MUMT 306 - Fall 2020

The beep-boop Jeff Mills Techno Machine

– by Pierre Talbot –

St.id: 260851456

Table of Content

Title Page 1

Table of Content 2

Background, Motivations & Goals 3

Results 4

Methodology 5-9

1. Arduino 5-6

2. Max 6-9

Design Choices 10-11

Challenges and Limitations 11-12

Challenges 11

Limitations 11-12

Moving forward… 12

Background, Motivations & Goals

Techno was invented in the early 1980s in Detroit, Michigan, from the release of Roland's drum machines, first the TR-808, and then the TR-909. One of the big names that originated from the early years of techno was Jeff Mills, nicknamed *The Wizard* for his technical abilities to perform live with just a TR-909 to accompany some records. At age 57, Jeff Mills has over 30 years of producing and is still active today. Here is a non-exhaustive list of a several favourite works by Jeff Mills:

The Bells, Changes of Life, Imagine, Step to Enchantment, If (We), The Extremist, Sugar is Sweeter, D.N.A., Gamma Player, Solid Sleep, (this could go on for a while so I'll stop here…)

Figure 1: The Purpose Maker record cover.

Figure 2: Jeff Mills playing with a TR-909 Drum machine. Please click on this image to understand why he is nicknamed The Wizard.

I wanted my final project to have to do with making techno, and since I could not cover all types of techno, I chose to target Detroit techno and Jeff Mills' style in particular, as it is an artist I've been listening to a lot for some time. Since Jeff Mills always has a TR-909 drum machine on hand, I knew where I wanted my samples to come from!

As I started to work, my goal for the project was still quite vague – though I knew that I wanted to produce some techno, I did not know how interactive I was going to want it nor if I would even be able to make it. I knew that I would have at least 4 different channels for the kicks, snares, cymbals and one for a filler instrument.
The main reason that made me attempt to make it user-interactive came after selecting a random 16-step sequence (out of a given bank of pre-made sequences) for all channels every 4 measures. This was surprisingly unsatisfying to hear and only made me want to adjust them to find the right combinations of step-sequences to be played together. This is how I ended up creating an interactive 16-step sequencer drum machine.

Here's just a quick explanation on the name of my project before we really go any further. Hopefully by now you'll have guessed why Jeff Mills has his place in the name of my project; same goes for "techno machine". The "beep-boop" part of the title comes from my friends and roommates who strongly disagree with the name of my major and minor at McGill (which one would call electrical Eng. & MST) and prefer to qualify my studies as a "beep-boop" degree.

Results

Eventually, I ended up managing to meet my goal in making a 16-step sequencer drum machine with an additional "filler instrument" channel. 4 channels were used in this version of the machine:

- ❖ Channel 0: Kicks
- ❖ Channel 1: Snares
- ❖ Channel 2: Filler instrument
- ❖ Channel 3: Cymbals

The step-sequencer is comprised of two complimentary interfaces: an Arduino hardware interface and a Max patch user interface.

Hardware interface:

- 4 buttons, used for selecting through a list of pre-made 16-step sequences. Each button targets a different channel.
- 2 potentiometers, one for the master tempo, the second for the filler channel volume.

Max patch user interface:

- Program number selection for the filler channel.
- Selection of instrument types for the other three channels*(#of samples available*): Kicks(8) / Snares(21) / Cymbals(19).
- 3 sliders to control the output level of the 1st, 2nd and 4th channel (Kick, Snare, Cymbal, resp.).

The beep-boop Jeff Mills Techno Machine is fully functional and is surprisingly more accurate timing-wise than I would've thought before starting the project. The wiring (figure 4 below) really isn't that complicated so you should definitely try it out as long as you have an Arduino board, 4 buttons and 2 potentiometers!

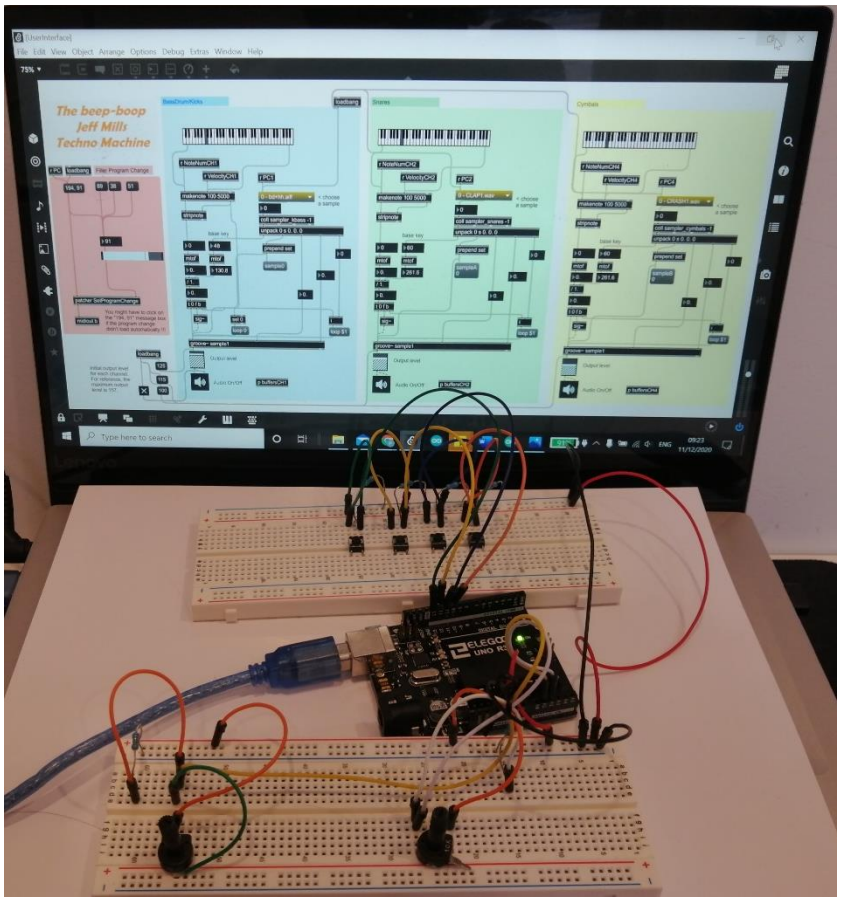

Figure 3: The beep-boop Jeff Mills Techno Machine, a view of the complete interface.

Methodology

The project uses both Arduino and Max. The Arduino is the part that chooses when/what note and instrument should be output at a specific step of the 16-step sequence. It sends the channel message to Max using the *midi2Max()* function (created by Prof. Gary Scavone).
While all messages sent use the MIDI channel messages format, only one channel (the 3rd channel for the filler instrument) is actually sent to a *midiout* using the full MIDI protocol (a *note_on* is always followed by an eventual *note_off*). The other 3 channels only require sending *note_on* messages because they will be routed to a sampler (that plays through a provided audio sample). The Max patch takes care of interpreting and sorting out the received MIDI channel messages by using the *midiparse* object.

1. *Arduino:*

Hardware Overview:
In the current form of the step sequencer, only 4 channels are used, hence there are 4 buttons for each channel. Two potentiometers are used as knobs to control the master tempo and the filler channel velocity.
This adds up to a total of 4 digital pins (out of the 12 of the Arduino Uno board) and 2 analog pins (out of 6). Since I only have 4 channels in the current version of the machine, power shortage and pin shortage were no threats to the project.

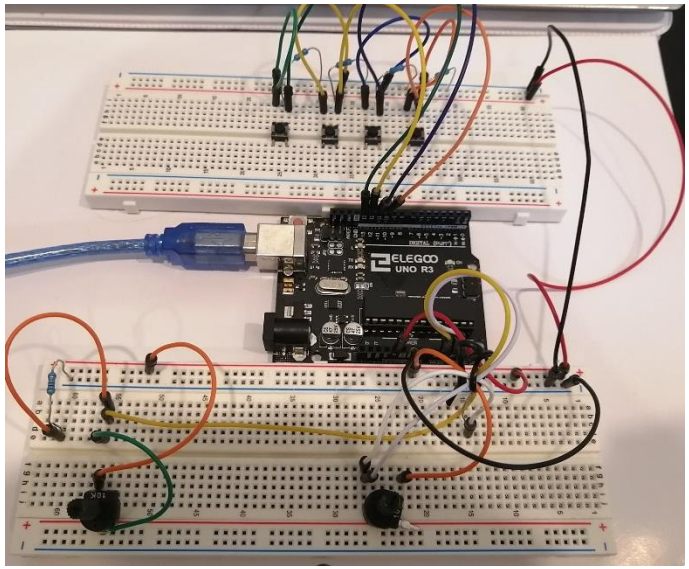

Figure 4: The hardware wiring

16-Step Sequence Patterns:
I created several sequence patterns for all 4 channels. These are represented as arrays of size 16 (i.e. 4 measures of 4 steps), which are all stored inside an array (1 array of arrays for each channel, e.g. *SnarePatternStorage[19][16]* contains 19 different sequences for the snare channel). These arrays of arrays offer a selection of different patterns that the user can select.
The individual sequence pattern arrays themselves are composed of either null values, or non-null values. A non-null value represents a note to be played for that step, whereas a null value will signify that no note should be played at that step.
In the current version of the machine, the 3rd channel (filler instrument channel) is the only one to make use of different note numbers, therefore the note numbers to be played at a specific step in the sequence are directly stored in the sequence pattern arrays. This is not the case for the other channels (1st, 2nd and 4th) since the notes for those channels are always the same, therefore a simple 1 suffices to represent a note to be played at a specific step.

Setup and other noteworthy variables:
At the beginning of the code, we specify the initial sequence pattern for each channel out of the bank of sequences, as well as the initial drum instrument (kick type, snare type, cymbal type). The *setup()* section of the code is used to specify the pins for the different channel buttons. It is also used to send the initial instruments type for the drum instruments to the Max patch using *midi2Max().*

Sequencer:
This is where the main action happens for the Arduino code. Essentially, every quarter of a quarter-note duration (i.e., every time a step duration passes), for each channel, the value at the index given by the current step number, of the current selected sequence array is read. If the value is a zero, nothing is sent through *midi2Max()* for that step, and the code moves on. If the value is different than 0, then a note gets sent through *midi2Max()* by means of a MIDI channel message.
Note that in the case of the filler channel, the value taken from the sequence pattern array corresponds to the note number to be played. Furthermore, for the filler channel, an extra stage involves checking the state of the boolean *FillerIsON* so that if the previous step corresponded to a *note_on*, a corresponding *note_off* can be sent before treating the current step.
At the end of this section, the integer *stepcount,* that takes care of counting the current step (from 0 to 15), is incremented by 1 for the next iteration (unless it has reached the last step, in which case *stepcount* is reset to 0).

Buttons:
At the beginning of the *loop()* section, the state of each button is read and stored in (instrument_type)*ButtonState.*
The logic for the implementation and functionality is the same for each button: if the current button state is different from the last time we checked and the current button state is a HIGH, then the sequence pattern of the corresponding channel changes to the next one in the bank of sequences. If we have reached the last sequence pattern for a channel, then it is reset to the first sequence pattern for that channel.
Finally, we store the current state of the button into *last*(instrument_type)*ButtonState.*
The reason for checking the last state of the button is that the Arduino is too fast and without checking the last state, we would skip several sequence patterns every time we pressed on the button.

Tempo & Filler Volume Knobs:
The master tempo and volume for the filler channel are controlled by means of potentiometers. Once a minimum and maximum value for the parameter is set, the value at the potentiometer is read using *analogRead()*. Since *analogRead()* takes values between 0 and 1024, we need a couple of operations to scale the value in the range of the defined minimum and maximum for the corresponding parameter. Tempo is expressed in beats per minutes (bpm) (this is the same as steps per minute) and gives the duration of a quarter-note in milliseconds by performing *60000 / tempo.*

2. ***Max:***

Communication patch (main patch):

This patch is heavily based on the one provided in the course for assignment 5. Once the user selects the correct port, this patch receives the stream of messages sent through the serial monitor of Arduino, that is the messages sent through the *max2Midi()* function. The *fromsymbol* object will output MIDI channel messages 1-by-1 (2 or 3 integers per message). The number of integers per channel message is obtained using the *zl.len* object and then sent through *send length* as it will be used as an argument for a *zl.group* object in the *UseMIDI* patcher (located inside the *parsing* patcher). The *parsing* patcher takes incoming channel messages and sends them through a *midiparser* object. However, the *midiparser* object requires the content of the incoming messages to be separated by commas (e.g "144 55 60" should be "144, 55, 60"). This is where the patcher *addCommas* comes in.

Patcher addCommas: on advice from our glorious TA Matt, adding commas between the integers of a channel message was made easy with the use of the *separator","* message, and the *tosymbol* and *fromsymbol* objects.

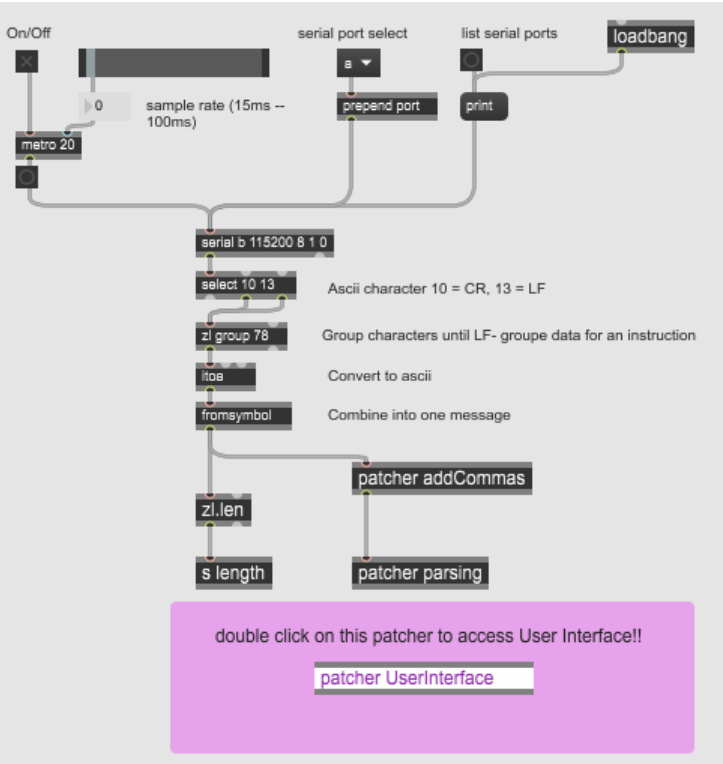

Figure 5: Communication/Main Max patch

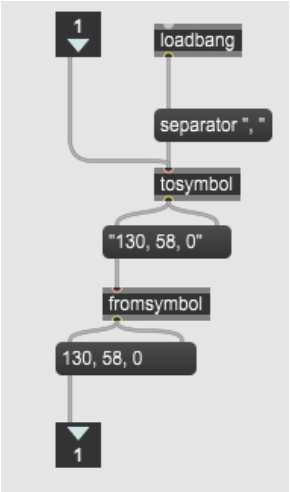

Figure 6: addCommas Max patcher

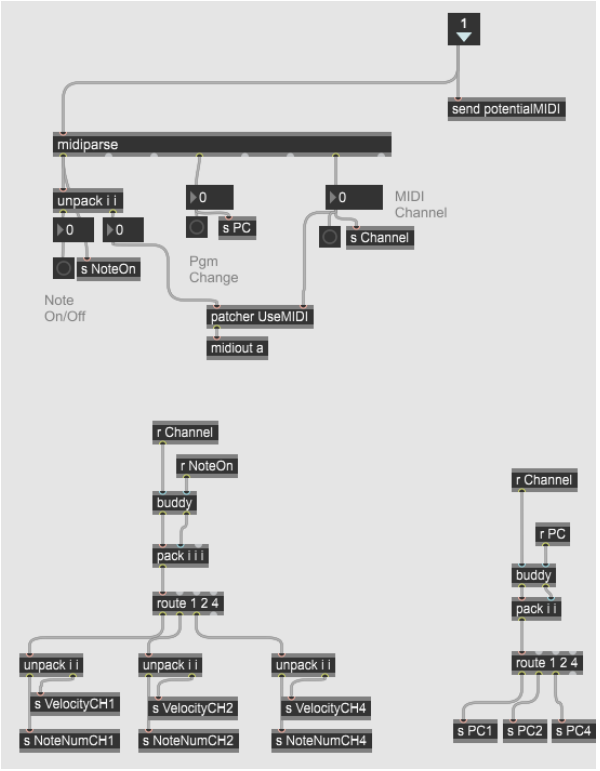

Figure 7: parsing Max patcher

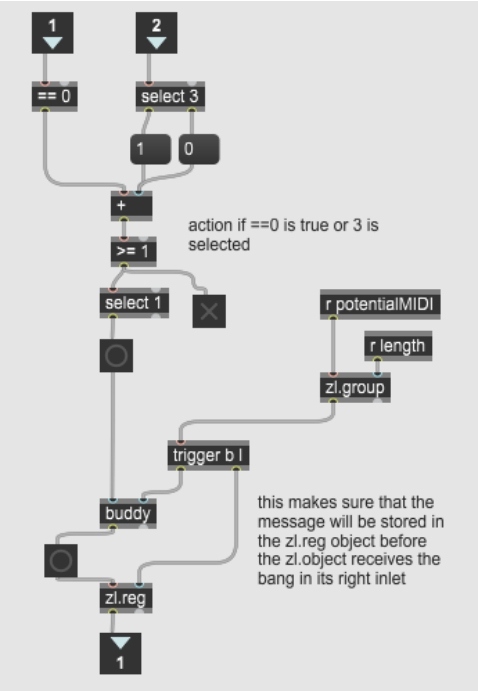

Figure 8: UseMIDI Max patcher.

Patcher parsing:

This patcher receives the message sent by *midi2Max()* with each element of the message separated by a comma, and takes care of sending the current channel message to the correct destination (either to *midiout* or to one of the samplers).

The *midiparse* object is convenient to use here because it allows to easily extract the elements of each channel message, in particular the channel, program change, and note number & velocity for *note_on* and *note_off* messages. The channel is used to determine where to send the message.

The *buddy* object is used to let the correct channel message type pass through (we are only dealing with two types here*: note_on/note_off* and program changes), then the *route* object routes the channel message to the correct sampler based on the channel number.

Since only the 1st, 2nd and 4th channels get their messages sent to the samplers, we need to find a way to make the difference between messages coming from the 3rd channel and messages coming from other channels. This is what the *UseMIDI* patcher is for.

UseMIDI patcher: this patcher takes two inputs via its inlets: the channel and the note velocity, both provided by the *midiparser* object. The message from the main patcher is sent to *midiout* if one of the following conditions is met: 1- The note velocity of a *note_on/note_off* message is 0 (which only happens for the 3rd channel since it is the only one to send *note_off* messages). 2- The channel determined by the *midiparser* object is the 3rd channel.

UserInterface patcher:

This is where the fun happens! This patcher provides user control over the output level of the Kicks/Snares/Cymbals, the selection of the type of Kick/Snare/Cymbal to use on each channel, as well as the MIDI program change number for the filler channel.

The 3 samplers (kick/snare/cymbal) all work the same way and are heavily based on a Max tutorial patch (MIDI Tutorial 3: MIDI Sampler). Each element of the channel message (i.e., Note number, note velocity, or program change number) was extracted in the *parsing* patcher and sent to the appropriate sampler.

To preserve the original audio sample, the base key of all samples of the same instrument type (kick/snare/cymbal) are set to the same value, which is the fixed note number set in the Arduino code for each channel (48 for kick samples; 60 for snare and cymbal samples).

Setting up the samplers was a long and repetitive process as for every additional sample we have to update:

- The text file for the *coll* object that contains the information required for reading through samples.
- The *umenu* object that contains the list of selectable samples.
- The *buffersCH* patcher that contains all the buffers for each channel.

The keyboards for each sampler are not essential at all but were kept to give an idea of what a sample played at a different note number sounds like, in case I ever want to implement the kick/snare/cymbal sequences with varying note numbers.

The red section is where you can modify the program number of the filler channel. If the correct port is selected quickly enough in the main patch upon compilating the Arduino code, the initial program number will automatically be set to number 91 (which doesn't sound too bad!). Or else you will have to click on the message box "*194, 91*" to set the first program change. The reason for this is that the message box contains two elements separated by a comma and therefore needs to be banged twice to pass both elements. The *setup()* section of the Arduino code sends program changes and so the *r PC* object should provide at least 1 bang while the *loadbang* provides the first bang. I tried using a *delay* object connected to the *loadbang* to produce 2 successive bangs upon opening the patcher however it did not work consistently and was not aesthetic either.
A slider was provided for the user to select any of the program changes however most do not sound great. For the user's convenience, suggested program changes are displayed as messages above the slider.

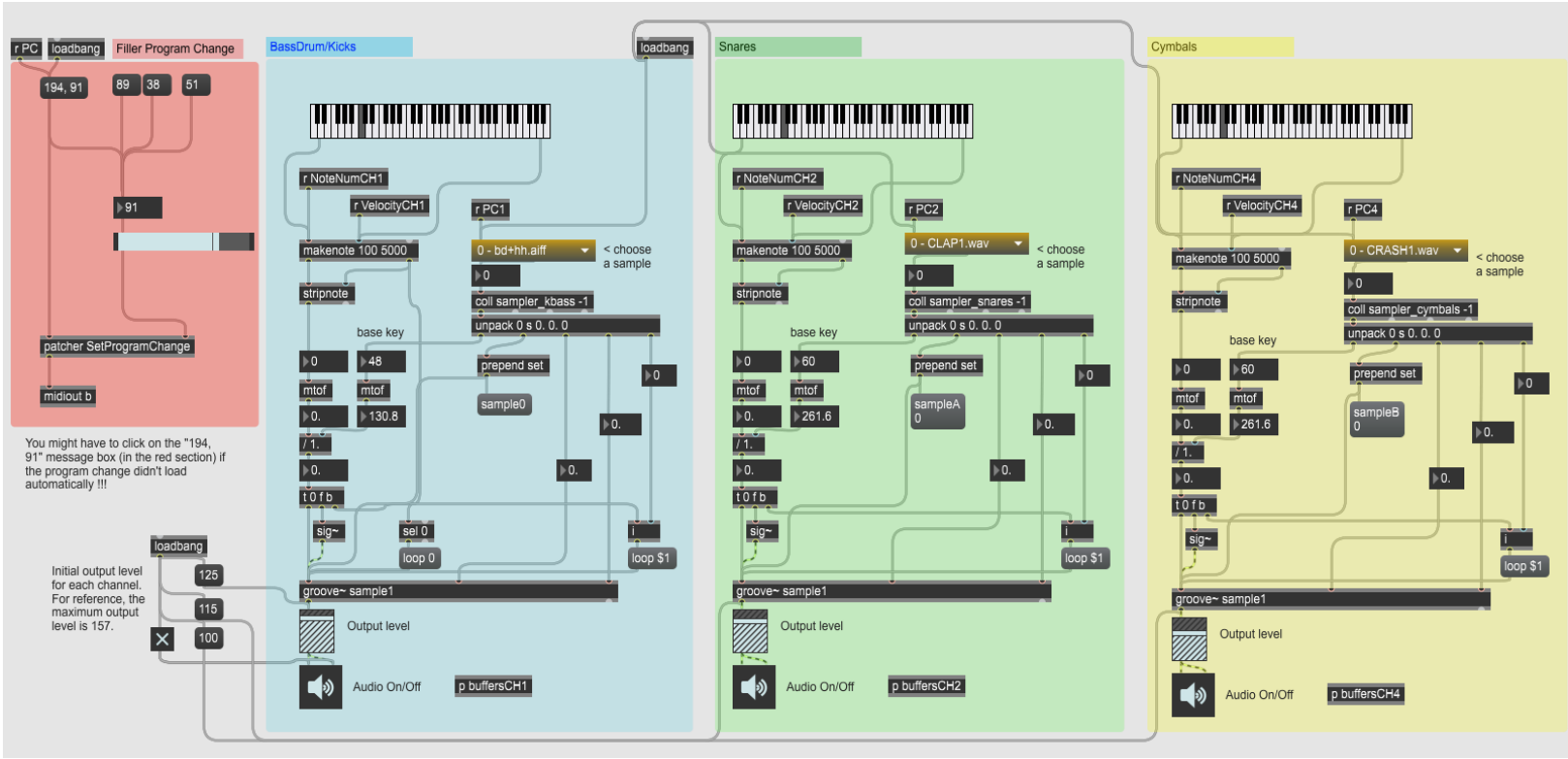

Figure 9: UserInterface Max patcher

Design Choices

The samplers:

While working on generating techno in the style of Jeff Mills, one of the first foreseeable concerns that arose was what sounds I was going to use. The best choice was obviously to have sounds similar to the ones produced by a Roland TR-909 drum machine since it is what Jeff Mills and Detroit techno is heavily based on. However, after going through all program changes (including note numbers on the MIDI percussion channel), I quickly realised that standard MIDI program changes were not going to come close to looking like any sound from the TR-909.

After several hours of researching for possible alternatives and going to Matt's office hours, the most realistic choice seemed to be to make use of samplers, as suggested by Matt. Using samplers also provides a lot of flexibility and offers an unlimited choice of sounds that can be incorporated provided that we have the sample.

Thanks to the extensive documentation provided by Max in the MIDI Tutorial 3: MIDI Sampler tutorial, which is what part of the *UserInterface* patcher is based on, understanding how to make use of the samplers was made easier.

Keeping a MIDI-protocol-based channel for the filler instrument:

In most of Jeff Mills' works, the TR-909 samples are accompanied by a filler sequence pattern with notes varying on a minor key.

One of the limits of the sampler is that the way it is used to obtain higher or lower pitches from a sample is by increasing or decreasing the speed at which the sample is played, respectively. Therefore, I found that a sample played at a note number more than 5 notes away from the base note will result in a sound that is not anything like the original sample.

Although this is not a problem for the kicks, snares and cymbals because they don't require changing notes, using a sampler is far from being the best way to go for the note-varying filler instrument. This is what led me to keeping one MIDI-protocol-based channel for the filler instrument, despite the additional challenge it created by having to route channel messages to either *midiout*, or a sampler.

Breaking-down the user controllability between the Arduino hardware and the user interface patcher:

Since anything to do with the timing was handled inside the Arduino code, having a potentiometer connected to the Arduino board to control the tempo was an easy decision.

Although I considered having buttons for traversing the different types of samples for the sampler channels, I quickly realised that it took off a lot of flexibility and intuition for the user. Having a list of selectable samples made much more sense than having to blindly go through all of them to play the desired sample. For this same reason, I decided to provide an extra effort into selecting the program change for the filler instrument inside the *UserInterface* patcher instead of sending MIDI channel messages from the Arduino code by means of a button.

The easiest way to control the filler channel volume was to do so in the Arduino code, and since I had a 2nd potentiometer at hand, I set the note velocity in the channel messages for the filler channel to be controlled similarly to the tempo.

Implementing the change of step sequence patters with buttons was an easy decision to make given that the banks of selectable step-sequences were stored in the Arduino code and that accessing the individual steps was made through the Arduino code.

This breakdown of controls was found to be very intuitive and convenient for live performance and was made final.

Challenges and Limitations

1. ***Challenges:***

The most challenging part came with the Max patches. The main challenge to tackle remained around correctly routing the received channel messages to the correct destination, between messages addressed to the samplers versus those addressed to *midiout*.

The part that required the most debugging was the *parsing* patcher. Its task was to filter the messages that were to be blocked and those that had to pass on. The most useful objects that were used to debug this without creating timing errors were the *buddy* object, the *trigger* object, and the *zl.reg* object. The latter receives a list that it stores and only outputs it if it receives a bang in its right inlet. The *pack/unpack* objects also came in handy for debugging as they allowed to extract individual elements of messages. This allowed for specific parameters of a message to be sent elsewhere in the max patch, or to induce other actions based on the extracted value (e.g., the channel number in the *parsing* patcher).

Another small but recurring challenge was dealing with comma-separated lists (i.e.. 144, 90, 100), and non-separated lists (i.e., 144 90 100). After finally finding a way to separate elements of a MIDI message of any size with commas (required for the *midiparser* object), I came to realise that in their new form (separated by commas) these messages were difficult to deal with. This is because most Max objects will not treat them as a list anymore. Therefore, I had to find a way to group those back together, in particular for the *midiout*-targeted messages.

A quickly resolved issue that is worth mentioning is the abandon of the *delay()* function in the Arduino code. I quickly came to realise that *delay()* would be impossible to use for our application because it puts the Arduino board to sleep for the specified duration. This would not be a problem had we a single channel sending messages 1-by-1, however this is not the case. The alternative was implemented by the sequencer part of the Arduino code, which is iterated through every quarter of a quarter-note duration.

2. ***Limitations***

Overall, the limitations to which my project is confronted remain, for the most, minor. They are listed below by order of increasing magnitude:

- Occasionally, there will be a single filler note playing in the background because its corresponding *note_off* message was not received. This occurs sometimes when changing the program number. I included it in the limitations list only because it is not something the user has control over. Nonetheless, the few times this happens are much appreciated as it only happens for program numbers 91 and 89 which provide a nice background ambient sound.

- When turning on serial communication in the main patch, the sequencer takes 1-2 seconds to get running properly.
- The potentiometers are picky on connecting to the breadboard and when the connection is lost the default reading will be 0 (which takes the tempo to the slowest bpm). To solve this, I attached one end of the copper wire directly to the potentiometer (I suggest you do the same).
- Sometimes the initial program change of the filler channel does not get automatically loaded and must be banged by the user in the *userInterface* patcher.
- Adding new samples in the *userInterface* patcher is a 3-step process (see page 8).
- Editing or scrolling through the *userInterface* patcher while the sequencer is running may cause momentary timing glitches (this is not too much of a problem as the *userInterface* patcher fits on my computer screen, so I do not usually need to scroll).
- The potentiometers provided in my Arduino kit have some sort of logarithmic reading – that is, the readings are accurate on both ends but middle values are extremely difficult to select. I was not able to find any solution online either.
- The memory of the Arduino Uno limits the size of multidimensional arrays. This limits the size of the banks of pre-made 16-step-sequences and I am currently on the threshold (adding a single new step-sequence pattern will mess up the functioning of the Arduino messages sent to Max).

Moving forward…

I am most definitely planning on continuing to work on *The beep-boop Jeff Mills Techno Machine* to upgrade and perfect it during the break. The starting point will be to add channels and samples (in particular for ambient sounds). At least one of the new channels would be for longer samples (such as an ambient note). An upgrade of the material I am using will also certainly be required, especially the Arduino board in order to add new step-sequence patterns. The potentiometers could also be replaced by more consistent ones. With enough free pins, we could also have 16 LEDs that would light up for the corresponding step of the step-sequence. This is however less essential and would be difficult to implement on more than one channel since it takes a pin per LED.

Thanks for teaching the course and I hope you enjoyed the read! Make sure to check out the demo video as well!