MUMT 307 – WINTER 2021

The beep-boop Machine

(The Return)

– *by Pierre Talbot* –

St.id: 260851456

Table of Content

Background, Motivations & Goals

As part of MUMT 306 last semester, I implemented a simple Arduino/MaxMSP-based drum machine – *The beep-boop Jeff Mills Techno Machine*, featuring 4 channels that could play different instruments following sequences from a predefined bank of sequences stored as arrays of 16 integers in the Arduino code. This implementation meant that the user wasn't able to create its own sequence on the spot, but rather had a choice from a limited set of sequence patterns. Furthermore, in *The beep-boop Jeff Mills Techno Machine*, the metronomes ticks was governed by the Arduino which meant that when multiple channels had an active beat at the same step (0-16), the messages were sent sequentially (given that Arduino's serial communication is sequential). This would ultimately result in timing errors and offsets at the output.

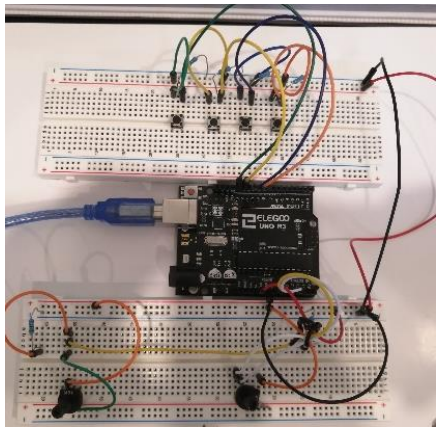

Figure 1: Final Project for MUMT306 Fall 2020, The beep-boop Jeff Mills Techno Machine, hardware interface.

In the light of these fundamental problems, I chose to expand on the drum machine idea for my final project in MUMT 307, this time reimplementing the drum machine such that the user would have real freedom without any timing errors, and with the addition of effects. Luckily, over the course of the winter break, I acquired some additional hardware equipment such as more precise potentiometers (linear knobs and logarithmic sliders), extra buttons and wires, extra breadboards, and last but not least, an Arduino Mega board to replace the Uno I used before.

As far as the choice for the name of the drum machine goes, my roommates found it easier to describe it as a "beep-boop machine" rather than anything else, and as a result it became the common saying.

Results

With online classes, I've had much more time on my hands and working on this project was much more fun than studying for classes like power-engineering, and provided a nice excuse to slack on those classes too. I was able to start working on the project very early which allowed me to implement all the ideas that came to my mind as I progressed (e.g: the LCD menu display, the *lock'n'loop* FX functionality, the two operating modes.) without really having a time constraint.
Therefore, I eventually ended up managing to meet my goal in making a 16-step sequencer drum machine with FXs. You can go check out the demo video!

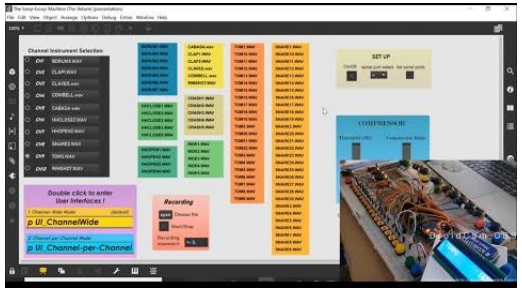

Figure 2: Demo video on YouTube

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

Hardware interface:

- An Arduino Mega.
- 16 potentiometers (analog pins):
    - Filters: 1 slider with logarithmic reading and 4 knobs.
    - Tune: 1 knob.
    - Reverb: 1 knob.
    - Flanger: 5 knobs.
    - Delay: 2 knobs.
    - Volume: 1 knob.
    - Tempo: 1 knob.
- 27 buttons (digital pins):
    - 16 step buttons (1 for each of the 16 steps).
    - 6 buttons for the FX *lock'n'loop* functionality (1 for each FX).
    - 5 menu/general buttons: Operating mode, On/Off, Reset, Next Channel, Previous Channel.

MaxMSP user interfaces:

- Home Menu Interface: Serial setup for communication with Arduino; Recording; Instrument attribution to channels; access to the User Interface patches.
- 2 User Interfaces (Channel-Wide and Channel-per-Channel).
    - User-friendly blocks for each FXs to provide visualization over the amount being applied and control over the state (On/Off) of each FX.
    - Chain of FX easy to modify (default order: Tune-Filter-Reverb-Delay-Flanger-Gain → Compressor).

The beep-boop Machine is fully functional and has no issues timing-wise. Unfortunately, the wiring of this circuit is much more complex to reproduce than that of last semesters work, which is why I linked a demo to my project.

Methodology

The beep-boop Machine uses both Arduino and MaxMSP. The Arduino Mega takes care of displaying the LCD screen, and continuously reading the potentiometers and button states. These readings are then communicated as MIDI formatted messages to MaxMSP through the serial monitor using the *midi2Max()* function. On the other end, MaxMSP takes care of decomposing the received messages and routing the information correctly to the channels. Furthermore, MaxMSP is in charge of the timing and therefore takes care of triggering the metronome ticks.
Although Arduino does not transmit directly the 16-step sequence pattern of each channel to MaxMSP, both MaxMSP and Arduino keep track of the same sequence for each channel. This is done by means of Arduino relaying the information of a single step being selected to MaxMSP, which then keeps track of which steps are active for each channel by itself.

1. ***Arduino:***

Hardware Overview:
In it's current form, *The beep-boop Machine* uses 27 digital pins of the Arduino Mega board out of 54, so the lack of digital pins was not an issue. On the other hand, the analog pins were all used (out of 16) and there had to be design choices over the choice of the potentiometer attributions.

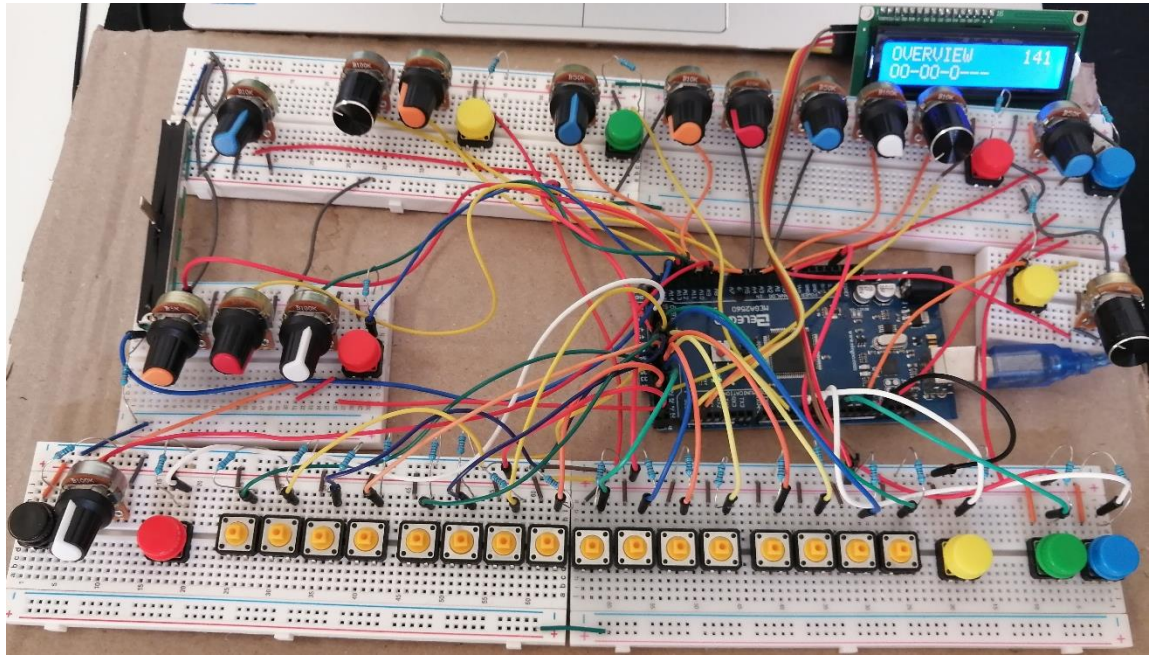

Figure 3: Picture of The beep-boop Machine's hardware interface.

Communication to MaxMSP:
All the analog and digital readings performed in Arduino are communicated to MaxMSP under the form of MIDI channel messages. The encoding of the messages is described in table 1 in the appendix.

Keeping track of the 16-Step Sequence Patterns with the LCD display:
Given that Arduino is not in charge of the metronome ticks, the only reason that the sequences for each channel is kept track of is for the LCD to display the channel sequence so that the user can see what is being done on each channel. Therefore the sequences of all the channels are stored within a two-dimensional array of size 11 by 16 (*channels[11][16]*). It contains the 16-step sequence for 10 channels, and an additional array for the "11th" channel which effectively provides an overview of which channels are active or inactive (any channel with at least 1 step that is ON is considered active).
As the user travels through the channels on the LCD using the Previous and Next channel buttons, Arduino keeps track of the channel that is currently being displayed upon the LCD screen. If at that moment, one of the 16-step buttons is selected/unselected, a message is sent to MaxMSP and the sequence for that channel is modified at the selected step index. The LCD then updates its display.
If the user goes to the last channel, effectively the "overview channel" (which is there strictly to provide visualisation over the states of each channel), the user will see "O"s and/or "-"s. An "O" indicates that the channel (whose number is given by the position of the "O") is currently active. An "-" means that that channel is inactive. The way Arduino keeps track of which channels are active and which ones are not is through a simple One-Hot Encoding scheme implemented in the *overviewCount*[number of channels] array. Every time a new step is selected in the nth channel, the value at index n in the *overviewCount* array is incremented by $2^{(step\ index)}$. Similarly, if a step is deselected, that value is decremented by $2^{(step\ index)}$. At the end, every index at which the *overviewCount* value is equal to 0 will be seen as a channel that is inactive. This algorithm is implemented in the *modifyArray()* function.

Other noteworthy aspects of the Arduino code:
For the user's convenience, when the user navigates from channel to channel using the *Previous* and *Next* channel buttons, all FXs for the previously viewed channel are automatically locked (Arduino sends a *lockAll* message). By doing so, if the user plays with the FXs on channel X, then goes to channel Y and tweaks with the knobs, then comes back to the channel X, the amount of the FXs will not frantically jump to the new potentiometer readings. Instead, the user will have to disable the lock on the FXs that are to be tweaked with.
Furthermore, if the user hits the reset button, Arduino will send a *reset1channel* message if the current channel is an actual channel (1 to 10), or else if that channel corresponds to the "overview channel", the message sent will be a *resetAll* message to reset all channels.

2. ***MaxMSP:***

Main Patch:

This patch has various purposes. First, it provides the set-up controls and contains the *SerialCommunication* patcher that receives the messages from Arduino. This patcher also contains the buffers for each channel, along with the channel-instrument selection, that make use of a *radiogroup* object (to select the channel) and 8 *chooser* objects (to select the instrument). Most importantly, this patch contains the last FX of the chain, that is a common compressor which receives the processed signals, before the signal is finally output through the *ezdac~* object. In addition, this patch provides access to the user interface for both operating modes, as well as a recording functionality to record the output.

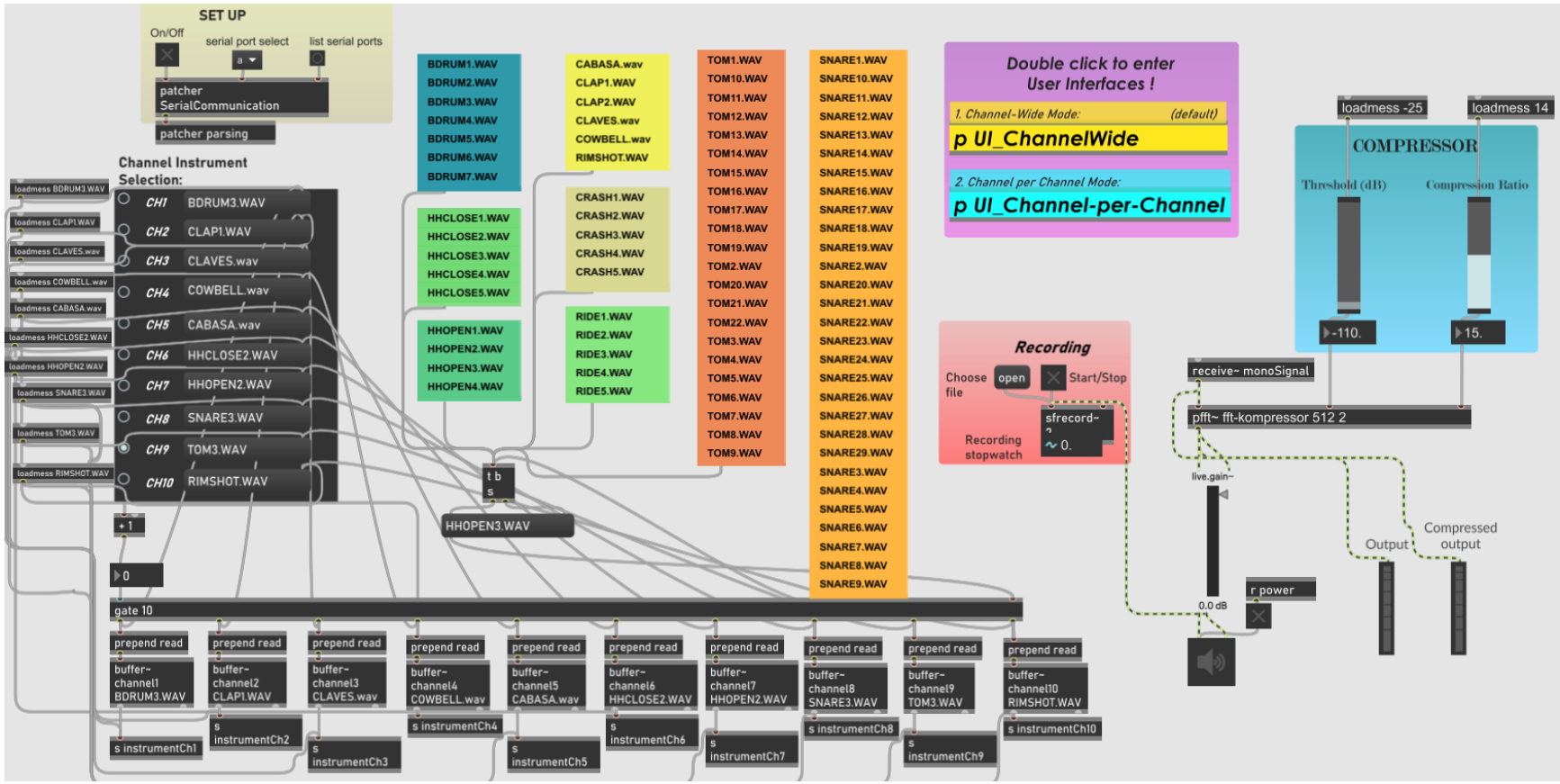

Figure 4: User interface of the home patcher in MaxMSP.

Patcher SerialCommunication:

The *SerialCommunication* patcher was taken from the communication part in *The beep-boop Jeff Mills Techno Machine.* 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. The *midiparser* object requires the content of the incoming messages to be separated by commas (e.g "160 2 90" should be "160, 2, 90"). This is what the patcher *addCommas* performs.

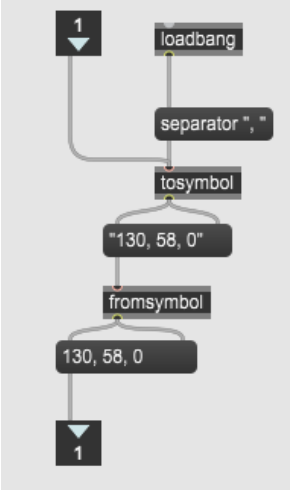

Figure 5: addCommas patcher in MaxMSP.

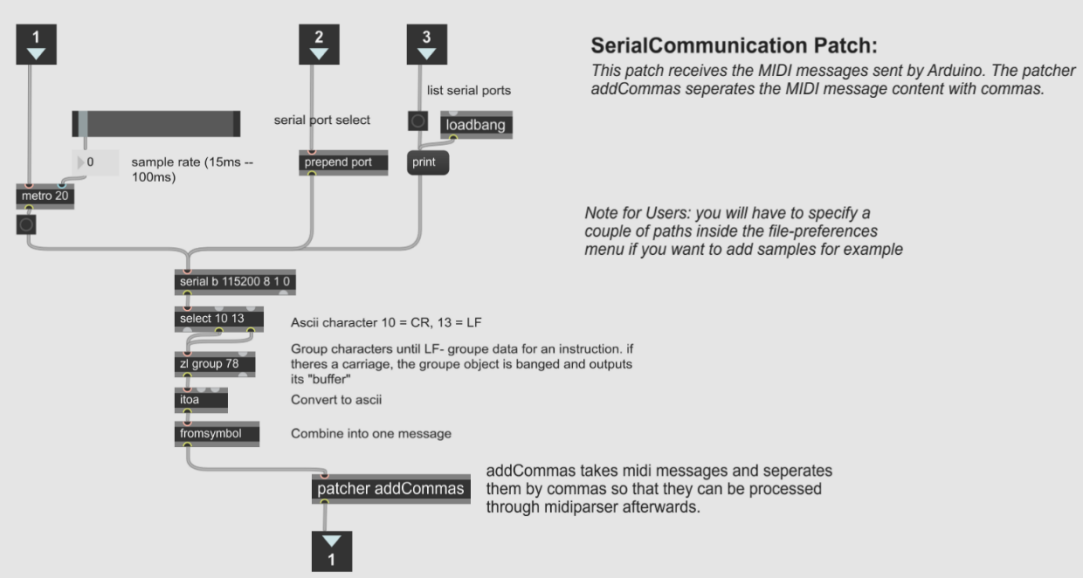

Figure 6: SerialCommunication patcher in MaxMSP.

Patcher parsing:

This patcher receives the message sent Arduino with each element of the message separated by a comma, and takes care of sending the information received to all the channel patchers (the message will then be gated in the channel patchers). The *Midiparse* object does most of the job in this patcher, for decoding the messages (see tables 1-3). As it can be seen in figure 7, some small manipulation had to be done for the Poly Key Pressure channel messages to correct the timing such that the FX parameter is sent first, followed just afterwards by the amount of that FX parameter.

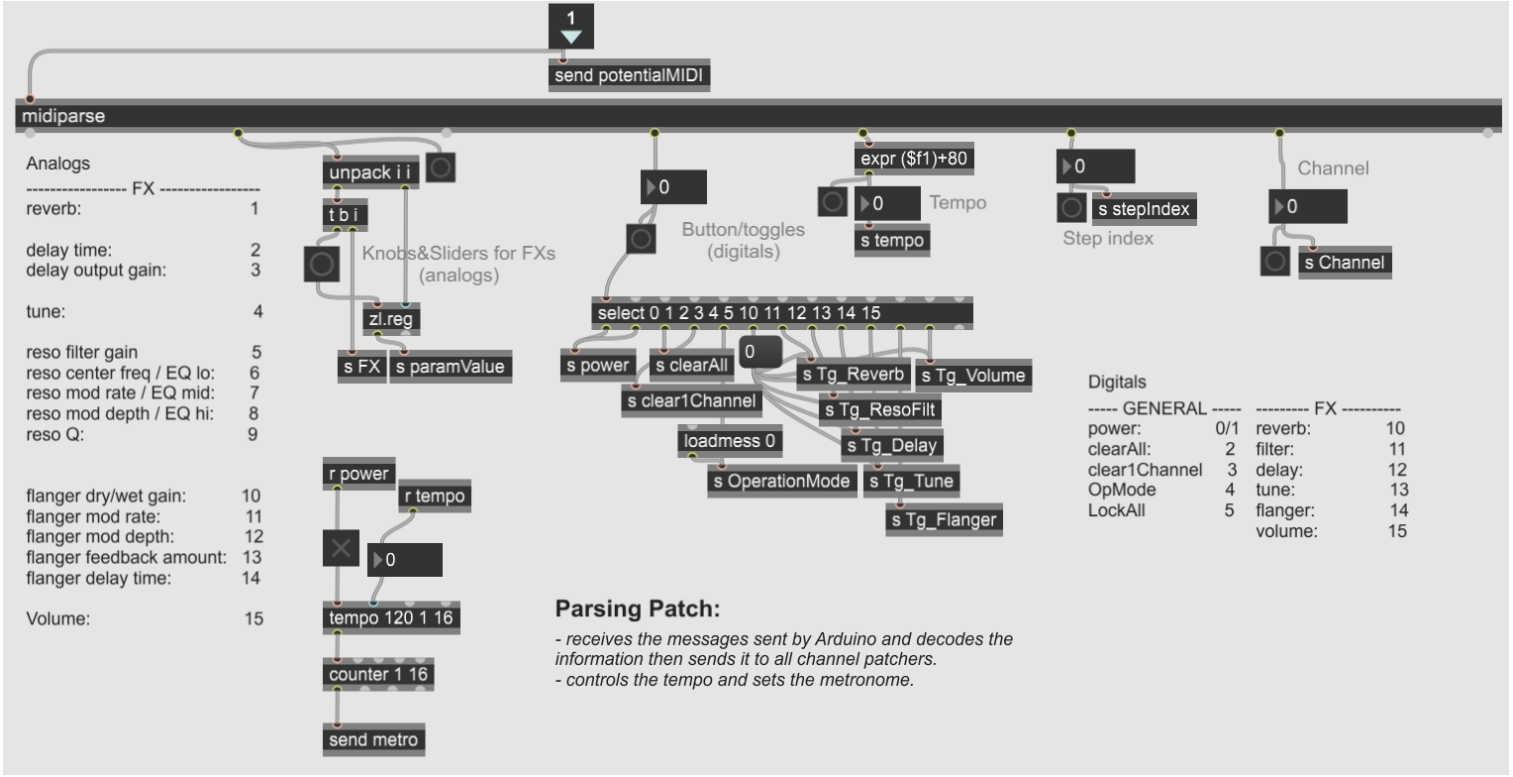

Figure 7: Parsing patcher in MaxMSP. Takes care of extracting and rerouting messages form Arduino.

Channel Patchers:

There are 10 channel patchers, one for each channel. They are all identical except for the *groove~* objects and the condition of the *gate* and *select* objects that differ. The information sent from the *Parsing* patcher is only processed in the channel patcher if it goes through the *gate/select* objects, that is, if the channel of the message sent from Arduino is the same as the channel of that channel patcher. If that is the case, if the message corresponds to the amount of an FX parameter, it will be sent to the FX parameter gate object where the value will be scaled appropriately (since all messages in MIDI have a value from 0-127) through one of the 15 objects below the long gate object. If the message corresponded to a step being selected/unselected from the channel pattern sequence, then the index of the step is passed through the *stepsequence* object. Similarly, if the message received is a *clear1channel*, then it will be passed to the *stepsequence* object where the sequence for that channel will be reset.

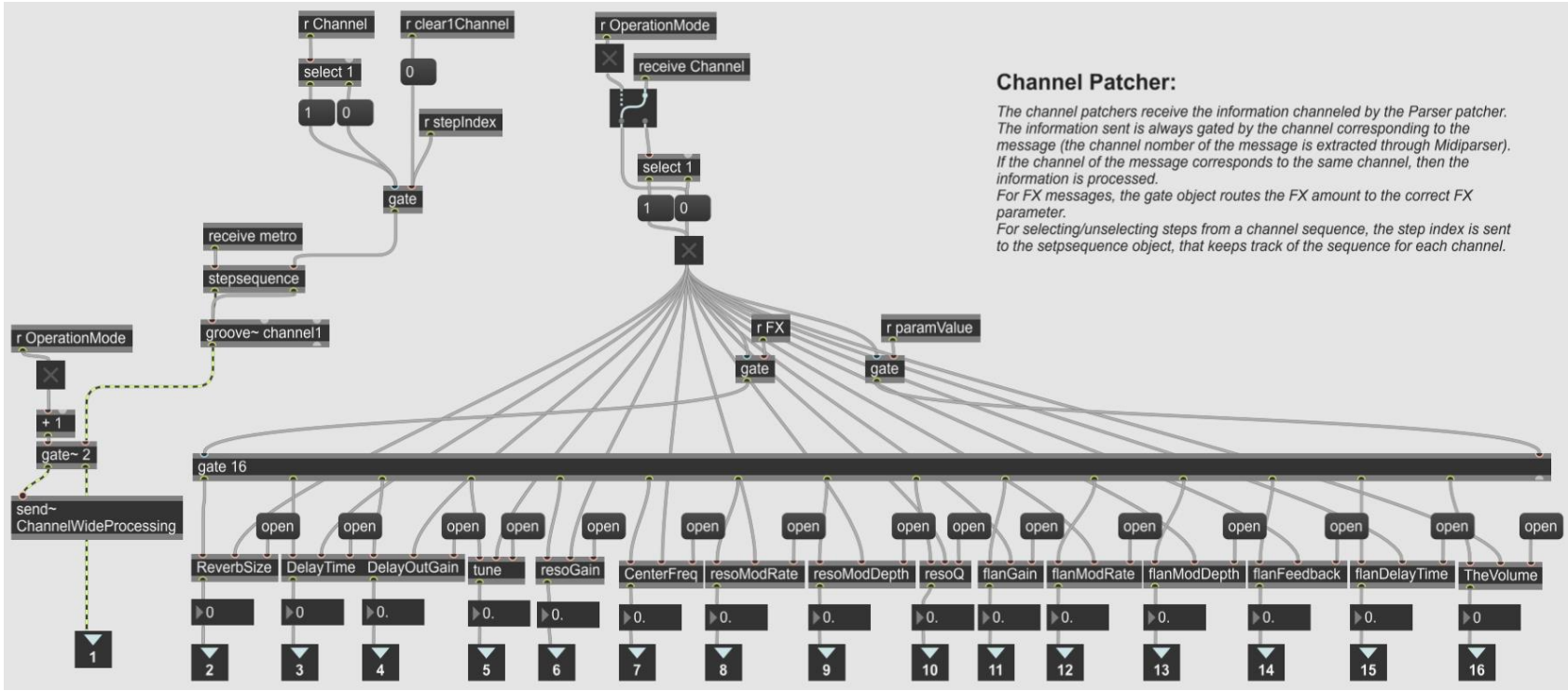

Figure 8: CH1 patcher in MaxMSP. (All Channel patchers are very similar)

Stepsequence (object):

The *stepsequence* object stores a 16-step sequence for a channel. It receives a counter that loops continuously from 1 to 16 at metronome speed, and triggers a buffer (connected to the object's outlet) every time it reads a step who's status is ON.

If the sequence is to be reset, it will a receive zero value in the select object which will reset all toggles to zero. Similarly, if a *clearAll* message is sent, the same will happen but for all instances of *stepsequence*.

At the output of the *groove~* object (which will play every time a step is active), the output goes to a gate that is gated by the operation mode. The two operation modes are *Channel-Wide* and *Channel-per-channel* mode. In *Channel-Wide* mode, the FXs are applied in the same amount to all channels, therefore the signal is sent through the *send~ ChannelWideProcessing* message to the *General* patcher, where all the channel's outputs will undergo the same amount of FXs.

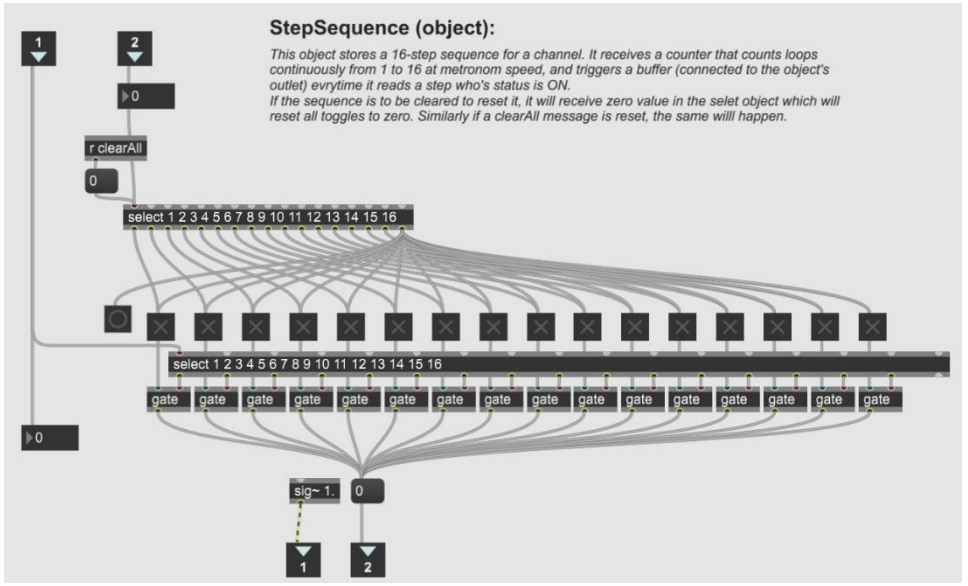

Figure 9: stepsequence object in MaxMSP, responsible for keeping track of the sequences of a channel.

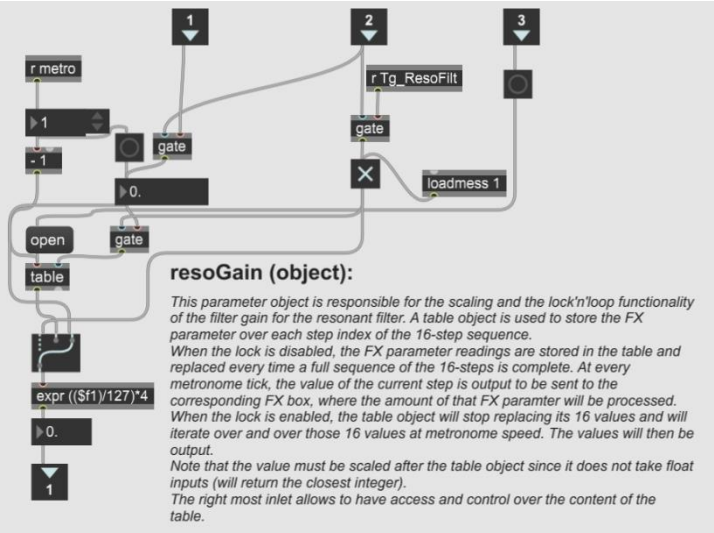

Figure 10: resoGain object in MaxMSP.

Parameter scaling patchers:

This refers to the 15 objects under the long gate object in the channel patchers. i.e: *ReverbSize, DelayTime, DelayOutGain, tune, resoGain, CenterFreq, resoModRate, resoModDepth, resoQ, flanGain, flanModRate, flanModDepth, flanFeedback, flanDelayTime, TheVolume.*

These 16 patchers oversee the *lock'n'loop* functionality of the FXs parameters, aswell as scaling appropriately the reading of the potentiometers from Arduino (a value ranging from 0 to 127), to the appropriate parameter value. Since all 15 patchers function the same way (they only differ in the amount by which they are scaled), the *resoGain* object will be taken as example in this report.

A *table* object is used to store the FX parameter over each step index of the 16-step sequence. When the lock is disabled, the FX parameter readings are stored in the table and replaced every time a full sequence of the 16-steps is complete. On the other hand, when the lock is enabled, the *table* object will stop replacing its 16 values and will iterate over and over those 16 values at metronome speed. At every

metronome tick, the value of the parameter on the current step is output to be sent to the corresponding FX block, where the amount of that FX parameter will be processed. Note that the value must be scaled after the *table* object since it does not take float inputs (will return the closest integer). The right most inlet allows the user to access the content of the table.

FX Patchers:

Whether the drum machine operates in *Channel-Wide* mode or in *Channel-per-channel* mode, the scaled FX parameters will be sent to their corresponding FX block patcher so that they can be used to process the signals.

Each FX block receives the values of the FX parameters, and contains the user interface and a *poly~* object that does the essence of the signal processing. Each On/Off toggle of the FXs are connected to a trigger object that takes care of selecting the signal to output (dry signal if Off, wet processed signal if On), aswell as sending a *mute* message to the *poly~* object if the FX is turned Off. The *mute* message disables the signal processing inside that object instance and hence, saves computaional power.

For the sake of non-redundency, the *Tune* FX is taken as example in this report. As seen in the *TuneFunc* object in figure 12, as well as in each object responsible for the signal processing of each FX, it contains a *thisPoly~* object. This is the object that takes the *mute* argument discussed previously.

The complexity of the patchers for processing varies from one FX to another. As it can be seen, the *TuneBlock* object was easier to implement since it has a single argument and that the *freqshift* object performs the frequency shift for us. Most FXs however have a more complex implementation (delay lines, feedback, etc). The essence of the implementation for the *Reverb*, the *EQ* and the *Compressor*, were provided by other people's work. They were found under "Sound Processing Techniques" in the MaxMSP documentation.

On the other hand, the *Tune*, *Delay*, *Resonant Filter*, and *Flanger* FXs were all built from scratch using various MaxMSP objects such as delay lines. A screenshot of the implementation of the different FXs can be found in the appendix.

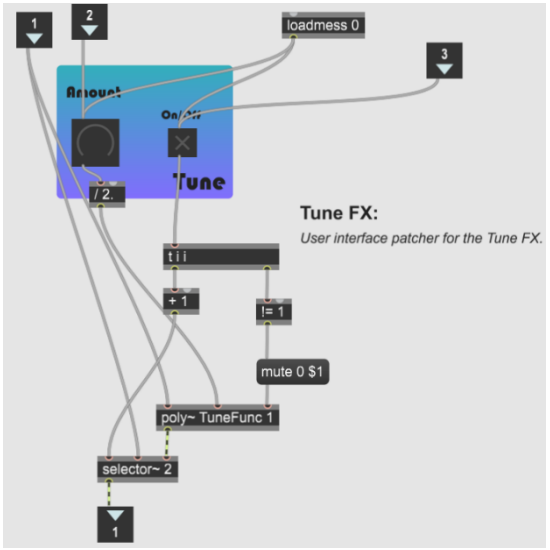

Figure 11: TuneBlock object in MaxMSP.

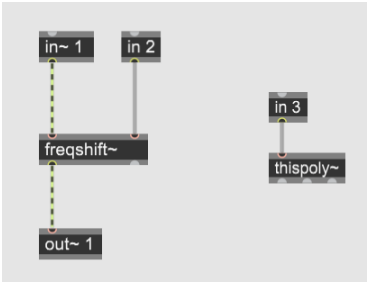

Figure 12: TuneFunc object in MaxMSP.

User Interface Patchers:

There are two different User Interfaces (UI) depending on the operating mode you choose. The *Channel-Wide UI* patcher has a single instance of each FX since all channels are sent this common FX chain when in Channel-Wide mode.

On the other hand, the *Channel-per-Channel UI* patcher has 10 different instances of each type of FX. This is, each channel has its own FX chain, which adds up to a total of 10x6=60 FX patchers (including the final gain slider).

A screenshot of both UIs is shown in figure 15 and figure 16, that can be found in the appendix. As shown in figure 13, the way the FX blocks were designed was to make it as easy as possible for the order of the FX chain to be modified. All the user must do is quit the MaxMSP presentation mode, unlock the patcher and play with the order at the user's convenience.

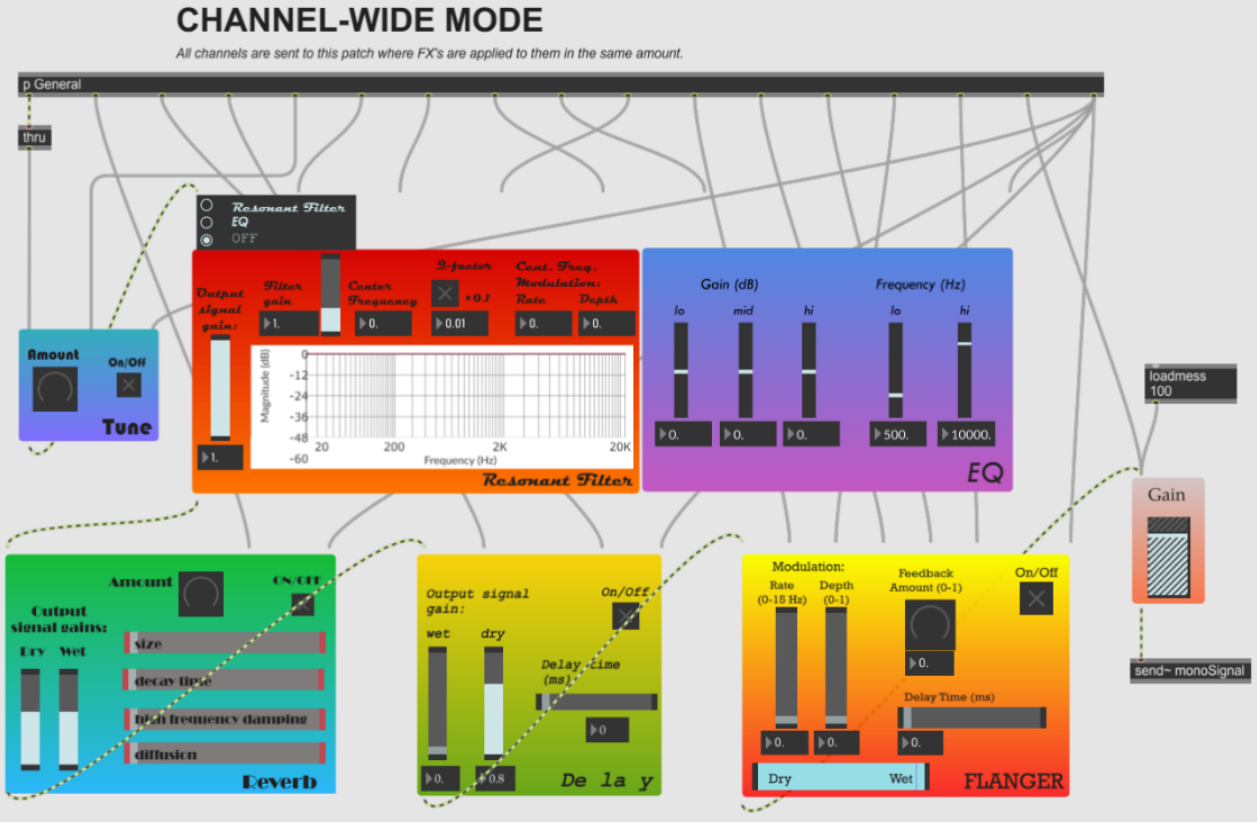

Figure 13: Unlocked view of the UI_ChannelWide patcher.

Design Choices

Reimplementation of last semester's simple drum machine ("*The beep-boop Jeff Mills Techno Machine"*):
The first task to be completed before any FXs could be added, was to change the fundamental implementation of the drum machine compared to the one from last semesters' final project. Therefore, the first task that was tackled was the implementation of a "live" and "free to modify however you want it" drum machine. The other part of reimplementing it had to do with timing performance. Unlike last semester, the metronome ticks of the drum machine were all controlled by MaxMSP instead of Arduino. This way, multiple channels could play simultaneously at the same time without getting timing offsets.

LCD display:
I didn't want to include the sequence pattern being played in the MaxMSP user interface (because it would take up to much screen space), however I needed a way for the user to keep track of the sequence being created. What made me choose the LDC display to fulfil this purpose was Jeremi's implementation of his drum machine last semester, which made use of the LCD display. Hence, on top of displaying the channels' sequence, the LCD allows to keep track of multiple functions: the state of the *lock'n'loop* functions for every FXs, an overview visualisation of the active/inactive channels, the option to reset all channels or one in particular, and the tempo.

Choice of FXs:
A lot of thought was put into which FXs I wanted to be included in my drum machine. At the end, I opted for a tune, a resonant filter or an equaliser, reverb, delay, a flanger, and a gain control. I would've liked to include an ADSR functionality, but this was found to be too difficult to implement on buffers, in addition to taking up 4 knobs.

Hardware Interface:
The design of the hardware interface took a lot of time, first, because there was a lot of hardware to setup, and second because I wanted to obtain the optimal disposal for the user. Hence, I choose to set the 1 button per step of the 16-step sequence all at the front-end of the drum machine. The Reset, On/Off, Operating Mode, Next and Previous Channel buttons were all placed at the front of the drum machine as well. The hardware for each FX is grouped together, from left to right: Filter, Delay, Reverb, Flanger, Tune, Volume.

Software Interface:
While creating *The beep-boop Machine*, I chose to give the MaxMSP software interface a lot of importance. It allows the user to turn FXs On or Off, and provides visualisation over the parameters that are being played with (e.g. the frequency response of the resonant filter). Furthermore, the choice of the instruments on each channel occurs in MaxMSP.

Lock'n'loop:
As I started working on this project early enough, I was able to implement my ideas as they came. The need for the *lock'n'loop* functionality arose from multiple inconveniences. The first one was discussed in the Arduino section, that is that without the lock enabled when navigating through channels, all the FXs would change suddenly when going from one channel that had different FX settings to the current readings of the knobs. Secondly, being able to vary FX parameters and then lock them added a tremendous functionality to *The beep-boop Machine*.

Channel-Wide vs Channel-per-Channel mode:
As explained before, this functionality provides the choice between either applying the FXs differently on each channel, or applying them in the same way to all channels.

Challenges and Limitations:

1. *Challenges:*

Routing messages:
One of the main challenges of this project was to get each message from Arduino to be delivered to the right place in MaxMSP. In the end, the combination of the *midiparser*, *gate* and *select* objects gated by the channel of the message received, and a couple of other tricks did the job. One particular problem I recall encountering had to do with the FX parameter amount messages that were received but channelled to the wrong parameters, that is, the delay time would be sent to the reverb, the reverb parameter would be sent to another FX. This was eventually resolved by placing a trigger object right after unpacking those message types from *midiparser*. This way it made sure that the parameter number would be sent before the amount of that FX parameter.

Lock'n'loop implementation:
Another challenge was the implementation of the *lock'n'loop* functionality. It took me some time to find a way of storing values and reading them over in a loop. Once I stumbled upon the *table* object, it took a bit of time but eventually I managed to implement the *lock'n'loop* functionality.

Limited number of analog pins:
An issue I anticipated was the hardware restrictions in terms of the limit of analog pins on the Arduino Mega board. With 16 pins available, of which one had to be used for the tempo, it left me with 15 FX parameters to implement. Although this seems to be a lot, when considering that some effects such as the Flanger, Reverb or Resonant filter all have 4 or 5 parameters each, I had to think of a way to reduce these numbers. I managed to decrease the number of knobs required by two means.
The first way I did it was for the reverb, for which I sought the right set of scaling factors by which the 4 parameters (size, decay time, high-frequency damping, diffusion) could be related. In the end, I expressed all 4 parameters as a function of the decay time: the size was scaled by

0.3 of the decay time, and both the high-frequency damping and diffusion are scaled by 0.5 of the decay time. This left me with a single knob instead of 4.
Similarly, for the EQ, I figured that I would rarely be using both the EQ and the resonant filter at the same time. Therefore, I reused 3 knobs of the resonant filter for the hi/mid/lo gains of the EQ.

Reduction of computational power:
This last challenge worth highlighting appeared very late in the project implementation. It only appeared when I started connecting all the FX blocks and made them play all together. The issue I was encountering was that my audio was lagging and could even come to a temporal stop at moments. I attempted to solve this problem by playing with the audio status settings of MaxMSP (vector sizes, etc.), however this effort was in vain. After some research, I found an old MaxMSP documentation (https://docs.cycling74.com/max6/dynamic/c74_docs.html#mspchapter05) that brought the solution to this problem by making use of *poly~* instances in combination to *mute* messages. *Poly~* provides instances of objects in MaxMSP that can contain *thispoly~* objects. When a *thispoly~* object receives a *mute* message, it will turn the signal processing off for that instance of the object. For example, when the *thispoly~* object in an instance of the flanger receives a *mute* message, that flanger instance will stop computing. This way, we prevent dozens of FX objects from continuously performing computations on null signals (e.g when an FX instance is turned off). By doing this, the issue was resolved.

2. ***Limitations***

Overall, the limitations to which my project is confronted remain mostly hardware oriented. That is the number of analog pins that are available. That being said, I think that it wouldn't be that great either to have a knob for every single FX parameter, as using a single knob for reverb proved to be sufficient and easier to use (instead of tweaking with 4 different knobs).

Moving forward…

If I were to pursue further the development of *The beep-boop Machine*, an immediate addition to the machine would be to have 16 LEDs right above each of the 16-step buttons, and have the LEDs turned on if the step is On, and turned off if that step is Off. This way it would be easier for the user to tell which steps are on and which ones are off. Despite having an LCD display where the sequence of the channel is displayed, it remains slightly difficult to tell the exact position of each step on the LCD display (for example I sometimes erroneously select say step 10 instead of step 9 that I wanted to select).

A part for that small accessory addition, I think if I want to go any further in the development of the drum machine, I will eventually have to switch from Arduino to some other more audio-suited microcontroller such as Teensy.
Finally, a nice feature for my drum machine would be to have it hard-cased. I have no idea where I can do such thing so I would be very happy if you have a website or such in mind where this can be done.

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

Appendix

Message Name ***(Status byte)***	**Data Byte 1 (0-127)**	**Data Byte 2 (0-127)**
Poly Key Pressure *(0xAc)*	FX parameter number (1 number attributed to each FX parameter)	FX parameter amount applied
Program Change *(0xCc)*	Toggle number (1 number attributed to each button)	_
After Touch *(0xDc)*	Tempo value	_
Pitch Bend *(0xEc)*	Index of the step selected (1 to 16)	_

Table 1: Encoding of the Arduino hardware readings in MIDI formatted messages.

Data Byte 1 (Prog. Change messages)	**Corresponding action**
0/1	Sequencer On/Off
2	Clear All Channels
3	Clear a single Channel
4	Operation Mode Toggle
5	Lock All FX's
10	Lock/Unlock Reverb
11	Lock/Unlock Filter
12	Lock/Unlock Delay
13	Lock/Unlock Tune
14	Lock/Unlock Flanger
15	Lock/Unlock Volume

Table 2: Encoding scheme for the toggle reading messages from Arduino to MaxMSP.

Data Byte 1 (Poly Key Pressure messages)	**Corresponding parameter *(FX)***
1	Reverb amount (*Reverb*)
2	Delay time (*Delay*)
3	Delayed output gain (*Delay*)
4	Tune amount (*Tune*)
5	Filter gain (*Resonant filter*)
6	Centre frequency/Lo (*Resonant filter/EQ*)
7	Modulation rate / Mid (*Resonant filter/EQ*)
8	Modulation depth / Hi (*Resonant filter/EQ*)
9	Q-factor (*Resonant Filter*)
10	Dry/Wet signal ratio (*Flanger*)
11	Modulation rate (*Flanger*)
12	Modulation depth (*Flanger*)
13	Feedback amount (*Flanger*)
14	Delay time (*Flanger*)
15	Gain

Table 3: Encoding scheme for the analog FX parameter reading messages from Arduino to MaxMSP.

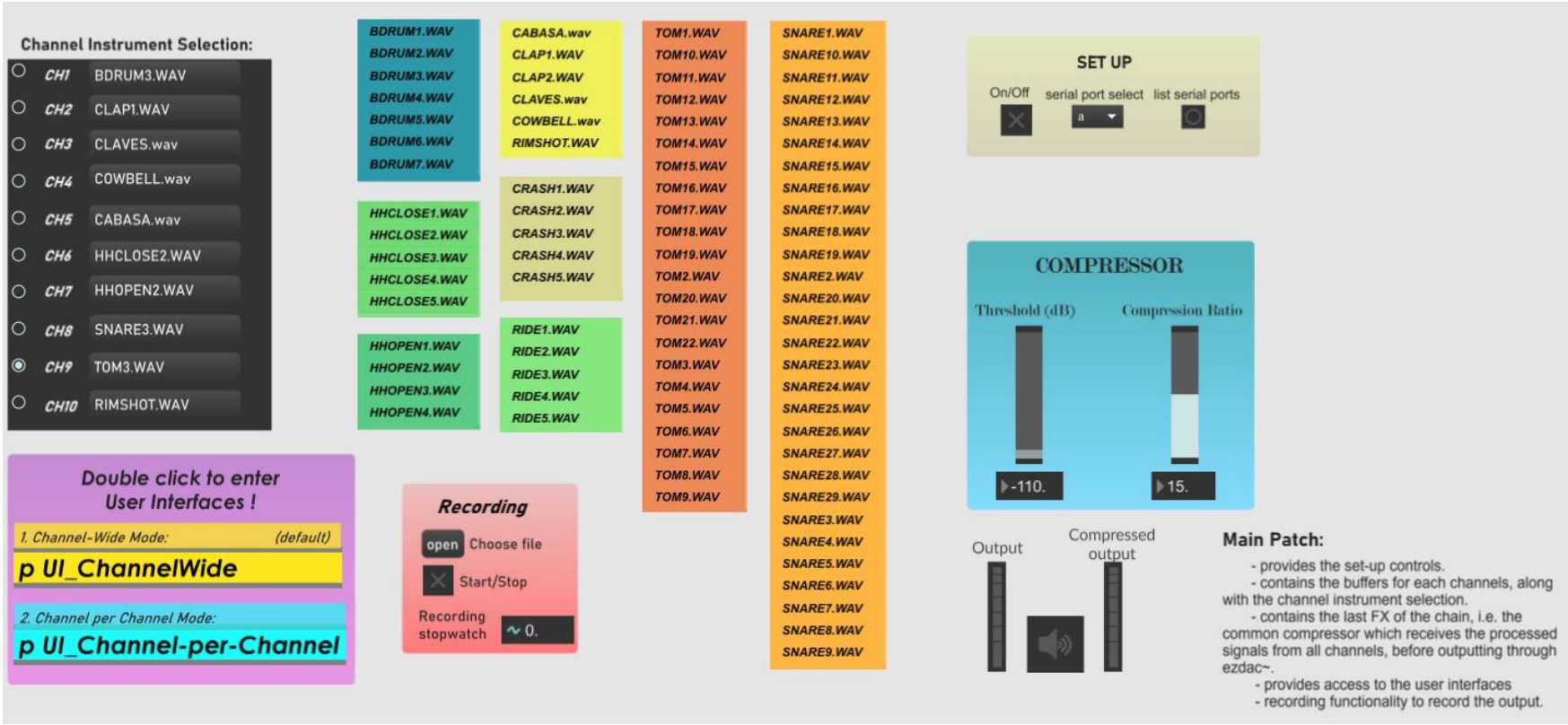

Figure 14: The MaxMSP User Interface of the home menu in presentation mode.

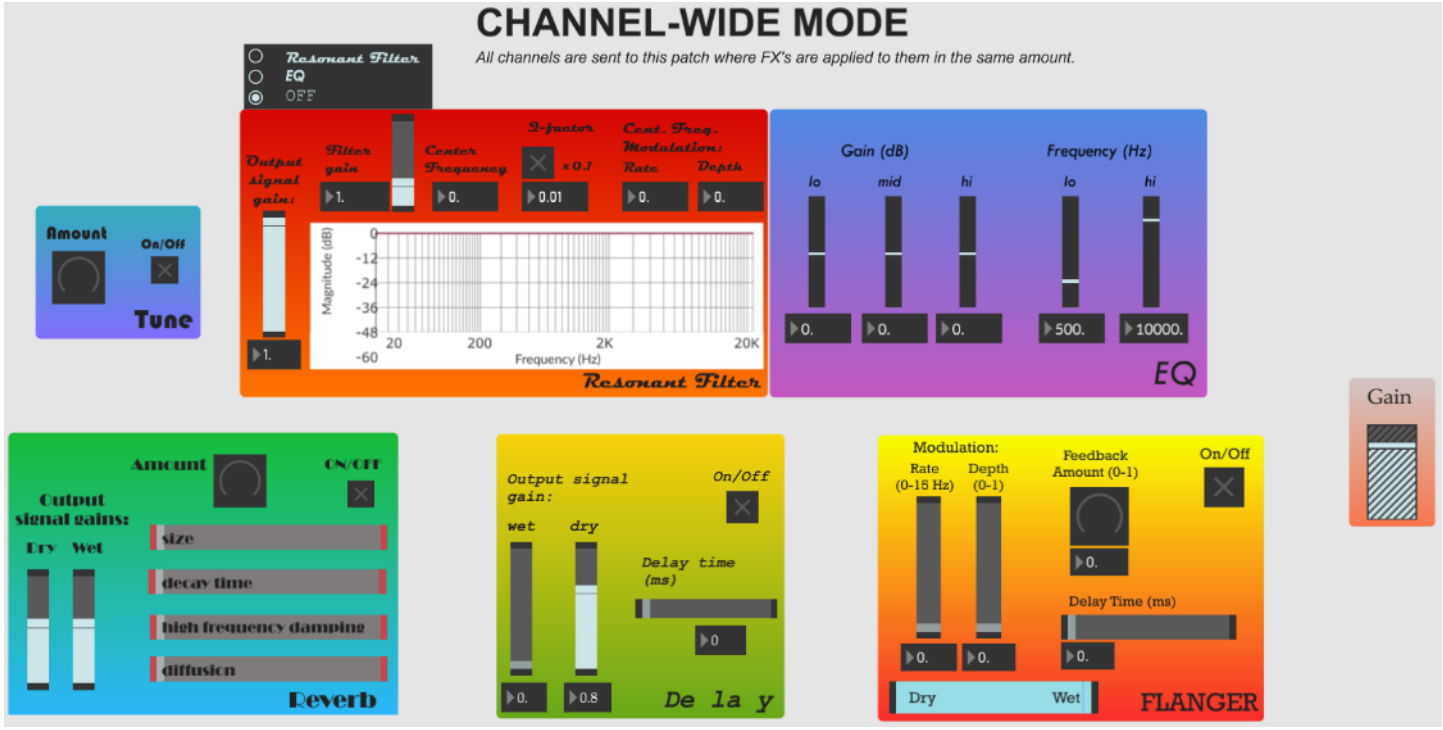

Figure 15: The MaxMSP User Interface for the Channel-Wide mode in presentation mode.

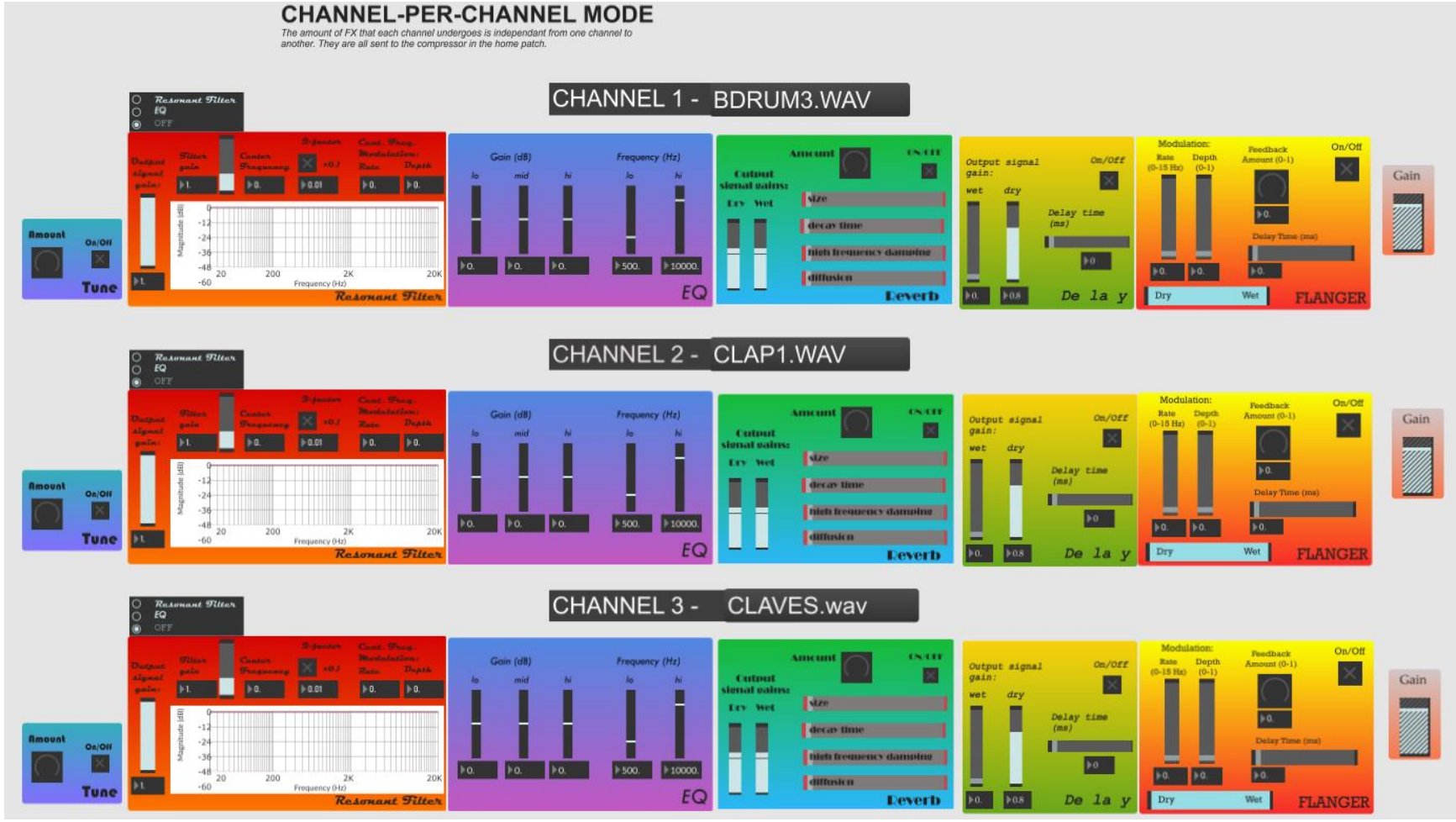

Figure 16: The MaxMSP User Interface for the Channel-per-Channel mode in presentation mode.

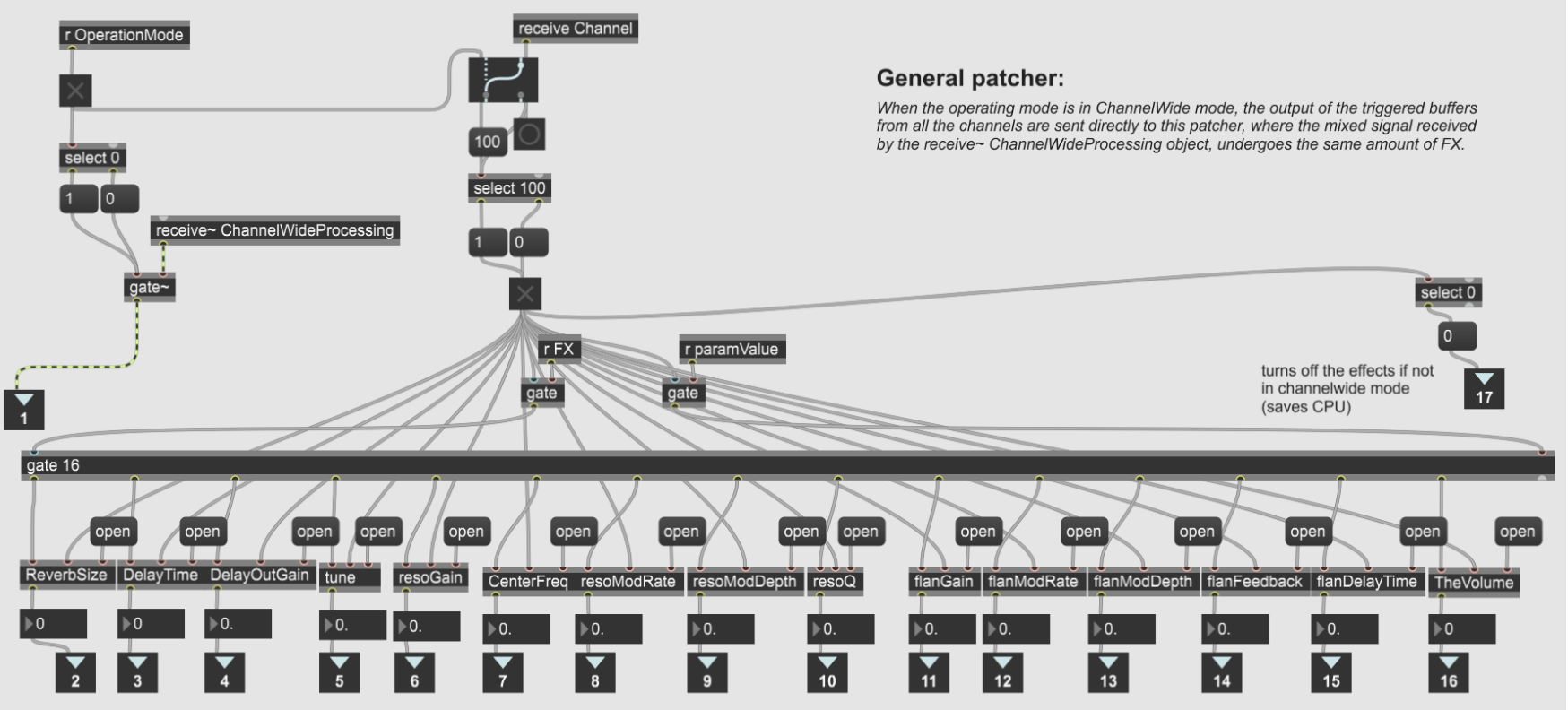

Figure 17: The General patcher in MaxMSP. The General patcher oversees receiving and distributing the FX parameters when operating in Channel-Wide mode.

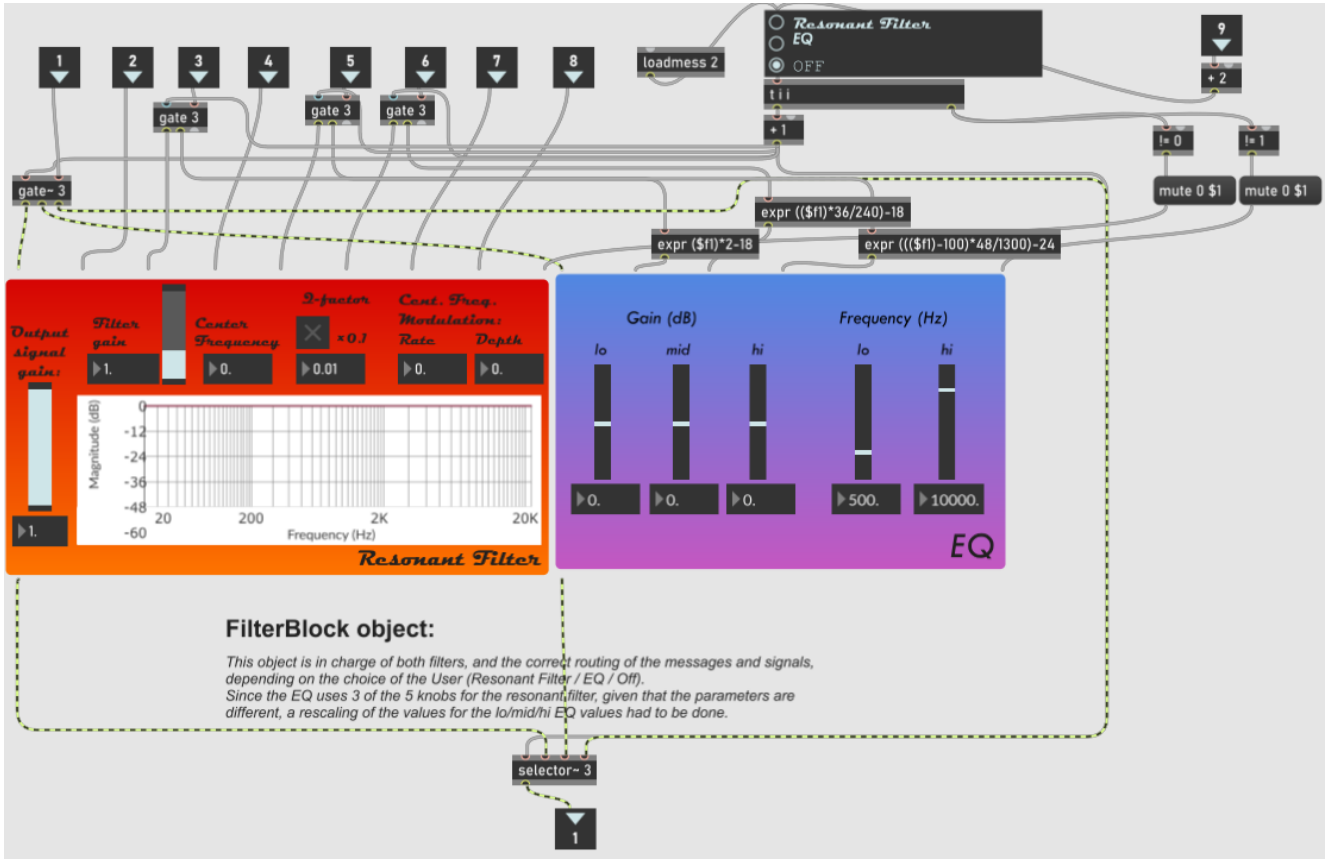

Figure 18: The FilterBlock object in MaxMSP.

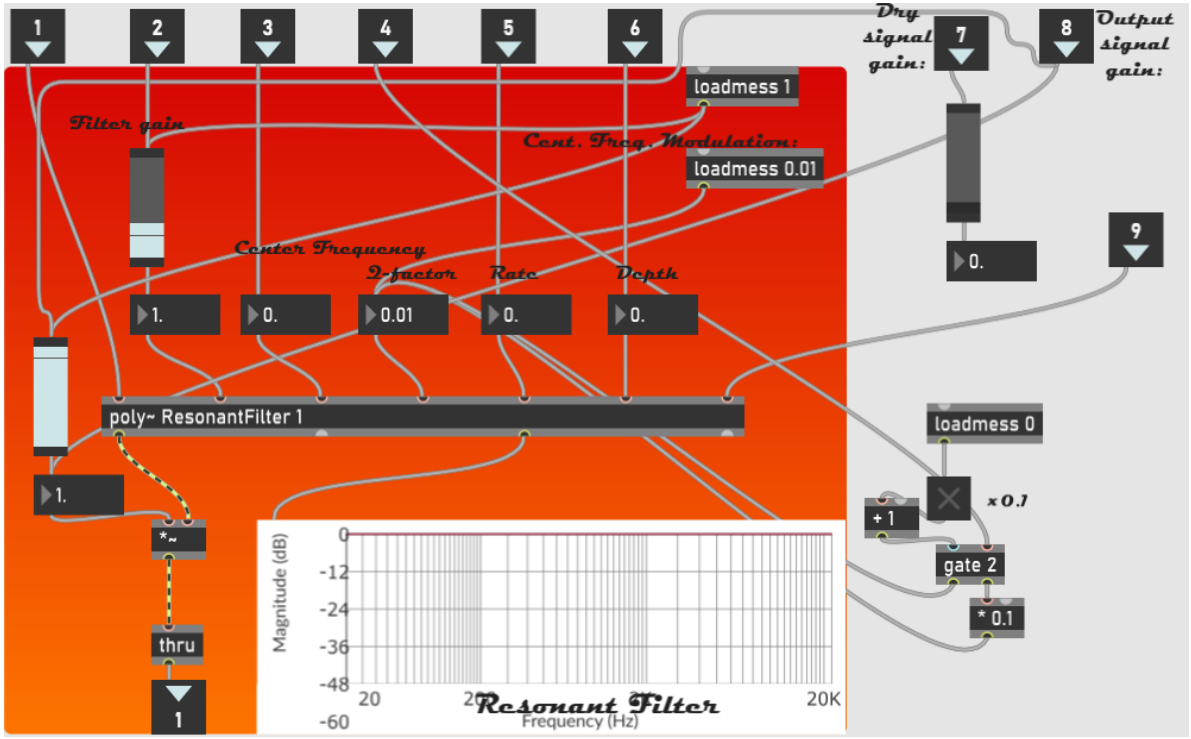

Figure 19: The ResonantFilterBlock object in MaxMSP.

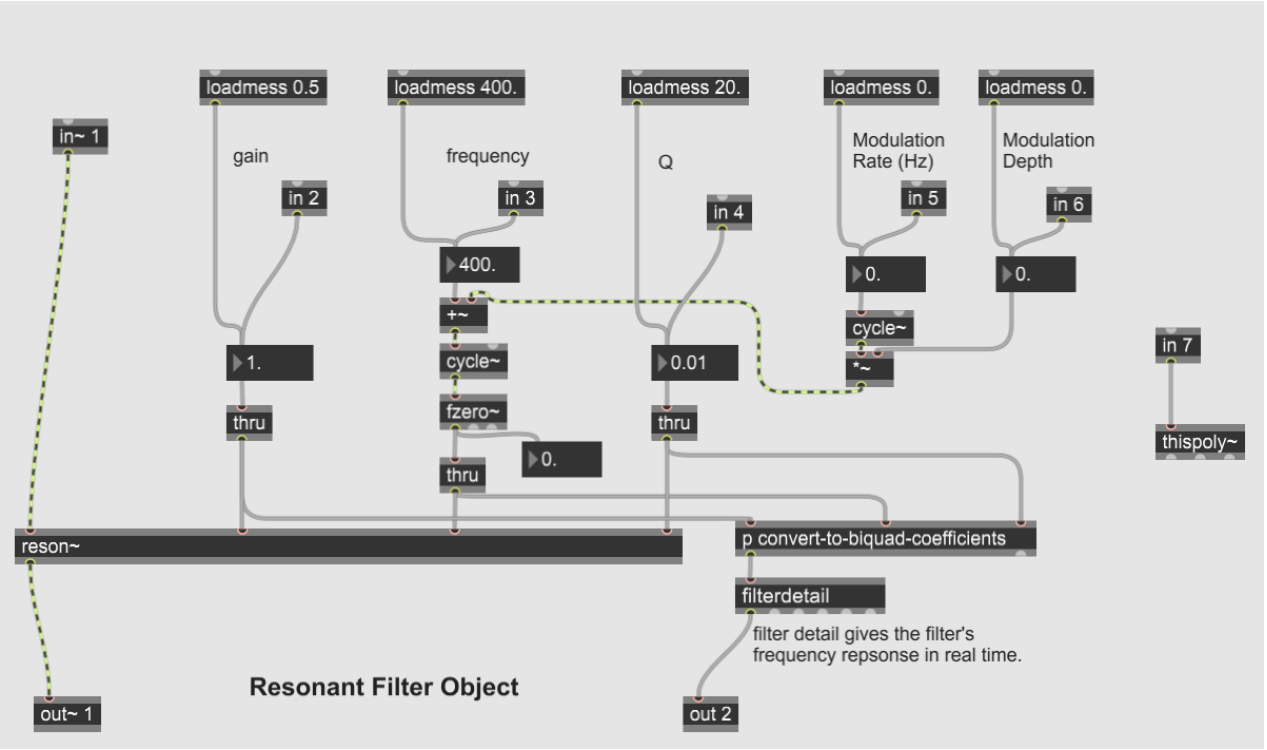

Figure 20: The ResonantFilter object in MaxMSP.

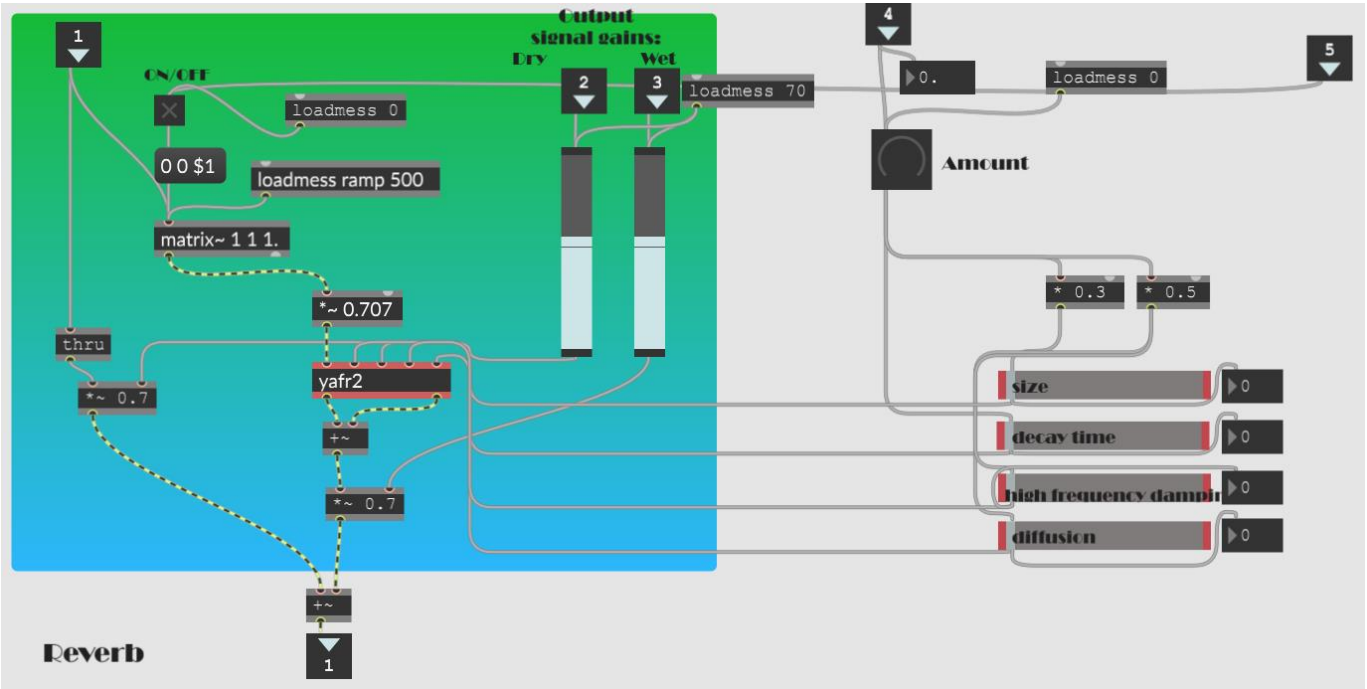

Figure 21: The ReverbBlock object in MaxMSP.

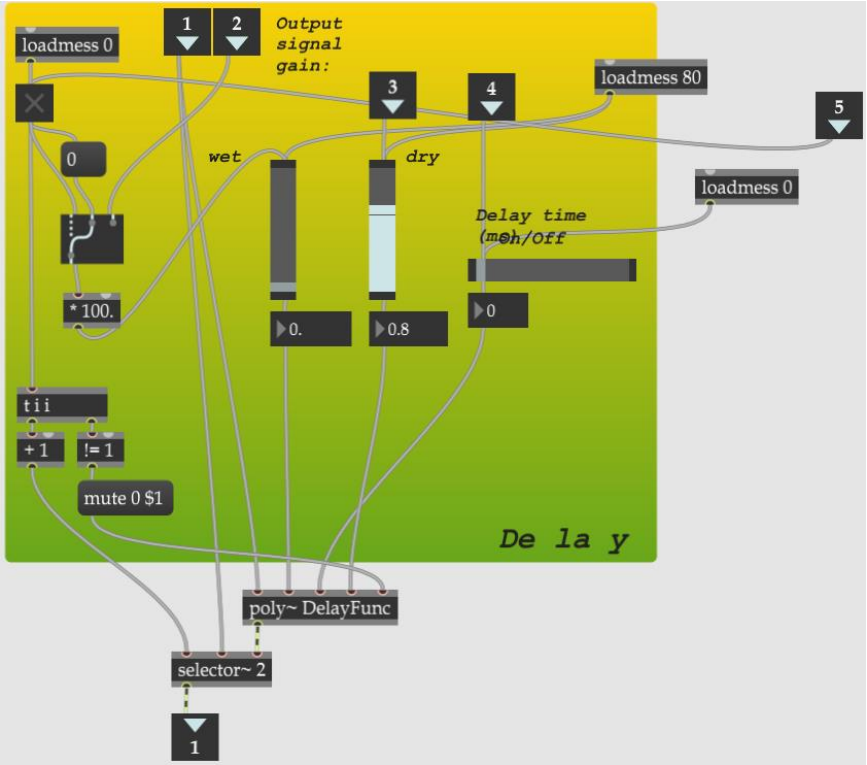

Figure 22: The DelayBlock object in MaxMSP.

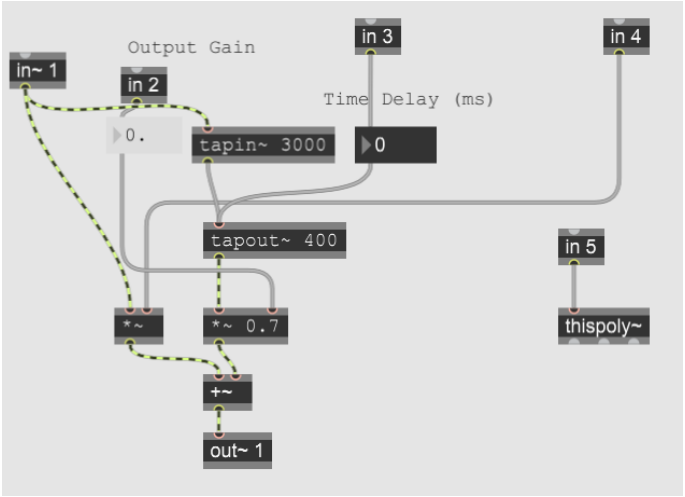

Figure 23: The DelayFunc object in MaxMSP.

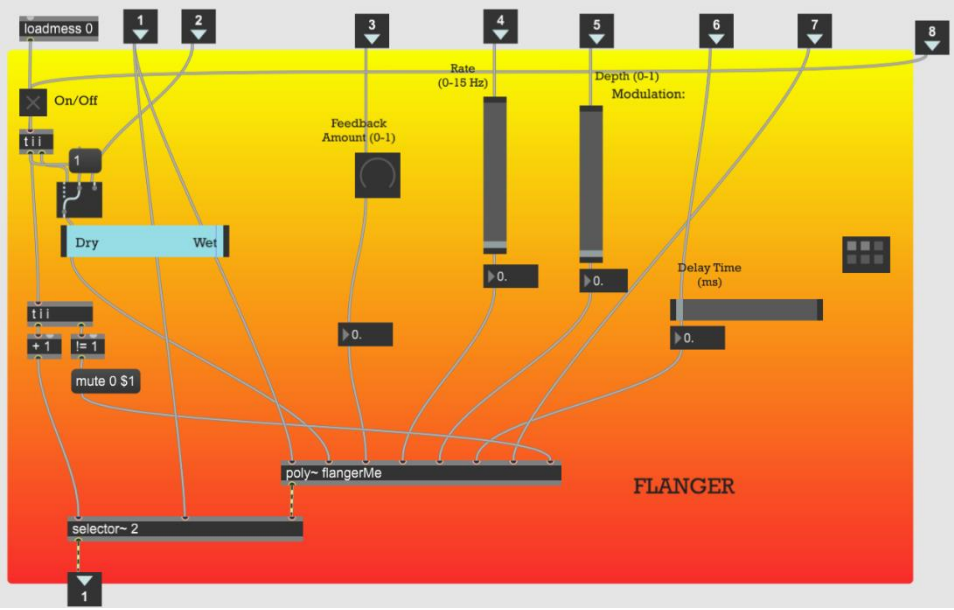

Figure 24: The FlangerBlock object in MaxMSP.

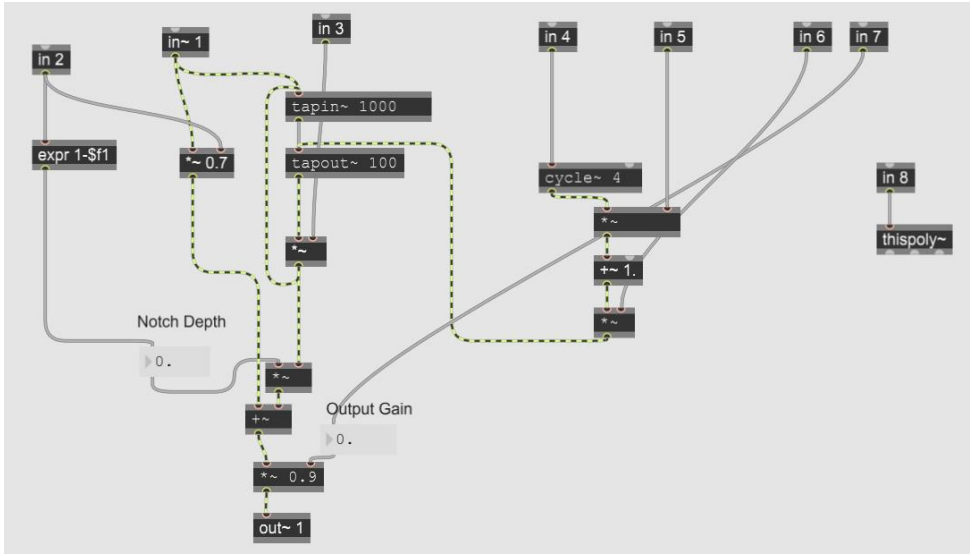

Figure 25: The flangerMe object in MaxMSP.

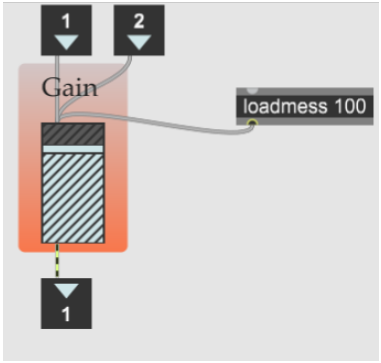

Figure 26: The GainBlock object in MaxMSP.