in a common pulse when a new score stage is ready to be entered.
- Even more sound effects shall be implemented using **re-sampling techniques** of sound wave audio picked up by external microphones. A teensy Audio Shield is the right way to think here.
- The next version's code should allow **non-4/4-beats** and be able to detect more complex rhythmical figures played on the drums, which, again, could be used as triggers.
- An **automatized calibration algorithm** for the evaluation of the contact mics' sensitivity should be integrated. This would make it much easier to get acquainted with the platform.
- After the assessment of the core features of SUPER MUSCLE, the necessary physical connectors should be defined and a long-lasting **packaging to hold a printed circuit board** must be designed. Let's get physical! SUPER MUSCLE shall become a modular production system using several communicative boards with a handful of effects implemented on each machine, one day.
- Lastly: **concerts!!** A lot of live experience still has to be gained using this setup. A good understanding of what the development of the machine requires can only be obtained an iterative, experimental process with direct aesthetic feedback.

6 Acknowledgments

I want to thank my cousin Thomas for all the C++ lessons. Without him, I would have never had the courage to fight the spaghetti monster my Code once was, and dive that deeply into the world of object oriented programming.

Thank you Tobias, Vincent, Jaroslaw, René, Julian, Fabi, Eike, Lukas and Robin for considering me the drummer of our joint music projects. Thank you Mama & Papa for letting me learn the drums, unlike my siblings, who all learned to play the piano (and thank you, siblings!). And thanks to all my friends loving music.

Big ups to my WG, and everyone living in this house, being awesome, and special thanks to Timo for proof reading and for helping out with the preparation and production of video footage (which will be available online).

Surprise-thanks to Dr. Petra Klusmeyer, who has, completely unrelated to this project, really introduced me to the philosophy of *Sonic Thinking*, and who I would have gladly asked to supervise this thesis, if it would have had a more theoretical nature.

A last, obligatory, yet honest acknowledgment goes to my supervisors Kilian Schwoon and Dennis Paul at the University of the Arts, Bremen, for the great teaching in the realms of Electroacoustics and Hardware Design. Thank you for always having open, musical ears and good advice, on-point or off-topic.