Building Live Loopers in Max

Design, understand, and perform with live looping systems in Max, gen~, and RNBO by building them from the ground up.

by Jeff Kaiser & Gregory Taylor

Building Live Loopers in Max

by Jeff Kaiser and Gregory Taylor

Copyright © 2026 Technophony
All rights reserved.
ISBN: 9798249419028

Dedication

This project is dedicated to the memory of our friend

Darwin Grosse

who constantly asked, *“where's the fun?!”*

reminding us why we build, listen, and play.

Table of Contents

Introduction: Thinking About Looping 1

Chapter 1: Tape-Style Looper (Max) 11

Chapter 2: Multitrack Looper (Max) 43

Chapter 3: Basic Looper (gen~) 75

Chapter 4: Varispeed Overdubbing Looper (gen~) 111

Chapter 5: Tape-Style Looper (rnbo~) 121

Chapter 6: Multitrack Looper (rnbo~) 135

Chapter 7: Basic Overdubbing Looper (rnbo~) 151

About the Authors

Jeff Kaiser is a trumpet player, composer, technologist, and scholar whose creative practice spans acoustic performance, interactive electronics, and software development. He is Associate Professor of Music Technology and Composition at the University of Central Missouri. Kaiser is the developer of the widely used KaiserLooper platform and several controller packages available through Cycling ’74’s Package Manager. An active international performer, Kaiser has presented concerts and research throughout Europe, Asia, and the Americas. He founded the pfMENTUM label (with over 150 releases), co-directs the MOXsonic Festival, and performs in multiple electroacoustic ensembles.

Gregory Taylor is the author of **Step by Step: Adventures in Sequencing with Max/MSP** and **Generating Sound & Organizing Time: Thinking with gen~ (Book 1).** Trained as a visual artist, he has studied central Javanese gamelan and electroacoustic music in the United States and the Netherlands. Taylor hosts a radio program devoted to contemporary audio on WORT-FM, and his recordings have been released on the Palace of Lights, c74, Clang, Nachtstück, Flood, Spectropol, Stasisfield, and ARTlevel labels. His work brings together deep technical engagement with Max programming and long-standing involvement in experimental, cross-cultural, and process-oriented musical practices.



The authors, Jeff Kaiser and Gregory Taylor, a long time ago on a rainy day in Amsterdam.

Chapter Descriptions

An Introduction — Thinking About Looping

- Frames live looping as both a musical practice and a way of thinking, situating it within broader musical traditions of repetition, pattern, and expectation. The introduction outlines the aesthetic, technological, and pedagogical concerns that shape the book and explains how the chapters and patching examples are structured.

Chapter 1 — Tape-Style Looper (Max)

- Introduces a fixed-length, tape-inspired looper using basic Max/MSP objects, providing a foundation for live looping in Max. Through hands-on patching, the chapter covers audio input/output, gating, feedback, loop length control, signal processing within the loop, and the creation of a performance-oriented user interface.

Chapter 2 — Multitrack Looper (Max)

- Explores the construction of a multitrack looper using standard MSP objects in Max. Topics include recording, playback, synchronization, and managing multiple asynchronous or synchronous looper tracks.

Chapter 3 — Basic Looper (gen~)

- Focuses on building an overdubbing looper in Max using gen~, emphasizing memory management, counters, and message sequencing. This chapter introduces techniques for layering recorded material while preserving loop structure.

Chapter 4 — Varispeed Overdubbing Looper (gen~)

- Recreates a delay-style looper in the gen~ environment, highlighting low-level signal processing, sample-accurate control, and the ability to overdub at variable speeds and directions—even while speed is being modulated. The chapter also introduces the gen~ codebox for advanced users, demonstrating how gen~ enables efficient and flexible looping architectures.

Chapter 5 — Tape-Style Looper (rnbo~)

- Builds a tape-style looper using RNBO, adapting earlier designs to a portable, exportable environment. The chapter introduces RNBO-specific workflows, parameter handling, and preparation for cross-platform deployment.

Chapter 6 — Multitrack Looper (rnbo~)

- Develops a multitrack looper in RNBO using abstractions and shared synchronization data. This chapter emphasizes scalable design, inter-track alignment, and preparing complex loopers for performance and export.

Chapter 7 — Basic Overdubbing Looper (rnbo~)

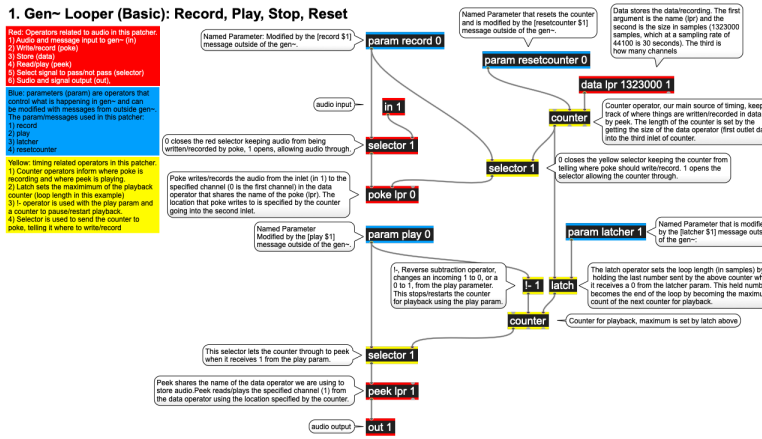
- Recreates the overdubbing looper in RNBO, integrating recording, playback, overdubbing, speed, direction, and declicking. The chapter highlights message ordering, parameter control, and performance-oriented UI design.

Downloads / Example Patches / Companion Software

- Download the complete companion software for *Building Live Loopers in Max*, including all patching examples from Chapters 1–7. This package works in Max 9 available from Cycling 74.
- https://jeffkaiser.com/product/building-live-loopers-max-patchers/

Introduction:

Thinking About Looping



This is a book about live looping—a musical practice that attracts fierce advocates and determined skeptics in equal proportions.

For skeptics, looping can seem reductive: a practice that risks flattening musical nuance, encouraging predictability, and linear development. Some might see it as a shortcut that substitutes mechanical repetition for musical development, while others critique what they perceive as aesthetic clichés that have grown around loop-based performance. These concerns—whether philosophical, stylistic, or practical—form a crucial critique that advocates might consider.

For advocates, however, looping is about the power, joy, and discipline of working with repetition and patterned time, as well as the inherent creative possibilities and challenges that arise from it. Live looping is

more than a technique—it becomes a way of thinking musically that emphasizes both structure and freedom. It is about the power and joy of building layered textures in real time, of listening closely and responding to patterns as they unfold, and of discovering subtle shifts in rhythm, harmony, and timbre that repetition can reveal. Looping allows performers to explore the interplay between anticipation and surprise. To its proponents, at its best, every repeated phrase is an opportunity: a chance to deepen expression, uncover hidden connections, and craft immersive sonic landscapes that reward careful attention. The practice demands focus, timing, and technological skill, yet offers a profound sense of satisfaction as small motifs can then transform into complex, dynamic musical structures.

Live Looping, Repetition, and Musical Thinking

The use of musical events that repeat over time is a defining feature of many musical traditions, spanning cultures and eras. Repetition lies at the heart of so many musical forms, such as rounds in folk music, call-and-response in blues and gospel, sonatas in classical music, trance-based musics, and, of course, minimalism.

This repetition establishes patterns of expectation that guide physical and imaginative engagement. Looping is the deliberate managing, shaping, and exploiting of these expectations through the intentional capture, configuration, reconfiguration, and deployment of repeated sonic material to serve creative ends.

Contemporary looping is also a practice with deep roots in experimental Western art music, including the work of Pierre Schaeffer, Terry Riley, Alvin Lucier, Pauline Oliveros, Gordon Mumma, Steve Reich, as well as collaborations by Robert Fripp and Brian Eno, and artists such as David Torn, Laurie Anderson, Pamela Z, and William Basinski. Looping has found love in popular music among artists like KT Tunstall, Imogen Heap, Reggie Watts, Ed Sheeran, St. Vincent, and many more.

In this book, we will explore technologies that make live looping possible. The advent of recording, and later the emergence of digital music, has enabled musicians to use technology to create forms of repetition that build on these familiar kernels of expectation, generating and organizing new forms of variety from them.

Technological developments inevitably shape the work musicians produce. For example, the invention of Western musical notation not only made music portable—allowing it to be learned independently of the composer—but also facilitated the creation of highly complex repeatable forms. A composed fugue, in this context, whose interweaving loops are subject to creative transformations, would be challenging to learn/explain without notation.

For practitioners of looping, the development of recording technology was equally transformative as the development of notation. The shift from ephemeral sound vibrations to a physically tangible medium enabled new forms of repetition, manipulation, and complexity. Many of these transformations are reflected in the software explored in this book. The first looper you will make recreates a foundational innovation: a fixed-length fragment of recorded sound—originally captured on tape—repeated to establish a pattern, which can then be modified through processes such as delay, slowing down, speeding up, reversal, and even ways to mimic the degradation of tape as it plays back.

The introduction of digital signal processing (DSP) further extended these possibilities. By decoupling sonic manipulation from the physical recording, musicians gained the ability to transform live input in real time, generating multiple, complex, interacting patterns that would be challenging, if not impossible, in the tape realm. This capacity for manipulation allows for the creation, organization, and repetition of sonic variety in entirely new ways.

Today, hardware loopers make multitracked recording accessible with remarkable ease, building directly on the principles of pattern and expectation described earlier. This book begins with emulation, before

progressing to software-based tools that move beyond the metaphor of the physical tape fragment: offering new creative possibilities for the modern practitioner.

Building Live Loopers offers a practical framework for understanding how repetition organizes perception, but it also functions as a pedagogical tool: a way to cultivate programming habits, internalize concepts, and reflect on the tools with which we think (and create).

In the chapters that follow, we construct a series of patchers that begin with that simple looper modeled on the familiar metaphor of a length of recording tape. From there, we progressively introduce and examine more advanced forms of manipulation made possible by DSP. None of these patchers is intended to represent a linear progression toward a single definitive design: instead, each stands as a complete, usable tool—examples to serve as a point of understanding, as well as rich and open enough to provide a point of departure to support exploration, modification, and personal adaptation.

Every looper we build serves as an invitation to develop your own approach. Programming in Max rewards clarity and directness, and one of its most productive lessons is the value of simple solutions deployed thoughtfully.

Just as repetition provides a framework through which listeners form expectations in music, we have structured this book to operate in a similar manner. And just as composition often relies on tension, the process of approaching and solving a series of small programming or musical problems introduces its own form of tension—one that can be productive, clarifying, and even reassuring. By revisiting ideas, techniques, and patching strategies in various patching environments (Max, gen~, RNBO) throughout the text, we aim to cultivate familiarity and confidence while introducing you to different solutions for similar situations. Each recurrence is meant to guide your understanding of what comes next by building on what you have already encountered. Our hope is that this return—conceptually and pedagogically—creates a

learning environment that mirrors the musical practice at the heart of the book: repetition as a means of grounding, orienting, and opening space for new possibilities.

How to Use This Book

This is a book about building loopers. So what do you need in order to begin?

We have designed this book to function equally well as a guide for self-directed study and as a resource in a classroom setting. You will not need an extensive mathematical background, nor do we assume prior experience with coding. We do, however, assume a basic familiarity with Max: enough to create and edit simple patches and to navigate the environment with reasonable comfort.

As you work through the chapters, there are several ways you might engage with the material and the accompanying patching examples:

- You can simply download the example patches and explore them as part of your own creative practice, without even reading the book.
- You can use the text as a guide to locate and modify specific components of a patcher, as each chapter sequentially steps through the addition of functions and features.
- Depending on your familiarity with Max/MSP and your own interests, you may pursue these ideas in a different order than the one we present. We have intentionally avoided building a single, “complete” looper. Instead, each chapter provides a fully functional variant that can serve as a starting point for your own investigation.
- Or, you may choose to “patch along”—individually or in a classroom—following the incremental construction of each patcher as a way to observe how an experienced Max programmer might frame, approach, and solve problems.

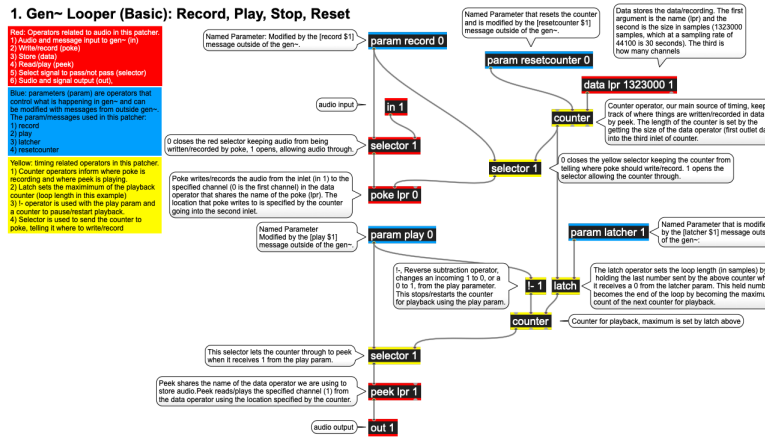
We encourage you to combine these approaches as needed, moving back and forth between text and examples, constructing the kinds of cognitive “loops” from which insight often emerges.

For readers new to Max, RNBO, M4L, or gen~, each chapter has been structured to minimize assumptions about technical expertise beyond what is stated above. When we introduce unfamiliar concepts or describe a technique that will recur throughout the text, we provide explanatory notes and guidance as to how to proceed. Our goal is to support you as you build fluency—whether this is your first encounter with these environments or a return prompted by new creative aims.

We will return frequently to the practice of building similar tools in different environments, exploring variations across Max, RNBO, and gen~. We have structured the chapters themselves as repeated approaches to explaining patchers, reinforcing the idea that programming is as much about habits of thought as it is about discrete technical skills.

We recognize that a “patch-along” format presents a practical challenge: it requires a clear view of the starting point, the intermediate steps, and the completed structure. To address this, the patches in most chapters are organized sequentially, with each step building directly from the previous one.

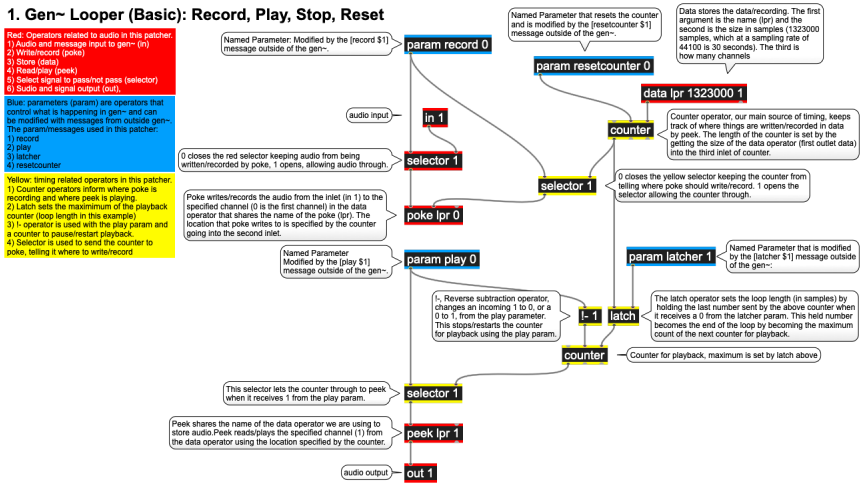
To make this process legible, we have color-coded the examples so that additions and changes are immediately visible. The color system is designed to highlight the functional roles of new objects or operators:



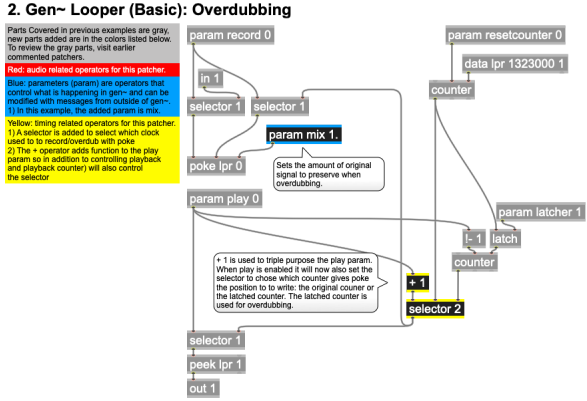
- **Red** indicates objects and operators related to audio-processing.
- **Yellow** highlights objects and operators related to timing.
- **Blue** marks objects and operators related to parameters that can be adjusted (either directly or via messages) to influence behavior inside the patcher.

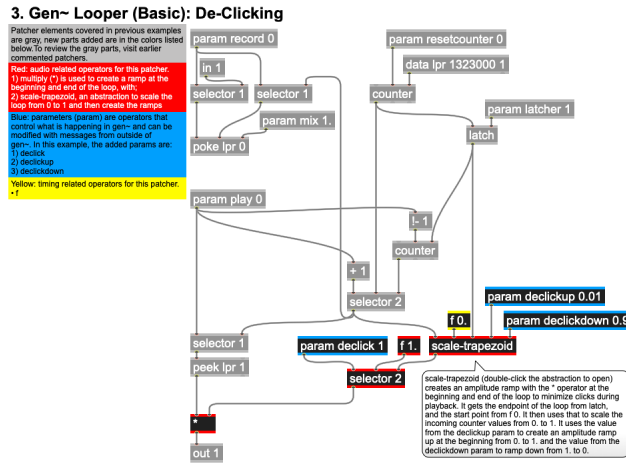
As you progress from step to step, only new elements will appear in the colors indicated above. Prior components will be either folded into a subpatcher or color-coded gray to help visualize what has changed, making it easier to “step backward” if you wish to trace the construction process.

1. Gen~ Looper (Basic): Record, Play, Stop, Reset



2. Gen~ Looper (Basic): Overdubbing





By now, we hope you have a sense of our aims and the ways we have tried to make them visible to you as both a reader and a practitioner. Although you could simply download the patches and explore them on your own, we have invested considerable effort in creating a book that guides you step-by-step through the construction of Max patchers, highlighting how experienced programmers think, make decisions, and adjust their strategies. Our hope is that this process will help you become the next iteration of this practice: not only a user of patchers, but a creator of them. In doing so, you contribute to a growing community of artists, technologists, and educators who build their own tools.

Thank you for engaging with this work, whatever your intentions for its use may be. We hope that what follows will help you generate, repeat, and organize variety in your own creative practice.

Let's get started…

Chapter 1:
Tape-Style Looper (Max)

This section introduces tape-style looping and is designed for those new to Max. However, it does assume a basic understanding of fundamental Max concepts: you should be familiar with the basic concepts of what a patcher is, what objects are, and how to connect objects before moving forward. Each subsequent chapter will build upon this foundational knowledge.

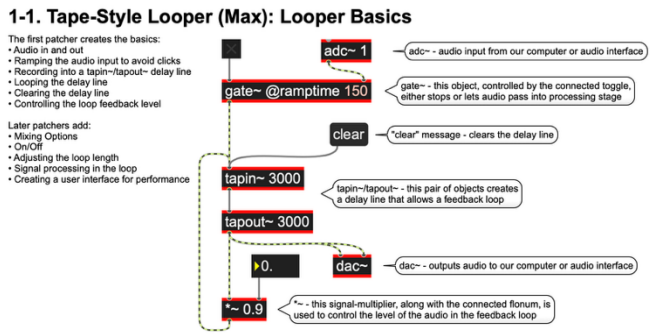


Figure 1-01

Our first looper is inspired by early tape-based loopers that loop audio elements in relation to the timing determined by the physical length of the tape loop. This type of looper is well-suited for experimental musicians, drone artists, and others who prioritize creative exploration over precise loop timing. It is not ideal for the more common approach to live-looping, which typically involves overdubbing beats, melodies, and bass lines in a metrically synchronized manner. In later chapters, we will build live loopers that allow the loop length to be set interactively during your performance, providing options for both synchronous and asynchronous

loops. The tape-style looper we are building here offers great creative possibilities, as demonstrated by artists such as Robert Fripp, Pauline Oliveros, and Brian Eno, among others, who have used tape-based systems.

For this project, we will use the **tapin~**/**tapout~** pair of objects, which allows us to create a feedback loop with a predetermined loop duration set in milliseconds.

Setting Up Audio Input and Output in Max

For this patcher, audio enters the computer through an audio interface or a built-in microphone, is processed (looped, etc.) within Max, and is finally output through the audio interface or built-in speakers/headphones.

The handling of audio input and output is managed by two primary objects: **adc~** (analog-to-digital converter) for audio input, and **dac~** (digital-to-analog converter) for audio output. The tilde (**~**) in the names of these Max objects signifies that they are part of the MSP category of objects in the Max environment, which deals with digital signals—specifically, audio in this context. Max also features a variety of specialized objects, each with its own distinct identifiers: Jitter (video objects prefixed with “jit”), Max for Live (M4L) objects starting with “live,” MC (multichannel objects beginning with “mc”), as well as gen, RNBO, and more.

To begin, create a new blank patcher by navigating to File and selecting “New Patcher.” Following this, we will begin adding the necessary objects.

Let’s start with an **adc~**. As with all things Max, there are multiple ways to do this. We highly recommend learning shortcuts from the beginning.

In the unlocked patcher (the lock/unlock icon is in the lower left corner), and with the patcher selected (just click on it), press the letter “n.” This

creates an empty object; type in '**adc~'** (it should autofill at some point) and press Return. You should now see this:



Figure 1-02

To view the audio settings for Max and your patcher, go to the "Options" menu and select "Audio Status…". You can also open "Audio Status" by double-clicking your **adc~** or **dac~** in a locked patcher, by sending them an "open" message, or by looking in Max preferences.

In the Audio Status window, you can view, select, and adjust the specifics of your computer and audio interface, as well as settings related to CPU performance and scheduling. For now, we will leave these settings at their default values. The image below shows my sampling rate set to 48,000 Hz, with both the I/O Vector Size and Signal Vector Size set to 32.

Generally, smaller vector sizes result in lower latency, however, at the expense of increased CPU usage. Larger vector sizes reduce CPU load while increasing latency. Max patching almost always involves balancing this tradeoff between responsiveness and computational cost, making these settings particularly important.

The Overdrive option prioritizes message handling and audio processing over screen drawing. Enabling it can improve timing for audio tasks, but may make the user interface feel less responsive. Similarly, enabling the Scheduler in Audio Interrupt can improve synchronization, with the same potential side effect of UI sluggishness.

If something feels off while using a patcher—whether increased latency or a laggy interface—these settings are often the first place to experiment. Depending on your machine and the complexity of the patch, minor adjustments here can make a significant difference.

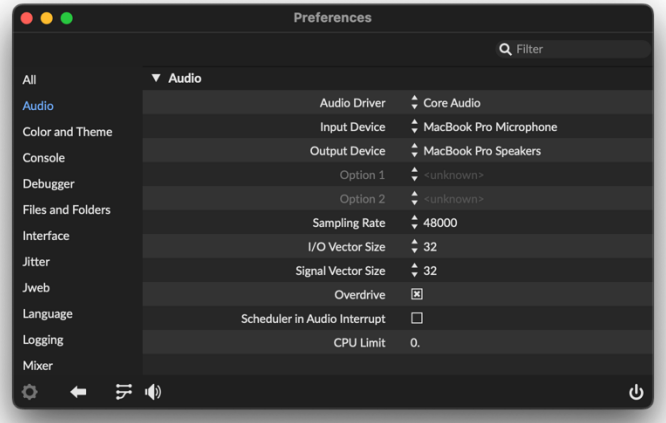


Figure 1-03

adc~ in its default state, as pictured below, is set to audio inputs 1 and (in your computer or interface), which enter your patcher through the outlets at the bottom of the **adc~** object. The top inlet is for messages (start, stop, on, off, open, et cetera).



Figure 1-04

You can assign specific inputs to the **adc~** by typing in a number or multiple numbers. As you can see below, there are now four outputs to the object that correspond to the input numbers entered.



Figure 1-05

Numbers need not be sequential; you can type in discontiguous numbers that will correspond to those matching inputs or outputs



Figure 1-06

You can also type in a single number. For example, on my personal audio interface, the microphone input is 13, and you can see there is now a single output from the object.



Figure 1-07

dac~ functions similarly regarding outputs. The object is set to outputs 1 and 2 by default, accessible from the inputs at the top of the object, which will be sent to your computer or audio interface.



Figure 1-08

As with **adc~** we can type in numbers (multiples or individual) that will correspond to your computer interface.



Figure 1-09

For this patcher, we will leave **adc~** set to input 1 and **dac~** in its default output settings of 1 and 2.

Audio input for the looper

Now that we have set up the input and outputs, let's start adding the looper section to our patcher.

In the input stage, we will first create a section that allows audio from the **adc~** to enter the looper. This will enable us to select the specific audio we want to loop. While we could have our patch continuously looping incoming audio, building this section gives you the option to be more selective about what you record into the loop.

For this patcher, we will use a Max **gate~** object that "gates" the audio, i.e., blocking the input or letting it pass. Other options for accomplishing this task that you can explore include **selector~**, **matrix~**, or a combination of **line** and ***~**.

In the example below, we have added a **gate~** between the **adc~** and **dac~**.

The audio input (**adc~**) is connected to the right inlet of the **gate~** object, and we have added a **toggle** to the left inlet. To quickly add a toggle, you can press the letter “t” in a blank space of the selected patcher. Alternatively, you can press “n” for a new object, then type “toggle,” or select the toggle option from the menu at the top of the patcher frame. It’s important to note that when typing the name, there is no “~” at the end. This absence of any additional identifier indicates that this is a Max object, which is message-based, whereas MSP objects (those with a “~”) are signal-based.

To connect the objects, click and drag from the corresponding output (located at the bottom of the object) to the appropriate input (found at the top of the object). You can also drag from an input to an output. It’s essential to remember that each inlet and outlet has a specific assigned task or function, so ensure that you connect the correct inputs and outputs carefully.

Each type of object, in this case, Max and MSP, will also have its own identifiable style of cable. The Max objects’ (messages) default cable is solid gray with an outline, and the MSP (signals) is the yellow-striped cable.

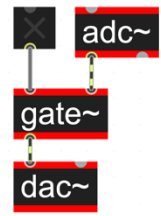


Figure 1-10

To use the patcher effectively, ensure that it is locked (indicated by the lock icon in the lower-left corner) and that the audio for the patcher is turned on (controlled by the on/off switch in the lower-right corner). When you press the **toggle**, it will highlight the “X,” indicating that the toggle is “on.” This action sends out a “1” from the **toggle**, which opens the **gate~** and allows audio to pass through. Conversely, when you deselect the “X”

on the **toggle**, it sends a “0,” closing the gate. Remember, the audio must be turned on for MSP objects to function properly.

When building the input section, it’s important to consider audio clicks. These clicks can happen for various reasons, but in this case, they often occur when audio is gated, causing an instantaneous switch from no signal to a signal or vice versa. Due to the laws of physics, speakers cannot move from one position to another instantaneously. As a result, the speaker's attempt to shift rapidly creates a clicking sound. One way to avoid this click is to create an amplitude ramp (fade in/out) up from 0. as you turn the gate on, and down to 0. as you turn off the gate. This ramp can be accomplished by adding the *ramptime* attribute to the **gate~**. This attribute can be adjusted to your preference, ranging from a short fade, such as 50 ms, to eliminate most clicks, to a longer fade that may better suit your creative goals. Here is the same configuration as above with the added **@ramptime** attribute. Ramptime can also be adjusted by sending a message. To add a message, use the shortcut “m” on an unlocked selected patcher, then click and drag to create a connection. While it is not necessary to use a message when setting it with the **@ramptime** attribute, sending a message allows for more creative control over ramptime while you are playing. You can incorporate a variable ramptime using the $1 sign in your messages, with a **number** object connected to the messages’ left input.

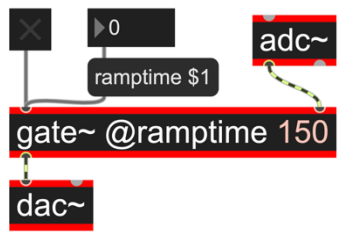


Figure 1-11

Creating the Looper with tapin~/tapout~

The Basics

(Example Patcher: 1-1-Basics)

One of the main characteristics of a tape loop is its fixed length. Similar to a tape loop system, the first step for us is to determine the length of the loop. In this section of the patcher, we will utilize the **tapin~/tapout~** pair to create a feedback loop that operates similarly to a tape loop on a tape player.

tapin~/tapout~



Figure 1-12

tapin~ takes as its input a signal and copies that signal into a delay line. The maximum available delay time (set in milliseconds) for this particular **tapin~** is 3000 (3 seconds). The **tapout~** object plays that signal back at its set delay time (3000, the maximum available, is used for this example), also set in milliseconds. The **tapout~** delay time can be any number equal to or less than the **tapin~** setting. This pair of objects is unique in the Max/MSP world in that they cannot function without each other, they must always exist in tandem. They are also unique MSP objects, in that the cable that connects them is not an audio cable, but a Max cable that exists to show the objects share a delay line. Remember: the delay used (set in **tapout~**) cannot exceed the delay available (set in **tapin~)**. You can add multiple delays to **tapout~** as long as each is equal to or less than the total delay available set by the **tapin~**. Each delay number will create its own output, as well as an input to adjust the delay value if desired.

The example below has three delays, each with its own output.



Figure 1-13

For this patcher, we will leave the **tapin~/tapout~** pair with a single delay line at 3000 ms to emulate a 3-second tape loop.

The next step is to add a feedback loop so that the recording continuously loops, allowing us to overdub any audio we input. One of the other unique features of the **tapin~/tapout~** pair is that you can create a feedback loop…and it won't crash the patcher. It can, however, create loud audio moments, so we will also need to add control for the amount of signal going into the feedback loop, as too much signal can lead to an audio explosion. Additionally, we may want the audio to fade out as it cycles through the feedback loop. For this, we will add a ***~** into the loop and have it default to multiplying the looping signal by ".9," a safe value to start with.

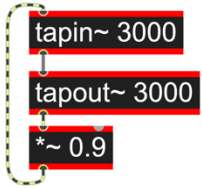


Figure 1-14

Adding Clear

One last thing, let's add the **clear** message to our **tapin~**, which, as the name suggests, will clear the delay line.

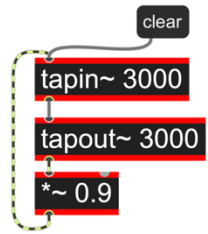


Figure 1-15

Now let's connect our **adc~** with our previously made **gate~** and **dac~,** and you have a basic three-second looper where you can let audio in with the **gate~**.

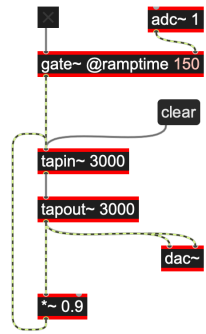


Figure 1-16

Adding Feedback and Amplitude Control

You can add a float number display/GUI by creating a new object and typing **flonum**, or you can simply press "f" in a blank spot of your unlocked patcher.

Select the **flonum** and in the Inspector (button with a circle and an "i" on the sidebar or select "Inspector" in the Object Menu), set the Minimum to

“0.” and the Maximum to “.9.” This will create a limit to the minimum/maximum level of the audio as it loops (feeds back) into itself, again, preventing audio explosions as well as allowing for creative fade outs of the audio as it loops through the system.

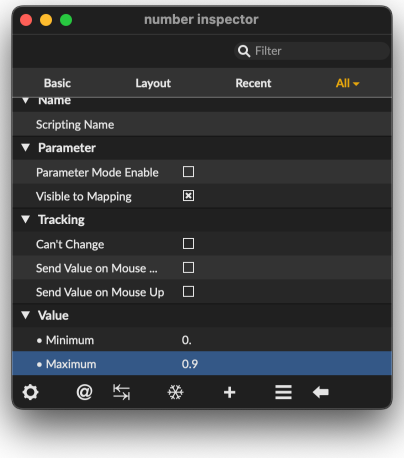


Figure 1-17

Connect the **flonum** to the right input of the ***~.** This will allow you to adjust the feedback from 0. to .9.

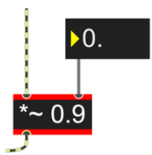


Figure 1-18

The patcher should now look something like this:

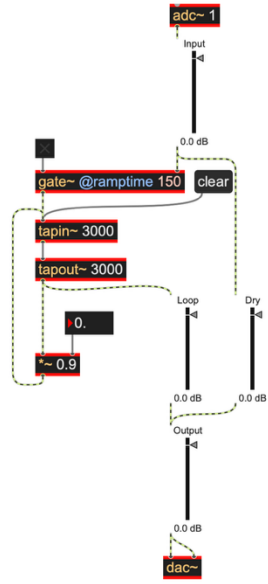


Figure 1-19

Mixing Options for Live Looping

The Basics

(Example Patcher: 1-2-Mixing)

An important part of live looping is the ability to mix the level of your voice/instrument in relation to the level of your loop. Let's add this capability to our patcher.

Add a mono **live.gain~** named Input. Throughout this text and in our private work, we prefer **live.gain~** because it includes metering and naming in the object, and many of these patchers may end up in Max for Live. To name them, select the object, open the Inspector, and find the name parameters.

- Add a mono **live.gain~** named Loop for the loop level

- Add a mono **live.gain~** named Dry for the dry (unprocessed) level of the Input
- Add a mono **live.gain~** called Output to monitor levels at the **dac~**. If we plugged both the Loop and Dry directly into the **dac~,** we would not see their combined level (sum), which could be a hidden source of clipping (generally considered undesirable distortion). This is important: do not sum audio where the summed level is invisible. Here we sum the audio at a **live.gain~** allowing us to see if there is any clipping.

Note the mostly vertical nature of this patcher in its basic state. This allows us to clearly see the flow of outlets to inlets in the patcher.

We could build the patcher with objects side by side, but as the patcher grows in complexity, the cable routing becomes increasingly challenging to follow.

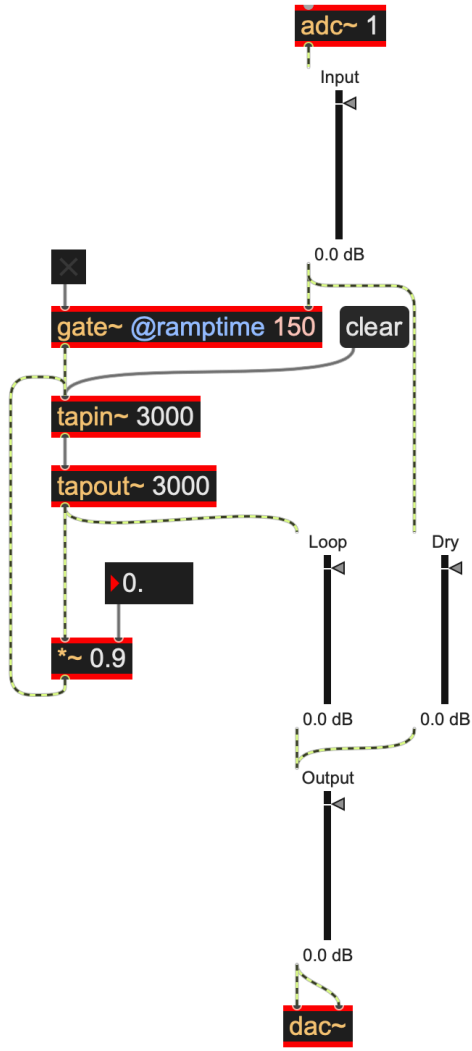


Figure 1-20

On/Off Control with the Spacebar

Before moving on to processing within the loop, let's add one more feature. Many audio programs use the spacebar to turn the software on and off. Let's build a spacebar on/off.

Add a toggle to the **dac~** to turn audio on and off. Click on the patcher where you would like the toggle, press "t," and connect the toggle to the **dac~.**.

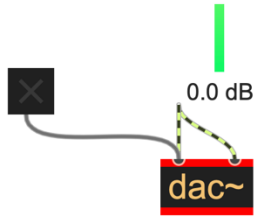


Figure 1-21

including, among other things, the ASCII key number of whatever you are typing. If you open the help file for the **key** object, you can see all the features it offers.

Knowing that the ASCII code for the spacebar is 32 (as discovered in the help file), we will add a new object, **select** (**sel** is the abbreviated name of the same object), with the argument 32. So, when the **select** object receives the message "32" from **key, select** outputs a **bang**.

Now whenever I press the spacebar, **key** outputs a “32” that **select** sees, that 32 then outputs a **bang** that will turn on or off the **toggle** (the opposite of what it is currently set), and turn on/off the audio via the **dac~**..

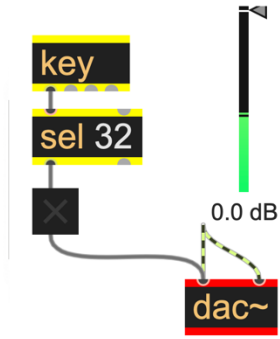


Figure 1-22

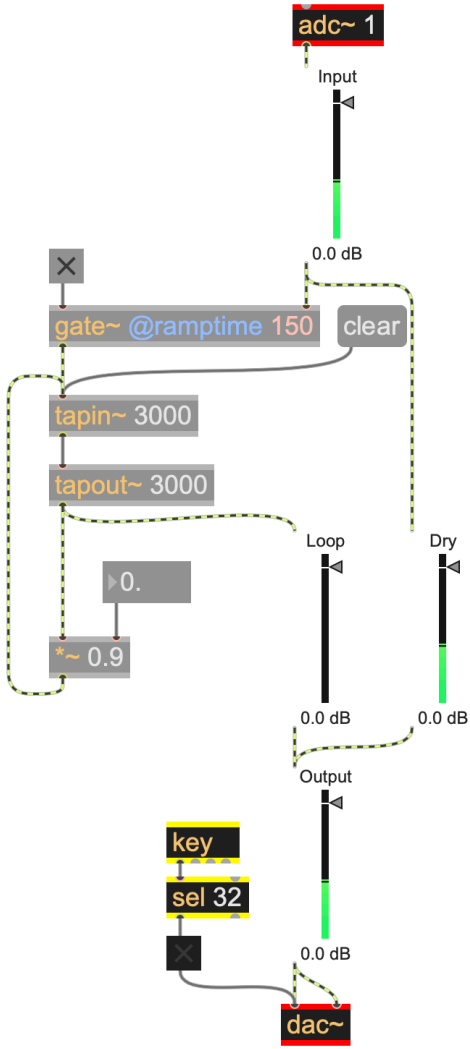

adc~ 1
Input
0.0 dB
gate~ @ramptime 150
clear
tapin~ 3000
tapout~ 3000
Loop
Dry
0.
*~ 0.9
0.0 dB
0.0 dB
Output
key
sel 32
0.0 dB
dac~

Figure 1-23

Varying the Length of the Loop

(Example Patcher: 1-3-Length)

To vary the length of the loop, we have added a live.dial named Length using the parameter settings below: Float, range of 1. to 60000., initial 3000, and Unit Style of “Time (ms)”

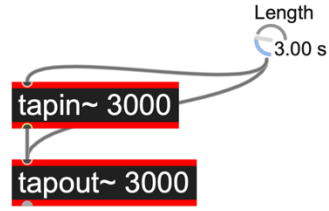


Figure 1-24

To vary the length of the loop, we have added a live.dial named Length using the parameter settings below: Float, range of 1. to 60000., initial 3000, and Unit Style of “Time (ms)”

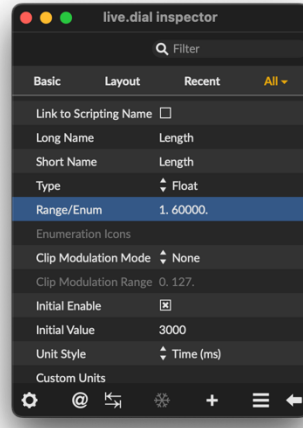


Figure 1-25

Be aware that changing the time of **tapin~** clears the delay line. This can make a noise that may or may not be considered undesirable. So it is best to change the length/time when the delay line is empty.

Adding Processing in the Loop

(Example Patcher: 1-4-Processing)

One of the joys of working with feedback loops is the ability to create variations of the audio as it passes through the loop each iteration. For the following examples, we will add several options for signal processing within the loop:

- We will add a switch, in this case **gate~,** to route audio straight to the loop with or without processing, commonly called a “bypass.”
- We will then create three options for processing in the loop, each with a bypass:
    a. Add the ability to transpose the output to change its pitch in the loop
    b. Add a lowpass filter to the processing chain in the **loop**
    c. Add a little judicious distortion/degradation to our processing chain in the loop
- We will add a switch, in this case gate~, to route audio straight to the loop with or without processing, commonly called a “bypass.”
- We will then create three options for processing in the loop, each with a bypass:
    1. Add the ability to transpose the output to change its pitch in the loop
    2. Add a lowpass filter to the processing chain in the loop

3. Add a little judicious distortion/degradation to our processing chain in the loop.

Building a Bypass

Adding the ability to route the audio signal into the processing stage or bypass it is the next step in building our looper. We can use the **gate~** object we have used earlier to do this. When we used it in the input/record stage, it either enabled or disabled audio input. However, now we want to have two outputs for our **gate**—one outlet to bypass it into the loop, and one outlet to route it to the processing stage. To do so, we add the **gate~** object and give it an initial argument of "2" to create two outlets. We also add a **@ramptime 50** attribute to minimize clicks, as we did before.



Figure 1-26

We already know that a toggle object sends either a 0 or 1 as its output. The **gate~ 2 @ramptime 50** object we have added now has three possible states:

- 0: no output at all
- 1: route output to the first output of the **gate~** object
- 2: route output to the second output of the **gate~** object.

For our purposes, we never want to send a value of 0 to the **gate~** object for this patcher (that would turn everything off); we need to send a value of 1 or 2. We can easily do this by adding 1 to the output value of the **toggle** object with the **+** object.

This makes its output of 0 or 1 become 1 or 2 with the addition.

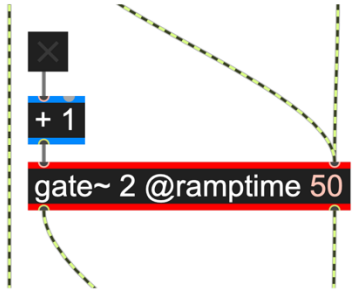


Figure 1-27

Since **gate~** defaults to 0, we need to ensure the **toggle** outputs 0 initially. This allows a 1 to be added, which then opens the first output. This initial output on opening is set in the toggle's inspector. Select the **toggle** in an unlocked patcher. Ensure the inspector window is open and displays "All" at the top. Then, select "Parameter Mode Enable."

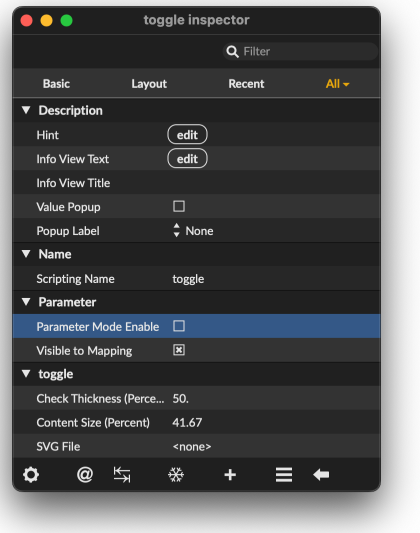


Figure 1-28

This will display a list of options. In these options, select “Initial Enable” and set the “Initial Value” to “0.”

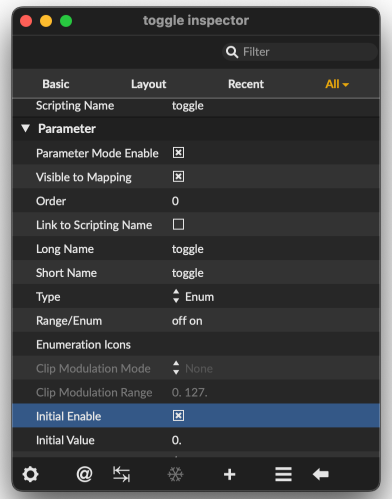


Figure 1-29

When the patcher is loaded, the **toggle**, in combination with the **+ 1**, will automatically set the **gate~** to the first outlet.

Pitch Change

Let’s create a named bypass for pitch processing, replacing the above toggle with a named **live.text** object.

As we continue to patch our way through this book, we will be using a **live.text** object where you might expect to see a **toggle** object. There are two reasons for doing this:

1. A **live.text** object lets us create buttons and toggles with clear labels referring to the function being controlled.

2. Many of the patches we are creating may ultimately become Max for Live Plugins, and the Live object set is designed to be used in this environment.

Add a **live.text** object. Use the **live.text** object's Inspector, locate the Value section of the Inspector, and type in the text we want to appear (since our text includes spaces, we will need to put what we type in in quotes). For the first example, we use "Pitch Off" and "Pitch On." We now have (compared to a highlighted "x") a clear label indicating the function being selected.

As you did with the basic toggle, select Initial Enable in the object's Inspector and set the Initial Value to 0 (you do not need to enable Parameter Mode in Live objects).

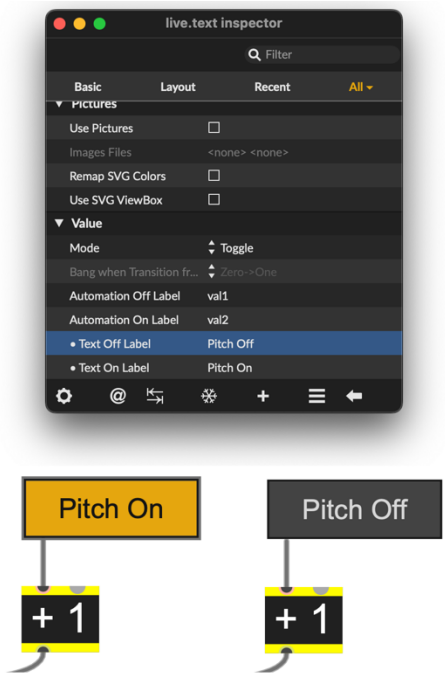


Figure 1-30

We will now introduce pitch transposition to our looper, allowing the pitch to change by a selected amount each time the audio passes through the loop. The Max **pitchshift~** object offers a straightforward method to implement this feature..

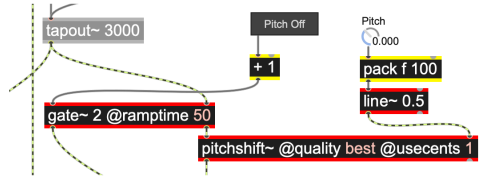


Figure 1-31

Create a new **pitchshift~** object and add the **@quality best** and **@usecents 1** attributes. Setting the *@usecents* attribute to 1 configures the **pitchshift~** object's second inlet to accept a pitch change number value and convert that to cents (cents is a unit of measure for the pitch transposition— each half-step interval is equal to 100 cents). We will use an incoming signal value along with a **line~** object to smooth the transition between settings as you select the value with the dial. Notice we used a **live.dial** object here, replacing the regular dial object for the same reason we did **toggle** with a **live.text** object: labeling of the function controlled and ease-of-use with Max for Live.

In the Parameter section of the **live.dial** object's Inspector, give the **live.dial** object a label "Pitch," set the range to -2400 and 2400 (which corresponds to a pitch change of two octaves up or down), and set the Initial Enable with an Initial Value of 0. We also changed the Unit Style to Custom and set the Custom Units to "%.3f" (a sprintf-style character string that instructs the dial to display three decimal places of accuracy).

We chose cents for pitch values—rather than semitones or another option—to support the microtonal pitch variation that decimal places allow us to create. (For more about sprintf, check out the **sprintf** object help file or Max Reference page.)

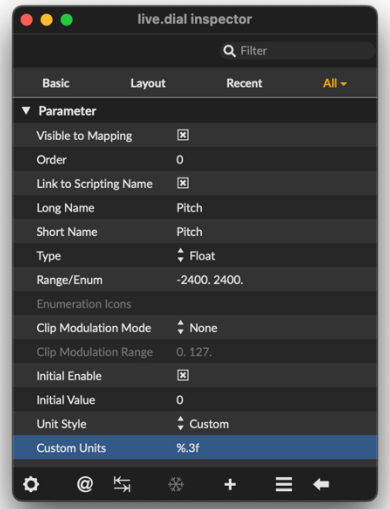


Figure 1-32

Lowpass Filter

Next, let's add a lowpass filter to the loop processing. For this example, we will use a **onepole~** object, which is a simple and efficient one-pole lowpass filter.

Once again, we will use a **gate~** object to set up our signal routing bypass. Notice that the first output from the previous **gate~** object we added earlier is connected to both the input of the new gate~ and the output of the **pitchshift~**. Since the signal in the first **gate~** is only going to one of the outputs (bypassed or pitch-shifted), only one output will be passed to this new **gate~**.

As before, we will add a combination of **live.text** object and **+ 1** to our gate and name it “LP Off” and “LP On” using the **live.text** object’s Inspector, with an initial value for the enable set to 0:

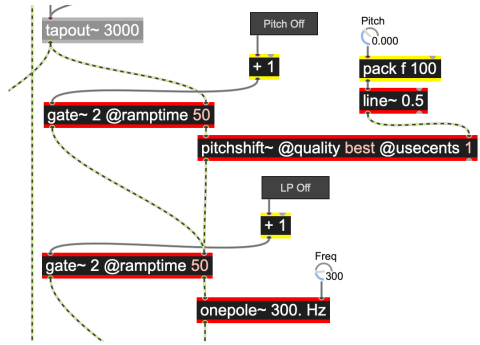


Figure 1-33

For **onepole~**, we add an initial argument of 300 Hz but make that adjustable with a **live.dial** named “LP Freq.”

Adding distortion

For our final bit of added signal processing, we will use the MSP **tanh~** object to add some nicely rounded “soft clipping” distortion to the loop. This type of distortion that musicians are accustomed to hearing and using is proportional to the input gain of the audio signal; the more the input gain is increased, the greater the distortion. The mathematical tanh() function neatly models the type of distortion associated with the physics of vacuum tubes—when you increase the input signal level to the **tanh~** object, it is as if an analog signal is being pushed towards its maximum value. The more it approaches the maximum, the more it distorts and acts like saturation (which is frequently described as sounding a lot warmer and more natural than digital clipping).

As we have done before, we are using a **gate 2 @ramptime 50** object for signal routing and a **live.text** object/**+1** pair as a UI and switch (setting Distortion On/Off for the name using the Inspector, and initial

enable of 0.). We've also added a labeled **live.dial** object and a ***~** signal multiplier object that "goes to 11" thanks to range setting using the Inspector) to control the amount of distortion/clipping we add to the signal.

Here is what the final result of the patching looks like:

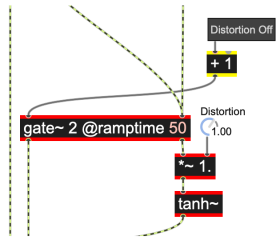


Figure 1-34

More Effects

As always with Max, you can add processing effects both in the loop and to the audio that bypasses the loop, creating a patcher as complex as you might desire.

Creating a Performance UI in Max

(Example Patcher: 1-5-Performance-UI)

Now that we have a functioning looper, it would be beneficial to modify our patcher so that only the parts we are controlling are clearly displayed, making them easier to use during performances. Additionally, we can view this as a way to design a user interface for a Max for Live plug-in in the upcoming chapters.

So far, we have arranged Max objects in our patcher, as we program, in a way governed by the flow of data and audio—rather than in a way that gathers all of our UI objects and arranges them in an orderly and meaningfully visible fashion governed by the demands of a live

performance. To create that order, we will use Max's Presentation Mode, which lets you arrange and resize user interface objects in your patcher independently of their functional position and size in Patching Mode. The Presentation Mode is a separate view from the Patching Mode we've been working in until now. Each mode has its advantages, and you can easily toggle back and forth between patching and presentation modes while you work, as well as lock/unlock the patcher window in each mode to switch between editing and running the patch. Presentation Mode, as the name suggests, allows you to create ("present") only what is needed to perform with, and you can also set your Max patcher to open in Presentation mode automatically when your patch is finished and ready to go.

Here is what Presentation Mode will look like when we're done patching:



Figure 1-35

We will start with our Max patcher, and begin by selecting only the objects in our patcher that we will want to see in our user interface part (Presentation Mode) of the patcher, and "adding them" to the Presentation Mode. To do this: while in patching mode, select any object you wish to add and select "**Add to Presentation…**" from the Max **Object** menu. When you do this, the selected object(s) will be surrounded by a salmon-colored border in the patcher window.

Here’s what the final patcher should look like in patching mode:

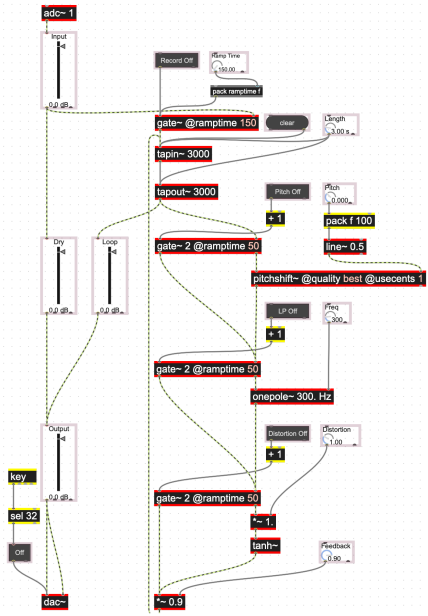


Figure 1-36

To view Presentation Mode, click on the Presentation/Patching Mode icon highlighted in the image below to toggle between Patching and Presentation.



Figure 1-37

You will now only see the objects you have added to Presentation Mode. It might be a mess at first. Arrange these objects to create your user interface. Moving things around doesn't change their location in patching mode, allowing you the tion.

Patcher Windows (double rectangles, icon next to the Presentation/Patching Mode icon) to create multiple views of the patcher you are working on, including simultaneous Presentation/Patching views, as shown below.

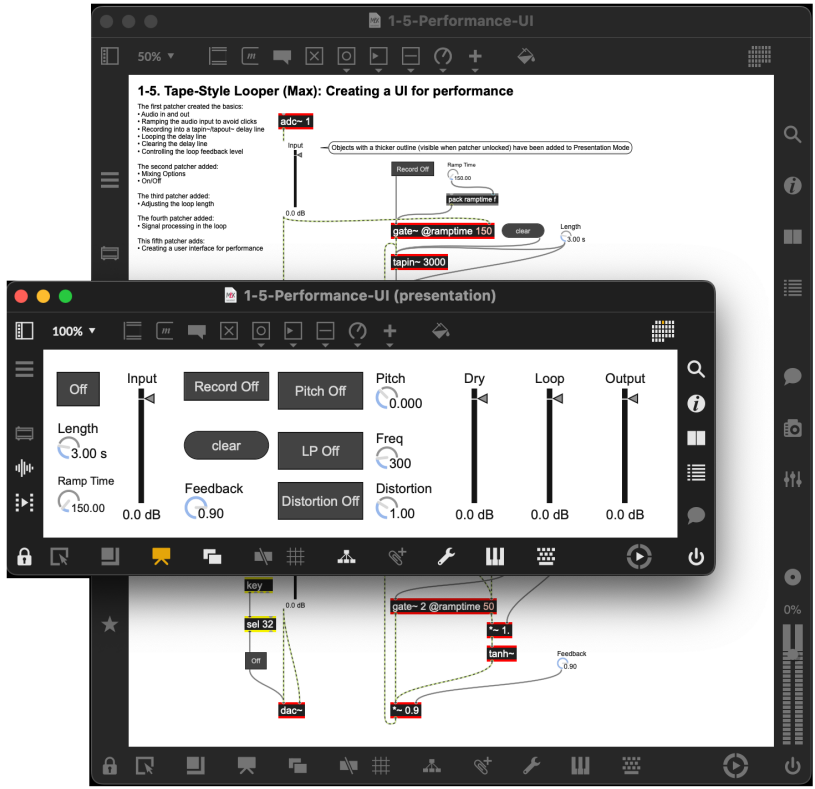


Figure 1-38

To set your Max patcher to open in Presentation mode, use the Patcher Inspector by selecting Inspector Window from the Max View menu (be sure the patcher itself is selectedtion.

now ready to loop in Presentation mode as soon as you open it.

Conclusion

For this chapter, we have developed a tape-style looper based on the concept of a predetermined, fixed tape-loop length. We also explored methods for processing the audio after it has been recorded in the loop. These processes used within the loop—gates with **gate~**, pitch with **pitchshift~**, lowpass with **onepole~**, and distortion with **tanh~—**can be repurposed in future loopers, or for building ways to affect sound outside of the loop as well. Nearly everything we've developed can be transferred to other types of projects. One key aspect of learning Max is recognizing that everything can be repurposed for other applications.

Chapter 2:
Multitrack Looper (Max)

Introduction

In this chapter, we will create a multitrack looper in Max with varispeed playback control and the ability to run multiple loops synchronously or asynchronously. In doing so, our multitrack looper will allow you to add as many or as few tracks/loopers as you desire and also enable the ability to process each track/looper individually. As we patch, we'll explore the use of Max abstractions as a way to create individual tracks/loopers. You'll see this technique used throughout the rest of this book.

Reset, Record, Play, Stop

(Example Patcher: 2-1—Reset-Record-Play-Stop)

Here are the basic elements we'll need to create for this looper:

- The ability to record and store our loop
- The ability to play a stored loop back
- The ability to stop recording, set the end of the loop, and enable playback
- The ability to reset our looper.

To achieve this, we'll use three MSP objects that will be the core of this patcher: a **buffer~** object (to store what we record), a **record~** object (to record input), and a **groove~** object (to play the audio recorded).

Since recording, storage, and playback will all be working with the same audio material, we will use a named **buffer~** object. Rather than being connected together with patch cords, the buffer object works by means of a shared name. A **buffer~** is given a unique name (it must be unique), and both **record~** and **groove~** objects use that same unique name to access the **buffer~** object. **Record~** records the audio into the **buffer~,** and the **groove~** object plays back the recorded audio from the **buffer~**.

But *how much audio* do we want to record? In addition to being assigned a name, the **buffer~** object lets us set a maximum duration (in milliseconds) for our looper, along with the number of channels we'll be using. For this example, we will be using a value of 60000 (1 minute in milliseconds). This number is arbitrarily high - while we will not always record a 60-second loop, we'll "set aside" that much memory and use a timer to set a loop length equivalent to the length of the material we record. Our example patch will be recording in mono, so we set the number of channels to 1. By default, the **record~**, **buffer~**, and **groove~** objects are set to 1 channel, but we'll use arguments to explicitly set the channel count to provide us with a visual reminder. In the following example, you'll notice that the **buffer~** object uses arguments to set its name (**lpr**), its length (**60000** milliseconds), and the number of channels (**1**). The following diagram illustrates how we used a shared name, along with the maximum length and number of channels set for the **buffer~** object using arguments. You'll see this used in patches throughout the rest of this book.



Figure 2-01

Now that we have our core objects set up, let's consider the basic tasks we need to perform to control the reset, record, play, and stop operations, as well as the order in which these tasks occur.

We'll start with how we'll record audio input. That task is simple: we need to provide an input source for our audio, and enable/disable recording of that input. We're going to use an **adc~** object as our live input source (just as we did in the previous chapter) and connect it to the **record~** object. Turning recording on and off can be done in several ways, but we'll keep it simple this time and add a **toggle** object that sends a 1 or 0 to enable or disable recording.

If you'd like to test your patch, you can use the setup shown in the following figure. Add the objects, connect them as shown, lock your patcher, turn on the patcher with the **toggle** attached to the **adc~**, and record something short by using the **toggle** attached to **record~**. When you've stopped recording, double-click the **buffer~** in your patcher. You'll see what was recorded in a pop-up window that shares the name of the **buffer~**.

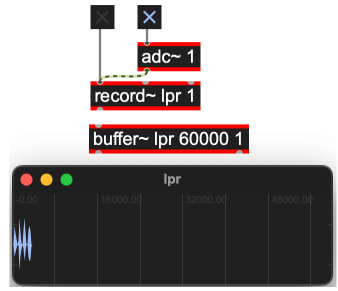


Figure 2-02

Setting up our patch to do playback is a bit more complicated. We'll need to implement a timer that allows us to set the endpoint for our recorded loop. For these examples, we're assuming that we'll start recording at the beginning of the buffer (0). We'll do this by creating a timer that we'll start running when we begin recording. When we stop recording, we'll stop the timer and use the value of the elapsed time of the recording (in milliseconds) as the endpoint for our loop in **groove~**. That way, we only playback and loop the part of the **buffer~** that we recorded into. Since we're using the timer output, we don't need to worry about any additional

space in our buffer. This additional space allows us to create loops of varying durations up to the maximum of the **buffer~**..

The Max **timer** object is just what we need for this task. Sending a first *bang* message to the object's right inlet starts the timer, and sending a second *bang* message to the object's left inlet outputs the value of the elapsed time between the two **bang** messages. This value then goes into the rightmost inlet of the **groove~** object to set the loop length.

To play our loop with **groove~** we need to do a few things:

We tell the **groove~** object what speed to play the audio back by providing a signal-rate value using a **sig~** to the left inlet of the **groove~** object. A value of 1.0 is normal playback speed. We can also change this to other positive floating-point values. Values greater than 1.0 speed up playback, values between 1.0 and 0 slow playback down, and negative values have the same effect, but the playback is in reverse.

Sending the message "*startloop*" to the right inlet of the **groove~** object starts the loop, and the message "*stop*" stops loop playback.

We will enable the playback to loop by adding the attribute **@loop 1** as an argument to the **groove~** object. By default, the **groove~** object has looping disabled (it defaults to **@loop 0**).

We're ready to start setting up the control portion of our patch. Let's review the four basic controls for our looper: reset, record, play, and stop.

To ***reset*** our looper requires the following sequence of actions:

1. Stop playback
2. Stop recording
3. Reset the timer. (We use a new object named onebang, more on this later.)
4. Clear the buffer~
5. Make sure the playback speed is set to 1.

Setting our looper to ***record*** requires the following sequence of actions:

1. Stop playback
2. Clear the **buffer~**
3. Start the **timer**
4. Start the recording

Playing back our looper requires the following sequence of actions:

1. Stop the recording
2. Stop the **timer** and use that value to set the maximum value for the timer (the loop maximum)
3. Start the loop with “startloop”

Stopping the loop is simple:

1. Message “stop” to **groove~**

The actions listed for each mode of operation should happen rapidly, and in the sequence you’ll see above. To accomplish this, we’ll use the **trigger** object (whose name is often shortened to **t)**. The **t** object is a go-to object in wide use and one of our favorites for good reason—it takes a single *bang* message as input and outputs multiple messages, symbols, integers, floats, bangs, rapidly from right to left, one after the other.



Figure 2-03

In the above example, we have created a **t** (**trigger**) object with a series of messages and integers associated with the actions listed above. A single *bang* message will output these in order, right to left. For example,

here are the five things that must happen for **reset**. Compare them to the figure above:

1. A *stop* message to the **groove~** object stops playback
2. A *0* to the **record~** object to stop recording
3. A *bang* message resets the **timer**,
4. A *clear* message clears the contents of the **buffer~** object
5. A *1* message resets our **sig~** (playback speed) to normal playback

The arguments to the **trigger** object for recording are arranged in a similar fashion. For record, right to left order:

- A *stop* message goes to the **groove~** object stops playback
- A *clear* message clears the contents of the **buffer~**, object
- A *bang* message resets the timer
- A *1* sets the **sig~** to normal playback speed



Figure 2-04

The arguments to the trigger object for playback are arranged similarly for playback, in right-to-left order:

- A 0 message to the record~ object stops the recording
- A *bang* stops the timer and sends the elapsed time to set the loop time for the **groove~** object

- A startloop message to the **groove~** object starts playback



Figure 2-05

Stopping playback requires a single *stop* message to the **groove~** object.



Figure 2-06

If you look carefully, you'll notice that the patcher contains an object you may not have seen before, called **onebang**. There is a subtle bit of patching here that needs some explanation. It starts with the question: What if we want to be able to click play to restart after stopping, but *without resetting the timer*?

This is a perfect example of why the **onebang** object exists. If you look carefully, you'll notice that both the **trigger** objects that control play and reset send *bang* messages to the **timer** object. Our problem is that we want to avoid resetting the timer if we move between stop and play. Unless we take action, the timer will always reset when we restart our looper after pausing it.

The **onebang** object does something really subtle that solves our problem: it acts like a **gate** object in Max that works with *bang* messages. Sending a *bang* message to the first inlet will only be passed through the object after a second *bang* message is sent to the right

outlet. By adding the **onebang** object, resetting our playback will send the *bang* message that “opens” the **onebang** object, allowing the *bang* message to pass through again. However, until then, the play trigger device will still be able to pass through without resetting the timer.

As we build our user interface, we’ll use objects that you’ll be familiar with by now. We will use a **live.tab** object with four tabs to act as buttons (Reset, Record, Play, Stop). These are set using the **live.tab** object’s Inspector and use a **select (sel**) object to route specific bangs associated with each index number for the tabs (0 for Reset, 1 Record, 2 Play, 3 Stop) to each **trigger** object.

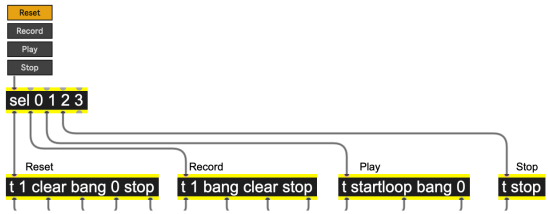


Figure 2-07

So far, we have worked to avoid patches that are difficult to read.

However, as our patches become more complex, we'll need to find ways to make our patching easier to read.

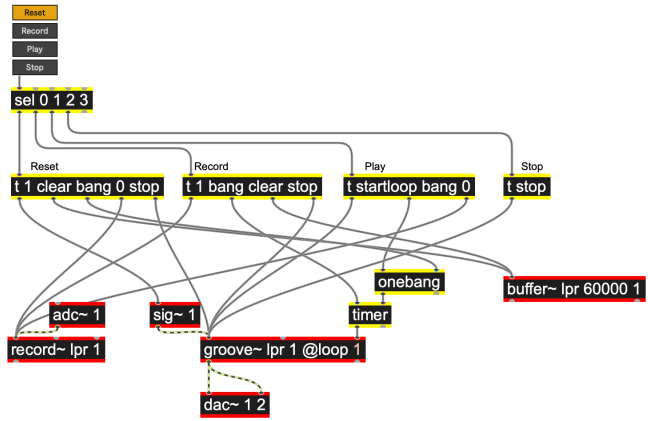


Figure 2-08

The messages, routed from each **trigger** to their intended destinations, can become visually chaotic. At this point, it's a good time to select all the patchcords and align them using Auto Align or Route Patchcords in the "Arrange" menu of Max. Either command will get it close, and then you can manually clean it up to suit your needs. You might also consider coloring the patch cords in their Inspector to create visual flow. At some point, we realize that any complexity in a visual coding environment might start to look…daunting. One method of making it easier to trace the working of a patch is by aligning the patch cords and objects. Later in this chapter, we will examine encapsulation and abstraction—features of the Max environment that will help streamline and refine your patching.

In the meantime, here's what our patch looks like:

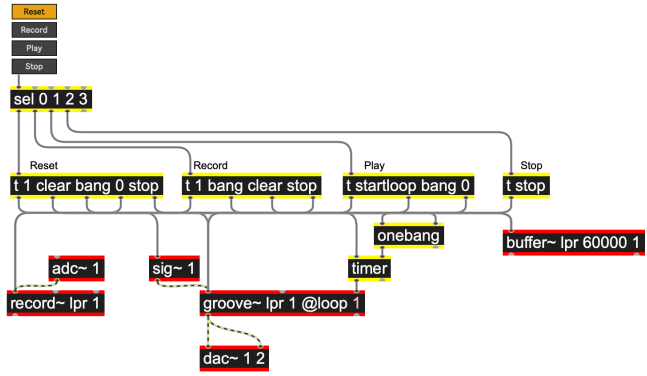


Figure 2-09

Speed/Direction Control

(Example Patcher: 2-2-Speed-Direction)

We're now ready to add two new features to our looper: control over the speed and direction of recording and playback. Controlling the playback speed is a relatively simple change to make. Since we're using a **sig~** object that outputs its value at signal rate. We can add a **live.dial** object attached to our **sig~** object that allows us to smoothly change the playback rate. As we said before, sending positive numbers greater than 1. to the **sig~** object speeds **groove~** playback up, and numbers between 1. and 0. slow it down. Negative numbers reverse the direction. -1 runs **groove~** at normal speeds but backwards, numbers less than -1. speed it up, and numbers between 0. and -1. slow it down. So we actually have control over both the speed *and* the direction of playback. We'll add a **live.dial** object and use its Inspector to change its name to "Speed/Direction," and set its initial value (under Initial Enable) to 1 and its range from -5. to 5.

We'll want to smooth out changes in our input to the **sig~** object, so we have also added a **line** object to handle the smoothing. (Remember to

add an argument of 0. to the **line** object, since we'll want the output to be a floating-point value rather than the **line** output's default integer output.) And finally, we'll add an argument of 1 to the **sig~** object so that the default playback state is forward at normal speed.

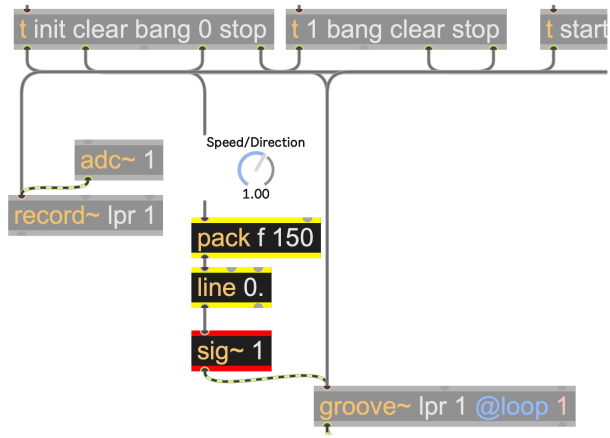


Figure 2-10

Declicking audio output

(Example Patcher: 2-3-Declicking)

One of the challenges to recording and manipulating audio loops involves handling clicks that occur at the beginnings and ends of loops. There are many ways to deal with this (that's often the case with Max). We're going to show you an easy way to do this by adding a quick volume ramp from 0. To 1. at the start of what we're playing back and add a corresponding ramp from 1. to 0. at the end. The **trapezoid~** object provides us with a way to do just that, and to do it in sync with the audio playback from the **groove~** object**.**

But how do we ensure that we can set a ramp based on the loop length the **groove~** object is working with? The **groove~** object is all set up to help us with that, using its right outlet. Whenever the **groove~** object

cycles through its loop, it outputs a signal-rate ramp that goes from 0. when the loop starts to 1. when the loop ends. In turn, the **trapezoid~** object is set up to create its output from exactly that kind of signal-rate input, which means that we don't need to worry about how long our loop is—the two objects work together to handle the heavy lifting.

So all we need to do is connect the right outlet of our **groove~** object to the inlet of the **trapezoid~** object, and take the resulting ramp from the **trapezoid~** object and use that to control the amplitude of the **groove~** object's left outlet using a ***~** object. And just in case you're wondering how steep the ramps output from the **trapezoid~** object are, we can set them by adding a pair of **live.dial** object (with names and with their initial values of .01 and .99 set using their objects' Inspectors) connected to **sig~** objects to allow us to adjust how steep the beginning and ending ramps are.

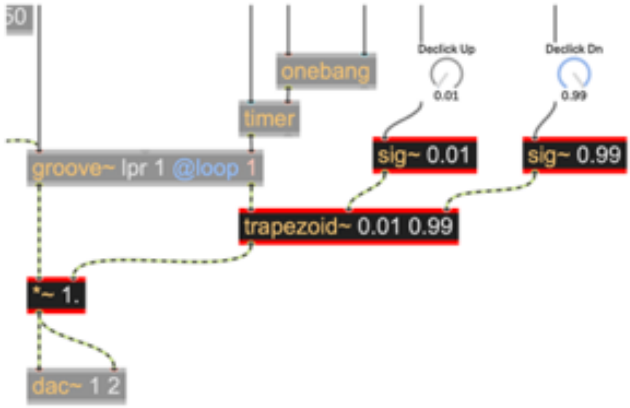


Figure 2-11

Reducing Visual Clutter: Encapsulation

Here's a look at our patch so far with the new features added:

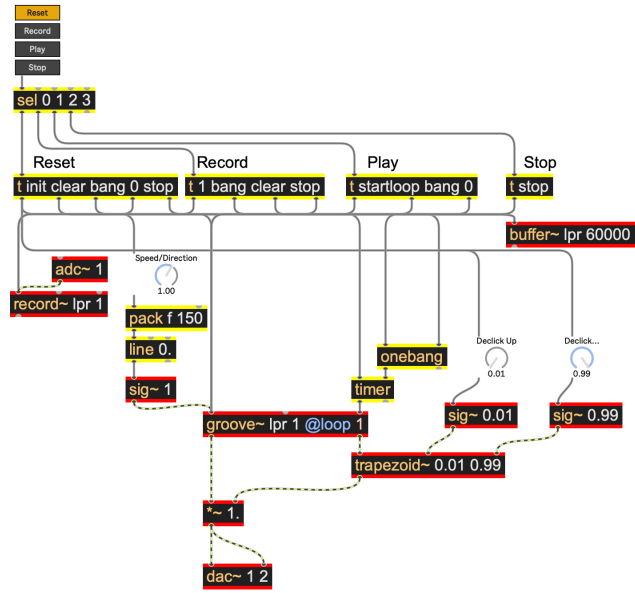


Figure 2-12

Since the subject of this chapter is *multitrack* loopers, the patch you see will become one track of our looper. You can imagine how complex our patch will look if we multiply this by *n* number of tracks. How can we streamline our patch as we duplicate it and still make it easy to work with?

One way to clean up a patcher is to use Max's "encapsulate" function (you can find the function under the "Edit" menu in Max), which takes the objects you select and automatically creates a Max subpatcher that contains what you've selected. The created subpatcher automatically adds **inlet** objects where needed and **outlet** objects as well. Here's how to automatically encapsulate a portion of our patch, including the creation of these **inlets** and **outlets**.

To prepare our patch for encapsulation, take a look at the UI objects that we'll want to access outside of our new subpatcher. In this case, UI elements, as well as the **adc~** and **dac~** audio input and outputs. Visually group them together. In the following example, below you can see that we moved objects that will send information into the subpatcher to the top of our patch, and those that will receive information from the subpatcher to the bottom. The **live.dial** objects both send and receive data, so we'll put them above, too.

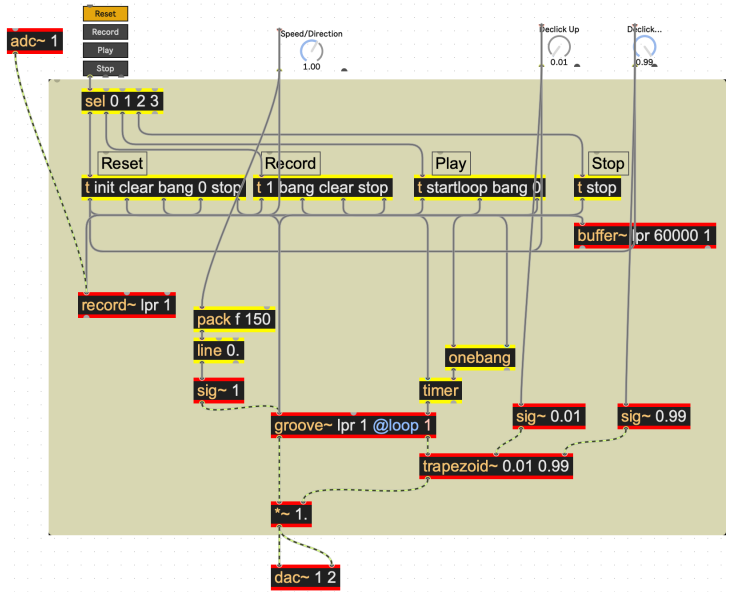


Figure 2-13

The resulting patcher elements in the middle (shown in the yellow/grey area) are what we want to include in our subpatcher. Click and drag to select those objects, and then select "Encapsulate" from Max's "Edit" menu. You should now see something like this:

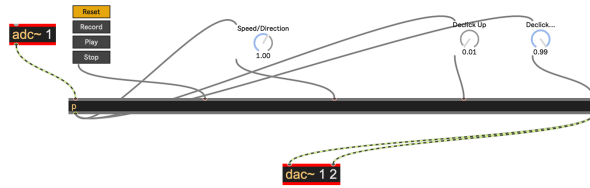


Figure 2-14

In your locked patcher, double-click the newly created **p** subpatcher to see its contents:

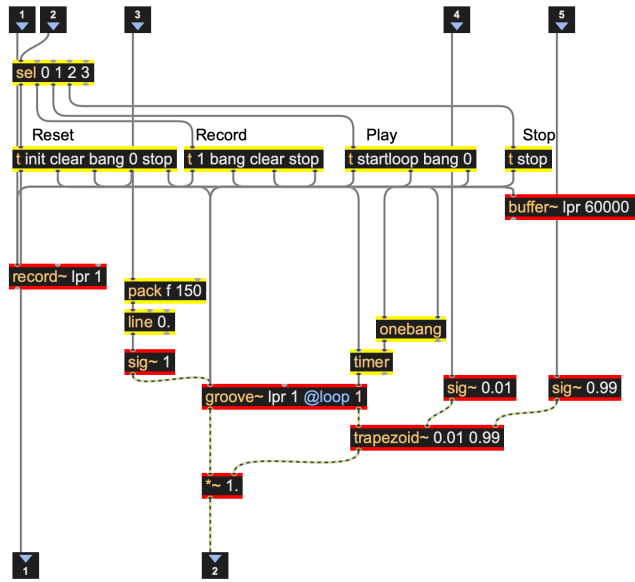


Figure 2-15

You'll see numbered square objects at the top and the bottom of the subpatcher. Encapsulation automatically adds these **inlet** and **outlet** objects, numbered automatically from left to right. You can rearrange the inlets and outlets for greater clarity (for example, we want to have our audio input and output on the far left of our subpatcher)—they'll automatically renumber themselves, and the position of inlets and outlets will also change in the parent patch.

Here's the new order we're using for the subpatcher:

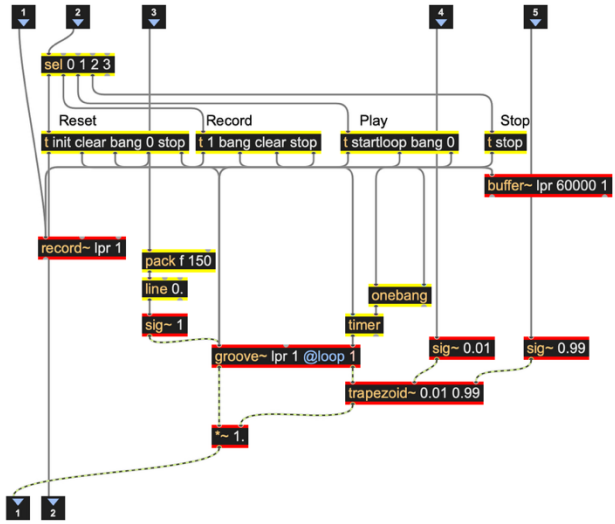


Figure 2-16

Here's the result of the rearrangement in the parent Max patch (we've resized the **p** patcher, too):

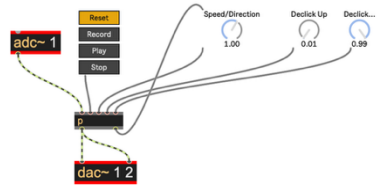


Figure 2-17

The result is smaller and easier to read.

From encapsulation to abstraction

(Example Patcher: 2-5-Abstraction)

You might think that creating multiple tracks for our multitracker looper simply requires copying and pasting as many instances of the patcher we just encapsulated as needed. But it's not quite that simple.

You'll recall that the trio of **buffer~**, **record~**, and **groove~** objects we created share a single *unique* name. If we copy our newly encapsulated patch, the trio of objects in the copy will still retain the same names as in the original, so they will not record/play different material. Fortunately, Max has a solution to this problem of assigning a unique shared name to each copy.

But first, we have a little preparatory work to do:

1. Save the main patcher with a unique name in a folder.
    1. Double-click to open the **p** subpatcher in your main patcher.
    2. With the sub-patcher opened and at the front of all other patchers (important)…
    3. Select "**Save As…**" in the File menu and save the **p** subpatcher in the same folder as your main patcher—give this new file a unique and memorable name. For this example, we'll use *lpr-abs* as an abbreviation for "looper abstraction."

After saving the **p** patcher with a new name, return to your main patcher. In the object box of the **p** subpatcher, rename it by entering the name of the subpatcher you just saved (lpr-abs). You can omit the "**p"** prefix at this **p**oint. When you hit return, Max will load the **lpr-abs** file you saved in place of the original **p** patcher.

Congratulations! You have successfully created a Max *abstraction*—an element of Max patching that behaves like an external object, except that

you can access and edit its contents by double-clicking the object's name.

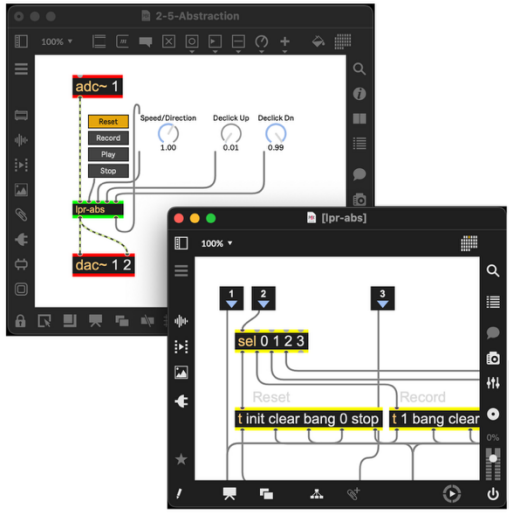


Figure 2-18

This is a very handy feature when creating a patcher that you'll reuse in multiple places within your patcher or in other patchers. When you edit the original saved abstraction, you'll change *all instances of it*, which saves a lot of time and energy. In fact, *any* of your patchers can be loaded into an empty object by typing in the name of the file, just as if it were a native Max object, as long as they are in the same folder as you're currently working in—hence the creation of a folder. There are other ways to handle Max's ability to find abstractions, including adding custom search paths and Projects, but for now, we'll use the folder method.

So, we now have a great piece of reusable patching, but we still need to address the fact that our abstraction's **record~**, **buffer~,** and **groove~** objects still reference the same buffer name (lpr).

Max abstractions have a special feature that allows us to generate unique names for our buffer and the objects that reference it. If you add

the 3-character string *#0_* (pound sign zero underscore with no spaces) to the beginning of the shared name objects inside your abstraction. Once we've done that, all instances of the *#0_* in the saved abstraction will have a unique numerical identifier generated for the character string that will be added every time that abstraction is loaded, with a different one for each abstraction. In the following example **groove~**,**buffer~**, and **record~** in the patcher have been renamed with *#0_lpr*.

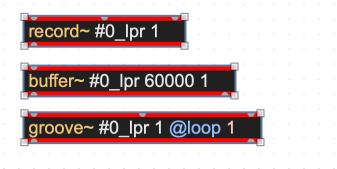


Figure 2-19

We can now load different copies of the abstraction, and when you open the abstractions, each copy will have its own unique shared name (the number that identifies the unique names is handled internally by Max and cannot be controlled). Please note that the underscore is not strictly required by Max, but we like it as it provides some visual space between the name and the unique identifier.

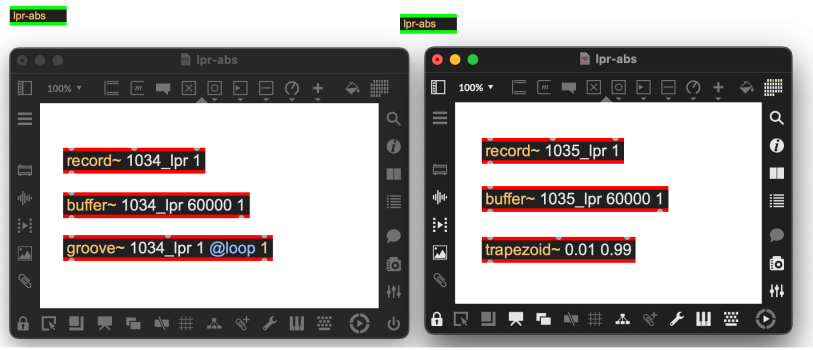


Figure 2-20

Here's what the abstraction patcher looks like with the added #0_ character strings to the **record~**, **buffer~,** and **groove~** objects:

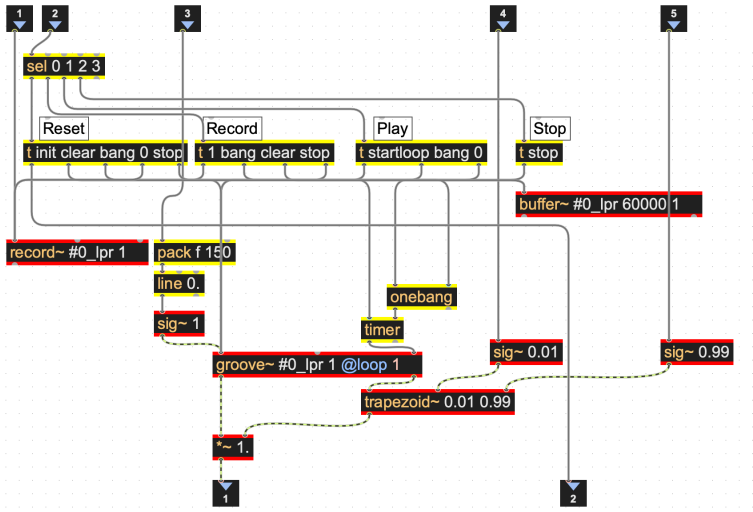


Figure 2-21

To track the functions of our abstraction's **inlets** and **outlets**, we can add a "Comment" to each of them using the Inspector. After we've done that, we will get a pop-up message that tells us what each inlet or outlet goes to, or comes from, like this:

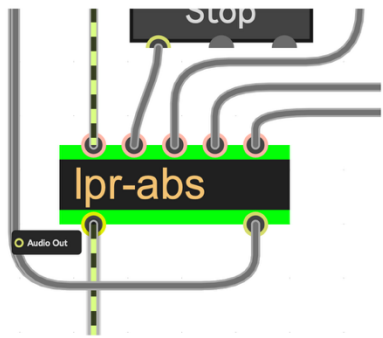


Figure 2-22

As we did in our previous chapter, add some **live.gain** UI objects, our spacebar on/off, and clean things up.

We now have a four-track, ***asynchronous*** looper perfect for experimental, ambient, and other musics.

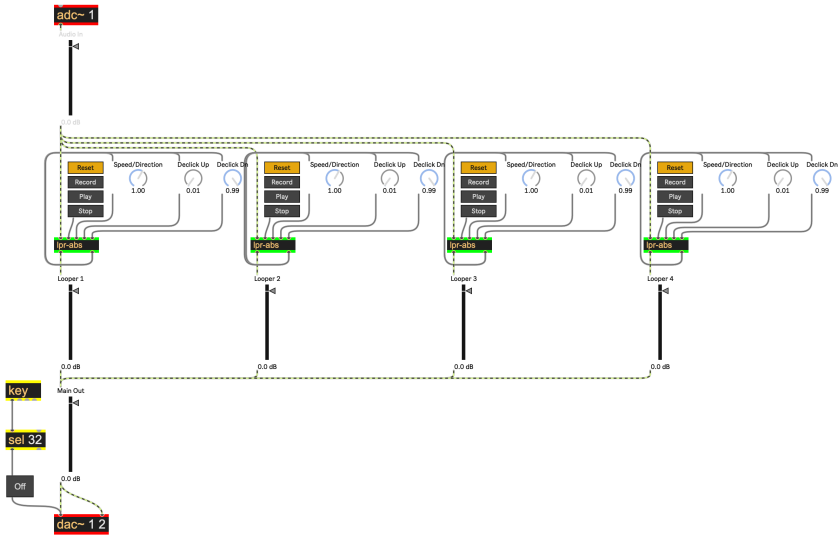


Figure 2-23

Synchronization

(Example Patcher: 2-7-Sync)

We now come to the trickiest part of our multitrack looper—synchronization. While you might think that syncing recording and playback of our multiple loopers sounds like it should be easy, it's a challenging problem, and one that we'll return to in different forms in our later chapters, where we'll be looking at the **gen~** and **rnbo~** objects' patching abilities. We will present one possible solution in the vanilla Max programming environment in this patcher.

For synchronization (the alignment of the beginning and end of loops when recording and subsequent playback), we will need a shared source

of timing. For this example, we will start by creating the primary looper abstraction, which provides the timing data, and then modify the secondary looper abstraction as needed to synchronize with the primary looper.

In this chapter's patcher, we've used the **groove~** object, which outputs a sync signal for synchronization with several objects. We used this feature earlier when adding a declicking control. For our secondary loopers, we can substitute the MSP **wave~** object for playback instead of **groove~.** We then use the "Loop Max" from the timer and sync output from our **groove~** object in the primary looper to sync all the **wave~**'s in our secondary loopers. (At the time of this writing, there are no Max objects for recording audio that can be controlled directly by this signal from the **groove~** object.) There is an object, **poke~**, that can be controlled by receiving an index (location by sample) to record, and **peek~** can use a counter, but both of these options would require some translation of the sync output of **groove~**, which is phase (0.-1.). Such a translation would introduce other challenges that might be a bit of a rabbit hole at this stage of our development.

The approach we've taken in this patch is to automate the start of the recording at the very beginning of the loop's playback in our primary **groove~** (0. from the sync output). We then automatically end the recording when the loop wraps back around to 0. This is relatively simple to achieve, with the limitation that the action of recording will only be in sync if recording happens *at normal speed*. The reason for this is that the **record~** object has no input control that takes into account any variation in speed or direction; it always records at normal speed. After that, playback will remain in sync with any changes to playback speed or direction from our primary looper. (You can still record at other speeds and directions, but the results will be unpredictable, which you might—or might not—find delightful.)

To create our primary looper, we need to set up one version of our looper abstraction that sends information to sync our secondary looper abstractions. We'll send the sync output of the **groove~** and the endpoint

(Loop Max) value we get from our **timer** object from our primary to our secondaries. We're going to add two additional outlets to the original patcher abstraction and save the result as our primary looper. It's a simple addition: we open up the **lpr-abs** abstraction and add two **outlet** objects in between the preexisting **outlet** objects. To the **outlet** number 2, we connect the "Loop Sync Out" from the **groove~** object, and to **outlet** number 3 we connect the output of the **timer** object**.** I named **outlet** 2 "Sync Out" in the Comment in its inspector, and **outlet** 3 "Loop Max." Here's what our primary looper looks like now:

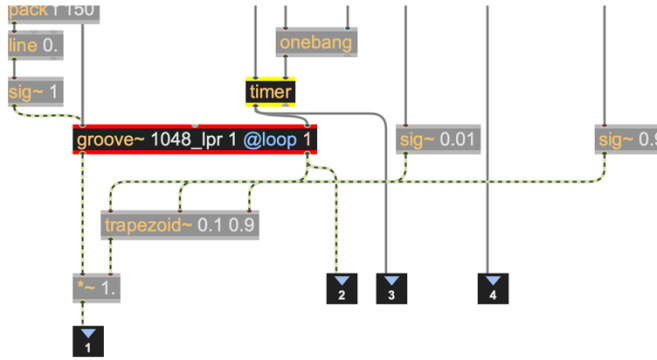


Figure 2-24

We now save this modified abstraction with a new name (**lpr-abs-sync-out).** This abstraction will serve as our primary looper, which we will refer to as "Looper 1" in the patcher. The other loopers will be secondary and identified by their respective numbers.

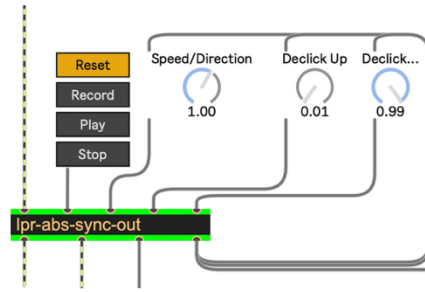


Figure 2-25

Secondary loopers

Since our secondary loopers will receive sync and loop endpoint information from our primary looper, they will have a reduced set of functions in their abstraction. We won't need a **timer** output or speed/direction in the secondary loopers, as they will receive that information from our primary looper. Our patching will need to focus on different ways of recording and playing.

The playback part is straightforward. In place of the **groove~** object, we can use the Max **wave~** object to access our **buffer~**. The **wave~** object plays back audio stored in the buffer~ object by means of a floating-point value at signal rate between 0. And 1. That means we can connect the sync outlet of our g**roove~** object in the primary looper to indicate which part of the **buffer~** to play back in the secondary looper. We will also use the "Loop Max" from the **timer** output from our primary looper to set the end of the loop in any and all secondary loopers using the third inlet to **wave~** called "Ending Table…". When it receives this signal from our primary looper, it will play at the speed and direction specified by the shared sync signal. When there is no signal, the **wave~** object will not play.

Things are a bit more complicated when it comes to recording. The solution we're going to use can be found under the tab called "sync" in the **groove~** object's help file. It shows us a way to tell when the **groove~** object loops back to the beginning using three objects: (**delta~**, **<=~**, and **edge~)**. Used in combination, they detect when the input ramp from the **groove~** object drops back to 0. and sends out a *bang* message at that time.

Since we're dealing with a loop, 0. is both the start of the loop and also signals the end, as well. (You may want to save that humble trio of objects as an abstraction—there are a lot of Max patching situations where you'll want to trigger something based on the wrap of a ramp signal.)

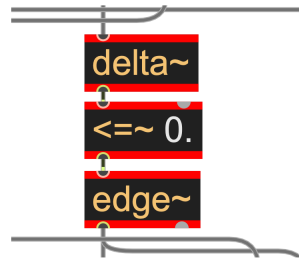


Figure 2-26

We will use this *bang* message to trigger recording using the **record~** object, and also to control playback with the **wave~** object. If we did this with a **toggle** object, the *bang* from 0. would start recording, turn it off, and play it —but it would cycle endlessly between record and play and record and play…

To avoid this problem, we need to create a structure where we have the *bang* message go into a **gate**, perform a series of actions for recording, then close that gate at the end of the loop, and open a different gate for the next *bang* message to start playback, and then close the second **gate**. To make the logic a little clearer, we're going to unpack the sequence step by step:

1. When we want to record, we "arm" record by opening the first **gate** object. This must happen *before* the loop starts.
2. That gate allows a *bang* message through when the sync signal from the primary looper goes to 0., signaling the beginning of the loop
3. This first *bang* message
    a. turns off any playback

b. clears the contents of the **buffer~** object in our secondary looper patcher

c. Sends a message to set a **live.text** object to indicate that recording has been enabled, “Recording,”

d. Triggers recording

e. Opens gate 2

f. Closes the first **gate** object so recording is not retriggered by more *bang* messages

Then…

1. When the next bang message is let through the second gate, it
    a. Sends a message to set up the **live.text** object to indicate that recording has been disabled

    b. Disables recording,

    c. Starts playback

    d. Closes its own gate

As before, we’ll use **trigger** (**t**) objects to perform these actions quickly and in sequence.

This is the most complicated part of our new abstraction for secondary loopers/tracks. Let’s go over all four elements of the patcher and then see what the sequence of messages we’ll use **trigger** (**t**) objects to send:

Reset

1. Turn off both of the gates with a “0”

2. Disable playback by sending a “0” to a **selector~** object that either lets the sync signal through or not. No signal means no playing back.
3. Disable recording by sending a “0” to the **record~** object
4. Clear the buffer by sending a “clear” message to the **buffer~** object

Arm/Record

1. Enable recording by sending a “1” to the first **gate**. This gate allows a *bang* message through when sync hits 0. When the *bang* gets through at the beginning of the loop:
    a. Turn off playback by sending a “0” to the **wave~** object
    b. Clear the buffer by sending a with a “clear” message to the to the buffer~ object
    c. Send a “1” message to set the **live.text** object to indicate recording has been enabled
    d. Starts record using the same “1” message and sending it to the record~ object
    e. Open Gate 2 by sending it a “1” message
    f. Close Gate 1 by sending it a “0” message
2. Then the next bang at the end of the loop
    a. Set the **live.tab** outside the abstraction to “Play” with “2”
    b. Send a “0” message to set the **live.text** object to indicate that recording has been disabled
    c. Disable recording using the same “0” message and sending it to the **record~** object
    d. Enable playback by sending a “1” to a **selector~** object that allows the sync signal to pass through.

a. Close Gate 2 by sending it a “0” message

Play

1. Sends a “0” to ensure recording is off
2. Sends a “1” to start play by opening the **selector~** object that lets the sync signal through

Stop

1. Disables playback by sending a “0” to a **selector~** object that either lets the sync signal through

In the following patch, you’ll notice that we have added the same **trapezoid~**-based technique for declicking, as it is what we use in our original abstraction. It runs with the same shared sync signal.

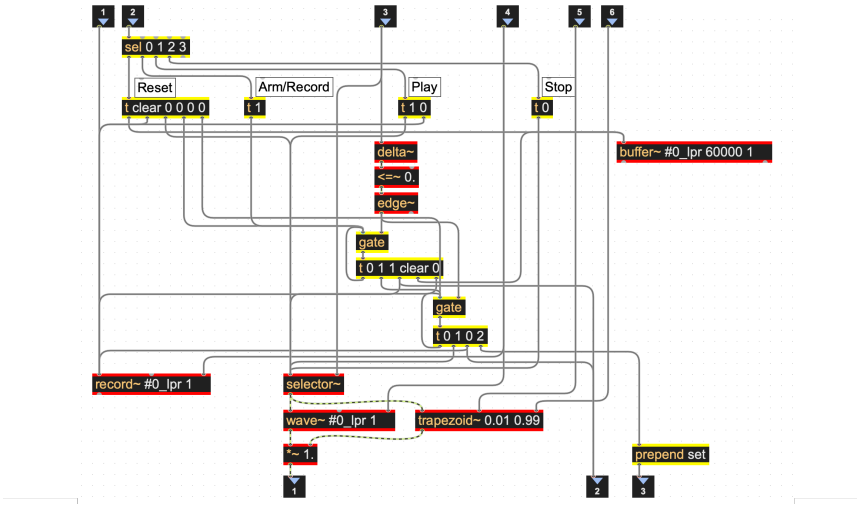


Figure 2-27

Performance UI

(Example Patcher: 2-8-Performance-UI)

As we've done with our previous looper, we use a **live.tab** object with the names of the four actions we need for our looper: Reset, Arm/Record, Play, and Stop. We've looked at the internals of our abstractions to verify the sequence of operations we need to perform, and we use a **select** (**sel**) object to pass a *bang* message as each mode is selected. There's one detail to pay attention to: Since Play is set automatically in the abstraction following the recording of a loop, we'll need to send a message "set 2" to the **live.tab** object that selects "Play" when play begins. (We use the "set" message to change this without outputting an index.) There is also a **routepass** object at the top of our patcher that is connected to our primary looper's **live.tab** object output so that Reset will be selected on all the secondary loopers' **live.tab** objects when Reset is selected on the primary looper.

We also have a **live.text** message to display "Recording" that is triggered when recording starts at the beginning of a loop and is then turned off when Play is automatically selected. Since the **live.text** objects in the secondary loopers are set by the primary looper's output, clicking on the **live.text** objects in the secondary loopers is disabled by using the objects' Inspectors and choosing the "Ignore Click" setting which turns them into display objects only..

Finally, we've added the same declicking patching using the **trapezoid~** object and its controls to the secondary looper patchers.

Once we have connected our sync and loop max message, we're ready to start looping.

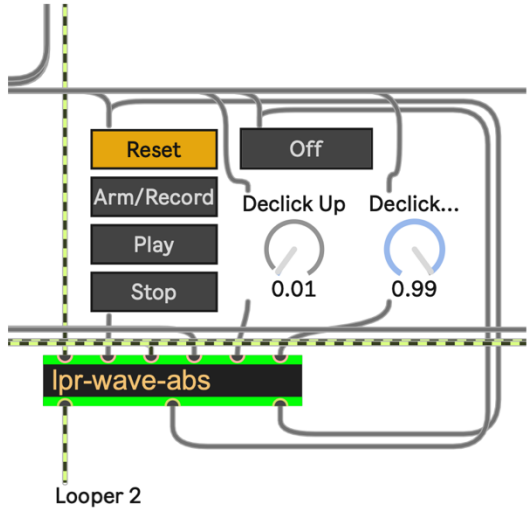


Figure 2-28

You can now copy and paste as many secondary looper patches as you'd like, and connect up the sync and loop outputs from the primary looper.

For this example, we have a primary looper and three secondary loopers.

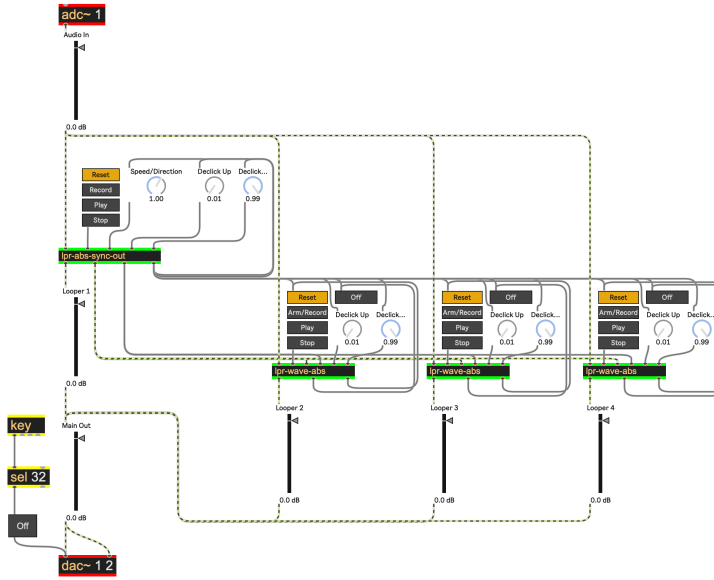


Figure 2-29

We have also added them to presentation mode to create a performance-ready interface.

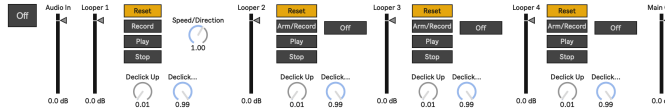


Figure 2-30

Here's the advantage to this version of a multitrack looper: you can mix asynchronous and synchronous loopers in the same patch, and you can also process the audio of each track/loop separately for complex variations.

The main disadvantage is that although playback will always be synchronous at any speed, synchronous recording must always take

place at normal speed. After you've recorded your input, you can change speed and direction as much as you want, and the audio will stay in sync.

Chapter 3:
Basic Looper (gen~)

Introduction to the gen~ Environment in Max

The **gen~** object provides a powerful and efficient environment for building audio loopers within Max. The **gen~** object allows you to program visually, similar to how you work in Max. However, it compiles your patching within the **gen~** patcher on the fly and runs it efficiently as C code. This combination provides the ease and familiarity of a visual programming environment alongside the performance of text-based code.

There are some key differences between working in Max and using the **gen~** object for patching.

1. Everything in **gen~** operates on a sample-by-sample basis
2. **gen~** utilizes a smaller set of objects, known as operators
3. The **gen~** environment does not support message types

While this chapter presumes a greater working knowledge of Max than previous chapters, we'll be working through patches sequentially in a way that breaks down the elements of a **gen~** patcher into bite-sized sections for the less experienced. Note: for Max for Live users (the version of Max included with Ableton Live): You will not be able to edit **gen~** patchers unless you own a standalone copy of Max.

To start patching in the Gen environment, add a **gen~** object to your Max patcher just as you would any other Max object.



Figure 3-01

Gen objects resemble Max abstractions in that they can be opened by double-clicking on them in a locked patcher, option/alt double-clicking in an unlocked patcher, or sending an “open” **message**. The initial default version of the object has two inlets and an outlet.

What you see when you open it should look something like this:

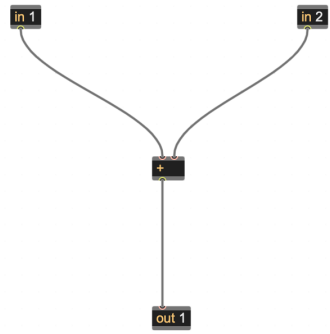


Figure 3-02

The **in** operators bring signals into the **gen~** patcher from Max. You then do the processing work inside the **gen~** environment and output the resulting signal via the **out** operator. Notice that the **in** and **out** operators are numbered, allowing for the creation of multiple unique ins and outs.

The **gen~** patching environment looks similar to a Max patcher in that we add **gen~** operators (each of which has a specific function) to a patcher and connect them together. As we do so, the **gen~** patcher compiles itself as we patch. What might appear similar to Max objects, in **gen~** are called “operators.” Like Max objects, they represent a specific

functionality: they have a task that they do. Unlike Max objects, the functional units of a **gen~** patcher can exist as **gen~** operators, or work with the GenExpr programming language using the **expr** and **codebox** operators, objects, or as GenExpr code. We will focus on operator-based object workflows for this chapter, and we will use the term “operators” to distinguish the contents of a **gen~** patcher from Max patchers.

One of the things that beginners to **gen~** patching sometimes find daunting is that **gen~** does its work one sample at a time, and that there are no messages inside a **gen~** object: every bit of data that passes between **gen~** operators is a single byte of data and that data can represent audio signals, operators that control the flow of data in a gen~ patcher, and other operators that handle timing. Learning to use **gen~** operators involves making sense of the different uses of single-sample operators.

To help you navigate the **gen~** environment, we’re going to use a color code similar to the colors used in the Max patchers in the previous chapters’ sample patchers to help keep operators and their functions—for this patcher—more visually apparent. Since we’re working with single-sample operators that pass data at signal rates, the labeling will be slightly different.

Color coding frameworks for this chapter

Red operators will be used to indicate audio in a patcher. Everything is data in **gen~**, but we will use the term audio to refer to data that is being used for audio. Color coding will focus on function, where operators will be used for data as audio, and others for data that is not audio, for example, a **selector** may have different colors based on its use.

Here are some examples of **gen~** operators we’ll be using as we create our loopers that will be colored red:

- Audio input to gen~ (the **in** operator)

- Writing/recording audio, again, as data (the **poke** operator)
- Operators that store audio in the patcher examples (the **data** operator)
- Operators that read/play audio (the **peek** operator)
- Operators that select an audio signal to pass/not pass (the **selector** operator)
- Audio and signal output (the **out** operator)

Blue: parameter operators (**param**). **Param** operators function as parameters that control what is happening inside a **gen~** patcher that can be modified with messages from outside the **gen~** operator. The first argument of a **param** is the name of the **param**, and can be arbitrarily named (we have found it best to use names that are short and meaningfully describe the functions controlled. The **param** names used in this patcher:

- record
- play
- latcher
- resetcounter

Yellow operators indicate timing-related operators in this patcher, which control when data is allowed to pass or how long a portion of the patcher takes to do its work. For example:

- A **counter** operator controls when a **poke** operator writes audio data (recording) and when a **peek** operator reads and outputs audio (playing).
- A **latch** operator can be used to sample and hold data. In our sample patcher, we'll use it to set the maximum of the playback counter (which corresponds to the loop length in the example patcher we'll create).

- **The !-** operator is used to stop/reset playback in conjunction with the play **param** and **counter** operators.
- A **selector** operator is used to send the output of a **counter** operator to a **poke**, operator, controlling where to send audio data to write/record when the looper is in record mode.

Gray: As with the Max patching examples, operators we have added to the previous iteration of a patcher are in gray to help keep focus on new operators and sections of our patcher.

Record, Play, Stop, and Reset in a gen~ Looper

(Example Patcher: 3-1-Record-Play)

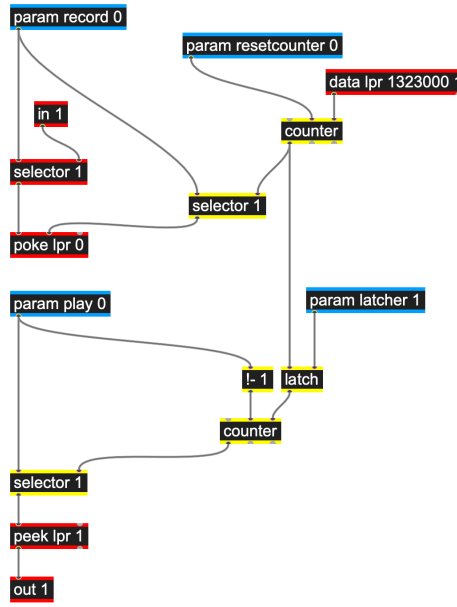


Figure 3-03

For our very first bit of patching in the **gen~** environment, we're going to focus on the basics of looping: recording, playing, stopping/pausing, and resetting.

The first step is to add a **gen~** to your Max patcher just as you would any other Max object.



Figure 3-04

Double-click on the **gen~** object to reveal the internals of the patcher. You'll see the default two **in** operators, an **out** operator, and a plus **(+)** operator that adds the two inputs together one sample at a time:

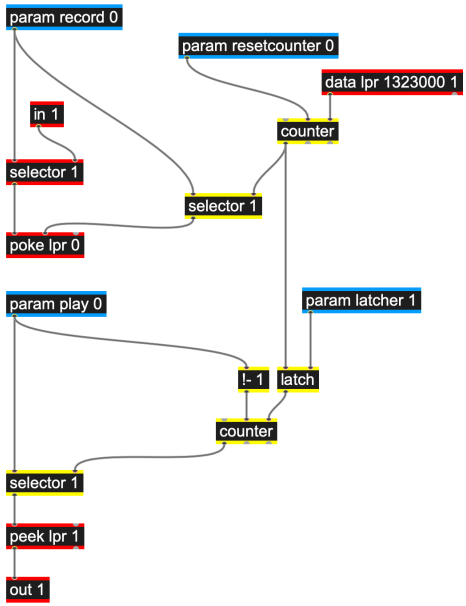


Figure 3-05

We'll begin by removing the operators from this patcher that we don't need. Select and delete everything in the patch except for the **in 1** and **out 1** operators.

We'll begin by adding a switch that will let us turn recording on using an operator similar to the **gate~** object we'd use in a Max patch: the **selector** operator. When we add the operator, we'll add the argument 1 to it to indicate that there will be one input (if you increase the number of the argument, it adds inputs—a **selector** 2 operator would have two inputs, for example).

Connect the output of the **in 1** operator to the right input of the **selector 1** operator to route the signal input of our patcher to the gate we've created.

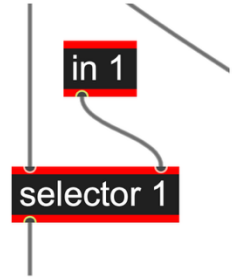


Figure 3-06

One of the differences from the Max environment you're used to working with is that **gen~** does not communicate internally using messages. You can, however, send messages from your Max patcher into the **gen~** object by adding a **param** (parameter) operator to the **gen~** patcher. Like the Max **gate~** object, the **selector** operator uses a value of zero to close the gate, and a 1 to pass the input through.

Add a **param** operator with a first argument “record” to specify the message from outside the **gen~** object, and a 0 to specify is the default setting for the param (so the gate is “off” by default).

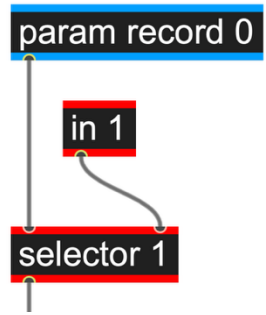


Figure 3-07

In the Max patcher that contains our **gen~** object, we’ll enable recording by adding a **message** box connected to the **gen~** object’s inlet that contains the **message** “record” with a replaceable variable **$1** and attaching it to a **toggle** object. When the toggle sends a 0 or 1, it sets the value of the **param** operator named “record” inside the **gen~** object to that value, opening and closing the selector as desired.

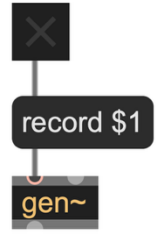


Figure 3-08

Next, we need a source of audio input to our **gen~** object. This first inlet of our **gen~** will also receive audio input from the Max patcher (where we’re using an **adc~** object for input). You’ll notice that both the audio input and the output of the message box are connected to the inlet of the **gen~** object: the input to **gen~** is smart in that it will sort the named

message and audio to their proper destinations inside the **gen~** object. We connect the **gen~** object's output to a **dac~** object in our patcher.

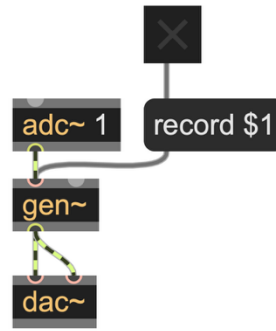


Figure 3-09

It's time to patch the inside of our **gen~** object to set up recording and playback. We'll need a place to store the audio we read and write, and we'll use a **data** operator for that purpose. We'll add a **data** operator with three arguments: a name (lpr) for the object, a number that sets the size of the operator in samples (1323000 = 30 secs at 44100 KHz), and the number of channels in our buffer (1).



Figure 3-10

We will be using two Gen operators for recording and playback, whose names you may recognize from their Max counterparts: **poke** to write (record) audio and **peek** to play (read) the audio. Together, the **data**, **poke**, and **peek** operators work together by sharing a common name "lpr" (like the previously examined trio of **record~**/**buffer~**/**groove~** examples).

For writing/recording audio, we'll add a **poke** operator with arguments that specify the name of the data object we'll write to and an additional argument of "0", which specifies we will only be using the first channel of audio (channel numbering starts at 0).

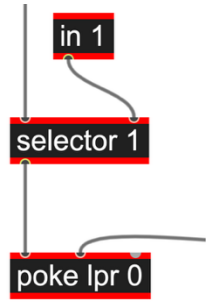


Figure 3-11

For reading/playing, we'll use a **peek** operator with arguments that specify a buffer name and channel offset.



Figure 3-12

Now that we have the recording and playback operators in place, we need a timing mechanism that does two things:

1. We need a timing mechanism that lets our trio of data/read/write operators know at what location in time to do all this writing/storing/reading in our buffer.
2. We also need to know *when* in time to do them.

While there are several ways to do this in **gen~**, we will start with basic combinations of the abovementioned operators using three new operators: **counter**, **latch**, and **!-** (reverse subtraction).

The **counter** operator is the heart of our timing control patching: it counts through the number of samples in our buffer at a single-sample rate. To

set the **counter**'s initial upper limit, we'll use the fact that the **data** operator sends the number of samples it contains from its outlet. All we need to do is connect the output of the **data** operator to the rightmost outlet (the upper limit) of the **counter** operator.

Whenever we begin our loop, we also want it to start counting from 0. We can do this by adding a named **param** operator (resetcounter) with an argument of 0 to set the default state (the value of 0 indicates that our counter is always running by default) and connecting its outlet to the middle inlet of the **counter** operator. Sending the message *resetcounter 1* to the **gen~** object in the Max patch will automatically reset our **counter** to 0.

Here's what our timing patch looks like with its upper limit set and a param object to reset the count:

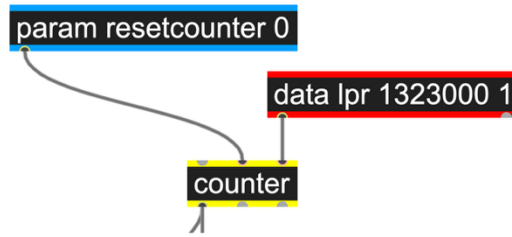


Figure 3-13

The output of our **counter** is used to set the point where we write (record) audio to the **data** operator using a **poke** operator. While that's happening, we'll also need to have the **counter** start running at zero by setting the resetcounter **param** to 0, indicating the location in the **data** operator to store the audio. To do that, we'll add a **param record 0** operator that will let us enable/disable recording by sending 0 or 1 from our MSP patcher and use the **param** object's operator together with a **selector** operator that will be opened or closed by the **param record 0** operator's output, letting in audio. We also connected the **param record 0** operator's output to the left input of the **selector** operator we're using to enable/disable writing to the **data** operator, letting in the count. This

way, we can use the record button in our MSP patch to handle two tasks simultaneously. Here's what happens when we toggle a record button in the parent MSP patch:

- Everything in the **gen~** object is reset using the generic reset **message**, which enables audio input (via the **selector** and the **param record 1** operators)
- The **counter** is reset and started (resetcounter 0)
- The count is sent to the **poke** operator via the **selector** and **param record 1** operators, and the **poke** operator's output is informed by the **counter** where to store the audio in the **data** operator

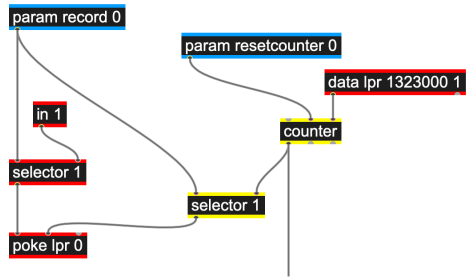


Figure 3-14

In our parent MSP looper patcher, we can control this using a single **message box** containing two messages separated by a comma. The order of the messages is important: the reset must be made *before* we start recording.

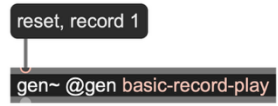


Figure 3-15

Once we have audio stored in the **data** operator, how do we read (play) it?

First, we need to set the end (length) of the loop to be played, i.e., when we stop recording. Since we restarted the counter to 0 when recording, once the end is set, we have a loop. To do this, we will use a second **counter** for playback set with the **latch** operator. The **latch** operator is similar to Max sample and hold objects—single-sample data (in this case, the current sample location in the buffer we are writing data to) flows through the operator until it receives a non-zero value in its left inlet, at which time the last value will be input until the next non-zero input. We'll use the **latch** operator to set our second playback **counter** to a maximum value. When the **counter** reaches *that* number, it loops back to zero. We've added a **param latcher 1** operator attached to the **latch** that tells it *when* to "latch" (set) the maximum length of the loop. Again, when the **latch** operator receives non-zero values, it simply passes the **counter** value through until it is instructed to store and hold that value as the end of the**counter**.

We also need to tell this **counter** operator to reset (1) or to run (0). However, the input from the **param play** operator comes from a toggle object in the parent patch. How can we send the *opposite* of the **param play 0** operator's input to the latch? We can do this using a **!- 1** operator, which will "flip" that toggled parameter input such that an incoming 1 becomes a 0 and an incoming 0 becomes a 1.

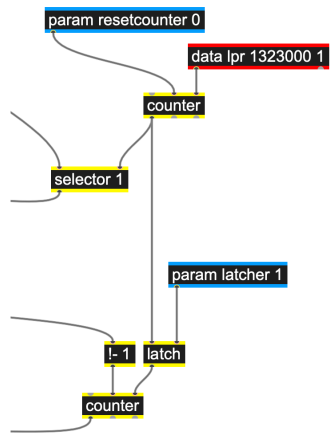


Figure 3-16

Finally, we need to inform the **peek** operator of the sample location from which it is retrieving data from the **data** operator. To do that, we add a **selector** operator to the output of the second **counter** operator. While playback is enabled, the **selector** operator sends the current sample count for the audio loop we've recorded to the **peek lpr 1** operator and outputs the result.

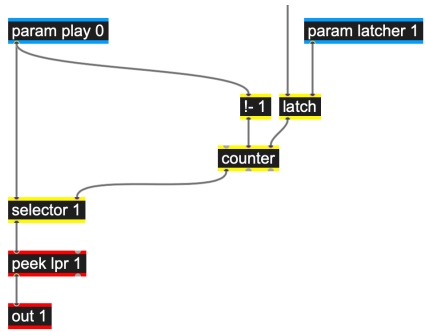


Figure 3-17

Here's what the completed **gen~** patcher now looks like:

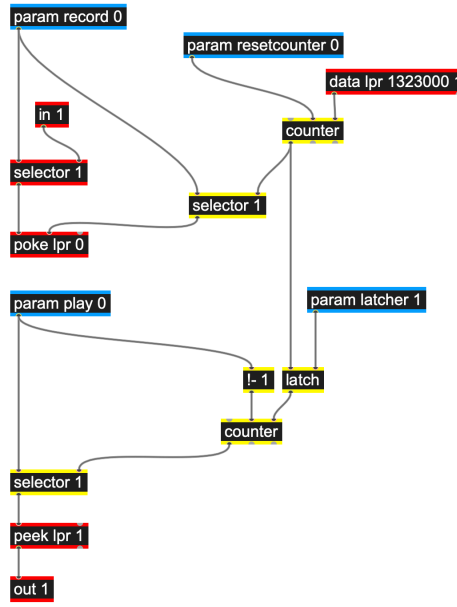


Figure 3-18

To see a large annotated **gen~** patcher for this section, open the example patcher and double-click the **gen~** to see its contents.

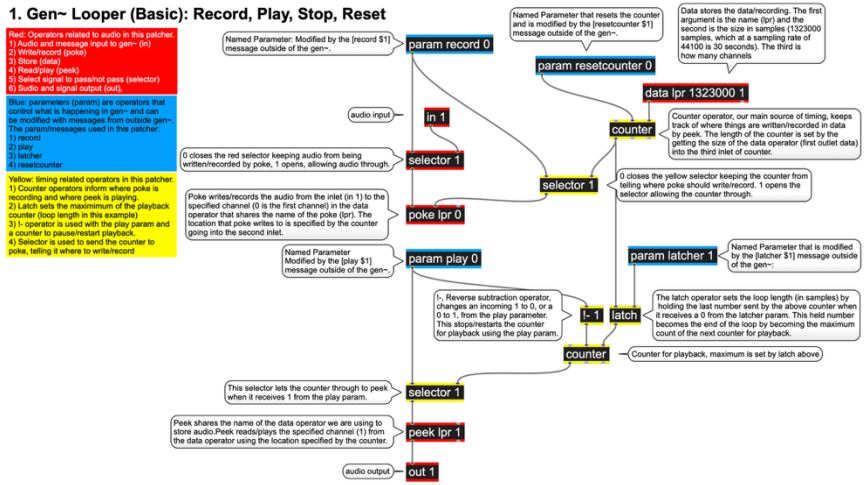


Figure 3-19

Wrapping it all up

Now that we've got a working Gen looper patch, it's time to connect it to our parent patcher, add some control patching, and create a UI for the patcher.

You may have noticed that all the **gen~** patchers in the Project folders distributed with this book are saved as *.gendsp* files. We do this for the same reason we save Max patches as abstractions—to give us bits of patching we can refer to by name and reuse. Gen patches saved as *.gendsp* files are just that. There's also a pedagogical benefit for doing this from your point of view, too: if you make an accidental deletion or error while you're patching, you can always reload the .**gendsp** abstraction to return to the original.

Sending messages from Max to gen~

There are a couple of situations where we'd like to control our **gen~** abstraction (which appears in the Max patcher as a **gen~ @gen basic-record-play** object).

- To stop everything, we send the message *reset*, which sets all **param** operators to their default state.
- To record, we send two messages to the **gen~** object: a *reset* message to reset everything, followed by the message *record 1*, which starts counting samples to record, and turns on the **poke lpr 0** object to start writing samples.
- To play back our recorded loop. A *record 0* message disables recording, the *latcher 0* message sets the maximum sample count for our loop, the *resetcounter 1* message starts counting at 0, and the *play 1* message starts reading from the lpr **buffer** operator. To stop playback, we simply set the play **param** to "0."

Here's what the **message box** objects that send those messages to the **gen~ @gen basic-record-play** object look like. (Note that the messages are sent in a specific order in each box, and each message is separated from the previous one by adding a comma.)

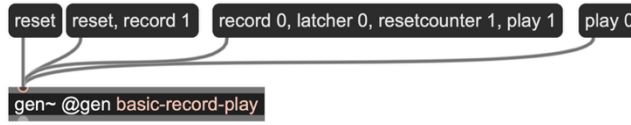


Figure 3-20

As you've probably noticed, we have four **message box** objects that contain all the messages necessary to record, play, stop playback, and reset our **gen~** patcher. Grouping things in this way means that we can use a Max **sel** (selector) object to choose all four modes with a single number value, which determines which of the four sets of messages we send to the **gen~** object.

This will make creating a user interface for our patcher much simpler.

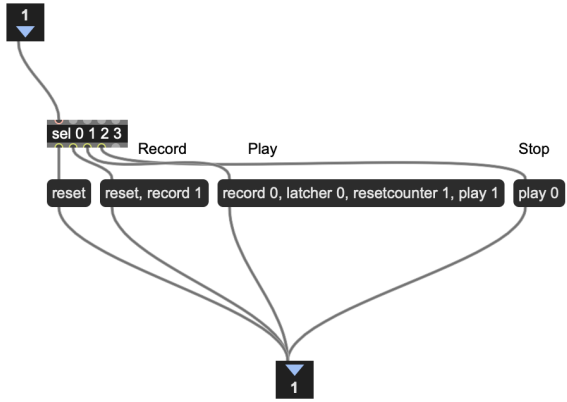


Figure 3-21

In our earlier patcher examples, we added mixing options and used the spacebar to turn the patcher on and off. For this patcher, we only need to select one of four messages (reset, record, play, stop). We can add a **live.tab** object to our patcher. It functions as a labeled multibutton object.

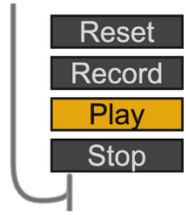


Figure 3-22

Create a new object and type “live.tab.” In the inspector for the object, in the “Range/Enum” section, select “edit” and type “Reset Record Play Stop.” Then, a bit below that, select Initial Enable and set it to 0 so that it selects the “Reset” option when the patcher is opened.

We will use the **live.tab** object with **sel** to send a bang to the desired message. Reset (0) goes to the reset **message**, Record (1) goes to the record **message,** etc.

Adding Overdubbing to a gen~ Looper

(Example Patcher: 3-2-Overdub)

For the next patcher, let's add the ability to overdub audio input to what we've already recorded.

The Gen **poke** operator already has an overdubbing function built into it. If you mouse over the third inlet to our named **poke** operator, you will see a bubble that says, "overdub mix: amount of original signal to preserve."

Based on the information about the overdub attribute, we will add a **param mix 1.** operator to our patch. A value of 1 for the argument sets the operator's initial state, preserving all the original audio signal data. The parameter's range is from 1.0 (100 percent of the original signal preserved) to 0 (no original signal preserved). This allows us to make creative choices in how our overdubbed audio is mixed.

When we use our patcher for overdubbing, we'll want to overdub onto what we just played. That means our **poke** for writing needs to receive the **counter** we are using for playing: if you use the **counter** that has as its length the full **data** operator, it might not overdub where we want. We need to tell the **poke** operator what our count is to write the overdubbed signal to the correct location. To do this, we will modify our **selector** operator—going into our **poke** operator—to select either the **data** operator counter (full-length) when first recording (1), or the play **counter** for overdubbing (2). We do this by adding a **+ 1** to the output of the play **param mix 1.** operator. This change changes the functionality to do three things:

1. Choosing between the two **counters**

2. Also, starting the play **counter**
3. Opening the **selector 2** operator to send the counter value to the **peek lpr 1** operator (play)**.**

One last thing: we have added a "**-1**" between the **selector** and **poke**. This offsets the recording counter by one sample relative to the playback counter, preventing **poke** from writing to the same data location that **peek** is reading at that moment. Without this offset, unpredictable behavior can occur.

The math here is a little fudged, but it works—and in this case, stability and simplicity is more important than elegance. A one-sample delay—1/44,100 of a second at a 44.1 kHz sample rate (or proportionally less delay at higher rates)—is effectively inaudible, while providing the predictability we need.

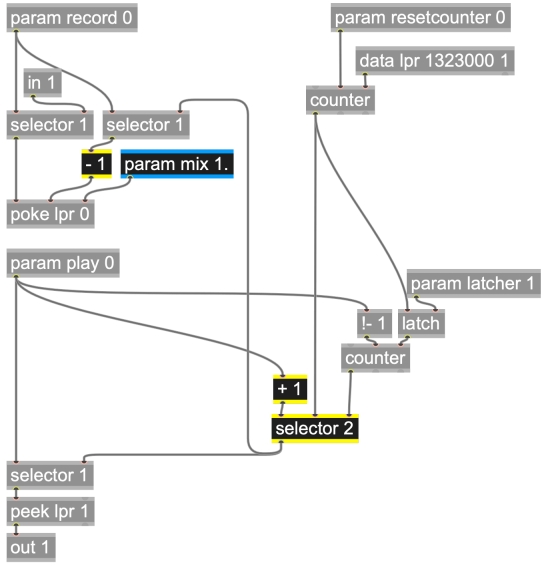


Figure 3-23

Performance UI

To simplify the patching for the set of control messages and our live tab object, we can move that part of our patch to a Max subpatcher (**p previously-covered-messages** in the example patch) created with the “Encapsulate” option in the “Edit” menu. As we move through each section of this chapter, we will use this to declutter the main window.

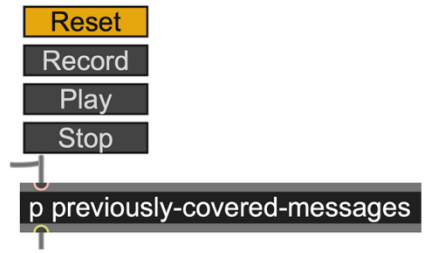


Figure 3-24

Here’s the inside of the sub-patcher:

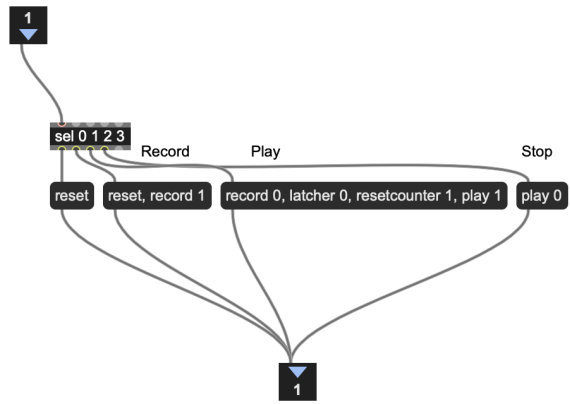


Figure 3-25

To enable overdubbing in the **gen~** patch, we’ll add two new messages going into **gen~**: record and mix. Then use the combination of **live.text**

object whose text will read Overdub Off/Overdub On which sends a *record 0 or 1* message to the **gen~** patcher. Along with this, a **live.dial** (as described in our first chapter) with messages that use a replaceable argument ($1) in a message box to set the amount of the original signal to preserve when overdubbing.

The *record* message for overdubbing is the same as we used earlier, but now we are using the play **counter** to know where to record.

We are now recording while playing and using the overdub feature of the **poke** operator to preserve the original signal as we overdub.

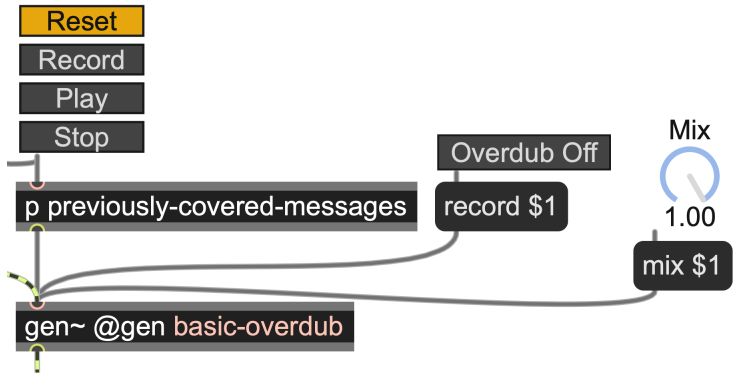


Figure 3-26

Removing Clicks in a gen~ Looper

(Example Patcher:3-3-Declicking)

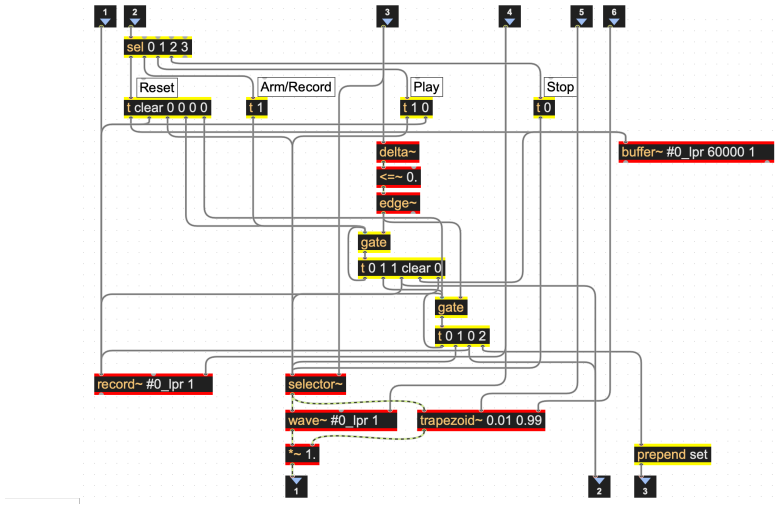


Figure 3-27

When capturing, manipulating, and playing back audio, we often encounter clicks in the output that need to be managed. (The clicks occur whenever we create discontinuities from one sample to another in the course of overdubbing or enabling recording.) While you may choose to embrace the clicks as an aesthetic decision, there are several methods you can use to remove clicks from your output.

We've already seen two examples earlier in this book:

1. We used a **gate~** with the **@ramptime** attribute when recording the audio into our patcher to prevent clicks in Chapter 1.
2. We used ramping, interpolation, and introduced the **trapezoid~** object to window our loop in Chapter 2.

This time, we will explore a way to use a gen~ patch similar to the **trapezoid~** object, differing in that it is inside our **gen~** patcher.

We have created a Gen abstraction (you can make abstractions inside of **gen~** just as you do in Max) called **scale-trapezoid**. As with the **trapezoid~** object, we specify the point at which the fade-in and fade-out functions are applied to the audio output (we use the **param declickup 0.01** and **param declickdown 0.99** operators to set the default values).

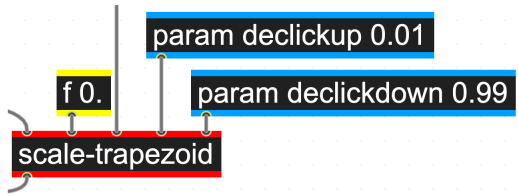


Figure 3-28

The release of Gen includes some patchers to create a trapezoidal function; the example can be found in Max->Help->Examples->gen.

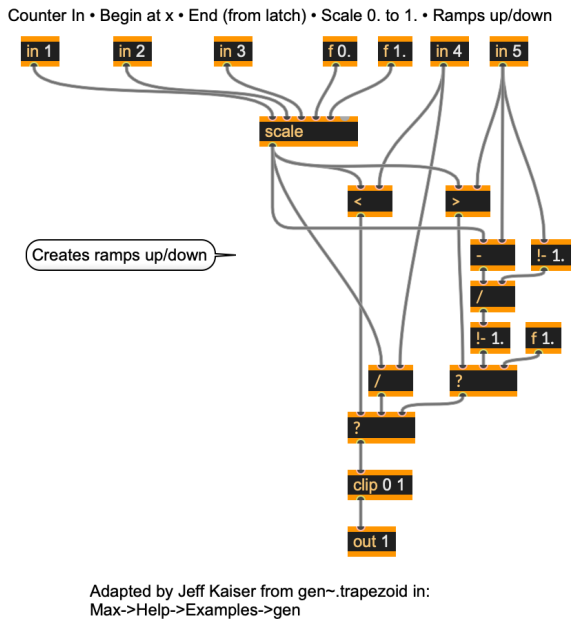


Figure 3-29

Double-click the **scale-trapezoid** abstraction to open it and examine its contents. Our modified version of the example creates an amplitude ramp at the beginning and end of the loop to help minimize clicking during playback.

It first gets the endpoint of the loop from the **latch** operator, uses the **f 0.** operator as the startpoint, and then scales those values from the incoming **counter** operator from 0. to 1. It then uses the value from the **param declickup 0.01** operator to create a ramp up at the beginning from 0. to 1. and the value from the declickdown **param** to ramp down from 1. to 0. It sends these changing ramp values to the multiply inputs (*) operator to create the amplitude ramp during playback.

The **selector** following the abstraction uses the **param declick** operator's value to select the constant float (**f 1.**) or select the changing ramp output of the abstraction **scale-trapezoid**.

The **float** operator (**f 1.**) sets the multiply inputs (*) operator to 1. when declick is not selected.

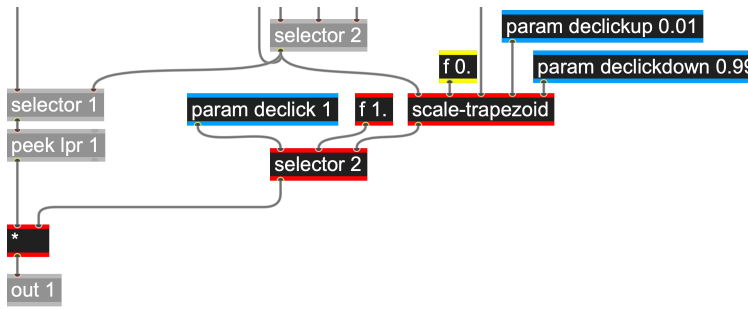


Figure 3-30

Performance UI

For the performance UI, we have added a **live.text** object used as a toggle with messages that use a replaceable argument ($1) in a **message box** to turn declicking on/off, and two **live.dial**s to set the ramp-up and ramp-down times using a similar m**essage box** technique.

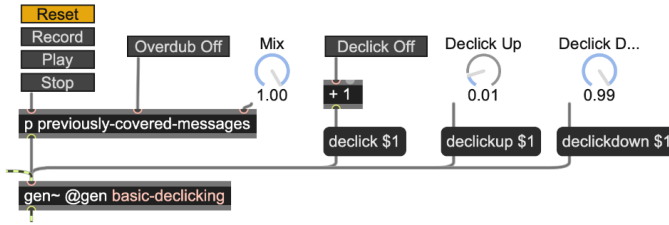


Figure 3-31

Shortening Loop Length in gen~

(Example Patcher: 3-4-Shortening)

We will next explore shortening the loop length and then using **selector** operators to switch between playing the full loop, the shortened loop, and the remainder from the shortened loop.

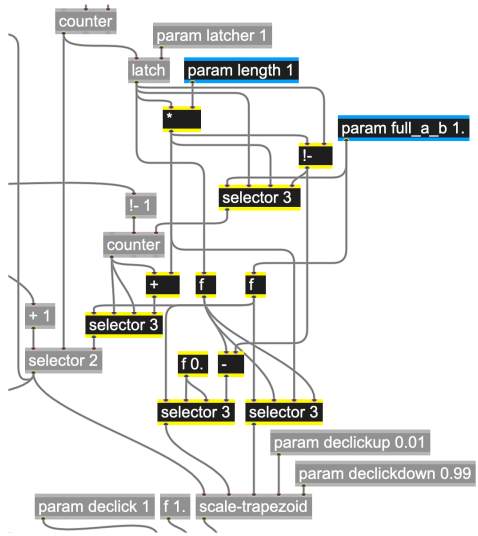


Figure 3-32

Shortening is accomplished simply by multiplying the length of the loop (given by our **latch** operator) by a number less than 1. For this example,

we use the **multiply** inputs operator (*) with a **param length 1** operator (for our patcher, we will refer to this shortened loop as A). We then also subtract the shortened length from the full length, and that value becomes our B. We can then get three loop lengths from the **latch** operator's output going into our first selector:

1. The full loop length in the **selector 3** operator's first inlet
2. The shortened loop length (A) in the **selector 3** operator's second inlet
3. The remainder (B) in the **selector 3** operator's third inlet.

This **selector** operator's output sends the result to both **latch** the playback **counter** and to the input of the declicking abstraction. The other **selector** operator chooses which one to use for playback: inlet one is "full," inlet two is "A," and for the third inlet, we add the shortened bit (A) to give us the starting point for the remainder "B."

Modifying Declicking

We are now managing playback from three different possible locations:

1. Counting samples from 0 to the value stored by the **latch** operator (full recorded loop).
2. Counting from 0. to a shortened latch value (A)
3. Counting from the end of the shortened latch value to the full latch value (B).

Since each of these possible playback locations involves different sample lengths, we will need to adjust the declicking section of our patcher to work with those ranges. We can use the **param full_a_b 1.** operator we already have to do this—its output is routed to two **selector** operators.

One **selector** operator calculates/sets values at the start of the loop, and one **selector** operator calculates/sets values for the end of the loop,

setting our **scale-trapezoid** abstraction as needed. The two **f** operators act as distribution points for **latch** and **param full_a_b** to not have so many patch cords going through the middle of the patcher.

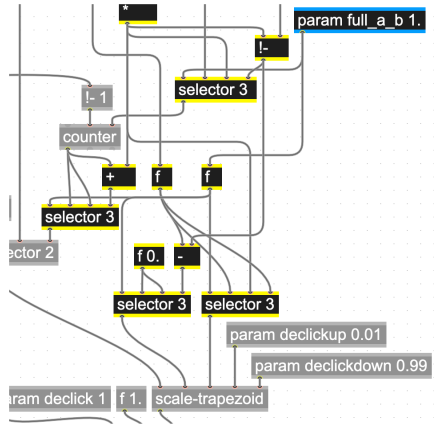


Figure 3-33

Performance UI

For the UI, we have added a **live.text/+ 1** object pair used as a switch, a message using a replaceable argument ($1) in a **message box** to select shortened loop lengths, and a **live.dial operator** using a replaceable argument ($1) in a **message box** to set the new loop length.

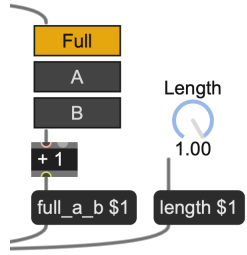


Figure 3-34

Auto-Insert and Rhythmic Overdubbing

(Example Patcher: 3-5-Auto-Insert)

Up until now, we've been adding features to this patch by modifying the contents of the **gen~** object. For this feature, we will provide metrical/rhythmic control of the overdub on/off for creative overdubbing—inserting/replacing audio depending on the mix setting—solely by patching in the Max environment.

If a Max user interface object, such as a **toggle** or **live.text** object receives a *bang* message, it reverses its current state from 0 to 1 or 1 to 0, and outputs the new value. Since a **metro** external object outputs bangs at a regular rate, we can use that output in combination with a **toggle** (or a **live.text** object, in this example) to rhythmically turn something on and off. We'll combine this with the overdub functionality we have already added to our **gen~** patcher to turn overdubbing on and off for creative, rapid-fire overdubbing or experimental rhythmic effect. (We're using the **metro** object in this example, but you could use random functions to create delightfully chaotic patterns and more.)

We start by adding a **metro** object connected to our overdub on/off that outputs *bang* messages at a default rate of 250 milliseconds. Adding a connected **routepass 0** object ensures that when we turn Auto-Insert off, it also turns off the Overdub On/Off toggle.

Next, we add a **live.dial** so we can vary the rate at which the **metro** object sends out *bang* messages (tempo).

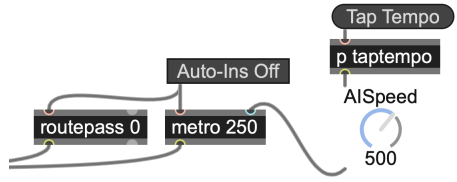


Figure 3-35

You'll also notice that we've added a new Max abstraction (**p taptempo**) and a button labeled Tap Tempo. This will allow us to set a rate for auto-insertion by tapping a button. Here's what the abstraction looks like (double-click the abstraction to see its contents):



Figure 3-36

This simple abstraction uses a **timer** to measure the time between taps. It then groups and sums every three taps and divides by 3 to create an average time, which is then passed on to our **live.dial** named AISpeed.

We have also added a **live.tab** that allows you to quickly select a predetermined amount of the original signal to preserve when overdubbing.

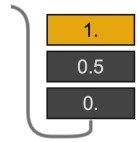


Figure 3-37

Randomizing Playback in a gen~ Looper

(Example Patcher: 3-6-Randomize)

For slicing and randomizing playback, we will combine objects both inside of **gen~** and outside in the Max patcher. There are many ways to accomplish the randomization of playback. For this example, we will be doing three things

1. Dividing the loop into multiple segments (slices) inside the **gen~** object.
2. Using a **metro** object to trigger and generate a **random** number in Max, with a message that will play the particular segment (division) corresponding to that random number
3. Adding a **live.text** toggle that lets us select either “Regular Mode” (uses the **counters** for playback that we previously added) or “Slice Mode” (that uses what we will call the **counter** operator we’ll add that is used for segment divisions.

Dividing the loop

Inside our Gen patcher, we add a **param segments 4** operator and **/** (divide) operator combination to set the number of segments (slices) we want. This divides our total loop length into the desired number of segments, giving us multiple new segment lengths to be played using the segment playback counter.

We instruct the **count** operator *to play a specific* segment by generating a random number within the range of the number of segment divisions in the parent Max patcher set, using the **param random 1** operator in our gen~. The random parameter value, the ***** (multiply) operator, and the **+** (addition) operator work together to set the point at which the patcher starts and ends playback.

We use the **selector 3** operator to select the full loop, section A, or section B, for slicing, which increases our randomization options.

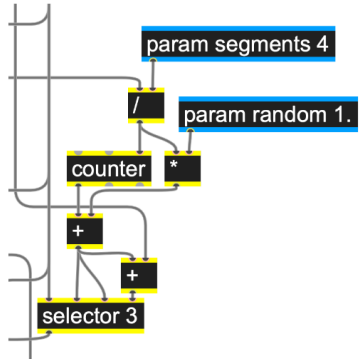


Figure 3-38

Finally, we use the new **param mode 1**. and **selector 2** operator pair to choose either Regular Mode, which plays back with our previously made playback counters in the first inlet, or the new segment **counter** (Slice Mode) in the second inlet.

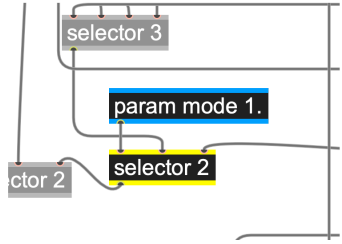


Figure 3-39

Performance UI

We'll need to add a user interface that lets us control loop slicing and random playback. In the figure below, you can see we added a **live.text** object to select Regular or Slice Mode, a **live dial** object to select the number of segments/slices we want, and a metro/random object pair of objects to choose which slice to play. (Notice that we're taking the output of the **live.dial** that sets the number of segments and using that value to

set the upper bounds for our random number generation. Finally, we have repurposed the Tap Tempo abstraction we created earlier to allow us to tap in the tempo for the metro slice.

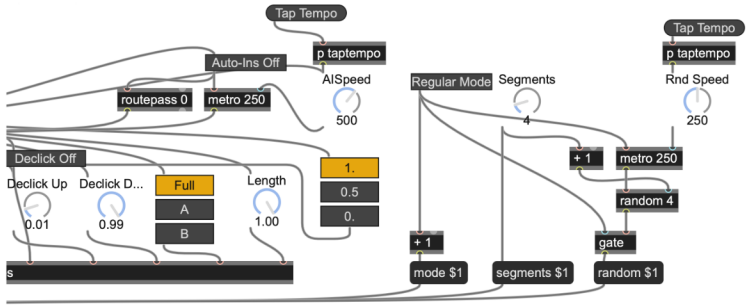


Figure 3-40

Changing Speed and Direction in a gen~ Looper

(Example Patcher: 3-7-Speed-Direction)

Changing speed and direction for playback with this version of a **gen~** looper is easy, but it comes with a caveat: overdubbing with *any* speed and direction other than 1 in a forward direction can create unexpected artifacts, which may or may not fit with your aesthetic. We will explore varispeed overdubbing in our next chapter.

To change the speed of our loop playback, we need only change the value by which the counter counts. A value greater than 1 will increase the speed, and a value between 0 and 1 will slow it down (0 stops the counter, of course).

We send this number to the **counter** operator by means of the **param speed 1** operator.

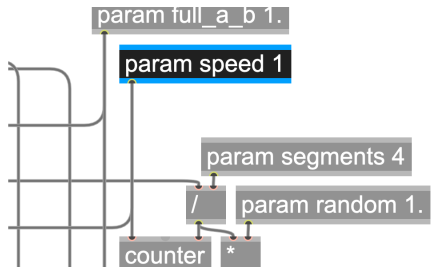


Figure 3-41

To change the direction of playback, we simply need to have our counter count backward. To do this, we once again add **!-**, the reverse subtraction operator that subtracts the first input from the second input. The **param direction 1** operator, in combination with an added **selector 2** operator, lets us choose inlet 1 (the regular and slice mode counters) or inlet 2 (that has those same counters counting backwards) with the **!-** set by the **latch** operator.

The **param direction** also serves double duty here. Because we still want the **poke** write position to remain one sample behind the **peek** read position, we previously applied a "-**1**" offset. When playback is reversed, however, that offset must become "**+1**" instead.

We accomplish this by using **param direction** to control a **selector** that chooses "**-1**" when playing forward and "**+1**" when playing backward, ensuring that the one-sample offset is preserved regardless of direction.

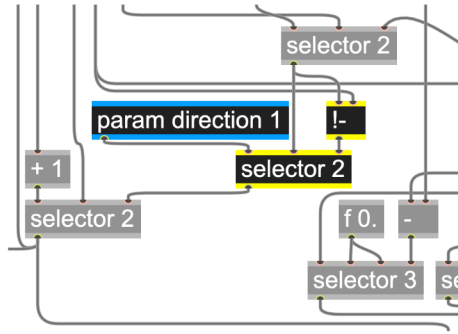


Figure 3-42

Performance UI

To control the speed and direction, we have added outside of the g**en~** a **live.dial** object for speed control and a **live.text**/**message box** object pair to select whether the counters in our patch count forward or backward.

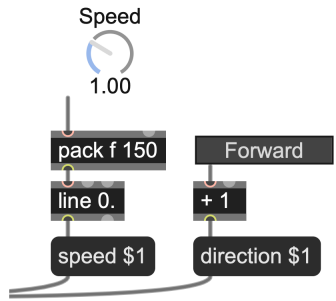


Figure 3-43

Performance UI

(Example Patcher: 3-8-Performance-UI)

For our final patcher, we need to put it all together in a way that makes sense for performance. As we did in previous patchers, we'll organize our user interface elements using Max's Presentation view, adding only those UI objects to Presentation View (once added, we can identify them by their salmon-colored outline.

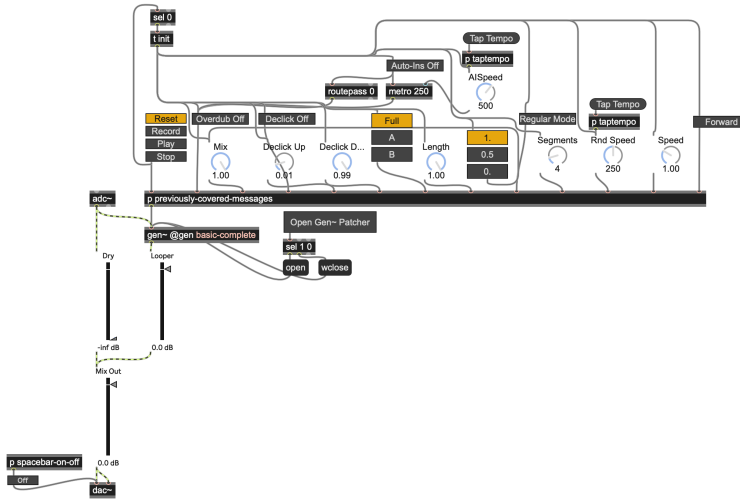


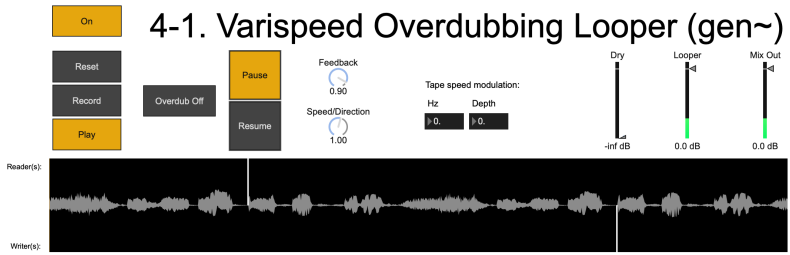
Figure 3-44

Toggling Presentation view lets us reorder the placements of objects added to the Presentation View in a way that makes performance easier and clearer.



Figure 3-45

Chapter 4:
Varispeed Overdubbing Looper (gen~)



Varispeed Overdubbing: Collaboration with Graham Wakefield

This chapter is something of an anomaly within the book. It addresses problems that require solutions more complex than are easily achieved using the usual visual, object-based approach to Max patching—at least at this time. As a result, it moves into more advanced territory.

Up until this point, we've used standard Gen operators in our patching. Doing so has enabled us to easily move from MSP to **gen~**-based visual patching, and to preserve the visual flow of data through the patchers we create.

Creating our new patcher requires us to use the Gen environment's **codebox** operator, which enables us to perform procedural text-based coding using the GenExpr language that is at the heart of **gen~** itself.

Everything in a **gen~**-based patcher—both the **gen~** operators and the contents of a **codebox** operator—is *merged* into a single representation used to generate and compile fast native CPU machine code on-the-fly. So, while there is absolutely no difference in terms of performance between using visual patching or using a **codebox** operator in **gen~** to do the same thing, there are two places where you absolutely need to use a **codebox** operator.

The **codebox** operator is the only way to have precise control of the order of reads and writes for working with delays and operations on **data** and **buffer** operators. Since we need to lay out a specific order for these operations in our looper, we'll be using a **codebox** operator. (The **codebox** operator is also necessary for something we don't need to do—to perform procedural branching and looping with if(), for(), or while().)

The patcher we're creating here requires approaches we cannot easily achieve using the usual visual, object-based approach to Max patching—at least at this time. As a result, it moves into more advanced territory.

Rather than following the step-by-step trajectory of earlier chapters, we recommend first reading below about the problem itself, and then examining the commented **codebox** in the example patcher to see one possible solution. You are, of course, free to skip the discussion entirely and simply use the patcher. However, we believe the ideas explored here are worth engaging with, as this chapter presents a varispeed overdubbing looper in **gen~** that overcomes key limitations by drawing on more technical DSP strategies.

Problems

There are problems that occur with the previous **gen~** patcher when overdubbing at different rates of speed and/or direction—in particular, when overdubbing at non-equal divisions of the speed—we get sonic artifacts that sound like distorted buzzing metallic sounds.

Additionally, there are clicks when the loop transitions from the end to the start. For some of us, the glitchiness can be embraced and used aesthetically, but for others, it is an unappealing mess.

The Fundamental Writing Problem

The distortion issue arises from the way digital audio buffers operate. A buffer is essentially an array of consecutive audio samples. When you write at speed = 1, each passing moment of real time corresponds to exactly one buffer location being filled. Sample 1 goes in slot 1, sample 2 in slot 2, and so forth—the buffer fills continuously with no gaps. It works and sounds great.

However, problems begin as soon as the write speed deviates from 1. For example, at speed = 1.5, the write head advances 1.5 samples per sample frame of real time. You write to position 0, then 1.5, then 3, then 4.5, and so on. The positions between those samples remain either unfilled or contain old data from previous passes. When the read head later encounters this patchwork of new and old samples, the result is a characteristic, distorted, buzzing, metallic sound.

The situation becomes even more complex when speed changes are smoothed, modulated, or reversed, since the write position is constantly moving at non-integer rates. No amount of interpolation or filtering on playback can repair audio data that was never written coherently in the first place.

In short, our original varispeed overdubbing method has somewhat corrupted the contents of the buffer, leading to those sonic artifacts. This behavior isn't a bug or an implementation mistake—it's a structural DSP problem, and a problem whose solution required careful thought.

We reached out to Graham Wakefield, the original author of **gen~** and co-author with Gregory Taylor of the book *Generating Sound & Organizing Time: Thinking with gen~ (Book 1)*. Graham, through a series

of emails and Zoom calls, worked with us to create a **codebox**-based patcher that solved our problem.

Two Viable Architectural Solutions

Once this problem was understood, Graham presented two possible solutions to us, unpacked each one, and then devised a hybrid patcher that combined elements of both solutions. The following descriptions are gleaned from our emails, Zoom calls, and example patchers he shared.

Solution 1: The Delay-Based Approach

The first solution treats the buffer not as a static loop, but as a continuously running delay line with feedback.

With this solution, the write head always advances at exactly one sample per sample frame of real time. A counter increments continuously, and its value is sent directly to **poke~**. This guarantees that every consecutive buffer position contains sequential audio samples. There are no gaps, and therefore, no distorted, buzzing, metallic artifacts.

Playback is handled by a read head that trails behind the write head by a fixed distance equal to the desired loop length. For example, a two-second loop at 44.1 kHz places the read head 88,200 samples behind the write head. This position is calculated by subtracting the loop length (in samples) from the current write position and reading from the buffer using **peek~** with interpolation enabled.

A conceptual shift is important: the “loop” is no longer a fixed region of the buffer. Instead, it is a moving window of audio that follows behind the present moment. The buffer itself continuously scrolls forward, wrapping around as needed.

Recording control can be handled in two ways.

1. One option is to freeze the write head when recording is disabled, locking the loop contents in place.
2. Another possibly more musically useful option is to keep the write head moving but stop mixing new input into it. This behaves like a delay pedal with infinite feedback: the loop continues, but no new material enters.

Because this system is fundamentally a feedback loop, audio is repeatedly read with interpolation. Over time, this results in a subtle loss of high frequencies. If you pitch a loop down and then back up, the sound will not return perfectly unchanged. Instead, it accumulates filtering, which produces a tape-like degradation that some musicians might find musically appealing.

Speed and pitch changes are achieved by changing the distance between the read and write heads. Bringing them closer together raises pitch and shortens the loop; moving them farther apart lowers pitch and lengthens it. This produces cascading pitch effects similar to those found in digital delay lines.

This approach is robust, efficient, and relatively simple to implement; however, it behaves more like a digital delay than a traditional tape loop.

Solution 2: The BBD-Like Variable-Speed Writer

Initially, the delay-based solution seemed like the answer, but the second solution Graham provided preserves the traditional looper metaphor: the entire buffer represents the loop, and playback speed can be adjusted independently of recording.

The challenge here is writing correctly at variable speeds. If speed = 2, two buffer samples must be written for every one sample of real time. If speed = 0.5, one buffer sample must be written every two samples of

real time. With non-integer or modulated speeds, the math becomes significantly more complex.

Solving this requires multiple write operations per sample frame. In **gen~**, this can only be achieved using a **codebox** containing a loop that calculates how many samples need to be written during each DSP tick. Fractional remainders must be tracked across frames to maintain accurate timing.

Interpolation is also required at the input stage. When writing fractional samples, simply duplicating input values introduces quantization noise. Instead, the input signal itself must be interpolated between successive samples before being written into the buffer.

According to Graham, this method has clear advantages. Once audio is written into the buffer, it remains pristine. You can pitch material up and down repeatedly, and when you return to the original speed, it will sound the same. However, the CPU cost scales directly with speed, making extreme modulation expensive. Additionally, any overdubbing performed at slowed speeds permanently bakes sample-rate conversion artifacts into the loop.

Additional Complications

Regardless of architecture, several other practical issues, in addition to the overdubbing artifacts, must be addressed:

- **Clicks** can occur when recording starts or stops, when overdub modes change, or when the read and write heads cross (especially during reverse playback or aggressive modulation). These are solved with short fades and crossfades.
- **Reverse playback** raises conceptual questions: should new overdubs play forward or backward relative to the reversed loop? A musically intuitive choice for some—overdubs playing forward—is technically harder and requires careful head management.

- **Latency compensation** is required to ensure that overdubbed material aligns correctly with the existing loop content.
- **Loop length changes without pitch effects** demand careful repositioning of the read head and crossfading to avoid audible jumps.
- **Feedback loops** can accumulate DC offset and spectral “dulling” as high frequencies are lost; this requires filtering that itself introduces latency and phase considerations.

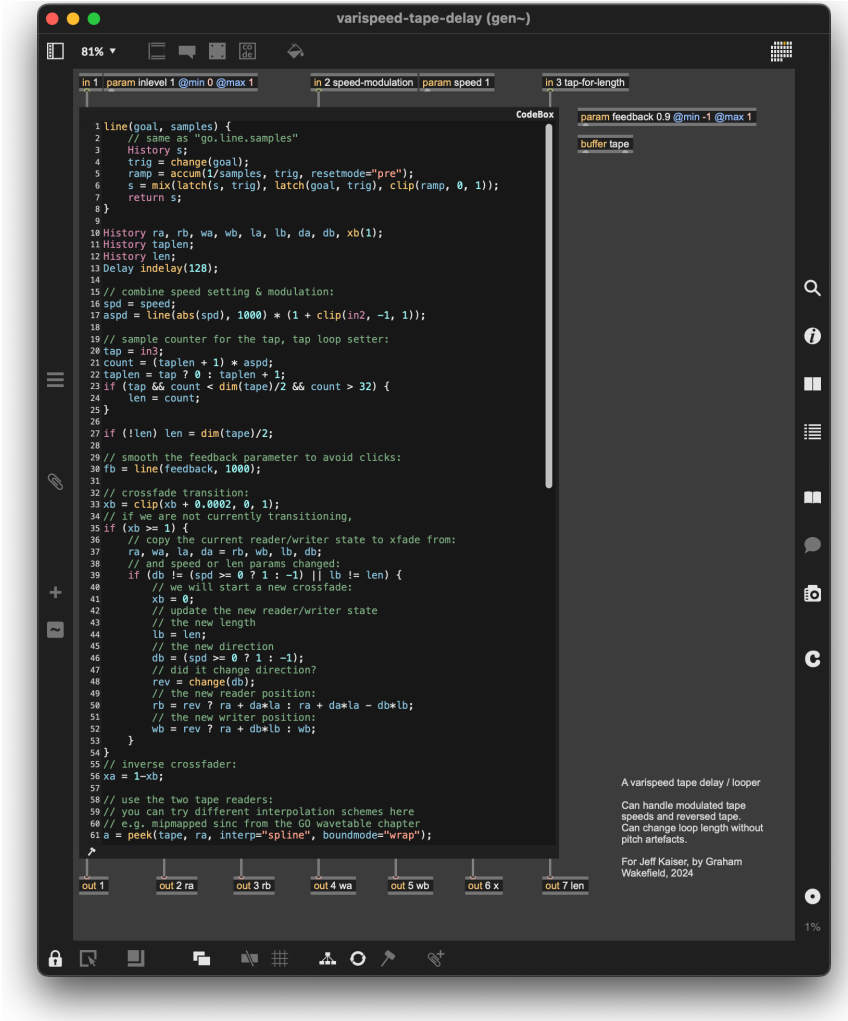
Graham’s Hybrid Tape-Style Solution

(Example Patcher: 4-1-Varispeed-Overdub-Gen)

The patcher presented in this chapter is based on a hybrid tape-style architecture developed in consultation with Graham. It combines the conceptual clarity of a tape metaphor with the technical gymnastics required to avoid buffer corruption.

In this design, speed controls the rate at which data moves through memory, including negative speeds for reverse playback. Graham created—using **codebox**—logic that swaps the heads’ reader and writer roles during direction reversals and uses rapid crossfades to prevent clicks. The beauty of this patcher is that overdubbing remains stable at all speeds and directions, even when using the speed modulation to simulate tape warble.

To explore the use of **codebox** in this patcher, open up the example, go into edit mode, and follow the comments to double-click and open **gen~**.



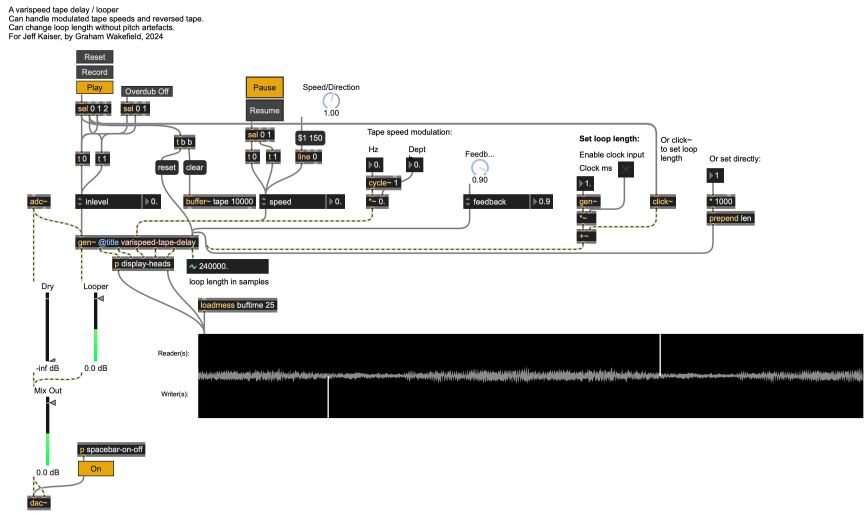
The loop length can be set in three ways: directly via a parameter, by tapping two impulses to define a duration, or by synchronizing continuously to an external clock. For this example patcher, we have foregrounded the tap method, as it is more common in live looping cases. However, if you take the patcher out of presentation mode, you

can see the other methods (in Edit mode). Smoothing the speed enables musically convincing tape-stop effects. A maximum speed limit is imposed to manage CPU load.

Some high-frequency loss will occur due to repeated interpolation, especially at extreme speeds. Rather than treating this as a flaw, the design embraces it as part of the tape-like character of the system. Achieving this required a substantial restructuring of the algorithm and the use of **codebox** for variable-speed writing loops.

While the complexity of this patcher goes beyond earlier examples in the book, it demonstrates the possibilities within the Max environment of **gen~** to build reliable, professional-quality loopers that support overdubbing, varispeed, and reverse playback without artifacts, and shows how far **gen~** can be pushed when its underlying DSP constraints are taken seriously. And like our other patchers, this has plenty of room to explore modification possibilities and make it work in a unique, personal manner.

4-1. Varispeed Overdubbing Looper (gen~)



Chapter 5:
Tape-Style Looper (rnbo~)

rnbo~ Introductions

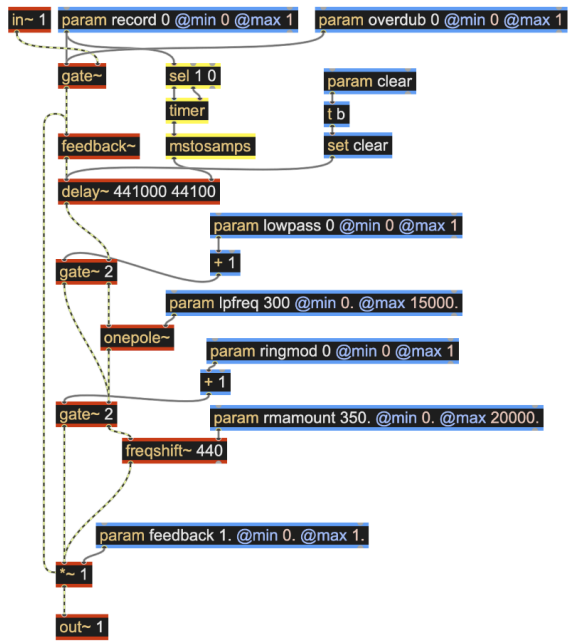


Figure 5-01

One of the goals of this book is to investigate the development of several kinds of loopers in the Max programming environment, and to do so in several forms using different tools available to us as Max users. In this chapter, we will build a tape-style looper using Max's RNBO object (**rnbo~**) and its programming environment.

Similar to the Gen environment we worked with in the previous chapter, **RNBO** is a visual programming environment that allows you to create patchers inside a **rnbo~** object. You add a **rnbo~** object to your Max patcher just as you would any other Max object, double-click on the object to open it, and add **rnbo~** operators and connect them just as you do in the Max and **gen~** environments.

What makes **RNBO** unique is the ability to export a patcher created inside of the **rnbo~** objects to several "targets." At the time of this writing, those targets include:

- VST3 plugins for Macintosh and Windows
- Audio Unit plugins for Mac
- C++ and Web Assembly code
- Raspberry Pi applications
- MSP external objects

RNBO is a visual programming environment that is a paid add-on to Max. However, even if you have not yet purchased a RNBO license, you can try out the patchers we will create in the following chapters for free in demo mode as long as your copy of Max is authorized and you are signed in. If you are using a copy of Max that comes with the Ableton Live Suite, you must sign in manually: select "User Account and Licenses" under the **Help** menu (you'll need to do this even if you are using Max in demo mode).

After signing in and authorizing, select "Show Package Manager" under the **File** menu and navigate to the **RNBO** package. Select and install it, RNBO will then be fully functional in demo mode, except that you cannot save your work. (Note that you also cannot copy and paste the code elsewhere to save it, either.) It is feasible that you could build a VST plugin and export it in demo mode, but then your patcher would be gone when you close Max.

This chapter assumes greater knowledge of Max than the previous chapters, but works sequentially in a way that unpacks the elements of the **rnbo~** patcher into understandable sections for those less experienced.

After you have installed the package, open a new patcher window and add a **rnbo~** object (just as we did with our first **gen~** patcher and then double-click on the **rnbo~** object to open the RNBO patcher window. This new patcher window looks similar to other Max patchers, but you will notice the addition of a page and arrow icon on the right-hand sidebar. Click on this icon to display the Export Sidebar.

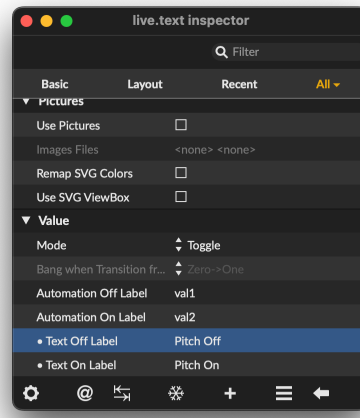


Figure 5-02

This is where we will export our patcher to a target. But first, we'll need to create the RNBO patcher.

The Basics

(Example Patcher: 5-1-Basics)

We'll start by building another tape-style looper using RNBO to help you get used to working within the RNBO patching environment.

Although RNBO objects very closely mimic their MSP and **gen~** counterparts, there are a few differences in the environments (many of these have to do with changes necessary to support export to the many RNBO targets). You'll recall that our first tape-style looper patch used the MSP **tapin~** and **tapout~** objects. At the time of this writing, the **tapin~/tapout~** pair of objects is not supported in **rnbo~** (it's currently not possible to implement the internal mechanisms that **tapin~/tapout~** takes use in MSP to manage feedback loops). We're going to use a delay/feedback loop in our patcher: the **feedback~** object in RNBO will introduce a single sample of delay in the loop, preventing it from spinning out of control. For the most part, Max has built-in protocols to avoid such out-of-control feedback loops. For example, if you tried connecting at **delay~** to itself in **rnbo~** without the **feedback~,** it just would not allow it.

Here is what the first looper patcher looks like:

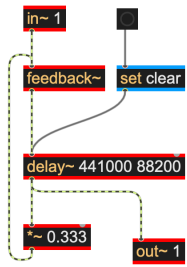


Figure 5-03

With these seven objects, we have created a working feedback loop that will form the basis of this patcher. Your experience with Max means it's pretty easy to see what's happening; **in~** and **out~** objects handle audio input and output from the **rnbo~** patcher.

The **feedback~** object adds a single sample of delay, allowing us to create a feedback loop using our **delay~** object. The **delay~** object includes two arguments that set (in samples) the maximum delay time available and the initial (default) amount of delay to use.

You'll also notice a new object you haven't seen before—a **set** object with an argument of **clear**. We're using this object because the RNBO environment (unlike MSP and Gen) only accepts *bang* messages or floating-point numbers, no symbol-based messages. To implement sending a *clear* message to the **delay~** object, we use the **set** object with an argument that describes the message we want to set, and then we use a button here that sends a *bang* message.

Finally, we've added a ***~** (multiply inputs) object that we use to lower the delayed signal's amplitude before sending it back into the **feedback~** object to prevent any volume explosions.

This loop is fully functional and will continue looping until you clear it, at which time it will start looping again. Next, we will add some controls to the looper.

Mixing options

(Example Patcher: 5-2-Mixing)

In this iteration of our **rnbo~** patcher, we have added a **gate~** object and three **param** objects. These objects work in the same way as their Gen counterparts. The **param** allows us to access messages from the **rnbo~** patcher's parent patch with arguments that set the message we will use to access this particular **param** object (e.g., **param *record*** and **param *feedback***). A **gate~** object works just as it does in the MSP and Gen worlds. In this patcher, the **gate~** works in conjunction with the **param record** object to open the **gate~** object, enabling audio input to the looper and allowing us to decide when and what to record.

You'll also notice that we can use arguments to set the initial state of our objects, and attributes **@min** and **@max** to set the range of numbers we want to work with. In this patcher, the **param record** object's default setting is off/0, with a range of 0 to 1, and the **param feedback** object's default is set to 1. with a specified range of 0. to 1.0. In this case, we want a 1 to open and a 0 to close the **gate~**.

The next **param** is named “clear.” This **param** will send a *bang* message to the **set clear** object. We do not have an argument in it, as when we send the message “reset,” it triggers a 0 out of the **param**, which the **trigger** (**t**) then uses to send a bang to the **set clear**.

The last **param** we added controls the volume level of the feedback loop. We limit this to a maximum of 1, allowing you to overdub while retaining 100% of the loop volume.

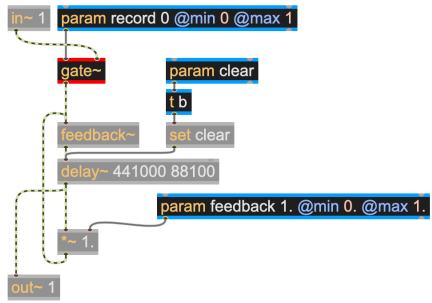


Figure 5-04

The **rnbo~** patcher accesses these **param** functions by sending messages from the parent Max patcher. The *reset* message is a generic message for all RNBO patchers that will reset all objects to their default state. The second (*record $1*) uses the replaceable argument $1 to send a 0 or 1 to the **param record 0 @min 0 @max 1** object, which opens or closes the **gate~** to enable audio input to the loop. The **param feedback 1. @min 0. @max 1.** allows us to send a *feedback $1* message to set the amount of signal preserved in the feedback loop; numbers less than one will create a fade-out effect.

Now we’ll turn to adding UI objects in the parent patch.

We have also added a few elements for mixing our dry audio and looped audio, including an on/off switch with the spacebar, and a “reset” message to initialize our Live UI objects to their default state.

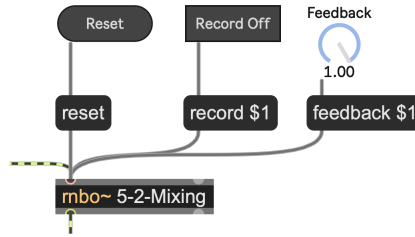

Reset
Record Off
Feedback
1.00
reset
record $1
feedback $1
rnbo~ 5-2-Mixing

Figure 5-05

We have also added a few elements for mixing our dry audio and looped audio, an on/off switch that uses the spacebar, and have the “reset” message to initialize our Live UI objects to their default state (set in the Inspector with “Initial Enable) with a **trigger** (**t**) “init.”

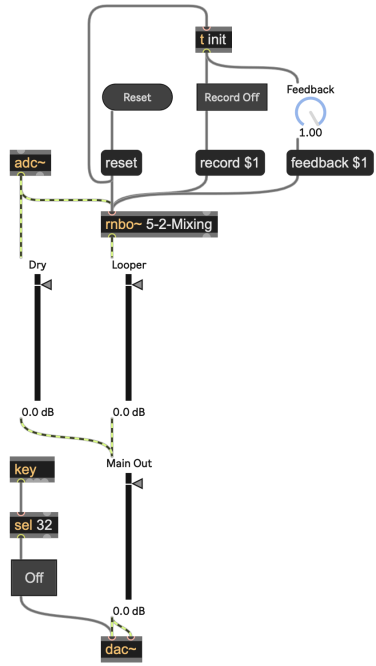


Figure 5-06

Setting the Length

(Example Patcher: 5-3-Length)

We have set the **delay~** to a maximum available duration with our first argument. Here we have a difference from our **tapin~/tapout~** pair, in that we can set the duration with a **timer** object and not clear the memory. We take the output of the timer (in milliseconds) and use a **mstosamps** (milliseconds-to-samples) object to convert the **timer** object's milliseconds into samples. For this patcher, we create a

multipurpose use of our record button. When we select record outside **rnbo~** (1) in addition to recording, it starts a **timer**. When we are done recording (0), it

1. Ends the **timer**
2. Outputs that value (milliseconds)
3. Converts the **timer** objects' milliseconds to samples and;
4. Uses that to set the length of the **delay~** (the loop).

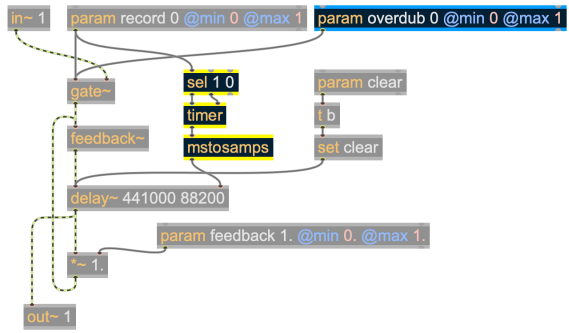


Figure 5-07

Since we are using our record **param** to set the length, we need to add an overdub **param** to open the **gate~** without retriggering the **timer** to avoid changing the loop's length every time we overdub.

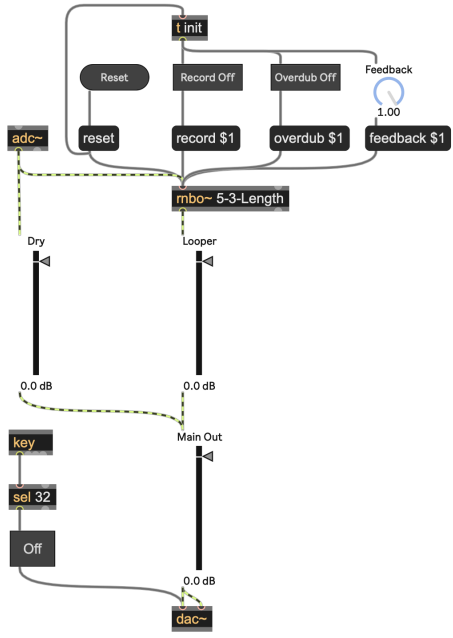


Figure 5-08

Processing Audio in the Loop

(Example Patcher: 5-4-Processing)

This section is very similar to the earlier chapter on processing in the loop with patcher 1-4-Processing. We will take a signal into a **gate~**, and then decide which outlet it goes to: the first outlet bypasses the audio effect (processing), and the second outlet sends it to the audio effect.

In the image below, you can see audio going into the right inlet of the **gate~**. The **param** for lowpass tells the **gate~** to either send it to the left outlet, bypassing the processing, or the right outlet with the **onepole~** to

lowpass the audio. Whichever is selected goes to the next gate, which repeats the process.

In addition to the **param** that chooses to bypass or process the signal, we have also added a **param** to set the value of the lowpass filter.

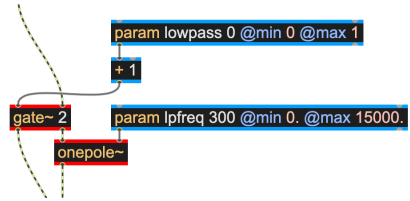


Figure 5-09

For this example, the next gate goes into a **freqshift~**, a type of ring modulator. (Remember, all these processing effects can be repurposed wherever desired.)

We have a **gate~** with **param** to bypass or process, as well as another **param** to set the **freqshift~** frequency.

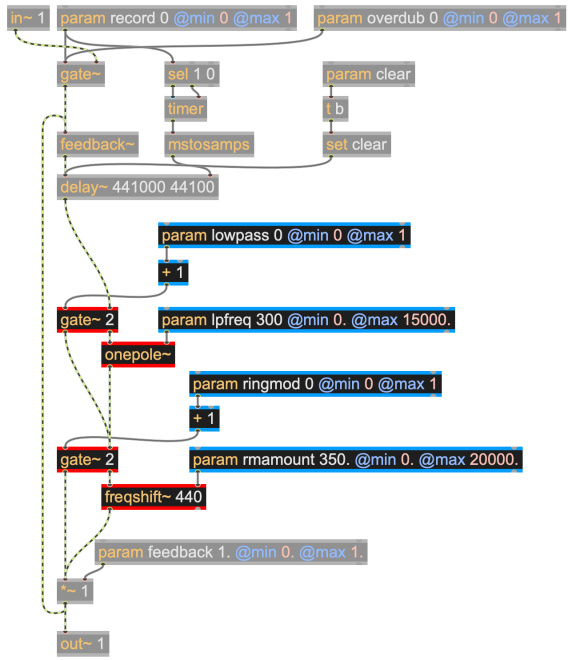


Figure 5-10

And these new **param** are accessed via additional messages from the parent patch what contains the **rnbo~** object:

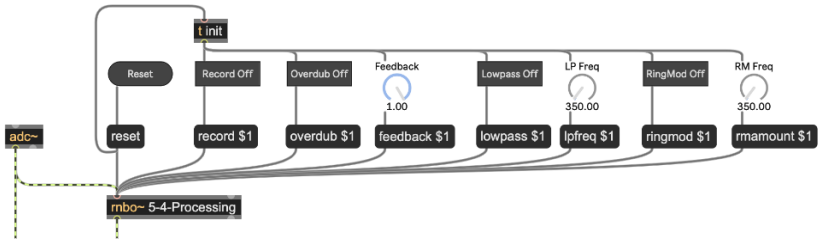


Figure 5-11

Creating a UI for Performance

(Example Patcher: 5-5-Performance-UI)

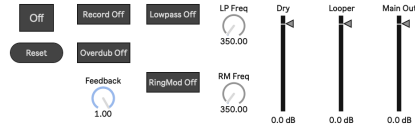


Figure 5-12

For the final UI, we have added our Live UI objects into Presentation Mode and organized them in a helpful manner. You can move them around as you desire.

As before, the outline around the objects shows that they are added to Presentation Mode.

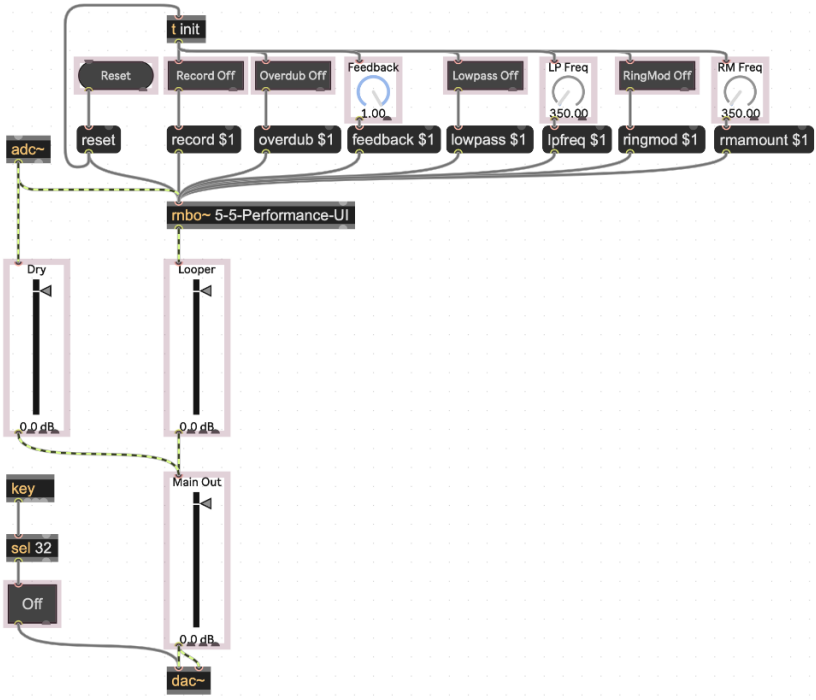


Figure 5-13

Chapter 6:
Multitrack Looper (rnbo~)

This **rnbo~** patcher is quite similar to our Multitrack Looper in Max. We have put it all together in **rnbo~** to leverage **rnbo~**'s ability to export the patcher to different targets. To achieve this, we have also done a few different things than in our other **rnbo~** chapters, for example, instead of using external messages at the top level of our Max patch, we have tried to include all but our Live UI objects within the **rnbo~** patcher. The main difference you will notice is the RNBO **set** operator, which is used when we want to send parameter values and attribute settings, to **gen~** subpatchers or parameters to a **rnbo~** object.

The Basics: Reset, Record, Play, Stop

(Example Patcher: 6-1-Reset-Record-Play-Stop)

The three RNBO operators at the core in this patcher are **record~** (for recording), **groove~** (for playing/looping), and **data** for storing audio data.

Note: For this chapter, and the RNBO-based looper we will build in the next chapter, we use a **data** operator with an attribute **@external 0** to make it local to the **rnbo~** patcher instead of a **buffer~** operator. The **data** and **buffer~** operators are similar, but some differences exist. We recommend reading their shared helpfile to see how they differ.

As with the multitrack looper we created in Chapter 2, we'll need to include patching that supports the basic elements we'll need for this looper:

- The ability to record and store our loop
- The ability to play a stored loop back
- The ability to stop recording and enable playback
- The ability to reset our looper when needed.

Our patcher includes the expected **in~**/**out~** operators for audio, as well as an **in** operator to handle input from our UI in the parent patch. You'll note that the **groove~**, **record~**, and **data** operators all share a name so they can work together. We have enabled looping for the **groove~** operator using the **@loop 1** attribute. In addition, we have set a maximum memory size as an argument for our **data** operator and added the attribute **@external 0** to make its contents local, i.e., invisible outside of the **rnbo~** patcher.

Moving all the commands associated with controlling our looper into values or settings inside the **rnbo~** object will make our patcher more visually complex in its connections. However, if we break it down into a sequence of operations and use a **trigger** (**t**) to help arrange the messaging and order of the messages, we can create a clearer version of our patcher.

For "reset," we send the following messages in order (reading the **trigger** operator from right to left:

1. Send a *bang* message to the **set** operator to stop playback with **groove~**
2. Send a 0 to the **record~** operator to cease recording
3. Send a *bang* message to the **onebang** operator to reset the timer operator
4. Send a *bang* message to the **set** to clear the contents of the **data** operator
5. Send 1 to the **sig~ 1** operator (this makes sure the speed of playback is normal)

For “record,” we send the following messages in order :

1. Send a *bang* message to the **set** operator to stop playback with **groove~**
2. Send a *bang* message to the **set** to clear the contents of the **data** operator
3. Send a *bang* message to enable the **timer** operator
4. Send 1 to the **sig~ 1~** operator to enable recording

For “play,” we send the following messages in order :

1. Send 0 to the **record~** operator to turn off recording
2. Send a *bang* message to the **timer** operator to stop timing, and send the **timer** operator’s output value to set the endpoint of the **groove~** operator
3. Send a *bang* message to the **groove~** operator to start playback

To “stop” we:

1. Send a *bang* message to the **set stop** operator to stop playback with **groove~**

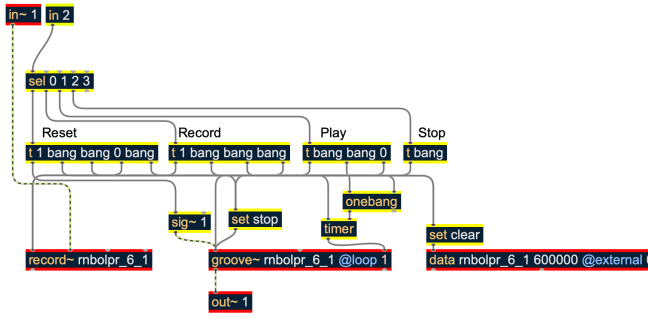


Figure 6-01

Although this reorganization of the **rnbo~** patcher may look a little more complex than the patcher in our previous chapter, these changes let us create a much cleaner interface in our parent patcher, which uses a single **live.tab** object to control the **rnbo~** patcher.

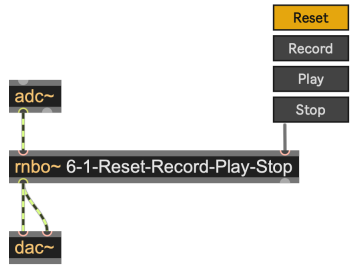


Figure 6-02

Changing playback speed and direction

(Example Patcher: 6-2-Speed-Direction)

As we did when patching our multitrack Looper in Max in an earlier chapter, we'll add speed and direction control to our RNBO looper using the groove~ operator; all we need to do is to change the value we send to the sig~ operator connected to the first inlet of the groove~ operator. Values between 0. and 1.0 slow down the playback, a value of 0. Stops playback, numbers greater than 1.0 speed up playback, and negative values function similarly but with playback reversed. To support changing the playback speed, we added an in 3 operator inside the **rnbo~** patcher whose value passes through a pack f 150 operator, setting the timing for the line operator to smooth values out before changing the value of the sig~ operator.

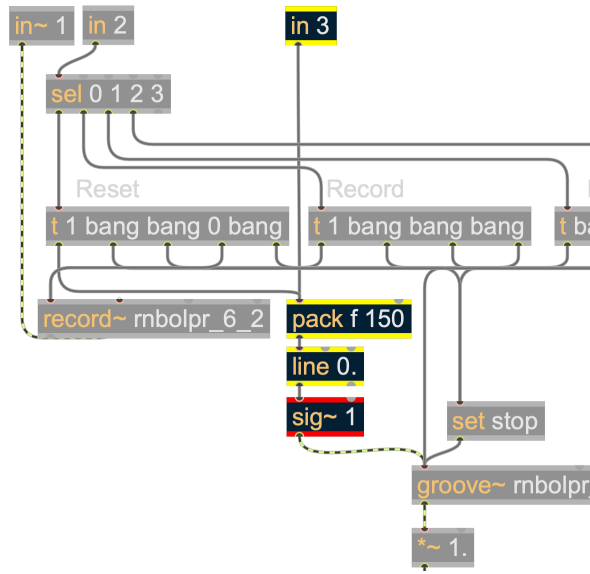


Figure 6-03

We control the playback rate in the parent patcher by adding a **live.dial** operator whose range (-5.0 - 5.0) and initial value (1) are set using the **live.dial** operator's Inspector.

To ensure that resetting our patcher will reset the playback speed to 1.0, we have added a **sel 0** and **t init** object pair.

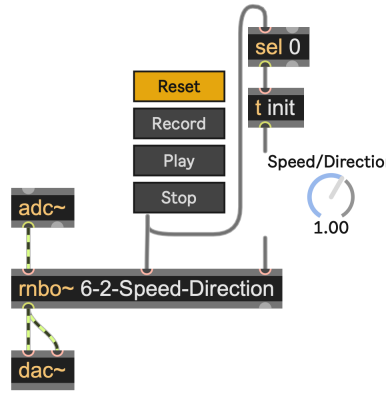


Figure 6-04

Declicking

(Example Patcher: 6-3-Declicking)

As we did when patching our multitrack Looper in Max earlier on, we will again use the RNBO **trapezoid~** operator for declicking our output. As with the MSP version of the groove~ object, the rightmost outlet of the **groove~** operator sends out a signal to synchronize other objects. We will use that output with the **trapezoid~** operator to create a ramp-up at the beginning and a ramp-down at the end of the loop playback, shaping the amplitude output of our looper. (Later in this chapter, we will use the same sync signal to align the playback of *multiple* tracks.)

We'll add two **in** and **sig~** object pairs to our RNBO patcher to allow us to set the values of the **trapezoid~** operator externally.

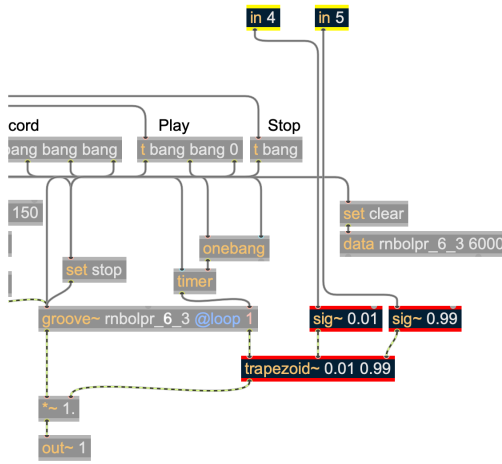


Figure 6-05

In the parent Max patcher, we'll add two **live.dial** objects to control the up and down ramps, and connect the output of the **t init** object so that resetting our looper will also return the up and down ramp values to their default settings.

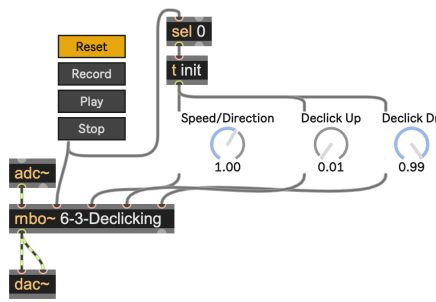


Figure 6-06

So far….

(Example Patcher: 6-4-So-Far)

We have now created a single-track looper in **rnbo~** with record and playback capabilities. To extend our patcher's functionality, we can create multiple instances of this looper patcher and use them as *asynchronous* loopers, offering new and interesting musical possibilities.

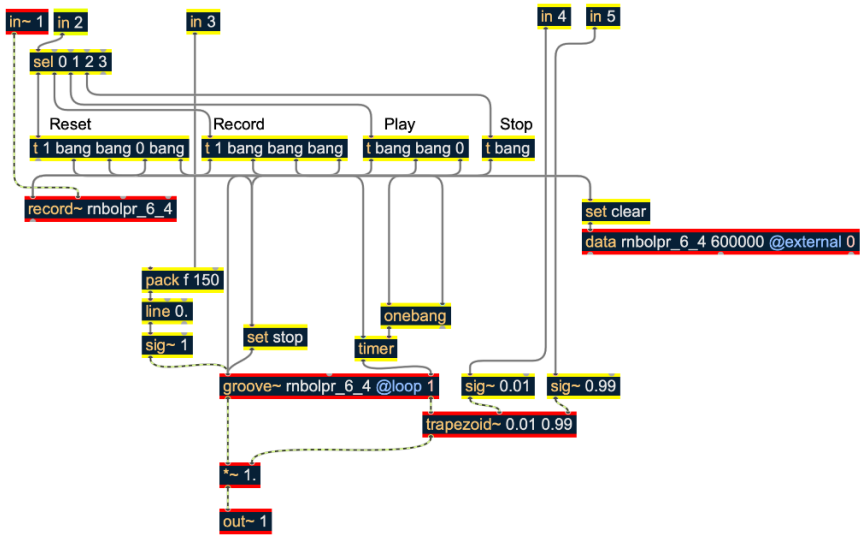


Figure 6-07

How might we modify our patching to create a multitrack asynchronous looper in rnbo~?

To create our multitrack asynchronous looper, we'll use the rnbo~ equivalent of a technique you'll recognize from the Max world — we'll create a series of *abstractions* (operato\atchers that will load as subpatchers into our **rnbo~** patcher) that share information to allow them to synchronize.

Using Abstractions and Synchronization

(Example Patcher: 6-5-Abstraction)

To create our patcher abstractions, we'll begin by making a few changes to the **rnbo~** patcher we have just created— a simple change to the **record~**, **groove~**, and **data** operators, as shown.

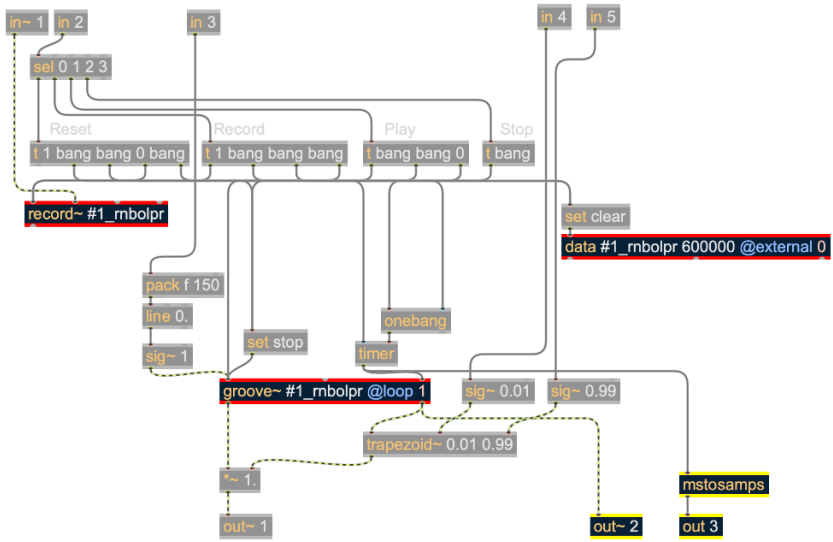


Figure 6-08

We'll change the first argument to all three operators to include the character string "#1_" in the names of the three operators. As we will see in a moment, this name change will allow us to add a unique name when we insert the resulting abstraction in a subpatcher using an *@args* attribute. More on this in a moment.

In addition, we have added an **out~ 2** operator for the sync outlet from the **groove~** operator and an **out 3** operator for the endpoint of our loop. We'll use both outputs to hand off to our secondary abstractions, aligning the playback location and loop endpoints.

We'll save this file with the name *6-5-Abstraction.rnbopat* from within our **rnbo~** patcher. (Since we have saved it in the same folder as our patcher, loading it as an abstraction is easy.)

Now that we've saved our abstraction, we can add a new object **p *@file 6-5-Abstraction* <name>** to our patcher, where <name> is unique (we'll use Peter, Paul, and Mary for our names here). This will load the file into a **rnbo~** subpatcher as an abstraction with the *@args* adding the name "Peter" to our **groove~**, **record~**, and **data** operators. We can do this as many times as our computer allows, with each uniquely named version sharing its own **groove~**, **record~**, and **data** operators.

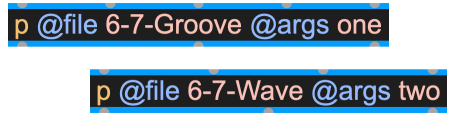


Figure 6-09

You'll notice that our abstractions have the same configuration of outlets and inlets. If we double-click them in a locked patcher, they will open and display the patcher.

This is a great way to create reusable patchers. Editing any abstractions will change the contents of all the other abstractions with the same name.

Secondary Loopers Using wave~

(Example Patcher: 6-6-Wave)

Our multichannel looper requires that we use one of our abstractions as the primary looper and synchronize any other loopers we add. These secondary loopers differ from our primary looper in that they use the **wave~** operator for playback, which facilitates working with endpoint and sync information from our primary patch. The challenging part of this patching lies in getting the **record~** to start at the beginning of the loop and end with the shared endpoint, which keeps everything in alignment.

The core elements in the patcher below are similar to those we have used before, but the differences lie in the message ordering. Here is what is happening with reset, arm/record, play, and stop sequences:

For “reset,” we send the following messages in order (reading from right to left in the **trigger** operator:

1. Close the second **gate** operator used for arming/recording (more on this later)
2. Close the first **gate** used for arming/recording
3. Close the **selector~** operator that sends the sync signal to our **wave~** operator for playback
4. Disable **record~**
5. Send a *bang* message to the **set** to clear the contents of the **data** operator

Setting arming/recording requires a somewhat more complex set of ordered messages:

1. Open a gate which
    a. Waits for and receives a *bang* message at the beginning of the loop (from the combo of **delta~**, **<=~**, and **edge~** operators), this *bang* message then uses a trigger (**t**) object to;
        i. Turn off the **selector~** for playback
        ii. Send a *bang* message to a **set** object, which clears the **data** object’s contents
        iii. Send a value of 1 to turn on a “Recording On” notification **and** enable recording.
        iv. Open the next gate
        v. Close the current gate

b. When the next gate is opened, it waits for and receives the *bang* message at the beginning of the loop, using a trigger (**t)** object to:

   i. Send a value of 2 to the main UI to set its display to Play

   ii. Send a value of 0 to turn off the “Recording On” button in the display **and** turn off recording

   iii. Open the **selector~** object so that the **wave~** object starts playback

   iv. Close itself.

See, easy! Well, a bit complicated. Let’s put it together in our next patcher.

Synchronizing Secondary Loopers

(Example Patcher: 6-7-Sync)

For this patch, we have indicated primary and secondary looper abstractions by name: the main abstraction with **groove~** (saved as 6-7-Groove), and the secondary **wave~**-based abstraction (saved as 6-7-Wave). They are loaded into the main patcher as **rnbo~** subpatchers and given unique names with the @*args* argument. We take sync and endpoint information from the main into the secondary abstraction to manage the connections between the primary and secondary loopers.

We have also added an **in~ 1** object for the audio that goes to both abstractions and other inputs required for each function.

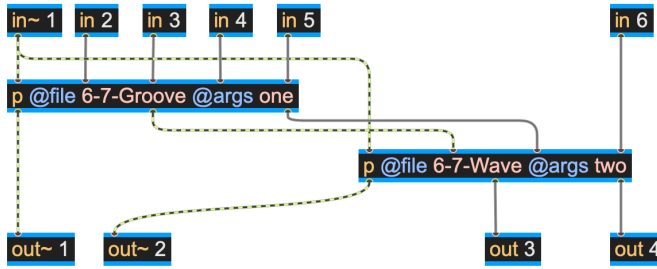


Figure 6-10

The result has been saved and reloaded as the **rnbo~ 6-7-Sync** object. Here is what it looks like in the parent patcher:

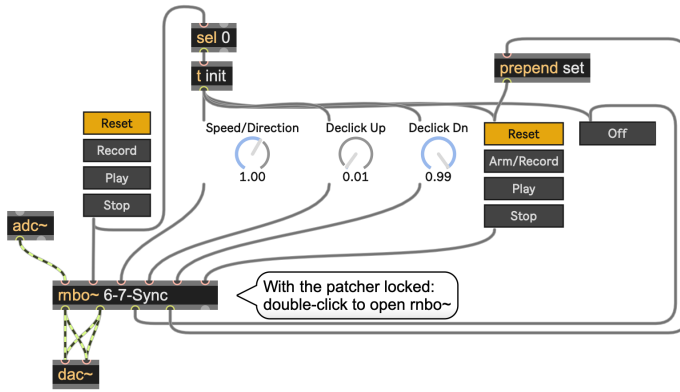


Figure 6-11

Adding a performance UI

(Example Patcher: 6-8-Performance-UI)

One of the best reasons to use abstractions in your patching is that they create an easy way to expand the resources used in your final patcher. For the final patcher in this chapter, we will make a four-track Multitrack

Looper. However, using abstractions for the secondary loopers means that it is easy to expand the patch for as many secondary loopers as you'd like—each of which can be spatialized differently, have different audio effects, mixes, and so on.

A **rnbo~** four-track looper patcher looks like this:

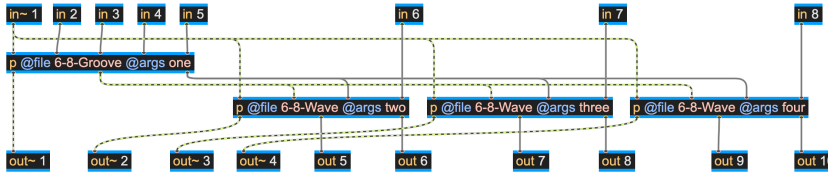


Figure 6-12

You'll note that we have assigned unique names to each of the secondary looper patchers and added the necessary **in~** and **out~** operators. The result has been saved and reloaded as the **rnbo~ 6-8-Performance-UI** object.

Here is what it looks like in the parent patcher. (While the UI connections appear a little busy, imagine how much more complex this would be if we were handling all the messaging at this level of the patcher!)

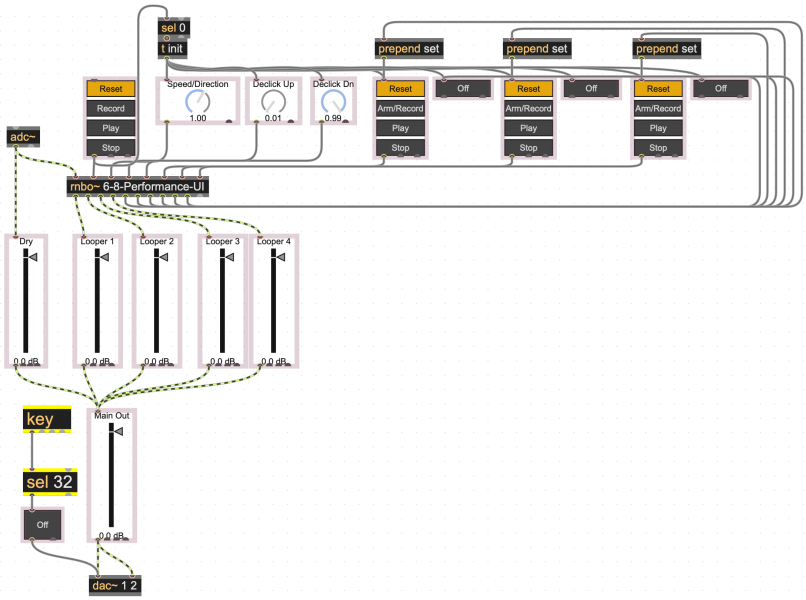


Figure 6-13

As with our earlier patchers, you'll notice the telltale salmon-colored outline that indicates that the Live UI objects have been added to Presentation Mode. Here's a final layout that is organized in a meaningful way (you may have your own ideas for organizing your UI, of course).

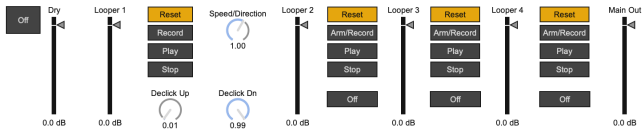


Figure 6-14

Chapter 7:
Basic Overdubbing Looper (rnbo~)

In this chapter, we recreate the basic overdubbing looper from Chapter 3 in the rnbo~ programming environment, adapting familiar recording, playback, and counter-based techniques to support overdubbing and performance-oriented workflows. As we've seen when creating our sequence of loopers and transitioning from the MSP to the **gen~** environment, we'll look at four interrelated parts of the patch that you'll recognize. Creating our RNBO patch will require revisiting those areas as we implement the new patch. We'll walk you through each of the areas to provide an overview of how our patcher needs to be modified. The overview is essential because, as before, recording/playing/overdubbing is never a single message, but multiple messages as you turn one section on and another off while starting or directing a counter to a new section.

The Basics: Record, Play, Stop, Reset

(Example Patcher: 7-1-Record-Play)

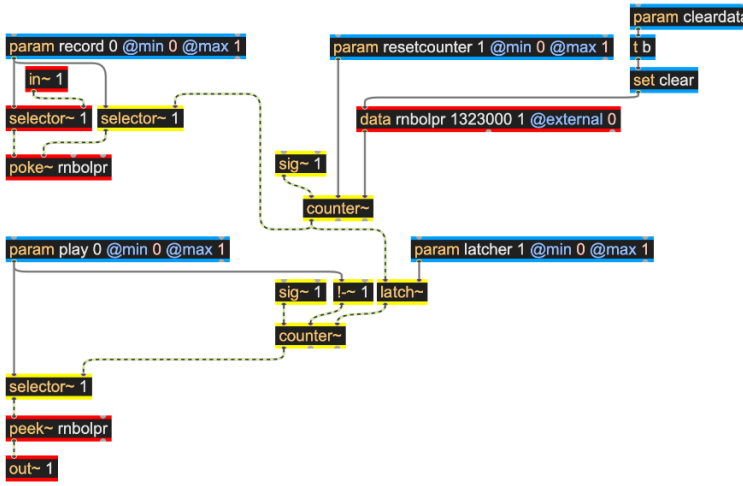


Figure 7-01

The core of this patcher is our **data** object, which strongly resembles the **gen~** operator of the same name (its relationship is made more evident by the fact that the help files for both the **buffer~** and the **data** operators are combined. We'll use the data object as we used the **data** operator in our **gen~** patching—as a purely internal (in this case, to the **rnbo~** patcher) place to store the recorded audio through adding the attribute *@external 0*. Since we only use it internally, the internal-only use is part of the exported code; should we export it to a target.

The arguments for our **data** operator include a name (rnbolpr), memory size in samples (1323000), and the number of channels (1). As in previous examples, we have added the patching used to clear the memory on the incoming message "reset" (**param cleardata)** using a **t** and **set** object with the message "clear."

The number of samples allocated to memory in **data** also sets the default length of the **counter~** operator, via the connection between the two.

The middle inlet of the **counter~** is used to tell it to reset the count to 0, and the first inlet has a **sig~** that instructs the **counter~** object to run at normal speed (1).

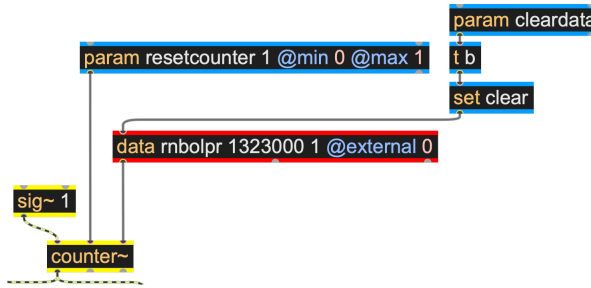


Figure 7-02

We use a **poke~** operator to write to our **data** object in the recording section of our patcher. Our **poke~** object has a single argument - a name associates it with its similarly named **data** operator**.** We have set up a **param** object named “record” with two functions: to open and close a **selector~** that lets audio in to be recorded, and also opens a **selector~** that lets in the **counter~** which tells **poke~** the location in our **data** object to write.

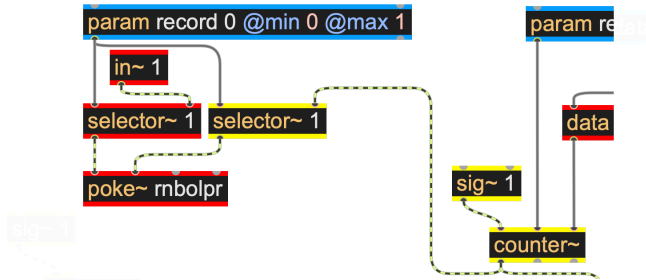


Figure 7-03

The playback portion of our patcher uses a **peek~** operator for playback (reading) from the **data** object. As with the **poke~** operator, our **peek~** object has a single argument—a name that associates it with the **data** operator of the same name. Our **param** named “play” opens a **selector~**

that sends a **counter~** to **peek~** telling it what location to read. This **counter~** is a second **counter~** that uses a **latch~** (controlled by the **param latcher** object), grabs the latching value from the **counter~** (to which it is connected) to set the end of the second **counter~** when we enable playback and turn off recording. The second latched **counter~** provides how audio stored in the **data** object loops as it counts to the end of the latched value and starts again from 0. Every time you press play, it also resets that **counter~** (using the middle inlet) to begin reading from 0. The **sig~** operator's argument of 1 ensures that the **counter~** object runs at normal speed.

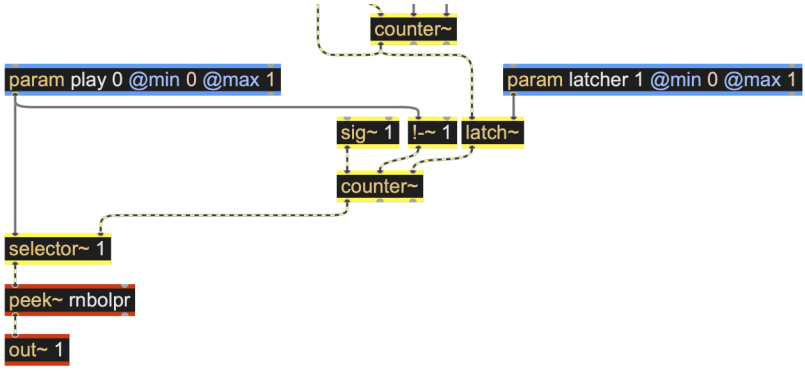


Figure 7-04

With this overview in mind, we see that many of our patcher's basic functions (reset, record, play, stop) require a sequence of multiple messages.

The parent patcher shows us that sequence of multiple messages, each contained in a message box for the record, play, and stop functions:

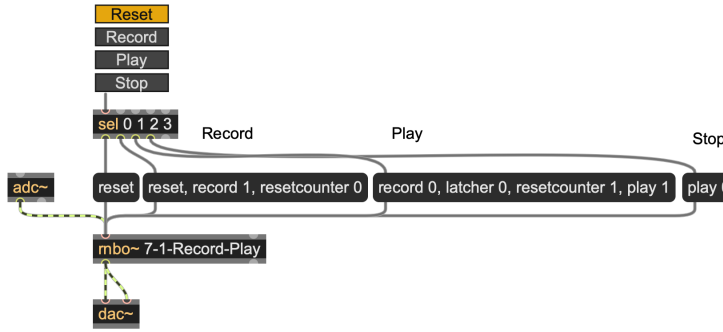


Figure 7-05

- For resetting, we send a generic *reset* message that sets the entire contents of our **rnbo~** patcher to their default state.
- To record, we send the messages “reset,” then “record 1” to turn on recording, and “resetcounter 0” to have it start counting from 0.
- For playback, we turn off recording, latch the loop's endpoint, reset the counter, and enable playback.
- To stop, we simply close the **selector~** so that no **counter** is sent to **peek~**.

Overdubbing

(Example Patcher: 7-2-Overdub)

To add overdubbing to our patcher, we’ll need to allow recording to work with the new latched **counter~** that sets our loop. We’ll need to add a **selector~** operator with two inputs to do this.

The first input is our main **counter~**, whose duration is specified by the connection to the **data** operator. This length sets the maximum possible duration for our initial recording.

The second input is our latched **counter,** which we'll use for our loop playback. **Param play** controls, which **counter~** is used via the **selector**. When overdubbing, we will also select the loop playback **counter~** to instruct the **poke~** operator on the location to record the overdub.

We will also add an **@overdub accum** attribute to our **poke~** operator. The attribute sets the **poke~** operator to add the new recording to the previously recorded content. We've also added **param mix 1. @min 0 @max 1.** object to set how much of the previous audio to preserve when overdubbing: 1. meaning that 100 percent of the previous audio recorded is preserved.

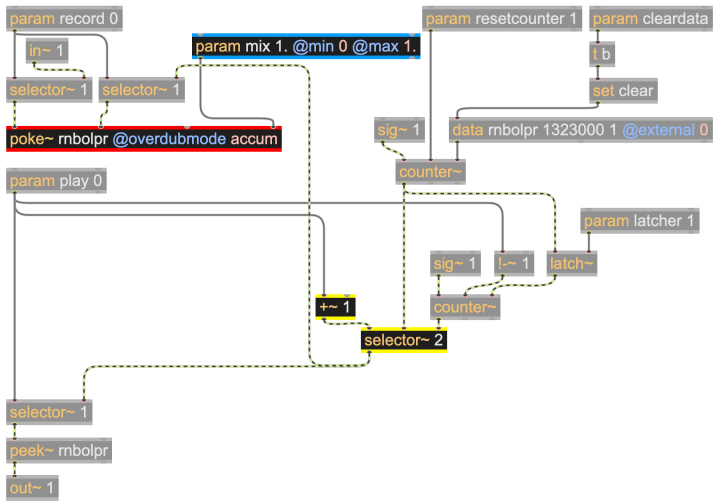


Figure 7-06

In our parent overdubbing patcher, we have added a "record" message and a "mix" message.

As in earlier patchers, when reset is selected, it triggers an “init” message to our Live UI objects.

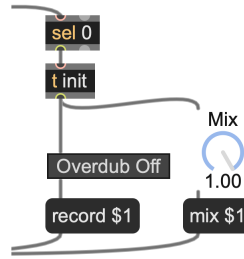


Figure 7-07

Declicking

(Example Patcher: 7-3-Declicking)

To enable declicking at the beginning and end of a loop, we again use the **trapezoid~** operator to provide a signal ramp at the loop’s start and end. In this example, it uses the playback (loop) **counter~** which is then scaled to phase (0. to 1.) to drive our ***~** (multiple inputs) that create the ramp. The **scale~**, **trapezoid~,** and ***~** objects are added after the **peek~** and before the **out~** operator. Finally, we’ll add two **param** objects to set how much ramp is used. To turn off declicking, set these values to 0. and 1.

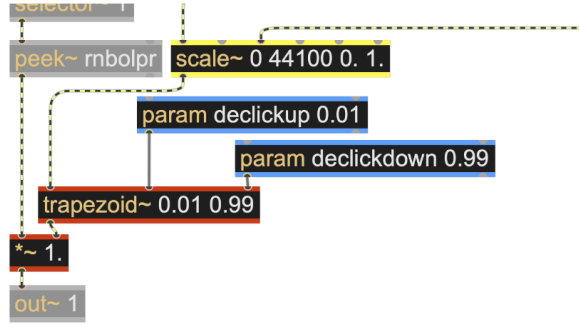


Figure 7-08

In our parent patcher, we have added two messages to control the new **param** operators**.**

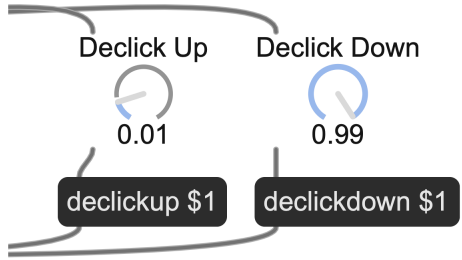


Figure 7-09

Speed and Direction

(Example Patcher: 7-4-Speed-Direction)

To add the control of speed and direction of our overdubbing, all we need to do is change the **sig~** operator that controls the speed of the latched **counter~** and get the same **counter~** operator to count backward from the latched value.

We have added a **param** operator named “speed” to control the value of the **sig~** going into our latched counter.

- A value of 1 is normal speed.
- A value less than 1 slows the playback.
- A value of 0 stops the playback.
- A value greater than 1 speeds up the playback.

One thing to keep in mind with speed changes: overdubbing is unpredictable and works best at normal speed.

For reverse, we use the **!-~** to subtract the incoming counter value from the latched value, making it count backward.

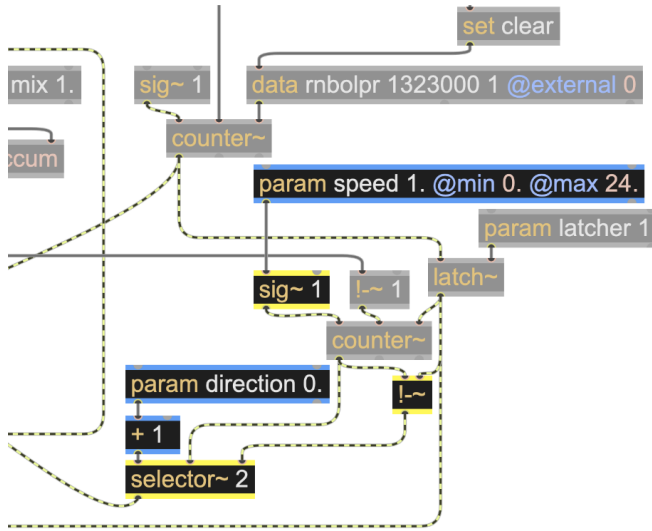


Figure 7-10

The trio of **param direction 0.**, **+ 1**, and **selector~ 2** operators are used to set the forward or reverse speed to the **counter~** operator that sets the speed (the **counter~** operator's **selector~** operator input is also used for overdubbing). Note that attaching the **!-~** operator is a handy way to set a negative value for the output of the **counter~** operator.

Here's how the updated speed and direction patching fits into our overall patcher:

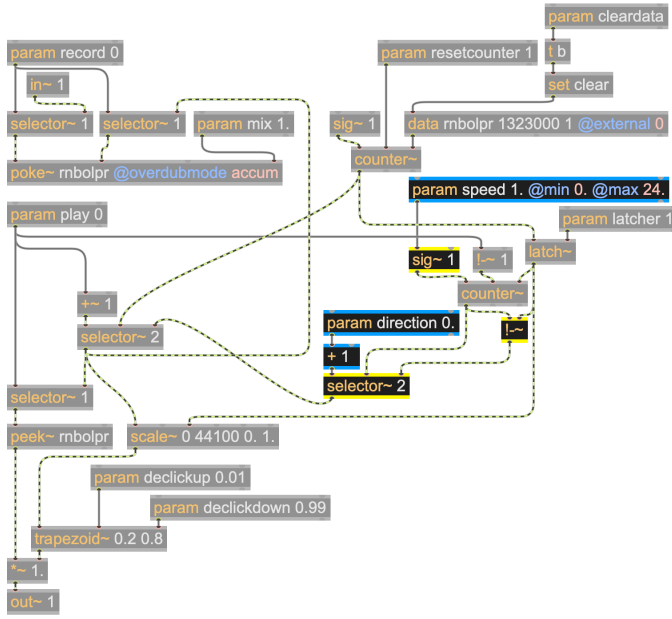


Figure 7-11

Now, let's add the UI changes to our parent patcher to support the new speed and direction features:

We'll add a **live.tab** object configured as a button and a message box object to send a *direction 0/1* message into our **rnbo~** object. To provide a UI for speed control, we'll use a **live.dial** object, a **message** box (*$1 150*), and a **line 0**. object to construct a set of messages that smooth out speed changes to minimize the stairstep/ripping effect we'd typically hear (if you like that sort of thing, just remove the **message** box and **line** objects!) the

And another message box (*speed $1*) to set the speed of our recording and playback.

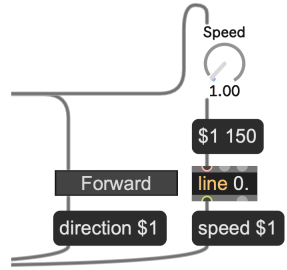


Figure 7-12

Creating a Performance UI

(Example Patcher: 7-5-Performance-UI)

Here's the final version of the parent patch. As before, you'll notice the Live UI objects that were added to Presentation Mode by their salmon-colored outlines.

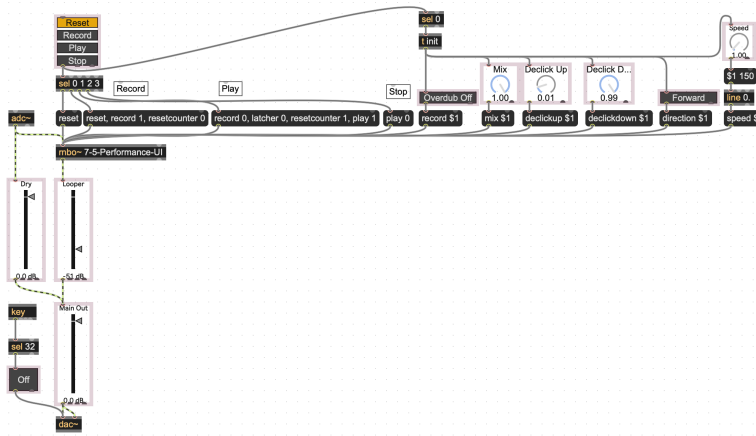


Figure 7-13

Finally, we've organized our UI in a meaningful way, and you can move them around as you'd like.



Figure 7-14