

Max 9 User Guide

Version 9.2.0-rev.1

Cycling ’74

cycling74.com

Contents

Intro **7**
- User Guide 8
- What's New in Max 9 10
- Glossary of Common Terminology 13

Audio **22**
- Ableton DSP 23
- Frequency Domain Processing 25
- MC 33
- MC and Gen 34
- MC Wrapper 35
- Multi-Channel Audio I/O 36
- Non-real-time Processing 41
- Audio Plugins 45
- Polyphony 51
- Recording and Exporting 63
- RNBO 69
- Sample Accurate Messages 71

Colors **74**
- Color Palette 75
- Color Themes 85
- Dynamic Colors 91
- Format Palette 95
- Styles 98
- Syntax Coloring 109

Data **114**
- Arrays 115
- Dictionaries 121
- Integers and Floats 129
- Strings 132

Debugging **138**

Debugging and Probing 139
Error Messages 147
Illustration Mode 152
Max Console 155

Files **164**
File Browser 165
File Types 171
Search Path 177

Gen **183**
Gen 184
Gen Common Operators 213
GenExpr 220
gen~ Operators 239
Jitter Operators 246

Jitter **249**
Depth Testing and Layering 250
Geometry Objects 255
Graphics Engine 263
Graphics Processing 265
Jitter expr 280
The JXS File Format 291
Jitter Matrix 308
Render Passes 332
Textures 341
Video 349
Video Engine 356

Max Interface **359**
Action Menu 360
Documentation Window 365
Extras Menu 376
Inspector 378
Object Reference 386
Preferences 401
Search 416
Patcher Toolbars 420

MIDI **455**
Mapping 456
MIDI 455

Parameters **471**
OpenSoundControl 472
Connecting Parameters 474
Parameter Mode 479
Presets and Interpolation 491
Saving State with pattr 498
Snapshots 508

Patching **518**
Conversion Cheat Sheet 519
Messages 541
Message Types 548
Objects 555
Patcher Lifecycle 567
Patching 518
Patch Cords 593
Patching Mechanics 601
Web Browser and jweb 604

Reuse and Organization **616**
Abstractions 617
Using bpatchers 624
Custom UI Objects 630
Externals 645
Packages 649
Package Manager 658
Projects 664
Prototypes 671
Snippets 674
Subpatchers and Encapsulation 678
Templates 682

Scripting **685**
Scripting 686
External Text Editor 689

JavaScript 692
jit.gl.lua 705
The define Message 724
Controlling Max with Messages 728
REPL 770

Sharing 782
Sharing Patchers 782
Projects 664
Standalones and Collectives 795

Timing 807
Scheduler and Priority 808
Time Value Syntax 817
Transport 821

Max for Live 825
Max for Live 826
Live API Overview 830
Creating Max for Live Devices 842
User Interfaces in Max for Live 844
Automation 846
Sharing Max for Live Devices 848
Timing and Synchronization in Max for Live 849

Max for Live Extended 850
Creating Audio Effect Devices 851
Creating Devices that use the Live API 853
Creating MIDI Effects 869
Device Parameters in Max for Live 871
Freezing Max for Live Devices 878
Max for Live Limitations 880
Max for Live MIDI Tools 884
Presets 902
Preview Mode 904
Resolving Conflicts in Frozen Devices 906
The Parameters Window 908
Unfreezing Devices 913
Using pattr in Live Devices 915
Using Symbols in Max for Live 917

Working With Files in Max for Live 918

Appendix A: MC Extended **920**
Gen Features for MC 921
Generating Values for All MC Wrapper Instances 923
MC and Max for Live 928
MC Channel Topology 929
MC Dynamic Routing 933
MC Event Objects 940
Multichannel Function Generators 941
MC Gen Instances 946
MC Gen Operators 948
MC Managed Polyphony 949
MC Mixing and Panning 955
MC Patch Cords 965
MC Polyphony 967
MC Recording and Playback 969
MC Signal Manipulation Objects 971
MC Spatialization 975
MC Visualization and Probing 977
MC vs MCS Objects 980
Messages to the MC Wrapper 983
Multichannel Delay Systems 988
Polyphony Using mc.poly~ 991
Polyphony with Multiple Patchers 994
Processing Events from MC Objects 995
Using mc.gen with the MC Wrapper 998
Using Plug-ins with MC 1001

Appendix B: Lua Extended **1006**
jit.gl.lua Color Bindings 1007
jit.gl.lua OpenGL Bindings 1019
jit.gl.lua OpenGL GLU Bindings 1051
jit.gl.lua Vector Math 1053

Credits **1067**

Intro

User Guide

New in Max 9 8
Demos 8
Tutorials 9
API Reference 9

Welcome to the Max user guide. This guide covers everything to do with how Max works, including the fundamentals of objects, messages, and patchers; the different parts of the Max interface; and how to do things like process sound and work with graphics.

> If you're new to Max, you might find it useful to read how to use this documentation
>
> This documentation is also available in PDF format, for viewing offine.

New in Max 9

If you're a long-time Max user and you want to see what's new in Max 9, check out this overview.

Demos

If you want to see some of what you can do with Max, check out these demos:

Name	Description
JavaScript Codebox	Mix textual and visual programming with inline codebox supporting modern JavaScript

Jitter Geometry	Half-edge structures for shapes that you can morph and distort
Ableton DSP	Use Ableton DSP algorithms for synthesis, modulation, and effects

Tutorials

These tutorials can guide you through working with Max step-by-step.

- **Working with Max** - The basics of working with Max, including setting up timers and events, building a user interface, handling input from the webcam, and generating sound.
- **Signal Processing** - How to generate and manipulate audio signals, extacting automation from real-time audio signals, and working with samples.
- **Video Processing** - Setting up a graphics environment, making dynamic visualizations, and generating audio-reactive effects.
- **Jitter Geometry** - Using the jit.geom objects, introduced in Max 9, to manipulate dynamic geometry using the half-edge structure.

If you want to get inspired with even more Max examples, check out our **Examples Gallery**

API Reference

Max exposes several APIs—Application Programming Interfaces—that let you use code to control various systems in Max. You can write short scripts to automate simple tasks, or large programs that completely change the way Max behaves. Some of the APIs that Max offers:

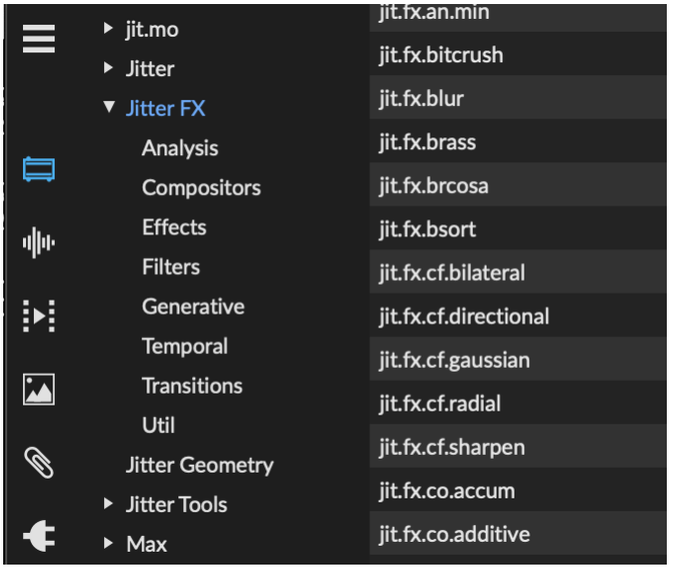
- **JavaScript API**: Use the v8, v8ui and v8.codebox to embed JavaScript in a Max patch. Define custom objects, programmatically create objects and patch cords, and operate the Max application.
- **Live Object Model**: Use Max objects like live.object and live.path to read and modify the state of Ableton Live from within a Max for Live device. Also accessible from the JavaScript API.
- **Node for Max**: Use the node.script and node.debug objects to launch custom Node.js scripts from Max. Send Max messages to a running Node process and fetch a result.

What's New in Max 9

Jitter 10
Audio 10
JavaScript and Coding 11
Patching Enhancements 11
User Interaction and Interface Tools 11

Jitter

- Jitter Geometry — A specialized group of Jitter objects designed to work with **Half-edge Geometry Structures**. This makes geometry manipulations like redistributing and adding vertices much more efficient.
- jit.fx - Tons of Jitter effects packaged as `jit.fx` objects, giving you access to effects, compositing, transitions, and analysis shaders. Each one has a help file and reference page.



- jit.ui — New Jitter UI objects for building interfaces and "heads-up displays".

Audio

- ABL Objects — These objects offer a patchable interface to the internal workings of some of Ableton Live Suite's most popular devices.
- loudness~ — Reports the loudness of a signal according to the EBU R 128 standard
- jweb~ — Load a web browser inside a patcher, capture its audio, and route the audio output into Max.
- Global Record — Record the output of a patcher with a single click, from the patcher toolbar.

JavaScript and Coding

- JavaScript V8 — Modern JavaScript engine, supporting JavaScript classes, the spread operator, destructuring assignment, typed arrays, and much more.
- Codebox — top-level codebox objects like v8.codebox for JavaScript and node.codebox for node.script.
- REPL — Read-evaluate-print-loop interface for controlling Max using text entry

Patching Enhancements

- Patcher List View — See all the objects in your patch in a hierarchical list.
- Illustration Mode — Understand how your patcher works by running it in slow motion.
- New Preferences Window — Updated, tabbed UI for Max preferences
- Syntax Coloring — Increased readability for object boxes with colored text for object names, arguments, attributes, and attribute arguments.
- New Documentation — Documentation rewritten and modernized.

User Interaction and Interface Tools

- Integrated OSC — Controll Max with any OSC-compatible device or application, and send OSC messages to control other OSC-enabled systems.
- hid — Modern human interface device input object
- Param Connect — Connect parameter-enabled UI objects to supported objects without

patch cords

- poly~ param Object — Named parameters in poly~ subpatchers
- Preset Interpolation — Blend between presets floats or the nodes object

Glossary of Common Terminology

#	13
A	14
B	14
C	15
D	15
E	15
F	15
G	15
H	16
I	16
J	16
L	16
M	17
N	17
O	18
P	18
Q	19
R	20
S	20
T	21

There are many terms used throughout the Max documentation that are unique to the application. Understanding what these terms refer to will help when learning the software. Below is a list of the most commonly used terms in Max, along with their respective definitions.

#

- **#** : The pound sign (#) is a special character that indicates a changeable argument specifically within an abstraction. Message boxes, object boxes, and some object attributes within an abstraction can be given a changeable argument by typing in a pound sign and a number (e.g. #1) as an argument. When the abstraction is used inside another patcher, an argument typed into the object box in the patcher replaces the # argument inside the abstraction. Using a zero with the pound sign (#0) at the beginning of a symbol argument

(e.g., #0_value), transforms that argument into an identifier that is unique to each instance of an abstraction (and its subpatchers) when it is loaded.

- **$** : The dollar sign ($) is a special character that indicates a changeable argument in a message box or an object. If used in a message box, $ should be followed by a number in the range 1-9 (such as $2). That argument will then be replaced by the corresponding argument in the incoming message. If used in an expr, if, sxformat, or vexpr object, $ must be followed immediately by the letter i, f, or s, which indicates whether the argument is to be replaced by an int, a float, or a symbol.
- --- : The three dash (---) identifier is used for coll, dict, and in the context of Max for Live. It serves as a way to create unique names for objects. In the context of Max for Live, if you want a named object to be unique to a device, use three dashes (---) to start the name of your buffer or send/receive destination.

A

- **Abstraction** : Any patch you have created and saved can be used as an object in another patch just by typing the filename of your patch into an object box as if it were an object name. Patches used in this way are called abstractions.
- **Argument** : Many objects have arguments which set the initial state of an object. Arguments are typed into the object box after the object's name. All possible arguments for an object are listed in their respective Object Reference Pages.
- **Attribute** : Many objects have attributes. An attribute is a setting or property that tells the object how to do its job. These can be set in a variety of ways which are outlined.
- **Attrui** : An attrui is an object which displays attribute values of the object it is connected to. If you connect an attrui to another object via a patch cord, it will intelligently display all attributes for that object via a dropdown menu and show their corresponding values when selected.

B

- **Bang** : Bang is a type of message that tells the receiving object to do whatever it is designed to do. It basically tells an object to "Go!". More information on bang messages can be found here.
- **BEAP** : The BEAP package is a collection of audio-processing modules that are automatically included with Max. Each module has a help file that explains its function.

C

- **Changeable Argument** : There are two special characters in Max that indicate changeable arguments: `#` and `$`. Please refer to their respective definitions for more information.
- **Collection** : A collection is a kind of virtual file folder where you can store patchers, media, saved searches, and any other kind of Max resources you would like to group together for convenience.
- **Console** : A window (Window >Max Console) that displays status information, error messages, and warnings related to your patch.
- **Contextual Menu** : A menu that appears when control-clicking (Mac) or right-clicking on the patcher background, an object, or a patch cord in an unlocked patch. This menu provides various options that are relevant to the overall patcher, the object selected, or the patch cord selected.

D

- **Device** : A device is a type of patch that is specifically designed to work within Ableton Live as an audio effect, MIDI effect, or instrument. Devices can also be used within Max. However, they are not compatible with other DAWs.
- **Dictionary** : A dictionary is a collection of key-value pairs that can be passed between objects. Each key in the dictionary is a unique symbol. Each value may be a number, symbol, list, array, or another dictionary.

E

- **Event** : An event is the passing of a non-signal message between two objects.
- **External** : A binary file that contains one or more Max objects.

F

- **Float** : A floating-point number, a.k.a. a number with a decimal point.

G

- **Gen** : Gen is an extension of the Max patching environment that converts what you build visually into efficient, compiled code as you go. Gen code can be used outside of Max with Code Export. Gen includes the gen~ and mc.gen~ objects for audio, and the jit.pix and jit.gen objects for matrix and texture processing.

H

- **Help File** : Every object in Max has a help file. Help Files are interactive patches that demonstrate how an object works. To access an object's help file you can either right-click (ctrl+click) on the object in an unlocked patch and select "Open Help", or click the "Open Help" button from the object's reference page.

I

- **Inlet** : A "port" at the top of an object where information is sent via a patch cord. Information is passed from the outlet of one object to the inlet of another object. Each inlet can receive a certain type of information. Hover over an inlet with your mouse in an unlocked patch to see what type of information it can receive.
- **Inspector** : There is both an Object Inspector and a Patcher Inspector. The Object Inspector is used to edit the attributes of Max objects. The Patcher Inspector is used to edit settings for a patcher window.
- **Integer** : A whole number, a.k.a. a number without a decimal point.

J

- **Jitter** : Jitter is the visual processing side of Max. All Jitter objects begin with "jit." For help getting started with Jitter, see this group of tutorials.

L

- **List** : A message with several numbers and/or symbols is called a list; the list is a mechanism for keeping data together into a single message that can be sent via patchcords to other objects. There is no inherent structure to a list; lists are organized simply as one item after the other.

- **Live Object Model (LOM)** : The LOM is used in conjunction with Ableton Live. It is essentially a roadmap to each of Live's parameters that are accessible via Max for Live. Using the LOM you can control everything in Live's API that is accessible to Max for Live.
- **Lock/Unlock** : A patch can either be locked or unlocked. You unlock a patch to edit it and you lock a patch to perform with it and adjust UI objects.

M

- **Matrix** : A matrix is a grid with each location in the grid containing some information. This term is specifically used in the context of Jitter, the visual processing side of Max. In Jitter, a matrix can have any number of dimensions from 1 to 32.
- **Max** : Max is the name of the application, but the word also stands for the data processing side of the software. For help getting started with Max-specific patching, see this list of tutorials.
- **Max for Live** : Max for Live is a separate application that is specifically designed to be integrated into Ableton Live. Max for Live allows you to use the Max development environment to create your own Live content. For more information on Max for Live see this page.
- **MC** : MC refers to a set of multichannel objects in Max. These objects allow for multiple channels of audio processing and pass multichannel audio signals using a single patchcord. MC objects typically have an mc. or mcs. prefix.
- **Message** : Max patches function by passing messages between objects. Messages tell objects what to do. For a list of what messages an object can receive, see the object's reference page.
- **Message Box** : A message box is a clickable object for storing, displaying, and passing messages between objects.
- **Method** : Another name for a message. See the "Message" entry above for further details.
- **MSP** : MSP is the audio processing side of Max. All MSP objects end with a "~". For help getting started with MSP, see this list of tutorials.

N

- **Node for Max** : Node for Max is a package, automatically included in Max. that lets you write custom applications using the Node JavaScript framework then control and

communicate with those applications from Max. For Node for Max documentation, open the Documentation Window in Max and look for the Node for Max package documentation.

- **Number** : Numbers refer to both floats and integers.

O

- **Object** : Objects are the building blocks of a patch – they perform specific tasks, and operate like miniature programs within the larger environment.
- **Object Action Menu** : The object action menu , accessible from the left side of every object, provides a quick way to access the help file, reference page, and object inspector for that object. It also provides sub-menus to discover messages and attributes associated with the object, and serves as an aid to Max patching.
- **Object Reference Page** : Every object in Max has its own reference page . This page serves as a manual for the object, explaining what the object does and how to control it with arguments, attributes, and messages.
- **Operate While Unlocked** : A mode that allows you to adjust user interface objects while patching. Sliders, dials, and other controls are all active in this mode which means you don't have to lock the patcher to operate them. To activate the Operate While Unlocked mode, click the "Enable Operate While Unlocked" button on the bottom patcher window toolbar.
- **Operator** : An operator is a Gen object.
- **Outlet** : A "port" at the bottom of an object where information is sent out. Information is passed from the outlet of one object to the inlet of another object via a patch cord. Each outlet sends a certain type of information. Hover over an outlet with your mouse in an unlocked patch to see what type of information is sent out.

P

- **Package** : Packages are a convenient way to bundle objects, media, patchers, and resources for distribution. A package is simply a folder adhering to a prescribed structure and placed in the 'packages' folder. Once placed in the correct location, the package contents are seamlessly integrated into Max.
- **Package Manager** : The Package Manager, which is accessed by selecting “Show Package Manager” from the File menu, provides access to a regularly updated, curated selection of Max add-on content. Some packages listed are created in-house, while others are created

by third-party developers. Typically, installing one of these packages will give you access to new objects, patchers, and other Max related content.

- **Parameter** : Parameters are a simple representation of the current state of that object, compatible with presets, pattr, and snapshots
- **Patch Cord** : Patch cords are used to connect objects together in Max. Information is passed from one object to another using a patch cord. There are four different types of patch cords — those used for Max objects, audio patch cords used for MSP objects, MC patch cords used specifically for multichannel objects, and Jitter patch cords. Each type of patch cord has a unique look.
- **Patcher** : A Max patcher is the graphical canvas that you move objects around on. A Max patch is the file or program that you create in Max. You create Max patches and you patch in Max.
- **Presentation Mode** : A different patcher view that allows you to arrange and resize user interface objects in your patch independently of their functional position and size in patching mode.Presentation Mode is typically used to design a user-friendly interface for a patch. When a patcher is in Presentation Mode, the Presentation Mode button in the bottom patcher window toolbar will turn green/yellow and the word (presentation) will appear in the title bar of the patcher window.
- **Probe** : Probing allows you to view data that is passed through patch cords. This can be very helpful when debugging a patch. There are three types of probes: matrix, signal, and event. The matrix probe allows you to monitor data flowing through a Jitter patch cord. The signal probe allows you to monitor data flowing through an MSP (audio) patch cord. The event probe displays event data that is passed through a Max patch cord. All three types can be activated via the Debug menu.
- **Project** : A Project is a collection of dependencies that are used in a patch. These dependencies may include patches, abstractions, JavaScript files, media files, data files, third-party externals, etc. Projects are an easy way to keep related files in the same place, and allow for Project-specific search path management outside of the standard Max Search Paths.

Q

- **Quickref Menu** : The Quickref Menu displays a list of all messages and attributes that the selected Max object accepts. To access this menu, control-click (Mac) or right-click on any inlet to a Max object. Selecting a menu item will automatically create a message box or attrui that is connected to the object via a patch cord.

R

- **RNBO** : A library and toolchain that can take Max-like patches and export them as portable code. RNBO can directly compile that code to targets like a VST, a Max External, or a Raspberry Pi. The online documentation for RNBO can be found here.

S

- **Scheduler** : The internal timing mechanism used to determine when and in what order events in Max are processed.Scheduler settings can be adjusted in the Preferences window.
- **Shader** : A shader is a program that performs a calculation to determine how a 3D object is rendered in a 2D frame. Determining factors include the object's color and position, lighting, texture coordinates, and material characteristics such as how shiny or matte the object should appear.
- **Signal** : This refers to audio data in Max. When audio data is passed between objects, the patch cord is yellow and grey striped.
- **Snapshot** : Snapshots let you save the state of parameters in your patcher. They are closely related to VST and Audio Unit presets. You can define presets for patchers, vst~, amxd~, and rnbo~ objects.
- **Snippet** : A snippet is a patcher file that typically contains a sequence of objects you use often and don't want to have to constantly re-create as you work. Dragging and dropping a snippet from the left-hand toolbar into your patcher will automatically add the contents of that snippet to your patch.
- **Standalone** : A standalone is basically an application built from a Max patch. Any patch can be turned into a standalone, which looks and acts like a standard Macintosh or Windows application. You do not need Max installed on your computer to run a standalone. Every standalone includes a runtime version of Max.
- **Style** : Styles allow you to standardize the look of objects by changing their colors and font settings. You can apply a style to a specific Max patch, create your own personal library of styles to be applied as you patch, or make use of "factory styles" that come with Max. To learn more about Styles, see this guide.
- **Subpatcher** : A subpatcher is a patch within a patch. Subpatches are often used to organize your work.

- **Symbol** : A symbol is essentially a word in Max. A message box with the word "hello" would be considered a symbol. However, numbers and lists can also be considered symbols if they are in quotation marks or converted using the tosymbol object. See this guide for information on how to include special characters in symbols.

T

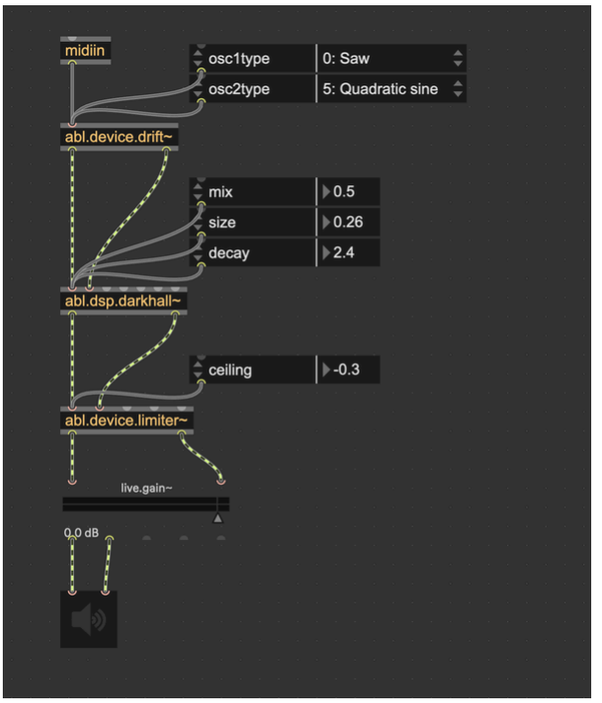
- **Template** : A template is a preconfigured Max patch that is intended to be a starting point for creating certain patches. Templates can include objects that you often use as well as specific patcher attributes for setting the look of your patching environment. Max comes with a selection of premade templates, or you can create your own.
- **Texture** : A texture is essentially an image that is overlaid upon geometry. Textures are used in the context of Jitter. Just like other images in Jitter, textures have an alpha, red, green, and blue component.
- **Theme** : As of Max 8, you have the ability to select from a number of color themes using the Color Theme preference in the Preference Window.Themes change the application-wide colors of patcher toolbars, the sidebar, and other interface elements (as well as default patcher colors).
- **Time Values** : Time Values refer to any measurement of time used in Max. There are two types of time values: fixed time values and tempo-relative time values. Fixed time values express an amount of time that doesn't change. These include milliseconds, hours/minutes/seconds, samples, and frequency. Tempo-relative time values vary according to the current tempo set in a transport object. These include ticks, note values, and bars/beats/units.
- **Toolbar** : Toolbars provide quick access to commonly used tools in Max, as well as audio and video samples, images, and plugins. There are two types of toolbars: patcher toolbars and toolbar mini browsers. For further information on each toolbar, see the Patcher Window documentation.
- **Transport** : A tempo-based timing mechanism that can be used to keep time and/or control the behavior of certain objects. There is both a transport object and a Global Transport found under the Extras menu.

Audio

Ableton DSP

abl.device vs. abl.dsp 23
Inlets and attributes 24
Internal smoothing 24

The Ableton DSP package is a collection of objects that bring Ableton Live devices and high-level DSP components into Max. From oscillators to modulators and filters to reverbs, these objects provide building blocks that speed up patch creation.

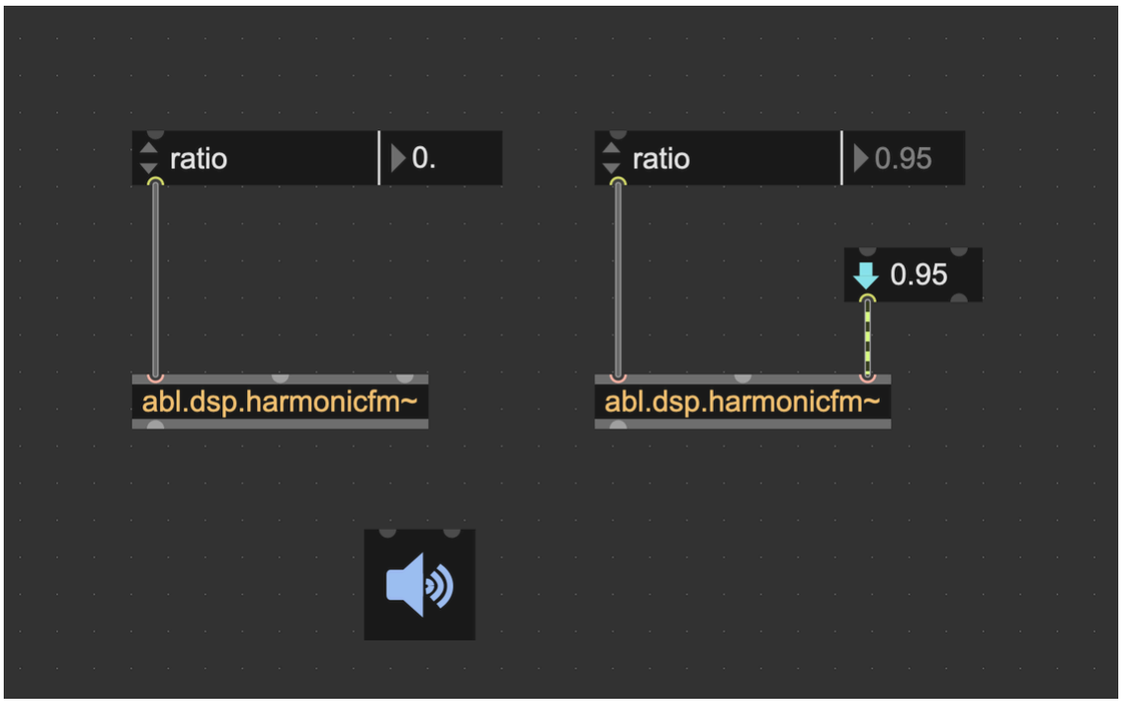


abl.device vs. abl.dsp

The prefixes "abl.device" and "abl.dsp" are used to distinguish between objects that wrap entire Live devices and objects that wrap DSP components. For instance, abl.device.utility~ has the same functionality as the Live Utility device, whereas abl.dsp.ramp~ wraps one of the modulators in Live's Meld instrument and abl.dsp.shimmer~ wraps one of Live's Hybrid Reverb audio effects.

Inlets and attributes

In most Ableton DSP objects, there are a select number of parameters that can be changed as either attributes or signals. For example, the `@ratio` attribute of abl.dsp.harmonicfm~ can be controlled via the third inlet. When a signal is connected to the inlet, the attribute will become disabled while the signal takes over control. If the signal is disconnected, the attribute will re-enable.



Internal smoothing

Unlike most Max objects, Ableton DSP objects offer internal parameter smoothing. Whenever a float-type attribute is changed at event-rate (from an attrui or float inlet, for example), a short ramp is applied instead of immediately stepping to the new value. This mitigates "zipper noise" as attributes are changed at event-rate. However, if you control a parameter at signal rate by attaching a signal to an inlet, no extra smoothing is applied.

Frequency Domain Processing

What is Frequency Domain Signal Processing? 25
The fft~ and ifft~ Objects 25
The pfft~ Object 27
Using gen~ 32

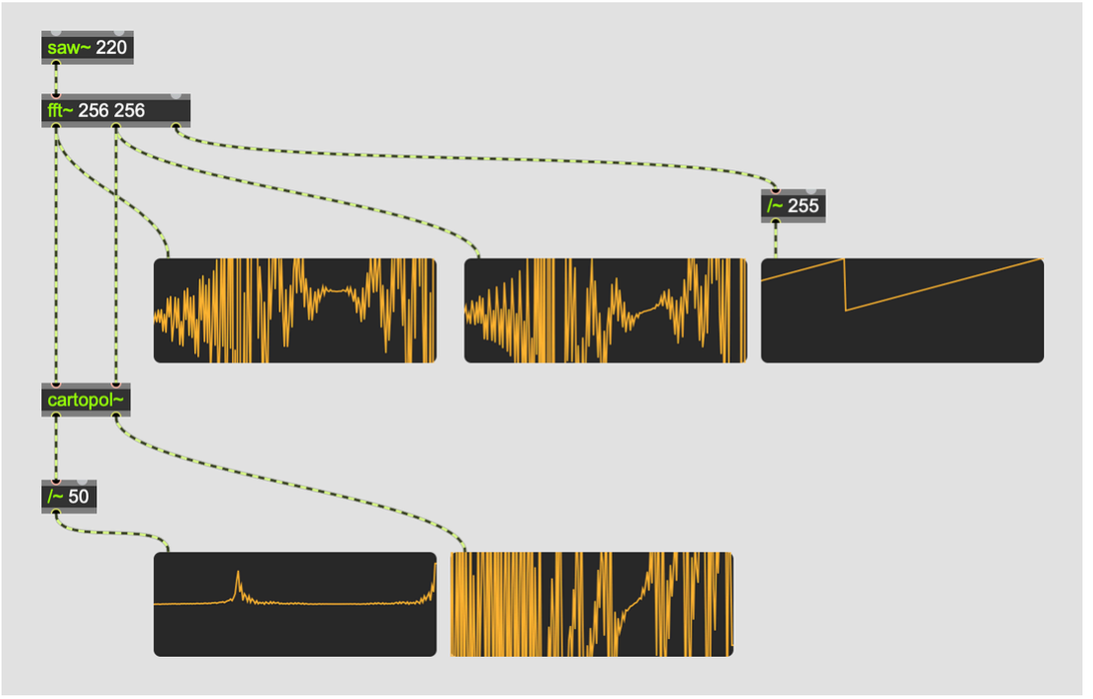
Max provides several objects for working in the **Frequency Domain**, including an object for performing a **Fourier Transform** fft~, as well as an object that can perform an **Inverse Fourier Transform** ifft~. Most of the time, it's easier to work with the convienience object pfft~, which abstracts over some of the technical details of working with the Fourier transform, including windowing and overlap-add. Very often, you'll see Fourier transform abbreviated to FFT, or **Fast Fourier Transform**.

What is Frequency Domain Signal Processing?

Fourier analysis transforms a signal from a **time domain** representation to one in the frequency domain. Instead of specifying the value of a discete signal (for example sound) at a specific point in time, a frequency domain representation gives the amplitude and phase of each of several sinusoidal functions that, when summed together, would reproduce the original function.

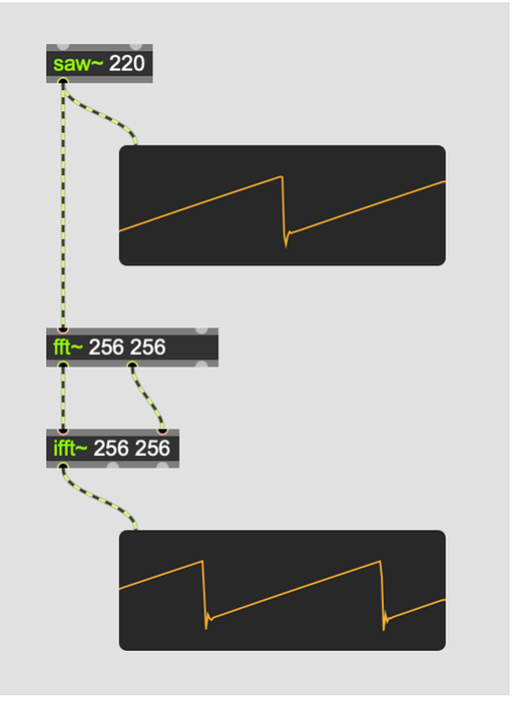
The fft~ and ifft~ Objects

The fft~ object takes a signal as input and outputs the real and imaginary components. The arguments to the fft~ object let you specify the size of the analysis window, as well as the spacing between analysis frames. You can also specify a global offset in frames, which will offset the Fourier analysis of this particular fft~ object against other fft~ objects. This is useful for performing overlap-add. The cartopol~ object converts between cartesian and polar coordinates, and will transform real-imaginary pairs to amplitude-phase pairs. The last outlet of fft~ and ifft~ is the current analysis bin, which is useful for synchronizing the Fourier transform with other processes.



Fourier analysis of a sawtooth signal at 220 Hertz, using an fft~ object. The cartopol~ object converts to the more intuitive amplitude and phase.

You can perform the inverse of a Fourier transform with the ifft~ object. The arguments to an ifft~ object are the same as to fft~, and passing a signal first through fft~ and then ifft~ should recover the original signal exactly.

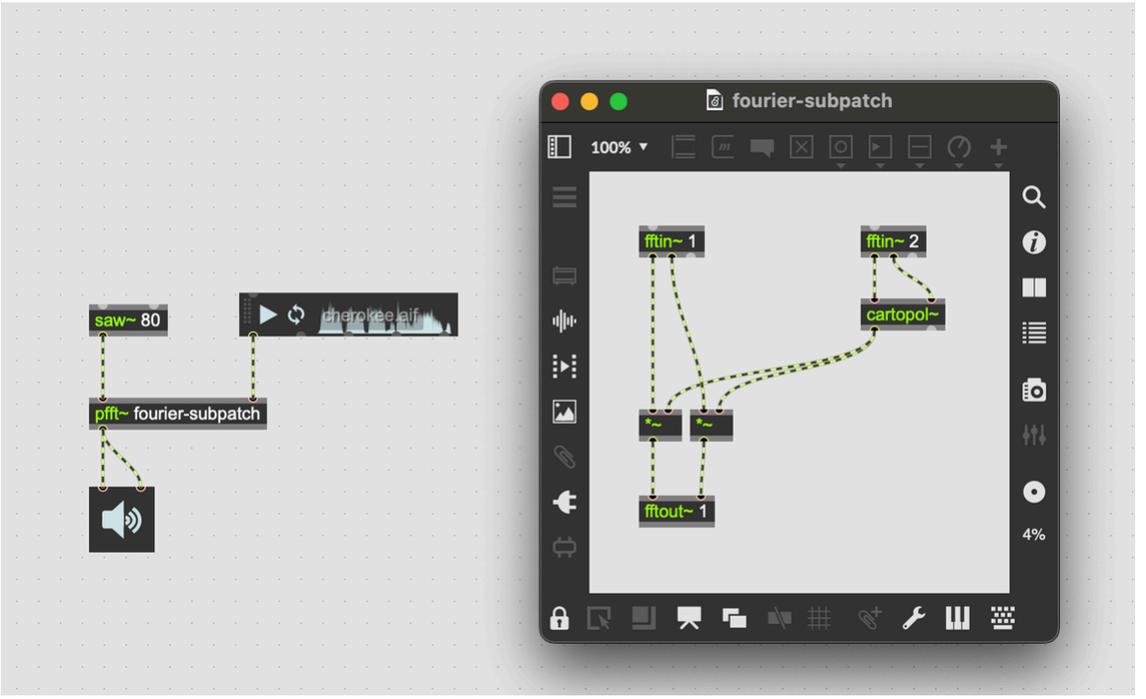


The pfft~ Object

Most of the time, we want to do something with the signal after taking the Fourier transform. Techniques like phase vocoding, comvolution, and spectral delay all work by taking an input signal, transforming that signal to a frequency domain representation, doing some operation, and then transforming the signal back. However, since we're working with successive frames of Fourier analysis, this can introduce discontinuities, which will become very audible distortions to the final signal.

Commonly, we combine a windowing technique with an overlap-add technique to smooth out the audio signal. Essentially, we analyze the same input signal multiple times with a sliding window, and add the output together after transforming back from the frequency domain.

It's possible to use multiple fft~ and ifft~ objects to do this, but it's usually much easier to use pfft~. The pfft~ object loads a patch as an **abstraction**, and then takes care of all of the windowing and overlap-add behind the scenes. The abstraction loaded by pfft~ is free to work entirely in the frequency domain, without worrying about the details of how the Fourier analysis is taking place.



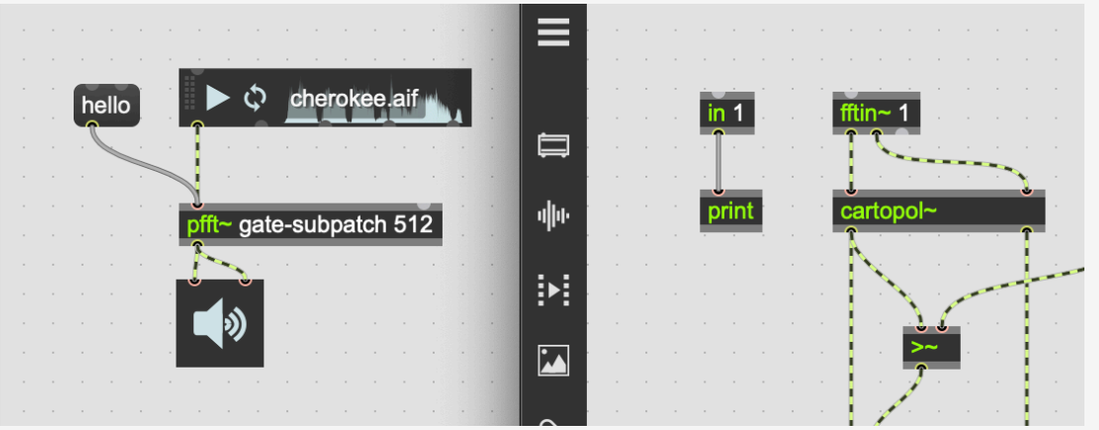
A classic vocoder effect, using pfft~. The pfft~ object takes care of windowing and overlap-add, without the user needed to think about these details.

The argument to a pfft~ object is simply the name of the patcher to load as an abstraction. Similar to fft~, it's also possible to specify the size of the FFT analysis frame, which must be a power of two. Because pfft~ is also performing overlap-add, it's also possible to set the overlap factor as an argument. Changing the analysis frame size and the overlap factor will affect the quality of the FFT output.

Inlets and outlets

Similar to **polyphonic** patches loaded with poly~, the pfft~ object uses special inlet and outlet objects. For regular Max messages, use the in and out objects. For signal, pfft~, uses the fftin~ and fftout~ objects.

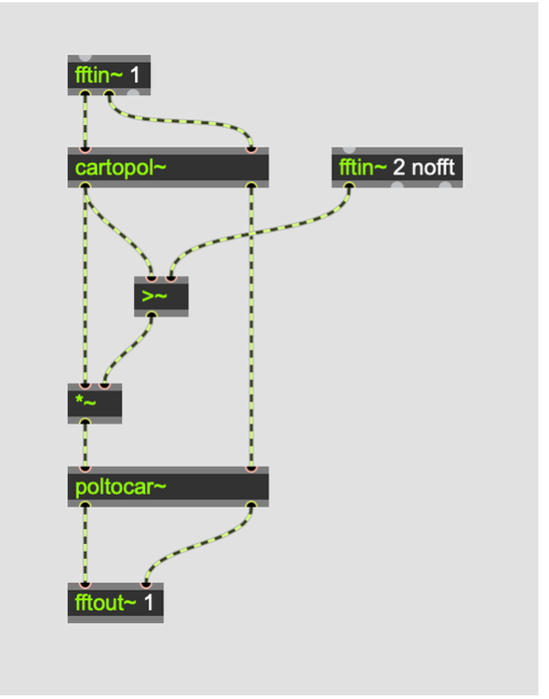
As is the case with poly~, if a signal inlet created with fftin~ shares an index with an in object, the parent pfft~ object will create in inlet that can accept messages or signals.



The in object has an index 1, and fftin~ has the same index. The parent inlet can accept a message 'hello' or a signal.

Unlike regular inlet and outlet objects, the fftin~ and fftout~ objects take an argument to specify their index. This means that multiple fftin~ objects can refer to the same inlet, and multiple fftout~ objects can refer to the same outlet. If a signal goes to two different fftout~ objects that refer to the same outlet, those signals will be summed.

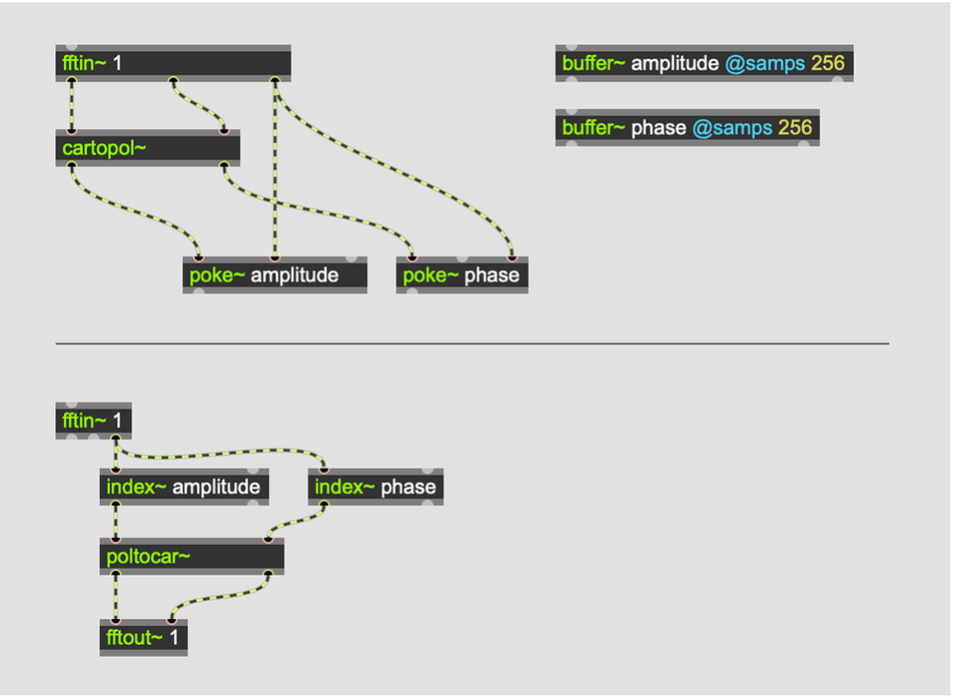
The fftin~ and fftout~ objects also let you specify a window function to use on the incoming signal. The special window name `nofft` disables the Fourier transform entirely, which lets you send audio signals into the subpatcher without transforming them.



A simplistic noise gate effect implemented in pfft~. The second inlet uses the 'nofft' window, which disables the FFT operation on the second inlet. This lets the user set the threshold for the gate using a signal.

Bin index

The last outlet of the fftin~ object specifies the **bin index**, a continuous ramp from 0 to $Fr/2 - 1$ Fr/2 − 1, where Fr Fr is the frame size. This is very useful for synchronizing the the FFT itself with other processes. For example, it's common to use the bin index in conjunction with poke~ and index~, to record and play back an FFT.



Using poke~ and index~ in conjunction with the bin index output of fftin~. This patcher wouldn't really do anything, since the amplitude and phase signals are being read back from the buffer as soon as they're written. However, this illustrates the principle.

fftinfo

In the context of a pfft~ subpatcher, use the fftinfo~ object to report information about the pfft~ parent patcher. This can be very useful if your pfft~ patcher needs to work differently depending on the way in which the FFT is being computed. For example, you might resize a buffer to hold a single frame of FFT analysis data.

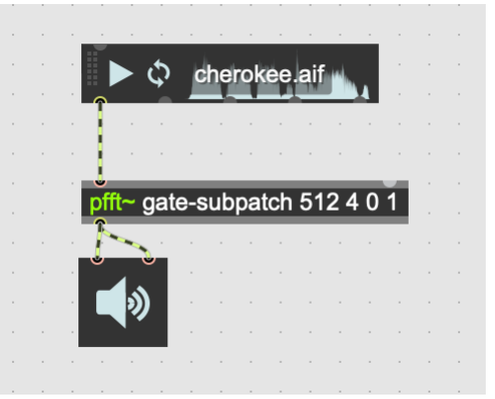
Window function

Specify a window function as an argument to fftin~ or fftout~. Supported window functions are `square` (no window), `triangle`, `hanning` (the default), `hamming`, and `blackman`. The special window name `nofft` disables the Fourier transform entirely, which lets you send audio signals into the subpatcher without transforming them.

Full spectrum

The second argument to pfft~ lets you specify the size of the analysis frame for the Fourier transform. For a given FFT frame size NN, this will generate NN complex pairs. This means that

the larger the analysis frame size, the greater the frequency resolution if the FFT (although because the frame is larger, this reduces the temporal resolution). However, the full FFT is actually symmetric, so even though an FFT frame size NN will yield NN complex pairs, the result has only really resolved the input into $N/2$N/2 unique frequencies. Normally, pfft~ abstracts over this, and simply leaves out the symmetric, complex pairs. However, you can use the `fullspectrum` argument to ask for the complete FFT analysis. This is an argument to pfft~, so all other arguments must be supplied as well.



A pfft~ object with the text 'pfft~ gate-subpatcher 512 4 0 1'. The arguments '512 4 0' specify the default values for the first three numeric arguments, but they must be supplied in order to specify a non-default value for the last argument, asking for a full spectrum analysis.

Using gen~

When using gen~ inside of a pfft~ object, you can use the gen objects fftinfo, fftfullspect, ffthop, fftoffset, and fftsize to retrieve information about the FFT context in which gen~ is operating. You can also make use of the gen~ constants `FFTFULLSPECT`, `FFTHOP`, `FFTOFFSET`, and `FFTSIZE`.

MC

Topics 33

MC supports multiple channels of audio in a single patchcord. It also allows standard Max audio objects to operate on many channels of audio at the same time.

Signal visualization objects such as meter~ and scope~will adapt themselves to multichannel input signals.

The MC Wrapper holds multiple instances of Max audio objects in a single object box. Wrapped object names start with **mc**, for instance mc.cycle~ is the MC-wrapped cycle~ object. The wrapper also offers many powerful features for controlling multiple objects.

In addition to MC wrapper objects, MC-specific objects aid in multichannel signal manipulation.

Topics

- Spatialization - Working with multiple channels over multiple speakers or output channels
- Polyphony - Managing polyphony with MC
- Gen - MC and Gen
- Events - Events and MC objects
- Mixing and Panning - Up- and down-mixing, multi-speaker panning, and signal routing
- Plug-ins - Hosting plug-ins in MC
- Dynamic Routing - Sending multi-channel signals to different places

MC and Gen

See Also 34

Gen and MC combine in the following objects:

- mc.gen~ contains individual instances of gen~ via the MC Wrapper, so it can take advantage of all the features available to any wrapped object.
- mcs.gen~, like other *mcs.** objects, combines the separate Gen inputs and outputs defined by `inX` and `outX` operators into a multi-channels input and output.
- mc.gen contains multiple instances of the event-based gen object. It has an additional outlet that informs Max of the voice number associated with any outgoing message.

See Also

- Using mc.gen With the MC Wrapper
- MC Gen Instances
- Gen Features for MC
- MC Gen Operators

MC Wrapper

When you type mc.cycle~ into an object box, one or more traditional cycle~ objects are embedded inside the **MC Wrapper**.

The wrapper holds multiple instances of audio objects such as cycle~. It permits you to control all of the objects at once with one message or target individual objects. It also manages multi-channel connections to other MC objects. For the most part, this is done transparently and you don't have to think about the wrapper too much. But wrapped objects share some useful messages and attributes not availble in the "unwrapped" versions.

A general rule of thumb is that if there is an MSP object xxx~, mc.xxx~ will be xxx~ in the MC Wrapper. But this is not always the case. Exceptions include:

- Objects that perform I/O: mc.adc~, mc.dac~, mc.ezadc~, mc.plugin~, mc.plugout~, mc.ezdac~, mc.sfplay~, and mc.sfrecord~. In these cases, you don't need multiple copies of objects, you just need multi-channel inputs and outputs to the outside world
- UI objects including gain sliders (mc.live.gain~, mc.gain~, mc.multigain~) and signal visualization objects including scope~, meter~, levelmeter~ and spectroscope~. The **mc.** is optional and does nothing for these objects, as they auto-adapt to the number of channels of a multichannel input signal.
- mc.tapin~, mc.tapout~, mc.send~, mc.receive~ : these can be thought of as I/O objects to internal memory buffers
- Any objects begining with mcs which are single instances of audio objects whose inputs and/or outputs are combined into a single multi-channel inlet and/or outlet
- MC objects specific to multi-channel signal manipulation

Multi-Channel Audio I/O

Logical and Physical Audio Channels 36
Assigning Logical to Physical Channels 37
MC Hardware Interfacing 38
Working with Multichannel Input 38
Working with Multichannel Output 39
Audio Hardware Routing 40
See Also 40

The first part of this guide describes how audio input and output works in Max, specifically how channels map to hardware. The second part describes how MC works with audio hardware. Work with patch cords and objects that deal with multiple channels simplifies working with audio hardware, but is not absoltely necessary.

Logical and Physical Audio Channels

Max provides up to 1024 logical audio channels that you can dynamically assign to the physical channels of your audio hardware.

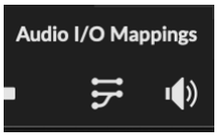
When you create an dac~ or adc~ object with different channel numbers as arguments, these numbers refer to logical channels. These channels are internal to Max, and they can be dynamically reassigned to physical device channels of a particular driver using either the Audio Status window, its I/O Mappings subwindow, or an adstatus object with an input or output keyword argument.

The purpose of having both logical channels and physical device channels is to allow you to create patches that use as many channels as you need without regard to the particular hardware configuration you're using. For instance, some audio interfaces use physical device channels 3 and 4 for *S/PDIF* input and output, others use channels 9 and 10 instead. Without logical mapping you would have to change the arguments on all of your adc~ and dac~ objects when you changed interfaces.

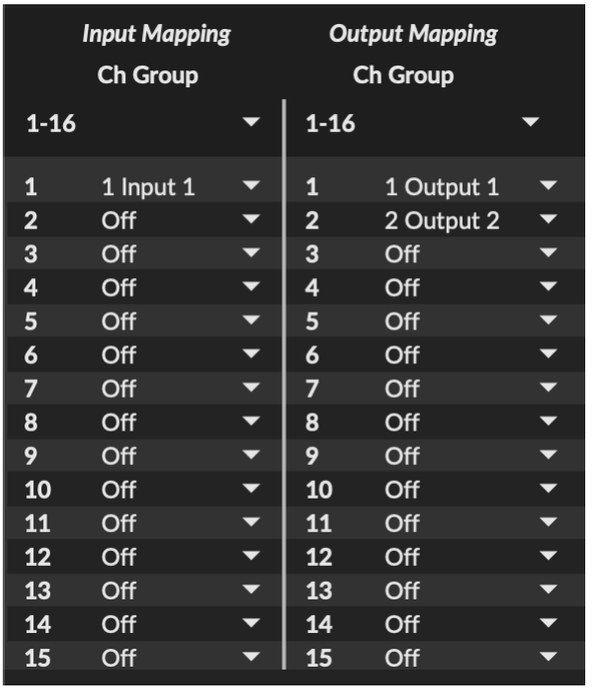
Note that logical channels can map to a single physical channel.

Assigning Logical to Physical Channels

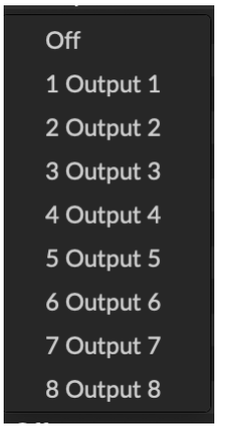
To configure the assignment of logical to physical audio channels, open the **Audio I/O Mappings** window from the Preferences window toolbar.



The window displays assignment menus for logical input and output channels in groups of 16. This is the default assignment for a single-channel audio input device and a two-channel audio output device.



The assignment menu contains the physical channels of the selected hardware input or output devices. For example, here is the output assignment menu when using a Focusrite interface that has eight output channels.



Use the menu to set the desired physical channel for each logical channel you want to use.

To move to another bank of 16 channels, use the menus for input and output at the top of the Audio I/O Mappings window.

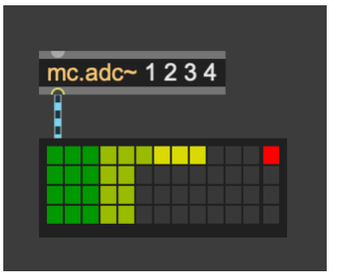
MC Hardware Interfacing

MC provides flexible managment of multichannel signals, but interfacing with your hardware requires system-specific setup and routing. The adc~ object for audio input has a multichannel counterpart (mc.adc~) with a single multichannel output, and the dac~ object for audio output has a multichannel counterpart (mc.dac~) with a single multichannel input.

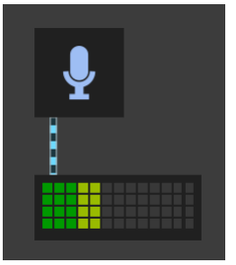
Working with Multichannel Input

Use mc.adc~ object to obtain a multi-channel input signal from your audio interface.

- Create an instance of the mc.adc~ with the input channels that you want to access. The channels are numbered starting with 1.
- Connect the multichannel signal output to any processing or monitoring objects to work with the MC content.



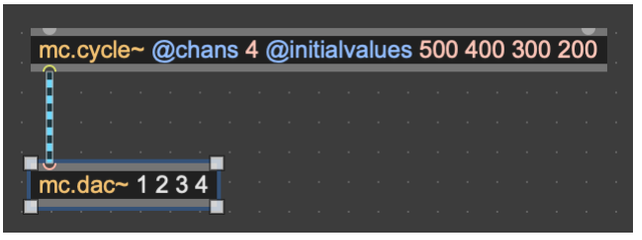
A multi-channel version of ezadc~ with two signals in one patch cord is availble with mc.ezadc~.



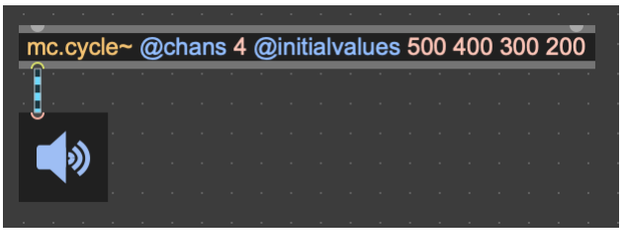
Working with Multichannel Output

Use mc.dac~ to send a multi-channel audio output signal to your audio interface.

- Create an instance of mc.dac~ with the output channels that you want to access. The channels are numbered starting at 1.
- Connect a multichannel signal to the input of the object to send the audio stream to your interface.



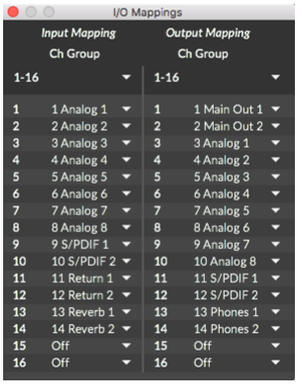
A multi-channel version of ezdac~ that outputs all channels of its input multi-channel signal is availble with mc.ezdac~.



Audio Hardware Routing

In addition to setting up the inputs and outputs of your patch, you will also need to verify the hardware input and output routings are correct. This is done using the Audio I/O Mappings window available in the Preferences window toolbar.

The Audio I/O Mappings window establishes mappings between channels on your audio interface hardware and Max's 1024 *virtual* channels.



For each input channel you wish to use for multichannel input (via mc.adc~), choose a hardware input from the pop-up menu. For each output channel you wish to use for multichannel output (via mc.dac~), choose a hardware output from the pop-up menu. You can assign the same hardware input or output to any number of virtual channels.

See Also

- Spatialization with MC
- MC in Max for Live

Non-real-time Processing

Non-real-time Audio 41
Non-real-time Video 43

Most of the time, Max is running in real-time. For every video and audio frame, Max produces just enough video and audio to fill that frame. If you tweak a parameter or modulate an effect, you can see and hear the result right away.

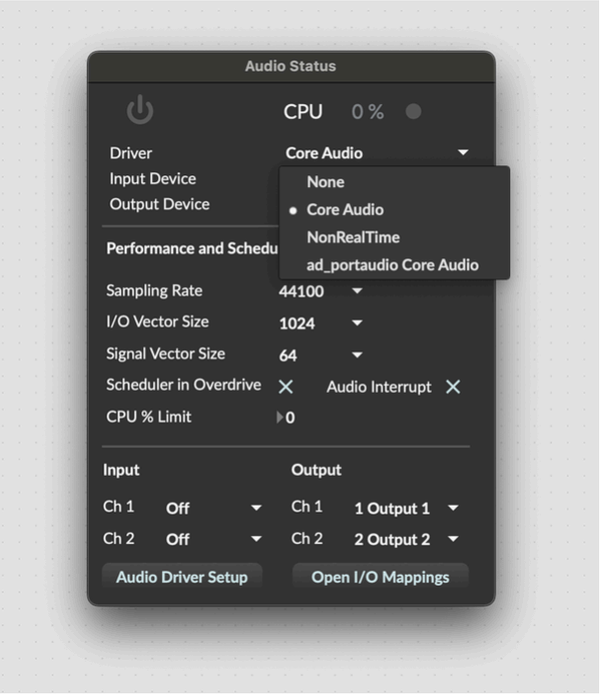
There are two main limitations to real-time processing:

- Some processes are too computationally intensive to run in real-time.
- It takes exactly as long to render a process in real-time as the resulting media. So, it takes 10 minutes to render a 10 minute video in real-time, even if it could have been processed faster.

By processing audio or video in non real-time, you can get around both of these limitations.

Non-real-time Audio

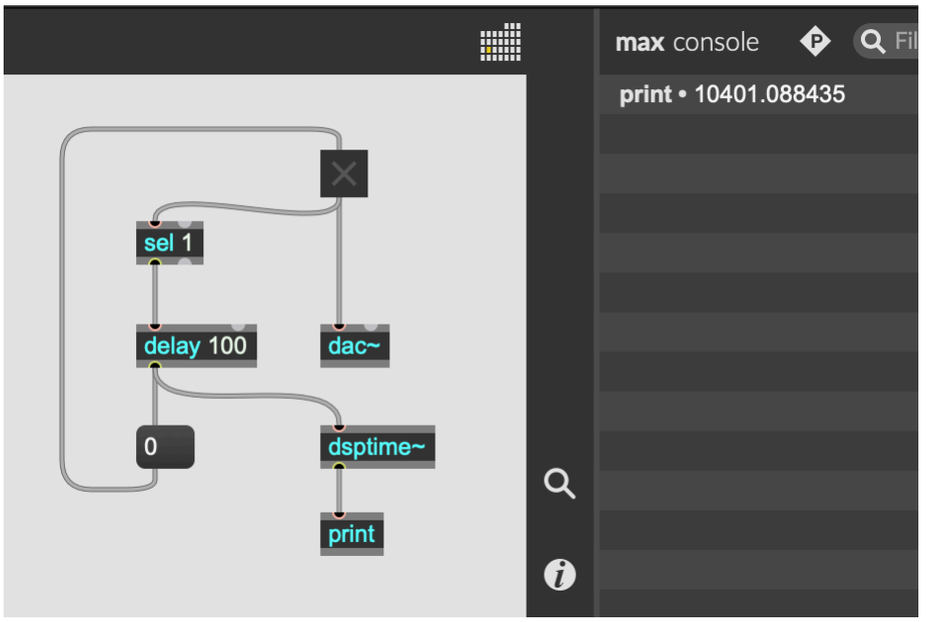
In the *Audio Status Window* select the *NonRealTime* driver.



In non-real-time mode, hardware input and output are disabled. Audio processing runs independently of any physical scheduling priority, which essentially means that Max runs the audio thread as fast as possible. This means that patches designed to run in non-real-time mode have a typical structure:

1. Some way to activate audio processing
2. Some way to monitor the state of audio processing (often polling dsptime~ to measure elapsed processing time)
3. An automatic way to disable audio processing

This patcher enables audio processing, waits 100 milliseconds, and then prints the elapsed time according to dsptime~, which measures how many milliseconds of audio have been processed.



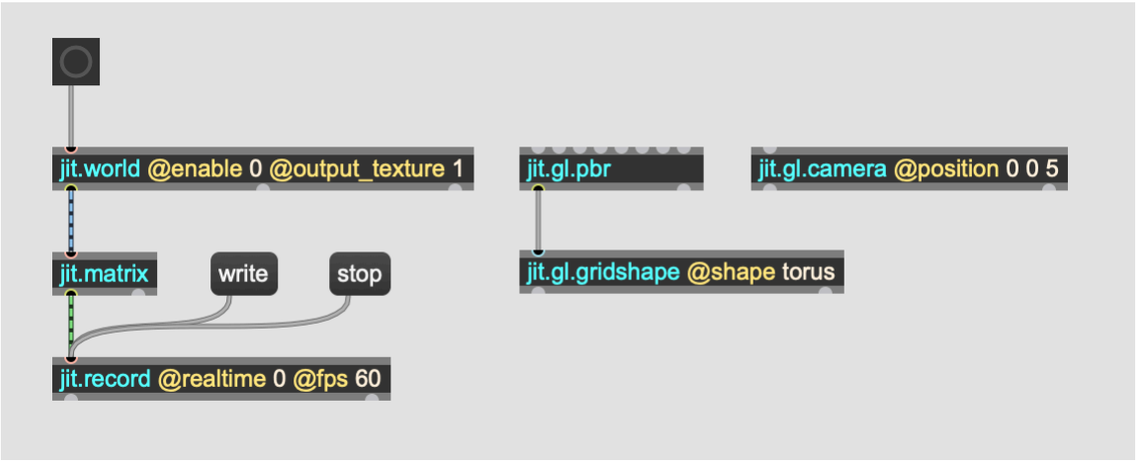
As you can see, about 10 seconds of audio have been processed in 100 milliseconds of real-time.

The typical way to capture audio output in non-real-time mode is to record to a file with sfrecord~. The following patcher synthesizes about a second of noise and captures the output.

Non-real-time Video

Recording video in non-real-time is a bit simpler than recording audio in non-real-time. The easiest method is to use jit.record with the `@realtime` attribute disabled. In this mode, every time jit.record receives a matrix, it will be appended to the currently open video file. The frame rate of the output will be determined by the `@fps` attribute of the jit.record object and *not* by the real-time rate at which new matrices are recieved.

If you're using jit.world or jit.pworld to manage your virtual scene, set the `@enable` attribute to `0` to disable automatic rendering. In this configuration, you can send a `bang` to jit.world to manually render a single frame. If you're using jit.gl.render to render a virtual scene to a texture, you can pass your texture through a jit.matrix to convert it before output.

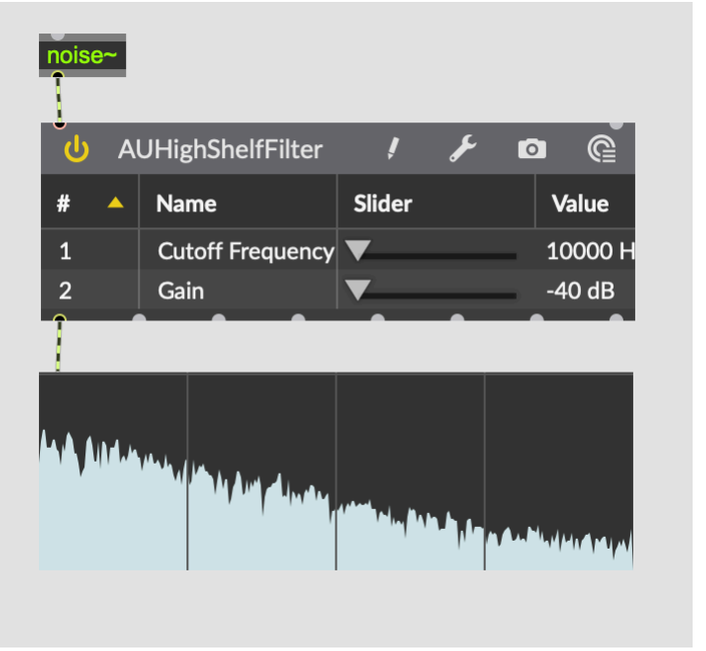

jit.world @enable 0 @output_texture 1
jit.gl.pbr
jit.gl.camera @position 0 0 5
jit.matrix
write
stop
jit.gl.gridshape @shape torus
jit.record @realtime 0 @fps 60

Audio Plugins

Loading a Plugin 46
User Interface 47

Most DAWs (Digital Audio Workstations) support plug-ins, which let third parties extend the audio processing capabilities of the main software. Max supports Audio Unit, VST, and VST3 plug-ins, and can load Max for Live devices (AMXDs) as plug-ins as well.

> The Max object wrapping VST and Audio Unit support is called vst~ or mc.vst~, and the object wrapping Max for Live devices is called amxd~ or mc.amxd~. The object plugin~ is for authoring Max for Live devices, and does not load plug-ins itself.

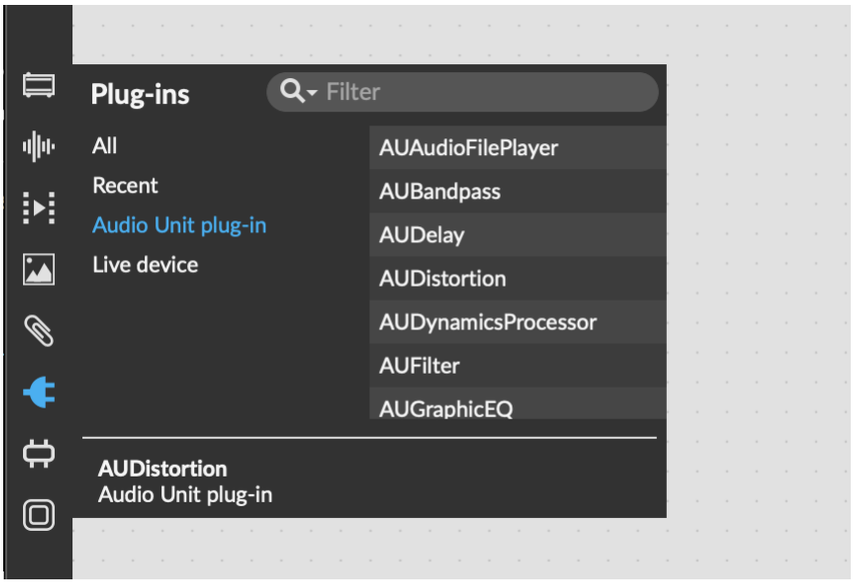


macOS includes an Audio Unit called AUHighShelfFilter, which Max can load as a plugin to implement that audio effect.

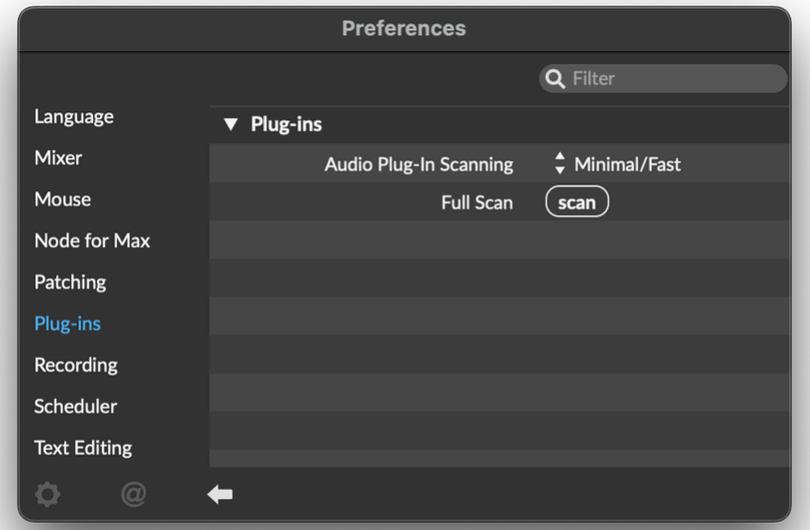
Loading a Plugin

If you have a `.component` or `.vst` file, you should be able to drop that file onto an unlocked patcher to load the plug-in. This will automatically create a vst~ object that points to that file.

The left toolbar also provides access to the **Plug-in Browser**. Clicking on the *Plug-ins* icon will open the browser, letting you filter for plug-ins and AMXDs by name and kind.

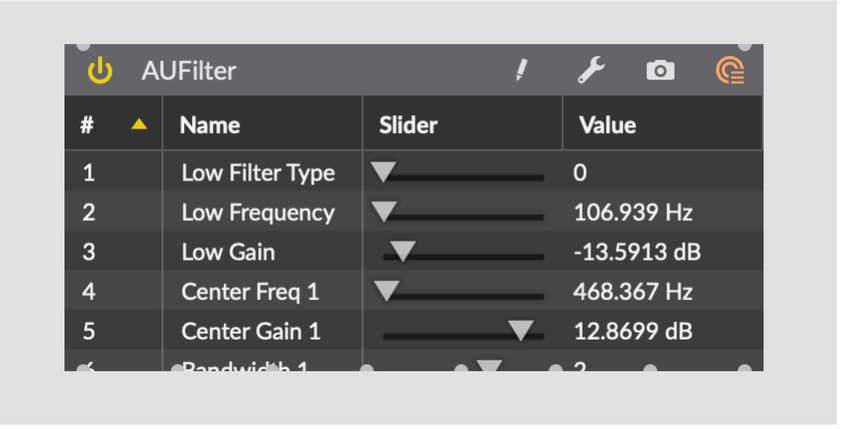


Max builds the list of plug-ins by scanning for them at launch. In the *Plug-ins* section, you can use the *Full Scan* button to initiate a scan manually, which can be useful if you're adding new plug-ins with Max open.



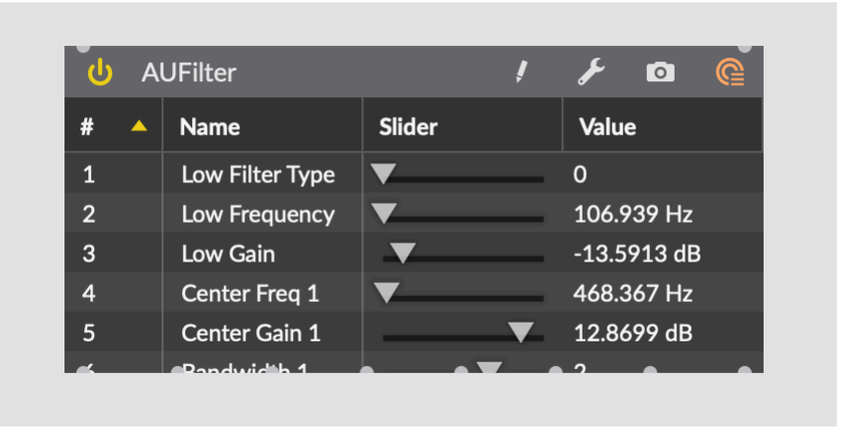
User Interface

When you drop a plug-in into your patcher, the vst~ or amxd~ object will enable the `@viewvisibility` attribute and show you a user interface for the plug-in. For an Audio Unit or VST plug-in, you'll see the generic interface. This lists all of the parameters in the plug-in, and lets you set their values.



The generic plug-in interface

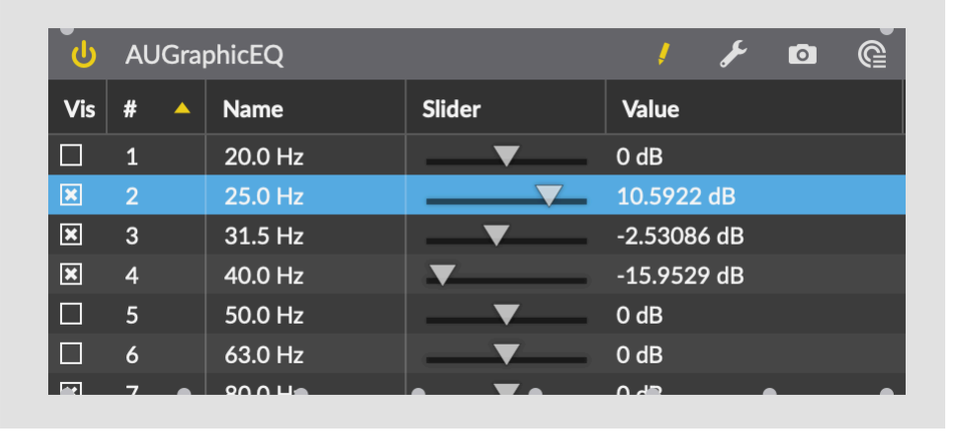
With Max for Live devices, you'll see the customized user interface for that device.



The custom user interface for the 'Additive Heaven' Max for Live device

Configuring parameter visibility

In the generic interface, click the pencil icon in the top toolbar to edit parameter visibility. Disable the *Visibility* checkbox to hide the parameter from view.



Viewing the native editor

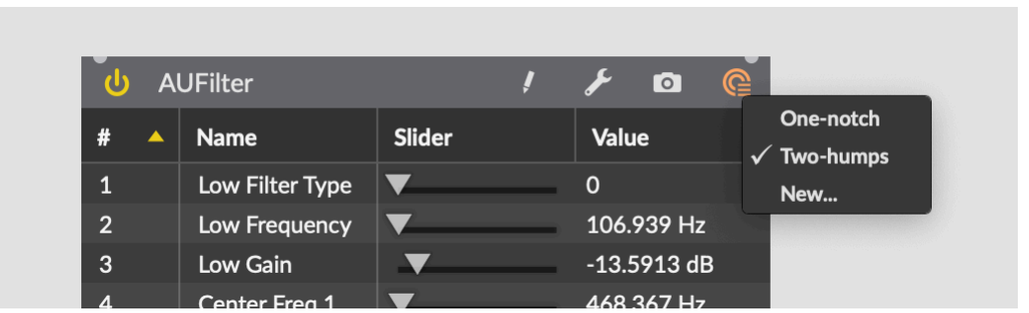
Most VST and Audio Unit plug-ins provide their own user interface. Click on the wrench icon in the top toolbar to open the native editor.



The native editor for the AUFilter Audio Unit

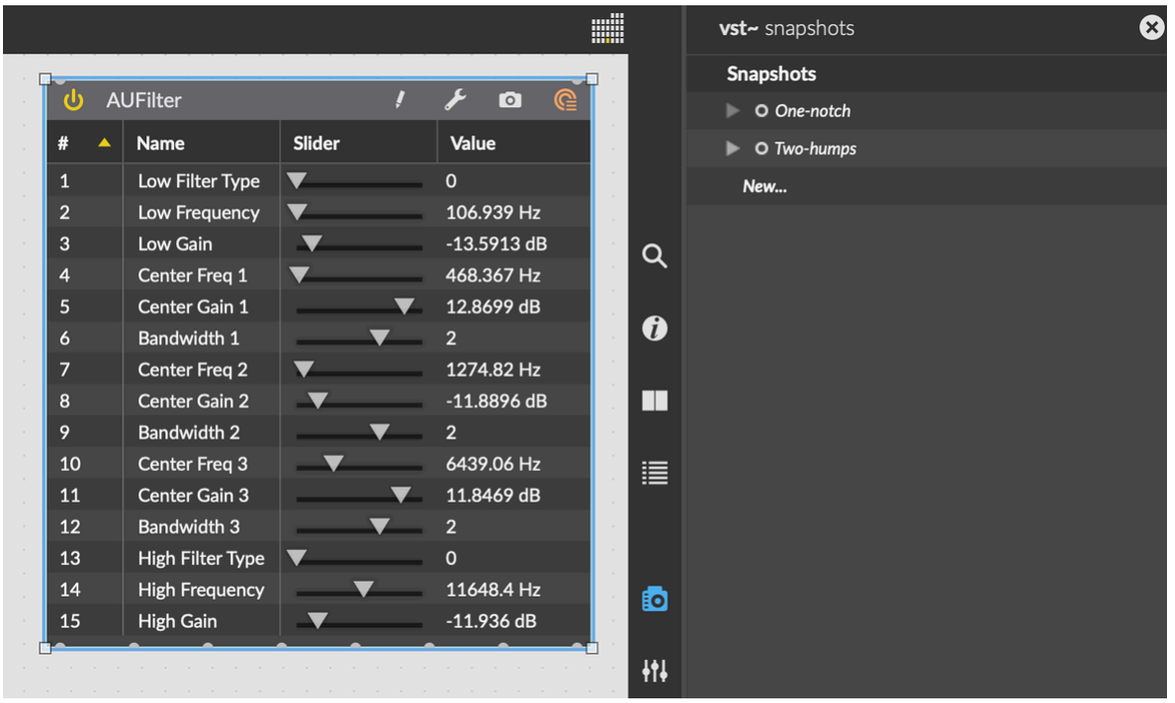
Saving and restoring snapshots

You can save the current state of a plug-in as a **Snapshot**, which will include the current value of all of the plug-in parameters. Click on the camera icon in the top toolbar to save the current parameter set to a snapshot. Click on the snapshot selection button in the top-right to list all saved snapshots.



Viewing saved snapshots for the plug-in

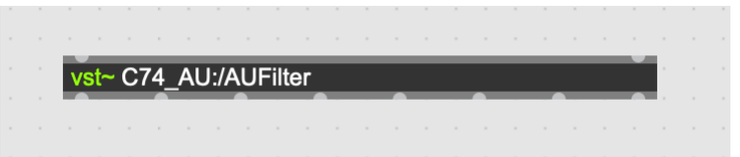
The plug-in object UI does not provide any way to edit or delete snaphots. Select the vst~ object and open the **Snapshot Sidebar** to edit the names of snapshots, or to delete a snapshot.



With the vst~ object selected, open the snapshot sidebar to edit snapshots.

Hiding the controls

The `@viewvisibility` attribute on a vst~ or amxd~ object determines whether or not the object will show an editable interface in the patcher. By disabling this attribute, you can get a much more compact representation of the wrapping object.



With @viewvisibility disabled, the object looks like a typical Max signal processing object.

Polyphony
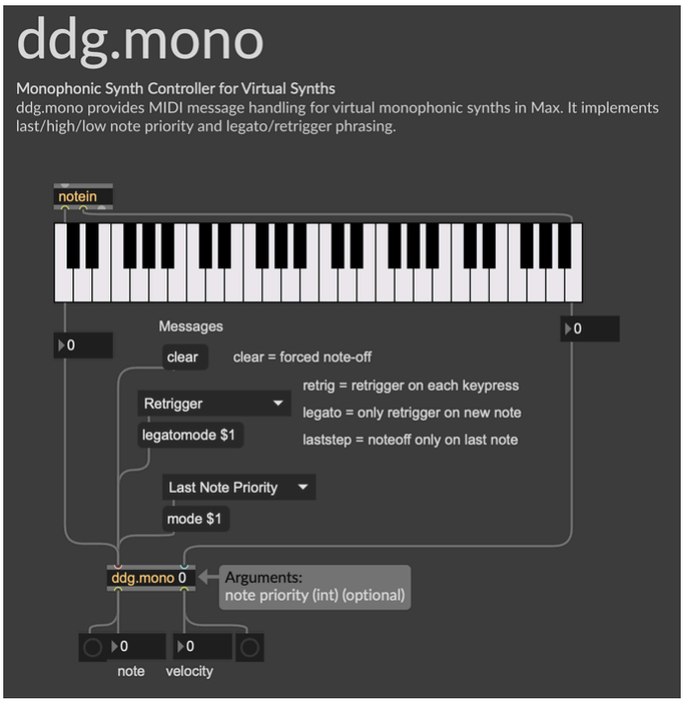
ddg.mono 51
Using poly 52
Using poly~ 53
Using MC Objects 62

Max provides a number of systems for managing **polyphony**. If you've every played an electric piano before, the concept of polyphony should be intuitive. When you press multiple keys at the same time, each key sounds independently, with each note lasting until it fades out, or until each key is released. However it's accomplished, handling polyphony always involves the same basic tasks:

- Routing an input to a specific voice
- Managing the independent state of each voice
- Deciding whether a particular voice is busy
- Muting a voice when it's not in use

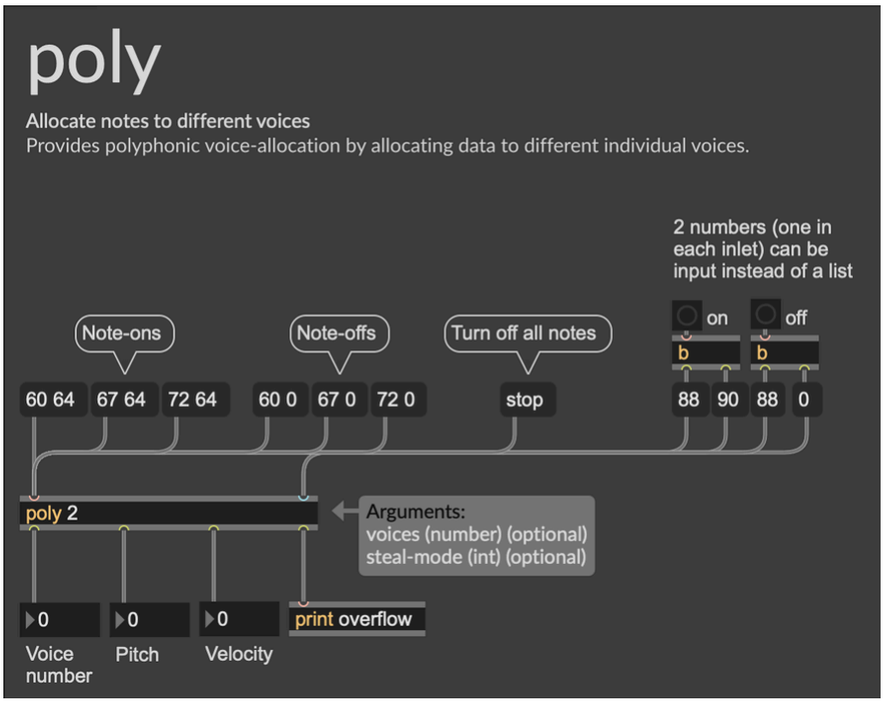
ddg.mono

The simplest way to handle polyphony is not to handle polyphony. The ddg.mono object implements a controller for a **monophonic** synthesizer with polyphonic input. If you were using a keyboard input, when pressing a second key, ddg.mono would shift the pitch to the new key, no longer tracking the first key.



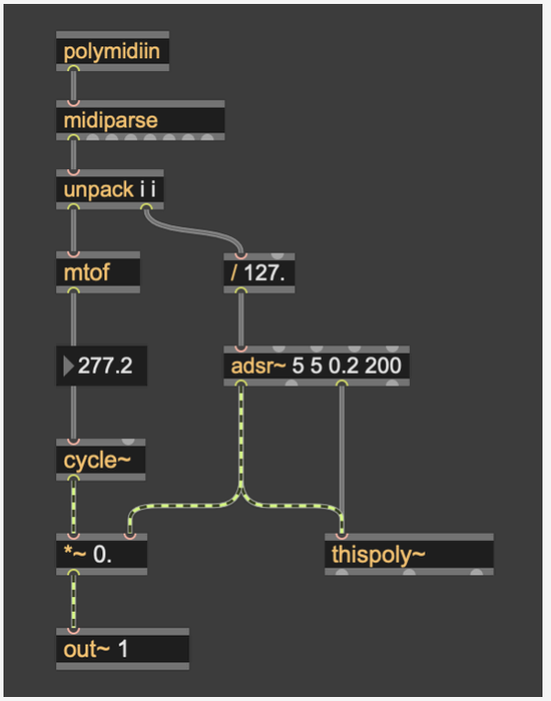
Using poly

The poly object is a polyphonic voice controller for MIDI note-on and note-off events. When it receives a note-on event, it assigns that event to the first available voice. When it gets a note-off event with the same pitch, it marks that voice as no longer in use.



The poly is a voice controller only. That means that it doesn't manage the voices themselves—it won't help you turn one oscillator into several. If you want to have multiple voices sounding at the same time, you should use mutiple patchers, the MC system, or the poly~ object.

Using poly~

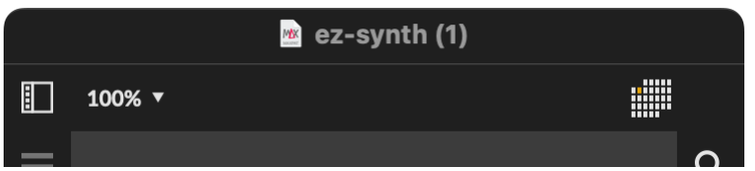


Maybe the "simplest possible" poly~ abstraction patch

The poly~ object combines the voice controller of the poly object with a system for managing multiple voices. When you load an abstraction into poly~, it will create one copy of that abstraction for each voice. Then, poly~ gives you the tools to route incoming messages to a specific voice, while also managing the individual state of each voice.

Viewing voices

Just like other abstractions, you can double-click on a poly~ object to view the patch that it's loaded as an abstraction. However, since poly~ loads one copy of the abstraction for each voice, you'll also see a number next to the name of the patch, indicating which voice you're currently looking at.



The number (1) next to ez-synth tells you that you're looking at the first voice.

You can send poly~ the messge `open` to view the loaded abstraction for a particular voice. The messages `open 1` will show the first voice, `open 2` will show the second voice, and so on.

Event inputs

When poly~ receives a message, it uses one of several systems to determine which voice will receive that message.

The simplest system uses the `@target` attribute. When you set the `@target` attribute to a particular voice number, all subsequent messages will be routed to that voice. If you set `@target 1`, then all messages will be routed to the first voice. Set `@target 0` to make all voices the target, in which case each messages will be copied and sent to each voice.

Messages starting with the symbol `note` will be automatically routed to the first available voice. You can tell poly~ when a voice is busy by using the thispoly~ object—see busy state for more details.

The poly~ object can also handle MIDI note-on and note-off events using the `midinote` message. When you send poly~ a message like `midinote 60 127`, it will get a free voice and send that note-on event to that voice. When you send poly~ a note-off event like `midinote 60 0`, it will find the voice associated with the pitch `60` and send the note off event to that voice. The poly~ object can also receive messages with the symbol `midievent`, like those coming from the rightmost outlet of the midiparse object. In this case, the system for assigning note-on and note-off events to voices is the same.

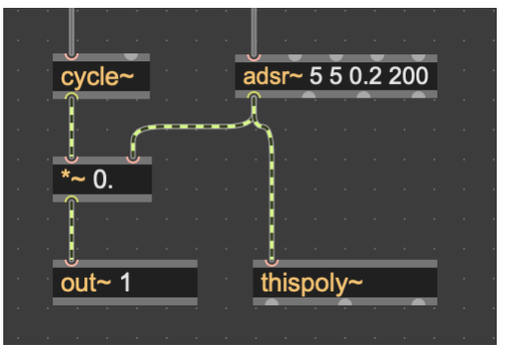
Finally, poly~ is also an MPE-compatible object. Use mpeparse and mpeformat to generate `mpeevent` messages, which poly~ will then route to the appropriate voice. The really cool thing about this approach is that it can manage not only note-on and note-off events, but also polyphonic aftertouch, pitch bend, and other MPE events.

> When using any of the MIDI-based systems for voice control, you can use the `notemessage` message to route events to the voice that has been assigned a particular MIDI pitch. This is different from using `target`, which always addresses the voice at a particular index.

Busy state

In order for poly~ to route `note` and `midinote` messages to the right place, it needs to know when a note is **busy**. From within a particular poly~ abstraction, use the thispoly~ object to manage busy state. Send thispoly~ the message `1` to mark the note as busy, and the message `0` to indicate that the voice is free, and ready to recieve new `note` or note-on messages.

> You can also send thispoly~ a signal, such as the output from adsr~, to mark the voice as busy/free.

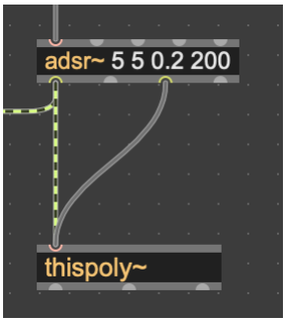


It's extremely common to see an adsr~ connected like this, where it's used both to modulate the signal and to control the busy state of the poly~ abstraction.

Muting

When a poly~ voice is **muted**, Max will skip signal processing for any signal-rate objects in that particular voice. Objects can still pass messages to each other, but no signals will flow between objects. Muting makes poly~ more efficient, since it won't waste processing power computing silent

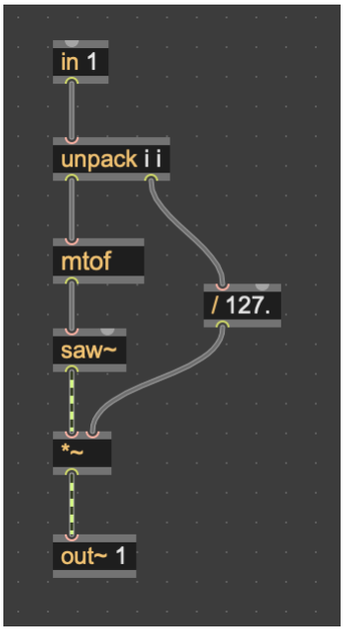
signals for unused voices. This can be especially important when you're working with multiple poly~ objects, or in the context of a custom Max for Live synthesizer.



Since the adsr~ object will also automatically send mute/unmute messages, you'll often see it connected to a thispoly~

Inlets and outlets

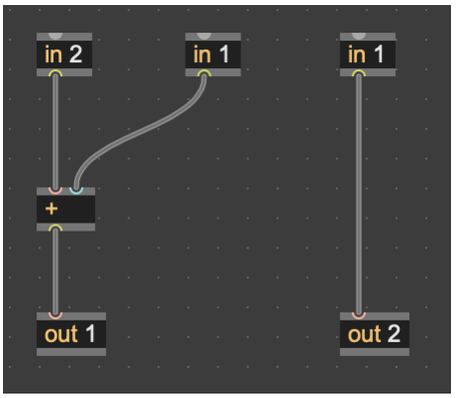
Unlike regular abstractions, poly~ uses the special objects in and out in order to declare inlets and outlets, and the objects in~ and out~ to declare signal inlets and outlets.



This patcher will have one message inlet and one audio outlet.

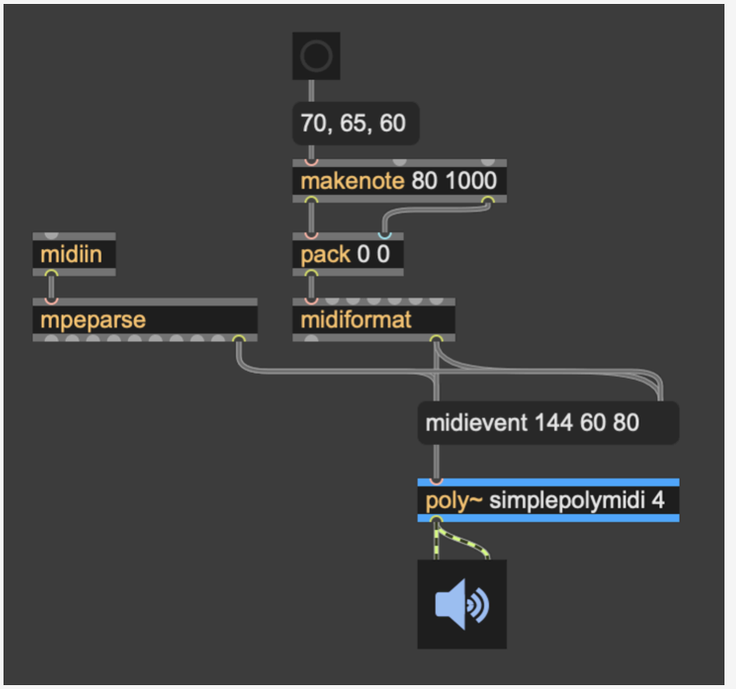
Unlike the inlet and outlet objects, these objects take an argument to specify their index. This has some upside; for example, you can reference the same inlet/outlet in multiple places, and you can

place the inlet/outlet objects in any left-to-right order that you like.



Two in objects share the index 1, and any messages sent to the first inlet will be received by both these objects. Also, the [in 2] object will receive messages sent to the second inlet, even though it appears to the left of the [in 1] object.

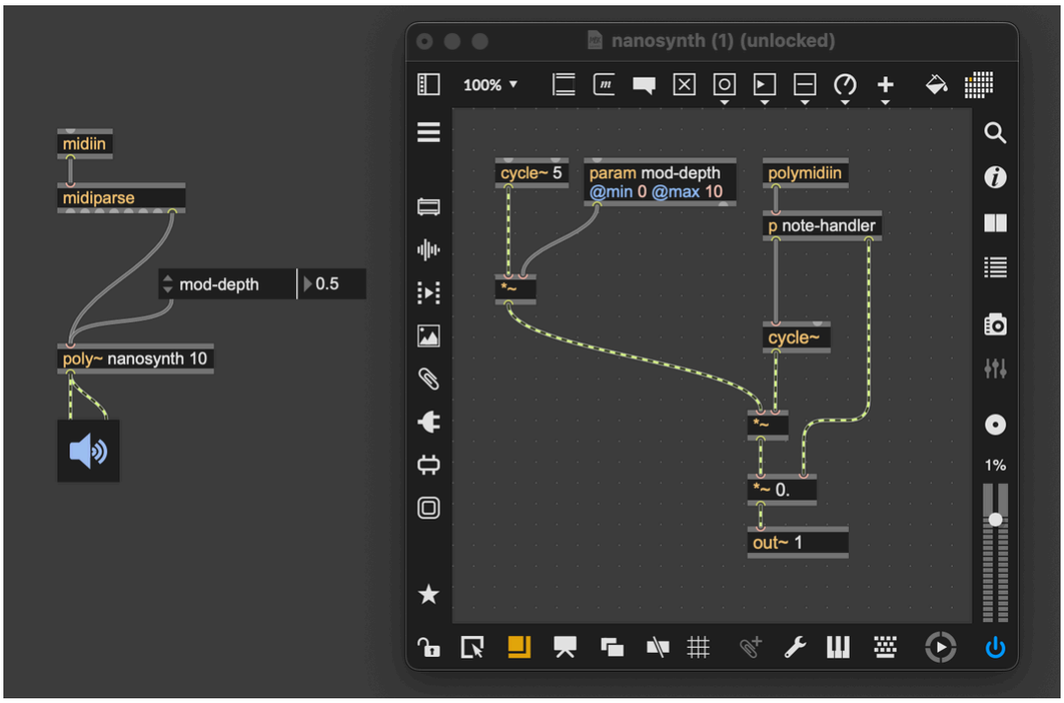
There's also a special object polymidiin, specific to poly~. This object receives any messages sent to poly~ with the `midievent` symbol, like the ones that come from the rightmost outlet of mpeparse and midiformat.



The polymidiin object (not shown but in the simplepolymidi abstraction) works with midiivent messages, like from mpeparse and midiformat.

Parameters

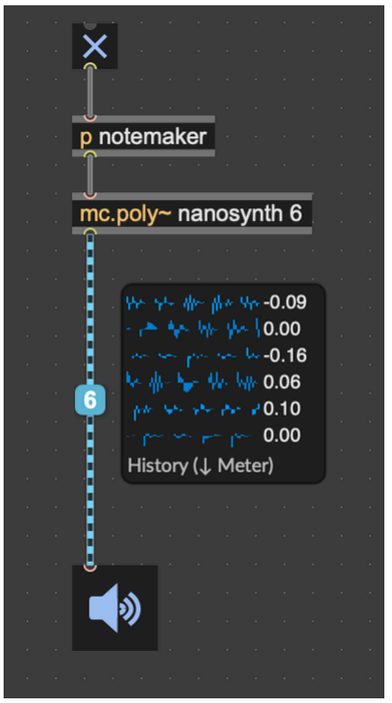
The poly~ object supports the use of the param object, which lets you define custom attributes on your poly~ object. You can control these with an attrui object, just like a regular object attribute. You can set a `@min` and `@max` value on your param too, constraining it to a given range. See the param help file for more information.



The param defines a custom attribute mod-depth" that can be controlled with an attrui object.

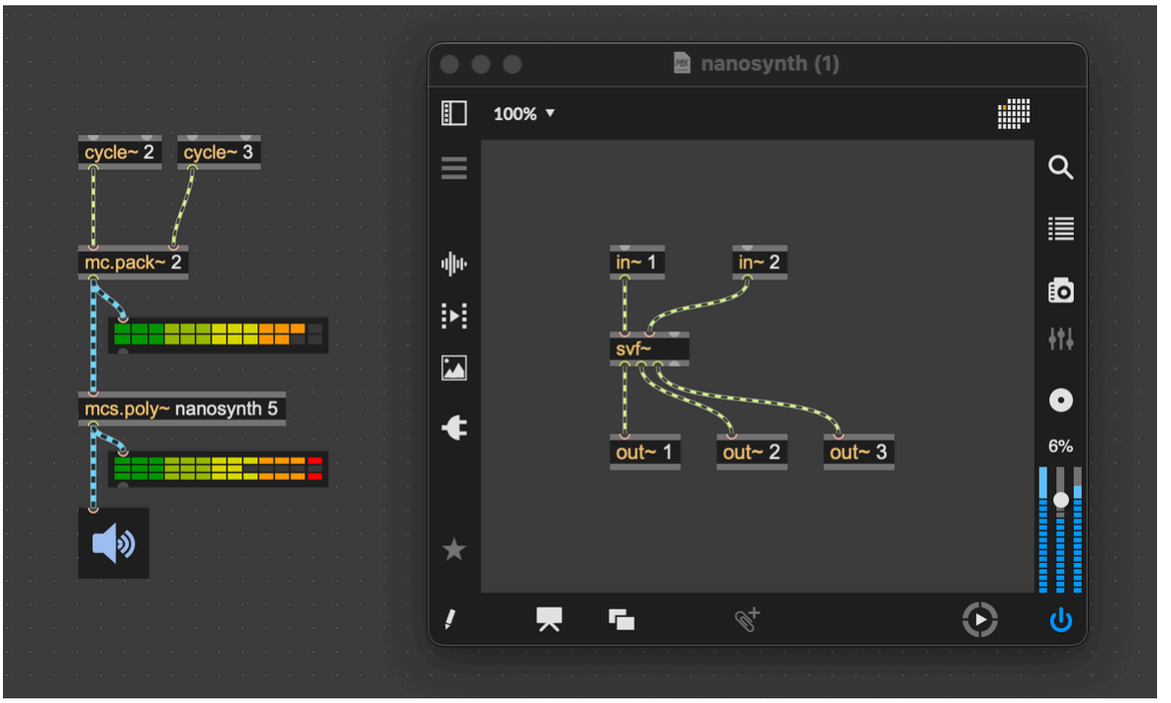
mc.poly~ and mcs.poly~

The poly~ object has two wrappers, mc.poly~ and mcs.poly~. These objects help to bridge the gap between the mc system and the poly~ object. The mc.poly~ object is maybe the simplest to understand: it turns each signal input to poly~ into a multichannel input, where the number of channels is the same as the number of polyphonic voices. Unlike regular poly~, the mc.poly~ object does not add together the signal output from each voice, but keeps each channel separate.



The number of channels depends on the number of polyphonic voices of the mc.poly~ object.

The mcs.poly~ object collapses all of the signal inputs and outputs of a poly~ object down to a single multichannel input and output. The number of input channels and the number of output channels depends on the number of in~ and out~ objects with different indices in the poly~ abstraction. Each audio input and output will be copied and sent to each polyphonic voice.



With mcs.poly~, the number of input and output channels depends on the in~ and out~ objects.

Using MC Objects

The MC wrapper can manage polyphony without the use of the poly~ object. The mc.noteallocator~ object works of of MIDI note-on and note-off events, similar to how poly~ responds to `midinote` and `midievent` messages. There's also the mc.voiceallocator~ object, which works more like poly~ in response to the `note` message. For more information, see the guide on MC Wrapper Polyphony.

Recording and Exporting

Recording Audio 63
Recording Video 68

Recording Audio

When it comes to recording the audio output of your patcher, Max tries to give you lots of options to best fit your needs. **Global Record** is a simple "one-button" recording option suitable for most everyday recording. If you need a little bit more control, or you want to record more than two channels, you can use the **Quickrecord** extra. For programmatic recording, and for recording more than eight channels, you can use the sfrecord~ object, or export directly from a buffer~ object.

File formats

Max objects can save audio to `aiff`, `wave`, `ogg`, `flac`, or `raw` formats. To record to an `mp3` or an `m4a`, export your audio from Max, and then process the result. Many programs can compress and convert audio; ffmpeg is a popular and free tool.

Global Record

When you want to record the output of your patcher, just click the *Global Record* button in the right toolbar.



This will immediately start recording the first two channels of all patchers to a new audio file. The *Global Record* button will change appearance while recording is taking place. If you want to see an animating red dot, instead of a filled white circle, enable the *Global Record Red Button* in Max's recording preferences.

Global Record will record to a folder named *Recordings* in the Max 9 Folder.

Quickrecord

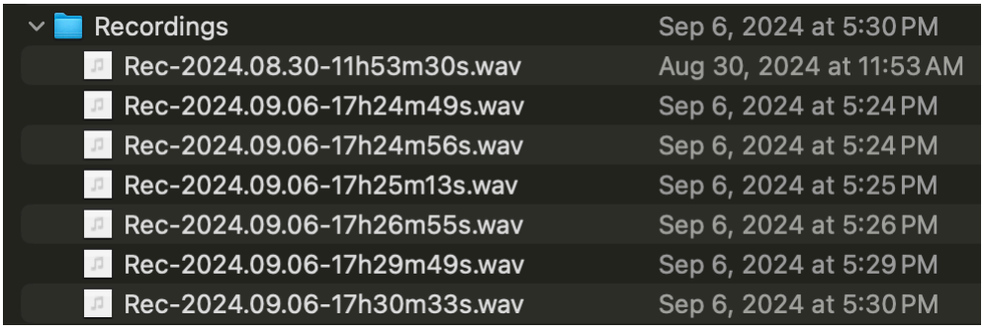
The **Quickrecord** extra also records the audio output of the Max application. You can open the Quickrecord extra by selecting `Quickrecord` from the *Extras* menu. Under the hood, this extra uses the adoutput~ object, which acts as a tap into Max's audio output.

Recording to a directory

By default, Quickrecored will record to a directory. You can press the *Choose a directory* button to select which directory Max should record to. When you do, you'll see the menu illuminate under the *Choose a directory* button.



Every time you press the *Record* button, Max will add a new recording to the directory that you've selected. Max will format the name of the recording to reflect the date and time when you started the recording.



> Remember to press the *Record* button again when you're done to stop the recording. This is necessary to finalize the audio file.

Recording to a file

Click on the *Open a file* button to select a file to which you'd like to record. Once you've picked a file, you'll see the user interface update to show the name of the file that you've chosen. You can

click on the illuminated menu under the *Open a file* button to see where your file will be recorded.



> If you start recording again without opening a new file, Max will record over your original recording.

Multichannel

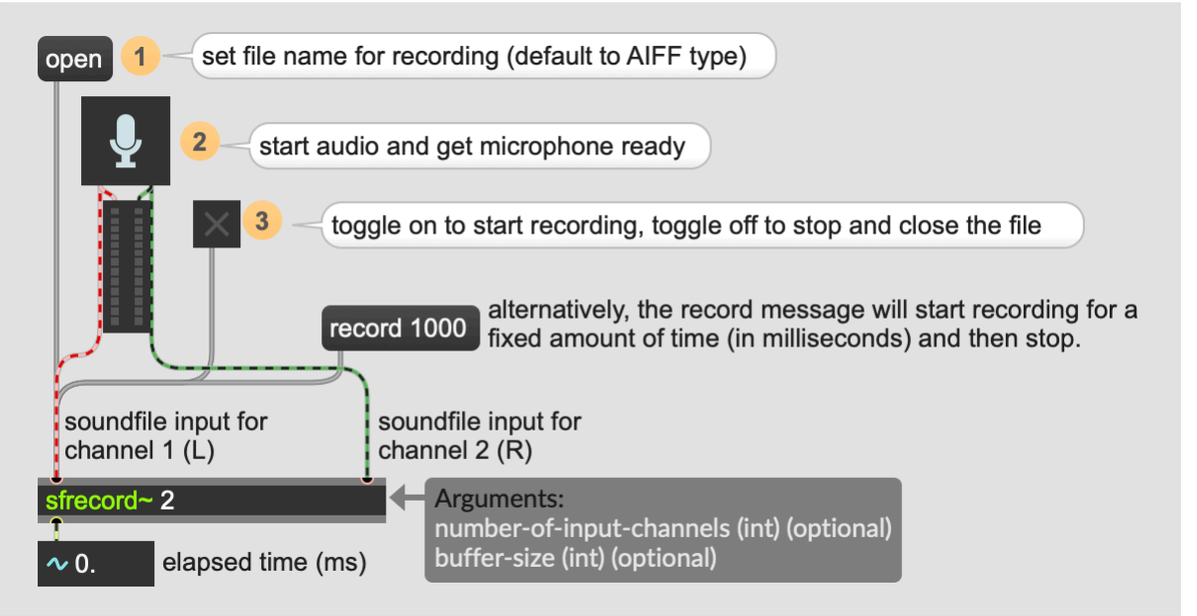
Quickrecord can record up to eight channels simultaneously. However, by default only the first two audio channels are enabled. In order to enable the other audio channels for recording, use the drop down menu under each channel to map each channel from Max to a channel in the recording.



Each channel strip in the Quickrecord view represents a different channel in the recorded audio file. Using the menus in each channel strip, you can map an audio output from Max to each recorded channel. The default configuration maps output channel 1 to recording channel 1, and output channel 2 to recording channel 2. Set any channel to *Off* to disable recording to that channel.

Using sfrecord~

The sfrecord~ object (and the mc.sfrecord~ object) is the programmatic interface to recording in Max. Each sfrecord~ object can record up to 64 channels at once, and multiple sfrecord~ objects can be active at the same time.



Using buffer~

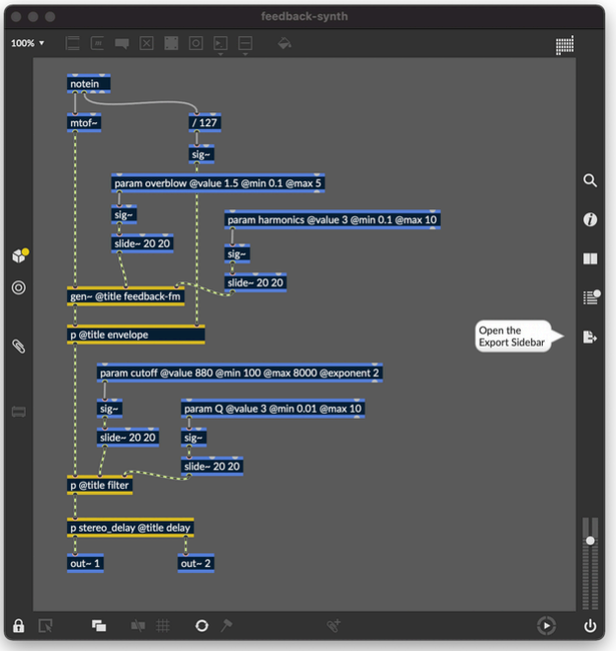
The record~ object can record directly to the contents of a buffer~ object, letting you record to memory in Max without saving your audio to disk. You can also use the poke~ object to write samples directly into a buffer~. Either way, after getting samples into a buffer~, you can send buffer~ a `write` messages to save the contents of the buffer to disk.

Recording Video

For a full discussion of video recording, see this Jitter documentation on Recording Video.

RNBO

RNBO is a visual, graph-based programming language. Similar to Gen, RNBO runs in a special Max object called rnbo~, a subpatcher with its own family of objects based off of Max objects.



RNBO objects in a rnbo~ object

Anything you make in a RNBO subpatcher can be exported, to hardware and software targets like Raspberry Pi, VST and Audio Unit plugins. You can also export to C++ and JavaScript code for use in custom desktop applications and Web Audio contexts.

Most RNBO objects are designed to work the same as their Max counterparts. So if you're familiar with Max, it should be easy to work with RNBO. Gen also works inside of RNBO, which means you can embed your Gen work in a RNBO patcher and export it from Max. Other RNBO features include sample accurate events, host transport sync, MPE & MIDI support, automatic polyphony, and Ableton Link support.

RNBO is a paid add-on to Max. There are several resources available online if you'd like to learn more about RNBO.

Resource	Description
RNBO Website	Main RNBO website
RNBO Guides	RNBO documentation and tutorials
RNBO Examples	RNBO in context, and examples of using RNBO
RNBO Objects	Object reference for RNBO namespace objects
RNBO C++ API	Programming interface reference to link RNBO-exported code in a C++ app
RNBO JS API	Programming interface reference to link RNBO-exported code in a JavaScript app

Sample Accurate Messages

Sample-accurate Events to Audio 71
Signals to Sample-accurate Events 72

When **Overdrive** and **Scheduler in Audio Interrupt** are enabled, certain signal-rate objects can handle high-priority events without losing timing accuracy. In other words, these objects are sample-accurate when handling high-priority events.

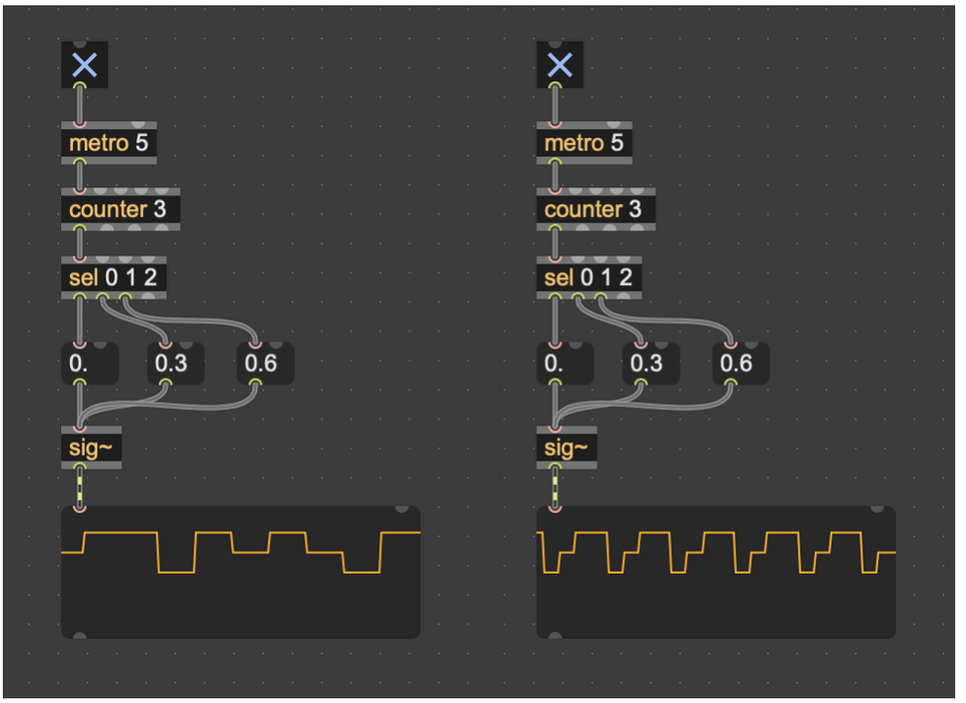
As a reminder, high-priority or scheduled events come from MIDI objects, or from timing releted objects like metro, pipe, and delay. Most other objects generate low-priority events, including UI events like clicking on a button or dragging a slider.

> Low-priority events never have any timing information to begin with, so it doesn't make sense to talk about handling them with sample accuracy.

Sample-accurate Events to Audio

With *Scheduler in Audio Interrupt* enabled, Max handles a chunk of message events before processing each signal vector. For example, when the signal vector size is 256 samples, the scheduler will handle all events scheduled within the next 5.8 milliseconds (equivalent to 256 samples at 44.1 kHz) before Max processes each signal vector.

This is what makes sample-accurate events possible. With SIAI on, signal-rate objects can handle high-priority events at the appropriate time within each audio processing block. As an example, consider a sig~ receiving the messages `0.`, `0.3`, and `0.6`. When SIAI is disabled, sig~ can only handle these messages once per audio block, which destroys the original timing information. With SIAI enabled, precise timing information is preserved, even when converting from message-rate to signal-rate.



On the left, Scheduler in Audio Interrupt is disabled, and sig~ must collapse together all the events received during the same audio block. On the right, SIAI is enabled, and sig~ can preserve timing information.

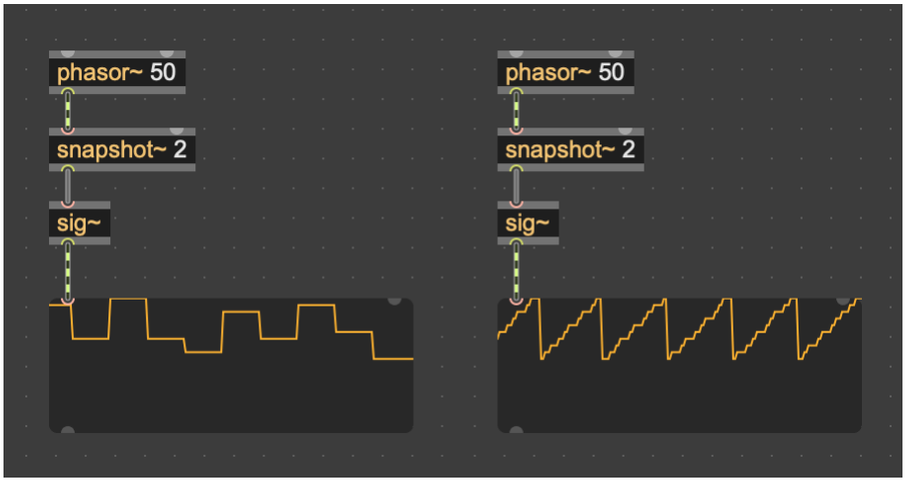
Objects that convert scheduled events to sample-accurate signals

- sig~ will convert events to audio signals sample-accurately
- click~ will start an impulse sample-accurately when it receives a scheduled bang
- line~ and curve~ will start a function sample-accurately
- vst~ receives MIDI events sample-accurately

Signals to Sample-accurate Events

Signal-rate objects that output events can generate sample-accurate, scheduled events when *Scheduler in Audio Interrupt* is enabled. For example, a snapshot~ object will output the value of an incoming signal at a specified rate. When *SIAI* is disabled, snapshot~ can only use the first audio sample in each signal vector, and the effective resolution is limited by the signal vector size. With *SIAI* enabled, snapshot~ can generate multiple events per signal vector, and those events maintain precise timing information.

> With Scheduler in Audio Interrupt enabled, scheduled events are handled before each audio vector is processed. So, events generated by snapshot~ are always delayed by exactly one signal vector. However, their relative timing information is still preserved accurately.



On the left, SIAI is disabled, and snapshot~ and sig~ lose all of the information in the original signal. On the right, with SIAI enabled, snapshot~ and sig~ can actually work together to accurately downsample the signal.

Signal objects that support sample-accurate events

- snapshot~ will sample incoming audio and generate events sample-accurately
- peakamp~ will record peaks in the incoming signal and generate events sample-accurately
- edge~ outputs the zero-to-non-zero signal transitions sample-accurately
- line~ and curve~ output their "done" bang event sample-accurately
- spike~ outputs the time between audio events sample-accurately
- what~ outputs detected signal values sample-accurately
- subdiv~ outputs step values sample-accurately

Colors

Color Palette

Displaying the Color Palette 75
Picking a Fixed Color 77
Managing Color Collections in Palettes 78
Using Dynamic Colors 79
Using the Dynamic Color Picker 79
Exploring Other Color Sets and Themes 81
Using the Color Picker 84
Copying and Pasting Colors 84

The **color palette** lets you select and edit colors used by Max objects. There are two types of colors used in Max, **fixed** and **dynamic**. Fixed colors are specified by RGB color values -- for example, black is 0, 0, 0 -- while dynamic colors are specified by name, and may change dynamically when the Max or Live theme changes.

You can use the color palette to select either fixed and dynamic colors. Certain color attributes in Max objects can also be specified as gradients -- continuous transitions between two fixed colors.

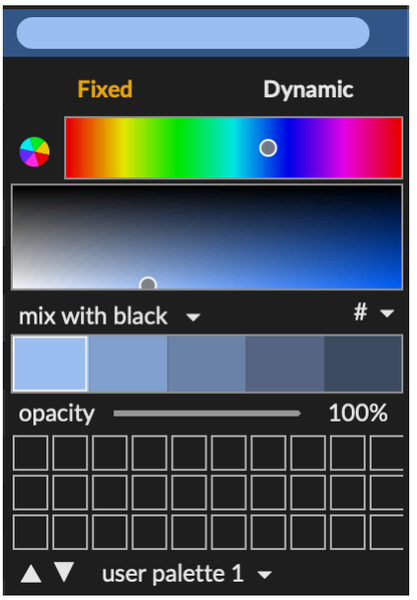
Displaying the Color Palette

The color palette can be accessed in two places, the Inspector or the Format Palette.

Colors in the Inspector

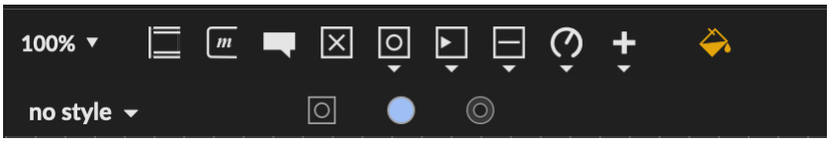
With a user interface object such as a button or toggle selected, open the Inspector. You'll see some color attributes:

Click any of the color swatches next to the name of the attribute. The Color Palette will open.

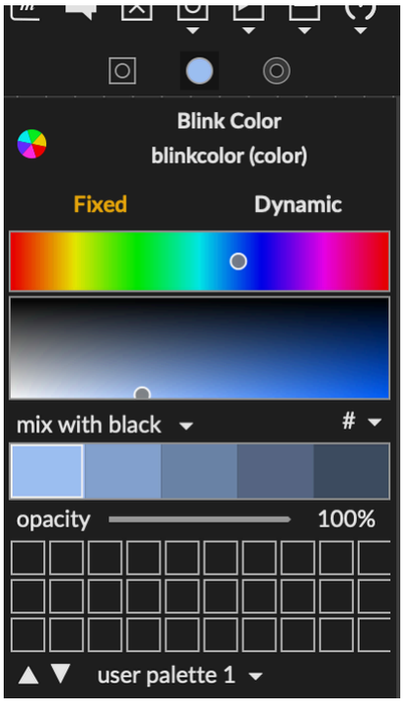


Colors in the Format Palette

Select a user interface object, then click the format palette icon in the top toolbar.



The format palette will open. Click any of the colors shown, and the color palette will drop down to edit that color.



Picking a Fixed Color

Select the *Fixed* tab at the top of the color palette if it is not already showing.

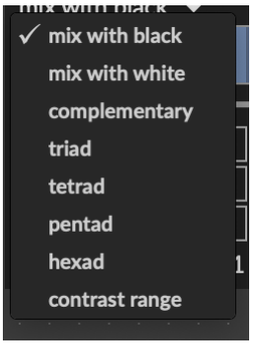


The top control shows the current color within a hue space.



The middle control shows the current color within a brightness and saturation space. Click and/or drag in either space to modify the current color. The Max object's color changes as you drag.

Below the color editors, you'll find some *variation controls*. By default, the chosen variation is *mix with black*. When you click on *mix with black*, you'll see a menu with other color variations.



Here's Pentad, which creates five variations of the selected color according to hue.

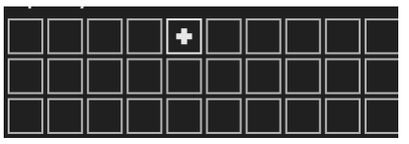


Click on any color variant to make it the selected color.

If you're not happy with your changes, choose *Undo* from the Edit menu to go back to the previously chosen color.

Managing Color Collections in Palettes

Below the color variation control, there is a 10 x 3 grid of squares. By default, the grid is empty, ready for you to save your own colors. As you move the mouse over an empty square, a plus sign appears. Click in a square to save the current color in a palette.



To save the current color over an existing color, hold down the shift key and click on the color you want to replace.

There are three **user palettes** for storing your own color collections. You don't need to save changes to the palette; the updated colors will be loaded every time you launch Max.

Saving Palettes

To export the current state of a user palette for sharing, click on the current palette name (`user palette 1` for example), then choose *Export user palette 1...* from the menu. Files are saved with the extension `.maxpalette`.

Built-in Palettes

The palette menu beneath the grid lists about a dozen read-only paletes built into Max. Choosing a named palette will replace the colors in the grid.

Importing Palettes

To import previously exported palette, choose *Import...* from the palette menu and select the desired `.maxpalette` file.

Using Dynamic Colors

Dynamic colors are named elements of color themes you can use to ensure the user interface elements in your patchers or Max for Live devices are coordinated with the larger environment. When the user changes the theme, dynamic colors update to reflect the color values in the theme.

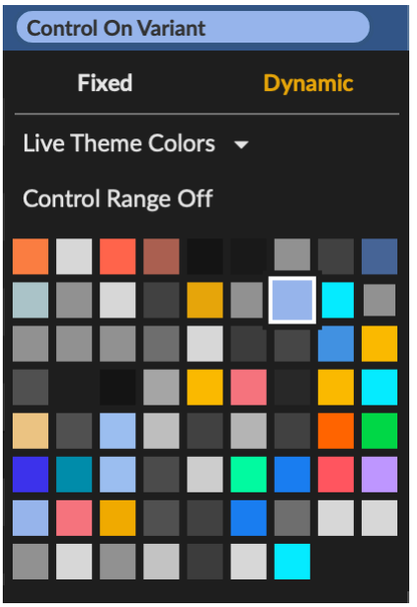
The live.dial object as well as other UI objects whose names begin with `live` use dynamic colors within a Live theme so'll they'll fit in with the rest of the Live UI. You can assign dynamic colors to any object however.

Using the Dynamic Color Picker

Click on the *Dynamic* tab of the color palette.

If the current color is already dynamic, it will be highlighted within its color set (for example, *Max Theme Colors*). If the current color is fixed, the *Live Theme Colors* will be shown. As you mouse over

the grid of colors, you'll see the color name displayed above the grid. Click on a color to choose it.



If you prefer to choose the dynamic color by name, click on the name above the grid to see a menu of all colors in the set by name.

About Live Colors

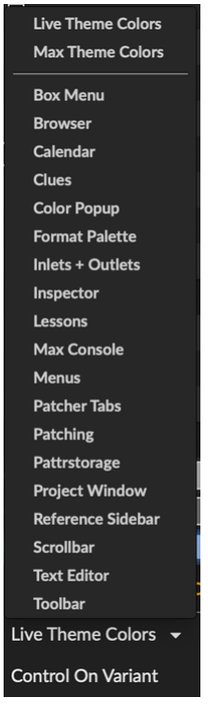
There are about 70 entries in the set of Live Theme Colors. The menu can be a bit overwhelming, so it can help to learn the most commonly used colors in objects. live.dial makes use of 11 colors of which the following are significant:

- **Control Range On** -- the range of the control when active
- **Control Range Off** -- the range of the control when inactive
- **Control On** or **Control On Variant** -- the control's value when active
- **Control Off** -- the control's value when inactive
- **Border Color (Focus)** -- the border around the control when it has keyboard focus
- **Control Needle On** -- the control needle when active
- **Control Needle Off** -- the control need when inactive
- **Text Color** -- the text color

Many of the other theme colors in the menu are used in the Live UI for indicating selection or a special modes, so they could potentially be confusing if used for controls in a UI.

Exploring Other Color Sets and Themes

To choose another set of dynamic colors, click *Live Theme Colors* and choose a set from the menu.



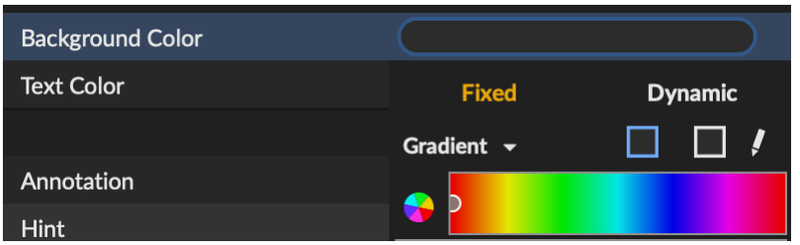
Most of the time you'll want to use either Live Theme Colors or Max Theme Colors; these two items appear at the top of the menu. The other sets represent different colors used in the Max interface. For example, *Max Console* is the collection of colors used to display text and backgrounds in the Max Console.

Using Gradients

A small number of Max user interface object background colors can be specified as gradients. The background of a message box is one example. The **Gradient** editor in the color panel is only available if the color you're editing can be a gradient.

To use the gradient editor:

- Make a new message box
- Show the inspector for the message box
- Click on the color swatch for `Background Color` to edit it. The color palette is in gradient mode.



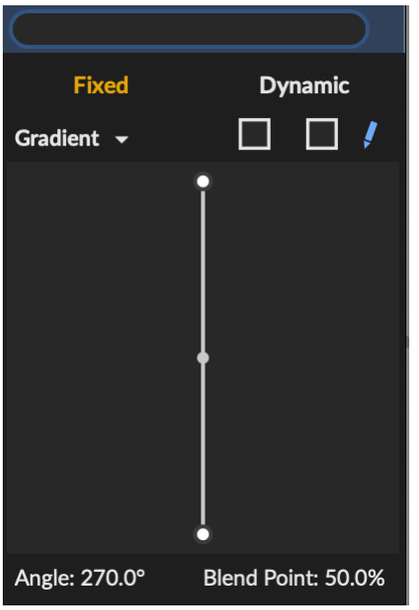
The gradient editor has three components. First, you can change either of the two colors that make up the gradient using the *active edit color selection*:



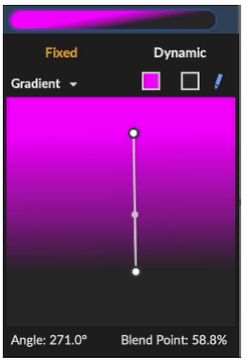
Second, you can edit the gradient parameters by clicking on the pencil icon.



The gradient editor will appear.

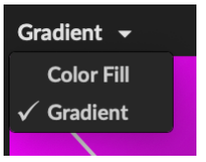


The following animation demonstrates the basic operation of the gradient editor, setting the gradient angle, the blend point, and the size of the blend region.



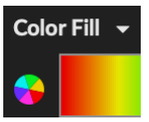
With gradient compatible background color attributes, you're not obligated to use a gradient.

To switch to a solid color, click on *Gradient* and choose *Color Fill* from the menu.



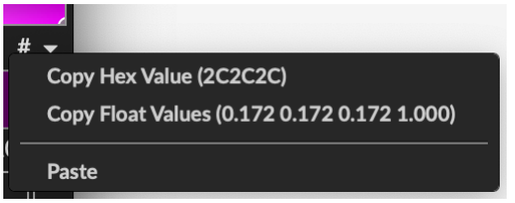
Using the Color Picker

A final option for editing fixed colors is the standard *Color Picker* accessed by clicking the circular color icon near the top of the color palette.



Copying and Pasting Colors

To moving color values from the color palette to other applications, click **#** above the color grid.



You can Copy the current color to the clipboard in either hex (*Copy Hex Value*) or float (*Copy Float Value*).

To import a color value into the color palette from another application, choose *Paste*. The color palette accepts values as four floating-point numbers or a hex string, which can optionally be preceded by a pound sign (`AEFC06` or `#AEFC06`).

Color Themes

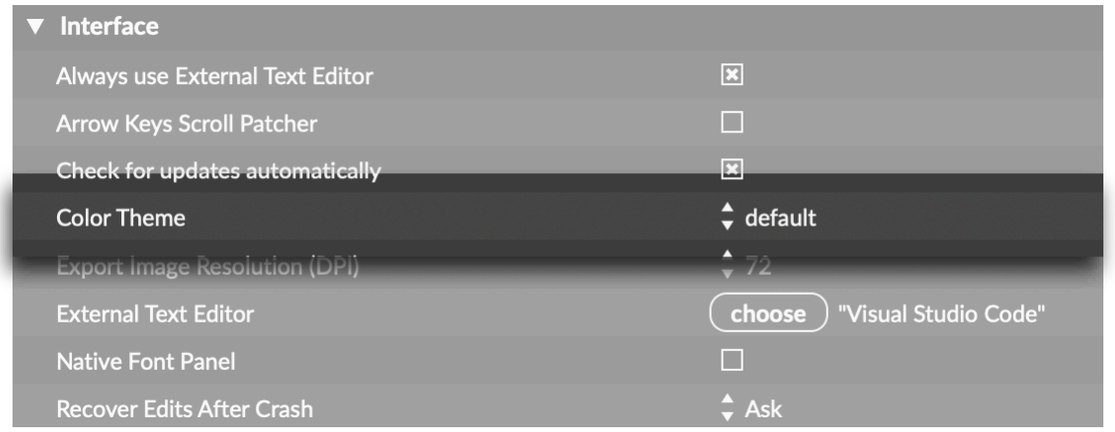
Choosing a Color Theme 85
Custom Color Themes 86
Designing Robust Patcher Interfaces 86
Themes and Max for Live 87
Accessing Theme Colors from JavaScript 87
Color Theme Format 88

A Max **Color Theme** defines the colors of elements of the Max user interface, like the toolbar color, as well as default values for object color attributes. You can choose the color theme that you like best, or define your own custom color theme.

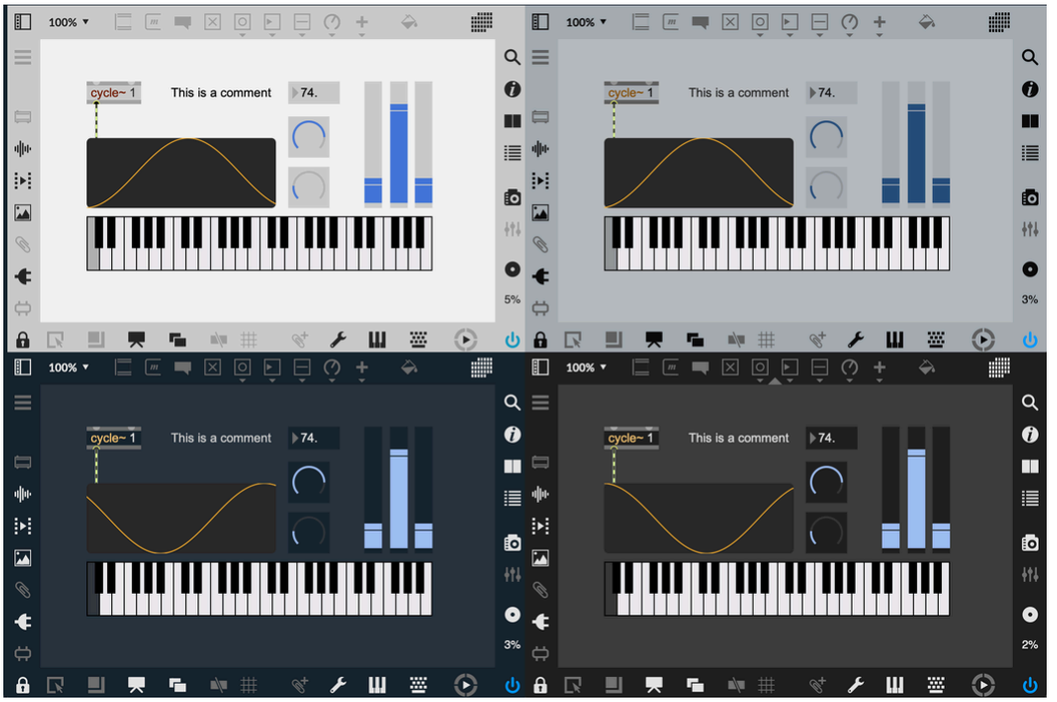
Note that a Max Style will take precedence over the current theme, if a custom style is set.

Choosing a Color Theme

You can change the colors of the Max interface by choosing a new theme. In the application preferences (Max > Preferences...), under the *Interface* the *Color Theme* option lets you select your preferred theme.



Max ships with several color themes, including light and dark themes, as well as themes from previous versions of Max.



clockwise from top left: light, midlight-steel, default, dark-lazurite

Custom Color Themes

If you want to create a custom Max theme, it must be part of a Max Package. Create a custom theme by creating a `.maxtheme` file, then placing that file in the `interfaces/themes` directory inside your package folder. Max will automatically make your custom theme available alongside the built-in themes in the application preferences.

The easiest way to make a custom color theme is to start from one of the existing themes. On macOS these can be found in the application bundle—right-click on the Max application, select "Show Package Contents" and navigate to `Contents > Resources > C74 > interfaces > themes`. On Windows these can be found in `Program Files > Cycling '74 > Max 8 > Resources > interfaces > themes`.

Designing Robust Patcher Interfaces

If you want to share your patches with other people, you should take the time to make sure that your patches are theme-aware. There are simple guidelines that you can follow if you want your patch to look good, even when the user switches between themes. This is especially important if you're trying to share a Max for Live device, as that device will follow whatever theme Live uses.

In order to make a patcher properly themed for Ableton (or Max) and robust in regard to theme changes, one of the following options should be followed for each UI element in your device (or patch):

- Use the default colors for all of a given UI element's colors. This should always follow the theme according to default theming behavior. If you require always dark or always light colors, please use appropriate dynamic colors that match these needs (see option #2).
- Use appropriate dynamic colors for all colors in any given UI element. This is especially important for any foreground and background colors. This allows for robust theme following, with different rules than the standard theme following behavior.
- Use fixed custom colors for all colors in any given UI element. This is especially important for any foreground and background colors. In this case, the colors will not adapt based on themes, but they will always use the authored colors and look the same under all themes.

Themes and Max for Live

The Max preference *Follow Live Theme* determines whether Max will update its own theme when the Live application changes themes. By default this preference is *On*. An individual user who wishes to retain legacy theme behavior, can turn this setting *Off*. And for Max users that wish for the last used Live theme to persist when using Max without having been launched via the Max for Live edit button, may set this preference to *Persist*.

It is generally not recommended to change the Max theme while running as the Max for Live editor with Follow Live Theme set to *On* or *Persist*. This leads to a confusing fight between custom Max themes and the Live theme being followed. If you find yourself in this situation. It is recommended to quit and restart Max (or simply change the theme in Live), to restore expected behavior. In general for testing Max for Live devices with Light and Dark themes, it is recommended to have Follow Live Theme set to *On* and to change the theme inside Live.

Accessing Theme Colors from JavaScript

Inside a JavaScript object like v8, v8ui, or v8.codebox, you can access theme colors using the max.getcolor function. For example, `max.getcolor("live_lcd_bg")` will return the color associated with the *LCD Background* color. If you're using this color with a custom UI object, it's recommended to fetch this color in your `paint()` method. Since `paint()` is called automatically whenever the theme changes, this guaranetees that your `paint()` function will always be called with up-to-date colors from the current theme.

It's also possible to set the value of a color in the current theme using the max.setcolor function. You could use this function to update the current theme dynamically. Note that this function is only available in **v8**.

> The JavaScript API max.getcolor and max.setcolor functions use theme color *ids*, while the themecolor object can use either ids or theme color display names.

Color Theme Format

A Max color theme is a set of Dynamic Colors combined with default values for the current Style. Together, these two sets of values define colors for the Max application as well as Max objects.

The `.maxtheme` file is simply a JSON file with two properties, `colors` and `styledefaults`. The `colors` property defines the values of Dynamic Colors within that color theme. This is a dictionary that maps identifiers to specific color values. When the application needs to know what color to draw a UI element, it can look in this dictionary to see what color would be consistent with the current theme.

```
// Example of the "colors" section of a .maxtheme file
{
  "colors": [
    {
      "id": "alignmentguide", // The internal name of the color
      "oncolor": [0.737255, 0.466667, 0.219608, 1.0], // RGBA format
      "category": "Patching",
      "label": "Alignment Guide" // The display name of the color
    },

    {
      "id": "assistance_background",
      "oncolor": [0.843137, 0.835294, 0.796078, 0.94],
      "category": "Patching",
      "label": "Assistance Background"
    }

    // ... more colors here
  ]
}
```

The `styledefaults` are similar, but work with the current Style to determine the final color of an object. These specify the default color of an object if no other color is defined.

```
// Exmaple of the "styledefaults" section of a .maxtheme file
{
  "styledefaults": {
    "bgcolor": [0.2, 0.2, 0.2, 1],
    "color": [0.807843, 0.898039, 0.909804, 1],
    "elementcolor": [0.34902, 0.34902, 0.34902, 1],
    "accentcolor": [0.501961, 0.501961, 0.501961, 1]

    // ... more colors here
  }
}
```

If an object needs to decide what color to draw, it first looks to its defined attibutes. For example, if a `button` object needs to draw its background, it looks to its `bgcolor` attribute. If the user has not defined a custom value for this color, it then looks at the object's style for a value for that color. If the style doesn't contain a value for that attribute, the object checks the patcher's style next. If the patcher doesn't have a style, or if the style doesn't define a color for the given attribute name, then the object will fetch the value from the current global color theme—this is what is specified by `styledefaults`.

Dynamic Colors

Max for Live 91
Selecting a Dynamic Color 91
Accessing Dynamic Colors from JavaScript 94
Limitations 94

Dynamic Colors automatically follow the active Max color theme, rather than remainining fixed. An object or a patcher can have a **Fixed Color**, like `1.0, 0.0, 1.0, 1.0`, which will not change when the current **Color Theme** changes. However, a color can also be specified by name. In this case, the actual color as it appears on screen is dynamic, and will change when the current color theme changes. You can get and set colors in the current theme using the themecolor object. You can also get dynamic colors using JavaScript.

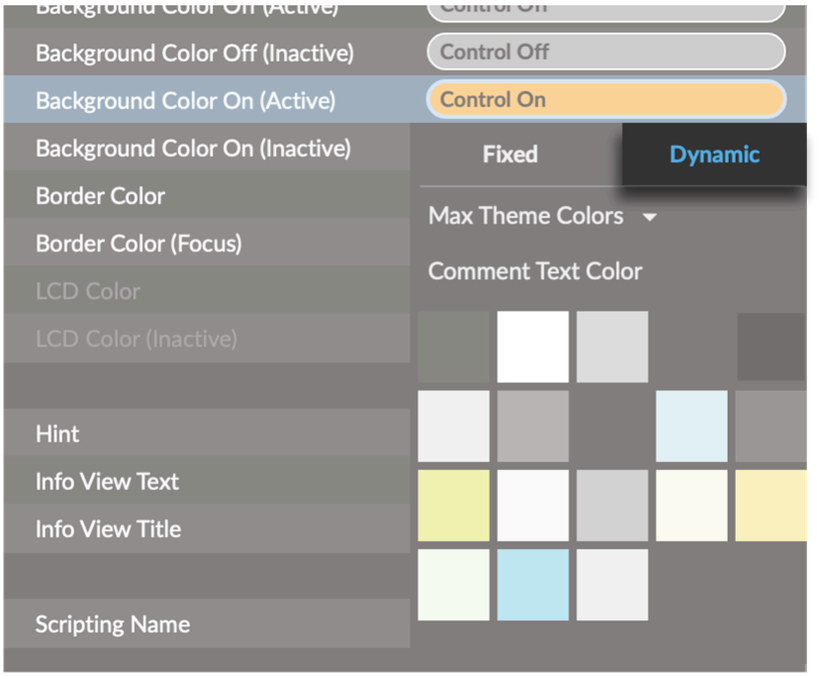
Max for Live

By default, dynamic colors are disabled for most Max objects, but are enabled for most Max for Live objects. When used in Max for Live, the dynamic colors of the Live category follow the active Live color theme as chosen in Live's Preferences. The Live dynamic colors and their values are also listed in the live.colors object's help patcher.

Selecting a Dynamic Color

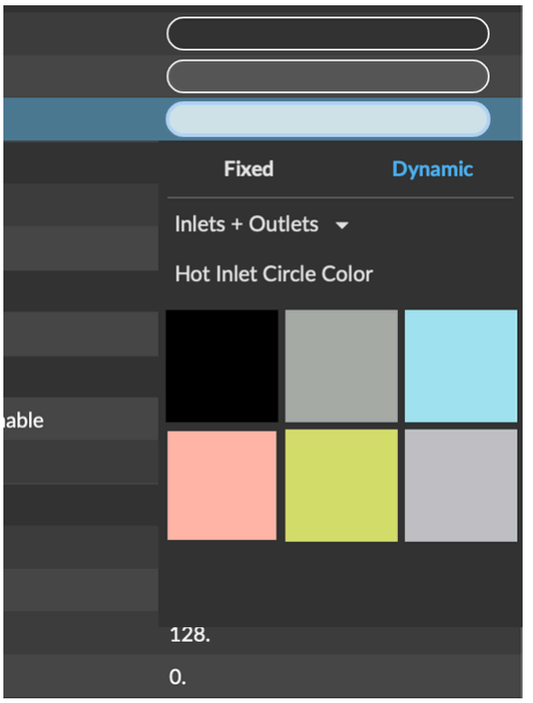
Using the Inspector

To set a dynamic color, open the Inspector for an object and find the color attribute you want to change. Click in the value column to open the color picker. Select the *Dynamic* tab at the top of the color picker.



Click the drop-down menu above the color swatch. From here you can select a color based on its dynamic color name. The first dropdown groups colors by category (e.g. Max Theme Colors, Inlet + Outlet), while the second dropdown lets you pick a specific color.

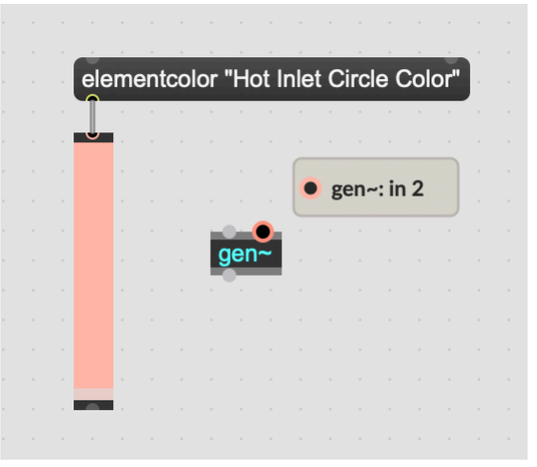
> Hold `alt` (Windows) or `Option` (macOS) while clicking the dropdown to display the full range of available dynamic colors.



Dynamic color picker showing the colors for Inlets + Outlets

Using a Message

Send an object a message like `elementcolor "Hot Inlet Circle Color"` to set the value of that attribute to a dynamic color. Since the name of a dynamic color might be multiple words, you may need to enclose the name of the dynamic color in quotes.



Someone set the elementcolor of this slider to be the same as a hot inlet.

> A full overview of all Dynamic colors and their names can be found in the "view all" tab of the themecolor object help patcher.

Accessing Dynamic Colors from JavaScript

Inside a JavaScript object like v8, v8ui, or v8.codebox, you can access dynamic colors using the max.getcolor function. For example, `max.getcolor("live_lcd_bg")` will return the color associated with the *LCD Background* color. If you're using this color with a custom UI object, it's recommended to fetch this color in your `paint()` method. Since `paint()` is called automatically whenever the theme changes, this guaranetees that your `paint()` function will always be called with up-to-date colors from the current theme.

It's also possible to set the value of a dynamic color in the current theme using the max.setcolor function. You could use this function to update the current theme dynamically. Note that this function is only available in **v8**.

> The JavaScript API max.getcolor and max.setcolor functions use theme color *ids*, while the themecolor object uses theme color display names.

Limitations

Dynamic colors do not work with styles. Choosing a style to override a color will not shut off dynamic colors, and dynamic colors can not yet reliably be part of a style.

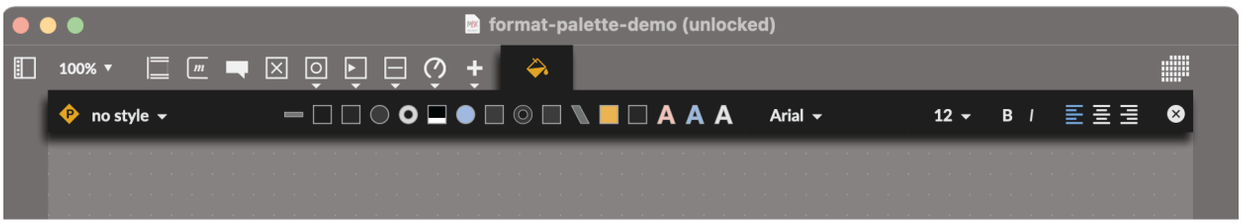
Format Palette

Opening the Format Palette 95
Patcher-level Formatting 96
Styles 96

The **format palette** is a specialized interface for customizing the look of your patcher. It lets you change object and patch cord colors, font size and style, syntax highlighting colors, and other attributes related to appearance. The format palette can configure specific objects, or it can apply patcher-level formatting, affecting all objects in the current patcher.

Opening the Format Palette

Open the format palette by clicking the *Format* icon in the top toolbar.



Click the Format icon to expose the format palette.

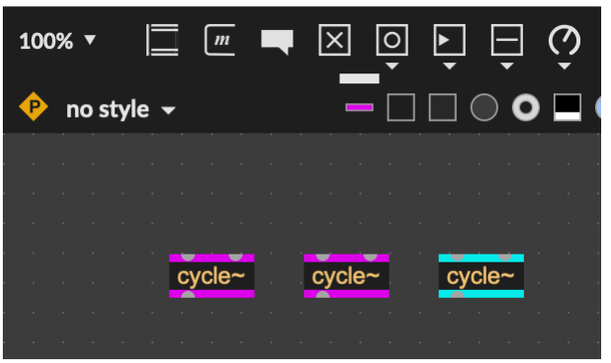
Along the top of the format palette, you'll see all of the configuration options for the current selection. Hover over a configuration option to see the name of the option.



Hover over a format option to see its name.

Patcher-level Formatting

With no objects or patch cords selected, the format palette shows patcher-level formatting options. Any change you make here will apply to all objects in the current patcher, unless those objects override those changes with individual settings.



Patcher-level formatting for the Object Accent Color changes the style of all objects in the patcher, except for those with individual settings.

> You can hide patcher-level formatting options by clicking the Hide Patcher-Level Format icon in the top-left.

Patcher-level formatting can be used to change the syntax highlighting for the current patcher, with options to change the color for object names, arguments, attributes, and attribute arguments.

> If a configuration option is missing from the format palette, you may need to expand the size of the window to make room for more options.

Styles

The format palette can be used to apply, create, and modify styles. See the Styles guide for an in-depth discussion.

Styles

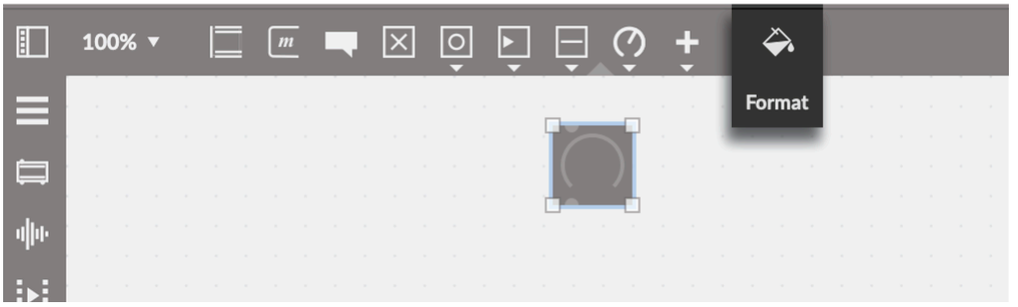
Applying a Style to Objects 98
Applying a Style to a Patcher 99
Accessing Style Colors from JavaScript 100
Creating a New Style 101
Removing a Style from an Object 103
Adding a Style to your Library 104
Removing a Style from your Library 105
Modifying a Style 107

A **Style** is a collection of colors and fonts used by Max user interface objects. You can apply a style to a Max patcher as a whole or to an individual object. A patcher-level style establishes defaults for colors and/or fonts that can be overridden by styles or colors applied to individual objects. You can define your own personal library of styles or make use of "factory styles" that come with Max. You can also use styles in connection with Max templates to create your own unique default patching environment.

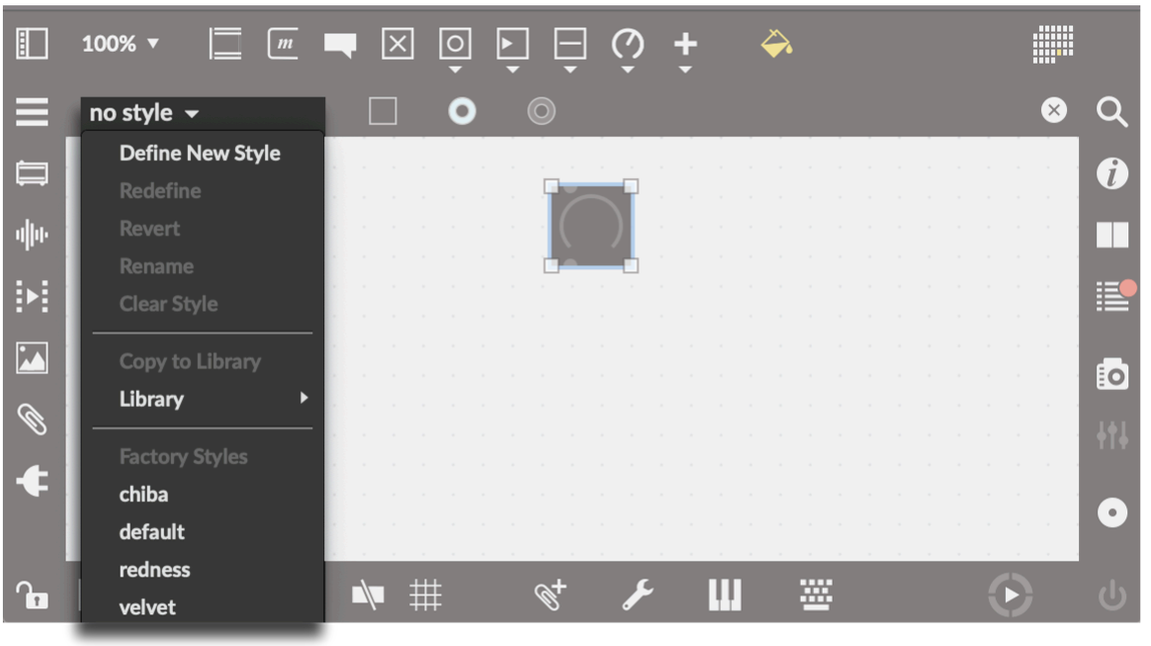
Styles are created and managed using the Format palette.

Applying a Style to Objects

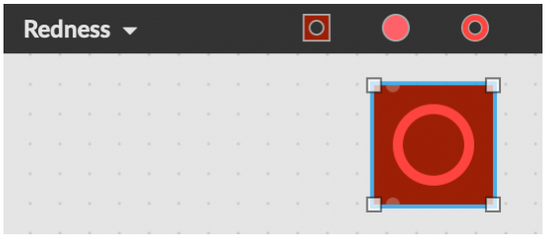
Select any number of objects, then click the *Format* button to open up the Format palette toolbar.



Click on the style menu in the top toolbar to to show the available styles.

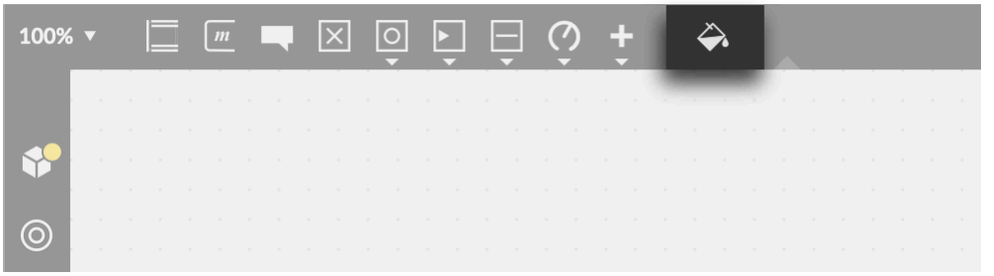


The style menu lets you choose from styles you have created and added to your own Max library (available using the Library entry), styles recently used in the current patcher (which will be listed just below the Library entry), or from Max Factory Styles (located at the bottom of the menu). Choose any of these styles from the pull-down menu. The menu will display the current style, and that style will be applied to your objects.

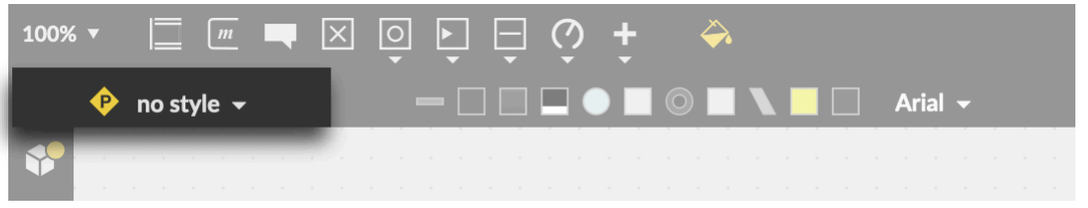


Applying a Style to a Patcher

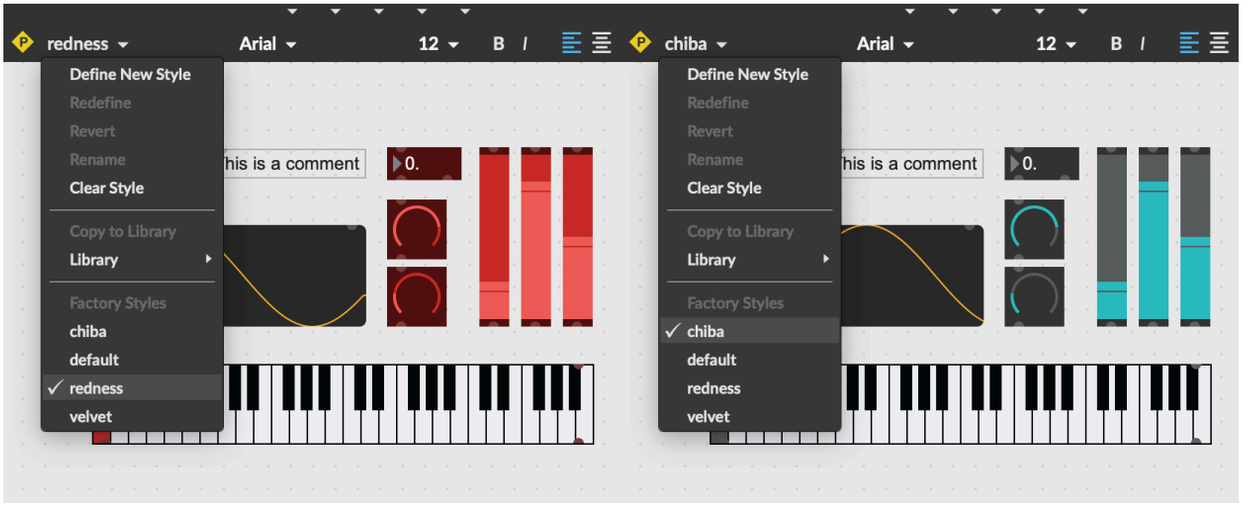
It's also possible to apply a patcher-level style. A patcher-level style establishes default colors and fonts for new objects. To apply a style to the whole patcher, first make sure no object is selected by clicking in the background of the current patcher, then click on the *Format* button to open the Format palette toolbar.



Once more, click on the style menu on the left-hand side of the toolbar to show the style menu.



Choose a style to apply a patcher-level style. If an object is using its default colors and fonts, the patcher-level style will replace the default values. This lets you change the look of a whole patcher at once.



Accessing Style Colors from JavaScript

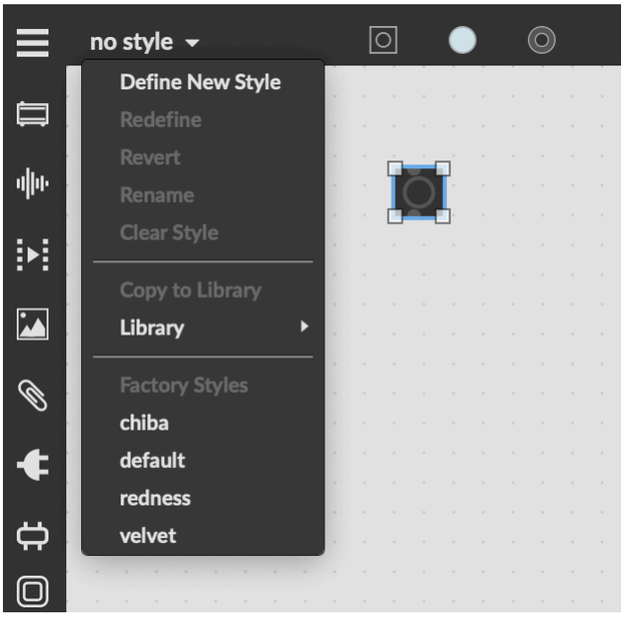
Inside a JavaScript object like v8, v8ui, or v8.codebox, you can access style colors using the associated properties on the `this.patcher` object. For example, `this.patcher.bgcolor` will

return the background color defined in the currently active style (or inhereted from the current theme, if no style is active). If you're using this color with a custom UI object, it's recommended to fetch this color in your `paint()` method. Since `paint()` is called automatically whenever the style changes, this guaranetees that your `paint()` function will always be called with up-to-date colors from the current style.

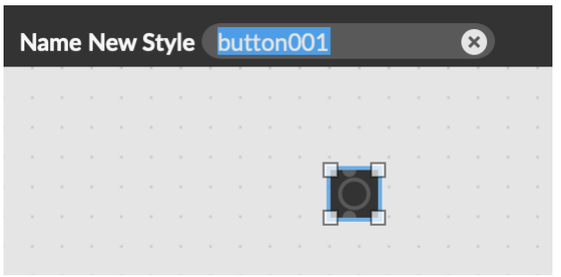
Creating a New Style

In the same way that you can apply a style either to a selection of objects, or at the patcher level, you can also create a style in the same way. In general, it's easiest to create a style based on a selection of objects.

Select the object (or objects) that you'd like to use as the basis of your new style, then click on the *Format* button to open up the Format palette toolbar. Configure the object as you want it to appear in your new style, then click on the style drop down and select *Define New Style*.

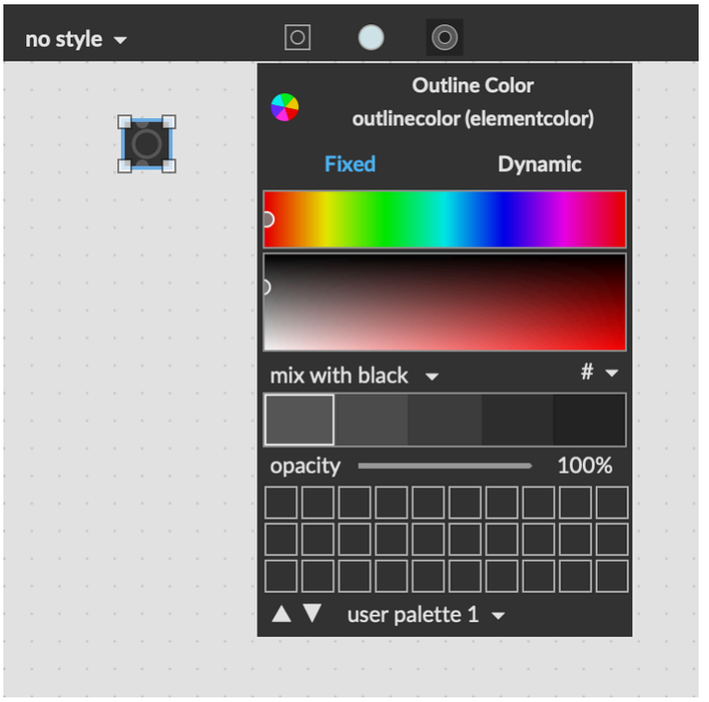


This will open a dialog box that lets you name your new style.



Attributes included in a style

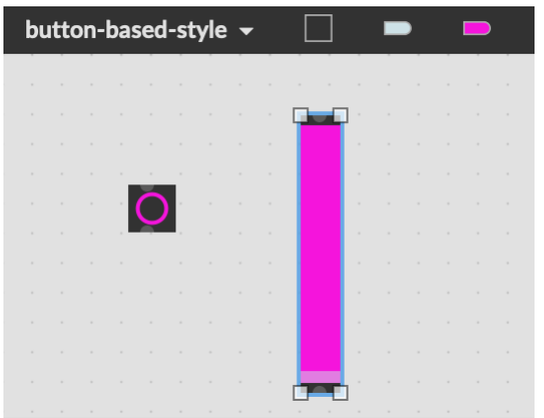
Not all attributes that affect the appearance of an object can be included in a style. It's easy to see which attributes will be included by looking at the format palette, which shows all the attributes that can participate in a style. By clicking on each icon in the toolbar, you can see the name of each attribute, as well as the name of the color theme value that affects that attribute. For example, a button object has an attribute called `outlinecolor`, and from the format palette you can see that this is affected by the color theme value `elementcolor`. This helps you see how the style of one object can apply to multiple objects, when creating a style from a single object.



> It's also important to note that a style will only include attributes that you've changes from their default. If you leave an attribute at its default value, then any style defined from that object will not include that value.

Single-object styles

When using a single object to define a style, Max will use the color theme value of a particular attribute to determine how to style other objects based on the original objects. Using the example of a button object, since the `outlinecolor` attribute is affected by the color theme value `elementcolor`, the `outlinecolor` of a button will affect the "Off Color" of a slider object with the same theme.



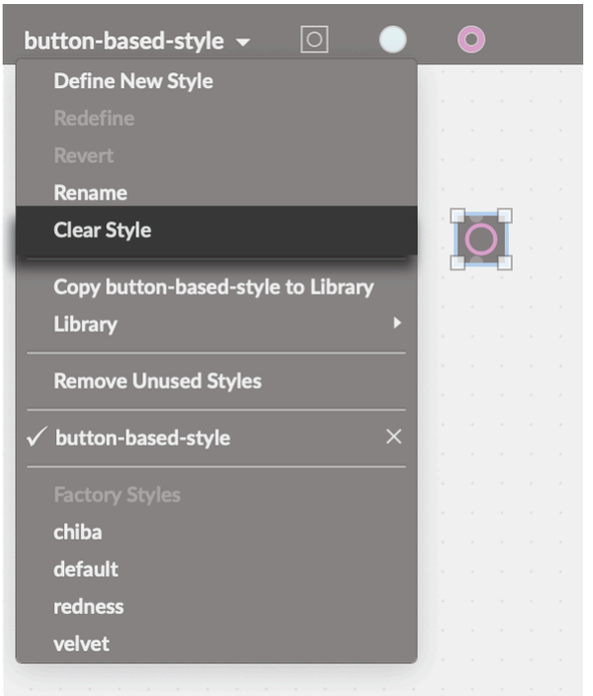
In the above, the appearance of the slider with the style "button-based-style" is determined by the appearance of the button.

Multi-object styles

When using multiple objects to define a style, Max will only apply the style to objects of the same class as the original objects. If you define a new style based on a button and a slider, then buttons will have the same appearance as the button, and sliders will have the same appearance as the slider, but other objects will be unaffected.

Removing a Style from an Object

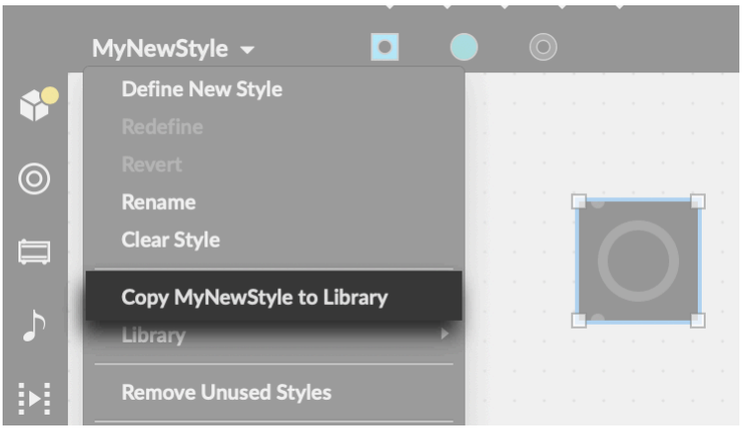
Select the object, then open the style menu and select "Clear Style".



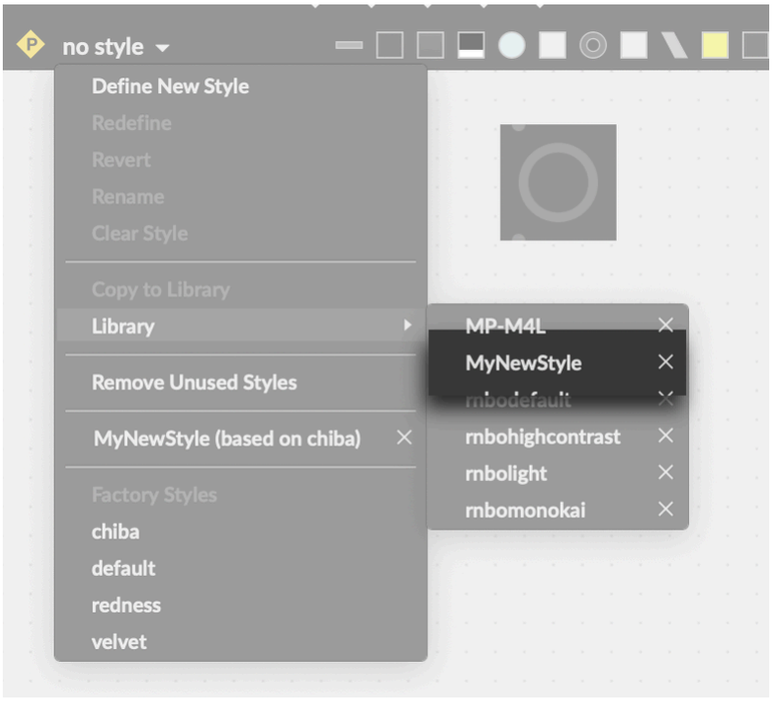
Adding a Style to your Library

A Library Style is dependency-free, saved to disk, and does not rely on any other styles.

- To add a Library Style, click on an empty section of a patcher, then click on the Format Button to open the Format palette toolbar. Select the style for your patcher that you want to save to the Library.
- Click on the style menu and choose *Copy* `<style name>` *to Library* from the pull-down menu.



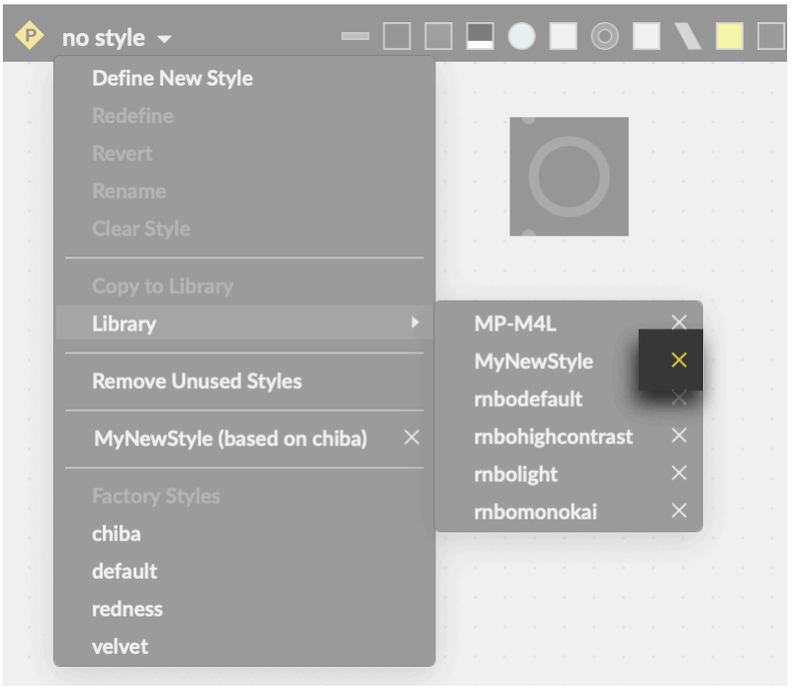
The style you have selected will now appear in the Library tab of the Style menu



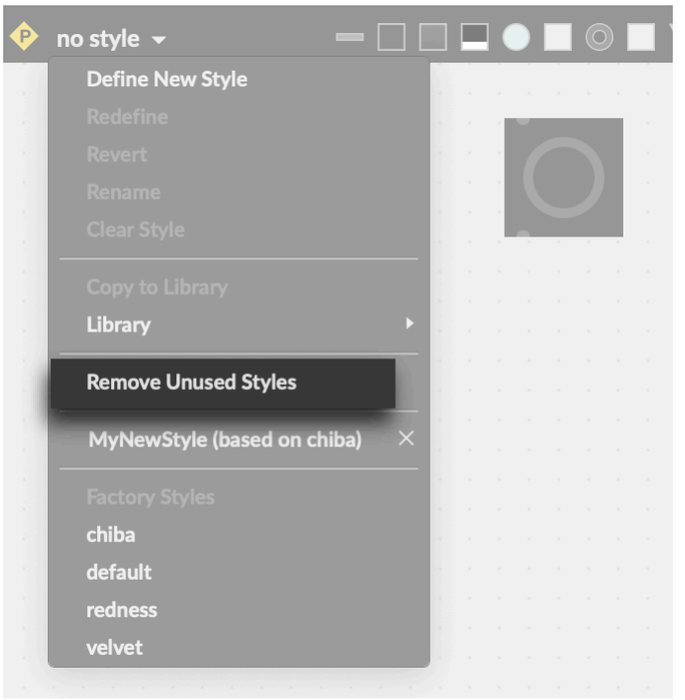
Library styles can be found at `~/Documents/Max 8/Styles` on a macOS, and `%HOMEPATH%\Documents\Max 8\Styles` on a Windows system.

Removing a Style from your Library

- Click on an empty section of a patcher, then click on the Format Button to open the Format palette toolbar.
- Click on the style menu on the left-hand side of the Format palette toolbar and hover over the Library menu entry. This will display all of the styles you have added to the Library.
- Click on the deletion button to remove a specific style.

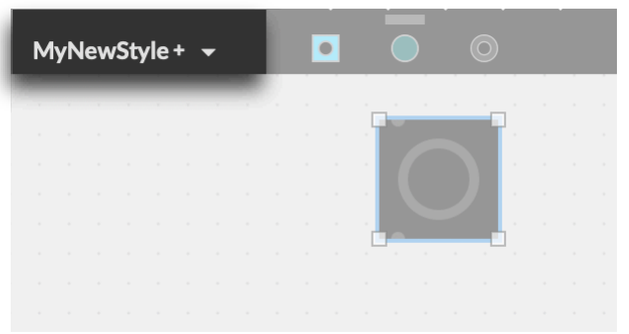


- To remove all styles that are not being used in a patcher, select *Remove Unused Styles* from the style menu. This will removed all unsed styles that may have been inherited from copying and pasting a different user's patcher, or opening a different user's patcher on your machine.

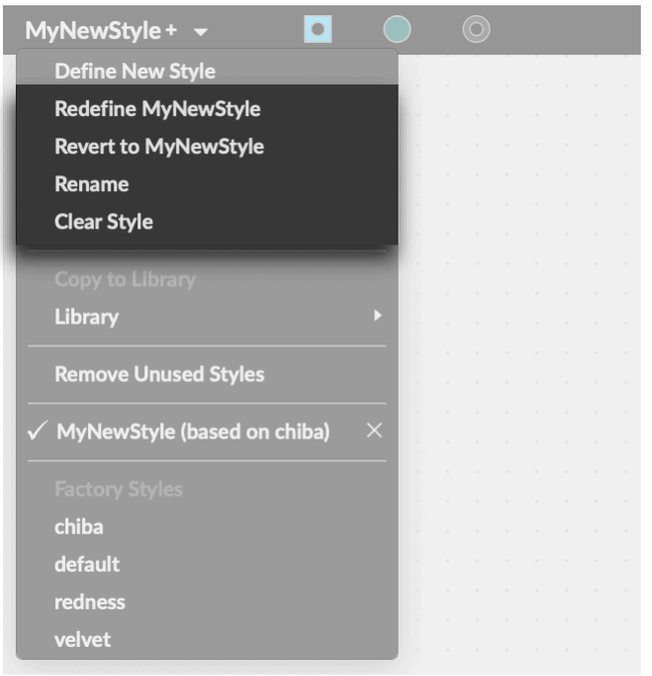


Modifying a Style

- Click on an empty section of a patcher, then click on the Format Button to open the Format palette toolbar. Use the Format palette to make your desired changes to the object. When you do, you will notice that a plus sign appears alongside the current style in the style menu to indicate that the style has been changed.



- Click on the style menu on the left-hand side of the Format palette toolbar. Making changes in an object with a style applied to it will enable several menu options in the pull-down menu.

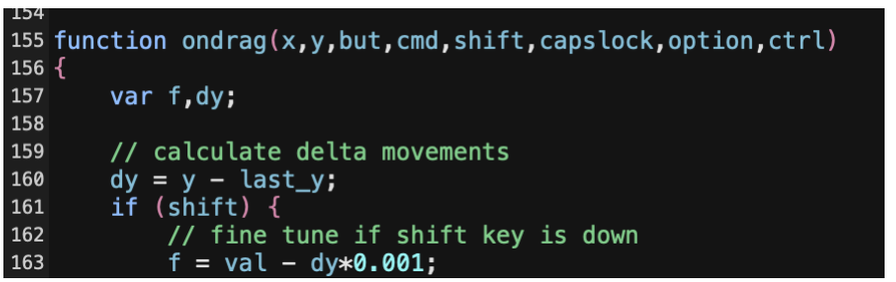


- Choose *Redefine* to redefine the current style to match your changes.
- Choose *Revert* to revert to the style you originally applied to the object.
- Choose *Rename* to rename the style you originally applied to the object. When you choose this option, the style dialog will appear. Type in the name of your style. When you hit a carriage return, the style will be created and be added to a list of patcher styles.
- Choose *Clear Style* to return the attributes of the object to their default state (precisely as if you had selected Set to Default Values from Max's Object menu).

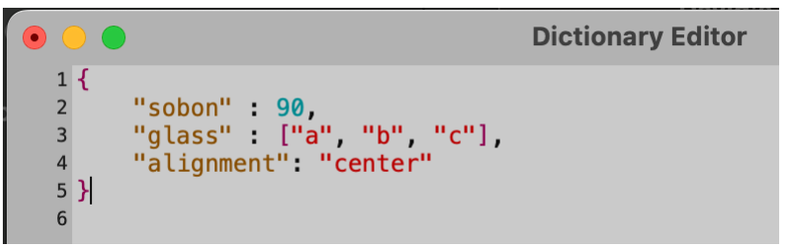
Syntax Coloring

Enabling Object Box Syntax Coloring 109
Customizing Object Box Syntax Colors 111

Syntax Coloring uses different colors for a word or number based on the type of token it is in a programming or data description language. Most modern text editors used for programming feature syntax coloring and Max is no exception. The standard text window will adjust text colors based on the theme and file type. Here are some examples:



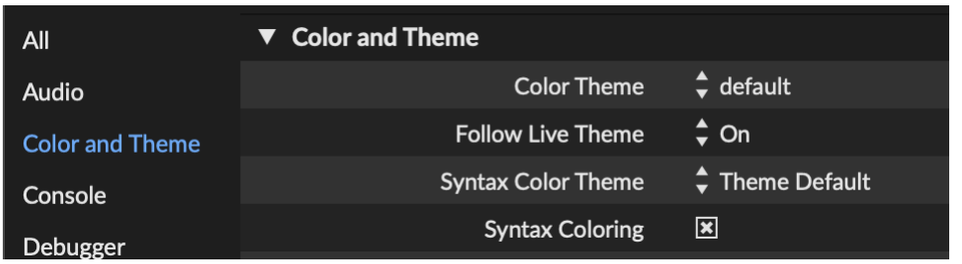
*JavaScript in the **default** theme*



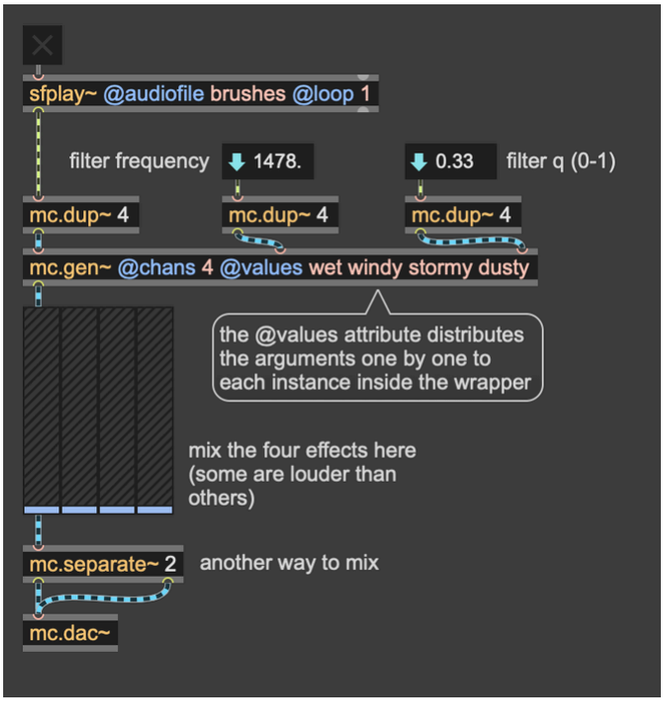
Dictionary Editor in the midlight-ash theme

Enabling Object Box Syntax Coloring

To apply syntax coloring to the text in object boxes in a patcher, enable **Syntax Coloring** in the **Color and Theme** tab of the Preferences window.



This applies the current theme's syntax colors to object box text.



In the above example, four colors are used to color the text of object boxes.

- the first word in the object box is the *object name*



- words starting with @ are *attribute names*



- words or numbers after the object name but before the typed-in attributes are *object arguments*



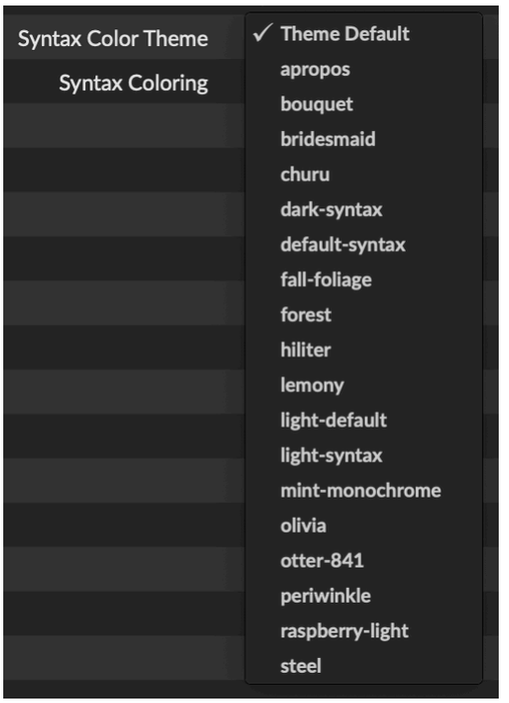
- words or numbers after the object name but before the typed-in attributes are *attribute arguments*



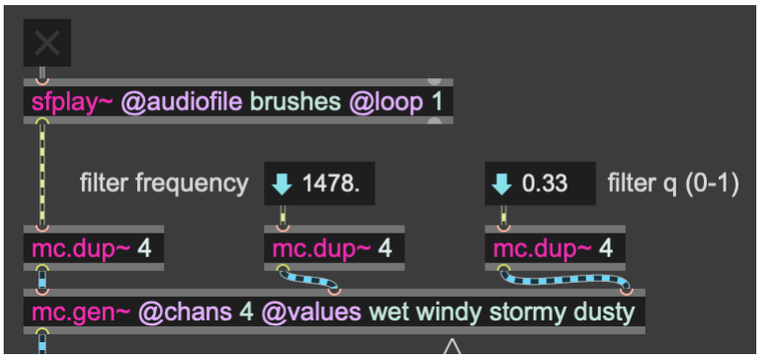
Customizing Object Box Syntax Colors

You can override object box syntax colors set by themes in two ways:

- Select a **Syntax Color Theme** other than *Theme Default* in the **Color and Theme** tab of the Preferences window.



The various themes in the menu will not be equally legible with all themes, but they may work better for you than the default theme colors. Here's the **olivia** Syntax Color Theme used with the **default** Color Theme.

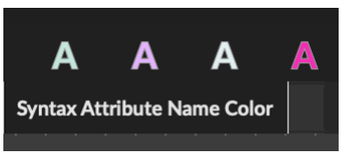


Note that Syntax Color Themes typically change only the four object box text colors, not the text editing window colors.

- On a per-patcher basis, you can edit the four object box syntax colors in the Format Palette for the patcher. With no objects in the patcher selected, show the Format Palette and click the P icon at the far left.



The Format Palette shows the default object, background, and text colors for the patcher. You'll want to locate the four **A** icons to the edit the syntax colors, Syntax Attribute Argument Color, Syntax Attribute Name Color, Syntax Object Argument Color, and Syntax Object Name Color.



It's helpful to edit these colors with object boxes showing the various syntax elements in the patcher visible.

Once changed, the customized colors will apply to all object boxes in a patcher (but not its subpatchers). To apply your customized colors more easily to new patchers, you can save the patcher as a template.

To apply the syntax colors to existing patchers, define a patcher-level style. Choose **Define New Style** from the Style menu in the Format Palette when editing the patcher fonts and colors.

Data

Arrays

When to Use Arrays 115
Creating an Array 115
Editing Arrays 115
Named Arrays 116
Converting to and from a List 117
Arrays and Dictionaries 119
JavaScript 119

Arrays are a data type in Max. Similar to lists, arrays store multiple items in order. Unlike lists, which are limited to storing numbers and symbols, arrays can store other Max data structures including strings, dictionaries, and other arrays. Similar to dictionaries and strings, arrays are stored in memory by name. Generally, arrays are a more powerful version of lists.

When to Use Arrays

Creating an Array

Create an array with the array object. Initialize the contents of the array by including the initial values as object arguments.

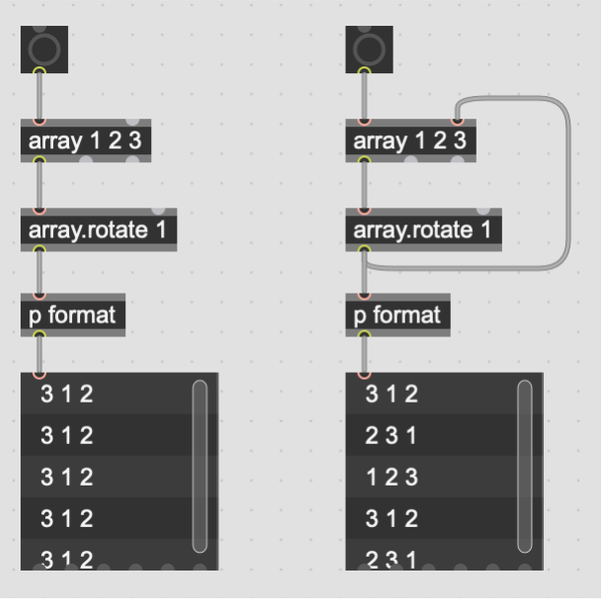


Editing Arrays

Edit an array in place by using the append, clear, delete, insert, prepend, and replace messages.

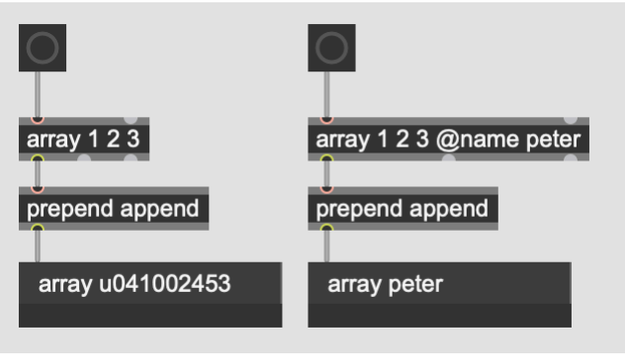


Objects that modify arrays as they pass through them, like `array.rotate`, don't change the original array, but create a new array with the result of their computation. In this example, the objects on the left always output the same result, but the objects on the right will update the original array with every rotation.

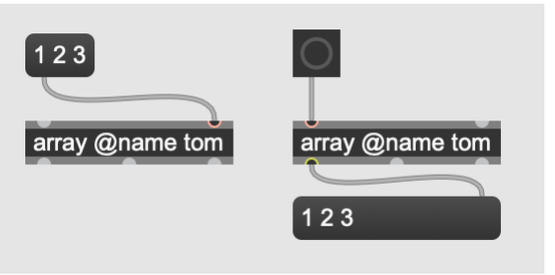


Named Arrays

Like dictionaries and strings, arrays always have a unique name. By default, the name will simply be a randomly generated unique identifier. You can also assign a name to an array using the @name attribute.

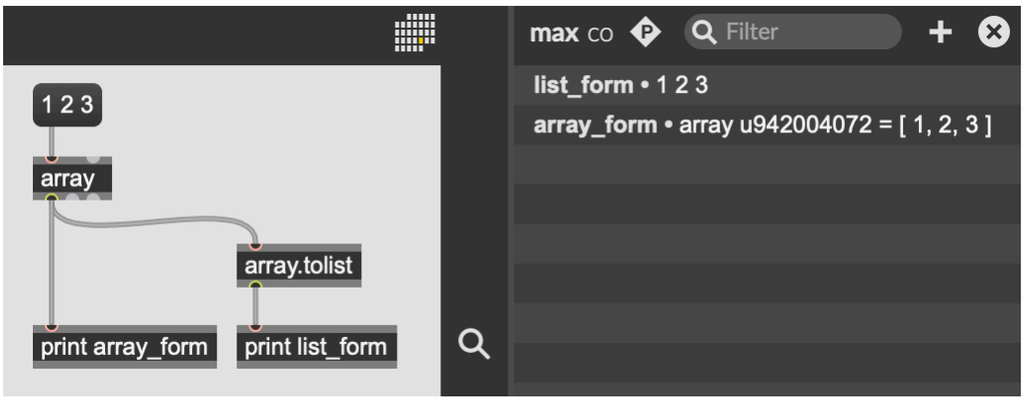


An array is identified by its unique name, so you can access the same array from any array object with the same name.

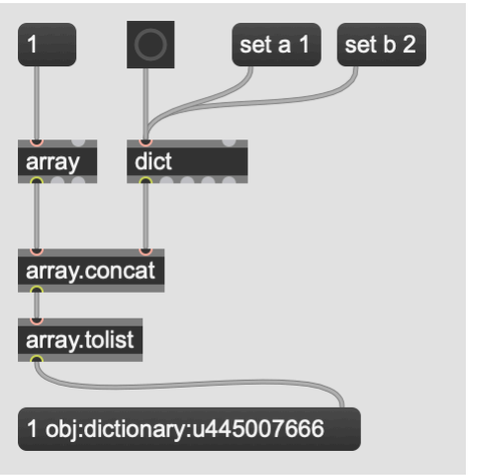


Converting to and from a List

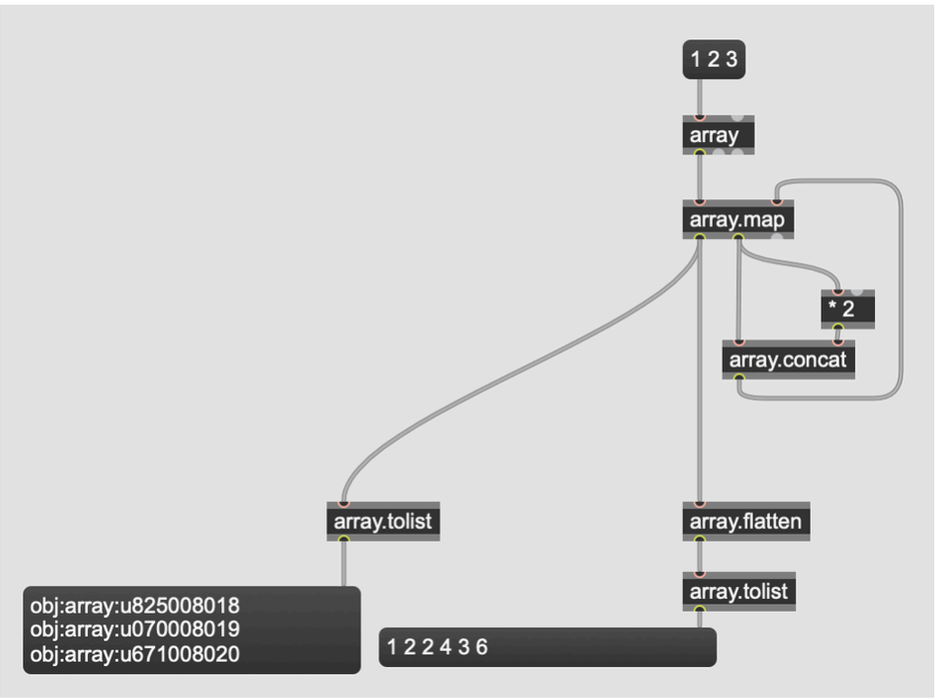
Any list sent to an array object will automatically be converted to an array. When it comes to working with arrays and lists, many objects will use arrays and lists interchangeably. However, in some circumstances, you might need to use a list and not an array. In these cases, you can use the array.tolist object to convert an array into a list.



If the array contains structured data, like dictionaries or other arrays, converting to a list will not unpack the contents of any structured data object. Instead, the list will simply contain a symbol representation of the object.

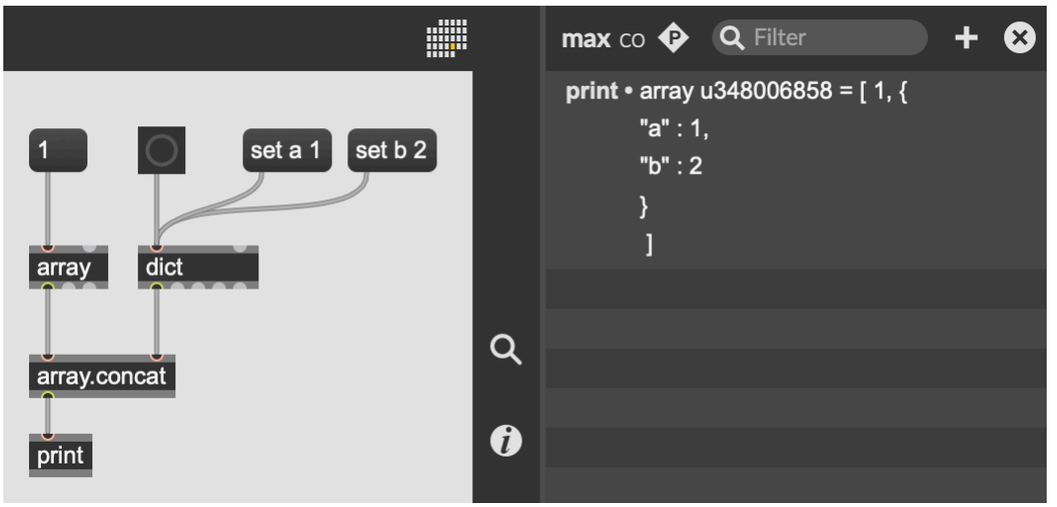


If your array contains only numbers, symbols/strings, and other arrays, you can use the object array.flatten to collect all sub-arrays and their contents into one long array. This can then pass through the array.tolist object to return a simple representation of your array's contents.



Arrays and Dictionaries

Dictionaries can contain arrays, and arrays can contain dictionaries. Printing an array containing a dictionary will output a JSON representation of the dictionary.



JavaScript

Use the `MaxArray` class to create a JavaScript reference to a Max Array. You can give it an initial value by passing a list or Array value to the constructor. Update the value of the array by calling `.set`.

```
var max_arr = new MaxArray(1, 2, "three", 4.0);
max_arr.set(10, 9, "eight", 7.0); // update the array contents
```

By setting the name property of the `MaxArray`, you can refer to a Array defined in the parent patcher.

```
var max_arr = new MaxArray();
max_arr.name = "frith"; // Now the MaxArray refers to an array named
"frith"
max_arr.set(2, 4, "six", 8); // Updates the array in the containing patcher
```

If you want to manipulate the Array value, call `.stringify` and `JSON.parse` to turn the Max Array into a JavaScript Array. From there, you can use regular JavaScript array manipulation functions. To convert back, use `JSON.stringify` and `.parse`.

```
var max_arr = new MaxArray();
max_arr.set(2, "3", "four", 6);
var js_arr = JSON.parse(max_arr.stringify()); // retrieve the value as a
JS string, convert to Array
post(JSON.stringify(js_arr) + "\n"); // prints "[2, '3', 'four', 6]"
js_arr[1] = 3;
max_arr.parse(JSON.stringify(js_arr));
post(max_arr.stringify() + "\n"); // prints "[2, 3, 'four', 6]"
```

To send a Max Array out of an outlet defined in JavaScript, send the string "array" followed by the name of the array.

```
function bang() {
  var arr = new MaxArray();
  arr.parse("[I, got, a, bang]");
  outlet(0, "array", arr.name);
}
```

Dictionaries

When to Use Dictionaries 121
Working with Dictionaries 122
Editing a Dictionary 122
Abbreviated Dictionary Syntax 125
JavaScript 126

A **Dictionary** is a container for data organized into key-value pairs. A key is always a symbol, but the value can be anything, including a number, symbol, list, or even another dictionary. You can create and manage dictionaries using the dict object, and the associated family of dictionary manipulation objects. Like arrays and strings, dictionaries have names and can be passed between objects with messages like `dictionary u12345678`.

For developers, a dictionary is most similar to a JSON object, and there are convenience functions in the API for converting a dictionary to a JSON string, as well as updating the contents of a dictionary with JSON.

When to Use Dictionaries

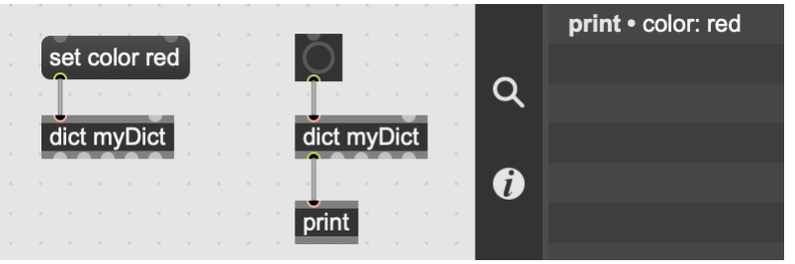
Dictionaries are useful for working with structured data, especially when that data has a hierarchical or labeled format. Imagine you were trying to simulate a large number of agents, each one modeling a simple creature with some kind of behavior. The state of each creature might look something like the following.

```
{
  "position": {
    "x": 2,
    "y": 5
  },
  "mood": "nostalgic"
}
```

With a dictionary, the values of your object not only have a place in a hierarchy, they also have labels.

Working with Dictionaries

Create a new dictionary using the dict object. The argument for the dict object is the unique name for the dictionary. If you do not give the dictionary a name, Max will generate a unique name automatically. Two dict objects with the same name will reference the same dictionary.



Dictionary basics

If you use a message object to display a string, you'll see the actual values that pass between Max objects when sending a dictionary in a message. In a message, a dictionary is represented by the symbol `dictionary`, followed by the unique name of the dictionary.

Editing a Dictionary

Edit the contents of a dictionary by double-clicking on a dict object, or by sending the dict object the `edit` message. This will open a text editor for modifying the contents of a dictionary.



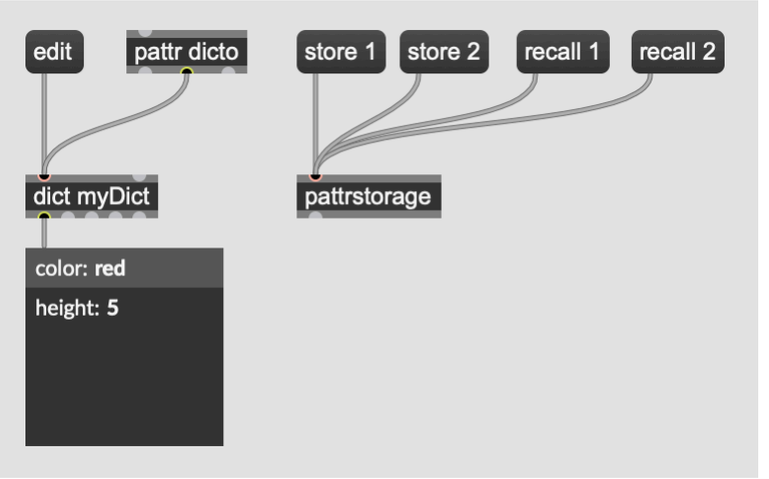
Dictionary editing

Dictionaries can be edited using JSON syntax (except that dictionaries don't support boolean (`true` / `false`) values). Keys must be strings, enclosed in quotation marks. Values can be numbers, strings, arrays (denoted with square brackets) or dictionaries (denoted by curly braces). Since the entire structure is itself a dictionary, curly braces must enclose the whole dictionary expression.

By sending the `export` message to the dict object, you can write the contents of a dictionary to a file, in either JSON or YAML format. Using the `import` message, you can then read such a file into a dict object.

pattr and pattrstorage

The pattr and pattrstorage objects will store the contents of a dictionary. Because a dictionary is not a simple numeric value, you can't interpolate between dictionary values using floating-point values for `recall`. However, in other respects, dictionaries are fully compatible with pattr and pattrstorage.



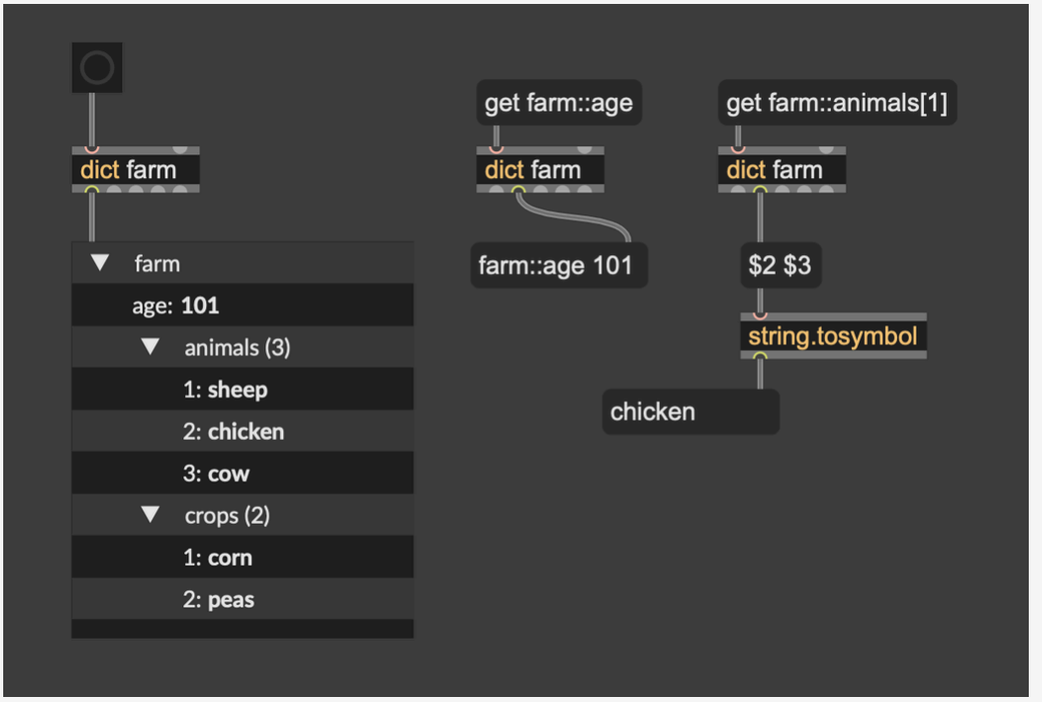
Dictionaries and pattr

Getting and setting values

Dictionaries can have nested dictionaries as values, and can contain array values an well. To simplify accessing nested elements within a dictionary, Max dictionary objects use a double-colon syntax.

```
{
  "farm": {
    "age": 101,
    "animals": ["sheep", "chicken", "cow"],
    "crops": ["corn", "peas"]
  }
}
```

Given a dictionary with the above contents, you can use the `get` or `set` messages to access values in the dictionary. Use a double-colon `::` to access a nested dictionary by its key, and use brackets `[]` to access elements in an array.



Convert a message to a signal without any smoothing

Abbreviated Dictionary Syntax

Dictionaries can be serialized to JSON, or initialized from a JSON string. However, dictionaries can also be parsed from a special abbreviated dictionary syntax, consisting of keys and values separated by a single colon. The following JavaScript code demonstrates initializing the same dictionary in two different ways.

```
let serial1 = `cow : 1 sheep : 2`;
let serial2 = `{ "cow": 1, "sheep": 2 }`;

let d1 = new Dict();
d1.parse(serial1);

let d2 = new Dict();
d2.parse(serial2);

// Dictionaries are the same
```



Initializing a dictionary with abbreviated dictionary syntax

JavaScript

Use the `Dict` class to create a JavaScript reference to a Max Dictionary. The name of the dictionary will be the first argument to the constructor, or an automatically generated unique name if none is provided. If the `js` object is in a patcher that contains a dictionary with the same name, then the `js` `Dict` object and the Max dict object refer to the same dictionary. Modifying the contents of one will change the contents of the other.

```
var max_dict = new Dict("mydict");
```

To manipulate the contents of the dictionary, use the `get` and `set` methods.

```
max_dict.set("color", "red");
console.log(max_dict.get("color")); // prints "red"
```

The `parse` method can initialize the contents of a dictionary using a JavaScript JSON serialization. This is a useful way to convert a JavaScript object to a Max dictionary.

```
let obj = { x: 1, y: 2 };
let serial = JSON.stringify(obj);
max_dict.parse(serial); // dictionary now has the keys x and y with values
1 and 2
```

To receive a dictionary in a JavaScript function, define a function named `dictionary`. The first argument to this function will be the name of the dictionary.

```
function dictionary(dict_name) {
  var myDict = new Dict(dict_name); // now "myDict" links to the passed-in
string.
}
```

To send a Max dictionary out of an outlet defined in JavaScript, send the symbol "dictionary" followed by the name of the dictionary.

```
function bang() {
  var myDict = new Dict();
  myDict.set("wood", "balsa");
  outlet(0, "dictionary", myDict.name);
}
```

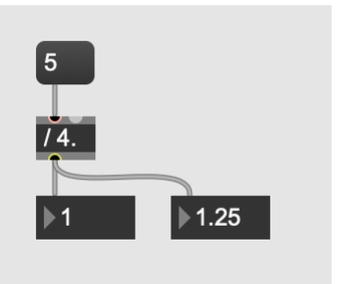
Integers and Floats

Integers and Floats in Max 129
Technical Details 131
Gen + RNBO 131

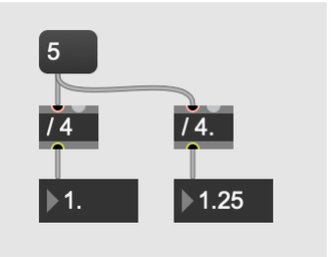
Max is fairly strict about how it deals with integer and floating point numbers. At a low level, integers and floats are two of the fundamental data types that objects can pass between each other, and many objects will handle the two data types differently.

Integers and Floats in Max

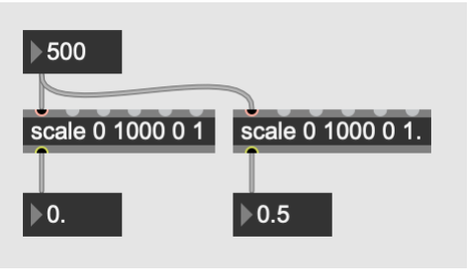
Integers are whole numbers, while floats can represent values with a decimal component. In Max, many objects operate in either a "floating point mode" or an "integer mode". This is a common source of bugs, since integer-mode objects will convert from float to int before processing, discarding any values after the decimal point.



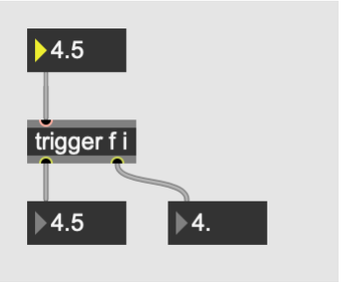
In this classic example, the result of the division `5 / 4` is computed as `1.25`, but displays as `1` in an integer numbox. This is the expected behavior, but sometimes it can be subtle when an object is in integer-mode.



The object containing a `/`, defining a division operation, can operate either in "integer mode" or "floating-point mode" depending on whether the argument has a decimal or not. As you can see, the integer-mode object will truncate any decimal component.



The scale object behaves similarly—so long as all of the arguments are integers, the object will be in "integer mode", and the output will be truncated to an integer value. The scale object on the left demonstrates this behavior. However, if any of the arguments to scale contains a decimal, then the object will be in "floating point mode", and the output will be a float, even if the input is an integer.



Some objects, like pack, pak, and trigger, can be configured to cast their inputs to floats or integers. A trigger object with the arguments `f` and `i`, as pictured, will cast its input to a float for the leftmost outlet, and to an integer for the rightmost. As you can see, the floating point box on the right displays the truncated value.

Lastly, it's worth mentioning the typeroute object, which can route messages by their type, separating out integers from floats.

A handful of other objects, in particular objects to do with simple math operations, will exhibit special behavior for integers and floats. When in doubt, check the help files and object reference documentation for more information.

Technical Details

In Max, integers are whole numbers. All integers are 64-bit, so the smallest integer that can pass between objects is -9,223,372,036,854,775,808, and the largest is 9,223,372,036,854,775,807. Floating point numbers in Max are also 64-bit (double precision). Messages can contain positive or negative numbers with a magnitude as large as 10^{308} 10308, or as small as 10^{-308} 10−308. Unlike integers, floating point numbers are not evenly spaced. There are as many floating point numbers between 0 and 1 as there are between 1 and 10^{308} 10308. This may or may not be spiritually significant.

Gen + RNBO

Unlike Max, Gen and RNBO do not use integers for any internal computation. If you really want Max-style integer math, for example truncating the result of a division operation, then you're best off using the trunc object for Gen, and the trunc object for RNBO.

Strings

When to Use Strings 132
Working with Strings 132
JavaScript 135
Strings vs Symbols 137

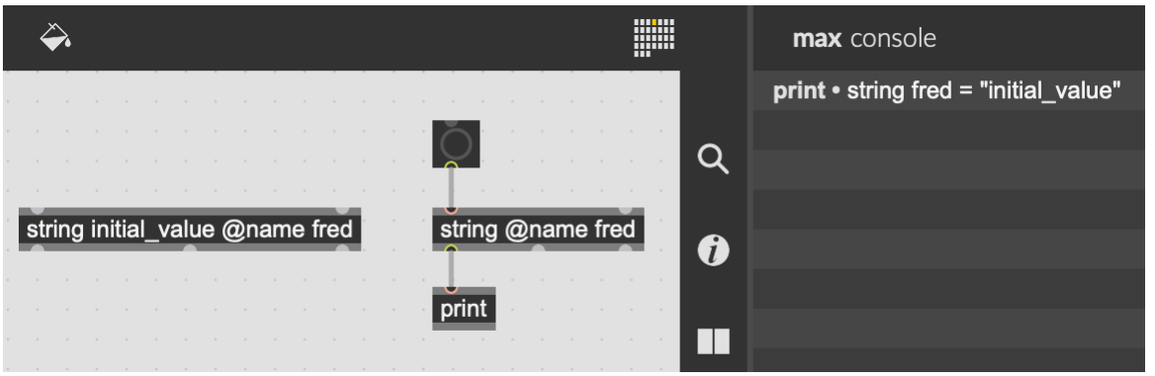
A **String** is a container for text (specifically UTF-8 text) that is independent of Max's Symbol Table. You can create and manage Strings using the string object. Like Arrays and Dictionaries, Strings have names and can be passed between objects with `string u12345678` messages. For developers, a String wraps Max's internal `t_string` object.

When to Use Strings

In addition to the string object, Max has several objects for editing, combining, filtering, and searching through strings. The string.at object searches a string for the position of a substring, and string.slice creates a new string from a character range within an existing string. If your patchers require this sort of text manipulation, it's easier and more efficient to use these purpose-built **String** objects than it is to use **Symbols**.

Working with Strings

Create a new **String** using the string object. The first argument is the initial value for the string, and you can (optionally) provide a name for the string as well using the `@name` attribute. Strings with the same name will share the same value, so you can create a string in one place and then refer to it somewhere else. When you print a string using the print object, Max will format the output to show the name and contents of the string.



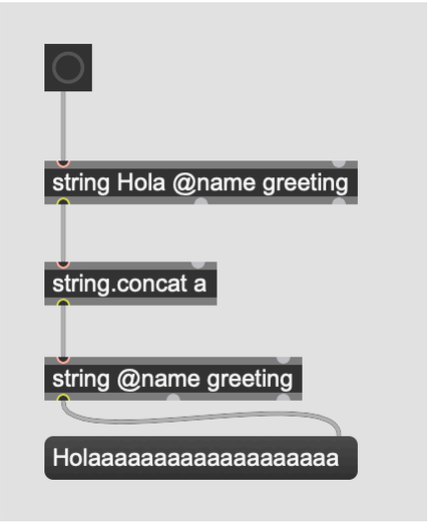
Strings basics

If you use a message object to display a string, you'll see the actual values that pass between Max objects when sending a string in a message. In a message, a string is represented by the symbol `string` followed by the unique identifier for that string. If you don't give your string an explicit name using the `@name` attribute, Max will assign one automatically.



> The message object has a `@convertobjs` attribute which will automatically convert received String objects into symbols for display.

If a string object changes the input string somehow, it will output a new string rather than modifying the original. This means that if you want to modify a string recursively, adding your changes back to the original string, you can use the name of the string to replace the original string.



pattr and pattrstorage

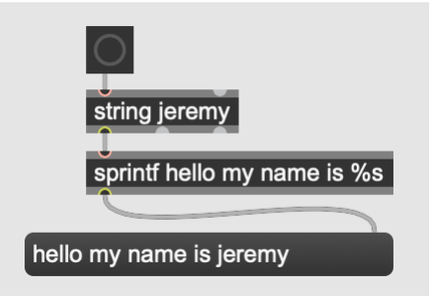
The pattr and pattrstorage objects will store the value of a string. They do not store the string itself, so if you modify the string after storing its value in a pattr object, the updated value will not appear in pattr until you send the string to pattr again.



In this example, changing the value of the string by sending the `new_value` message to the second string object will not update the value stored in pattr.

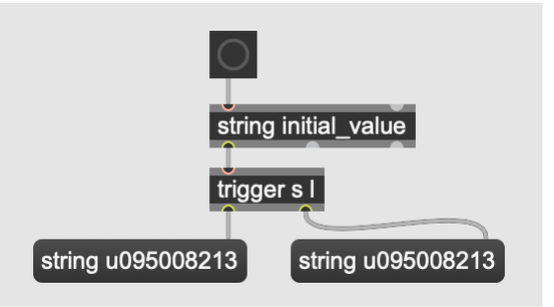
Backwards compatibility

If a receiving object does not understand the new `string` type, then Max will automatically convert that string into a symbol to maintain compatibility.



A handful of control objects will always pass strings unmodified, so that they can still be used to route strings between objects. Those objects include:

- append
- prepend
- route
- routepass
- trigger
- match
- router
- universal
- typeroute~
- gate/switch



The trigger object passes the string message through without decomposing it into a symbol. This works even when we use the symbol formatter for trigger.

JavaScript

Use the `MaxString` class to create a JavaScript reference to a Max String. You can give it an initial value by passing a string value to the constructor. Update the value of the string by calling `.parse` or `.set`.

```
var max_str = new MaxString("initial_value");
max_str.parse("new_value"); // update the string contents
```

By setting the name property of the `MaxString`, you can refer to a String defined in the parent patcher.

```
var max_str = new MaxString();
max_str.name = "fred"; // Now the MaxString refers to a string named "fred"
max_str.set("new_value"); // Updates the string in the containing patcher
```

If you want to manipulate the String value, call `.stringify` or `.get` to turn the Max String into a JavaScript string. From there, you can use regular JavaScript string manipulation functions.

```
var max_str = new MaxString();
max_str.set("the original string value");

var js_str = max_str.get(); // retrieve the value as a JS string
var updated_str = js_str.replace("original", "new");
max_str.parse(updated_str);
post(max_str.stringify()); // prints "the new string value"
```

To send a Max String out of an outlet defined in JavaScript, send the symbol "string" followed by the name of the string.

```
function bang() {
  var str = new MaxString();
  str.parse("I got a bang");
  outlet(0, "string", str.name);
}
```

Strings vs Symbols

When working in Max, most of the time, objects pass around text in the form of symbols. When you include text like `bgcolor` or `set` in a list, you're using a symbol. Max doesn't pass the text of a symbol directly, but instead generates a unique identifier for each symbol, passing that identifier between objects instead. This makes certain operations on symbols very efficient, for example comparing the value of two symbols. However, it also means that every time you use a new symbol, it must be assigned to a unique identifier, and that identifier must be added to the **Symbol Table**. The identifiers added to the Symbol Table are never removed -- the table will grow forever until Max is quit, or runs out of memory.

Strings, on the other hand, do not interact with the Symbol Table. Instead, Max manages Strings in a similar way to **Buffers** or the contents of a dict object. The text contents of a String are located somewhere in memory, and Max gives that memory a name that can be used to locate the contents of the String. The object interface to a block of audio samples is buffer~, and the object interface to a Dictionary is the dict object. In a similar way, the string object is the interface to a text String. When a buffer~ or a dict is cleared, the underlying memory is released back to the operating system to be used for new storage. The same applies to Strings, which can be, depending on your requirements, a more efficient way to store text data.

One further difference is that Max symbols are restricted to 32767 characters. Strings have no such limitation, and support the storage of huge blocks of text data.

Debugging

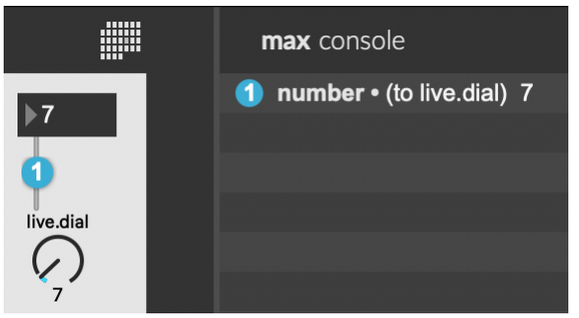
Debugging and Probing

Debugging 139
Enabling/disabling Debug Mode 140
Stepping Through 140
Illustration Mode 142
Probing 143

Debugging

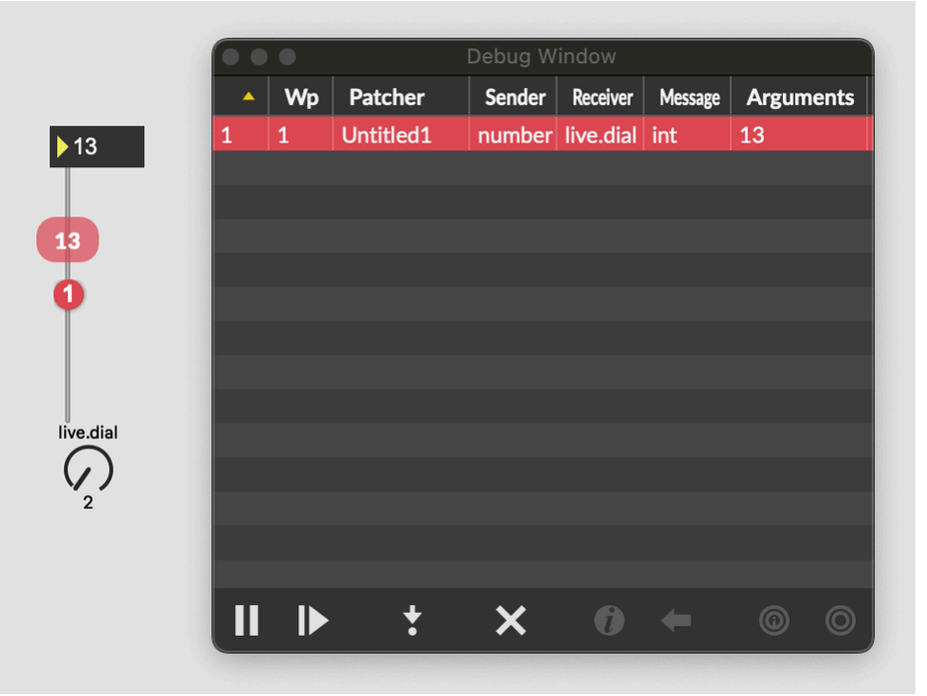
With **Debugging**, you can monitor any messages passing along a patchcord, or pause execution and walk through processing in the patcher step by step. Debugging starts with **Watchpoints**, which you can configure to *monitor*, *print*, or *pause*. To add a watchpoint, right-click on any patchcord and select *Add Watchpoint*, or select a patchcord and select *Add Watchpoint* from the *Debug* menu.

- **Print Watchpoint**: A Watchpoint that will print a message to the Max Console whenever a message passes through the patch cord. This message indicates the source and destination of the message, along with its contents.



A print watchpoint printing its message

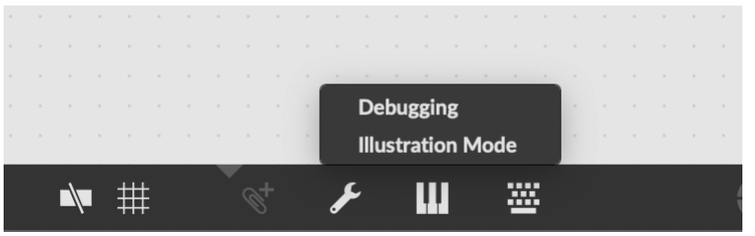
- **Monitor Watchpoint**: Display a popup in the Max patcher when a message passes through the patchcord.
- **Break Watchpoint**: Pause execution and open the Debug Window whenever a message passes through the patchcord.



A break watchpoint pausing execution. You can see that the value 13 has not yet reached the live.dial, which still displays the value 2.

Enabling/disabling Debug Mode

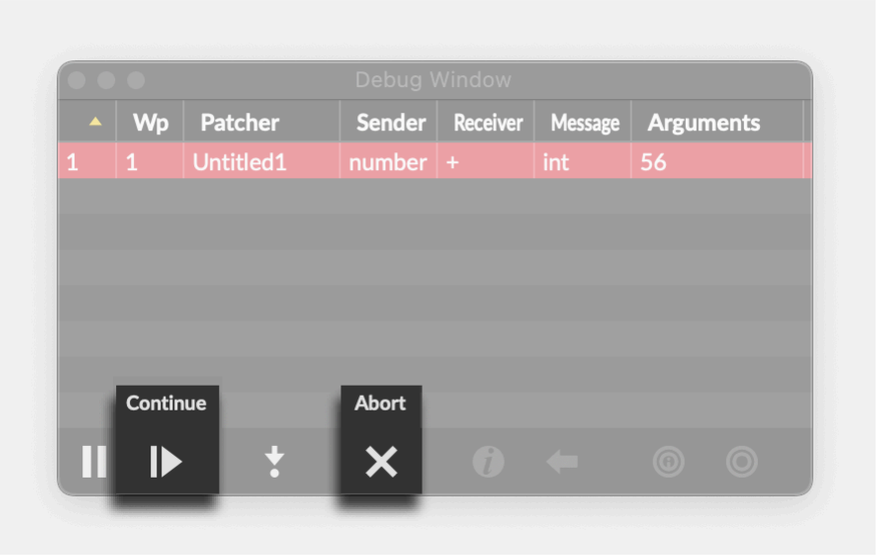
Choose *Debugging* from the *Debug* menu to enable debugging. You can also use the *Debug* icon in the bottom toolbar to toggle debugging, as well as enabling Illustration Mode.



Stepping Through

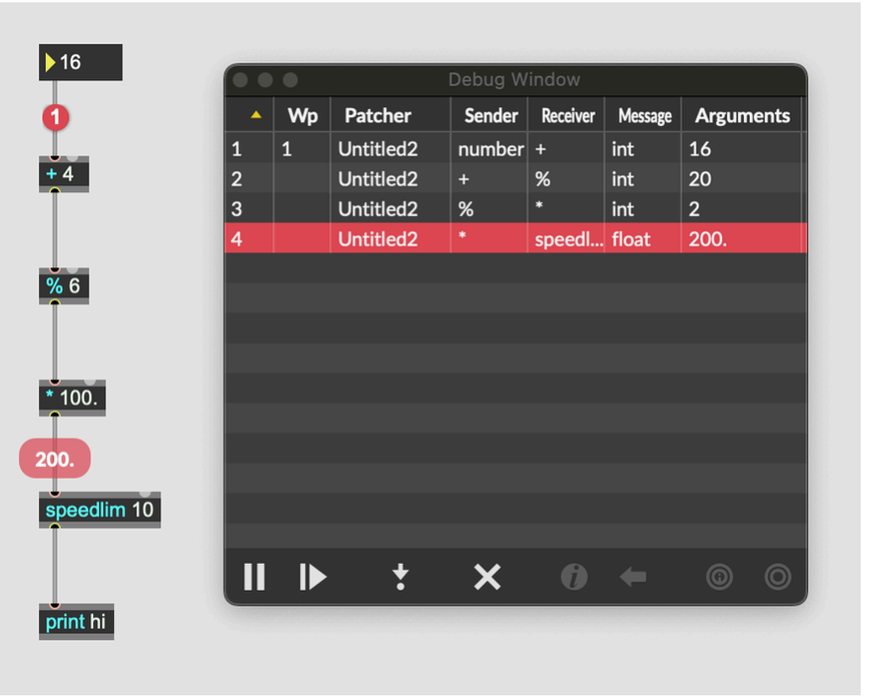
When a Break Watchpoint is triggered, execution pauses and Max will focus on the **Debug Window**. From here you can see the sender, receiver, and contents of the message that triggered the watchpoint.

After triggering a breakpoint, The *Continue* button will let you resume execution, and the *Abort* button will effectively remove the message from Max's scheduler and exit debugging. This can be really useful, especially if the message is about to do something that you don't want.



The Continue and Abort buttons

The *Step* button is a very powerful tool in the Debug Window, allowing you to walk through the flow of messages in a Max patcher one step at a time.



Pressing the step button moves through the patcher and adds layers to the call stack.

When stepping, whenever sending a message to an object triggers a new message, the new message will appear at the bottom of the execution stack. In this way, you can see the whole processing chain in response to a message.

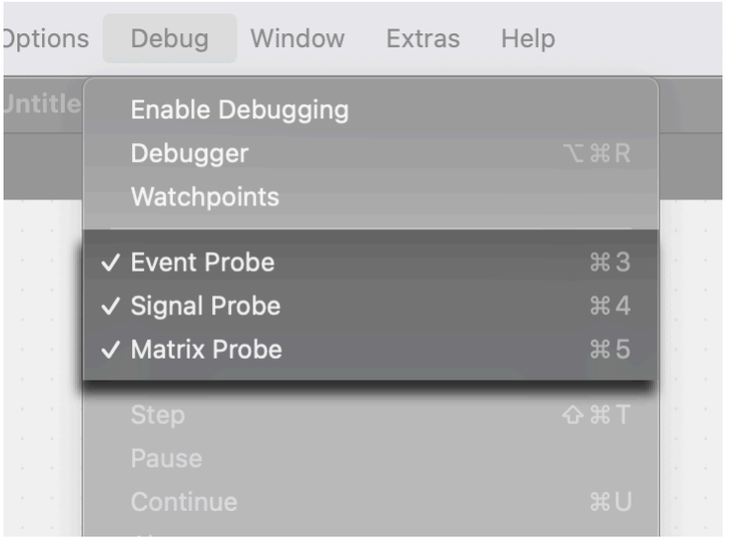
> If you are in the middle of debugging, you cannot operate your patcher. In addition, you cannot close the patcher window being debugged, and you cannot quit Max. To exit debugging and enable these functions again, choose *Abort* from the *Debug* menu, and you will be able to operate Max normally.

Illustration Mode

- Introduction
- Activating illustration mode
- Clearing pending messages

Probing

Probing lets you see the last message that passed through a patch cord by hovering over the patch cord you want to inspect. With probing you can see messages, matrices and textures, as well as audio vectors passing between objects. *Event Probing*, *Signal Probing*, and *Matrix Probing* must all be enabled from the *Debug* menu before you can use them.



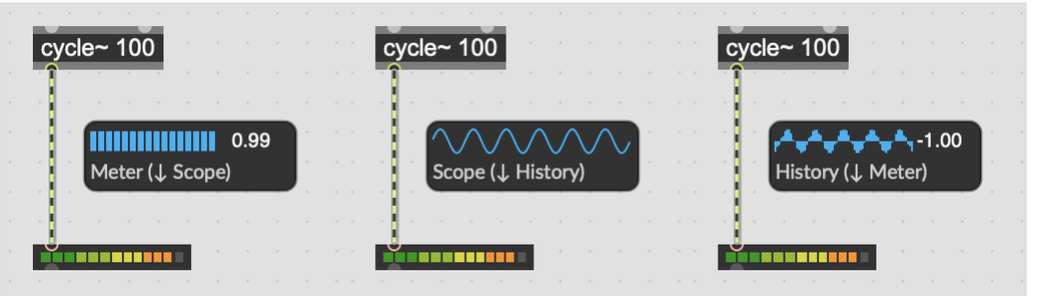
Probing in the Debug menu

Event Probing

With Event Probe enabled, hovering over any patchcord will display the last message that passed through, or else `no data` if no message has passed through.

Signal Probe

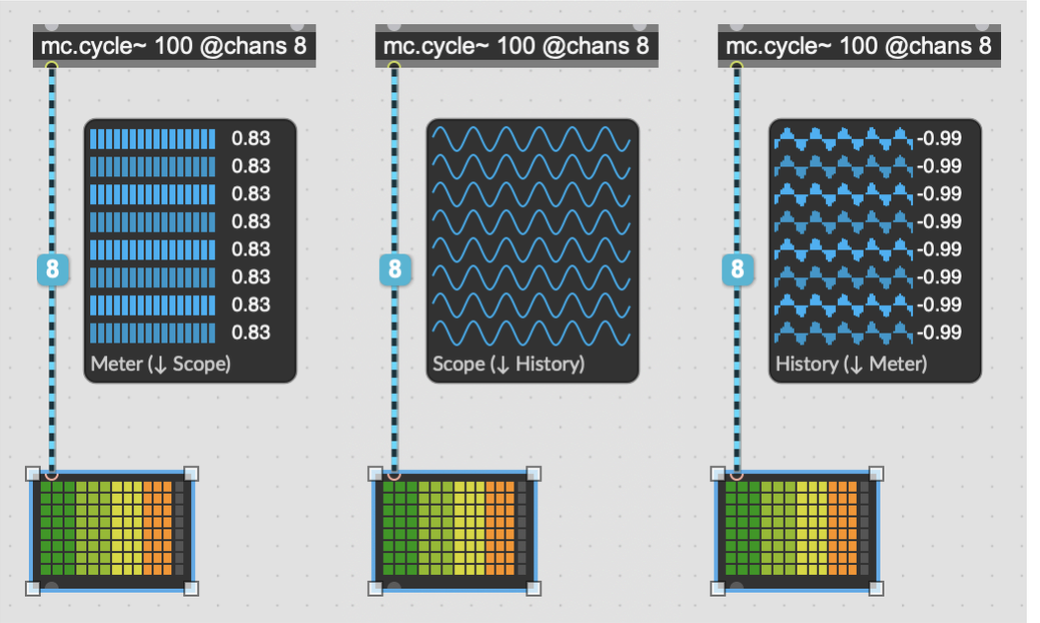
The Signal Probe lets you see the audio data passing between two objects. With Signal Probe enabled, hover over any audio patch cord to get a visualization of the data. While the signal probe popup is visible, you can press the up or down arrow keys to cycle between **Meter**, **Scope**, and **History** views.



The Signal Probe popup (all three views).

> Signal Processing must be enabled in order to use the Signal Probe.

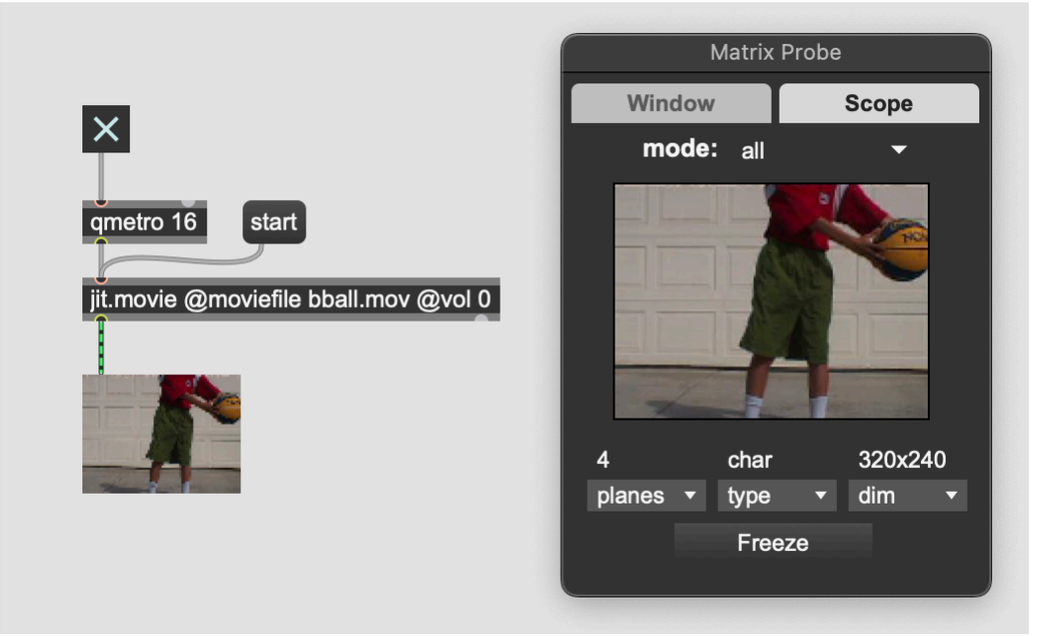
The Signal Probe also works with **mc.*** objects.



The Signal Probe with an mc object (all three views).

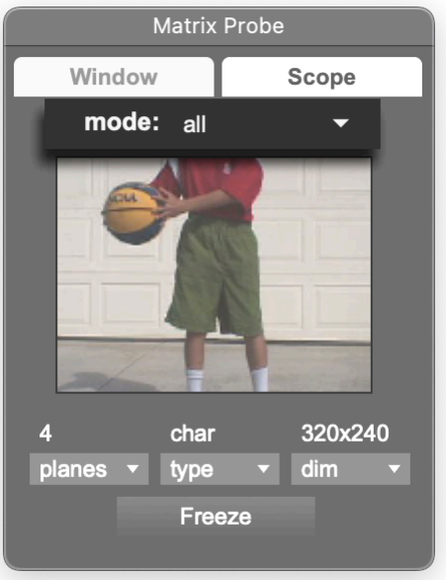
Matrix Probe

Unlike the Signal Probe and Event Probe, the Matrix Probe displays in a separate window. Enabling *Matrix Probe* from the *Debug* menu will display the Matrix Probe window.



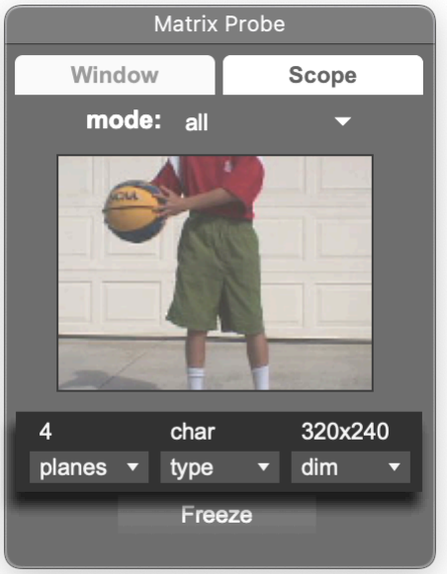
Viewing a matrix with the Matrix Probe.

From the *Window* tab, the *Mode* chooser will let you choose which plane of the matrix to inspect—alpha, red, green, blue, or a composite of all four.



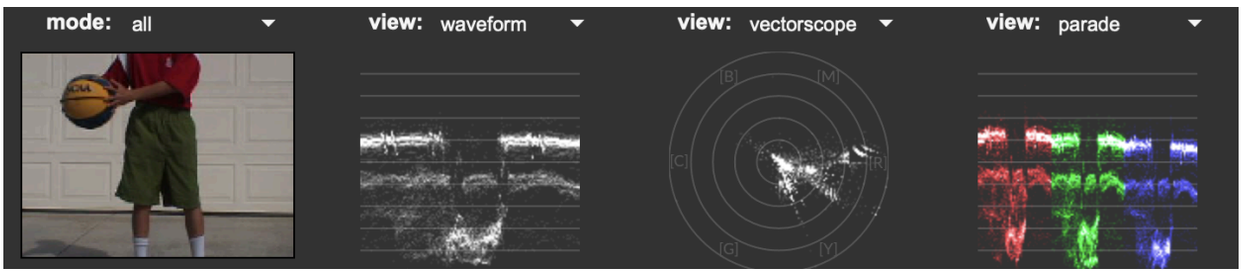
Choosing the plane to display

The dropdown menus at the bottom of the window will let you view additional information about the matrix, including the number of planes, the type of data contained in the matrix, and the dimensions of the matrix.



Get more information about the matrix

The *Scope* tab shows useful statistics about the matrix. For example, the *vectorscope* view shows the distribution of color intensities, which can help you visualize which colors are most common in the matrix.



*The *vectorscope* shows that red and orange are strong colors in the matrix.*

Error Messages

Common Error Messages 147

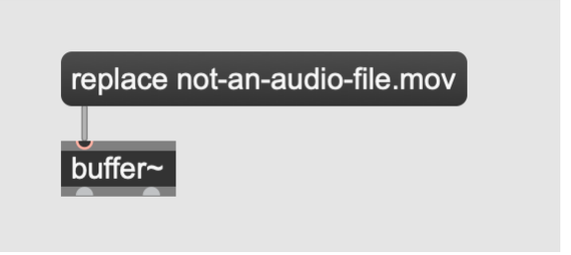
When Max encounters an error, it will print a message to the Max Console. This message is often specific to the object that generated it, and the message will usually contain whatever information you need to figure out what went wrong. However, there are some common errors that you may encounter often when working with Max

Common Error Messages

Files

`<objectname>: error opening file`

The object was able to locate the file with the name, but there was a problem reading or opening the file.



`<objectname> : <filename> : can't open`

The object was not able to open the file. This could be because the file contained data that the object was not able to understand, or because the file is missing, or because Max doesn't have permission to read the file. It may be that the file is not in Max's Search Path, so make sure that Max can locate the file.

`<filename> : bad magic number`

The Max file you tried to open is corrupted or is not a properly formatted Max document. Restore the file from a backup copy if available.

`<filename> : corrupt binary format file`

The Max file you tried to open is corrupted or is not a properly formatted Max document. Restore the file from a backup copy if available.

`<filename> : error creating file`

There was an error writing a file; the disk may be write-protected or full.

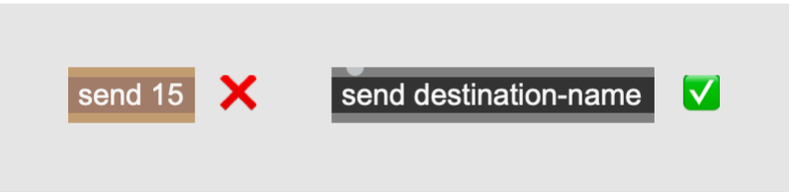
`<filename> : out of memory writing file`

There is insufficient memory to write the file you're trying to save. If possible, close other files and windows that don't relate to the file you're saving.

Objects and Patching

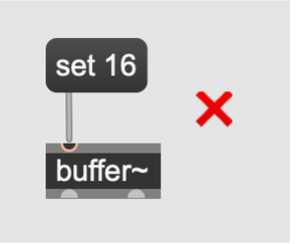
`<objectname> : bad argument creating object`

The object was given an argument that it doesn't understand. This can happen when an object expects a symbol as an argument, but gets a number instead.



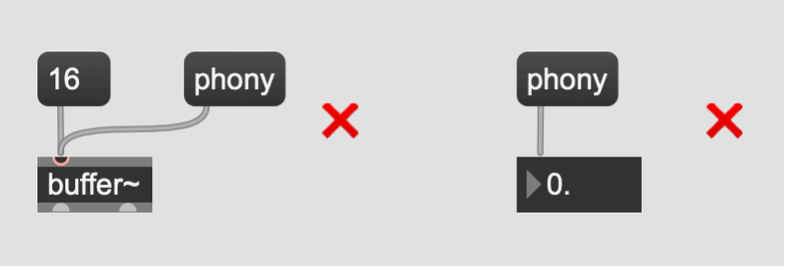
`<objectname> : bad arguments for message <message>`

The object received a message that it understood, but with a bad argument. For example, trying to set the name of a buffer~ object to be a number will display this error message, because the name of a buffer~ cannot be a number.



`<objectname> : doesn't understand <message-selector>`

The object received a message, but it doesn't understand that message. This often occurs when sending a symbol to a number box, when meaning to send an interger or float.

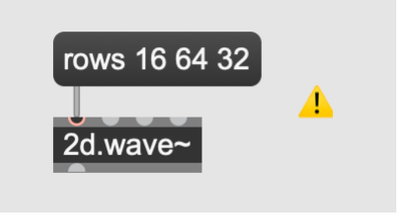


`<objectname> : message too long <message>`

A message was sent that contained more elements than the object can handle.

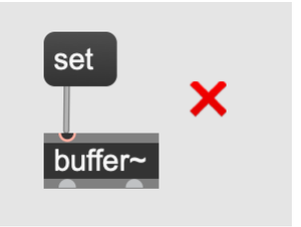
`<objectname> : extra arguments for message <message-selector>`

The object received a message that it understands, with valid arguments, but it received more arguments than it expected. This will display as a warning, not an error, since the extra arguments are simply ignored.



`<objectname> : missing arguments for message <message>`

The object received a message that it understands, but missing some arguments to that message. When a buffer~ receives the `set` message, without any arguments, it will display this message, because buffer~ expects an argument to the message `set`.



`<objectname> : No such object`

Max cannot find an object with the given name. Whether you're trying to load an Abstraction or an External, make sure that the relevant file is in Max's Search Path. If you're opening a patch and you see this error, the author may have forgotten to include an abstraction or external that the patch depends on. It may also be that you need to install a Package from the Package Manager that includes the missing dependencies.

`can't connect <objectname> to <objectname>`

Advisory message produced when you try to connect an outlet to an inlet that doesn't understand the message sent by the outlet. Typically you'll only see this message when opening a saved patcher, if for some reason an object doesn't load as expected.

`No help available for <objectname>`

A help file could not be located for the given object. Make sure that the help file (with the format `<objectname>.maxhelp`) is in the Search Path.

`patcher connect: inlet <number> out of range`

Occurs when editing the name or arguments of an object that has already been created in a patcher, and patch cords that used to be connected to the object can no longer be connected. Very often this occurs when changing the number of inlets or outlets of an object like gate or route, while that object is created.

Audio

`ad_mme: <message>`

(Windows only) Please check that you have the latest driver update for your audio device. Please exit all other audio applications, reboot if necessary, and try again. Also, please check your settings in the Audio Status window to insure appropriate choices are selected for Input Device, Output Device, Sampling Rate, IO Vector Size, and Signal Vector Size. If the problem persists, contact Cycling '74 support.

`ad_directsound: <message>`

(Windows only) Please check that you have the latest driver update for your audio device. Please exit all other audio applications, reboot if necessary, and try again. Also, please check your settings in the Audio Status window to insure appropriate choices are selected for Input Device, Output Device, Sampling Rate, IO Vector Size, and Signal Vector Size. If the problem persists, contact Cycling '74 support.

`ASIOCreateBuffers error`

(Windows only) A problem was encountered initializing the ASIO device. Please check that you have the latest driver update from your audio device manufacturer. Please also try different settings for the device buffer sizes and latency in the control panel for your audio device provided by your device manufacturer. Check that another audio application is not using the audio device. Also check that the audio device is not the default audio device for Windows System Sounds.

`midi_mme: <message>`

(Windows only) Max was unable to open the MIDI input or output device. Please exit from all other MIDI applications and try again.

`MSP/ASIO: <message>`

(Windows only) A problem was encountered initializing the ASIO device. Please check that you have the latest driver update from your audio device manufacturer. Please also try different settings for the device buffer sizes and latency in the control panel for your audio device provided by your device manufacturer. Check that another audio application is not using the audio device. Also check that the audio device is not the default audio device for Windows System Sounds

Illustration Mode

Enabling Illustration Mode 152
Illustration Mode and Debugging 153
Debug Event Queue 153

Observe your patcher running in slow motion to understand and/or debug its operation.

Enabling Illustration Mode

Choose *Illustration Mode* from the *Debug* menu. You can also click the Debug icon in the bottom patcher toolbar to show the debug menu.

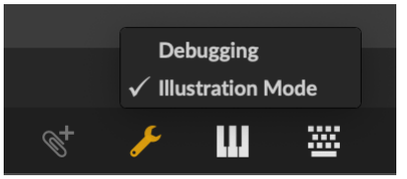


Illustration Mode is generally a global setting for all patcher windows.

A Simple Example

Whlie it might seem as if everything in Max happens all at once, there's a predictable order of operations that becomes visible once you slow things down. The animation below demonstrates the right-to-left sorting of patch cords from a single outlet.

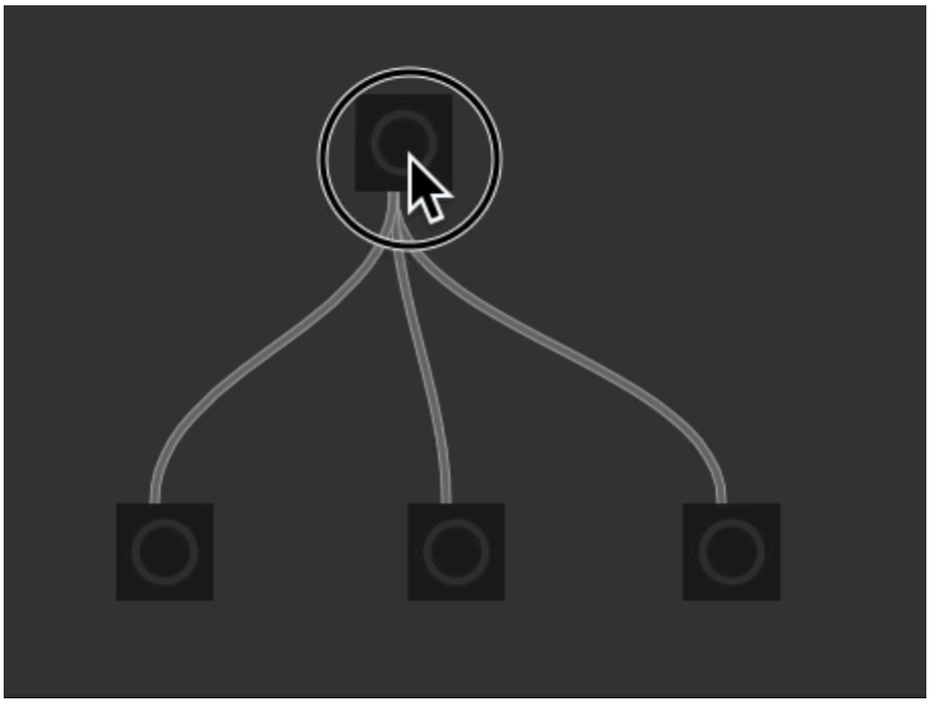


Illustration Mode in Action

Illustration Mode and Debugging

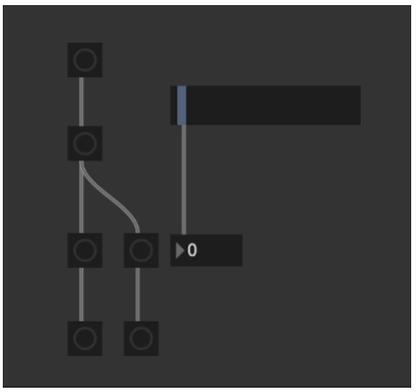
Most Max debugging features work in Illustration Mode. Your patcher will run in slow motion and messages wil "travel" down patch cords, unlike traditional Max debugging where the patcher runs at regular speed until you stop execution at a Break watchpoint. If a Break watchpoint is encountered in Illustration Mode, excecution will pause. You can then choose Continue from the Debug menu to resume Illustration Mode execution, Abort to cancel execution entirely, or Step to advance to the next outlet and pause again. Choosing Pause from the Debug menu will also pause execution.

When paused, unchecking Illustration Mode in the Debug menu will continue normal execution.

Debug Event Queue

When using either Illustration Mode or debugging, Max is not completely frozen and other events could occur. Examples include changing UI objects or incoming MIDI messages. Events that occur while debugging or in Illustration Mode are placed into event queues before they are sent out an outlet and illustrated. When the current computation sequence completes, the next event will be removed from the event queue and processed.

In the example below, events are queued when you move a slider, then output one by one once the processing chain of objects connected to the button completes. The pending event queue for each outlet is represented by an orange bubble showing the count of events remaining to process.



Debug Event Queue

To empty the event queue for an outlet, cancelling all pending events, click the close button on the orange bubble. To cancel pending events at all outlets, click the Debug icon in the bottom toolbar and choose Cancel Pending Events.

You can set the maximum size of the Illustration Mode Event Queue in the Preferences window. (The size defaults to 0 which is unlimited.)

As an example, if you set the maximum queue size to 15, up to 15 events will be stored in the queue at each outlet. Once the queue is full, new events will begin overwriting the oldest events.

Max Console

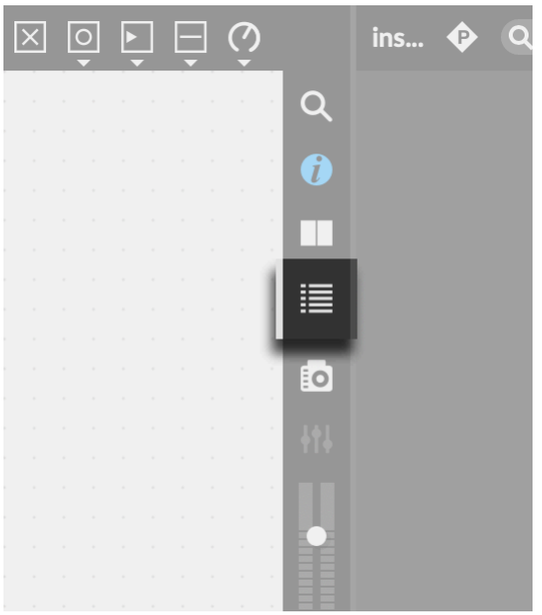
Showing the Max Console 155
Using the Max Console 157
Adjusting the Font and Size 161
Color Coding and Errors 161
The Max Console Contextual Menu 162
Console-related Objects 163

The **Max Console** displays printed text messages including status information, error messages, and warnings. If your patcher isn't working the way you expect, one of the first things you should do is check the Max Console to see if any error messages appear here. You can write to the Max console directly using the print object, or in **Debug Mode** by using a *Print Watchpoint* on a patchcord (see Debugging and Probing).

Showing the Max Console

Max console sidebar

Click on the Max Console button on the right-hand toolbar to open the Max Console in the sidebar. To close the sidebar, click on the Close button, or click on the Max Console button.



The Max Console sidebar provides all of the functions of the Max Console window. Additionally, you can limit the messages viewed in the Max Console sidebar by clicking on the *Messages for This Patcher Only* icon at the top of the sidebar.



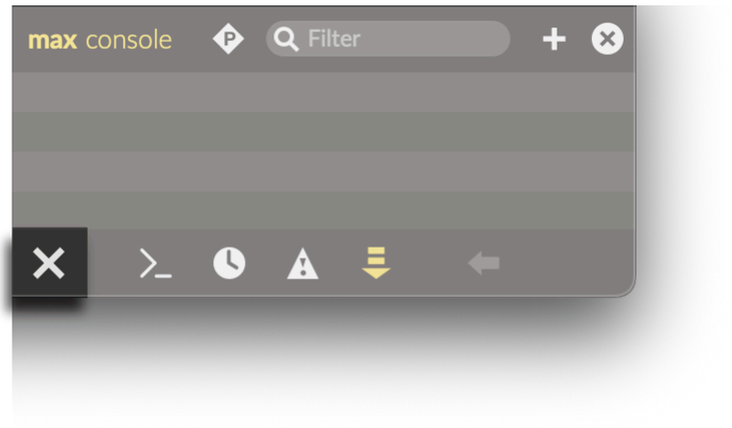
Dedicated window

Open the *Window* menu and choose `Max Console`. This will display the console in its own window. The Max application keeps track of whether or not the Max Console is visible when you quit the program, and will hide or show the window at its last size and position when you relaunch Max based on its state when you last used Max.

Using the Max Console

Clearing the console

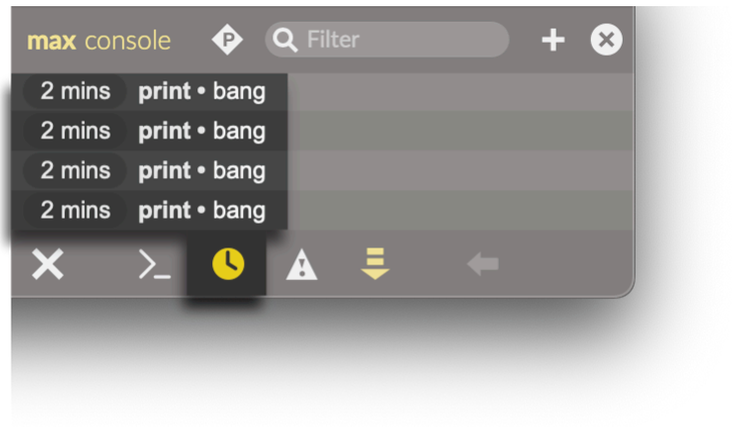
Clear the Max Console by clicking the *Clear All* button in the bottom toolbar.



You can also select the lines that you'd like to delete, and choose *Delete* or *Cut* from the *Edit* menu.

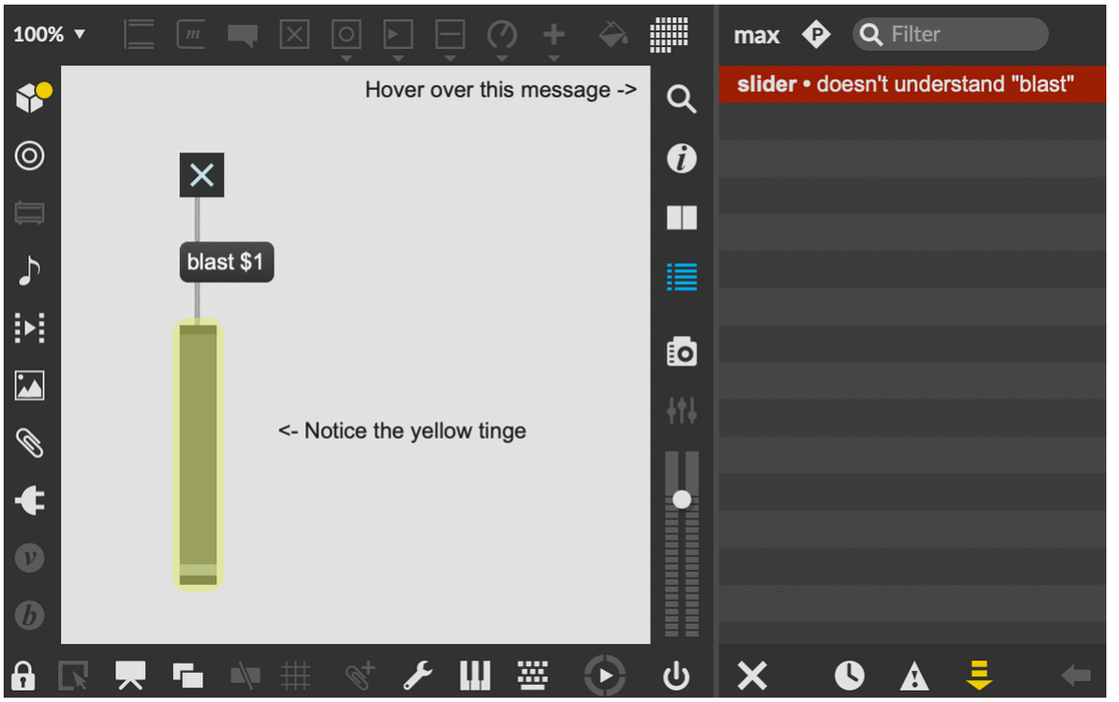
Viewing message time

The *Show Message Time* button will display the time since each message was generated.



Finding the object that generated a message

If a row of the Max Console displays the name of an object, hover over the row and a green tinge will appear over the object. Double-click on the row to cause the object that generated the error to be scrolled into view and brought to the front. You can also click the *Show Object* button in the bottom toolbar.



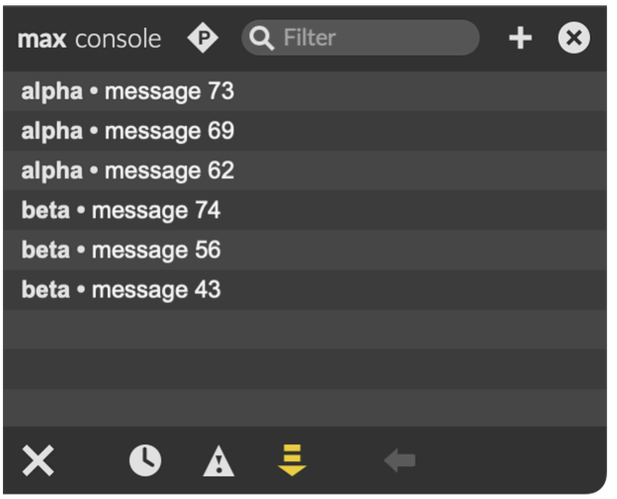
> You can also use the Max Console Contextual Menu to find the object that generated a message.

Copying text

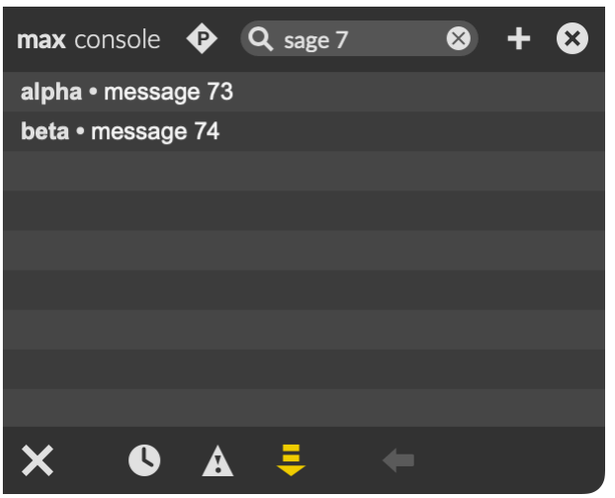
To copy text from the Max Console, select the lines you want to copy, then choose *Copy* from the *Edit* menu.

Filtering the contents of the Max Console

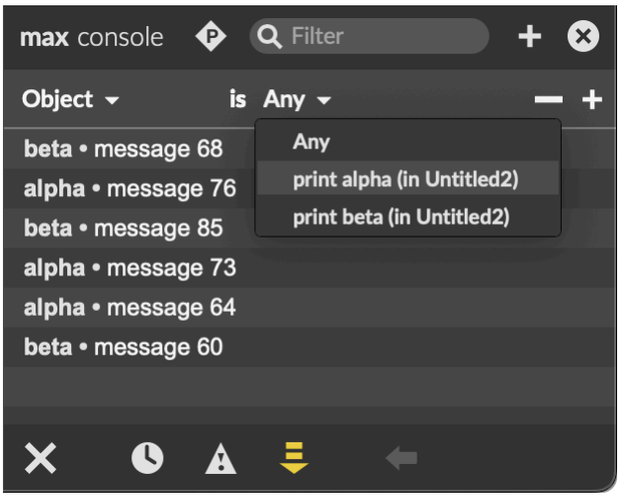
Suppose the Max Console contains the following:



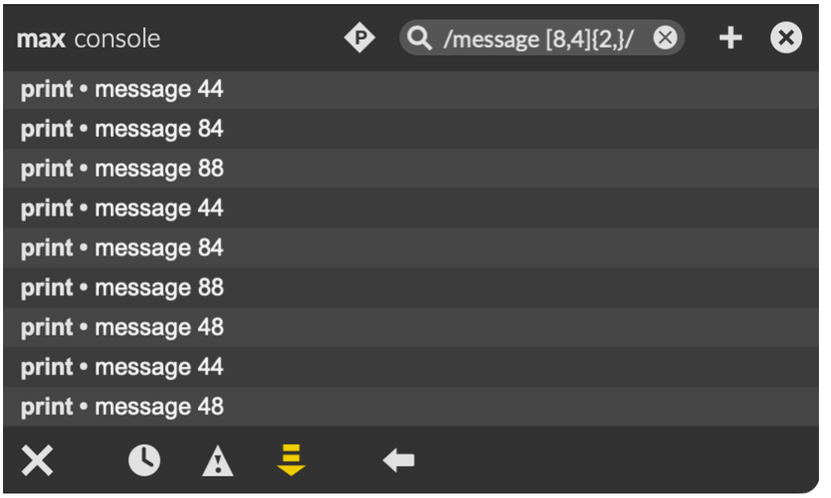
Enter any text into the filter field to display only messages that contains that text.



Click the "+" button to the right of the filter field to add search criteria. This will let you filter for messages from a particular object, patcher, or class of objects. You can also command-click or right-click on a console line to display the contextual menu, then select "Filter By This Object" to show only messages generated by that single object.



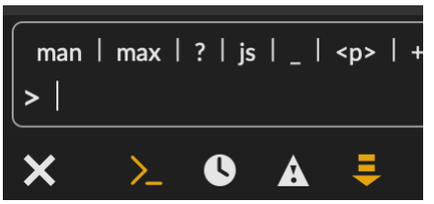
You can use regular expressions in the search field as well by beginning and ending your search query with a `/` character.



Finally, the *Show Only Errors* button button in the bottom toolbar will hide all messages except for errors.

The REPL

The REPL (Read-Evaluate-Print-Loop) at the bottom of the Max Console lets you interact with the current Max patcher using text.



Using the REPL, you can:

- Send messages to Max objects in your patcher
- Evaluate JavaScript expressions
- Control the Max environment with text-based commands
- Access the documentation

Check out the REPL documentation for more information.

Adjusting the Font and Size

To adjust the font and size, bring the dedicated Max console window to the front, then choose *Show Fonts* from the *Object*. Your chosen font and size will be saved for the next time you launch Max.

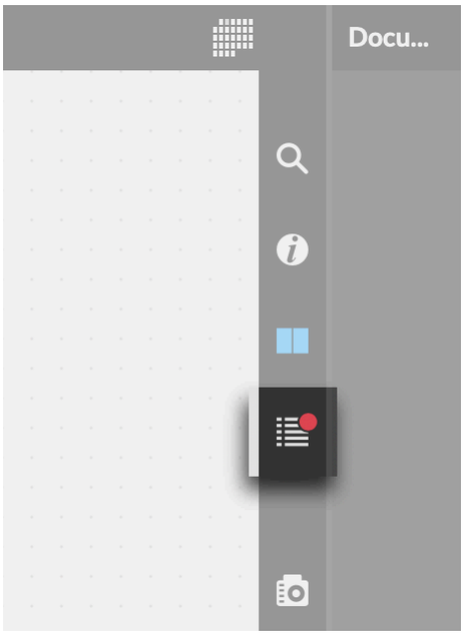
Color Coding and Errors

Rows of the Max Console are color coded.

- **gray/white**: status information or messagess sent to a print object
- **yellow**: warnings
- **red**: error messages
- **blue**: internal errors (bugs in Max itself)

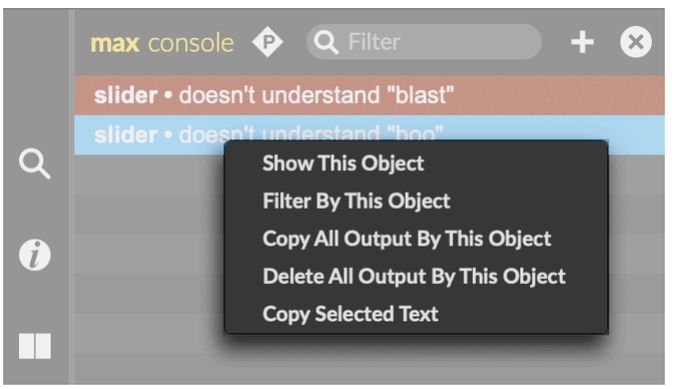
You can find a listing of common error messages here.

On the patcher toolbar, the Max Console button will display a badge if there are any errors or printed messages to be displayed. The badge can be cleared by setting focus to the Max Console window.



The Max Console Contextual Menu

Command-click or right-click on a Max Console message to display the contextual menu. There are several options for using the messages as well as managing the Max Console contents.



- *Show This Object* will bring the object into view and apply a green tinge over the object.

- *Select This Object* will select the object that generated the message.
- *Filter By This Object* will filter the contents of the Max Console to show only the current object's messages. To remove the filter, click the Remove Criteria button at the top of the window.
- *Copy All Output* By This Object will copy the text of all of an object's messages to the clipboard. This can be useful for capturing data or documenting error conditions.
- *Delete All Output* By This Object will remove all messages generated by the object that generated the selected message. Once deleted, the text cannot be retrieved.

Console-related Objects

The print and error Objects

Use the print object to log a message to the Max console. You can also use the error object in the same way, except any messages sent to the error object will have special "error" formatting.

The console Object

The console object lets you receive messages that would print to the Max console directly in your patcher. The console object can receive warnings, errors, or just general messages. You can also use the `textfilter` message to set a filter for the console object specifically, the same as the text filter in the console itself. See the console object helpfile or the console object reference for more information.

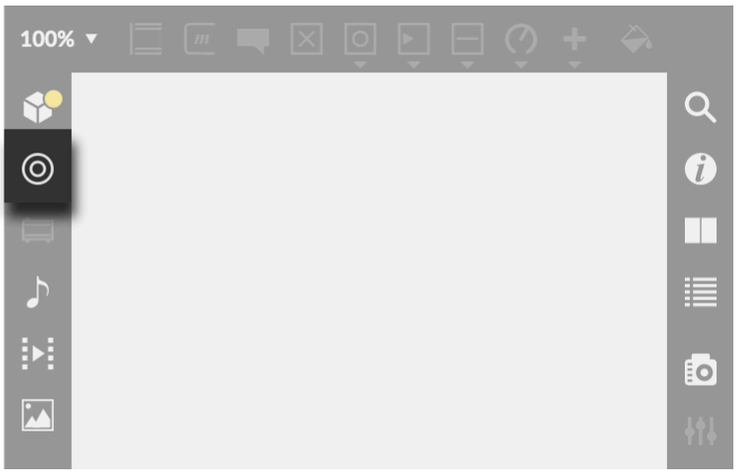
Files

File Browser

Browsing 165
Adding Files to the Search Path 166
Advanced Search 167
Bookmarks 168
Collections 169

The **File Browser** is a graphical interface to Max's Search Path, letting you view, search, and organize all the files that Max can access. It's helpful not only for finding your own files, but also for finding all of the content that comes with Max. You can also create **Collections**, which are like virtual folders that group together patchers and other media.

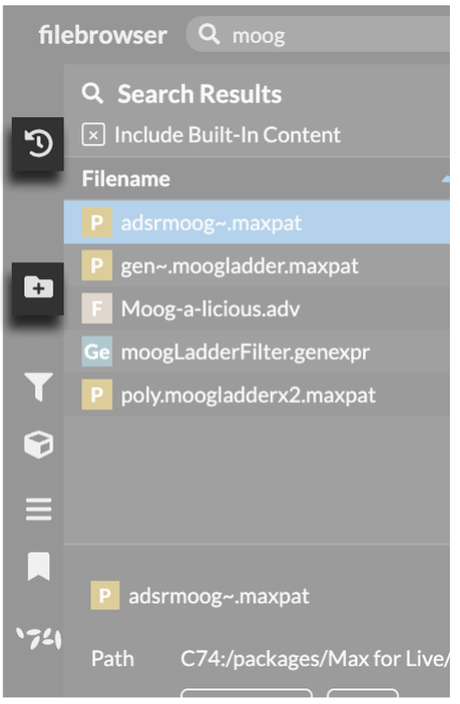
You can open the file browser from a Max patcher window by clicking the button in the left toolbar.



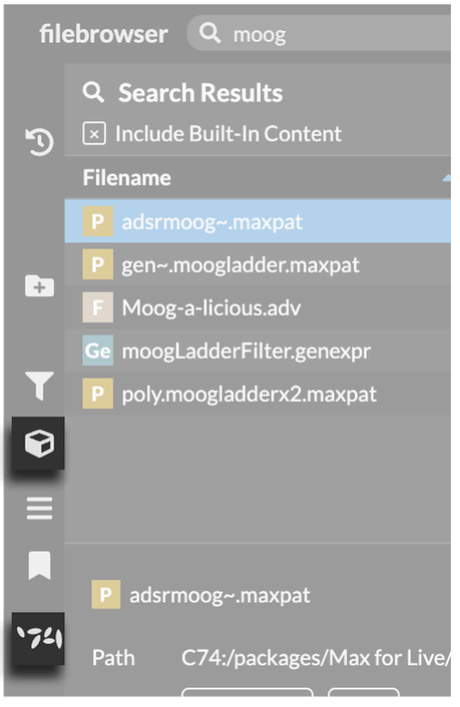
You can also select the *Show File Browser* option from the *File* menu.

Browsing

From the navigation bar on the left of the File Browser window, you can view **Recently Used** items, **Recently Added** items.



You can also browse content by package, or browse all of the content that comes built-in with Max.



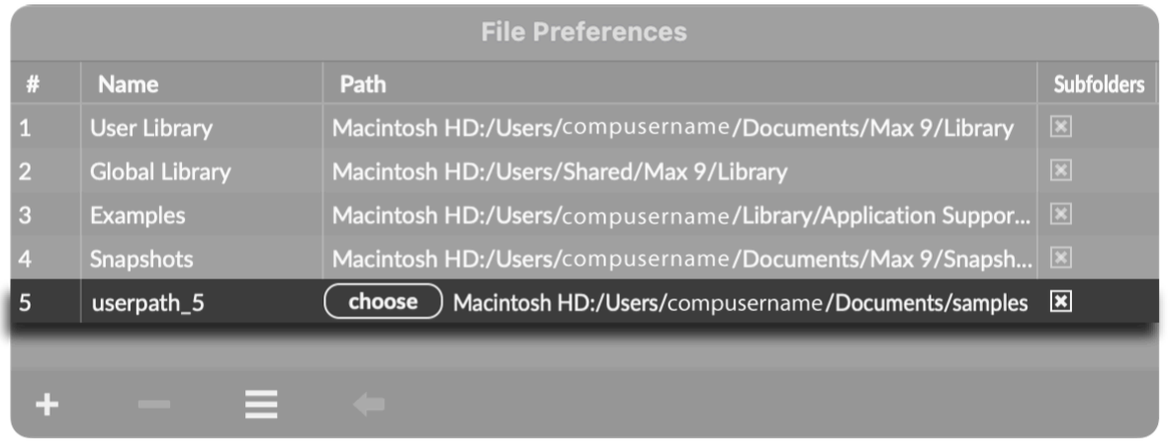
Adding Files to the Search Path

At the bottom of the File Browser window, there's a button that you can use to add files to the search path and, by extension, the file browser.



Clicking this button will bring up a system dialog box that you can use to browse for files or folders to add to Max's search path. If you select a file, that file will be visible and searchable in the file browser. To remove that file from the search path, right-click on the file and select *Remove from Search Path*.

From the same dialog box you can also add a whole folder to Max's search path. Once you do, you can select *File Preferences* from the *Options* menu to view the path that you added. By default this will add that folder and all subfolders to the Max search path. You can remove the folder by selecting the folder and clicking the *Remove Path* button at the bottom of the *File Preferences* window.



Advanced Search

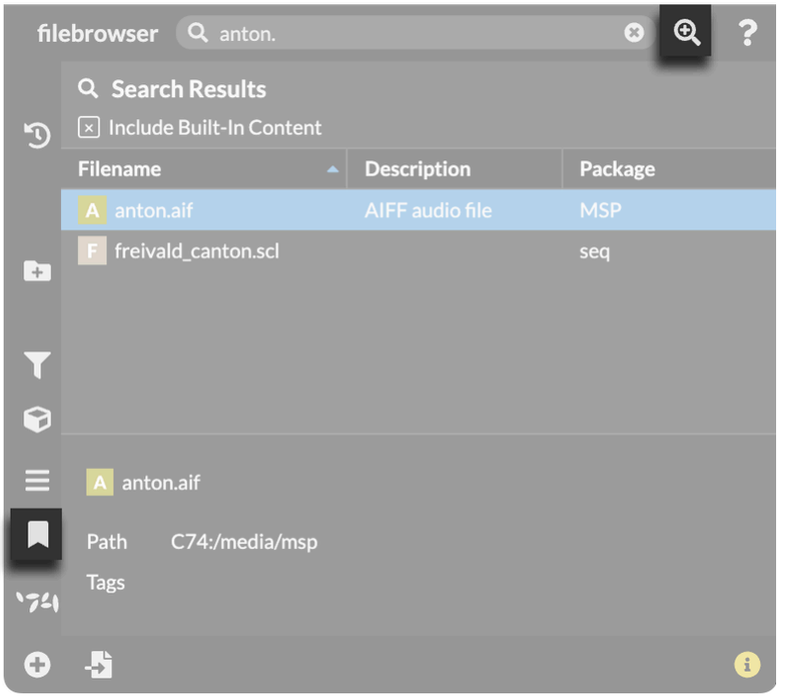
When you open the File Browser for the first time, or when you click on the *Question Mark* button in the top-right of the window, you'll see a description of the advanced search syntax for the file browser. This lets you build search queries to narrow in on just the content you're looking for. For example, the search

```
package:BLOCKS kind:audio
```

will search only for audio files in the package *BLOCKS*. When you click on buttons in the left sidebar, you may notice that these change the contents of the search box. In fact, these buttons are just shortcuts to using the advanced search terms. Clicking on the *Recently Used* button is exactly the same as starting a search with `recent:true`. Clicking on a button in the left sidebar and then adding additional search terms is a convenient way to build up a complex search term.

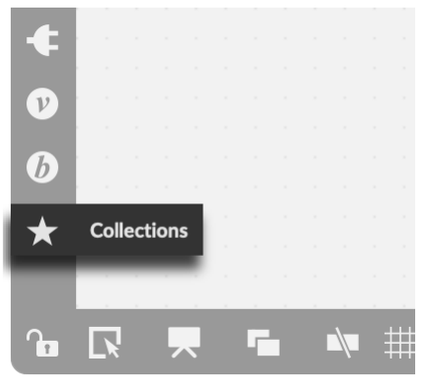
Bookmarks

If you want to save a search query for later, you can click on the *Bookmark Search* button in the top-right of the File Browser window, next to the search bar. This button creates a new **Saved Search** or **Bookmark** so you can easily find it again later. And once you've created a Saved Search, you can find it using the Saved Search button in the left sidebar.



Collections

Collections group together patchers, media, and saved searches. In addition to helping you stay organized, anything that you add to a Collection will also be added to the Max search path. You can also access a Collection from the sidebar in any Max patcher, making them convenient for accessing files that you use frequently.



Creating a Collection

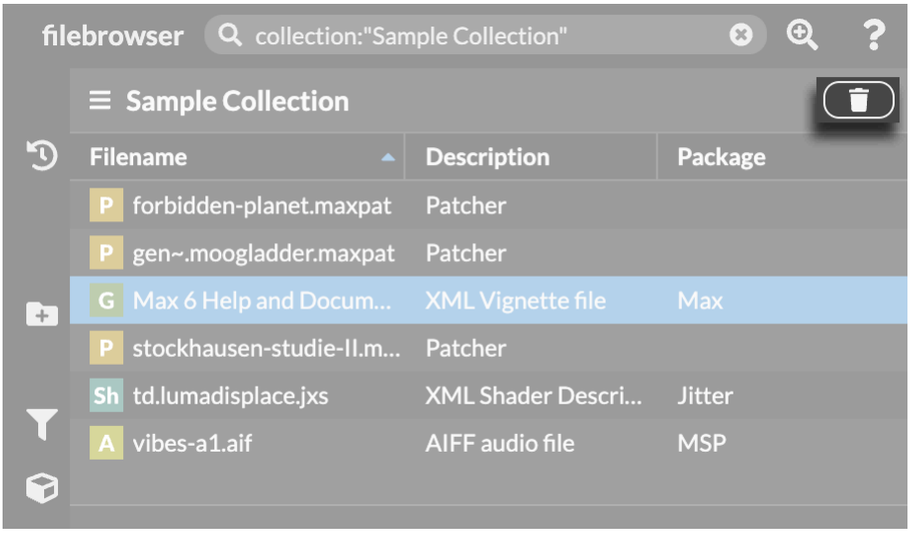
Create a new Collection by clicking the *Create New Collection* button in the bottom-left of the File Browser window.



It's also possible to right-click on any file in the File Browser and select *Add to Collection* or *Create Collection with selected file*.

Removing a Collection

To remove a collection, first open the collection in the File Browser. Then, click the *Garbage Can* icon in the top-right of the collection viewer.



File Types

Several Max objects such as folder, umenu, and opendialog can refer to specific File Types. These can be either an explicit file extension beginning with a dot (.png, .wav, .mp3, etc.), or Macintosh four-character code. Even though they're called Macintosh codes, Max will recognize these on any platform.

Here's a list of those file types, together with a description and the standard extensions associated with each type.

Code	Description	Extensions
8BPS	Photoshop file	.psd
AFxB	FXB file	.fxb
AFxP	FXP file	.fxp
AIFF	AIFF audio file	.aif
AIFF	AIFF audio file	.aiff
ampf	Ableton Live Max Device	.amp
amxd	Ableton Live Max Device	.amxd
aPcs	VST Plug-In	.dll
aPcs	VST plug-in	.vst
APPL	Application	.app
APPL	Application	.exe
AUin	Audio Unit Plug-in	.auinfo
AUpi	Audio Unit Plug-in	.component
BMP	BMP file	.bmp
CAF	CAF audio file	.caf

cafe	Java class file	.class
css	CSS file	.css
DATA	audio file	.data
DICT	Dictionary file	.json
dict	Dictionary Instance	.maxdict
FBX	Autodesk FBX Model File	.fbx
FLAC	FLAC audio file	.flac
fold	Folder/Directory	Folder
gDSP	dsp.gen File	.gendsp
GenX	GenExpr file	.genexpr
GIFf	GIF file	.gif
gJIT	jit.gen File	.genjit
iLaF	External object	.mxd
iLaF	External object	.mxe
iLaX	External object	.mxo
iLaX	Macintosh External Object	.mxo
jar	Java Archive file	.jar
Jb3d	BlitzBasic 3D Model File	.b3d
Jbvh	Biovision BVH Motion Capture File	.bvh
Jdae	Collada Model File	.dae
JiT!	Jitter data file	.jit
JiT!	Jitter data file	.jxf
jlua	Lua Source File	.lua
Jmtl	Jitter Material File	.jitmtl

Jobj	Wavefront Object Model File	.obj
Jogr	Ogre3D XML Model File	.mesh.xml
JPEG	JPEG file	.jpeg
JPEG	JPEG file	.jpg
Jply	Stanford Polygon Library Model File	.ply
JSON	JSON	.json
JSON	Defaults file	.maxdefaults
JSON	Defaults definitions	.maxdefines
JSON	Help Patcher	.maxhelp
JSON	Patcher	.maxpat
JSON	Preferences file	.maxpref
JSON	Preset file	.maxpresets
JSON	Prototype	.maxproto
JSON	Query file	.maxquery
JSON	Swatches file	.maxswatches
Jstl	Stereolithography Model File	.stl
M4a	AAC audio file	.m4a
M4V	Video file	.m4a
maxb	Help file	.help
maxb	Patcher	.mxb
maxb	Patcher	.pat
maxc	Max Collective	.clct
maxc	Max Collective	.mxc
Midi	MIDI file	.mid

Midi	MIDI file	.midi
Midi	MIDI file	.syx
mLsn	Max Lesson File	.maxlesson
mMap	Max Mapping File	.maxmap
MooV	Video	.mov
Mp3	MP3 audio file	.mp3
mPak	Packed Package	.maxpack
MPEG	Video	.mpeg
MPEG	Video	.mpg
mpg4	Video	.mp4
mPrj	Project	.maxproj
mQur	Max Collection File	.maxcoll
mSnp	Snippet file	.maxsnip
mx@c	Collective	.mxf
mx64	Windows x64 external object	.mxe64
mxPL	Max Palette File	.maxpalette
mZip	Packed Project	.maxzip
NxTS	NeXT/Sun audio file	.snd
PICS	PICS file	.pics
PICT	PICT file	.pct
PICT	PICT file	.pict
PNG	PNG file	.png
PNG	PNG file	.png
PNGf	PNG file	.png

pSto	Pattrstorage Preset File (JSON)	.json
pStx	Pattrstorage Preset File (XML)	.xml
svg	SVG file	.svg
SWFL	Flash file	.swf
TEXT	OpenGL Shading Language file	.glsl
TEXT	Help file	.help
TEXT	Web page	.htm
TEXT	Web page	.html
TEXT	Java source file	.java
TEXT	Javascript file	.js
TEXT	XML OpenGL Pass Description file	.jxp
TEXT	XML Shader Description file	.jxs
TEXT	XML Reference file	.maxref.xml
TEXT	XML Tutorial file	.maxtut.xml
TEXT	XML Vignette file	.maxvig.xml
TEXT	Patcher	.mxt
TEXT	Patcher	.pat
TEXT	Text file	.txt
TEXT	Web page	.xhtml
TEXT	XML file	.xml
TIFF	TIFF file	.tif
TIFF	TIFF file	.tiff
ULAW	SND file	.snd
VfW	AVI video	.avi

WAVE	WAVE audio file	.wav
WMV2	Video File	.asf
WMV3	Video File	.wmv
WMVA	Video File	.wmv
WVC1	Video File	.wmv
xdll	Windows external object	.mxe
xQZZ	Support file	.hibundle
XSLT	XSLT file	.xsl
YAML	YAML	.yaml
YAML	YAML	.yml
ZIP	Zip archive	.zip

Search Path

Editing the Search Path 177
Max 9 Folder 179
Path Objects 180
Projects 182
Standalones 182

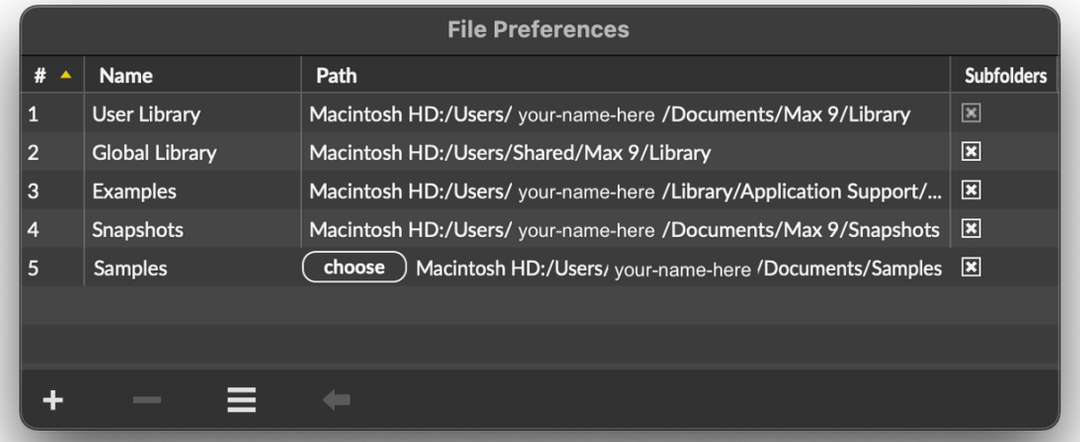
The **Search Path** is a collection of folders that Max looks through whenever it needs to find a file. It lets you reference files based on just their name, rather than their absolute path. To locate a file, Max looks in the following places, in order:

- The folder containing the currently open patch
- The **project** to which the current patcher belongs, if any
- The Max 9 Folder, including built-in patches and media, installed **Projects**, and others
- The folders in the current search path, in order

The **File Browser** lets you search, organize, and filter files in the current search path.

Editing the Search Path

Edit the search path by selecting *File Preferences...* from the *Options* Menu. This will open the *File Preferences* window.

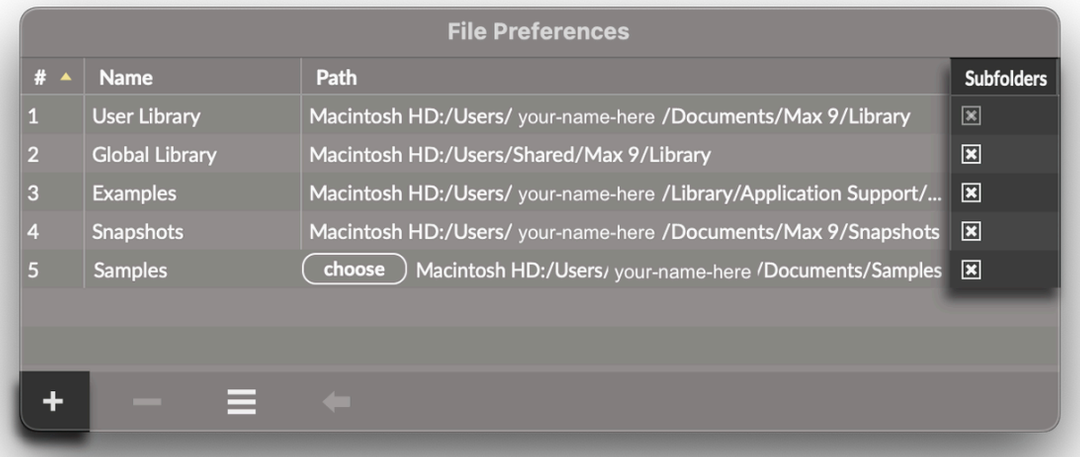


The paths to the folders for *User Library*, *Global Library*, *Examples*, and *Snapshots* are managed by Max and cannot be modified. If you want Max to search other folders when looking for files, you can add your own folders using this window.

> Select any path and click the *Reveal in Finder* buttom in the bottom toolbar to open that folder in *Finder* or *File Explorer*.

Adding a file to the search path

Click on the *Add Path* button in the bottom-left of the window. This will add a new, empty row to the search path list. Click the *Choose* button to open a dialog box that lets you browse for the folder you'd like to add to the search path. You can also name your path by double-clicking on the *Name* field. If you'd like to include subfolders as well, check the *Subfolders* box.



Removing a file from the search path

Select any path and click the *Remove Path* button in the bottom toolbar to remove it from the search path.

Listing all folders in the search path

You can click the *List Path* button in the bottom toolbar to list all of the folders currently in the search path. This includes subfolders and can include *a lot* of folders.

Max 9 Folder

Max adds a folder to the *Documents* directory called `Max 9`, containing files of different kinds that get added to Max as you work with it. This folder is located at `%HOMEDRIVE%%HOMEPATH%\Documents\Max 9` on Windows and `~/Documents/Max 9` on other operating systems.

> The *Library* and *Snapshots* folders are automatically added to the search path. Folders in the *Paackages* folder are also added, as each package is loaded.

- *Collections* - Collections that you make using the File Browser get stored here
- *Library* - This folder is for your use. Whatever files you put in this folder will be included in Max's search path.
- *Packages* - Packages instlaled by the Package Manager get installed here.
- *Palettes* - Palettes that you create using the Color Picker get stored here.
- *Projects* - This is the default location to save **projects**. Max will also unpack **amxd**s here.
- *Prototypes* - Any saved **Prototypes** will be stored here.
- *Recordings* - The default directory for audio **Recordings**.
- *Snapshots* - New **Snapshots** get saved here.
- *Snippets* - When you create a **Snippet**, it gets stored here.
- *Styles* - New **Styles** get saved here.
- *Templates* - Saved **Templates** go to this directory.

Path Objects

Several objects facilitate working with Max's search path:

Name	Description
absolutepath	Convert a file name to an absolute path
conformpath	Convert file path styles
filepath	Manage and report on the Max search path
relativepath	Convert an absolute to a relative path
strippath	Separate filename from a full pathname

Path Prefixes

When resolving a file path, Max can use special path prefixes to locate files relative to a known location. For example, you could use the path `Patcher:/sources/my_patcher.maxpat` to locate a patcher relative to the current patcher, even if the patcher `my_patcher.maxpat` is not in the current *search path*. This works for all objects that can load files. The absolutepath and conformpath objects can be used to illustrate where Max is resolving these relative paths on disk.

Prefix	Description	Example
~:	path relative to the user's home folder	`~:/sources/my_patcher.maxpat`
C74:	path relative to the Cycling '74 resources folder (on macOS, this is inside the application bundle `Max.app/Contents/Resources/C74/`, on Windows, this is the resources folder next to the `Max.exe` executable)	`C74:/sources/my_patcher.maxpat`
C74_AU	Max-specific plugin directory for AudioUnit plugins. Mostly used with the `plug` message to vst~ to disambiguate plugin types.	`C74_AU:/MyPluginName
C74_VST	Max-specific plugin directory for VST plugins. Mostly used with the `plug` message to vst~ to disambiguate plugin types.	`C74_VST:/MyPluginName
C74_VST3	Max-specific plugin directory for VST3 plugins. Mostly used with the `plug` message to vst~ to disambiguate plugin types.	`C74_VST3:/MyPluginName
Usermax:	path relative to the Max 9 folder in the user's Documents folder	`Usermax:/sources/my_patcher.maxpat`
Desktop:	path relative to the user's Desktop folder	`Desktop:/sources/my_patcher.maxpat`

Tempfolder:	path relative to Max's temp folder (this folder will be automatically emptied when Max quits)	`Tempfolder:/sources/my_patcher.maxpat`
Package:	path relative to the package specified in package-name	`Package:/miraweb/misc/app.js`
Project:	path relative to the project (if any) containing the file loading object	`Project:/sources/my_patcher.maxpat`
Patcher:	path relative to the patcher (if any) containing the file loading object	`Patcher:/sources/my_patcher.maxpat`

Projects

Projects collect and organize dependencies. All files in a given project will be able to locate other files in the same project. In addition, projects support Project Search Paths, which are extra search path folders specified by that project. For more details, see the documentation for Projects.

> Max for Live devices are just projects, and follow the same rules as projects when locating files using the search path.

Standalones

Search paths in standalones work more or less the same as in regular Max, with a couple of small differences. Check out the documentation for Standalones for more details.

Gen

Gen

Why Use Gen? 185
Working with Gen 186
Creating a Gen Patch 186
Patching in Gen 187
Auto-compile 188
Gen Operators 188
Standard Operators 190
Argument Expressions 191
Send and Receive 191
Subpatchers and Abstractions 192
Subpatcher/Abstractions and Parameters 194
Setting Parameter Defaults 197
The gen~ Object 198
gen~ Operators 199
History 200
Delay 200
Data and Buffer 201
Event-rate Gen 203
Technical notes 204
Jitter Gen Objects 204
Jitter Operators 204
jit.gen 208
jit.pix 209
jit.gl.pix 210
Technical notes (Jitter Gen) 210
See Also 212

Gen is a visual, graph-based programming language. It's syntax is extremely similar to Max, so similar that we program Gen patchers using the Max environment. However, Gen uses a different set of objects from Max—within a gen~ subpatcher, you can only use Gen objects.

The main difference between Max and Gen is that while Max works by sending messages between instantiated objects, a Gen patcher is more like a description of how a patcher should be. In order to

process audio or computer graphics, a Gen patcher is first used to generate code, which is then compiled and run in the Max environment.

Why Use Gen?

- You want to do low-level, visual programming, while still getting the performance of compiled C or GLSL code.
- It's easier to describe your process using text code, but you want to use codebox, rather than compiling a C object.
- You need single-sample delay for signal processing techniques like filtering, synthesis, and physical modeling.
- You want to design a graphics shader, but you don't want to write GLSL code.
- You want to export your C or GLSL code, for use in an environment other than Max.

Examples

- arbitrary new oscillator and filter designs using single-sample feedback loops with gen~
- reverbs and physical models using networks of short feedback delays with gen~
- sample-accurate buffer~ processing such as waveset distortions with gen~
- efficient frequency-domain processing such as spectral delays using gen~ inside pfft~
- custom video processing filters as fast as C compiled externals with jit.pix, and graphics card accelerated with jit.gl.pix
- geometry manipulation and generation with jit.gen
- particle system design with jit.gen
- iso-surface generation with distance fields in jit.gen

Performance Improvements

- A chain of Gen objects compiles down into one single meta-object, removing the usual overhead that Max encounters when passing messages and signals between objects.
- replacement for jit.expr with performance and interface improvements

- You want to be able to have a simple way to make use of the GPU for image processing both in visual and textual form

Working with Gen

Gen patchers are specialized for specific domains such as audio (MSP) and matrix and texture processing (Jitter). The Max Gen object is called gen. The MSP Gen object is called gen~. The Jitter Gen objects are jit.gen, jit.pix and jit.gl.pix. Each of these Gen objects contains within it a Gen patcher. While gen patchers share many of the same capabilities, each Gen object has functionality specific to its domain. For example, Gen patchers in gen~ have delay lines while Gen patchers in jit.gen have vector types.

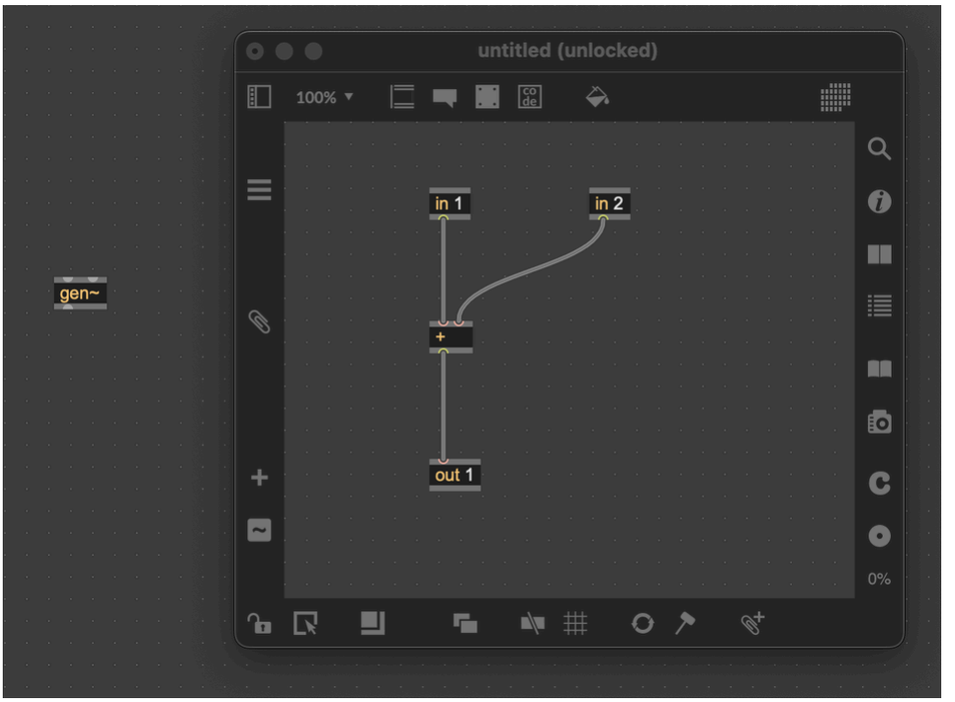
- A listing of operators common to all Gen objects
- A listing of operators common to all gen~ objects
- A listing of operators common to all Gen Jitter objects

Creating a Gen Patch

Create a gen or gen~ object for message and signal processing, and create a jit.gen, jit.pix, or jit.gl.pix for image processing.

- gen — create a processing object that runs at message rate, using gen common and gen dsp operators.
- gen~ — create a signal object that runs at signal rate, in Max's DSP chain, using gen common and gen dsp operators.
- jit.gen — create a matrix processing object that runs on the CPU. Uses gen common and gen jitter operators.
- jit.pix — create a matrix processing object that runs on the CPU, but that's optimized to work on frames of video as opposed to more general matrix types. Always locked to 4 planes, but somewhat more efficient than jit.gen. Uses gen common and gen jitter operators.
- jit.gl.pix — create a texture processing object that runs on the GPU. Always locked to 4 planes, but much faster and more efficient than jit.gen or jit.pix. Compiles to GLSL code, which can be exported and run in other graphics processing environments. Uses gen common and gen jitter operators.

Whichever object you make, you'll get a Gen object that contains a Gen subpatcher. Just like a patcher object, you can double-click on the Gen object to see its contents. When you do, you'll see the default Gen patcher, which simply adds together its inputs.



A gen~ object, and its contents

Gen patchers can be embedded within the gen, gen~, jit.gen, etc. object, or can be loaded from external files (with `.gendsp` or `.genjit` file extensions respectively) using the `@gen` attribute of gen, gen~, jit.gen, etc. objects.

Patching in Gen

Patching in Gen should feel very similar to patching in Max. The basic Max paradigms—making objects, connection them—are all the same.

The Gen patcher window has some small differences from the standard Max window, in order to facilitate patching in Gen.

Conceptual differences between Max and Gen

- In Max, objects send messages to each other. In Gen, there are no messages. All operations are synchronous, much like signal flow in Max. Because of this, there are no UI objects (sliders, buttons etc.). However the param operator can be used to receive message-rate controls from the normal Max world. There is no need to differentiate hot and cold inlets, or the order in which outlets "fire", since all objects and outlets always fire at the same time.
- There are no `send` and `receive` operators in Gen patcher. Gen patchers are connected to the outside world through the in, out, and param operators. In gen~, there are some additional operators such as history, data and buffer that are controllable with messages to gen~.
- The usual distinction between int and float numbers does not apply to Gen patchers. At the Gen patcher level, everything is a 64-bit floating point number.
- The `codebox` is a special operator for Gen patchers, in which more complex expressions can be written using the GenExpr language.

Auto-compile

Remember that Gen patchers must be compiled before they can run. By default, the compilation process occurs in the background while you are editing, so that you can see or hear the results immediately. This auto-compilation process can be disabled using the *Auto-Compile* toggle in the Gen patcher toolbar. With auto-compilation disabled, click the *Compile* icon to compile manually.

See the Gen patcher differences section of the Patcher Window guide for more.

Gen Operators

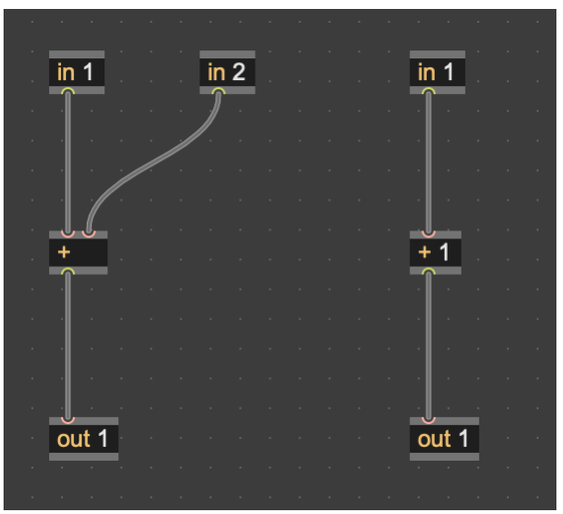
The fundamental processing object in Max is the Object, and the fundamental processing unit in Gen is the Gen Operator. You can call an operator by creating an object box in Gen, or by calling the operator as a function in GenExpr code.



Patching into a change operator is the same as calling the change function in GenExpr

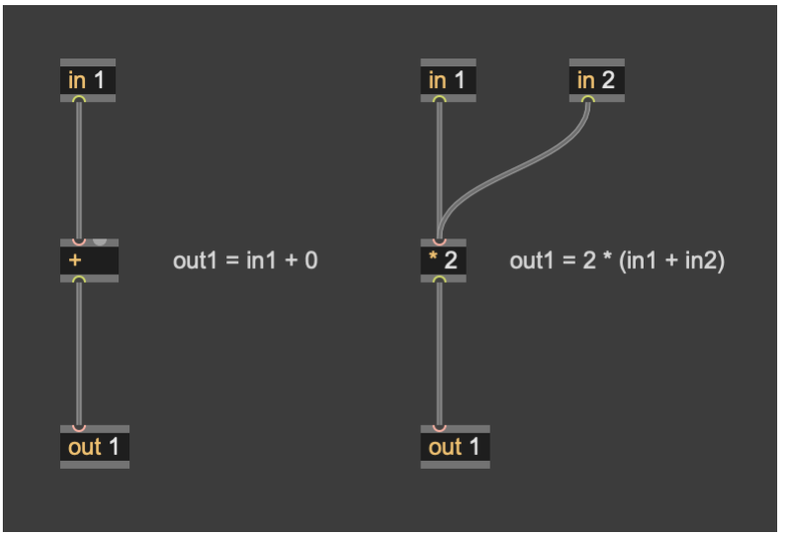
Gen operators take arguments and attributes just like Max objects, but these are purely declarative. Since there is no messaging in Gen patchers, the attribute value set when the operator is created does not change. Attributes are most often used to specialize the implementation of the process the operator represents (such as setting a maximum value for `param` using the `@max` attribute.)

In many cases, the specification of an object's argument effectively replaces the corresponding inlet. This is possible in Gen because there is no messaging and all processing is synchronous. For example, the `+` operator takes two inputs, but if an argument is given only one input needs to be specified as an inlet.



If the + operator has an argument, it no longer needs a second inlet

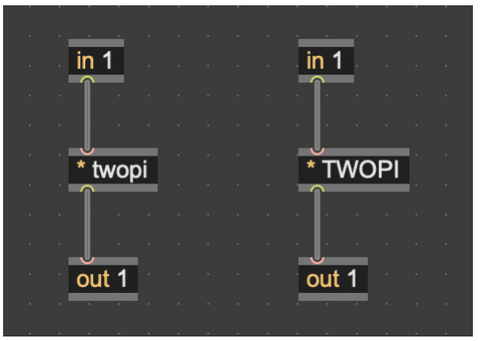
An inlet with no connected patchcord uses a default value instead (often zero, but check the inlet assist strings for each operator). An inlet with multiple connections adds them all together, just like with signal patchcords.



In gen and gen~, two patch cables connected to the same operator inlet will add together.

Standard Operators

Many standard objects behave like the corresponding Max or MSP object, such as all arithmetic operators (including the reverse operators like `!-`, `!/` etc.), trigonometric operators (`sin`, `cosh`, `atan2` etc.), standard math operators (`abs`, `floor`, `pow`, `log`, etc.), boolean operators (`>`, `==`, `&&` (also known as `and`) etc.) and other operators such as `min`, `max`, `clip` (also known as `clamp`), `scale`, `fold`, `wrap`, `cartopol`, `poltocar` etc. In addition there are some operators in common with GLSL (`fract`, `mix`, `smoothstep`, `degrees`, `radians` etc.) and some drawn from the jit.op operator list (`>p`, `==p`, `absdiff` etc.). There are several predefined constants available (`pi`, `twopi`, `halfpi`, `invpi`, `degtorad`, `radtodeg`, `e`, `ln2`, `ln10`, `log10e`, `log2e`, `sqrt2`, `sqrt1_2` and the same in capitalized form as `PI`, `TWOPI` etc), which can be used in place of a numeric argument to any operator.



Multiply by the constant pi using twopi or TWOPI.

Argument Expressions

For all objects that accept numeric arguments (e.g. `[+ 2.]` or `[max 1.]`) argument expressions can be used in their place. Argument expressions are simple statements with known inputs such as constants, Gen patcher inputs, and parameter names. Many gen operators can be used as argument expressions, particularly the math operators (sqrt, cos, etc.). Argument expressions can help simplify Gen patchers where all that is needed is the calculation of a constant that isn't pre-defined, such as $3 * pi/23 *$ pi/2. For example, in the patch below, there is a scale operator with an argument of `sqrt(2)*2`.

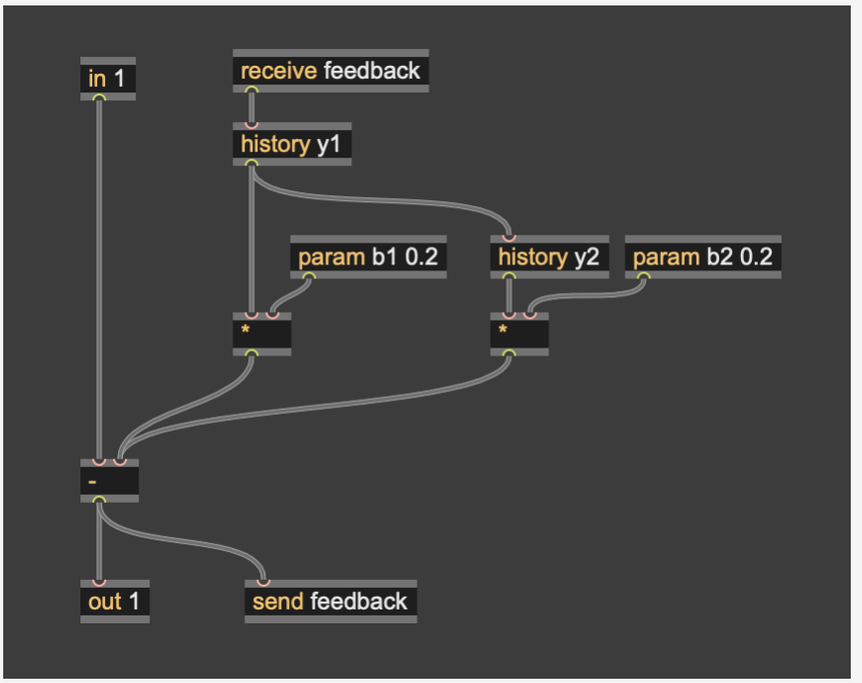


Similarly, the mul (*) operator has an argument expression of `1+in2`. Since `in2` is the GenExpr equivalent of `[in 2]`, it can be used in an argument expression.

Send and Receive

send and receive within gen patchers can be used to connect objects without patchcords. In gen patchers, send and receive can only be used locally. They will not connect to send and receive objects in other gen patchers or gen subpatchers.

send and receive take a name argument that determines connectivity.



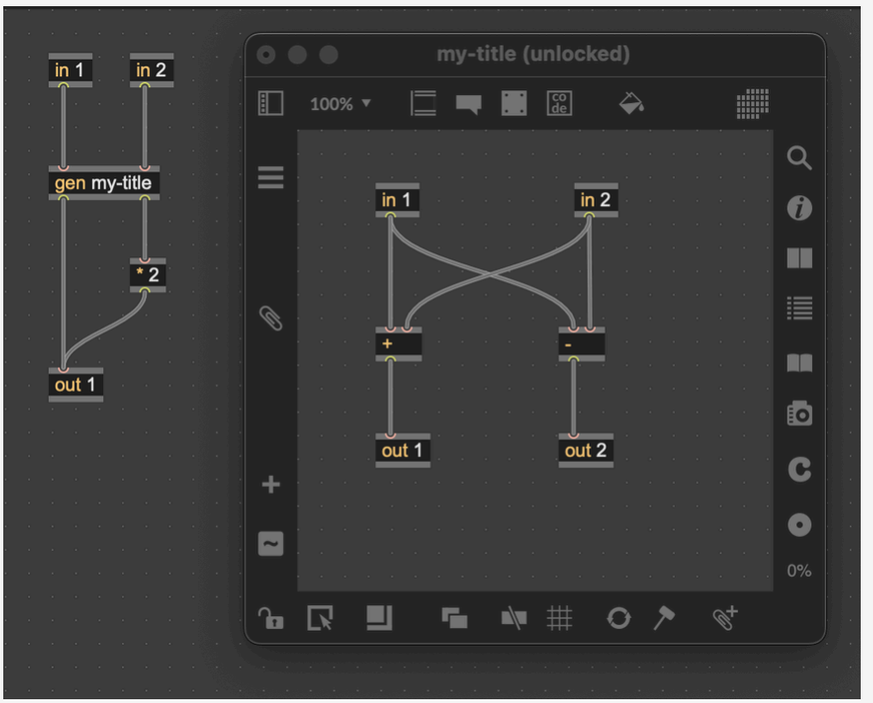
Using send and receive in a Gen patcher

There can be multiple send and receive objects with the same name without issue. If there are multiple send objects with the same name, they will be summed just as if multiple patchcords were connected to the same inlet. If there are multiple receive objects with the same name, they will all receive identical input from their corresponding send objects.

Subpatchers and Abstractions

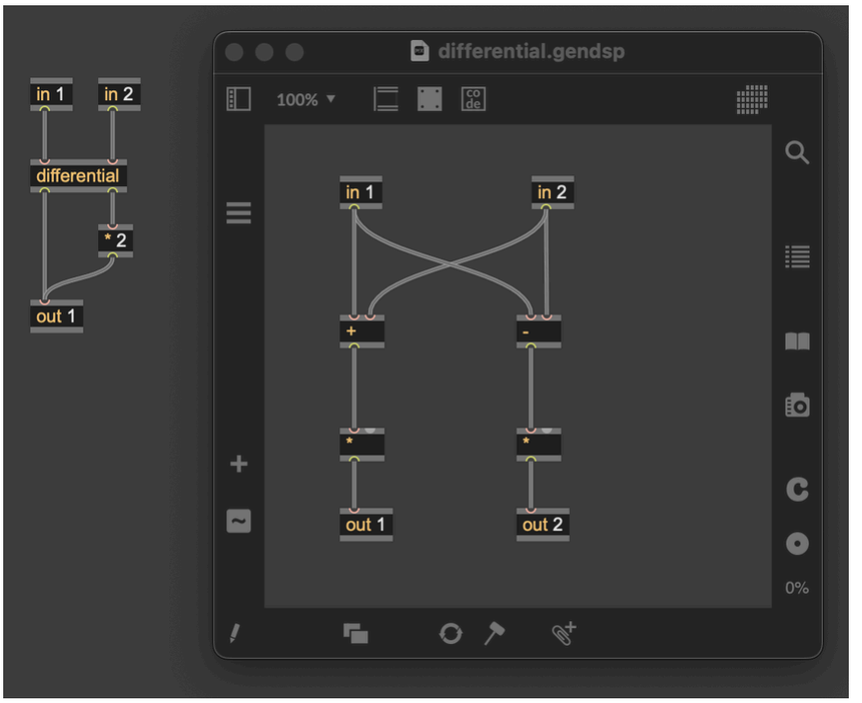
Subpatchers and abstraction in Gen objects behave practically identically to standard Max subpatchers and abstractions. In Gen objects, subpatchers are created with the gen operator. If the

gen operator is given the name of a Gen patcher as an argument, it will use it to set the titlebar of the subpatcher.



Using the gen operator to make subpatchers in Gen

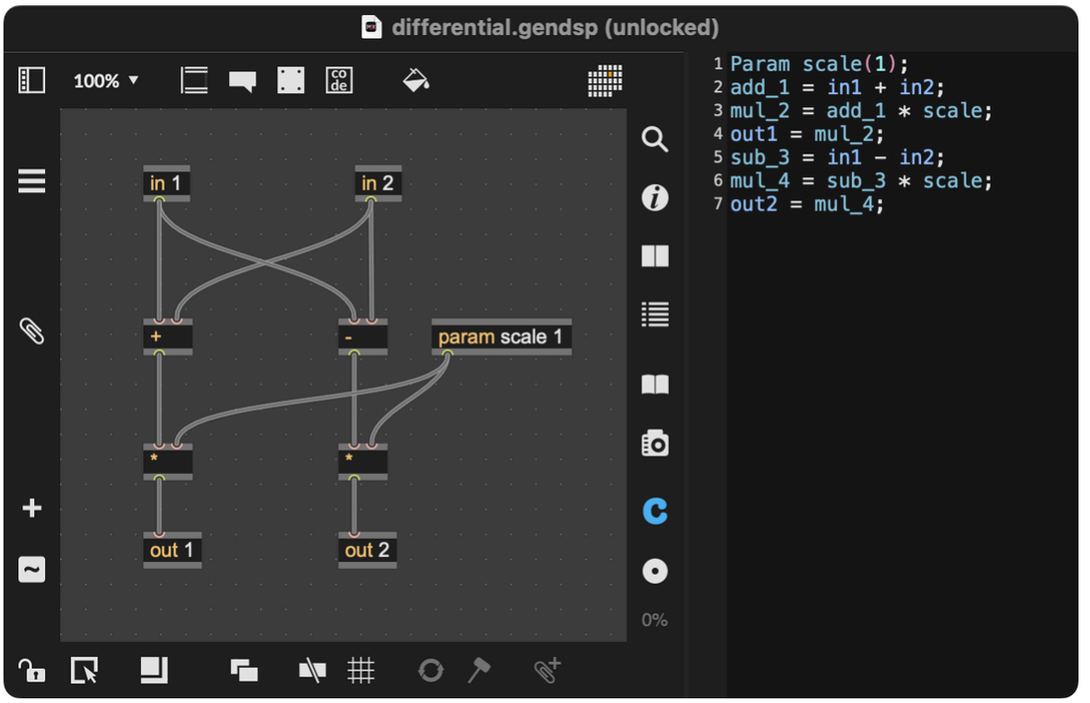
Abstractions, as with standard max abstractions, are created by saving a Gen patcher, then instantiated by creating an object with the name of the saved Gen file to load as the abstraction. For example, if an operator named `differential` is created, gen will look for the file `differential.gendsp` with gen~ and `differential.genjit` with the Jitter Gen objects. Instantiating abstractions this way is shorthand for setting the `@file` attribute on the gen operator. For example, creating an operator `differential` is equivalent to `gen @file differential`. Abstractions of gen~ and gen patchers save with the `.gendsp` file extension and abstractions of jit.gen, jit.pix and jit.gl.pix save with the `.genjit` file extension.



> Save a Gen abstraction by choosing *Save As...* from the *File* menu when focused on a Gen subpatcher.

Subpatcher/Abstractions and Parameters

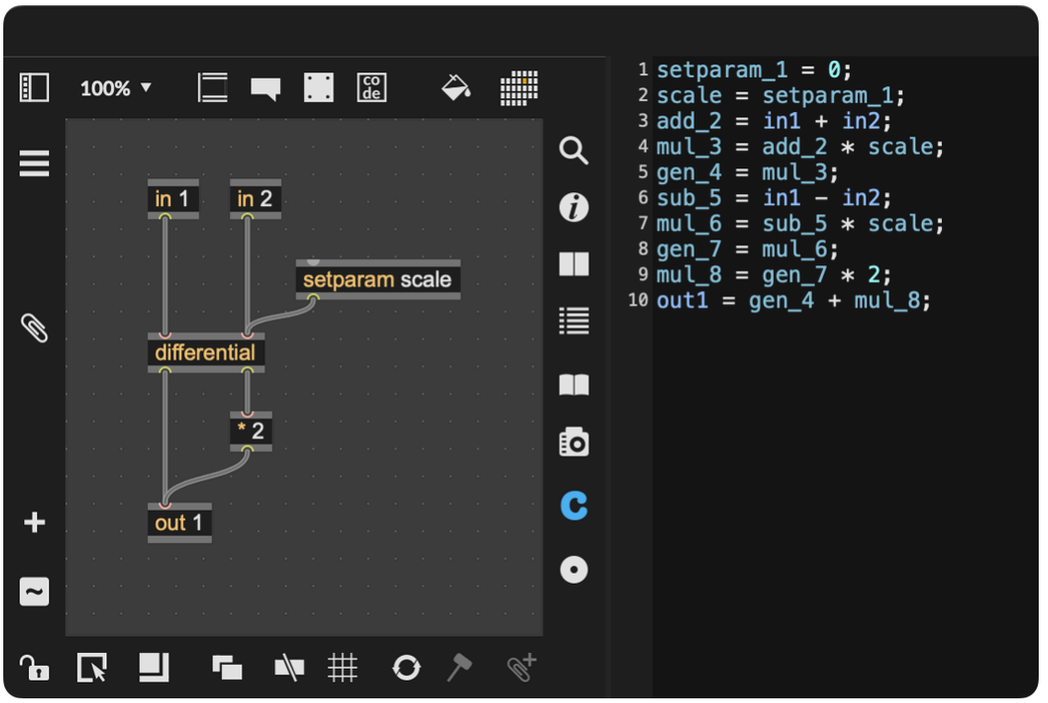
Just like normal gen patchers, Gen subpatchers and abstractions can also contain parameters. When used in subpatchers and abstractions, parameters behave like named inlets with default values. If nothing is connected to a parameter in a subpatcher or abstraction, the parameter will be a constant and its value will be its default.



In the above example, the subpatcher has a parameter `@scale` with a default of 1. In the subpatcher's sidebar, we see this represented in the GenExpr code as

```
Param scale(1.);
```

However, in the parent Gen patcher, the parameter gets converted into a constant because nothing is connected to the parameter.



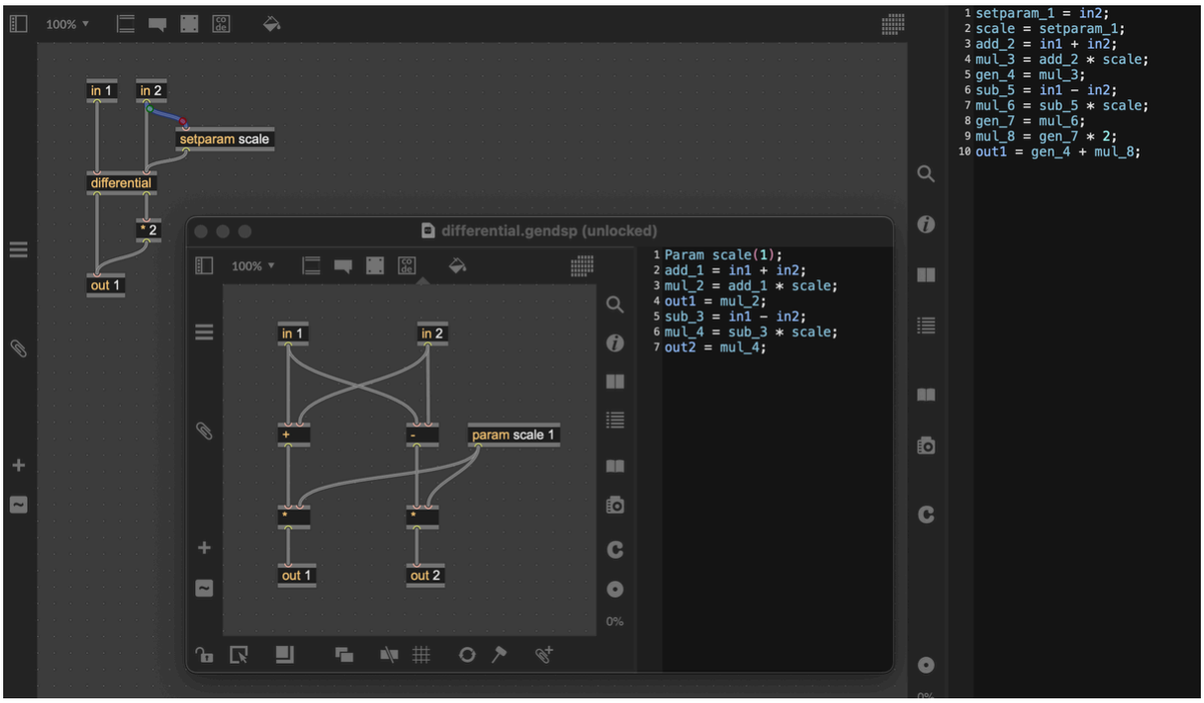
The first line in the parent patcher's GenExpr sidebar reads:

```
scale_1 = 0.;
```

This sets the `@scale` parameter to a constant value.

Since subpatcher and abstraction parameters don't create their own inlets to connect objects to, there is a special operator called setparam that can be connected to any inlet for this specific purposes. setparam connects all of its inputs to a named parameter in a subpatcher or abstraction. It requires an argument specifying the name of the parameter to connect to.

When setparam is connected to a parameter, the parameter changes from being a constant to a dynamic variable equivalent to the value at the input of the setparam object.



Notice that the code in the parent subpatcher has changed from a constant to:

```
setparam_1 = in2;
```

`in` is conected to the inlet of the setparam object so the scale parameter takes on that value.

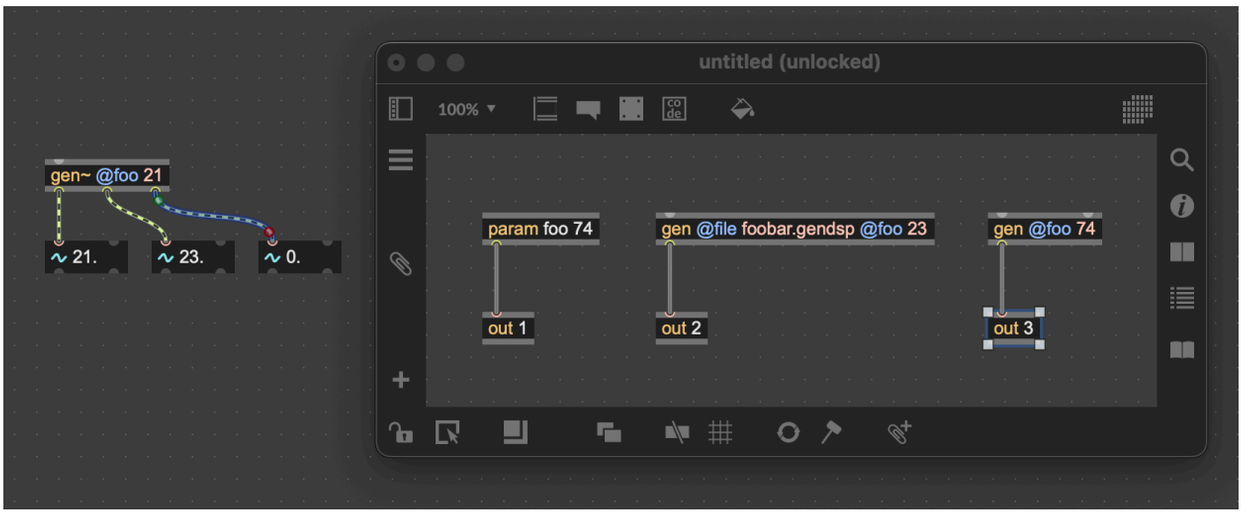
Setting Parameter Defaults

The default value for param objects within Gen patchers and subpatchers can be set either directly in the param object in the form `param paramnamevalue(s)` or in the containing gen~ object box in the form `gen~ @paramname value(s)`. If a default is declared in both the param object and in the containing gen object box, the object box will override the value declared in the param object.



The parameter value @foo 21 in the top level overrides the default value of 74.

This also applies to Gen subpatchers and abstractions; however, object box declared values only go one patcher deep. So `gen~ @foo 10` would set the default for any param object named `foo` in the top-level gen~ patcher, but not param objects named `foo` contained in Gen subpatchers and abstractions.



The gen~ Object

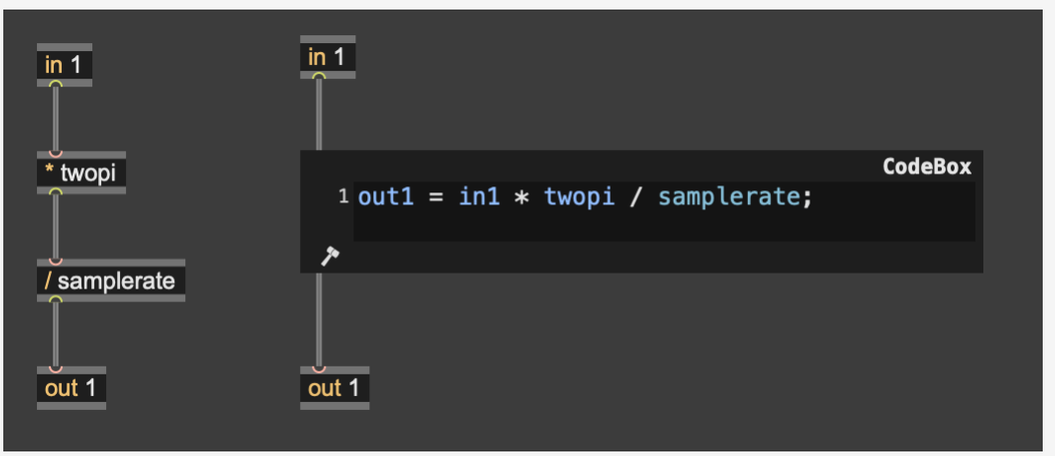
The gen~ object is specifically for operating on MSP audio signals. Unlike MSP patching however, operations in a Gen patcher are combined into a single chunk of machine code, making possible many more optimizations that can make complex processes more efficient, and allow you to design processes which must operate on a per-sample level, even with feedback loops.

Working in gen~ opens up scope to design signal processes at a lower level, even per-sample. Because of this, many operators take duration arguments in terms of samples (where the equivalent MSP objects would use milliseconds).

gen~ Operators

In addition to the standard Gen operators , which are often similar to the equivalent MSP objects (such as `clip` , `scale` , `minimum` , `maximum` , etc.), many of the operators specific to the gen~ domain mirror existing MSP objects to make the transition to gen~ easier. There are familiar converters (`dbtoa` , `atodb` , `mtof` , `ftom` , `mstosamps` , `sampstoms`), oscillators (`phasor` , `train` , `cycle` , `noise`), and modifiers (`delta` , `change` , `sah` , `triangle`). In addition there are some lower-level operators to avoid invalid or inaudible outputs (`isnan` , `fixnan` , `isdenorm` , `fixdenorm` , `dcblock`).

A global value of `samplerate` is available both as an object, and as a valid value for an argument of any object.



The samplerate value is available as an object and as a constant in codebox

History

In general, the Gen patcher will not allow a feedback loop (since it represents a synchronous process). To create a feedback loop in gen~, the history operator can be used. This represents a single-sample delay (a $Z-1$Z − 1 operation). Thus the inlet to the history operator will set the outlet value for the next sample (put another way, the outlet value of the history operator is the inlet value from the previous sample). Multiple history operators can be chained to create $Z-2$ Z − 2, $Z-3$Z − 3 delays, but for longer and more flexible delay operators, use the delay operator.



You can use the History operator as an object or in codebox.

> A history operator in a Gen patcher can also be named, making it available for external control, just like a param parameter.

Delay

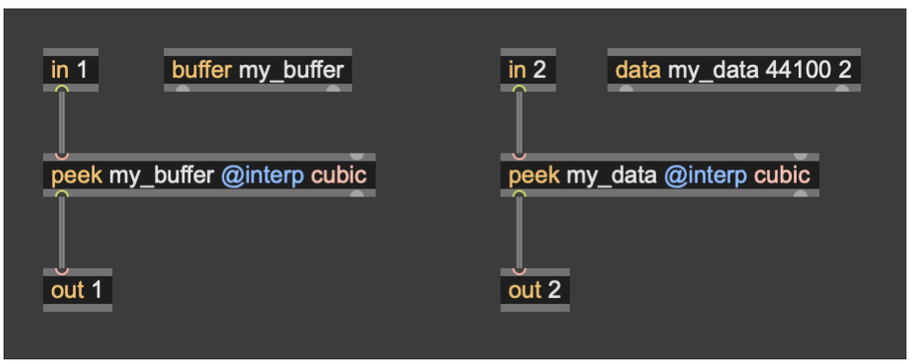
The delay operator delays a signal by a certain amount of time, specified in samples. The maximum delay time is specified as an argument to the delay object. You can also have a multi-tap delay by specifying the number of taps in the second argument. Each tap will have an inlet to set the delay time, and a corresponding outlet for the delayed signal.

The delay operator can be used for feedback loops, like the history operator, if the `@feedback` attribute is set to 1 (the default). The `@interp` attribute specifies which kind of interpolation is used:

- **none** or **step** : No interpolation.
- **linear** : Linear interpolation.
- **cosine** : Cosine interpolation.
- **cubic** : Cubic interpolation.
- **spline** : Catmull-Rom spline interpolation.

Data and Buffer

For more complex persistent storage of audio (or any numeric) data, gen~ offers two objects: data and buffer, which are in some ways similar to MSP's buffer~ object. A data or buffer object has a local name, which is used by various operators in the Gen patcher to read and write the data or buffer contents, or get its properties.



Reading the contents of a data or buffer can be done using the peek, lookup, wave, sample or nearest operators. The first argument for all of these operators is the local name of a data or buffer. They all support single- or multi-channel reading (the second argument specifies the number of channels, and the last inlet the channel offset, where zero is the default).

All of these operators are essentially the same, differing only in defaults of their attributes. The attributes are:

- **index** specifies the meaning of the first inlet:
- **samples** : The first inlet is a sample index into the data or buffer.
- **phase** : Maps the range 0..1 to the whole data or buffer contents.
- **lookup** or **signal** : Maps the range -1..1 to the whole data or buffer contents, like the MSP lookup~ object.
- **wave** : Adds extra inlets for start/end (in samples), driven by a phase signal between these boundaries (0..1, similar to MSP's wave~ object). `@boundmode` specifies what to do if the index is out of range:
- **ignore** : Indices out of bounds are ignored (return zero).
- **wrap** : Indices out of bounds repeat at the opposite boundary.
- **fold** or **mirror** : Indices wrap with palindrome behavior.
- **clip** or **clamp** : Indices out of bounds use the value at the bound. `@channelmode` specifies what to do if the channel is out of range. It has the same options as the `@boundmode` attribute.
- **interp** specifies what kind of interpolation is used:
    - **none** or **step** : No interpolation.
    - **linear** : Linear interpolation.
    - **cosine** : Cosine interpolation.
    - **cubic** : Cubic interpolation.
    - **spline** : Catmull-Rom spline interpolation.

Operator	Defaults
nearest	`@index phase @interp none @boundmode ignore @channelmode ignore`
sample	`@index phase @interp linear @boundmode ignore @channelmode ignore`
peek	`@index samples @interp none @boundmode ignore @channelmode ignore`

lookup	`@index lookup @interp linear @boundmode clamp @channelmode clamp
wave	`@index wave @interp linear @boundmode wrap @channelmode clamp`

Accessing the spatial properties of a data or buffer objects is done using the `dim` and `channels` operators (or the outlets of the data or buffer object itself), and writing is done using `poke` (non-interpolating replace) or `splat` (interpolating overdub).

Briefly, data should be thought of as a 64-bit buffer internal to the gen~ patcher, even though it can be copied to, and buffer should be thought of as an object which can read and write external buffer~ data. The full differences between data and buffer are:

- A data object is local to the Gen patcher, and cannot be read outside of it. On the other hand, a buffer object is a shared reference to an external MSP buffer~ object. Modifying the contents in a Gen buffer is directly modifying the contents of the MSP buffer~ object it references.
- The data object takes three arguments to set its local name, its length (in samples) and number of channels. The buffer object takes an argument to set its local name, and an optional argument to specify the name of an MSP buffer~ object to reference (instead of using the local name).
- Setting the gen~ attribute corresponding to a named data object copies in values from the corresponding MSP buffer~, while for a named buffer object it changes the MSP buffer~ referenced. The buffer object always has the size of the buffer~ object it references (which may change). The data object has the size of its initial definition, or the size of the buffer~ object which was copied to it (whichever is smaller).
- The data object always uses 64-bit doubles, while the buffer object converts from the bit resolution of the MSP buffer~ object (currently 32-bit floats) for all read and write operations, and may be less efficient.

Event-rate Gen

The gen~ object also has an event-rate version, called gen. Internally, the event-rate gen object supports all of the same operators as gen~. The only difference is that gen is an event-rate object, and is not part of the audio graph. Like most event-rate objects, the first inlet to gen is hot, and will trigger computation. However, you can also use the `@interval` attribute in conjunction with the

`@active` attribute to enable an internal metronome, which will trigger computation at a consistent rate. When using the event-rate gen object, the samplerate operator will return $\frac{1000}{@interval}$ @interval1000, and the vectorsize operator will always return 1.

Technical notes

All operations in gen~ use 64-bit doubles.

The compilation process for gen~ Gen patchers and GenExprs includes an optimization that takes into account the update rate of each operator, so that any calculations that do not need to occur at sample rate (such as arithmetic on the outputs of param operators) instead process at a slower rate (determined by the host patcher vector size) for efficiency.

Jitter Gen Objects

There are three Gen objects in Jitter: jit.gen, jit.pix, and jit.gl.pix. The jit.gen and jit.pix objects process Jitter matrices similar to jit.expr. The jit.gl.pix object processes textures and matrices just like jit.gl.slab. The jit.gen object is a generic matrix processing object that can handle matrices with any planecount, type and dimension. jit.pix and jit.gl.pix, on the other hand, are specifically designed for working with pixel data. They can handle data of any type, but it must be two dimensional or less and have at most four planes.

Jitter Operators

Coordinates

Jitter Gen patchers describe the processing kernel for each cell in a matrix or texture. As the kernel is processing the input matrices, a set of coordinates is generated describing the location of the current cell being processed. The objects are just like the operators in jit.expr. They are norm, snorm, and cell, with the dim operator giving the dimensions of the input matrix.

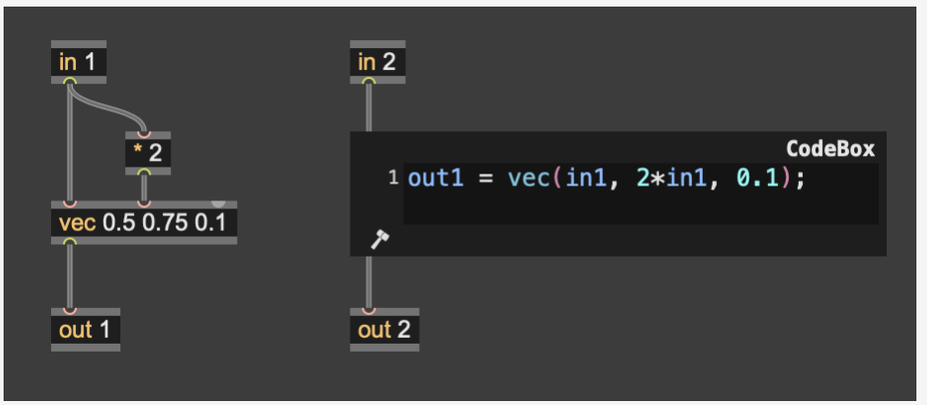
- norm ranges from `[0, 1]` across all matrix dimensions and is defined as *norm = cell/dim*.
- snorm ranges from `[-1, 1]` across all matrix dimensions and is defined as *snorm = cell/dim*2-1*.

- cell gives the current cell index.

Vectors

Since Jitter matrices represent arrays of vector (more than one plane) data, all Gen operators in Jitter can process vectors of any size, so Gen patchers once created work equally on any vector size. The basic binary operators `+`, `-`, `*`, `/`, and `%` can take vector arguments as in `[+ 0.5 0.25 0.15]`, which will create an addition operator adding a vector with the three components to its input. Also, the param operator can take vector default values as in `[param 1 2 3 4]`. Parameters can have up to 32 values in jit.gen and 4 values in jit.pix and jit.gl.pix.

The vec operator creates vector constants and packs values together in a vector. It takes default arguments for its components and casts all of its inputs to scalar values before packing them together.



You can use the vec operator as an object or in a codebox

The `swiz` operator applies a swizzle operation to vectors. In GLSL and similar shading languages, vector components can be accessed by indexing the vector with named planes. For example in GLSL you might see

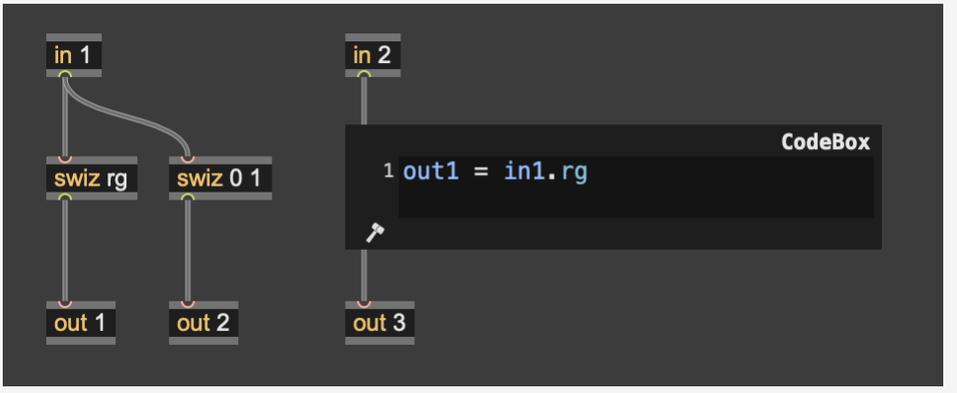
```
red = color.r
```

or

```
redalpha = color.ra
```

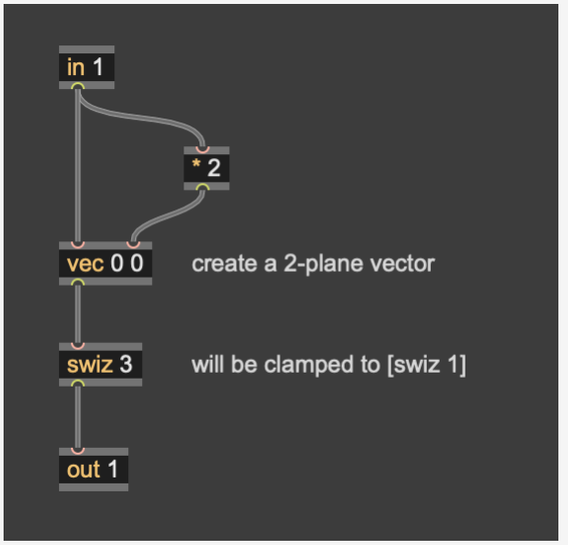
or even

```
val = color.rbbg
```



Use the swiz operator as an object to pick certain planes. In a codebox, do swizzling with a dot operator.

This type of operation is referred to as *swizzling*. The swiz operator can take named arguments using the letters `r`, `g`, `b`, `a`, as well as `x`, `y`, `z`, `w` in addition to numeric indices starting at `0`. The letters are convenient for vectors with four or less planes, but for larger vectors numeric indices must be used. The compilation process automatically checks any swiz operation so arguments indexing components larger than the vector being processed will be clamped to the size of the vector.

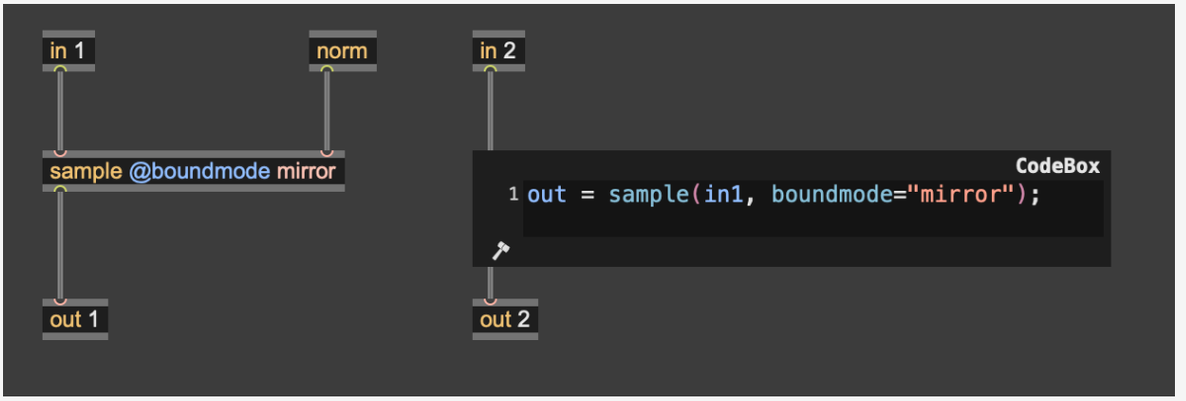


Out of bounds swiz operations will be clamped

In addition, there are the basic vector operations for spatial calculations. These are length, normalize, cross, dot, and reflect.

Sampling

Sampling operators are one of the most powerful features of Jitter Gen patchers. Sampling operators take an input and a coordinate in the range `[0, 1]` as an argument, returning the data at the coordinate's position in the matrix or texture. The first argument always has to be a Gen patcher input while the second argument is an N-dimensional vector whose size is equal to the dimensionality of the input it is processing. If the coordinate argument is outside of the range `[0, 1]`, it will be converted to a value within the range `[0, 1]` according to its boundmode function. Possible boundmodes are `wrap`, `mirror`, and `clamp`, where `wrap` is the default.



You can use the sample operator as an object or in codebox.

The two sampling operators in Jitter Gen patchers are sample and nearest. The sample operator samples values form a matrix using N-dimensional linear interpolation. The nearest operator will simply grab the value from the closest cell.

Geometry

Jitter Gen patchers include a suite of objects for generating surfaces. These include most of the shapes available in the jit.gl.gridshape object. Each surface function returns two values: the vertex position and the vertex normal. The geometry operators are sphere, torus, circle, plane, cone, and cylinder.

Color

There are two color operators in Jitter Gen patchers. They are rgb2hsl and hsl2rgb. These convert between the Red Green Blue color space and the Hue Saturation Luminance color space. If the input to these objects has an alpha component, the alpha will be passed through untouched.

jit.gen

The jit.gen object is a general purpose matrix processing object. It compiles Gen patchers into native machine code representing the kernel of an N-dimensional matrix processing routine. It follows the Jitter matrix planemapping conventions for pixel data with planes [0-4] as the ARGB channels.

jit.gen can have any number of inlets and outlets, but the matrix format for the different inputs and outputs is always linked. In other words, the matrix format (planecount, type, dimensions) of the first inlet determines the matrix format for all other inputs and outputs. jit.gen makes use of parallel processing just like other parallel aware objects in Jitter for maximum performance with large matrices.

How a matrix is processed by jit.gen is dependent on the input planecount, type, and dimension of the input matrices. In addition, there is a `@precision` attribute that sets the type of the processing kernel. The default value for precision is `auto`. Auto precision automatically adapts the type of the kernel dependent upon the matrix input type. In `auto` mode, the following mapping between input matrix type and kernel processing type is used:

- **char** maps to fixed
- **long** maps to float64
- **float32** maps to float32
- **float64** maps to float64

Other possible values for the precision attribute are `fixed`, `float32`, and `float64`. Fixed precision is the only setting that doesn't correspond to a Jitter matrix type. Fixed precision specifies a kernel type that performs a type of floating point calculation with integers using a technique called fixed-point arithmetic. It's very fast and provides more precision than 8-bit char operations without incurring the cost of converting to a true floating-point type. However, fixed-point arithmetic calculations have more error that can sometimes be visible when using the sampling operators. If there are noticeable artifacts, simply increase the internal precision to `float32`.

jit.pix

The jit.pix object is a matrix processing object specifically for pixel data. When processing matrices representing video and images, jit.pix is the best object. Internally, data is in RGBA format always. If the input has less than four planes, jit.pix will convert it to RGBA format according to the following rules:

- 1-plane, Luminance format, L to LLL1 (Luminance for RGB and 1 for Alpha)
- 2-plane Lumalpha format, LA to LLLA (Luminance for RGB and Alpha for Alpha)

- 3-plane RGB format, RGB to RGB1 (RGB for RGB and 1 for Alpha)
- 4-plane, ARGB format, ARGB to RGBA (changes the order of the channels internally)

The output of jit.pix is always a 4-plane matrix in ARGB format, which is the standard Jitter pixel planemapping. Like jit.gen, jit.pix compiles Gen patchers into C++ and makes use of Jitter's parallel processing system. jit.pix also has a precision attribute that operates exactly the same was as it does in jit.gen.

jit.gl.pix

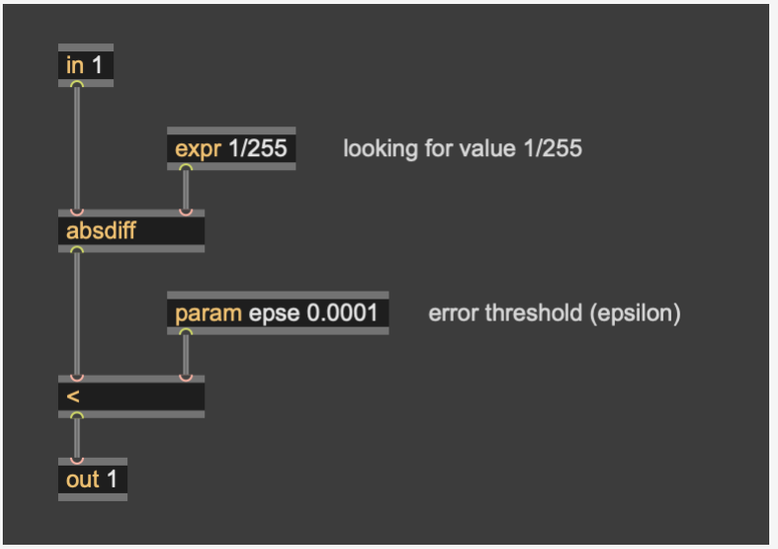
The jit.gl.pix object is a matrix and texture processing object specifically for pixel data that operates just like jit.gl.slab. The only difference between the two is that jit.gl.pix compiles its patcher into GLSL while jit.gl.slab reads it from a shader file. Like jit.pix, jit.gl.pix uses an internal RGBA pixel format.

Technical notes (Jitter Gen)

Numerical Values in the Kernel

All numerical values in Jitter Gen patches are conceptually floating point values. This is the case even for fixed precision kernels. It is particularly important to remember this when dealing with char matrices. All char matrices are converted to the range `[0, 1]` internally. On output, this range is mapped back out to `[0, 255]` in the char type. A char value of 1 is equivalent to the floating point value of 1/2551/255.

When using the comparison operators (`==` , `!=` , `<` , `<=` , `>` , `>=`), it's particularly important to keep in mind the floating point nature of Gen patcher internal values because of their inherent imprecision. Instead of directly testing for equality for example , it's more effective to test for whther a value falls within a certain small range (epsilon). In the example below, the `absdiff` operator calculates how far a value is from 1/255 and then the `<` op tests to see if it's within the threshold of error.



Rather than testing for equivalence, test whether values are within some epsilon distance of a target value.

jit.pix vs. jit.gl.pix

For the most part jit.pix and jit.gl.pix will behave identically despite one being CPU-oriented and the other GPU-oriented. The differences have to do with differences in behavior between how matrix inputs are handled with jit.pix and how texture inputs are handled with jit.gl.pix.

All of the inputs to jit.pix will adapt in size, type, and dimension to the left-most input. As a result, all input matrices within a jit.pix processing kernel will have the same values for the cell and dim operators. In jit.gl.pix, inputs can have different sizes. In jit.gl.pix, the values for the cell and dim operators are calculated from the properties of the left-most input (*in1*). A future version may include per-input cell and dim operators, but for now this is not the case.

Since the sampling operators take normalized coordinates in the range `[0, 1]`, differently sized input textures will still be properly sampled using the norm operator since its value is independent of varying input sizes. However, in jit.gl.pix the sample and nearest operators behave differently than with jit.pix. How a texture is sampled is determined by the properties of the texture. As a consequence, sample and nearest behave the same in jit.gl.pix. To enable nearest sampling, set the `@filter` attribute to nearest. For linear interpolation, set `@filter` to linear (the default).

See Also

- mc_gen

Gen Common Operators

Comparison 213
Constant 214
Declare 215
Expression 215
Ignore 215
Input-output 215
Logic 215
Math 215
Numeric 216
Powers 216
Range 217
Route 217
Subpatcher 218
Trigonometry 218
Waveform 219

The following Gen operators are common to all of Max's Gen family of objects. They can be used as operators in the gen, gen~, jit.gen, jit.pix, and jit.gl.pix objects.

Comparison

- !=p, neqp : Returns in1 if it does not equal in2, else returns zero. Equivalent to in1 * (in1 != in2)
- >, gt : Returns 1 if in1 is greater than in2, else returns zero.
- ==, eq : Returns 1 if in1 equals in2, else returns zero.
- ==p, eqp : Returns in1 if it equals in2, else returns zero. Equivalent to in1 * (in1 == in2).
- >=, gte : Returns 1 if in1 is equal to or greater than in2, else returns zero.
- >=p, gtep : Returns in1 if in1 is equal to or greater than in2, else returns zero. Equivalent to in1 * (in1 >= in2).

- >p, gtp : Returns in1 if in1 is greater than in2, else returns zero. Equivalent to in1 * (in1 > in2).
- <, lt : Returns 1 if in1 is less than than in2, else returns zero.
- <=, lte : Returns 1 if in1 is equal to or less than in2, else returns zero.
- <=p, ltep : Returns in1 if in1 is equal to or less than in2, else returns zero. Equivalent to in1 * (in1 <= in2).
- <p, ltp : Returns in1 if in1 is less than in2, else returns zero. Equivalent to in1 * (in1 < in2).
- max, maximum : The maximum of the inputs
- min, minimum : The minimum of the inputs
- !=, neq : Returns 1 if in1 does not equal in2, else returns zero.
- step : Akin to the GLSL step operator: 0 is returned if in1 < in2, and 1 is returned otherwise.

Constant

- constant : A constant value
- degtorad, DEGTORAD : Multiplicative constant to convert degrees to radians
- e, E : Base of the natural logarithm
- f, float : Either outputs a constant float or converts its input value to a float
- halfpi, HALFPI : One half of the constant pi
- i, int : Either outputs a constant integer or converts its input value to an integer.
- invpi, INVPI : One over pi
- ln10, LN10 : The natural log of 10
- ln2, LN2 : The natural log of 2
- log10e, LOG10E : Log base 10 of the constant e
- log2e, LOG2E : Log base 2 of the constant e
- PHI, phi : $\frac{1+sqrt(5)}{2}$21+sqrt(5), the "golden" ratio
- pi, PI : The constant pi, the ratio of a circle's circumference to its diameter
- radtodeg, RADTODEG : Multiplicative constant to convert radians to degrees
- sqrt1_2, SQRT1_2 : One over the square root of 2

- sqrt2, SQRT2 : The square root of 2
- twopi, TWOPI : Two times pi

Declare

- param, Param : Named parameters can be modified from outside the gen patcher. The first argument specifies the name of the parameter, the second argument the initial value.

Expression

- expr : Evaluates GenExpr code. Standard mathematical operators (+, -, *, / etc.) and gen patcher operators can be used. See the GenExpr documentation for more detail.

Ignore

- pass : Passes the value through unchanged.

Input-output

- in : Defines an input for a gen patcher.
- out : Send output from a gen patcher

Logic

- !, not : An input value of zero returns 1, any other value returns zero.
- &&, and : Returns 1 if both in1 and in2 are nonzero.
- bool : Converts any nonzero value to 1 while zero passes through.
- or, || : Returns 1 if either in1 or in2 are nonzero.
- ^^, xor : Returns 1 if one of in1 and in2 are nonzero, but not both.

Math

- !%, rmod : Reverse modulo (remainder of second input / first input)
- !-, rsub : Reverse subtraction (subtract first input from second)
- %, mod : Modulo inputs (remainder of first input / second input)
- +, add : Add inputs
- -, sub : Subtract inputs
- /, div : Divide inputs
- absdiff : Compute the absolute difference between two inputs using the equation $abs(in1 - in2)$abs(in1 − in2).
- cartopol : Convert Cartesian values to polar format. Angles are in radians.
- *, mul : Multiply inputs
- neg : Negate input
- poltocar : Convert polar values to Cartesian format. Angles are in radians.
- !/, rdiv : Reverse division (divide second input by first)

Numeric

- abs : Negative values will be converted to positive counterparts.
- ceil : Round the value up to the next higher integer
- floor, trunc : Round the value down to the next lower integer (toward negative infinity)
- fract : Return only the fractional component
- sign : Positive input returns 1, negative input returns -1, zero returns itself.
- trunc : Round the value down to the next smaller integer (toward zero)

Powers

- exp : Raise the mathematical value e to a power
- exp2 : Raise 2 to a power
- fastexp : Approximated e to a power

- fastpow : Approximated in1 to the power of in2
- ln, log : The natural logarithm
- log10 : The logarithm base 10 of the input
- log2 : The logarithm base 2 of the input
- pow : Raise in1 to the power of in2
- sqrt : The square root of the input

Range

- clamp, clip : Clamps the input value between specified min and max. Ranges are inclusive (both min and max values may be output)
- fold : Low and high values can be specified by arguments or by inlets. The default range is 0..1.
- scale : Similar to the Max scale and MSP scale~ objects. Inputs are: 1) value to scale, 2) input lower bound, 3), input upper bound, 4) output lower bound, 5) output upper bound, 6) exponential curve. Default lower and upper bounds are zero and one; default exponential curve is 1 (linear). No bound clamping is performed. The high and low values can be reversed for inverted mapping.
- wrap : Low and high values can be specified by arguments or by inlets. The default range is 0..1.

Route

- ?, switch : Selects between the second and third inputs according to the boolean value of the first. If the first argument is true, the second argument will be output. Otherwise, the third argument will be output.
- gate : Similar to the MSP gate~ object. It takes an argument for number of outputs (one is the default) and lets you choose which the incoming signal (at the right inlet) is sent to according to the (integer) value in the left inlet. A value of zero or less to the left inlet will choose no output; a value greater than the number of outlets will select the last outlet. Like gate~, un-selected outlets will send zero.
- mix : Mixes (interpolates) between inputs a and b according to the value of the third input t, using linear interpolation. The factor (t) should vary between 0 (for a) and 1 (for b). If one argument is given, it specifies the mix (interpolation) factor.

- r, receive : Receive values from a named send. The send/receive pairs are only visible to each other within the same gen patcher. They will not send across gen patchers or sub-patchers.
- s, send : Send values to a named receive. The send/receive pairs are only visible to each other within the same gen patcher. They will not send across gen patchers or sub-patchers.
- selector : Similar to the MSP selector~ object. In a Gen patcher it takes an argument for number of choices (one is the default). In GenExpr, the number of choices is determined by the number of arguments. The first input lets you choose which of the remaining inputs is sent to the output. A value of zero or less to the first input will result in a zero signal at the output; a value greater than the number of choices will select the last input.
- smoothstep : Smoothstep is a scalar interpolation function commonly used in computer graphics. The function interpolates smoothly between two input values based on a third one that should be between the first two. The returned value is clamped between 0 and 1. The slope (i.e. derivative) of the smoothstep function starts at 0 and ends at 0.

Subpatcher

- gen : Gen subpatcher or abstraction
- setparam : Set a param in a subpatcher from a parent patcher

Trigonometry

- acos : The arc cosine of the input (returns radians)
- acosh : The inverse hyperbolic cosine of the input
- asin : The arc sine of the input (returns radians)
- asinh : The inverse hyperbolic sine of the input
- atan : The arc tangent of the input (returns radians)
- atan2 : Returns the angle to the coordinate (x,y) in radians.
- atanh : The inverse hyperbolic tangent of the input
- cos : The cosine of the input (in radians)
- cosh : The hyperbolic cosine of the input
- degrees : convert radians to degrees

- fastcos : The approximated cosine of the input (in radians)
- fastsin : The approximated sine of the input (in radians)
- fasttan : The approximated tangent of the input (in radians)
- hypot : Returns the length of the vector to (in1, in2).
- radians : convert degrees to radians
- sin : The sine of the input (in radians)
- sinh : The hyperbolic sine of the input
- tan : The tangent of the input (in radians)
- tanh : The hyperbolic tangent of the input

Waveform

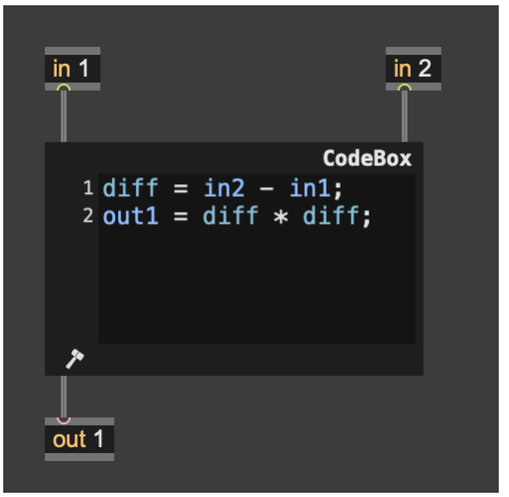
- noise : A random number generator

GenExpr

The Gen Patcher and the codebox Object 221
Language Basics 221
Parameters 223
Comments 224
Multiple Return Values 224
Defining GenExpr Functions 228
Operator Attributes 230
Abstractions as GenExpr Functions 232
Requiring GenExpr Files 232
Branching and Looping 233
GenExpr and Jitter Inputs 236
GenExpr and Jitter Coordinate Operations 237
Technical Notes 238

GenExpr is the internal language used by Gen patchers. It is used to describe computations in an implementation agnostic manner. To perform actual computations, it is translated into machine code for the CPU or GPU by the various Gen objects (gen, gen~, jit.gen, etc.). The GenExpr language can be used directly in Gen patchers with the expr and codebox objects. These objects analyze the expressions written in them and automatically construct the the appropriate number of inlets and outlets so that patchcords can be connected to the computations described within.

Note that there is absolutely no difference in terms of performance between describing computations with object boxes and the GenExpr language. When a Gen patcher is compiled, it all gets merged into a single representation, so use the approach that is most convenient for the problem.

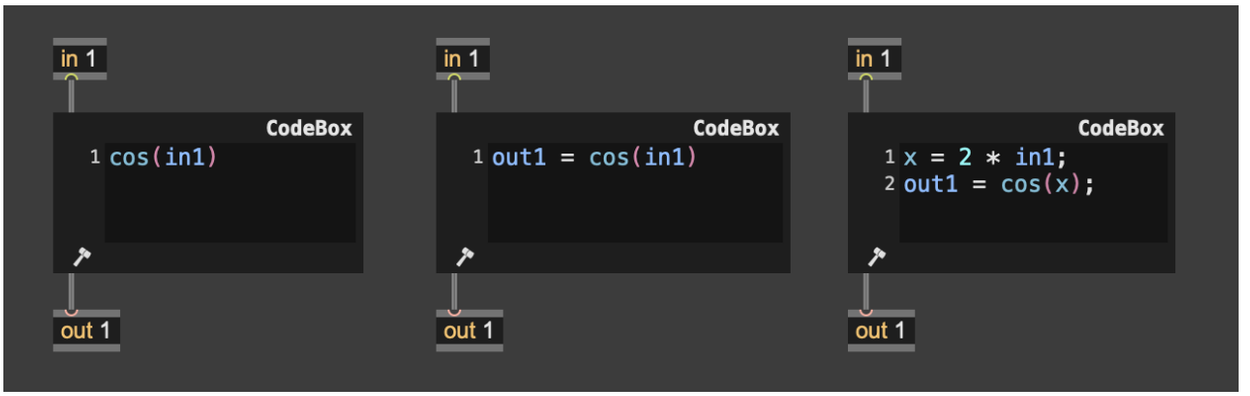


The Gen Patcher and the codebox Object

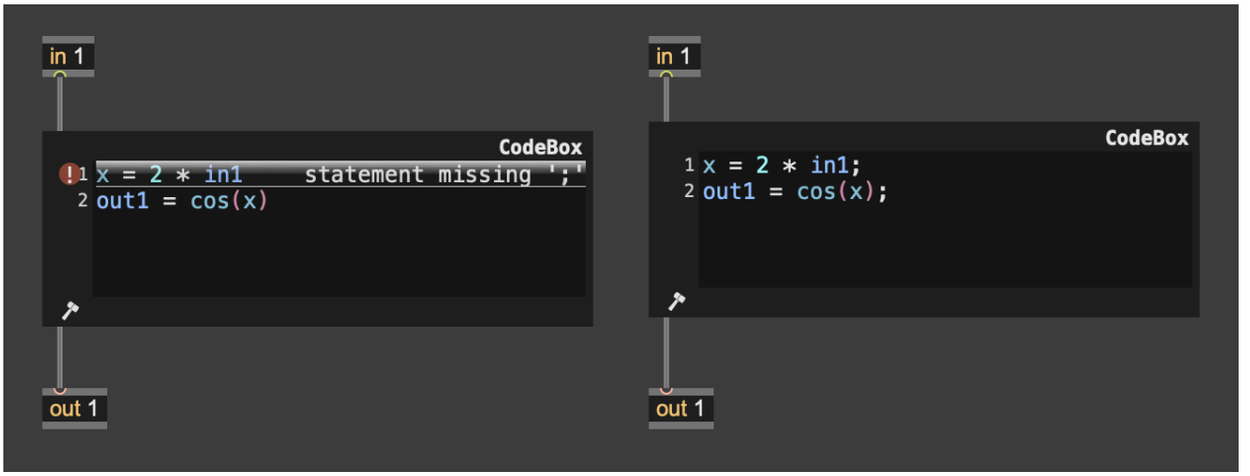
The GenExpr language is designed to complement the Max patching environment within Gen patchers. It provides a parallel textual mechanism for describing computations to be used in concert with the graphical patching paradigm of Max. As one example, the structural elements of user-defined GenExpr functions correspond closely to the structural elements of Max objects with their notions of inlets, outlets, arguments and attributes. Furthermore, the GenExpr language has keywords `in`, `in1`, `in2`, ... and `out`, `out1`, `out2`, ... that specifically refer to the inlets and outlets of the expr or codebox the GenExpr code is embedded in.

Language Basics

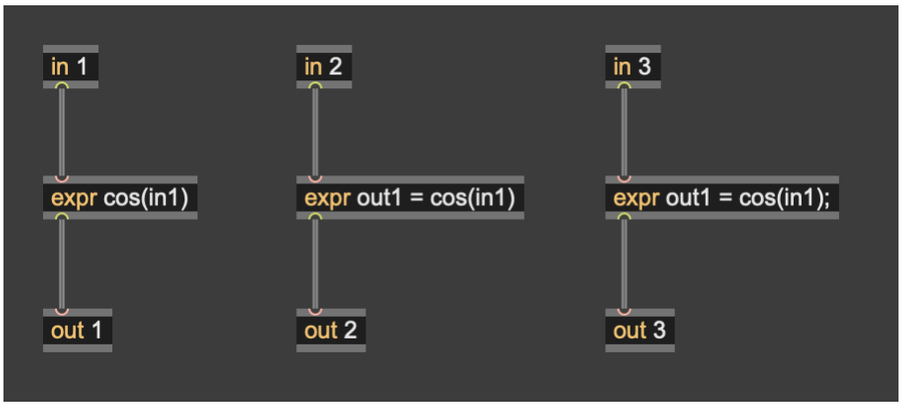
The GenExpr language resembles C and JavaScript for simple expression statements; however, semicolons are only necessary when there are multiple statements. The codeboxes below all contain valid expressions in GenExpr. When there is a single expression with no assignment like in the far left codebox, the assignment to out1 is implied. Notice that it also doesn't have a semicolon at the end. When there is only one statement, the semicolon is also implied.



For multi-line statements however, semicolons are required. The codebox on the left doesn't have them and will generate errors. The codebox on the right is correct.



The expr operator is functionally the same as a codebox but lacks the text editor features such as syntax highlighting and multi-line text display and navigation. expr is most useful for short, one-line expressions, saving the effort of patching together a sequence of objects together that operate as a unit.



An expr or codebox determines its number of inlets and outlets by detecting the `inN` and `outN` keywords where `N` is the inlet/outlet position. `in1` and `out1` are the left-most inlet and outlet respectively. For convenience, the keywords `in` and `out` are equivalent to `in1` and `out1` respectively.

Almost every object that can be created in a Gen patcher is also available from within GenExpr as either a function, a global variable, a declaration, or a constant. The number of inlets an object has corresponds to the number of arguments a function takes. For example, the object atan2 has two inlets and takes two arguments as follows:

```
out  = atan2( in1 ,  in2 )
```

Parameters

Parameters declared in GenExpr behave just like param operators in a patch. They can only be declared in the main body of GenExpr code where inlets and outlets (*inN* and *outN*) exist because they operate at the same level as object box Gen operators in the patcher.



A param declaration in GenExpr names a parameter and creates it if necessary. If there is a param object box with the same name as a param declared in GenExpr, the GenExpr param will simply alias the object box param. If there isn't a param object box with the same name, one will be implicitly created. In the code above, offset aliases the object box param offset, while amp creates a new global param.

Comments

Comments in GenExpr follow the C style syntax for both single-line and multi-line comments. Single-line comments start withand multi-line comments are defined by.

```
// this is a comment, below we sample an input
pix = sample( in1 , norm);
```

Multiple Return Values

Just as object boxes can have multiple inlets and outlets, function in GenExpr can take multiple arguments and can return multiple values. The object cartopol has two inlets and two outlets. Similarly, in GenExpr the cartopol function takes two arguments and returns two values. In code, this looks like r, theta = cartopol(x, y). Functions that return mutiple values can assign to a list of variables. The syntax follows the pattern:

```
var1, var2, var3, … = <expression>
```

When a function returns multiple values but assigns to only one value, the unused return values are simply ignored. When a return value is ignored, the GenExpr compiler eliminates any unnecessary calculations. The function cartopol could be expanded out to

```
r, theta = sqrt(x*x+y*y), atan2(y, x)
```

If we remove theta and have instead

```
r = sqrt(x*x+y*y), atan2(y, x)
```

the compiler simplifies it to

```
r = sqrt(x*x+y*y)
```

In the reverse case where we only use theta, the Gen compiler will strip out the calculations for r

```
notused, theta = sqrt(x*x+y*y), atan2(y, x);
  out  = theta;
```

effectively becomes

```
theta = atan2(y, x);
  out  = theta;
```

Even for more complex examples where the outputs share intermediate calculations, the compiler eliminates unnecessary operations, so there is no performance penalty for not using all of a function's return values.

Just as the left-hand side list of variable names being assigned to are separated by commas, the right-hand side list of expressions can also be separated by commas:

```
sum, diff = in1 + in2, in1 - in2
out1, out2  = diff, sum
```

If there are more values on the left-hand side than on the right-hand side, the extra variable names are given a value of zero. For example,

```
out1, out2  =  in1
```

becomes

```
out1, out2  =  in1, 0
```

If any of the expressions in the right-hand side return more than one value, these additional values will be ignored unless the expression is the last item in the right-hand side list. This is complex to describe, but should be clear from these examples:

Unused Return Values The second return value gets discarded and cartopol is optimized:

```
r = cartopol(x, y)
```

Extra Assignment Values Zeros are assigned to extra assignment values:

```
x, y =  in1
```

becomes

```
x, y =  in1 ,  0
```

Multiple Return Values in an Expression List Only the last expression can return multiple values. cartopol's second return value discarded, as it is not the last expression in the right-hand side:

```
r,  out1  = cartopol(x, y),  in1
```

Here cartopol returns both values, since it is in the last position:

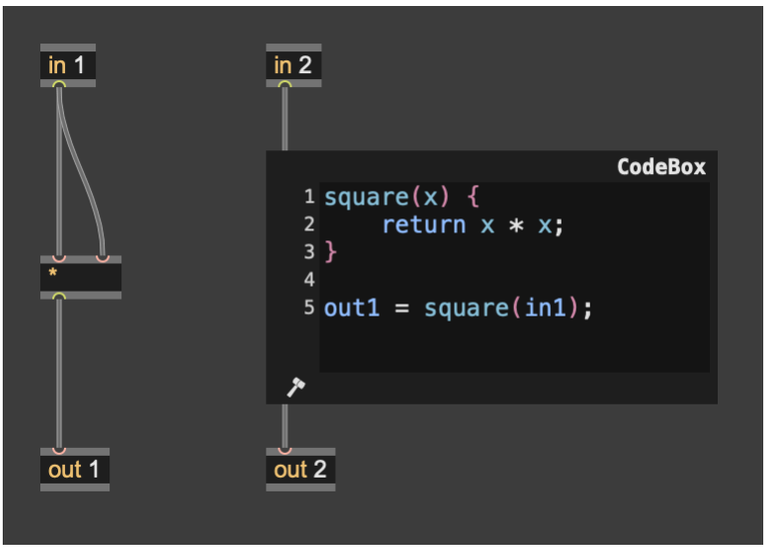
```
out1 , r, theta =  in1 , cartopol(x, y)
```

The same principle applies when expressions are used as arguments to a function call. In this example, the two output values of poltocar connect to the two input values of min:

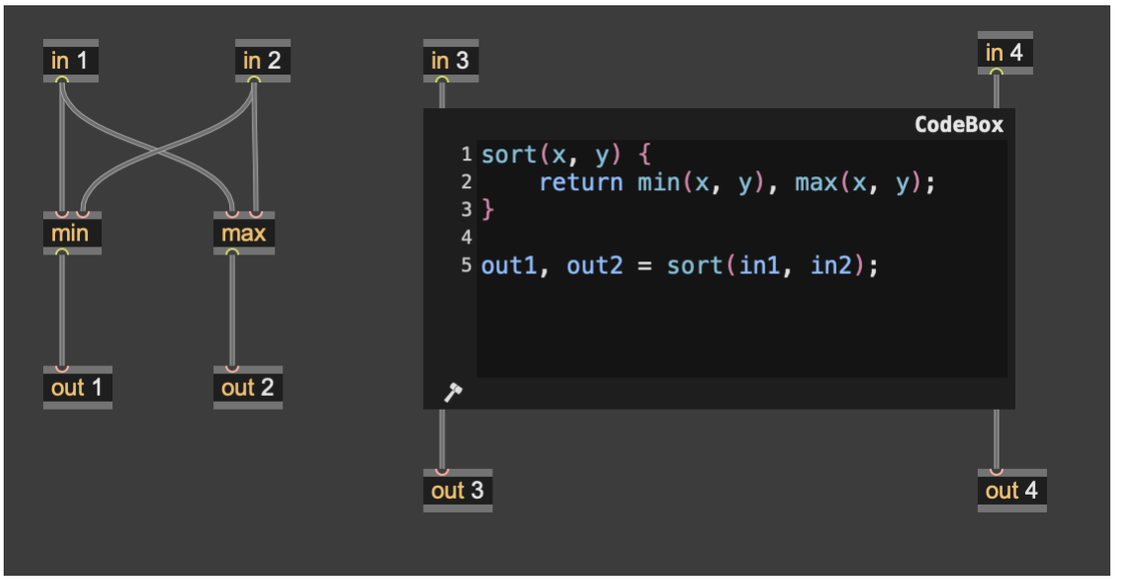
```
out  = min(poltocar( in1 ,  in2 ))
```

Defining GenExpr Functions

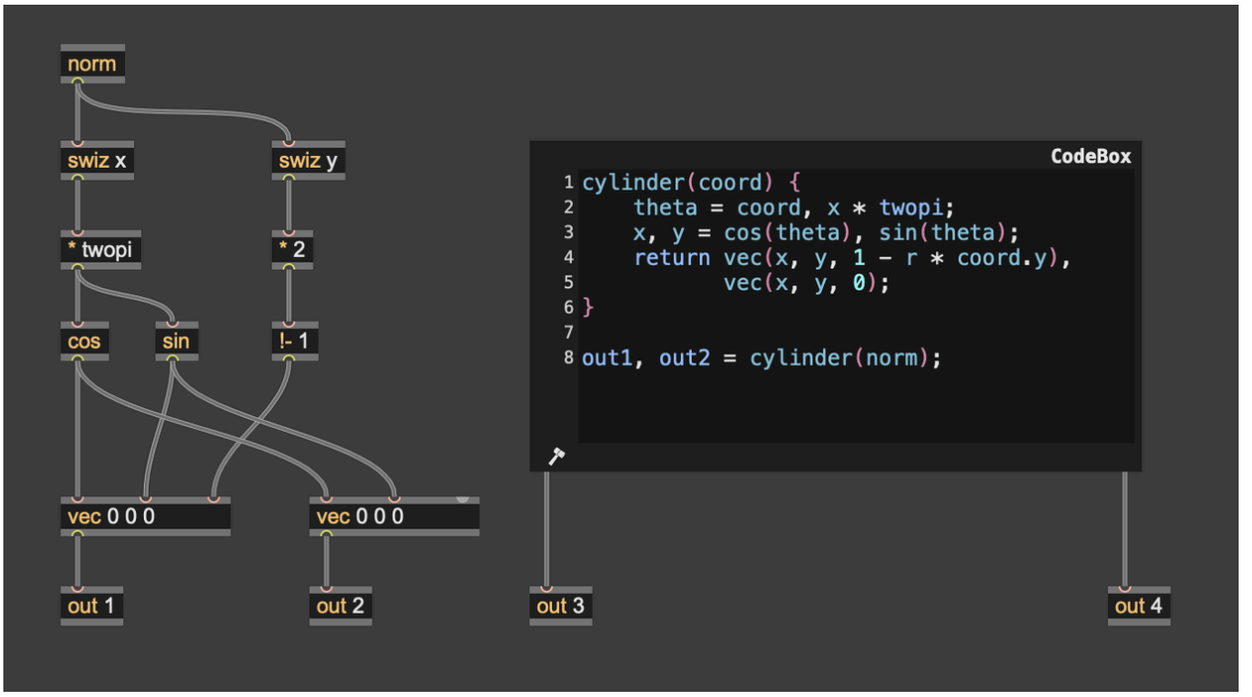
Defining new functions in GenExpr happens in much the same way as other familiar programming languages. Since there are no types in GenExpr function arguments are specified simply with a name. A basic function definition with an equivalent patcher representation looks like the following. Note that functions must be declared **before** all statements:



A function returning multiple values looks like:

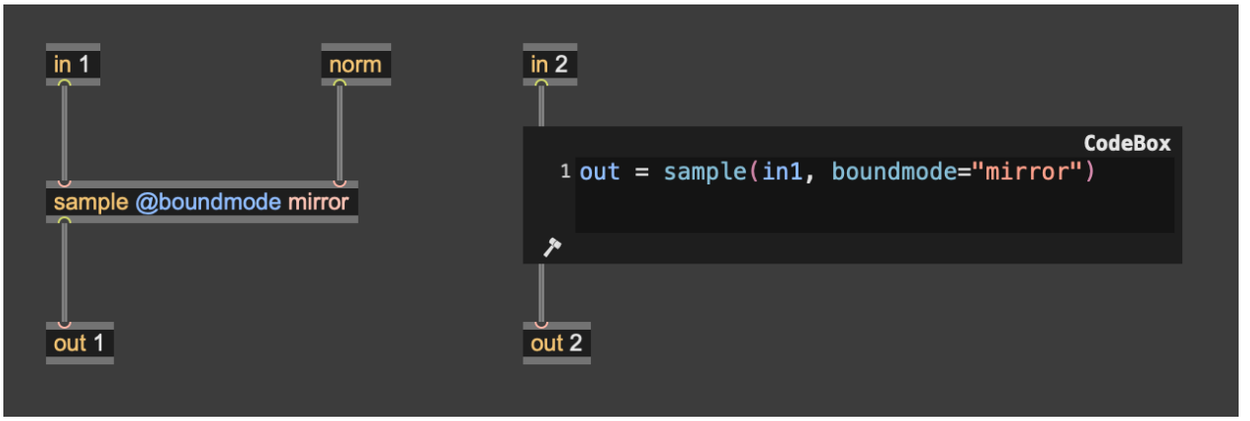


The cylinder operator in Jitter Gen objects is defined as:



While simple functions in GenExpr can be easily patched together, more involved functions like the above cylinder definition start to become unwieldy, especially if the function is used several times within the GenExpr code. This is the advantage of textual representations.

Operator Attributes



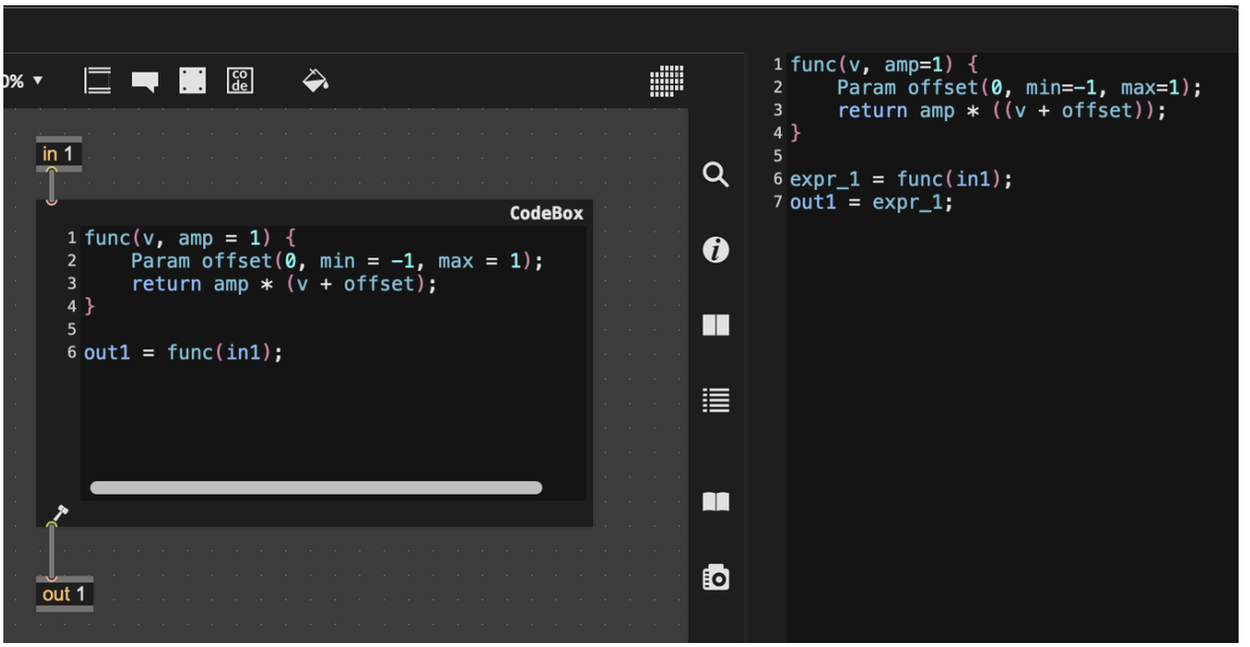
In Gen patchers, some objects have attributes like the Jitter Gen operator sample, which as a `boundmode` attribute. In GenExpr, function arguments correspond to operator inlets and function return values correspond to outlets. Attributes are set using a key/value style argument. For example:

```
out  = sample( in , norm, boundmode= "mirror" );
```

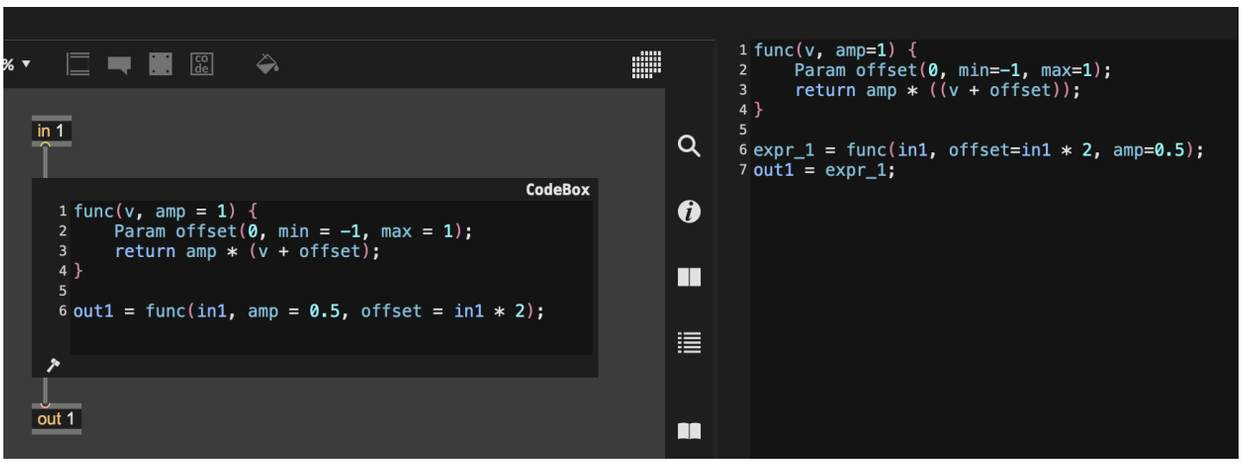
will use a version of the sample operator with a mirroring boundary behavior. The attribute is set with boundmode as the *key* and "mirror" as its *value*. Since the concept of Max messages doesn't exist within Gen patchers, attributes for built-in operators are not dynamically modifiable. This holds equally in GenExpr. Such attribute values must be constants. If the attribute takes a numerical value, it cannot be assigned to from a variable.

Attributes in Function Definitions

Attributes can also be defined for function definitions. Here, attributes can be dynamic, behaving in a similar manner to how setparam interacts with subpatcher parameters. Attributes can be defined in one of two ways. With one syntax, the attribute is defined and given a default in the function signature. With the other, a Param object is declared in the function, adding the name of the parameter as an attribute to the function.



With the first method, only the default value of the attribute can be defined. With the second method, other properties such as minimum and maximum values for the attribute can be set. By declaring a param object, you get more control over how the attribute operates.



As with built-in operators, attributes of function definitions can be set with key-value syntax. In the above example, the `amp` attribute is given a value of 0.5 while the `offset` attribute is given a value of `in1 * 2`, which is an expression that is not constant but valid because `func` is a function definition. Note, however, that the `offset` attribute has minimum and maximum values defined, so any expression assigned to it will be clamped to that range.

Abstractions as GenExpr Functions

Structurally, GenExpr functions are equivalent to Gen patchers. Both can have inputs, outputs and named parameters. In GenExpr, Gen patchers save as abstractions (.gendsp or .genjit files) can be used as functions. When the GenExpr interpreter encounters a function it can't find the definition of, it will use the current Max search paths to look for a Gen abstraction with the name of the function.

```
out1  = myAbstraction( in1 )
```

There is no definition of the function myAbstraction in the above code and it isn't a built-in operator, so Max will try to find a Gen abstraction with that name. The GenExpr interpreter will look for `myAbstraction.gendsp` in the case of gen~ or `myAbstraction.genjit` in the case of the Jitter Gen objects jit.gen, jit.pix, or jit.gl.pix.

There are some caveats with using abstractions as GenExpr functions. GenExpr function names must be valid identifiers. An identifier in GenExpr is a sequence of characters starting with a letter or an underscore (`[a-z]` , `[A-Z]` , `_`) followed by any number of letters, numbers or underscores (`[a-z]` , `[A-Z]` , `[0-9]` , `_`). It is not uncommon for Max subpatchers to have other chartacters such as `~` or `.` in them. These are invalid characters when it comes to GenExpr function names, so if they're used in the name of a Gen abstraction, they cannot be used as GenExpr functions.

Requiring GenExpr Files

When defining operators in GenExpr, it can be handy to keep them in a separate file so that they can be reused frequently without having to constantly re-type the operator definition in a codebox. To include GenExpr operators defined in a separate file, use the `require` operator. The `require` operator takes the name of a `.genexpr` file and loads its definitions. The following are all valid ways to require a `.genexpr` file:

```
require( "mylib" )
require( "mylib" );
require "mylib"
require "mylib" ;
```

In the above code, calling `require` triggers the codebox to look for the file 'mylib.genexpr' using the Max search path. Required `.genexpr` files can also require other files. If a file is required more than once, it will only be loaded once.

GenExpr files can be created and edited using the built-in Max text editor. If a GenExpr file is required in a gen object and the file is edited and saved, the Gen object will detect that a file it depends on has changed through filewatching and recompile itself to reflect the new changes.

Branching and Looping

Branching and looping is supported in GenExpr with `if/else`, `for` and `while` constructs. `if` statements can be chained together using `else if` an arbitrary number of times such that different blocks of code will be executed depending on different conditions. For example:

```
if (in  >  0.5) {
  out = cos(in * pi);
}
else if (in  >  0.25) {
  out  = sin( in *pi);
}
else {
  out  = cos(in * pi) * sin(in * pi);
}
```

Note that in the Jitter gen objects, `if` statements with vector-valued conditions will only use the first element of the vector to determine whether a block of code should be tested or not. `while` loops in GenExpr are similar to those in many other languages:

```
i = 0;
val = 0;
while (i < 10) {
  i += 1;
  // accumulate
  val += i;
}
out  = val;
```

`for` loops are also similar to this in many other languages although there is no `++` operator in GenExpr to increment a loop counter. Instead, += can be used:

```
val = 0 ;
  for (i= 0; i <  10; i += 1) {
      // accumulate
    val += i;
}
out  = val;
```

Since GenExpr is compiled on-the-fly, it can be easy to make a programming mistake and create an infinite loop. All of the gen objects have protections against infinite loops, so while an infinite loop might slow things down, it won't cause Max to get stuck and freeze.

Also, note that in many cases values in GenExpr are floating point, even loop counters. Floating point numbers can't exactly represent every number, sometimes a little fudge factor to account for this might be necessary. For example, this loop:

```
val = 0;
for (i = 0; i <= 1; i += 0.05) {
  // accumulate
  val += i;
}
out = val;
```

will likely not reach 1.0 despite the `<=` operator because of floating point precision. Instead, write something like:

```
val = 0;
for (i = 0 ; i <= 1.04 ; i +=  0.05) {
  // accumulate
  val += i;
}
out  = val;
```

to ensure that the `i` variable goes all the way to 1.

Continue and Break

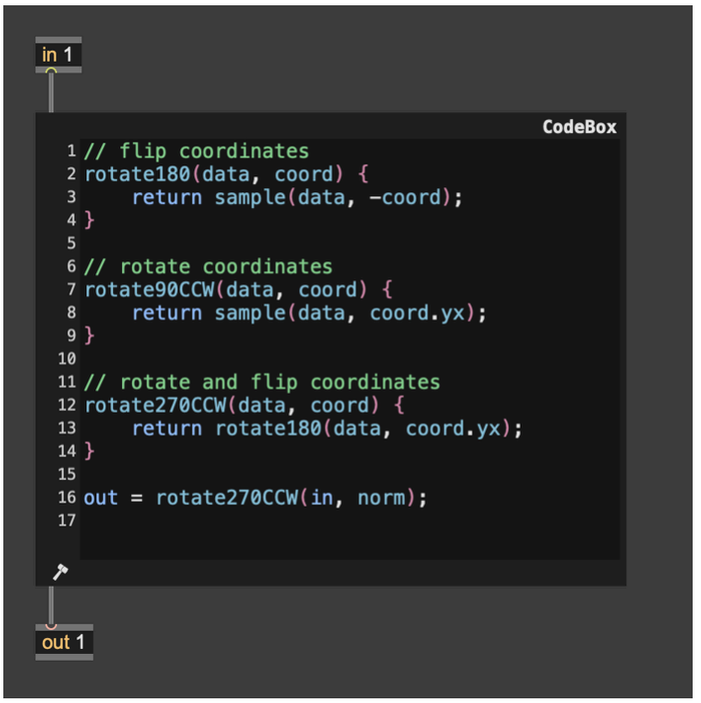
With looping constructs, GenExpr supports `break` and `continue` statements. `break` causes an early exit from a loop while `continue` causes the loop to start the next iteration without finishing the current one.

```
val = 0;
for (i = 0 ; i < 10; i += 1) {
  if (val > 20) {
    // bail early
    break;
  }
  val += i;
}
out = val;
```

```
val = 0 ;
for (i = 0 ;i <  10; i +=  1) {
  if (val ==  6) {
    // skip an iteration
    continue;
  }
  val += i;
}
out  = val;
```

GenExpr and Jitter Inputs

Jitter Gen objects take both Jitter matrices (jit.matrix) and/or textures (jit.gl.texture) depending on the object. Within the Gen patcher the operator `in` represents both the input matrix or texture in its entirety *and* the current cell of that input being processed. In most cases, the `in` operator represents the current cell being processed. The only time where it represents the entire input is with the sample and nearest operators. Only an `in` operator can be connected to the left input of sample and nearest, which are used to grab data from arbitrary locations within the input. The same holds true when sample and nearest are used in GenExpr.



When GenExpr code is compiled, the inputs to sample and nearest are validated to ensure that their first arguments are actually Gen patcher inputs and an error thrown if this isn't the case. The validation process can track inputs even through function calls so sample and nearest can be used within functions without issue.

GenExpr and Jitter Coordinate Operations

The coordinate operations in Jitter Gen patchers (norm, snorm, cell, and dim) are special-case operators that are a hybrid betwen operator and global variable. There are two equally valid syntaxes for using these operators:

```
out1  = norm * dim();
```

In the first instance above, norm is syntactically a global variable while dim is syntactically a function call. All of the coordinate operators follow this convention.

Technical Notes

GenExpr is a type-less language. Variables are given types automatically by the compiler depending on the Gen domain and the Gen object's inputs. Gen variables are also local-to-scope by default so they don't have to be declared with a keyword like var as in JavaScript. Note that GenExpr has no array notation `[index]` as there is currently no notion of an array structure.

gen~ Operators

Buffer 239
Convert 241
Constants 241
DSP 241
Feedback 242
FFT 243
Filter 243
Global 244
Integrator 244
Numeric 245
Waveform 245
See Also 245

The following Gen operators are unique to the gen~ object, and operate in the audio domain.

Buffer

- buffer : References a named buffer~ object in the gen~ object's parent patch. The first argument specifies a name by which to refer to this data in other objects in the gen patcher (such as peek and poke); the second optional argument specifies the name of the external buffer~ object to reference (if ommitted, the first argument name is used). The first outlet sends the length of the buffer in samples; the second outlet sends the number of channels.
- channels : The number of channels of a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher.
- cycle : An interpolating oscillator that reads repeatedly through one cycle of a sine wave. By default it is driven by a frequency input, but if the `@index` attribute is set to 'phase', it can be driven by a phase input instead.
- data : Stores an array of sample data (64-bit floats) usable for sampling, wavetable synthesis and other purposes. The first argument specifies a name by which to refer to this data in other objects in the gen patcher (such as peek and poke); the second optional argument specifies the length of the array (default 512 samples); and the third optional argument

specifies the number of channels (default 1). The first outlet sends the length of the buffer in samples; the second outlet sends the number of channels.

- dim : The length (in samples) of a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher.
- lookup : Index a data/buffer object using a signal, for waveshaping. The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels. Input signals in the range -1 to 1 are mapped to the full size of the data/buffer, with linear interpolation. The last inlet specifies a channel offset (default 0).
- nearest : Multi-channel lookup a data/buffer object (no interpolation). The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels. The input phase ranges from 0 to 1, and wraps outside this range. The last inlet specifies a channel offset (default 0).
- peek : Read values from a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels. The first inlet specifes a sample index to read (no interpolation); indices out of range return zero. The last inlet specifies a channel offset (default 0).
- poke : Write values into a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument (or third inlet if omitted) specifies which channel to use. The first inlet specifies a value to write, while the second inlet specifies the sample index within the data/buffer. If the index is out of range, no value is written.
- sample : Linear interpolated multi-channel lookup of a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels. The last inlet specifies a channel offset (default 0).
- splat : Mix values into a data/buffer object, with linear interpolated overdubbing. The first argument should be a name of a data or buffer object in the gen patcher. The second argument (or third inlet if omitted) specifies which channel to use. The first inlet specifies a value to write, while the fractional component of the second inlet specifies a phase (0..1) within the data/buffer (indices out of range will wrap). Splat writes with linear interpolation between samples, and mixes new values with the existing data (overdubbing).
- wave : Wavetable synthesis using a data/buffer object. The first argument should be a name of a data or buffer object in the gen patcher. The second argument specifies the number of output channels. The first inlet specifies phase (0..1), while the second and third inlets specify start/end sample positions within the data/buffer. The last inlet specifies a channel offset (default 0).

Convert

- atodb : Convert linear amplitude to deciBel value
- dbtoa : Convert deciBel value to linear amplitude
- ftom : Frequency given in Hertz is converted to MIDI note number (0-127). Fractional note numbers are supported. The second input sets the tuning base (default 440).
- mstosamps : Convert period in milliseconds to samples
- mtof : MIDI note number (0-127) is converted to frequency in Hertz. Fractional note numbers are supported. The second input sets the tuning base (default 440).
- sampstoms : Convert period in samples to milliseconds

Constants

- fftfullspect, FFTFULLSPECT : The pfft~ full spectrum flag (0/1)
- ffthop, FFTHOP : The pfft~ FFT hop size
- fftoffset, FFTOFFSET : The pfft~ FFT offset
- fftsize, FFTSIZE : The pfft~ FFT frame size
- samplerate, SAMPLERATE : The DSP samplerate. Will return $\frac{1000}{@interval}$@interval1000 for event-rate gen.
- vectorsize, VECTORSIZE : The DSP vectorsize. Will return 1 for event-rate gen.

DSP

- fixdenorm : This operator detects denormal numbers and replaces them with zero. Note: As of Max 6.0 the x87 control flags are set to flush to zero and disable exception handling in audio processing, so denormal fixing should only be required for exported code. A denormal number is a floating point value very close to zero (filling the underflow gap). Calculations with denormal values can be up to 100 times more expensive, so it is often beneficial to replace them with zeroes. Denormals often occur in feedback loops with multipliers, such as filters, delays and exponential decays. Denormal detection is based on a bitmask. Note that feedback operators in gen~ (delay, history) apply fixdenorm to their input signals by default.

- fixnan : This operator replaces NaNs with zero. A NaN (Not a Number) is a floating point data value which represents an undefined or unrepresentable value, such as the result of dividing by zero. Computations on NaNs produce more NaNs, and so it is often preferable to replace the NaN with a zero value. Note that division and modulo operators in gen~ protect against generating NaNs by default.
- isdenorm : This operator detects denormal numbers and returns 1 if the input is denormal, and zero otherwise. Note: As of Max 6.0 the x87 control flags are set to flush to zero and disable exception handling in audio processing, so denormal fixing should only be required for exported code. A denormal number is a floating point value very close to zero (filling the underflow gap). Calculations with denormal values can be up to 100 times more expensive, so it is often beneficial to replace them with zeroes. Denormals often occur in feedback loops with multipliers, such as filters, delays and exponential decays. Denormal detection is based on a bitmask. Note that feedback operators in gen~ (delay, history) apply fixdenorm to their input signals by default.
- isnan : This operator detects the presence of NaN values, returning 1 if the input is NaN, and zero otherwise. A NaN (Not a Number) is a floating point data value which represents an undefined or unrepresentable value, such as the result of dividing by zero. Computations on NaNs produce more NaNs, and so it is often preferable to replace the NaN with a zero value. Note that division and modulo operators in gen~ protect against generating NaNs by default.
- t60 : Given an input T, returns a number X such that, after T multiplications of a signal by X, that signal would be attenuated by 60 decibels. Roughly, -60db = 0db * pow(X, T). This could be used as a per-sample multiplier (X) to ensure a decay time (of T samples), for example. The name t60 is borrowed from the RT60 time used to measure reverberation time, which specifies the time taken for a signal to decay by 60db, as an approximation of fading to inaudibility.
- t60time : Estimates the decay time (in samples) of a given decay factor. That is, given a multiplier X, returns a number T such that, after T multiplications of a signal by X, that signal would be attenuated by 60 decibels. It is the dual of the t60 object.

Feedback

- delay : Delays a signal by a certain amount of time (specified in samples). The first argument specifies the maximum delay time (in samples, default samplerate). The second argument specifies the number of tap inlet/outlet pairs (default 1). The first inlet is the signal to be delayed. Additional inlets specify the delay time per tap (in samples). With `@feedback` 1, like history, delay allows feedback loops in the patcher, but minium delay is 1 sample. With

`@feedback` 0, minimum delay time is zero samples (or more if `@interp` is cubic, spline, or spline6)

- history : The history operator allows feedback in the gen patcher through the insertion of a single-sample delay. The first argument is an optional name for the history operator, which allows it to also be set externally (in the same way as the param operator). The second argument specifies an initial value of stored history (defaults to zero).

FFT

- fftinfo : fftinfo gets constant data about the FFT frames in a patcher loaded by pfft~. If it is used in a patcher that is not loaded by pfft~, it returns default constants instead.

Filter

- change : Returns the sign of the difference between the current and previous input: 1 if the input is increasing, -1 if decreasing, and 0 if unchanging.
- dcblock : A one-pole high-pass filter to remove DC components. Equivalent to the GenExpr: `History x1, y1; y = in1 - x1 + y1*0.9997; x1 = in1; y1 = y; out1 = y;`
- delta : Returns the difference between the current and previous input.
- interp : Smoothly mix between inputs, according to an interpolation factor in the range of 0 to 1 (first inlet). The @mode attribute can choose between linear or cosine interpolation to mix between two additional inlets, cubic or spline to mix between four inlets, and spline6 to mix between six inlets. The default mode is linear.
- latch : Conditionally passes or holds input. The first inlet is the 'input' and the second inlet is the 'control'. When the control is non-zero, the input value is passed through. When the control is zero, the previous input value is output. It can be used to periodically sample & hold a source signal with a simpler trigger logic than the sah operator.
- phasewrap : Wrap input to the range -pi to +pi
- sah : The first inlet is the 'input' and the second inlet is the 'control'. When the control makes a transition from being at or below the trigger value to being above the trigger threshold, the input is sampled. The sampled value is output until another control transition occurs, at which point the input is sampled again. The default threshold value is 0, but can be specified as the last inlet/argument. The `@init` attribute sets the initial previous value to compare to (default 0).

- slide : Use the slide operator for envelope following and lowpass filtering. Related to the MSP slide~ object.

Global

- elapsed : The number of samples elapsed since the patcher DSP began, or since the last reset.
- mc_channel : If used within a patcher inside mc.gen~, the mc_channel operator will return the current channel index. Otherwise, it always returns 1.
- mc_channelcount : If used within a patcher inside mc.gen~, the mc_channelcount operator will return the channel count of the mc.gen~. Otherwise, it always returns 1.
- voice : If used within a poly~ patcher, the voice operator will return the current voice index (similar to thispoly~). Otherwise, it always returns 1.
- voicecount : If used within a poly~ patcher, the voicecount operator will return the current voice count. Otherwise, it always returns 1.

Integrator

- *=, mulequals : The object multiplies by, and then outputs, an internal value. This occurs at sample-rate, so the stored value can grow very large or very small, very fast. The value to be multiplied by is specified by either the first inlet or argument. The internal sum can be reset to the minimum by sending a nonzero value to the right-most inlet. The minimum value is 0 by default, but can be changed with the @min attribute. An optional maximum value can be specified with the @max attribute; values will wrap at the maximum.
- +=, accum, plusequals : The object adds to, and then outputs, an internal sum. This occurs at sample-rate, so the sum can grow very large, very fast. The value to be added is specified by either the first inlet or argument. The internal sum can be reset to the minimum by sending a nonzero value to the right-most inlet. The minimum value is 0 by default, but can be changed with the @min attribute. An optional maximum value can be specified with the @max attribute; values will wrap at the maximum.
- counter : Accumulates and outputs a stored count, similarly to Max's counter object, but triggered at sample-rate. The amount to accumulate per sample is set by the first input (incr). The count can be reset by a non-zero value in the second input (reset). The third inlet (max) sets a maximum value; the counter will wrap if it reaches this value. However if the maximum value is set to 0 (the default), the counter will assume no limit and count indefinitely. The first outlet outputs the current count, the second outlet outputs 1 when

the count wraps at the maximum and zero otherwise, and the third outlet outputs the number of wraps (the carry count).

Numeric

- round : Returns the integral value that is nearest to the input, with halfway cases rounded away from zero.

Waveform

- phasor : A non-bandlimited sawtooth-waveform signal generator which can be used as LFO audio signal or a sample-accurate timing/control signal.
- rate : The rate operator time-scales an input phase (such as from a phasor) by a multiplier. Multipliers less than 1 create several ramps per phase cycle.
- train : train generates a pulse signal whose period is specifiable in terms of samples. The first input sets the pulse period (in samples). The second input sets the pulse width (default 0.5). The third inlet sets the phase of the 'on' portion (default 0.)
- triangle : A triangle/ramp wavetable with input to change phase offset of the peak value. The phase ranges from 0 to 1 (and wraps outside these values). With a duty cycle of 0, it produces a descending sawtooth; with a duty cycle of 1 it produces ascending sawtooth; with a duty cycle of 0.5 it produces a triangle waveform. Output values always bounded in 0 to 1.

See Also

- mg_gen

Jitter Operators

Color 246
Coordinate 246
Quaternion 246
Sampling 247
Surface 247
Vector 247

The following Gen operators are unique to the jit.gen, jit.pix, and jit.gl.pix objects - unlike the Common Gen Operators , they are only used in the matrix/texture domain.

Color

- hsl2rgb : Convert HSL to RGB, preserving alpha
- rgb2hsl : Convert RGB to HSL, preserving alpha

Coordinate

- cell : Cell coordinates of input matrix [0, dim-1]
- dim : Dimensions of input matrix
- norm : Normalized coordinates of input matrix [0, 1]
- snorm : Signed normalized coordinates of input matrix [-1, 1]

Quaternion

- qconj : Get the conjugate of a quaternion.
- qmul : Multiply quaternion inputs
- qrot : Rotate a vector by a quaternion. The equation of the rotation is $q * v * q^{-1}$
  q * v * q−1.

Sampling

- nearest : Nearest neighbor sample a matrix at a given coordinate (normalized). Nearest has a boundmode attribute that can be set to wrap, mirror or clamp.
- nearestpix : Nearest neighbor sample a matrix at a given coordinate (in pixels). Nearest has a boundmode attribute that can be set to wrap, mirror or clamp.
- sample : Sample a matrix at a given coordinate (normalized) with linear interpolation. Sample has a boundmode attribute that can be set to wrap, mirror or clamp.
- samplepix : Sample a matrix at a given coordinate (in pixels) with linear interpolation. Pixel centers are located at PIXEL+0.5. For example, the center of the upper-left pixel is (0.5, 0.5). Samplepix has a boundmode attribute that can be set to wrap, mirror or clamp.

Surface

- circle : Equation of a circle taking input coordinates ranging from [0, 1]
- cone : Equation of a cone taking input coordinates ranging from [0, 1]
- cylinder : Equation of a cylinder taking input coordinates ranging from [0, 1]
- plane : Equation of a plane taking input coordinates ranging from [0, 1]
- sphere : Equation of a sphere taking input coordinates ranging from [0, 1]
- torus : Equation of a torus taking input coordinates ranging from [0, 1]

Vector

- concat : Concatenate vector values into a larger vector
- cross : Take the cross product of two vectors
- dot : Take the dot product of two vectors
- faceforward : Return a vector pointing in the same direction as another
- length : Returns the length of a vector
- normalize : Normalize a vector to unit length
- reflect : Reflect a vector off a surface defined by a normal

- refract : Refract a vector through a surface defined by a normal and a refraction index
- rotor : Return a quaternion that can rotate the first input into the second.
- swiz : Unpack and remap vector components
- vec : Pack scalar values into a vector

Jitter

Depth Testing and Layering

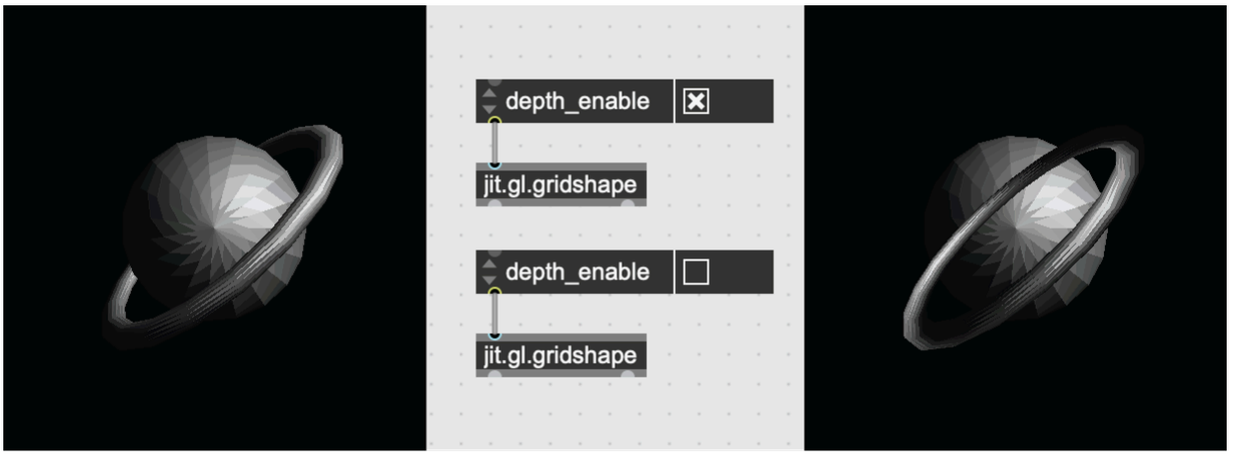
Depth Testing 250
Layering 251
Blending 252
Combining the Techniques 253

Composing refers to the various ways of drawing multiple objects in the same graphics context. One object might partially obscure another, or another object might be visible behind a transparent object. Compositing generally uses one of two techniques: **depth testing** and **layering**. To facilitate these techniques, all Jitter objects representing entities in a rendered scene share the attributes `@layer`, `@depth_enable`, `@blend_enable`, and `@blend`.

Depth Testing

Depth testing models the way objects closer to the camera lens obscure objects further away. Every pixel that would be drawn writes its distance from the camera—its depth—into the **depth buffer**. When a new object is drawn, the renderer checks the depth (distance from the camera) of every pixel and compares that with the depth of whatever pixels may have been previously drawn in the same location. If the new pixel is closer it will be drawn, otherwise it will be discarded.

By default, every context has a depth buffer. Whether or not a context has a depth buffer can be set on jit.window and jit.pwindow with the `@depthbuffer` attribute. Additionally, every `jit.gl.*` object that supports depth testing has a `@depth_enable` attribute. This is on by default, but can be used to control depth testing on a per-object basis.

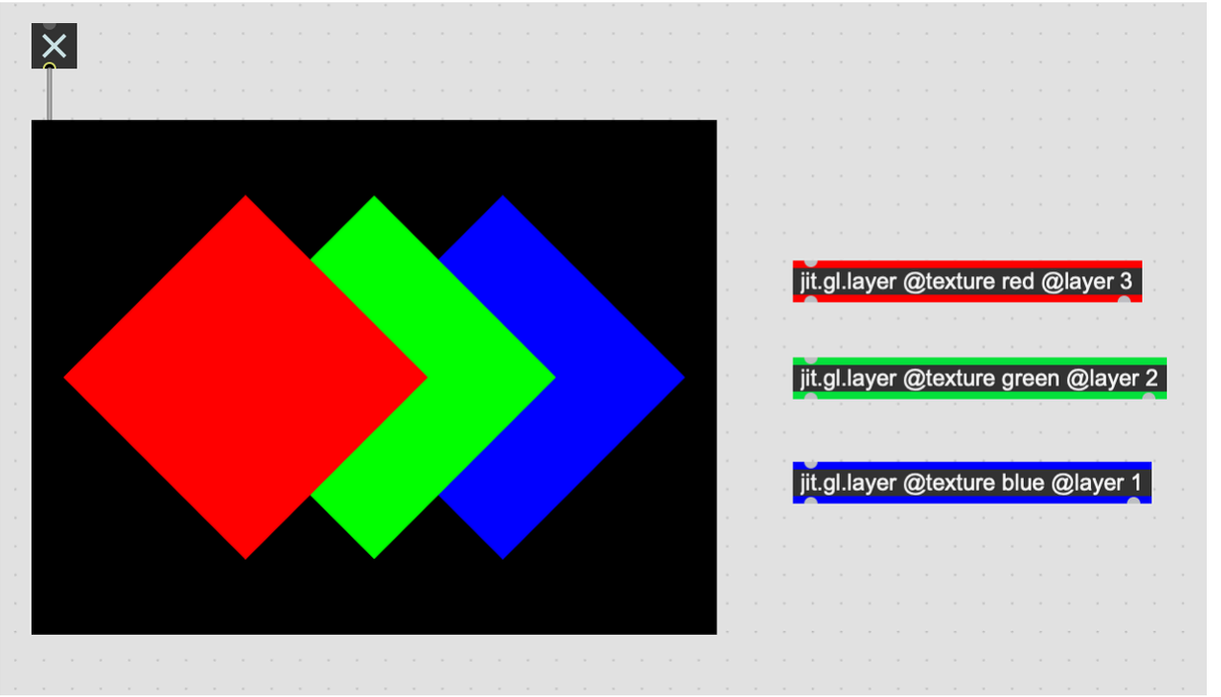


When @depth_enable is disabled, the object will no longer write to the depth buffer, and the renderer will not use depth information when compositing the object into the scene.

Layering

Layering gives you direct control over the order in which objects are drawn. Objects in a lower layer are rendered first, then objects in higher layers. By default, all 3D objects are added to layer 0, and therefore their drawing order is indeterminate. You can change the `@layer` attribute to move the object to a different layer, where higher values are drawn on top of lower values.

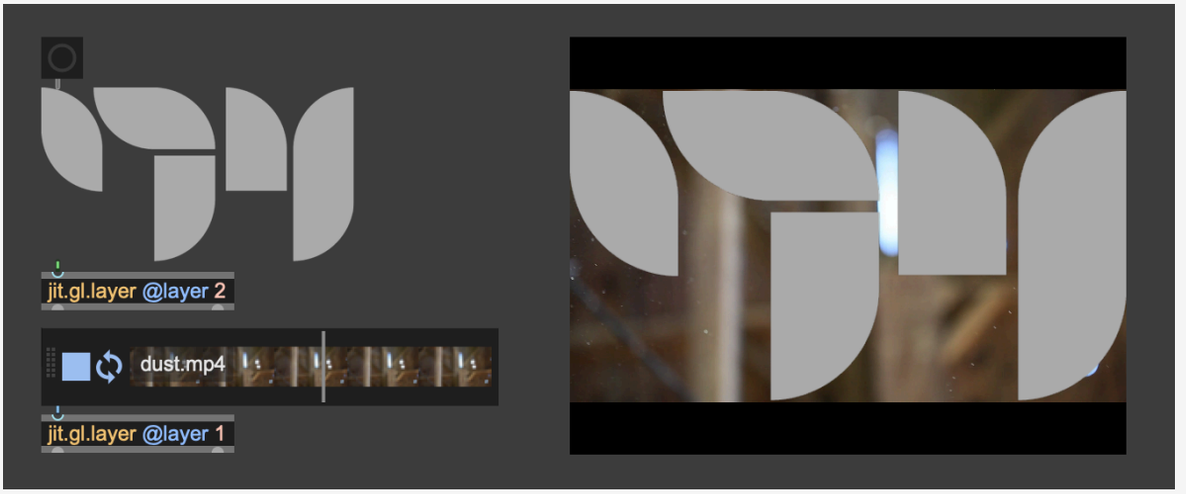
> Be careful to disable depth testing with `@depth_enable 0` if you want to use layering, otherwise depth testing will override the layer value. Unlike other `jit.gl.*` objects the jit.gl.layer object disables depth testing by default.



The red layer (3) is drawn on top of the green layer (2) which is drawn on top of the blue layer (1)

Blending

Enable blending to allow overlapping layers to blend together. Blending is disabled by default, but you can enable it for relevant objects by setting `@blend_enable 1`. You can change the way pixels from different layers mix together by changing the **blend mode**, controlled with the `@blend` attribute. By default, blend is set to **alpha blend**, which uses the alpha value of the higher layer to determine how much of the higher and lower pixels to blend together. An alpha value of 0 means the pixel in the top layer is completely transparent, while an alpha value of 1 means the top pixel is totally opaque. Images and movie files that contain alpha values (e.g. PNG, TIFF, Animation codec, and ProRes 4444) can be displayed with proper transparency by texturing an object (like jit.gl.layer or jit.gl.gridshape) with alpha blend enabled. Unlike other `jit.gl.*` objects the jit.gl.layer object enables blending by default.



An image with an alpha channel, rendered on top of a video, using alpha blending

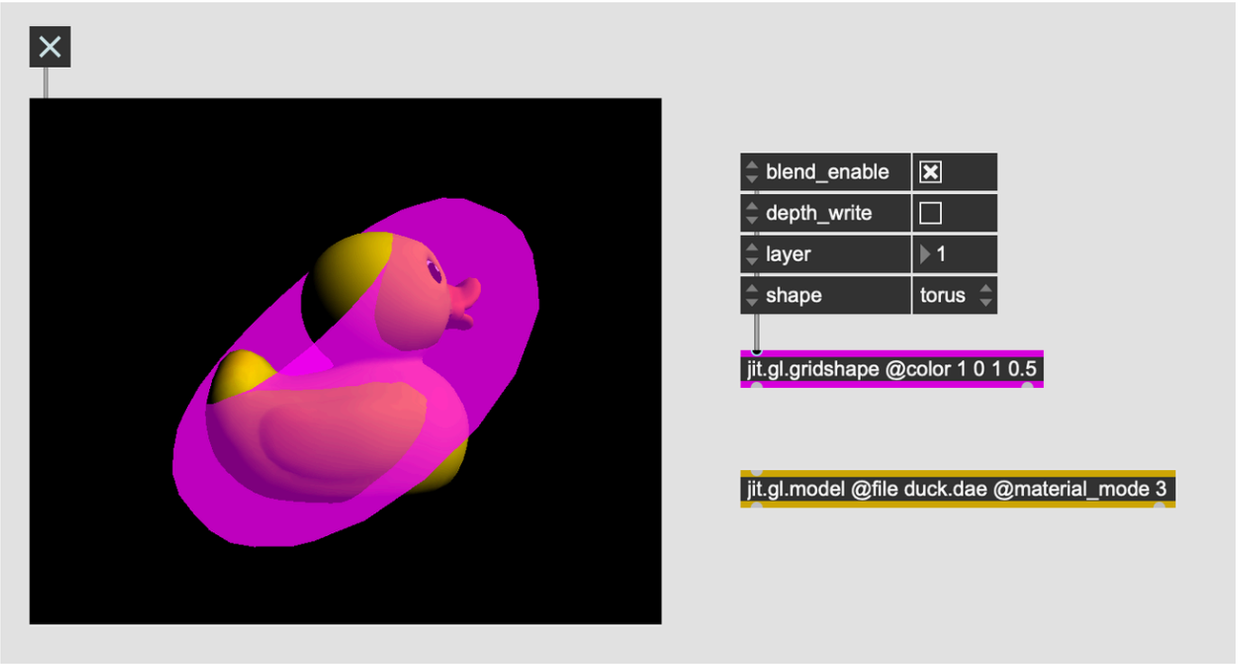
Combining the Techniques

Drawing an overlay

Layering and blending can work together with depth testing to draw a transparent overlay on top of a 3D scene. If objects in the scene have `@depth_enable 1`, while elements in the transparent overlay have `@depth_enable 0`, `@blend_enable 1`, and `@layer 1`, then the overlay will always be drawn on top of elements in the 3D scene. Whether or not the overlay is transparent or opaque will be determined by the alpha values of pixels in the overlay.

Drawing transparent objects.

Sometimes, you want to have a transparent object in your scene that still uses the depth buffer. However, even if you give this object a transparent texture, it will still block objects behind it because the it still writes to the depth buffer. The solution is to set `@depth_write 0` and `@layer 1` on the transparent object. With `@depth_write 0`, the object will not update the depth buffer with its current depth values. That way, the object will be obscured by opaque objects in the scene, but other objects will be visible through your transparent object.



A transparent torus ensconcing a rubber duck. By disabling depth_write, other objects are still visible through the transparent object.

Geometry Objects

Learning & Examples 255
Half-edge Structure 256
Creating Geometries 257
Remeshing, Subdividing, Decimating 258
Texture Cooridinates and Surface Normals 259
Effect Chains 259
Drawing Geometries 261
Asynchrony 261

The `jit.geom` family of objects, like jit.geom.shape, are a specialized group of Jitter objects designed to work with **Half-edge Geometry Structures**. This way of representing geometries makes certain kinds of geometry manipulations much easier and more efficient:

- *Generating shapes* - Use jit.geom.shape to create simple geometries with an arbitrary number of subdivisions.
- *Vertex adjustment* - The jit.geom.remesh object lets you rearrange the vertices of a geometry, making them more evenly distributed without changing the overall shape. You can add and remove vertices with jit.geom.subdivide and jit.geom.decimate.
- *Transformations* - Smooth out a geometry with jit.geom.smooth, add roughness or procedural displacement with jit.geom.dimples and jit.geom.displace, or perform more radical transformations with jit.geom.twist, jit.geom.waves, and jit.geom.xform.
- *Property generation* - Create dynamic, adjustable shapes with lifelike surface properties, using jit.geom.texgen and jit.geom.normgen to generate texture coordinates and surface normals.

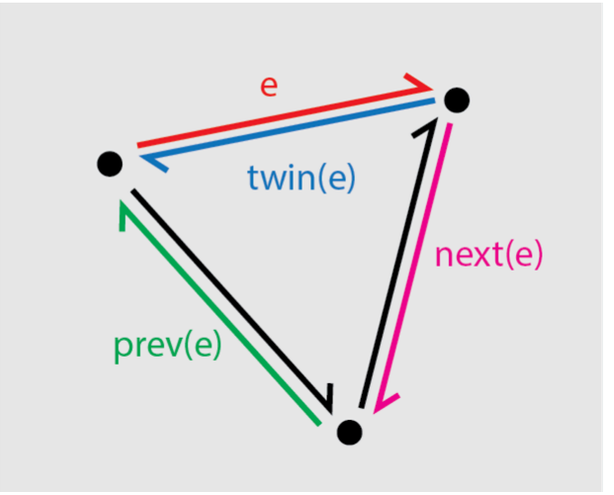
Learning & Examples

Follow the Jitter Geometry tutorial series, or check out some examples of what's possible with these objects:

Example	Description
Contours	Example of how to make a Jitter geometry look hand-drawn
Fractals	Fractal geometries using jit.geom + JavaScript
Point Cloud	Generate a point cloud using jit.geom
Thickness	Extrude triangles to create volumes

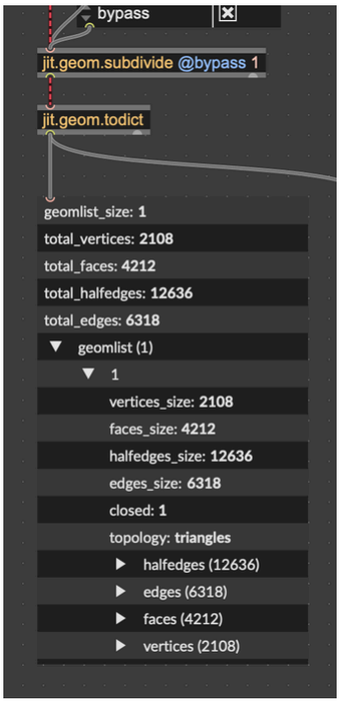
Half-edge Structure

The Jitter geometry objects represent geometries using a half-edge structure. This structure makes it easy to determine which vertices are adjacent to each other, so that manipulations like extruding, decimating, and remeshing are all possible. The basic idea is that each vertex should maintain a pointer to the next vertex in the shape. This pointer from one vertex to the next is the **half-edge** that gives the structure its name. Each half-edge also has a twin (sometimes called its opposite) that points from the next vertex back to the original. These two half-edges together make up one edge.



A single triangular face, in a half-edge representation.

One more requirement completes the definition of the half-edge structure, that following the half-edges from one vertex to the next must loop around one face of the 3D shape. You can view the structure of a half-edge geometry as text using jit.geom.todict, which converts the geometry to a structured dictionary.



A half-edge structure as a dictionary

Creating Geometries

Use the jit.geom.shape object to create a simple geometry, like an icosphere, torus, or cube. You can also use the jit.geom.togeom object to convert a Jitter matrix of triangles into a geometry. Like you see here, it can work with the output of jit.gl.model, but it works just as well with shapes from jit.gl.gridshape, or with any kind of triangular mesh.



Use jit.geom.togeom to convert a Jitter matrix to a geometry

> If you're not working with the output of jit.gl.model or jit.gl.gridshape, then you'll need to make sure that the input to jit.geom.togeom follows the standard for 3D surfaces as represented by a Jitter matrix.

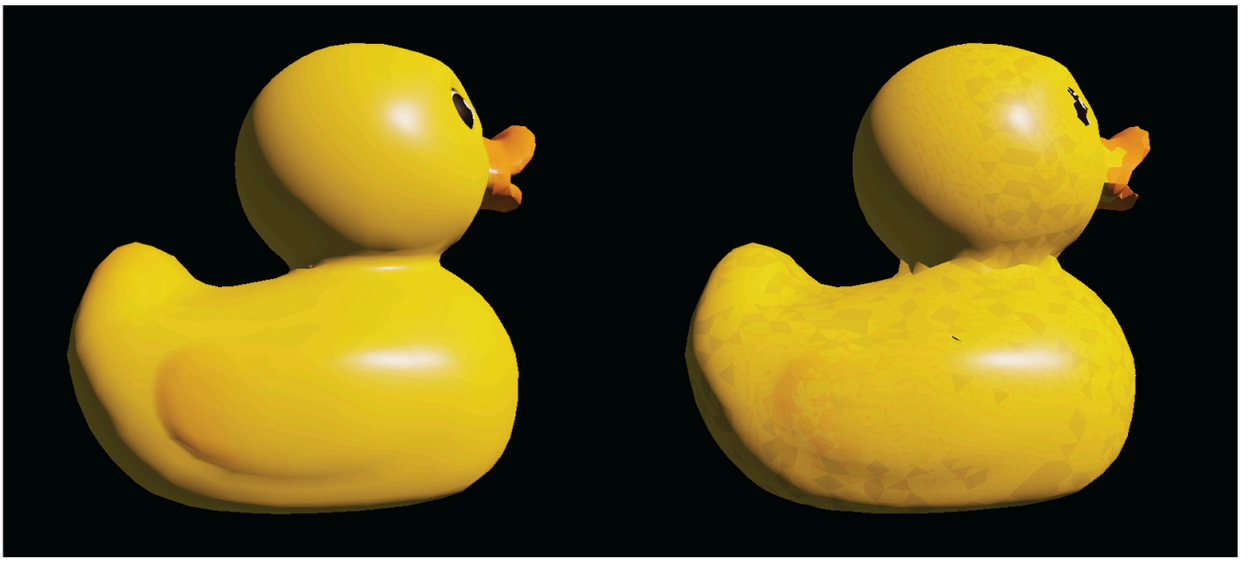
Remeshing, Subdividing, Decimating

You can adjust the shape of a Jitter geometry using various objects.

Object	Description
jit.geom.remesh	Evenly redistribute vertexes without changing the overall shape
jit.geom.subdivide	Subdivide each face into multiple faces, creating new vertices without changing the overall shape
jit.geom.decimate	Combine vertices, reducing the number of vertices wihtout changing the overall shape
jit.geom.smooth	Smooth out the surface vertices, which will distort the surface of the shape

Texture coordinates

The jit.geom.remesh algorithm cannot generate texture coordinates. Once a geometry passes through this object, there's no guarantee that the texture coordinates will contain anything useful. You can try to use jit.geom.texgen to generate new texture coordinates, although these won't necessarily correspond to the originals.



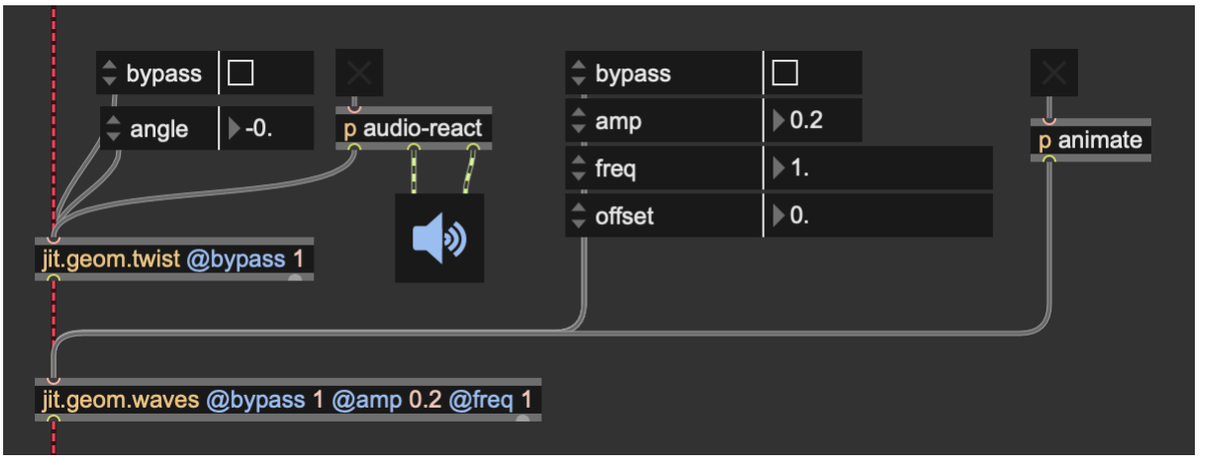
Texture coordinates are no longer usable after remesh

Texture Cooridinates and Surface Normals

Generate new **texture coordinates** for a Jitter geometry with jit.geom.texgen. The jit.geom.texgen objects has a few different algorithms that it can use to compute new texture coordinates, based on the shape of the geometry. Generate new **surface normals** with jit.geom.normgen.

Effect Chains

When chaining multiple Jitter geometry transformation effects together (effects like jit.geom.twist or jit.geom.dimples), you can use the `@bypass` attribute to route geometries through an object unchanged. This is the right way to work with geometry effects.



Geometry routed through a jit.geom.twist and a jit.geom.waves, using @bypass to skip an effect

Generally, it's not a good idea to treat geometry objects like matrix or texture processing objects, when it comes to using a regular gate object to route geometry computations.



The wrong way to bypass a geometry effect

The `jit.geom` objects are careful to only trigger new computation when necessary. Unlike video effects, where every new frame passes through the whole render chain, `jit.geom` objects only trigger new computation when their internal state changes. When you change the state of the gate object in this example, you're not changing the state of any `jit.geom` object. This won't trigger any new computation, and you won't actually see any change to the geometry.

> Of course, you could bang the first object in the `jit.geom` chain to re-trigger computation manually.

Drawing Geometries

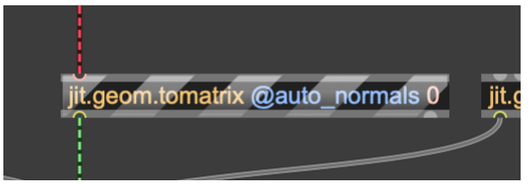
You can draw a Jitter geometry using jit.gl.mesh, after having first converted the geometry to a matrix. The jit.geom.tomesh is slightly more performant, and should be used if you're trying to draw a geometry directly with jit.gl.mesh. If you want to use the geometry for some other purpose, where a Jitter matrix would be more useful, there is alto the jit.geom.tomatrix object to convert a geometry to a full matrix.

> If you connect jit.geom.tomesh to a jit.gl.mesh object, it will automatically set the `@drawmode` attribute of jit.gl.mesh to draw the geometry correctly. Otherwise, you should set `@drawmode triangles` on jit.gl.mesh to draw a geometry.

Both of these objects have an attribute `@auto_normals` that will compute surface normals, which may make an explicit jit.geom.normgen unnecessary.

Asynchrony

By default, the Jitter geometry objects are *asynchronous*. They process their input on a special thread reserved for geometry calculations. This allows them to handle large, complex inputs, without slowing down video, audio, or other computations. When a Jitter geometry object is actively processing, its appearance will change to show that it's currently working.



When a Jitter geometry object is actively processing, it shows zebra stripes.

Because of this potential asynchrony, when you change a Jitter geometry object's attributes, or when you send it a new geometry, the result may not come out of the object's outlet right away. You can bypass this behavior by setting the `@async` attribute to `0`.

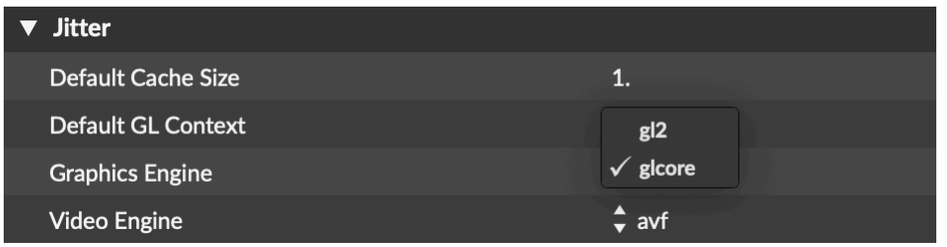
Graphics Engine

Changing the Graphics Engine 263
Engine Comparison 263

The **Graphics Engine** manages the interface between Max and the real time graphics API that interacts with your computer's GPU. It's an abstraction layer that translates your instructions about rendering a 3D or 2D scene into hardware instructions that your computer can understand. Operations like positioning virtual objects, calculating light and shadow, running postprocessing effects, and simulating physics interactions are all handled by the graphics engine.

Changing the Graphics Engine

Under most circumstances, it's not necessary to change the graphics engine. Currently Jitter ships with two engine variants, legacy OpenGL (`gl2`) and core profile OpenGL (`glcore`). As of Max 8.5 the default graphics engine used by Jitter is `glcore` , which was formerly termed `gl3` . Users needing legacy behavior can set the Graphics Engine preference to `gl2` .



Engine Comparison

For a full engine comparison, see the legacy OpenGL discussion. Summarized briefly, the following have been deprecated:

- Fixed-function vertex and fragment processing, as the rendering pipeline is now fully programmable.

- Immediate-mode vertex attribute specification and client-side vertex arrays.
- The GL_QUAD and GL_POLYGON primitive types.
- Fixed-function lighting, materials and color materials, fixed-function shadow mapping and bump mapping.

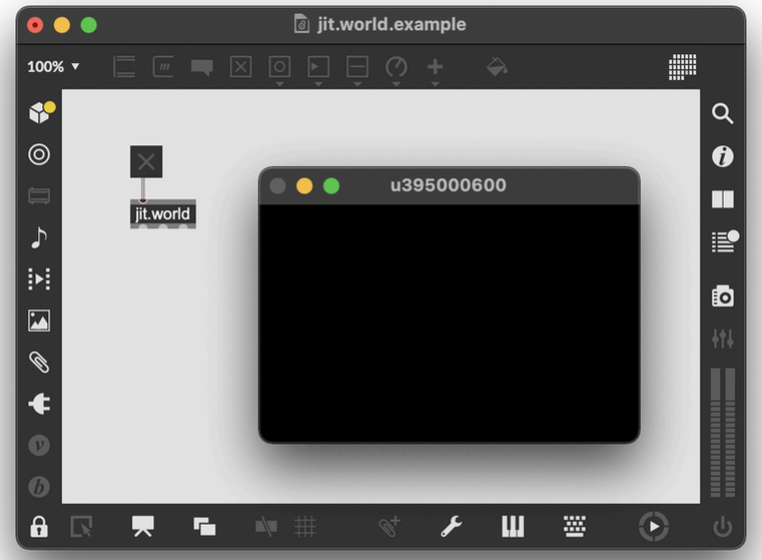
Graphics Processing

Graphics Contexts 265
Cameras 268
Geometry 270
Materials 273
Lights 276
Instancing 276
Depth Buffer 278
Handles 278
Models 279

Whether you're using 3D modeling software, a game engine, or writing your own shaders, creating and displaying *Computer Graphics* invovles many of the same concepts. Max has many capabilities when it comes to working with computer graphics, and this document will describe how to make use of them.

Graphics Contexts

In order to draw, you must have some place to draw to. The Max object jit.world is the simplest way to get a graphics context. This object wraps together many other, lower level objects. The two most important of these objects are jit.window, which creates a graphics context in a separate window, and jit.gl.render, which can be used to direct Max to render a single frame of graphics.

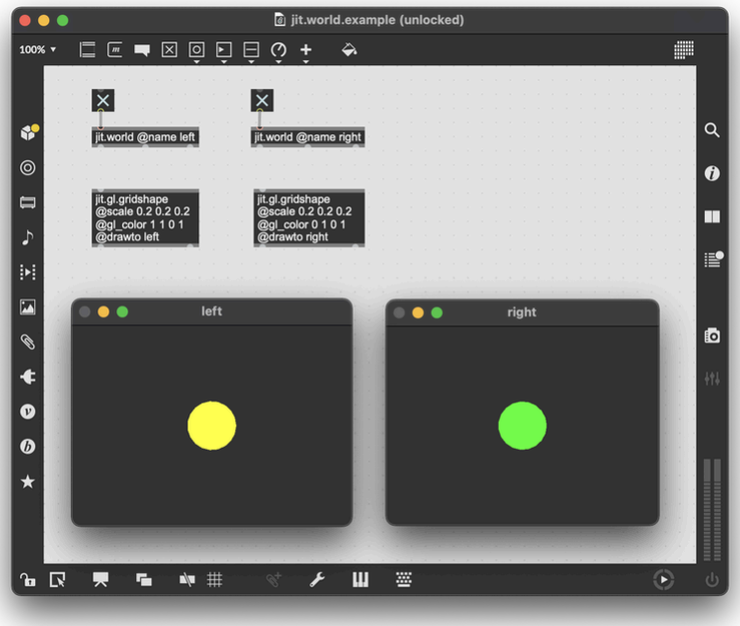


Creating a simple graphics context with {jit.world}

While jit.world creates a separate window, you can also use the jit.pworld object, which has similar functionality as jit.world, except the graphics context appears as a frame in the Max patcher.

Context names

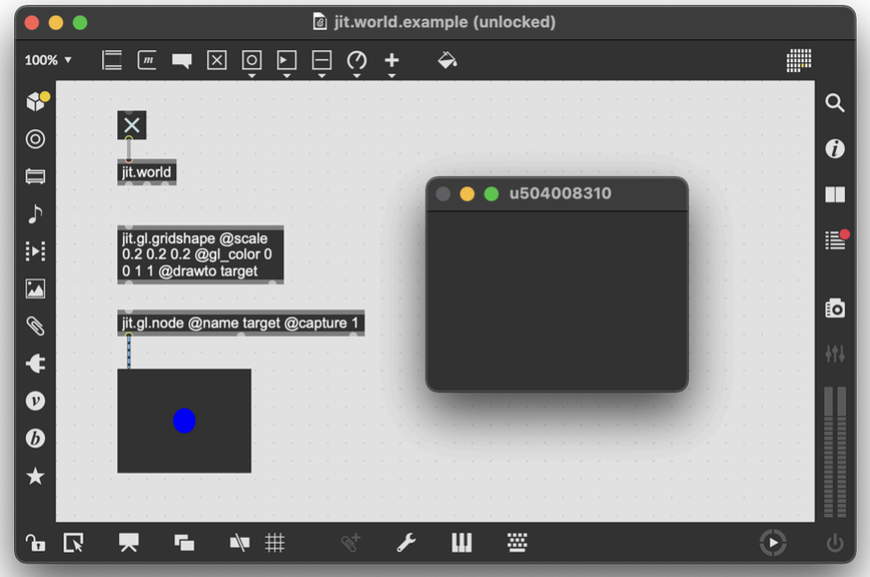
Unlike some Max objects, Jitter objects that deal with graphics processing don't need to connect to each other to pass data around. Most graphics processing objects (usually with the `jit.gl` prefix) refer to their parent context by name. Even if you don't assign an explicit name to your graphics context using the `@name` attribute, Max will create a unique name automatically when you create it. If you add a graphics processing object to your patcher, by default it will bind to whatever graphics context is available in the current patcher. However, if you have multiple graphics contexts in a single patcher, you can use an attribute to bind the object specifically to a particular context. Most objects use the `@drawto` attribute to choose their parent context explicitly. An alternative way to set `@drawto` is to provide the context name as the first argument following the object name.



Using `@drawto` to pick a parent context

Offscreen contexts

It is possible to hide the window created by jit.world using the `@visible` attribute. However, the best way to create an offscreen context is with the jit.gl.node object. Using the `@name` attribute, you can give jit.gl.node a name, and then use the `@drawto` attribute to bind objects to the jit.gl.node. Finally, enable texture capture with `@capture 1` on the jit.gl.node object. In this configuration, the jit.gl.node object functions as an offscreen render target. The output of jit.gl.node will be a texture to which you can add postprocessing.



Capturing to an offscreen context with {jit.gl.node}

Note that here, since the jit.gl.gridshape is drawing to the context owned by jit.gl.node, the shape is not visible in the window context owned by jit.gl.world. However, for jit.gl.node to function, it must still have its own parent context. So the jit.world object must be present for this patcher to function.

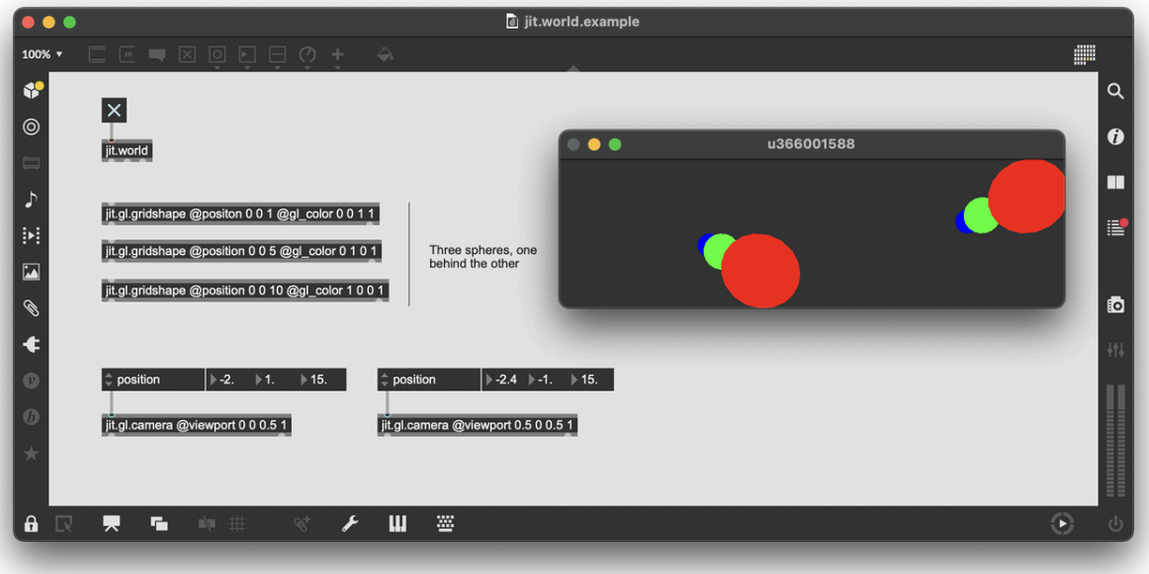
Cameras

Create a camera with the jit.gl.camera object. This object has attributes for managing rendering according to perspective.

Attribute	Type	Description
`@lookat`	vec3	Point the camera towards this position. Setting this attribute will override `@direction`.
`@position`	vec3	The position of the camera. Combines with `@lookat` to determine the orientation of the view.

`@rotatexyz`	vec3	Rotates the camera. Setting `@direction` or `@lookat` will override this, since they are all three different ways of defining the orientation of the camera.
`@near_clip`	float	Only vertices further from the camera than this minimum distance will be rendered.
`@far_clip`	float	Only vertices nearer to the camera than this maximum distance will be rendered.
`@lens_angle`	float (degrees)	The angle between the limits of the camera's view. A narrower lens angle increases the size of visible objects, while decreasing the amount of the scene that can be seen.
`@ortho`	enum (int)	Manages camera perspective. `0`, the default value, imitates natural perspective vision by making more distant objects appear smaller. `1` instead uses orthographic projection, in which all objects are the same size on matter their distance. `2` is also orthorgraphic, but with a fixed lens angle.
`@tripod`	bool	`1` enables tripod mode, in which the camera vertical orientation will be locked pointing upwards. Avoids unintended camera roll.

Multiple cameras can belong to the same graphics context. When rendering, each camera will render one after the other, compositing their output if blending is enabled. The `@viewport` attribute can be used to render the camera to only part of the graphics context.



Rendering with two cameras to two different viewports

Geometry

Special purpose objects

The easiest way to create a geometry is to use the jit.gl.gridshape object. This object generates geometry for one of several standard 3D shapes. By default, it will also render that shape to the current graphics context. The jit.gl.gridshape is one of several **special purpose** objects that generate procedural geometry, and which can render that geometry in the current context.

Object	Description
jit.gl.gridshape	Generate simple geometric shapes as a grid
jit.gl.plato	Generate platonic solids
jit.gl.nurbs	Generate Non-Uniform Rational B-Spline surfaces
jit.gl.text3d	Generate three dimensisional text
jit.gl.model	Load 3D models
jit.gl.graph	Generate 3D graphs

jit.gl.sketch	State machine interface
jit.gl.lua	State machine interface using Lua programming

Half-edge geometry objects

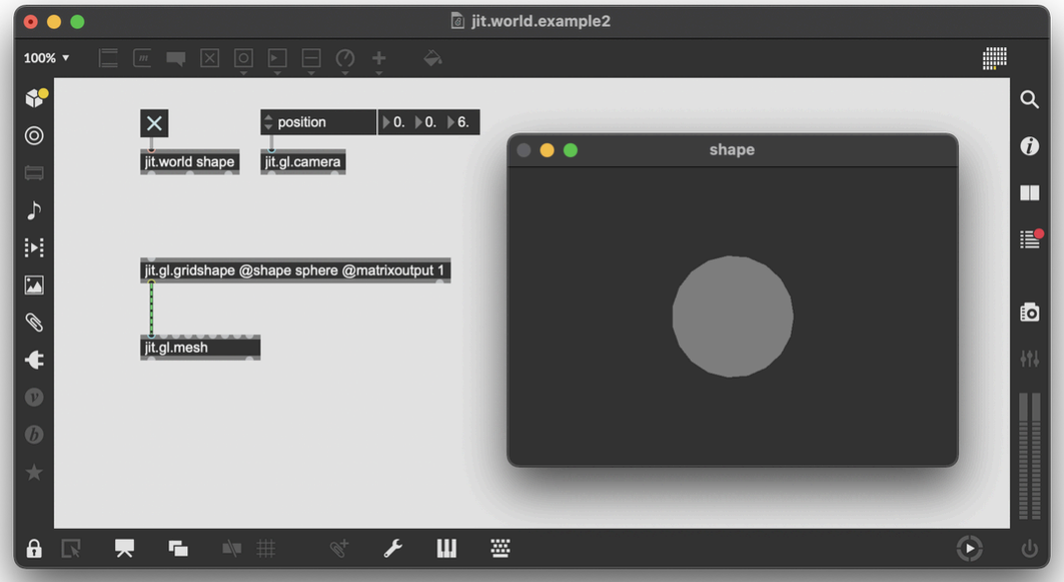
The `jit.geom` family of objects, like jit.geom.shape, are a specialized group of Jitter objects designed to work with **Half-edge Geometry Structures**. See the Jitter Geometry guide for more information.

Using @matrixoutput with jit.gl.gridshape

If you enable the `@matrixoutput` of jit.gl.gridshape, then the object will no longer render its shape to the current context. Instead, the procedurally generated geometry will come out of the first outlet as a matrix. The output is a Jitter matrix with multiple planes describing the geometry. The dimension of the output is equivalent to the number of points in the mesh.

Planes	Type	Description
0-2	vec3	Vertex position (using **tri_grid** order)
3-4	vec2	Texture coordinates
5-7	vec3	Normal vector
8-11	vec4	Color (rgba)

To render geometry as generated with a jit.gl.gridshape object, use the jit.gl.mesh object. By default the mesh uses a totally flat shader for rendering, so the 3D geometry from jit.gl.gridshape will appear as a flat 2D shape.



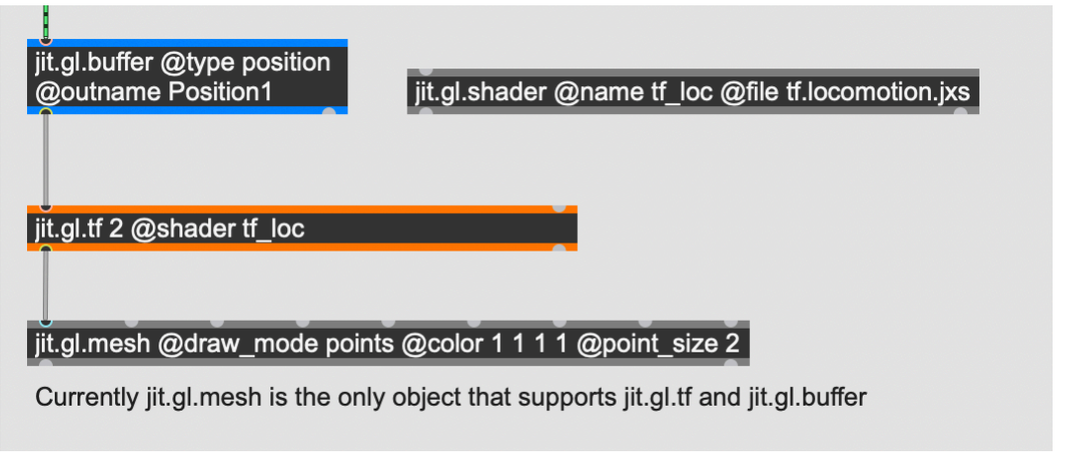
Rendering the matrix output of {jit.gl.gridshape} using a {jit.gl.mesh} object.

Custom geometry

In addition to using jit.gl.gridshape to generate a simple geometry from a standard list of shapes, you can also supply your own Jitter matrices to jit.gl.mesh. You can use whatever Jitter processing technique you like to generate these matrices, but one really convenient way would be to use jit.gen. This object lets you manipulate Jitter matrices on a cell-by-cell level, which will be familiar to anyone who has worked with geometry shaders before.

Processing geometry on the GPU with vertex shaders

The jit.gl.gridshape and other special purpose procedural geometry objects, including jit.gen, run entirely on the CPU. Because of this, they can be slow for very large, complex, dynamic geometries. For more sophisticated geometry processing, using vertex shaders, use the jit.gl.tf object. When used in conjunction with jit.gl.buffer and jit.gl.mesh objects, along with a vertex shader held in jit.gl.shader, the jit.gl.tf object can do complex geometry processing on the GPU, with feedback.



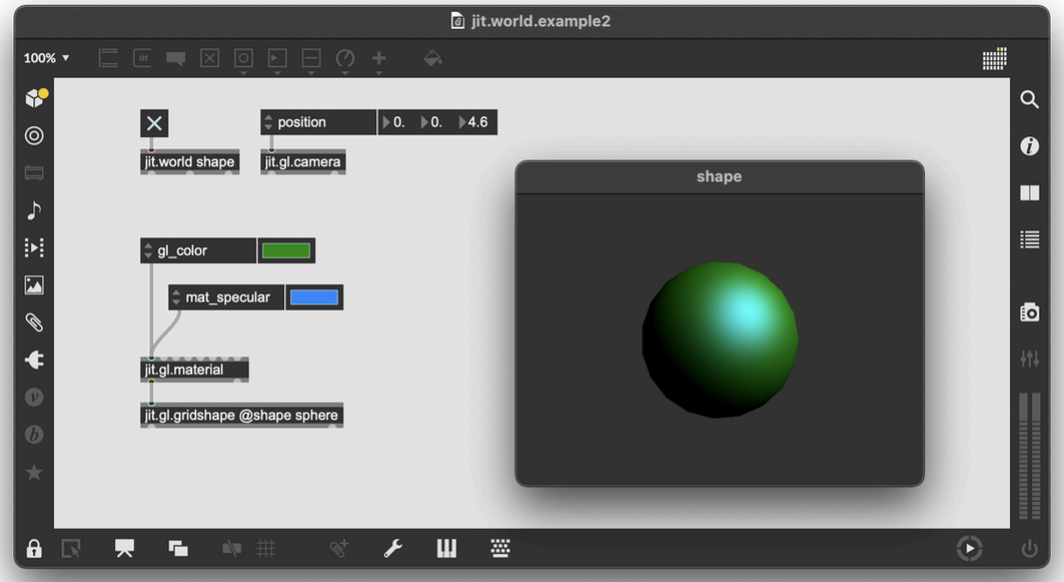
Close up on the {jit.gl.tf} object, in conjuction with other objects needed for transform feedback

Materials

To determine the final appearance of a 3D object, a graphics program will combine information about the object's shape with information that describes the surface of the object and how it interacts with light. This information is called the object's **material**. You can define the material for an object by connecting a jit.gl.material object to the target object, or by setting the `@material` attribute on that object.

jit.gl.material

The simplest object for working with material is jit.gl.material. This object uses a simple lighting model that assumes that the apparent color of an object is the sum of contributions from ambient, diffuse, and specular light.

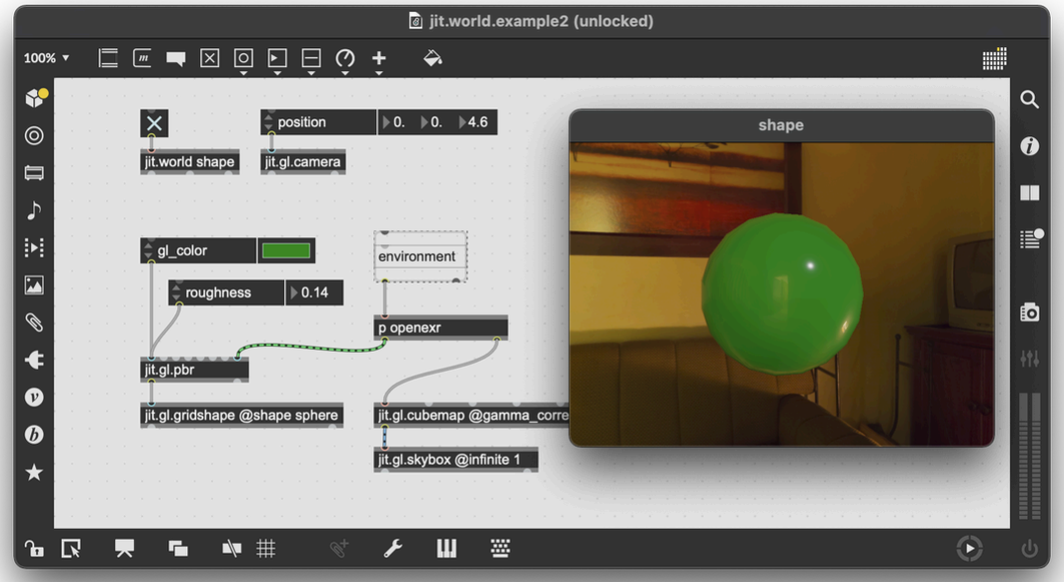


Close up on the {jit.gl.tf} object, in conjuction with other objects needed for transform feedback

In addition to diffuse, ambient, and specular textures, the jit.gl.material object also has inputs for a custom normal map, height map, and environment map. When using a custom height map, the `@heightmap_mode` attribute gives access to two different ways of handling the height map. With `@heightmap_mode parallax`, the material will use **parallax mapping** to manipulate the texture map for a more realistic look. By contrast, `@heightmap_mode vtf` will manipulate the object's geometry, changing its silhouette and recalculating its surface normals.

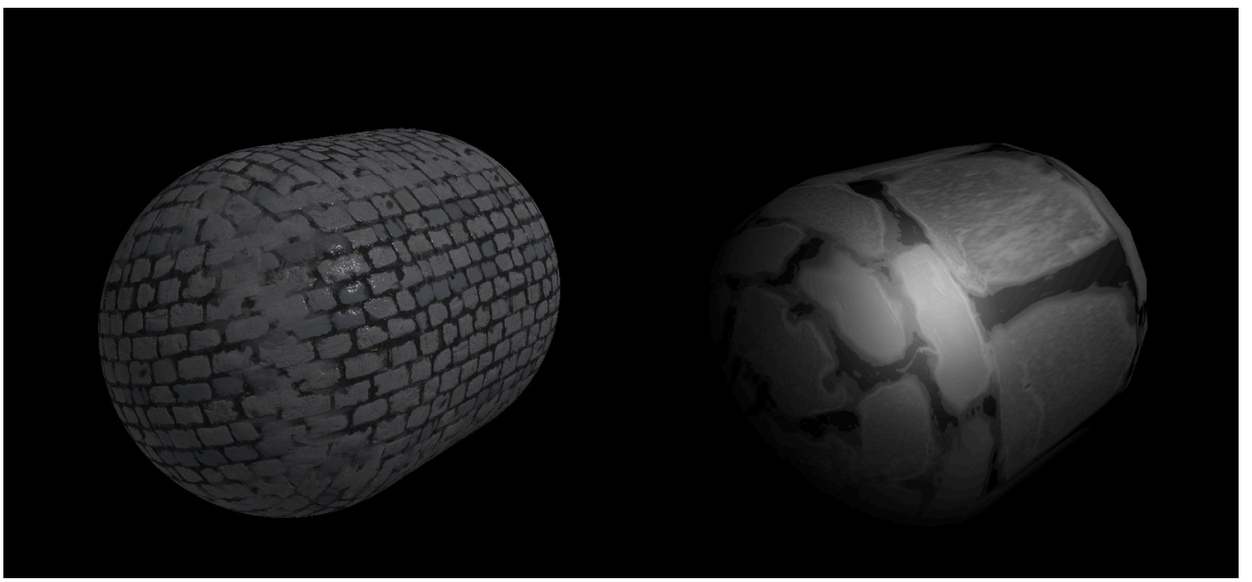
jit.gl.pbr

The jit.gl.pbr object implements a more realistic material model for 3D objects (the "pbr" stands for "physically based rendering"). Unlike jit.gl.material, the **pbr** model uses the **metalness** and **roughness** of an object to determine how it reflects light.



jit.gl.pbr with more realistic lighting

Similar to jit.gl.material, the jit.gl.pbr object can take a height map as input. The height map can be used for parallax mapping, and for recomputing surface normals. The jit.gl.pbr object can also automatically generate texture coordinates based on the geometry of the attached mesh.



On the left, automatically generated texture coordinates with {jit.gl.pbr}. On the right, default texture coordinates with {jit.gl.material}

When using an environment map with jit.gl.pbr, the material can take specular highlights from the environment, greatly adding to the sense of physicality.

Lights

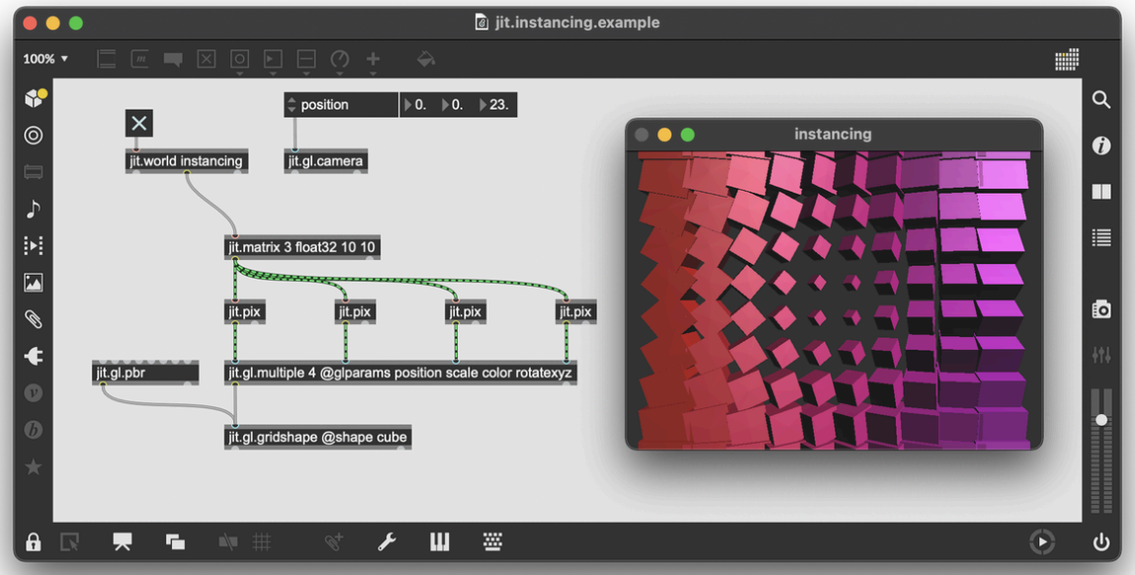
You can add lights to a graphics context with the `jit.gl.light` object. Through the `@type` attribute, this object supports four different standard types of lights.

Type	Description
point	Light emitted from a single point in all directions. Similar to the light from a lightbulb.
directional	Light emitted as parallel rays from a distant source. Similar to light from the sun.
spot	Light emitted from a point in th eshape of a cone.
hemisphere	Light from two hemispheres. The `@diffuse` attribute defines the color of one hemisphere, and the `@ambient` attribute defines the color of the other hemisphere.

Instancing

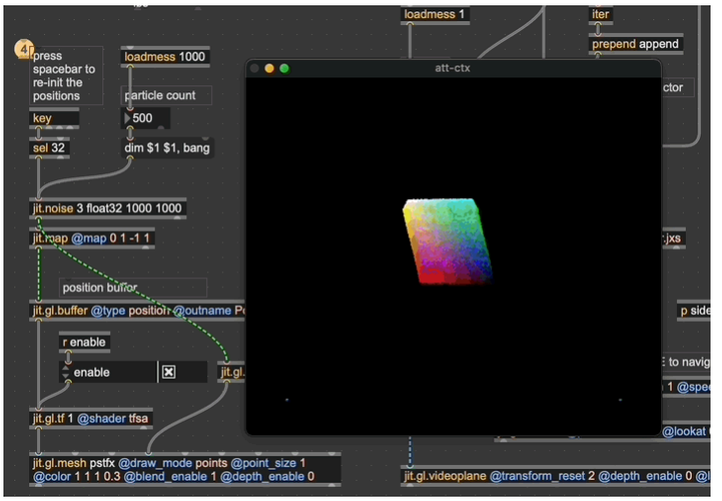
When rendering the same object multiple times, as when rendering snow or smoke or a flock of birds, we can gain efficiency through **instancing**. The idea is, we try to keep as much data as possible on the GPU (where the drawing happens), because copying data from the CPU (where we work in Max) is computationally expensive.

The first way to do instancing is with jit.gl.multiple. To multiply an object with jit.gl.multiple, connect to the object that should be instanced.



The jit.gl.multiple object takes a matrix at each input. The number of cells in the matrix determines the number of instances to create, and the data contained at each cell maps to a different parameter. This example uses simple parameters like *position* and *scale*, but it's also possible to map a different material to each instance.

The second is with jit.gl.buffer. This object manages a named reference to a buffer of data stored on the GPU. It's analogous to jit.gl.multiple, except the data managed by jit.gl.buffer never leaves the GPU, and can be manipulated with a shader program managed by jit.gl.tf. For particle effects and simulations involving a very large number of agents that update according to simple rules as defined in a vertex shader, the objects jit.gl.buffer, jit.gl.tf, and jit.gl.mesh are the best choice.



Strange attractor particle simulation using 1,000,000 points, using jit.gl.buffer and jit.gl.tf

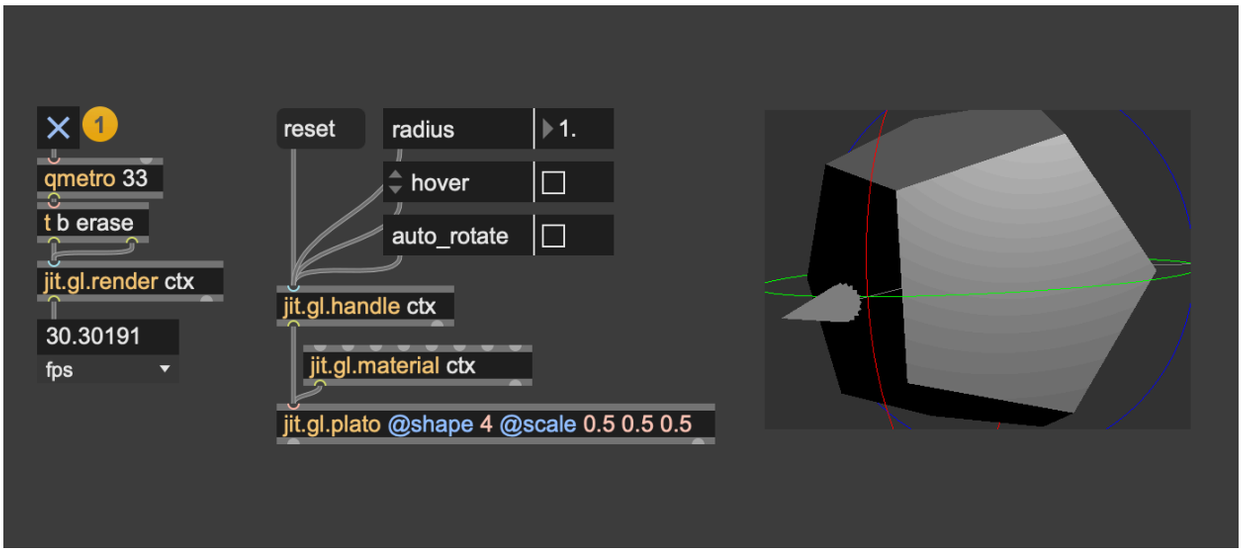
Depth Buffer

Objects rendered in Jitter write to the depth buffer by default, which the renderer will use to draw objects on top of each other in the expected way. The attributes `@depth_write`, `@depth_clear`, and `@depth_enable` can be used to configure how an object interacts with the depth buffer—whether it writes to, clears, or uses the depth buffer at all. For an in-depth discussion of depth testing and layering, see the Depth Testing and Layering guide page.

You can retrieve the depth buffer using a jit.gl.pass object. A custom render pass can get a source from `NORMALS`, which includes both the surface normals as well as the depth for each fragment. See Render Passes for more details.

Handles

The jit.gl.handle object an bind to another Jitter object, mouse interaction with a Jitter graphics context to matrix transformations. In othre words, attach a jit.gl.handle to an object like jit.gl.gridshape if you want to click and drag on your graphics context to to rotate, scale, and move that object.



Use jit.gl.handle to manipulate the transforms of objects in your scene

Models

Load models into your scene with the jit.gl.model object. This object can load most common model formats, like OBJ, Collada, and Blender. Check out the jit.gl.model help file for details on how to work with rigged models, how to fetch materials descriptions from the model, and how to read and play animations.

Jitter expr

Inputs 281
Functions and Operators 282
Matrix Operators 286
Basis Function Generators 287
Constants 289

This document describes the Jitter expression language syntax. This is the language used for the jit.expr object, as well as the `exprfill` message to a jit.matrix objects. This expression will be evaluated once for each cell in the input, in order to generate an output matrix with the same matrix dimensions as the input.

You could make a jit.expr that passes through input unchanged.

```
jit.expr @expr in[0]
```

Or you could use an expression to add the first and second inputs together:

```
jit.expr @expr in[0] + in[1]
```

Or do something more sophisticated, like apply a vignette mask:

```
jit.expr @expr (1-hypot(snorm[0],snorm[1]))*in[0]*2.
```

Inputs

Reference the inputs to the expr object with `in`. Get the first input with `in[0]`, the second with `in[1]`, and so on. Use `p` to get a specific plane of the input, where `in[0].p[0]` accesses the first plane of the first input, `in[0].p[1]` gets the second plane of the first input, and so on.

> The jit.expr object defaults to two inputs, but you can increase the number of inputs with the `@inputs` attribute.

To get the cell coordinates, use either `cell`, `norm`, or `snorm`. The index is used to select the dimension, so `cell[0]` is the cell coordinate in the x direction (first dimension), `cell[1]` is the cell coordinate in the y direction, and so on. You can also use `dim` to get the dimensions of the matrix in any direction.

Finally, it's also possible to access any named matrix by simply using the name of that matrix instead of `in`. With a matrix named `my_matrix`, use `my_matrix` to access the contents of that matrix at the current cell coordinates, or `my_matrix.p[0]` to access the first plane of that matrix.

Name	Description
in[0-31]	Input matrix cell contents, corresponding plane
in[0-31].p[0-31]	Input matrix cell contents, specific plane
cell[0-31]	Cell coordinates of the current cell
norm[0-31]	Normalized (0 to 1) cell coordinates
snorm[0-31]	Signed normalized (-1 to 1) cell coordinates
dim[0-31]	Matrix dimension size
matrixname[0-31]	Named matrix cell contents, corresponding plane

matrixname[0-31].p[0-31]	Named matrix cell contents, specific plane

Functions and Operators

Within a Jitter expression, Jitter operators can be applied as functions. For the most part, these are the same operators as are defined for the `jit.op` object. Some of these can be applied as infix operators, for example `in[0] + in[1]` for the `+` operator. Others are called as functions, with commas between arguments, like `absdiff(in[0], in[1])`.

> Instead of the Jitter operators `>p` and `<p`, use the function calls `gtp(val, test)` and `ltp(val, test)`.

Arithmetic

Name	Description
`pass`	pass left input, no operator
`*`	multiplication (also `mult`)
`/`	division (also `div`)
`+`	addition (also `add`)
`-`	subtraction (also `sub`)
`+m`	addition modulo (char only) (also `addm`)
`-m`	subtraction modulo (char only) (also `subm`)
`%`	modulo (also `mod`)
`min`	minimum
`max`	maximum

`abs`	absolute value (unary)
`avg`	average
`absdiff`	absolute value of difference
`fold`	folding/mirrored modulo (float only)
`wrap`	wrapping/positive modulo (float only)
`!pass`	pass right input, no operator
`!/`	right input divided by left input (flipped)
`!-`	right input minus left input (flipped)
`!%`	right input modulo left input (flipped)
`ignore`	leave previous output value

Trigonometric

> (float32/float64 only, unary except atan2)

Name	Description
`sin`	sine
`cos`	cosine
`tan`	tangent
`asin`	arcsine
`acos`	arccosine
`atan`	arctangent
`atan2`	arctangent (binary)

`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`asinh`	hyperbolic arcsine
`acosh`	hyperbolic arccosine
`atanh`	hyperbolic arctangent

Bitwise

> (long/char only)

Name	Description
`&`	bitwise and
`|`	bitwise or
`^`	bitwise xor
`~`	bitwise compliment (unary)
`>>`	right shift
`<<`	left shift

Logical

Name	Description
`&&`	logical and
`||`	logical or

`!`	logical not (unary)
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`==`	equal
`!=`	not equal
`>p`	greater than (pass)
`<p`	less than (pass)
`>=p`	greater than or equal to (pass)
`<=p`	less than or equal to (pass)
`==p`	equal (pass)
`!=p`	not equal (pass)

Exponential/Logarithmic/Other

> (float32/float64 only, unary except hypot and pow)

Name	Description
`exp`	e to the x
`exp2`	2 to the x
`ln`	log base e
`log2`	log base 2

`log10`	log base 10
`hypot`	hypotenuse (binary)
`pow`	x to the y (binary)
`sqrt`	square root
`ceil`	integer ceiling
`floor`	integer floor
`round`	round to nearest integer
`trunc`	truncate to integer

Matrix Operators

Many Jitter **Matrix Operators** can be used inside of a Jitter expression as well. These are the functional equivalent of Jitter objects like `jit.clip`, objects that perform a simple operation on their matrix inputs. There is no exhaustive list of these, but if there's a simple Jitter object that you'd like to use in an expression, chances are it will work. You could use the following expression to convolve two matrices.

```
jit.expr @expr "jit.convolve(in[0], in[1])"
```

If the Jitter matrix operator can be configured with attributes, those can be supplied as an attribute list following the other arguments to the matrix operator. The following would constrain the values in each plane to be between 0.2 and 0.8.

```
jit.expr @expr "jit.clip(in[0], @min 0.2 @max 0.8)"
```

Basis Function Generators

The basis function generators from the jit.bfg object can also be used inside of an expression. These work in much the same way as matrix operators.

```
jit.expr @expr "noise.gradient(norm[0]*2, norm[1]*2, @seed 313)"
```

Distance functions

Name	Description
distance.chebychev	Absolute maximum difference between two points
distance.euclidean	True straight line distance in Euclidean space
distance.euclidean.squared	Squared Euclidean distance
distance.manhattan	Rectilinear distance measured along axes at right angles
distance.manhattan.radial	Manhattan distance with radius fall-off control
distance.minkovsky	Exponentially controlled distance

Filter functions

Name	Description
filter.box	Sums all samples in the filter area with equal weight
filter.gaussian	Weights samples in the filter area using a bell curve
filter.lanczossinc	Weights samples using a steep windowed sinc curve
filter.mitchell	Weights samples using a controllable cubic polynomial
filter.disk	Sums all samples inside the filter's radius with equal weight
filter.sinc	Weights samples using an un-windowed sinc curve

filter.catmullrom	Weights samples using a Catmull-Rom cubic polynomial
filter.bessel	Weights samples with a linear phase response
filter.triangle	Weights samples in the filter area using a pyramid

Transfer functions

Name	Descrpition
transfer.step	Always 0 if value is less than threshold, otherwise always 1
transfer.smoothstep	Step function with cubic smoothing at boundaries
transfer.bias	Polynomial similar to gamma but remapped to unit interval
transfer.cubic	Generic 3rd order polynomial with controllable coefficients
transfer.saw	Periodic triangle pulse train
transfer.quintic	Generic 5th order polynomial with controllable coefficients
transfer.gain	S-Shaped polynomial evaluated inside unit interval
transfer.pulse	Periodic step function
transfer.smoothpulse	Periodic step function with cubic smoothing at boundaries
transfer.sine	Periodic sinusoidal curve
transfer.linear	Linear function across unit interval
transfer.solarize	Scales given value if threshold is exceeded

Noise functions

Name	Description
noise.cellnoise	Coherent blocky noise
noise.checker	Periodic checker squares
noise.value.cubicspline	Polynomial smoothed pseudo-random values

noise.value.convolution	Convolution filtered pseudo-random values
noise.sparse.convolution	Convolution filtered pseudo-random feature points
noise.gradient	Directionally weighted polynomially interpolated values
noise.simplex	Simplex weighted pseudo-random values
noise.voronoi	Distance weighted pseudo-random feature points
noise.distorted	Domain distorted combinational noise

Fractal functions

Name	Description
mono	Additive fractal with global simularity across scales
multi	Multiplicative fractal with varying simularity across scales
multi.hybrid	A hybrid additive and multiplicative fractal
multi.hetero	Heterogenous multiplicative fractal
multi.ridged	Multiplicative fractal with sharp ridges
turbulence	Additive mono-fractal with sharp ridges

Constants

There's also a handful of constants ready as well. An expression like `in[0] * DEGTORAD` would convert a matrix full of degrees to a matrix full of radians.

Name	Description
PI	Ratio of a circle's circumference to its diameter
TWOPI	Twice the value of pi
HALFPI	Half the value of pi

INVPI	One over pi
DEGTORAD	Scale factor to convert degrees to radians
RADTODEG	Scale factor to convert radians to degrees
E	Base of the natural logarithm
LN2	Natural logarithm evaluated for 2
LN10	Natural logarithm evaluated for 10
LOG10E	Log base 10 evaluated for e
LOG2E	Log base 2 evaluated for e
SQRT2	Square root of 2
SQRT1_2	One over the square root of 2

The JXS File Format

The jittershader Tag 293
The description tag 293
The texture tag 293
Parameters and the param tag 294
Shaders and the language tag 304

Jitter objects that work with shaders use a special shader description file format called **JXS**, or *Jitter XML Shader*. This file tells Max how to load a list of shaders and connect them up to Max. It includes a description, a list of textures and parameters, and a list of shaders. There must always be one vertex and fragment shader, and there can be an optional geometry shader. A typical JXS file looks like this:

```
<jittershader name="myshader">
  <!-- optional description -->
  <description>This is my shader</description>

  <!-- optional list of texture objects to bind -->
  <texture file="chilis.jpg" rectangle="0"/>

  <!-- optional list of shader parameters -->
  <param name="myparam" type="vec3" default="1 2 3" >
    <description>This is my parameter</description>
  </param>

  <!-- optional list of shader state parameters -->
  <param name="modelViewProjectionMatrix" type="mat4"
state="MODELVIEW_PROJECTION_MATRIX" />
  <param name="jit_position" type="vec3" state="POSITION" />
  <param name="jit_texcoord" type="vec2" state="TEXCOORD" />

  <!-- optionally include other glsl sources -->
  <include source="diffuse.glsl" program="vp" />

  <!-- list of language implementations -->
  <language name="glsl" version="1.5">

    <!-- list of binding targets for shader parameters -->
    <bind param="modelViewProjectionMatrix" program="vp" />
    <bind param="jit_position" program="vp" />
    <bind param="jit_texcoord" program="vp" />
    <bind param="myparam" program="fp" />

    <program name="vp" type="vertex">
      <![CDATA[
      #version 330 core

      in vec3 jit_position;
      in vec2 jit_texcoord;
      out jit_PerVertex {
        vec2 texcoord0;
      } jit_out;
      uniform mat4 modelViewProjectionMatrix;

      void main() {
```

```
  jit_out.texcoord0 = jit_texcoord;
        }
        ]]>
      </program>
      <program name="fp" type="fragment">
        <![CDATA[
        #version 330 core

        in jit_PerVertex {
          vec2 texcoord0;
        } jit_in;
        layout (location = 0) out vec4 outColor;
        uniform sampler2D tex0;
        uniform vec3 myparam;
        void main() {
          outColor = vec4(myparam, 1) * texture(tex0, jit_in.texcoord0);
        }
        ]]>
      </program>
    </language>
  </jittershader>
```

The `jittershader` Tag

A JXS file opens with a `jittershader` tag. This tag has a single, optional attribute, the `name` attribute. This attribute is currently unused by Max, but by convention a JXS author should provide a descriptive name.

The `description` tag

A `jittershader` may include a `description`. This description is also optional, but by convention this is a good place to document the intended behavior of a JXS file. Max may use this information when listing available shaders.

The `texture` tag

A `jittershader` may have one or more textures, defined using a `texture` tag. The `file` attribute loads the texture from an image file, which may be any file in Max's Search Path.

```
<texture file="mytexture.jpg"/>
```

A `texture` may be modified, using any attribute of the jit.gl.texture object.

```
<texture file="gn.gradperm.png" rectangle="0" filter="none none"
wrap="repeat repeat" mipmap="none" anisotropy="0" />
```

Bind a texture to a parameter using the `unit` attribute. This defaults to zero, and so may be omitted if there is only one texture parameter.

```
<!-- The texture has unit="1", so binds to texture tex_rand with
default="1" -->
<param name="tex_normals" type="int" default="0" />
<param name="tex_rand" type="int" default="1" />
<texture file="random-tex.png" type="float16" unit="1" rectangle="0"
filter="none none" wrap="repeat repeat"/>
```

See texture parameters for more details about binding Max textures to shader programs.

Parameters and the `param` tag

Use the `param` tag to create parameters, which define bindings between values available in Max and shader variables. These can bind to numerical Max values, to Jitter textures, or to specific state variables calculated by the graphics engine. A parameter may contain a description.

```
<jittershader>
  <param name="myparam" type="float" default="1.0">
    <description> A parameter description </description>
  </param>
</jittershader>
```

A parameter must also have a type. Parameter types can be most of the primitive types defined by the *glsl* standard.

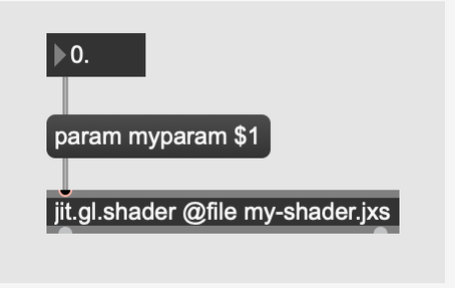
Type	Description
bool	boolean type
int	integer type, **also used for texture input**
float	float-width number type
double	double-width number type
bvec2, bvec3, bvec4	boolean vector of length 2, 3, or 4
ivec2, ivec3, ivec4	integer vector of length 2, 3, or 4
vec2, vec3, vec4	float vector of length 2, 3, or 4
dvec2, dvec3, dvec4	double vector of length 2, 3, or 4
mat2	two-by-two matrix
mat3	three-by-three matrix
mat4	four-by-four matrix

Numeric parameters

Create a binding to a numerical value in Max by creating a `param` tag without a `state` attribute.

```
<jittershader>
  <param name="myparam" type="float" default="1.0" />
</jittershader>
```

Set this value in Max using the `param` message.



Set a numeric parameter with a Max message.

Texture parameters

To define a texture parameter, a `param` should have the type "int" and a default value that matches the parameter index. Max will recognize that this "int" param represents a texture if the `param` is later bound to a sampler type, for example a *sampler2DRect* or a *sampler2D*.

```
<jittershader>
  <!-- Two input textures -->
  <param name="image1" type="int" default="0" />
  <param name="image2" type="int" default="1" />

  <param name="modelViewProjectionMatrix" type="mat4"
state="MODELVIEW_PROJECTION_MATRIX" />
  <param name="textureMatrix0" type="mat4" state="TEXTURE0_MATRIX" />
  <param name="textureMatrix1" type="mat4" state="TEXTURE1_MATRIX" />
  <param name="jit_position" type="vec3" state="POSITION" />
  <param name="jit_texcoord" type="vec2" state="TEXCOORD" />
  <language name="glsl" version="1.5">
    <bind param="image1" program="fp" />
    <bind param="image2" program="fp" />
    <bind param="modelViewProjectionMatrix" program="vp" />
    <bind param="textureMatrix0" program="vp" />
    <bind param="textureMatrix1" program="vp" />
    <bind param="jit_position" program="vp" />
    <bind param="jit_texcoord" program="vp" />
    <program name="vp" type="vertex">
      <![CDATA[
      #version 330 core

      in vec3 jit_position;
      in vec2 jit_texcoord;
      out jit_PerVertex {
        vec2 texcoord0;
        vec2 texcoord1;
      } jit_out;
      uniform mat4 modelViewProjectionMatrix;
      uniform mat4 textureMatrix0;
      uniform mat4 textureMatrix1;

      void main() {
        gl_Position = modelViewProjectionMatrix*vec4(jit_position, 1.);
        jit_out.texcoord0 = vec2(textureMatrix0 * vec4(jit_texcoord, 0.,
1.));
        jit_out.texcoord1 = vec2(textureMatrix1 * vec4(jit_texcoord, 0.,
1.));
      }
    ]]>
    </program>
```

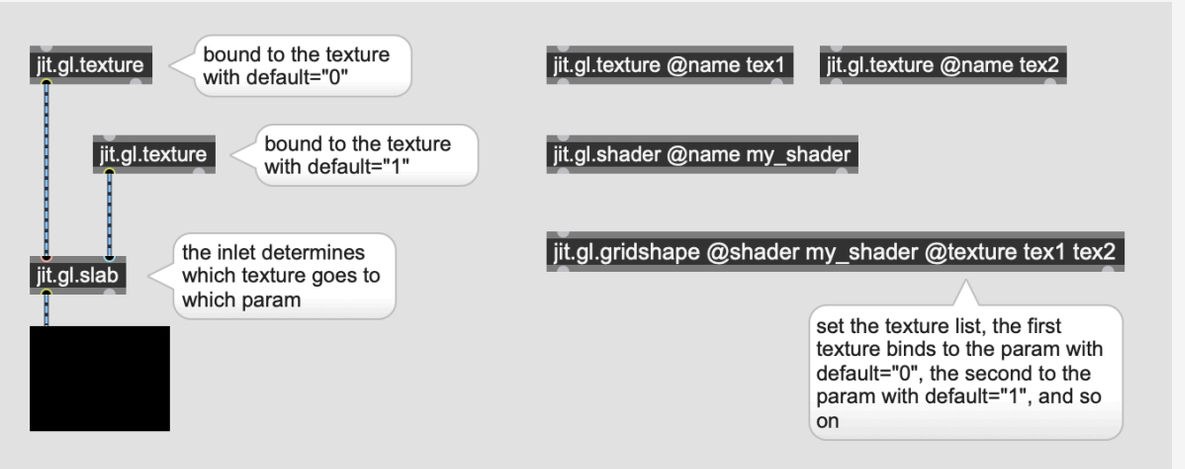
```
  <![CDATA[
        #version 330 core

        // texcoords
        in jit_PerVertex {
          vec2 texcoord0;
          vec2 texcoord1;
        } jit_in;

        layout (location = 0) out vec4 outColor;

        // samplers
        uniform sampler2DRect image1;
        uniform sampler2DRect image2;
        void main() {
          vec4 im1 = texture(image1, jit_in.texcoord0);
          vec4 im2 = texture(image2, jit_in.texcoord1);
          outColor = im1 + im2;
        }
      ]]>
      </program>
    </language>
  </jittershader>
```

In Max, you can set these textures using a texture list, or using the inlets of a jit.gl.slab object.



Setting the texture inputs of a shader

State parameters

Parameters can also bind to shader state variables, built-in uniform variables that expose the graphics engine state to JXS shader programs. To create a `param` bound to a state variable in this way, use the `state` attribute.

```
<param name="itvmat" type="mat3" state="NORMAL_MATRIX" />
```

Available shader parameter state bindings are listed below.

Model View and Projection Matrices

WORLD_MATRIX	mat4	transforms into world coordinates, also known as model matrix
VIEW_MATRIX	mat4	transforms into current camera view, also known as eye
MODELVIEW_MATRIX	mat4	combined view and model (world) transform

PROJECTION_MATRIX	mat4	current camera projection transform
VIEW_PROJECTION_MATRIX	mat4	combined projection and view transform
PREV_VIEW_PROJECTION_MATRIX	mat4	previous frame VIEW_PROJECTION_MATRIX
MODELVIEW_PROJECTION_MATRIX	mat4	combined projection view and model transform
PREV_MODELVIEW_PROJECTION_MATRIX	mat4	previous frame MODELVIEW_PROJECTION_MATRIX
NORMAL_MATRIX	mat3	orients the normals in eye space
CAM_PROJECTION_MATRIX	mat4	provides the current rendering camera's projection matrix; in most cases is equivalent to PROJECTION_MATRIX except in cases where a full-screen quad is rendering, e.g. in a jit.gl.slab/pix (or jit.gl.pass).

Camera

CAMERA_POSITION	vec3	camera position in world space
CAMERA_DIRECTION	vec3	camera direction in world space
VIEWPORT	vec2	the pixel size of rendering window
INVERSE_VIEWPORT	vec2	the inverse of the viewport dims
NEAR_CLIP	float	camera near clipping distance
FAR_CLIP	float	camera far clipping distance
FAR_CORNER	vec3	far corner of the view frustrum

Light

Shaders used in Jitter can reference up to eight simultaneous lights. To create a `param` bound to the state of a particular light, use a `state` attribute with a value like LIGHT0_POSITION to bind to the position of the first light, LIGHT1_POSITION to bind to the position of the second light, and so on.

LIGHT_VIEWPROJ_MATRIX0-7	mat4	scene seen from the light position
LIGHT_RANGE0-7	float	distance reached by the light
LIGHT0-7_POSITION	vec3	position of the light
LIGHT0-7_DIRECTION	vec3	direction of the light
LIGHT0-7_AMBIENT	vec4	ambient light color
LIGHT0-7_DIFFUSE	vec4	diffuse light color
LIGHT0-7_SPECULAR	vec4	specular light color
LIGHT0-7_CUTOFF	float	the spotlight cutoff in degrees
LIGHT0-7_EXPONENT	float	the spotlight exponent defining dropoff from cone center

Material

AMBIENT	vec4	material ambient color
DIFFUSE	vec4	material diffuse color
SPECULAR	vec4	material specular color
EMISSION	vec4	material emission color
COLOR0	vec4	object color

Texture

Similar to lights, Jitter shaders can reference up to eight texture inputs. For each one, create a `param` bound to the state of a particular texture with an indexed key for `state`. So TEXTURE0_MATRIX will bind to the texture transform matrix of the first texture, TEXTURE1_MATRIX to bind the texture transform matrix of the second texture, and so on.

TEXTURE0-7_MATRIX	mat4	Texture tranform matrix for textures 0-7
TEXDIM0-7	vec2	Texture dimentions for textures 0-7

Time

Jitter provides time, frame and date based state parameters, useful for tasks like animating values and seeding random number generators in your shader program. `TIME` and `FRAME` will be specific to a particular shader program, whereas `GLOBAL_TIME`, `DELTA_TIME`, `CONTEXT_FRAME`, and `DATE` will be uniform for all shader programs running in a particular context.

TIME	float	time in seconds since program compilation
GLOBAL_TIME	float	time in seconds since context initialization
DELTA_TIME	float	time in seconds since previous frame
FRAME	int	frame count since program compilation
CONTEXT_FRAME	int	frame count since context initialization
DATE	vec4	year, month, day, time in seconds

Matrix transformations

You can apply a matrix transformation to any `param` with a *mat4* `type`, using the `transform` attribute.

```
<param name="itvmat" type="mat4" state="VIEW_MATRIX"
transform="INVERSE_TRANSPOSE" />
```

Recognized matrix transformations include `IDENTITY`, `TRANSPOSE`, `INVERSE` and `INVERSE_TRANSPOSE`.

Vertex attributes

Several built-in vertex attributes are available via the following state tags:

POSITION	vec3
TEXCOORD	vec2
NORMAL	vec3
TANGENT	vec3
BITANGENT	vec3
COLOR	vec4
VERTEX_ATTR	user-defined
VERTEX_ATTR0	user-defined
VERTEX_ATTR1	user-defined
VERTEX_ATTR2	user-defined
VERTEX_ATTR3	user-defined

The state tags VERTEX_ATTR and VERTEX_ATTR0 through VERTEX_ATTR4 are for custom vertex attributes. Define the `param` tag like so:

```
<param name="pvel" type="vec4" state="VERTEX_ATTR" />
```

and in the vertex program:

```
in vec4 pvel;
```

To set the values of custom vertex attributes from the patcher, send jit.gl.mesh the `vertex_attr_matrix` message followed by the name of a jit.matrix containing the attribute values. The example patcher custom.vertex.attribute demonstrates this.

Shaders and the `language` tag

After the description, textures, and parameters are declared, the `language` tag includes the shader definitions themselves.

```
  <!-- list of language implementations -->
  <language type="glsl" version="1.0">

  <!-- list of binding targets for shader parameters -->
  <bind param="myparam" program="vp"/>

  <!-- vertex and fragment programs -->
  <program name="vp" type="vertex" source="sh.passthru.xform.vp.glsl"/>
  <program name="fp" type="fragment">
    <![CDATA[
      // entry point
      void main()
      {
        gl_FragColor=vec4(0.5, 0.5, 0.5, 1.0);
      }
    ]]>
  </program>
</language>
```

The language tag includes the `type`, which will always be "glsl", as well as the `version`. Shader versions 1.5 or higher are treated as "modern" shaders, whereas anything lower will be treated as legacy and transformed automatically before getting passed to the graphics engine.

The `bind` tag

The `bind` tag binds declared parameters to variables in any one of the shader programs.

```
<jittershader>
  <param name="myparam1" type="vec3" default="3.0 4.0 5.0" />
  <param name="myparam2" type="float" default="0" />

  <language type="glsl" version="1.5">

  <bind param="myparam1" program="vp" />
  <bind param="myparam2" program="fp" />

  <program name="vp" type="vertex">
  <![CDATA[
    uniform vec3 myparam1;

    void main() {
      // ... the vertex program
    }
  ]]>
  </program>

  <program name="fp" type="fragment">
  <![CDATA[
    uniform float myparam2;

    void main() {
      // ... the fragment program
    }
  ]]>
  </program>

  </langauge>
</jittershader>
```

The `param` attribute of a `bind` tag identifies a parameter with that name. The `program` attribute directs that parameter to a particular program. Within the program, the parameter is bound to the variable with the same name.

The `type` of the `param` tag should match the type of the variable in the vertex, fragment, or geometry shader program. One exception is for texture parameters. The type of the `param` for a texture parameter should be `int`, and it should be bound to a variable with a "Sampler" type, like `Sampler2D` or `Sampler2DRect`.

The `include` tag

After binding parameters but before declaring shaders, you can use the `include` tag to include external *glsl* sources. These can be any files in the Max search path.

```
<bind param="FrontMaterialParameters" program="vp" />
<bind param="LightingParameters" program="vp" />

<include source="diffuse.glsl" program="vp" />
<include source="specular.glsl" program="vp" />

<program name="vp" type="vertex">
```

The `program` tag

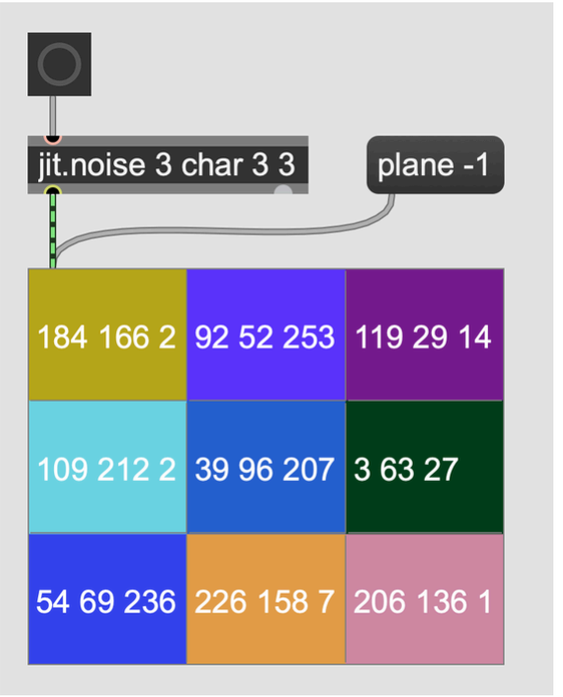
A `program` tag contains the actual shader program. The `name` of a `program` tag can be anything, and is used by the `bind` tag to bind external parameters to program variables. The `type` attribute of a `program` tag can be `vertex`, `fragment`, or `geometry`. To define the shader program itself, a program can use the `source` attribute to specify a `.glsl` file, or the contents of the `program` tag can be character data containing the glsl shader program itself.

```
<language type="glsl" version="1.0">
  <program name="vp" type="vertex" source="sh.passthru.xform.vp.glsl"/>
  <program name="fp" type="fragment">
    <![CDATA[
    void main()
    {
      gl_FragColor=vec4(0.5, 0.5, 0.5, 1.0);
    }
    ]]>
  </program>
</language>
```

Jitter Matrix

Quick Reference 309
Matrix processing 310
Creating a Matrix 310
Sending and Receiving from a jweb Browser 312
Coordinates 312
Frames, Colors, ARGB 313
Data Types 314
Getting and Setting Values 314
Adapting and Interpolating 317
Inspecting Matrices 319
Splitting and Recombining Matrices 322
Coercing 330

A **Matrix** holds multidimensional, numeric data. Each matrix has one or more dimensions, in addition to one or more planes. You can think of a matrix as a grid, where each cell of the grid holds a list of numbers. The length of the list in each cell is the number of planes.



A three-by-three matrix. You can see that, since this is a three-plane matrix, each cell contains three values as well.

When a matrix holds an image or a frame of video data, it's equivalent to a two-dimensional array, where each cell represents one pixel. The dimension of the matrix would correspond to the resolution of the image or frame, and there would be one plane for each channel of color data (typically red, green and blue), possibly with one additional channel for alpha. Every cell of a matrix must have the same data type, with the possible types being `char`, `long`, `float32`, and `float64`.

Quick Reference

Operation	Objects
Create a matrix	jit.matrix
Display a matrix	jit.window, jit.pwindow
Getting and setting values	jit.fill, jit.spill, jit.iter
Viewing matrix data	jit.cellblock, jit.print
Getting matrix descriptors	jit.matrixinfo, jit.fpsgui, getattr

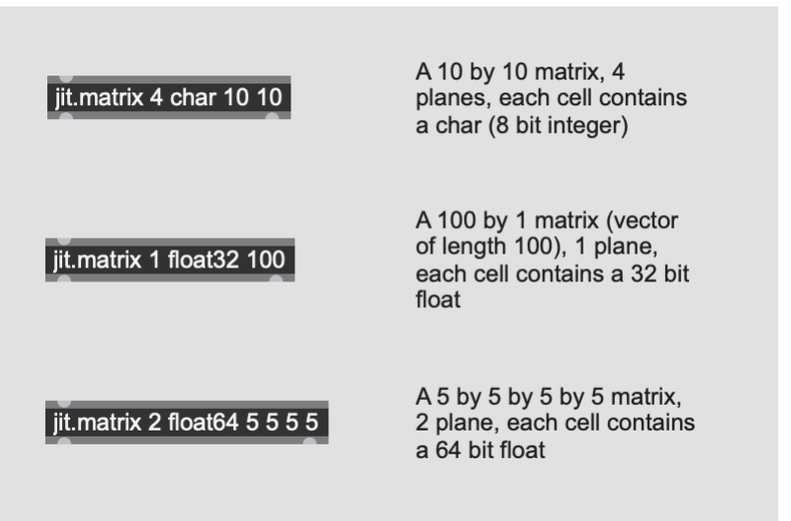
Splitting and combining matrices	jit.pack, jit.unpack, jit.submatrix, jit.concat, jit.dimmap
Re-interpreting matrix data	jit.coerce

Matrix processing

Matrix processing will always happen on the CPU, which has advantages and disadvantages. In general, graphics processing can happen on the CPU (Central Processing Unit) or the GPU (Graphics Processing Unit). CPU processing is sequential and generally slower, but can support certain operations that GPU processing cannot. GPU processing is highly parallel and often much faster, but can be more restricted. For video and graphics programming, Max has objects and data structures supporting both kinds of processing. For GPU-based processing, use the `jit.gl.*` and `jit.fx.*` families of objects, which make use of textures instead of matrices.

Creating a Matrix

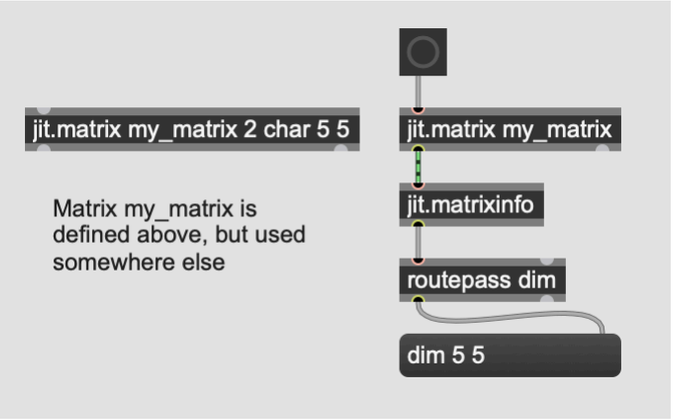
Create an empty matrix by creating a jit.matrix object. The object arguments determine its plane count, data type, and dimension size.



Named matrices

Like other named storage types, each matrix has a unique name. Objects that send matrices to each other send messages like `jit_matrix u12345678`, where `u12345678` is the name of the matrix.

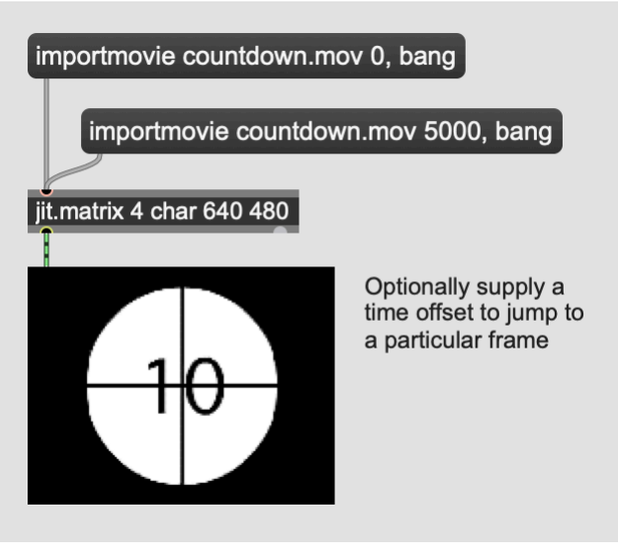
Since you can refer to matrices by name, It's possible to define a matrix in one part of your patcher and use it in another.



The matrix named my_matrix is defined in one part of the patcher, but is referred to by name and used in another part of the patcher.

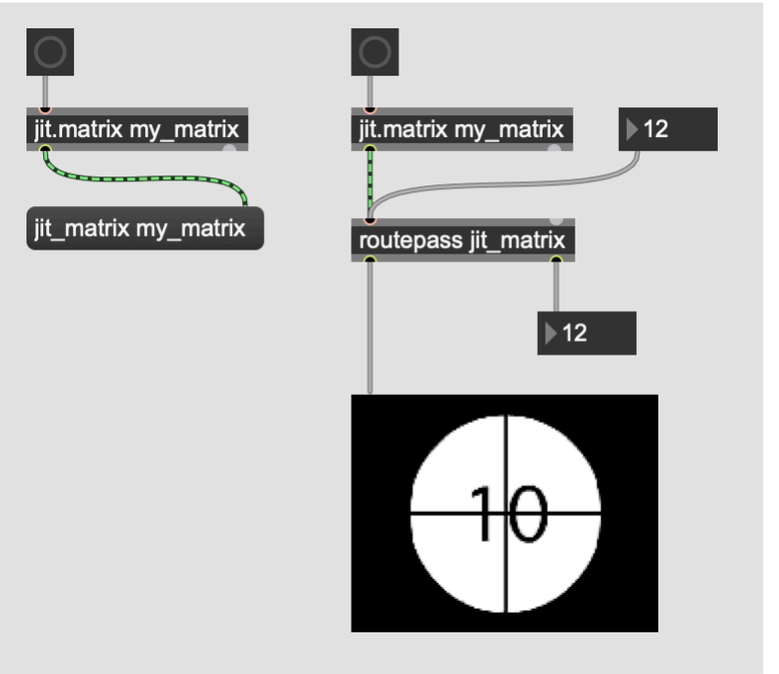
Loading from a video or image

You can fill a matrix with the contents of a video or image file with the `importmovie` message. If you're loading a video file, you can supply a time offset to load a particular frame. The jit.matrix object can only load a single image frame this way. To work with videos, use jit.movie, jit.playlist, or jit.matrixset.



Patcher cords

Objects that send out a Jitter matrix will style their outgoing patch cord with black and green stripes. This is only cosmetic, as these just carry regular Max messages of the form `jit_matrix u12345678`. However, the special styling helps to distinguish matrix patch cords from other patch cords.



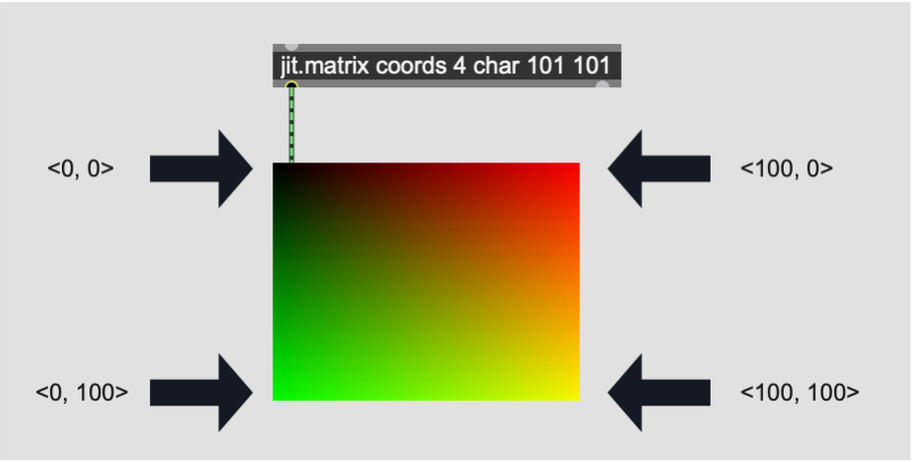
Even though they have special styling, patch cords carrying Jitter matrices are just carrying regular Max lists. You can work with them in the same way as any other Max message.

Sending and Receiving from a jweb Browser

You can read the display of a jweb webpage as a matrix using the jit.web object (or jit.gl.web for textures). You can also use JavaScript to send/receive matrices as well. See Web Browser and jweb for more.

Coordinates

Jitter matrixes index from the top left, meaning the origin `<0, 0>` is in the top left corner. If the matrix is 101 cells wide, then `<50, 50>` is in the center, `<100, 0>` is the top right, and `<0, 100>` is the bottom left.



Frames, Colors, ARGB

Matrices can hold arbitrary data, but Jitter objects that operate on image frames will interpret each plane as corresponding to a particular color. Most Jitter objects work in ARGB format, expecting the first plane of a matrix to represent alpha, the second the red color value, the third green, and the final plane blue. If the type of the matrix is `char` or `long`, then the maximum value of a cell, when interpreted as a color, is 255. For `float32` or `float64` types, the maximum value for a particular color chanel is 1.0.

Value	Type	Color
255 255 255	char	White
1.0 1.0 1.0	float32	White
255 0 0	char	Red
1.0 0 0	float32	Red
128 128 128	char	Medium gray
0.5 0.5 0.5	float32	Medium gray

> When viewing a three plane matrix in a jit.window or a jit.pwindow, Jitter will assume RGB format instead. You can change the color display mode of a jit.pwindow or a jit.window by setting the `@colormode` attribute

Data Types

Jitter matrices support four data types:

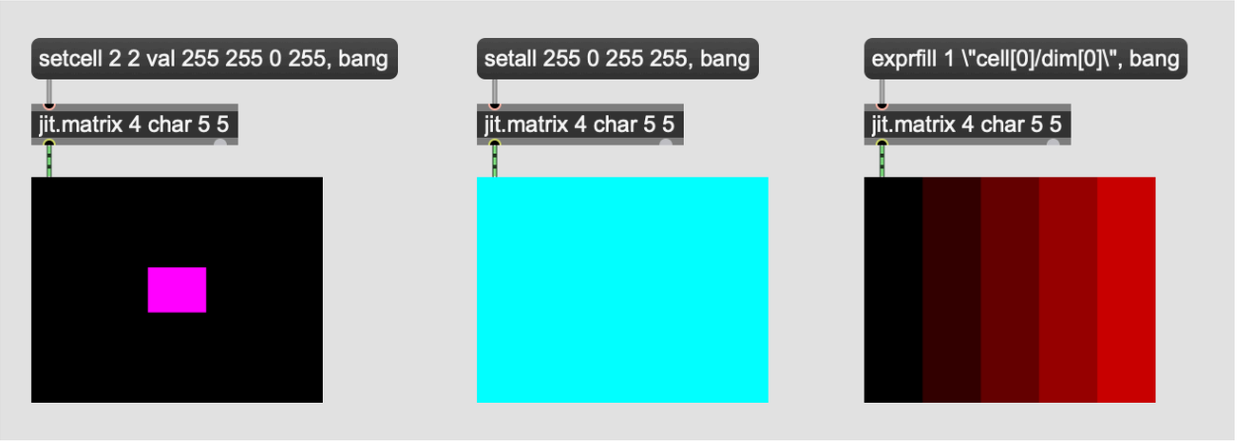
Type	Description
char	unsigned 8-bit integer (0 to 255)
long	signed 32-bit integer
float32	32-bit floating point number
float64	64-bit floating point number

> There aren't very many use cases for a `long` matrix. One of the few is when working with jit.repos, which would let you specify more than 255 pixel offsets.

Getting and Setting Values

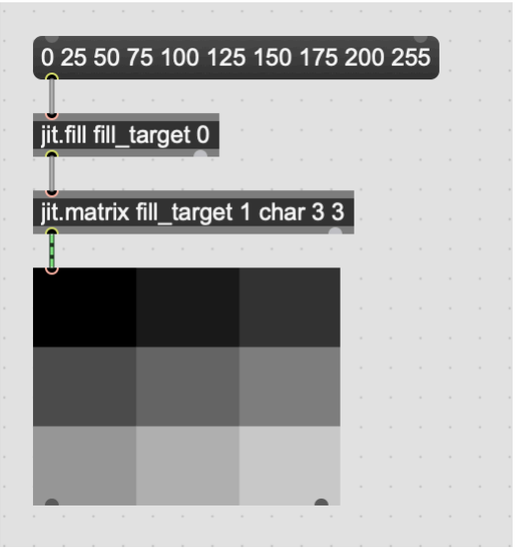
Fetch values from a matrix with the `getcell` message. For example, the message `getcell 9 19` would retrieve the value in the 10th column (since we're indexing from zero) and the 20th row. The jit.matrix object will send the contents of the cell as a list out of its right outlet. The length of the list will be the plane count of the matrix.

For setting the values of a matrix, the most common ways are with the `setcell`, `setall`, and `exprfill` messages. The `setcell` message simply sets the value of one cell to a given value. The message `setcell 1 2 val 255 255 0 255` would set the cell at coordinates `<1, 2>` to the value `255 255 0 255`. The `setall` message fills the whole matrix with a particular value, so the message `setall 0 255 0 255` would set every cell of the matrix to the value `0 255 0 255`. Finally, the `exprfill` message lets you fill a matrix parametrically, according to a function that will be evaluated for each cell of the matrix. For more on the expression language used here, see Jitter expr.

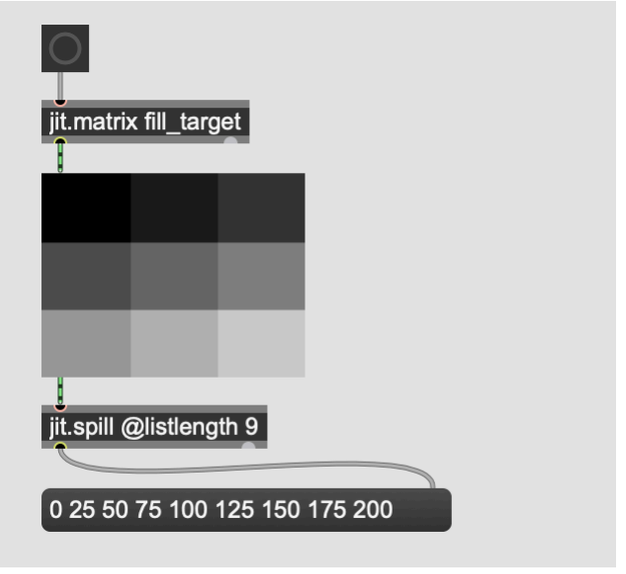


Helpful objects for getting and setting values

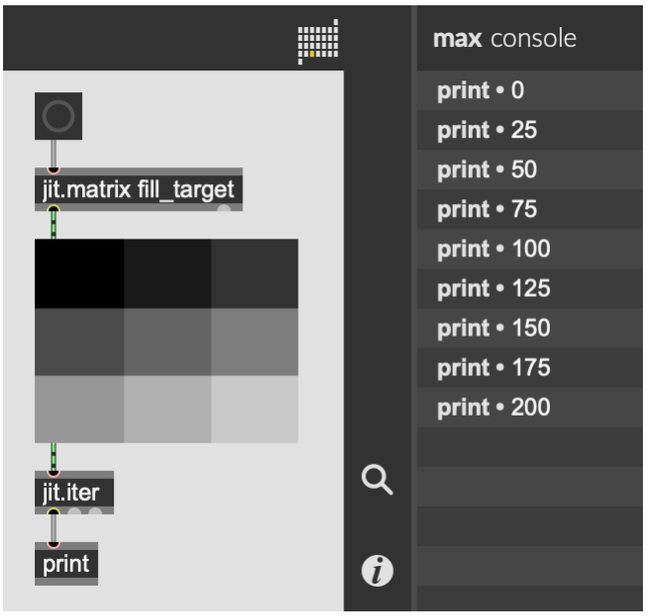
A handful of jitter objects support setting and retrieving values from a matrix using lists. The jit.fill object lets you "fill" a matrix using a list. Note that jit.fill references a matrix by name, and doesn't output a matrix itself.



Going in the other direction, jit.spill can output a matrix as a list. The attributes `@plane` and `@offset` determine the plane and offset into the matrix. The `@listlength` attribute lets you set the length of the output list.

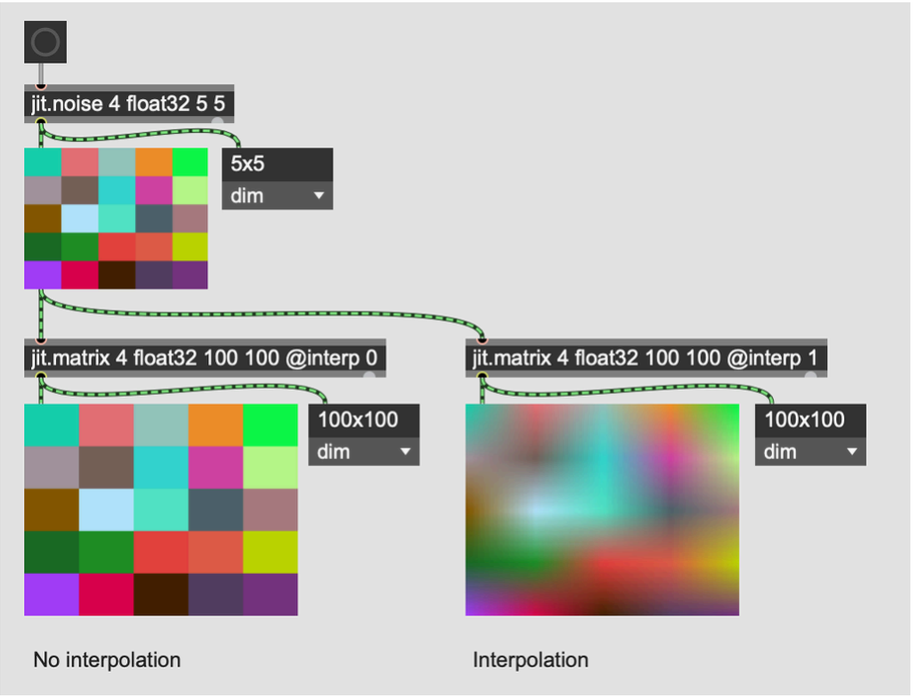


One more object that serves a similar function is the object jit.iter. This object will output each each cell of the input matrix one at a time, along with the coordinates of each cell.



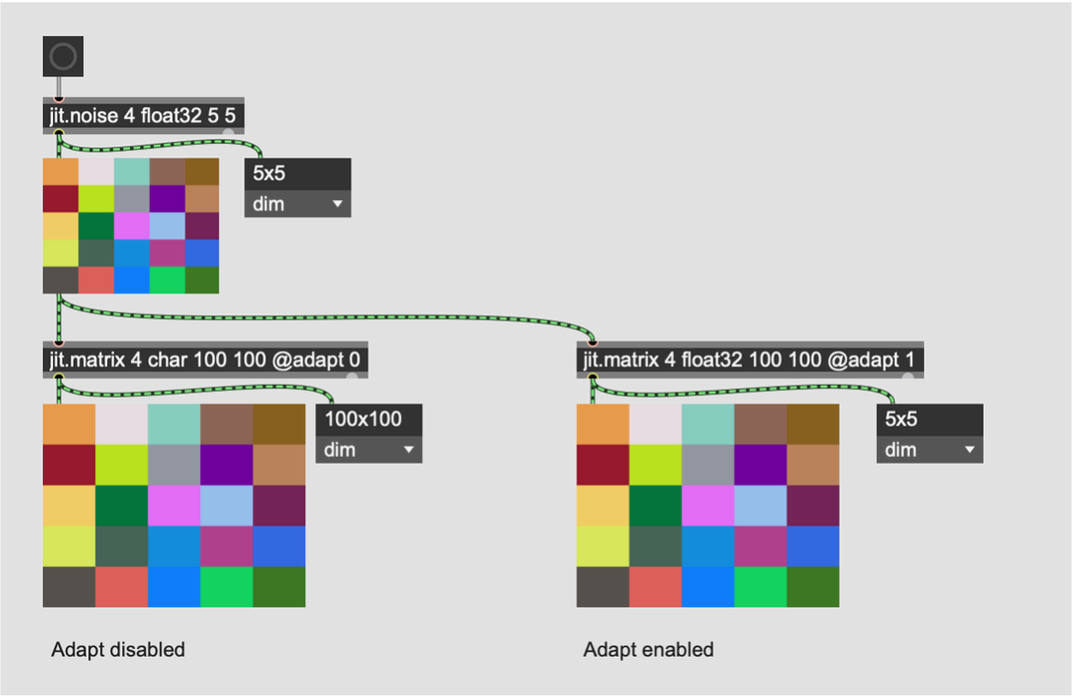
Adapting and Interpolating

If a jit.matrix object receives a matrix as input that does not match the dimensions of the internal matrix, the object can adapt to or interpolate the incoming data. Interpolating means creating in-between values to fill in gaps where needed. If a 100 by 100 cell jit.matrix object receives a 10 by 10 cell input, it can interpolate to fill in the missing values. Interpolation is off by default. Use the `@interp` attribute to control interpolation.



When enabled, jit.matrix will smoothly interpolate when asked to scale up an input matrix

By default, the arguments to a jit.matrix object determine the plane count, type, and dimensions of the internal matrix. If you enable the `@adapt` attribute, then jit.matrix will change its properties to match the dimensions of the incoming matrix, including plane count, dimensions, and type. An empty jit.matrix object with no arguments will have `@adapt` enabled by default.

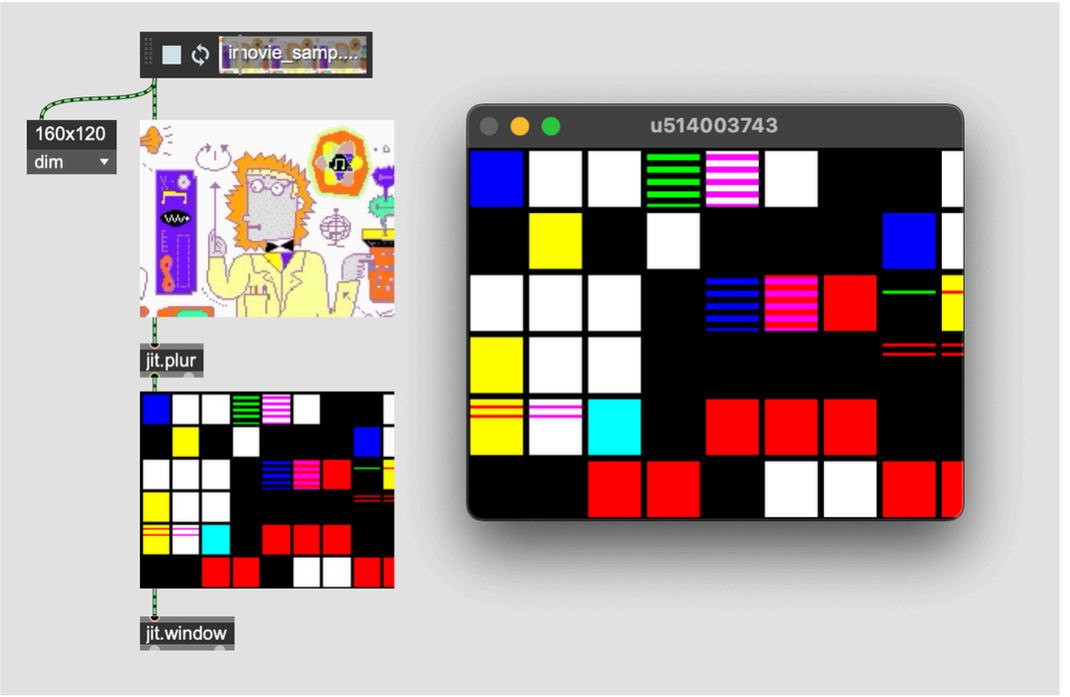


If @adapt is enabled, a jit.matrix will update its dimensions to match an input matrix.

Inspecting Matrices

Viewing as images

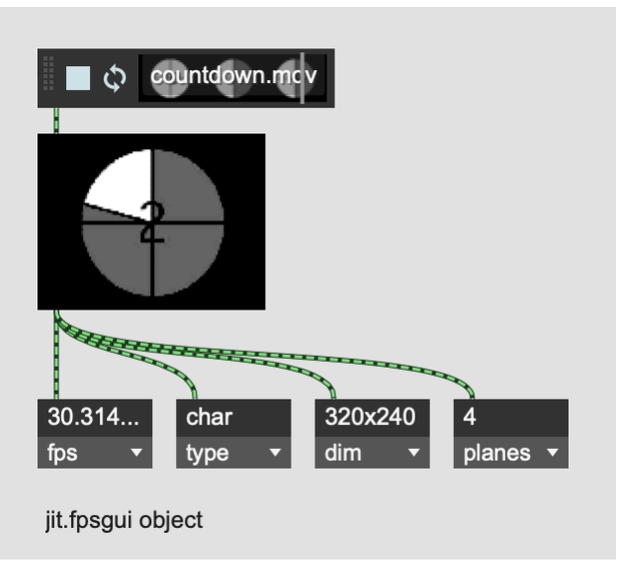
The jit.window and jit.pwindow objects will display whatever matrix they receive as an image. So, jit.pwindow is a great way to monitor a video stream in a running patcher. The jit.window object is often used as a final render destination, where a series of Jitter processes modify a matrix, and then a jit.window presents the final result.



jit.pwindow and jit.window are useful for viewing matrices, and they can also be used to view textures.

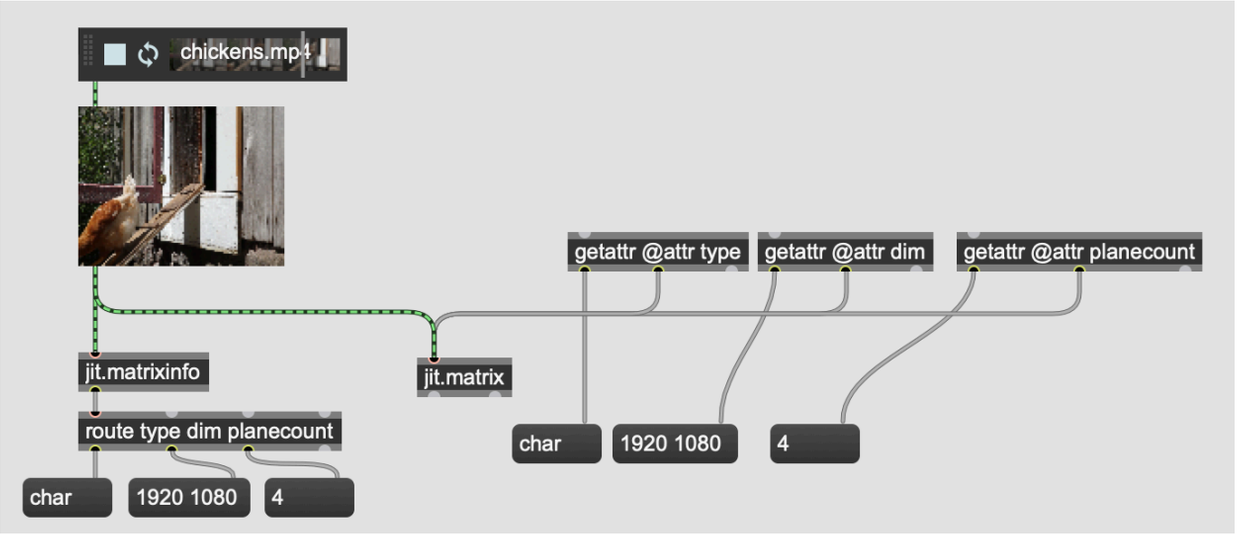
Matrix descriptors

Several objects can get information about the dimension, plane count, type, or other descriptors of a particular matrix. The jit.fpsgui object can be configured to display any of these descriptors, in addition to the frame rate of a stream of matrices.



The jit.fpsgui object shows useful info about a matrix

If you want to retrieve these same matrix descriptors as Max messages, for further downstream processing, use either the jit.matrixinfo object, or use the getattr object to fetch these values as attributes from a matrix. Both methods are equally efficient and idiomatic, the main difference is that jit.matrixinfo processes an incoming matrix, while getattr must attach to a jit.matrix object.

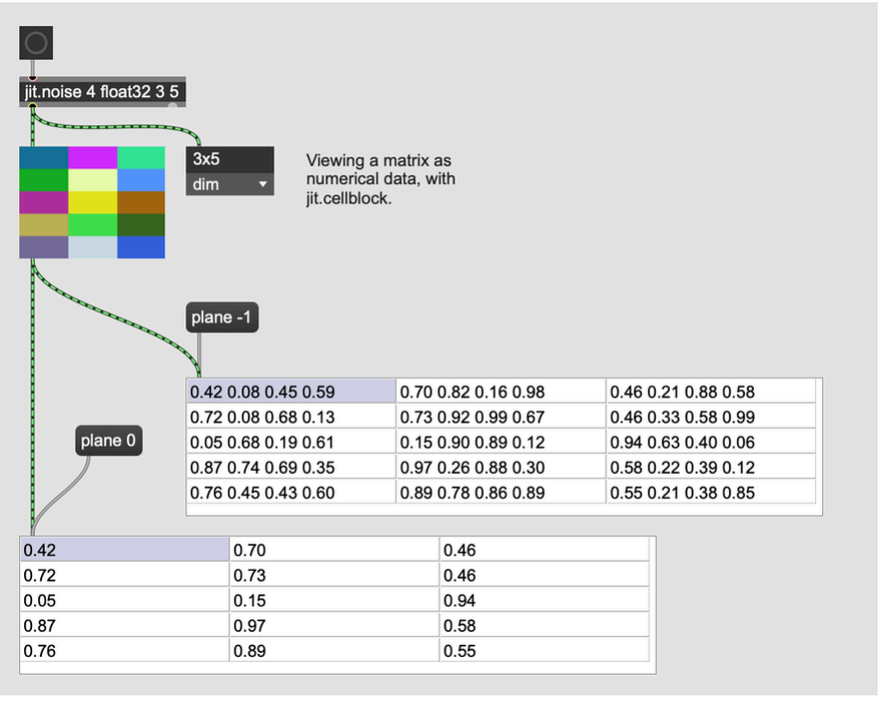


The jit.matrixinfo object reports on the descriptors of an incoming matrix. These same descriptors are available as attributes, so the getattr object can also retrieve them.

Matrix contents

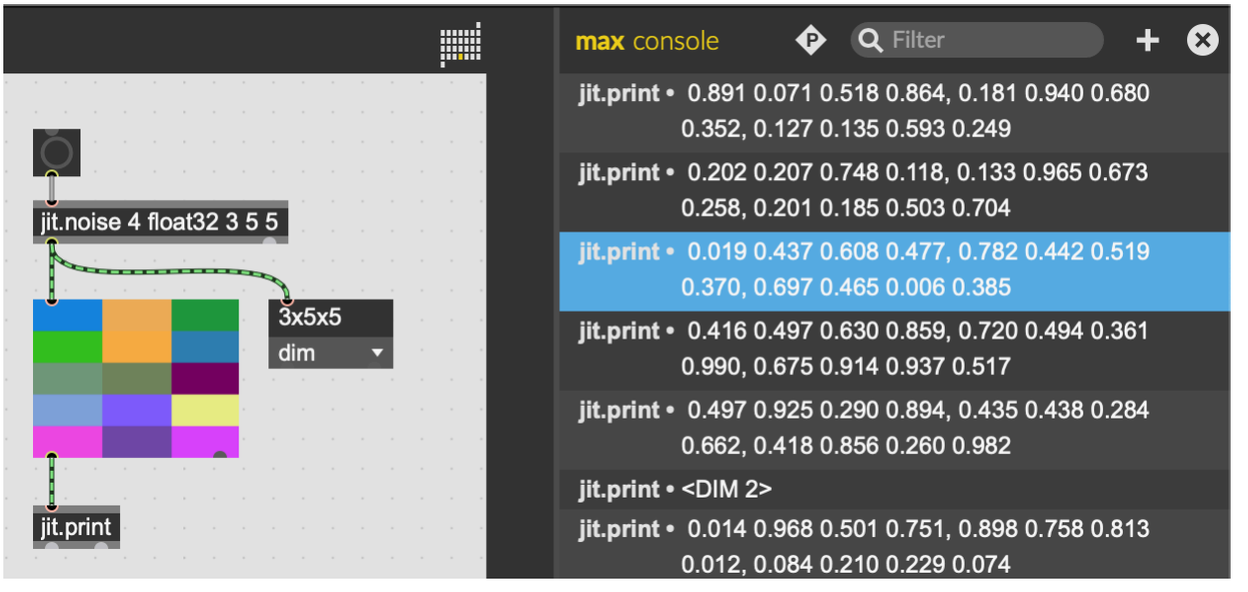
For viewing the contents of a jit.matrix object as numerical data, use the jit.cellblock object. It can display a one or two dimensional matrix in a "spreadsheet" format. You can choose which plane of the matrix to view by sending the message `plane $1`, where `$1` is the plane of the matrix that you want to display. You can also send the message `plane -1` to view all planes of the matrix simultaneously, with each cell containing a list.

> The jit.cellblock object can only display matrices with less than two dimensions. For larger matrices, first split the matrix with jit.submatrix, pass the matrix through jit.dimmap to cut out redundant dimensions, then pass the result through a jit.matrix to copy the matrix reference to a format that jit.cellblock can use. See Splitting by dimension for more details.



Use the jit.cellblock object to view a matrix as numerical cells

Finally, the jit.print object can print the entire contents of any matrix of any number of dimensions and planes.

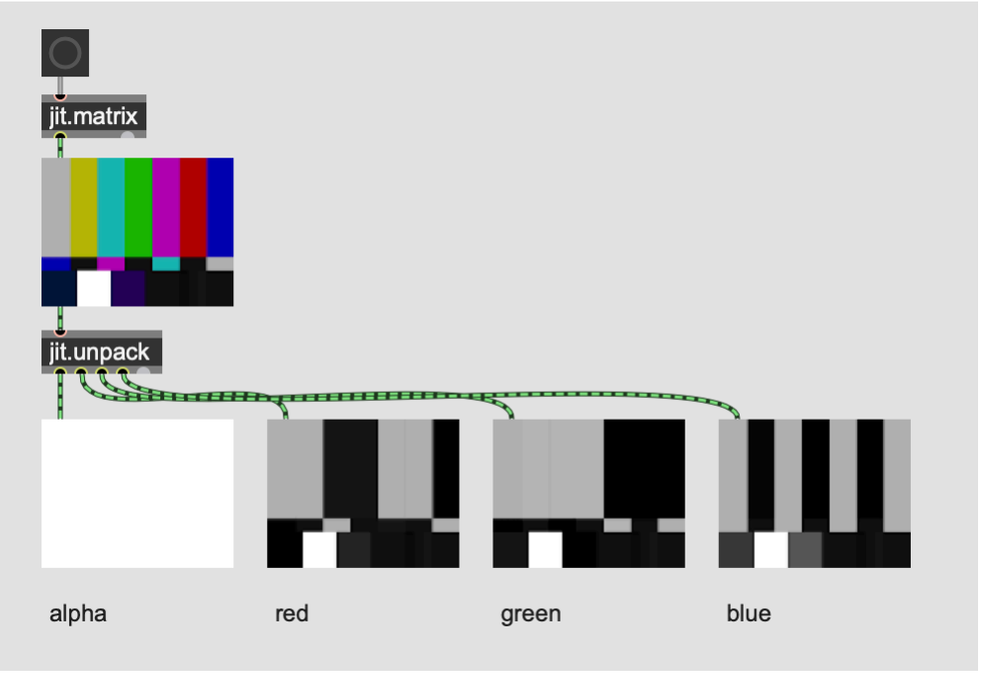


Splitting and Recombining Matrices

It's possible to split and recombine matrices across planes or across dimensions. Use jit.unpack and jit.pack to recombine matrices by plane, and use jit.submatrix and jit.concat to recombine by dimension.

Splitting and remapping planes

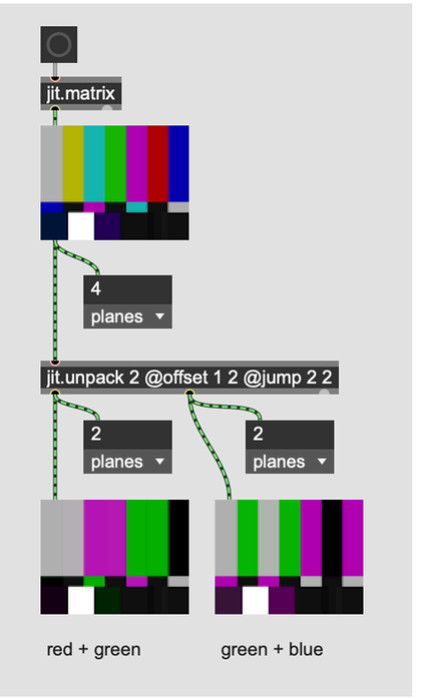
By default, the jit.unpack object will split a 4-plane matrix into four single-plane matrices, but you can pass an argument to split into a different number of single planes.



The default configuration of jit.unpack splits a 4-plane ARGB matrix into four single-plane matrices.

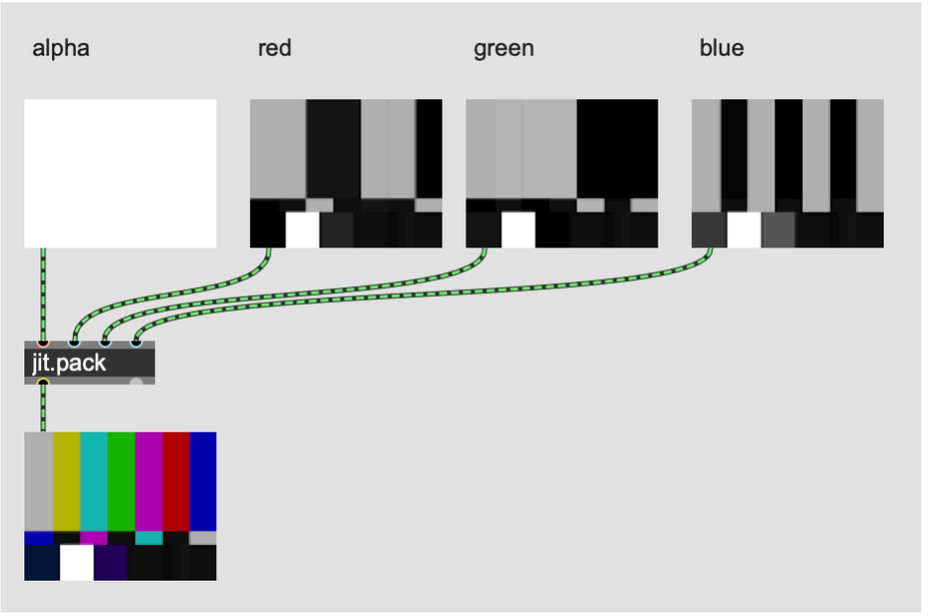
If you want to split a multiple-plane matrix into other multiple-plane matrices, use the `@offset` and `@jump` attributes to configure jit.unpack. Arguments following `@offset` indicate, for each outlet, which plane in the input matrix to start reading from. Argments following `@jump` indicate, for each outlet, how many planes to read from the input.

Setting	Outlet 1	Outlet 2	Outlet 3
`jit.unpack 2 @offset 0 1 @jump 2 2`	two-plane matrix, from planes 0 and 1 of the input matrix	two-plane matrix, from planes 1 and 2 of the input matrix	none
`jit.unpack 2 @offset 0 3 @jump 3 1`	three-plane matrix, from planes 0, 1, and 2 of the input matrix	one-plane matrix, from plane 3 of the input matrix	none
`jit.unpack 3 @offset 1 2 3 @jump 1 1 1`	one-plane matrix from plane 1	one-plane matrix from plane 2	one-plane matrix from plane 3



With @offset and @jump, you can configure jit.unpack to output multiplane matrices.

The jit.pack object combines input matrices into a single multi-plane matrix. Like jit.unpack, it by default expects four input matrices, and the first argument determines the number of inlets.



jit.pack combines mutiple input matrices into a single multi-plane matrix

Using `@offset` and `@jump`, you can combine several multi-plane matrices together. The `@offset` attribute controls the offset into each input matrix (at each inlet), and the `@jump` attribute determines how many planes to pull from each input matrix. The final plane count will the the sum of all the arguments to `@jump`.

Setting	Output

`jit.pack 2`
`@offset 0 0`
`@jump 2 2`
	four-plane matrix, with two planes from the first inlet and two planes from the second inlet

`jit.pack 2`
`@offset 0 0`
`@jump 1 3`
	four-plane matrix, with one plane from the first inlet and two planes from the second inlet

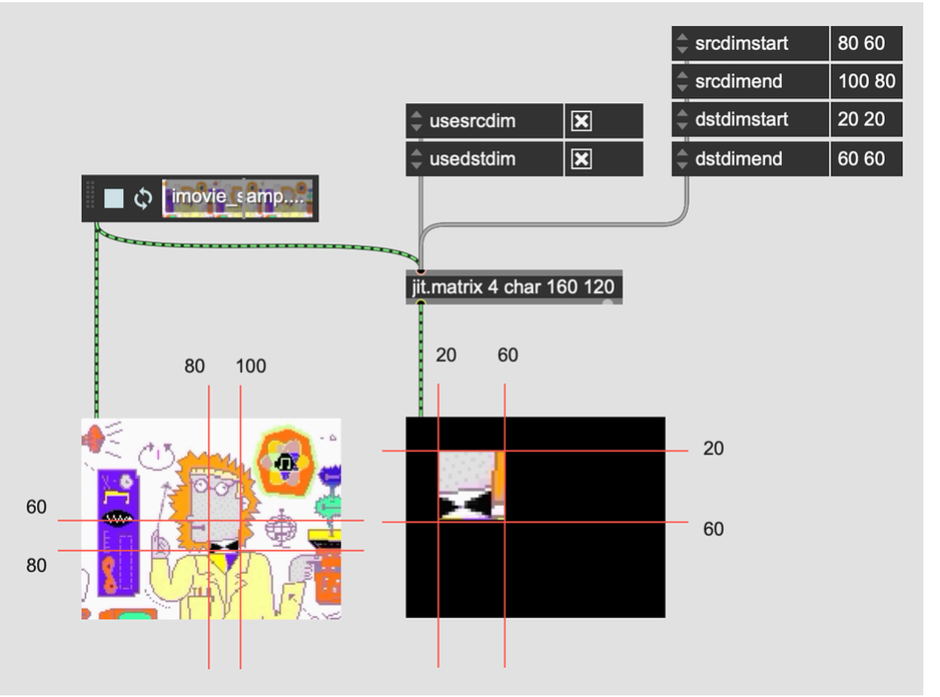
`jit.pack 2`
`@offset 0 2`
`@jump 2 2`
	four-plane matrix, with two planes from the first inlet and two from the second inlet. The first two planes of the second input matrix are skipped. This would combine the alpha and red channels of the first input with the green and blue channels of the second.

The jit.matrix object can also remap planes directly, without using any other objects. The `@planemap` attribute controls the mapping between input and output planes.

Setting	Result
`jit.matrix 4 @planemap 0 1 2 3`	Map the first plane of the input (at index zero) to the first plane of the output, the second plane of the input to the second plane of the output, and so on. In other words, leave the input unchanged.
`jit.matrix 4 @planemap 0 0 0 0`	Map the first plane of the input to each of the four output channels. This makes a 4-plane matrix from just the first plane of the input.
`jit.matrix 4 @planemap 1 2 3 0`	Shift input planes over by one, effectively converting ARGB format to RGBA format.

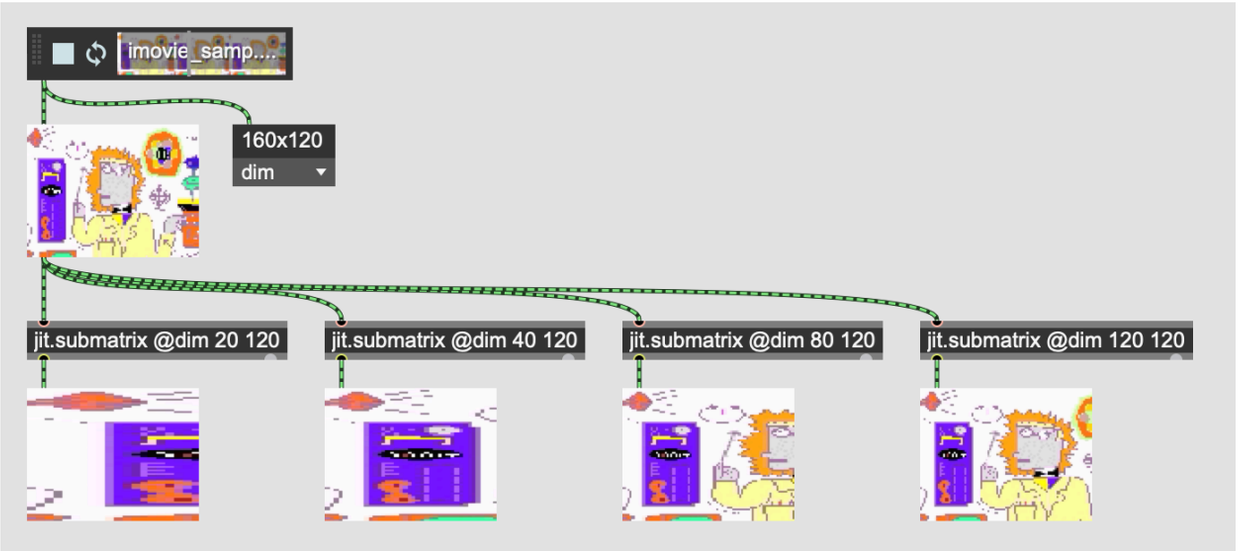
Splitting and remapping dimensions

One way to split an input matrix is with the `@srcdimstart`, `@srcdimend`, `@dstdimstart`, and `@dstdimend` attributes. When a jit.matrix object receives an input matrix, it will look at these attributes to determine how the input is copied into the internal matrix. It's important to note that these attributes will have no effect unless the `@usesrcdim` and `@usedstdim` attributes are enabled.



The source and destination dimensions can control how the input matrix is mapped to the output. This example also shows how if the start and end areas are different in size, the input will grow or shrink as it's mapped to the output.

The jit.submatrix object will let you reference just a portion of the input matrix. The `@dim` attribute determines the size of the portion in each dimension, corresponding to width and height for a two-dimensional matrix. The `@offset` attribute adjusts the offset of the portion, which you could also think of as the x- and y-coordinate of the submatrix.

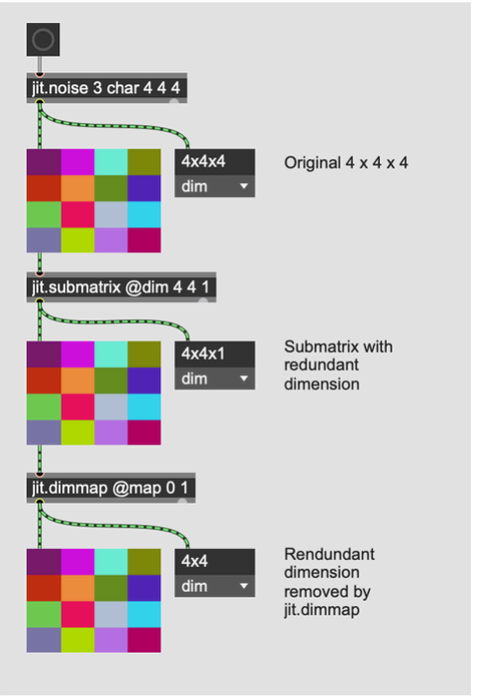


The @dim attribute determines the size of the submatrix region.



The @offset attribute can shift the submatrix region.

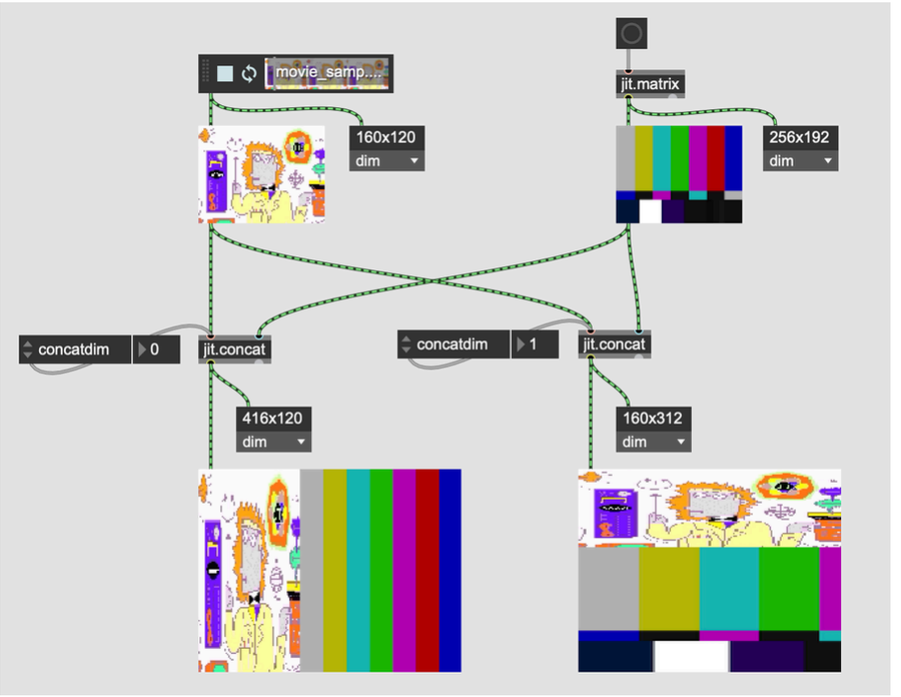
The output of jit.submatrix will have smaller dimensions than the input, but it will never reduce the number of dimensions, even a redundant dimension of size 1. The jit.dimmap object can remove whole dimensions from an input matrix, or insert a redundant dimension. It can also remap and invert dimensions, making it useful for transposing and reversing a matrix.



jit.submatrix will never change the number of dimensions, but jit.dimmap can remove and remap dimensions arbitrarily

Setting	Output
`jit.dimmap @map 1 0`	Map input dimension 1 to output dimension 0, effectively transposing the input matrix.
`jit.dimmap @map 0 2`	Map input dimension 0 to output dimension 0, and input dimension 2 to output dimension 1. This will remove dimension 1 from the input, resulting in a two-dimensional output matrix from a three-dimensional input.
`jit.dimmap @map 0 -1 1`	Insert a redundant, size-one dimension between the first and second input dimensions.
`jit.dimmap @map 0 1 @invert 1 0`	Leave the input dimensions mapped normally, but flip the first dimension, effectively mirroring an input image from left to right.

To combine or recombine two matrices, use the jit.concat object. The `@concatdim` attribute will determine the dimension along which the two matrices will be concatenated. They do not need to have the same size, although there may be empty space in the resulting concatenated matrix, or some data might be lost if the edges of the two matrices don't line up.



Concatenating two matrices. The @concatdim attribute determines the dimension along with the two matrices are combined.

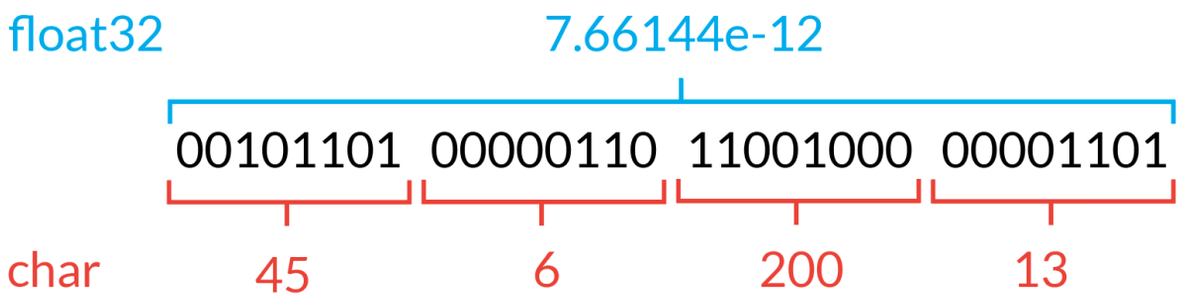
More ways to split

There are still other matrix manipulation objects that may be helpful when splitting and recombining matrices:

Object	Description
jit.split	Split a matrix into two
jit.scissors	Cut up a matrix into equal parts
jit.glue	Recombine a matrix from multiple parts
jit.multiplex	Multiplex (interleave) two matrices into one
jit.demultiplex	Demultiplex (deinterleave) one matrix into two

Coercing

The jit.coerce object can be used to change the interpretation of the contents of a matrix, without copying or changing those contents. For example, you could treat a `long` matrix as if it were a `float32` matrix, or you could treat a 4-plane `char` matrix as a 1-plane `float32` matrix. This is similar to a `cast` operation in many programming languages.

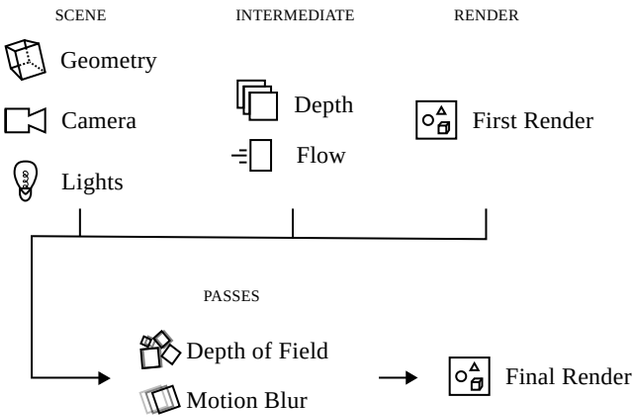


The jit.coerce can change the way the underlying bits of a matrix are interpreted

Render Passes

Using jit.gl.pass 332
The JXP File Format 333

Rendering means taking the elements of a scene, including the lighting, camera position, and the geometry of the objects in the scene, and using those to compute an image. After the initial pass, we can perform additional render passes to create postprocessing effects. These passes can use the initial image output, along with intermediary structures computed during the previous pass—for example the depth buffer—to produce another image. Depth of field is a classic example of a postprocessing effect, using the depth buffer to apply a blur proportional to the distance from the camera. Multiple passes can be chained together to produce complex effects.



Typical graphics processing rendering pipeline

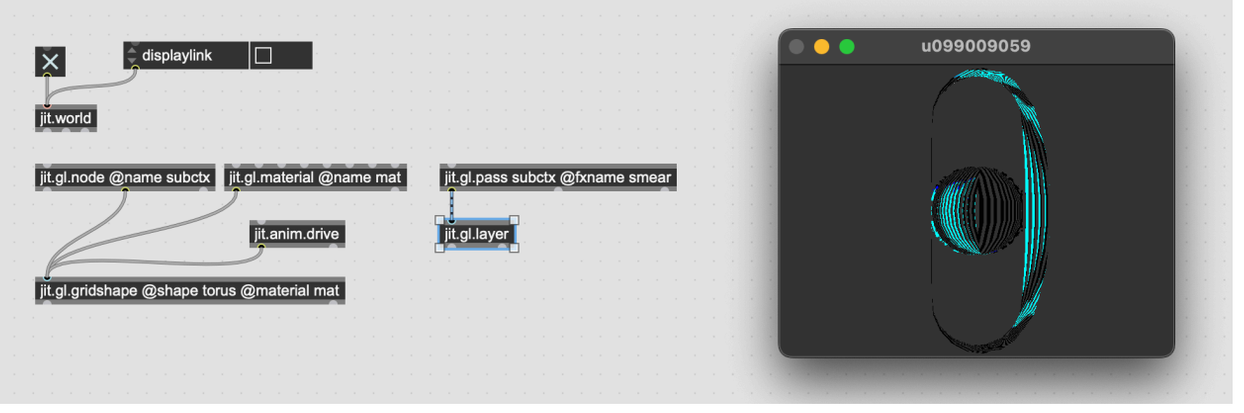
Using jit.gl.pass

The jit.gl.pass object provides a high-level, text-based way to describe a render pass. Working with a pass description file (a .jxp file), you can combine Jitter shaders defined in text (.jxs files) with Gen-based Jitter shaders (.genjit files) to create a final, complex shader with multiple stages.

Because a jit.gl.pass object works with intermediate render targets like the depth buffer, it works differently from objects like jit.gl.slab or jit.gl.pix, which can work as simple video effects. The jit.gl.pass object always works with a jit.gl.node object, even if it's the internal jit.gl.node object in a jit.world object.

> To work with a jit.gl.pass, geometry must have a material attached to it, for example by attaching a jit.gl.material or a jit.gl.pbr object.

The typical use pattern for jit.gl.pass is to bind jit.gl.pass to a jit.gl.node object with the same name. Multiple jit.gl.pass objects can be connected together to create complex effects. Finally, the texture output of the last jit.gl.pass object is connected to a jit.gl.layer object to be displayed in the root rendering context.



It's typical to use jit.gl.pass with jit.gl.node and jit.gl.layer.

> When a jit.gl.pass object is connected to a jit.gl.node, the `@capture` attribute is controlled by the jit.gl.pass object. It will be set automatically to accommodate for the render targets requested by all jit.gl.pass effects bound to that node.

The JXP File Format

To determine its behavior, a jit.gl.pass object loads a Jitter Pass file: a specially-formatted XML file that defines one or more render passes. Every JXP file has a `<jitterpass>` node as its root.

```
<jitterpass>
  <!-- Pass definitions go here -->
</jitterpass>
```

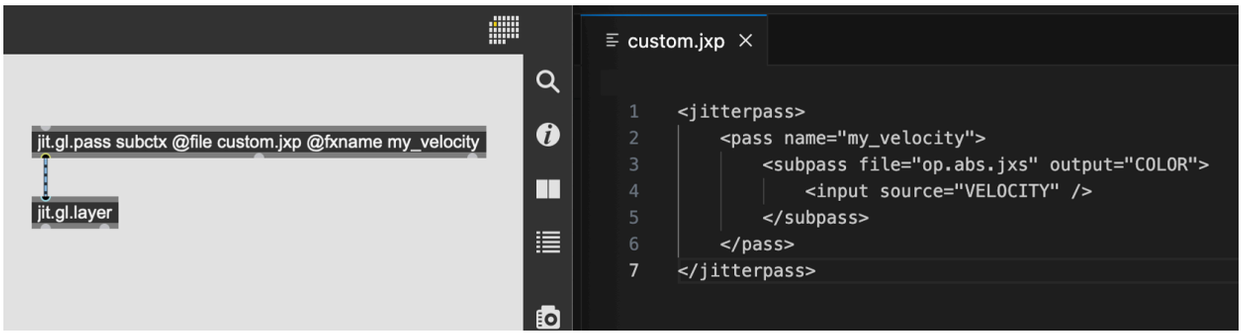
> Max includes several examples of common post-processing effects, implemented in the JXP format. Start with motionblur and look for other examples in the same folder.

The <pass> tag

The `<jitterpass>` tag can contain multiple `<pass>` tags, each of which defines one pass.

```
<jitterpass>
  <pass name="passname">
    <!-- Pass definition here -->
  </pass>
</jitterpass>
```

Each `<pass>` must have a name attribute, which is the identifier that a jit.gl.pass will use with the `@fxname` attribute to load the pass definition. For built-in pass definitions, the `@fxname` attribute is sufficient to select the pass. To load a custom `pass` definition, use the `@file` attribute to load the custom pass file, followed by `@fxname` to load the specific pass.



jit.gl.pass loading a custom JXP file

The <bind> tag

At the top of a `<pass>` tag, you can can use the `<bind>` tag to create a binding between Max attributes and shader parameters. Use the `name` attribute to define the name of the attribute as it will appear in Max. The `param` attribute will specify which shader parameter to bind to. The `param` attribute will bind to the first shader parameter among `<subpass>` tags with a matching name.



The <bind> tag lets you bind shader parameters to Max attributes. A custom attribute will be created for each <bind> tag.

The <inputs> tag

A `<pass>` tag can optionally contain an `<inputs>` tag containing one or more `<input>` tags specifying input source properties (see source input). The properties are exposed via jit.gl.texture attributes on the underlying render targets of the bound jit.gl.node context. A common use case might be inverting the depth buffer clear value. Render target `erase_color` defaults to 0 0 0 0. Since the alpha component determines the depth, this default implies a depth clear value 0. To invert, specify a 1 for the alpha component of `erase_color` in the `<input>` tag for `source` NORMALS. The code below demonstrates this, and specifies `type` and `erase_color` per target.

```
<jitterpass>
  <pass name="passname">
    <inputs>
      <input source="COLOR" type="char" erase_color="0 0 0 1" />
      <!-- inverted depth clear value -->
      <input source="NORMALS" type="float16" erase_color="0 0 0 1" />
    </inputs>
    <!-- subpasses -->
  </pass>
</jitterpass>
```

Important to note, input sources are shared by all pass effects bound to a particular jit.gl.node context. As a concequence any `<input>` tag attribute defined here will change that attribute for all active passes in the context.

The <subpass> tag

Each `<pass>` tag contains one or more `<subpass>` tags. These `<subpass>` tags define which shader program to load, as well as how that shader should connect to the various input sources available (see subpass sources). To load a shader, set the `file` attribute to load a Jitter shader (a `.jxs` file), or set the `gen` attribute to load a jit.gl.pix shader (a `.genjit` file).

```
<jitterpass>
  <pass name="passname">
    <subpass file="cf.radialblur.jxs">
      <!-- subpass inputs -->
    </subpass>
    <subpass gen="kaleido.genjit">
      <!-- subpass inputs -->
    </subpass>
  </pass>
</jitterpass>
```

A `<subpass>` tag can also have a name, allowing it to be referred to as an input by other subpasses. Finally, a subpass can also use the `dimscale` attribute to scale the size of any

dimension of the output texture, and the `rect` attribute to adjust or invert the subpass texture coordinates. See the object reference for jit.gl.pix for more.

```
<jitterpass>
  <pass name="passname">
    <!-- a named subpass can be referred to by other subpasses -->
    <subpass file="cf.radialblur.jxs" name="subpass_name">
    </subpass>

    <!-- here dimscale will downsample (shrink) the output texture -->
    <subpass file="cf.radialblur.jxs" dimscale="0.5 0.5">
    </subpass>

    <!-- rect defines normalized texture coordinates, left-bottom-right-
top -->
    <!-- used in this way, we could mirror the texture left to right-->
    <subpass file="cf.radialblur.jxs" rect="1 0 0 1">
    </subpass>

    <!-- >
  </pass>
</jitterpass>
```

Subpass sources

The number of texture inputs to a subpass will depend on the number of texture inputs of its loaded shader. You can connect each input to a different source using the `<input>` child tag. The input to a subpass can be a named texture, another named subpass, or one of several special sources.

Named texture input

Use the `name` attribute to specify a named texture as input.

```
<jitterpass>
  <pass name="passname">
    <subpass file="cf.radialblur.jxs">
      <input name="named_texture" />
    </subpass>
  </pass>
</jitterpass>
```

Source input

The `source` tag can get input from any of several special sources.

```
<jitterpass>
  <pass name="passname">
    <subpass file="hdr.bloom.jxs">
      <input source="VELOCITY" />
    </subpass>
  </pass>
</jitterpass>
```

Name	Description
COLOR	RGBA color render target
NORMALS	normals in RBG, depth buffer in A
VELOCITY	horizontal velocity in R, vertical velocity in G
PREVIOUS	the preceeding subpass output
HISTORY	the previous output (output from the previous frame) of the entire `<pass>`
TEXTURE0...N	any of the textures, input as a list, to jit.gl.pass. If three texture names are bound to the `@texture` attribute of jit.gl.pass, refer to the first one with TEXTURE0, the second with TEXTURE1, and the third with TEXTURE2.

SUBPASS0...N	any subpass output of the current `<pass>`. The first `<subpass>` is SUBPASS0, the second is SUBPASS1, and so on.
ALBEDO	the albedo material component (non-jit.gl.pbr materials use diffuse).
ROUGHMETAL	material roughness in R and metalness G (non-jit.gl.pbr materials use inverted shininess in R and shininess in G).
ENVIRONMENT	the environment cubemap texture of the active jit.gl.environment. Specifying will disable environment input on bound materials.

> Pass effects that request NORMALS, VELOCITY, ALBEDO, ROUGHMETAL, or ENVIRONMENT as input sources require jit.gl.material or jit.gl.pbr generated materials.

Subpass input

Use the `subpass` attribute to specify another subpass to use as input.

```
<jitterpass>
  <pass name="passname">
    <subpass file="cf.laplace.jxs" name="laplace">
      <input source="COLOR" />
    </subpass>
    <subpass file="hdr.bloom.jxs">
      <input subpass="laplace" />
    </subpass>
  </pass>
</jitterpass>
```

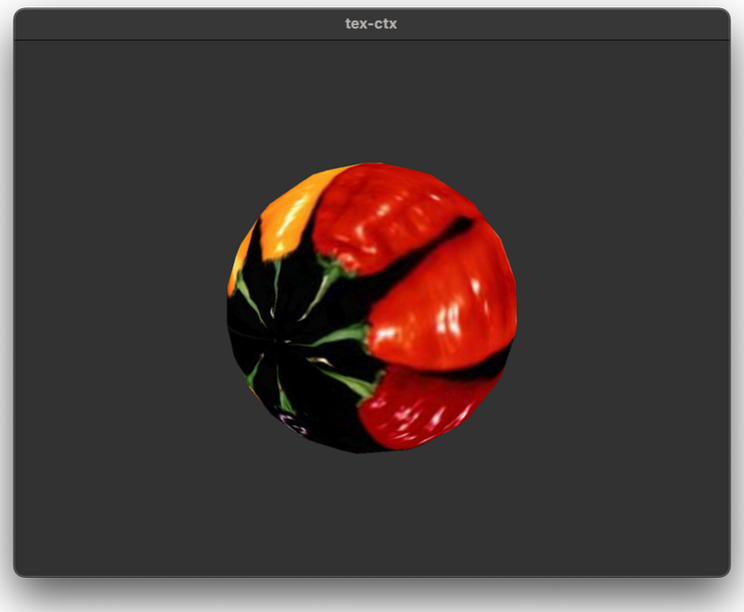
The output index of a multi-output `subpass` input type is specified using an "output" attribute and 0-based indexing. The previous frame of subpass input types can be requested using the history keyword and setting it to 1. For example to set second output of the previous frame as the input.

```
<input subpass="mysubpass" output="1" history="1" />
```

Textures

Creating a Texture 342
Texture output 344
Converting to a Matrix 345
Procedural Textures 345
3D Textures 346

A **Texture** is a collection of data, organized as an array of pixel data and stored in texture memory on the GPU. Textures commonly hold image data—they have the name "texture" because they're often applied to the surface of a 3D model to change its appearance. However, a texture can hold any kind of data that can be expressed as an array of numbers.

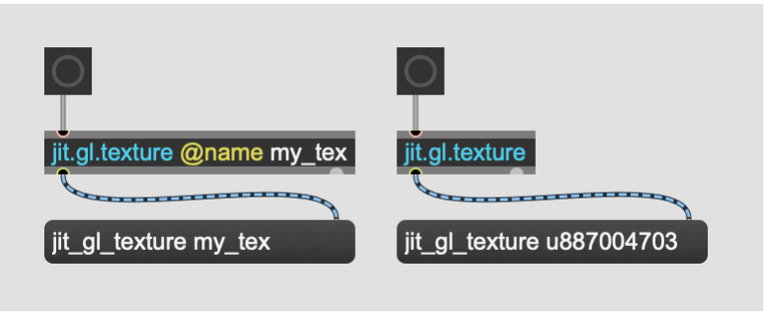


An image of chili peppers, mapped as a texture onto the surface of a sphere

Because textures are managed by the GPU, they can be processed much faster than the equivalent matrix, which is managed by the CPU. For tasks like distorting or manipulating an image, it's almost always more efficient to work with a texture than a matrix.

Creating a Texture

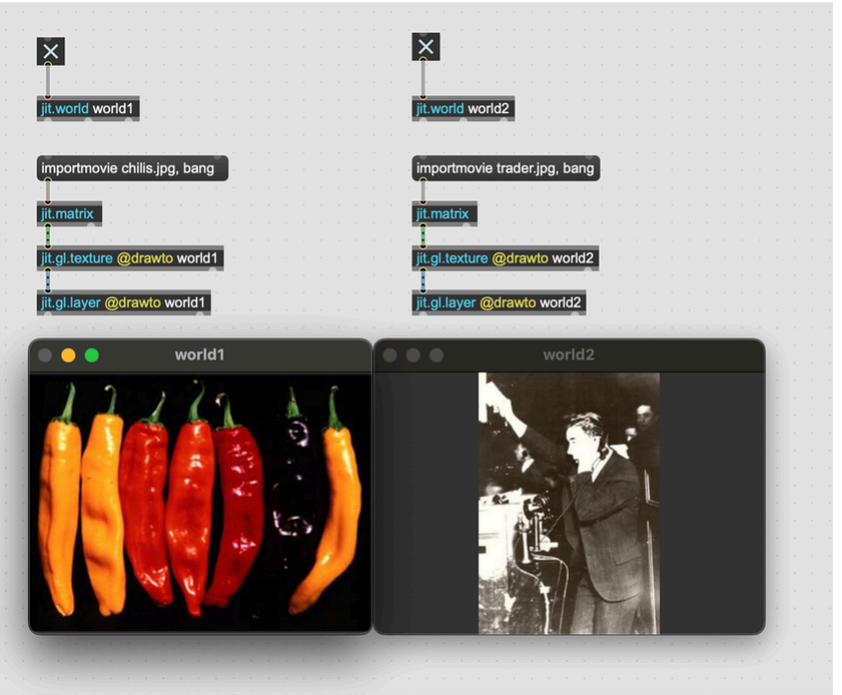
The Max object for managing textures is called jit.gl.texture. Similar to objects like jit.matrix and buffer~, every Jitter texture has a name, and objects send a message like `jit_gl_matrix u1234567` to other objects to refer to a specific texture. Like matrices, Jitter textures have a *dimension* and a *type*. The type of a texture determines the resolution of each pixel. For example, a `char` type can have 256 different values, while a `float32` type can have millions of different values.



If a texture isn't given an explicit name, Max will assign it a unique name automatically.

Graphics context

All textures are belong to a particular graphics context, usually a window managed by jit.world, or else a view managed by jit.pworld. Use the `@drawto` attribute to explicitly assign a texture to one context or another.

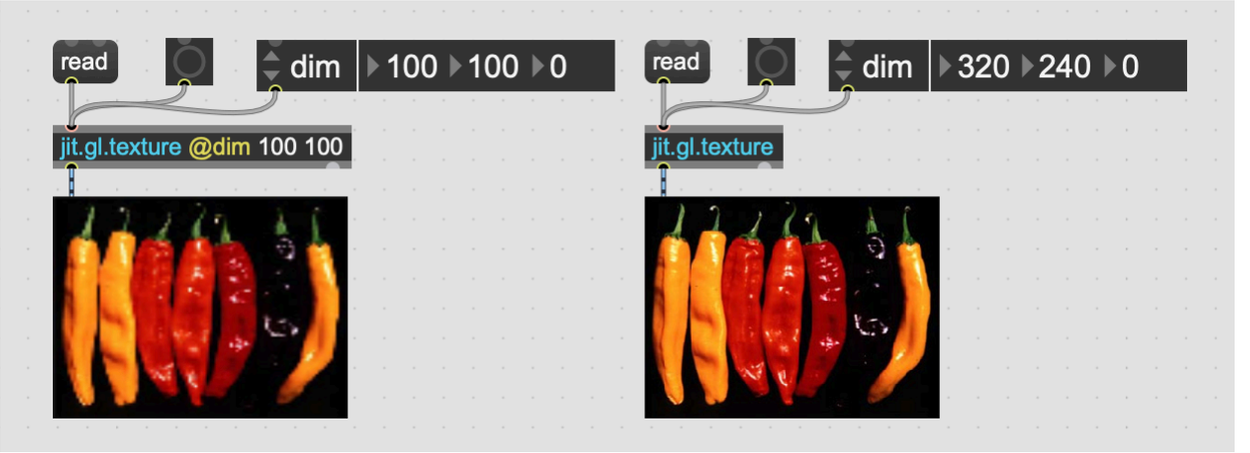


Use @drawto to bind a jit.gl.texture to a particular graphics context.

> If you don't assign jit.gl.texture a value for `@drawto`, it will bind to the first graphics context that it can find. If there is more than one graphics context, it's undefined which context it will bind to. It's common to see `@drawto` omitted when there is only one graphics context.

Loading from a video or image

You can fill a texture with an image by sending it the `read` message. After reading, the texture will store the contents of the image in memory. If you don't give the texture an explicit value for `@dim`, then the dimensions of the texture will adapt to fit the image.



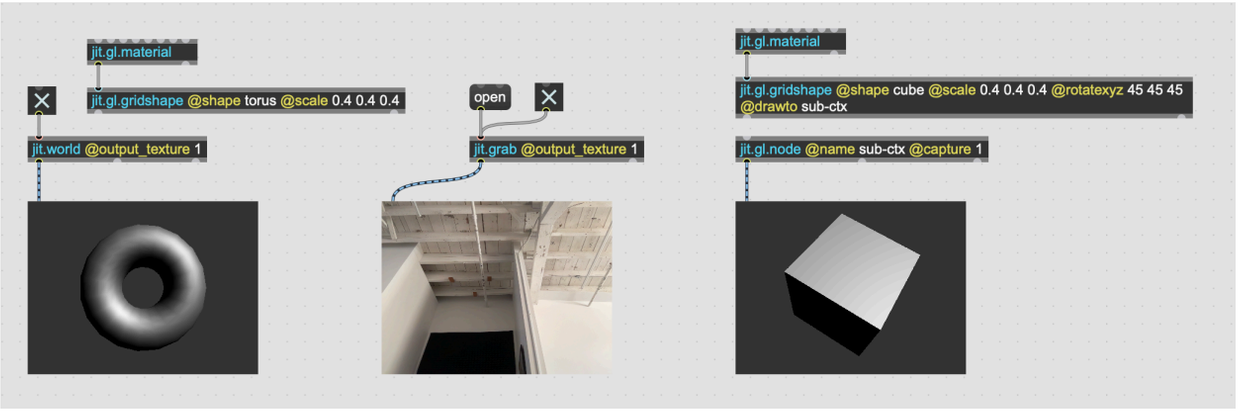
Even though both texture objects loaded the same image, the dimensions are different because one texture does not have an explicit value for @dim.

Adapt

When a jit.gl.texture receives another texture as input, it copies that input texture. If the `@adapt` attribute is enabled, the jit.gl.texture will also update its dimensions and type to match that of the input. The `@adapt` attribute defaults to active, unless an explicit value for `@dim` is set.

Texture output

Many Jitter objects have an attribute `@output_texture` that can be enabled in order to configure them for texture output. When streaming video, either from a camera with jit.grab or from a file with jit.movie or jit.playlist, enable `@output_texture` to stream to a texture as opposed to a matrix. Similarly, toggle `@output_texture` on a jit.world object to capture the entire render output as a texture. When using jit.gl.node to render to an offscreen context, use `@capture 1` instead of `@output_texture 1` to get a texture output from jit.gl.node.



Three important objects that can be configured for texture output

Converting to a Matrix

Remember that Jitter matrices reside on the CPU, while Jitter textures are managed by the GPU. Because of this, converting an texture to a matrix or a matrix to a texture requires copying data from one processor to the other. This can take an undefined amount of time, and so Max provides both a synchronous and an asynchronous method to convert matrices to textures.

jit.matrix

You can send a Jitter texture directly to a jit.matrix object. Max will read the data from the texture and store it in the matrix. This is the synchronous way of converting a texture to a matrix. It's the fastest way to get the result, but it can block other rendering commands during the data copy. If you're trying to copy a texture to a matrix during a live performance, it's usually better to use jit.gl.asyncread.

jit.gl.asyncread

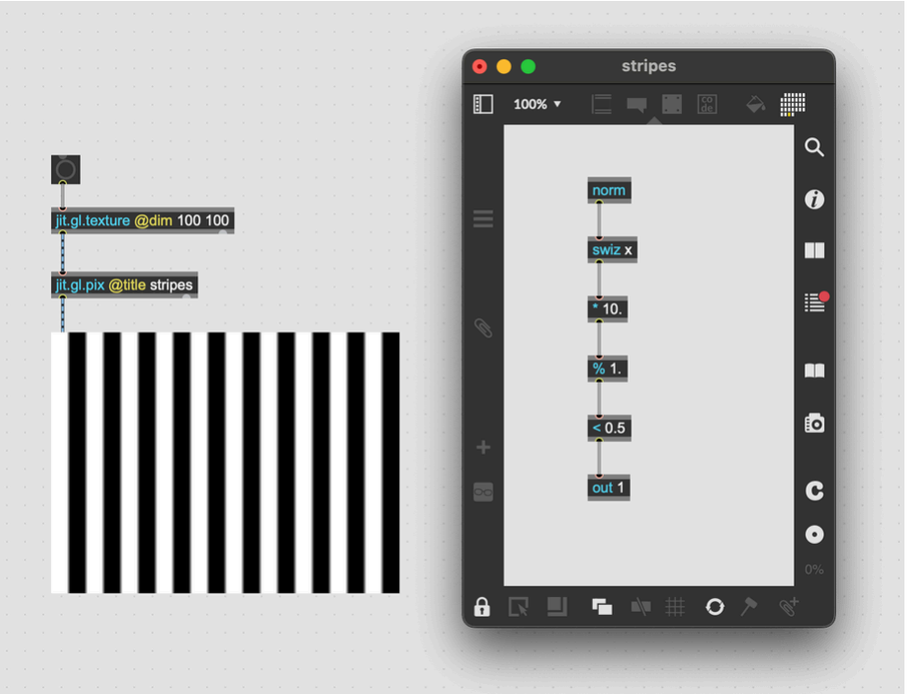
This object uses a double-buffering technique to copy data to a matrix without blocking. Because of this, you're safe to use jit.gl.asyncread as much as you want, even during other real-time rendering. However, since the process is asynchronous, you're guaranteed to incur one frame of delay durin the copy. Most of the time this doesn't matter, and it's fine to use jit.gl.asyncread unless you're doing something very specific.

Procedural Textures

In addition to loading textures from an image file, or copying them from a Jitter matrix, you can also generate texture procedurally.

jit.gl.pix

The jit.gl.pix object uses the Gen object namespace to define a pixel shader. One way to make a procedural texture is by using the cell, dim, norm, and snorm objects.



A simple procedural shader defined with jit.gl.pix

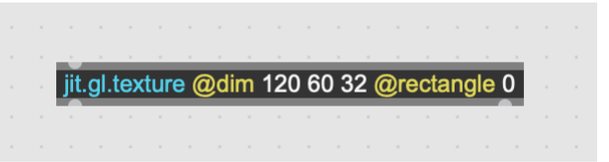
jit.gl.slab

The jit.gl.slab object works very similarly to jit.gl.pix, except it load a shader definition from a Jitter XML Shader or JXS description file. Generator shaders like `gn.stripes.jxs` can use texture coordinates to fill a texture procedurally, just like you might do with jit.gl.pix.

3D Textures

Most textures have two dimensions, but Jitter supports 3D textures as well. This can be useful when generating your own textures, or when using textures to do simulations or other kinds of

computation. To create a 3D texture, simply set the `@dim` attribute of a texture to be a list of three numbers, one for the size of each dimension.



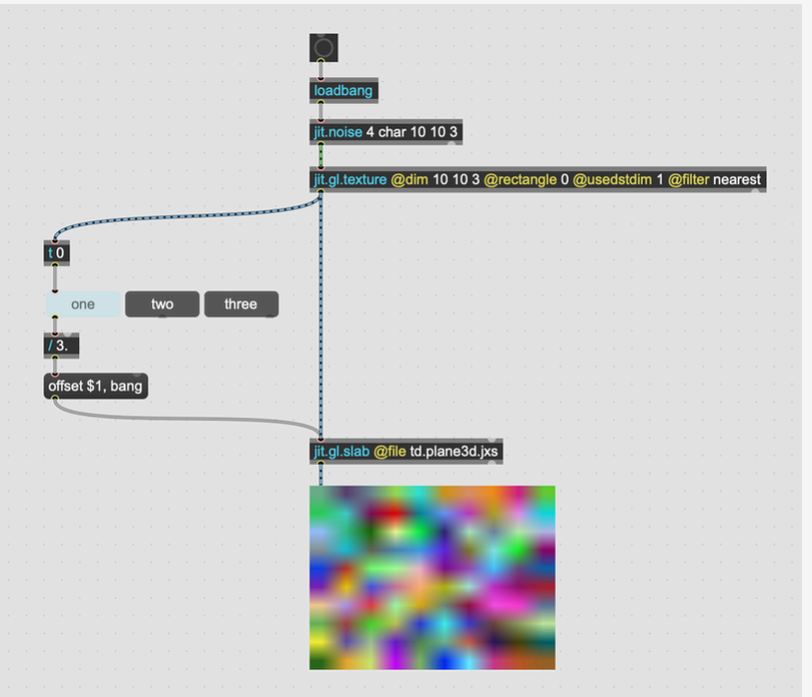
A three-dimensional texture

> When working with a 3D texture, you must set the `@rectangle` attribute to zero. This may change with a future version of Max, but it's necessary for now.

You can fill a 3D texture directly by sending it a 3D matrix. As usual when converting from a matrix, Max will simply copy the matrix to the texture. However, you can also use two-dimensional images to fill each slice of the 3D texture one-by-one. If you want to fill a 3D texture using a series of 2D matrices, use the message `subtex_matrix` in conjunction with the `@dstdimstart` and `@dstdimend` attributes. Open the example patcher subtex.3d for a demo.

You can also fill a 3D texture using a series of 2D textures. For this method, set the `@slice` attribute of the 3D texture object before sending it a two-dimensional texture. See the example patcher tex3d.warpy to see how this works in practice.

To pull a 2D texture slice from a 3D texture, you must use a shader program that takes a 3D texture input. The built-in JXS file `td.plane3d.jxs` implements a shader program that can sample a specific slice from a 3D texture using either an `offset` parameter or with a secondary *map* texture. Use this shader with jit.gl.slab to convert a 3D to a 2D texture.



Taking a 2D slice from a 3D texture with td.plane3d.jxs and jit.gl.slab

Video

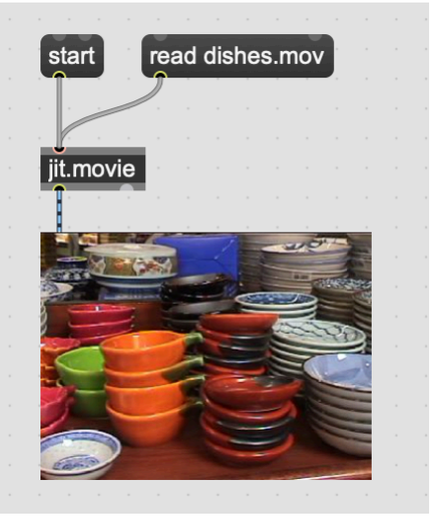
Playing a Video File 349
Using a Video Capture Device 351
Sending and Receiving from a jweb Browser 351
Recording 351
Further Reading 355

The Jitter **video engine** supports streaming video from a webcam or other video capture device, as well as playing a video from disk. Using the third party Syphon and Spout packages, it's also possible to stream video to other applications.

For applications like random video access or rapid resequencing of video material, you might want to consider loading your video into memory, or using a specially designed encoding like HAP.

Playing a Video File

Use the jit.movie object to stream a video from disk. This object uses the current video engine to manage reading and streaming the video.



The standard jit.movie object plays back audio directly using the video engine, rather than routing audio through Max. Use the jit.movie~ object to use the audio tracks of the video file in Max. The mc.jit.movie~ file makes the audio tracks available as multichannel audio.

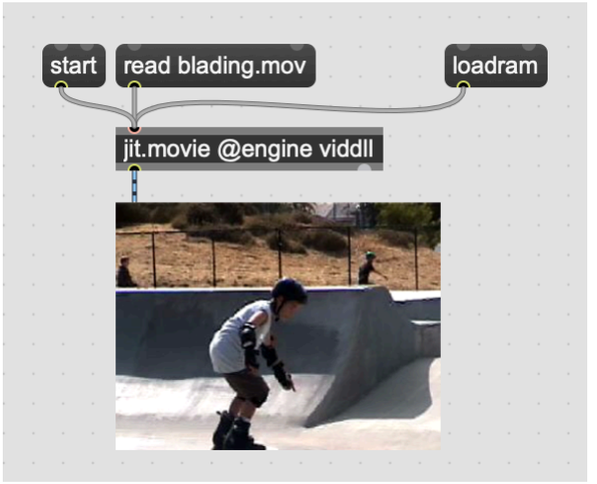


Random Access

Use the `rate` message to change the playback speed of a jit.movie object. Use the `frame` and `jump` messages to move between different parts of the video file.

Keep in mind that videos files are generally encoded in a way that optimizes linear playback from start to finish. Depending on the size of the video file, your computer, and the specifications of your hard drive, you might need to make a couple of changes to improve performance when jumping around a video file.

When using the viddll engine, you can use the `loadram` message to pull an entire video file into Max's application memory. Accessing application memory can be significantly faster than accessing hard disk memory, and this might improve performance.



The HAP codec (collection of codecs actually) is a specially designed video encoding that's made for VJs and creative codeers. Depending on how you're using your video in Max, you might find that jit.movie is more responsive when using the HAP codec.

Using a Video Capture Device

The jit.grab object manages access to video capture devices. Use the `open` and `close` messages to get access to a particular capture device. The `getvdevlist` and `getformatlist` messages let you fetch a list of capture devices and formats, which you can set with the `vdevice` and `format` messages. Check the jit.grab helpfile for more details.

Sending and Receiving from a jweb Browser

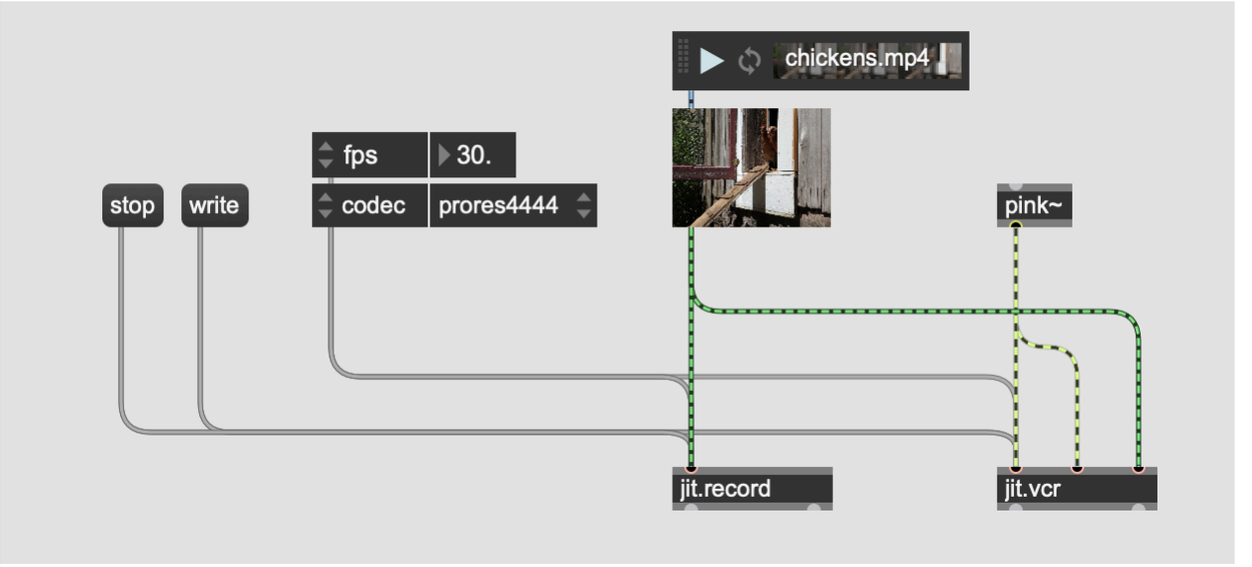
You can read the display of a jweb webpage as a matrix using the jit.web object (or jit.gl.web for textures). You can also use JavaScript to send/receive matrices as well. See Web Browser and jweb for more.

Recording

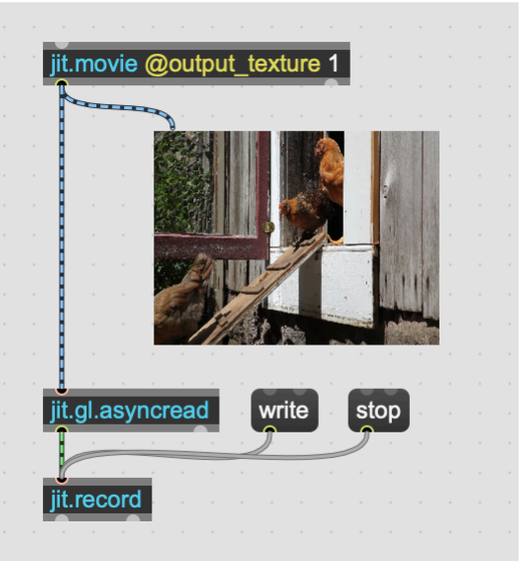
Max gives you a lot of flexibility when it comes to recording video, but there are three general approaches: using jit.record/jit.vcr, using Syphon or Spout to send video to an external recorder like OBS, or frame-by-frame export with jit.matrix and `exportimage`.

Recording with jit.record and jit.vcr

The jit.record and jit.vcr objects let you record a stream of Jitter matrices to disk. In terms of video recording the two objects are the same, except jit.vcr can record audio as well as video. This is the simplest and most straightforward way to record video. For higher quality recording, consider recording with Syphon, Spout, and OBS.



You can use the `@codec` attribute to change the codec used for recording, from `prores444` for a high quality recording with alpha, to `h264` for a lossy encoding and a smaller file size. The jit.record object can record frames as it receives them (non-realtime mode, the default), in which case you'll need to be sure that your recording FPS (frame per second) matches the FPS of your Jitter patcher. Finally, note that jit.record and jit.vcr both record matrices, not Jitter textures. If you want to record a texture, use jit.gl.asyncread to convert textures to matrices.

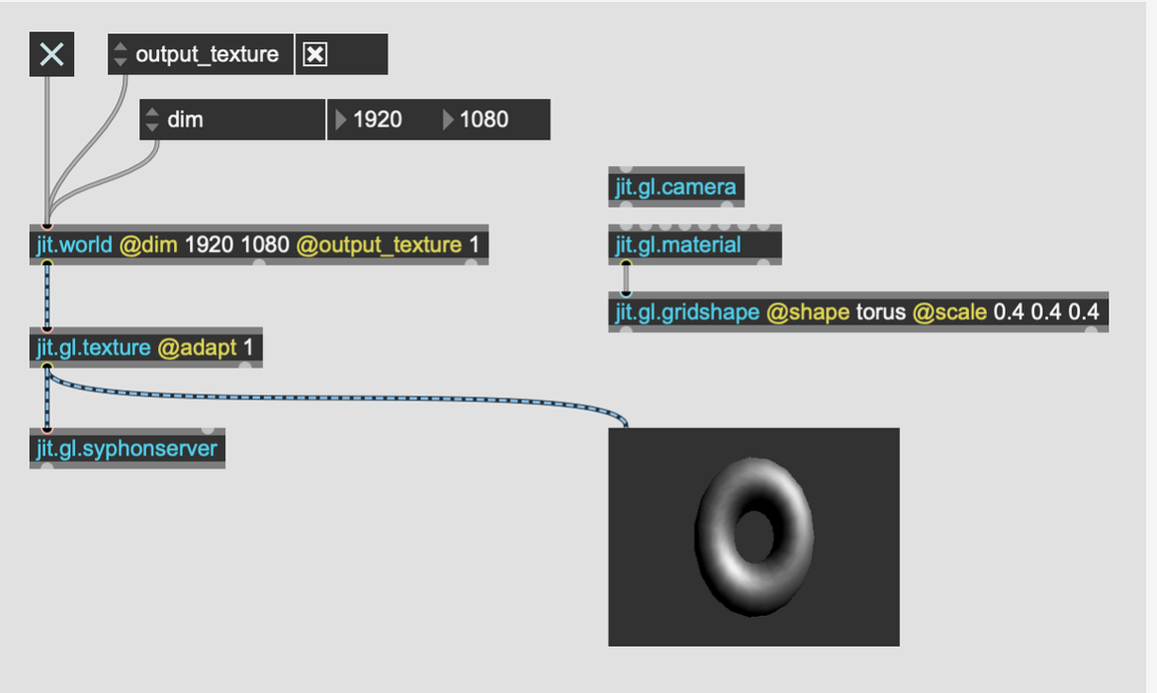


The jit.gl.asyncread object converts textures to matrices for recording

Recording with Syphon, Spout, and OBS

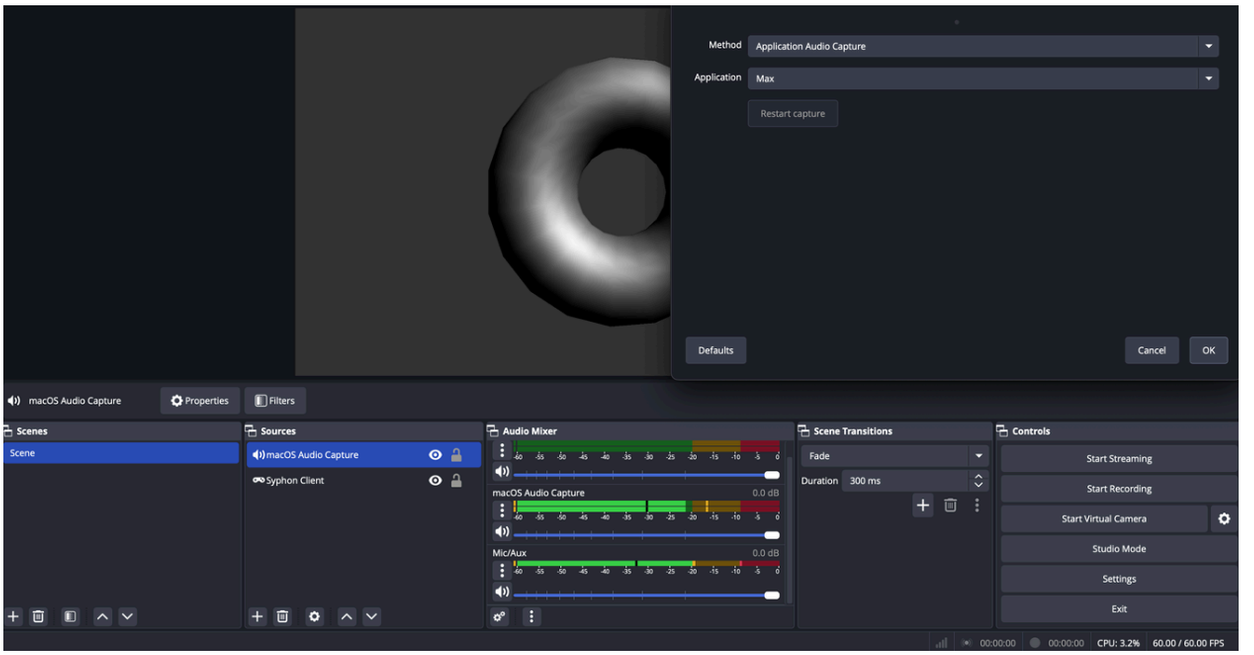
If you want to make a higher quality recording, often it's better to send textures through Syphon or Spout to an external application like OBS. After installing Syphon or Spout, create the apporpriate server object: `jit.gl.syphonserver` for Syphon, and `jit.gl.spoutsender` for Spout. If you're using jit.world, enable the `@output_texture` attribute to make sure that jit.world is sending its texture as an output.

> If you're using jit.world, you'll need to make sure that the texture size coming out of jit.world matches what OBS is expectind to record. Set the `@dim` attribute to match the canvas resolution in OBS. You'll also need to include a `jit.gl.camera` object in your patcher, to make the viewport size independent of the `jit.world` window size.



Using jit.gl.syphonserver to send textures from Max to OBS

To record audio as well as video, create a new Audio Capture source in OBS, and configure this for Application Capture. Set the application to Max, and you can record audio from Max at the same time as you record video.



A simple OBS setup for recording video with audio from Max

> OBS audio sources may not support Application Capture on Windows. You can use Desktop Capture, or you can use a third-party application to route audio from Max to OBS. We recommend Voicemeeter or JACK.

Recording with frame-by-frame with jit.matrix and `exportimage`

For total control over the recording process, you can explore a non-realtime recording setup. This will look something like the digital equivalent of stop motion filmmaking. After rendering your scene, capture the output in a jit.matrix object. Send that object the `exportimage` message to save the image to disk. Later, use FFmpeg to render a final video from your sequence of images.

Further Reading

For a more in-depth tutorial concerning recording video from Max, check out this article.

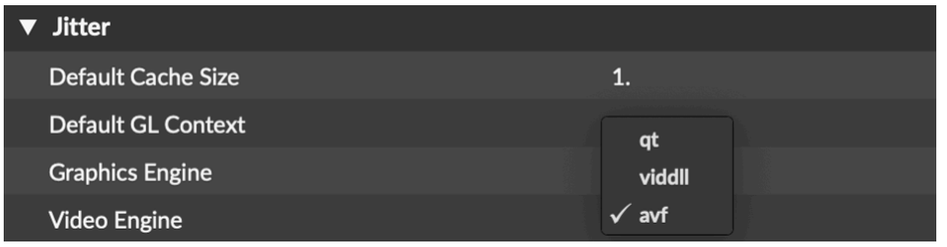
Video Engine

Changing the Video Engine 356
Platform Specifics 357
Codec and Format Support 357
jit.grab 357

The **Video Engine** is responsible for interfacing with the operating system to manages access to video hardware devices and to video files on disk.

Changing the Video Engine

Max's *Video Engine* preference allows users to switch the backend video implementation for all video objects.



Objects affected include jit.movie, jit.record, jit.playlist, and jit.matrixset. Still image loading may be affected by the video engine for jit.matrix and jit.gl.texture. Individual jit.movie, jit.grab and jit.record objects may override the Video Engine application preference by typing `@engine`, followed by the engine name argument, into the Max object box.

> Objects previously initialized are unaffected by a preference change, therefore open patches should be closed and reopened after switching the video engine.

Platform Specifics

Max ships with support for two video engines on Mac platforms, **avf** (AVFoundation - the default) and **viddll** (Viddll - FFmpeg), and two on Windows, **viddll** (the default) and **qt** (DirectShow). The DirectShow based engine is named **qt** for historical reasons, and has limited functionality. Windows users wishing to install third-party codecs for the **qt** engine should follow the instructions here. The **viddll** engine utilizes the FFmpeg library to provide support for a wide variety of file formats and codecs. Both **avf** and **viddll** engines provide native playback support for HAP encoded video files.

Codec and Format Support

Common supported codecs for movie file reading with jit.movie and jit.playlist and file writing with jit.record and jit.matrixset:

- H264
- Photo-Jpeg
- ProRes (422 and 4444)
- Animation (**viddll** only)
- Many additional formats and codecs when using **viddll**

Supported image types for file reading with jit.matrix and jit.gl.texture:

- JPEG
- PNG
- TIFF
- GIF

jit.grab

The jit.grab object is unaffected by the video engine preference. On Mac, jit.grab will use AVFoundation as the video digitizer, and on Windows DirectX is used. Additionally both platforms

include native support for Blackmagic video input devices. See the *Blackmagic* tab of the jit.grab help file for more information.

Max Interface

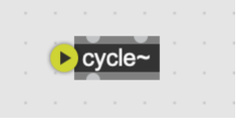
Action Menu

Viewing the Action Menu 360
Parts of the Action Menu 360
Using the Action Menu 362

The Action Menu lets you explore, transform, and interact with objects in your patcher. It groups together common operations on an object, like viewing its attributes or messages. It also gives you access to Transformations, which are a powerful way to refactor your patcher.

Viewing the Action Menu

With the patcher unlocked, hover near the middle-left side of the object until a green arrow appears.



Click the arrow to reveal the action menu.

Parts of the Action Menu

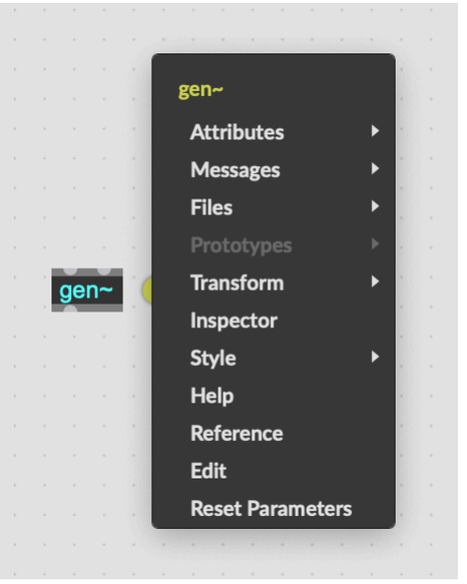


Name	Description
Attributes	Lists the current state of all the object's attributes. Select an attribute from the Attributes submenu to create an attrui object attached to the object, configured with the given attribute.
Messages	Select a message from the Messages submenu to create a message box attached to the object containing the given message. You'll typically want to type in an argument after the message name in the newly created message box.
Files	If the object can read in a file, like sfplay~ or jit.movie~, then this option will list all compatible files in Max's search path. Selecting any one will load that file into the object.
Prototypes	Select a prototype from the submenu to replace the object with a new version containing a collection of attributes. See prototypes.
Apply Changes	Choose a prototype from the submenu to apply the collection of attributes to the object. See prototypes.
Connect	(UI objects only.) Assign a connectable parameter. See Connecting Parameters.
Transform	Transformations defined for this object. See transform.
Inspector	Open the inspector for this object.
Style	Choose a style for this object.
Help	Open the help patcher for this object.

Reference	Open the object reference for this object.
Edit	Perform the same action as double-clicking on the object. For example js and coll open a text editor to edit their contents. MIDI objects such as noteout display a menu of MIDI ports.

Additions

Some objects, like gen~ for example, may add their own actions to the action menu. In this case, gen~ adds the "Reset Parameters" option which, as you might expect, resets all the parameters of the gen~ object in question.



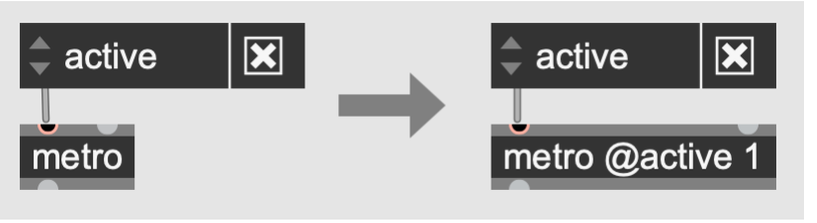
Using the Action Menu

The action menu provides quick access to two of the most powerful ways to modify an object in place: transforms and prototypes.

Transform

Transformations let you change the way an object is represented, without changing its behavior. Usually you'd do this because it's more convenient to work with an object in a different representation.

Transform > Changed Attributes to Arguments



This option takes all of the attributes on the given object that have been modified from their original state, and includes them as initial attribute values in the object box. This can be a useful alternative to freezing attributes, and is a handy way to "lock in" the current state of an object.

Transform > Multi Channel Version / Single Channel Version



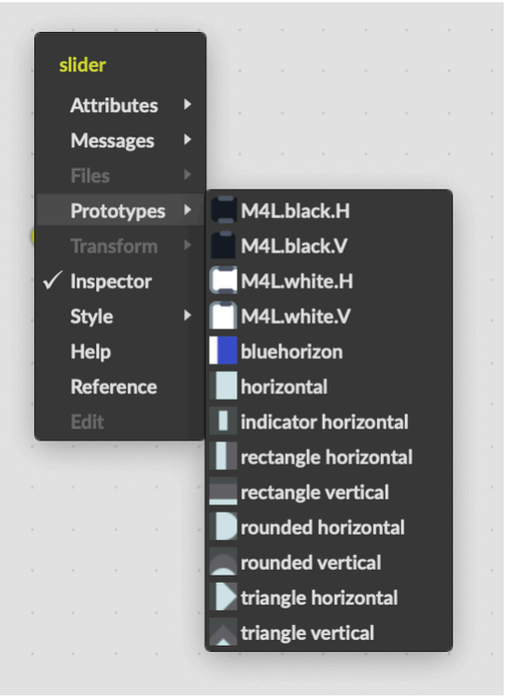
Convert between single- and multi-channel versions of an MSP object. Usually this will involve adding or removing the `mc.` prefix.

Transform > Patcher to Bpatcher



This extremely useful option converts a subpatcher to a bpatcher. If you end up adding interface controls to a subpatcher, this is a great way to expose the presentation view of your subpatcher at the top level.

Prototype



Prototypes store configurations of a given object, such as the range and colors of a slider. Choosing an item from the Prototype submenu replaces the object with the prototype. Choosing an item from the Apply Changes submenu updates the existing object with the changed attributes stored in the prototype file.

Documentation Window

Using the Documentation Window 365
Using the Object Reference 368
Using the Package Documentation 372
Searching the Documentation 374

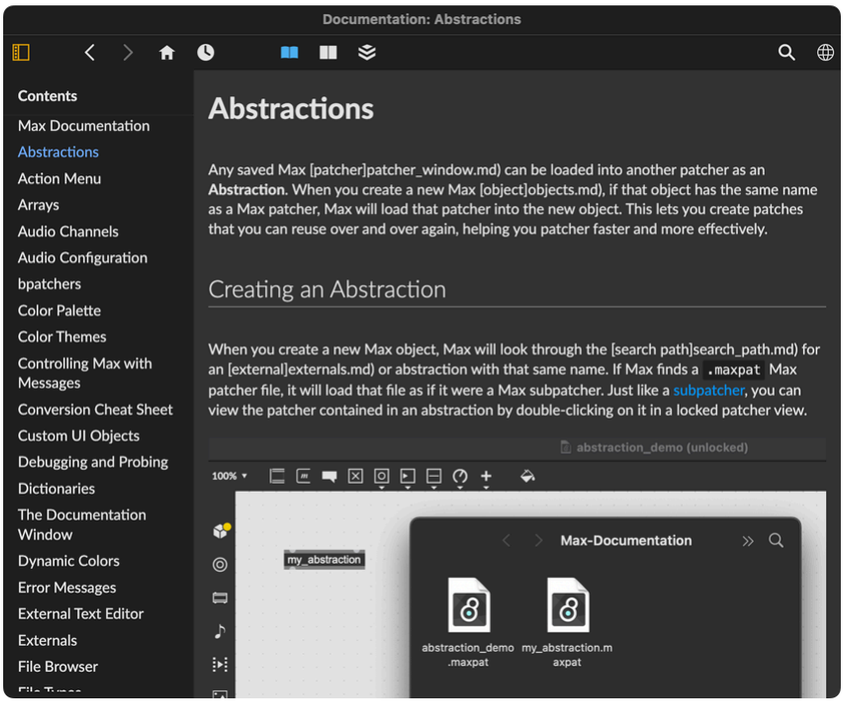
The **documentation window** gives you combined access to core parts of Max's documentation.

- **User Guide**: Full-page explanation of Max, its systems, and how to work with them.
- **Object Reference**: Detailed descriptions of individual objects, including all the messages and attributes that each object understands
- **Package Documentation**: Guides specific to a particular package, written by the package author

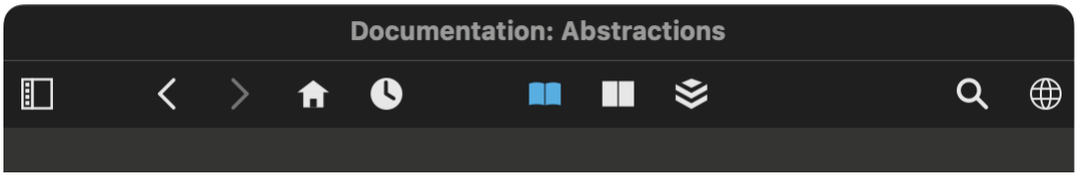
In addition to this, each object also has its own help file, which demonstrates the object's functionality in the context of a working patcher. And there is even more documentation online. The **JavaScript API** is one example of API documentation, listing all of the functions and classes you can use to extend Max with JavaScript. Finally, you can find examples and tutorials online as well, which introduce concepts gradually and show some of what's possible with Max.

Using the Documentation Window

Open the documentation window by selecting *User Guide* from the *Help* menu. This will open the documentation window, focusing on the User Guide.



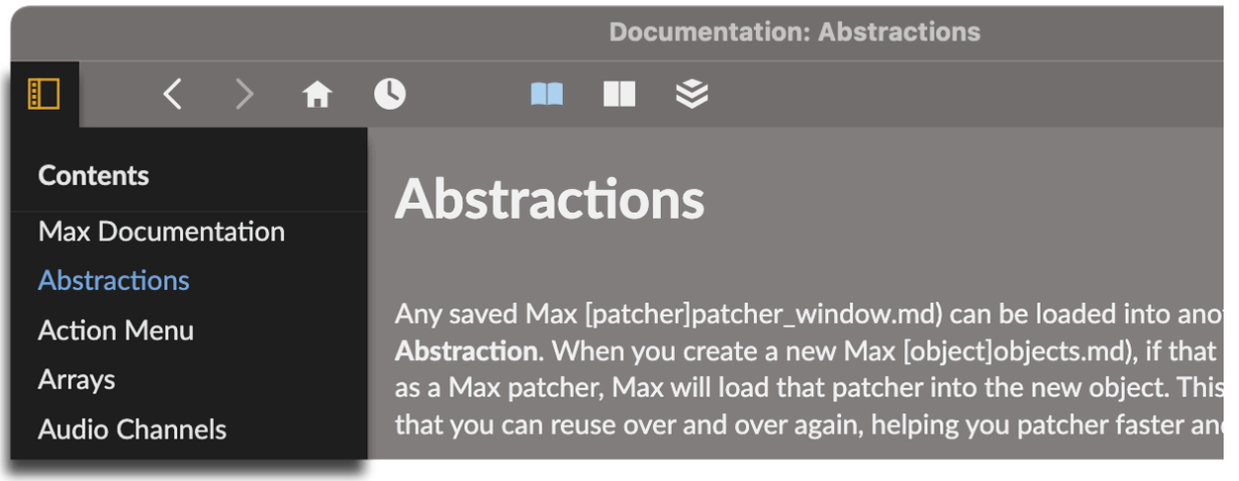
As mentioned earlier, the in-app documentation is divided into three main sections: the User Guide, Object Reference, and Package Documentation. You can navigate through the different parts of the documentation window using the icons at the top of the window.



- **Back**: Navigates to the previously visited page
- **Forward**: After navigating back, returns to the original page
- **Documentation Home**: Go to the documentation home page
- **History**: Click to display a list of recently visited pages
- **User Guide**: Show the User Guide
- **Object Reference**: List and search the available Max objects
- **Package Documentation**: Show documentation for installed packages
- Search: Open the search view

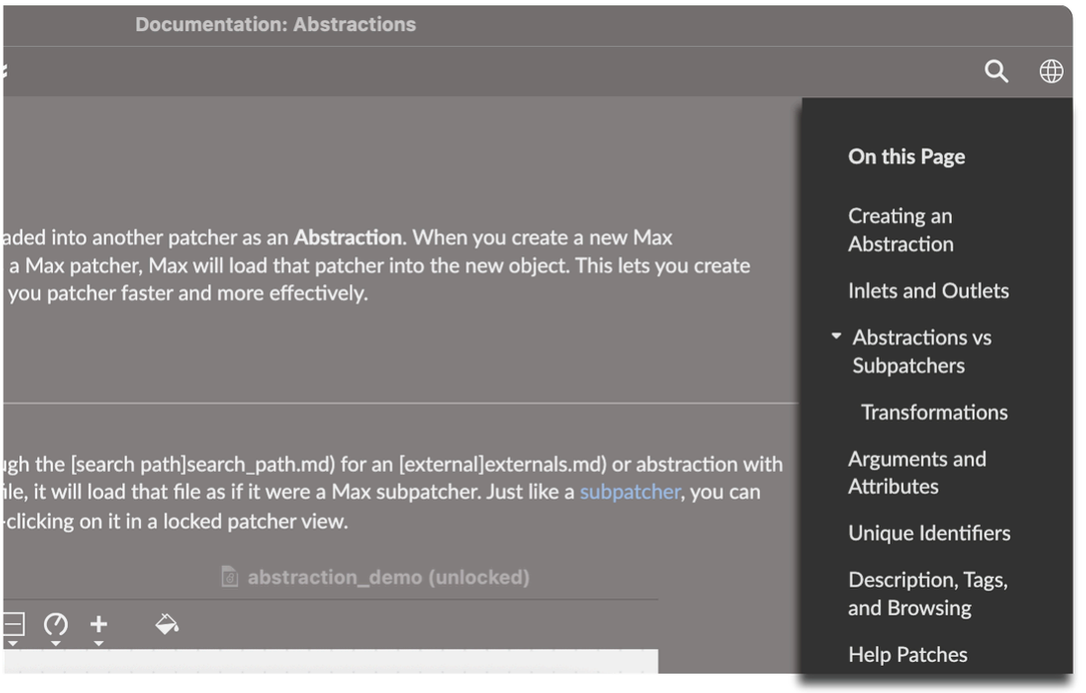
- **Open Online Version**: Open the documentation online

The left side of the documentation window shows the **navigation** for the current documentation area. In the User Guide, this lists user guide pages organized by topic. You can click the *Hide Navigation* icon to toggle the navigation display.



The Navigation icon lets you hide and show the navigation.

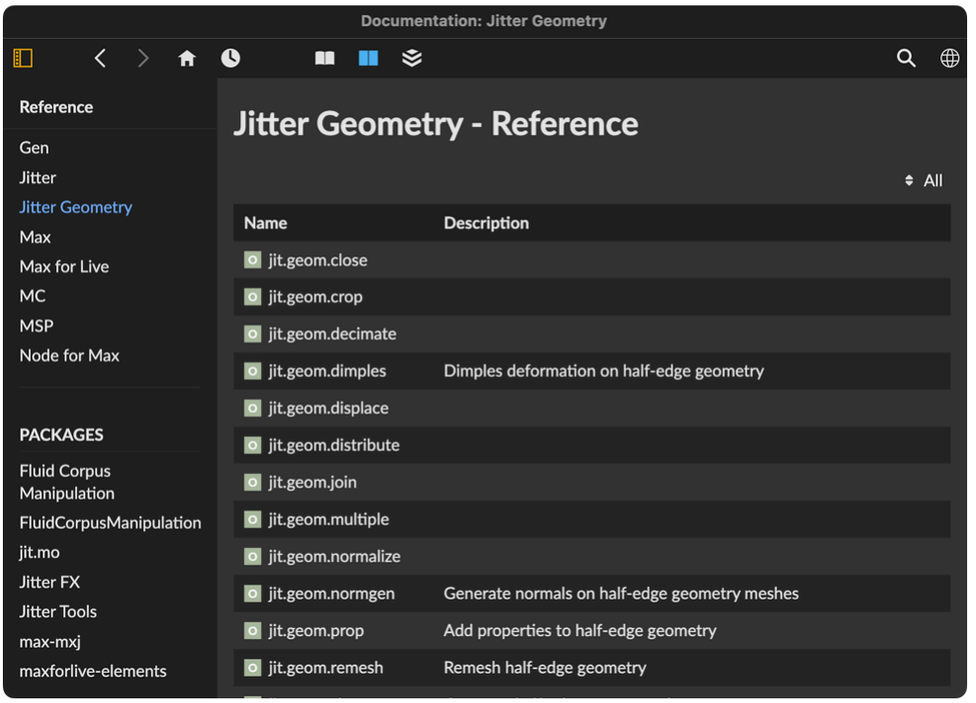
The navigation lets you jump from page to page, but you can also quickly jump through the contents of a particular page by using the in-page navigation. If you make the documentation window wider, the *On this Page* navigation will appear.



The page navigation appears when the documentation window is wide enough

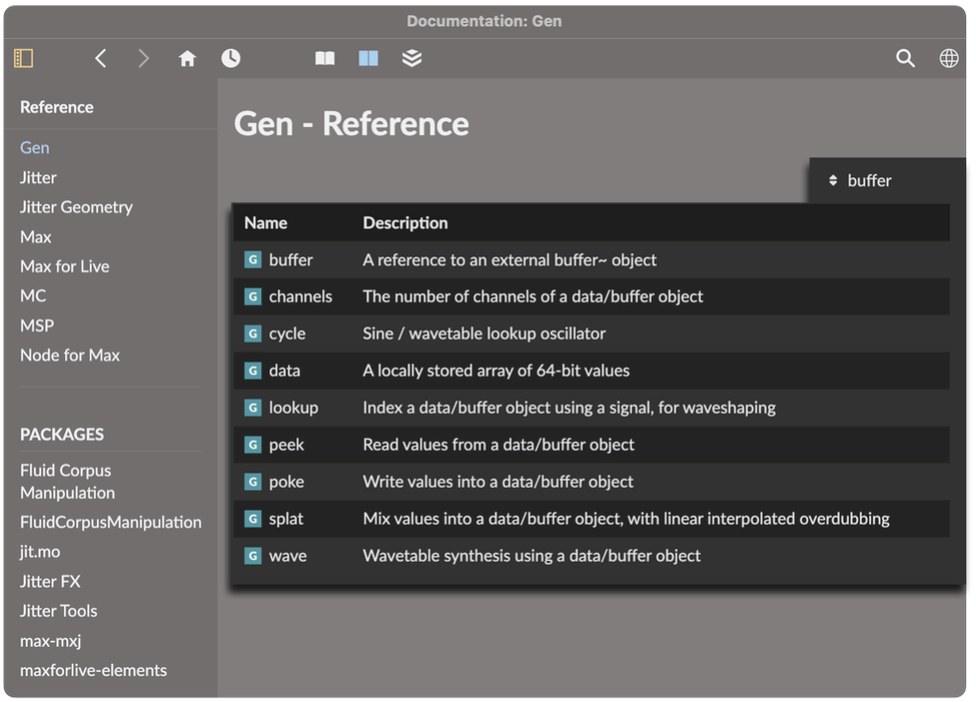
Using the Object Reference

Click the *Reference* icon to display the object reference list.

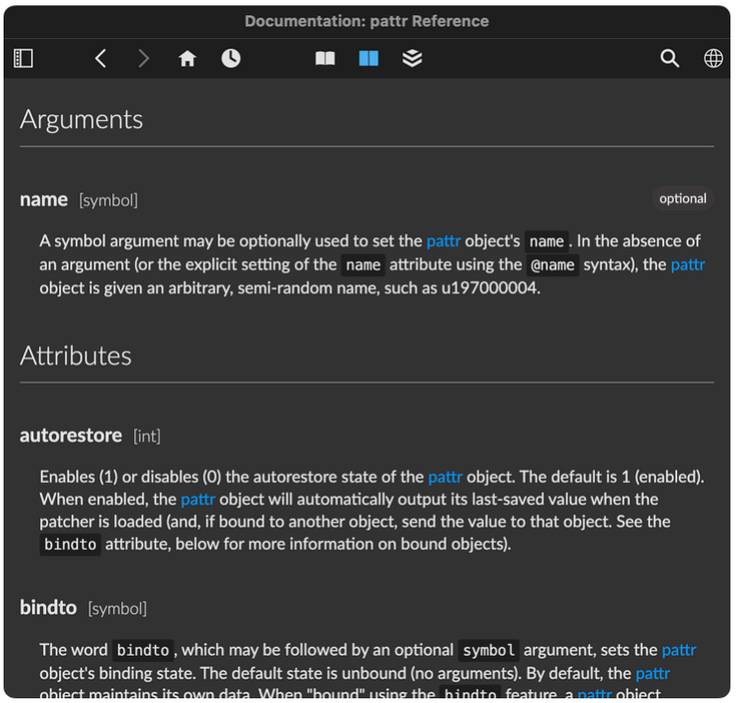


On the left side of the page, you'll see the object reference navigation, which groups objects by built-in section for native objects, and by package name for objects from third-party package. Click on any entry in the navigation to list objects from that section.

On the right side of the page, above the object listing, you can use the drop down menu to further refine the list of objects by category. For example, in the *Gen* section, you can select the *buffer* category from the drop down to see only Gen objects that deal with buffer objects.



Click on an object to view the reference documentation for a specific object. Use the *Open Help* button at the top of the object reference page to open the help patcher for that object. Under that button, you'll see extensive reference for the object, including the arguments, attributes, and messages that the object understands.



The reference documentation for the pattr object

Next to the entry for every attribute, argument, and message, you'll see the expected type for that entry. For arguments, you'll also see the keyword *optional* appear if the value is optional, as well as any default value the argument may have.

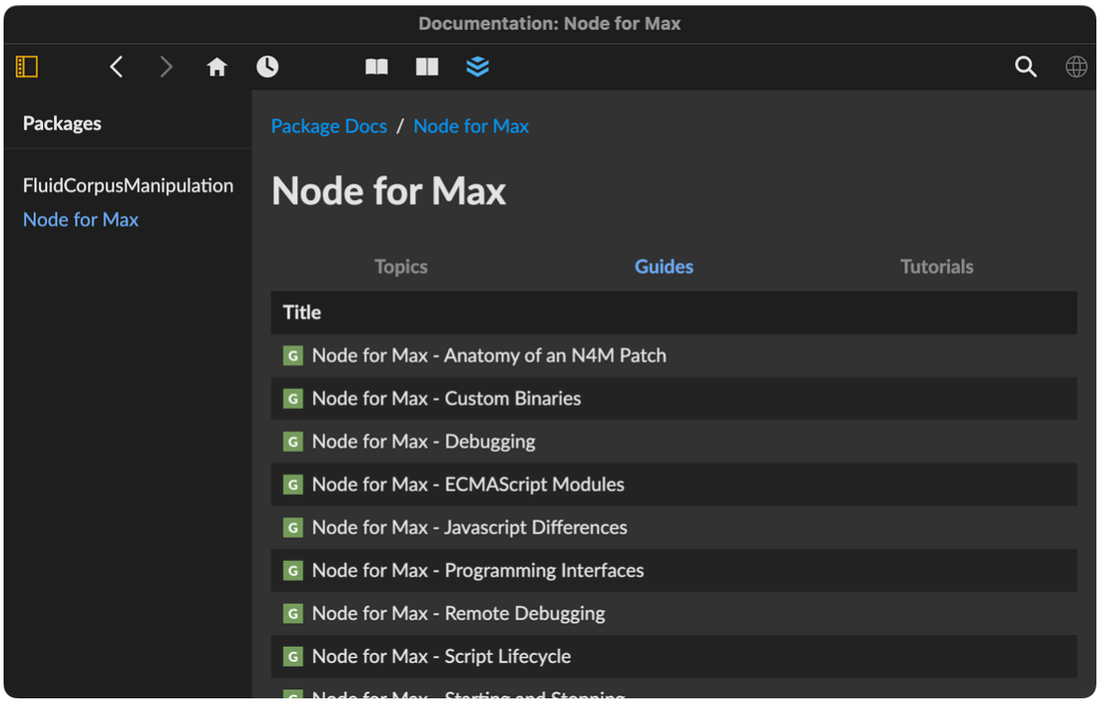
If you make the documentation window wide enough, you'll see *On this Page* navigation, including a disclosure triangle to list arguments, attributes, and messages.



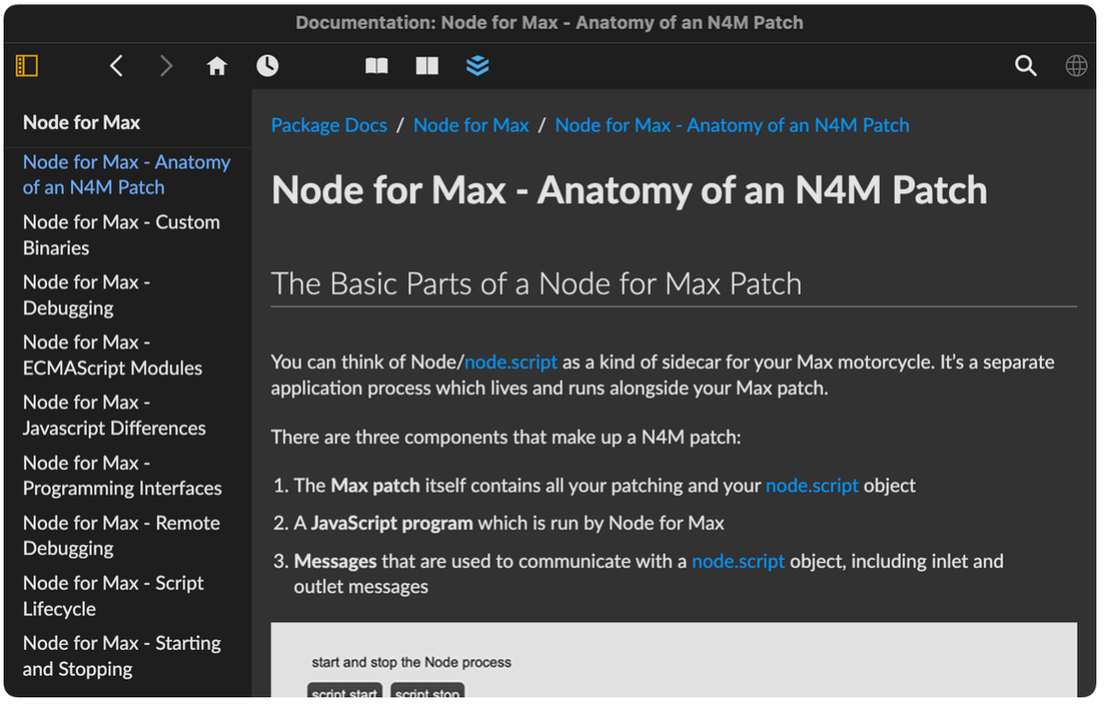
On this Page navigation for the pattr object

Using the Package Documentation

Click the *Packages* icon to display the package documentation.



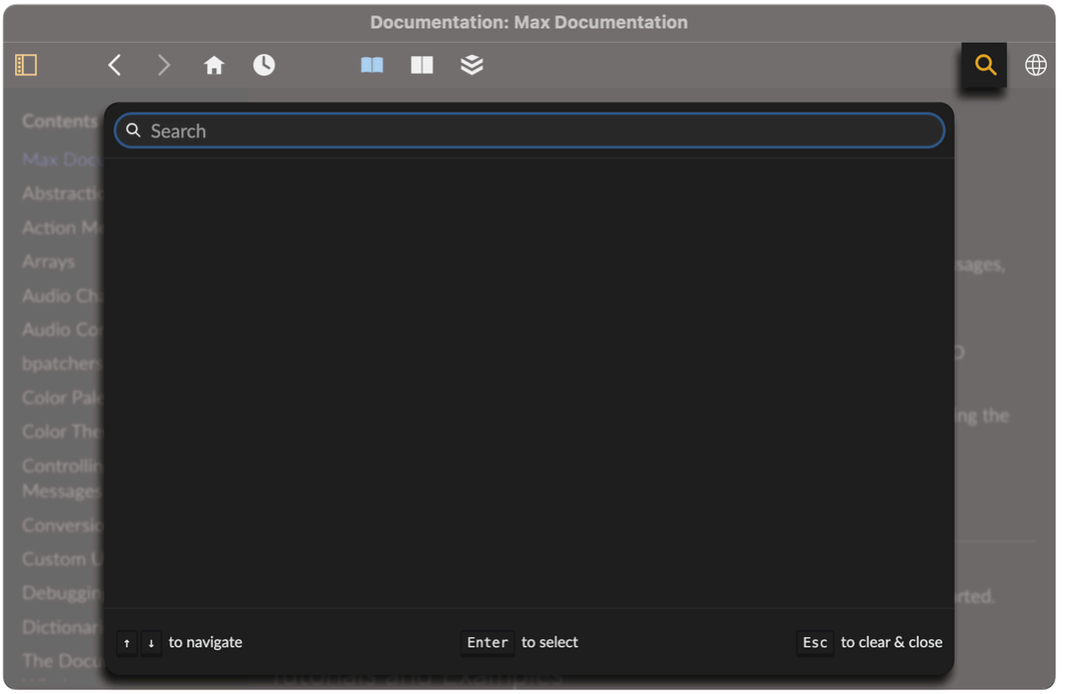
On the left side of the page, you'll see a list of installed packages. If the package authors have written any Guides, Topics, or Tutorials, you'll see those appear in the center of the documentation window. Once you click on a particular entry, you'll see that entry appear.



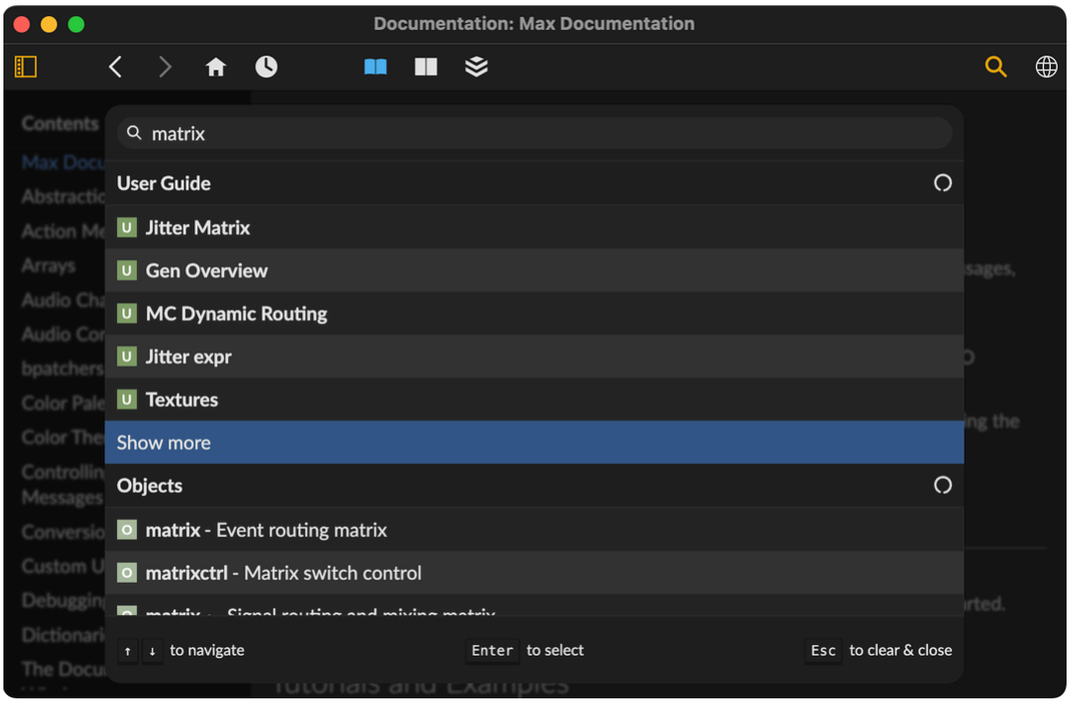
At the top of a package documentation page, you'll see **breadcrumbs** showing you the path to the current page. You can click *Package Docs* to return to the package documentation home page, or on the name of the page to see just the documentation for that package.

Searching the Documentation

Click the *Search* icon in the top-right of the window to display the search view.



When you search, you'll see results from the User Guide, Object Reference, and Package Docs, but you'll also see results from online as well, including API Reference and RNBO results. If there are more than a few results in a given category, click *More Results* to view a page of results from just that category.

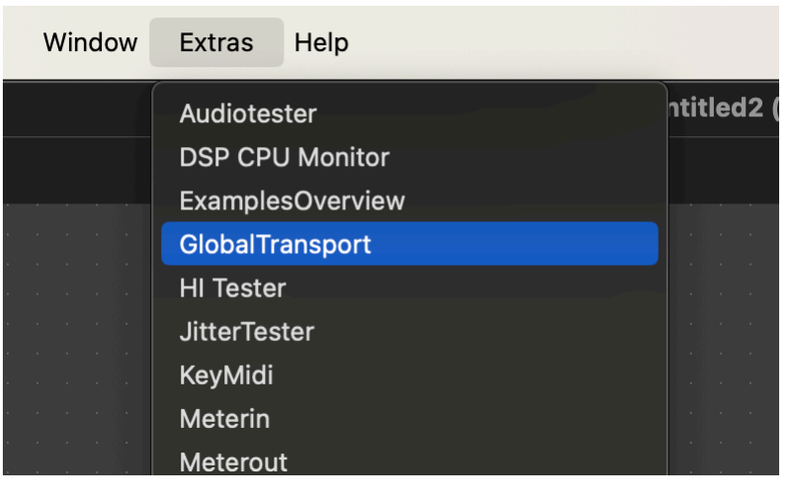


Documentation search results with the search term 'matrix'

Extras Menu

Included Extras 376
Package Extras 377

The *Extras* menu provides quick access to frequently used patchers. Max comes with a set of utility patchers for everyday functions like testing MIDI input and opening the Global Transport, and you can also add your own patchers to the Extras menu.



Included Extras

Extra	Description
Audiotester	Measures input and output levels for your audio setup, and can be used to test your audio hardware and sound system.
DSP CPU Monitor	Shows the current approximate audio CPU utilization.
ExamplesOverview	Browse and launch examples included with Max.
GlobalTransport	Start, stop, and display the passage of time for objects that use Max Time formats.

Human Interface Driver Tester	Set up and test any object that support the hi object.
JitterTester	Tests video input devices.
KeyMidi	Use your computer keyboard as a MIDI keyboard.
Meterin and Meterout	Provide 24 channels of meters for audio input and output.
MIDI Tester	Display all MIDI input and output devices and lets you test your connections.
Mousemeter	Tracks the mouse.
ObjectHelpLauncher	Fast way to open a help file.
Quickrecord	Quickly record the audio output, up to 8 channels.
UDP Tester	lets you test the sending of Max messages over a network using UDP.

You can add your own patchers to the Extras menu by creating a *Package* with an `extras` folder.

Package Extras

Packages that you add to Max (from the *Package Manager* or by adding folders to the `Packages` directory) will add patchers to the *Extras* menu as well. Most packages include a "launcher" patcher that acts as an overview for the package, and some packages will have an *extras* folder that adds even more patchers.

Adding extras

If you want to add your own patchers to the *Extras* menu, you can create a custom package and add your patchers to the `extras` subfolder. See Packages for more information.

Inspector

Opening the Inspector 378
Inspector contents 379
Attribute Names 382
The Patcher Inspector 383
Freezing Attributes 383
Modifying and Resetting Attributes 384

View and modify the internal state of any Max object using the inspector.

Opening the Inspector

Open the inspector by clicking the *Inspector* icon in the right toolbar.



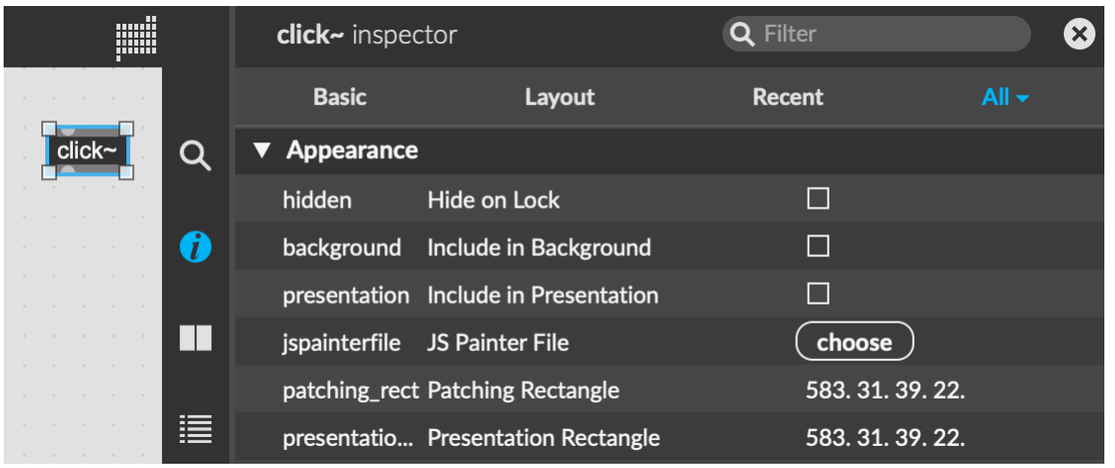
Opening the Inspector

.
You can also select *Inspector* from the *Object* menu as another way to open or close the inspector. If you'd prefer to see the inspector in a separate window, select *Inspector Window* from the *View*

menu instead.

Inspector contents

The contents of the inspector are determined by the current selection. With a single object selected, the inspector will show all attributes for that object.



Inspecting an object

With multiple object selected, the inspector will show all attributes that are shared by the selected objects. Changing the value of a single attribute will update that attribute's value for all selected objects.

Inspector Toolbar

The toolbar at the bottom of the inspector exposes several handy functions.



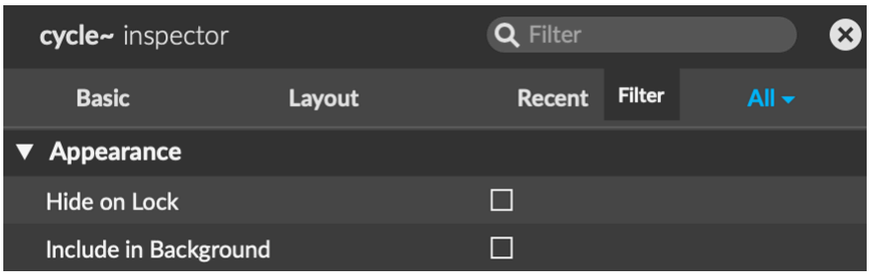
The inspector toolbar

Icon Name	Function
Modify Selected Item	Copy or change the value of an attribute.

Show Attribute Names	Show the programmatic ID of an attribute.
Show Column Header	Show the column headers for sorting.
Freeze Attribute	Freeze or unfreeze an attribute.
Make Attribute in Patcher	Create an attrui object in the patcher to modify the selected attribute.
Show in Reference	Open the reference documentation for the selected object attribute.
Show Object	Highlight and focus the selected object in the patcher view.

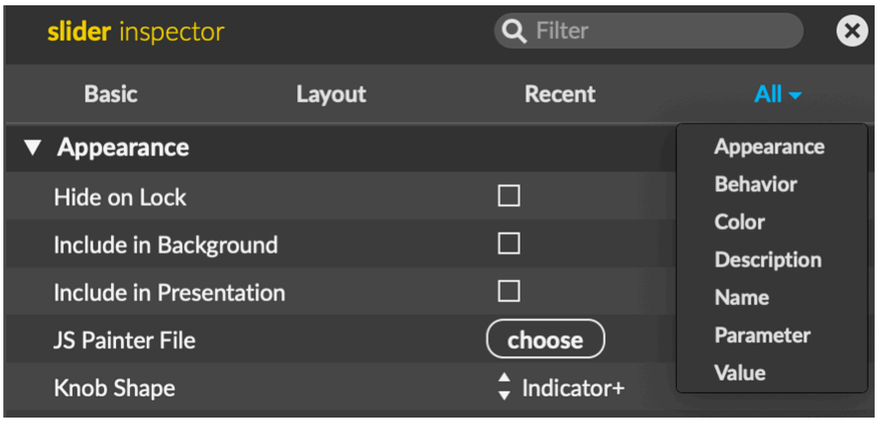
Finding your attribute

Some objects can have a lot of attributes. At the very top of the inspector window, a text input lets you display only those attributes whose text matches the contents of your filter. The match includes the attribute name, not just its display name, and so the filter text "var" will match the attribute `varname` with the display name "Scripting Name", even if *Show Attribute Name* is not enabled.



Filtering the inspector

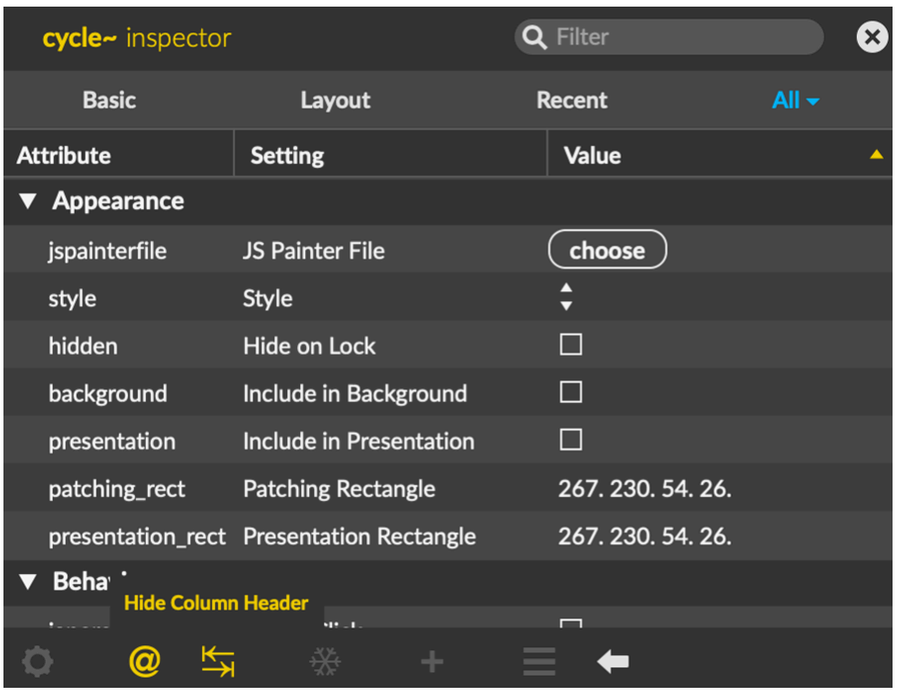
The tabs underneath the filter input select for attributes matching a given category. Attributes in the *Basic* tab are the most common attributes for the selected object. Attributes under *Layout* handle the positioning and appearance of the object. *Recent* shows attributes most recently modified for the object, and the *All* tab shows all attributes. Click and hold the *All* tab to show the subcategories of all attributes, and pick one to open just the disclosure tab for that subcategory.



Show subcategories under All

Sorting attributes

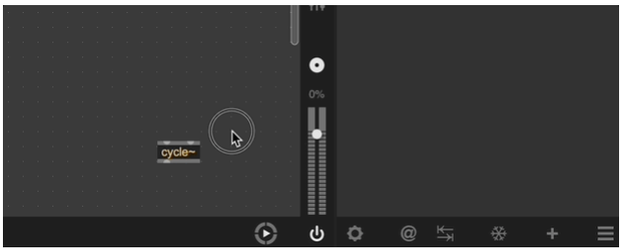
It's possible to sort attributes by name or by value. In the bottom toolbar, click on the *Show Column Header* icon in the bottom toolbar to reveal the headers of the inspector table.



With the headers revealed, click on any header to sort all attributes based on the value of that column. Click on the header again to switch between ascending and descending sort.

Dragging attributes

You can drag rows from the inspector into your Max patch. When you do so, Max will create a new attrui object, configured to display the selected attribute. In this way you can quickly build an interface for controlling a particular set of attributes.

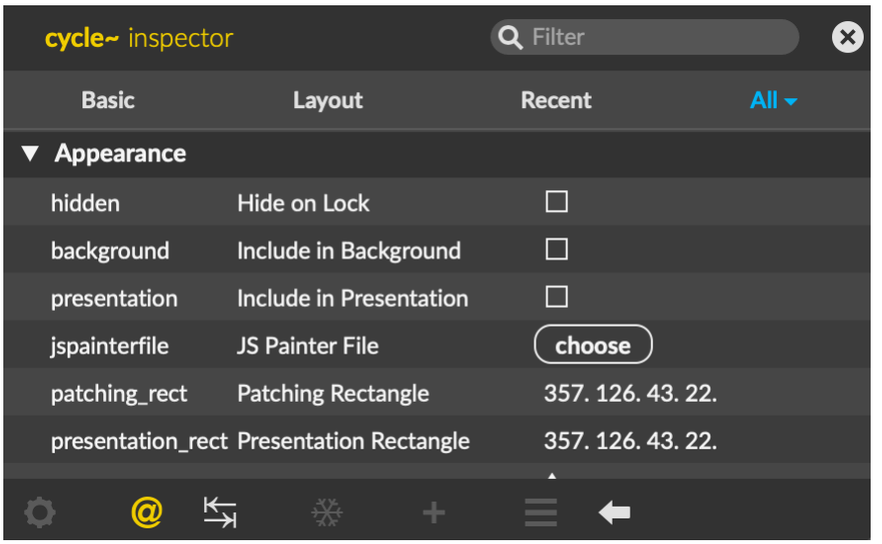


Drag and drop attributes from the inspector to create attrui objects

If you drag the attribute on top of an existing object, Max will automatically connect the attrui object to the target object. Finally, you can hold down the `option` key while dragging to display a popup menu with more options, including an option to create a message box containing the current value of the attribute.

Attribute Names

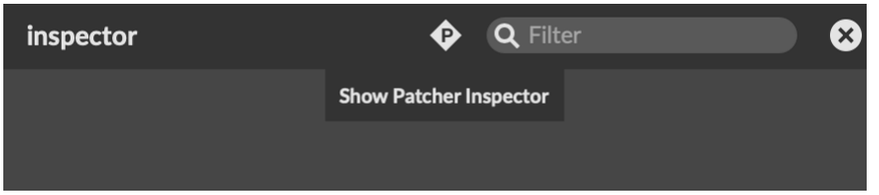
Attributes can be identified by their *Display Name*, a brief, human-readable description, or by their *Scripting Name*, a unique identifier used to fetch the attribute programatically. By default, the inspector hides the scripting name and shows only the display name. The *Show Attribute Name* button in the inspector toolbar lets you toggle the visibility of the scripting name of each attribute.



Enable 'Show Attribute Names' to display the scripting name of each attribute.

The Patcher Inspector

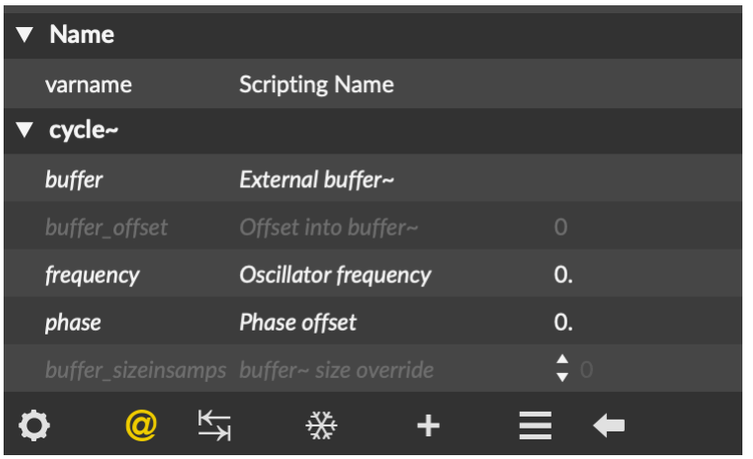
The *Patcher* itself is, behind the scenes, just a Max object like any other. Many properties of the patcher, like the patcher background color, can be controlled by modifying the attributes of the patcher. To access the patcher inspector, open up the inspector with no object selected. An icon will appear at the top of the empty inspector view, which you can click to access the *Patcher Inspector*.



An empty inspector, revealing the 'Show Patcher Inspector' icon at the top of the inspector view.

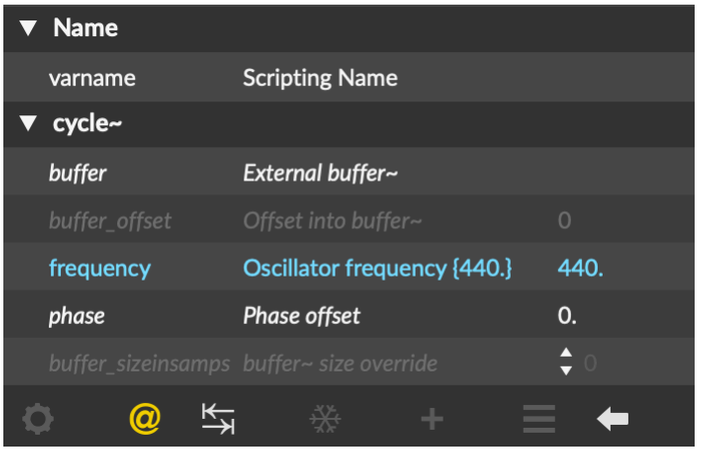
Freezing Attributes

Most attributes, like font size or scripting name, are saved with the patcher and will be restored when you reopen the patcher later. However, some attributes are not stored by default, and will appear italicized in the object inspector.



Unsaved attributes are shown in italics

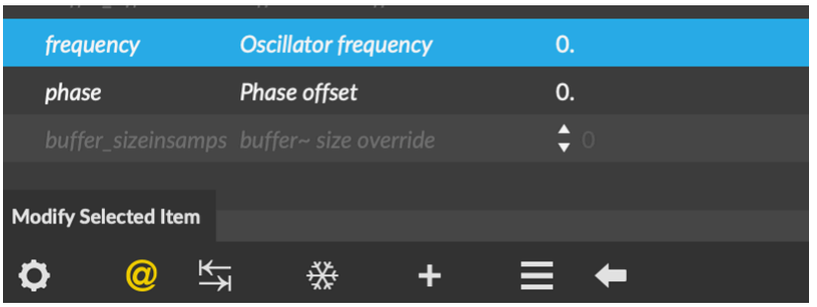
If you want to save the value of an attribute, you can use the snowflake icon in the inspector toolbar to *freeze* the attribute. Frozen attributes will embed their current value with the patcher, so that this value can be restored when the patcher is next opened. Once the attribute is frozen, the display name will show the frozen value of the attribute. It is also possible to freeze attributes that are normally saved with the patcher. Once an attribute is frozen in its way, the frozen value is the value that will be restored when the patcher is closed and reopened. It might be useful to freeze an attribute like this to "anchor" it to the frozen value, rather than its current value.



The frequency attribute, after it's been frozen.

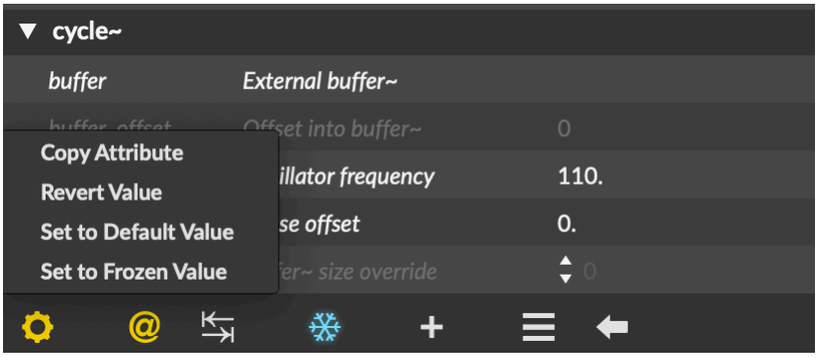
Modifying and Resetting Attributes

The *Modify* icon in the bottom toolbar lets you copy, revert, and reset the value of a given attribute.



The gear icon in the left of the bottom toolbar

Select an attribute, then click on this icon to access several options related to the value of the attribute.



The expanded 'Modify Selected Item' menu.

Menu Item	Description
Copy Attribute	Copy the value of the attribute to the clipboard. Once the value is copied, you can paste the value to another attribute using the *Paste* command from the *Edit* menu.
Revert Value	If the value of the attribute has been modified since the last time the inspector was opened, this option lets you set the attribute back to its original value.
Set to Default Value	Reset the value of the attribute to its default. Not all attributes have a default value, so this option might not be enabled for all attributes.
Set to Frozen Value	If you've frozen the attribute, establishing a new saved value for the attribute, you can use this option to set the attribute to the frozen value.

Object Reference

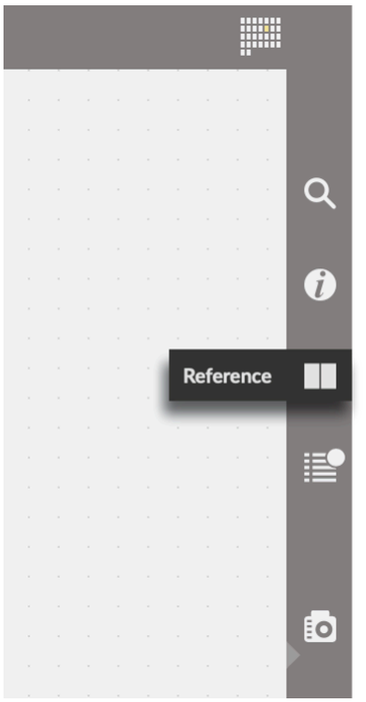
Sidebar 386
Full Reference 392

Every object in Max not only has a dedicated **help file**, but also a reference page. This page completely describes the object's behavior, inlcuding:

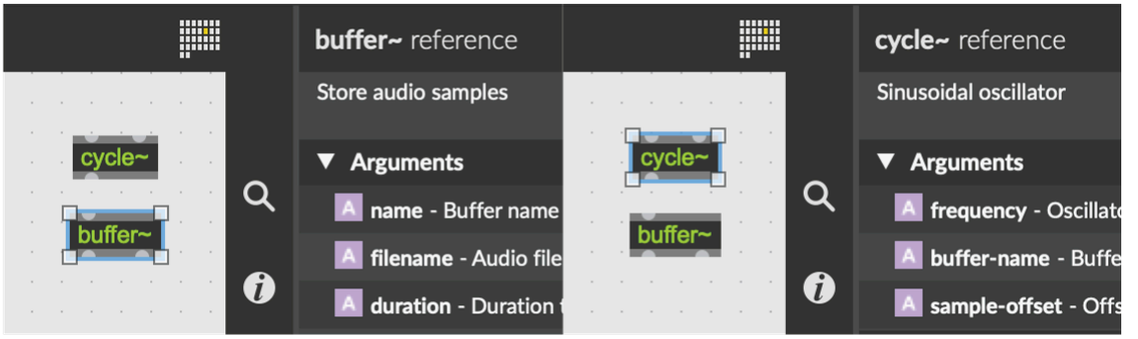
- A short and long description of the object's functionality
- The Arguments and attributes that can configure the object
- The symbols that the object understands
- Other related object and documentation

Sidebar

You can view an abbreviated form of the full reference for an object by clicking on the *Reference* icon in the right sidebar.



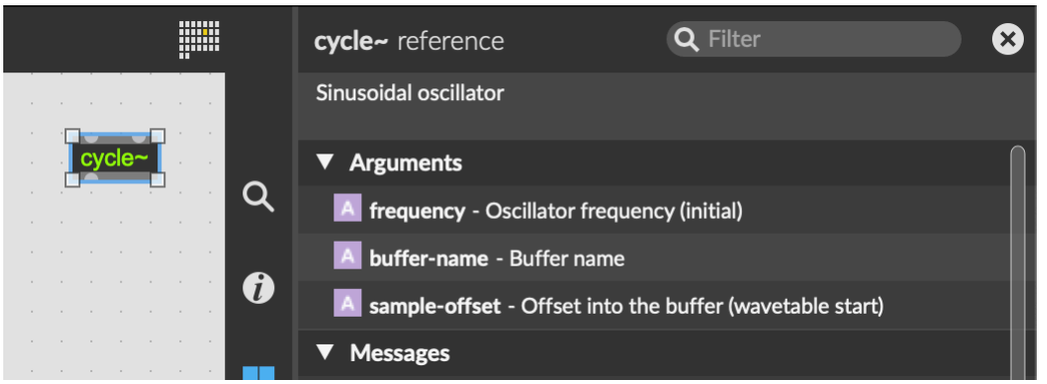
The reference sidebar view will display reference documentation for whichever object is currently selected in the patcher.



With the buffer~ object selected, the sidebar view displays reference documentation for that object.

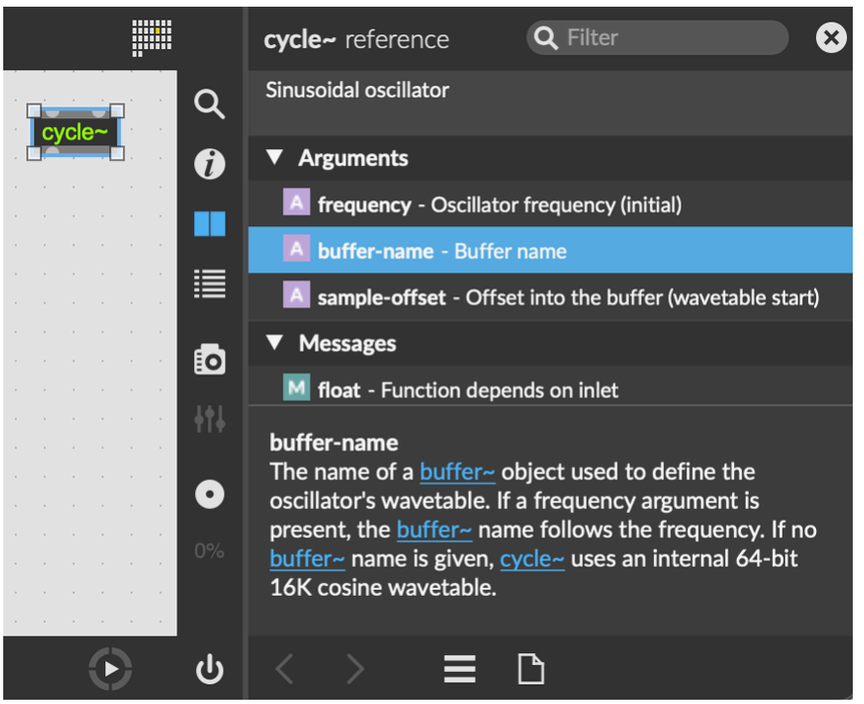
Filtering the sidebar

The top of the sidebar view shows a short description of the object. Using the text field above the description, you can filter the contents of the sidebar view to find the entry that you're looking for.



Arguments

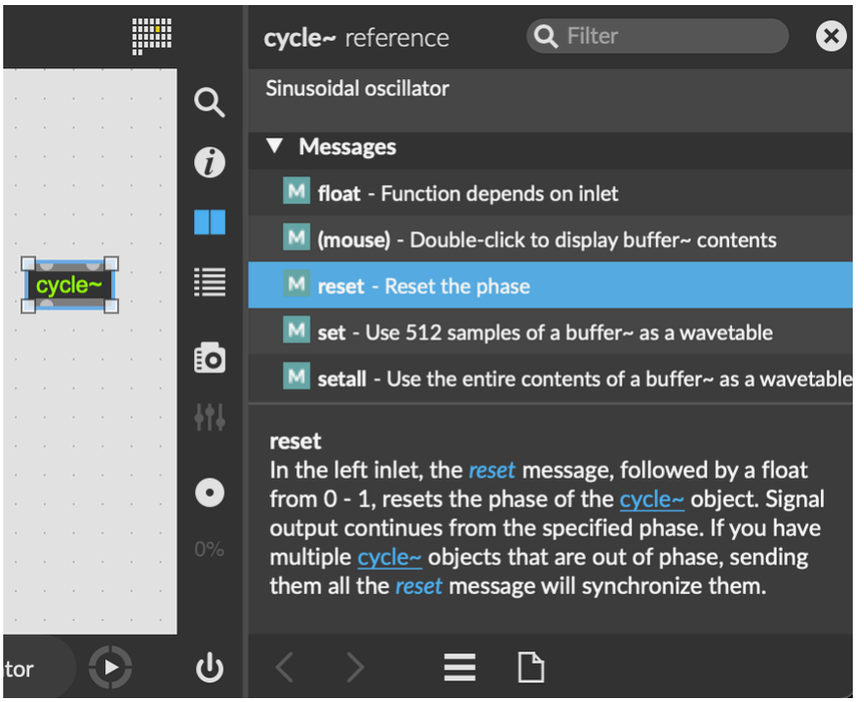
The first section of the reference list view is for arguments. In this section, you can see all of the arguments that the object expects. Next to the name of the argument, you will see a short description of what the argument does. Click on an argument to select it, and a detailed description of that argument will appear at the bottom of the view.



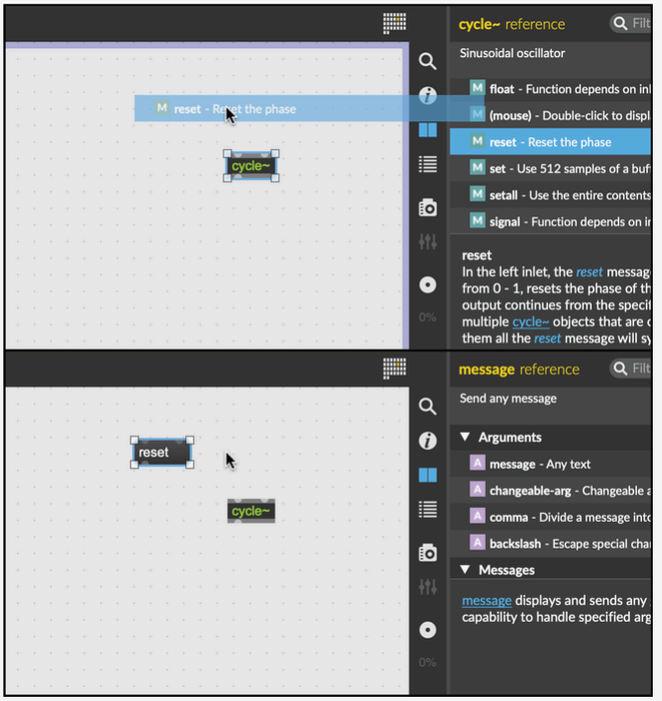
Messages

Under the arguments section, the messages section lists all of the messages that the object can understand. Click on a message to see a detailed description of that message at the bottom of the

view.


cycle~ reference
Filter
Sinusoidal oscillator
Messages
float - Function depends on inlet
(mouse) - Double-click to display buffer~ contents
reset - Reset the phase
set - Use 512 samples of a buffer~ as a wavetable
setall - Use the entire contents of a buffer~ as a wavetable
reset
In the left inlet, the reset message, followed by a float from 0 - 1, resets the phase of the cycle~ object. Signal output continues from the specified phase. If you have multiple cycle~ objects that are out of phase, sending them all the reset message will synchronize them.
cycle~
0%

You can drag and drop a row from the *Messages* section into an unlocked patcher. When you do, a message box will appear in the patcher.



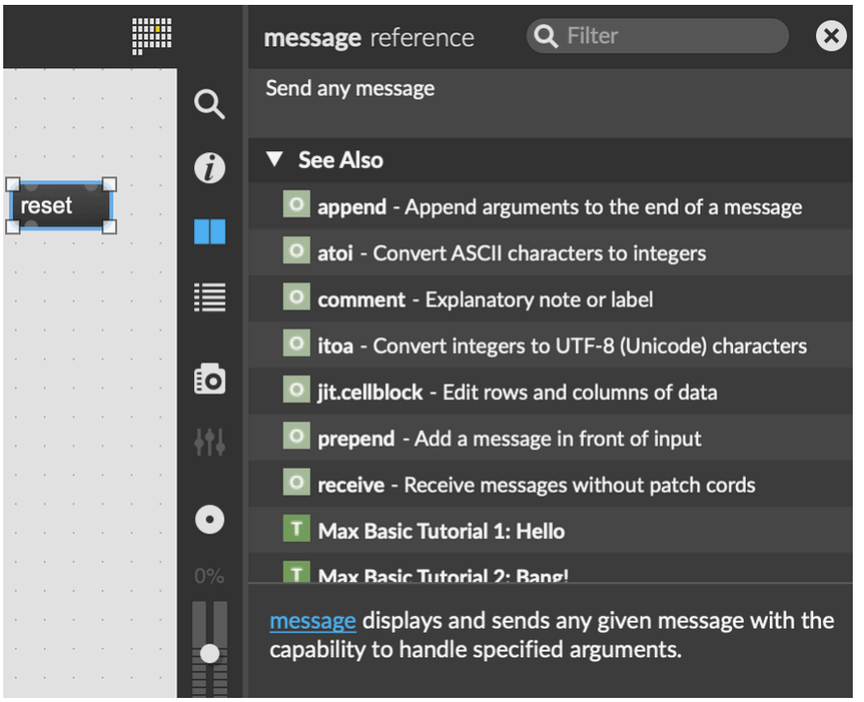
Attributes

The **attributes** section lists all of the selected object's attributes. As in the other sections, click on the attribute to see a detailed description of the attribute in the detail view at the bottom of the sidebar reference.

You can also drag and drop a row from the *Attributes* section into an unlocked patcher. When you do, an attrui object will appear in the patcher, pre-configured to select for the named attribute.

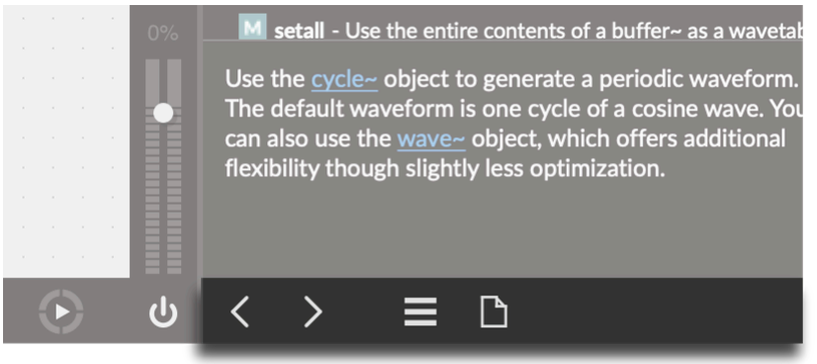
See also

The last section of the sidebar reference is the *See Also* section. This section lists related objects and documentation. You can double-click on any object in this section to open the help file for that object, and you can double-click on any piece of documentation in this section to open it.



Navigation

Use the buttons in the *Navigation Bar* at the bottom of the view to quickly jump to related pages.



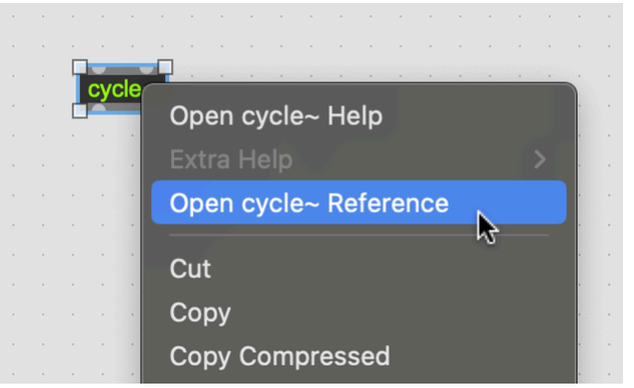
- *Show Previous Object* — Jump to the sidebar reference for the last selected object.
- *Show Next Object* — After pressing *Show Previous Object*, navigate forwards again.
- *Open Full Reference* - Show the **Full Reference** for the selected object.
- *Open Help File* — Open the help file for the selected object.

Full Reference

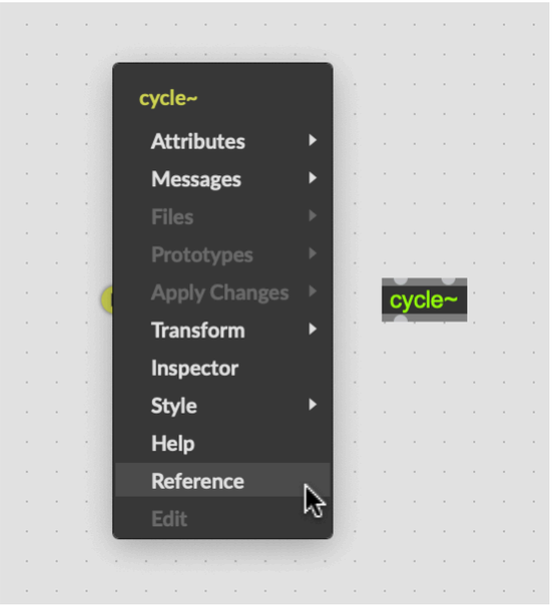
Accessing the full reference

You can access the full reference for a selected object in a variety of ways.

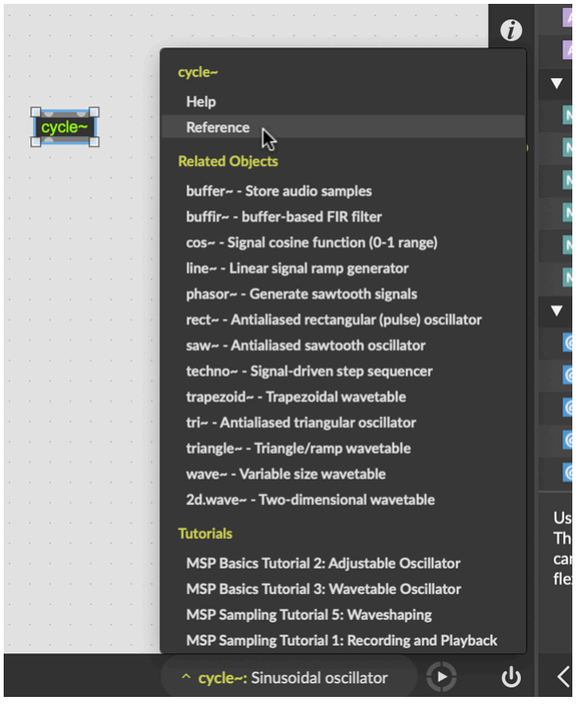
Right-click on the object and select *Open Reference* from the contextual menu.



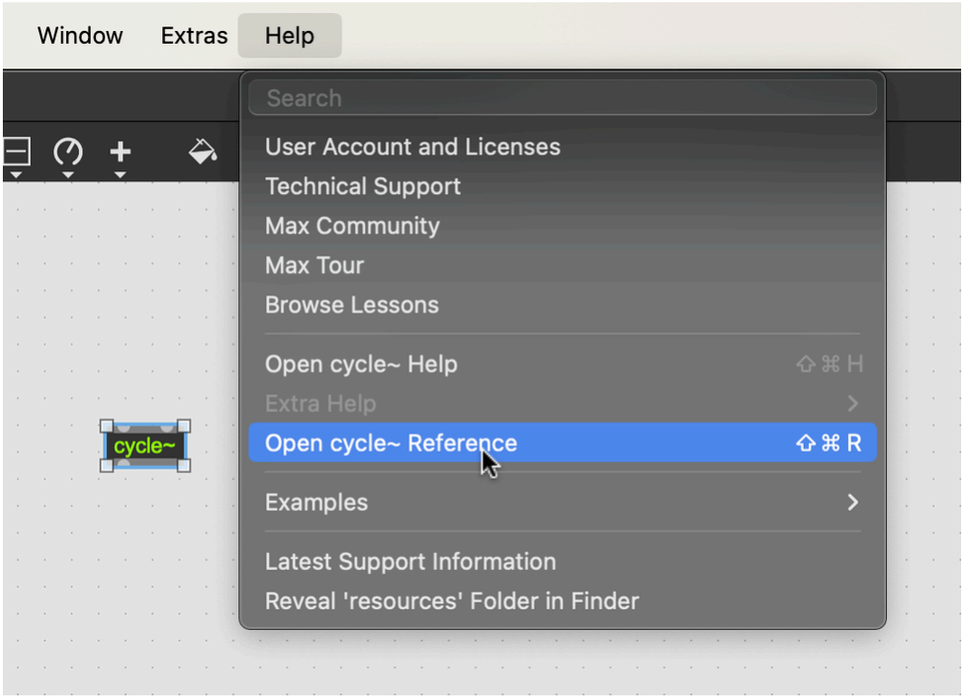
From the object action menu, select *Reference*.



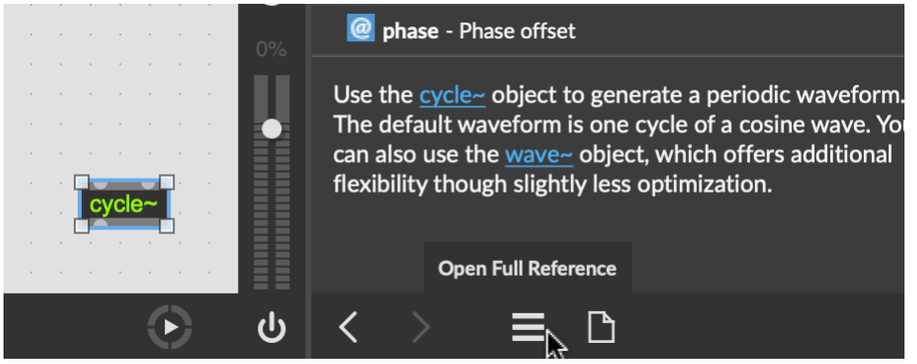
With the object selected, click on the object name in the clue bar, and select *Reference* from the menu.



Select the object, then select *Open Reference* from the *Help* menu.

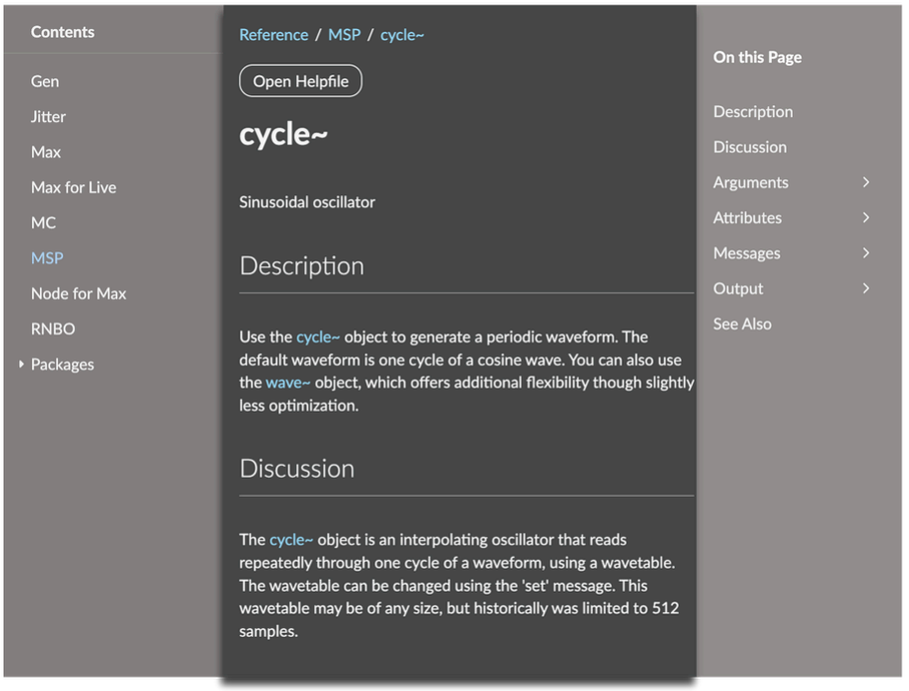


Finally, you can open the reference sidebar, and then click the *Open Full Reference* button in the bottom navigation bar.

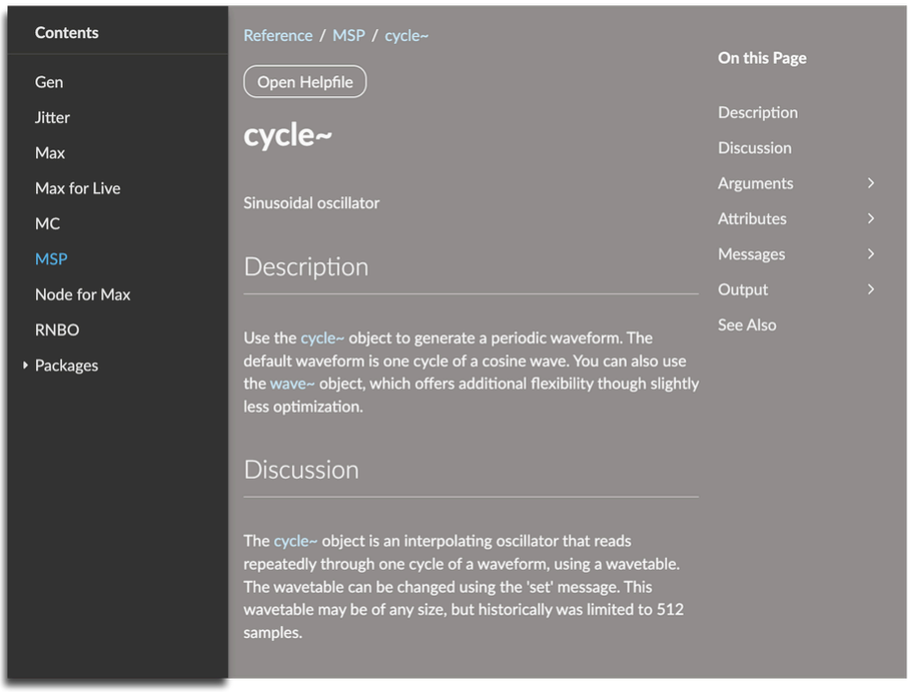


Using the full reference

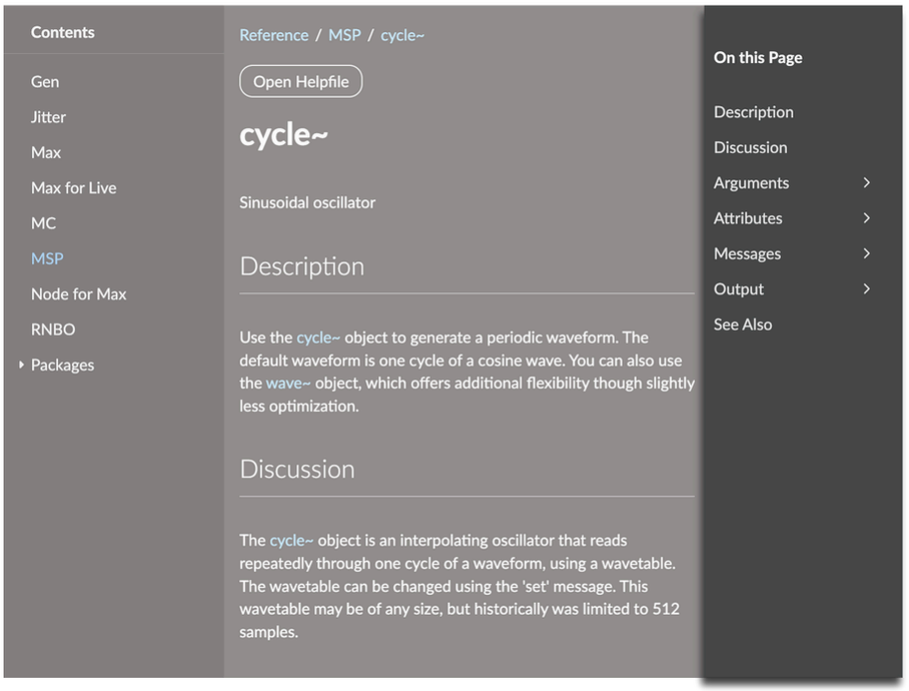
The full reference for an object is an extended version of the abbreviated reference available in the sidebar view. At the top of the reference document, you'll see the name of the object, a short and long description, a longer discussion about the object, and a button to open the help file. At the very top, you'll see breadcrumbs that show the path to the reference file in Max's documentation.



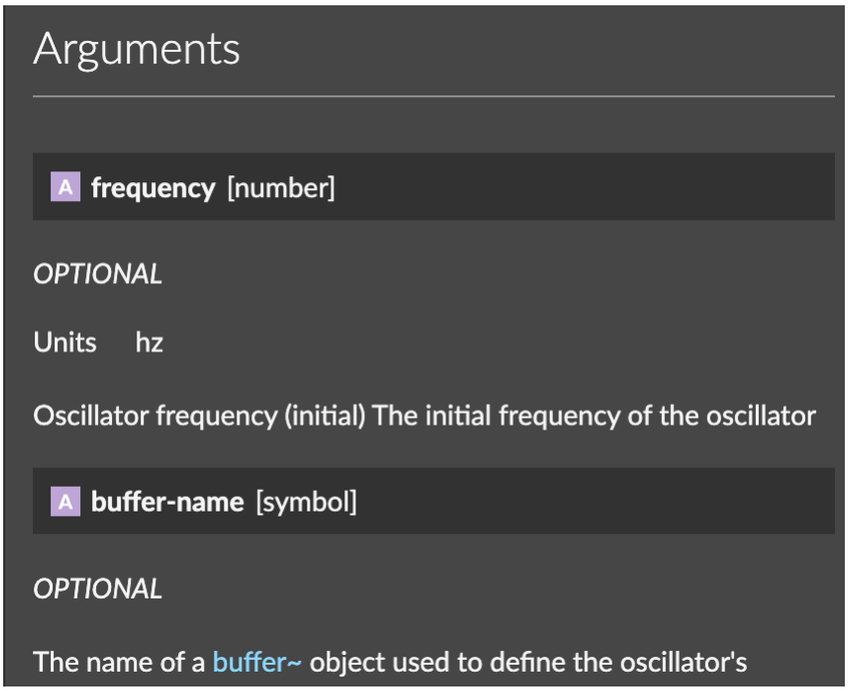
The left side of the page shows the location of this reference document in Max's overall documentation. The primary Max documentation categories are listed here, in addition to a section *Package Documentation* that lists all the documentation for installed packages.



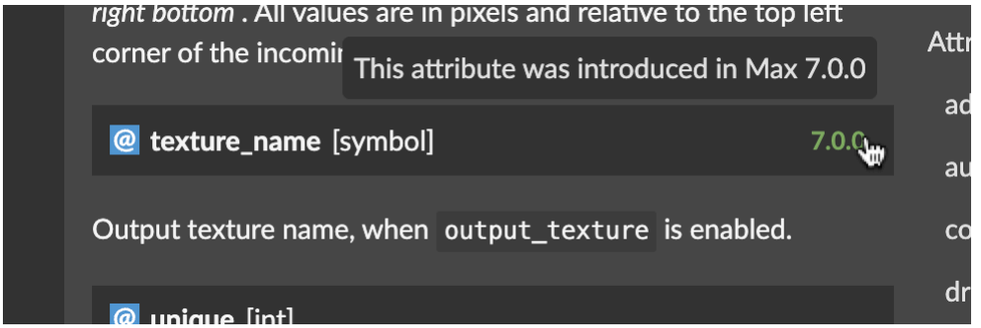
On the right side of the page, you'll see a navigation menu similar to the section categories from the sidebar reference view. From here you can jump to any section on the page, including the documentation for each argument, attribute, and message that the object supports. Additionally, the *Output* section describes what messages or signals the object will send out.



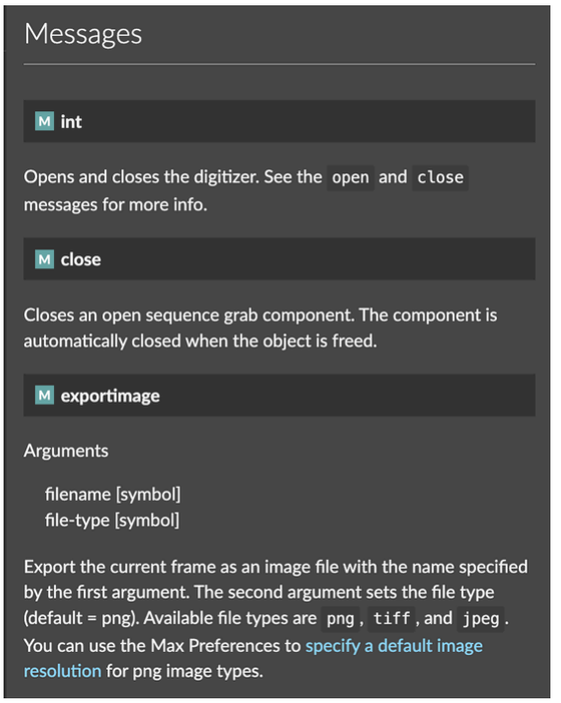
In the *Arguments* section, you'll see a detailed description of each argument. In addition to a description, you'll see the text `OPTIONAL` if the argument is optional, and you'll see the expected type of the argument as well. If the type is `[number]`, it means that the argument can be an int or a float.



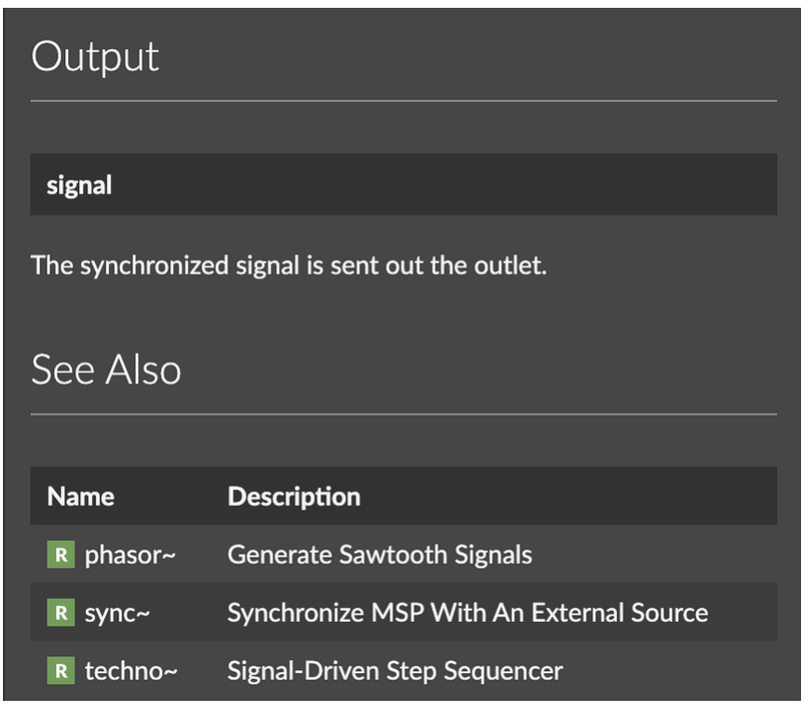
The *Attributes* section lists attributes in a similar way. Note that for some attributes, you may see a special label indicating the version of Max in which this attribute was introduced.



The *Messages* section lists all of the messages to which the object responds. A message will have the special symbol `(mouse)` to indicate how the object will respond to mouse clicks. The symbol `signal` indicates how the object will handle signal inputs.



Finally, the *Output* section will describe what kinds of messages and signals the object generates. This optional section is most common for signal objects.



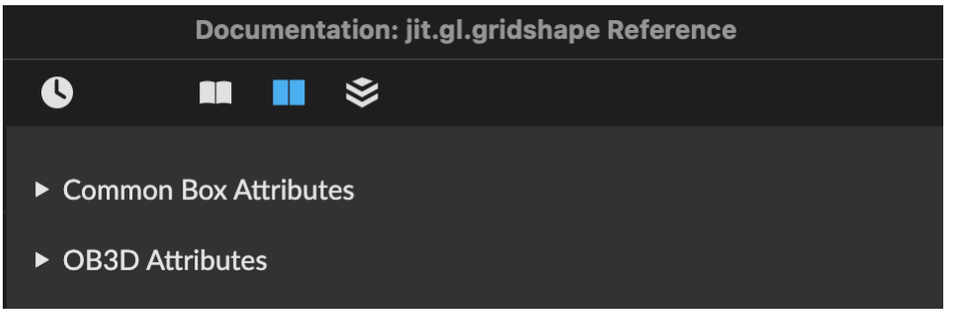
Object parent classes

The full reference for an object documents every message and attribute for that object. Some objects have many, many messages and attributes, especially objects that have a **parent class**.

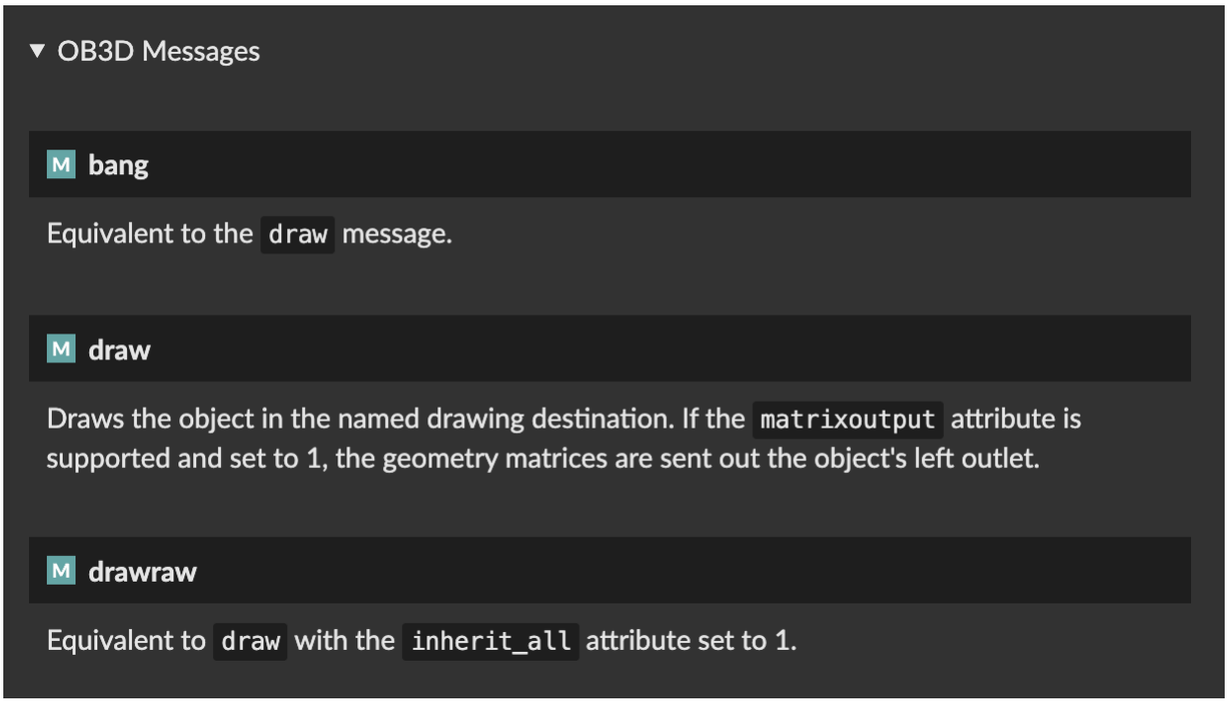
Max objects don't have a strict notion of inheritance like you might find in object-oriented programming languages like C++. However, certain Max object do have a parent class from which the inherit many common messages and attributes. For example, the jit.gl.gridshape object inherits from the *Common* object class and the *OB3D* object class.

- **Common** - The class that all objects with an *object box* inherit from. Adjust things like the font, background color, and the annotation.
- **OB3D** - The parent class for all objects that manage an object in a 3D computer graphics scene. Attributes let you control things like the matrix transform and the color.

In the reference page for an object like this, you'll see a list of object parent classes with a disclosure triangle next to each.



Click on the disclosure triangle to see the attributes or messages that the current object inherited from the given parent class.



Some of the messages that all objects from the OB3D object class will respond to

Preferences

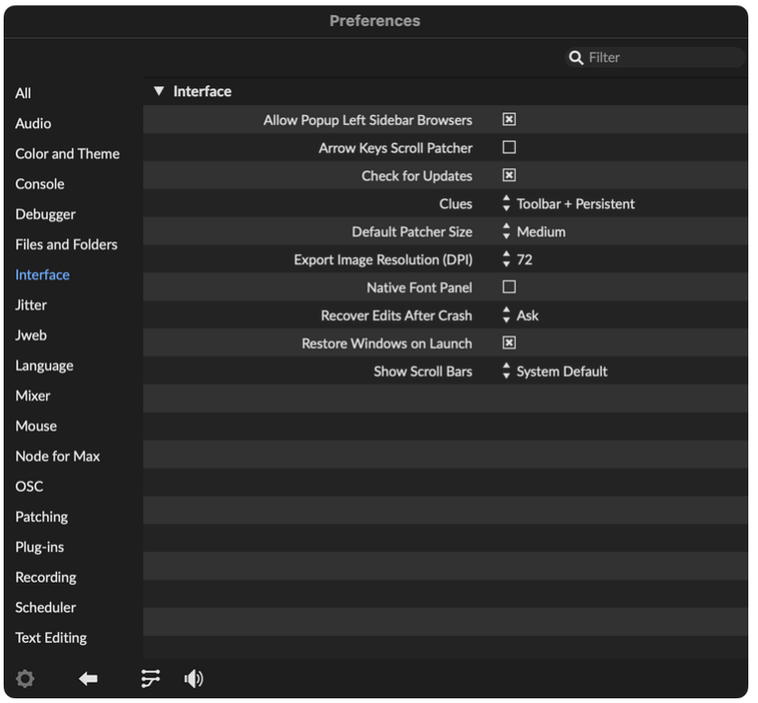
Audio	403
Color and Theme	404
Console	405
Debugger	405
Files and Folders	405
Interface	406
Jitter	407
jweb	408
Language	408
Mixer	408
Mouse Wheel	409
Node for Max	409
OSC	409
Patching	410
Plugins	412
RNBO	412
Recording	412
Scheduler	413
Text Editing	413
Preferences Window Toolbar	414

Use the **Preferences** window to control the behavior of the whole application, including audio driver settings, the default appearance of the patcher window, the behavior of the Max interface, scheduler parameters, and more. On Mac, *Preferences...* ⌘, (macOS) appears in the application (Max) menu. On Windows, *Preferences...* CTRL , (Windows) appears in the Options menu.

- Audio - Configure audio I/O, sample rate, and other audio performance preferences
- Color and Theme - Choose a preset color scheme
- Console - Configure the Max Console window
- Debugger - Set the debug event queue size
- Files and Folders - Configure default folders

- Interface - Customize the user interface
- Jitter - Select the graphics and video engines
- Jweb - Configure integrated web browsing
- Language - Localize the user interface
- Mixer - Configure the audio mixer
- Mouse Wheel - Enable the mouse wheel for zooming
- Node for Max - Set options for using Node
- OSC - Open Sound Control network setup
- Patching - Customize your patching experience
- Plugins - Set up plug-in scanning
- RNBO - Configure RNBO startup and logging
- Recording - Choose the audio quick recording format
- Scheduler - Customize scheduler performance settings
- Text Editing - Customize text editing

Preferences Toolbar - Using the Preferences window toolbar



The preferences window with the Interface tab active

Audio

Audio Driver	Choose an audio driver from the menu. Mac users will typically use *Core Audio* while Windows user will use *MME* (is this right?) or *ASIO*. The *NonRealTime* driver permits you to capture audio processing to a sound file. For more information on the NonRealTime driver, refer to Recording. To disable audio, choose *None* as the driver.
Input Device	Choose a device for audio input. For the selected audio driver, the Input Device menu will display the available audio input devices.
Output Device	Choose a device for audio output. For the selected audio driver, the Input Device menu will display the available audio output devices.
Other Driver Options	Depending on the selected audio driver, there may be more options available below the Output Device.

Sampling Rate	Choose a sampling rate from the menu. Depending on the selected audio driver and hardware devices, the menu may display different available sampling rates. Typical available sampling rates are 44100, 48000, 88200, and 96000 Hz.
I/O Vector Size	Set the number of samples received and set to the audio driver at one time. Smaller values may be more computationally expensive but typically result in lower I/O latency.
Signal Vector Size	Set the number of samples computed at one time. Smaller values may be more computationally expensive. The maximum signal vector size is equal to the current I/O vector size. Very small vector sizes (1, 2) may not work properly.
Overdrive	Set whether the Max scheduler runs in a separate high-priority thread. Overdrive is required for accurate timing.
Scheduler in Audio Interrupt	Set whether the Max scheduler runs synchronously with audio processing, before each signal vector. When enabled, scheduler events are sample-accurate with respect to audio in most cases.
CPU Limit	Establish whether audio processing should be skipped if the computation exceeds a set percentage of the available CPU. The default setting of 0 does not limit the amount of audio processing.

Color and Theme

Color Theme	Select an available Theme that configures colors used for the user interface. Themes also supply default colors for the patcher and objects.
Follow Live Theme	Select how Max responds to color theme changes in Live. When set to `On` the color theme selected in Live will be used when Max is launched from Live to edit a Max for Live device. When set to `Persist`, the last color theme selected in Live when Max launched to edit a Max For Live device will be used in Max when it is launched directly as an application. When to `Off` the Live theme will never be used in Max, potentially leading to color differences in the appearance of a device in Live and Max.

Syntax Color Theme	Select a theme for patcher syntax coloring that overrides the default colors in the chosen theme. When set to *Theme Default* the default theme colors are used for syntax coloring. Note that the syntax color theme is reset to *Theme Default* when you choose a different color theme.
Syntax Coloring	When enabled, text in object boxes will be colorized. The colors used will reflect the curent Color Theme or Syntax Color Theme.

Console

Max Console Dequeue Chunk Size	The maximum number of lines posted in the Max Console per update. Increasing the value above the default (128) will make the console more responsive but could slow down redrawing elsewhere.
Max Console Font Name	Choose a font for the console from the menu or open the font panel. You can also change the console font and size by choosing Show Fonts from the Object menu when the Max console is the frontmost window.
Max Console Font Size	Size of the font used in the Max console.
Max Console Queue Size	The total number of lines to the Max console that can be posted by objects such as print before an update occurs.

Debugger

Illustration Mode Event Queue Size	When using Illustration Mode or the debugger, this value, if non-zero, limits the number of pending events stored from actions such as a running metro object, moving a slider, or incoming MIDI.

Files and Folders

Add Patchers to the Search Path on Save	If enabled, patcher files outside the search path will be added to the search path and become visible in the File Browser.
Default Folder for Max for Live Device Projects	Newly created Max for Live projects will be saved to this folder by default.
Default Folder for Projects	Newly created Projects will be saved to this folder by default.
Default Patcher Template	The selected Template will be used each time you create a patcher using New Patcher in the File menu. Any template can be used via the New From Template submenu.
Save Dependency Paths	When enabled, Max saves the full paths for some dependencies in patcher files. Max can use these paths as a fallback if a file isn't currently in the search path. Disabling this option is useful for source control applications, so that the contents of a patcher file doesn't change based on the computer or user where it was last edited.
URL Proxy	Enter a URL to use as a proxy service for web access. This is used by the maxurl object as well as other web-based Max features such as the search sidebar.

Interface

Arrow Keys Scroll Patcher	When enabled, you can scroll a window using the keyboard arrow keys.
Clue Bar Shown by Default	When enabled, the Clue Bar will be shown for newly opened patcher windows. Changing this preference will not change the Clue Bar visibility in existing windows.
Clues in Toolbar	When set to *Enabled*, clues will be shown in the bottom toolbar and disappear after a few seconds, permitting you to use the documentation menu if shown. When set to *Enabled (Hide on Mouse Over)*, the clue text will disappear when the

	cursor enters the bottom toolbar. To show the documentation menu in this case, press the ? or ^ keys while the clue is visible. Clues are always shown in the Clue Bar instead of the bottom toolbar if the Clue Bar is currently visible.
Check for Updates Automatically	When enabled, Max will alert you if new software updates are available.
Export Image Resolution (DPI)	Sets the DPI (dots per inch) resolution for PNG images exported from Max via the Export Image... command in the File menu. Supports 72, 96, 150, and 300 DPI.
Native Font Panel (Mac only)	When enabled, uses the OS font panel instead of the one provided by Max.
Recover Edits After Crash	Sets how edits to a patcher are restored after a crash. When set to *Always*, previous patcher edits will be restored when Max is restarted. When set to *Never*, no edits will be restored. If set to *Ask*, you'll be prompted at startup whether to restore patcher edits.
Restore Windows on Launch	If enabled, Max will attempt to re-open all previously open patcher windows when launched.
Show Scroll Bars	Sets the window scroll bar style. When set to *Dynamically*, scroll bars appear when using a mouse wheel or mousing over the edge of a window. When set to *Always*, scroll bars appear if content in a window can be scrolled. When set to *System Default*, the system-wide user preference determines the scroll bar style.
Space Bar Accepts Autocompletion	If enabled, a space bar will accept the currently highlighted autocompletion value. If disabled, the space bar will insert a space into an object box without inserting the autocompletion text.

Jitter

Default Cache Size	Set the default cache size (in gigabytes) for jit.movie and jit.playlist objects when instantiated using viddll the engine.
Default GL Context	Set the name of the default GL context. When a context is set, any GL or animation object that doesn’t have a user provided context will be added to the

	default context. A jit.world, jit.window, or jit.pwindow object of the same name as the default must also be instantiated.
Graphics Engine	Select the Graphics Engine
Video Engine	Select the Video Engine

jweb

Remote Debugging Port	Sets the remote debugging port for jweb / CEF. After a port is set, restart Max and open a patcher with a jweb object that has web content loaded. Then open a web browser, navigate to `http://localhost:[port number]`, and a link to each jweb instance in max will apear for debugging.
Force jweb Render Mode	When set to *Onscreen* or *Offscreen*, all jweb objects will render in one of these two modes, no matter how they are configured individually. Mostly useful for debugging. Requires a Max restart to take effect.

Language

Language	Select the language used in the interface. The menu will only show *English (en)* unless Max was installed with the available Japanese installer.

Mixer

Enable Mixer Crossfade (Adds Latency)	When set to *On*, Max will crossfade between new and old versions of your editing operations when audio is turned on.
Enable Mixer Parallel Processing	When enabled, audio in each top-level patchers will be processed in its own thread.
Mixer Crossfade Latency	Set the latency of the mixer when crossfading editor operation in milliseconds.

Mixer Crossfade Ramp Time	Sets the cross-fade time (in milliseconds) used during an editing crossfade (if enabled).

Mouse Wheel

Mouse Wheel Zoom Direction	If set to *Standard*, zooming follows the same direction as mouse wheel scrolling. If set to *Reversed*, zooming increases in the opposite direction to scrolling.
Mouse Wheel Zoom Sensitivity	Sets the mouse scroll wheel sensitivity when used for zooming in and out of a patcher window. Set to 0 to disable zooming with the mouse wheel.

Node for Max

Enable Node for Max Logging	When enabled, Node for Max objects will write `console.log` messages to a log file.
Debug Log Filename	File name of the log file that Node for Max will write, if Node for Max logging is enabled.
Debug Log Folder	Folder in which Node for Max will write its log file, if Node for Max logging is enabled.

OSC

Send OSC Default	Enable / disable sending OSC. Can be overridden by each patcher in the patcher inspector.
Default Remote UDP Address	Remote UDP address to send OSC to when enabled. Can be overridden by each patcher in the patcher inspector.
Default Remote UDP Port	Remote UDP port to send OSC to when enabled. Can be overridden by each patcher in the patcher inspector.

Receive OSC Default	Enable / disable listening for OSC. Can be overridden by each patcher in the patcher inspector.
Default Local UDP Port	Local UDP port to listen for OSC on when enabled. Can be overridden by each patcher in the patcher inspector.
Enable OSCQuery Server	Enable / disable an http server to serve OSCQuery requests. Individual patchers can add or remove themselves from the list of OSCQuery sources.
OSCQuery Port	Local http port to listen for OSCQuery requests on.
OSC Address Prefix Type	What kind of prefix to add, if any, to OSC addresses generated by Max.
OSC Address Prefix	The prefix to add to OSC addresses generated by Max.
OSC Value To Send	Whether to send raw (scaled) values, normalized values if they exist, or both.
Use /param Prefix For Parameters	Whether to add the prefix `/param` to addresses generated by Max.
OSC Enabled Default	Enable / disable OSC by default for individual objects. Can be overridden in the inspector for each OSC-capable object.

Patching

Assistance Bubbles	When enabled, bubbles appear to describe an object's inlets and outlets when you move the cursor over them. When disabled, the descriptions appear in the Clue Bar.
Box Snap Margin	Sets the horizontal snap margin in pixels. The snap margin is the space within which the position or size of an object will automatically be changed to match nearby objects.
Curved Patch Cords	When disabled, patch cords will be straight and have sharp corners. See examples below of the appearance of patch cords depending on the setting of Curved Patch Cords.



Curved patch cords enabled



Curved patch cords disabled

Disable Window Animation (Windows only)	Disabling window animation may improve real-time performance of the application.
Edit Operations Trigger Loadbang	When enabled, any loadbang or loadmess objects present in what is pasted or restored when performing an Undo or Redo operation will produce output.
Enable Patching Mechanics	When enabled, shortcuts as defined in Patching Mechanics will be available.
Exit on Last Window Closed (Windows only)	When enabled, Max will quit when the last application window is closed. When disabled, the Max console window will appear once all windows are closed to avoid quitting the application.
High DPI Rendering (Windows only)	On compatible Windows OS versions, enables a higher resolution display for the application which may involve using more than one physical screen pixel per logical pixel. This can improve the appearance of fonts and other graphics at the possible cost of decreated overall graphics performance.
Keep Duplicated Objects in View	When enabled, duplicated objects will be placed inside the current visible area of the patcher window. When disabled, objects will be dulplicated relative to original object position and may end up outside the visible area of the patcher window.
Layout Bubbles	When enabled, coordinate information will appear in a bubble when moving or resizing an object. When disable, this information will appear in the Clue Bar.
Mouse Position Determines Auto-Connection	If enabled, a new auto-connected object will be created just below or above an inlet or outlet that is closest to the current mouse location. If disabled, new auto-connected objects will always connect to the first inlet or outlet, though the object will be created at the mouse location.

Patch Cord Wiggle Time (ms)	Sets the time, in milliseconds, of the patch cord "wiggle" animation during patching. Set this value to 0 to disable wiggling.
Prioritize Patch Cords	When enabled, patch cords will be selected if they are over a box.
Segmented Patch Cords	When enabled, clicking on outlet starts a segmented patch cord. When disabled, you need to shift-click in order to make a segmented cord.

Plugins

Audio Plug-In Scanning	If set to *Minimal/Fast*, audio plug-in scanning will look for audio plug-ins files, but will not test or verify them. If set to *Complete/Slow*, audio plug-in scanning will load and verify each plug-in before making it available to the application. Complete scanning will also scan through "shell" plug-ins that contain multiple other plug-ins.
Full Scan	Click the *scan* button to start a manual scan for audio plug-ins.

RNBO

Start RNBO Server on Launch	When enabled, the RNBO server will start when Max is first launched. When disabled, the RNBO server will not start until the first rnbo~ object is created.
RNBO Log Filename	File name of the log file that RNBO will write. If unspecified, RNBO will not write to a log file.
RNBO Log Folder	Folder in which RNBO will write its log file, if RNBO logging is enabled.
RNBO Log Level	Set to *Debug* to log all messages, set to *Error* to log only error messages.

Recording

Global Record Format (WAV)	The bit depth and numeric type for audio files recorded with Global Record
Global Record Red Button	When enabled, the Global Record button will be red while recording is active.

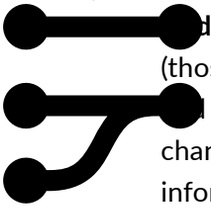
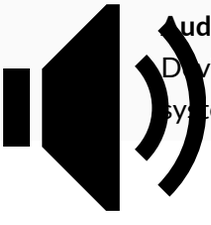
Scheduler

Event Interval (ms)	The approximate minimum time, or throttle limit, (in milliseconds) between handling of low-priority events. For advanced use.
Overdrive	When enabled, time-critical tasks in the Max scheduler run at a higher priority, increasing timing accuracy.
Prioritize Scheduler Accuracy Over CPU Usage	When enabled, improves the accuracy of the scheduler potentially at the cost of increased CPU usage. Only relevant when Overdrive is on and Scheduler in Audio Interrupt is off.
Poll Throttle	The number of events that are handled together in one tick of the scheduler. For advanced use.
Queue Throttle	The number of events that are handled together at low priority. For advanced use.
Redraw Queue Throttle	Scheduler performance parameter that sets the maximum number of patcher UI update events to process at a time. Lower values can lead to more processing power available to other low-priority Max processes, and higher values make the user interface more responsive (especially when using many bpatchers). For advanced use.
Refresh Rate (fps)	The rate limit (in fps) at which Max will update the UI for user interface objects.
Scheduler Slop (ms)	Scheduler performance parameter that, roughly, balances accuracy with CPU efficiency. For advanced use.

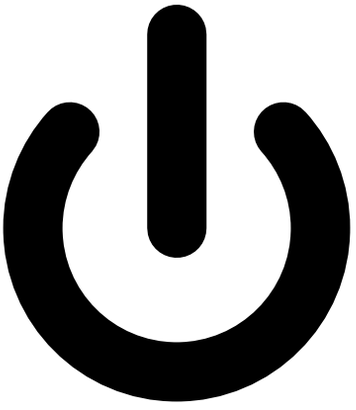
Text Editing

Always Use External Text Editor	When enabled, Max will use an external text editor you specify using the External Text Editor preference when editing text files for objects such as coll and dict.
Edit Box Text on Click	When enabled, one click will begin editing the text in an object box, rather than two clicks.
External Text Editor	Choose an application to use as an external text editor.
Typing Edits Selected Box	When enabled, typing with an object selected will automatically start editing that object's text.

Preferences Window Toolbar

	**Modify Selected Item** displays a menu for acting on the selected preference. *Copy Attribute* copies the current value of the preference to the clipboard. *Revert Value*, which will be enabled if you've changed a preference, restores the value that was set when you first opened the Preferences window.
	**Show Preferences Folder** switches to the Finder (Mac) or Explorer (Windows) and opens the Settings folder where Max stores its preferences. The file `maxpreferences.maxpref` contains the settings edited in the Preferences window.
	**Audio I/O Mappings** opens an editor window where you can assign virtual channels (those used in adc~ and dac~) to real channels on the currently selected audio input and output devices. If you're using stereo audio input and output devices, virtual channels 1 and 2 are assigned to real channels 1 and 2 by default. For more information, refer to Audio Channels.
	**Audio Driver Setup** opens Audio MIDI Setup (Mac) or the Sounds and Audio Devices Properties panel (Windows) to configure your audio setup at the operating system level.

]> **Audio On/Off** turns audio on or off globally.



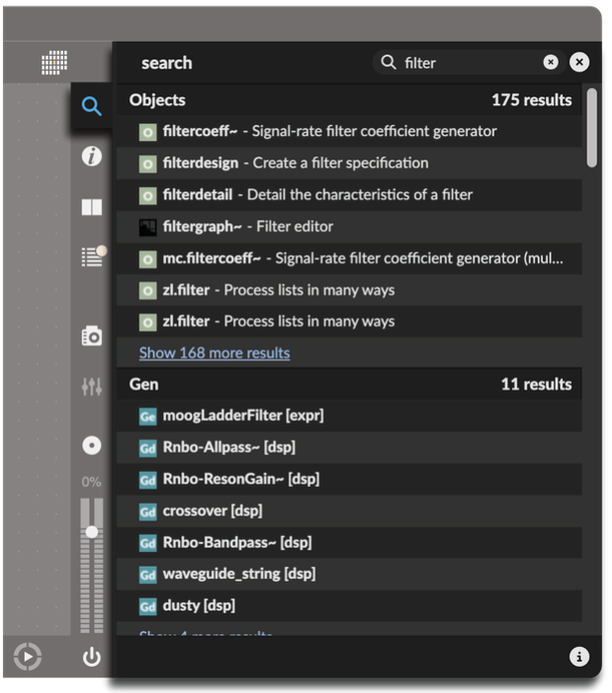
Search

Using Search 417

Search in Max is a convenient way to search across many different areas all from a single text prompt. Searching for the text "filter" will return signal processing objects, examples of filtering, and online articles about filter design. Search in Max includes:

- Max, Gen, and RNBO objects.
- User Guide, installed Package documentation, and API Reference
- Patchers and Snippets in the current search path
- Plugins
- Packages in the Package Manager
- Forum posts

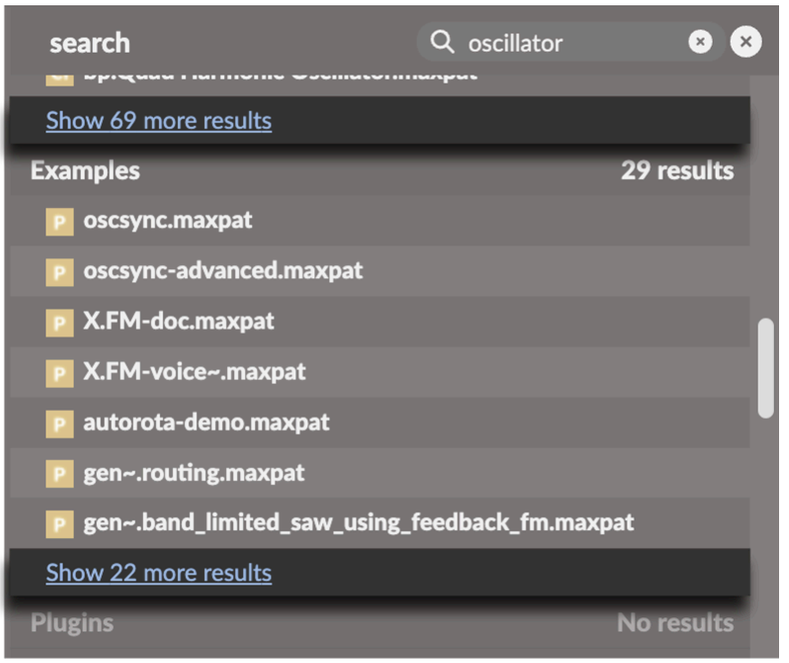
Click on the *Search* icon in the right toolbar to access search in the **Search Sidebar**.



Search in one place and find resources from the in-app documentation, your local files, and online articles.

Using Search

Start typing in the search box in the top-right to initiate a search. You'll see results grouped by category, including objects, documentation, and examples. If more than a certain number of results appear, you can click *Show more results* at the bottom each category to reveal more results.



When there's more than a few results per category, you can click to show more results.

Viewing results

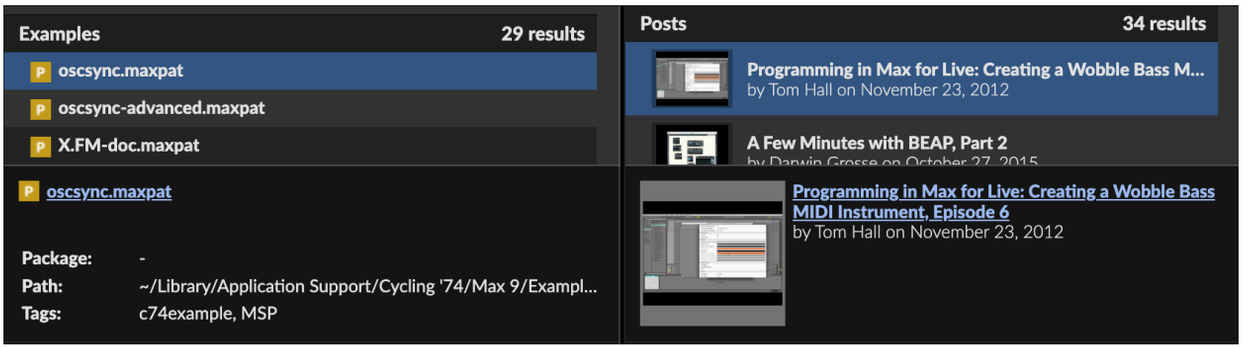
Double-click on any search result to view it. The way Max opens a particular result depends on the kind of result. For example, Max patchers will open in a new patcher window, and forum posts will open a new web page in your default web browser.

Dragging results into your patch

You can add results from certain categories directly to your patch, like *Objects*, *Examples*, and *Patcher & Snippets*. Click and drag a search result into your patch to add it. As mentioned in dragging and dropping, you can hold `option` (macOS) or `alt` (Windows) while dragging to customize how Max adds the resource to your patch.

Search details

At the bottom of the search sidebar, the *Details Pane* shows you more information about each search result. The kinds of results that you see will depend on the type of each result. For example, forum posts will show the author and post date, while Max patcher results will show the path to the `.maxpat` file on your local disk, along with any semantic tags associated with that file.



Max patcher file results (left) will show you the path to the file, along with tags. Post results will show the author and the post date.

Patcher Toolbars

Customizing Toolbars 420
Left toolbar 421
Top toolbar 436
Right toolbar 438
Bottom toolbar 443
Max for Live Window 448
Gen Window 449
RNBO window 452

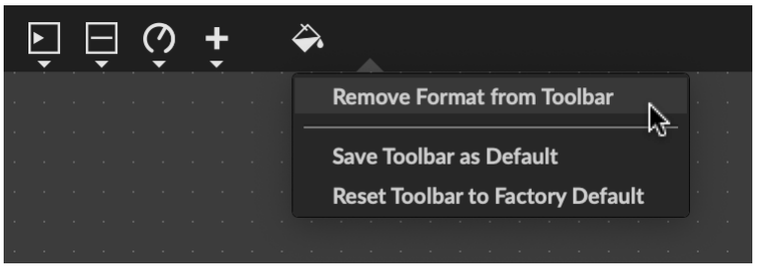
The top, left, right, and bottom of every Max window contain the toolbars. These icons allow you to access built-in Max content, configure the behavior of your patcher, quickly access objects to add to your patcher, and more.

Customizing Toolbars

If you want, you can pin/unpin any of the toolbars by hovering over the toolbar and clicking on the triangular tab that appears in the middle of the toolbar. Hover over the hidden toolbar, near the border of the patcher, to bring the toolbar back.

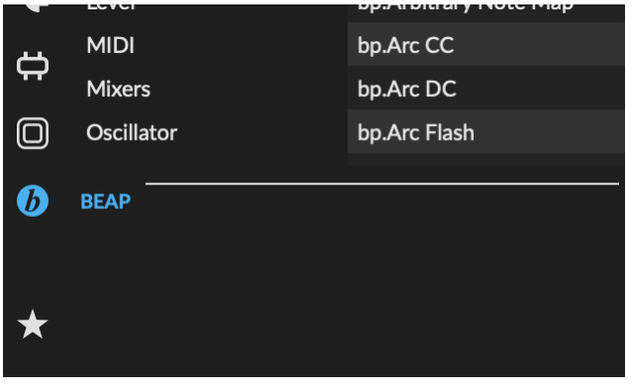


You can also customize each toolbar by adding or removing icons. To remove an icon, simply right-click on it and choose the *Remove* option from the contextual menu.



You can also add icons to your toolbars by right-clicking and selecting an *Add* option from the contextual menu. Each toolbar has different options for which icons can be added.

- **Right Toolbar**- Select *Add Browse Lessons* to add a lesson browser.
- **Bottom Toolbar**- This toolbar can add *Mute* and *Solo* buttons, which mute and solo the audio output of the patcher.
- **Left Toolbar**- The most customizable of all, since you can add a browse icon for any installed package to this toolbar. You can also add an icon to open the Package Manager.



The left toolbar, customized to include an icon for the BEAP package

If you want your toolbars to keep their configuration the next time you open Max, right-click and select *Save Toolbar as Default*. If you want to go back to the original toolbar configuration, select *Reset Toolbar to Factory Default*.

Left toolbar

The left toolbar provides access to different collections of resources that you can use in your patcher. For a more detailed, dedicated view of all the files in Max's search path, use the File Browser.

From top to bottom:

1. **Patcher List View**- List, sort, and filter all of the objects in the current patcher.
2. **Objects**- Browse and filter all the objects in Max, including objects from installed packages.
3. **Audio**- Contains all the audio in Max's search path, with options to filter by name and length.
4. **Video**- Browse all the videos in Max's search path.
5. **Images**- Browse all the images in Max's search path.
6. **Plug-ins**- Displays both VST and Audio Unit plug-ins, as well as Max for Live devices (also known as AMXDs).
7. **Max for Live**- Special snippets and objects for Max for Live device development.
8. **Modules**- Categorized selections of objects and snippets from installed packages
9. **Collections**- Resources from collections as defined in the File Browser.

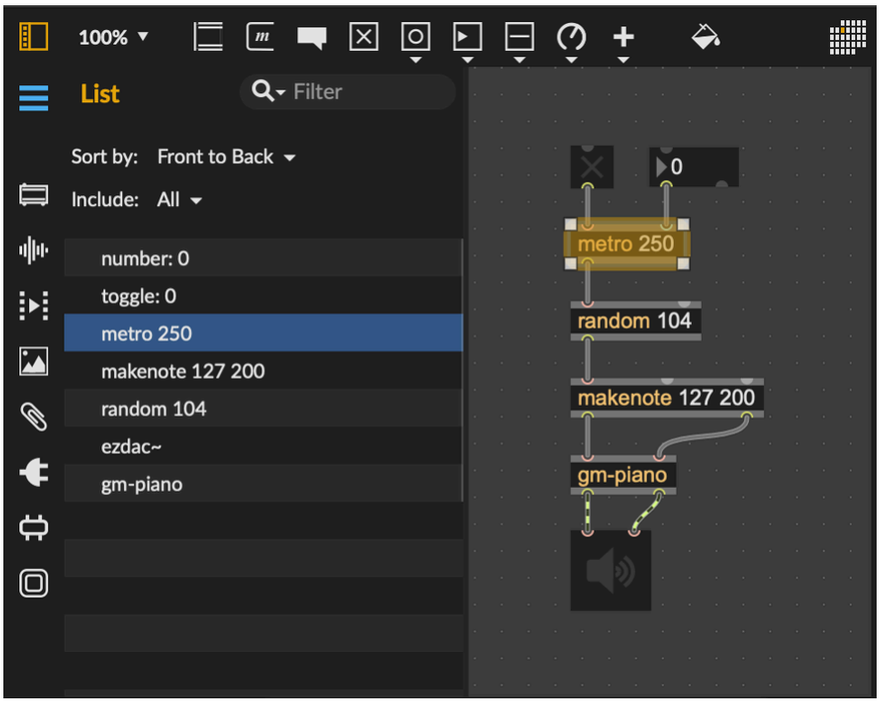
Optionally, you can also add a browser view for *Gen DSP*, *Gen Jitter*, or any installed package.

Patcher List View

The Patcher List View, which only functions when the patcher is unlocked, shows all of the objects in the current patcher. Unlike the patcher view itself, which shows each object in its current patching rectangle, this view simply displays the text of each object in a flat list.

The list view is helpful for locating, selecting, and operating on objects that might otherwise hard to find in complex patchers.

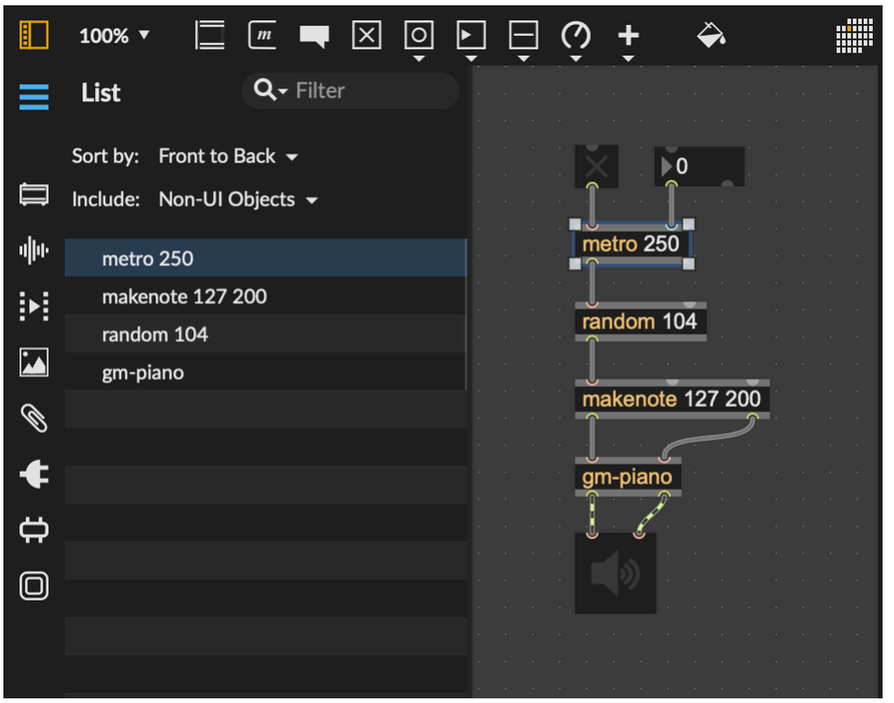
Move the cursor over any element in the list to highlight the associated object, and click to select it.



> Objects within a subpatcher won't be included in the list, but you can double-click on a subpatcher object to open that subpatcher.

Type in the *Filter* text entry at the top of the view to filter for objects matching specific text.

Use the *Sort by* drop-down to change how objects are sorted in the list, and use the *Include* drop-down to filter for *UI Objects*, *Non-UI Objects*, or *All Objects*.



With Non-UI Objects selected, user interface objects like the toggle, numbox, and ezdac~ are all excluded from the object list.

When open, the list view will reflect the current selection in the patcher window. In addition, the list view display of objects with values such as slider or number will update as the values of those objects changes.

Operations on List View Items

- Click any item to select it in the patcher. Shift-click to select multiple items.
- Click on the round **button** that appears at the left edge of a list view item to open the Object Action Menu for the object.



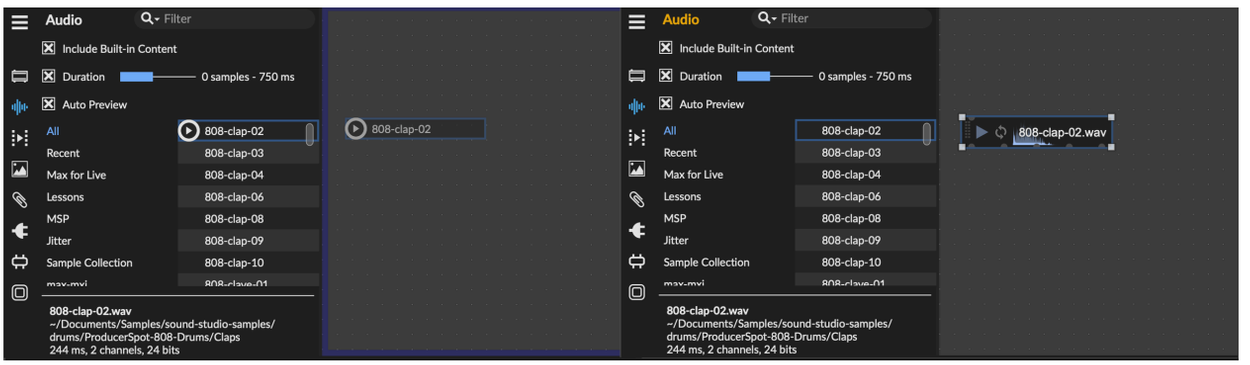
- **Double-click** on any list view item to perform the same action as double-clicking on the item in the patcher would perform. For instance, double-clicking on a patcher will open the object's patcher window.

- Press **return** or enter on any selected object to send that object a `bang` message. For example, selecting a button and pressing return will act as if you clicked on the button.

Dragging and Dropping

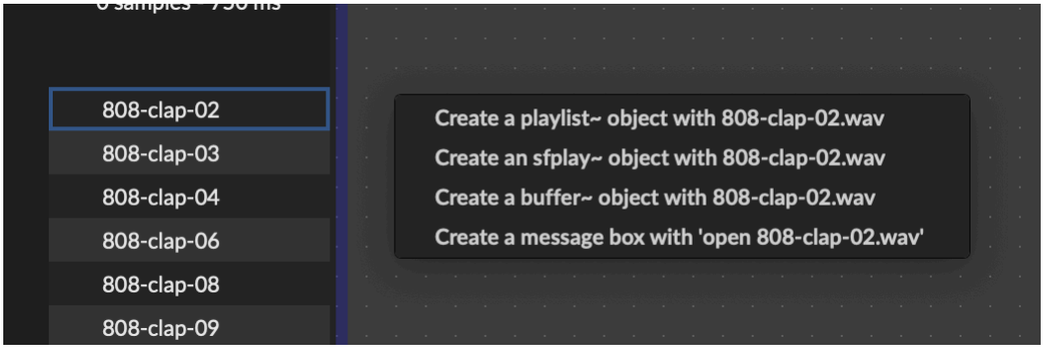
The following browsers, including the Object Browser, Audio Browser, etc., are convenient ways to find resources that you can add to your patch. Once you've found what you're looking for, you can just drag the resource into your patch to add it.

Depending on what kind of resource you're trying to add, the Max patcher will handle the drop in different ways. If you drag an audio file into your patch, Max will create a playlist~ object to play that file back. If you drag in a video file, Max will make a jit.playlist object.



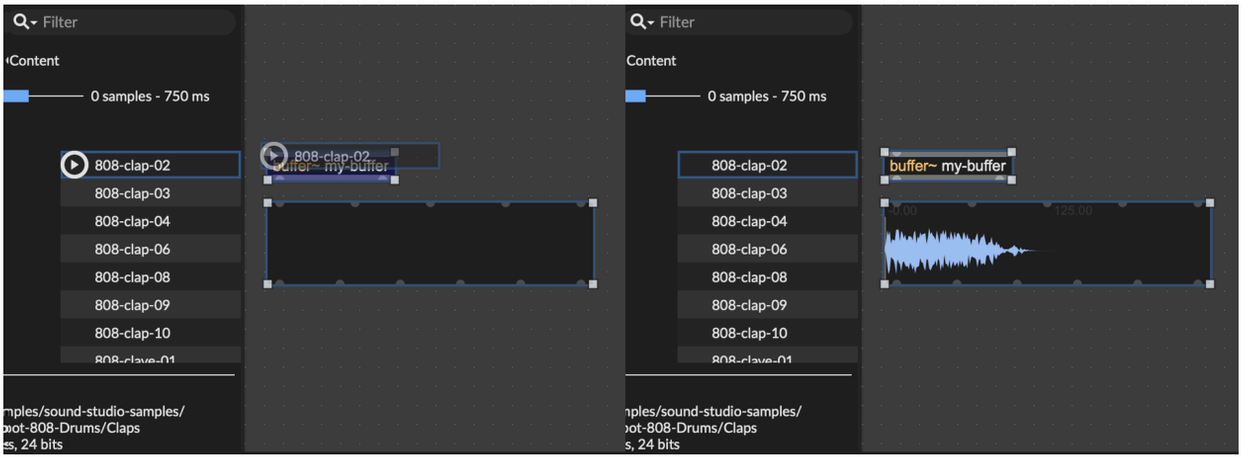
When you drop an audio file into your patch, Max will create and configure a playlist~ object to play back that file.

Some resources can give you multiple options as to how they should be handled. Hold down `Alt` (Windows) or `Option` (macOS) while dragging a resource into the patcher view to see all the available options.



Hold down Alt/Option as you drag an audio file into your patcher to create a playlist~ object, an sfplay~ obect, a buffer~ object, or a message box configured for that audio file.

Finally, many Max object can handle resources of the appropriate kind. For example, if you drag an audio file over a buffer~ object, you'll see a blue border appear inside buffer~, indicating that it can perform a special action when you drop the file. Usually, if an object can load a file in response to an `open` or `read` message, then it can handle a drag-and-drop action with that same file type.

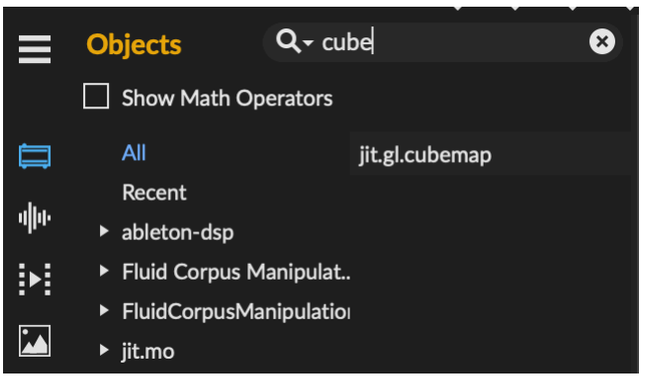


When you drag-and-drop an audio file on top of a buffer~ object, the buffer~ will read the audio file and resize itself to fit.

Object Browser

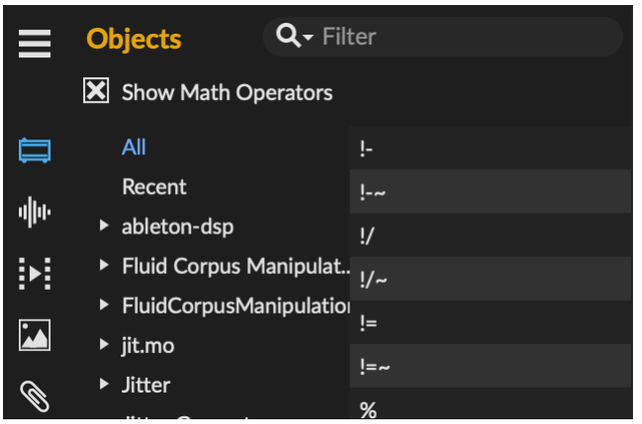
The Object Browser shows you all the objects that Max has to offer, with a couple of controls to make it easier to find the object you're looking for.

Type into the *Filter* text entry at the top to find objects matching specific text.

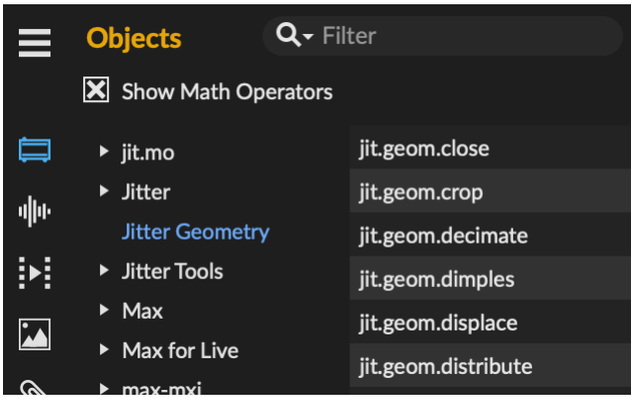


Max has several *Math Operators*, small objects that perform a simple math operation like addition or multiplication. By default, these objects are filtered from view, but you can select the *Show Math*

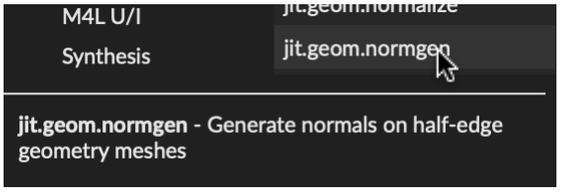
Operators checkbox to reveal them.



The browser view on the left groups objects by package, and then by category within a package. Click on a package to show all objects included in that package, and on a category to show just objects in that category.



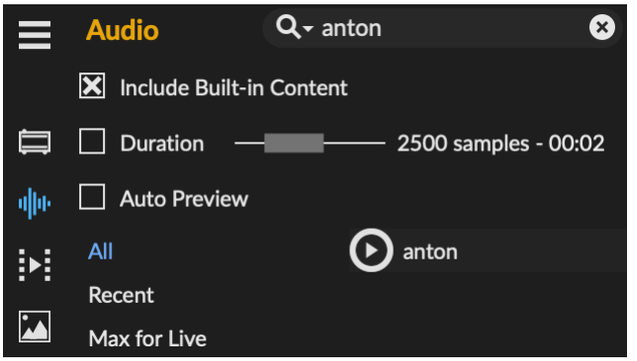
Finally, notice the description view at the bottom of the browser, which will show a short summary when you roll over an object.



Audio Browser

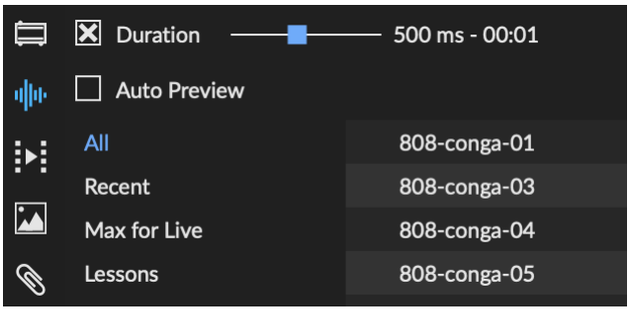
The Audio Browser shows all of the audio samples in Max's search path. There are controls to filter by name and length, and a preview option to audition samples as well.

Type into the *Filter* text entry at the top to find objects matching specific text.



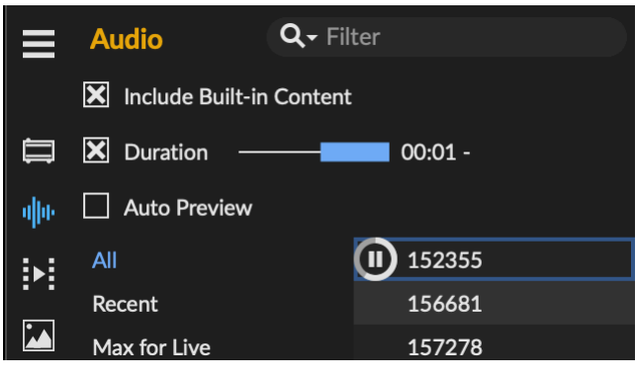
You can deselect *Include Built-in Content* to filter out any content that shipped with Max. This includes content from built-in packages, like *BEAP* and *Jitter*.

With the *Duration* filter active, you can use the slider to select only audio files that fall within a certain length.



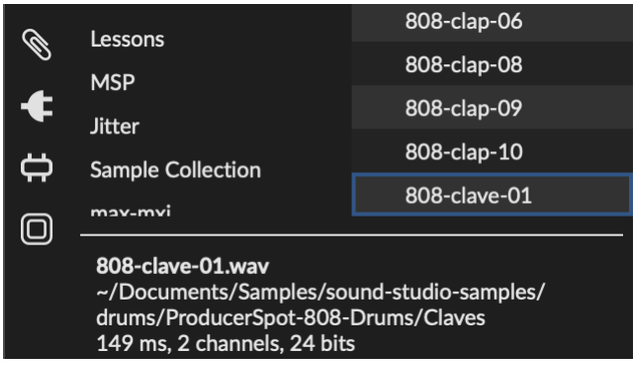
You could use a short duration filter to show only one-shots.

Hover over a sample and click the *Play* triangle to preview the sample. While the sample is playing, the triangle will change to a *Pause* icon that you can use to pause playback.



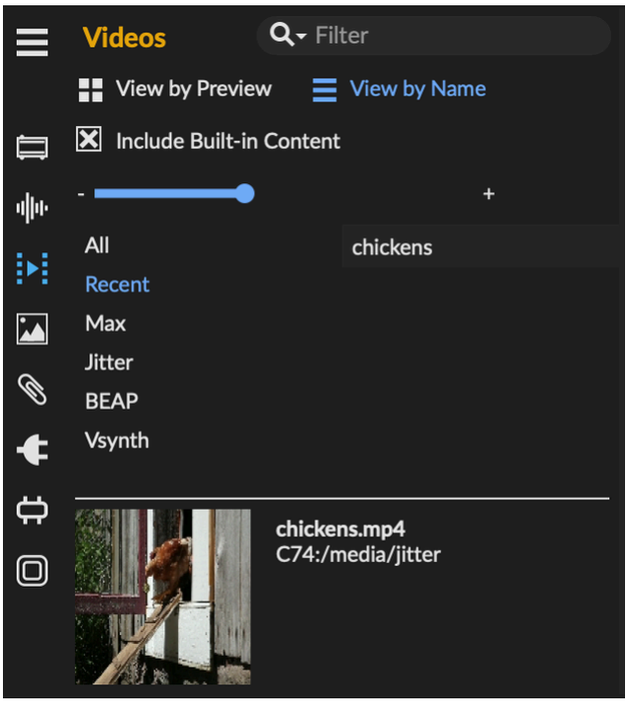
If *Auto Preview* is enabled, each sound sample will start playing as soon as you select it. With this option, you can use the arrow keys to move your selection up and down, quickly auditioning a large number of files.

Finally,the *Description* view at the bottom of the browser will display the name of an audio file, the full path to that file on disk, the length of the file, the number of audio channels in the file, and the bit depth of samples in the file.

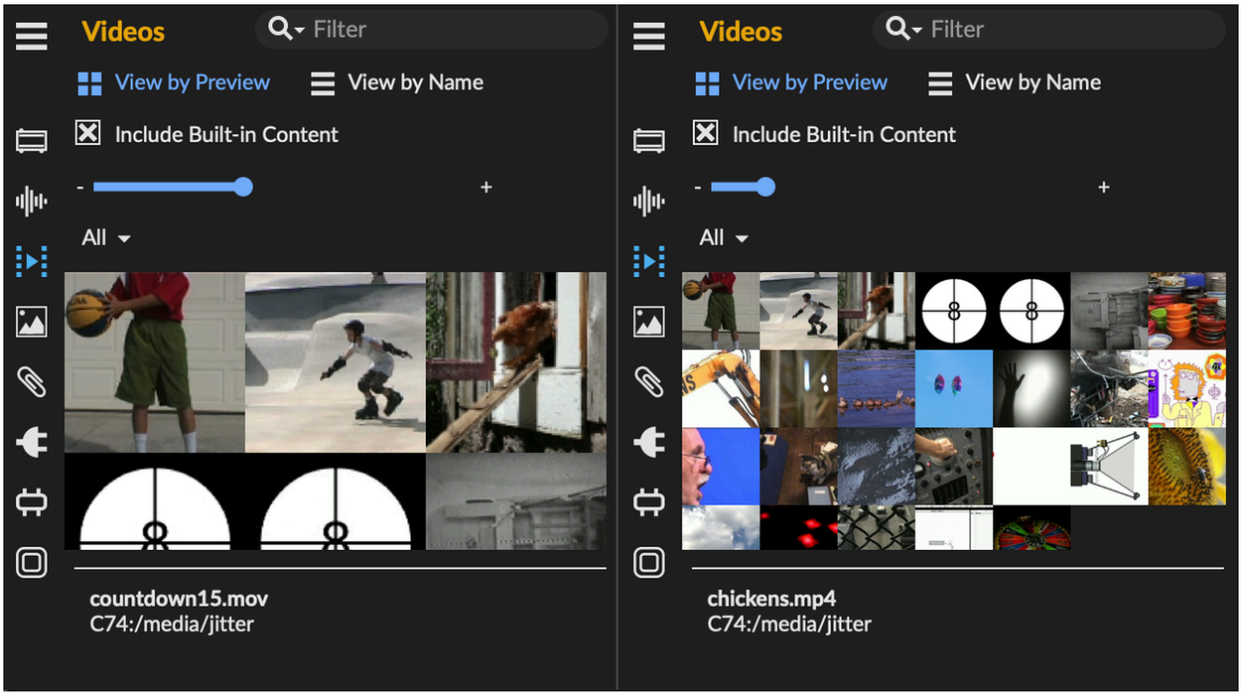


Video Browser

The Video Browser shows all of the video files in Max's search path. There are controls to filter by name, and you can choose whether to view files as a list or by preview thumbnails.



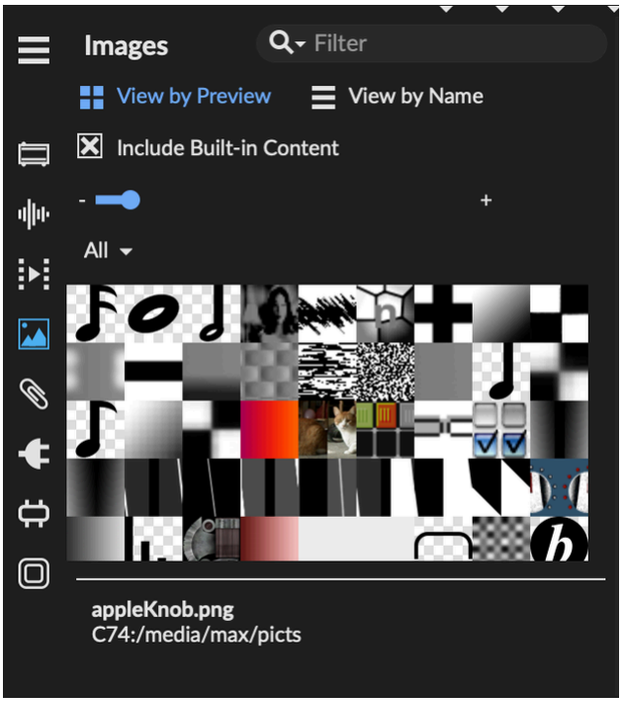
With *View by Preview* enabled, video files will appear in a grid of video thumbnails. Hover over a thumbnail to see the name of the file, and click on it to preview the video. Finally, use the size controller above the thumbnails to adjust the size of the previews in the view.



Change the size of the previews to see more video files at once.

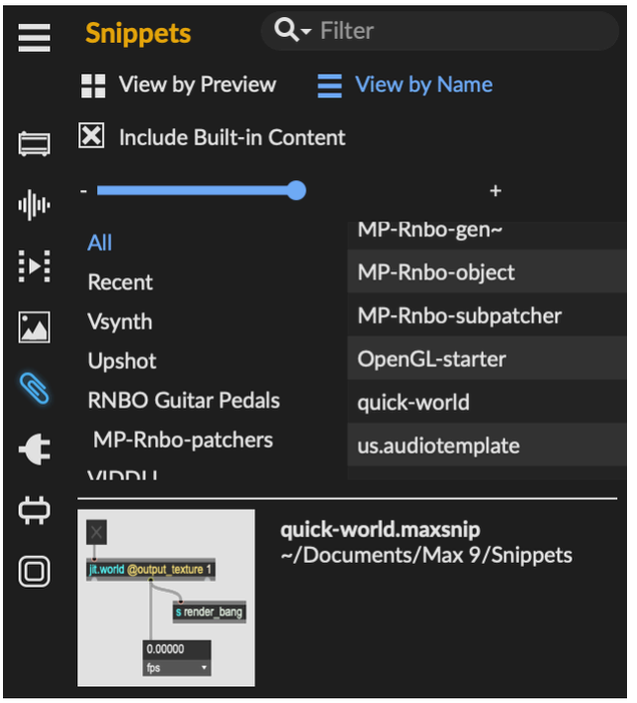
Image Browser

The Image Browser shows all of the image files in Max's search path. Similar to the Video Browser, there are controls to filter by name, and you can choose whether to view files as a list or by preview thumbnails.



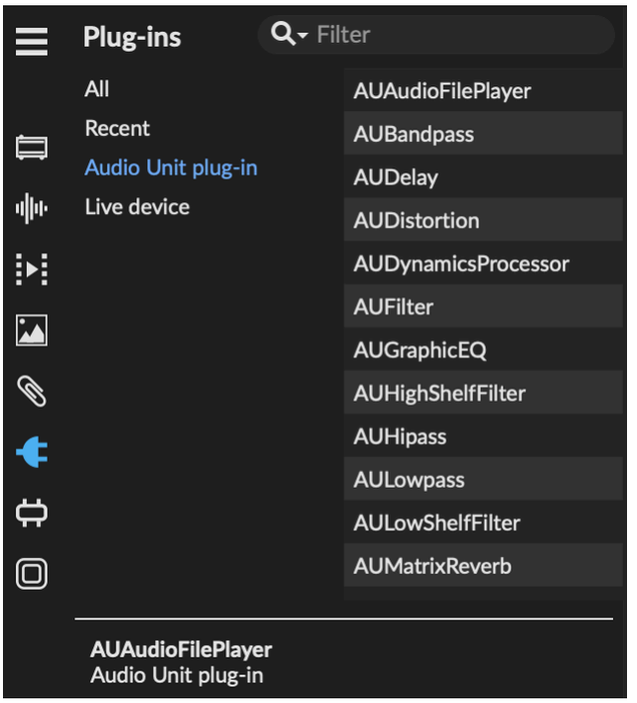
Snippet Browser

The Snippet Browser shows you all of the Snippets that Max has access to. Like the Video and Image Browsers, there are controls to filter by name, and you can choose whether to view files as a list or by preview thumbnails.



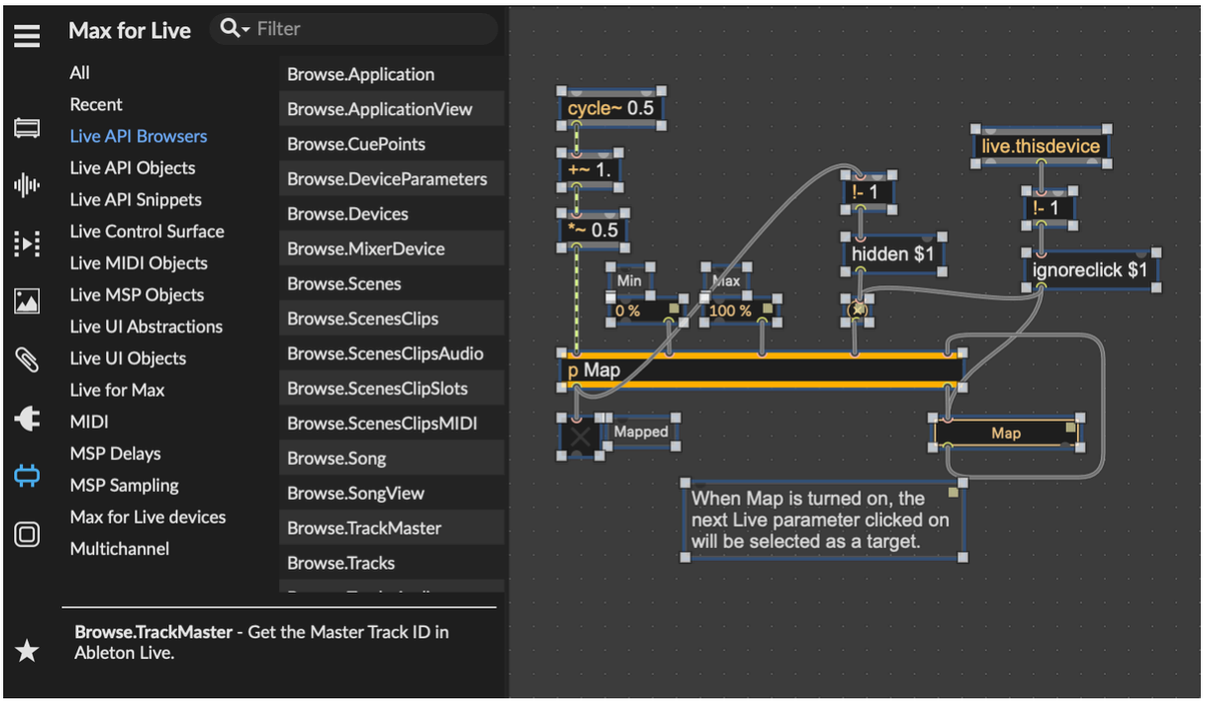
Plug-ins Browser

The Plug-ins Browser shows you both VSTs and Audio Units plug-ins that Max has scanned, as well as any Max for Live devices (AMXDs) in the search path. Use the *Filter* text entry to filter by name, and the *Recent* tab so see plug-ins that you've used recently.



Max for Live Browser

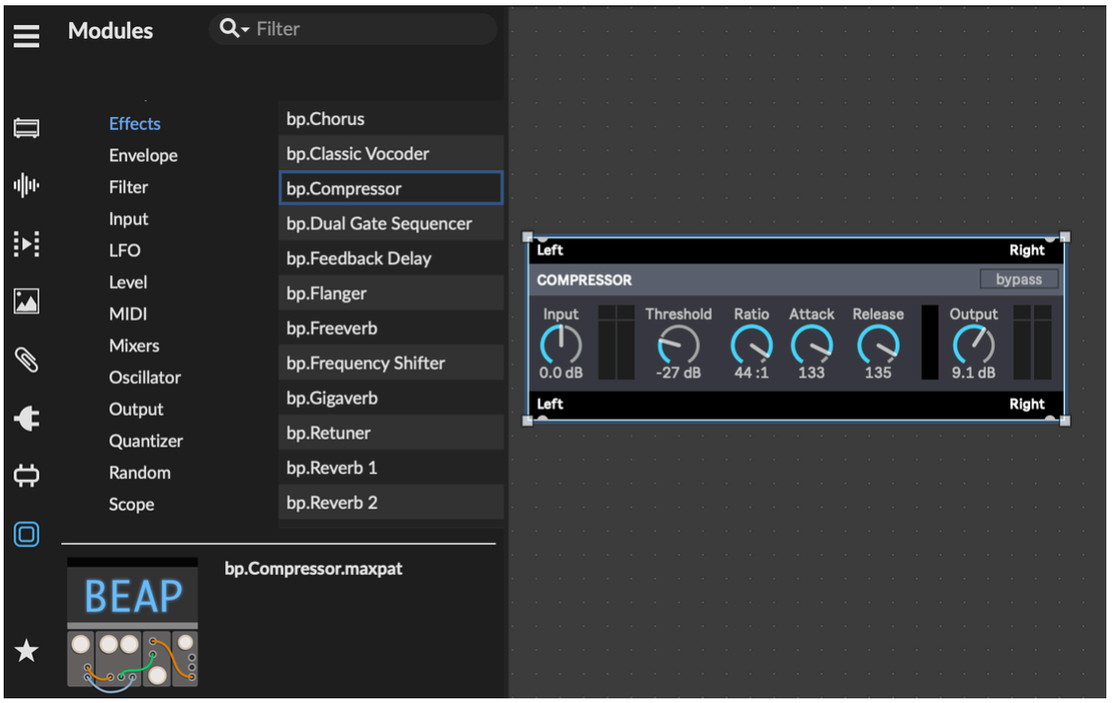
The Max for Live Browser is organized to make Max for Live device development easier by bringing together objects, abstractions, and snippets for common tasks like handling MIDI, voice allocation, and audio synthesis. The snippets and abstractions under *Live API Browsers*, *Live API Snippets* and *Live API Objects* are especially useful, since these provide solutions to many of the common programming challenges you'll run into when working with the Live API



One of the helpful snippets available in the Max for Live Browser

Module Browser

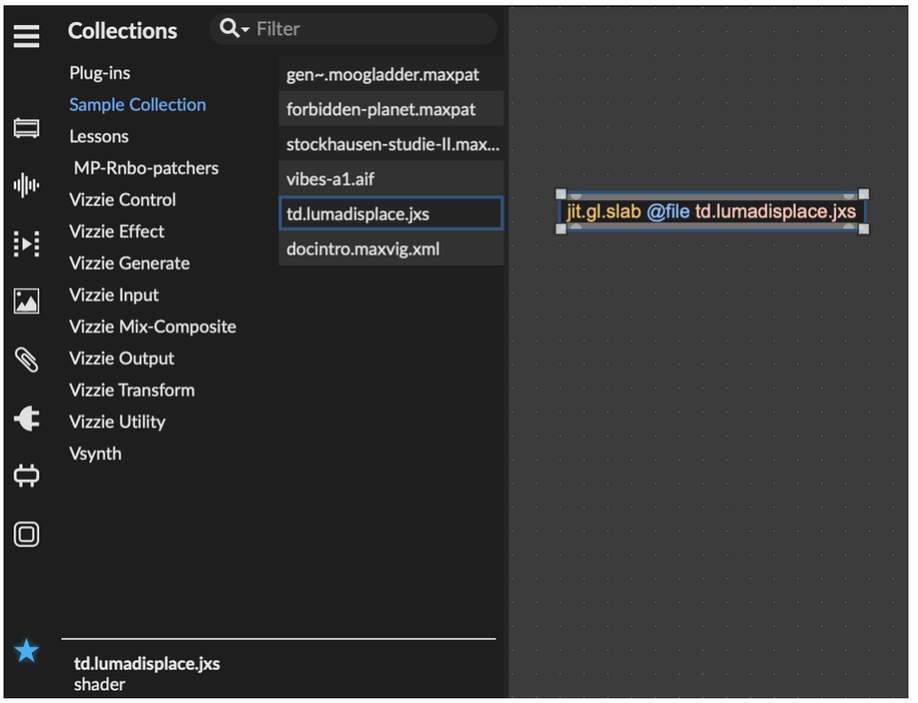
The Modules Browser presents snippets and abstractions from built-in and third party packages that you've installed. It's a powerful way to package authors to give you quick access to the best of what their tools are capable of. For example, the built-in BEAP package has categorized abstractions related to audio analysis, effects, and quantization, all organized into modules.



The BEAP compressor effect, loaded from the Modules Browser.

Collection Browser

The Collection Browser shows you all the collections that you've defined using the File Browser. Collections give you total control over how your work is grouped, since a single collections can contain any kind of resource, including video clips, text files, and JavaScript code. You can define collections specific to a project or workflow, and use the Collections Browser to quickly access resources in that collection.



The built-in collection 'Sample Collection' demonstrates what a collection can do, and includes files of multiple media types.

Top toolbar

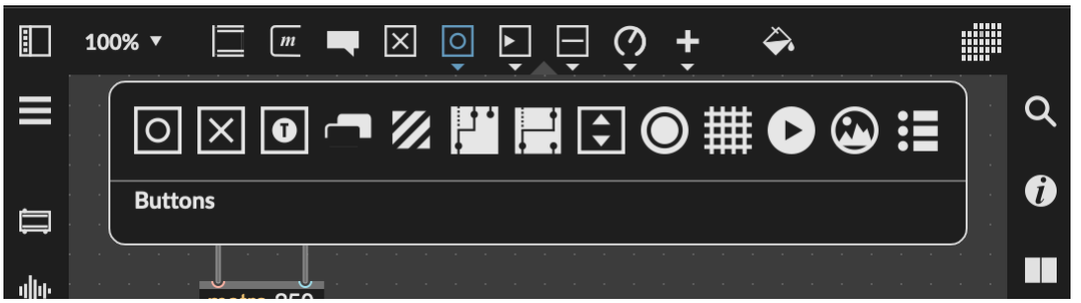
The top toolbar contains various controls for changing the appearance of your patcher, along with quick access to many of the user interface objects available in Max.



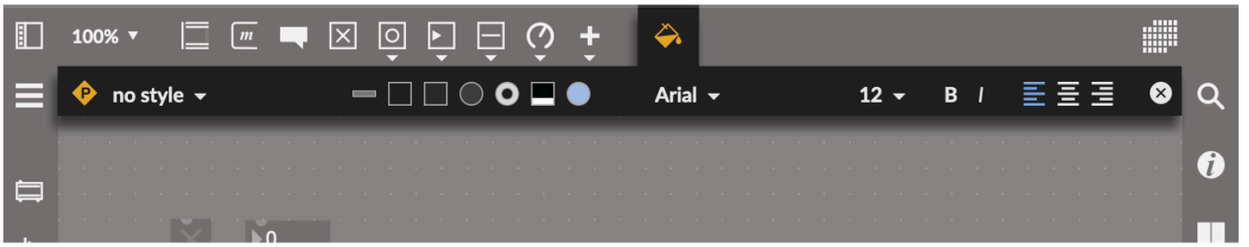
The *Show Browser* icon reveals the most recent Browser view in the Left Toolbar.

The *Zoom Dropdown* lets you adjust the level of zoom in your Max patcher. You can also adjust the zoom level by pressing ⌘= (macOS) or CTRL= (Windows) to zoom in, and ⌘- (macOS)or CTRL - (Windows) to zoom out.

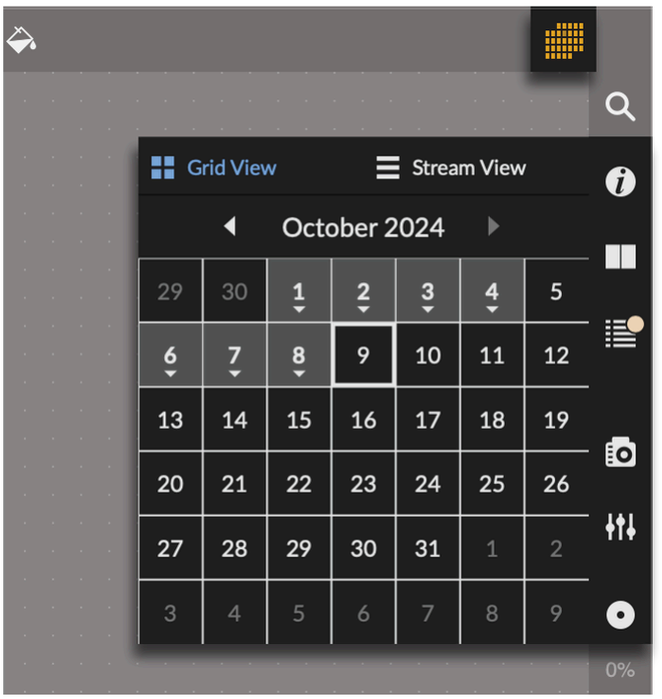
The *UI Object Palette* gives you quick access to Max UI objects, organized by function. Click on icons with a disclosure triangle to see a selection of options within that category.



The Format Palette button lets you adjust the style and appearance of objects in your patcher.

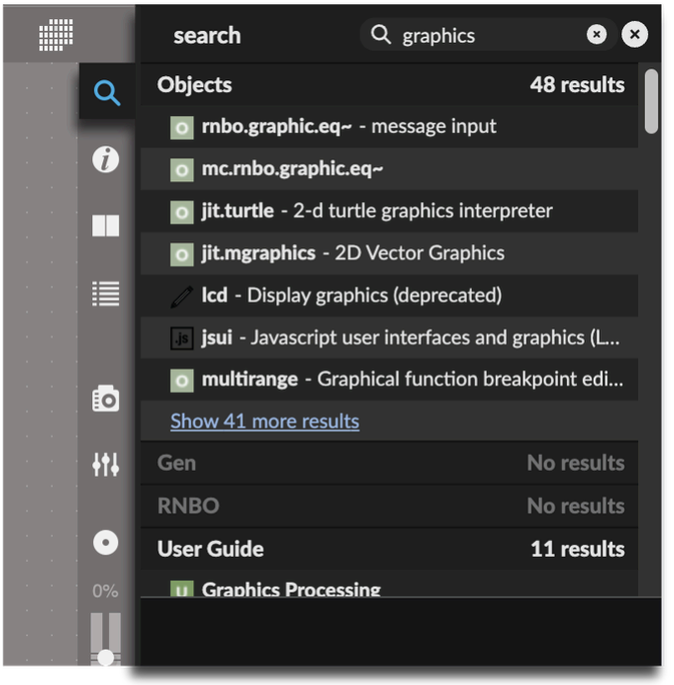


Finally, the calendar button lets you access the calendar. This can be extremely useful when you want to know what patches you opened on a specific date.

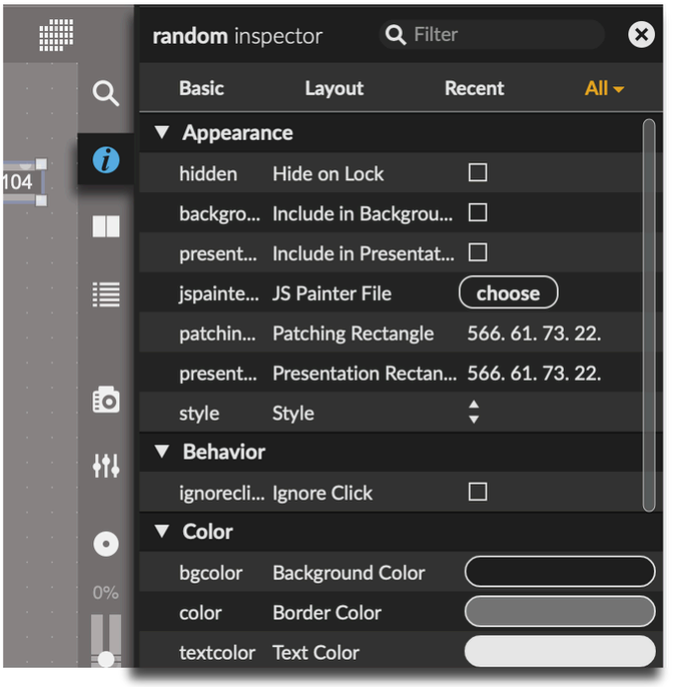


Right toolbar

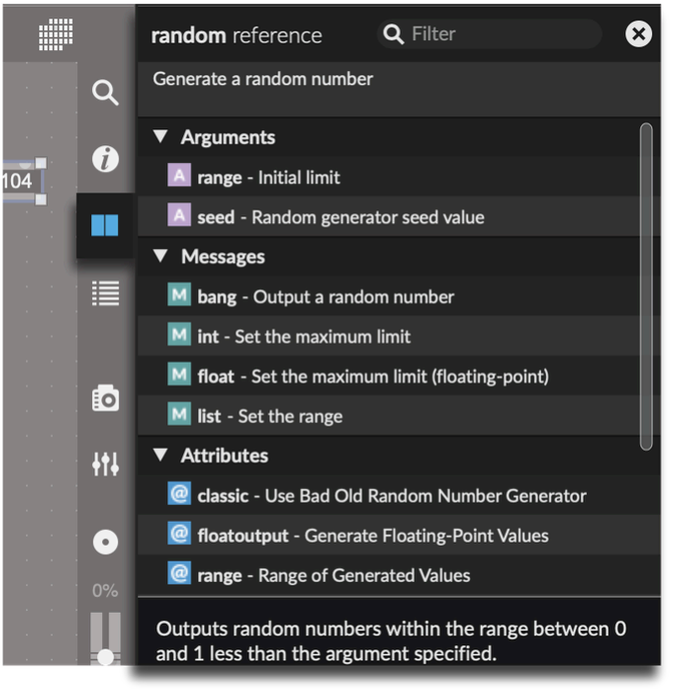
The right toolbar lets you access the sidebar. Each button icon opens a different sidebar view. The Search icon lets you access Max's search, which can help you find objects, reference documentation, examples, and forum posts.



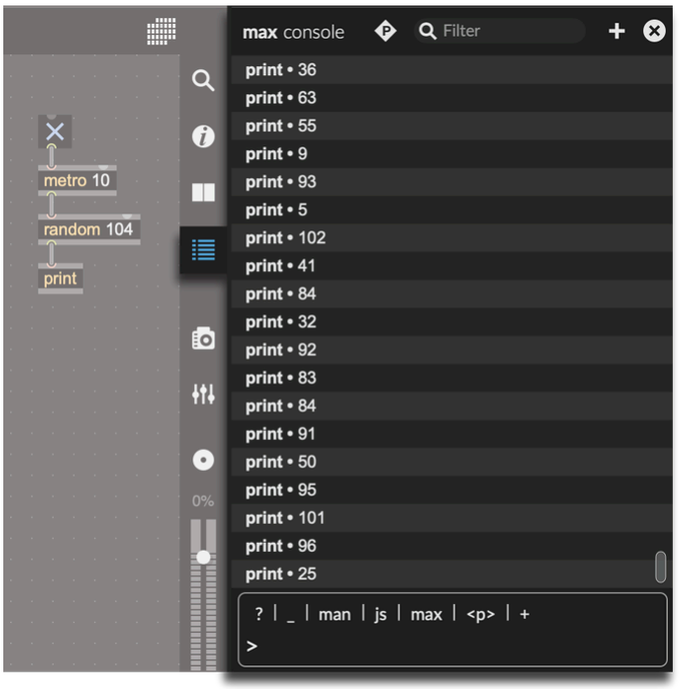
The Inspector icon will open the patcher and object inspector, which lets you view and edit the configuration of the objects in your patcher.



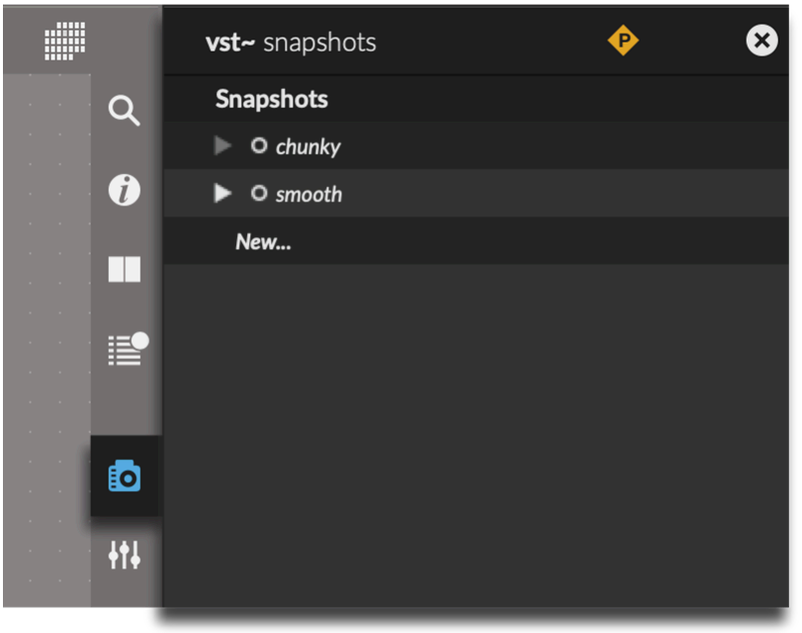
The Reference Sidebar gives a quick summary of the function of the selected object, along the messages and attributes that the selected object understands.



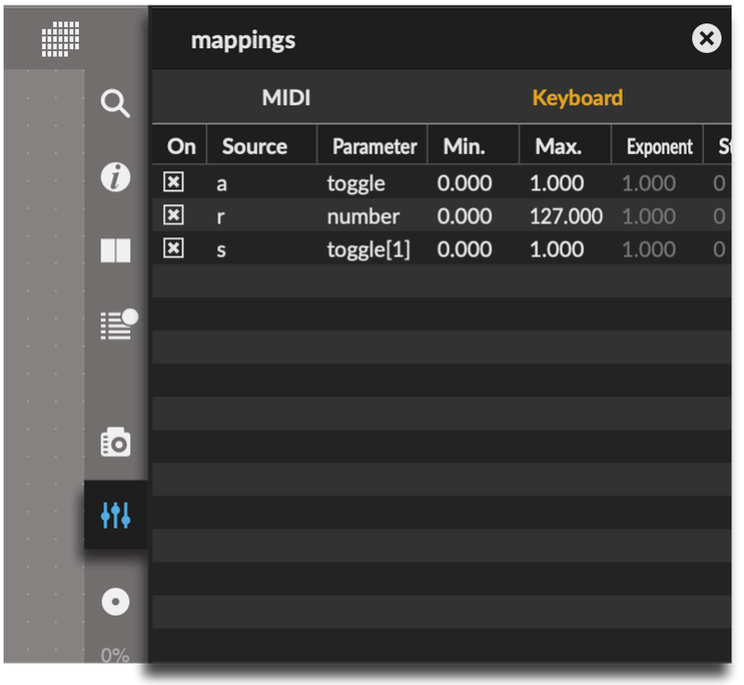
The Max Console displays errors and warnings, in addition to the output of any print objects.



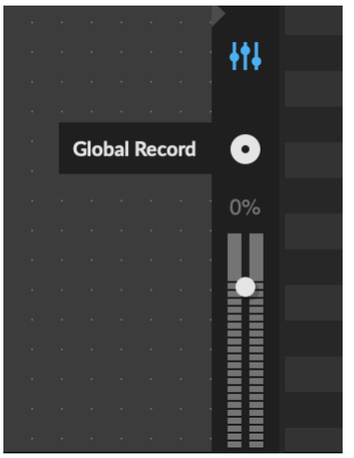
The Snapshot editor lets you view and edit any snapshots belonging to the current patcher.



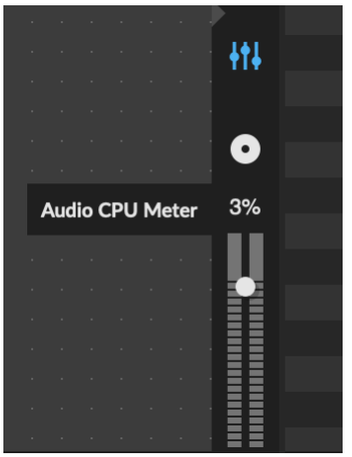
The Mapping button will be enabled if any objects in the current patcher support parameter mapping. From the mapping editor, you can view and configure MIDI and Keyboard mappings.



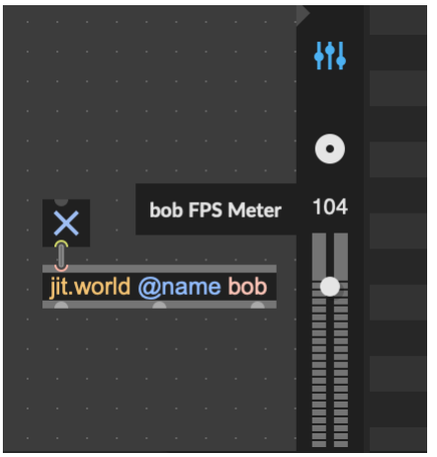
You can use the Global Record button to quickly record the audio output of your patcher.



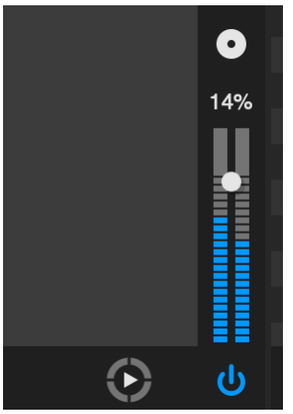
Above the volume control, the *Audio CPU Meter* tells you how much of your computer's processing power you've used up with signal computation.



If you have a jit.world object in your patch, you can click on the *Audio CPU Meter* to toggle the *FPS Meter* for your graphics context. This shows you the rate of graphics processing in frames per second.

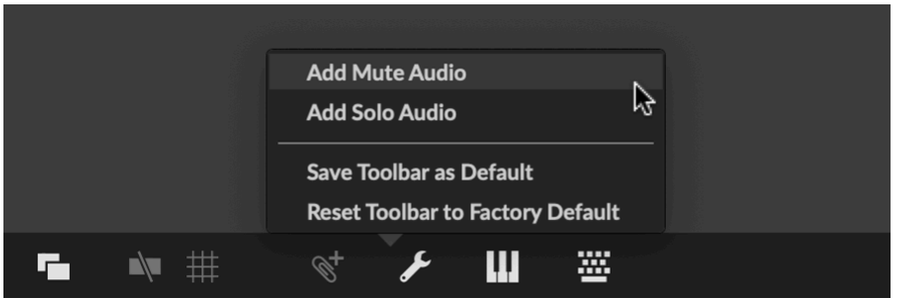


Finally, at the bottom of the right sidebar, the volume control lets you adjust the gain for any audio generated from this patcher. Each patcher has its own gain control.



Bottom toolbar

The bottom toolbar contains controls for changing how you interact with your patcher, including enabling/disabling an alignment grid, turning on signal processing, and more. You can right click on the bottom toolbar and select "Add Mute Audio" or "Add Solo Audio" to enable these optional icons.



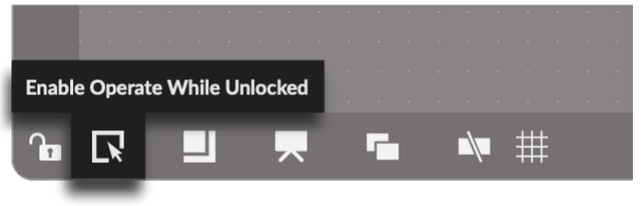
From left to right, the first icon in the bottom toolbar controls Locking, letting you lock/unlock your patcher.



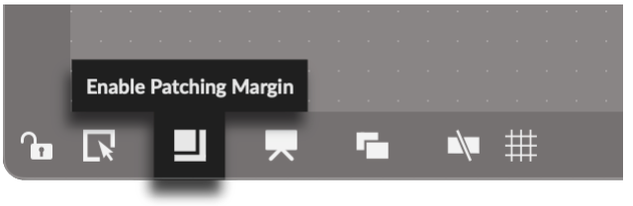
If you're looking at an instance of an Abstraction, then the lock icon will change to a crayon. Clicking this icon will let you modify the orignal patcher.



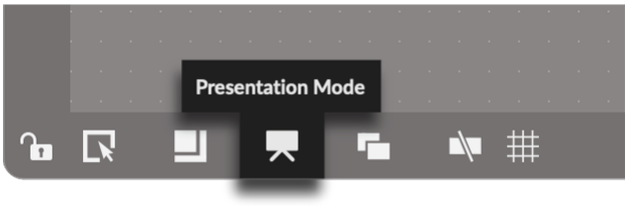
The Operate While Unlocked icon enables an interaction mode that lets you control UI objects in your patcher, even while the patcher is unlocked.



The *Patching Margin* icon gives you a bit more room to work with, at the border of a patcher that fills up the entire view.

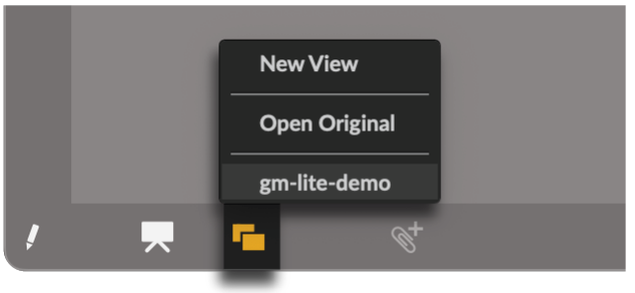


Clicking the Presentation Mode icon will enable/disable presentation mode.

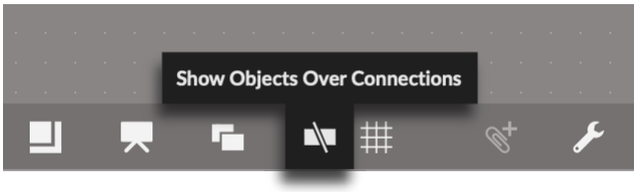


The *Patcher Windows* button lets you access different views of the current patcher. Click "New View" to open a new window displaying the same contents as the current patcher. This can be useful if you want to look at two different parts of a large patcher at once, or if you want to view a patcher in presentation and patching mode at the same time. If you're looking at an instance of an

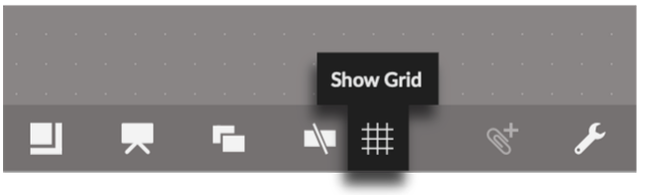
Abstraction, the option "Open Original" will be enabled, and selecting this option will open the original version of the abstraction. Finally, if you're looking at a subpatcher, the bottom of the menu will let you navigate up the patcher hierarchy to a parent patcher.



Clicking the *Show Objects Over Connections* button toggles between displaying objects over patch cords, or patch cords over objects.



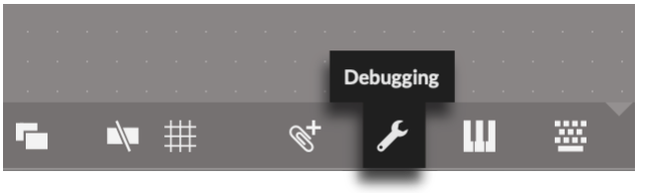
The *Show Grid* button will let you enable and disable an alignment grid for your patcher. You can control this same option by selecting *Grid* from the *View* menu, and this option works in conjunction with *Snap to Grid* from the *Arrange* menu.



With some objects selected in your patcher, the Snippet button will let you save a new snippet from your selection.



The *Enable Debugging* button will toggle Debug Mode.



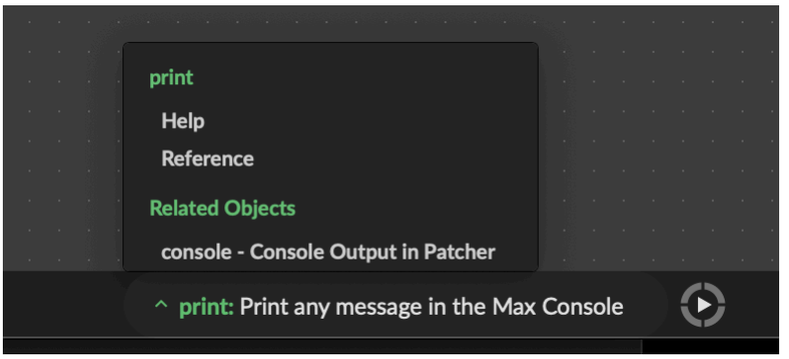
You can configure MIDI Mapping by clicking the *MIDI Mapping* icon.



And you can configure Keyboard Mapping by clicking the *Keyboard Mapping* icon.

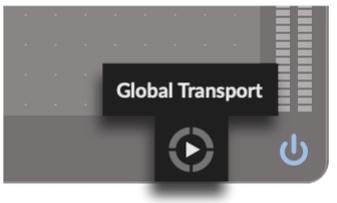


When you hover over an object in your patcher, the **Clue Bar** will appear in the bottom toolbar. You can configure the appearance and behavior of the Clue Bar using the *Clues* preference in the Preference Window. If you click on the name of an object in the Clue Bar, Max will show you additional information about that object.

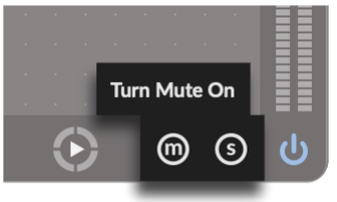


> You can set the `@annotation` attribute on an object to customize the text that appears in the *Clue Bar*.

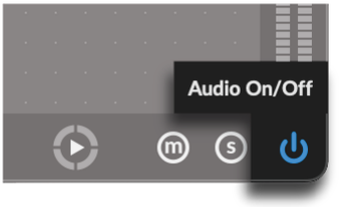
You can enable and disable the Global Transport with the *Transport* icon near the end of the bottom toolbar.



If you enable the optional *Mute* and *Solo* icons, these will appear to the right side of the bottom toolbar. These will let you silence audio in the current patcher, or silence all other non-soloed patchers, respectively.



Finally, the *Audio On/Off* button in the right corner or the bottom toolbar will let you enable or disable audio processing.

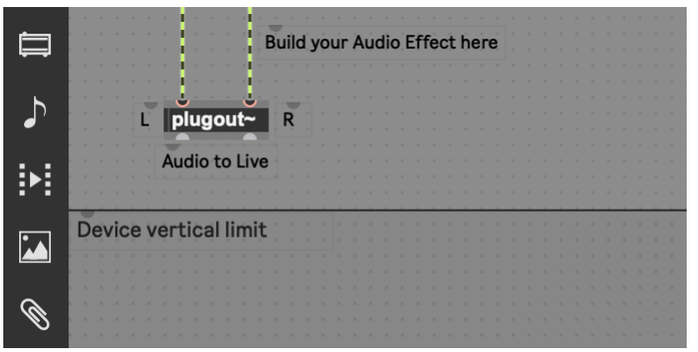


If you enable local audio processing by way of a dac~ object with `@local 1`, or with the `startwindow` message, the *Audio On/Off* button will glow orange instead of blue.



Max for Live Window

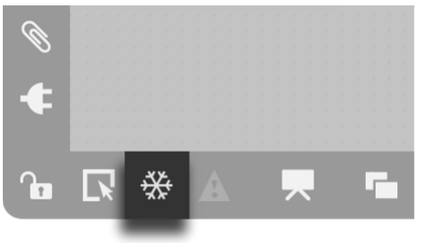
The Max patcher window will look slightly different when displaying a Max for Live device.



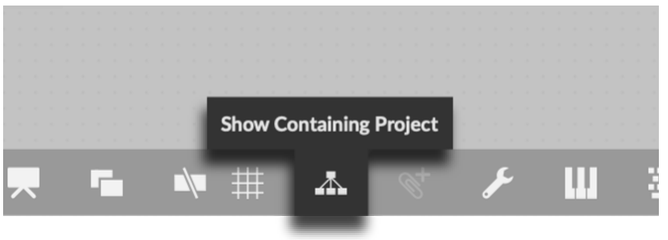
While in edit mode, the patcher will show a line indicating the vertical limit of the device. When editing the device directly from Live by pressing the *Edit* buton, you cannot adjust the height of this line. However, if you open the `.amxd` file in Max, you may adjust the position of this line. Changing the position of this line will not affect the height of the device in Live.

Toolbar changes

In the bottom toolbar, the *Freeze Device* button lets you **Freeze** and unfreeze your Max for Live device.



The *Show Containing Project* button will reveal the Project that contains the Max for Live device. The project is the place to configure properties of your Live device like whether it is an Audio Effect, MIDI Effect, or a MIDI Instrument, among others.



Finally, the *Preview* button will let you enable/disable preview for the Max for Live device (this button is only available when opening a device from Live by pushing the *Edit* button). With **Preview** enabled, audio and MIDI will pass to and from Live, directly into your Max for Live device as you edit it. With Preview disabled, audio and MIDI will bypass your device until you finish editing it.

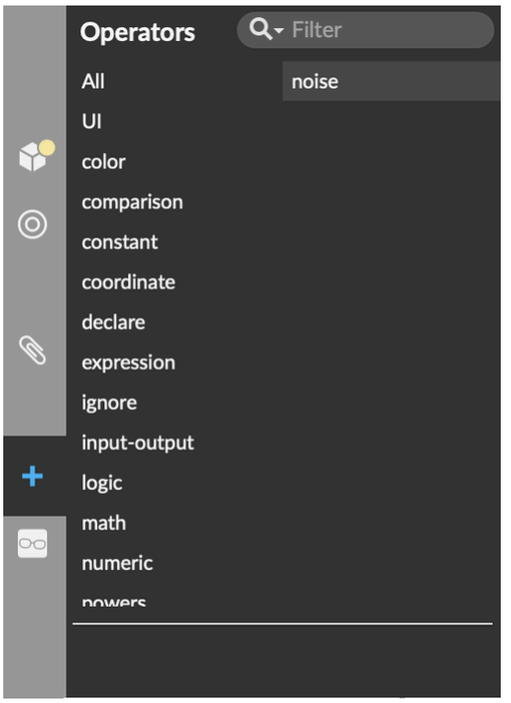


Gen Window

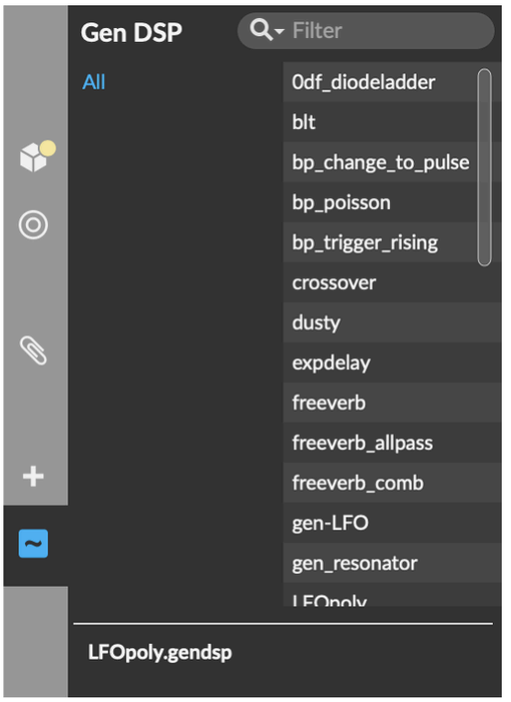
The window for a Gen patcher has its own toolbar buttons as well. These let you access the special objects supported in Gen, and give you control over when Gen compiles its code.

Toolbar changes (Gen)

In the left toolbar, the *Gen Operators* button gives you access to all of the operators supported in Gen, sorted by category. This will display a different set of objects depending on whether the patcher is a Gen DSP (gen and gen~ objects) or Gen Jitter (jit.gen, jit.pix, and jit.gl.pix) patcher.



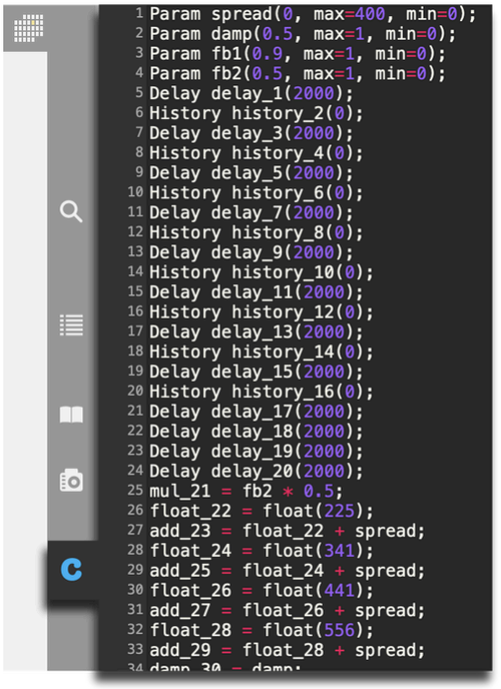
The next button in the left toolbar will be titled either *Gen DSP* or *Gen Jitter*, again depending on whether the patcher is a Gen DSP (gen and gen~ objects) or Gen Jitter (jit.gen, jit.pix, and jit.gl.pix) patcher. This button lets you see built-in `.gendsp` and `.genjit` patchers, which are essentially Abstractions that are restricted to the Gen domain. Many of these Gen Abstractions are helpful starting points or useful building blocks.



In the bottom toolbar, the *Enable Auto-Compile* and *Compile* buttons let you decide when your patcher compiles. With **Auto-Compile** enabled, your patcher will compile whenever you make a change. If Auto-Compile is disabled, then you can press the *Compile* button to direct Gen to compile your patcher whenever you're ready. Auto-Compile is enabled by default—disable it if you find that your patcher is taking a long time to compile.



Finally, the *Code* button in the right toolbar lets you open the **Code** sidebar. This lets you examine the Gen code that is generated from your patcher. If you want, you can copy-paste this into a Gen codebox.

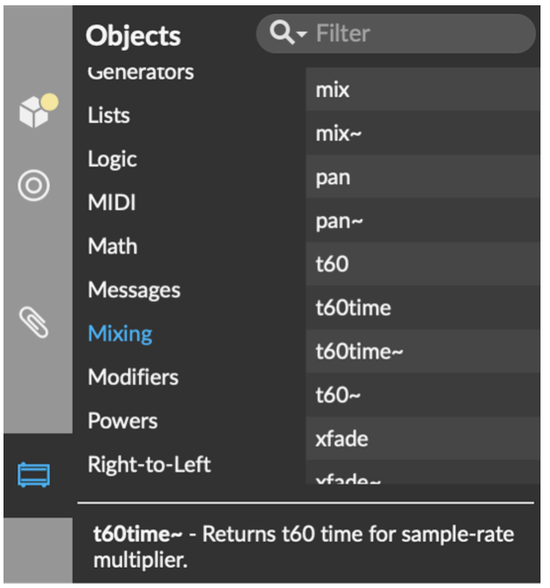


RNBO window

The RNBO window is very similar to the Gen window, which shouldn't be surprising given that both generate code. The most important difference is the *Export* button in the right toolbar, which gives you access to the **Export Sidebar**.

Toolbar changes (RNBO)

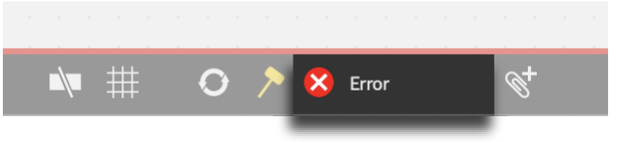
In the left toolbar, the *RNBO Objects* button will open a browser view for objects belonging to RNBO. You can sort these by category and filter them by name.



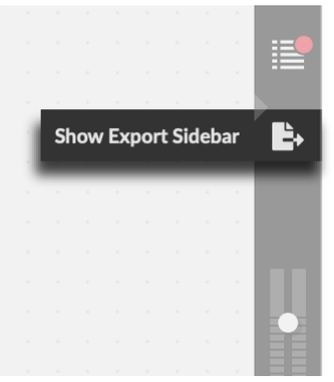
Similar to Gen, RNBO patcher windows have *Enable Auto-Compile* and *Compile* buttons to let you decide when your patcher compiles. With **Auto-Compile** enabled, your patcher will compile whenever you make a change. If Auto-Compile is disabled, then you can press the *Compile* button to direct RNBO to compile your patcher whenever you're ready. Auto-Compile is enabled by default—disable it if you find that your patcher is taking a long time to compile.



If RNBO encounters an error while trying to compile your patcher, an error indicator will appear in the bottom toolbar. Click on the indicator to display the generated source code, which will show on which line the error occurred.



Finally, the *Show Export Sidebar* button in the right toolbar lets you show and hide the **Export Sidebar**, from which you can export your RNBO patcher to any of its supported targets.


Show Export Sidebar

MIDI

Mapping

MIDI Mapping 456
MIDI Setup 457
Key Mapping 459
Deleting Maps 459
Editing Maps 460
Saving/Loading Maps 463

The Mappings system allows you to connect MIDI controllers and the computer keyboard directly to user interface objects. If you've used custom MIDI mapping in a DAW or other program, the concept should be familiar. Mapped parameter values can be changed directly from a MIDI controller, without routing MIDI messages through Max objects.

Mapping uses object Parameters under the hood, so any Max object with a float, integer, or enum parameter can be mapped. Detail on the individual mappings is provided in the 'Show Mappings' sidebar pane. Mappings are specific to individual patchers, and are not global Max preferences. Mapping is also not available within the Max for Live context.

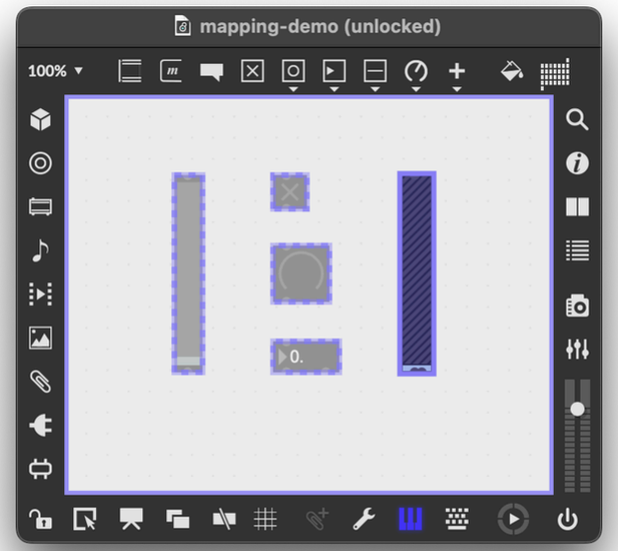
MIDI Mapping

Enable MIDI mapping by clicking the *Assign MIDI Map* button in the bottom toolbar.



The whole patcher view should get a purple border, and any MIDI-mappable objects will get a purple hue as well. Objects that have Parameter Mode enabled will be solid purple and ready for mapping. Objects that could have parameter mode enabled will have a dashed purple outline, and you can click on these to enable MIDI mapping.

> There's a shortcut to assign a MIDI map for any UI object: right-click on the object and select *Assign MIDI Map* from the contextual menu.

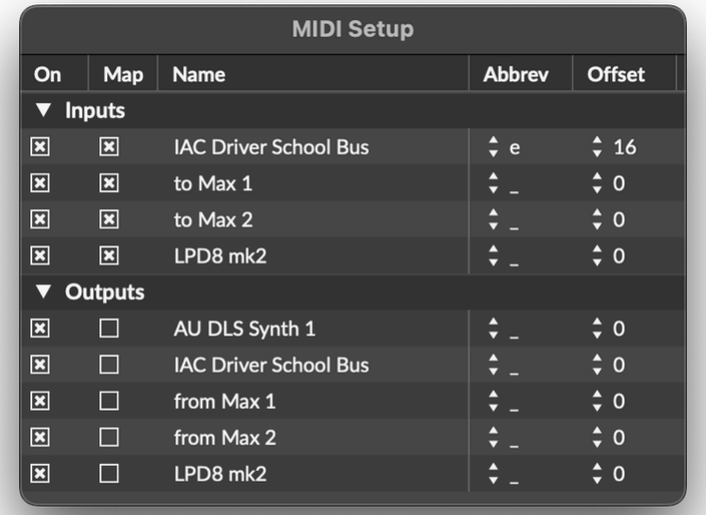


Click on any mapping-ready object to focus it, then use any connected MIDI device to send a control change or note message. You should see the value of the object change, and the top-right corner of the object will show a solid blue box to indicate an active mapping. You can also open the Mappings Sidebar to verify that your MIDI message was mapped to the object.

> You can also shift focus between mapping-ready objects by hitting the tab key.

MIDI Setup

You can configure which MIDI ports participate in mapping using the MIDI Setup window. If the value in the `Map` column is enabled, then Max will listen to MIDI messages from that port when in MIDI mapping mode.



By default, `Map` is enabled for all MIDI inputs (for devices sending MIDI *to* Max) and disabled for all MIDI outputs (where Max sends MIDI to an output port).

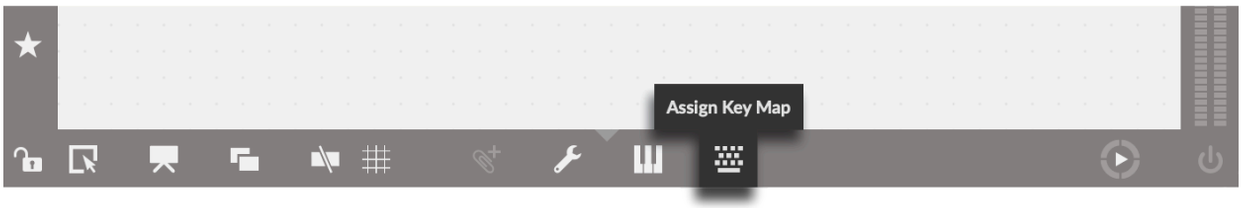
MIDI Output and Mapping

If you choose to enable `Map` for a MIDI output port, then any MIDI-mapped UI object will automatically send MIDI control change or note events to that port. You don't need a midiout, cltout, or noteout object. The specific controller number or note pitch that the UI object will send depends on how the object is mapped.

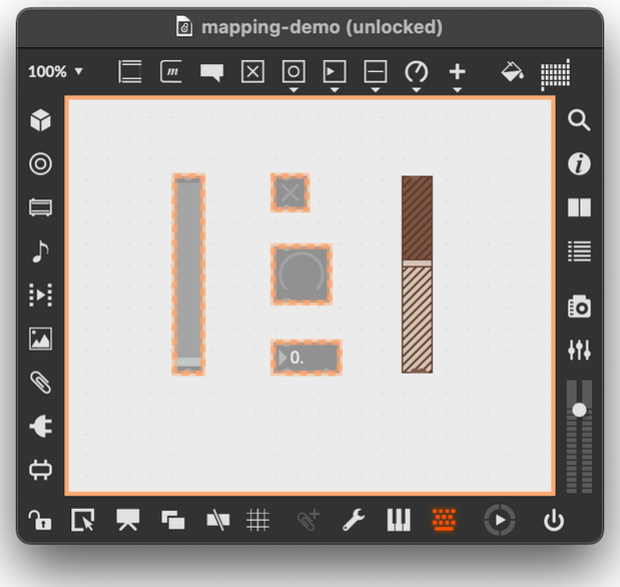
As concerns MIDI output mapping, A subtle but potentially important consideration has to do with what caused the object's value to change. Value changes produced by a MIDI or Key mapping will send a final value at the completion of the event, after Max has finished processing the event. However, value changes from other sources (clicking on a UI object, or sending it a message) will produce events as they occur.

Key Mapping

Key mapping works similarly to MIDI mapping. Enable key mapping by pressing the *Assign Key Map* button in the bottom toolbar.



The patcher view will get an orange border, and objects that support a key mapping will get an orange hue. Just like for MIDI mapping, a dashed orange line means that you need to click on the object to enable key mapping (by turning on Parameter mode for that object).

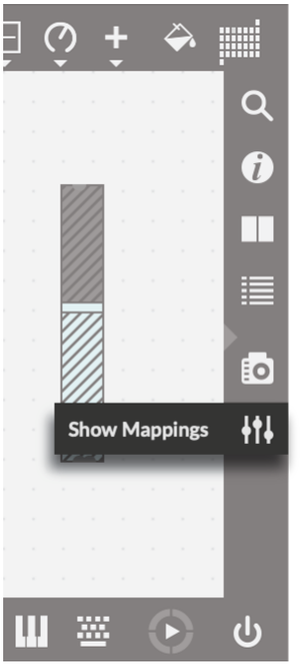


Deleting Maps

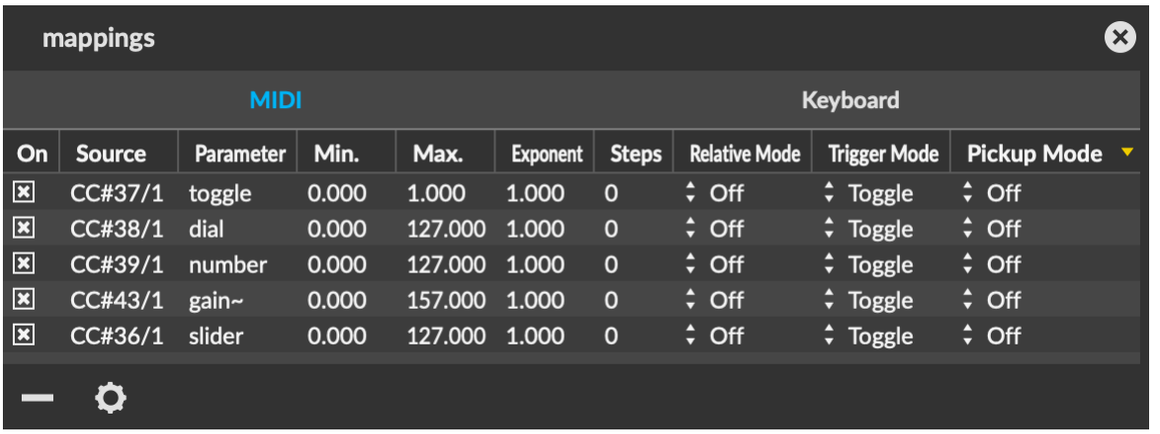
- With MIDI or Key mapping active, select an object and press the *delete* key to remove the mapping.
- You can also delete the mapping using the Mappings Sidebar.

Editing Maps

View and edit all of the mappings in the current patcher by opening the Mappings Sidebar.



All of the mappings will be visible here, organized into MIDI and Keyboard tabs. You can customize the behavior of each mapping by editing the row in the table corresponding to your that mapping.



Min, Max, Exponent, and Steps

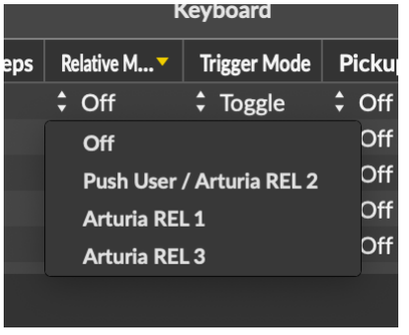
These values change how the input, whether keyboard or MIDI, maps to the object parameter. The values `min` and `max` refer to the min and max of the mapping itself, rather than the parameter, so they can have any value between the minimum and maximum value of the parameter. If you set `min` to a value greater than `max`, then the map will be inverted, with larger MIDI values corresponding to smaller parameter values. The `exponent` value lets you adjust the curvature of the mapping, giving you more fine-grained control towards the upper or lower end of the input value. The `steps` value lets you divide the input value into discrete steps.

> A Max parameter also has min, max, exponent, and steps, but these don't have to match the values of min, max, exponent, and steps for the mapping. Your parameter could, for example, have an exponent of 1, while the input mapping has an exponent of 3.

Relative Mode

Relative Mode affects the behavior of a mapped value when using an endless rotary encoder. If you're mapping from a fixed-position knob or slider — common on most MIDI controllers — you shouldn't need to change the relative mode. If you are using a controller like the Ableton Push, which uses endless rotary encoders, one of the Relative Mode options should correspond to the output of your device:

Mode	Description
Off	The controller sends absolute values, relative mode is off.
Push User / Arturia REL 2	Positive change is indicated by values > 0; negative change by values <= 127. Zero (no change) is 0.
Arturia REL 1	Positive change is indicated by values > 64; negative change by values <= 63. Zero (no change) is 64.
Arturia REL 3	Positive change is indicated by values > 16; negative change by values <= 15. Zero (no change) is 16.



Trigger Mode

Trigger Mode sets the way the object will respond to trigger-like events. Note-on events and key-down events are trigger-like — Max will treat the note on/key down event as the start of the trigger, and the note-off/key-up event as the end.

Mode	Description
Toggle	The value toggles between minimum and maximum.
Momentary Switch	The value stays at maximum until the event is completed.
Cycle	The value toggles between several options.
Bang	The object is treated as if it had received a bang message.

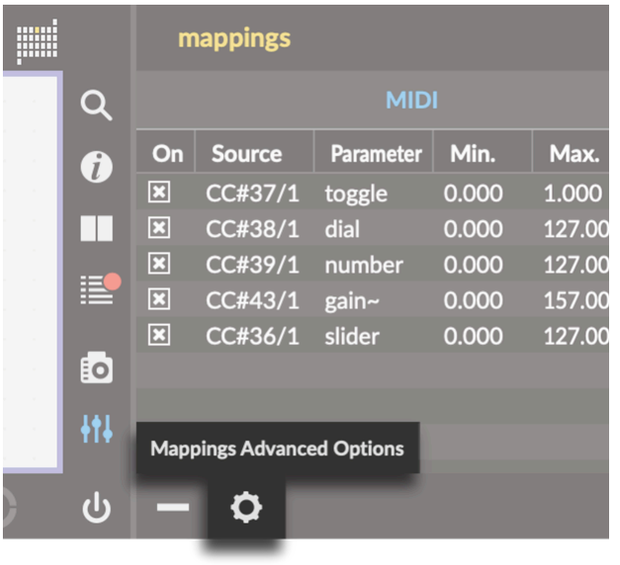
Pickup Mode

Pickup mode changes the way the mapped object behaves when the MIDI controller gets out of sync with the parameter value of the object. One way this could happen is if you click on a MIDI-mapped object to override its current value. In this case, pickup mode determines how the object will handle new MIDI values, until the two get back in sync.

Mode	Description
Off	The received MIDI value is treated like an absolute value, and the object will jump to the new value.
Pickup	The object will ignore incoming MIDI values until they cross over the current value of the object.
Scale	Incoming MIDI values will scale the parameter value between its current value and the minimum or maximum value, until the object reaches its minimum or maximum.

Saving/Loading Maps

Click on the Mappings Advanced Options button to reveal the Advanced options.



From here, click *Save Mappings to File* to save the current mapping configuration, and select *Load Mappings from File* to restore

MIDI

MIDI Setup 464
Objects for MIDI 466
MIDI Ports and Devices 466
MPE 467
Default Devices for MIDI Objects 468
Configuring the Built-in MIDI Synthesizers 469

MIDI stands for Musical Instrument Digital Interface. It's a method for exchanging information among compatible devices and programs. Originally created on top of a serial protocol, MIDI messages are often sent entirely within a computer, either between applications or from an application to a plug-in.

MIDI Setup

Choose *MIDI Setup...* from the *Options* menu to open the **MIDI Setup Window**.



The MIDI Setup window provides information on the availability and status of all available MIDI inputs and outputs, and lets you perform mapping between MIDI devices and names or numbers.

Max can receive MIDI on any MIDI *Input Port*, and can send MIDI to any MIDI *Output Port*. These ports have the following properties:

- *Name* - The unique identifier for the port
- *Abbrev* - An optional, single-letter abbreviation that acts as a shorthand for the port name.
- *Offset* - An optional channel offset, in multiples of 16.
- *On* - Whether or not the port is enabled
- *Map* - Whether the port is available for mapping

Enabling/disabling MIDI ports

Toggle the checkbox in the *On* column to enable and disable MIDI port. All ports are enabled by default.

Mapping mode

The checkbox in the *Map* column determines whether or not the given port is available for **MIDI Mapping**. Mapping output is disabled by default, see MIDI Output and Mapping for more details.

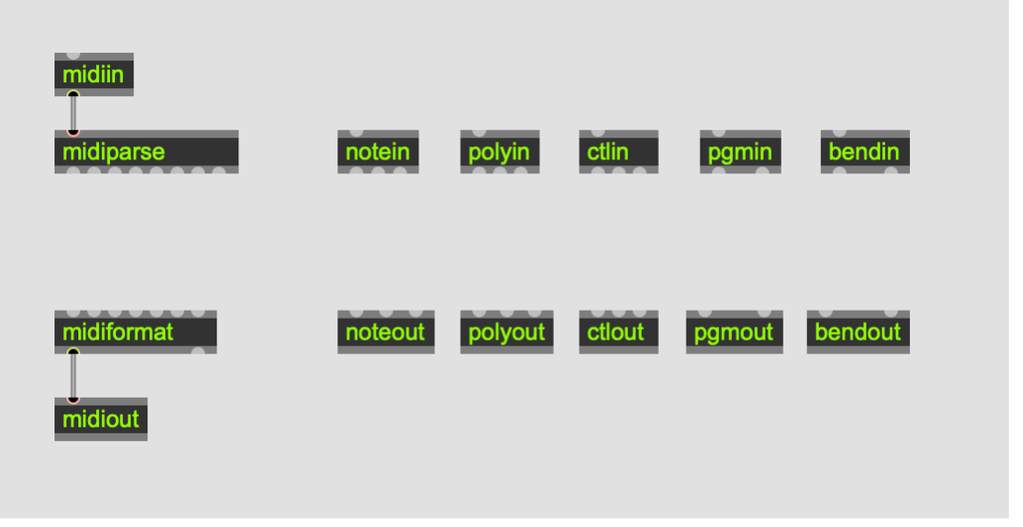
Abbreviation and channel offset

Use the *Abbrev* and *Offset* columns to configure the abbrevation and channel offset for each MIDI port. To create a letter mapping for a MIDI input or output, click in the Abbrev column for the row corresponding to a MIDI input or output and choose a letter from the pop-up menu that appears. The MIDI input or output device can now be identified using the letter abbreviation you have selected.

Additionally, you can set a channel offset for each MIDI port. To create a channel offset for a MIDI input or output, click in the Offset column for the row corresponding to a MIDI input or output and choose a channel offset value from the pop-up menu that appears. The MIDI input or output device can now be identified by sending or receiving a message whose MIDI channel number falls in the range *Offset Value* + 15. For example, if you configure a MIDI port to have an offset of 32, then setting the channel offset of 35 on a MIDI object will address that MIDI port on channel 3.

Objects for MIDI

The simplest objects for working with MIDI are midiin and midiout, which connect to a MIDI port and route MIDI byte streams to or from those ports. The MIDI objects midiparse and midiformat convert MIDI streams to and from Max messages. MIDI objects like notein/noteout, bendin/bendout, ctlin/ctlout, polyin/polyout, and pgmin/pgmout combine the functionality of midiin/midiout and midiparse/midiformat.



Basic objects for working with MIDI streams

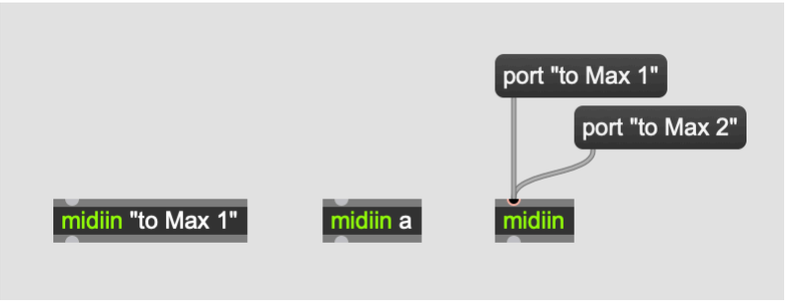
Receive MIDI real-time messages with rtin. This includes MIDI `start`, `stop`, and `tick` messages.

To work with MIDI system-exclusive messages, use sysexin to receive sysex messages and sysexformat to format them.

MIDI Ports and Devices

A MIDI **Port** is a named MIDI source or destination. Often, a MIDI **Device**, or a physical or virtual system that can send or receive MIDI, will open a single MIDI port, whose identifier is simply the name of the device. Because of this, you'll sometimes see *port* and *device* used interchangeably in MIDI object help files.

MIDI sending and receiving objects all respond to the `port` message, which configures them to bind to a particular MIDI port. The name of a MIDI port can also be an abbreviation, which can be configured in the _MIDI Setup Window, or using the `portabbrev` message to Max. The midiinfo object can list MIDI devices.

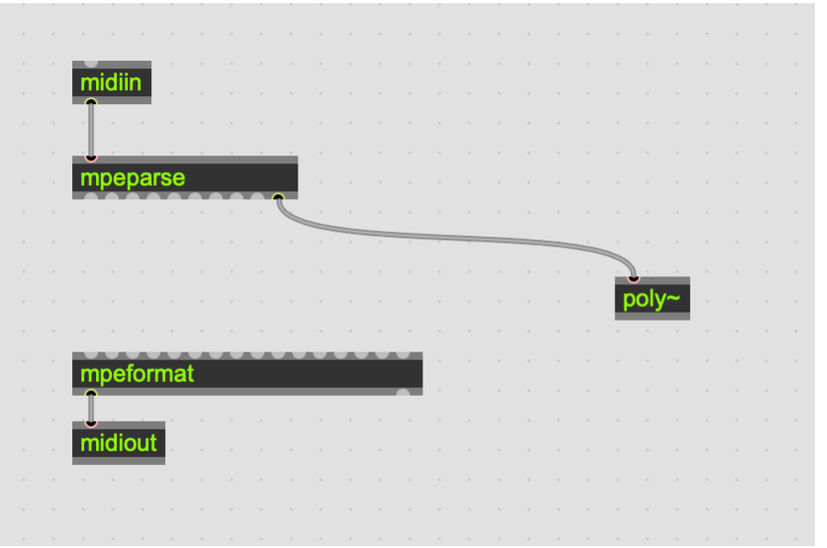


From left to right: a midiin object, bound to the port named 'to Max 1'; a midiin object bound to the port with the abbreviation 'a'; a midiin object that will listen on all ports, with messages to select between ports.

MPE

MPE stands for MIDI Polyphonic Expression. The MPE specification assigns each voice its own MIDI Channel, so that expression messages like pitch bend and aftertouch can be applied to indivicual notes.

The mpeparse object will format an incoming MIDI stream into MPE message events, and the corresponding mpeformat object will convert message events into an MPE-compatible midi stream. The mpeconfig object will let you configure a MPE device, for example by defining zones. Finally, the polymidiin object works in conjunction with the poly~ object, facilitating working with MPE in the context of a polyphonic subpatch.



The last outlet of mpeparse formats the MIDI stream into 'mpeevent' messages, which are understood by a poly~ containing a polymidiin object.

Default Devices for MIDI Objects

If you create a MIDI output object without specifying a device name, the object transmits MIDI to the first device (the default) in the list of output devices in the MIDI Setup window.

On the Macintosh, the default MIDI output is an AudioUnit DLS synthesizer. The AU DLS synth supports its own set of internal sounds as a General MIDI bank, and also provides support for Level 2 SoundFont files.

On Windows systems, the default MIDI output is the Microsoft GS Wavetable Synthesizer. (Note: the Microsoft GS Wavetable Synthesizer does not support SoundFont files)

If you don’t assign MIDI input or output to a specific destination or port (i.e., one that does not have a MIDI device name or abbreviation as an argument) , Max will automatically merge all input devices. This is useful if you want to treat all MIDI input identically regardless of its source, But this also means that the only way for a Max patcher to determine which device is actually the source of input to these objects is to compare the incoming MIDI channel number to the MIDI channel offset specified for each device.

> Note: The midiin object is an exception to this behavior — if you don't use an argument to specify a device, it receives data only from the first device in the input device list. When multiple devices share the same letter abbreviation, Max will use the first one in the list at the time a MIDI input or output object is created with that abbreviation as an argument. Changing the abbreviations of devices has no effect on pre-existing objects, although it will have an effect on the meaning of subsequent port messages sent to MIDI output objects with abbreviations as argument.

Configuring the Built-in MIDI Synthesizers

Max uses your machine's built-in MIDI synthesizer as its default MIDI output destination. With the OS-provided, built-in synthesizer, you can make sound without any external MIDI gear.

On Macintosh, the default MIDI output is the AudioUnit DLS (Down-Loadable-Sounds) synthesizer. The AU DLS synth supports its own set of internal sounds as a General MIDI bank, and also provides support for Level 2 SoundFont files.

On Windows systems, the default MIDI output is the Microsoft GS Wavetable Synthesizer.

On both systems, you can configure the built-in synthesizer by sending messages to Max directly.

Creating a new port for a DLS synth (Mac only)

Only one port for the built-in synth is created by default. (augraph on Mac OS X or midi_mme on Windows). However, on Mac, you can work with more than one DLS synth by creating additional MIDI synthesizer ports and assigning new DLS sound bank files to each one. This feature is not available on Windows machines.

- Use the `createoutport` message.
- The name of the driver will be `augraph` on macOS. For the port name, you can use the name you want to give to your virtual output port.

- For example, the message `;#SM createoutport synth2 augraph` create a second DLS synth called “synth2”.

Deleting a DLS synth port (Mac only)

- Use the `deleteoutport` message.
- Again, on macOS the name of the driver will be `augraph`. The port name will be the name of the virtual port (created with `createoutport`) that you want to delete.

Loading a bank of sounds on a DLS Synth (type 1 or 2) (macOS only)

- Use the `driver loadbank` message.
- The filename should be the name of an existing DLS bank file, and the port name will be the name of the port that will use this bank. If you omit the port name, the bank you specify will be loaded by all DLS ports. On macOS, the folder `/Library/Audio/Sounds/Banks` is added to the search path when looking for a DLS bank file (i.e. it is the default location for searching for sound bank files).
- If you specify a zero (0) for the filename, the default GM (General MDI) sound bank will be loaded (e.g., ;#SM driver loadbank 0 <portname>).

Turning the DLS synth reverb on and off (Mac only)

- Use the `driver reverb` message.

Setting MIDI output latency (Windows only)

- The Windows *midi_mme* will let you configure the output latency.
- Use the message `driver latency`.
- For example, the message `;#SM driver latency 10 midi_mme` will set the driver latency to 10 milliseconds.

Parameters

OpenSoundControl

osc.codebox 473
OSCQuery 473
The FullPacket Message 473

With OpenSoundControl (OSC) support, Max can be controlled by any OSC-compatible device or application, and can also send OSC messages to control other OSC-enabled systems.

OSC can be enabled in Max in two ways:

1. Through the built-in UDP server, which you can activate for the whole application in Max's preferences, or for a specific patcher using the patcher inspector.
2. By using the param.osc object.

Once enabled, **parameters** are automatically given *OSC addresses*, which have the following structure:

```
/<patcher name>/param/<parameter name>/<attribute name>
```

The different components that make up the address are configurable globally and locally in each patcher. Using these addresses, you can get and set the values of the following parameter attributes:

Parameter attribute	OSC address	Description
Raw Value	`/raw`	The raw (scaled) value of the parameter. This address container is optional if the raw value is the only value present in the OSC bundle. See the OSC Value Mode attribute.

Normalized Value	`/normalized`	The normalized (`[0,1]`) value of the parameter. This address container is optional if the raw value is the only value present in the OSC bundle. See the OSC Value Mode attribute.

osc.codebox

osc.codebox can be used to display the contents of an OSC packet using JSON syntax. It's worth mentioning that OSC is a binary format--it has no human-readable form. The use of JSON syntax to represent OSC should be considered an approximation of the underlying binary data.

OSCQuery

OSCQuery is a method of describing the capabilities of an OSC server. An http server can be configured to serve OSCQuery requests in Max by enabling OSCQuery in the general preferences.

Once enabled, an OSCQuery request can be generated with a URL like `http://localhost:30339`. All patchers with parameters exposed as OSC will be present in the OSCQuery response. Individual patchers can be excluded from the OSCQuery response using the option in the patcher inspector.

The `FullPacket` Message

Objects that accept and produce OSC do so using a message called `FullPacket`. This message is passed with two arguments, and should be considered opaque, i.e. these arguments are not to be manipulated as normal max values.

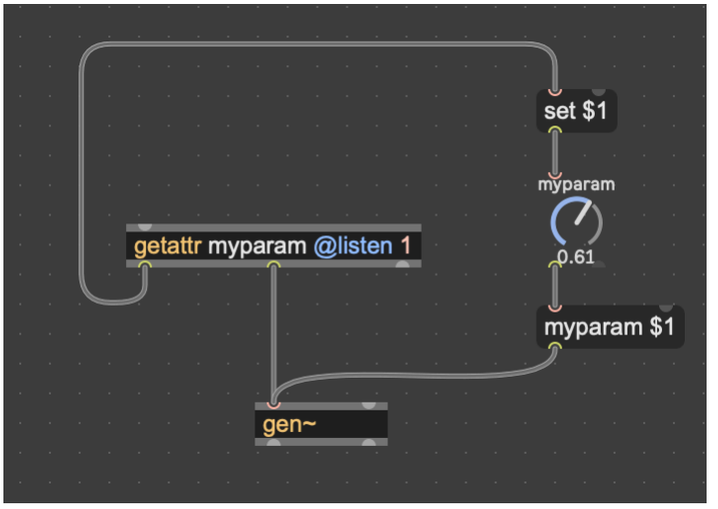
> An important property of the FullPacket message is that it is transient and **must not be stored** in objects like the message box, zl.reg, etc.

Connecting Parameters

Supplier Objects 475
UI Objects 476
Establishing Connections 476
Visualizing Connections 477
Limitations 478

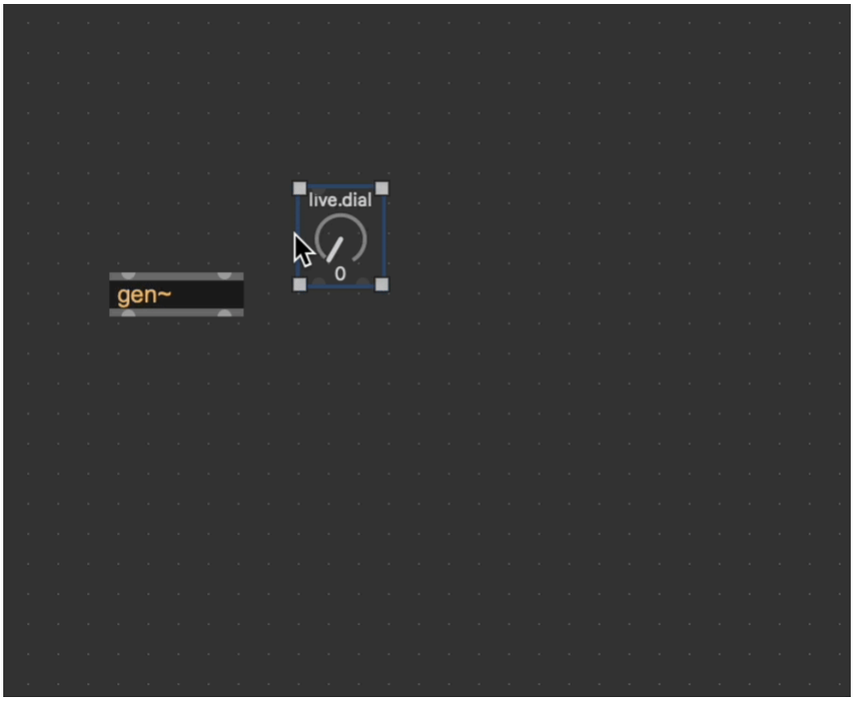
You can include an object called *param* in Gen and RNBO that declares a **parameter**. Gen and rnbo~ parameters can be changed as **attributes** of the object. If you wanted to link a live.dial to control a gen~ parameter you would have to do the following:

- copy the Gen parameter's min, max and default to the live.dial's `parameter_range`, `parameter_inital`, and `parameter_initial_enable`.
- use getattr to listen to the value of the Gen parameter, then patch the live.dial to gen~ and getattr using a `set` message to avoid creating a feedback loop and another message box to prepend the attribute name to the value of the dial



Note that you have to copy this patching for *every* parameter you want to control.

The `param_connect` attribute of many Max UI objects (including live.dial) provides a way to handle all of this with one step, as demonstrated below. In this example, we will connect a live.dial to a parameter inside gen~ called `xyz`.



Once this connection is established, the live.dial will control the `xyz` parameter inside gen~ and if the `xyz` parameter's value changes in Gen, the live.dial will update to reflect the new value. Furthermore, because the Gen parameter is linked to a live.dial, it can be automated in a Max for Live device or used to for state management in pattrstorage or preset.

Supplier Objects

The list of **suppliers**—objects that define parameters connectable to UI objects via `param_connect` will continue to expand over time. It currently includes:

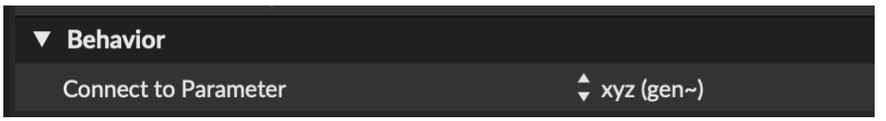
- Gen (gen~, gen, jit.gen, jit.gl.pix, and jit.pix as well as all the Gen codebox variants such as gen.codebox~).
- RNBO (rnbo~)
- The ABL objects

- v8 with attributes defined via scripts that call `declareattribute`
- poly~ objects that contain subpatchers with param objects. For more information, refer to Polyphony.
- jit.gl.slab (shader parameters)

Not all supplier objects will support every `param_connect` feature but all provide for bidirectional control.

UI Objects

To verify if a UI object can be connected to a parameter, look in the *Behavior* category in the Inspector. If the object is compatible, you'll see a *Connect to Parameter* attribute:



Generally, any UI object that is *parameter-aware*—in other words, an object with `parameter_enable` attribute—will be able to connect to a supplier object. While some UI objects such as multislider handle multiple values, only the first value will interact with a parameter of a supplier.

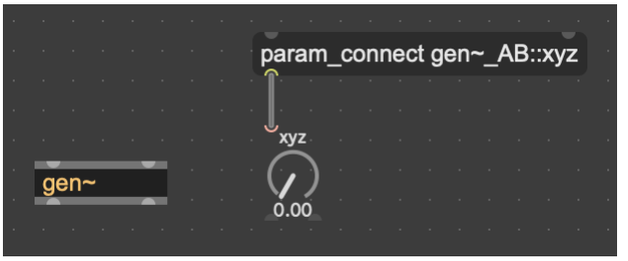
Some non-UI objects—pattr for example—are also parameter-aware but they cannot connect to parameters of supplier objects.

Establishing Connections

Connections to parameters are always established with the UI object you want to use for control and display. There are three ways to establish a connection:

- Use the *Connect* submenu of the Object Action Menu
- Using the Inspector on the selected UI Object, choose the desired parameter from menu for the *Connect to Parameter* attribute

- Send the message `param_connect <path ID>` to the UI object. To determine the path ID, combine the scripting name of the supplier object with the parameter name separated by double colons as follows: `gen~_AB::xyz`



To view the scripting name of an object, select the object and open the Inspector or choose *Name...* from the Object menu.

Note that while multiple UI objects can be connected to the same parameter, a UI object can only control a single parameter.

To disconnect a UI object from its parameter, choose *None* from the *Connect* submenu of the Object Action Menu.

Visualizing Connections

To see the connected parameter for a UI object, hold down the Option / Alt key while the cursor is over the object. A line will connect the UI object and supplier as shown below:



You can also check the connection via the menu in Object Action Menu or Inspector. The menu shows a check mark next to the connected parameter name.

Limitations

Connections between UI objects and supplier objects can only occur in the same patcher. If you put the supplier object in a subpatcher (for example, via encapsulation), the connection will disappear.

Parameter connections are generally single-valued. Multi-valued parameter data and user interface objects are not yet fully supported.

Parameter Mode

Enabling Parameter Mode 480
Parameter Attributes 480
Initial Value 482
Parameter Names 483
Parameter Modulation 483
Parameter Data Types 485
Custom Unit Styles 486
Parameter Visibility 486
Parameter Window 488

Some Max objects can define a **Parameter**, which is a simple representation of the current state of that object. Parameters define an interface between a Max patcher and some outside system for the purpose of presetting, automation and modulation. In a Max for Live device, parameters can be saved and recalled in the form of presets, and are exposed to Live **Automation** and **Modulation**. Even outside of Live, parameters let you set the initial value of an object, and can be saved and recalled by interacting with the pattr family of objects or Snapshots. They can also map to MIDI events and the computer keyboard through Max's Mapping System.



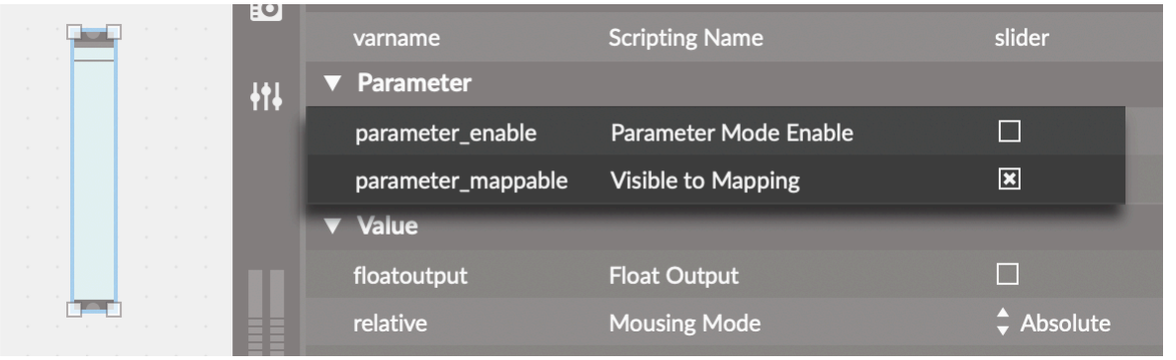
This slider has enabled parameter mode, which creates a parameter 'Frequency' that can be controlled by a MIDI message.

Most Max UI objects and Max for Live UI objects support **Parameter Mode**. The pattr and vst~ objects can also participate.

The Parameters Window shows all parameters currently associated with a patcher (or device), and permits you to change parameter attributes in a single place. You can also change parameter attributes for individual objects by using the Parameter tab of the Inspector.

Enabling Parameter Mode

Many objects that support *Parameter Mode* will have an attribute `@parameter_enable` or "Parameter Mode Enable". The Max for Live UI objects *always* have their parameter enabled, and don't display that attribute.



The Parameter Mode Enable attribute

Activating *Parameter Mode* for an object will create a parameter for that object. The object owns that parameter, and you can customize the behavior of the parameter through the parameter-related attributes on that object. Enabling the *Parameter Mode Enabled* attribute will reveal all of the parameter-related attributes in the Inspector.

Parameter Attributes

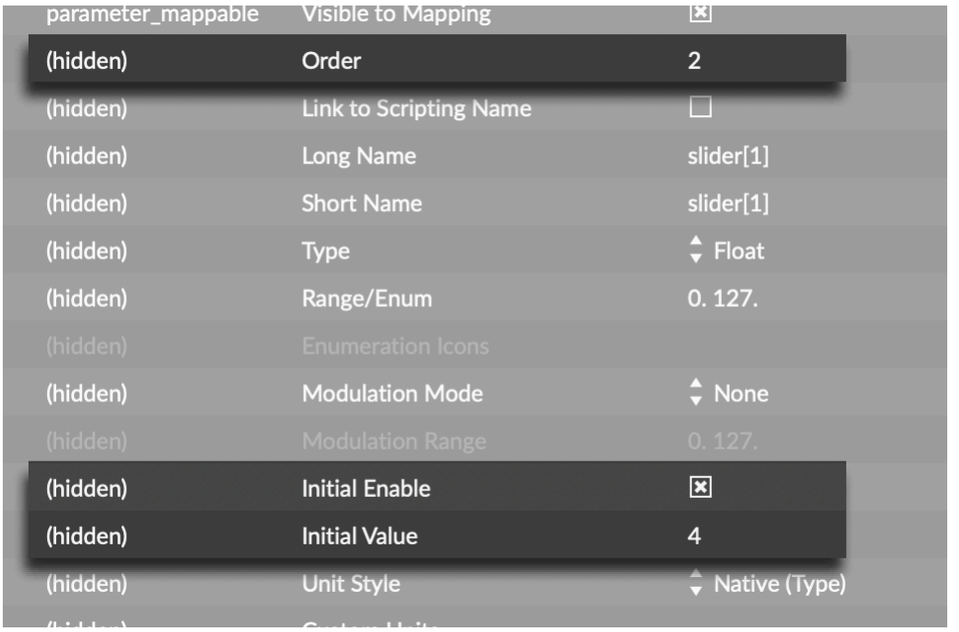
Name	Description
Visible to Mapping	When enabled, the parameter will be available for mapping to keyboard or MIDI input using Max Mapping
Order	Sets the order of recall of this parameter. Lower numbers are recalled first. See Initial Value.
Link to Scripting Name	When enabled, the *Scriping Name* attribute is linked to the *Long Name* attribute. See Parameter Names.
Long Name	The internal, programmatic name of the parameter. Must be unique within the patcher hierarchy. See Parameter Names.

Short Name	The display name of the parameter, used in the user interface. See Parameter Names.
Type	Type of the parameter. In general this will be *Int* (0-255), *Float* (32-bit float), *Enum*, or *Blob*. Some parameter types are disabled for certain objects.
Range/Enum	The range of the parameter (for *Int* or *Float*) or the members of the enumeration (for *Enum* types). Unsupported for *Blob* type parameters. See Parameter Data Types.
Enumeration Icons	Image files to be used in place of text on the Ableton Push controller, for parameters with the *Enum* type.
Modulation Mode	See Parameter Modulation
Modulation Range	The *Modulation Range* of the parameter, if enabled.
Initial Enable	If enabled, the parameter will be set to an initial value when the patcher or device is loaded. When you turn this on, the current value of the parameter is stored as the initial value. See Initial Value.
Initial Value	The parameter's initial vlaue, if *Initial Enable* is enabled. See Initial Value.
Unit Style	Changes how the value of the parameter will be displayed. For example, the *Pan* style will display negative numbers as panning left, positive numbers as panning right, and zero as center pan.
Custom Units	The format string used to to display the parameter's value, if *Custom* is selected for *Unit Style*. Accepts sprintf-style format strings. See Custom Units.
Exponent	Scales the exponential weight of the parameter's range. Values above `1` give the parameter more fine grained control near the low end of the parameter's range, while values between `0` and `1` give more fine grained control near the top end.
Steps	The number of discrete steps between the minimum and maximum values of the parameter's range. Values are inclusive, so steps `4` with a range `10 40` will have the possible values `10`, `20`, `30`, and `40`.
Update Limit (ms)	Limits the rate of updates when new values are triggered by automation.

Defer automation output	When enabled, value updates that are triggered by automation are sent to the back of the low priority queue.
Parameter Visibility	Determines whether parameters are hidden (to a host like Live), and whether their values are automatable. See Parameter Visibility.

Initial Value

With Parameter Mode enabled, you can turn on the "Initial Enabled" attribute in order to set an initial value for the object. The object will restore this value when the patcher loads, as well as sending this value to any connected objects. For Live UI objects, you can also double-click on the object to restore its initial value. The "Order" attibute affects the order in which parameter-enabled objects are initialized. Objects with a lower value for their "Order" attribute will be initialized first.



Attributes related to parameter initialization

Reinitialize

Select *Reinitialize* from the *Edit* menu to set all parameters in the current patch to their *Initial Value*. See the patching guide for more information.

Parameter Names

Parameter-enabled objects have three names associated with them: a **Scripting Name**, a **Short Name**, and a **Long Name**. The Long Name and Short Name attributes only affect the display of Live UI objects, and the visible name of the parameter when working in Live.

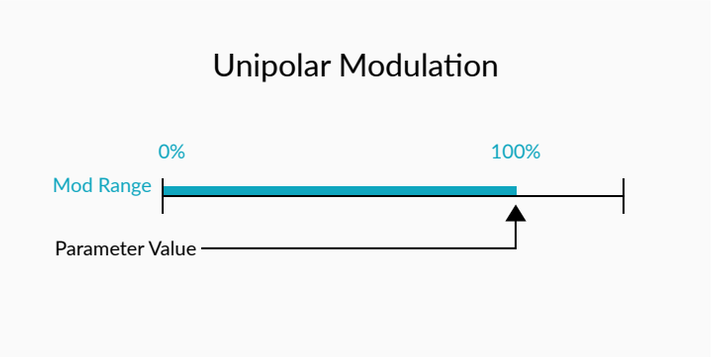
- **Scripting Name**: The name of the *object* as it appears to Max, this is a unique identifier that can be used to refer the object when you're scripting or using pattr. Must be unique to the patcher.
- **Long Name**: The name of the *parameter* attached to the object. If you set up MIDI or Keyboard Mapping for the parameter, this is the name that you'll see in the **Mappings Sidebar**. It is also how the parameter will appear in Live. Must be unique to the entire patcher hierarchy.
- **Short Name**: The display name of the parameter. This affects the display of Live UI objects like live.slider and live.dial, which have a visible text label.

Parameter Modulation

The "Modulation Mode" and "Modulation Range" attributes determine how **Modulation** from Live affects the value of the parameter (for more about Modulation, see Live's documentation). When the parameter is modulated, the modulation value is combined with the current value of the parameter and scaled by the modulation range to determine its final value.

Unipolar

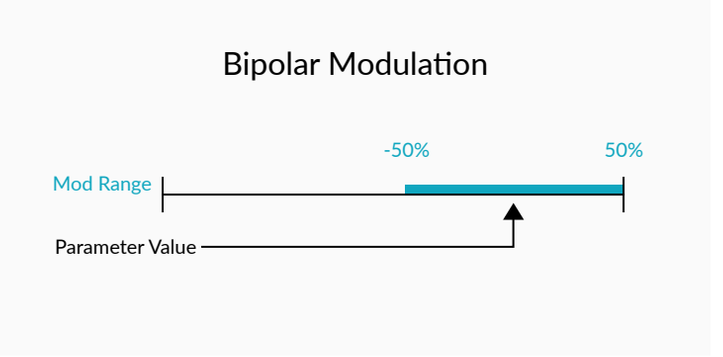
Modulation is between 100% and 0%. At 100%, the parameter isn't modulated at all. At 0%, the parameter takes on its minimum modulation value value.



Unipolar modulation mode

Bipolar

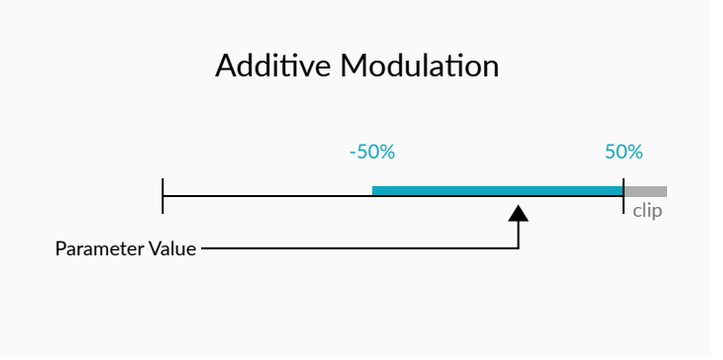
Modulation is between -50% and 50%. The range of modulation depends on the current value of the parameter. The range shrinks as the parameter approaches its minimum or maximum value, so that even as it modulates it never exceeds those values.



Bipolar modulation mode. Notice that the range of the modulation is squished to 27 units on either side, so that the value after modulation never exceeds the maximum of 127.

Additive

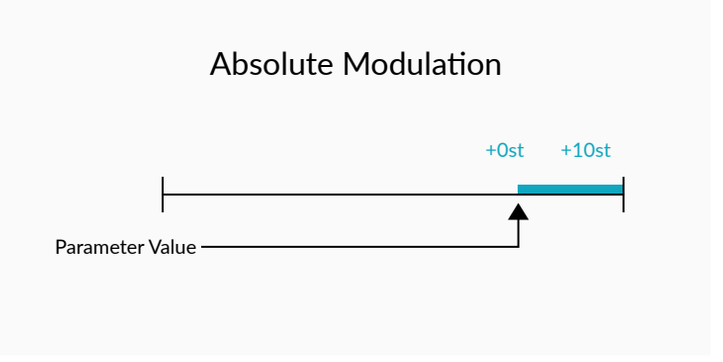
Additive modulation is the same as Bipolar, except the range of modulation never changes. Instead, modulation that would cause the parameter value to exceed its minimum or maximum value instead clips to that value.



Additive modulation mode.

Absolute

Absolute modulation is not based on a percentage but rather the units of the parameter. No matter the range of the parameter, a modulation by 10 absolute units will always modulate by the same amount. Absolute modulation cannot be negative.



Absolute modulation mode

Parameter Data Types

The **Data Type** of a Max Parameter determines the internal storage format for the data.

- **Float**: Can take on any value, including floating-point values, and can participate in Modulation from Live. The default storage type, and perfect for most applications.
- **Int**: Can represent 256 distinct values, with a default range 0 to 255.
- **Enum**: A list of items with user-configurable names. Cannot be modulated (but can be

automated).

- **Blob**: Parameters that cannot be automated or modulated, but can be stored in presets. These non-automatable parameters may be any type of data you can store with a pattr object: single values, lists, dictionaries, arrays or symbols.

> The Data Type only affects the internal storage format of the parameter, and does not change how the parameter's value is displayed. A parameter with the "Data Type" `Int` and the "Unit Style" `Float` will appear to be a decimal number, even though the value after the decimal will always be zero. Similarly, a parameter with the "Data Type" `Float` and the "Unit Style" `Int` will still display as if it were a whole number. In fact, this is the way to have a parameter with more than 256 values that still appears to be a whole number.

Custom Unit Styles

The "Unit Style" attribute lets you change the units associated with your parameter, which for some objects will affect the way the value is displayed.



Millisecond, decibel, and pan unit styles

If you like, you can define your own custom Unit Style. Select `Custom` as the value for the "Unit Style" attribute. The "Custom Units" attribute now lets you create your own units to follow the parameter value. You can use C-style format strings here, so `%0.2f Bogon(s)` would cause a parameter value of 15.5678 to display as `15.56 Bogon(s)`.

Parameter Visibility

The *Parameter Visibility* attribute determines how a parameter is exposed to Ableton Live when your device loads. It affects:

- **Automation:** Whether the parameter is available to Live's automation system
- **Stored:** Whether the parameter is saved in a Live Set, included in Live Presets, and recorded in Live's undo history
- **Visible:** Whether control surfaces (e.g., Push) can access the parameter
- **Type:** Whether the parameter supports multi-value data (e.g., multislider or dictionary) or is restricted to single values (int, float, enum)

Mode	Automation	Stored	Visible	Type
Automated and Stored	✅	✅	✅	int, float, enum
Stored Only	❌	✅	❌	multi-value
Visible	❌	✅	✅	int, float, enum
Visible (Not Stored)	❌	❌	✅	int, float, enum
Hidden	❌	❌	❌	multi-value

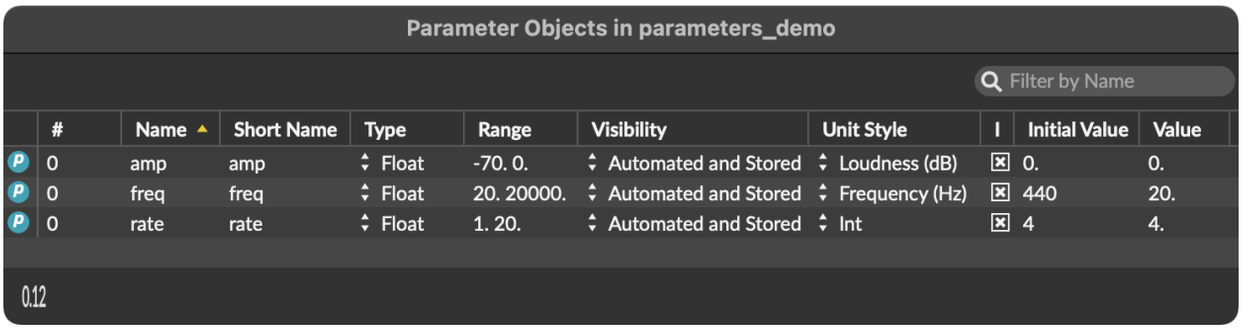
- **Automated and Stored**: This is what you want most of the time for most parameters, for any value that you want to be able to automate in Live's Automation View. Only single-value parameters (int, float, enum) can be automated.
- **Stored Only**: Use for parameters that are part of the device's persistent state but should not be automated or exposed to control surfaces. Typical examples include UI state (e.g., which tab is selected). This mode may include multi-value data (e.g., multislider or dictionary).
- **Visible**: Similar to *Stored Only*, but exposed to control surfaces such as Push. Suitable for mode selectors or simple state controls (e.g., filter type, show/hide a UI panel, apply a temporary gain offset). With this option enabled, you can use the *Undo When Visible* attribute to exclude the parameter from Live's Undo system. This mode is limited to single-value parameters (int, float, enum).
- **Visible (Not Stored)**: Same as *Visible*, but not saved or included in preset storage. With this option enabled, you can use the *Undo When Visible* attribute to exclude the parameter from Live's Undo system. Suitable for momentary or transient controls (e.g., pad-triggered actions).

- **Hidden**: For internal parameters that use *Parameter Mode* for behavior or notifications but should not be exposed to Live or control surfaces. Not stored in the Live Set or presets, not visible to control surfaces, and not recorded in Live's undo history. Effectively equivalent to a standard Max object with *Parameter Mode* enabled.

> If a parameter primarily exists to drive or modulate other parameters, consider setting it to *Hidden*. This can reduce pressure on Live's undo buffer and avoid unintended preset storage interactions.

Parameter Window

From the *View* menu, select *Parameters* to open the **Parameter Window** (this option will be disabled if your patcher does not contain any parameter-enabled objects). This window lists every parameter in the given patcher hierarchy.



Updating parameters from the list

Because the Parameter Window lists all parameters in your patcher, it's a convenient place to change the name, range, initial value, and many other attributes of parameters in your patcher, without you needing to find the specific object that owns the parameter. Double-click on any cell in the parameter table to update the corresponding value for that parameter.

Filtering and ordering parameters

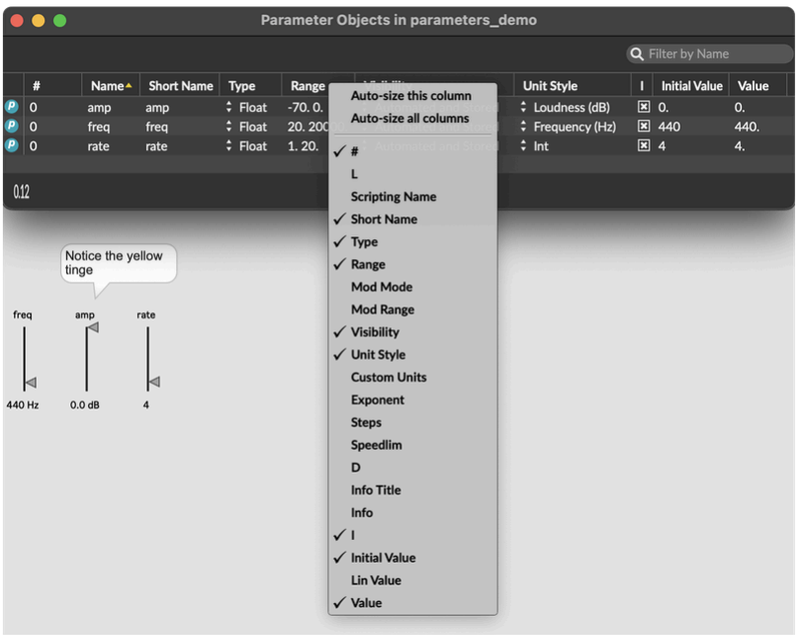
In the top-right of the window, the *Filter Field* lets you filter parameters by name. For example, typing "freq" into this field would show only parameters whose names contained the text string

"freq".

Along the top of the parameter list, the table headers identify the kind of information displayed in each column. Click on a table header to sort parameters by the values in that column. Click again on the same header to toggle between an ascending and descending order. You can also drag on these column headers to reorder them.

Customizing column headers

Right click on any column header to bring up a customization menu for column headers.

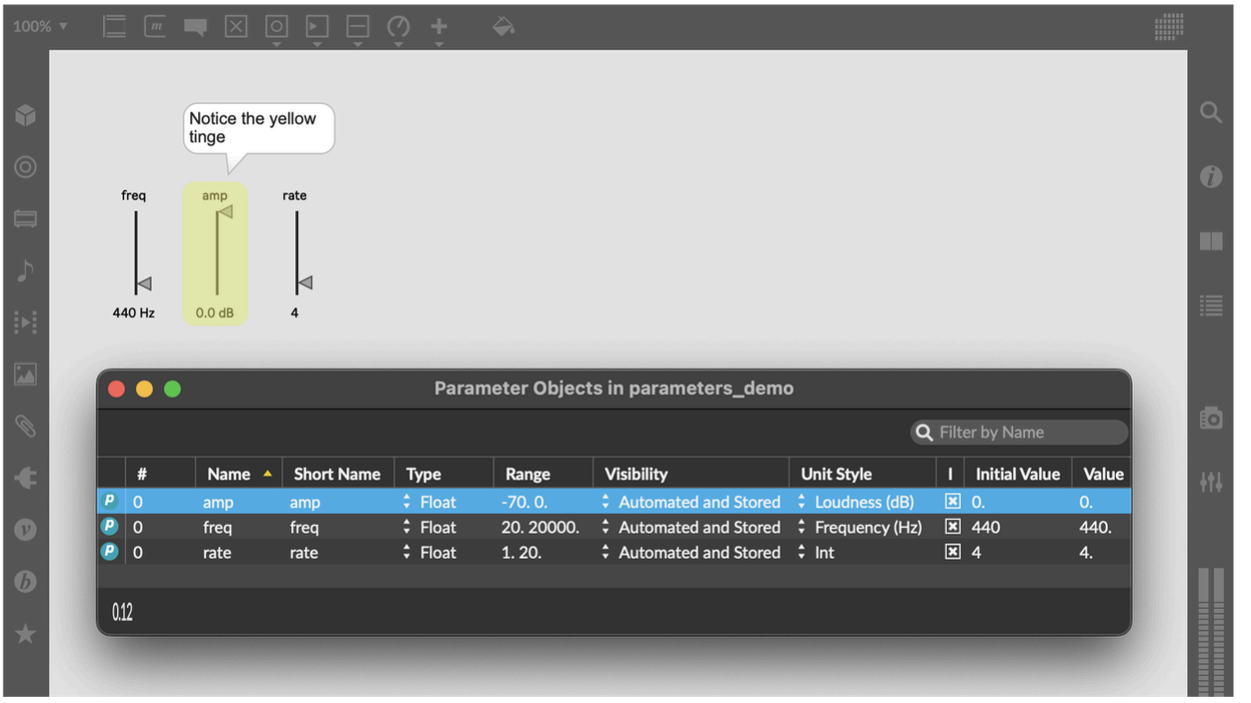


The commands "Auto-size this column" and "Auto-size all columns" will shrink the width of the column or columns to fit the text in the respective column.

Below these, the other menu options let you customize which columns will appear in the parameter list. Some columns have abbreviated names, for example the list option "I" refers to the parameter attribute "Initial Enable". You can hover over the header of any column to see the full name of that header, even if the title is abbreviated.

Finding parameter objects

When you click to select a row in the parameter table, the object that owns that parameter will get a yellow highlight. You can also click the blue "P" button in that parameter's row and select "Reveal in Patcher" to show the same yellow highlight.



Presets and Interpolation

Storing and Recalling Values 491
Interpolating Between Two Presets 494
Multi-Preset Interpolation 495
Non-Interpolating User Interfaces 496
Interpolation User Interfaces 497
See Also 497

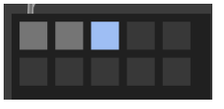
A **preset** is a set of values of one or more objects in your patcher you have stored for later recall. Max has two objects, preset and pattrstorage, whose primary purpose is storing and recalling these sets of values. You can use other objects for this purpose, including table, coll, and dict, but preset and pattrstorage have two principal advantages:

- they can store and recall the state of objects without patch cord connections
- they offer the ability to **interpolate** between two or more preset states

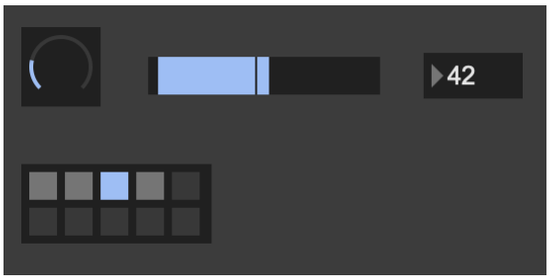
Storing and Recalling Values

The preset Object

The preset object is an easy-to-use way to capture the state of user interface objects. The preset object is itself a user interface object. Each square in its grid represents a different preset.



In the example below, shift-clicking on any square in the preset will capture the current state of all the user interface objects in the patcher -- the dial, slider, and number box. Clicking on the square (without holding down the shift key) will recall the stored state asociated with the sequare (and draw that square to indicate it's current).



Any preset data you store will be saved in the patcher file containing the preset object.

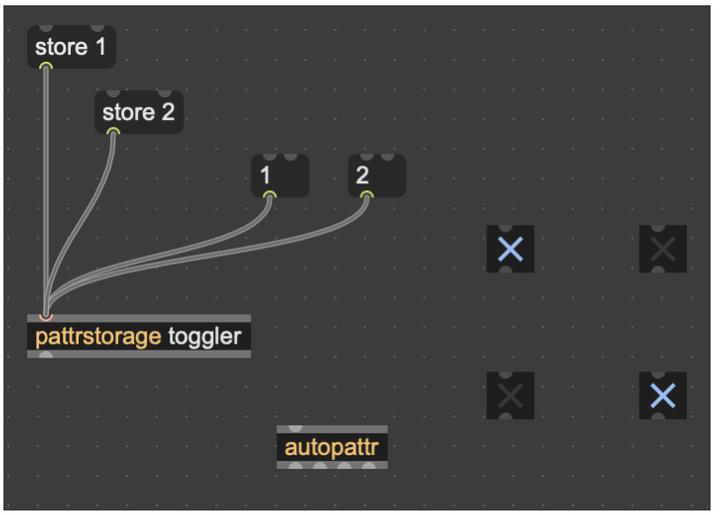
The pattrstorage Object

The pattrstorage object has many features and operating modes that go far beyond what preset is capable of doing. In this brief introduction, we'll discuss how to use pattrstorage in a manner roughly similar to preset.

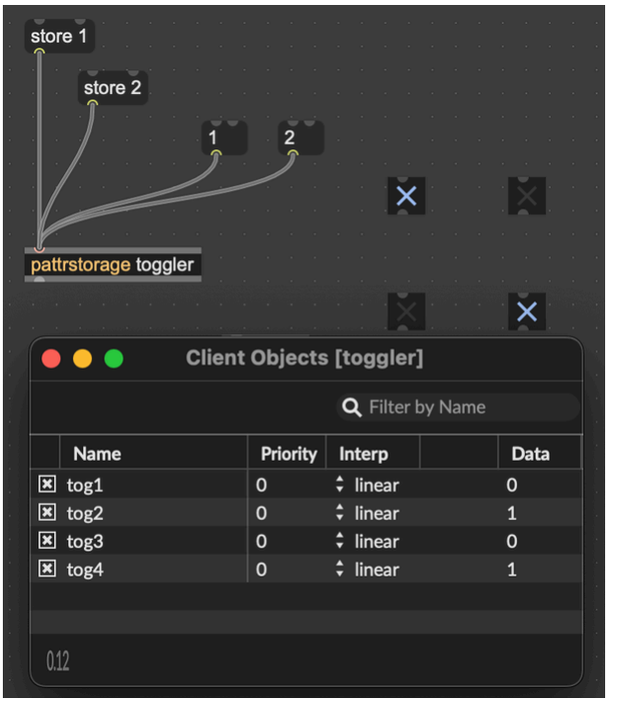
The name pattrstorage comes from its role in storing the values of pattr objects. "Pattr" is short for *patcher attributes* and is an object that creates a single named storage location. While preset stores the values of user interface objects directly, pattrstorage needs a pattr object before it will store anything. Conveniently, it's not necessary to make pattr objects explicitly to store the values of user interface objects. Instead, two things are required:

- ensure each object whose value you want to store has a scripting name -- this will be true for objects such as live.dial that default to parameter mode on but not for toggle, dial or other UI objects that do not enable parameter mode by default. (Parameter mode itself is not required to use pattrstorage, just the scripting name.)
- add an autopattr object to your patcher.

The autopattr will ensure every user interface object with a scripting name has a hidden pattr to go with it.



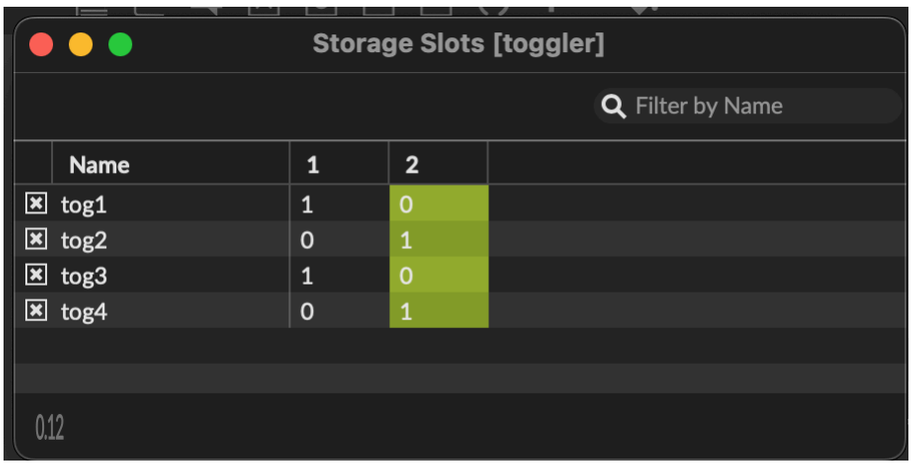
If you double-click on the pattrstorage object, you'll see all of the pattr slots and their current values.



To store a preset with the current values managed by pattrstorage, send the message store followed by a slot number starting at 1.

Send that same number back to recall to pattrstorage to recall the preset. (The preset object accepts the same messages.)

To see the contents of all your presets, you can open the **storage window** by sending the message `storagewindow` to pattrstorage.



Saving pattrstorage data in a file offers more options than preset. The default option is to prompt you to save any newly stored data in a separate file when you close the parent patcher. You can read this data file back into pattrstorage using the `read` message.

Interpolating Between Two Presets

Both preset and pattrstorage offer the ability to *interpolate* between two adjacent presets. Interpolation is an operation that produces a value in between two values. In both objects, you use a float value between two storage slots to specify where in the distance between the slots you want the resulting values.

Here's a simple example with a single object. Suppose the value of a slider for preset 1 is 0 and 1 for preset 2. An float value 1.5 sent to preset or pattrstorage will produce an output value halfway between the two presets, resulting in a value of 0.5 (which is half way between 0 and 1). A float value of 1.9 will produce a value closer to preset 2 than preset 1 -- in this case the slider will be set to 0.9.

When you interpolate between two presets with the preset object, the display of the active slot will show the relative contribution of each of the two slots. The display can help you keep in mind that a value of 1.7 does not mean preset 1 contributes 70% to the value; in fact, just the opposite. The interpolation calculation is based on the *distance* between the value and the whole number. The farther away the float value is from the whole number, the less that preset slot will contribute to the final interpolated outcome. And the closer the float value is to a whole number, the more that preset slot will contribute. Thus 1.99 will be 99% preset 2 and 1% preset 1.



While preset is limited to linear interpolation, pattrstorage offers several other interpolation curves and behaviors via its `interp` message.

Multi-Preset Interpolation

Both preset and pattstorage offer the ability to interpolate using the data in several preset slots with the `recallmulti` message. Multi-preset interpolation is based on a list of weights, where the integer part of the value is the preset slot number and the decimal part is the relative weight of the preset contributing to the outcome. Weights are normalized so they don't have to add up to 1. Here's an example using the preset object below where we want to use preset slots 1, 4, and 7.



`recallmulti 1.5 4.5 7.5` will weigh each preset slot equally. So will `recallmulti 1.2 4.2 7.2`. `recallmulti 1.8 4.0 7.2` will be 80% preset 1 and 20% 7, with nothiing from preset 4.

As with two-preset interpolation, the preset object will display the weights for the most recent `recallmulti` message.



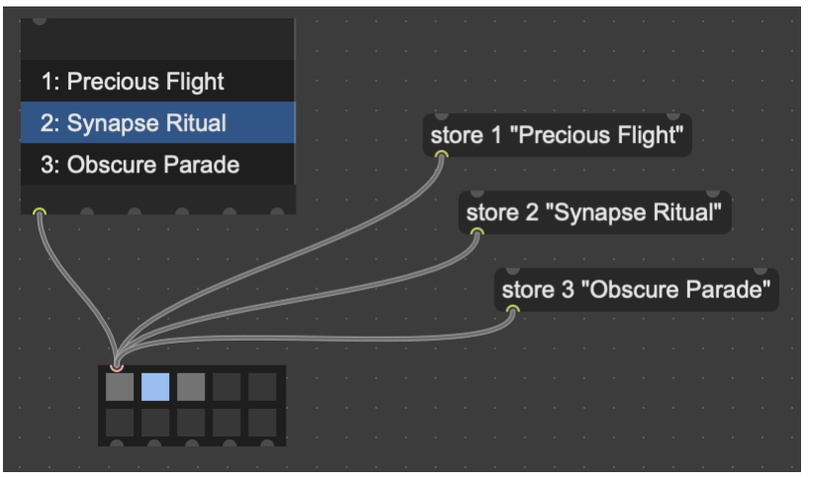
Non-Interpolating User Interfaces

preset As a Front End for pattrstorage

You can use a preset as a front-end for storage and recall of presets in pattrstorage. Give pattrstorage a name with its first argument, then set the `pattrstorage` attribute of preset to this name. After doing this, shift-clicking on a slot in preset actually stores a preset at the same slot in pattstorage, and clicking on a slot in preset recalls the corresponding slot in pattrstorage.

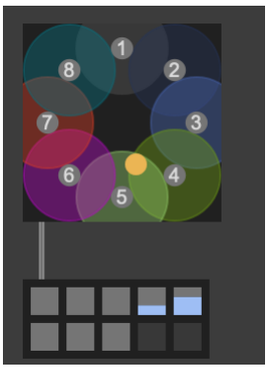
umenu and chooser As a Front End for preset

To display presets in a list format rather than the grid of the preset object, connect a umenu or chooser to the left inlet of preset. The contents of the umenu and chooser will be kept in sync with the storage state and most-recently recalled slot of preset. You can also display names for each preset slot by appending an optional name to the `store` message. `store 2 Cyclical` will name the new preset `Cyclical`. You can see the name when moving the cursor over the stored slot in preset or, if you have a connected umenu or chooser it will show up in the item text.



Interpolation User Interfaces

The nodes object is ideal for multi-preset interpolation with either preset or pattrstorage. Connecting nodes to the inlet of preset creates a visual "node" for each stored preset up to a maximum set by the Max Dynamic Nodes (`maxdynamicnodes`) attribute, which defaults to 8. Dragging the mouse over nodes performs multi-preset interpolation.



Using nodes with pattrstorage requires more work including manually creating the set of nodes that will correspond to the preset slots you want to interpolate and some patching logic to translate the values coming out of nodes to the correct values for the `recallmulti` message. An example is found in the *pattrstorage* tab of the nodes object help file.

See Also

- Snapshots

Saving State with pattr

Saving State with pattr 498
Quickstart 499
Parameters and pattr 499
Including Objects in pattrstorage 501
Binding Objects to pattr 502
Working with preset 502
Paths, Hierarchy and pattrhub 503
Managing Storage 504
Usage with Max for Live Devices 506

When you save a patcher to a `.maxpat` document, you save its structure but not the state of objects in that patcher. The pattr object, along with object like autopattr, pattrstorage, pattrforward, and pattrhub, lets you save and recall the state of a patcher, or a subset of objects within the patcher.

Saving State with pattr

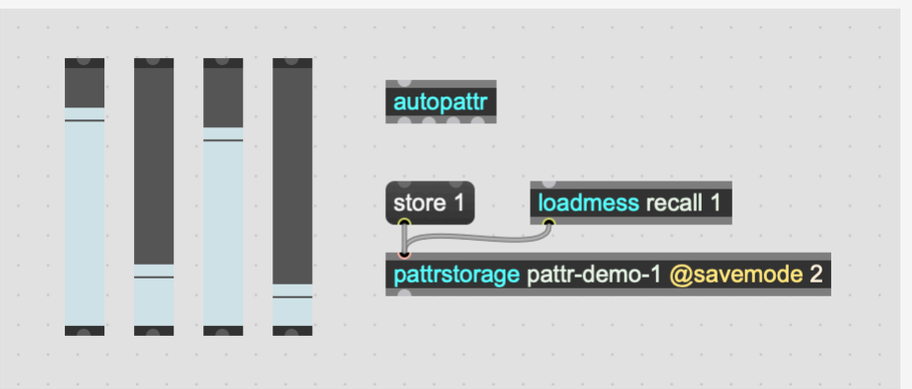
Unlike some applications, Max does not save the *state* of a patcher along with its *structure*. When you save a patcher normally, you're saving the names of all the objects, their connections, their arguments, and their frozen attributes. However, the *state* of the patcher—the value of all your sliders, the contents of your number boxes—is not usually saved. The pattr object family is responsible for helping with saving the state of a patcher.

- We can maintain sets of data from objects throughout a Max patcher hierarchy. For example, with pattr you can control the state of objects inside of patcher and bpatcher objects all from the top level of the patcher.
- We can use pattr objects to remotely set and query the state of objects controlled by the pattr system from anywhere within the patcher.
- The pattr objects store groups of settings as JSON or XML files, allowing us to easily read and edit saved data outside of Max.

- The pattr objects can recall the state of objects in a specific order, avoiding difficulties with, for example, the recalling of a toggle object that starts a process before all the variables are in place for the process to function correctly.
- Not only can we store the states of many objects under a single address, we can also interpolate between these states, allowing for a seamless crossfade between multiple settings.
- The pattr objects feature a high-level interface for viewing and managing the current state of controlled objects and the states that have been saved.

Quickstart

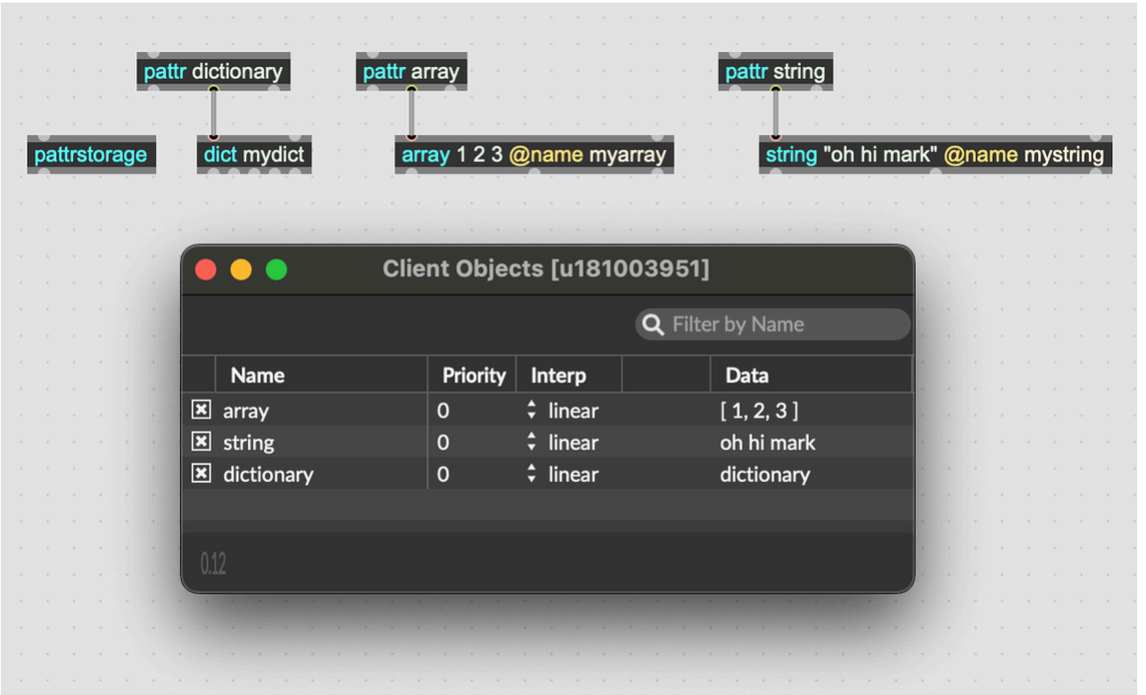
The simplest way to save the state of your patcher using pattr is to add an autopattr object and a pattrstorage object to your patcher. With autopattr, objects that have a **scripting name** will be added to pattrstorage automatically. Now store the state of your patcher by sending pattrstorage the message `store 1`. If pattrstorage has the attribute `@savemode 2`, then when you save your patcher, pattrstorage will save the state of your patcher to a JSON file. When you open your patcher, if Max can locate that JSON file, pattrstorage will load it automatically, and you can send pattrstorage the message `recall 1` to reload the state of your patcher.



Simple setup to save the state of a patch

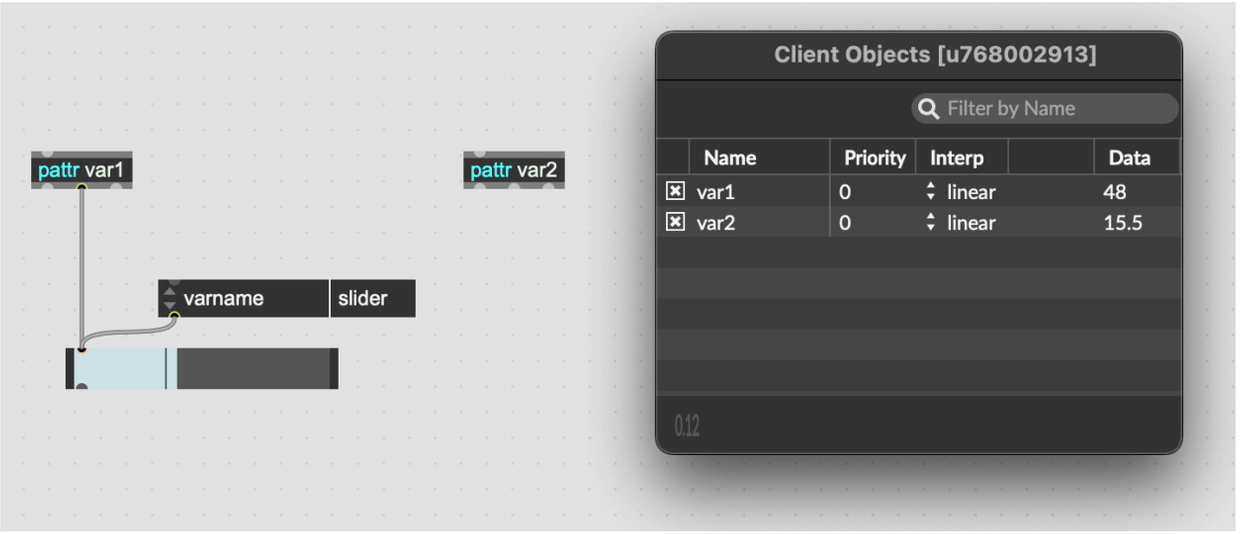
Parameters and pattr

Any object whose internal state can be represented as a parameter can be stored in the pattr system. This includes most UI objects (slider, multislider, nodes), along with any object with the `@parameter_enable` attribute. Also, dictionaries, strings, and arrays can be included in the pattr system.



An array, string, and dictionary, all stored in pattr

The pattr object itself can either be bound or unbound. When bound to an object, via either its middle outlet or the `@bindto` attribute, pattr simply references the internal state of the bound object. If pattr is unbound, then it instead manages its own, internal state.



The pattr 'var1' is bound to the slider. Changing the state of the slider updates the data stored in 'var1', but the data is stored under the key 'var1'. The data in 'var2' is managed internally by the other pattr object, and is not bound to any object.

Complex internal state can be represented as a parameter, so even objects like live.grid can be included in the pattr system and saved to pattrstorage. However, some objects cannot. For example, the contents of a jit.matrix or buffer~ cannot be stored with pattr.

Including Objects in pattrstorage

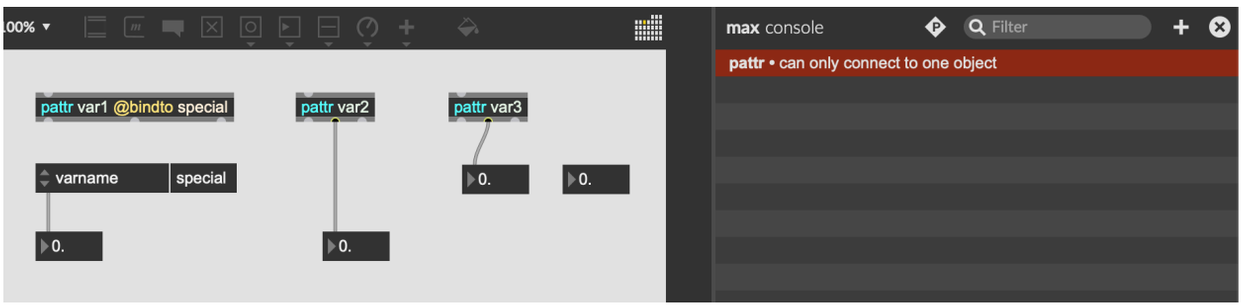
In order to save the state of an object, it must be included in the pattr system.

1. All pattr objects are included in the pattr system automatically. Both bound and unbound pattr objects will be addressable via pattrhub and saved with pattrstorage.
2. Any object with a scripting name (`@varname` attribute) will be included in the pattr system if it's in a patcher with an autopattr object. If autopattr has the attribute `@autoname 1`, it will automatically give all valid objects a scripting name.
3. Any object connected to the left outlet of autopattr.

> The autopattr object will also collect the state of plugins included with vst~ or amxd~. See the autopattr help file for more.

Binding Objects to pattr

Bind a pattr object to pattr-compatible object either using the `@bindto` attribute, or using the middle outlet of pattr. Only one object may be bound to a given pattr at a time.



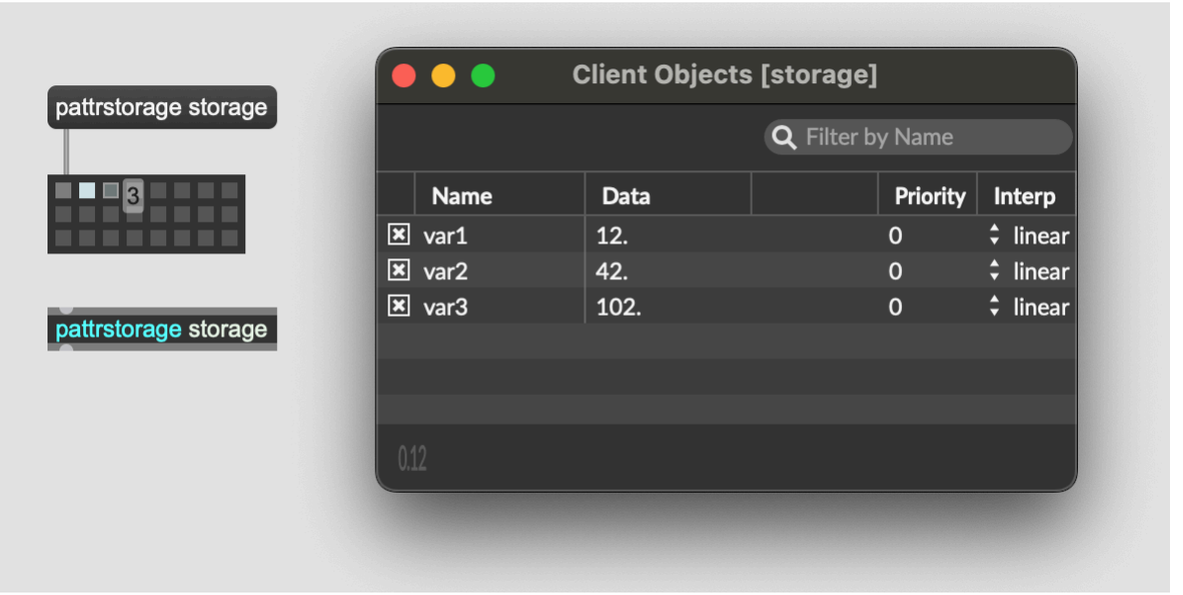
The pattr 'var1' is bound to the number box with @varname special. The pattr 'var2' is bound to another number box using its middle outlet. Finally, the pattr 'var3' tried to bind to two object, which caused an error message.

Automatic Naming

Like many named entities in Max, every pattr object must have a name. If you don't give it a name, Max will assign one automatically. Also, every object that pattr binds to must itself have a name. If you bind a pattr to an object without a name, Max will assign a unique `@varname` automatically.

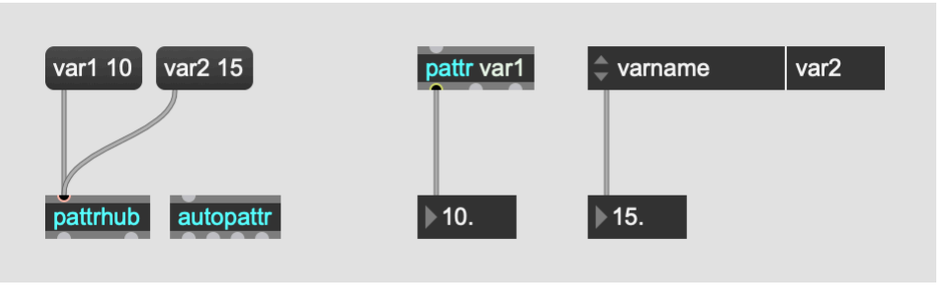
Working with preset

The preset object can be used as an interface to the pattrstorage object. Set the `@pattrstorage` attribute of a preset object to turn that object into a visual representation of the pattrstorage object. Each preset slot in the preset object represents a different storage slot in the pattrstorage object. Use preset to create, update, and delete presets normally, and the state of the pattrstorage object will update automatically.



Paths, Hierarchy and pattrhub

The pattrhub object can forward messages to any object in the pattr system. That includes pattr objects, as well as any object included in the pattr system with autopattr. If a pattr object has the name "var1", send a message like `var1 10` to pattrhub to send a message 10 to the pattr object "var1".



Using pattrhub to send messages to objects in the pattr system, including a pattr object as well as a number box with a scripting name.

Other patchers and pattrmarker

In typical usage, a pattr system is linked to a single patcher hierarchy. If you have two distinct root patchers, then each will have a completely independent pattr system.

However, you can use the pattrmarker object to give a root patcher a global name that can be referenced from anywhere in pattr. If your root patcher contains a pattrmarker object with the text `pattrmarker earth`, then you can send messages to objects in that patcher by using the path prefix `::earth`. See the pattrmarker help file for details.

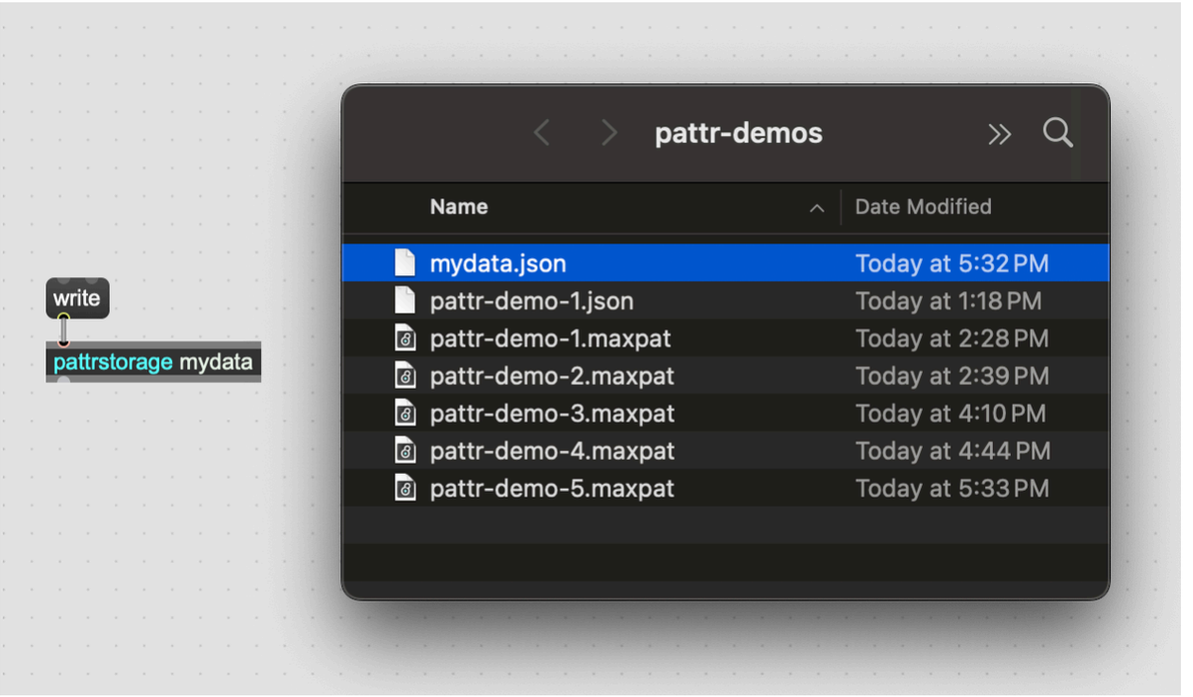
Managing Storage

Similar to the preset object, store the state of pattrstorage by sending it a message like `store 1` to store the current state in slot 1. Recall a given state by sending it a message like `recall 1` to recall the data in slot 1. A floating-point value for recall, like 1.5, will interpolate between two stored slots. The interpolation curve and other parameters can be configured, see the pattrstorage help file for details.

Saving and loading pattr data

Send pattrstorage the `write` message to save its contents to a JSON file. Later, use the `read` message to load the contents of a saved pattrstorage data file.

> When you open a Max patcher containing a pattrstorage object, it will try to automatically load data from it's JSON file. The name of a pattrstorage object is important if you want this automatic loading to work, since Max will look up the data file using the same name as the pattrstorage object.



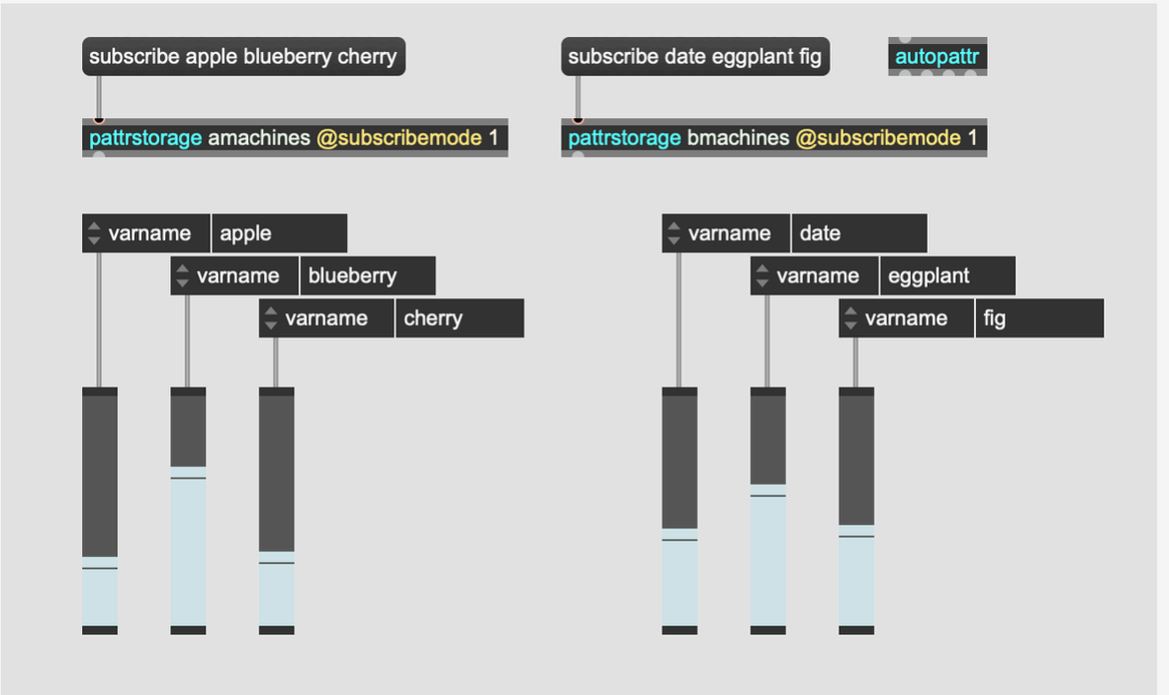
Because the pattrstorage object is named 'mydata', Max will look for the file 'mydata.json' to initialize pattrstorage when the patcher loads.

Saving pattrstorage data can be automated somewhat. If you set the `@savemode` attribute to something other than 0, Max will prompt you to save the contents of your pattrstorage.

@savemode	Description
0 (no autosave or prompt)	Never save automatically or ask to save
1 (prompt to save when object is freed)	When you delete a pattrstorage object, or when the patcher that contains it gets freed or deleted, prompt the user to save. This keeps you from accidentally losing data held in pattrstorage.
2 (attempt autosave when patcher is saved else prompt)	When you save the patcher, pattrstorage will try to save to the same place as it saved last time. If it can't, it will prompt the user to save.
3 (attempt autosave when patcher is closed else prompt)	When you close the patcher, pattrstorage will try to save to the same place as it saved last time. If it can't, it will prompt the user to save.

Multiple pattrstorage objects

Generally you'll only have one pattrstorage object per patcher. However, using the attribute value `@subscribemode 1`, you can explicitly assign objects to pattrstorage with the `subscribe` message.



Use the subscribe message to assign objects to pattrstorage.

Usage with Max for Live Devices

Live-specific user interface objects such as live.dial save their state within Live documents and presets. Standard Max objects will not. If you want to use standard Max objects and have Live save their state, you'll have to attach those objects to a pattr object with `@parameter_enable` enabled. The "Parameter Visibility" attribute must also be set to "Automated and Stored" or "Stored Only".



Because this Max dial is bound to a pattr with @parameter_enable 1, its state will be saved with Live and included in Live presets.

There are some other important differences to keep in mind when building a Max for Live device with pattr objects:

- The autopattr object does not work with Max for Live's parameter system. It will not work with Live's parameter system, so objects that are attached to an autopattr will not be seen by Live. If you want a pattr parameter to appear for modulation etc, you will need to add a pattr object for each instance.
- The pattrstorage object mostly works the same in Max for Live, but there are a few important distinctions to keep in mind, *if the object is in Parameter Mode*.
    - The value of the pattrstorage object in Parameter Mode is its entire storage state (what is ordinarily saved to an external file), rather than the currently recalled slot. This means that Live can save the contents of pattrstorage in the Live set itself, eliminating the need for a separate JSON or XML file. Use of an external file can be disabled by setting `@savemode` to 0.
    - If the pattrstorage object has an Initial Value (Initial Enable is turned on in the Inspector), the `@savemode` and `@autorestore` attributes are ignored and file-less use of the object is assumed.
    - The pattrstorage object has an additional attribute when in Parameter Mode: Auto-update Parameter Initial Value. When this is enabled and Initial Enable is turned on, all changes to the object's storage state will cause the Initial Value to auto-update to the new state.

Snapshots

Patcher Snapshots 508
Effects Snapshots 510
Managing Snapshots 511
Embedding Snapshots 515
Usage with pattr 516
Snapshot-enabled Messages 516
JavaScript Snapshot API 517

Snapshots let you save the state of parameters in your patcher. They are similar to the pattr system, but with some key differences:

- Snapshots can be embedded with a patcher, and do not need to be saved as a separate JSON file.
- There is no built-in mechanism to interpolate between snapshots.
- Snapshots are global, containing all of the parameters in a given patcher, plugin, or device.
- You can give snapshot slots a name

In general, snapshots give you less granular control than working with pattr, but require less configuration to work. Snapshots were designed with a few particular use cases in mind:

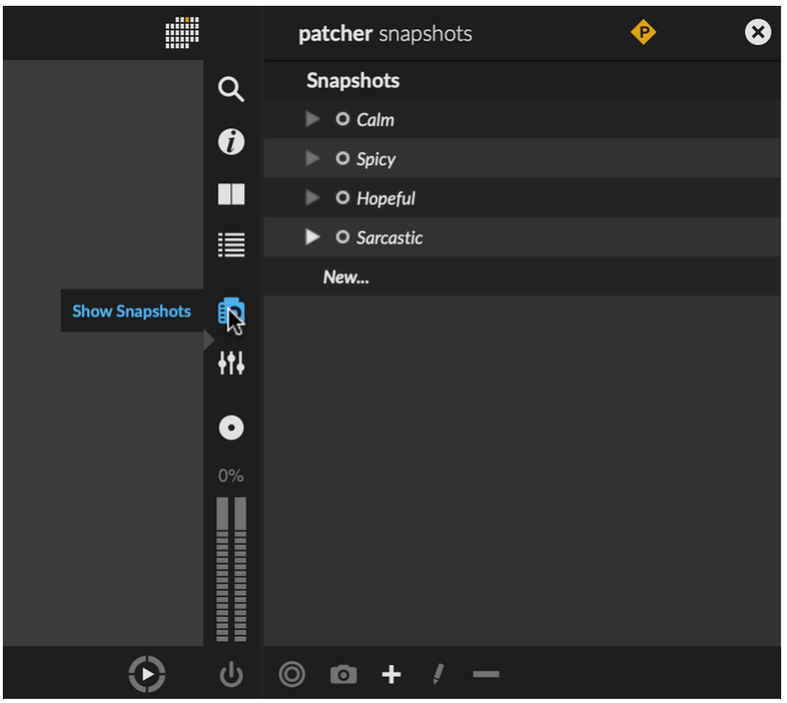
- Saving and restoring the state of all parameter-enabled objects in a patch
- Saving and restoring the state of a VST or Audio Unit plugin, or a Max for Live Device
- Saving and restoring the state of a rnbo~ object.

Snapshots use the Parameter system to determine what to save; you will need to set the Parameter mode enable for each UI object whose state you wish to save.

Patcher Snapshots

A patcher snaphot contains the state of all parameters in a patcher hierarchy. Since snapshots work with the parameter system, an object must have parameter mode enabled in order to be saved in a snapshot.

> If you're unable to create new patcher snapshots, it might be because there are no parameter-enabled objects. Snapshots will only be enabled if there is at least one parameter-enabled object in your patcher. You can enable parameter mode on most UI objects from the inspector.



Here at the top of the *Snapshots* sidebar view, you can see the text "patcher snapshots", indicating that these are snapshots that belong to the patcher itself. With nothing selected, the *Snapshots* sidebar view will display patcher snapshots. If you select a vst~, rnbo~, or amxd~ object, you'll see that the sidebar view updates to display snapshots for that object.

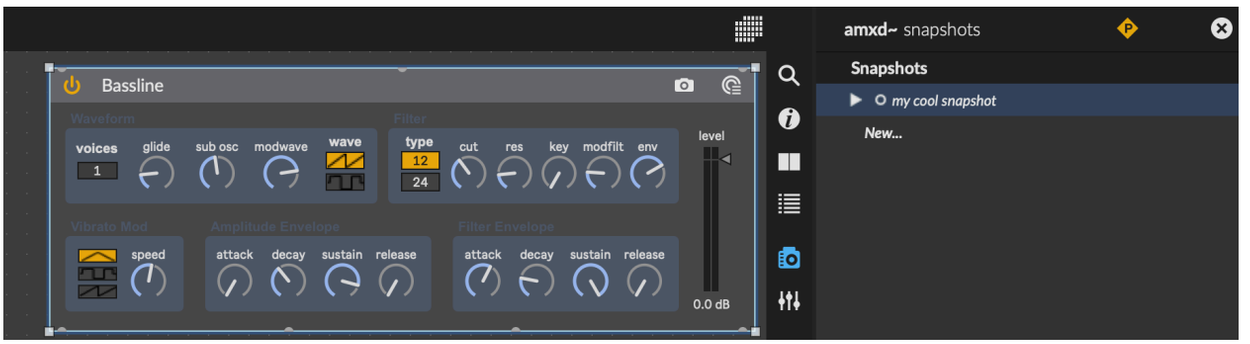
Patcher snapshots are saved and recalled at the top-level of a patcher hierarchy, but will save subpatcher parameters.

> Subpatches can't have snapshots separate from the root patcher. If you need this kind of behavior, you can create a Max for Live device and host it in a patcher with the amxd~ object.

Effects Snapshots

Effects snapshots store the parameter states of plugins, Max for Live devices, or RNBO devices hosted by vst~, amxd~, or rnbo~ objects.

When you select one of these objects in your patcher window, the snapshots pane will display any snapshots associated with they selected device.



Unlike patcher snaphsots, which are always saved with the patcher, effects snapshots are not embedded by default. Instead, they are saved in the Max 9 Folder in a folder called *Snapshots*. This means that when you create a snapshot for a VST, Audio Unit, or Max for Live Device, it's available throughout Max, no matter where it was created. See embedding snapshots for more.

> If you want to share a snapshot that you created, you can share the associated snapshot file located in the Max 9 Folder. You can also use Max projects to automatically collect snapshot dependencies, which may make sharing snapshots easier.

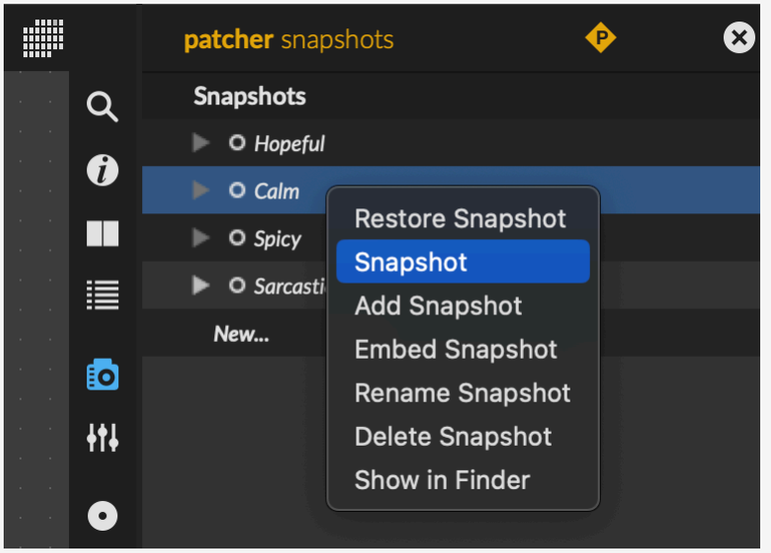
The header/title bar for vst~ and amxd~ objects lets you create and recall new snapshots when the patcher window is locked. The camera icon creates snapshots, and the circular icon to the right will recall snapshots.



Managing Snapshots

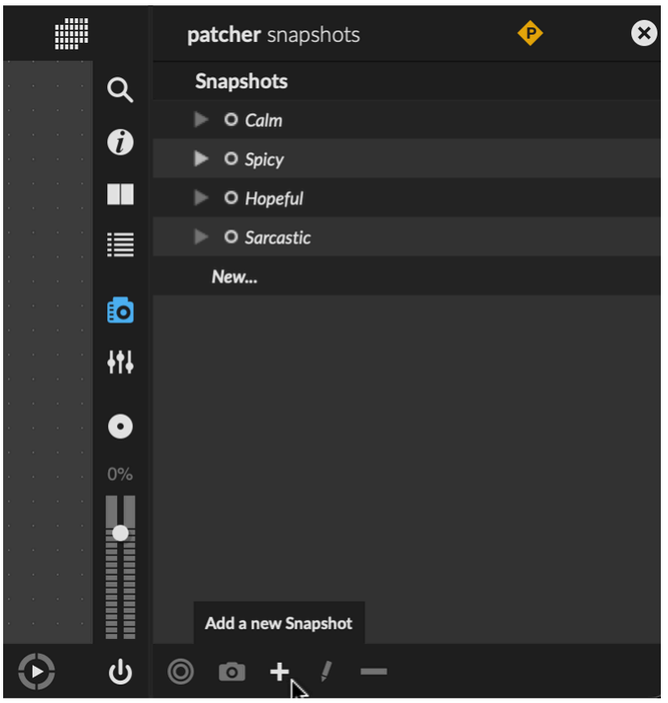
To view, create, load, rename, and delete snapshots, open the *Snapshots* sidebar view by clicking the *Show Snapshots* icon in the right toolbar.

All of the following snapshot options are also available by right-clicking on any snapshot.



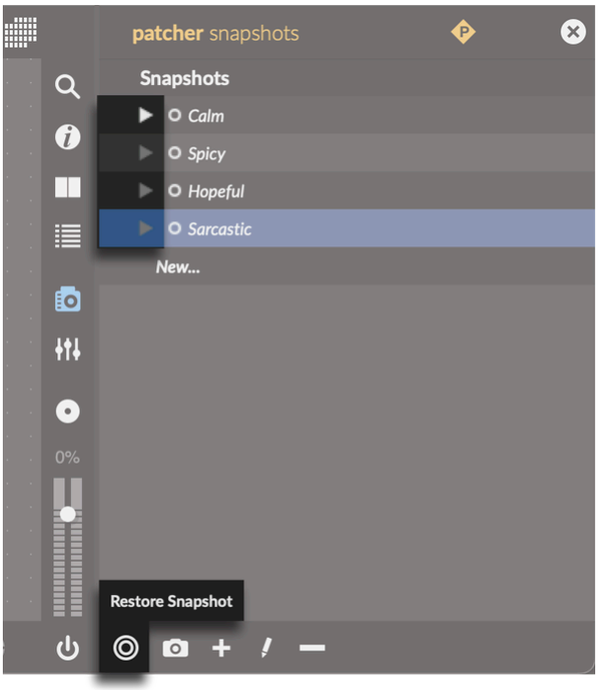
Creating Snapshots

To create a new snapshot, click on the *Add a new Snapshot* button in the bottom of the *Snapshots* sidebar view.



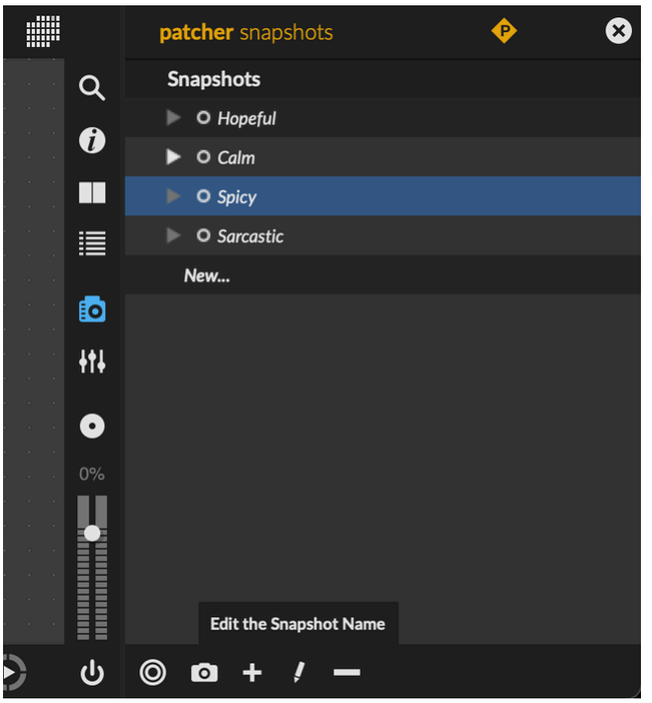
Recalling Snapshots

Recall a snapshot by either clicking on the triangle next to a snapshot, or by clicking on the *Restore Snapshot* icon in the bottom toolbar.



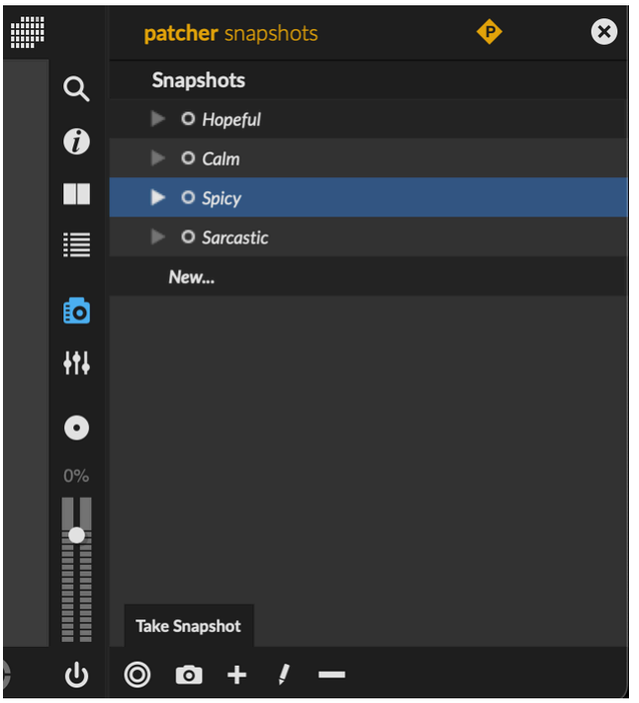
Renaming Snapshots

Rename a snapshot either by double-clicking on the name of a snapshot, or by clicking on the *Rename Snapshot* icon in the bottom toolbar.



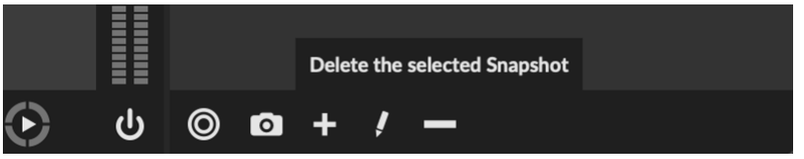
Modifying Snapshots

You can modify a snapshot by clicking the *Take Snapshot* button in the bottom toolbar. This will overwrite the currently selected snapshot with the current parameter values of the selected patcher or, device, or plugin.



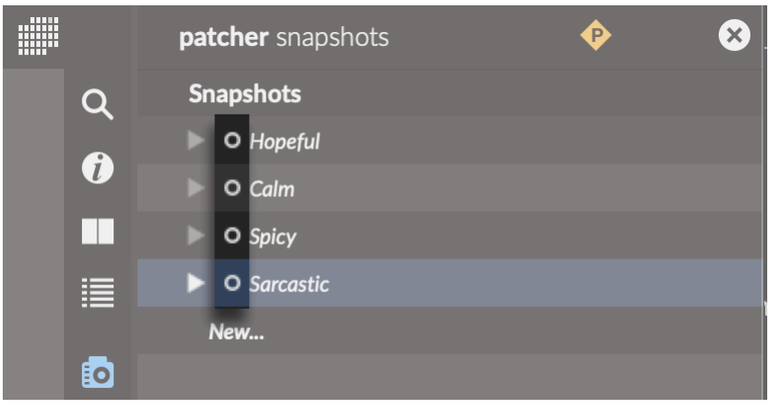
Deleting Snapshots

Delete a snapshot by clicking the *Delete the selected Snapshot* icon in the bottom toolbar.



Embedding Snapshots

Patcher snapshots are always embedded with the patcher, but snapshots for plugins, amxds, and RNBO devices are not. Click on the circle icon next to any snapshot to embed it with the current patcher.



Whether they are embedded or not, snapshots are always saved in the *Snapshots* folder in the Max 9 Folder. These snapshot files are named according to the name of the current patcher, so it's good practice to name your patcher file prior to creating snapshots.

Usage with Projects

When you use a vst~, amxd~, or rnbo~ object as part of a Max Project, any snapshots of that object become dependencies of the project. When you consolidate your project, those snapshots will be copied to the project directory. This lets you share your project with others, including any snapshots that the project might depend on.

Usage with pattr

Snapshots can be easily integrated into your pattr workflow. Using a pattrstorage object along with pattr objects or an autopattr object, the internal state of your VST, AU or AMXD can be recalled.

See the pattr and autopattr help files for example usage (under the "snapshots" tab).

Snapshot-enabled Messages

All snapshot-enabled objects (amxd~, vst~, rnbo~, and thispatcher) understand the messages:

- `snapshot` [userpath (optional)] [index (optional)] [name (optional)]
- `restore` [index (optional)]

- `addsnapshot` [userpath (optional)] [index (optional)] [name - (optional)]
- `deletesnapshot` [index]
- `setsnapshotname` [index] [name]
- `deletesnapshot` [index]
- `setembedsnapshot` [index] [embedstate]
- `movesnapshot` [srcindex] [dstindex]
- `exportsnapshot` [srcindex] [userpath]
- `importsnapshot` [dstindex] [userpath]

JavaScript Snapshot API

For advanced users and those creating standalone patchers, Snapshots can be accessed via the Snapshots API. See the JavaScript Snapshot API for more information.

Patching

Conversion Cheat Sheet

Message to Audio 519
Audio to Message 520
Signal to Event 521
Unit Conversions 522
Multichannel 524
List to buffer~ 525
buffer~ to List 526
Audio to Matrix 526
Matrix to Audio 528
Matrix to Texture 529
Texture to Matrix 529
Matrix to Messages 530
Message to Matrix 531
Matrix Upsampling/Downsampling 532
Matrix Color Conversion 533
Thread Priority Conversion 534
Dictionaries 535
Number Formats 536
String to Array 538
Array to String 538
buffer~ to Array 538
Array to buffer~ 539
Array to buffer~ (multiple channels) 539

Message to Audio



Convert a message to a signal without any smoothing

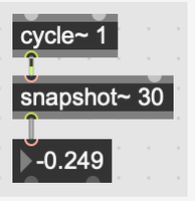


Convert a message to a signal with linear smoothing

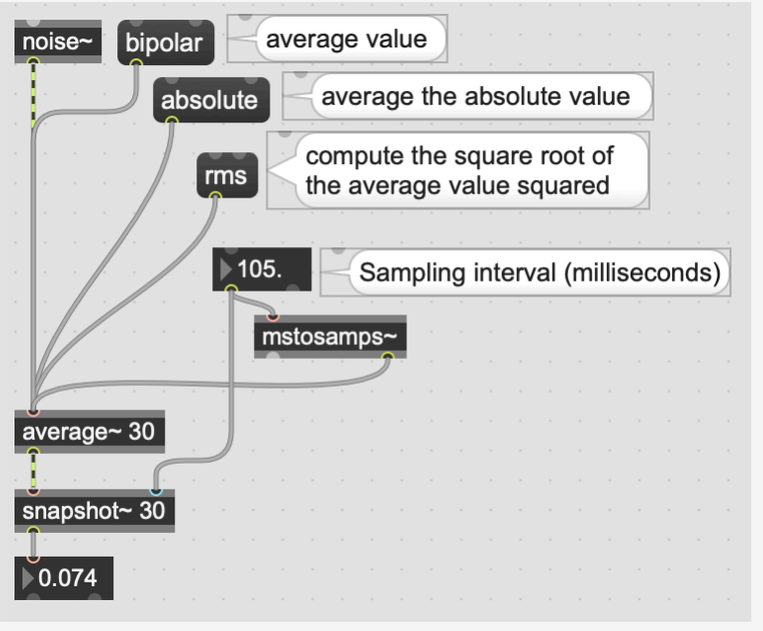


Convert a message to a signal with logarithmic smoothing

Audio to Message

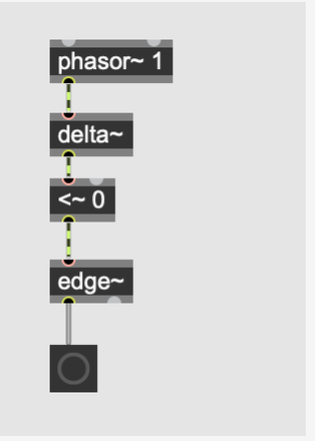


Convert audio to a stream of messages



Analyze an audio stream for peaks, average, or RMS

Signal to Event

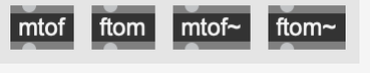


Signal to Event

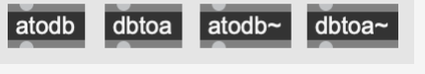
Unit Conversions



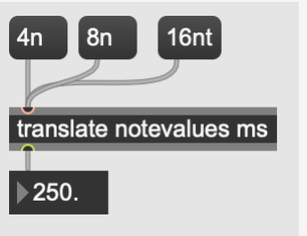
Samples to Milliseconds



Frequency to MIDI Pitch

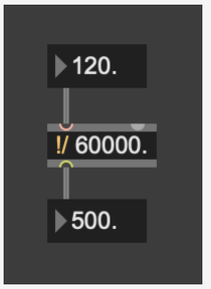


Amplitude to Decibel



Note Value to Millisecond

> See Time Value Syntax for more details.



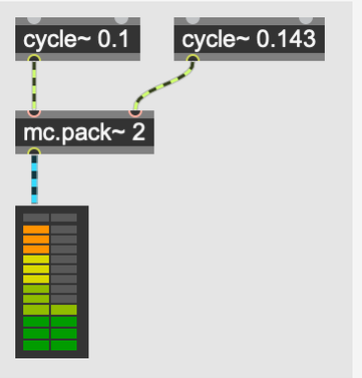
Tempo to Millisecond

> This same patch will convert milliseconds to tempo (kinda cool huh?)

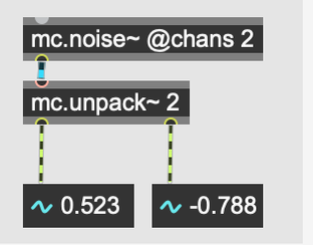


Linear to Quadratic

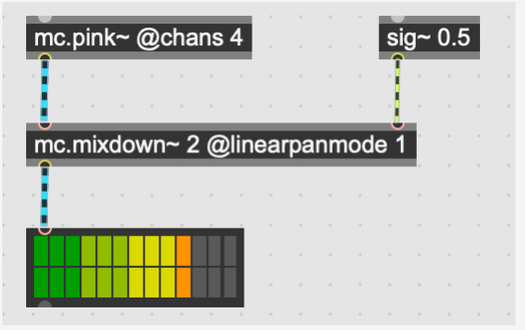
Multichannel



Single channel to multichannel

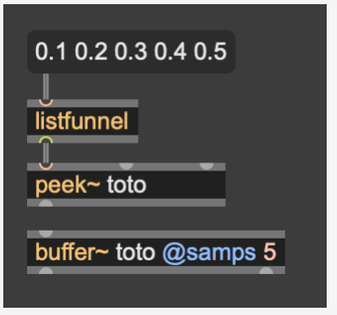


Multichannel to separate channels

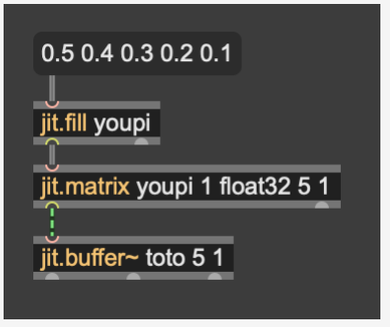


Multichannel to stereo mixdown

List to buffer~

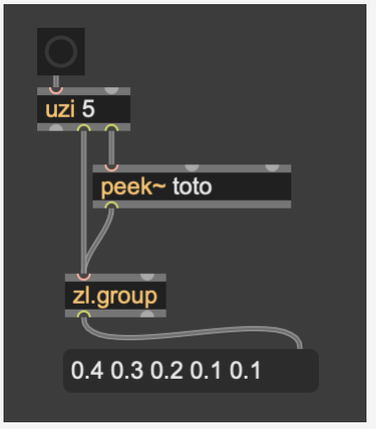


Fill using peek~

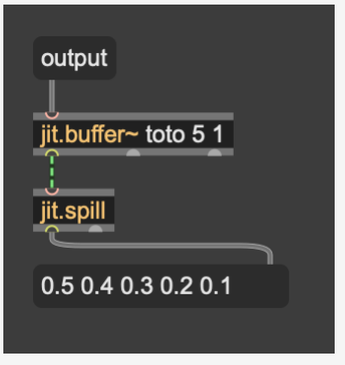


Fill using jit.buffer~

buffer~ to List

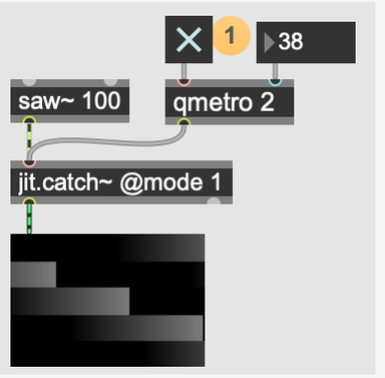


Using peek~ and zl.group

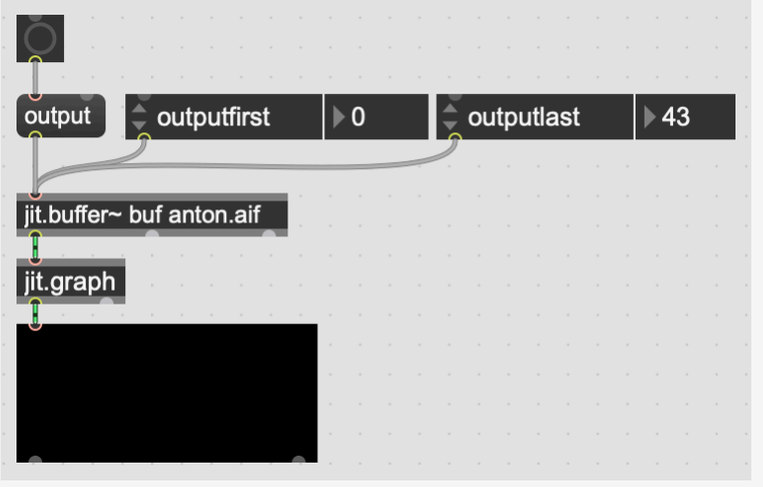


Using jit.buffer~ and jit.spill

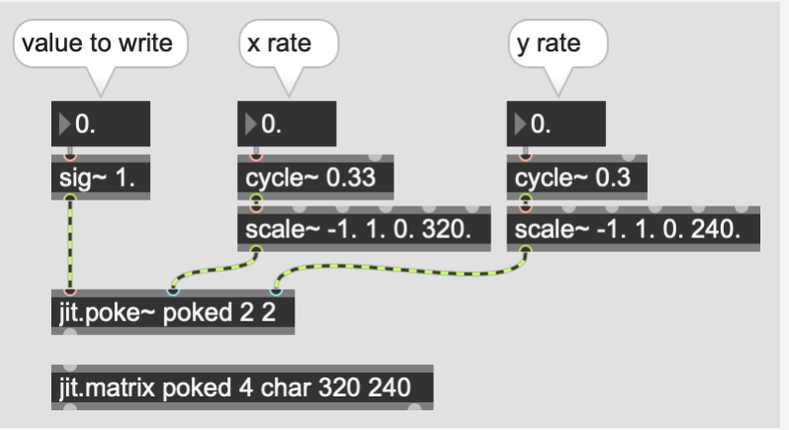
Audio to Matrix



Filling up a matrix

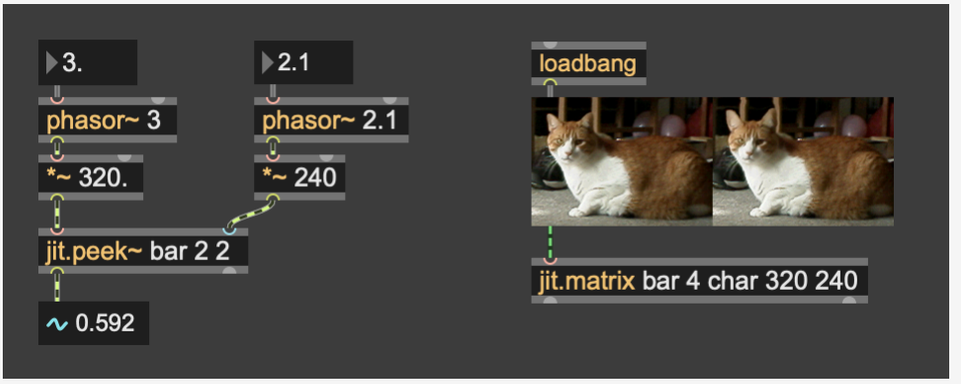


Reading from a buffer

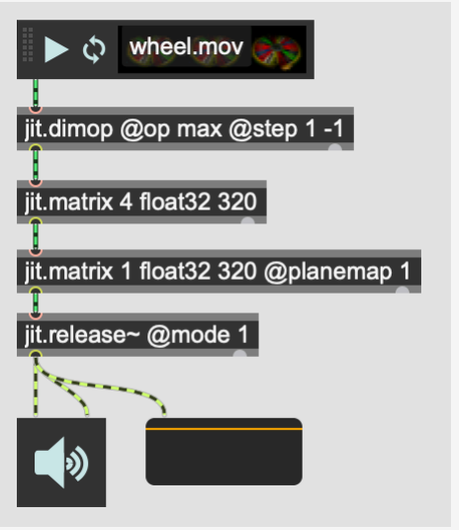


Writing into a matrix

Matrix to Audio

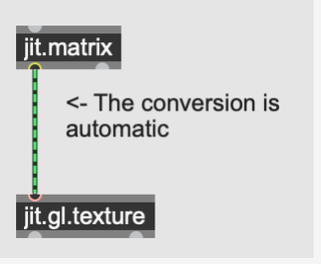


Scanning a matrix



Read a video as an audio signal

Matrix to Texture



Convert a matrix to a texture. It really is that simple.

Texture to Matrix

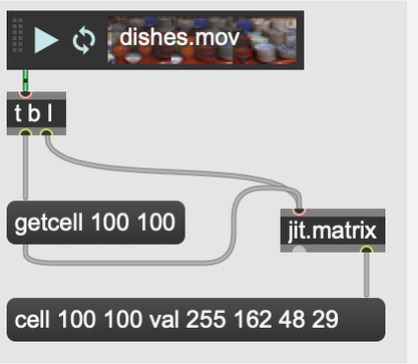


Convert a texture to a matrix

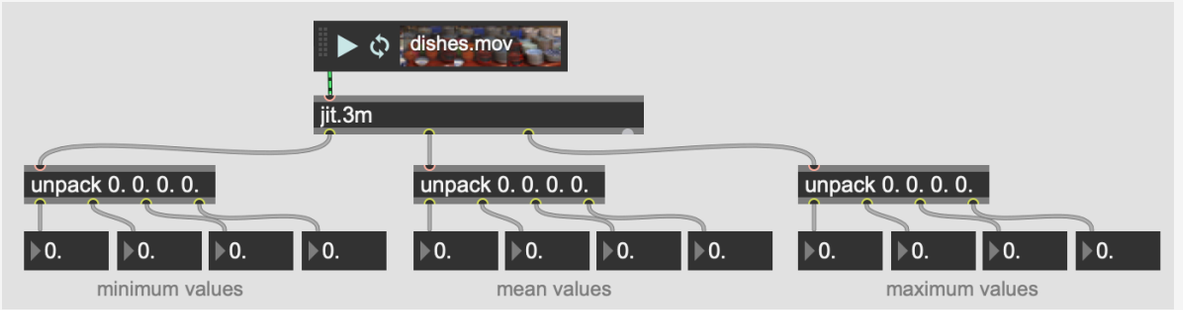
Matrix to Messages



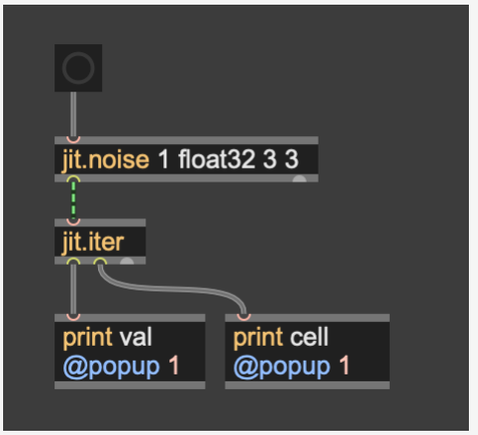
Get cell values from a matrix as a list



Read a specific cell from a matrix

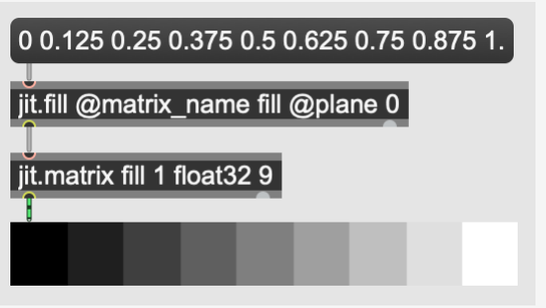


Get a representative number from a matrix



Output the cells of a matrix one by one

Message to Matrix



Fill up a matrix using a list



Set a single cell in a matrix

Matrix Upsampling/Downsampling



Downsample a matrix, no interpolation

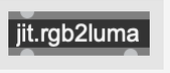


Upsample a matrix with interpolation

Matrix Color Conversion



Red-Green-Blue to Hue-Saturation-Lightness

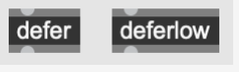


Red-Green-Blue to Luminance

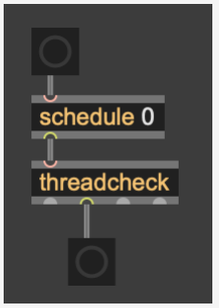


More color space conversions

Thread Priority Conversion



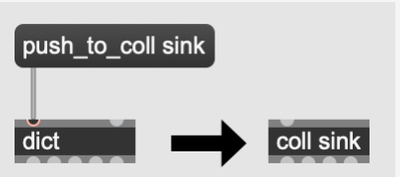
Move a high-priority message to the low priority queue



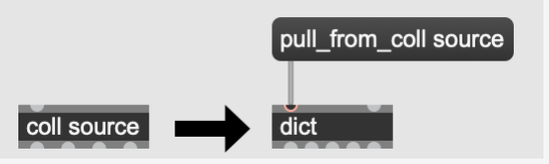
Move a low-priority message to the high-priority scheduler

> Note that this won't make the message get processed "sooner", but if you know what you're doing there may be situations where it's useful.

Dictionaries



Convert a dictionary to a coll

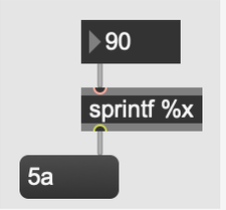


Convert a coll to a dictionary

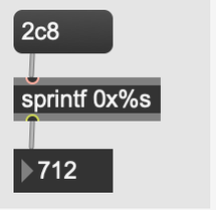


Add an array to a dictionary

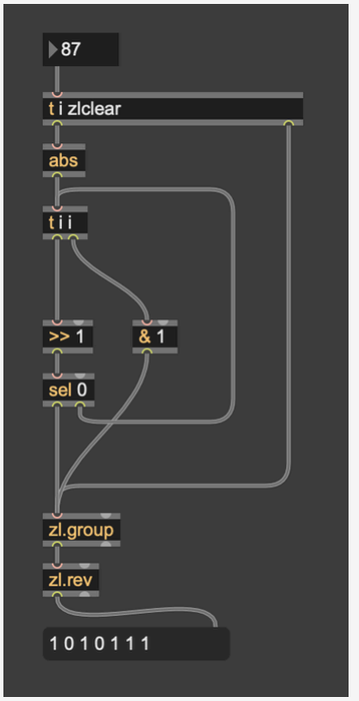
Number Formats



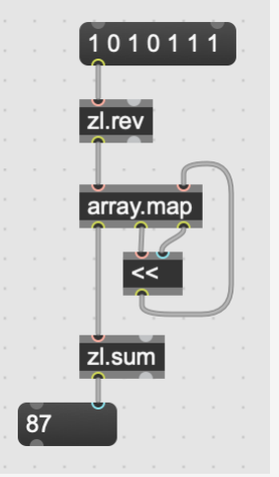
Decimal to Hex



Hex to Decimal



Decimal to Binary



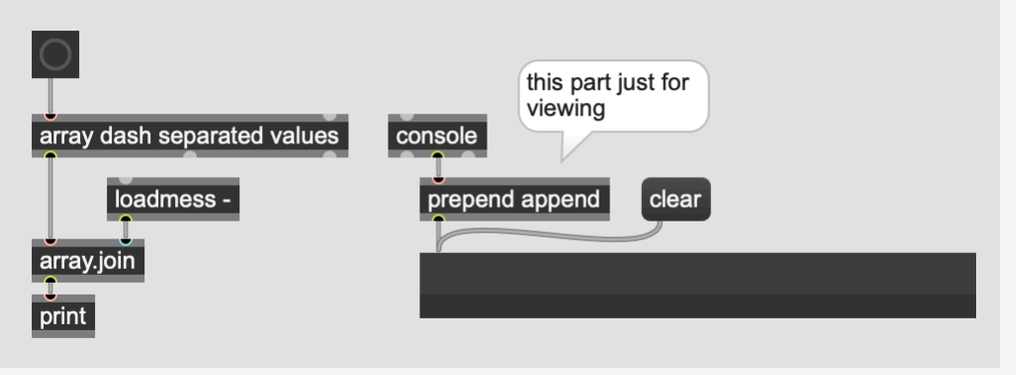
Binary to Decimal

String to Array



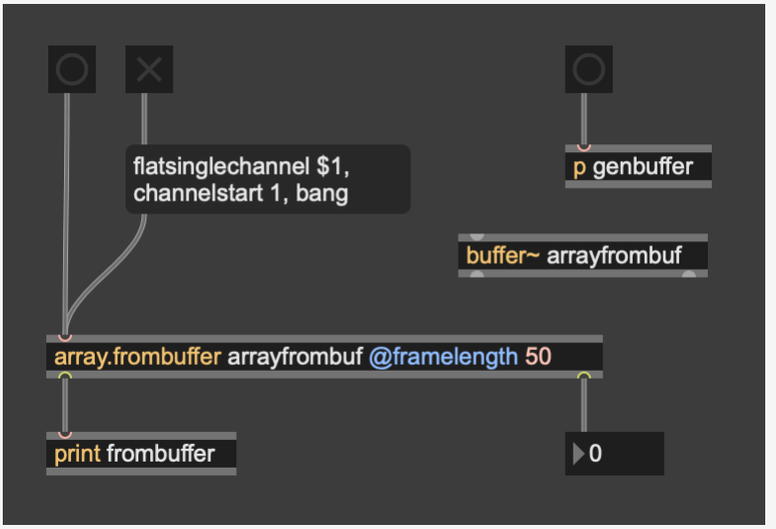
String to Array

Array to String



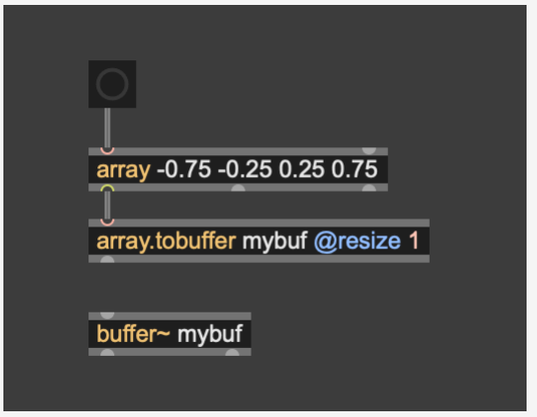
Array to String

buffer~ to Array



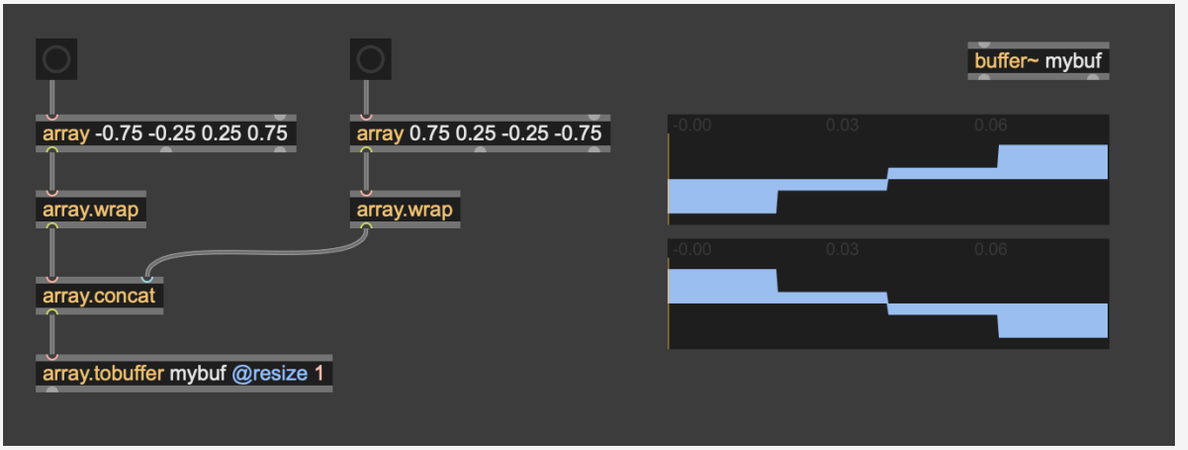
buffer~ to Array

Array to buffer~



Array to buffer~ using array.tobuffer

Array to buffer~ (multiple channels)



Each nested array is its own channel

Messages

Messages Overview 541
Interpreting Messages 542
Message Box 543

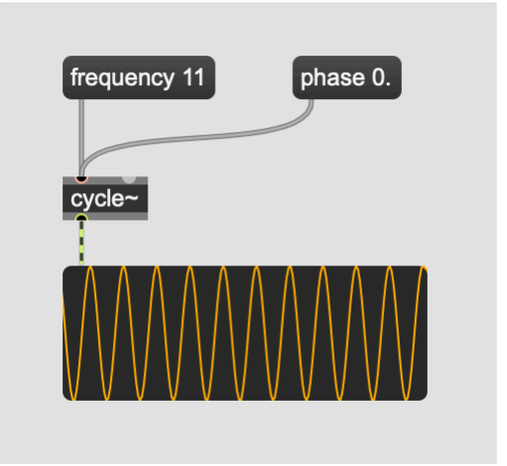
Messages Overview

Max objects communicate with each other by sending each other *Messages*. A message consists of one or more *Atoms*, where each atom can be a *bang*, a *symbol*, an *int*, or a *float*.

- bang: empty atom, a bang means "do it"
- symbol: one or more alphanumeric characters
- int: a whole number
- float: a number with a decimal part

If a message contains multiple atoms, it's referred to as a list. The atoms in a list are separated by spaces. If a symbol atom must contain spaces, the whole symbol should be escaped with double-quotes. When an object receives a message, what the object does depends on the first element of the message. For example, the cycle~ oscillator object can respond to messages by changing its frequency or phase.

> This first element of a Max message, which determines how an object will interpret the message, doesn't have a special name in Max, but you may sometimes hear it called a *selector*, since it functions in a similar way to a *selector* in programming languages like Smalltalk or Objective-C.

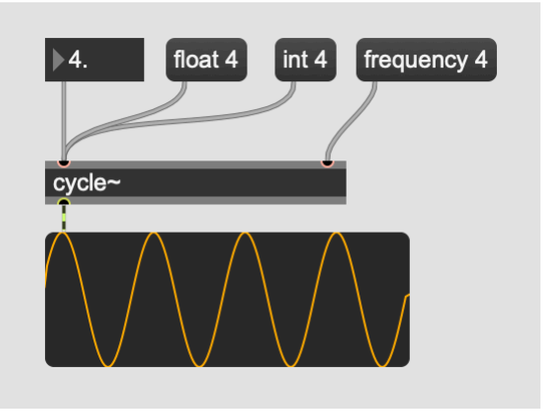


When a cycle~ object gets a message starting with 'frequency', it responds by changing its internal frequency. When it gets a message starting with 'phase', it responds by changing its phase instead.

Interpreting Messages

The way an object responds to a message depends on two things: the contents of the message, and the inlet that receives the message. Generally, every inlet is bound to a specific function, but you can override that function depending on how you format your message. For example, the second inlet of the cycle~ object sets the phase of the oscillator, and when cycle~ receives a *float* or *int* value in its second inlet, it will interpret that number as changing the phase of the oscillator. However, if you really want to, you can send the cycle~ a message like `phase 0.5` to any inlet, and the cycle~ object will always update its phase. This might be a confusing way to design your patcher, but it's allowed.

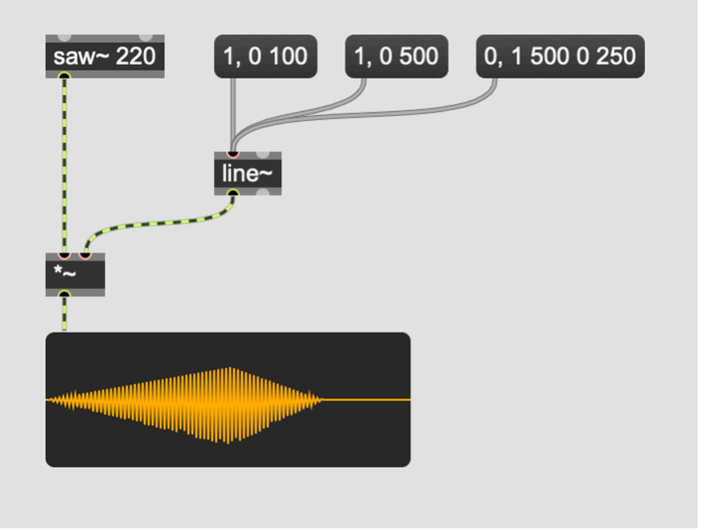
Because of this flexibility, an object could respond to multiple messages in the same way. When a cycle~ object gets a float in its first inlet, it will change its frequency. The same is true if it gets an *int* message, or the message `float 440`. Also, you could send the message `frequency 440` to any inlet, and the result would be the same.



These messages, when sent to cycle~, all have the same effect. The `frequency 4` message will set the frequency, even though it's being sent to the right inlet.

Message Box

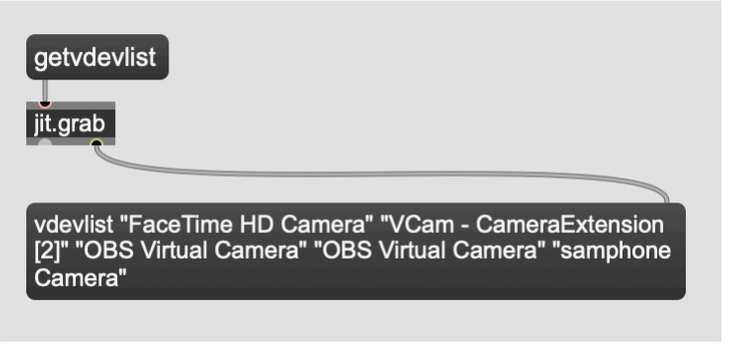
A *Message Box* is a special kind of UI object that contains a single message. When you click on a message box or send it a *bang*, the message box will respond by sending out its contents as a message. Message boxes are extremely flexible and fulfill all kinds of functions in a Max patcher. You can use a group of message boxes to store values, to format other messages, or to view the messages that are passing between other objects.



Clicking on any of these messages would send a different list to line~, triggering an enveelope with a different shape

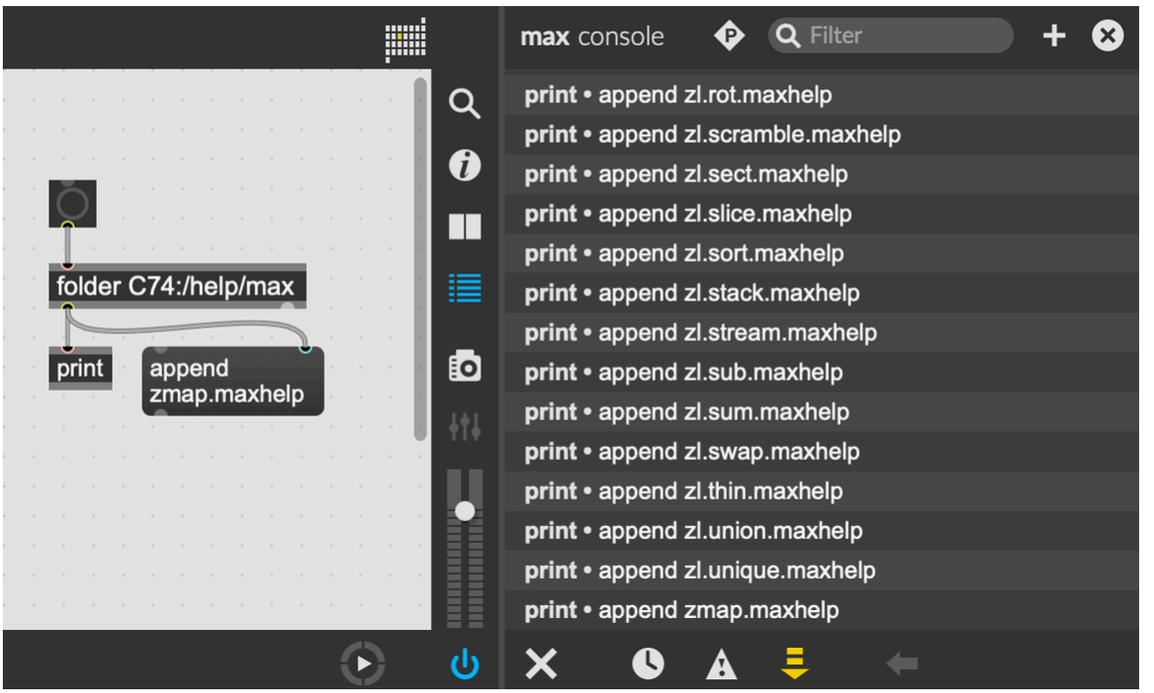
Viewing Messages

The easiest way to view a message is to connect a patch cord to the right inlet of a message box. Lists of any kind can easily be viewed this way.



Using the right inlet of a message box, it's possible to view even very long and complex messages

One thing to keep in mind when using a message box this way is that the message box will only display the last message that it received. Sometimes it's not clear that a message box has actually received multiple messages, since only the last message received will be displayed. When you want to view all of the messages that have been sent along a given patch cord, you can use the print object to log messages to the Max console.

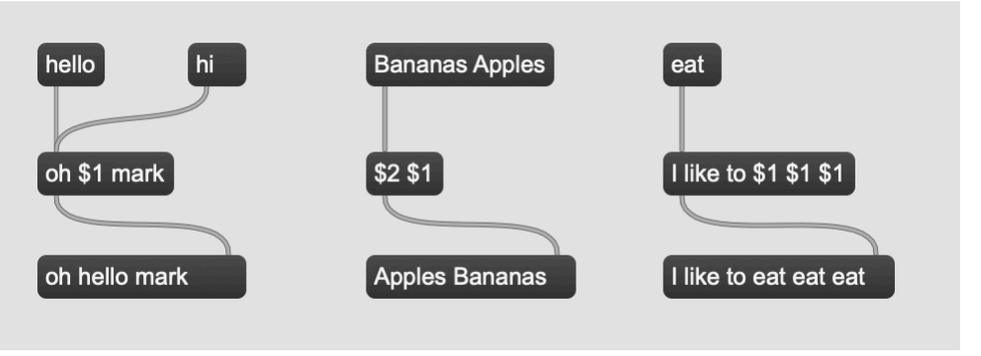


The print object will log all received messages to the console, where the message object itself will only show the most recent message.

Finally, you can set **breakpoints** and **watchpoints** on your patch cords, for even more precise control over how messages move through your patcher. See Debugging and Probing for more details.

Dollar sign replacement

If a message box contains the special `$` character, followed immediately by one of the digits `1-9`, that message box will re-format any messages that come through it. The characters `$1` will be replaced by the first element of the incoming message, the characters `$2` will be replaced by the second element, and so on. Characters like `$1` and `$2` do not need to appear in order, and they can appear any number of times. So the message `$2 $1` will reverse the first two element of an incoming message, and the message `$1 $1 $1` will repeat the first element three times. If you want a symbol to begin with `$1` and for those characters not to be replaced, add the backslash `\` character before the `$` to escape it.



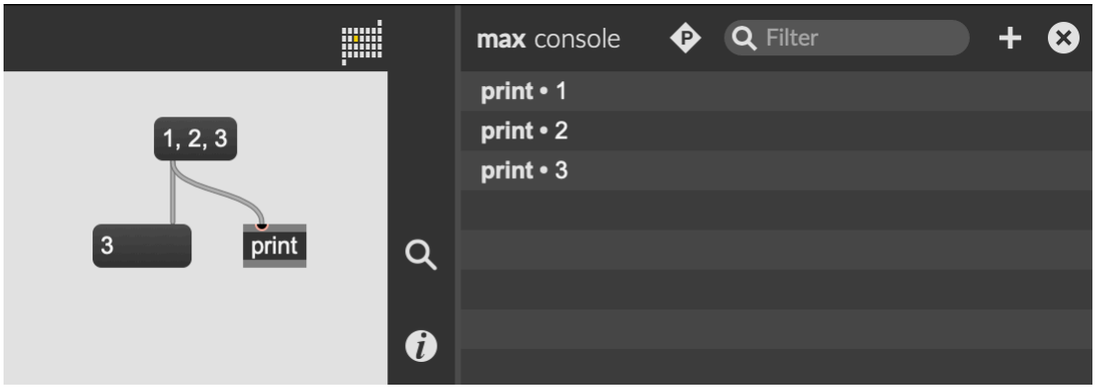
The special $1 characters are replaced by the first element of the incoming message

> Dollar-sign replacement will only work if the characters like `$1` appear at the beginning of a word. So sending a message like `34` to `$1-bis` would result in `34-bis`, but sending the same message to `bis-$1` won't cause any replacement to happen.

Commas

Commas in a message box will cause a single message box to send multiple messages. If you click on a message box containing the text `1, 2, 3`, that message box will send the message `1`,

followed by the message `2`, followed by the message `3`. If you're new to Max, probably the first place you'll see this is in conjunction with the line~ object, where a single-element message sets the value of line~ immediately, and the following list message gives line~ an envelope to follow.



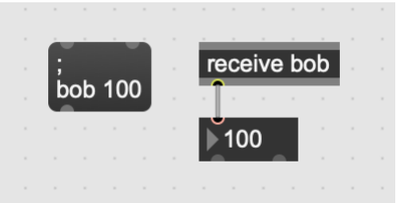
Commas in a message box cause a single message box to send multiple messages.

Semicolon prefix

Finally, the semicolon character `;` has a special meaning whenever it appears at the beginning of a message. When you trigger a message box beginning with a semicolon, instead of sending that message to its first outlet, the message box will send a message to the named destination in the global namespace.

For example, a message like `; max clearmaxwindow` will send the message `clearmaxwindow` to the global object named `max`. In this case, this message will tell Max to clear the Max console. Messages like this are commonly used to script and control the Max application itself. This is useful for controlling Max with messages, for controlling Jitter with messages, and for controlling DSP with messages.

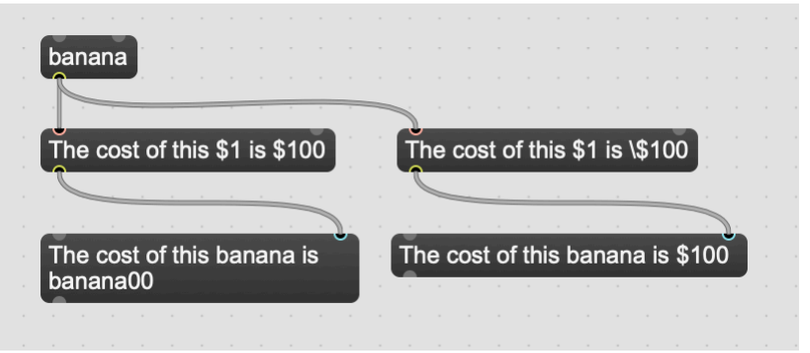
Because `send` and `receive` objects are also in the global namespace, you can send messages to these objects as well using messages with the semicolon character. This is not a common pattern, but you're welcome to do it anyway.



You can send messages to a receive object using a semicolon message, even though it's not very common

Escaping characters

You can use the backslash character to escape a single character anywhere in a message box. For example, you can use a backslash before the `$` character if you want to make sure that the `$` is not interpreted as a placeholder for dollar-sign replacement.



Use the backslash to escape a $ if you want to avoid dollar-sign replacement

To escape multiple characters, or a whole sequence of characters, you can enclose everything you want to escape in double quotes. Note that single quotes will not work, so the message box containing `'hi there'` contains two symbols: `'hi` and `there'`. Double quotes are often useful if you want a block of text containing spaces to be interpreted as a single symbol.



Use double quotes to escape multiple characters, especially multiple spaces.

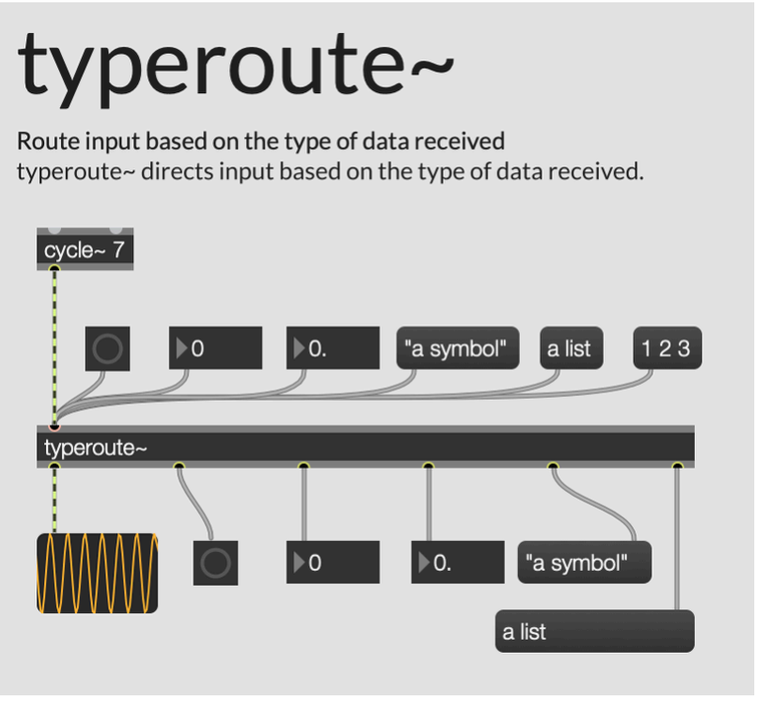
Message Types

Atoms 548
Lists 550
Named Storage Types 551
Matrices 551
Textures 552
Dictionaries 553
Strings 553
Arrays 553

All Max messages are built up of atoms, which can be a **bang**, a **float**, an **int**, or a **symbol**. These are simple types which hold a small, fixed amount of data. An ordered group of atoms is called a **list**. These are the fundamental Max data types. For larger, more complex data types, Max messages use atoms to refer to the place where that more complex data is stored.

Atoms

There are no more fundamental types in Max than *bang*, *float*, *int*, and *symbol*. An ordered group of these types is called a *list*, which also acts a lot like a fundamental Max data type. In a **message box**, Max treats each space-separated element as an atom. You can use the typeroute~ object to separate messagse traveling on a single patch cord by their data type.



Name	Description
bang	*bang* means "do it"—most objects respond to a bang by performing their primary function.
int	A whole number without a decimal component
float	A number with a decimal component
symbol	Any combination of characters, including numeric characters

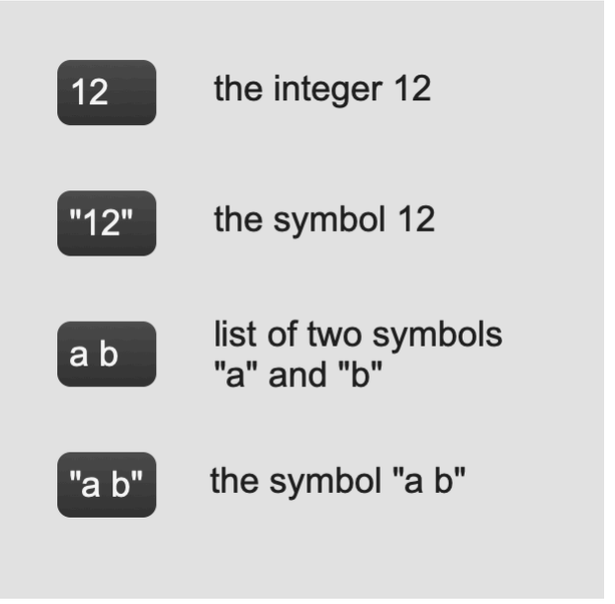
Floats and Ints

In Max, the difference between an int and a float can be significant. Some objects will have different behavior depending on whether they receive a float or an int, and some might refuse to accept one type of number when they expect the other. For a more in-depth discussion, see Integers and Floats.

Symbols

Symbols can contain any combination of characters. Max will automatically treat any non-numeric group of characters as a symbol (the one exception is the characters `bang`, which Max will

recognize as a *bang*). If you want to treat a group of numbers as a symbol, or you want your symbol to contain spaces, use quotation marks.



> A symbol is a fixed-length, immutable entity. Adding or removing characters from a symbol will create a new symbol and add that to the **Symbol Table**. Often this is a technical detail that you can safely ignore. If you want to know more, you can read about symbols and strings.

Lists

A list is just an ordered group of atoms. Often, if the first element of a list is a symbol, the object receiving the list will interpret the leading symbol as a *selector*, and the following list elements as *arguments*.

Building, manipulating, and routing lists is fundamental to working in Max. You can route a list to one part of your patcher or another, based on the first element of the list, using the route object. It's also very common to build lists using dollar-sign replacement in a message box. The `list.*` family of objects, like list.append, list.iter, and list.rot, provide even more ways to work with lists.

> Technically, a list must always start with either a float or an integer. If a list starts with a symbol, then the symbol is the selector, and what follows are the arguments for that selector. Usually you can ignore this technicality, but the distinction can be important to remember when working with Max's C or JavaScript APIs. There are also some sneaky situations that expose what's really going on "under the hood". For example, any object that accepts a message like ::message[list 1 2 3] will respond the same way to the message ::message[1 2 3]. So, the "length" of the message ::message[list 1 2 3] is actually 3.



The list selector instructs the object to treat the arguments that follow as a list.

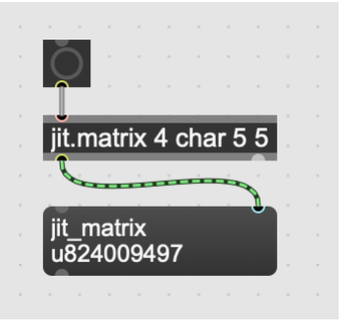
Named Storage Types

Atoms and lists are primitive in the sense that the name of the data is the same as the data. The number 12 and the atom `12` are the same. However, for larger and more complex data, it's not feasible to put the entire block of data into a message box. When one Max object wants to tell another Max object to process an image that's stored in a **matrix**, it doesn't send a message containing the data, but rather the name of the matrix that stores the data.

Matrices

Matrices store multidimensional data, where every *cell* has the same data type. Matrices are often used to store images, 3D models, and 3D transformations. The object that manages a reference to a matrix is called jit.matrix.

When you view a matrix in a message box, you'll see that matrices are identified by a list with two parts: the symbol `jit_matrix`, followed by the unique name of the matrix. For more info on matrices, see matrix.

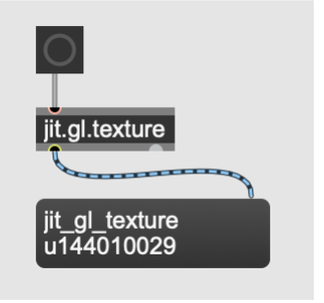


> Patcher cords that carry matrices also get a special, striped-green style. This is just cosmetic—as you can see it's simply carrying a normal message.

Textures

Textures are similar to matrices, in that they store multidimensional data of all the same type. However, the big difference between Max matrices and textures is that textures reside on the Graphics Processing Unit, or GPU. Max itself manages the data and the life cycle of matrices, but it asks the graphics API to manage textures on its behalf.

The object that manages a reference to a texture is called jit.gl.texture. When you view a texture in a message box, you'll see that a texture is identified by the symbol `jit_gl_texture` followed by the unique name of the texture. For more info on textures, see textures.



> Like matrices, texture patch cords get their own styling.

Dictionaries

Dictionaries store structured data. That data is organized into *keys* and *values*, and you can use the key to look up the value. A value can be a number, a list, a symbol, a string, an array, or even another dictionary.

Dictionaries are managed by the dict object, and you can work with dictionaries using the `dict.*` family of objects. In a message box, you'll see that dictionaries are identified by the symbol `dict` followed by the name of the dictionary. For more info on dictionaries, see dictionaries.

Strings

Strings store an ordered collection of characters. Unlike a symbol, strings are mutable, which means that a string can be changed without creating a new string.

Strings are managed by the string object and manipulated with the `string.*` family of objects. In a message box, you'll see strings represented as by the symbol `string` followed by the name of the string. For more info on strings, see strings.

Arrays

Arrays are an ordered collection of arbitrary data. Unlike lists, arrays can store complex data types like dictionaries, strings, and other arrays. Max provides the handy array.map and array.reduce for functional-style programming on arrays.

Arrays are managed by the array object and manipulated with the `array.*` family of objects. In a message box, you'll see arrays represented as by the symbol `array` followed by the name of the array. For more info on arrays, see arrays.

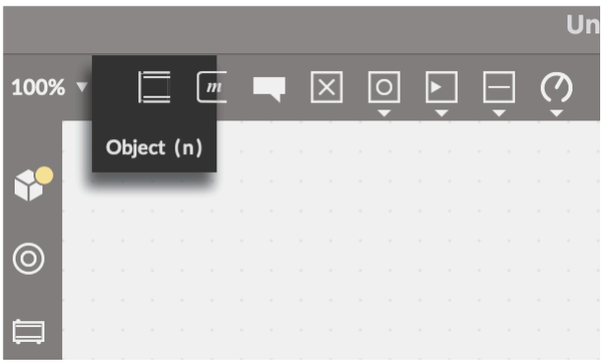
Objects

Creating an Object 555
Help Files and Reference 559
Arguments 559
Attributes 561
Inlets and Outlets 562
Action Menu 565
Annotations and Hints 566

In Max, you define the behavior of a patcher by connecting together objects. Each object has a unique character, defined by its name. If you're familiar with programming, you can think of an object's name as its class name. Objects are connected to each other by patch cords, which define how messages and data flow between objects.

Creating an Object

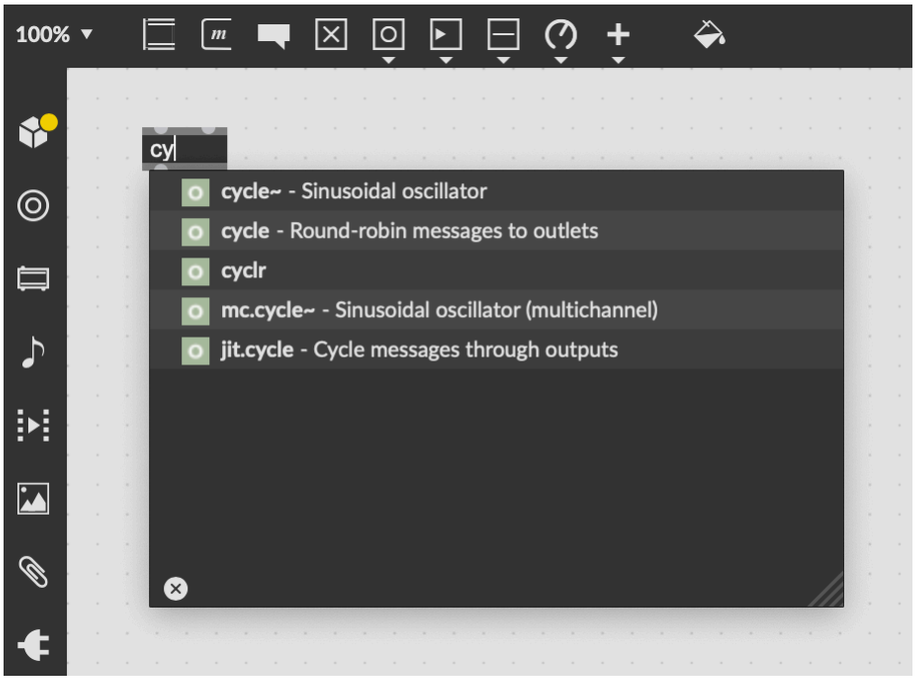
You can create a new object in any unlocked patcher by pressing the n key, or by double clicking, or by dragging in a new object box from the top toolbar.



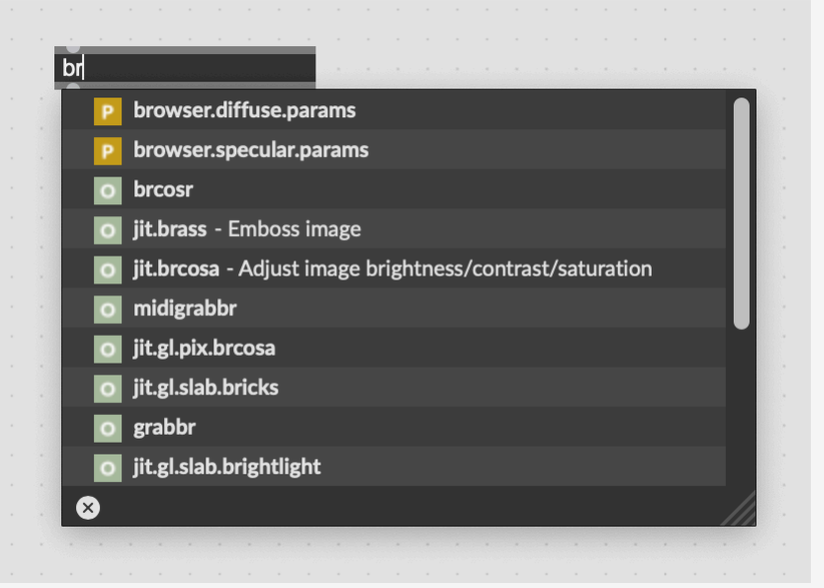
Autocomplete

Once you've created a new, empty object, simply start typing into the new object box. Autocomplete should appear, showing you the names of Max objects that are a close match for

your text. Once you've found the name of the object you want to create, press `enter` or click outside of the object box to finalize your text.



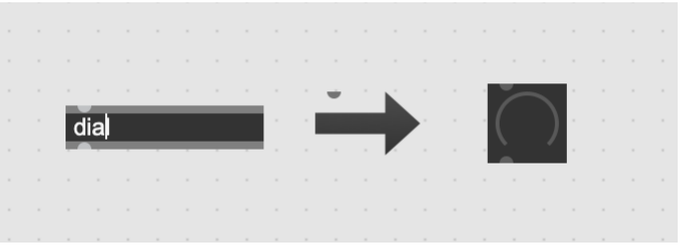
The small icons to the left of the object name in the autocomplete box tell you what kind of object matches the current object text. Most of the time, you'll see an `o`, indicating an object. Object boxes can also reference abstractions, and if you start to type the name of an abstraction, you'll see a `p` icon instead of the usual `o`.



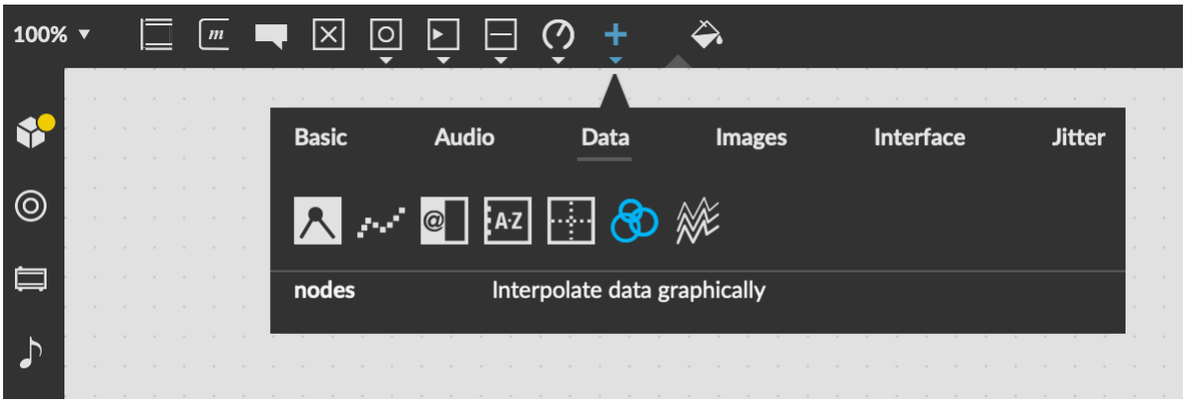
Creating UI objects

Some objects, like sliders, dials, and level meters, are user interface or **UI Objects**. These objects don't look like most objects, instead they have a custom way of being drawn that's unique to each one. They may also define some kind of interaction behavior, where clicking and dragging on the object may have some effect. For example clicking and dragging on a dial will change its value and cause it to send that value as a message.

You can create UI objects in the same way as other objects. Create a new object box and type the name of the UI object into the box. As soon as you finalize the object's text, Max will replace the generic object box with the custom UI display.



You can also use the top toolbar to create UI objects. In fact, the top toolbar contains a large palette of available UI objects. By clicking in the top toolbar, you can find simple interaction objects like sliders and dials, audio UI objects like gain controls and level meters, and widgets like a step sequencer interface. Click on any of these to create a new object in your patcher, or drag and drop the object to position it exactly where you want.



Resizing objects

When you select an object, resize handles will appear at the object's corners. Click and drag on these to resize the object. For UI objects, you can hold down the `shift` key to lock the object's aspect ratio as you resize it. For regular text objects, holding down `shift` will let you change the font size by dragging. This is especially useful for comment objects. Finally, with an object selected, you can select the "Fix Width" option (`command-j` on MacOS, `control-j` on Windows) to shrink the object to just fit its text contents.

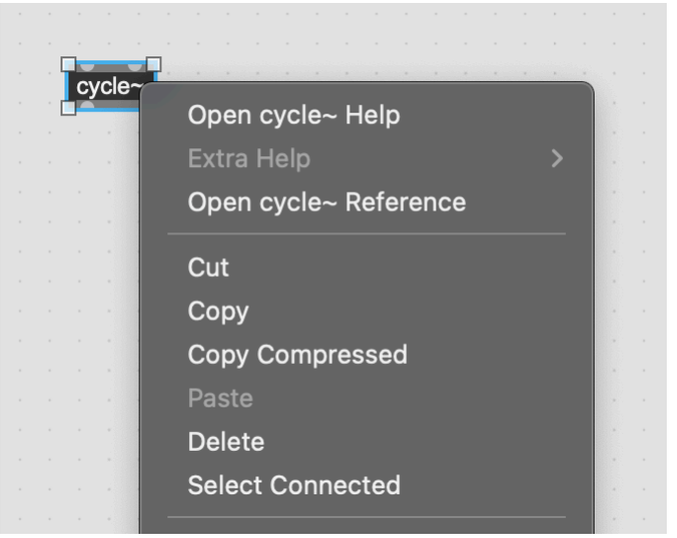
Editing object text

You can edit the text of an existing object by selecting the object and pressing `return`, or by double-clicking on the object (the patcher must be unlocked). Changing the text of an object will usually replace the original object with a new instance, resetting the internal state of the object to its original values.

> If you need to use special characters in a object's text box, for example a comma or a quotation mark, you can use the escape rules for special characters. See *messages* for more information.

Help Files and Reference

Every object has an associated help file, which describes the object's use and demonstrates its behavior. You can open the help file for an object by holding option and clicking on it, or by right-clicking on the object and selecting *Open Help* from the contextual menu.

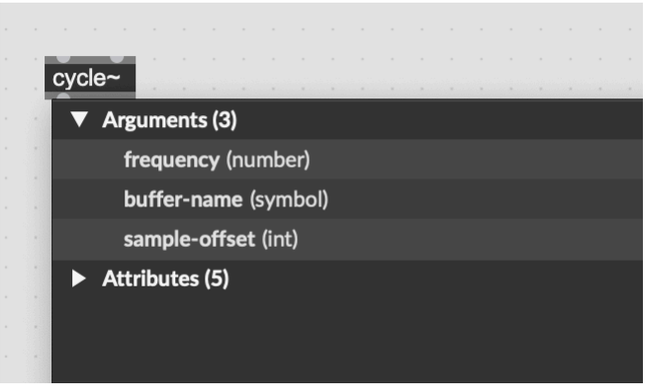


Every object also has an Object Reference Page, which completely describes the object's behavior.

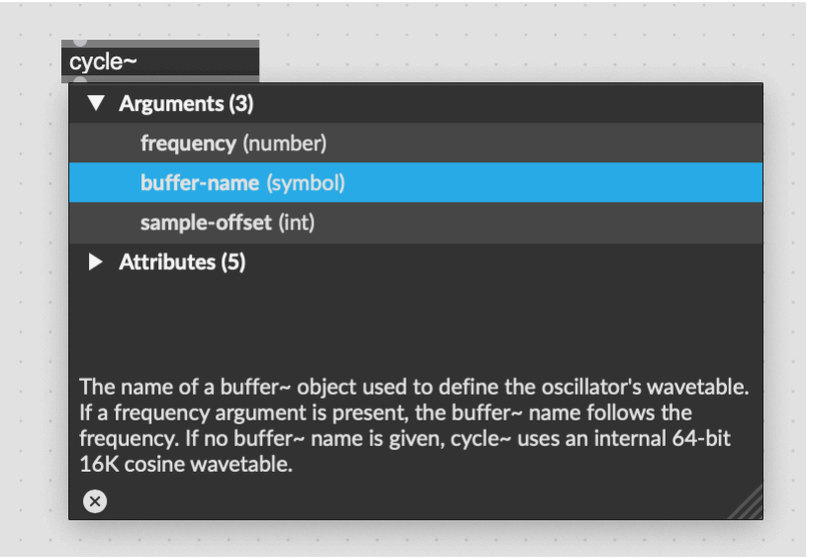
- A short and long description of the object's functionality
- The Arguments and attributes that can configure the object
- The symbols that the object understands
- Other related object and documentation

Arguments

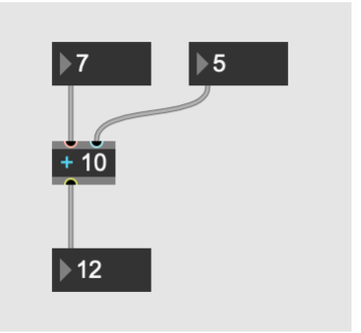
Some objects take arguments that initialize their state and further define their behavior. For example, the object cycle~ can take three arguments, the first of which defines its initial frequency. Once you've typed the name of an object into an object box, hit space to start typing the object's arguments. The autocomplete should update to show the possible arguments for the object.



You can also click on the name of a particular argument to get more information about what that argument actually does. Clicking on the `buffer-name` argument for the `cycle~` object, for example, shows some explanatory text about that argument.



The arguments to an object are used to initialize it, and do not reflect the current state of an object. This is a quirk of working with Max that can be confusing to new users. As a classic example, sending a message to the right inlet of a + object will change its behavior, but not its appearance.



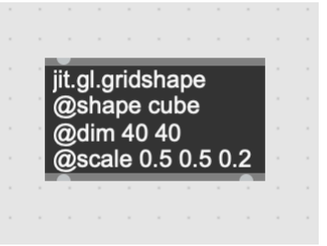
The + object has 10 as an argument, but its internal state has been changed by a message to its right inlet.

A more subtle point is that arguments might not map easily to something internal to the object that can be changed after the object is created. For example, the arguments to a gate object change the number of inlets, and there's no way to update this value after creating the object. This makes arguments different from attributes, most of which can be modified in the inspector.

> Arguments are almost always optional, but some objects do require arguments to initialize correctly.

Attributes

Attributes are similar to arguments, except unlike object arguments, attributes can appear in any order, but must be identified by their name. In the text of an object box, attributes always come after arguments. Attribute names are prefixed with the `@` symbol, so an object box with the text `cycle~ @frequency 440` will create a cycle~ object with the frequency `440`. Attributes are especially useful for complex objects with lots of configuration options. A classic example is jit.grab, which gets video from a camera device. This object has lots of configuration options, including the size and format of the video stream. Attributes make it possible to define the state of a complex object using only the object's text.



An object called `jit.gl.gridshape`, with some attributes defining its state.

Inspecting attributes

Most attributes come with a default *setter* method, meaning the attribute can be set by sending the object a message with the name of the attribute and its new value. For example, you can set the `@bgcolor` attribute on a toggle object by sending it a message starting with `bgcolor`.



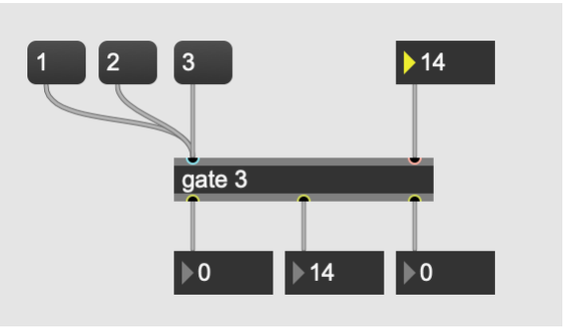
Object attributes can also be viewed and modified in the patcher using the attrui and getattr objects. Connect an attrui to an object to get a dynamic dropdown with all of that objects attributes. Use getattr to listen to the state of attributes as the change.

Finally, select an object and open the **Inspector** to see and filter all of an object's attributes at once. This is also the place for **freezing attributes**, especially handy for saving the state of changed attributes.

Inlets and Outlets

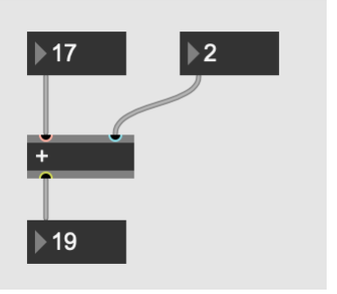
Almost all objects have at least one inlet or outlet. Objects connect to each other via Patcher Cords, where a patch cord always connects the outlet of one object to the inlet of another. Some objects have many inlets or outlets, and sometimes the way an object is configured can change the number of inlets or outlets that it has.

The way that an object responds to a message will usually depend on the inlet that receives the message. For example, the first inlet to a gate object lets you set the active outlet, and the second inlet receives messages to be routed to the active outlet.

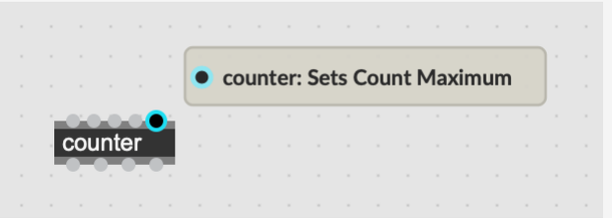


You can imagine that the messages to the first inlet choose between the first, second, and third outlets. It looks like messages are being routed to the second outlet, so the first inlet seems to have received the message `2`

Inlets actually come in two kinds: hot and cold. **Hot Inlets** will trigger output whenever they receive a message input, while **Cold Inlets** do not trigger output, and instead change something about the object's state. The gate object is a good example—you can see that the left inlet has a blue hue, while the right inlet has a pinkish tint. Math objects are another classic example, where the cold right inlet changes how the object acts but does not trigger computation, while the left inlet makes the computation happen.



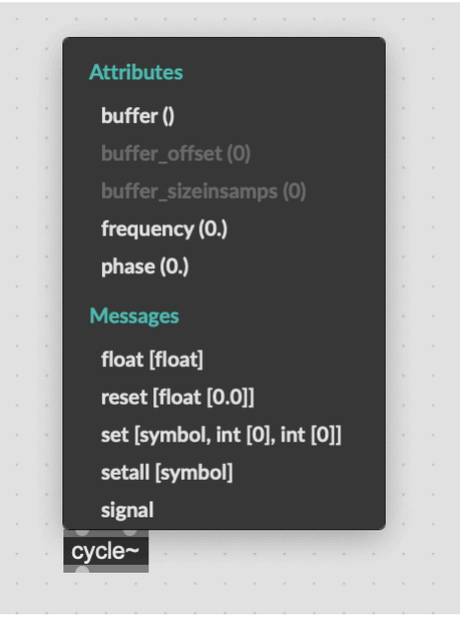
The right inlet is a cold inlet, since it just changes the amount of addition but causes no output. The left inlet is a hot inlet, since it will trigger the actual computation.

> With the patcher unlocked, you can hover over any inlet or outlet to see a quick explanation of its behavior.
>
> 
>

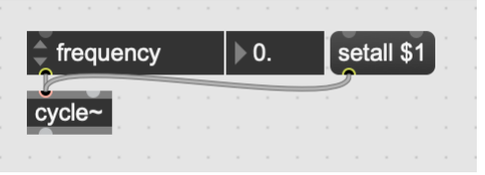
If you're making a subpatcher or an abstraction, you can set the `@comment` attribute to add your own description that will display when you hover over the inlet or outlet. See subpatcher inlets and outlets for more information.

Viewing messages and attributes (Quickref)

Right-click on an object inlet (or hold `control` and click) to see the **Quickref** for that object. This is a short list of all of the attributes that the object has, and all of the messages that it will respond to. This can be an extremely useful way to remind yourself how to work with an object, without needing to open its help file.



Click on any of these to create a new object in the patcher for controlling that attribute or message. Clicking on a message will create a message box that can send a formatted message to the object. Clicking on an attribute will create an attrui object that will let you change that attribute.



Action Menu

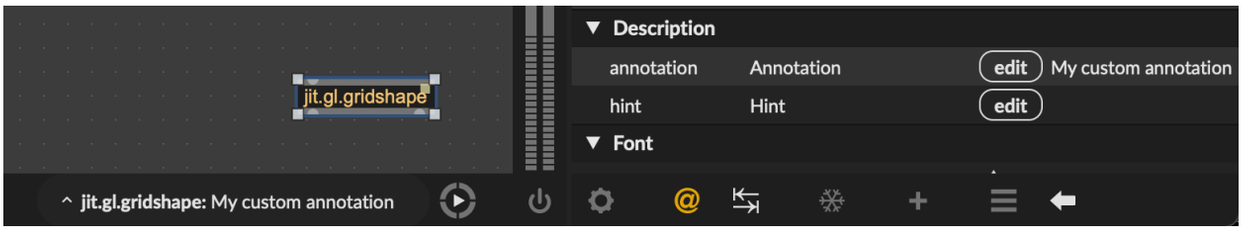
Hover over the left edge of an object to show the Action Menu button. Click on this button to show the **Action Menu**, which lets you transform and manipulate the object in several useful ways. For more, see the action menu documentation page.



The action menu button

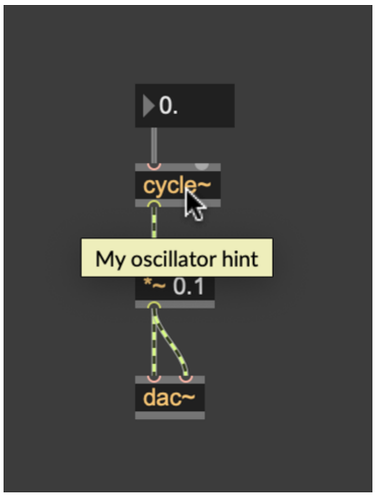
Annotations and Hints

When you hover over a Max object, the *Clue Tab* will display information about that object. You can customize the text that Max displays here by setting the `@annotation` attribute for a given object.



Setting the @annotation attribute lets you customize the text that displays when you hover over an object.

You can also set the `@hint` attribute on an object, which will add an "alt-text" style text pop-up that will display when you hover over the object. This text will only appear if the patcher is locked.



Set the @hint attribute on an object for another way to show descriptive text.

Patcher Lifecycle

Opening a Patcher 567
Closing a Patcher 569

Opening a Patcher

In general, it's best practice not to make any assumptions about the order in which things will happen when Max opens up a saved patcher. If there's a way to use an object like trigger to make the order of messages explicit, it's usually a good idea to do so. However, when Max loads a new patcher it does initialize the patcher in specific phases.



Max patcher stages of initialization

1. Object Initialization
2. Patchcord Connection
3. Parameter Initialization
4. *pattr* Restoration
5. *loadbang* and *loadmess*
6. *live.thisdevice*
7. Window Activity
8. Patcher Arguments
9. *dspstate~* and Signal Processing

Subpatchers

Within a given phase, Max subpatchers are initialized before their parents. If a Max patcher contains a loadbang object, and its subpatcher also contains a loadbang object, the loadbang in the subpatcher will always output a `bang` message before the parent. However, a buffer~ object in a parent patcher will initialize before any loadbang objects output a `bang`, even if the loadbang objects are in a child patcher, because *Object Initialization* comes before *loadbang and loadmess*.

Object initialization

Max creates all objects in the current patcher. For some objects, this may have synchronous behavior, where the behavior might otherwise be asynchronous. For example, when a buffer~ object is initialized, if a *file* argument is provided, that file is loaded synchronously. After the patcher has loaded, sending a `replace` message to a buffer~ object triggers an asynchronous operation.

Patchcord connection

After all of the objects in the patcher have been initialized, Max will rebuild all of the patchcord connections between objects. Importantly, since this happens after object initialization, Max messages sent between objects during the object initialization step might not be routed between objects as expected.

Parameter initialization

Next, Max checks all objects for which Parameter Mode is enabled. For any such objects, if they have an initial value set, those objects will set their internal parameter value to that initial value, and then output their value.

> Select *Reinitialize* from the *Edit* menu to set all parameters in the current patcher to their initial value.

pattr Restoration

If there are any pattr objects with `@autorestore` enabled, Max will restore the last set value of each pattr object, and then each pattr object will output a value.

loadbang and *loadmess*

Now that all objects have been initialized, loadmess and loadbang objects will generate their output.

live.thisdevice

The live.thisdevice object functions similarly to loadbang in the context of a Max for Live device. In a regular Max patcher, live.thisdevice is functionally the same as a loadbang except live.thisdevice objects will trigger after loadbang objects.

Window activity

After all objects have been initialized, and any loadbang or loadmess objects have sent their output, Max will create the window and bring it into focus. Any active objects will now send an output.

Patcher Arguments

Patcher arguments (arguments and attributes on a patcher object) are parsed by patcherargs objects at the same time as loadbang and loadmess objects send their respective messages. However, the initial output of a patcherargs object is deferred, and will be sent after the window is ready.

dspstate~ and signal processing

Finally, Max is ready to construct the signal processing graph and start audio processing for the patcher. This is the stage when all dspstate~ objects will send their outputs, reporting the current sample rate, DSP on/off status, etc.

Closing a Patcher

Similar to the way Max builds up a patcher in stages when opening a new patcher, closing a patcher also breaks the patcher down in a standard order. Similar to opening a patcher, subpatches close before parent patches So, a freebang in a subpatcher will send its bang before a freebang in a parent patcher.

1. *closebang*

2. Freeing Objects

closebang

First, the closing the Max patcher window causes any closebang objects to send a `bang` message. One interesting thing is that if you close a parent patcher window, with an open subpatcher containing a closebang object, that object will *not* send a `bang` message at this time. However, if you close the subpatcher window manually first, closebang will send a `bang`.

Freeing objects

Max goes through all of the objects in the patcher and frees them. This frees any memory or other resources that each object might have been holding on to. It also causes any freebang objects to send a `bang` message.

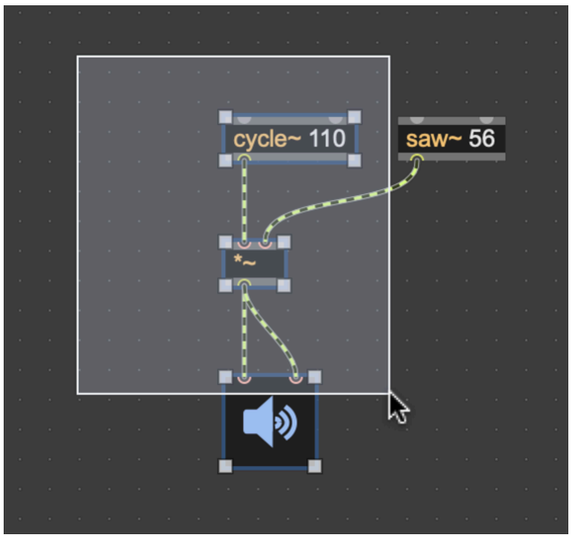
Patching

Locking/Unlocking 571
Creating Objects 574
Positioning Objects 574
Copying Objects 576
Resizing Objects 576
Aligning Objects 578
Distributing Objects 579
Grouping Objects 580
Presentation Mode 580
Object Ordering 582
Foreground and Background 582
The Grid 583
Routing Patch Cords 585
Changing Colors 586
Zooming 586
Multiple Views 587
Patcher Attributes 587
Key Commands 588
Patching Mechanics 589
Finding Objects and Text 589
Patching Margin 590
Reinitialize 592

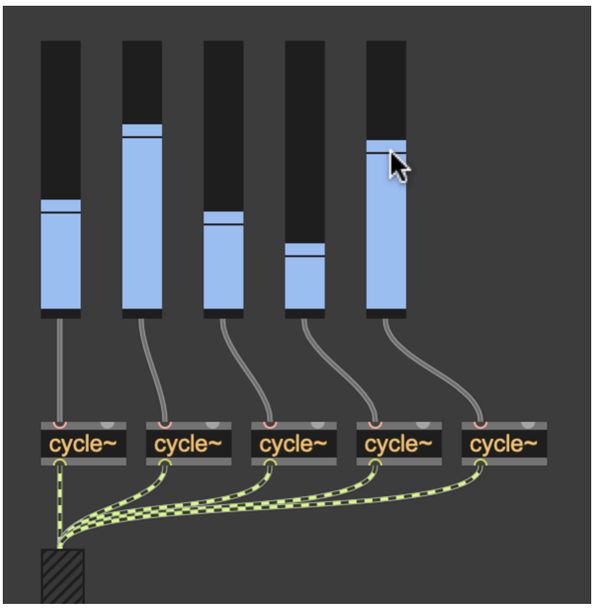
The word **patching** describes everything that you do as part of creating a Max patcher, including adding, positioning, configuring, and connecting objects.

Locking/Unlocking

A new Max patcher is **unlocked** by default. Unlocking a patcher is also called putting the patcher in **edit mode**.. When unlocked, you can use the mouse and keyboard to create, delete, move, and connect objects. You can click and drag in the background of a patcher to start a selection, and drag to select multiple objects.



When the patcher is locked, you can no longer modify objects using the mouse and keyboard (you can still edit the patcher via scripting). Instead, you can operate user interface (UI) objects (sliders, dials, buttons, etc.) by clicking and dragging on them. You could say that you unlock a patcher to work on it, and then lock the patcher to perform with it.



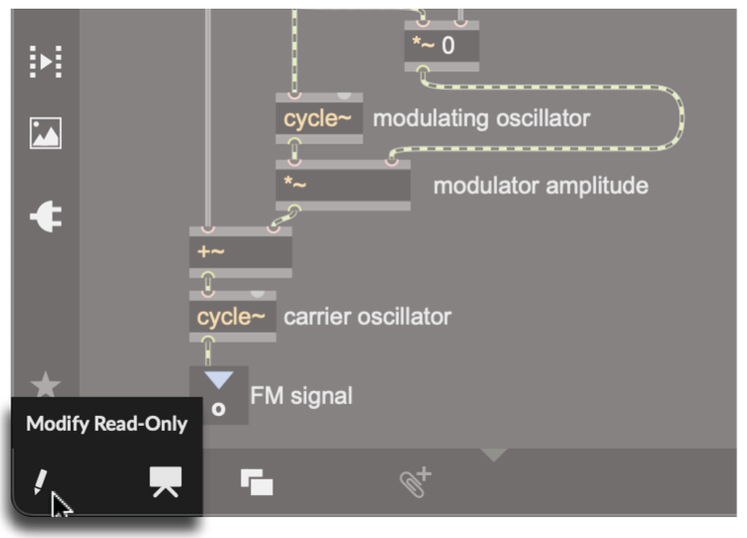
Lock the patcher to control UI objects like sliders

> While the patcher is unlocked, you can still operate UI objects as if the patcher were locked by holding the ⌘ (macOS) key or the CTRL (Windows) key. This is convenient if you want to adjust an object quickly without locking.

You can lock the patcher by clicking the *Lock* icon in the bottom toolbar, or by selecting *Edit* from the *View* menu.

Modify read-only

If you open an *abstraction*, or if you're viewing the contents of a *bpatcher*, poly~, pfft~, or any other object that loads a separate patcher, the *Edit* icon will be disabled and replaced by the *Modify read-only* icon. A read-only patcher cannot be unlocked by pressing ⌘e (macOS) key or the CTRLe (Windows) key—you must click on the *Modify read-only* icon to unlock a read-only patcher. This is to help you avoid accidentally modifying a patcher that other patchers might depend on.



Patchers that load in read-only mode must be unlocked by clicking the Modify read-only icon.

Operate While Unlocked

With *Operate While Unlocked* enabled, you can adjust user interface objects using the mouse even while the patcher is unlocked (in *Edit* mode). You can enable this mode by clicking the *Operate While Unlocked* icon in the bottom toolbar, or by selecting *Operate While Unlocked* from the *View* menu.

In this mode, hold down `shift` while clicking to select UI objects without changing their value. You can also select a UI object by clicking its border. Hold down `shift` and start dragging a UI object to move it, then release `shift` once you've started dragging.

> To perform actions with UI objects that require the `option` key, hold down the `option` and ⌘ (macOS) or CTRL (Windows) keys.

Creating Objects

With the patcher unlocked, create a new object by pressing the `n` key, or by double clicking. You can also drag in an object from the top toolbar. Once a new object is created, keyboard focus moves to the text field in the new object.

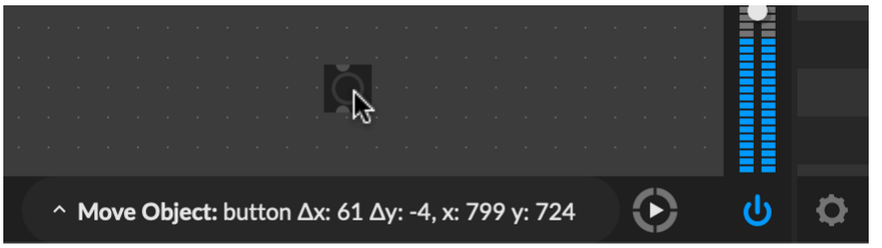


A new object, with keyboard focus.

For more on creating objects, understanding objects, and learning about objects, see the dedicated Objects page in the User Guide.

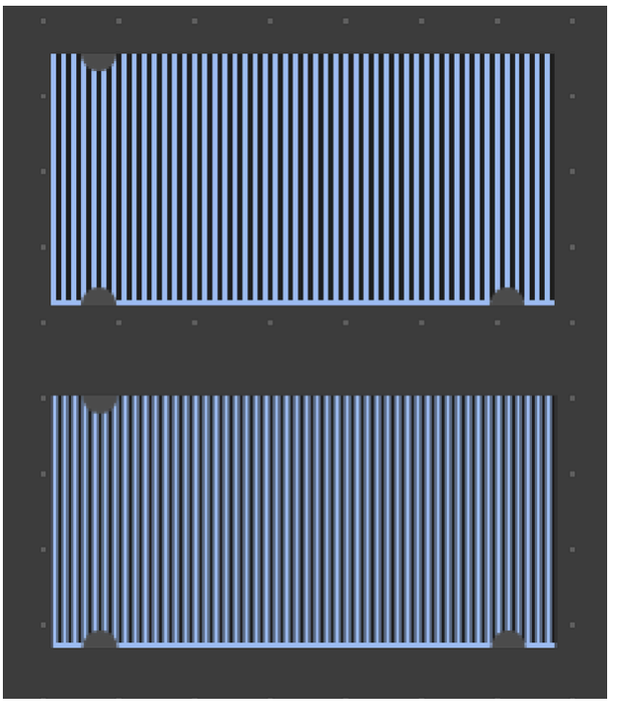
Positioning Objects

With the patcher unlocked, click and drag on an object to move it. If multiple objects are selected, the objects will move as a unit. You can also use the arrow keys to make fine adjustments, or hold `shift` and press the arrow keys to make large adjustments. As you move objects around your patcher, the the *Clue Bar* will show you how your action will affect the object's position.



Pixel alignment

If an object has a position with fractional values, it will no longer align precisely with the pixels used for display on your screen. This can cause subtle aliasing issues that can affect how objects look. One way this can happen is if you try to reposition an object while the patcher view is zoomed.



These two multislider objects are displaying the same data, but because the object on the bottom is not pixel-aligned, the slider bars look blurry.

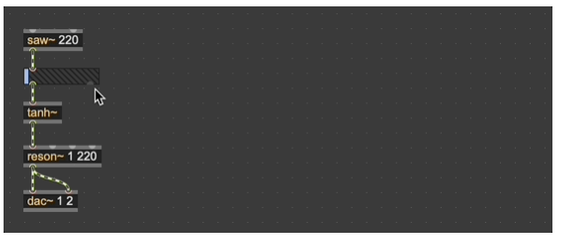
There is a patcher attribute called `@integercoordinates` or *Snap to Pixel* that enforces whole number coordinates for all objects in the patcher. You can also select any number of objects that you'd like to align to the pixel grid and select *Apply Grid > Apply 1x1 Pixel Grid* from the *Arrange* menu to move all selected objects to integer coordinates.

Copying Objects

Select a group of objects and select *Copy* from the *Edit* menu to copy them to the clipboard. Select *Paste* to paste those objects into the patch. You can also select *Duplicate* to copy-paste with a single command. Finally, you can hold `option` (macOS) or `alt` (Windows) while dragging a group of selected objects to copy them before dragging.

After you *Paste* or *Duplicate* a group of objects, you can reposition the newly created group of objects. When you do, Max will remember the offset from the original object group. This lets you quickly create a large number of evenly-spaced object groups.

1. Select a group of objects
2. Copy and paste (or duplicate) that group
3. Reposition the pasted group
4. Select *Paste* or *Duplicate* to paste a new group with the same spacing.



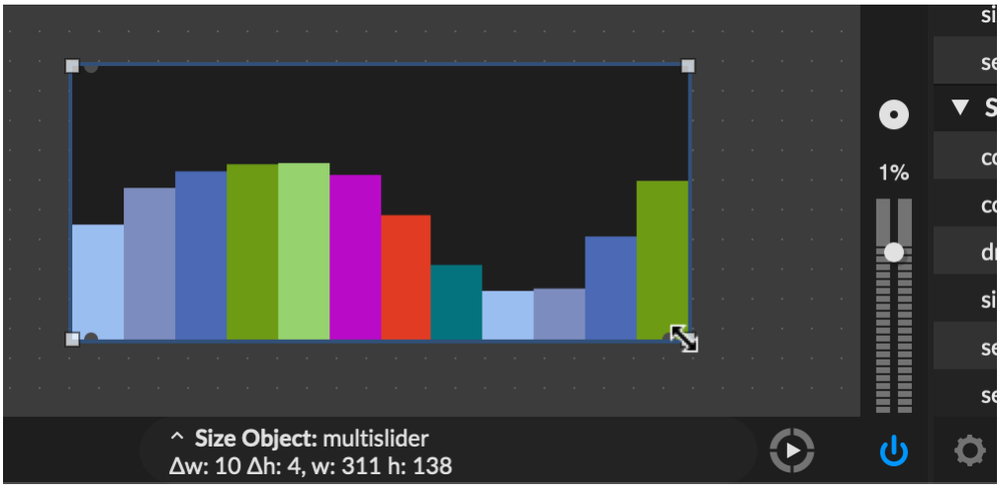
Paste or duplicate multiple times after spacing.

Copy Compressed

The *Copy Compressed* command from the *Edit* menu will copy a group of objects as compressed text. You can share this text anywhere, and restore the objects from compressed text simply by pasting, or by using the *New From Clipboard* command from the *File* menu. See the copy compressed section of the Sharing page for more information.

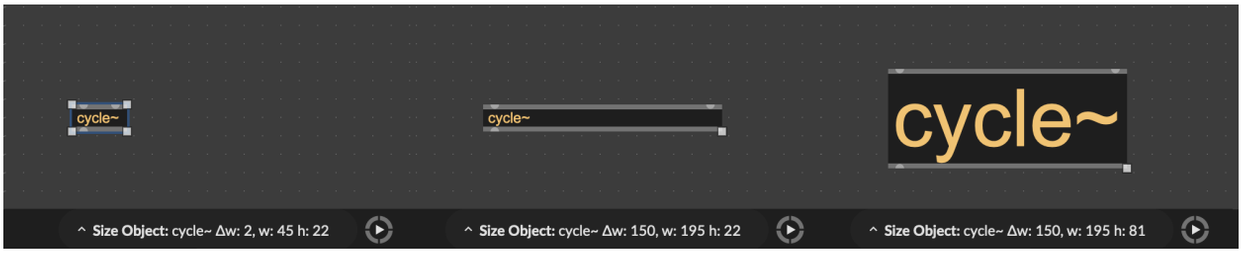
Resizing Objects

When any number of objects are selected, you can resize those objects by clicking and dragging on the white resize handles that appear. Hold shift while dragging to maintain aspect ratio while adjusting size. Note that some objects, like the toggle object, have an intrinsic aspect ratio that cannot be changed.



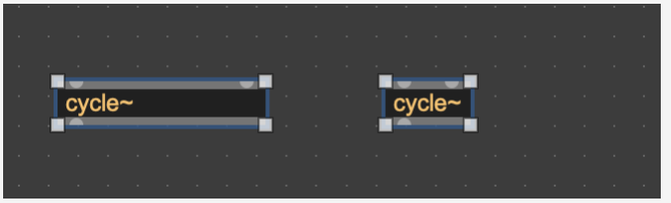
Resize an object using the white resize handles that appear when an object is selected.

Objects that display text, like non-UI objects, comments and message boxes, have a fixed height that will not change while resizing. Instead, changing the width of the object will reflow its text. Hold shift while dragging to change the size of object's font while adjusting its width.



Hold shift while adjusting the size of a text object to change its font size along with its height.

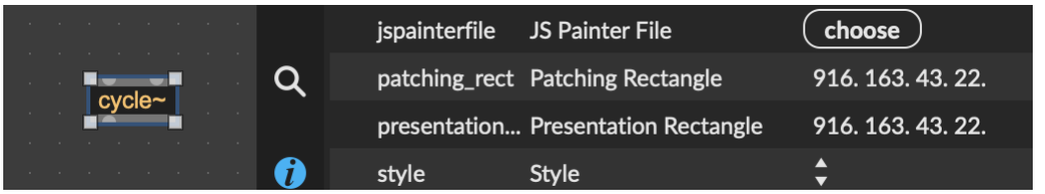
For text objects like comment and message, you can select *Fix Width* from the *Object* menu to adjust the width of the object to fit its text.



Use the Fix Width menu command to adjust the width of text objects.

Patching Rectangle

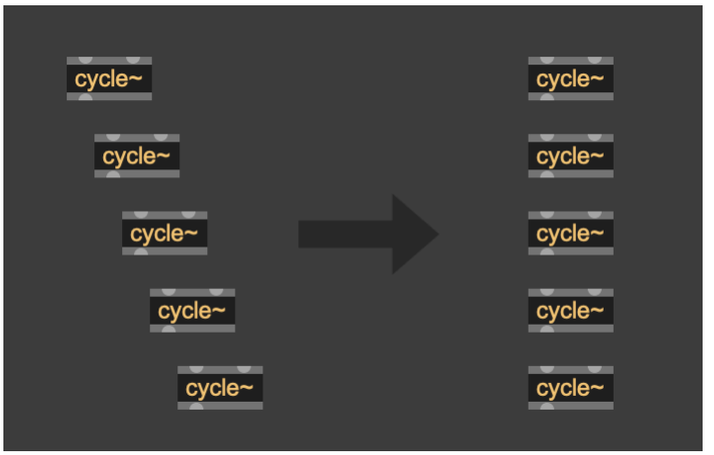
The position of an object in a patcher is called its **Patching Rectangle**. These four numbers determine the distance of the object from the left and top edge of the patcher, as well as its width and height. This value is also exposed as an object attribute called `@patching_rect`, and can be adjusted from the inspector.



An object's position is determined in part by its @patching_rect attribute

Aligning Objects

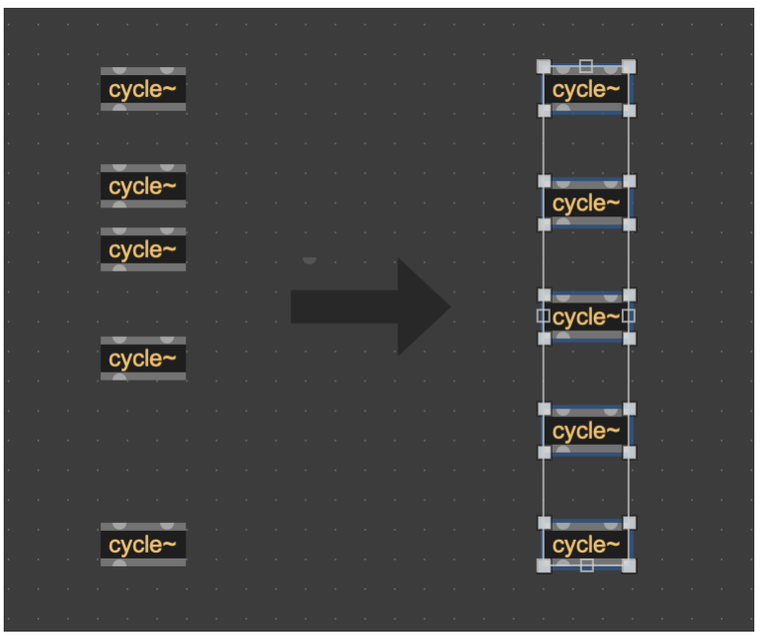
You can align objects by selecting one of the *Align* options from the *Arrange* menu. You can align objects by their left, top, right, or bottom edge, or by their vertical or horizontal center. Most of the time, you can simply press ⌘j (macOS) or CTRL j (Windows) for the *Auto Align* command, which will align objects automatically based on whether they take up more vertical or horizontal space.



Since these objects take up more vertical space, the Auto Align command will align their left edge.

Distributing Objects

You can distribute objects, either vertically or horizontally, equalizing the space between each object in either direction. Select one of he *Distribute* options from the *Arrange* menu, or press `shiftoption⌘h` (macOS) or `shiftoptionCTRLh` (Windows) for horizontal spacing, or press `shiftoption⌘v` (macOS) or `shiftoptionCTRLv` (Windows) for vertical spacing. After distributing, adjustment handles will appear around the objects, which you can use for further equal-space adjustments.



After distributing, you can use handles for further refinement.

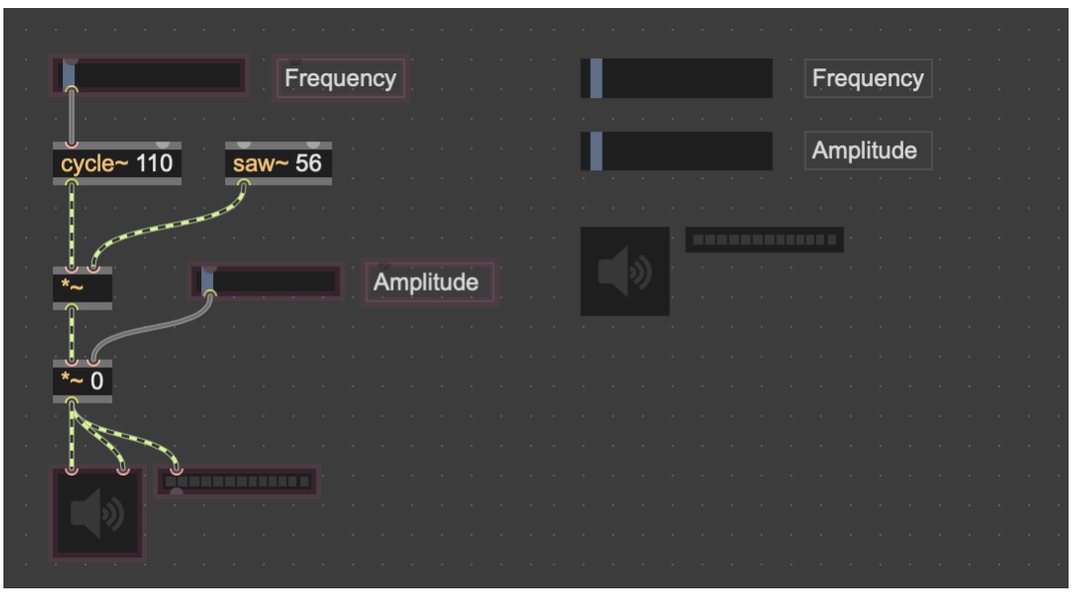
Grouping Objects

If you want to maintain the spacing between objects, even if one of them is moved, you can assign them to a **group** by selecting *Group Objects* from the *Arrange* menu. When one memeber of a group is selected, the whole group will have a thin black border.

When you resize a group using the resize handles, Max will scale the spacing between objects, rather than the objects themselves.

Presentation Mode

Max is designed both as a visual programming environment and as a tool for building user interfaces. It's easy to design an interface simply by sizing and positioning objects as you'd like them to appear. However, when you want to perform with or demonstrate your patch, you might want to hide or reposition certain objects. By using **Presentation Mode**, you can configure a special appearance for your patch, showing just the most important objects.

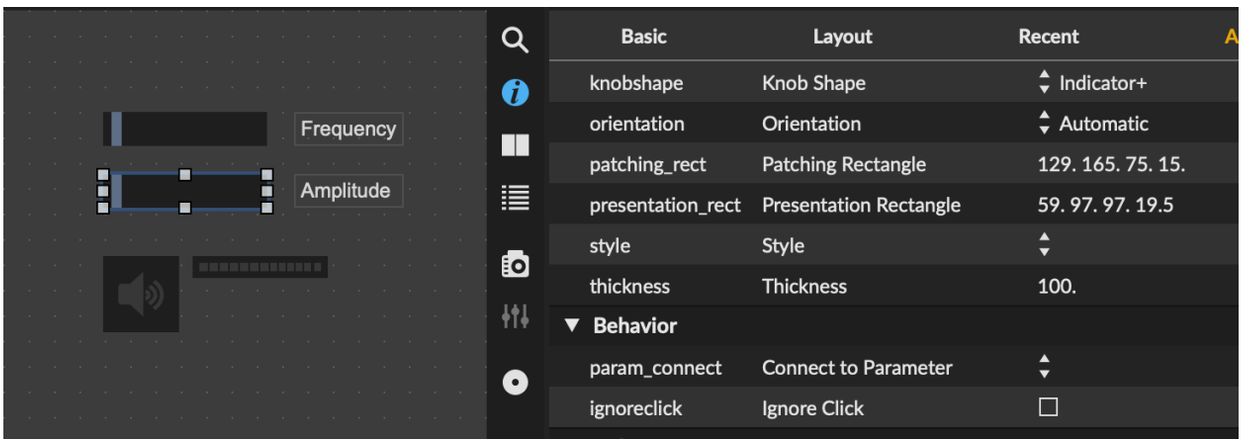


On the left, the patcher as it appears normally. On the right, the same patcher in presentation mode. Only the objects that have been selected for presentation mode appear.

In order to add an object to Presentation Mode, select it and then choose *Add to Presentation* from the *Object* menu. Once added, the object will have a pink border. If you no longer want to see the object in Presentation Mode, select *Remove from Presentation* from the *Object* menu.

> Adding an object to Presentation Mode simply enables the `@presentation` or *Include in Presentation* attribute.

You can toggle between Presentation Mode and Patching Mode by selecting the *Presentation* icon from the bottom toolbar, or by selecting *Presentation* from the *View* menu. An object can have a different size and position in Presentation Mode than in Patching Mode. When you toggle between the two, you will see objects animate to their new positions and sizes. In addition, you'll notice that the `@patching_rect` and `@presentation_rect` object attributes will be different, reflecting the two distinct positions for the object.

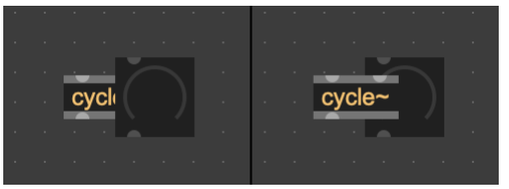


The selected slider has different values for @patching_rect and @presentation_rect, and the @presentation attribute is enabled.

> If you want a patcher to open in Presentation Mode by default, enabled the *Open in Presentation* attribute in the patcher inspector. This can be especially useful for bpatchers that you intend to use as high-level modules.

Object Ordering

Whenever you add a new object to your patcher, Max adds it on top of any existing objects. By selecting *Send Backward* or *Send to Back* from the *Arrange* menu, you can instead have other objects draw on top of the selcted object. The commands *Bring Forward* and *Bring to Front* instead cause an object to be rendered on top of other objects.



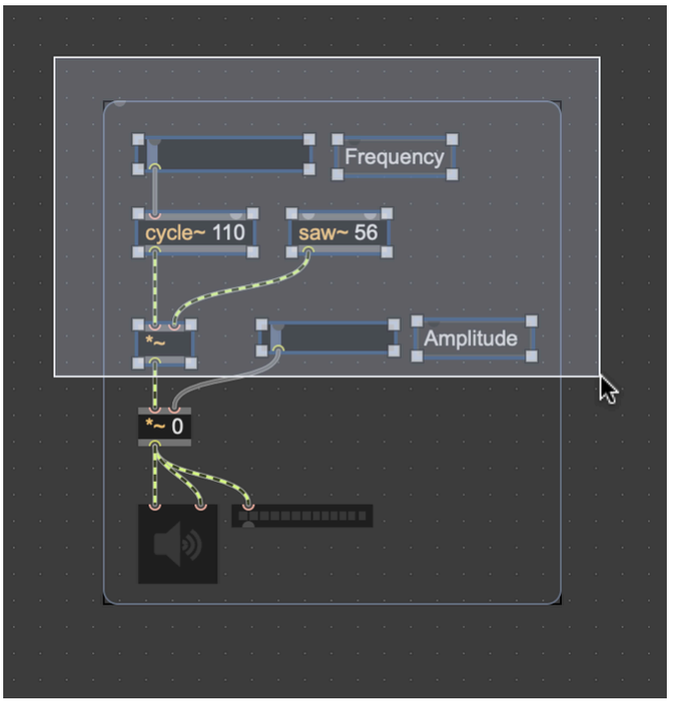
After selecting Send Backward, the newly created dial renders behind the cycle~ object.

> By making parts of some objects transparent, you can create compound UI object by layering existing objects in a clever way. You might find the attribute `@ignoreclick` useful in this case.

Foreground and Background

Max patchers provide a simplified implementation of a layering system, with a foreground and a background layer. The main goal of this system is to make it easier to separate documentation objects like comment and panel from logical objects that affect the behavior of the patcher. Objects in the background will always be rendered behind objects in the foreground, though both may appear in Presentation Mode.

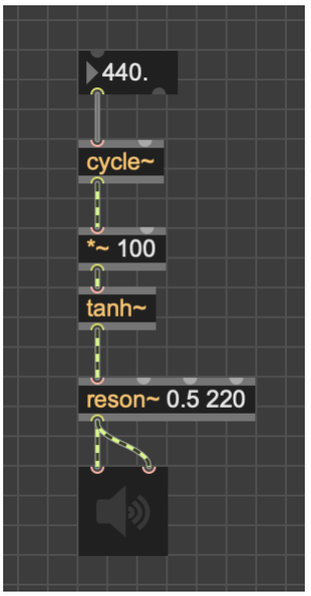
By default, all objects are added to the foreground. You can add an object to the background by selecting *Include in Background* from the *Arrange* menu. You can hide and show the background and foreground by selecting *Hide Background* and *Hide Foreground* from the *View* menu. It can also be very useful to lock the background, which you can do by selecting *Lock Background* from the *View* menu. When the background is locked, objects in the background cannot be selected. This makes it possible to rearrange objects in the foregroudn without background objects getting in the way.



The panel object (drawing the rounded rectangle border) cannot be selected, because it's included in the background and the background is locked.

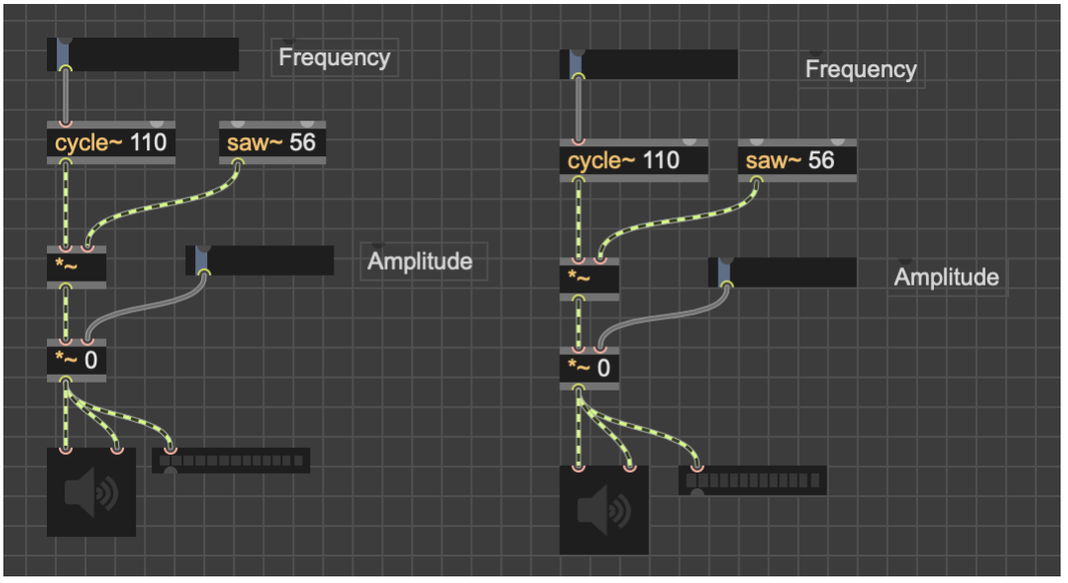
The Grid

To make it easier to lay out objects visually, you can enable the grid by selecting *Grid* from the *View* menu, or by clicking the *Grid* icon in the bottom toolbar. The grid is only visible when the patcher is in *Edit* mode, and will disappear when the patcher is locked.



Enabling the grid may make object layout more consistent and readable.

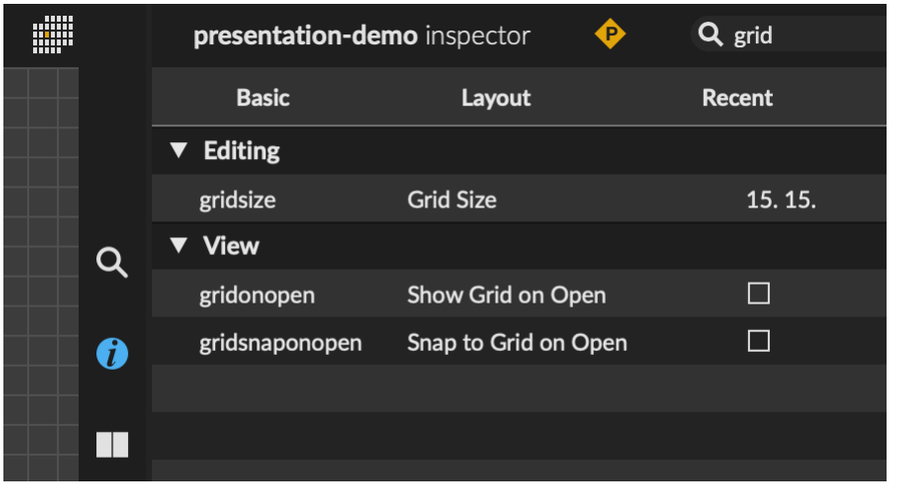
By default, the grid is simply a visual indicator, and objects will not be aligned to the grid while dragging. Choose *Snap to Grid* from the *Arrange* menu, and objects will always be alignd to the grid when you reposition them with the mouse. With this option enabled, objects will also snap to the grid when you resize them. While repositioning or resizing, you can hold down the ⌘ (macOS) or CTRL (Windows) key to temporarily disable snapping.



Object before and after grid-alignment

After enabling *Snap to Grid*, objects that were positioned without this option enabled may not be aligned to the grid, and will not be aligned automatically. To align existing objects to the grid, select *Apply Grid* > *Apply Current Grid to Position* or *Apply Grid* > *Apply Current Grid to Position and Size* from the *Arrange* menu.

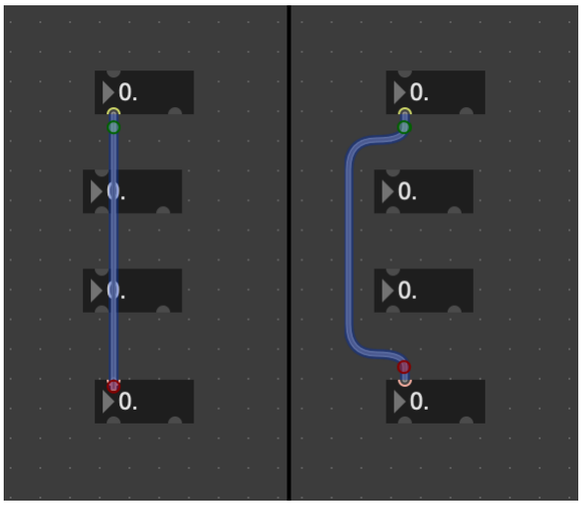
You can change the size of the grid using the *Grid Size* patcher attribute, visible from the patcher inspector.



The Grid Size attribute in the patcher inspector

Routing Patch Cords

To keep your patcher looking clean and organized, you can create segmented patch cords, or you can have Max route the patch cords for you automatically. Routed patch cords are segmented in a way that tries to use only right angles while also avoiding intersecting other objects.



A patch cord after automatic routing

Changing Colors

You can change the appearance of most parts of the patcher, including the color of objects, patch cords, and the patcher background. Color changes can be applied individually, or as a style across the whole patch.

- Change the color and appearance of the patcher itself using the patcher inspector.
- Change the appearance of objects by adjusting each object's individual attributes, by setting a style for each object, or by setting a style for the patcher.
- You can also change the color of patch cords in your patcher.

Zooming

You can adjust the zoom level of your patcher, either zooming in to make more precise adjustments, or zooming out to see more objects at once.

- Find a precise zoom level using the *Zoom* control in the top toolbar.
- Select *Zoom In* from the *View* menu to zoom in, or *Zoom Out* from the *View* menu to zoom out.
- Press `z` to zoom in, or `shiftz` to zoom out.

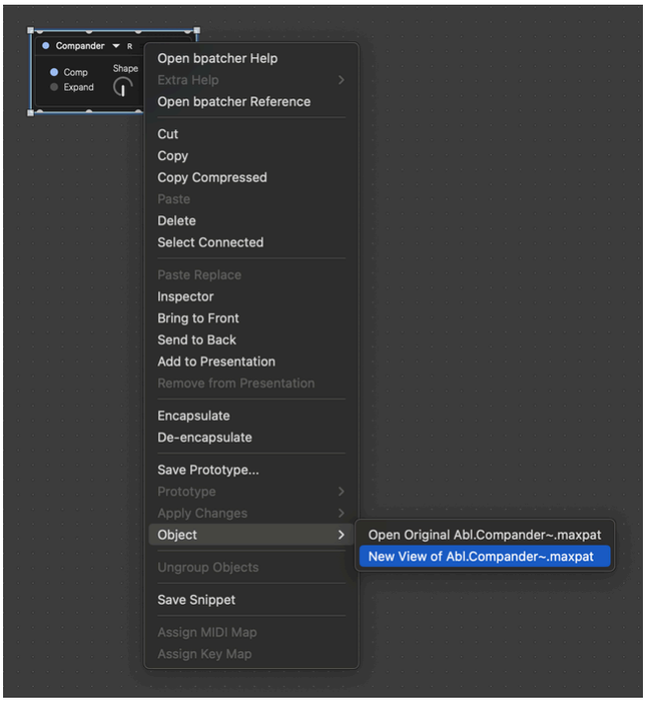
- Use the two finger "pinch" gesture to zoom in or out with precise control.

Multiple Views

It's possible to open multiple views of the same patcher. This could let you view a patcher in patching and presentation mode at the same time, or view a patch at two different zoom levels at once.

Create a new view of the current patcher using the *Patcher Window* icon in the *Bottom Toolbar*.

You can also create a new view of a *bpatcher* by right-clicking the bpatcher and selecting `Object > New View` from the contextual menu.



Patcher Attributes

Behind the scenes, the patcher is a Max object like any other, and many aspects of the patcher can be controlled with attributes.

- Whether the patcher opens in presentation mode
- The color of the patcher background
- The spacing of the grid
- Whether the toolbars are pinned or unpinned

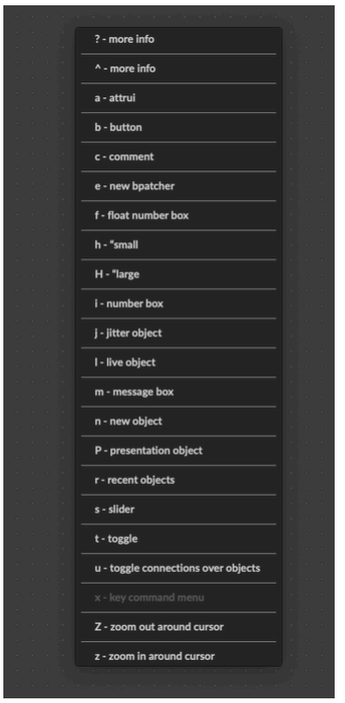
Access these patcher attributes by using the patcher inspector in the Inspector sidebar.

Key Commands

As you patch, you'll find there are several things that you'll do very frequently:

- Creating a new object
- Adding a comment
- Showing a highlight
- Viewing a list of recently created objects
- Adding a button, slider, or toggle

All of these common actions have a **Key Command** associated with them. You can press a single keyboard key (`n` to make a new object, `r` to view recent objects, etc.) to perform the associated action. Most importantly, you can press `x` at any time to view a list of all available key commands.



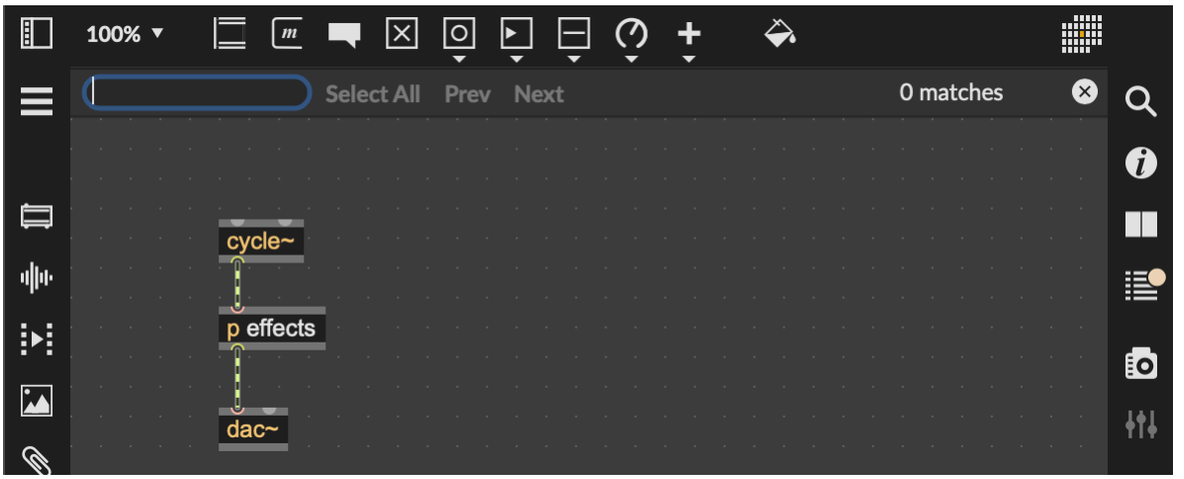
Press x anywhere in an unlocked patcher to view a list of key commands.

Patching Mechanics

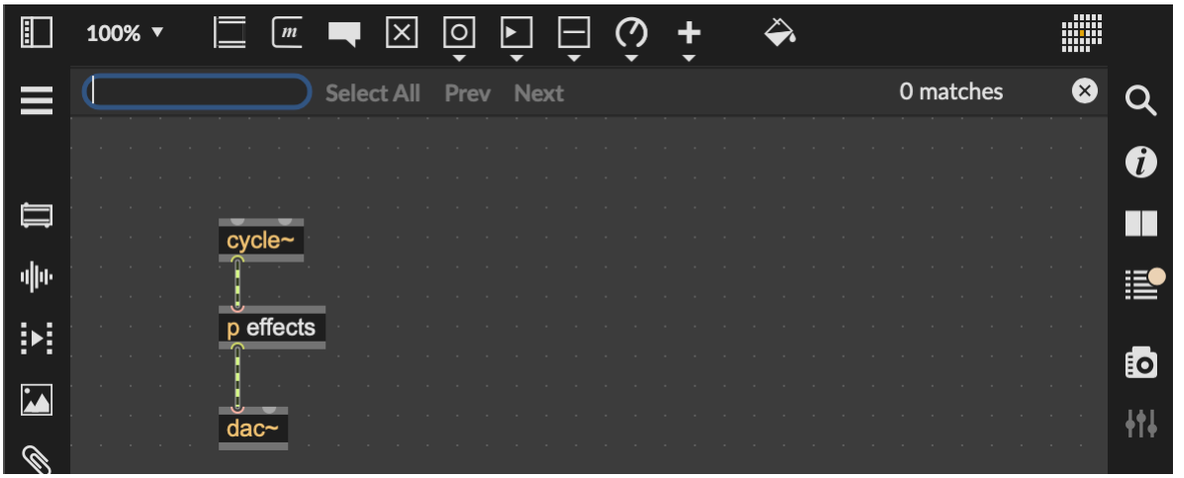
The Max patching interface supports a handful of useful shortcuts called Patching Mechanics. These are enabled by default, and you can toggle them on and off using the *Enable Patching Mechanics* preference in Max's preferences. These shortcuts make it easier to work with your patcher by reducing the amount of precise clicking needed to create and arrange objects. See Patching Mechanics for more information.

Finding Objects and Text

Select *Find...* from the *Edit* menu to search for text in your patcher. Max will display a search interface at the top of your patcher view.



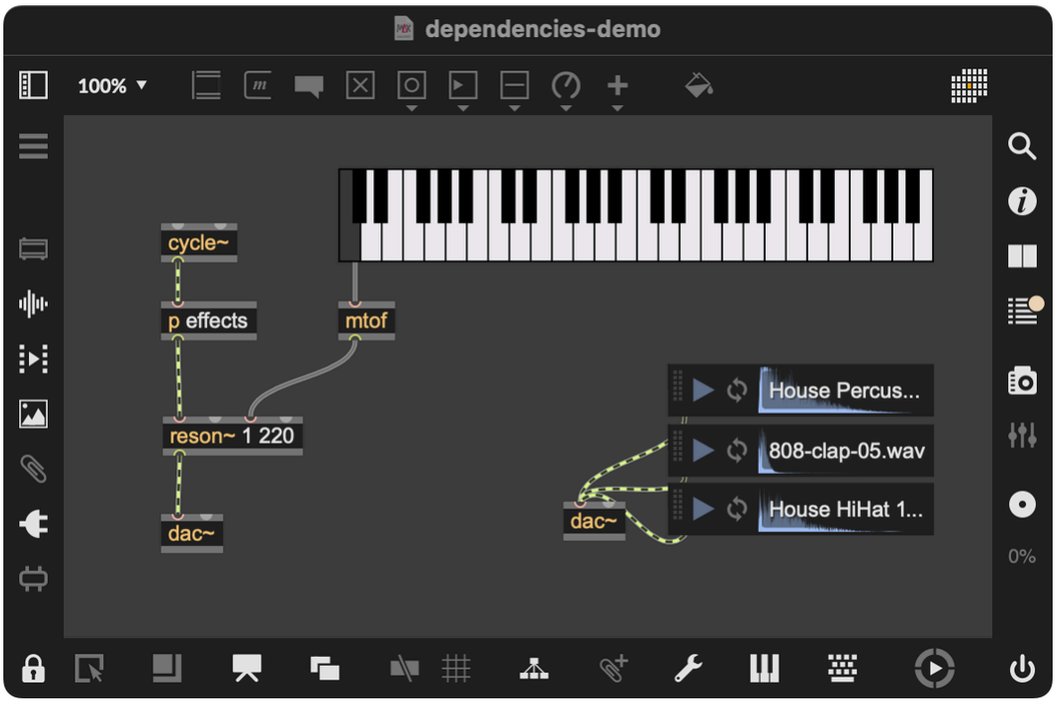
As you enter your search text, Max will show you all of the objects whose text matches your query. It can also identify objects that do not match in the current patcher, but which match in some subpatcher. Click on the *in a subpatcher* text in the search interface to reveal the subpatcher with the matching object.



The 'effects' subpatcher contains a cycle~ object, so the search view indicates that there are additional matches within a subpatcher.

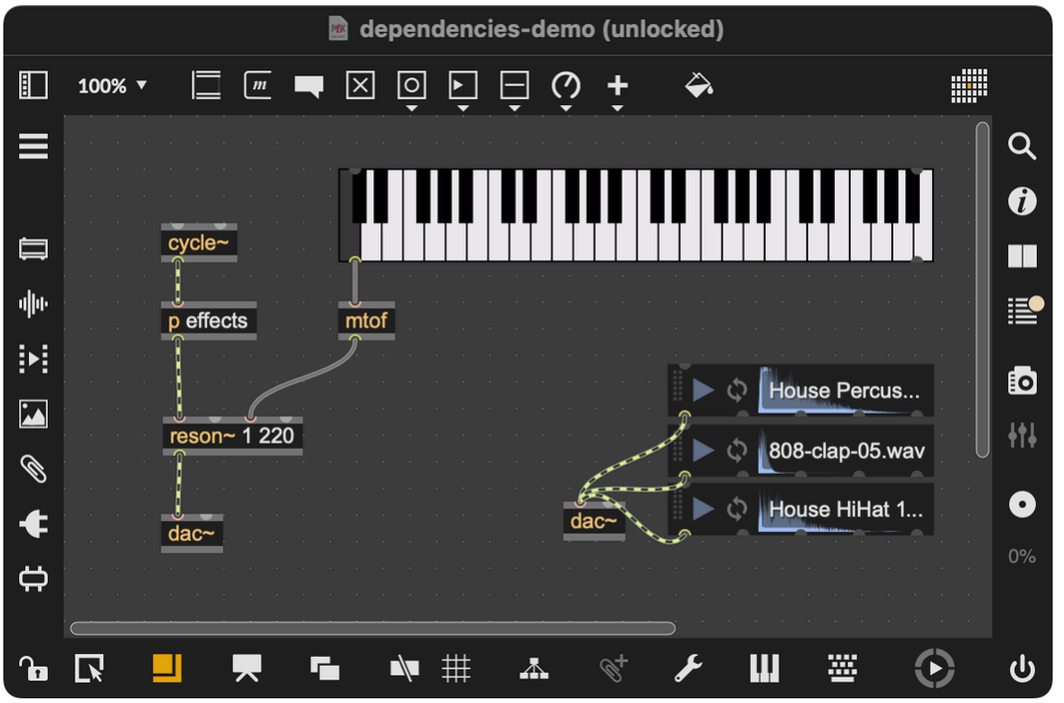
Patching Margin

By default, Max will only allow scrolling only until the object in the bottom-right of your patcher is just in view. That means that even if the current view is full of objects, you won't be able to scroll to reveal more empty space.



Even though the patcher view is full, there still aren't any scroll bars.

You might prefer to allow scrolling no matter what, so that you can always scroll in order to have more empty space to patch in. Enable the *Patching Margin* icon in the bottom toolbar in order to add additional space to the patcher view to the bottom and right.



With Patching Margin enabled, scroll bars appear when in Edit Mode, giving you more room to patch in

Reinitialize

You can bring a patcher back to an initial state by selecting *Reinitialize* from the *Edit* menu. This has two effects:

- Sends a `loadbang` to the patcher, triggering any loadbang objects in the patcher.
- Resets all parameters to their initial value.

Reinitialize works especially well with patchers that take advantage of parameter-aware objects

Patch Cords

Types of patch cords 593
Creating Patcher Cords 594
Editing Patcher Cords 594
Disabling patch cords 596
Viewing Patcher Cord Contents 597
Styling Patcher Cords 598

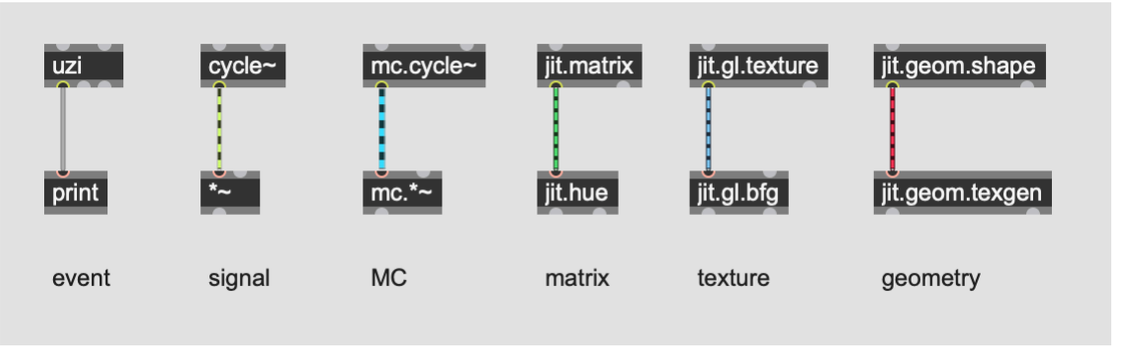
The inlets and outlets Max objects are connected together using patch cords.

Types of patch cords

There are six kinds of patch cords:

1. Event - Messages sent between Max objects, handled by the scheduler
2. Signal - Audio processed in blocks
3. MC - Multichannel signals
4. Jitter matrix - Video and other multidimensional data, processed on the CPU
5. GL texture - Multidimensional data residing on the GPU, part of graphics processing
6. Jitter geometry - Half-edge geometry structures

Patcher cords can be distinguished by their stripe patterns and colors.



Creating Patcher Cords

You can connect objects with patch cords in a few ways:

- Clicking on an inlet/outlet and dragging the mouse to another inlet/outlet
- Clicking on an inlet/outlet and then clicking on another inlet/outlet
- Hovering or selecting a patch cord, and then using the green or red circles by clicking or dragging to new inlets/outlets

To disconnect patch cords, you can:

- Select a patch cord and use the backspace or delete key on your keyboard
- Hovering or selecting a patch cord, and then dragging the green or red circle to an empty spot on the patcher window

> When creating patch cords, if you hold shift as you finalize a connection, Max will automatically start a new connection from the same inlet or outlet. This can be *extremely* useful when creating many patch cords from the same inlet/outlet.

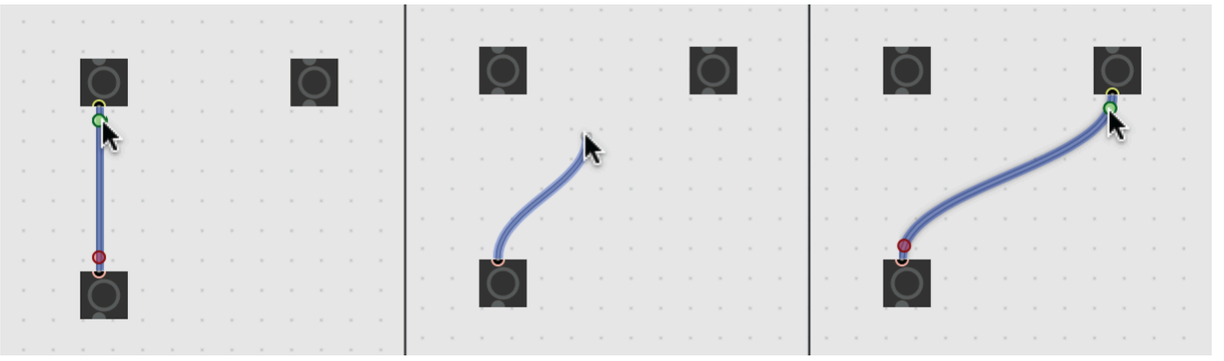
Editing Patcher Cords

Selecting patch cords

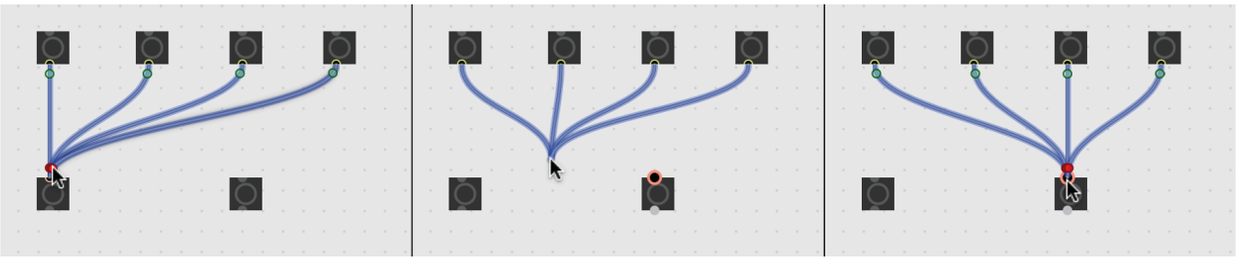
Click on a single patch cord to select it. To select multiple patch cords, hold down `Option` (macOS) or `alt` (Windows) while drawing a selection rectangle. This will select objects as well as patch cords.

Re-connecting patch cords

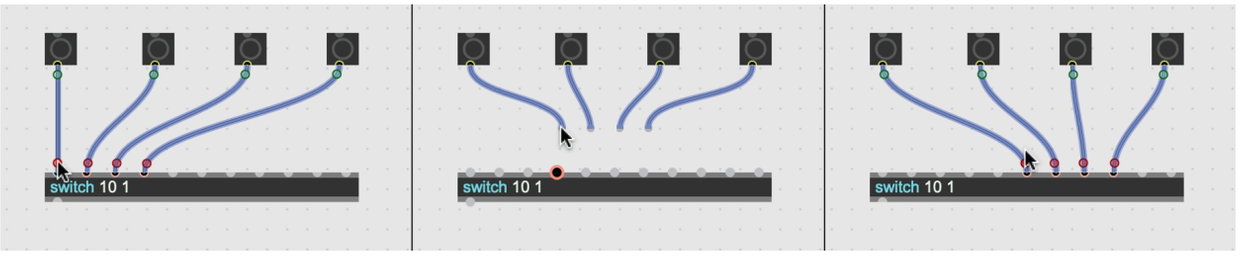
When a patch cord is selected, a green and red circle will appear next to the inlet and outlet. Click and drag on either of these to move the patch cord, leaving either the inlet or outlet connected.



With this same technique, you can move multiple connections from one object to another.

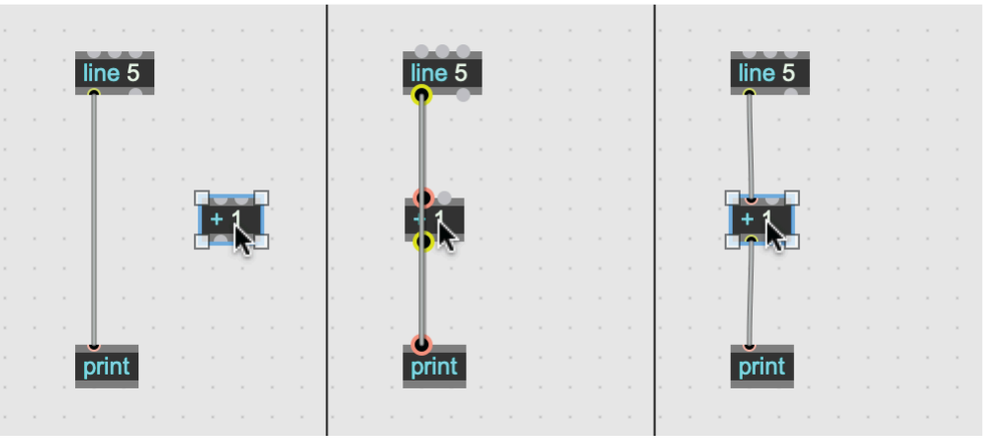


You can also shift patch cords along the inlets or outlets of an object.



Inserting/removing objects from a patch cord

Insert an object into the middle of a patch cord by holding `Shift` while dragging the object.



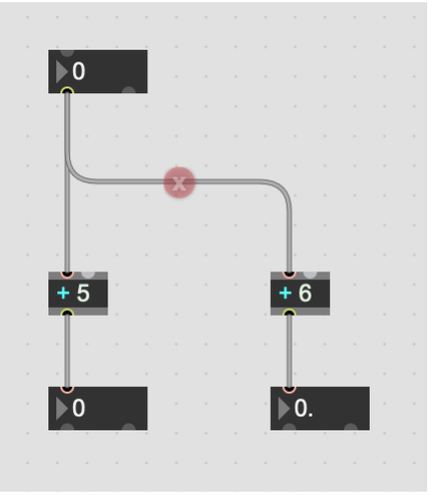
You can also drag an object out of a patch cord by holding shift while dragging the object. This will only work if the object has a single connection to the first inlet, and a single connection to the first outlet.

Patching mechanics

For mouse-free patching, Max supports Patching Mechanics. With this feature activated, you can do things like create and delete patch cords without using the mouse.

Disabling patch cords

Right-click on any patch cord and select *Disable Patcher Cord* in the contextual menu to disable it. Select *Enable Patcher Cord* from the contextual menu to re-enable the patch cord.



Viewing Patcher Cord Contents

Enabling probing

With Probing enabled, you can view the contents of a patch cord by hovering over it. Max supports Event Probing, Signal Probing, and Matrix Probing.

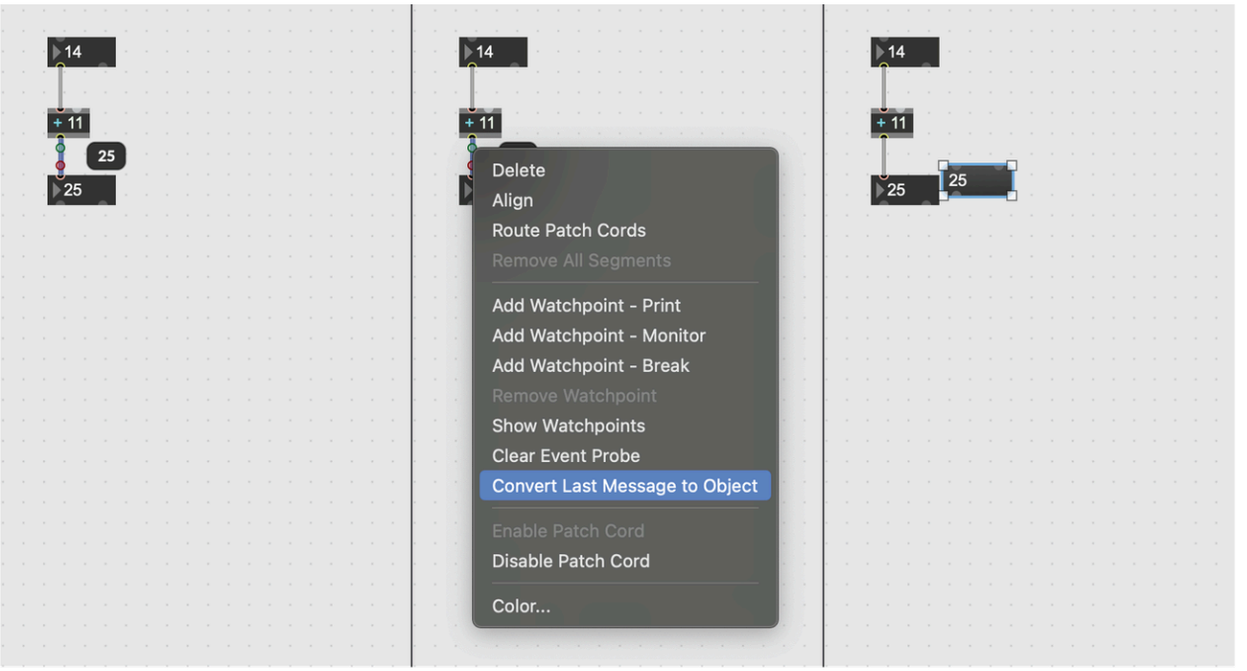
Debugging patch cords

With **Debugging** enabled, you can attach Break Points and Watch Points to a patch cord. These will show the contents of a patch cord in a separate window, and pause execution when a message flows through a patch cord.

Illustration Mode is another helpful way to visualize the flow of messages along patch cords.

Extracting a message from a patch cord

With the Event Probe enabled, right-click on an event patch cord and select *Convert Last Message to Object*. This will create a message box containing the contents of the last message to pass through that patch cord.



Styling Patcher Cords

Segmented Patcher Cords

By default, patch cords are drawn in a curved style, but it is also possible to use a "segmented" style which has joints and corners. You can change the default style in the Max Preferences with the *Segmented Patcher Cords* option. To create a segmented patch cord when the *Segmented Patcher Cords* option is not checked in the Max Preferences menu (or to create a curved patch cord when Segmented Patcher Cord is enabled), hold the Shift key down when clicking on an outlet.

When creating segmented patch cords, click at each point where you want the patch cord to bend, and then click on the inlet/outlet of the other object.

To correct a segmented patch cord while you draw, Option-click (macOS) or Alt-click (Windows) to erase the most recent patch cord line segment. To remove a patch cord completely, command-click (macOS) or control-click (Windows) anywhere in the Patcher window.

To automatically create a segmented patch cord from a curved patch cord, right click on the patch cord and select *Align*. To automatically create route a segmented patch cord around objects, instead,

select *Route Patch Cord*. To change a segmented patch cord to a curved one, right click on the patch cord and select *Remove All Segments*.

If you are making a segmented patch cord and want to make a corner over an object, hold down the control key to disable the normal auto-connection feature.

Coloring patch cords

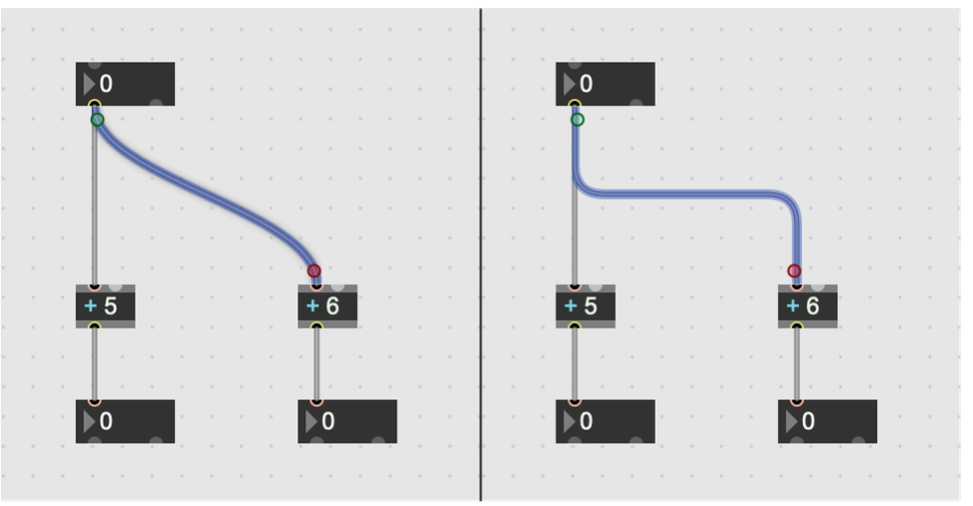
Patch cords can be individually colored, or styled as a whole using the **Format Palette**.

To color an individual patch cord, or a selection of patch cords, select all of the patch cords that you want to color and right-click on one to open the contextual menu. Select *Color...* to open the color picker.

To style all of the patch cords in a patcher at once, open the Format Palette and edit the *Patchline Color*.

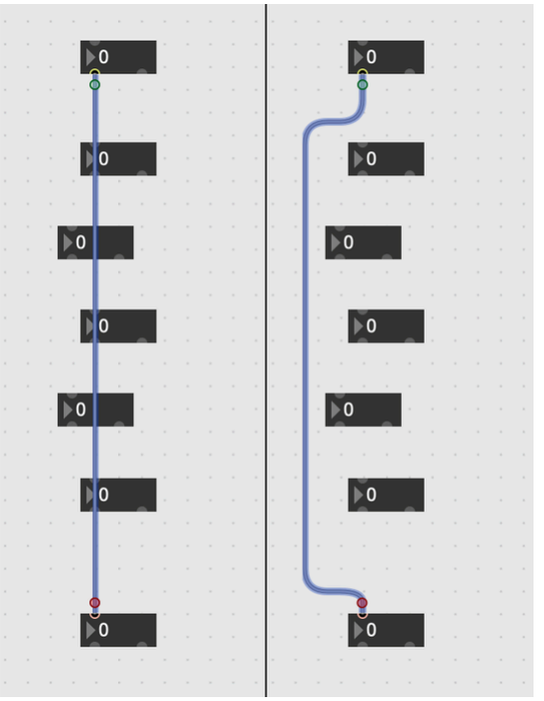
Aligning and routing patch cords

With one or more patch cords selected, open the *Arrange* menu and select *Auto Align* to create aligned, segmented patch cords.



You can also route patch cords automatically, creating segmented patch cords that move between objects. With one or more patch cords selected, open the *Arrange* menu and select *Route Patcher*

Cords.



Hiding patch cords

To hide patch cords in locked patcher mode, right click on a patch cord and select *Hide on Lock*. To make hidden patch cords visible again, right click on the patch cord and select *Show on Lock*.

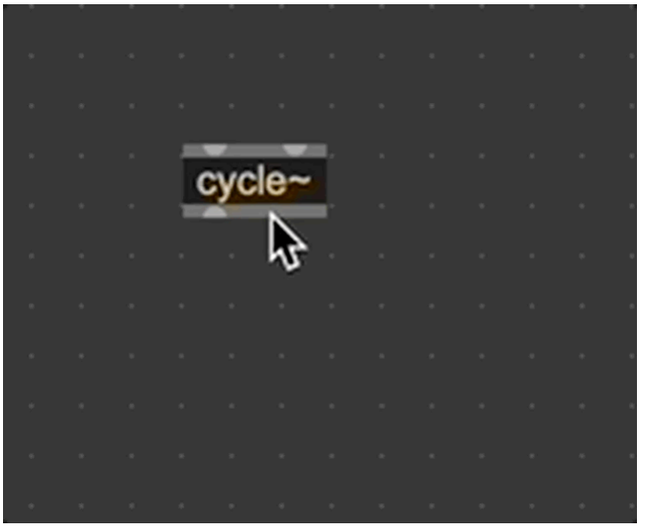
Patching Mechanics

Drag to create new objects 601
Create a new object along a patch cord 601
Insert and remove an object from a patch cord 602
Navigating the patcher selection with the keyboard 603

The Max patching interface supports a handful of useful shortcuts called Patching Mechanics. These are enabled by default, and you can toggle them on and off using the *Enable Patching Mechanics* preference in Max's preferences. These shortcuts make it easier to work with your patcher by reducing the amount of precise clicking needed to create and arrange objects.

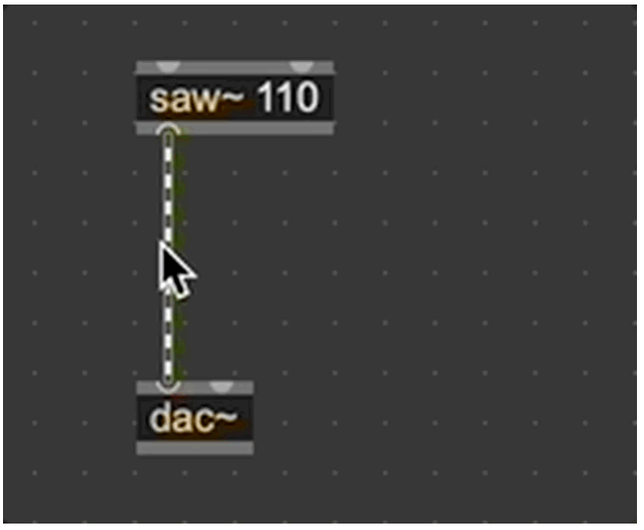
Drag to create new objects

With the patcher unlocked, hold down `ShiftAlt` and drag an object to instantly create a new object connected to the first.



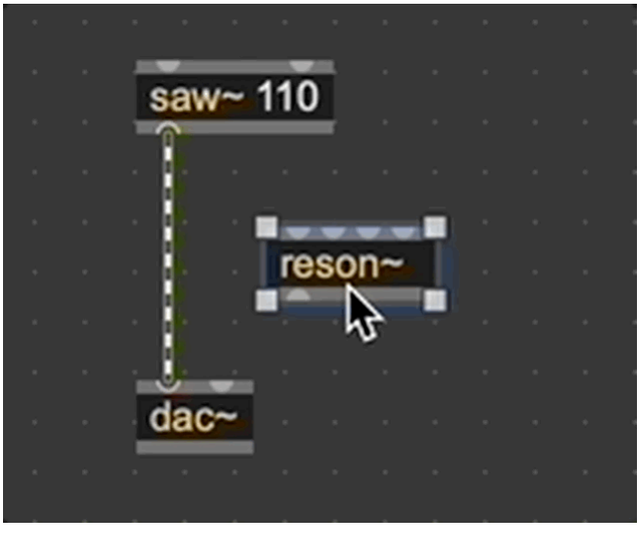
Create a new object along a patch cord

With a patch cord selected, press `Shiftn` to create a new object and insert it into the selected patch cord.



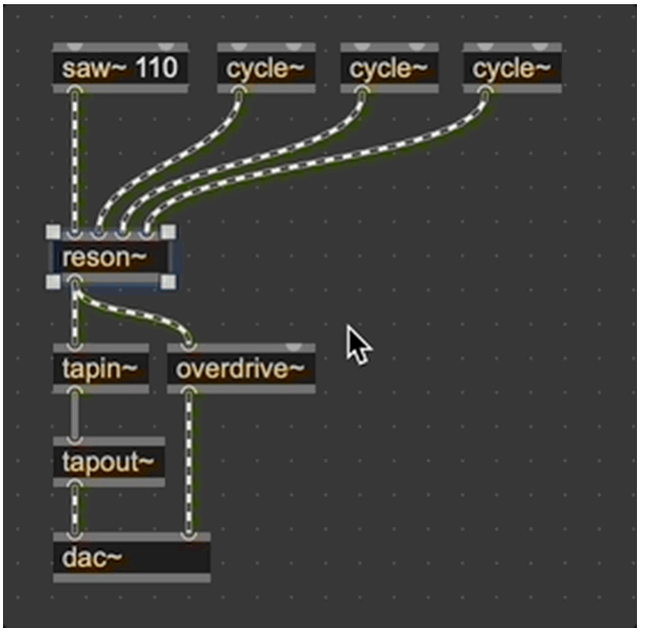
Insert and remove an object from a patch cord

Hold down `Shift` and drag an object into an existing patch cord, aligning its first inlet with the patch cord. Max will insert the selected object into the patch cord, replacing its existing connection. In the same way, hold down `Shift` and drag an object out of a patch cord to remove that object from the connection. Max will replace the connection with the selected object removed.



Navigating the patcher selection with the keyboard

With an object selected, you can press `AltUp` to select a patch cord leading into the object. Pressing `AltDown` moves the selection to patch cords coming from the object. Then, `AltLeft` and `AltRight` rotate your selection through patch cords.



Web Browser and jweb

jweb 604
JavaScript Communication 605
Receiving Messages 606
Sending Messages 607
Interacting with Max Dictionaries 608
Jitter Matrices 609
Debugging 612

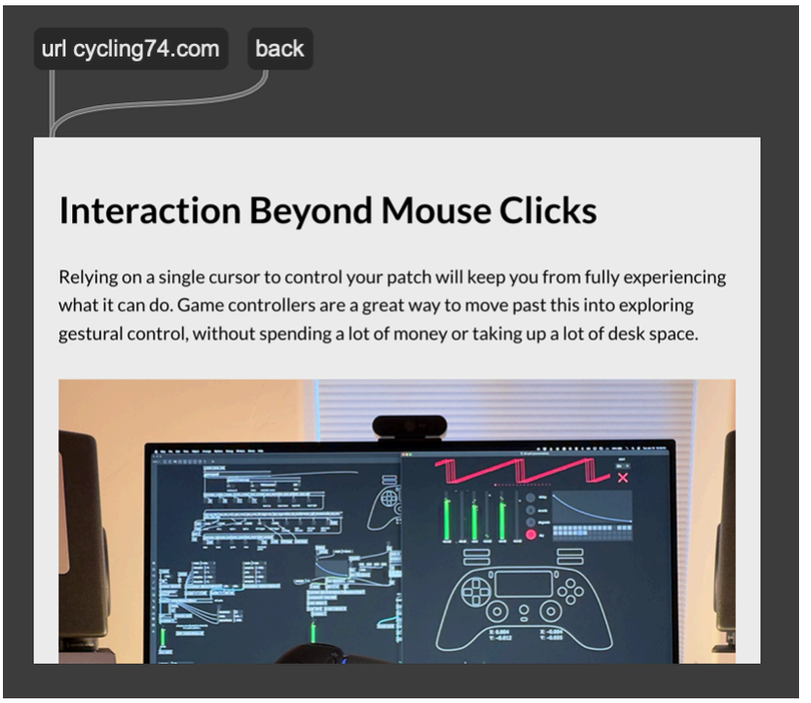
Max contains a web browser, implemented with the Chromium Embedded Framework (CEF). You can access this browser through the jweb object, which lets you embed a web browser in your Max patcher. This object has multiple versions for supporting streaming audio and video as well:

- jweb~: captures audio from whatever webpage you're on, letting you route that audio through your patcher
- jit.web: outputs the rendered page as a Jitter Matrix
- jit.web~: outputs a Jitter matrix as well as captured audio
- jit.gl.web: outputs the rendered page as a texture
- jit.gl.web~: supports texture and captured audio output

Finally, you can use the MaxBridge API to receive messages from Max, to send messages to the object outlets, to receive Jitter matrices, to get and set Max dictionaries, and to get/set Jitter matrices by name.

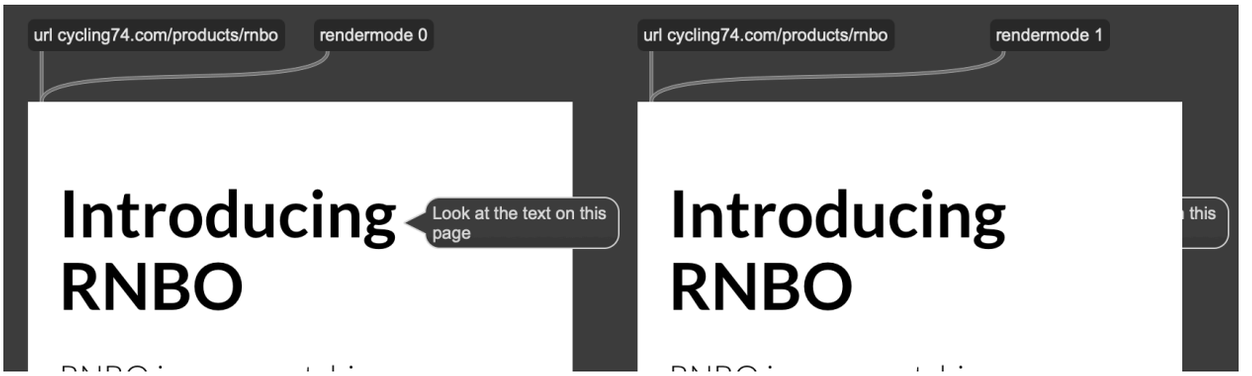
jweb

Create a jweb object to make an instance of an embedded web browser.



A jweb object, showing a web page.

The `@rendermode` attribute determines whether the page is rendered directly in the object view, or whether it's rendered offscreen and then composited into the patcher. Onscreen rendering is slightly more efficient, but the jweb browser view will always render on top of other objects.



On the left, offscreen rendering. On the right, onscreen rendering. Onscreen is more efficient, but always renders on top of any objects in the patcher.

JavaScript Communication

You can communicate with the contents of a page loaded with jweb using the MaxBridge API through the `max` object. When jweb loads a page, it adds this object to the global `window` object. You can use this to determine, from JavaScript, whether the page was loaded in Max.

```
if (window.max) {
  console.log("This webpage is loaded in Max, from a jweb object");
} else {
  console.log("This webpage is not loaded in Max.");
}
```

Asynchronous messaging

> It's important to note that CEF runs in a separate process from the rest of Max. If you send a message to a jweb object, it will be handled asynchronously. Any messages that come out of jweb from a call to `window.max.outlet` will always be handled on the low-priority queue. For the same reason, MaxBridge API functions that fetch data from Max, like getJitterMatrix or getDict, will also work asynchronously.

Receiving Messages

Use the `bindInlet` function to receive messages from Max inside a JavaScript callback.

```
window.max.bindInlet("something", () => {
  /*
   * code here will be executed whenever jweb receives
   * the symbol "something"
   */
});
```

The `bindInlet` function can also register a callback that will accept arguments.

```
window.max.bindInlet("addNumbers", (a, b) => {
  console.log(`${a} plus ${b} is ${a + b}`);
});
```

You can use the spread operator to handle lists, or messages with a variable number of arguments.

```
window.max.bindInlet("printLength", (...values) => {
  console.log(`The list has ${values.length} elements`);
});
```

Sending Messages

Use the `outlet` function to send messages to the outlet of the jweb object.

```
// output a string
window.max.outlet("foo");

// output a list
window.max.outlet("foo", 1, 2);

// output contents of array with prepended "foo" message
let ar = [1, 2, 3, 4];
window.max.outlet.apply(window.max, ["foo"].concat(ar));
```

You can also send a message out of jweb using the `href` attribute of an `anchor` tag. Note that a message send this way will be output with the symbol "maxmessage" prepended. The contents of the message should be separated by the "/" character.

```
<a href="maxmessage:name/param1/param2"></a>
```

Interacting with Max Dictionaries

Use `getDict` to get the contents of a Max dictionary.

```
let nestedValue;

// access dictionary
window.max.getDict("dictName", (dict) => {
  // dict is a JavaScript object. Dictionary keys
  // will be JavaScript object properties, so you
  // can fetch values using typical JavaScript syntax.
  nestedValue = dict.a;
});
```

Use `setDict` to set the contents of a dictionary.

```
let obj = {
  a: "1",
  b: "2",
  c: "3",
};

window.max.setDict("dictName", obj);
```

There is no special function to change just one value of a dictionary. To update a dictionary, call `getDict` followed by `setDict`.

```
window.max.getDict("dictName", (dict) => {
  // Increment a numerical value
  dict.inc = dict.inc + 1;

  window.max.setDict("dictName", dict);
});
```

Jitter Matrices

You can read/write Jitter matrices as well. The implementation uses shared memory to transfer the matrix data between processes, and this transfer happens impressively fast. There are two approaches:

- using bindJitterMatrix and bindJitterImage to stream matrices received via the `jit_matrix` message
- similar to dictionaries, using the window.max.getJitterMatrix and window.max.setJitterMatrix functions

For matrix streaming, register a handler to receive Jitter matrices pushed from Max via the `jit_matrix` message. These are received as either raw binary matrices, or as images. Both are main-frame only (not exposed in iframes).

```
// "binary" — verbatim matrix
window.max.bindJitterMatrix(onMatrix, opts);
// "image"  — 4-plane char → RGBA
window.max.bindJitterImage(onImage, opts);
```

If you call bindJitterMatrix, your `onMatrix` handler function will receive the contents of the matrix in its first argument. That argument object has `type`, `planecount`, and `dim` properties, as well as a `data` property, containing the matrix data itself.

Use the bindJitterImage function if you want to register an `onImage` handler instead. This function will be called with an argument containing `width`, `height`, and `data` properties. Image data is pre-swizzled from Jitter's ARGB to RGBA, so it drops straight into a canvas or webgl context with no per-pixel work.

```
function onImage(image) {
  ctx.putImageData(new ImageData(image.data, image.width, image.height),
0, 0); // 2D
  gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    w,
    h,
    0,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    image.data,
  ); // WebGL
}
```

Additionally both variants include metadata properties containing timing and other performance metrics. Check the MaxBridge API Reference for more details.

- `timestamp`: producer send time, wall-clock ms (Date.now()-comparable)
- `seq`: monotonic source frame number
- `received`: performance.now() time the moment the frame arrives from Max
- `dropped`: frames discarded since the previous delivered frame

The optional `opts?` argument allows you to configure the received matrix queue (each handler has its own queue).

option	default	meaning
queue	8	max frames buffered for this binding before dropping
drop	'oldest'	when full: 'oldest' discards the stalest queued frame (lowest latency); 'newest' rejects the incoming one

Alongside the streaming receive API, a page can copy matrix data to and from a named Max matrix on demand. This is similar to getDict/setDict, but for matrices instead of dictionaries. You address a matrix by the name it's registered under in Max (e.g. a jit.matrix `mybuf` in the patch). Both calls are main-frame only (not exposed in iframes).

window.max.setJitterMatrix(name, m) copies a matrix from the page into the named Max matrix.

```
window.max.setJitterMatrix("mybuf", {
  type: "char", // "char" | "long" | "float32" | "float64"  (default
"char")
  planecount: 4,
  dim: [640, 480],
  data: myUint8Array, // a TypedArray (or ArrayBuffer) of samples
});
```

The named matrix is resized to the shape you describe (type / planecount / dim), then your data is copied in. It's a no-op if a name isn't a registered matrix or the resize can't produce exactly the shape you asked for. The target matrix has to already exist in the patch.

`window.max.getJitterMatrix(name, callback)` reads the named Max matrix and hands a copy back to the page.

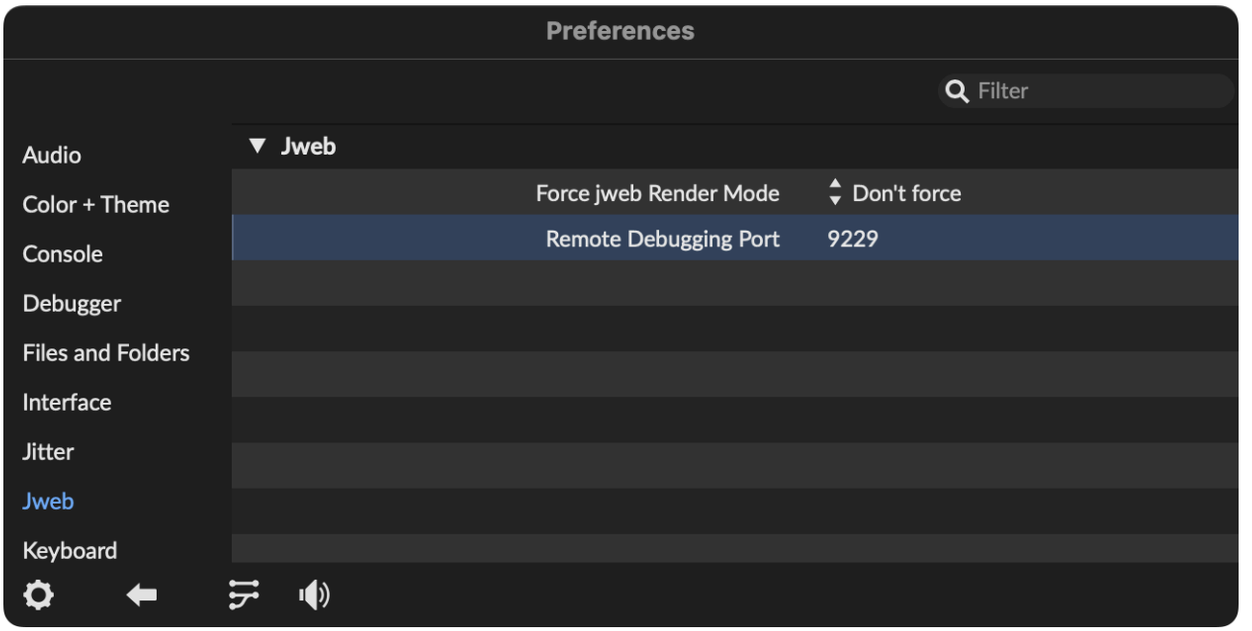
```
window.max.getJitterMatrix("mybuf", function (m) {
  if (!m) return; // matrix wasn't available — see below
  // m.type / m.planecount / m.dim describe m.data
  console.log(m.dim, m.data.length);
});
```

The callback is called exactly once, with either an object shaped like the received matrix core fields or null if the matrix couldn't be read (the name isn't registered, it has no data, or it's a type the bridge doesn't support). Always guard for null before touching the result. The read is asynchronous: it happens over in Max and the result arrives a little later, which is why it's a callback rather than a return value.

Debugging

You can debug a webpage loaded in jweb using Google Chrome. After you open a remote debugging port in Max, each jweb instance will be visible as a separate device in the Chrome debugger. When you open the instance in Chrome, you'll be able to view the JavaScript console, set breakpoints, and monitor network traffic.

First, make sure to enable a debug port in Max. From Max's preferences, enable remote debugging by picking a port. Chrome defaults to browsing devices on port 9222 and 9229, so these are good choices.



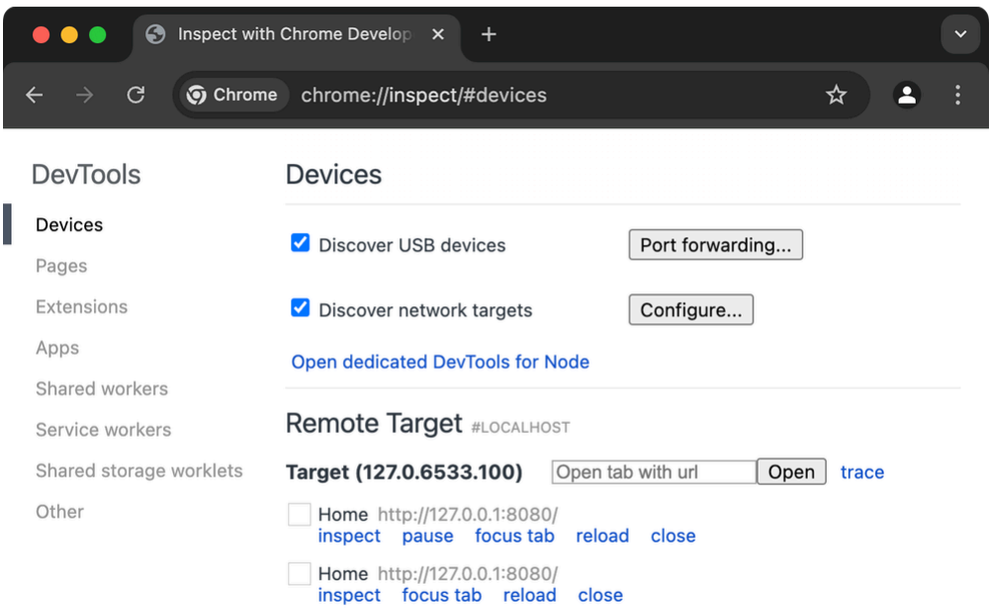
> You'll need to restart Max for these changes to take effect.

Now, load the webpage you want to debug in jweb.

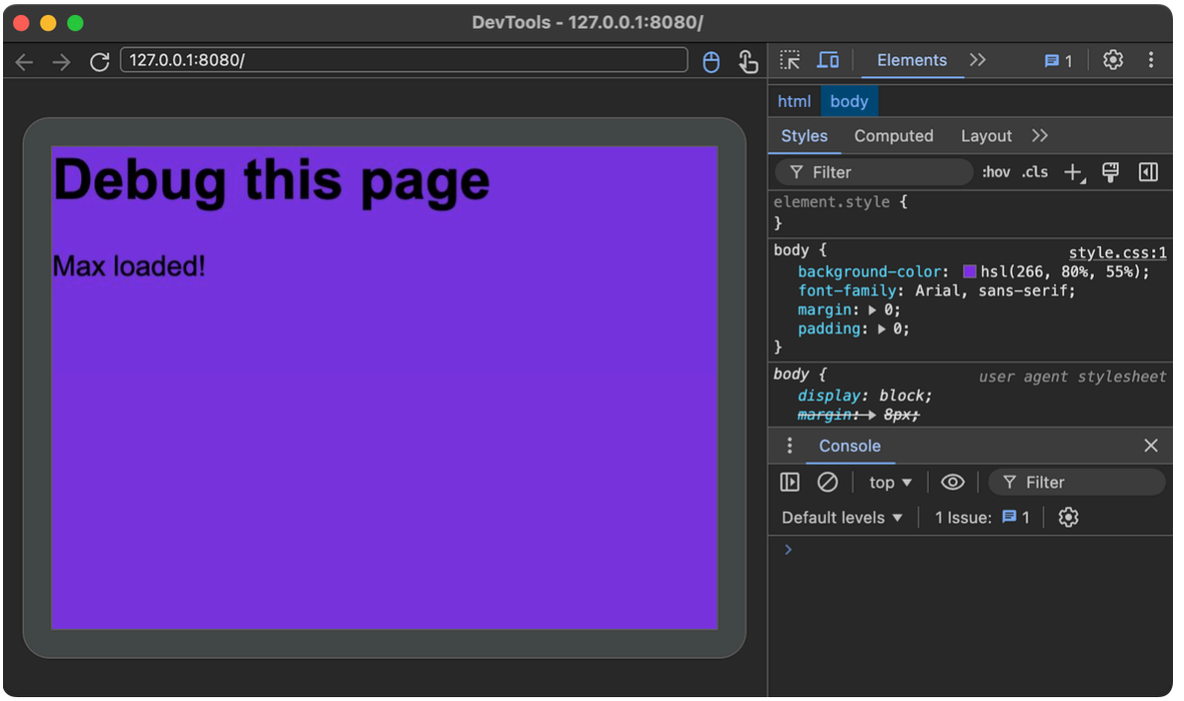


A webpage loaded in jweb. This one happens to be served locally.

With the webpage loaded in jweb, open Chrome and navigate to `chrome://inspect/#devices`.



Click "inspect" under the page that you want to debug. Chrome will open a new window to debug your page. (If you don't see your jweb instance listed, it might be because you set Max to a jweb debug port other than 9222 or 9229. Click *Configure...* to enable your desired port.)



From here, you can debug this webpage just like you would any other. Look for guides releated to web development and JavaScript programming for more information.

Reuse and Organization

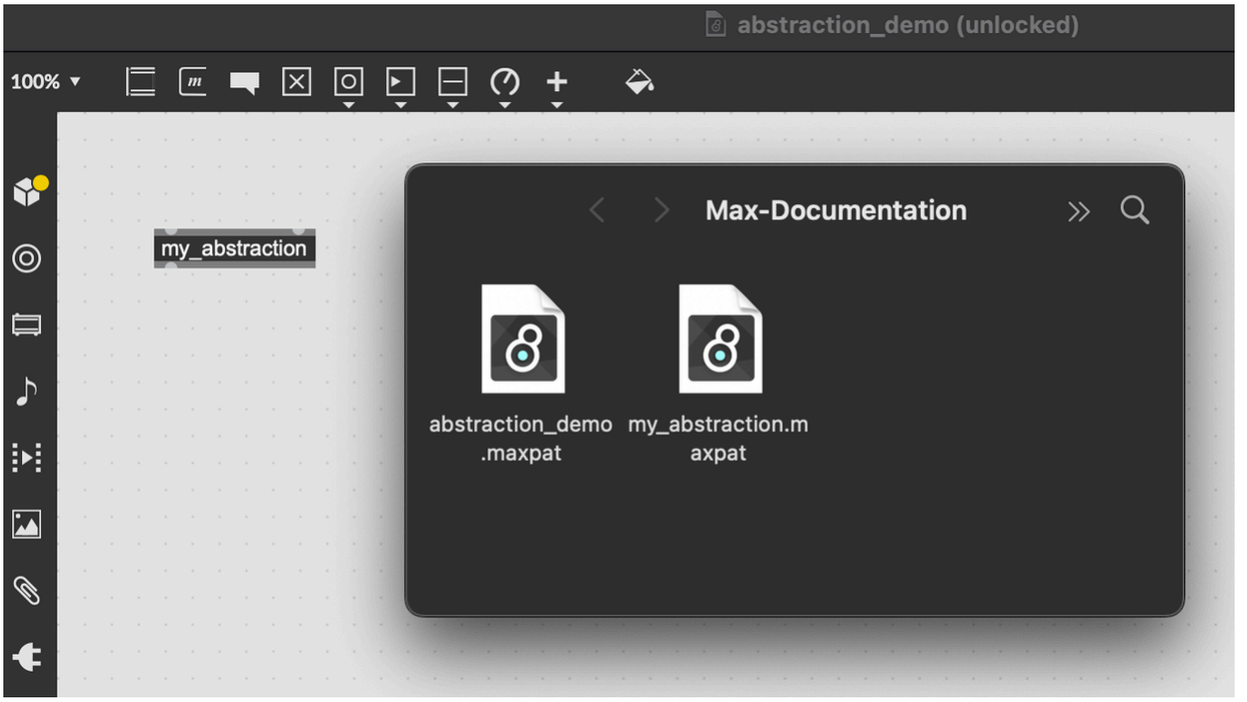
Abstractions

Creating an Abstraction 617
Editing an Abstraction 618
Inlets and Outlets 618
Abstractions vs Subpatchers 619
Arguments and Attributes 620
Unique Identifiers 621
Description, Tags, and Browsing 622
Help Patches 623

Any saved Max patcher can be loaded into another patcher as an **Abstraction**. When you create a new Max object, if that object has the same name as a Max patcher, Max will load that patcher into the new object. This lets you create patches that you can reuse over and over again, helping you patcher faster and more effectively.

Creating an Abstraction

When you create a new Max object, Max will look through the search path for an external or abstraction with that same name. If Max finds a `.maxpat` Max patcher file, it will load that file as if it were a Max subpatcher. Just like a subpatcher, you can view the patcher contained in an abstraction by double-clicking on it in a locked patcher view.



The object named `my_abstraction` loads the file `my_abstraction.maxpat` as an abstraction

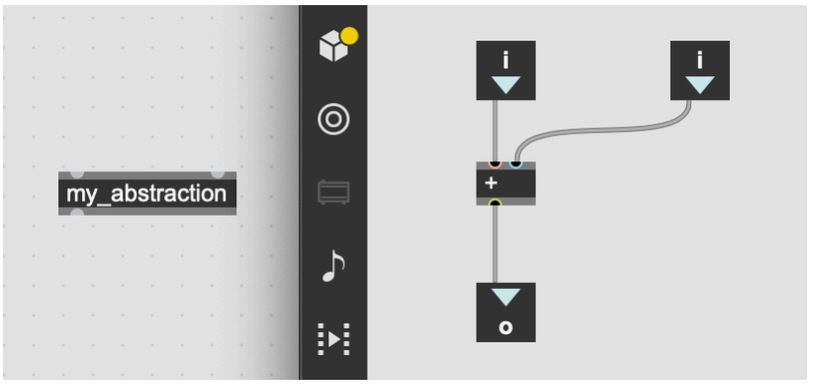
> Your Max patcher can only load another patcher by name if it appears in the current search path. That means it's very common to see abstractions either saved in the same folder as the parent patcher, or else included as part of a Project. Of course, you can load abstractions from anywhere in the Max search path, but managing abstraction dependencies is a common use case for projects too.

Editing an Abstraction

If you open an abstraction from a parent patcher, it may open in read-only mode. In this mode, you must click the *Modify read-only* icon to unlock it for editing. See Modify read-only for more information.

Inlets and Outlets

An abstraction will have as many inlets or outlets as it has inlet or outlet objects. If you create a new inlet and save your abstraction, the parent patcher (the patcher containing the loaded abstraction) will update to include the new inlet. The order of the inlets in the parent patcher corresponds to the order of inlets in the abstraction. So, if you swap the position of two inlet objects in the abstraction, those objects will map to different inlets in the loaded abstraction.



The abstraction has two inlet objects and one outlet object, so the loaded abstraction object has two inlets and one outlet.

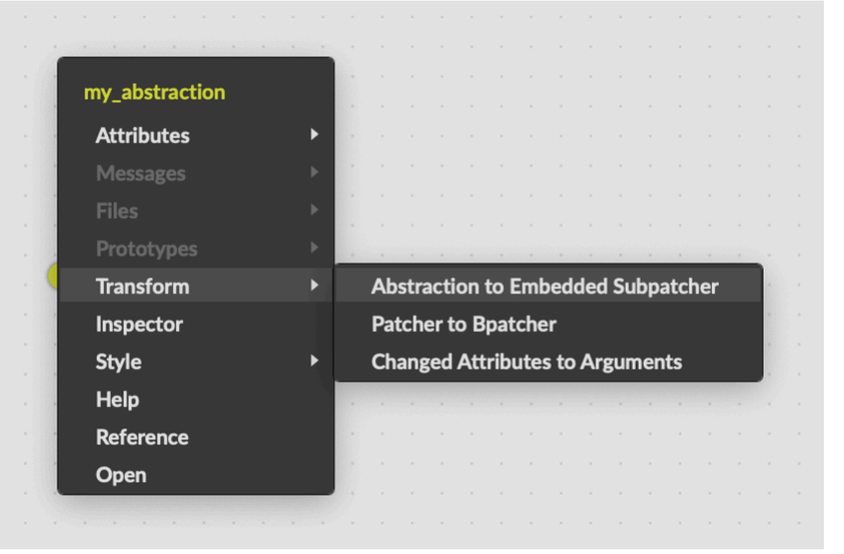
Just like a regular Max object, an abstraction can add **Comments** to its inlets and outlets. If you set the `@comment` attribute on an inlet or outlet object, then when you mouse over the inlet or outlet you will see that text displayed.

Abstractions vs Subpatchers

Abstractions and subpatchers are very similar, but they have a few key differences. For a start, abstractions reference a saved `.maxpat` file, while subbpatchers are embedded in their parent patcher. That means that you can't edit an abstraction without changing the original file. If you open an abstraction patcher, you will see that the normal *Unlock* icon in the patcher toolbar has been replaced with a pencil icon. Hovering over the icon, you will see the text "Modify read only". If you click this icon to enable editing, then any changes you make to the abstraction will change the original file. Consequently, changes you make here will affect all instances of the abstraction. This is different from embedded subpatchers, where each subpatcher has its own data, and changing one does not affect the others.

Transformations

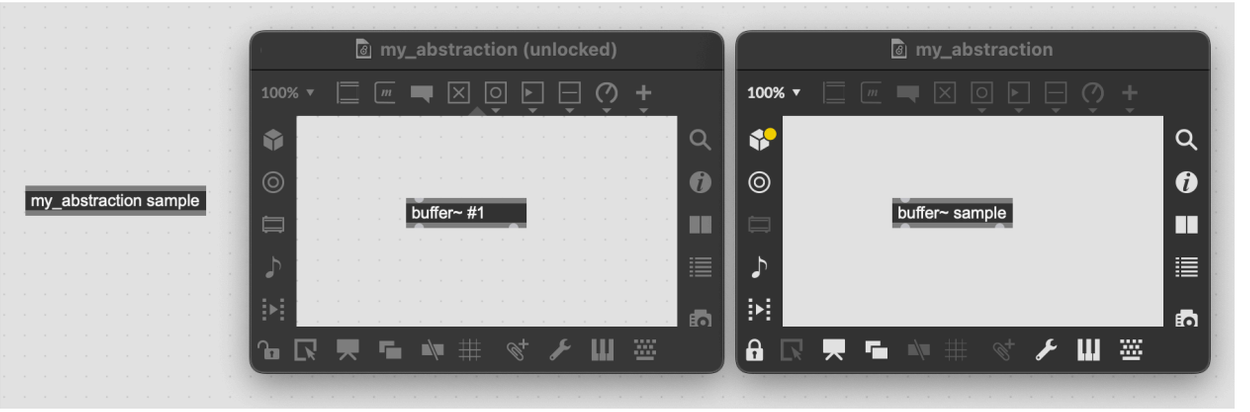
It's possible to convert a subpatcher into an abstraction using the action menu, with the *Abstraction to Embedded Subpatcher* command. This command will copy the contents of the abstraction file to a new subpatcher, which can be really useful if you want to modify an abstraction, but you don't want to change the original file.



Arguments and Attributes

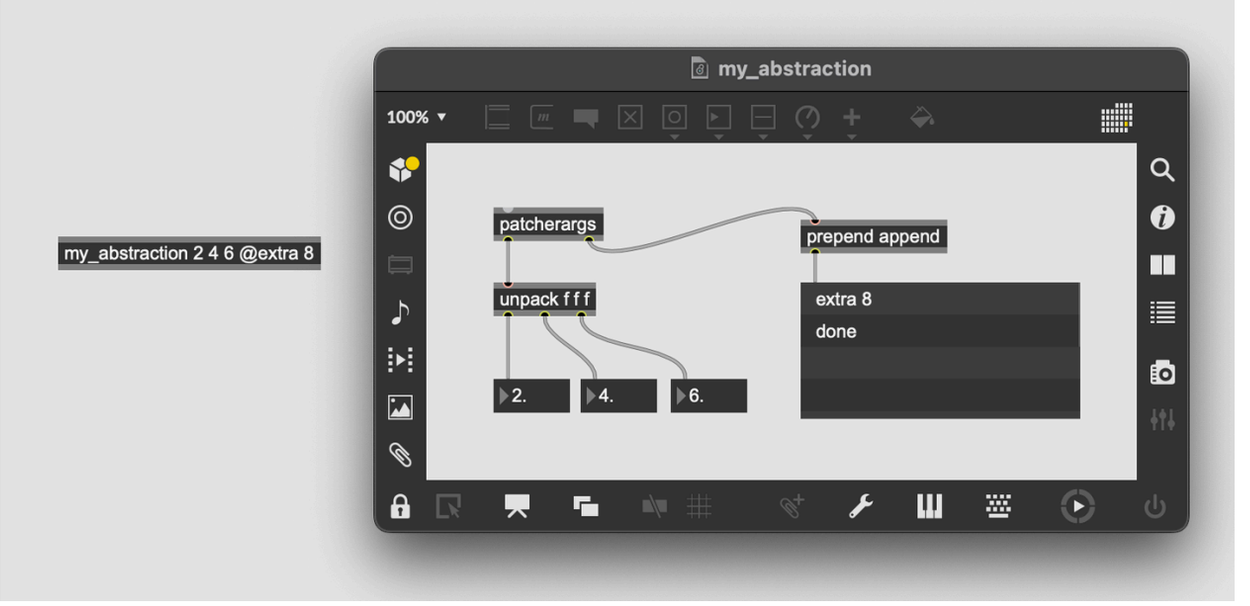
Abstractions can have arguments and attributes, just like a regular Max object. You can handle arguments in one of two ways, either using the patcherargs object, or using the special `#`-sign syntax for abstractions, Max for Live devices, and poly~ patches.

The `#`-sign syntax lets you quickly define "insertion points" where the text of any argument will replace the `#`-sign text. For example, if you have an object inside an abstraction with the text `buffer~ #1`, then the text `#1` will be replaced by the first argument passed to the abstraction object. If you lock and unlock the abstraction, you can alternate between viewing the text before and after replacement.



The #1 symbol in an abstraction will be replaced by the first argument in the parent object.

If you put a patcherargs object into your abstraction, then any arguments that you give to the abstraction object will come out of the left outlet of patcherargs when the abstraction is loaded. The patcherargs object can also process attribute-style arguments. These will come out of the right outlet of patcherargs, one at a time, followed by the `done` messages. In some sense, patcherargs works a lot like a loadbang or loadmess object.

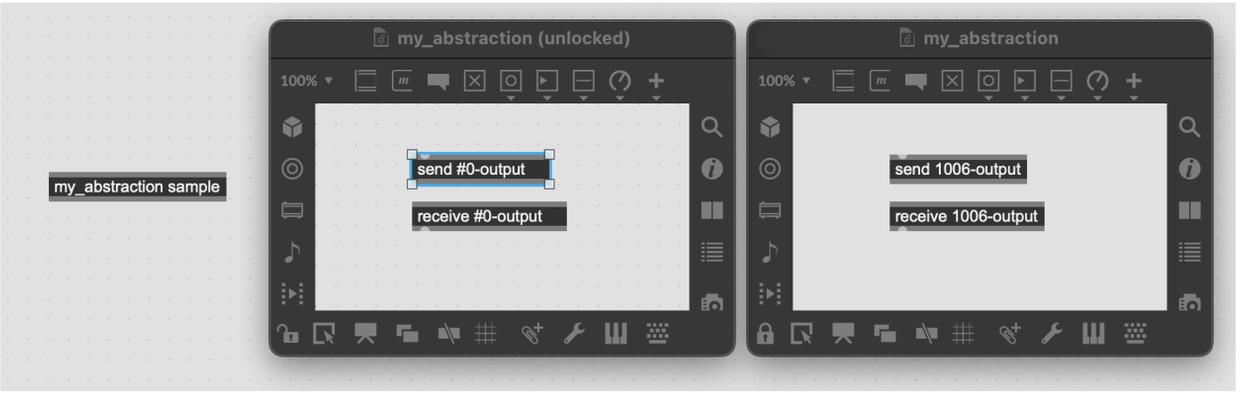


A patcherargs object outputs the argumetns of the parent object when the abstraction is loaded.

Unique Identifiers

Sometimes it's useful to create unique identifiers inside of an abstraction. Suppose you have a send/receive object pair inside your abstraction. If you create two instances of the abstraction, any messages sent to the send object in one will be received in the receive objects in both instances. If this isn't the behavior you want, you can use the special `#0` prefix to create a unique number. This `#0` prefix will be replaced by a number unique to each patcher instance, so different instances of the same abstraction will get a different unique number. In this way, you can use send and receive in an abstraction, without sharing messages among all instances of that abstraction.

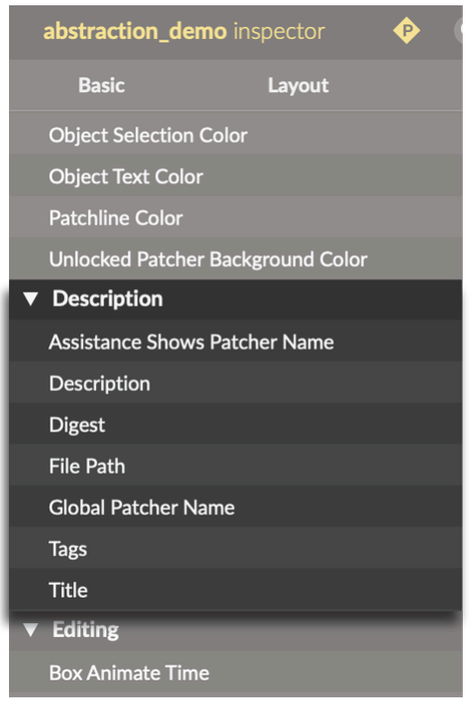
> The `#0` prefix will only be replaced with a unique number if it appears at the beginning of the word. In a message box containing the text `#0-text`, the `#0` will be replaced. However, if the same message box contained the text `text-#0`, then the `#0` would not be replaced.



The special #0 prefix will be replaced by a number unique to the specific patcher instance. Different instances of the same abstraction will get a different unique number.

Description, Tags, and Browsing

The Patcher Inspector has several attributes under the header `Description`, all of which let you describe your patcher in one way or another. Max will parse these attributes into its internal database, meaning you can search for patcher by their description using the File Browser. Tags can be especially useful, since you can search for tags using advanced search, with a search pattern like `tag:<your-tag>`.



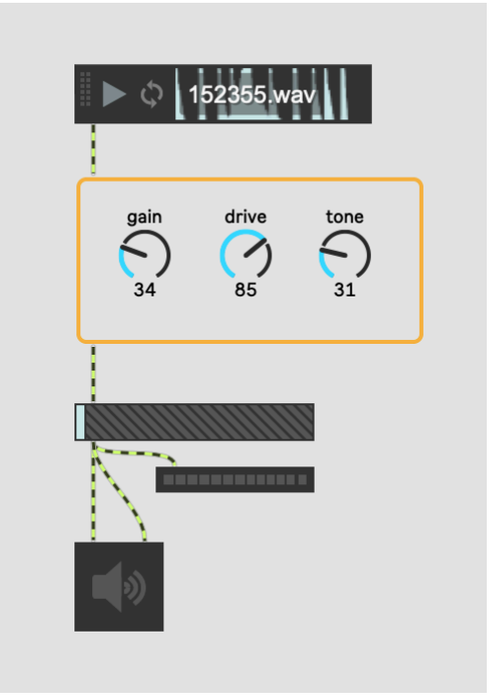
Help Patches

Abstractions can have Help Patches, just like regular Max objects. To create a help file for your abstraction, all you have to do is save a Max patcher with the `.maxhelp` extension, where the name of the patcher file is the same as the name of your abstraction. So, if your abstraction is called `my_abstraction.maxpat`, the help file should be named `my_abstraction.maxhelp`. If the help file is in Max's search path, then when you look up the help file for your abstraction, your custom help file should open.

Using bpatchers

Creating a bpatcher 624
Opening the Contained Patcher 626
Presentation View 626
Changing the Patcher's Offset 626
Embeded vs Referenced Patchers 627
Patcher Appearance in bpatchers 627

A bpatcher embeds the interface of a patcher inside a box in its parent patcher.



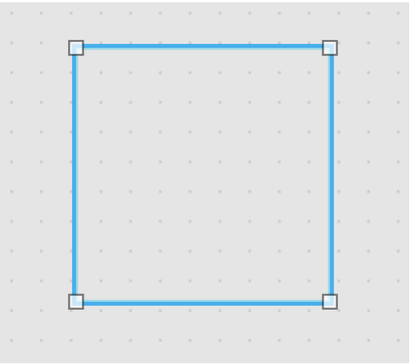
The audio processing patcher in the center of this patcher is inside a bpatcher.

Creating a bpatcher

You can create a bpatcher either by creating one from scratch, or by transforming an abstraction or subpatcher into a bpatcher.

Starting with an Empty bpatcher

Create a new object (by pressing the n key for example), then type the name `bpatcher`. A bpatcher will replace the object box when you click outside the box.



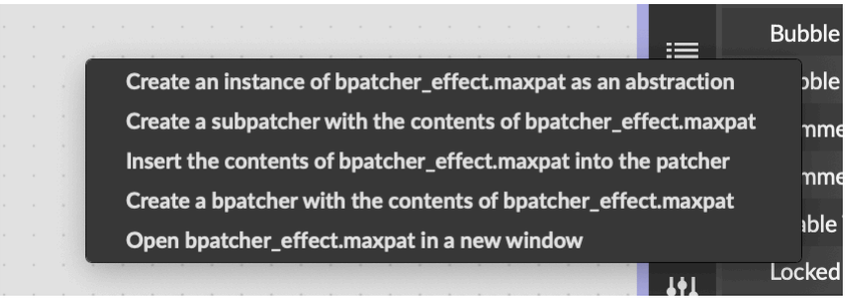
An empty bpatcher, immediately after creation

An empty bpatcher won't do anything. Setting the bpatcher's `@name` or "Patcher File" attribute will assign a patcher file within the search path to load. After a file is loaded, enable the `@embed` (*Embed Patcher in Parent*) attribute to save the contents of the file in the parent patcher.

You can also assign a file to a bpatcher by dragging a patcher file onto the object from the File Browser or one of the left sidebar browsers.

Starting with a Patcher File

- To turn a named patcher file into a bpatcher, start by pressing the e key. After the object box appears, type the file's name into the box, then click outside of the box. A bpatcher containing the file will replace the object box. For more details, see Patcher Appearance in bpatchers below.
- If you hold down the Option key while dragging a patcher file into a empty space in a patcher window, you'll see a contextual menu with several options. The *Create a bpatcher* option will let you create a bpatcher directly with the contents of the .maxpat file.



- Finally, you can drag any patcher file onto a bpatcher to replace the file the bpatcher currently uses.

Converting a Subpatcher or Abstraction

Given any subpatcher or abstraction, you can also open the Action Menu and select *Transform* > *Patcher to Bpatcher* to convert to a bpatcher. You can also transform a bpatcher to a subpatcher or abstraction with the same menu.

Opening the Contained Patcher

Choose *Open* from the bpatcher's Action Menu to view and edit the bpatcher's contained patcher in a separate window.

Editing an embedded patcher

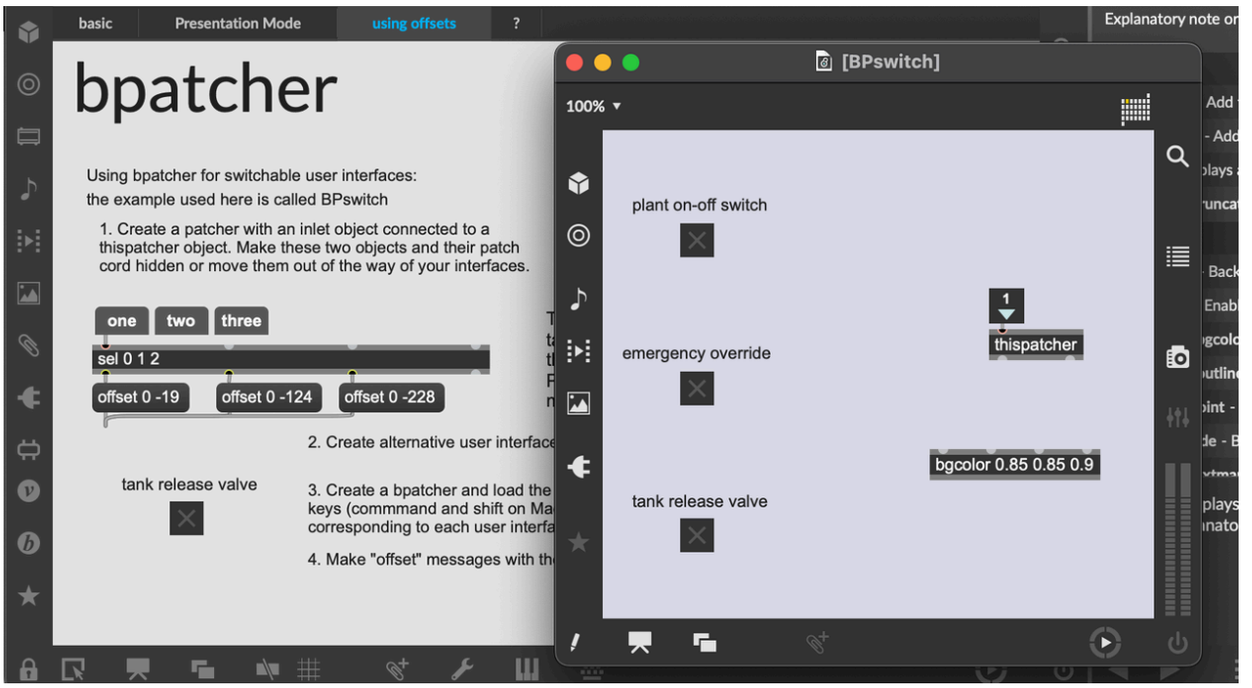
If you open an embedded bpatcher from a bpatcher object in the parent patcher, it may open in read-only mode. In this mode, you must click the *Modify read-only* icon to unlock it for editing. See Modify read-only for more information.

Presentation View

By default, a bpatcher will display its patcher's patching view. If you want to display the Presentation View instead, enable the `@openinpresentation` (*Open in Presentation*) attribute in the patcher file the bpatcher contains (*not* the bpatcher itself) using the Patcher Inspector.

Changing the Patcher's Offset

Often you'll want to set up a bpatcher to have a dynamic display, changing its presentation depending on some value. The best way to achieve this is by changing the `@offset` attribute of the bpatcher object, which will shift the origin of the displayed view by the desired horizontal and vertical amount. Since bpatcher is a slightly unusual object, you have to set this attribute using a thispatcher object in the embedded subpatcher. See the bpatcher help file for more details.



Overview of the offset technique for adjusting the display of a bpatcher

Embeded vs Referenced Patchers

As mentioned above, a bpatcher can either refer to an existing patcher file or embed its contents in its parent. The differences are similar to subpatchers and abstractions. When you enable the `@embed` attribute on a bpatcher, the contents of the bpatcher will be saved with the parent patcher.

If `@embed` is not enabled, the bpatcher references a patcher file specified in its `name` attribute. As with abstractions, changes to the original file will update every bpatcher that refers to that file. See abstractions for more detailed information on working with bpatchers that reference patcher files.

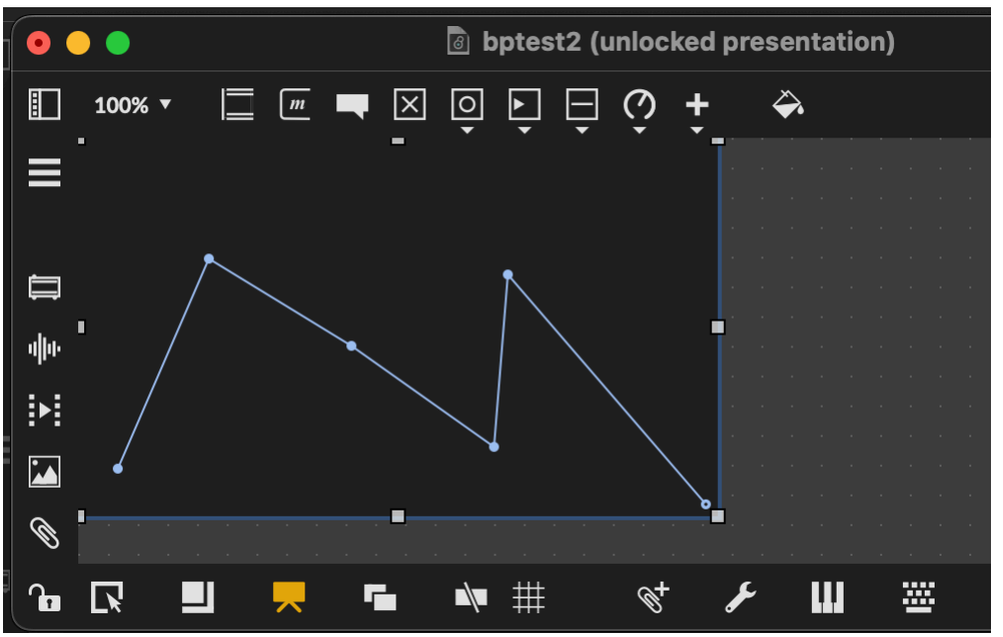
Patcher Appearance in bpatchers

As mentioned above, when loading a patcher file, the bpatcher will use the `openinpresentation` attribute of the patcher to determine whether to show it in patching or presentation mode.

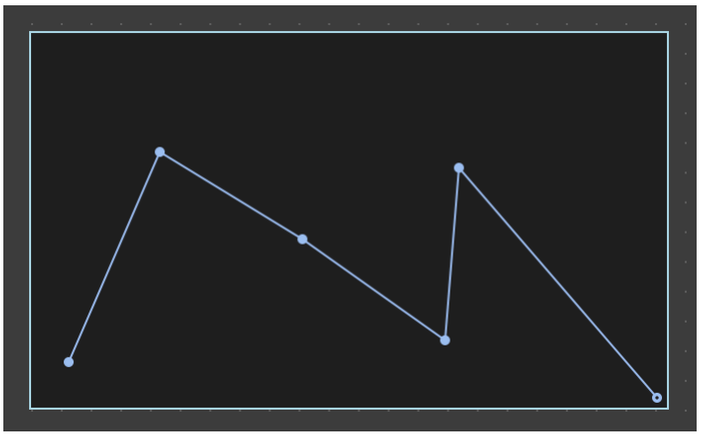
In addition, the patcher file can control the bpatcher's initial size and appearance when using the e command to create the object.

- To assign a default size for the bpatcher, use the `openrect` (*Fixed Initial Window Location*). The x and y coordinates of the openrect are ignored, but the width and height will be assigned to the bpatcher's initial width and height.
- If `openinpresentation` is enabled and `openrect` is not set, the enclosing rectangle for all objects that belong to the presentation will be used as the bpatcher's initial size.

As an example, this patcher contains a function object that has been added to the presentation.



When using the e command and typing this patcher file's name, the resulting bpatcher frames the function object as shown below.



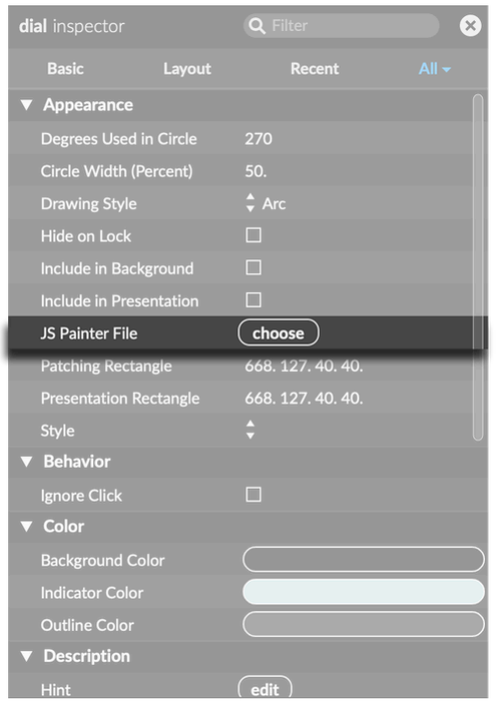
Custom UI Objects

Customizing Existing UI Objects 630
The jsui Object 637
MGraphics 640
Sketch 642
JSPainter vs jsui in Depth 644
See Also 644

In addition to creating a completely new object by writing your own Max External, you can also customize the appearance of Max objects using JavaScript. If you want to change the look of an existing user interface object, you can set its `@jspainter` attribute to a JavaScript file with your custom drawing code. If you want to create a UI object with custom behavior, you can use the jsui object, which combines the js object with a context you can draw to. Most custom drawing uses the **MGraphics** API for "painter's canvas" style drawing, and in fact this in the only drawing API available with `@jspainter`. With jsui you can use either **MGraphics** or the older **Sketch** API, which wraps a lower-level OpenGL 2.1 drawing context.

Customizing Existing UI Objects

To customize the appearance of an existing Max object, set the `@jspainterfile` attribute of that object to a JavaScript file containing JavaScript drawing code.



In that JavaScript file, you must implement the `paint` function to define the appearance of the object, where you can use the **MGraphics** API for drawing. A very simple implementation that replaces the existing paint function with a solid color would look like this:

```
function paint() {
  var viewsize = mgraphics.size;
  var width = viewsize[0];
  var height = viewsize[1];
  mgraphics.set_source_rgba(box.getattr("bgcolor"));
  mgraphics.rectangle(0, 0, width, height);
  mgraphics.fill();
}
```

Attributes and Value

When using `@jspainterfile`, call `box.getvalueof()` to return the **value** of the parent object. This lets you change your drawing based on the current value of the object, for example changing the length of a slider or the angle of a dial. This will always be an array value, so if the object has a single number as its value, it will be stored in an array of size 1.

```
// Custom drawing for a 'toggle' object
function paint() {
  var val = box.getvalueof(); // this is an array of size 1
  var isToggleOn = val[0] > 0;
  var viewsize = mgraphics.size;
  var width = viewsize[0];
  var height = viewsize[1];

  if (isToggleOn) {
    mgraphics.set_source_rgba(box.getattr("checkedcolor"));
  } else {
    mgraphics.set_source_rgba(box.getattr("uncheckedcolor"));
  }
  mgraphics.rectangle(0, 0, width, height);
  mgraphics.fill();
}
```

Mapping styles to attributes

Using the `box.attrname_forstylemap()` function, you can retrieve the name of the attribute for a given **Style** element. For a list of valid **Style** element names, check your Theme. For example, the style element names for the standard Max theme are

```
"bgcolor", "color", "elementcolor", "accentcolor", "selectioncolor",
"textcolor", "textcolor_inverse", "patchlinecolor", "clearcolor",
"locked_bgcolor", "editing_bgcolor", "stripecolor", "darkcolor",
"lightcolor", "bgfillcolor_type", "bgfillcolor_color1",
"bgfillcolor_color2", "bgfillcolor_color"
```

This can be useful if you want to look up an object's attribute name based on its function, rather than its internal name. For example, here's generic code for a custom, solid-color toggle, using colors from Max's current theme.

```
// Solid-color toggle, using theme colors
function paint() {
  var val = box.getvalueof(); // this is an array of size 1
  var isToggleOn = val[0] > 0;
  var viewsize = mgraphics.size;
  var width = viewsize[0];
  var height = viewsize[1];

  var onColorName = box.attrname_forstylemap("color");
  var offColorName = box.attrname_forstylemap("elementcolor");
  var oncolor = box.getattr(onColorName);
  var offcolor = box.getattr(offColorName);

  if (isToggleOn) {
    mgraphics.set_source_rgba(oncolor);
  } else {
    mgraphics.set_source_rgba(offcolor);
  }
  mgraphics.rectangle(0, 0, width, height);
  mgraphics.fill();
}
```

Calling the original `paint` function

The **mgraphics** API includes a method `parentpaint` that will call the original object's paint method. This lets you draw on top of (or under) the standard interface for the object. For example, here's a way to render a border on top of the standard native object UI using the underlying object's "elementcolor" style color:

```
function paint() {
  var viewsize = mgraphics.size;
  var width = viewsize[0];
  var height = viewsize[1];

  // call original object paint method
  mgraphics.parentpaint();

  // the actual underlying attribute name may be different
  // so we use the attrname_forstylemap() method to map
  // the style color to object attribute name
  var colorname = box.attrname_forstylemap("elementcolor");
  var bordercolor = box.getattr(colorname);

  // draw border rectangle over it
  mgraphics.set_source_rgba(bordercolor);
  mgraphics.rectangle(0.5, 0.5, width - 1, height - 1);
  mgraphics.stroke();
}
```

Note that calling `parentpaint` can recalculate custom automation dot positions, so any custom values to the automationpoint attribute should be set after calling `parentpaint`.

Hidden jspainter attributes

To make customizing parameter control objects more flexible, a number of object attributes can be set only from jspainter code. These attributes are not visible in the inspector.

Customizing the focus border

- `"jspainterfocus", -1 | 0 | 1` - read and write. Assuming the control is focussed / selected:
  - `-1` (default): uses default / legacy behavior:
    - For live.gain/slider/dial, the original focus border is always drawn, regardless of whether `parentpaint` is called.

    - For other objects, the original focus border is drawn only when `parentpaint` is called.
  - `0`: never draws the original focus border, even when calling `parentpaint`. Only works for live.gain/slider/dial.
  - `1`: always draws a focus border, even if `parentpaint` is not called. Note that this draws a standard focus border of four corners, always using the `focusbordercolor` color attribute, regardless of the object type or its `appearance` attribute.
- `"hasfocus", 0 | 1` - read only. Returns `1` if the user currently has this parameter control focussed / selected or `0` otherwise.

For example, this draws a heavy 4 pixels thick focus frame instead of the standard one:

```
// Don't draw the standard focus border, even when using parentpaint
box.setattr("jspainterfocus", 0);

function paint() {
  var focusBorderColor = box.getattr("focusbordercolor");
  var width = mgraphics.size[0];
  var height = mgraphics.size[1];

  mgraphics.parentpaint();

  if (box.getattr("hasfocus")) {
    // Draw extra heavy focus frame
    mgraphics.set_source_rgba(focusBorderColor);
    mgraphics.set_line_width(4);
    mgraphics.rectangle(2, 2, width - 4, height - 4);
    mgraphics.stroke();
  }
}
```

Customizing the automation dot behavior

Controls how the automation, modulation, and macro dots are shown when a parameter is automated, modulated or assigned to a macro knob.

- `"jspainterautomation", -1 | 0 | 1` - read and write.
  - `-1` (default): dots are drawn only when using `parentpaint`.
  - `0`: dots are never drawn, allowing the jspainter script to draw custom dots, even when `parentpaint` is used.
  - `1`: dots are always drawn, regardless of whether `parentpaint` is used.
- `"automationpoint", x, y` - read and write. Set the position of the upper-left corner of the dots.
  - Note: calling `parentpaint` can recalculate this position. For a custom dots position, make sure to update the `automationpoint` value *after* calling `parentpaint`.
- `isautomated, 0 | 1` - read only. Returns `1` if automation is applied to the parameter or `0` otherwise.
- `ismodulated, 0 | 1` - read only. Returns `1` if modulation is applied to the parameter or `0` otherwise.
- `ismacroassigned, 0 | 1` - read only. Returns `1` if the parameter is assigned to a macro knob or `0` otherwise.

Here is an example that, when applied to live.dial, draws the regular parameter visual only when the parameter is automated and also draws the automation dot on the right side of the control.

```
// Always draw the dots even if parentpaint is not called
box.setattr("jspainterautomation", 1);

function paint() {
  if (box.maxclass != "live.dial") {
    mpgraphics.parentpaint();
    return;
  }

  var width = mgraphics.size[0];

  // Only draw the live.dial's original content if it is automated
  if (box.getattr("isautomated")) mpgraphics.parentpaint();

  // Move the automation dot to the right side of the control, leaving room
  // for both the automation and the modulation dots, which are stacked
  // if both are present. Note: the macro dot, if present, hides both
others.
  box.setattr("automationpoint", width - 10, 10.5);
}
```

The jsui Object

Like the js object, jsui refers to a separate JavaScript file to define its behavior. In most ways, jsui is the same as a js object, except jsui also has a drawing surface that you can use to implement your own custom appearance.

The easiest way to implement custom drawing with jsui is using **mgraphics**. Call `mgraphics.init` somewhere in the global scope of your JavaScript file (typically near or at the top) and then make a `paint` function where you define your custom drawing. Max will call this function when the object needs redrawing, or after you call `mgraphics.redraw()`.

When using jsui, you also have access to the older **Sketch** API. This is really a completely separate drawing model, and while it's possible to use **sketch** at the same time as **mgraphics**, it's not intended. The **Sketch** API currently uses a legacy OpenGL 2.1 context, and so it supports low-level graphics functions, but not a very modern implementation. Unlike MGraphics, with **sketch** you redraw your object manually by calling `refresh()`.

Event handlers with jsui

The jsui object can also handle simple UI events, including mouse and resize events. To respond to these events, implement a function with a name like `on<eventname>` where `<eventname>` is the name of the event you want jsui to respond to. For example, to handle a mouse click event:

```
function onclick(x, y) {
  someOtherFunction(x, y);
  mgraphics.redraw();
}
```

The full arguments to any of the mouse events include `x, y, button, mod1, shift, caps, opt, mod2`, which will be the x, y position of the mouse click in the jsui object, the `phase` of the event, the state of the left modifier key, the state of the shift key, the state of caps lock, the state of the option key, and the state of the right modifier key. You can see full documentation for these functions in the jsthis JavasScript documentation The supported functions are:

```
/**
 * Receives initial click events.
 * @remarks
 * The "button" argument will always be 1.
 */
function onclick(x, y, button, mod1, shift, caps, opt, mod2) {}
```

```
/**
 * Receives double click events.
 * @remarks
 * The "button" argument will always be 1.
 */
function ondblclick(x, y, button, mod1, shift, caps, opt, mod2) {}
```

```
/**
 * Receives drag events.
 * @remarks
 * The "button" argument will be 1 while dragging, and 0 when dragging
stops.
 */
function ondrag(x, y, button, mod1, shift, caps, opt, mod2) {}
```

```
/**
 * Receives mouse events over the object.
 * @remarks
 * Equivalent to a "mouse over" event. The "button" argument will always
be 0.
 */
function onidle(x, y, button, mod1, shift, caps, opt, mod2) {}
```

```
/**
 * Mouse event as the cursor leaves the object boundaries.
 * @remarks
 * Equivalent to a "mouse out" event. The "button" argument will always be
0.
 */
function onidleout(x, y, button, mod1, shift, caps, opt, mod2) {}
```

There is one more event that will be received when the object resizes.

```
/**
 * Receives the new size of the object in width and height
 */
function onresize(width, height) {}
```

MGraphics

MGraphics uses a "painter's canvas" drawing model. You call functions that create shapes, paths, text, or other things to draw, and then fill or stroke them to add to the current canvas. Whether you're using MGraphics with jsui or the `@jspainter` attribute, you don't ever draw to the canvas directly. Rather, you implement a `paint` function, which Max will call whenever the object needs to be redrawn. You can also call `mgraphics.redraw()`, which will tell Max to redraw the object as soon as it can. A simple paint function might look like this:

```
// Implement this somewhere in your custom drawing code
function paint() {
  mgraphics.rectangle(-0.2, 0.2, 0.2, -0.2);
  mgraphics.fill();
}
```

Initialization

In the global scope of your JavaScript file, usually near the top, call `mgraphics.init()` to initialize your object for MGraphics drawing. This is also a good time to configure the MGraphics context globally—you could configure `mgraphics` for relative coordinates, or tell `mgraphics` not to fill shapes automatically when you use the `stroke()` functions by disabling `autofill`.

```
// Set up the object for mgraphics drawing
mgraphics.init();

// Optionally, configure the global mgraphics instance
mgraphics.relative_coords = 1;
mgraphics.autofill = 0;
```

Shape Drawing

Whether you're drawing a path, shape, or text, the formula for drawing with MGraphics is more or less the same:

- Set the colors and properties of the thing you want to draw.
- Create the path, shape, text, or other thing to draw.
- Call a function to execute the drawing.

So a drawing routine might look like:

```
function paint() {
  mgraphics.set_source_rgba(0.2, 0.2, 0.2, 1);
  mgraphics.set_line_width(0.03);
  mgraphics.move_to(-1.0, -1.0);
  mgraphics.line_to(1.0, 1.0);
  mgraphics.stroke();
}
```

This drawing function is using the relative coordinate system: the `move_to` call moves to the top-left of the drawing context, and `line_to` draws a line ending in the bottom-right.

Coordinate System

MGraphics supports two coordinate systems: relative and absolute. With absolute coordinates, the origin `(0, 0)` is the top-left of the canvas, and `(width, height)` would be the bottom-right of the canvas. If your display was `400` by `200`, you could draw an 'X' filling this area with code like this:

```
/**
 * Draw a cross using absolute coordinates. This shape will always be the
 * same size, no matter how the mgraphics context is resized.
 */
function paint() {
  mgraphics.move_to(0, 0); // the top-left
  mgraphics.line_to(400, 200); // the bottom-right
  mgraphics.stroke(); // draw it

  mgraphics.move_to(400, 0); // the top-right
  mgraphics.line_to(0, 200); // the bottom-left
  mgraphics.stroke();
}
```

Switch to the relative coordinate system by setting `relative_coords` to `1` on your MGraphics object. After enabling relative coordinates, the origin `(0, 0)` will be at the center of the canvas, the point `(-1, -1)` will be at the top-left, and `(1, 1)` will be at the bottom-right.

```
/**
 * Draw a cross using relative coordinates. This drawing will stretch to
 * fill the full width and height of the drawing context
 */
mgraphics.relative_coords = 1;

function paint() {
  mgraphics.move_to(-1, -1); // the top-left
  mgraphics.line_to(1, 1); // the bottom-right
  mgraphics.stroke(); // draw it

  mgraphics.move_to(1, -1); // the top-right
  mgraphics.line_to(-1, 1); // the bottom-left
  mgraphics.stroke();
}
```

Sketch

JavaScript files loaded by the jsui object have access to an instance of `Sketch` through the global `sketch` object, which can be used for drawing using the Sketch API. Sketch is an older API than MGraphics, wrapping a legacy OpenGL 2.1 drawing context. It is not the most modern way to implement custom interfaces with jsui or `@jspainterfile`, though it's still available if you need to use the low-level OpenGL functions that the API exposes.

The Sketch API provides high-level functions like `sphere` and `torus`, as well as low-level functions like `glclearcolor`. These lower level functions, all prefixed with `gl`, are thin wrappers around OpenGL functions. You can find the most complete, accurate documentation for these functions by checking the OpenGL documentation.

Colors and coordinates

Color values are floating point numbers in the range 0. to 1. Color support an alpha channel, with all colors in RGBA format. If an alpha value is not provided, it is assumed to be 1—totally opaque. To disable alpha blending, use `sketch.gldisable("blend")`. If you want to disable depth buffering, which can interfere with alpha blending, use `sketch.gldisable("depth_test")`.

Unlike some graphics APIs, the OpenGL API does not distinguish between 2D and 3D drawing. Conventional 2D drawing is simply a subset of 3D drawing calls with specific graphics state--e.g. no lighting, no depth testing, orthorgraphic projection, et cetera. High level utility methods are provided as a convenience to setup up the OpenGL graphics state to something typically used for 2D or 3D graphics. If assuming 2D drawing conventions, one can ordinarily use z coordinates of zero for all methods that require them.

Coordinates in OpenGL are also given in terms of floating point relative world coordinates, rather than absolute pixel coordinates. The scale of these world coordinates will change depending on the current graphics transformation--i.e. translation, rotation, scaling, projection mode, viewport, etc. However, our default mapping is that Y coordinates are in the range `-1`. to `1` from bottom to top, and X coordinates are in the range `-aspect` to `aspect` from left to right, where `aspect` is equal to the ratio of `width/height`. In the default case, `(0,0)` will be center of your object, `(-aspect,1.)` will be the upper left corner, and `(aspect,-1.)` will be the lower right corner.

User events like mouse clicks will be in absolute screen coordinates. Use `sketch.screentoworld()` and `sketch.worldtoscreen()` to convert between OpenGL world coordinates and screeen coordinates.

OpenGL conventions and differences

- Sketch methods are all lowercase, so the OpenGL function `glBegin` is wrapped by the Sketch function `glbegin`.
- Instead of using symbolic constants like `GL_LIGHTING`, Sketch simply uses lowercase JavaScript strings without the `GL_` prefix. So instead of `GL_LINE_STRIP`, use `"line_strip"` with Sketch.
- Sketch doesn't have special vector versions of its functions, and only floating point numbers are supported. So `glColorv4fv()` is simply `sketch.glcolor()`.
- Sketch functions can take arrays as well as individual arguments, so `sketch.glcolor(0.5, 0.5 0.5)` and `sketch.glcolor([0.5, 0.5, 0.5])` are both supported.

JSPainter vs jsui in Depth

When using `@jspainterfile`, all of the underlying logic of the object is still handled in native Max C code. Your JavaScript drawing function changes how the object appears, but nothing else. Like a standard Max object, message scheduling happens in the high priority scheduler thread, and painting happens in the low priority application thread. The jsui object has the same limitation as the js object: all object logic is executed in the low priority application thread. However, with jsui you're also free to change the object's behavior, just like with the js object.

JSPainter limitations

JSPainter is powerful but has some limitations. While drawing with JSPainter, any use of the **Task** object to support timing information in the UI is not supported. Also, objects with text fields will have some issues. Text may be rendered with standard mgraphics API calls, but as is the case with JSUI, there is no automatic linewrapping, and if the underlying object has a textfield (as do number boxes, message objects, comment objects and the standard text object), there will be conflicts or missing text. For this reason the jspainterfile attribute is hidden for such objects. And finally, there is no support for `@autowatch` or double clicking to open the `jspainterfile` in a text editor.

See Also

- Max JS API

Externals

Installing Externals 645
Security 646
Developing Externals 646
Resolving Errors and Troubleshooting 647

Under the hood, Max objects are basically small programs. The essence of Max is to create new behaviors by connecting these small programs together. Because of this modular design, third parties can extend the functionality of Max by programming their own objects. These objects are called **Externals**, since they're authored *externally* to the main Max program.

Externals are like audio plug-ins for a DAW, they allow Max to be extended to support new objects at any time. Max actually ships with many externals and many more are available for free on the Internet.

Externals are bundles of executable code, stored in `.mxo` files on non-Windows operating systems, and `.mxe64` files on Windows. When an object receives a message, or when Max calculates a vector of audio data, it calls a function inside each object to perform the necessary computation.

Installing Externals

In order to install an External, simply put the `.mxe64` or `.mxo` file into Max's Search Path. When Max launches, it scans its Search Path for any External files. If it finds any, it automatically adds them to internal database. Those objects will appear in Max's new object autocomplete, and you will be able to create a new object from any dicovered externals.

> More complex externals may have dependencies that will need to be installed as well. For example, an external might rely on a Dynamic Library or `dll` to function. In general, follow the instructions provided by the external author to install correctly.

If the external you want to install is part of a Max Package, then you don't need to do anything special to install it. The external wil be included automatically along with the rest of the package. If you're trying to install an external on its own, you can put it anywhere in the current Search Path. The `Library` folder, found at `%HOMEDRIVE%%HOMEPATH%\Documents\Max 9\Library` on Windows, and at `~/Users/Max 9/Library` on non-Windows operating systems, is a generic folder for storing files that you want to be included in Max's Search Path. It can be a useful place to store externals if you'd like them all to be in one place.

Security

> Externals are programs. Just like you wouldn't download and run a program from someone you don't trust, don't install any externals from untrusted sources either. Also, keep in mind that externals run as part of the main Max program. Whatever permissions you give Max, your externals will have those same permissions.

Developing Externals

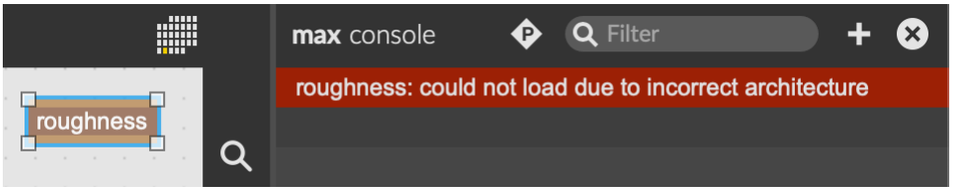
In order to communicate with the Max application, an external needs to tell Max what messages it responds to, how many inlets and outlets it has, whether it processes audio, and much more. All of this is accomplished using the Max SDK, a library of functions that can program can use to register as a Max external.

You can get the SDK, and learn to write your own externals, using the SDK Documentation.

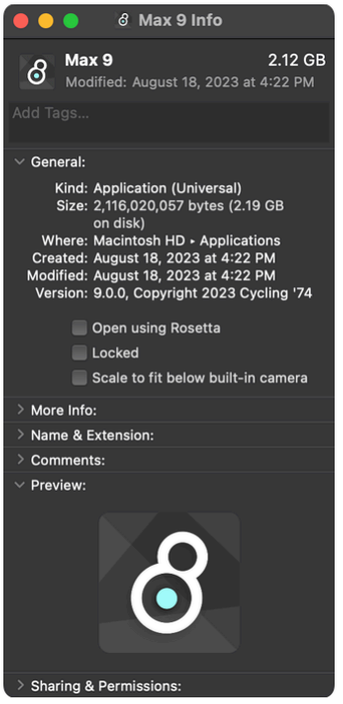
In addition to the C SDK, there is also a C++ SDK called Min. This version of the SDK can streamline writing Max externals, and supports some advanced features not readily available to the C SDK. You can learn more about the Min SDK from the Min SDK Documentation.

Resolving Errors and Troubleshooting

When developing or installing new externals, you may see a message like the following when you try to load your external.



This message, "could not load due to incorrect architecture", means that Max recognized the `.mxo` or `.mxe` file as an external, but could not execute the code inside that object because it was compiled for a different hardware architecture. This can happen when trying to use a 32-bit external while running a 64-bit version of Max, or when trying to use an external compiled for an Intel architecture while runnning an ARM (Apple Silicon). Usually the best solution is to find a version of the external that's compiled for the same architecture that you're trying to use (or to rebuild your own external with new build settings). However, on MacOS, it's also possible to run Max using Rosetta, which will let you use externals compiled for intel architecture. To do so, first quit Max, then right click on the application, choose `Get Info`, and then enable "Open using Rosetta".


Max 9 Info
Max 9
2.12 GB
Modified: August 18, 2023 at 4:22 PM
Add Tags...
General:
Kind: Application (Universal)
Size: 2,116,020,057 bytes (2.19 GB on disk)
Where: Macintosh HD ▸ Applications
Created: August 18, 2023 at 4:22 PM
Modified: August 18, 2023 at 4:22 PM
Version: 9.0.0, Copyright 2023 Cycling '74
Open using Rosetta
Locked
Scale to fit below built-in camera
More Info:
Name & Extension:
Comments:
Preview:
Sharing & Permissions:

Packages

Packages Folder 649
Package Folder Structure 650
package-info.json 652
The Left Toolbar 656
Submitting a Package 657

Packages are a convenient means of bundling objects, media, patchers, and resources for distribution. Most of the time, if you're just looking to manage resources specific to a project, then a **Project** is probably what you're looking for. However, if you'd like to share your work more broadly, especially if you're building a reusable set of tools to submit to the **Package Manager**, then packages are right choice.

A package is simply a folder adhering to a prescribed structure and placed in the appropriate folder. When Max launches, it will iterate through these packages and load them, making their resources available for use.

Packages Folder

If a package is available in the Package Manager, then you should use the package manager to install and uninstall that package. However, you can also install packages yourself, which can be useful if you want to use a package that's not currently in the package manager.

You can install a package by copying the package folder into Max's *packages* folder. The user-specific packages folder is at `%HOMEDRIVE%%HOMEPATH%\Documents\Max 9\Packages` on Windows and `~/Documents/Max 9/Packages` on macOS. The other (system wide) location is in `/Users/Shared/Max 9/Packages` on MacOS, and `%HOMEDRIVE%\ProgramData\Max 9\Packages` on Windows.

You can uninstall a package by deleting it from the packages folder.

Package Folder Structure

The name of each folder in a Max package determines how Max will load the files in that folder. Some of these folders will also be automatically included in Max's Search Path.

Folder	Search Path	Description
clippings	✅	Patchers to list in the "Paste From..." contextual menu when patching
code	✅	Gen patchers
collections		Collections to list in the File Browser that are associated with the package
default-definitions		Definition info for Object Defaults support in UI externals
default-settings		Saved color schemes for Object Defaults
devices	✅	Max for Live devices (AMXDs)
docs	✅	Reference pages and Vignettes to be accessible from the Documentation Window
examples	✅	Example patchers and supporting material
extensions	✅	Special external objects loaded on Max launch
externals	✅	External objects
extras	✅	Patchers to be listed in the "Extras" menu
fonts	✅	Custom fonts available to Max when the Package is present
help	✅	Help patchers and supporting material
icon.png		A PNG graphic file (500x500px) for display in the Package Manager
init		Text files interpreted by Max at launch

interfaces		Supporting files for objects to display in the top patcher toolbar and other Max integration.
java-classes	✅	Compiled Java classes for use in mxj/mxj~. Place .jar folders in a 'lib' subfolder.
java-doc		Documentation for Java classes
javascript	✅	Javascript files to be used by js
jsextensions	✅	Extensions to JS implemented as special externals or js files
jsui	✅	Javascript files to be used by jsui, and listed in the contextual menu for jsui
license.txt or *license.md*		Terms of use / redistribution of your package, in plain text or markdown
media	✅	Media files to be included in the searchpath
misc	✅	Anything
patchers	✅	Patchers or abstractions to be included in the searchpath
projects	✅	Projects to be included in the searchpath. Note that only the project file will be added to the searchpath.
object-icons	✅	An SVG-format object icon for a particular Max object (named <objectname>.svg), used in the Object Browser
object-prototypes		Object Prototypes will be listed in the contextual menu for a selected UI object
readme.txt or *readme.md*		Information about your package, in plain text or markdown
snippets		Snippets associated with this package
source		Source code for external objects, ignored by Max
support		Special location for DLL or dylib dependencies of external objects. Added to the DLL search path on Windows.
templates		Template patchers to be listed in the "File > New From Template" menu

package-info.json

The **package-info.json** file is the manifest of your package. It includes metadata about your package like its name and description, as well as including a list of important files in your package. If you're not familiar with the JSON format, it's a common format structured text built around numbers, strings, arrays and dictionaries.

name

The name of the package. Should be unique among packages. This doesn't affect how the name of the package will appear in Max.

displayname

This is the name of the package as it will appear in Max, both the Package Manager as well as other places.

version

A semantic versioning compatible string.

```
{
  "version": "0.6.0"
}
```

author

Author string, for a single author.

authors

Array of author strings, for multiple authors

```
{
  "authors": ["Arthur Author", "Otter Auter"]
}
```

description

Brief description. It's a good idea to list any dependencies here, or additional relevant information.

tags

An array of strings containing developer-defined tags, if any.

```
{
  "tags": ["audio", "DSP", "visualization"]
}
```

website

If the package or author has a website, note it here.

extensible

An integer (0 or 1) to indicate whether this package can be extended by another package. Package extension means that the extending package becomes logically part of the extensible package in terms of how it appears in the File Browser and Reference Window. For instance, the RISE package extends the extensible BEAP package. As such, RISE patchers and documentation are included in the BEAP package entries in the File Browser and Reference.

extends

If the package *extends* an *extensible* package, note that package name here.

max_version_min

The minimum Max version this package supports (or "none" if there is no specific minimum version).

```
{
  "max_version_min": "8.0.0"
}
```

max_version_max

The maximum Max version this package supports (or "none" if there is no specific maximum version).

```
{
  "max_version_max": "none"
}
```

os

Optionally restrict the installation of this package to certain platforms or architectures.

```
{
  "os": {
    "macintosh": {
      "min_version": "none",
      "platform": ["x64", "aarch64"]
    },
    "windows": {
      "min_version": "none",
      "platform": ["x64"]
    }
  }
}
```

The info above would indicate that this package can run on 64-bit Intel Windows and macOS computers, as well as 64-bit Apple Silicon-based computers (`aarch64`). `"x32"` would indicate 32-bit compatibility, although recent Max versions no longer support 32-bit OSs.

Note that the `min_version` fields on Windows use some specific strings to indicate specific OS versions: `"none"` , `"11"` , `"10"` , `"8.1"` , `"8"` , `"7"` and `"7SP1"` are the only ones which are relevant for modern Max.

On macOS, the `min_version` field should be a Semantic Versioning string matching a macOS version (e.g. `"10.5.2"` or `"12.5"`).

homepatcher

If the package has a "landing patcher" in its `patchers` folder, note the patcher name here. This will be the package that opens when you click the *Launch* button in the package's detail page in the Package Manager.

```
{
  "homepatcher": "name of the home patcher.maxpat"
}
```

toolbar_icon

This lets you identify an SVG in your package that will be used as the package's icon in the left toolbar. Read more about this below.

package_extra

Optional fields. You can put anything you want here, but Max won't use these values. The one exception is the `forcerestart` key.

package_extra.forcerestart

Set to 1 if you would like Max to prompt the user to restart after installing your package from the Package Manager.

```
{
  "package_extra": {
    "reverse_domain": "com.cycling74",
    "copyright": "Copyright (c) 2022 Cycling '74",
    "forcerestart": 1
  }
}
```

filelist

While you may see this field in some `package-info.json`, it's automatically generated and not used for most packages. You should not include this property in your own `package-info.json`.

c74install

You may also see this property in installed packages, but do not add this to your `package-info.json` file. It is automatically added by the Package Manager when installing packages.

installdate

This is another field that is managed by the Package Manager. Do not add this to your `package-info.json` file.

The Left Toolbar

If your package includes clippings or **snippets**, those will be shown in the *Modules* menu in the left toolbar. The user can also add a dedicated icon for your package by right-clicking on an empty space in the left toobar and adding your package. You can customize the appearance of this icon by adding an appropriate SVG file to your package.

You can point to this icon file in your package's `package-info.json` file, using the `"toolbar_icon"` field. For example,

```
{
  "toolbar_icon": "my_icon.svg"
}
```

Alternatively, if no "toolbar_icon" is defined in `package-info.json`, you can name the file `<packagename>_toolbar.svg` where `<packagename>` is case-sensitive and must match the name of your package, and Max will automatically use that file as the toolbar icon.

> If you use this second method to name your icon, and your package name contains a space, then the icon file should use underscores instead of spaces. For example, if your package is named "My Package", then the icon file would need to have the name My_Package_toolbar.svg.

In either case, the SVG file should be placed in some subfolder of your package which is included in the search path -- we recommend the `misc/` folder.

If you do not include an icon in this way, a two-letter icon will be auto-generated from your package name.

Submitting a Package

If you'd like your package to appear in the offical Package Manager, you should share it with us. Find the submission form on our website at https://cycling74.com/support/submit-packages.

Package Manager

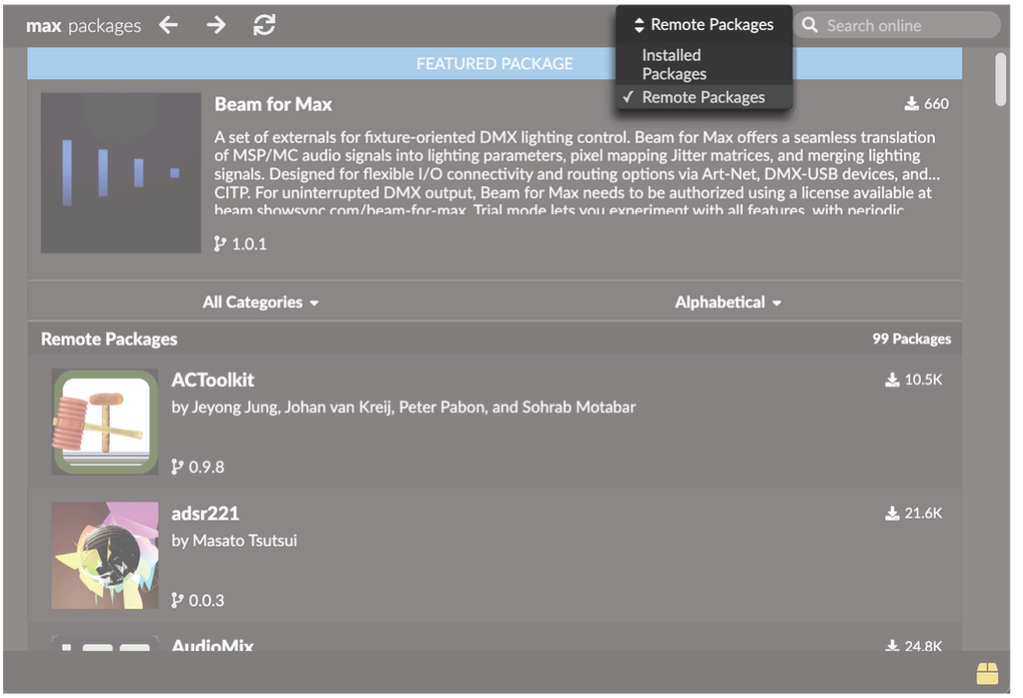
Browsing and Installing Packages 658
Managing Installed Packages 660
Package Install Location 660
Package Updates and Versions 661

The **Package Manager** provides instant access to a regularly updated, curated selection of Max add-on content and tools. The Package Manager allows you to manage which **Packages** are currently installed in Max. From the package manager you can install new packages, enable/disable existing packages, update or downgrade installed packages, and launch the default patcher for a given pacakge.

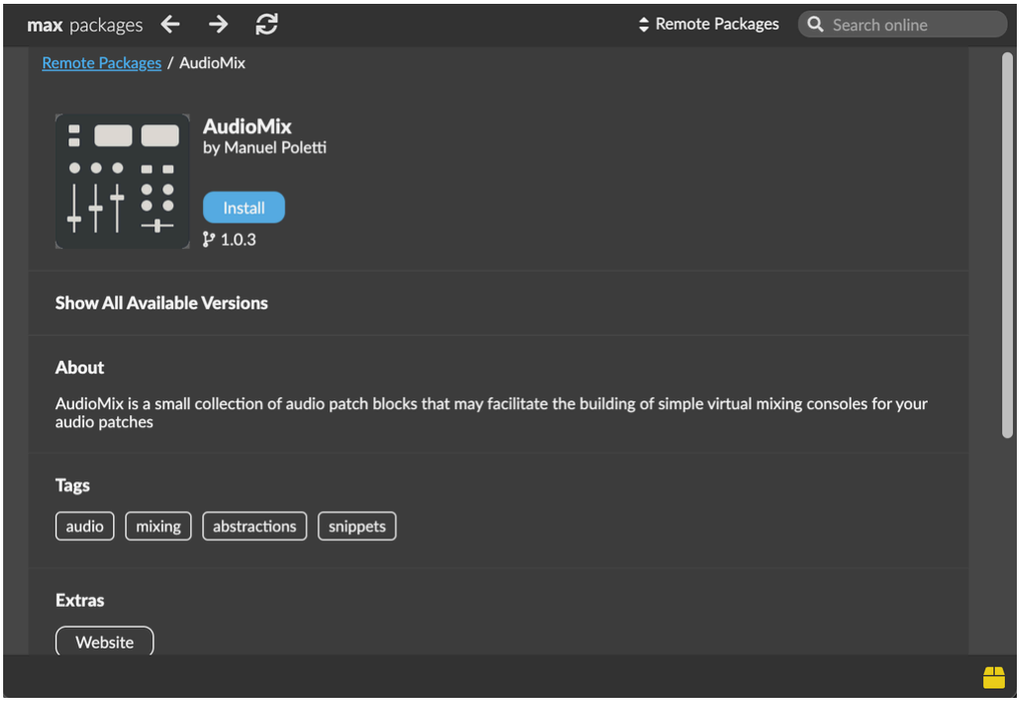
To access the Package Manager, select *Show Package Manager* from the *File* menu.

Browsing and Installing Packages

By default, the package manager opens in the *Browse Remote Packages* view. letting you browse through packages that you do not currently have installed. You can instead view the list of currently installed packages by selecting *Installed Packages* from the drop-down menu at the top of the window.



Click on a package to inspect its details. This will show you information such as system requirements, a description of the package, and a link to the package author's website.



To install a package, click the blue *Install* button. Once a package is installed, click the *Launch* button (if available) to view the Launch Patcher for the package. The *Show in Filebrowser* button will open the package in the File Browser, letting you see all the files included in the package.

Managing Installed Packages

At the top of the package manager window, click the drop-down menu and select "Installed Packages". This will list all installed packages, including both those you installed through the package manager, as well as those you might have added by dropping them in the *Packages* folder.

To disable a package without deleting it entirely, click the *Disable* button after selecting the package. Max will ignore any files in a disabled package, but the package will remain downloaded and installed.

> Many packages, including any packages that contain externals, will require a restart of Max before changes take effect.

You can remove a package entirely by clicking the red *Uninstall* button.

Package Install Location

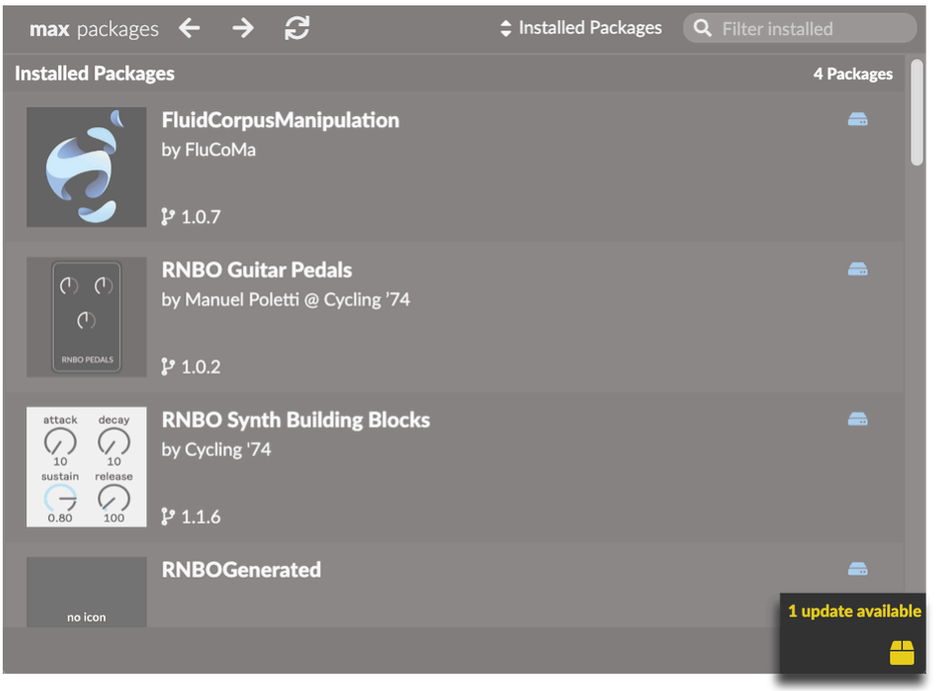
All packages are installed in your *User Packages Folder*, located at `%HOMEDRIVE%%HOMEPATH%\Documents\Max 9\Packages` on Windows, and at `~/Users/Max 9/Packages` on macOS. You can install packages that aren't available in the package manager, including your own custom packages, by dropping them in this directory.

Note that if you install a package that is itself available on the Package Manager by dropping them in this directory, the Package Manager will warn of a conflict. We recommend installing a package via the Package Manager if available in order to stay informed of updates to that package.

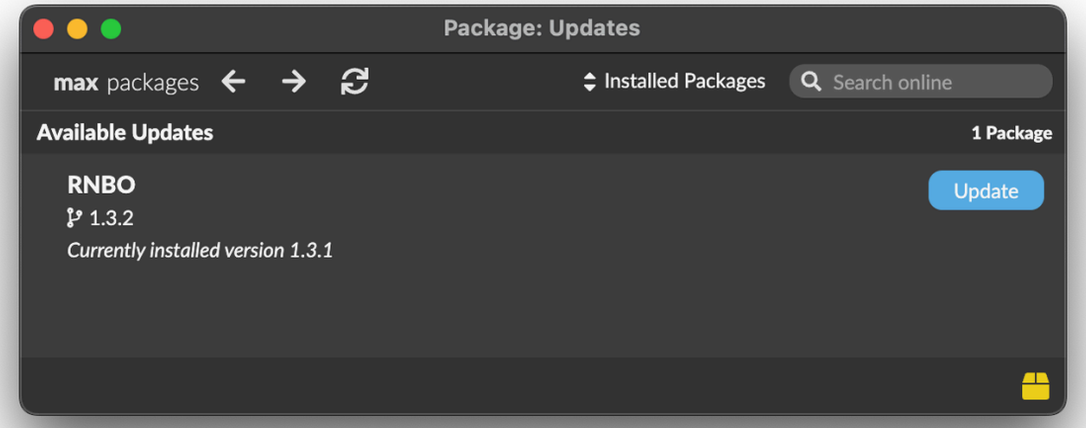
> Don't change the folder names for any packages installed using the package manager.

Package Updates and Versions

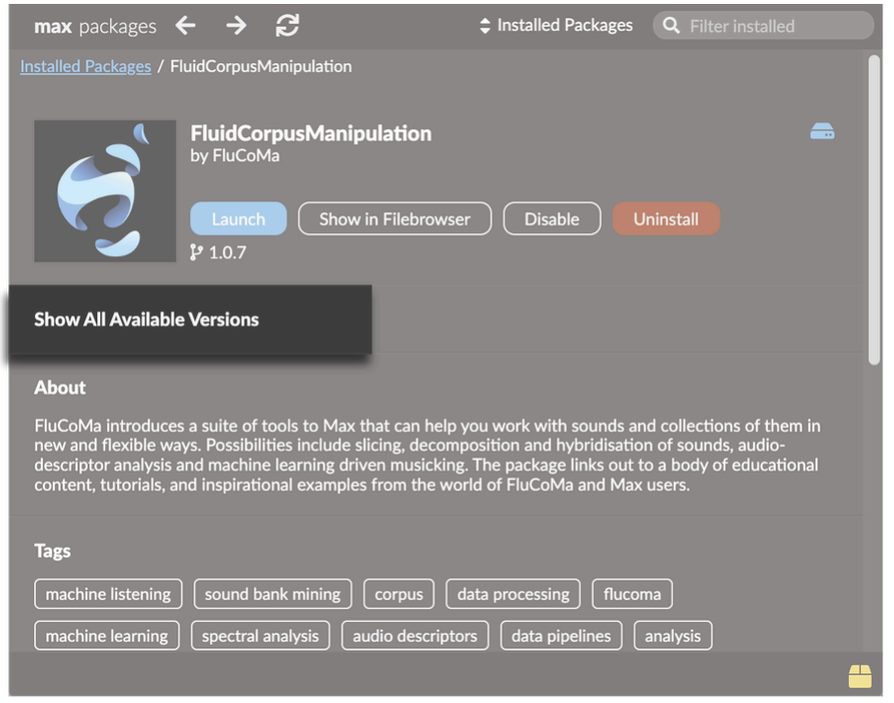
The *Package Update* icon in the bottom-right of the package manager window will highlight if there are updates available for any installed packages.



Click on the *Package Update* icon to show all packages with an available update.



You can change the installed version of any package by clicking "Show All Available Versions" in the package detail view. If you have a self-installed Package that conflicts with a Package available in our online library, you will have the option to overwrite it with our version or ignore it and leave it unchanged.



> The Package Manager itself is automatically updated with each version of Max. Occationally, you may be prompted to update the Package Manager when you open the window if an update is available.

Projects

Creating Projects 664
Project Window 665
Project Inspector 666
Editing Files in a Project 667
Project Search Paths 668
Open Actions 669
Max for Live Device Projects 669

Projects collect and organize dependencies associated with a patcher or set of patches. These dependencies can include patches, media files, code, and third-party externals.

Projects also extend the main Max **search path**. All files in a project can locate other files in the same project.

- Project-specific elements are loaded before any other files on the Max search path.
- Project-specific assets don't have to be added to the main Max search path, isolating them from other patches and Projects.
- You can switch between global and Project-local assets.
- Using projects lets you reduce the number of folders in Max's global search, which can improve patcher loading and editing speed.

Creating Projects

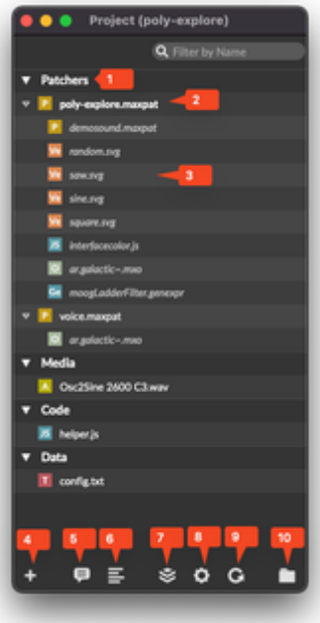
You can start with a new empty project, or create a project from an existing patcher.

- To create a new empty project, choose *New Project* from the *File* menu. This action creates a Max Project at the location specified. By default, this is `~/Documents/Max/Projects` (Mac) or `(User Folder)\My Documents\Max\Projects` (Win). Projects can be saved anywhere, but it is recommended to use the default location.

- To create a project from your existing open patcher, choose *Save as Project* from the *File* menu. This will close and re-open your patcher as a project.

> Once the project is created, you don't need to save it: projects are saved automatically as you edit them.

Project Window



1. Category header: Files in your project are automatically assigned to a category, like "Patchers", "Media", "Code", or "Data". Categories are only shown if there are matching files in the project.
2. Patcher opened with project: A project may contain one or more "top-level patchers". These are displayed in bold in the *Project* window, and are automatically opened when the project is loaded.
3. Dependencies: Files local to a project are shown in regular, non-italic text. Dependencies of project files are shown in a darker color, and if those files are only found on the global Max

search path, they are shown in italics. Dependencies cannot be removed from a project from the *Project* pindow.

4. Add files to Project: To add files to a project, they can be created from scratch, patchers can be created from templates, and existing files can be added to projects. Files can be removed by right-clicking on the file name in the *Project* window and selecting "Remove from Project." Alternatively, you can highlight the name and use the backspace or delete key on your keyboard.
5. Details view: When toggled on and a file from the list is selected, details about the file are shown, including the name, kind and absolute path of the file. If the file is a dependency (whether explicit or implicit), a link is shown to highlight the patchers in the file list which use it. Also shown is whether the file is opened with the project or not.
6. Hierarchical and flat view modes: Hierarchical view lists all dependencies underneath the file which depends on them. Flat view sorts all files, whether dependencies or not, into a flat list.
7. Manage project: From the *Manage Project* menu, you can

    - consolidate and deconsolidate. Consolidation copies all dependencies to the local project folder. De-consolidation removes any files which can be found on the global Max search path from your local project directory.
    - archive. Similar to consolidation, archiving creates a copy of all files needed by the project. Unlike consolidation, archiving creates a new ".maxzip" archive file that can be saved and distributed as a single file.
    - build collectives and applications.
    - export as a Max for Live device.

8. Project settings: Open the *Project Inspector* window, the *Project Seach Path* window, or the *Open Actions* text window.
9. Reload Project: Refreshes the *Project* window.
10. Show Project folder: Opens the project's location on disk.

Project Inspector

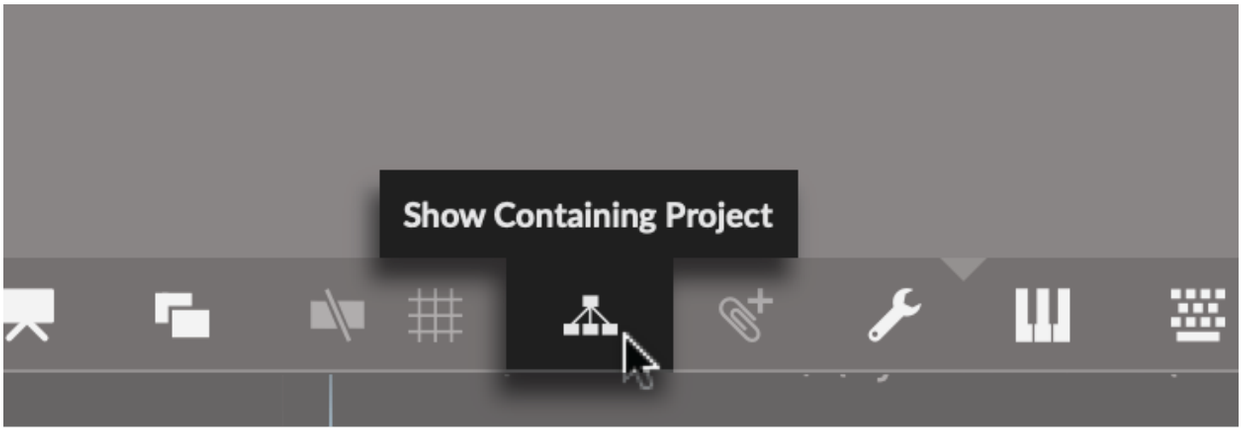
The Project Inspector allows you to set preferences for the project's behavior.

- Always Localize Project Items: When enabled, files added to a project will always be copied to the project's folder before being included in the project. (default = off)
- Hide Project Window After Opening: When enabled (and only if the project contains patchers which are marked "Open on Project Load"), the project window will not be visible after the Project opens. You can open the *Project* window later by click on the *Show Containing Project* button in the patcher's toolbar. (default = off)
- Keep Project Folder Organized: When enabled, files in the project's folder are automatically sorted into appropriate subfolders based on the file's category. (default = on)
- Show Patcher Dependencies: When enabled, implicit dependencies are displayed in the *Project* window. (default = on)
- Development Path Type: When enabled, allows you to specify the path type for the folder specified in the Development Path setting. Options are disabled, relative, and absolute.
- Development Path: Allows you to choose a project-specific folder for development of Max for Live projects. When set, all device files for the project will be saved to and referenced from this location. When disabled, the project uses the global folder `~/Documents/Max 8/Max for Live Devices` (Mac) or `(User Folder)\My Documents\Max 8\Max for Live Devices` (Win).

Editing Files in a Project

To edit a patcher or text file from a project, double-click on the file's name in the project window. Alternatively, right-click on the file's name in the project window and choose Open from the contextual menu.

When editing a patcher from a project, a new icon will appear in the bottom toolbar. You can use this icon to return to the project window from the patcher.



> If you edit a patcher by opening the patcher file directly, rather than opening the parent project first, Max won't be able to use the project context to resolve any *search path* dependencies. Some dependencies may be found in the wrong location (globally rather than project-locally), and some may not be found at all. For this reason, if you're working with a project, be sure to open the project before editing any child patchers.

Project Search Paths

Within a project, patchers and other kinds of files may be marked either global or local. Files marked global tell Max to use the file found on the global search path, if available. Files marked local tell Max to use the file found on the Project's search path first, if available. You also have the ability to publish a file to the global search path or to localize a file to your project's search path.

The project-local search path consists of:

1. Project-local folders: Projects maintain a folder on your hard drive. The contents of this folder (and its subfolders) are preferentially searched when Max is looking for files requested by project members. New files added to a project are created inside this folder by default, and you can manually add files to it, too.
2. Singleton folders: Projects can reference files which are neither in the Max global search path, nor in the project folder. When searching for files, the folders containing these "singletons" are searched (non-recursively), as well.

3. Project search paths: Projects also maintain a list of additional folders to be searched when locating project assets. This list works similarly to the list found in Max's File Preferences... window, but is used only by the project.

Dependencies are always presumed to be local. If no local version can be found, the global version will be used.

Open Actions

Project Open Actions allow you to specify what Max configuration options are loaded when apProject opens.

In the Open Actions text edit window, you can write various messages to Max. For example, messages such as `;dsp takeover 1` or `;max overdrive 1` will enable Scheduler in Audio Interrupt and Overdrive, respectively. When the project is loaded (or reloaded via the *Reload Project* button in the *Project* toolbar), the Open Actions will be executed.

Max for Live Device Projects

All Max for Live devices are projects. There are a few differences between regular Max projects and Max for Live device projects:

- The default location for Max for Live Device Projects is `~/Documents/Max 8/Max for Live Devices` (Mac) or `(User Folder)\My Documents\Max 8\Max for Live Devices` (Win)
- Unfreezing a device will unpack the device's assets to the project's folder, rather than to the "Unfrozen Max Device Files" folder on the Desktop.
- Conflicts are now auto-resolved in favor of the Device version (although you can override this in the Resolve Conflicts window if you need to).
- The Archive... item in the *Manage Project* toolbar menu creates a datestamped ".zip" archive of a frozen copy of the current device.
- Freezing a device will include all files listed in the *Project* window (both explicit and implicit items).

Freezing a device which utilizes 3rd party externals will automatically collect and freeze the externals for other Max platforms if they can be found in Max's (or the project's) search path. For instance, freezing a device which uses `myextern.mxe64` on Windows will include `myextern.mxo` (macOS) externals if they are available. None of these need to be added explicitly to the project.

Prototypes

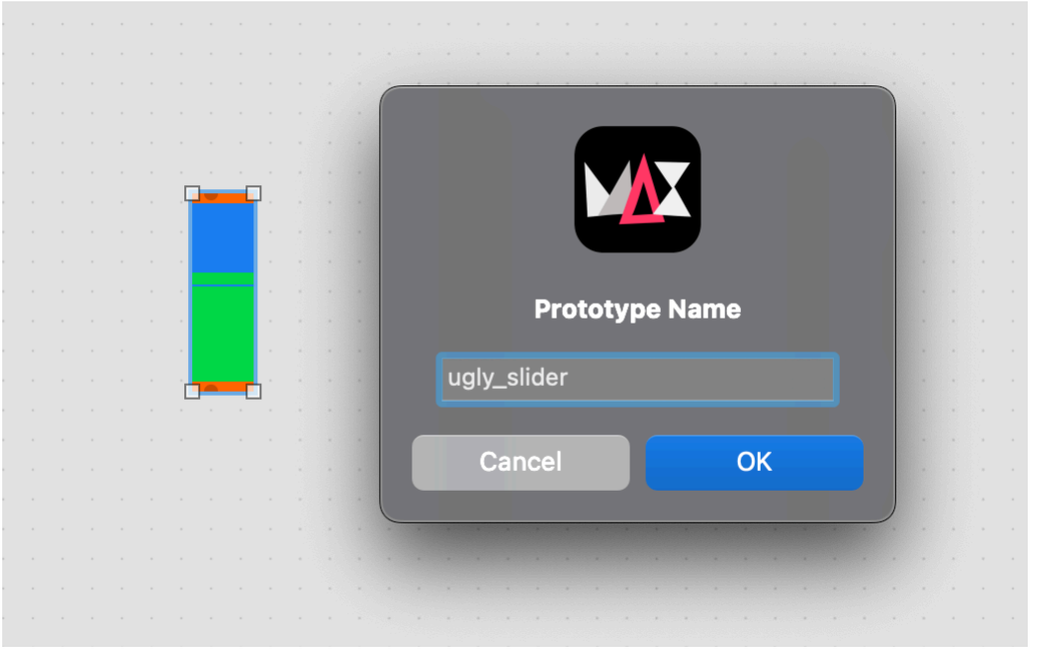
Defining a Prototype 671
Applying a Prototype 672
Deleting a Prototype 672
Prototypes in Packages 672

Prototypes are configurations of object attributes that can be saved and recalled later. For example, you might have an object like a slider that you've set up to have a certain size and to look a certain way. You could define a prototype from that slider, and then apply that prototype to make other sliders look the same way.

> Prototypes are mostly used with UI objects, but you can use them for generic Max objects as well. For example, you might have a Jitter object with a complex state that you find yourself reusing often.

Defining a Prototype

With an object selected, open the *Object* menu and select `Save Prototype...` to define a new prototype. A dialog box will appear, allowing you to give the prototype a name.



Defining a new prototype

Applying a Prototype

Select the object to which you'd like to apply your prototype. Open the *Object* menu and choose `Prototype > <name>`, where `<name>` is the name of your saved prototype. The object should change its text and/or appearance to match your saved prototype.

Deleting a Prototype

Max prototypes are stored in files with the `.maxproto` extension. You can find these files the `Prototypes` folder in the Max 9 Folder, which on macOS is in `~/Documents/Max 9` and on Windows is in `%USERPROFILE%\Documents\Max 9`. If you delete a `.maxproto` file from the `Prototypes` folder, it will no longer appear in the list of prototypes for the specified object.

Prototypes in Packages

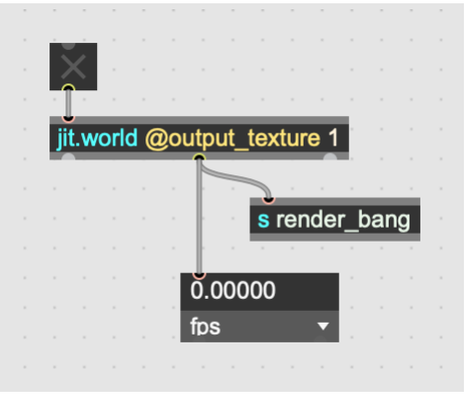
To include a prototype as part of a Package, create a folder in your package directory named `prototypes`, and put any `.maxproto` files that you want to include into that directory. When a

user installs your package, those prototypes will be available to them as well. This can be useful if your package has a consistent look and feel that you want to enable other users to reproduce.

Snippets

Creating Snippets 674
Using Snippets 675
Adding Snippets to a Patcher 675
Snippet Location 677
Using Snippets with Packages 677

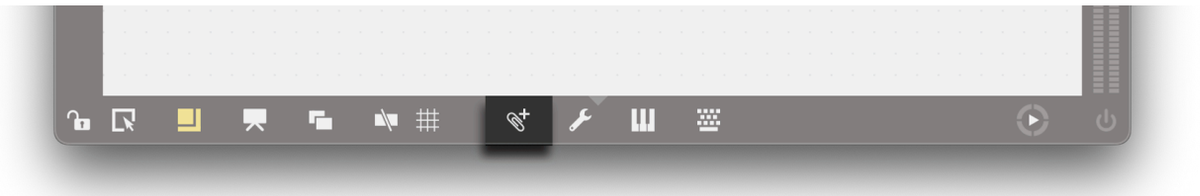
A **Snippet** is a small group of objects, saved in a special format for easy reuse. You can create snippets from parts of a patcher that you find yourself using often, saving you the trouble of having to recreate the objects and connections from scratch.



A snippet for a basic video display patch.

Creating Snippets

Select the objects you'd like to include in a snippet. Click the *Save Snippet* button in the bottom toolbar.



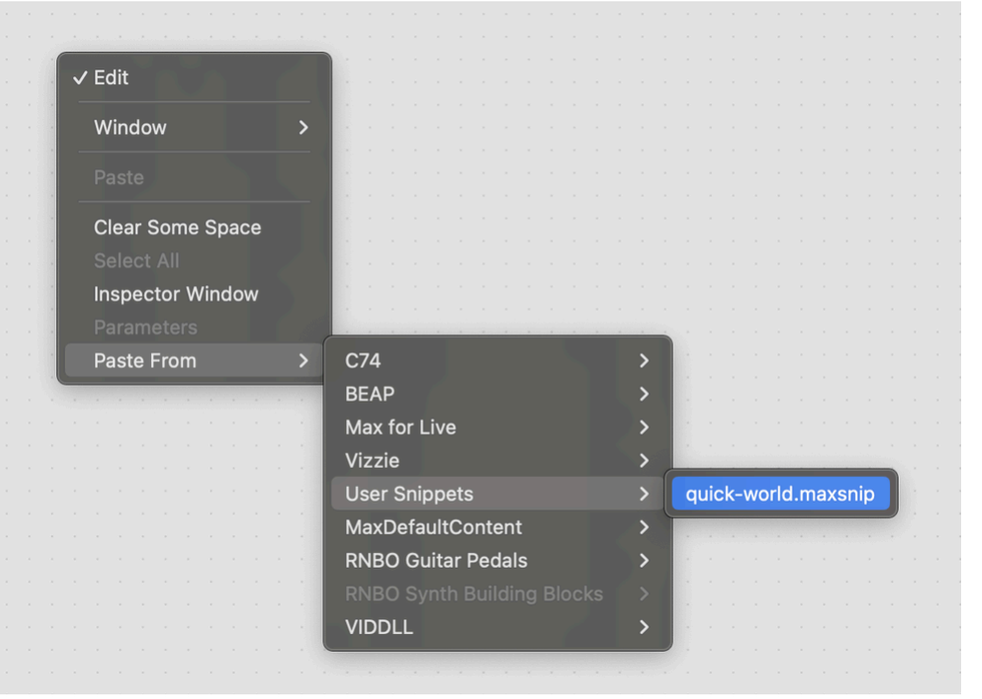
Using Snippets

A dialog will appear, letting you name your snippet. Once you've saved your new snippet, you'll be able to add it to any patcher either using the contextual menu or using the left sidebar.

Adding Snippets to a Patcher

Via the contextual menu

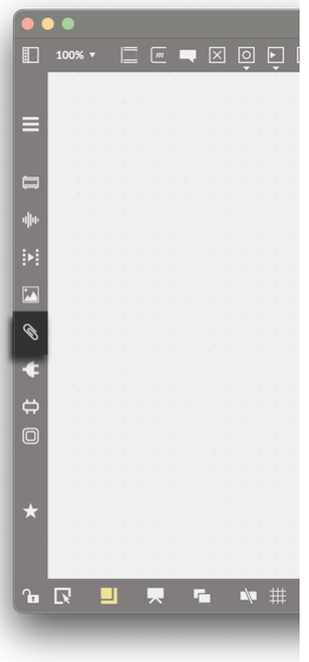
Right- or control-click an unlocked patcher where you'd like to paste your snippet. A contextual menu will appear. Select `Paste From > User Snippets` then choose the desired snippet from the submenu.



> The `Paste From` menu also lets you quickly add patchers and snippets from installed packages.

Via the left sidebar

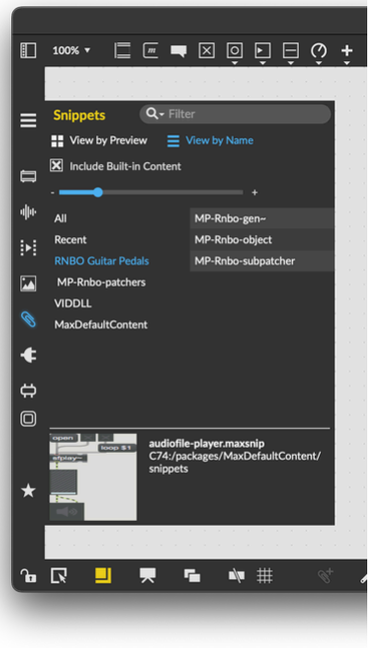
Click on the *Snippets* tab in the left sidebar to open the *Snippets* browser.



Click to open the Snippets browser

The *Snippets Browser* lets you browse snippets my name or by preview image. When viewing snippets by name, use the left column to filter by package. When viewing by preview image, use the pop-up menu to filter by package. To filter snippets by name, use the **filter** text entry field at the top of the browser.

To add a snippet to your patcher, just drag it from the browser and drop it into an unlocked patcher.



Snippet Location

Snippets are saved in the *Snippets* folder inside the Max 9 folder. Snippet files have the `.maxsnip` extension. They are `.maxpat` format files that include a preview image.

> You can add regular Max patcher files to the *Snippets* folder as well. This will add that patcher to the `Paste From` section of the context menu.

Using Snippets with Packages

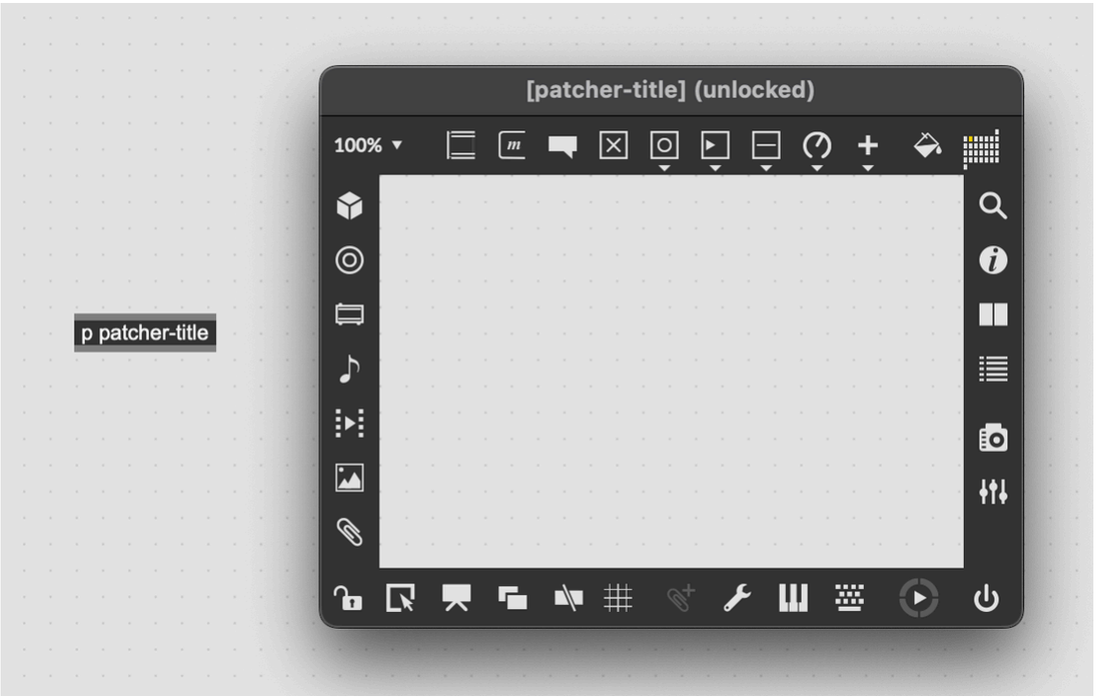
Packages can include snippets as well. If you're authoring a package, either for your own use or for distribution through the Package Manager, you can add snippets to your package. Create a folder called `Snippets` inside your package folder, and put whatever `.maxsnip` and `.maxpat` files you like into that folder.

Subpatchers and Encapsulation

Inlets and Outlets 679
Abstractions vs Subpatchers 679
Encapsulating and De-encapsulating 680

Subpatchers let you collapse a group of objects down to a single object. You can treat the subpatcher as if it were like any other Max object, but you can also double-click on the subpatcher object to see its contents at any time. Subpatchers can be extremely useful for managing the complexity of large patchers, and for grouping together objects that perform a single function.

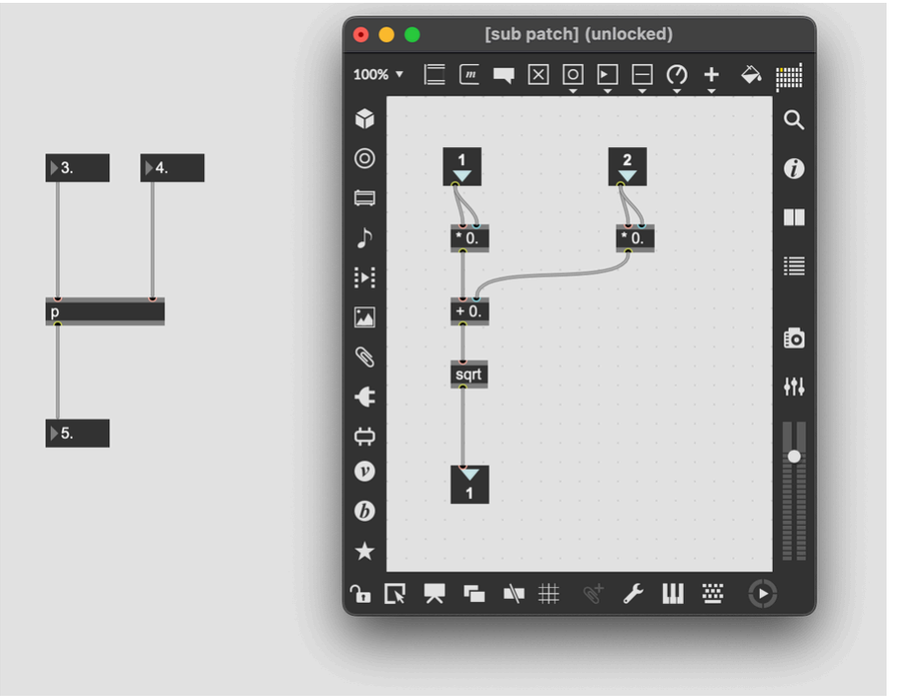
Create an empty subpatcher object by making a patcher object, which can also be abbreviated to `p`, or create a subpatcher from an existing group of objects using **encapsulation**.



An empty subpatcher, using the abbreviated object name 'p'. The title of the subpatcher window is the first argument of the subpatcher object.

Inlets and Outlets

Subpatchers and abstractions handle inlets and outlets in the same way. A subpatcher will have as many inlets or outlets as it has inlet or outlet objects. If you create a new inlet or outlet object in your subpatcher, the parent patcher will update to include the new inlet or outlet. The order of the inlets in the parent patcher corresponds to the order of inlets in the subpatcher. So, if you swap the position of two inlet objects in the subpatcher, those objects will map to different inlets in the parent subpatcher object.



The subpatcher has two inlet objects and one outlet object, so the parent subpatcher object has two inlets and one outlet.

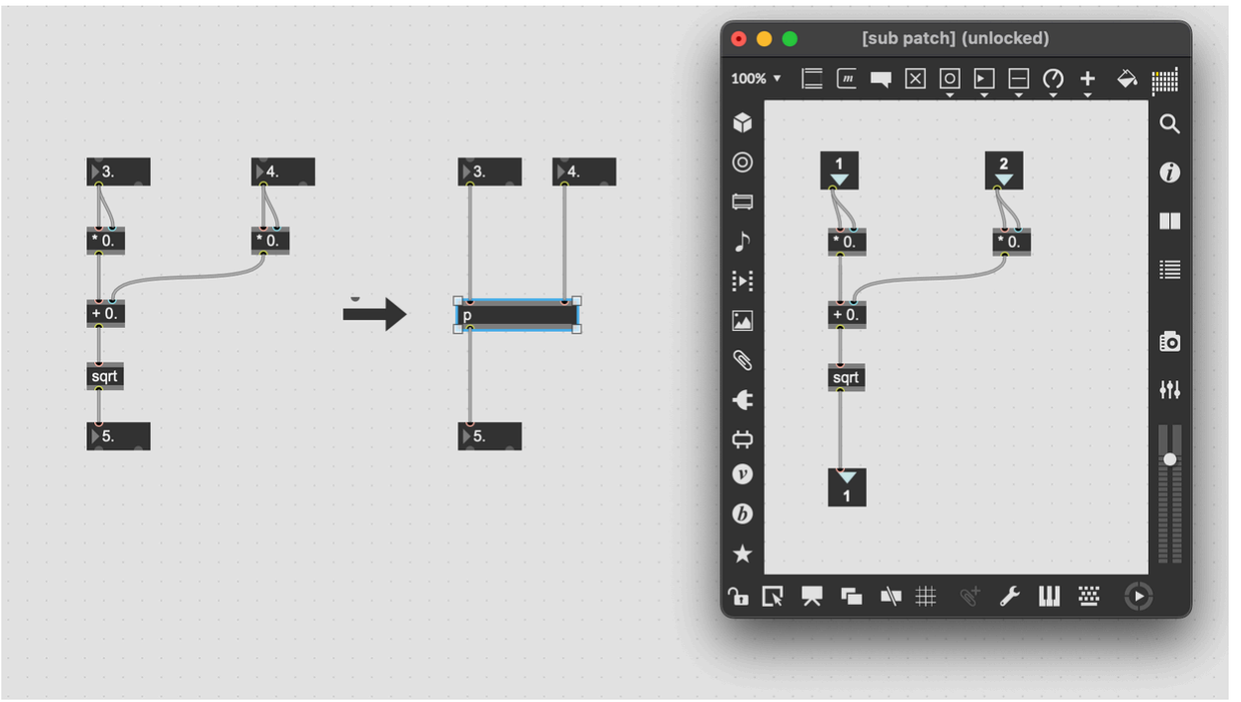
Just like a regular Max object, a subpatcher can add **Comments** to its inlets and outlets. If you set the `@comment` attribute on an inlet or outlet object, then when you mouse over the inlet or outlet you will see that text displayed.

Abstractions vs Subpatchers

As mentioned in the article on abstractions, subpatchers are embedded within their parent, while abstractions reference a saved `.maxpat` file. Changes that you make to a subpatcher only affect that subpatcher, whereas changes to an abstraction will modify the original file, and hence all copies of that abstraction. Abstractions can also take advantage of arguments and unique identifiers, while subpatchers cannot.

Encapsulating and De-encapsulating

With a group of objects selected, press ⌘`shifte` (macOS) or CTRL`shifte` (Windows), or select *Encapsulate* from the *Edit* menu to move those objects to a new subpatcher. If those objects were connected to other objects that weren't part of the encapsulation, Max will create inlet and outlet objects automatically, and connect the new subpatcher object in the correct way. The logic of your patcher will always be preserved after encapsulation.



A group of objects that express the Pythagorean theorem, computing the length of the hypotenuse of a right triangle. After encapsulation, the objects are contained in a subpatcher that implements the same logic.

It's also possible to de-encapsulate a subpatcher, copying all of the contained objects into the parent patcher and removing the subpatcher object. Select any subpatcher and press ⌘`shiftd` (macOS) or CTRL`shiftd` (Windows), or select `De-encapsulate` from the *Edit* menu to perform a de-encapsulation.

Re-initializing

> It's worth mentioning that encapsulation (as well as de-encapsulation) copies the selected objects into a new subpatcher and then deletes the original objects. The objects in the subpatcher are new objects, meaning their internal state will be reset to its initial value. This is a common source of confusion when starting out with Max.

Templates

Using Templates 682
Creating Templates 682
Default Template 683
Templates in Packages 684

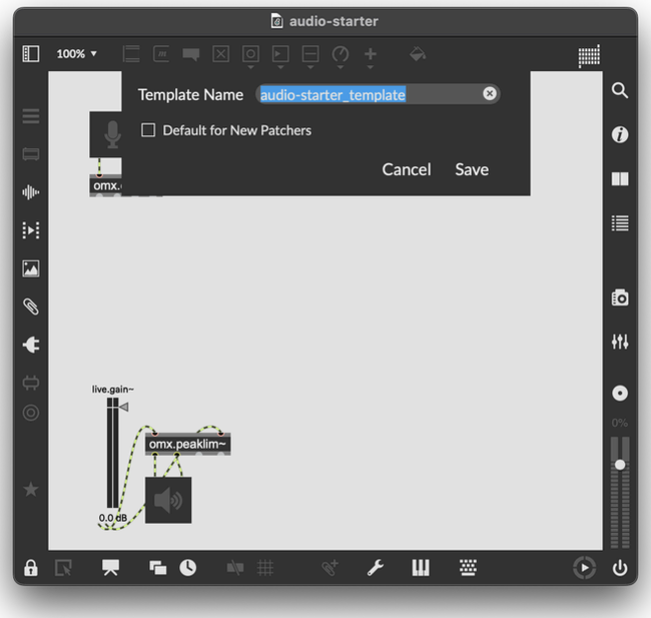
Templates are a starting point for a Max patcher. You can save any patcher as a template, and then rather than starting from a blank patcher, you can use your saved patcher as a starting point. Templates maintain patcher-level formatting including fonts and colors, so if you create a template with a patcher style, the style will be included in your template.

Using Templates

Select *New From Template* from the *File* menu to see a list of saved templates, including built-in templates. Choose a template from the list and it will open in a new patcher window.

Creating Templates

To create a template from a Max patcher, choose *Create Template...* from the *File* menu. A dialog box will appear that contains the name of a Max template file (the default filename will be based on the filename of the currently open Max patch).



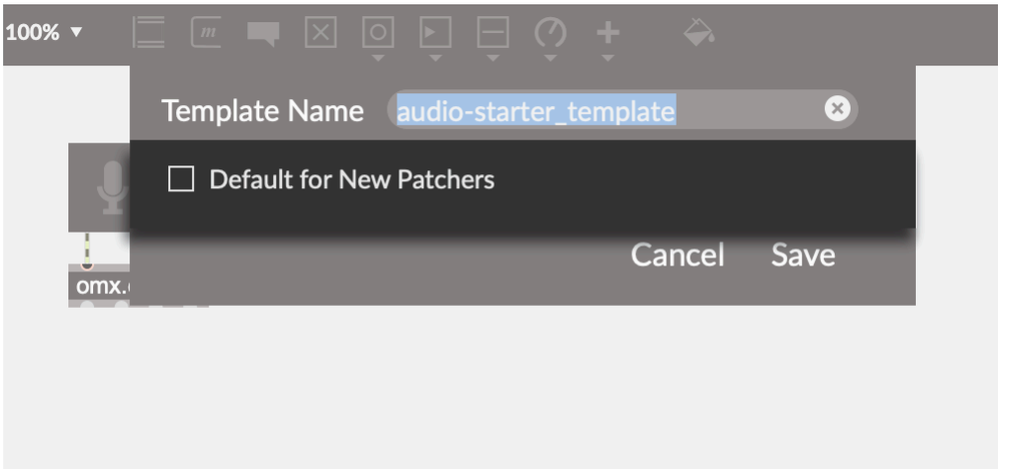
The modal dialog asks for a name for the new template.

Default Template

You can choose a template to be the **default template** for the whole Max application. Once you do, any new patcher that you create will be based on that template, rather than the typical empty patcher.

Setting a new default template

When you save a new template, you can check the `Default for New Patchers` checkbox to make the saved template the new default.



If you want to make an existing template the default, set the *Default Patcher Template* preference in the *Preferences* window. See the Default Patcher Template section in the Preferences reference for details.

Templates in Packages

If you're authoring a **Package**, whether for your own use, to share with your colleagues, or to publish to the **Package Manager**, you can include templates in that package as well. This can be really useful, for example as a way to demonstrate the functionality of your package.

To add a template to your package, create a folder called `templates` in your package folder. Any `.maxpat` files that you put in this folder will be available as templates. Once someone installs your package, they should see your custom templates listed in the dropdown whenever they choose `File > New From Template`.

Scripting

Scripting

Identifying Objects with Scripting Name 686
thispatcher 687
JavaScript 687
Messages to Max 688

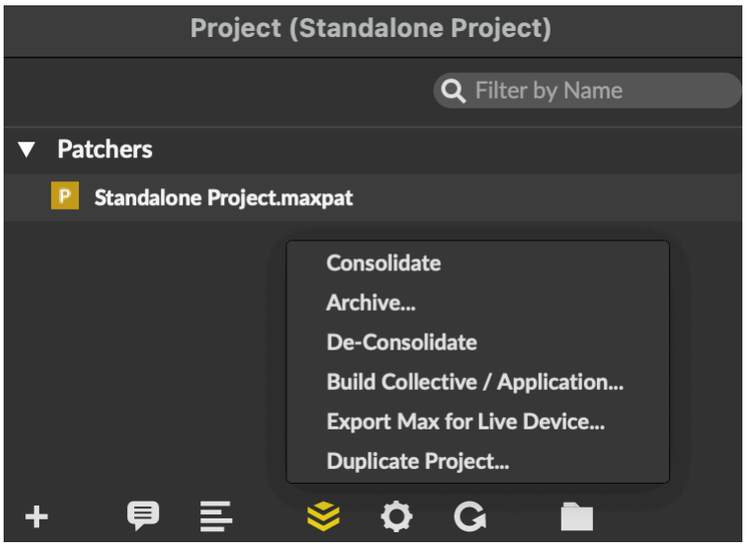
Max provides several mechanisms for **Scripting** the behavior of a patcher. Through scripting, you can accomplish many of the same things that you would normally do using the mouse and keyboard, but through an automated or programmatic mechanism. Some examples of things you could do with scripting:

- Create, delete, and connect objects
- Open and close patcher files
- Customize the appearance of UI objects
- Walk up and down the patcher hierarchy

Identifying Objects with Scripting Name

All Max objects have an attribute called `@varname`, also known as the **Scripting Name** of the object. This is the name by which the object can be uniquely identified within a patcher, and so no two objects in a patcher can share a scripting name. The scripting name is usually optional, but some objects require a scripting name. For example, any object with **Parameter Mode** enabled, including all `live.*` UI objects, must have a scripting name.

You can use the thispatcher object to send messages to an object using its scripting name.



Using JavaScript, you can also get a reference to an object using its scripting name.

```
function find_object(name) {
  let obj = this.patcher.getnamed(name);

  if (obj) {
    post(`Found an object named ${name}\n`);
  } else {
    post(`No object with name ${name}\n`);
  }
}
```

thispatcher

The thispatcher object is the object-level interface to Max's scripting capabilities. The help file for thispatcher provides an overview of its capabilities.

JavaScript

The v8 and v8ui objects let you embed JavaScript code directly in your patcher. In addition to all of the functionality of the JavaScript engine itself, you can also call on the Max JavaScript API to

interact with the Max application. See JavaScript for more.

Messages to Max

Finally, you can send messages directly to the Max application itself (or to global resources that the Max application owns, like the audio engine). This uses a special syntax with a semicolon `;` at the beginning of a message. See Controlling Max with Messages for more information, including a list of all available messages.

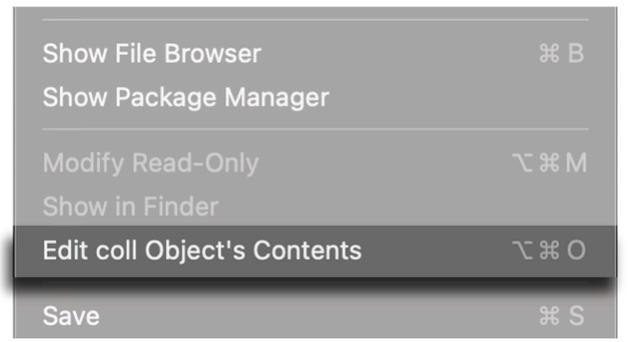
External Text Editor

Editing an Object's Text 689
Using an External Editor 689
Required File on Disk 690

Objects like coll, js, jsui, and node.script have an internal state that references a body of text. For editing the state of these objects, Max has a built-in text editor. However, you can also use an external text editor if you prefer.

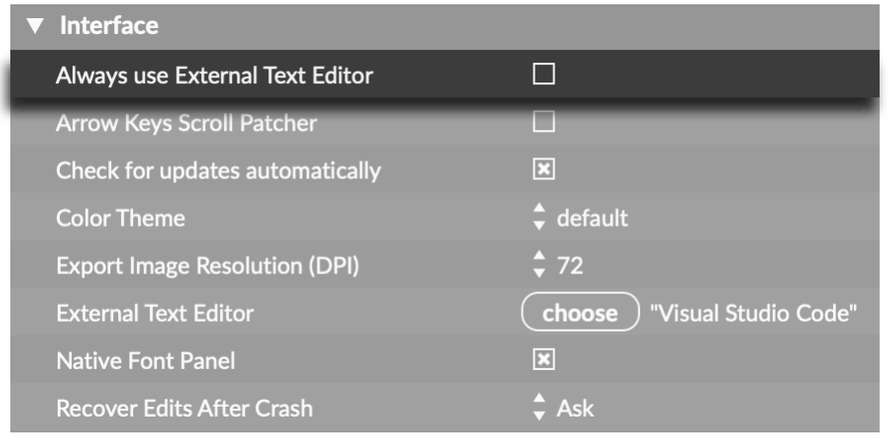
Editing an Object's Text

For any objects that have some internal text, you can open the text editor for that object by double-clicking on the object. There's also a menu command to open the text editor. For example, with a coll object selected, choose *Edit coll Object's Contents* from the *File* menu to open the text editor. By default, this will use Max's internal text editor.

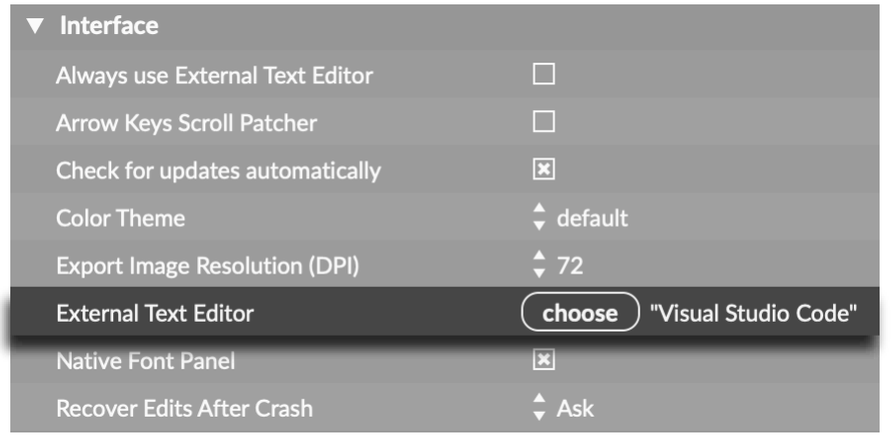


Using an External Editor

To use an external editor, open Max's preferences and enable *Always use External Text Editor*.

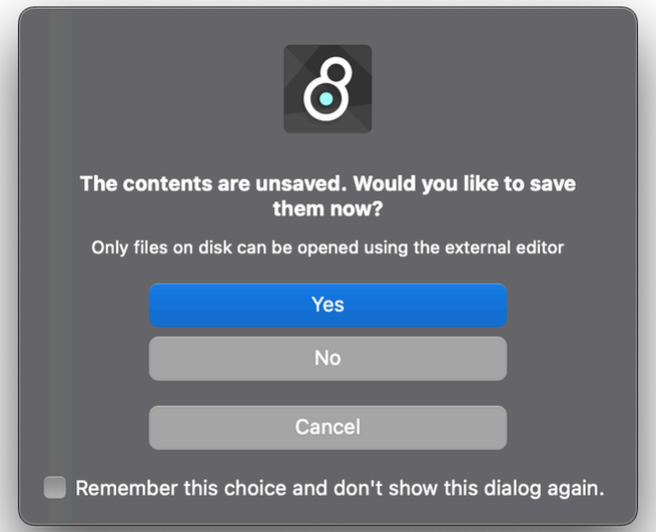


With this option enabled, Max will use whatever text editor application is the system default when opening a given file. If you set a value for "External Text Editor", them Max will always use that application when opening a text file.



Required File on Disk

If you've enabled *Always use External Text Editor*, you may see a dialog appear when you try to open up a text-based object for editing.



When using Max's internal text editor, Max can display the contents of an object as text without creating an actual file on disk. However, when using an external text editor, Max must save an actual file before it can open the object contents with an external editor. Choose "Yes" to save a file on disk. After creating the file, Max will open the file in your chosen external text editor.

JavaScript

JavaScript Objects 693
jsthis 694
Arguments 695
Input 695
Special Functions 696
Output 700
Global Code and Initialization 700
require 701
Private (Local) Functions 702
Available APIs 703
JavaScript, Threading, and Priority 703
See Also 703

Using **JavaScript**, you can script the behavior of Max by writing text code. Most of the things that you could do with a Max external, you can also do with a JavaScript object.

- Receive messages from inlets
- Manage internal state
- Send messages to outlets
- Perform custom drawing and handle mouse events
- Schedule events
- Query the state of the patcher
- Other functions exposed through the JavaScript API



JavaScript Objects

The JavaScript family of objects v8 and v8ui (js and jsui using the older JavaScript engine) let you define a custom Max object using JavaScript. The JavaScript code that you write to define the behavior of your object can be loaded as a separate `.js` file, or embedded in your patcher.

> Max is in the process of updating its JavaScript engine, and so there are currently two engines available in Max. There is an older JavaScript engine, running version 1.8.5, as well as a newer v8 engine. The older version is only maintained for backwards compatibility, and in the near future the v8 engine will be a drop-in replacement for the older engine. Developers using the old engine through the js object don't need to take any action—when the old engine is replaced the js object will also use the v8 engine. If you're writing new JavaScript code, use the v8 and v8ui objects to take advantage of the new engine.

There are two JavaScript objects: v8 and v8ui (and v8.codebox, which is functionally the same as v8). The main difference between the two is that v8 is a simple Max object, while v8ui provides a drawing and interaction context. Use v8 to implement a generic Max object using JavaScript, and use v8ui to implement a user interface object. You can also replace the default drawing behavior of any UI object using the `@jspainter` attribute. See **Custom UI Objects** for a full description of working with `@jspainter` and jsui.

> Max also provides an object called node.script, which lets you start and interact with Node processes from Max. Node uses JavaScript, but code running in node.script runs in a separate process.

jsthis

Before calling into your JavaScript code, Max binds an instance of jsthis to `this`. Doing so puts a number of Max-specific functions into scope. Using `this` is optional when referring to these functions.

```
function bang() {
  // The `post` function is a method of jsthis
  post("hi\n");

  // Using "this" is optional, but makes explicit the reference
  // to the bound instance of jsthis
  this.post("nice to see you\n");
}
```

See the JS API Docs for a full list.

Number of Inlets and Outlets

Set the number of inlets and outlets on your JavaScript object by setting the value of `inlets` and `outlets` on `jsthis`.

```
inlets = 3; // object will have three intles
outlets = 2; // object will have two outlets

function bang() {
  outlet(1, "hi"); // send "hi" out the second outlet.
}
```

Arguments

Arguments supplied to the js or v8 object after the filename will be passed to the JavaScript code as arguments. These will be available in a `jsthis` property called `jsarguments`.

```
// The first argument will always be the name of the file
const filename = jsarguments[0];

const argumentsLength = jsarguments.length;

// Get an array of arguments supplied by the user
const userArguments = jsarguments.slice(1, argumentsLength);

// Fancy way to do the same thing with array destructuring
const [filename1, ...userArguments1] = jsarguments;
```

> With jsui and v8ui, use the `@arguments` attribute to specify the argumetns to your JavaScript code.

Input

A message received in the inlet of a v8 or v8ui object will invoke a function with the same name. Arguments following the message name will be passed to the JavaScript function. This method

would print "1, 2, 3" in response to the message `foo 1 2 3`.

```
function foo(a, b, c) {
  post(a, b, c);
}
```

With `v8` you can use destructuring assignment to get the arguments to the function as an array.

```
function foo(...args) {
  post(args.length);
}
```

You can even get the first argument as a single value and the rest as an array.

```
function foo(first, ...rest) {
  post(first, ...rest);
}
```

Special Functions

You can define a number of functions with special names to respond to specific hooks from Max.

bang

Invoked in response to a bang message.

msg_int, msg_float

Invoked in response to an integer or a float, respectively.

```
function msg_int(a) {
  post(`received an int: ${a}\n`);
}

function msg_float(a) {
  post(`received a float: ${a}\n`);
}
```

If you define only `msg_int`, any float received will be truncated and passed to `msg_int`. Similarly, if only `msg_float` exists, an int received will be passed to the `msg_float` function.

list

Invoked in response to a list (a message with more than one element that starts with a number).

```
function list(...elements) {
  post(`elements length: ${elements.length}\n`);
  post(`first element: ${elements[0]}\n`);
}
```

anything

You can define an `anything` function that will run if no specific function is found to match the message symbol received by the v8 or v8ui object. If you want to know the name of the message that invoked the function, use the `messagename` property. If you want to know what inlet received the message, use the `inlet` property.

```
function anything(...args) {
  post(`message: ${this.messagename}\n`);
  post(`inlet: ${this.inlet}\n`);
  post(`arguments: ${args}\n`);
}
```

loadbang

Invoked when the patcher file containing the v8 or v8ui object is loaded. This function will not be called when you instantiate a new v8 or v8ui object and add it to a patcher; it will only be called when a pre-existing patcher file containing a JavaScript object is loaded.

```
function loadbang() {
  post("loadbang\n");
}
```

getvalueof

Defining a `getvalueof` function lets your JavaScript object participate in the **pattr** system, enabling Max to save the state of your JavaScript object using pattr and pattrstorage. The return value of `getvalueof` can be a `number`, a `string`, an `Array` of `number` and `string`, or a Max Dict.

```
let myvalue = 0.25;
function getvalueof() {
  return myvalue;
}
```

setvalueof

If you've defined a `getvalueof` function, you can define a `setvalueof` function to enable pattr and related object to restore the state of your JavaScript object from a preset. Like the return value of `getvalueof`, the arguments to `setvalueof` can be a `number`, a `string`, an `Array` of `number` and `string`, or a Max Dict.

```
let myvalue;
function setvalueof(v) {
  myvalue = v;
}
```

save

Defining a function called `save` allows your script to embed state in a patcher file containing your JavaScript object. Max will automatically restore your saved state when the patcher is loaded.

Saving your state consists of storing a set of messages that your script will receive shortly after the JavaScript object is recreated. These messages are stored using a special global function called `embedmessage` that only works inside the `save` function.

Suppose you have a function `cowbells` that sets the number of cowbells your object currently has:

```
let numcowbells = 1;
function cowbells(a) {
  numcowbells = a;
}
```

When the patcher containing the JavaScript object is saved, you would like to preserve the current number of cowbells, so you define a `save` function as follows:

```
function save() {
  embedmessage("cowbells", numcowbells);
}
```

The first argument to embedmessage is the name of the function you want to call as a string. Additional arguments to embedmessage supply the arguments to this function. These additional

arguments will typically be the values of the state you want to save. For each call to `embedmessage`, Max will call that function with the provided arguments as it restores the state of your JavaScript object.

notifydeleted

The `notifydeleted` method is called when the JavaScript object is freed.

Reserved names

The v8 and v8ui objects already do something in response to the `compile` message. So, if you define a function named "compile" in your JavaScript code, there will be no way to call that function from Max. However, you can still call the function locally, from your own JavaScript code.

Output

Call the `jsthis` method `outlet` to send messages out of a given outlet.

```
function bang() {
  outlet(0, "bang");
}
```

Global Code and Initialization

When the JavaScript object is loaded, Max will execute the script once from beginning to end. Variables defined in the global scope will persist through the life of the object, and you can use these to hold on to internal state.

```
let counter = 0;

function count() {
  post(`count: ${++counter}`);
}
```

In fact, Max will attach global variables to the `this` context when executing functions from an inlet. The above code is equivalent to the following.

```
this.counter = 0;

function count() {
  this.counter++;
  post(`count: ${this.counter}`);
}
```

> During the execution of global code, the JavaScript object is still being initialized. The object does not have any outlets, nor is it yet part of any patcher. If you want to send messages to an outlet immediately after your JavaScript object is created, define a `loadbang` function.

require

Use `require` to include code from other JavaScript files.

```
const lib = require("my-lib.js");

function call(a) {
  const computedValue = lib.compute(a);
  outlet(0, computedValue);
}
```

The included file should be a CommonJS module. To export functions and variables, it should set the properties of `exports` or else replace `module.exports` with an object containing exported properties.

```
// An example implementation of my-lib
function compute(a) {
  return a + 10;
}

module.exports = {
  compute: compute,
};
```

Private (Local) Functions

If you want to use a function locally, but you don't want it to be called from Max, you can set its `local` property to `1`. For example, suppose the function `foo` is not something we wish to expose to the outside world.

```
foo.local = 1;
function foo() {
  post("what does Pd *really* stand for?");
}
```

Now, when we send the message `foo` to the JavaScript object, we see the following error in the Max window:

```
error: js: function foo is private
```

Available APIs

Since the Max JavaScript engine is not running inside of a web browser, certain APIs may not be available.

> The v8 engine provides several runtime libraries, which provide some functionality that is not available in the legacy JavaScript engine. For example, setImmediate and setTimeout.

- JSON serialization with `JSON.stringify` and `JSON.parse` are available.
- Timing functions like `setImmediate` and `setTimeout` are not available in the legacy JavaScript engine (though they are available with the v8 engine). Use **Task** instead.
- There is no DOM, so document methods like `document.getElementById` are not available.

JavaScript, Threading, and Priority

Max schedules events using a high priority thread for events that require high timing accuracy, like MIDI, and a low priority thread for long-running operations like decompressing video. The JavaScript engine in Max always executes code in the low priority thread. That means that if a JavaScript object receives a message from a MIDI object, or from any other object that schedules events on the high priority queue, that event will be deferred to the low priority queue.

See Also

- Max JS API

jit.gl.lua

The 'this' Variable	706
Messages and Functions	706
OpenGL Callback Functions	708
Creating Inlets and Outlets	709
Inlet Detection and Data Conversion	710
Outlets and Data Conversion	711
Creating Jitter Objects	711
jit.matrix	712
jit.listener	713
jit.gl Functions	714
Vector Math Functions	718
OpenGL Bindings	719
Color Functions	721

---

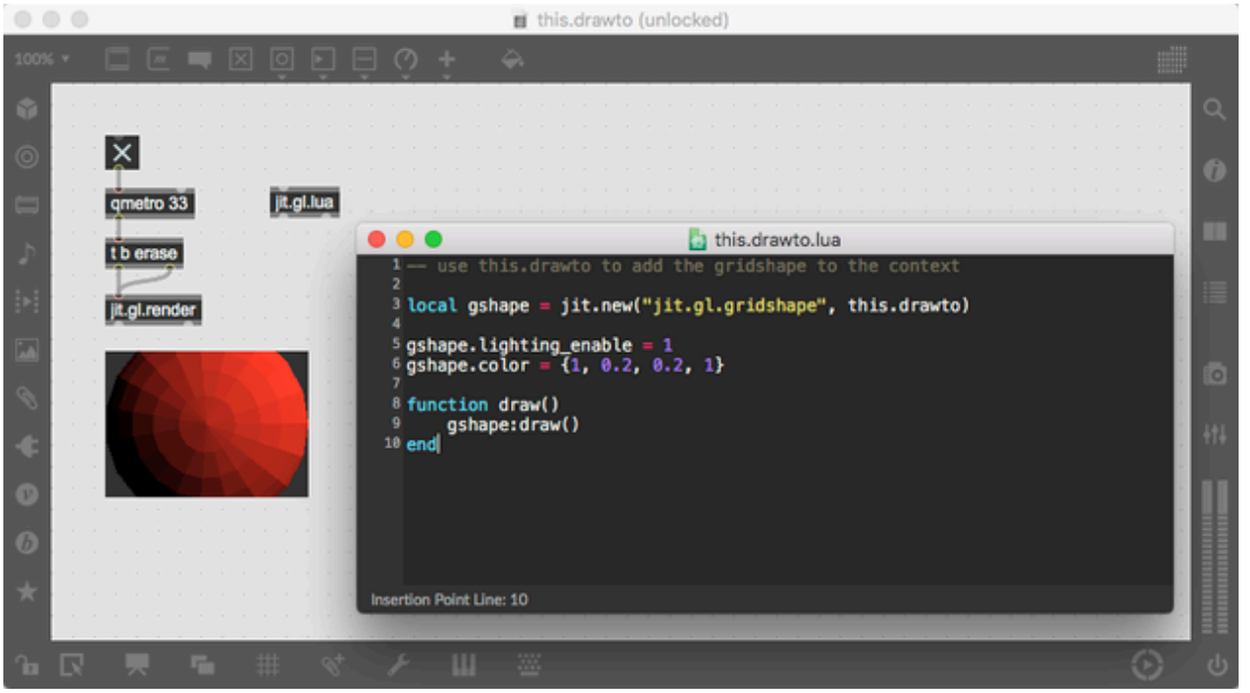
jit.gl.lua embeds the Lua scripting language inside a Jitter OpenGL object. jit.gl.lua serves as both a general purpose Lua scripting object and a 3D graphics scripting object. It is very similar to the js object for JavaScript with the addition that OpenGL commands can be used directly. OpenGL is what Jitter uses for 3D graphics rendering. Normally, access to the 3D drawing functionality is wrapped in higher level jit.gl objects (e.g. jit.gl.gridshape). jit.gl.lua however exposes OpenGL directly in its scripting environment for more precise, dynamic control over OpenGL's 3D drawing commands.

The Lua scripting environment inside jit.gl.lua interacts with Max and Jitter in two ways: through patcher messages and through commands from Jitter's OpenGL rendering system. If you have done Lua scripting in other contexts, that knowledge is directly applicable to working with jit.gl.lua. jit.gl.lua simply adds extra functionality to the standard Lua environment, and this is what is covered in this documentation. For more information on Lua, visit the Lua homepage.

For documentation on the OpenGL commands, what they do, and how to use them, please visit the OpenGL website. Of particular note are the OpenGL man pages, which are available for each command. They explain what type of arguments a command requires, what it does, and any possible errors that may arise from improper use.

The 'this' Variable

In jit.gl.lua, there is a special variable called `this`. `this` represents the jit.gl.lua object the Lua script is running inside. Whenever you need to modify or query the properties of embedding jit.gl.lua object, use the `this` variable. A common idiom when scripting with jit.gl.lua is to use the drawto attribute of jit.gl.lua when creating other jit.gl objects. All jit.gl objects must belong to a rendering context in order to be used. In Jitter, rendering contexts are given names, so jit.gl objects join a rendering context using the context's name. This attribute is called `drawto`.

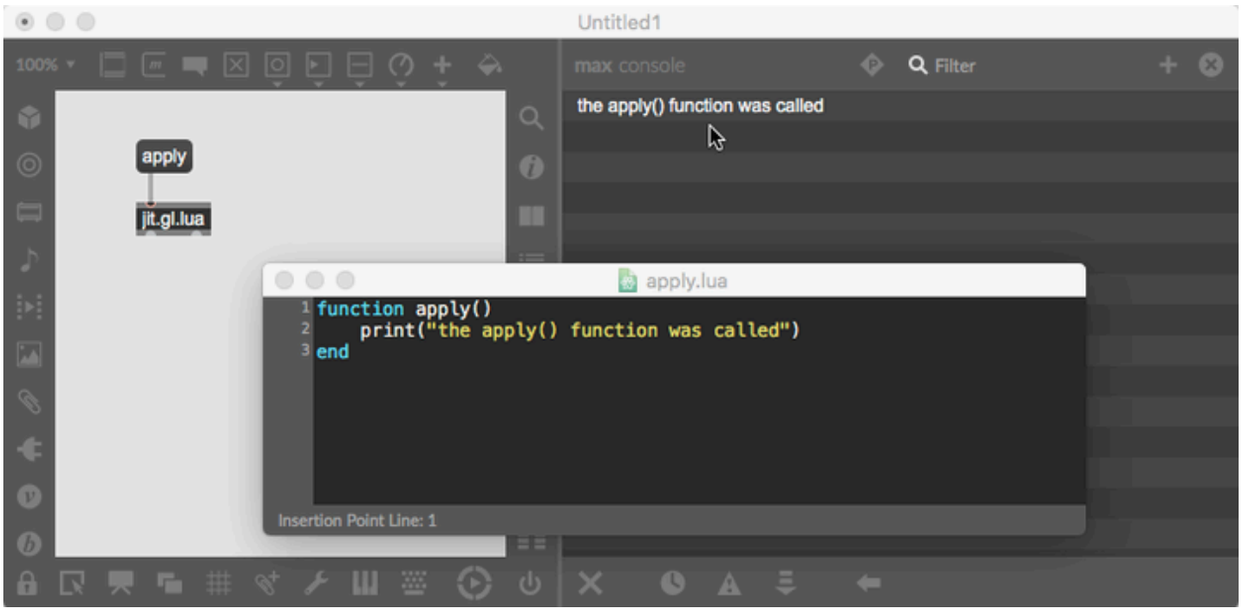


When other jit.gl objects are created in a script, they too have to belong to a context, so we typically attach them to the same context as the jit.gl.lua object, which can be accessed by referring to `this.drawto`.

Messages and Functions

Global functions in jit.gl.lua can be called by sending messages to the jit.gl.lua object. If a script contains the code

```
function  apply()
   print( "the apply() function was called" )
 end
```



sending the message apply to a jit.gl.lua object will call the function above. If the message is followed by any extra strings or numbers, these will be passed as arguments to the function. While a function can have any name and be called by a message, there are a few function names that have special significance. There are two groups of these functions: one for handling patcher input and another for Jitter OpenGL messages.## Patcher Interaction Functions The patcher handling, the functions are `float`, `int`, `list`, `loadbang`, `closebang`, and `scriptload`.

```
function  float(v)
   print( "float" , v,  "inlet" , this.last_inlet)
 end

 function  int(v)
   print( "int" , v,  "inlet" , this.last_inlet)
 end

 function  list(...)
    local  values = {...}
   print( "list" , table.concat(values,  ", " ),  "inlet" ,
this.last_inlet)
 end
```

When a number is sent to a jit.gl.lua inlet, the `float` messages is called and passed the value. When an integer is sent, the `int` message is called. Similarly, a list of values triggers the `list` function with the elements of the list set as arguments to the function. The list can be captured into a table by using the Lua idiom

```
-- variable arguments captured into a table
  local  thelist = {...}
```

When the patcher is loaded, the `loadbang` function is called, and when it is closed, the `closebang` function is called. `loadbang` will only be called when the patcher containing the jit.gl.lua object is called. This happens once during the patcher lifetime, so if a script is reloaded, `loadbang` will not be called again. To perform any setup each time a script is loaded, use the `scriptload` function.

OpenGL Callback Functions

The Opengl methods are `draw`, `dest_changed`, `dest_closing`. They are called in response to events in the Jitter OpenGL rendering system. The OpenGL callback functions are used to manage OpenGL resources and draw to the rendering destination. It is important to remember that OpenGL

commands can *only* be used inside the three OpenGL callback functions. Whenever a jit.gl.lua object is asked to draw itself by the jit.gl.render object, it calls the `draw` function. When the `draw` function is called OpenGL commands can be used to draw to the rendering destination. Any OpenGL related attributes set on the jit.gl.lua object objet such as `blend` will be applied before the draw function is called, so they can be used initialize the drawing state before any OpenGL commands are called in the draw function.

The `dest_changed` and `dest_closing` functions on the other hand are used to manage OpenGL resources. `dest_changed` is called when the OpenGL context is created and the first frame is about to be rendered. `dest_closing` is called when the context is about to be destroyed. When both functions are called, the OpenGL context is active so any OpenGL command can be used. Typically the `dest_changed` function is used to create OpenGL resources such as display lists while the `dest_closing` function is used to destroy them.

```
local  gl = require("opengl")
 local  GL = gl

 function  dest_changed()
   -- context is new
 end

 function  dest_closing()
   -- context is closing
 end

 function  draw()
   -- draw a line
  gl.Begin(GL.LINES)
    gl.Vertex( -1 ,  0 ,  0 )
    gl.Vertex( 1 ,  0 ,  0 )
  gl.End()
 end
```

Creating Inlets and Outlets

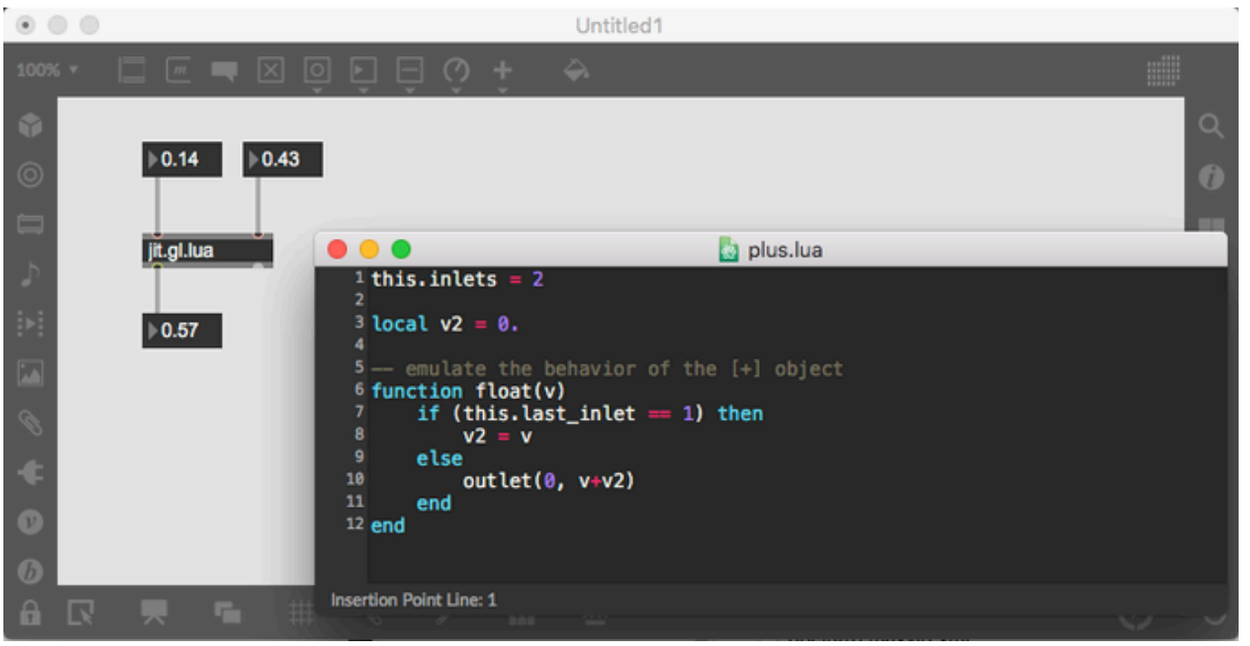
jit.gl.lua can have a variable number of inlets and outlets. The number of inlets and outlets are specified by setting the `inlets` and `outlets` attriubutes respectively on the jit.gl.lua object itself.

```
-- 3 inlets, 2 outlets
this.inlets = 3
this.outlets = 2
```

Since jit.gl.lua has dynamic inlets and outlets, whenever the `inlets` or `outlets` attriubutes are set, the change is immediately reflected in the patcher.

Inlet Detection and Data Conversion

Lua has a single number type, so integer and float inputs are both converted to a Lua number. When an input is sent to an inlet, jit.gl.lua 's `last_inlet` attribute is set, allowing a script to determine what inlet a message was sent to so that inlet-specific behaviors can be designed if needed.



In the script above, the `last_inlet` attribute of jit.gl.lua is used to save the input number when it comes in the right inlet. When input comes in the left inlet, it triggers an addition operation and the result is sent out the outlet.

Outlets and Data Conversion

Outlets in jit.gl.lua are accessed through the `outlet` function. `outlet` takes two or more arguments. The first argument is the outlet index with 0 as the left-most outlet. The other arguments are the values to be sent out the outlet. Outlets can be sent both string and numerical values. If a numerical value is an integer, jit.gl.lua will convert it to an Max integer type, otherwise it will be a Max float type.

Creating Jitter Objects

Jitter objects can be created in a jit.gl.lua script using the `jit.new` function. When called, `jit.new` will create and return a Jitter object. The arguments to `jit.new` are the classname of the object and any additional arguments that the object's constructor might take. For example, to create a jit.xfade object:

```
xfade = jit.new( "jit.xfade" )
```

to create a jit.gl.gridshape object:

```
-- classname of the object, the context name
gshape = jit.new( "jit.gl.gridshape" , this.drawto)
```

Once created, the messages and attributes of the object are exposed to Lua in the usual Lua fashion. In Lua, functions belonging to an object are called using the ':' operator, which is a kind of object-oriented syntax sugar. This operator passes the object itself in as an implicit first argument.

```
gshape = jit.new( "jit.gl.gridshape" , this.drawto)

  function  draw()
        -- use ':' to call the gridshape's draw function
        -- equivalent to gshape.draw(gshape)
      gshape:draw()
  end
```

All Jitter object messages can be called this way. Attributes are acessed more like properties stored in a table and can be accessed using the '.' operator. For example:

```
gshape = jit.new("jit.gl.gridshape", this.drawto)
  -- set some of the gridshape's attributes
gshape.shape =  "cylinder"
gshape.automatic =  0
gshape.poly_mode = { 1 ,  1 }   -- set an attribute that takes a list of
values

  -- print the shape attribute
print("gridshape using shape", gshape.shape)
```

Some attributes consist of a array of values such as gridshape's `poly_mode` above. These can be set with a list of values stored in a table. When getting an array attribute, the list of values will be returned in a table as well.

jit.matrix

There are two ways to create a jit.matrix object in jit.gl.lua. One is using the `jit.new` function as described above. The other is by using the convenience `jit.matrix` function. Both are equivalent in terms of functionality. The `jit.matrix` function takes the same arguments as are used when creating a jit.matrix object in a patcher. The arguments are the matrix name (optional), planecount, type, and dimensions.

```
-- equivalent to jit.new("jit.matrix", 4, "char", 720, 480)
mat1 = jit.matrix(4,  "char" , 720, 480)

  -- a named matrix
mat2 = jit.matrix( "frame" )
```

jit.listener

Certain Jitter objects such as jit.window send messages out an outlet in response to patcher and user events. Since Jitter objects in scripts have no outlets, this information has to be accessed another way. In jit.gl.lua, the jit.listener object is used to attach to an object and listen to any notifications it may send out.

```
-- listener function
  function  wincb(event)
    print(event.subjectname)
    print(table.concat(event.args,  ", " ))
  end

win = jit.new( "jit.window" ,  "x" )
  -- list to the window with wincb as the callback function
listener = jit.listener(win.name, wincb)
```

jit.listener takes two arguments: the name of the object to listen to and the function to call when the listener gets notified of some information. The second argument can either be a function directly or the name of a global function. This code is equivalent to the code above:

```
function  wincb(event)
      print(event.subjectname)
      print(table.concat(event.args,  ", " ))
  end

win = jit.new( "jit.window" ,  "x" )
listener = jit.listener(win.name,  "wincb" )
```

Notice that the second argument in this code is a string naming the function to call as opposed to the function itself.

jit.gl Functions

In addition to the low-level OpenGL bindings discussed below, jit.gl.lua has bindings to Jitter-specific OpenGL functionality for working with Jitter textures and jit.gl objects at a lower level than usual.

```
-- v2 and v3 are optional
jit.gl.texcoord(v1, [v2], [v3])

  -- also can take a table
jit.gl.texcoord({v1, [v2], [v3]})
```

`jit.gl.texcoord` creates multi-texturing texture coordinates so that if there are textures bound on different units, they will each get texture coordinates to be properly displayed on the geometry.

```
local  gl = require( "opengl" )
 local  GL = gl

 function  draw()
   gl.Color( 1 ,  1 ,  1 ,  1 )
   jit.gl.bindtexture( "tex1" ,  0 )
   jit.gl.bindtexture( "tex2" ,  1 )
   gl.Begin(GL.QUADS)
      jit.gl.texcoord( 0 ,  0 ) gl.Vertex( -1 ,  -1 ,  0 )
      jit.gl.texcoord( 1 ,  0 ) gl.Vertex( 1 ,  -1 ,  0 )
      jit.gl.texcoord( 1 ,  1 ) gl.Vertex( 1 ,  1 ,  0 )
      jit.gl.texcoord( 0 ,  1 ) gl.Vertex( -1 ,  1 ,  0 )
   gl.End()
   jit.gl.unbindtexture( "tex2" ,  1 )
   jit.gl.unbindtexture( "tex1" ,  0 )
 end
```

```
jit.gl.bindtexture(texname, texunit)
  -- draw some geometry
jit.gl.unbindtexture(texname, texunit)
```

`jit.gl.bindtexture` and `jit.gl.unbindtexture` are used to bind and unbind texture. The first argument is a texture name and the second argument is the texture unit to assign the texture to. Every graphics card has a fixed number of slots called texture units. The number of texture units a card has determines how many textures can be used simultaneously. This number is usually 8 but can be 16 or even 32. To see how many texture units your card has, go to the Options Menu > OpenGL Status. Under the OpenGL Limits > Textures item, you'll see MAX_TEXTURE_UNITS and a value next to it. This is the number of texture units your card supports.

```
jit.gl.begincapture(texname)
  -- draw some geometry
jit.gl.endcapture(texname)
```

`jit.gl.begincapture` and `jit.gl.endcapture` are used to render drawing commands to a texture instead of to a window. The only argument is the texture name.

```
local  gl = require( "opengl" )
  local  GL = gl

  local  pi = math.pi

  local  tex = jit.new( "jit.gl.texture" , this.drawto)
tex.dim = {1024, 1024}

  function  draw()
        -- capture to texture
      jit.gl.begincapture(tex.name)
            gl.Color( 1 ,  1 ,  1 ,  1 )
            gl.Begin(GL.LINES)
            for i= 0 , pi, pi/ 100  do
                  gl.Vertex(math.cos(i), math.sin(i* 2.4 ))
                  gl.Vertex(math.cos(i+pi), math.sin(i* 2.4 +pi))
            end
            gl.End()
              -- end capturing to texture
      jit.gl.endcapture(tex.name)

        -- draw the result
      gl.Color( 1 ,  1 ,  1 ,  1 )
      jit.gl.bindtexture(tex.name,  0 )
      gl.Begin(GL.QUADS)
            gl.TexCoord( 0 ,  0 ) gl.Vertex( -1 , - 1 ,  0 )
            gl.TexCoord( 1 ,  0 ) gl.Vertex( 1 , - 1 ,  0 )
            gl.TexCoord( 1 ,  1 ) gl.Vertex( 1 ,  1 ,  0 )
            gl.TexCoord( 0 ,  1 ) gl.Vertex( -1 ,  1 ,  0 )
      gl.End()
      jit.gl.unbindtexture(tex.name,  0 )
  end
```

```
-- arguments can be either a table or list of values
screenpos = jit.gl.worldtoscreen(worldpos)

  -- arguments can be either a table or list of values
worldpos = jit.gl.screentoworld(screenpos)
```

`jit.gl.worldtoscreen` converts world coordinates into screen coordinates. `jit.gl.screentoworld` performs the inverse operation, converting screen coordinate into world coordinates. Both functions can take either a table of values or a list of x, y, z values. The z-coordinate in `jit.gl.screentoworld` is typically a value in the range [0, 1] where 0 represents the near clipping plane and 1 the far clipping plane, which can be used to cast a ray into the OpenGL scene as the code below demonstrates:

```
function  castray(x, y)
   local  raystart = jit.gl.screentoworld(x, y,  0 )
   local  rayend = jit.gl.screentoworld(x, y,  1 )
   return  raystart, rayend
 end
```

```
jit.gl.draw_begin(jit_gl_object)
jit.gl.draw_end(jit_gl_object)
```

`jit.gl.draw_begin` and `jit.gl.draw_end` operate on jit.gl objects. All jit.gl objects share a set of common attributes know as ob3d (or object 3D) attributes. Whenever a jit.gl object draws itself, it sets up the OpenGL state to reflect the settings of its attributes such as `depth_enable` and `blend` among others. When an object is drawn, the following sequence of calls takes place:

```
draw_begin(ob3d)
draw(ob3d)
draw_end(ob3d)
```

Sometimes it's useful to have explicit control over this sequence of calls within a jit.gl.lua script and `jit.gl.draw_begin` and `jit.gl.draw_end` enable this kind of control. `jit.gl.draw_begin` sets up OpenGL state based on the object's ob3d attributes while `jit.gl.draw_end` reverses the process. These calls must be used in pairs, otherwise OpenGL errors may occur.

```
gshape = jit.new( "jit.gl.gridshape" , this.drawto)
gshape.automatic = 0

  function  draw()
        -- equivalent to gshape:draw()
      jit.gl.draw_begin(gshape)
      gshape:drawraw()
      jit.gl.draw_end(gshape)
  end
```

Vector Math Functions

The vector math functions in jit.gl.lua are located in the `vec` module and are organized into six categories. These categories are:- vec.vec2

- vec.vec3
- vec.vec4
- vec.quat
- vec.mat3
- vec.mat4 For more detailed documentation on the `vec` module, see the jit.gl.lua Vector Math Overview.

OpenGL Bindings

Detailed documentation on each function in the OpenGL module can be found on the jit.gl.lua OpenGL Bindings page. There are also bindings for the OpenGL Utility (GLU) functions at jit.gl.lua OpenGL GLU Bindings. This section describes common usage and techniques.

The OpenGL bindings in jit.gl.lua enable direct access to OpenGL commands. Nearly all of the OpenGL commands are available through the bindings. Wherever possible the arguments to an OpenGL function in Lua match the arguments described in the OpenGL man page for that command. Since OpenGL is a C interface, type information and the number of arguments a command must take are fixed. Lua does not have this restriction, so in an effort to simplify the interface into OpenGL, functions that only vary based on type and the number of arguments have been collapsed into a single function. For example, the OpenGL command for specifiying a vertex of geometry as the following variations:

```
void glVertex2s(GLshort x, GLshort y);
void glVertex2i(GLint x, GLint y);
void glVertex2f(GLfloat x, GLfloat y);
void glVertex2d(GLdouble x, GLdouble y);
void glVertex3s(GLshort x, GLshort y, GLshort z);
void glVertex3i(GLint x, GLint y, GLint z);
void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
void glVertex3d(GLdouble x, GLdouble y, GLdouble z);
void glVertex4s(GLshort x, GLshort y, GLshort z, GLshort w);
void glVertex4i(GLint x, GLint y, GLint z, GLint w);
void glVertex4f(GLfloat x, GLfloat y, GLfloat z, GLfloat w);
void glVertex4d(GLdouble x, GLdouble y, GLdouble z, GLdouble w);
```

In Lua, all of the variations are contained within a single function:

```
-- The same function, different number of arguments
gl.Vertex(x, y)
gl.Vertex(x, y, z)
gl.Vertex(x, y, z, w)
```

In OpenGL, enumerations play an important role for specifying different modes of behavior. An enumeration is simply a number with a particular meaning. For example, OpenGL has a lot of functionality that can be enabled and disabled such as depth testing, blending, etc. To enable or disable a particular bit functionality, an enumeration specifiying the functionality is passed to the glEnable or glDisable command. In C this looks like:

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_BLEND);
```

The Lua bindings for OpenGL also have enumerations. These are stored in the same location as all of the OpenGL functions, which is simply a giant table. Often times it is convenient to have the Lua code look as much like the C code as possible so that when copying example code or switching between C and Lua there is minimal cognitive overhead. We can do this by aliasing the OpenGL module table to the variable 'GL'. When jit.gl.lua loads a script, it autmatically makes available the `opengl` table, which contains all of the OpenGL functions and enumerations. The following idiom allows us to write code that more closely resembles C-style OpenGL code:

```
-- set the OpenGL module to the variable 'gl'
  local  gl = require( "opengl" )
  -- alias the OpenGL module to the variable 'GL' to emulate the C
enumeration style
  local  GL = gl

  function  draw()
      gl.Enable(GL.DEPTH_TEST)
      gl.Disable(GL.BLEND)

        -- draw some geometry
  end
```

Color Functions

In addition to the OpenGL module, jit.gl.lua also has a color module built in. The color module contains funtions for translating betwee RGB space and Hue-Saturation-Luminance (HSL) space. It also contains a large number of pre-defined color values. The color values are given in RGB form. Here are some examples:

```
chocolate = { 0.823529 ,  0.411765 ,  0.117647 }
lightcoral = { 0.941176 ,  0.501961 ,  0.501961 }
slateblue = { 0.415686 ,  0.352941 ,  0.803922 }
```

There are 115 colors in total. For the full list, see the jit.gl.lua Color Bindings documentation. The main functions in the color module are `RGBtoHSL` and `HSLtoRGB`. While colors in both spaces are defined by three values, an optional fourth value representing the alpha channel can also be passed in. The alpha channel is not involved in any of the calculations, but is simply passed through untouched.

```
hsl = RGBtoHSL(rgb)
hsl = RGBtoHSL(r, g, b)
hsla = RGBtoHSL(rgba)
hsla = RGBtoHSL(r, g, b, a)

rgb = HSLtoRGB(hsl)
rgb = HSLtoRGB(h, s, l)
rgba = HSLtoRGB(hsla)
rgba = HSLtoRGB(h, s, l, a)
```

For example, the code below converts an RGBA color to an HSLA color, lightens it, and then gets back the result into RGBA space.

```
color = { 1 ,  0.2 ,  0.2 ,  0.5 }
hsla = RGBtoHSL(color)
  -- lighten the color
hsla[ 3 ] = hsla[ 3 ]* 1.1
color = HSLtoRGB(hsla)
```

The other three functions in the color module are for manipulating HSL colors. They are designed so that the color manipulation functions can be chained without having to assign any results to a variable. These functions are `hue`, `saturate`, and `lighten`.

```
res = hue(hsla, hue_offset)
res = saturate(hsla, saturation_scale)
res = lighten(hsla, luminance_scale)
```

`hue` offsets the hue of an HSLA color by a given amount. `saturate` and `lighten` scale the saturation and luminance components of an HSLA color respective.

```
-- lighten the color
color = HSLtoRGB(lighten(RGBtoHSL{ 1 ,  0.2 ,  0.2 ,  0.5 },  1.1 ))
```

The define Message

Using define 724
Recovering the Original 726
Storing Your define in a Package 727

The **define** message to the `max` object (see Controlling Max with Messages) lets you declare an object that loads a file of code for one of Max's powerful supported language-based objects: JavaScript (v8), Gen, and jit.gl.slab.

Once defined the object can have its own help file and reference page and will appear in object box autocompletion.

The define message was used to create the **jit.fx** series of shader-based effects using a combination of v8 and jit.gl.slab.

Using define

Text Substitution

> For this example we are including the leading semicolon needed if you are entering a message to max in a message box. Later when you create a text file with your defines in it, you will omit the leading semicolon.

Suppose you have a JavaScript file called mycode.js. Here's how you can use the define message so that merely typing mycode will create a v8 object with `mycode.js` as an argument.

```
; max define mycode v8 mycode.js
```

The word after `define` is the name of the object you'll type into an object box, in this case `mycode`. Everything after that will become the actual contents of the object box, but it's not substituted for the text you type in; instead it's hidden in the background.

Arguments

Any arguments you type after the defined object name will be appended after the specified subtitution text. For example, if you typed `mycode 1 2 3` the resulting object box (which remains a secret) contains `v8 mycode.js 1 2 3`.

Typed-in Attribute Handling

It's also possible to specify preset values for typed-in attributes. To continue with our JavaScript example, imagine that mycode.js declares two attributes:

```
declareattribute("attr1");
declareattribute("attr2");
declareattribute("attr3");
```

You can set the initial values for any of these attributes as part of your define.

```
; max define mycode v8 mycode.js @attr1 25 @attr2 freedie
```

If the user of your define also adds arguments to their use of the define, those are substituted after the arguments in the define (in this case, `mycode.js`) but *before* the typed-in attribute values. With the above define, `mycode 1 2 3` would produce
`v8 mycode.js 1 2 3 @attr1 25 @attr2 freddie`

Finally, if the user of your define includes typed-in attributes, those are placed at the very end. And, if they supply values for attributes already in the define, the user's attribute values are used instead of the ones in the define. As an example, if the user enters `mycode 1 2 3 @attr3 1000` the resulting object will be created with
`v8 mycode.js 1 2 3 @attr1 25 @attr2 freddie @attr3 1000`. And if the user enters

`mycode 1 2 3 @attr1 50 @attr3 1000` the result will be `v8 mycode.js 1 2 3 @attr1 50 @attr2 freddie @attr3 1000`.

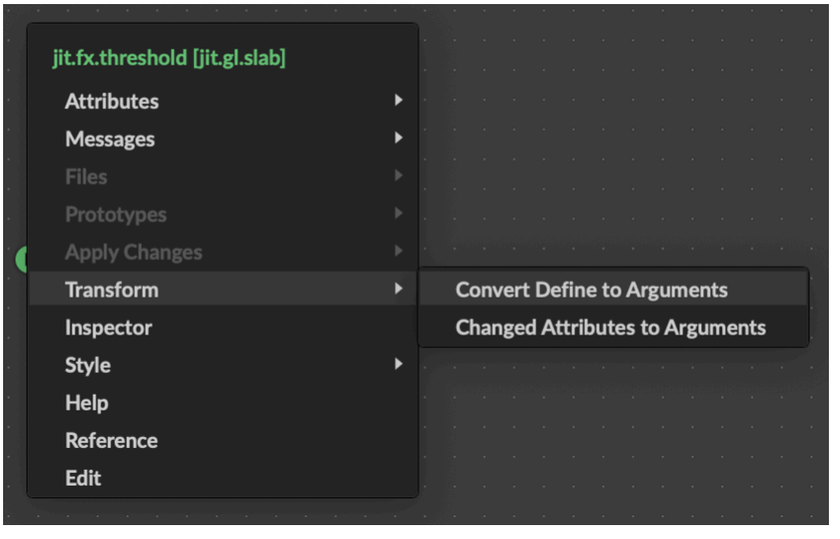
For Gen, typed-in attributes set the initial values of Gen **param** objects.

Inspector Behavior

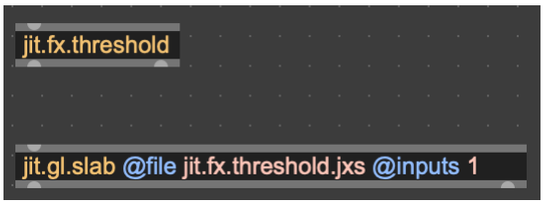
In order to put the focus on the settings the user can change, Gen, v8, and jit.gl.slab hide certain attributes in the inspector when you are using a define. For example, you typically can't read in a different file.

Recovering the Original

To recover the original object and arguments used to create an object using define, use the Object Action Menu and Transform > **Convert Define to Arguments**.



As an example, after using Convert Define to Arguments, the jit.fx.threshold object becomes a jit.gl.slab hosting the shader file `jit.fx.threshold.jxs`.



Storing Your define in a Package

Currently, the best way to create permanent access to a define is to use packages. Here is the recommended folder structure for a package that uses this feature:

```
package_folder/
├── init/
│   └── text file with max define messages
└── code/
  ├── JavaScript files
  ├── Shader files
  └── Gen files
```

The text file that will be stored in the init folder should end in `.txt`. Omit the leading semicolon required by the message box but include a trailing semicolon at the end of your message.

```
max define mycode v8 mycode.js @attr1 25 @attr2 freedie;
max define yourcode v8 yourcode.js;
```

After your package is assembled, copy it into the Max Packages folder. To test, relaunch Max, wait for the database to rebuild (as indicated in the left toolbar) and try typing your define into an object box. It should be displayed in autocompletion after typing the first two or three characters.

Controlling Max with Messages

Messages to max 728
MIDI Configuration Messages 753
Messages to midi 758
Messages to dsp 760
Messages to jitter 765

Just like you can control objects by sending them messages, you can also send messages directly to the Max application. This lets you do things like:

- Get/set the value of various **Max Preferences**
- List all of the currently loaded Max externals
- Move the mouse cursor
- Hide/show the menu bar

To send a message directly to Max, create a message box starting with a semicolon followed by the symbol `max`, `jitter`, or `dsp`, to send a message directly to max itself, the Jitter engine, or the audio engine. You don't need to connect the message box to anything, simply trigger it to send the message.



Messages boxes beginning with a semicolon send their message directly to the Max application

Messages to `max`

buildcollective

Builds a collective using a patcher previously opened with openfile. The collective is named with the output filename. The filename should be a full path (e.g. `Desktop/somecollective.mxf`), and it

must have the `.mxf` extension.

```
; max buildcollective patchername outputfile
```

Name	Type	Description
patchername	symbol	The symbol associated with the patcher that should be bundled into a collective
outputfile	symbol	Name of the output file

checkpreempt

Sends the current Overdrive mode to a receive object

```
; max checkpreempt $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

clean

Clear the dirty flag, causing Max not to show a *Save Changes* dialog when you close a window or quit, even if there are patchers that have been modified. This is useful in conjunction with the quit message below.

```
; max clean
```

clearmaxwindow

Clear the Max Console.

```
; max clearmaxwindow
```

closefile

Closes a patcher file previously opened with the openfile

```
; max closefile $patchername
```

Name	Type	Description
patchername	symbol	Symbol associate with the file

crash

Terminate Max, generating a standard crashlog. When relaunched, The Max application will perform standard crash recovery (if crash recovery is enabled in the Max preferences).

```
; max crash
```

db.exportmetadata

Write the metadata information currently stored in the database to a file. An optional argument can be used to specify a filename. If no filename is specified, the metadata is backed up to a file in your preferences folder.

```
; max db.exportmetadata $filename
```

Name	Type	Description
filename	symbol	(optional) backup file filename

db.importmetadata

Load metadata information from a previously stored file into the Max database. An optional argument can be used to specify a filename - when no argument is specified, Max will look for a backup file from a previous call to db.exportmeta in your preferences folder.

```
; max db.importmetadata $filename
```

Name	Type	Description
filename	symbol	(optional) backup file filename

db.reset

Rebuild the internal database, used by the File Browser and others.

```
; max db.reset
```

debug

Enable/disable sending Max's internal debugging output to the Max Console. This debug information may be of limited use for anyone who isn't debugging Max itself.

```
; max debug $onoff
```

Name	Type	Description
onoff	number	0 or 1 to enable/disable debug messages

disablevirtualmididestinations

Activate to disable the creation of virtual sources by the Core MIDI driver. When set to zero (enabling virtual sources) the virtual sources are created again.

```
; max disablevirtualmididestinations $offon
```

Name	Type	Description
offon	number	0 or 1 to disable/enable virtual sources

enablepathcache

Enable/disable Max's search path cache. This should only be done if you notice unusual behavior when opening files.

```
; max enablepathcache $onoff
```

Name	Type	Description
onoff	number	0 or 1 to enable/disable path cache

externaleditor

Sets the text editor used for editing text file content, such as saved coll files, text files and JavaScript code.

```
; max externaleditor $editorname
```

Name	Type	Description
editorname	symbol	Name of the external text editor

externs

Print all of the external objects currently loaded to the Max Console.

```
; max externs
```

fileformat

Associate a filename extension with a particular four-character filetype. For example, the message `max fileformat .tx TEXT` associates the extension .tx with TEXT (text) files. This allows a user to send a message `read george` and locate a file with the name `george.tx`. It also ensures that files with the extension `.tx` will appear in a standard open file dialog where text files can be chosen.

```
; max fileformat extension filetype
```

Name	Type	Description
extension	symbol	File extension to associate with a character code
filetype	symbol	Filetype character code

fixwidthratio

Set the ratio of the box to the width of the text when the user chooses *Fix Width* from the *Object* menu. The default value is 1.0. A value of 1.1 would make boxes wider than they needed to be, and a value of 0.9 would make boxes narrower than they need to be.

```
; max fixwidthratio $ratio
```

Name	Type	Description
ratio	number	box width to text width ratio

getarch

Send the currently running Max architecture (always 'x64') to the named receive object.

```
; max getarch $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getdefaultpatcherheight

Send the current default patcher height in pixels to the named receive object (See also the setdefaultpatcherheight message to Max.)

```
; max getdefaultpatcherheight $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getdefaultpatcherwidth

Send the current default patcher width in pixels to the named receive object (See also the setdefaultpatcherheight message to Max.)

```
; max getdefaultpatcherwidth $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getenablepathcache

Report whether the path cache is enabled to the named receive object. (See also the enablepathcache message to Max.)

```
; max getenablepathcache $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

geteventinterval

Report the event interval to the named receive object. (See also the seteventinterval message to Max.)

```
; max geteventinterval $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getfixwidthratio

Report the current fix width ratio value to the named receive object. (See also the fixwidthratio message to Max.)

```
; max getfixwidthratio $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getpollthrottle

Report the current poll throttle value to the named receive object. (See also the setpollthrottle message to Max.)

```
; max getpollthrottle $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getqueuethrottle

Report the current queue throttle value to the named receive object. (See also the setqueuethrottle message to Max.)

```
; max getqueuethrottle $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getrefreshrate

Report the current refresh rate in frames per second (fps) to the named receive object. (See also the refreshrate message to Max.)

```
; max getrefreshrate $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getruntime

Send a 1 to the named receive object if the current version of Max is a runtime version, and a 0 if not.

```
; max getruntime $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getslop

Send the scheduler slop value to the named receive object. (See also the setslop message to Max.)

```
; max getslop $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getsysqelemthrottle

Send the maximum number of patcher UI update events processed at a time to the named receive object. (See also the setsysqelemthrottle message to Max.)

```
; max getsysqelemthrottle $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getsystem

Send the name of the system (macintosh or windows) to the named receive object.

```
; max getsystem $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

getversion

Send the Max version number as a decimal value, which needs to be converted to a hexidecimal value (e.g. Max version 7.3.4 is reported as '1844'), and output from the named receive object.

```
; max getversion $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

hidecursor

Hides the cursor if it is visible.

```
; max hidecursor
```

hidemenubar

Hides the menu bar.

```
; max hidemenubar
```

htmlref

Look for a file called `<filename>.html` in the search path. If found, a web browser is opened to view the page. This opens *local* html files, not remote web addresses.

```
; max htmlref $filename
```

Name	Type	Description
filename	symbol	Name of the file to open

launchbrowser

Opens a web browser to view the given URL.

```
; max launchbrowser $url
```

Name	Type	Description
url	symbol	URL to open

maxcharheightforsubpixelantialiasing

Set a threshold font size (in points) for native subpixel aliasing. Since the look of subpixel antialiasing may be undesirable when working with large fonts as compared to regular antialiasing, this attribute lets you specify a threshold font size; if a font is larger than the specified size, it will be rendered using regular rather than subpixel antialiasing.

Note that Max honors your computer's system preferences - Max won't use subpixel aliasing if you've disabled it for your system. Setting this attribute value to zero value is 0 will always use regular antialiasing, and setting a very high value will always use subpixel antialiasing (unless it is disabled in system preferences).

```
; max maxcharheightforsubpixelantialiasing $pointheight
```

Name	Type	Description
pointheight	number	threshold font size (in points) for native subpixel aliasing

maxinwmenu

Enable/disable the special *Status* option in the *Window* menu bar item. This is only available when using runtime version of Max *and* an active custom menubar object. The *Status* option will allow users to see the Max Console. Defaults to 1 (enabled).

```
; max maxinwmenu $onoff
```

Name	Type	Description
onoff	number	Enable/disable the *Status* option to show the Max Console

maxwindow

Displays the Max Console. If the Max Console if not currently open, the window will be displayed. If the window is currently open, it will bring it to the front.

```
; max maxwindow
```

midilist

Prints the names of all current MIDI devices in the Max Console. (See also MIDI Messages to Max, below.)

```
; max midilist
```

notypeinfo

(macOS only) Enable/disable saving files with traditional macOS four-character type information. By default, Max does save this information in files.

```
; max notypeinfo $onoff
```

Name	Type	Description
onoff	number	Enable/disable saving files with traditional macOS four-character type information

objectfile

The word objectfile, followed by two symbols that specify an object name and a file name, Create a mapping between an external object and its filename. For example, the `*~` object is in a file called `times~` so at startup Max executes the command `max objectfile *~ times~`.

```
; max objectfile objectname filename
```

Name	Type	Description
objectname	symbol	Name of the object to associate
filename	symbol	External object filename to associate

openfile

The word openfile, followed by two symbols that specify an reference name and a file name or path name, attempts to open the patcher with the specified name. If successful, the patcher is associated with the reference symbol, which can be passed as argument to the buildcollective and closefile messages to Max. The `openfile` message is intended for batch collective building.

```
; max openfile namehandle filename
```

Name	Type	Description
namehandle	symbol	Name to associate with the open patcher
filename	symbol	Name of the patcher to open

paths

List the current search paths in the Max Console. There is a button in the File Preferences window that does this.

```
; max paths
```

preempt

Toggle Overdrive mode.

```
; max preempt $onoff
```

Name	Type	Description
onoff	number	Enable/disable overdrive

pupdate

Move the mouse cursor to a global screen position.

```
; max pupdate xposition yposition
```

Name	Type	Description
xposition	number	Horizontal screen coordinate
yposition	number	Vertical screen coordinate

purgemididevices

Purge the missing MIDI device cache. Max maintains a cache of the MIDI Setup settings for known, but detached MIDI devices. Send this message to 'forget' any missing devices.

```
; max purgemididevices
```

quit

Quit the Max application, equivalent to choosing *Quit* from the *File* menu. If there are unsaved changes to open files, and you haven't sent Max the clean message, Max will ask whether to save changes.

```
; max quit
```

refresh

Update all Max consoles.

```
; max refresh
```

refreshrate

Set the rate limit, in frames per second, at which the UI for user interface objects is updated. Better visual performance can be achieved - at the cost of a slight performance decrease in Jitter, and little or no performance decrease for audio processing - by specifying a higher frame rate.

```
; max refreshrate $rate
```

Name	Type	Description
rate	number	Refresh rate (fps)

relaunchmax

Close and relaunch the Max application.

```
; max relaunchmax
```

runtime

Conditionally send a message to Max if the given value matches the "runtime" status of Max. For example, the message `; max runtime 0 externs` will send the `externs` message to Max *only* if the active Max application is not a runtime-only version.

```
; max runtime runtimeflag message
```

Name	Type	Description
runtimeflag	number	Flag to match for the current runtime status
message	symbol	Message to send if the runtime flag matches

sendinterval

Send the current scheduler interval to the named receive object.

```
; max sendinterval $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

sendapppath

Send the path of the Max application to the named receive object.

```
; max sendapppath $receive
```

Name	Type	Description
receive	symbol	Name of the target receive object

setdefaultpatcherheight

Set the default patcher height in pixels. Must be greater than 100.

```
; max setdefaultpatcherheight $height
```

Name	Type	Description
height	symbol	Default patcher height

setdefaultpatcherwidth

Set the default patcher width in pixels. Must be greater than 100.

```
; max setdefaultpatcherwidth $width
```

Name	Type	Description
width	symbol	Default patcher width

seteventinterval

Set the time between invocations of the event-level timer (The default value is 2 milliseconds). The event-level timer handles low priority tasks like drawing user interface updates and playing movies.

```
; max seteventinterval $interval
```

Name	Type	Description
interval	symbol	Low priority timer interval

setmixergbitmode

Set the state of the Enable Mixer Crossfade preference for top-level patcher mixers. A value of 0 sets the preference to `Off`, 1 to `On`, and 2 to `Auto`.

```
; max setmixergbitmode $mode
```

Name	Type	Description
mode	number	Enable Mixer Crossfade preference

setmixerlatency

Set the Mixer Crossfade Latency preference for top-level patcher mixers to the specified number of milliseconds.

```
; max setmixerlatency $latency
```

Name	Type	Description
latency	number	Mixer latency in milliseconds

setmixerparallel

Enable/disable the Enable Mixer Parallel Processing preference for top-level patcher mixers.

```
; max setmixerparallel $onoff
```

Name	Type	Description
onoff	number	Enable/disable mixer parallel processing

setmixerramptime

Set the Mixer Crossfade Ramp Time preference for top-level patcher mixers to the specified number of milliseconds.

```
; max setmixerramptime $time
```

Name	Type	Description
time	number	Ramp time in milliseconds

setmirrortoconsole

Enable/disable mirroring of Max Console posts to the system console (default off). The system console is available on macOS using Console.app, or on Windows using the DbgView program (free download from Microsoft).

```
; max setmirrortoconsole $onoff
```

Name	Type	Description
onoff	number	Enable/disable mirror to system console

setsleep

Set the time between calls to get the next system event, in 60ths of a second. The default value is 2.

```
; max setsleep $interval
```

Name	Type	Description
interval	number	Interval in 60ths of a second between system event polls

setpollthrottle

Set the maximum number of events the scheduler executes each time it is called (The default value is 20). Setting this value lower may decrease accuracy of timing at the expense of efficiency.

```
; max setpollthrottle $chunksize
```

Name	Type	Description
chunksize	number	Number of events the executed per scheduler tick

setqueuethrottle

Set the maximum number of events handled at low-priority each time the low-priority queue handler is called (The default value is 2). Changing this value may affect the responsiveness of the user interface.

```
; max setqueuethrottle $chunksize
```

Name	Type	Description
chunksize	number	Number of events executed per low-priority queue handler tick

setslop

Set the scheduler slop value - the amount of time a scheduled event can be earlier than the current time before the time of the event is adjusted to match the current time. The default value is 25 milliseconds.

```
; max setslop $slop
```

Name	Type	Description
slop	number	Slop time in milliseconds

setsysqelemthrottle

Set the maximum number of patcher UI update events to process at a time. Lower values can lead to more processing power available to other low-priority Max processes, and higher values make the user interface more responsive (especially when using many bpatchers).

```
; max setsysqelemthrottle $chunksize
```

Name	Type	Description
chunksize	number	Number of patcher UI events to process per tick

showcursor

Show the cursor if it is hidden.

```
; max showcursor
```

showmenubar

Show the menu bar if it was hidden with hidemenubar

```
; max showmenubar
```

size

Print the number of symbols in the symbol table in the Max Console.

```
; max size
```

system

Conditionally send a message to Max if the Operating System condition matches. The operating system can be `windows` or `macintosh`. For example, the message `; max system windows externs` will send the message `externs` to Max only on a Windows operating system.

```
; max system systemflag message
```

Name	Type	Description
systemflag	number	Flag to match for the current operating system `windows` or `macintosh`
message	symbol	Message to send if the system flag matches

useexternaleditor

Enable/disable using an external editor for text. If enabled, any situation where an external editor can be used will launch the editor. If disabled, an external editor will only be used when selected from the menu.

```
; max useexternaleditor $onoff
```

Name	Type	Description
onoff	number	Enable/disable the external editor

useslowbutcompletesearching

Enable/disable complete file searching. When enabled, it causes files not found in Max's cache of the search path to be searched in the file system. This is necessary only in extremely rare cases where the file cache does not update properly. One such case is copying a file into the search path using a version of the Mac OS prior to 10.5.5 over a network. This option may cause patcher files to be loaded more slowly. The setting defaults to off with each launch of the application, and is not stored in the user's preferences.

```
; max useslowbutcompletesearching $onoff
```

Name	Type	Description
onoff	number	Enable/disable the complete file searching

MIDI Configuration Messages

createoutport

Creates a new port for the specified driver. This is only possible on macOS machines. Windows is unsupported. On macOS, specifying the `coremidi` driver name creates a virtual output port you can use to communicate with other MIDI applications, while specifying the `augraph` driver name creates another port exclusively assigned to the DLS synthesizer.

```
;#SM createoutport portname drivername
```

Name	Type	Description
portname	symbol	Name of the port to create
drivername	symbol	Name of the driver to create

deleteoutport

On macOS, deletes a port created with the createoutport message. `drivername` and `portname` should be the same as the arguments originally passed to `createoutport`.

```
;#SM deleteoutport portname drivername
```

Name	Type	Description
portname	symbol	Name of the port to delete
drivername	symbol	Name of the driver to delete

driver loadbank

On macOS, loads a type 1 or 2 DLS Bank, where `filename` is the name of an existing DLS bank file, and `portname` is the name of the port that will use this bank. If portname is omitted, all DLS ports will use the bank. The folder `/Library/Audio/Sounds/Banks` is added to the search path when looking for a DLS bank file.

```
;#SM driver loadbank filenaem portname
```

Name	Type	Description
filename	symbol	Name of the DLS bank file to load
portname	symbol	Name of the port that will use the bank

driver loadbank 0

Load the default General MIDI DLS bank

```
;#SM driver loadbank 0 $portname
```

Name	Type	Description
portname	symbol	Name of the port that will use the bank

driver reverb

Enable/disable reverb. Off by default in `augraph`

```
;#SM driver reverb onoff portname
```

Name	Type	Description
onoff	number	Enable/disable reverb
portname	symbol	Name of the port

driver latency

(midi_mme only) Set the MIDI Output Latency in milliseconds

```
;#SM driver latency mstime portname
```

Name	Type	Description
mstime	number	Latency in milliseconds
portname	symbol	Name of the port

inportinfo

Send information about MIDI input ports to the named receive objects. The information is contained in an list message with the following elements:

Element	Type
the port's name	symbol
the port's driver name	symbol
the port's unique ID	int
the port's abbreviation	int
the port's channel offset	int
whether the port is enabled or disabled	one if enabled, zero if disabled
whether the port was created dynamically	one if yes, zero if no

```
;#SM inportinfo portname receivename
```

Name	Type	Description
portname	symbol	Name of the port
receivename	symbol	Name of the target receive object

outportinfo

Send information about MIDI output ports to the named receive objects. The information is contained in an `infolist` list message. See inportinfo for a description if the `infolist` list elements.

```
;#SM outportinfo portname receivename
```

Name	Type	Description
portname	symbol	Name of the port
receivename	symbol	Name of the target receive object

createinport

(macOS only) Add a virtual MIDI input port, where `portname` is the name you assign to the port, and drivername should be set to `coremidi`. Other MIDI applications can send messages to Max using this port.

```
;#SM createinport portname drivername
```

Name	Type	Description
portname	symbol	Name of the port to create
drivername	symbol	Name of the driver, should always be `coremidi`

deleteinport

Deletes a port created with the createinport message. `drivername` and `portname` should be the same as the arguments originally passed to `createinport`.

```
;#SM deleteinport portname drivername
```

Name	Type	Description
portname	symbol	Name of the port to create
drivername	symbol	Name of the driver, should always be `coremidi`

> Ports created with the `createoutport` and `createinport` messages are not saved as a part of your MIDI setup preferences.

Messages to `midi`

These are technically messages to Max, but specifically to MIDI configuration object owned by Max. So to send one of these messages, use a format like:

```
; max midi <message> <message-arguments>
```

portabbrev

Set the abbreviation for a given MIDI port.

```
; max midi portabbrev inorout portname $abbrev
```

Name	Type	Description
inorout	symbol	Must be `innum` for an input port or `outnum` for an output port
portname	symbol	Name of the port
abbrev	number	The abbrevation to use for the port. 0 is for no abbrev (- in menu), 1 for 'a' and 26 for 'z'.

portenable

Enable/disable a given MIDI port. All ports are enabled by default.

```
; max midi portenable portname onoff $inorout
```

Name	Type	Description
portname	symbol	Name of the port
onoff	number	1 for enable, 0 for disable
inorout	number	1 for output, 0 for input

portoffset

Similar to portabbrev, but offset is the channel offset added to identify input or output ports when a MIDI object can send to or receive from multiple ports by channel number. Must be a multiple of 16 (e.g. `max midi portoffset innum PortA 16` sets the channel offset for PortA device to 16).

```
; max midi portoffset inorout portname $offset
```

Name	Type	Description
inorout	symbol	Must be `innum` for an input port or `outnum` for an output port

portname	symbol	Name of the port
abbrev	number	The port offset, which must be a multiple of 16.

Messages to `dsp`

cpulimit

Sets a utilization limit for the CPU. Above this limit, MSP will not process audio vectors until the utilization comes back down, causing a click. If the cpu limit is set to either 0 or 100, there will be no limit checking done.

```
; dsp cpulimit $limit
```

Name	Type	Description
limit	number	The cpu limit in a range of `[0-100]`

inremap

Maps a physical device input channel to a logical input channel.

```
; dsp inremap physical_input logical_input
```

Name	Type	Description
physical_input	int	Physical device input number
logical_input	int	Logical device input number

iovs

Sets the I/O vector size.

```
; dsp iovs $size
```

Name	Type	Description
size	type	Vector Size, should be a power of 2

open

Opens the Audio Status window.

```
; dsp open
```

outremap

Maps a physical device output channel to a logical output channel.

```
; dsp outremap physical_output logical_output
```

Name	Type	Description
physical_output	int	Physical device output number
logical_output	int	Logical device output number

set

Turns the audio on (`1`) or off (`0`). It is equivalent to clicking on a ezadc~ or ezdac~ object.

```
; dsp set $status
```

Name	Type	Description
status	int	New audio status - on / off

setdriver

Sets a new audio driver based on its index into the currently generated menu of drivers created by he adstatus driver object.

If the argument is a symbol instead of an index and names a valid driver, the new driver is selected by name. An additional symbol argument may be used to specify a "subdriver" (for example, ASIO drivers use *ASIO* as the name of the driver and *PCI-324* as a subdriver name that specifies a specific device).

```
; dsp setdriver index subdriver_name
```

Name	Type	Description
index	int or symbol	The index or name of the driver
optional subdriver_name	int or symbol	The name of the subdriver

sigvs

Sets the I/O signal vector size.

```
; dsp sigvs $size
```

Name	Type	Description
size	type	Vector Size, should be a power of 2

sr

Sets a new sampling rate in Hertz.

```
; dsp sr $samplerate
```

Name	Type	Description
samplerate	number	Samplerate in Hertz

start

Turns the audio on.

```
; dsp start
```

status

Opens the Audio Status window.

```
; dsp status
```

stop

Turns the audio off.

```
; dsp stop
```

takeover

Turns Scheduler in Audio Interrupt mode on (`1`) or off (`0`). It is equivalent to clicking on the `Audio Interrupt` checkbox in the Audio Status window. `Overdrive` must be on in order for this change to be reflected.

```
; dsp takeover $status
```

Name	Type	Description
status	int	On (`1`) or off (`0`)

timecode

Starts (`1`) or stops (`0`) timecode reading by any audio drivers that support the feature (ASIO 2).

```
; dsp timecode $status
```

Name	Type	Description
status	int	Start (`1`) or stop (`0`) reading

wclose

Closes the Audio Status window.

```
; dsp wclose
```

Messages to `jitter`

cursor

> Macintosh only

Toggles cursor visibility on / off.

```
; jitter cursor $status
```

Name	Type	Description
status	int	On (`1`) or Off (`0`)

html_ref

Launches the reference file for an object in the Max search path.

```
; jitter html_ref $name
```

Name	Type	Description
name	symbol	Name of the object

javaload

Toggles Jitter Java support on / off.

```
; jitter javaload $status
```

Name	Type	Description
status	int	On (`1`) or Off (`0`)

launch_browser

Launches the specified URL in the default system web browser.

```
; jitter launch_browser $url
```

Name	Type	Description
url	symbol	URL to open

menubar

> Macintosh only

Toggles menubar visibility on or off. Similar to Max's `showmenubar` , and `hidemenubar` messages.

When using in conjunction with the jit.window object's fullscreen attribute, it is recommended that the jit.window object's *fsmenubar* attribute is used instead of Jitter's `menubar` message, in order to prevent possible "pixel trash".

```
; jitter menubar $status
```

Name	Type	Description
arg	int	On (`1`) or Off (`0`)

parallel

Toggles parallel processor support. The default is on if the machine has multiple processors (or cores), and otherwise off.

```
; jitter parallel $status
```

Name	Type	Description
arg	int	On (`1`) or Off (`0`)

parallelthreads

Specifies the number of threads used for parallel processor support. Default is the number of processors (or cores).

```
; jitter parallelthreads $count
```

Name	Type	Description
count	int	Number of threads to use

parallelthresh

Specifies matrix cellcount above which parallel processors are used. Default is 10000 cells.

```
; jitter parallelthresh $cellcount
```

Name	Type	Description
cellcount	type	Treshold at which parallel processors are used.

pollthrottle

Sets the number of scheduler events to process per scheduler tick. Equivalent to Max's `setpollthrottle` message.

```
; jitter pollthrottle $count
```

Name	Type	Description
count	int	Events to process per scheduler tick

queuethrottle

Sets the number of low priority queue events to process per low priority queue service. Equivalent to Max's `setqueuethrottle` message.

```
; jitter queuethrottle $count
```

Name	Type	Description
count	int	Number of low priority queue events to process

REPL

Introduction 770
Accessing the REPL 770
Using the REPL 771
REPL History 773
Shortcuts 773
Patcher-Specific and Global Targets 774
receive Objects Are Global 775
Scripting Name Targets Are Patcher-Specific 776
Evaluating JavaScript 777
JS Redirection 779
Patcher Scripting with the <p> Target 779
Object Values and Attributes 780
The repl Object 780

Introduction

The Max REPL (an acronym for *Read-Evaluate-Print Loop*) is a text entry area at the bottom of the Max Console that lets you interact with patchers via text commands.

Using the REPL, you can:

- Send messages to Max objects in your patcher
- Evaluate JavaScript expressions
- Control the Max environment with text-based commands
- Access the documentation

Accessing the REPL

To show the REPL, open either the sidebar Max Console or the standalone Max Console.

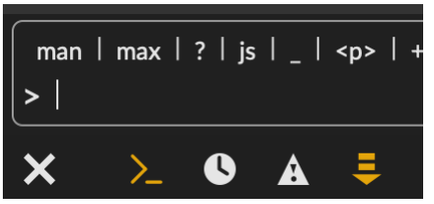


If the command prompt at the bottom of the Max console is not showing, click the REPL icon in the bottom Max console toolbar to show it.



Using the REPL

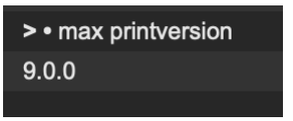
Let's see a simple example of the REPL in action. Click once in the text entry box at the bottom of the Max Console to highlight the REPL for text input. The caret will begin blinking.



Enter `max printversion` and press Return.

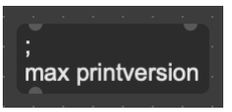


After pressing Return, you'll see `max printversion` posted to the Max console followed by the current version number. For example:



What's going on in this example? The REPL is sending a **message** to a **named object**. The "read" in this case is reading what you type. It's then "evaluating" the text you typed by sending the message. Finally, the result of the message is (often) "printed" back to the Max console. And after reading, evaluating, and printing, the REPL is ready for another command (the "loop").

In the above example `max` is the named object and `printversion` is the message. This simple form -- receiver followed by message -- is the same as a message box that begins with a semicolon.



The REPL is a faster and simpler way to send messages to `max` and other named objects because you don't need to create and then click on a message box, and you don't need a semicolon.

Finally, notice that after you press return, the text entry field clears, but the cursor is still blinking, waiting for another command. This permits you to send REPL commands in quick succession.

Quick Documentation Access

Ever forget what a Max object does? The REPL can help. Enter `man funnel` and press Return. (If `man` is too much typing, you can also use `?`.) The Max window posts the following:

funnel: Identifies the inlet of incoming data. It can be used to store values into a table or coll based on their source, or used to set a destination with an object such as spray.

REPL History

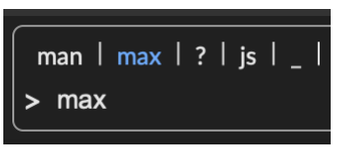
To repeat a previous command, you can use the **up arrow** key to scroll through the history of what you've previously typed into the REPL. Pressing the up arrow once will enter the most recently typed command into the REPL's text entry box, ready for you to send it again by pressing the Return key. The **down arrow** scrolls the opposite direction back to the most recently typed command. Once you pass the most recent command, pressing the down arrow will clear the text entry box.

Shortcuts

Above the text entry field of the REPL you'll see the **shortcuts bar**:



These **shortcuts** are named objects (*targets*) where you can send messages. Clicking on a shortcut makes it persistent, so you can send it repeated messages without having to type the object name. For example, click on the **max** shortcut. It will appear at the beginning of the text entry box:



Type `printversion` after the word `max` and press return. Now, after the version number is posted to the Max console, `max` remains at the beginning of the text entry box, ready for you to send it another command.

There are several other standard shortcuts that appear in the REPL area of the Max console:

- **?** is the same as **man**, for documentation queries (currently limited to object descriptions)
- _ is a way to send messages to the REPL itself
- **man** (short for *manual*) is for documentation queries (currently limited to object descriptions)
- **js** is the global JavaScript evaluator
- **max** is the `max` object; messages are documented in Controlling Max with Messages
- **<p>** is the patcher for which this console is the sidebar (**<p>** does nothing in the standalone Max console window)
- **+** opens a menu showing other available named object targets; if none are currently available, the menu will not appear when you click **+**

Managing Shortcuts

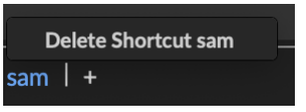
To set the current shortcut, click on the desired name (or choose it from the **+** menu). It will highlight and the text will appear at the beginning of the text entry box.

To clear the current shortcut, click its name in the shortcuts bar again. The text at the begining of the text entry box will be removed.

To make the current named object the persistent REPL target so it doesn't go away when you pretty Return, press Shift-Return instead.

To clear the current shortcut after sending it a message, press Option- or Alt-Return

To remove non-default shortcuts (ones you added by the **+** menu) from the shortcuts bar, hold down the control key and click on the shortcut name until you see a menu that says *Delete Shortcut*.



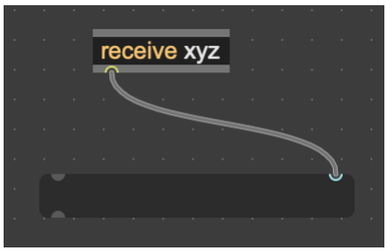
Patcher-Specific and Global Targets

Two named object REPL targets already introduced are `max` and `man`. These are **global** targets that work in any Max console window, whether it's the sidebar or the standalone. In fact, all the standard shortcuts listed above are global except for **<p>**.

receive Objects Are Global

The name argument to a receive object is its global name. While you can have many receive objects that share the same name in different patchers, using a send object with the same name will send to all receive objects. The same is true for the REPL.

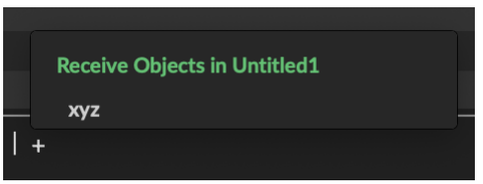
Make the following simple patcher:



You can use `xyz` as a named object target in the REPL in any Max console window, not just the one connected to the patcher where the receive object lives.

Enter `xyz 74` and press Return. The message box text will be set to 74. This does the same thing as entering `; xyz 74` into a message box and clicking it.

Once a receive object exists in the Max environment, it will be available as a shortcut in the menu you see by clicking **+**.

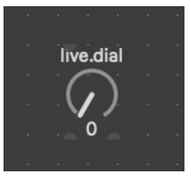


You can send any message to a receive object via the REPL, not just numbers. Given the above example, `xyz cat + dog` will set the message box contents to `cat + dog`.

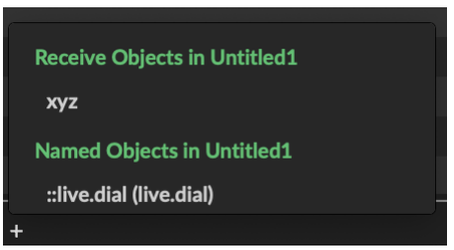
Note that unlike a message box, the REPL does not currently support sending multiple messages in sequence to the same named object using commas.

Scripting Name Targets Are Patcher-Specific

In addition to receive objects, you can use the REPL to send messages to any object with a scripting name. Here's a simple example. In a new patcher window, open the sidebar Max console. Then return to the patcher and create new a live.dial object.



The default scripting name for a live.dial is `live.dial`. But we don't even have to look in the inspector to see what it's called because having created the live.dial it will show up in the shortcut menu under **Named Objects** when we click the **+** above the text entry area in the REPL.

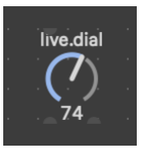


Choosing **::live.dial** from the **+** menu will make it the current shortcut (and add it to the shortcuts bar).



Objects with scripting names are preceded with two colons (**::**) to distinguish them from receive objects.

With `::live.dial` are the REPL target, type `74` and press return. The dial is now set to 74.



Using a scripting name gives you direct access to an object, so you can also set its attributes and send the object any message it understands. Let's set the dial to a deep blue color -- enter `activedialcolor 0 0 1 1` after `::live.dial` and press Return.



Evaluating JavaScript

The Global js Environment

Using the special `js` global target, the REPL has its own JavaScript environment that can be used for evaluating math expressions and defining variables.

When working with the JavaScript evaluator, it's helpful to click on **js** in the shortcut bar to make it the current shortcut.



Here are a couple of examples of using the evaluator:

- Enter `js 3 + 4` and press Return. The Max console should print `7`.
- Need a random number? Enter `js Math.random()` and press Return. You'll get a number between 0 and 1.
- You can declare a variable for later use. Enter `let b = 20` and press Return. The Max console will print `undefined` because there was no result for this variable assignment operation.
- Now you can retreive the variable `b` and use it in other expressions. For example, try `b * 100` and you should see `2000`.

In the Redirection discussion below, we'll see how you can send the result of a Javascript expression to a named object target.

Javascript with Named v8 Objects

If you give a v8, v8ui, or v8.codebox JavaScript object a scripting name, you can use the REPL to interact with the environment in that object using JavaScript. You can call functions defined in the object's script or get and set the values of variables or properties.

A v8 object with a scripting name, like any other object, can be a target for Max messages. For example, if you define a function bang inside a script in v8 object named `calum` you can call that function with `::calum bang`.

By using the special `.js` "sub-target" you can interact with the v8 object in JavaScript itself. For example, to call the defined function `bang` using the `.js` sub-target: `::calum.js bang()`.

Note the parentheses are necessary because you are not sending `calum` a Max message, you are using Javascript to call a function inside the object's script itself. Similarly if you have declared a variable at global scope in your script named `bob` you can print out its value with `::calum.js bob`. Or you can change the value of `bob` with `::calum.js bob = 74`.

Everything else you can do in the global JavaScript evaluator is possible with a named v8 object and the `.js` sub-target.

JS Redirection

Using the **redirect operator** (**<**) you can send the result of evaluating a JavaScript expression to a named object. For example, we can generate a random value between 0 and 127 with `js Math.random() * 127`. Now we want to send the random value we generated using JavaScript to a live.dial.

To set a live.dial object with scripting name `::live.dial` to a random value between 0 and 127, enter `::live.dial < js Math.random() * 127`.

The **<** *redirects* the result of the Javascript expression to become the *message* sent to the dial.

The result of expressions evaluated by named v8 objects can also be redirected as Max messages. The same example as above using a v8 object with scripting name `calum`:
`::live.dial < ::calum.js Math.random() * 127`

The REPL currently supports only one redirection per line. One way to circumvent this limitation would be to write a function in a script in a v8 object that sends its arguments out an outlet as a Max message. This would allow you to call that function with more than one argument. For example, to generate a random color sent to a function called `sendit()`, you could do something like this: `::calum.js sendit(Math.random(), Math.random(), Math.random(), 1.0)`

Patcher Scripting with the <p> Target

The REPL target **<p>** represents the patcher next to the sidebar Max console where you are typing commands. It is an undefined name in the standalone Max console window.

Using the same messages you can send to the thispatcher object or the `patcher` object in the JavaScript API, the REPL allows you control and script your patcher. Here are a couple of simple examples:

- To change the unlocked background color of the patcher to a dull medium gray, enter `<p> editing_bgcolor 0.5 0.5 0.5 1.0`.
- To create a new button object with a scripting name, enter `<p> script newobject button @varname ddd`. Now you can send the newly created button messages such as `::ddd bang` or `::ddd bgcolor 0.7 0.2 0.3 1.0`

Object Values and Attributes

To change the value of an object's attribute, type the attribute name and value after the object name. For example, to set the `ignoreclick` attribute of a slider with scripting name `biffy`: `::biffy ignoreclick 1`

As a reminder, you can also change the value of the slider with `::biffy 25`.

You can use redirection to *access* the value of an object or one of its attributes and send that value to another object. In this example, we have a toggle with scripting name `togo` that we will set with the current value of the `ignoreclick` attribute of the slider named `biffy`. To access an attribute of an object, use the `::` notation to separate the attribute name from the scripting name.

```
::togo < ::biffy::ignoreclick
```

If have set `ignoreclick` in the slider to 1, after entering this command, the toggle should have a value of 1.

To access the value of an object such as a slider, the name itself is sufficient after the redirect. In this example, we have another slider named `coppy` that we want to set to the current value of `biffy`:

```
::coppy < ::biffy
```

The repl Object

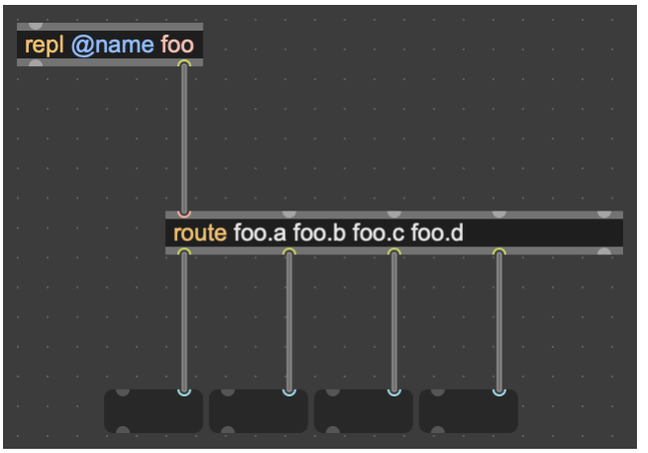
The repl object is an alternative to the receive object with some REPL-specific features, including the ability to set either global or patcher-specific scope and the ability to use sub-targets.

Create a repl with a name:



Once created, the object is available as a target `foo` in the REPL. However, you can also append additional names (*sub-targets*) to `foo` using dot notation. For example, `foo.a 50 100` will still be sent to the `repl @name foo` object.

The right outlet of repl precedes the message you send from the REPL with the target name. You can use the target name with the route to direct messages aimed at different sub-targets to different outlets.



Sharing

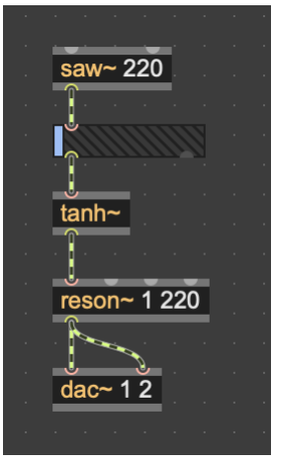
Sharing Patchers

Copy Compressed 783
Screenshots and Image Export 784
Screen Recording 786
Projects 786
Tools and Abstractions 787
Packages 787

There are several ways to share patchers, depending on whether you want to share a large project with multiple dependencies, or a just a small number of Max objects.

Copy Compressed

To share a small selection of objects, select the object and select *Copy Compressed* from the *Edit* menu. This copies the objects into compressed text that can be easily shared.



In compressed form, this group of objects looks like this:

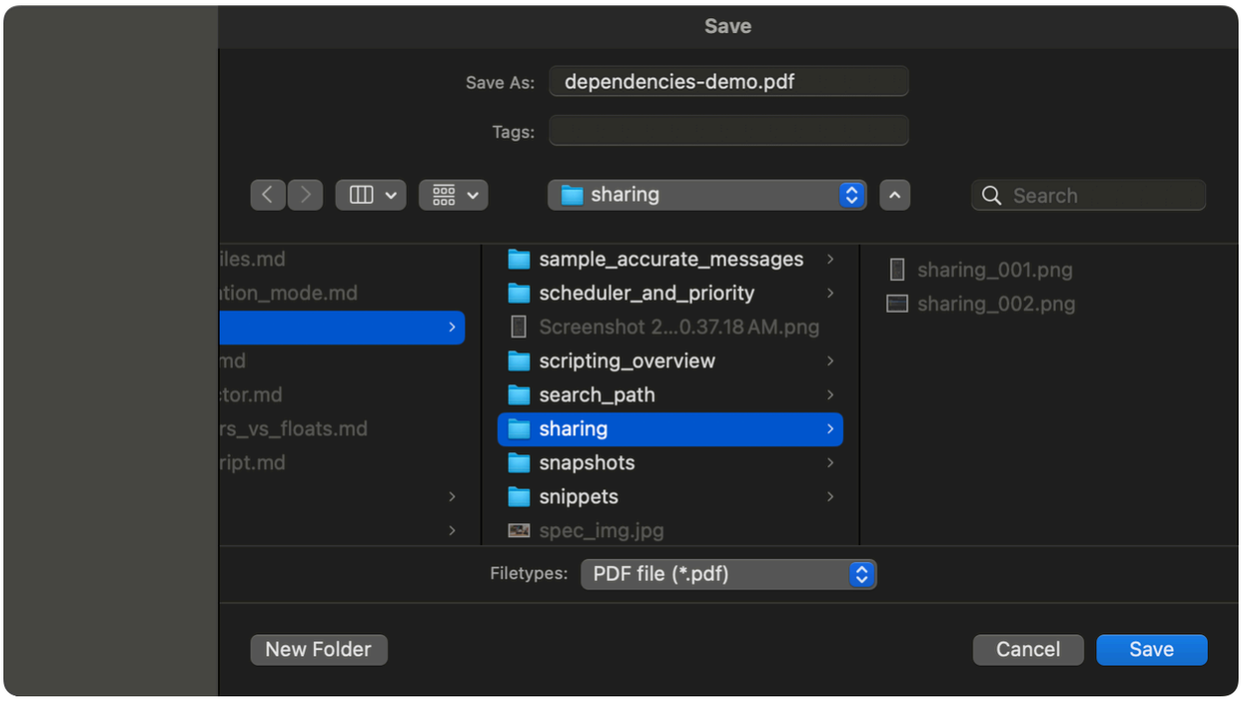
```
<pre><code>
----------begin_max5_patcher----------
438.3ocuUsraiCCC7r8WgfN6sPV4Uy9qTTTv3HjnBaJCI4zTTz7suRT1nY2l
GtHA6EKvwLb3Xpg4i7L9Jydkiy9M6IVV1G4YYDTDHqONi2.6qpAGkFGUuYV8
JuH8JuZumfWCUGXkL4vKZAe0VMt4Eqpxmp+iKePTvjylDOlMiBjOHXO2+Szq
oJEp9ujSGpC10nwZkmHW9EnoyOfJhfelmGeTbapvpbFjzgTbckHdLdrX4EUx
jSpjomTIk8nIH+6spDebmdCB0b1y2Oo5Ab6gqpwx4SiGStrFkmTik++z3FPi
iPLSDz.aAEr3Lh4nAuEZTdk8EEBqpo9TLdgNbYsoq1qq1BHpp2AVMf9iqz49
NTv3204sCd6vntVurL9bt3RS7x4+.+4OZhSrwq03+tUh52H9eKcmoyVMTvAC
G6qldsx40H30F73jB6WBt7gjL10J64Wkb+oV7cpKuMpkig59961XRLFlj2Al
hWxtNShuwT5JDz1tSYc8YSjD7IuZnO1KKnPMlBIuXX4+N8P9IDvFrH9f+nyR
MFe+7zeLwaLgwF1o6mbA4Enj7fXXmgqERJgrp4el+GfYabKx
-----------end_max5_patcher-----------
</code></pre>
```

If you copy this text and paste it into a patcher, Max will re-create the original objects.

> You can also create a new patcher from a block of compressed text. Select *New From Clipboard* from the *File* menu to create a new patcher using the existing contents of the clipboard. This is especially useful for large patchers that have been shared in compressed form.

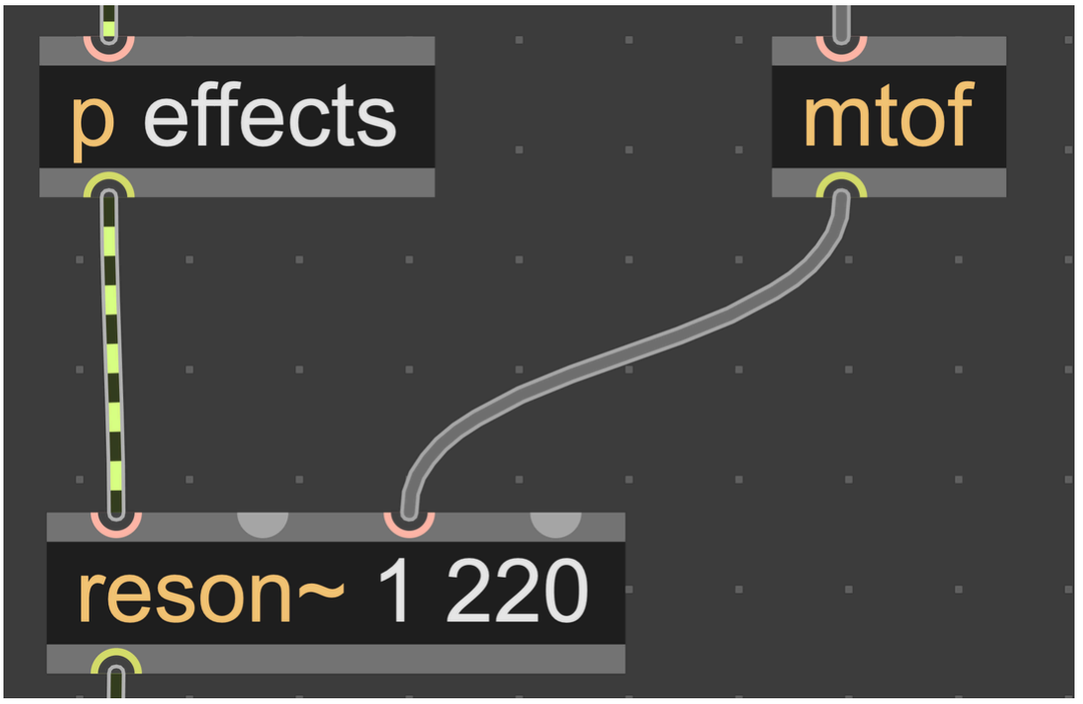
Screenshots and Image Export

Max can export patches in PNG and PDF format, although PDF export is currently available only on macOS. To export a patch as an image, select *Export Image...* from the *File* menu. In the dialog box that appears, you'll be able to choose between PNG and PDF export.



With the save dialog open, select the PDF format to export a PDF.

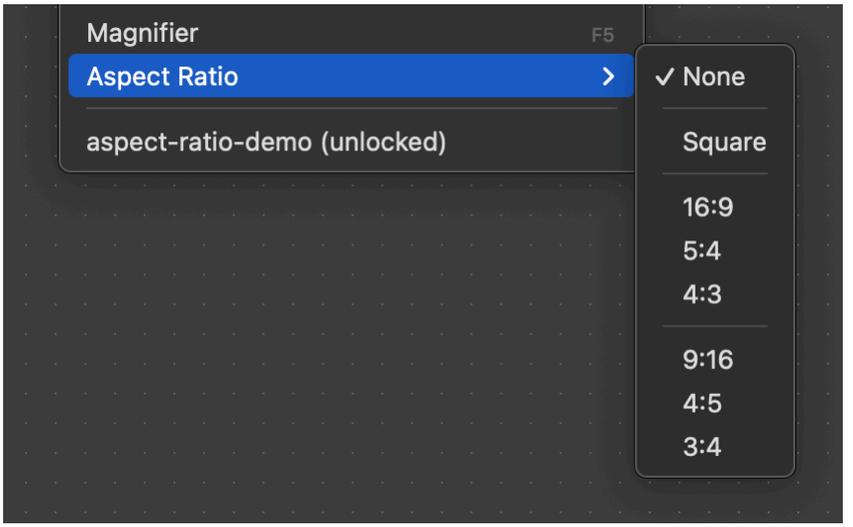
PDF image export saves Max patches in a vector format, with means the image can render at any scale without losing resolution.



PDF image export will render without loss of resolution, even at extreme zoom levels.

Screen Recording

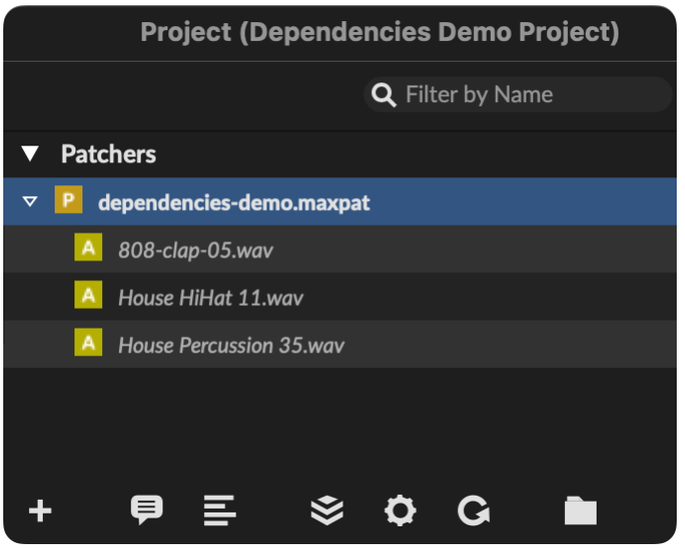
You can lock the aspect ratio of your patcher window by selecting *Aspect Ratio* from the *Window* menu. This can be useful when you want to record your patch for sharing on a platform that expects a certain aspect ratio.



You can lock the aspect ratio of your patcher window to one of several common options.

Projects

For sharing large patchers with lots of dependencies, you might want to use a Max Project. For each patcher in a project. Max keeps track of the resources (audio samples, text documents, video files, etc.) that the patcher depends on. You can use the consolidate option to bring those resources into the project folder. You can then use the *Archive* command to create a `.maxzip` project archive, which can be opened by anyone with Max.



A Max project window, showing the implicit dependencies of a patcher that loads audio samples.

Tools and Abstractions

You may find yourself building a library of Max tools, perhaps in the form of abstractions. You might for example compile a library of audio effects or JavaScript utilities. In order to share these, you can simply share a folder of the `.maxpat` files and other resources. For anyone who wants to use these resources, than can install them anywhere on their computer, provided the installation path is in Max's search path.

Packages

Over time, your library of tools, abstractions, and other resources may reach the point where you want to share it with a large number of users. You might want to use the library as part of a class or workshop, or you might want to distribute it widely to the whole Max community. A Max package provides a wrapper for libraries like this, making it easier to organize, present, distribute, and update them. See the user guide page on Max Packages and the Package Manager for more information.

Projects

Creating Projects 664
Project Window 665
Project Inspector 666
Editing Files in a Project 667
Project Search Paths 668
Open Actions 669
Max for Live Device Projects 669

Projects collect and organize dependencies associated with a patcher or set of patches. These dependencies can include patches, media files, code, and third-party externals.

Projects also extend the main Max **search path**. All files in a project can locate other files in the same project.

- Project-specific elements are loaded before any other files on the Max search path.
- Project-specific assets don't have to be added to the main Max search path, isolating them from other patches and Projects.
- You can switch between global and Project-local assets.
- Using projects lets you reduce the number of folders in Max's global search, which can improve patcher loading and editing speed.

Creating Projects

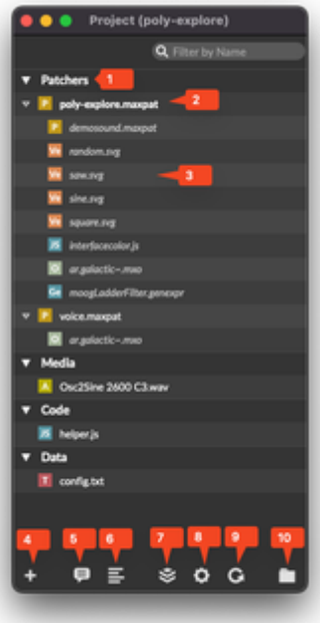
You can start with a new empty project, or create a project from an existing patcher.

- To create a new empty project, choose *New Project* from the *File* menu. This action creates a Max Project at the location specified. By default, this is `~/Documents/Max/Projects` (Mac) or `(User Folder)\My Documents\Max\Projects` (Win). Projects can be saved anywhere, but it is recommended to use the default location.

- To create a project from your existing open patcher, choose *Save as Project* from the *File* menu. This will close and re-open your patcher as a project.

> Once the project is created, you don't need to save it: projects are saved automatically as you edit them.

Project Window



1. Category header: Files in your project are automatically assigned to a category, like "Patchers", "Media", "Code", or "Data". Categories are only shown if there are matching files in the project.
2. Patcher opened with project: A project may contain one or more "top-level patchers". These are displayed in bold in the *Project* window, and are automatically opened when the project is loaded.
3. Dependencies: Files local to a project are shown in regular, non-italic text. Dependencies of project files are shown in a darker color, and if those files are only found on the global Max

search path, they are shown in italics. Dependencies cannot be removed from a project from the *Project* pindow.

4. Add files to Project: To add files to a project, they can be created from scratch, patchers can be created from templates, and existing files can be added to projects. Files can be removed by right-clicking on the file name in the *Project* window and selecting "Remove from Project." Alternatively, you can highlight the name and use the backspace or delete key on your keyboard.
5. Details view: When toggled on and a file from the list is selected, details about the file are shown, including the name, kind and absolute path of the file. If the file is a dependency (whether explicit or implicit), a link is shown to highlight the patchers in the file list which use it. Also shown is whether the file is opened with the project or not.
6. Hierarchical and flat view modes: Hierarchical view lists all dependencies underneath the file which depends on them. Flat view sorts all files, whether dependencies or not, into a flat list.
7. Manage project: From the *Manage Project* menu, you can

    - consolidate and deconsolidate. Consolidation copies all dependencies to the local project folder. De-consolidation removes any files which can be found on the global Max search path from your local project directory.
    - archive. Similar to consolidation, archiving creates a copy of all files needed by the project. Unlike consolidation, archiving creates a new ".maxzip" archive file that can be saved and distributed as a single file.
    - build collectives and applications.
    - export as a Max for Live device.

8. Project settings: Open the *Project Inspector* window, the *Project Seach Path* window, or the *Open Actions* text window.
9. Reload Project: Refreshes the *Project* window.
10. Show Project folder: Opens the project's location on disk.

Project Inspector

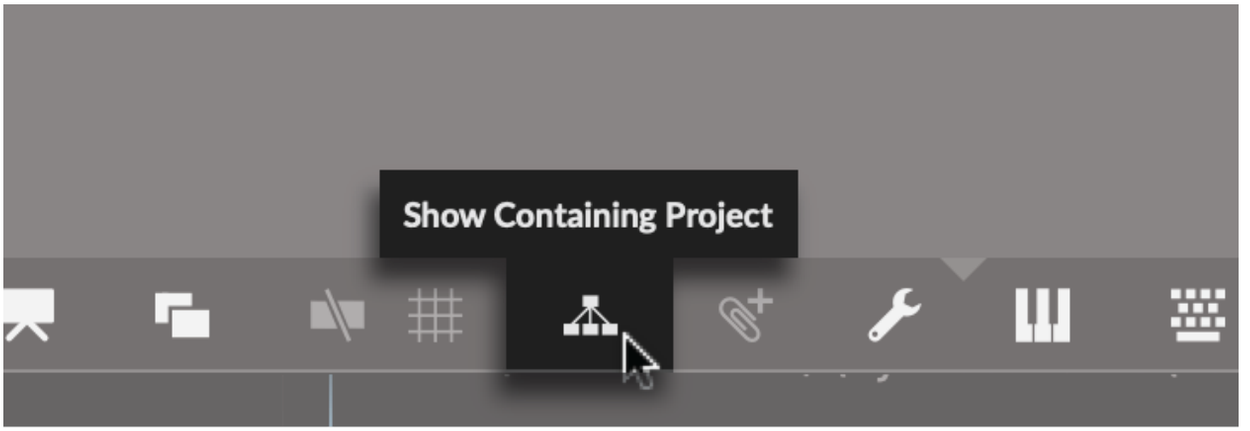
The Project Inspector allows you to set preferences for the project's behavior.

- Always Localize Project Items: When enabled, files added to a project will always be copied to the project's folder before being included in the project. (default = off)
- Hide Project Window After Opening: When enabled (and only if the project contains patchers which are marked "Open on Project Load"), the project window will not be visible after the Project opens. You can open the *Project* window later by click on the *Show Containing Project* button in the patcher's toolbar. (default = off)
- Keep Project Folder Organized: When enabled, files in the project's folder are automatically sorted into appropriate subfolders based on the file's category. (default = on)
- Show Patcher Dependencies: When enabled, implicit dependencies are displayed in the *Project* window. (default = on)
- Development Path Type: When enabled, allows you to specify the path type for the folder specified in the Development Path setting. Options are disabled, relative, and absolute.
- Development Path: Allows you to choose a project-specific folder for development of Max for Live projects. When set, all device files for the project will be saved to and referenced from this location. When disabled, the project uses the global folder `~/Documents/Max 8/Max for Live Devices` (Mac) or `(User Folder)\My Documents\Max 8\Max for Live Devices` (Win).

Editing Files in a Project

To edit a patcher or text file from a project, double-click on the file's name in the project window. Alternatively, right-click on the file's name in the project window and choose Open from the contextual menu.

When editing a patcher from a project, a new icon will appear in the bottom toolbar. You can use this icon to return to the project window from the patcher.



> If you edit a patcher by opening the patcher file directly, rather than opening the parent project first, Max won't be able to use the project context to resolve any *search path* dependencies. Some dependencies may be found in the wrong location (globally rather than project-locally), and some may not be found at all. For this reason, if you're working with a project, be sure to open the project before editing any child patchers.

Project Search Paths

Within a project, patchers and other kinds of files may be marked either global or local. Files marked global tell Max to use the file found on the global search path, if available. Files marked local tell Max to use the file found on the Project's search path first, if available. You also have the ability to publish a file to the global search path or to localize a file to your project's search path.

The project-local search path consists of:

1. Project-local folders: Projects maintain a folder on your hard drive. The contents of this folder (and its subfolders) are preferentially searched when Max is looking for files requested by project members. New files added to a project are created inside this folder by default, and you can manually add files to it, too.
2. Singleton folders: Projects can reference files which are neither in the Max global search path, nor in the project folder. When searching for files, the folders containing these "singletons" are searched (non-recursively), as well.

3. Project search paths: Projects also maintain a list of additional folders to be searched when locating project assets. This list works similarly to the list found in Max's File Preferences... window, but is used only by the project.

Dependencies are always presumed to be local. If no local version can be found, the global version will be used.

Open Actions

Project Open Actions allow you to specify what Max configuration options are loaded when apProject opens.

In the Open Actions text edit window, you can write various messages to Max. For example, messages such as `;dsp takeover 1` or `;max overdrive 1` will enable Scheduler in Audio Interrupt and Overdrive, respectively. When the project is loaded (or reloaded via the *Reload Project* button in the *Project* toolbar), the Open Actions will be executed.

Max for Live Device Projects

All Max for Live devices are projects. There are a few differences between regular Max projects and Max for Live device projects:

- The default location for Max for Live Device Projects is `~/Documents/Max 8/Max for Live Devices` (Mac) or `(User Folder)\My Documents\Max 8\Max for Live Devices` (Win)
- Unfreezing a device will unpack the device's assets to the project's folder, rather than to the "Unfrozen Max Device Files" folder on the Desktop.
- Conflicts are now auto-resolved in favor of the Device version (although you can override this in the Resolve Conflicts window if you need to).
- The Archive... item in the *Manage Project* toolbar menu creates a datestamped ".zip" archive of a frozen copy of the current device.
- Freezing a device will include all files listed in the *Project* window (both explicit and implicit items).

Freezing a device which utilizes 3rd party externals will automatically collect and freeze the externals for other Max platforms if they can be found in Max's (or the project's) search path. For instance, freezing a device which uses `myextern.mxe64` on Windows will include `myextern.mxo` (macOS) externals if they are available. None of these need to be added explicitly to the project.

Standalones and Collectives

Building Standalones 795
Customizing Your Standalone 796
Standalone Applications for MacOS 797
Standalone Applications for Windows 798
Search Paths in Standalones 800
The Collective Editor 802

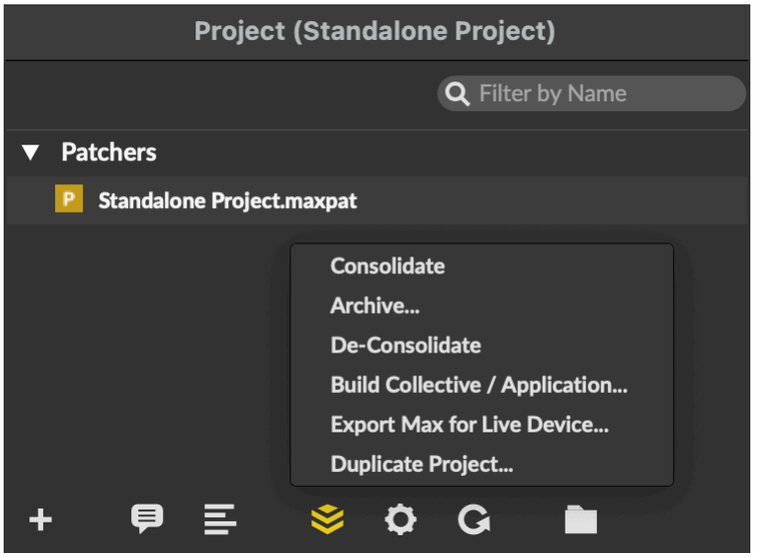
A **Standalone** lets you turn your Max patcher into an application that can run on its own, without needing to have the main Max application installed. The resulting file looks and acts like a standard macOS or Windows application. This can be a handy way to share your work with someone who doesn't have Max on their machine, or to run your work in a context like an art installation.

Under the hood, a standalone is a bundle containing the Max runtime along with a **Collective**. A collective contains all of the dependencies, including media files and Max externals, that a patcher needs to run. Most of the time, collectives are only used internally by a standalone, and you won't need to work with them directly. However, you can still open a collective using Max.

> If you want to share work with another Max user, consider using a **Project**. In particular, a project is the ideal document for collaboration when using Max with a version control tool like Git.

Building Standalones

The simplest way to build a standalone is to start with a Max project. From the *Project* window, click the *Manage Project* button in the bottom toolbar and select `Build Collective / Application...`



In the dialog box that appears, be sure to select *Application* from the *Filetypes* dropdown. From there, simply press *Save*. After Max finishes bundling up your project, you'll see the exported application appear in your file browser.

You can also build a standalone by selecting `Build Collective / Application...` from the *File* menu. If you build a standalone this way, without a Max project open, Max will open the **Collective Editor**.

Customizing Your Standalone

To configure options for your standalone, place a standalone object in your toplevel patcher. Open the inspector for this object to see all of the configuration options available.

Adding a custom application icon

Add a standalone object to your top level patcher. Find the `@appicon_mac` attribute, and specify the path to the app icon image to use for macOS. Use the attribute `@appicon_win` to specify an app icon for Windows.

Managing standalone size

By default, standalone export includes the whole Max runtime. By setting attributes on the standalone object, you can include or exclude certain parts of the Max application, which can

reduce the size of your exported standalone. See the standalone object reference for a full description, but some options you can configure include:

- **@gensupport** - Enable support for gen~, jit.gen, jit.pix, and jit.gl.pix. Disable to reduce standalone application size.
- **@cefsupport** - Enable support for the jweb object. Disable to reduce application size.

Standalone Applications for MacOS

Max builds **universal binary** standalone applications on macOS. The standalone is an application package , which is also a folder that looks like a file in the Finder. Double-clicking on the file's icon launches the application. Right-click on the application file and select *Show Package Contents* from the contextual menu to look inside the application bundle.

```
YourApplication.app [ note: the .app is not shown in the Finder ]/
└── Contents [ folder ]/
  ├── Frameworks [ folder ] – needed for external object support
  ├── Info.plist [ copied from your Info.plist if included in a collective
build script ]
  ├── MacOS [ folder ]/
  │   └── YourApplication [ actually the Max Runtime executable ]
  └── Resources [ folder ]/
    ├── [ custom icon file goes here ]
    └── C74 [ folder containing all supporting files]
```

Manually including Max resources (macOS)

If you disabled the `@copysupport` attribute on your toplevel standalone object, Max will not copy its `Resources` folder into the standalone application. If you need support for certain features, you can try to manually copy Max resources into your application bundle.

Including Jitter components in your Mac standalone application

Although they are included by default, if you are not including the C74 Resources, be sure to add any shader, material, model, passes, or volume-dataset files that your application uses, copying to

`<package>/Contents/Resources/C74/media/jitter/*`. Many of the included image-processing shaders rely on one of the "passthru" vertex shader programs that reside in the shared folder inside the jitter-shaders folder.

Including Java components in your Macintosh standalone application

Although they are included by default, if you are not including the C74 Resources, copy any necessary Java class files to `<package>/Contents/Resources/C74/java-classes/classes`.

Distributing your Mac standalone application

In most cases, Max standalone apps will be distributed outside of the Mac App Store. In order for the application to open properly, the standalone can be signed (if one pays the $99/year developer membership) or users of the standalone can informed that the Security and Privacy settings in System Preferences must be set to 'Allow apps downloaded from Anywhere'. To read more about signing your application, visit the Apple Developer Site.

For distribution on macOS versions starting with Big Sur (macOS 11), the developer will need to code sign their standalone application with the appropriate Entitlements in order to avoid the user being asked for microphone, camera, and disk access every time the app is launched. Learn more in this article.

Standalone Applications for Windows

When you create a standalone application on Windows, you're actually creating a folder containing the Max Runtime executable plus a .mxf collective file containing your patches and files. In addition, the standalone folder contains a support folder with files necessary to run your application.

```
YourApplication [ folder ]
├── YourApplication.exe [modified MaxRT.exe – launch this to launch your
app ]
└── YourApplication.mxf [ collective containing your patches ]
  └── support [ folder ]
    ├── ad [ folder containing MSP audio driver objects ]
    ├── mididrivers [ folder containing Max midi driver objects ]
    ├── MaxAPI.dll [ Max API for external objects ]
    ├── MaxAudio.dll [ MSP library ]
    └── interfaces [ folder containing menu specifications, icons, etc. ]
```

Runtime library dependency

Max depends on the Microsoft C Runtime Library. To use a standalone on a computer that has not run the Max installer you must first run the *Microsoft Visual C++ 2013 Redistributable Package*. This can be downloaded from Microsoft. An alternative is to simply download and install Max. The Microsoft redistributable files can be found at this URL:

https://www.microsoft.com/en-us/download/details.aspx?id=40784

Manually including Max resources (Windows)

If you disabled the `@copysupport` attribute on your toplevel standalone object, Max will not copy its `Resources` folder into the standalone application. If you need support for certain features, you can try to manually copy Max resources into your application bundle.

Including Jitter components in your Windows standalone application

- Copy the file `/Program Files/Cycling '74/Max 9/support/jitlib.dll` to `{standalone folder}/support/jitlib.dll`

Including Jitter shader components in your Windows standalone application

- Copy the file `/Program Files/Common Files/Max 9/Cycling '74/jitter-shaders` to `{standalone folder}/support/jitter-shaders`. Be sure to include any shader files that the included shaders depend on; many of the included image-processing shaders rely on

one of the "passthru" vertex shader programs that reside in the `/Program Files/Common Files/Max 9/Cycling '74/jitter-shaders/shared/ folder`.

- Copy the file `/Program Files/Cycling '74/Max 9/support/cg.dll` to `{standalone folder}/support/cg.dll`
- Copy the file `/Program Files/Cycling '74/Max 9/support/cgGL.dll` to `{standalone folder}/support/cgGL.dll`

Including Java components in your Windows standalone application

- Copy the file `/Program Files/Common Files/Max 9/Cycling '74/java/lib/max.jar` to `{standalone folder}/support/java/lib/max.jar`.
- Copy any other necessary jar files (e.g. `jitter.jar`) to `{standalone folder}/support/java/lib/max.jar`
- Copy / `Program Files/Common Files/Max 9/Cycling '74/java/classes/*.` class to `{standalone folder}/support/java/classes/*.class` (all your necessary class files).

Search Paths in Standalones

A standalone application you create with Max searches for files in a slightly different order than that used in the Max search path when you're editing patches. If you want to include non-standard Max objects or other types of files, you'll need to understand how search paths and File Preferences work in standalone applications.

In a standalone application, here is the order in which folders in the standalone package (Macintosh) or folder (Windows) will be searched:

1. The collective file (and any other open collective files)
2. The support folder

You can add additional locations to this list by checking the Utilize Search Path option in the standalone object's Inspector.

On Macintosh systems, checking the Utilize Search Path option, does the following for the search path of a standalone:

- Adds all of the folders inside of the folder containing the standalone application (i.e., the Contents folder and all of its subfolders).
- Adds the Cycling '74 folder. A Cycling '74 folder at the same location as the application is used if it exists. If it does not, the location /Library/Application Support/Cycling ’74 is used if it exists.

On Windows systems, checking the Utilize Search Path option, does the following for the search path of a standalone:

- Adds all of the folders inside of the application folder (i.e. the folder containing YourApplication.exe).
- Adds the Cycling '74 folder. A Cycling '74 folder at the same location as the application is used if it exists. If it does not, the location c:\Program Files\Common Files\Cycling '74\ is used if it exists.

You can use the *Utilize Search Path* option to aid in testing your standalone application.

If some of the supporting files used by Max objects in your patcher will not be included in the collective itself, check the Search for Missing Files option. (It is checked by default.) Unchecking this option can be useful for ensuring that you have included all necessary files in the collective that you are making into a standalone application. If you create your application with this option turned off, your application will not look outside the collective for any files it cannot find, such as missing sequences or coll files that your application attempts to load. So, you can make your application with Search for Missing Files unchecked, and then run it to see if it works properly. If your application is unable to find a file that it needs, you will get an error message to that effect, and you will know that you have to rebuild your standalone application. In some cases, however, you may want your application to look for a file outside of the collective. For example, you may want it to look for a MIDI file that can be supplied by the user of your application. In that case, you will naturally want the Search for Missing Files option to be on. Please also note that this feature restricts itself to looking in folders nested only three levels deep.

When your application searches for files outside the collective, you can control where it looks with the Utilize Search Path in Preferences File option. If this option is on (which it is by default), your application will use the search path settings stored in the Max Preferences file instead of using the default search path. You can instruct your application to use its own Preferences file instead of the default Max Preferences by supplying a preferences file name in this field. If the Utilize Search Path

in Preferences File is checked and you type in a name other than the default Max Preferences, your application will make its own unique preferences file (in ~ /Library/Preferences , where ~ represents your home directory) the first time it is run. From then on, your application will use that preferences file to recall the settings for options such as Overdrive or *All Windows Active* (Windows systems only).

On Macintosh systems, the Finder uses a four-character ID to distinguish one application from another. When you create a standalone, a default generic creator (`????`) is assigned to the application automatically. You can use the standalone object Inspector to change this to a unique ID.

Testing your standalone search path

Open the inspector for your toplevel standalone object. Disable the *Utilize Search Path* setting, and build your application. If you see any errors indicating `no such object` or `can't find files` in the Max Console when you launch the program, you will know which supporting files you have not properly included.

Checking the current file paths in use

Add a message box containing the message `; max paths` to your patcher and then activate the message box. The current file paths in use will be printed to the Max Console.

The Collective Editor

The **Collective Editor** window lets you define the steps that Max will take when it bundles together a collective. You can include patchers, files, and folders, so you can be sure that all your dependencies will be included in your standalone or collective.



The *Collective Editor* window includes a script area, and a number of buttons for configuring your collective. The script shows each of the steps involved in building the collective. The first entry in the script `open thispatcher` is added for you. This tells the collective to include the patcher that was the topmost window when the collective editor was opened. The `open` keyword means that the patcher will be opened when the collective file is opened.

> If you're managing a complex system with lots of potential dependencies, it's often easier to work with a project than to configure your collective manually using the collective editor. When you build a standalone from a project, Max will automatically use the implicit and explicit dependencies of the project to define your collective.

Collective dependencies

For your main toplevel patcher, and for any other patchers that you include in your standalone/collective, Max will try to automatically include any dependencies of that patcher. Any object with an explicit dependency will be able to share that dependency with the standalone builder.



This v8 object has an explicit dependency, so that dependency will be included automatically in any standalone or collective.

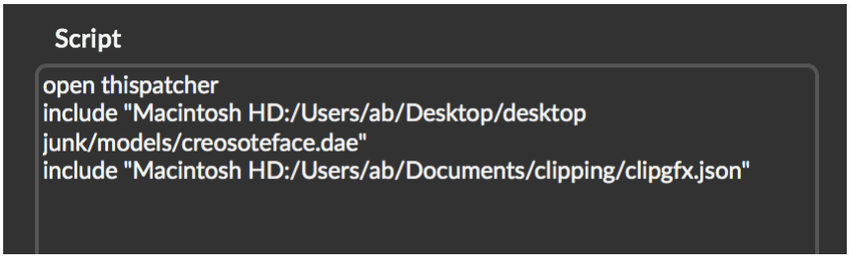
If you load a file dynamically, or include the name of a dependency in a message box, that dependency might not be registered automatically. In this case, you would need to add the dependency manually when defining your collective.



This buffer object does not have an explicit depednency. The audio file 'my-sample.aif' would need to be added manually to any exported standalone or collective.

Adding toplevel patchers

A **Toplevel** patcher window opens when the collective opens. You can have more than one toplevel patcher (you need at least one, however). To add a toplevel patcher, plick the *Toplevel Patcher...* button in the *Collective Editor* window and choose a patcher file from the open file dialog that appears.



Once you select the file, a new line using the `open` keyword is added to the script. If you were to open this standalone or collective, both of the files listed with the open keyword would initially be visible.

Adding files to a standalone/collective

- Click the *Include File...* button in the *Collective Editor* window and select the file you want to add from the open file dialog.
- The full path of the file chosen will be added to the build script, preceded by the `include` keyword.

Adding folders to a standalone/collective

If you have an entire folder of data files you want to include, you can include all the files at once.

- Click the *Include Folder...* button in the *Collective Editor* window and select the folder you want to add from the open file dialog.
- The full path of the folder chosen will be added to the build script, preceded by the `include` keyword.

> Including a folder will only include files in the folder itself. Folders inside the folder (subfolders) will not be included.

Saving and reloading the build script

After you've configured your standalone/collective by adding files and folders, you can save your configuration to a **build script**. This lets you load that script later if you want to make changes to your configuration.

- Click the *Save Script...* button in the *Collective Editor* window and type in a name for your script in the file dialog box. A copy of the current build instructions will be written to a text file.

If you have previously saved a build script for your collective, you can open the file and use it for quick editing and/or rebuilding.

- Click the *Open Script...* button in the *Collective Editor* window and choose the build script you have created for your standalone/collective.
- The *Collective Editor* window will display the contents of the file you have loaded. At this point, you can click the *Build* button to build your collective, or modify the script if necessary.

> Since a collective script uses absolute path names to refer to files, the script is not necessarily transferrable from one machine to another.

Timing

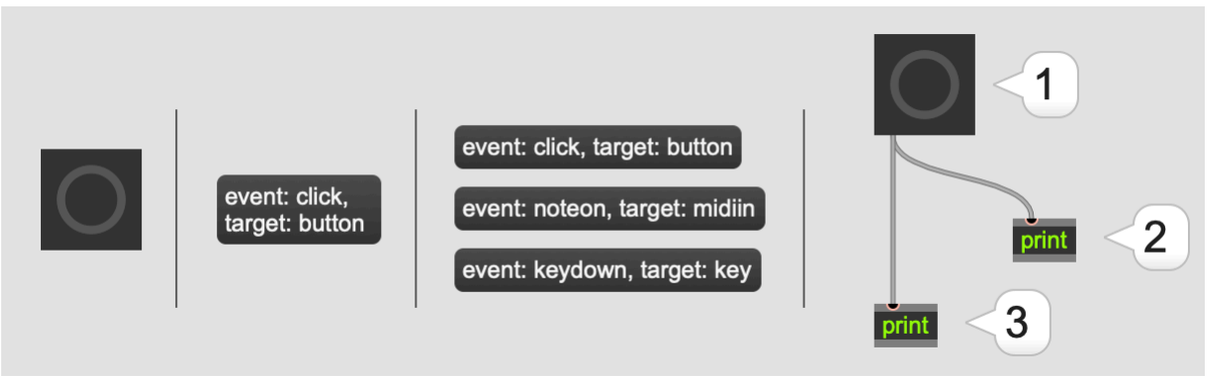
Scheduler and Priority

High Priority and Low Priority	809
Configuring the Scheduler	809
Infinite Loops and Stack Overflow	812
Starting and Stopping the Scheduler	813
Changing Priority	814
Event Backlog and Data Rate Reduction	815

Whenever something happens in a Max patcher, it starts with an **Event**.

Events can be generated from different sources—for example, a MIDI keyboard, the metro object, a mouse click, a computer keyboard, etc. After an event is generated, Max either passes the event to the **high-priority scheduler**, or puts it onto the **low-priority queue**. Periodically, Max **services** the queue, during which it removes an event from the queue and **handles** it.

Handling an event usually means sending a message to an object, which might send messages to another object, until finally the event has been completely handled. The way in which these messages traverse the patcher network is depth first, meaning that a path in the network is always followed to its terminal node before messages will be sent down adjacent paths.



Loose visual representation of how Max works. Clicking a button generates a 'click' event, which Max puts onto a queue. When it services the queue, it handles the event by sending a message to the target object and propagating messages through the patcher graph.

High Priority and Low Priority

Some events have timing information associated with them, like the ones that come from a MIDI keyboard or a metro object. We call these events **high priority** events, since they happen at a specific time, and Max wants to prioritize handling them at that time. These are also called scheduler events, since they are handled by Max's **scheduler**. All other events are **low priority** events. Max tries to process these as fast as possible, but not at any specific time. These are also called queue events, since Max doesn't decide when to process them based on timing information, but simply by processing them in first-in-first-out order.

Configuring the Scheduler

The two most important settings for the scheduler are **Overdrive** and **Scheduler in Audio Interrupt**. Overdrive is so important that you can toggle it directly by selecting *Overdrive* from the *Options* menu. You can also enable overdrive from the **Max Preferences Window**, which is also the place to enable *Scheduler in Audio Interrupt*.

For all other scheduler settings, including low-level, explicit control over all the details of how the scheduler operates, you can tweak the scheduler preferences.

Overdrive and parallel execution

With **Overdrive** enabled, Max uses two separate threads to process high priority and low priority events. This means that high priority events can be handled before a low priority event has finished executing, resulting in better timing accuracy for high priority events. Using multiple threads also has the advantage that on multi-processor machines, one processor could be executing low priority events, while a second processor could be executing high priority events.

Normally, Max processes both kinds of events in the same thread, neither one interrupting the other. A high priority event would have to wait for a low priority event to be handled completely before the high priority event itself may be executed. This waiting results in less accurate timing for high priority events, and in some instances a long stall when waiting for very long low priority events, like loading an audio file into a buffer~ object.

By default, overdrive is disabled. This is because it adds a small amount of complexity to the way patches work, since high priority and low priority messages will pass through the patcher network simultaneously. Usually you can ignore this, but sometimes it might be important to remember that with overdrive on, the state of a patcher could change in the middle of an event being processed.

> In addition, the debugging features of Max only work if overdrive is disabled.

Scheduler in Audio Interrupt

When *Scheduler in Audio Interrupt* (SIAI) is turned on, the high-priority scheduler runs inside the audio thread. The advancement of scheduler time is tightly coupled with the advancement of DSP time, and the scheduler is serviced once per signal vector. This can be desirable in a variety of contexts, however it is important to note a few things.

First, if using SIAI, you will want to watch out for expensive calculations in the scheduler, or else it is possible that the audio thread will not keep up with its real-time demands and hence drop vectors. This will cause large clicks and glitches in the audio output. To minimize this problem, you may want to turn down *poll throttle* to limit the number of events that are serviced per scheduler servicing, increase the I/O vector size to build in more buffer time for varying performance per signal vector, and/or revise your patches so that you are guaranteed no such expensive calculations in the scheduler.

Second, with SIAI, the scheduler will be extremely accurate with respect to the MSP audio signal, however, due to the way audio signal vectors are calculated, the scheduler might be less accurate with respect to actual time. For example, if the audio calculation is not very expensive, there may be clumping towards the beginning of one I/O vector worth of samples. If timing with respect to both DSP time and actual time is a primary concern, a decreased I/O vector size can help improve things, but as mentioned above, might lead to more glitches if your scheduler calculations are expensive. Another trick to synchronize with actual time is to use an object like the date object to match events with the system time as reported by the OS.

Third, if using SIAI, the scheduler and audio processing share the same thread, and therefore may not be as good at exploiting multi-processor resources.

Best practices

When Overdrive is enabled, Max gives priority to timing and MIDI processing over screen drawing and user interface tasks such as responding to mouse clicks. If you are primarily going to be using Max for MIDI or audio processing, Overdrive should be enabled. If you are primarily going to be using Jitter, Overdrive should be disabled.

As for SIAI, remember that with SIAI enabled, the scheduler uses the audio sample counter, rather than "real-world time", to schedule events. If your patcher is generating events that affect your audio processing, and the sync between audio and events is important, you should enable SIAI. If it's more important that your scheduler syncs with the outside world, including synchronization with external hardware, then disable SIAI.

If you need a balance of both, use SIAI with as small an I/O vector size as you can afford to without dropouts. Just be aware that this is more demanding on your computer, and you might have to change your settings if you hear audio dropouts.

Advanced settings

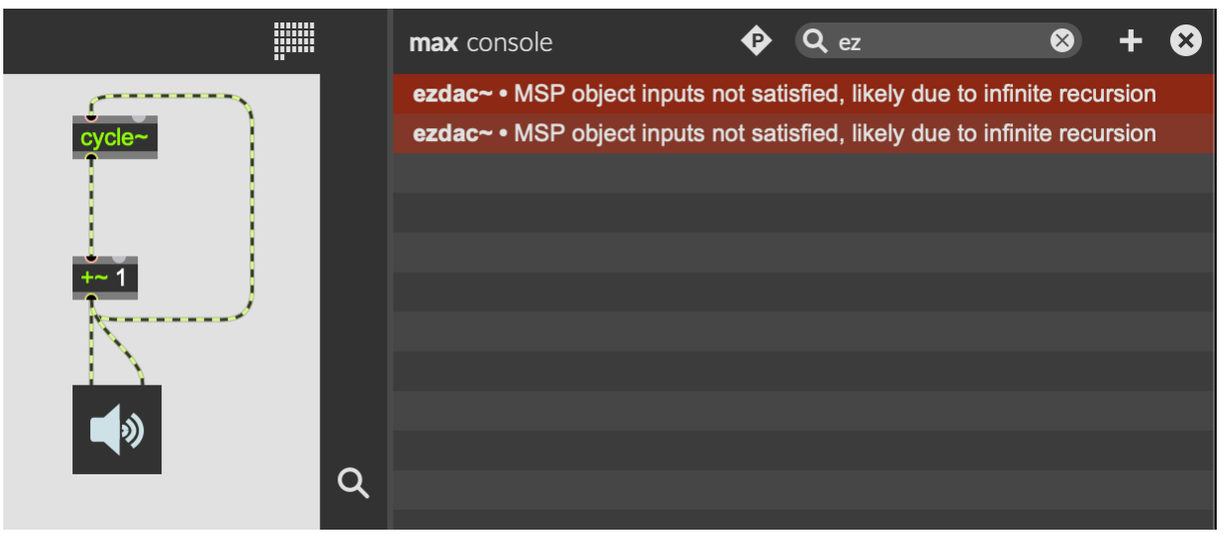
Sor all other scheduler configuration, see the scheduler section in Max's preferences. This section discusses these settings in detail.

- *Event Interval* sets the interval, in milliseconds, at which low-priority (main thread) queue events are throttled. Historically, Max would poll for events at the event interval. Recent versions of Max avoid polling, for increased efficiency, but incoming events are throttled at the event interval. The default value of is 2 ms.
- *Poll Throttle* sets the number of high-priority events processed per by the scheduler at one time. High-priority events include MIDI as well as events generated by metro and other timing objects. A lower setting (e.g., 1) means less event clumping, while a higher value (e.g., 100) will result in less of an event backlog. The default value is 20 events.
- *Queue Throttle* sets the number of events processed per servicing of the low priority event queue (Low priority events include user interface events, graphics operations, reading files from disk, and other expensive operations that would otherwise cause timing problems for the scheduler). A lower setting (e.g. 1) means less event clumping, while a higher value (e.g. 100) will result in less of an event backlog. The default value is 10 events.

- *Redraw Queue Throttle* sets the maximum number of patcher UI update events to process at a time. Lower values can lead to more processing power being made available to other low-priority Max processes, and higher values make user interfaces more responsive (especially for patches in which large numbers of bpatcher objects are being used). The default value is 1000 events.
- *Refresh Rate* sets the minimum time, in milliseconds, between updating of the interface. A lower setting (e.g., 5) means that Max will devote more time to redrawing the screen and less to responding to user input, while a higher value (e.g., 60) will mean that the interface is faster and more responsive. The default value is 33.3 ms.
- *Scheduler slop* is the amount of time, in milliseconds, the scheduler is permitted to fall behind actual time before correcting the scheduler time to actual time. The scheduler will fall behind actual time if there are more high priority events than the computer can process. Typically some amount of slop is desired so that high priority events will maintain long term temporal accuracy despite small temporal deviations. A lower setting (e.g., 1) results in greater short term accuracy, while a higher value (e.g., 100) will mean that the scheduler is more accurate in the long term. The default value is 25 milliseconds.

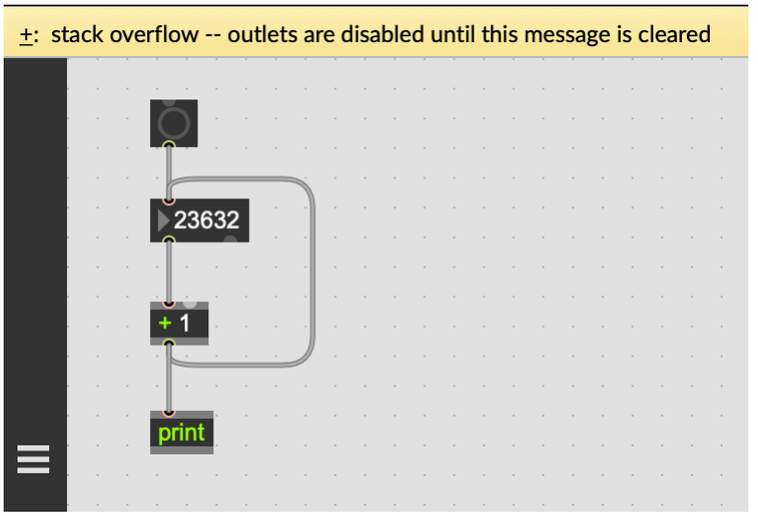
Infinite Loops and Stack Overflow

If you try to connect DSP objects in a loop, Max will post an error message to the console, and audio processing will be disabled.



However, Max has no problem letting you make regular message connections that form a loop. In fact, this can be a useful way to process data recursively, or to implement an iterative algorithm.

However, if sending a message from one object to the next causes messages to be sent forever in an infinite loop, this will cause a stack overflow.



The stack overflow results since a network containing feedback has infinite depth, and the computer runs out of memory attempting to handle the event. When Max encounters a stack overflow, it will disable all outputs until you clear the message at the top of the patcher window. This will give you a chance to look at the patcher and figure out where the stack overflow occured.

> You can break an infinite loop into chunks by scheduling a new event, either using a high-priority timing object like pipe or delay, or using the low-priority deferlow object.

Starting and Stopping the Scheduler

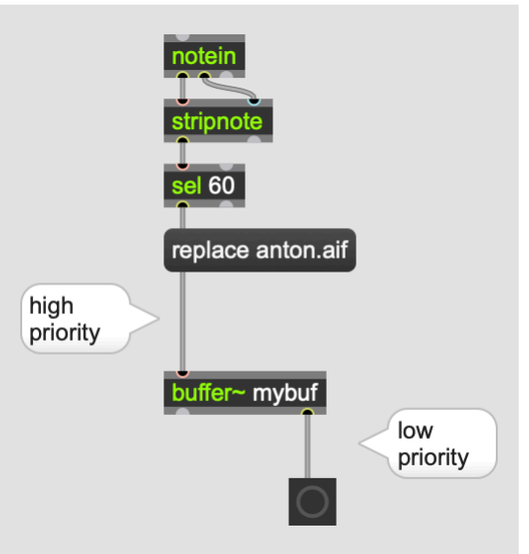
Press ⌘. (macOS) or CTRL . (Windows) to stop the scheduler. You can also select *Stop Scheduler* from the *Edit* menu. With the scheduler stopped, timing objects like metro, pipe, and delay will no longer function. MIDI input and output objects like midiin and midiout will stop working too.

To start the scheduler again, press ⌘r (macOS) or CTRL r (Windows), or select *Resume Scheduler* from the *Edit* menu.

Stopping the scheduler is useful if you want to pause a patcher that's doing a lot of automated execution. You might have a bunch of metro objects, each on a one millisecond delay, each causing a bunch of processing to happen. By stopping the scheduler, you can temporarily disable everything, giving yourself a chance to make edits to your patcher.

Changing Priority

Certain objects will always execute messages at low priority, even when they receive a message during a high-priority event. For example, if a MIDI messages causes a buffer~ to load a new audio file, buffer~ will still load that file at low priority. Messages that cause drawing, read or write to a file, or launch a dialog box are typical examples of things which are not desirable at high priority.



The notein object generates messages in response to MIDI events, which execute at high priority. However, buffer~ loads files at low priority, so the notification that buffer~ has finished loading a file will come out at low priority.

You can use the defer and deferlow objects to do this same kind of deferral explicitly, generating a new event that will be handled from the low-priority queue. The defer object will put the new event at the front of the low-priority queue, while the deferlow object will put it at the back. This is an important difference, since it means that defer can reverse the order of a message sequence, while deferlow will preserve order.

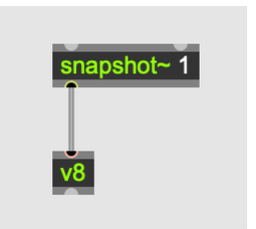
> The defer object will not defer event execution if executed from the low-priority queue, while the deferlow object will always put a new event at the back of the low-priority queue, even when handling a low-priority event. If it receives a message from a low-priority event, the defer object will simply pass that messages through, without putting a new event on the queue.

To move an event from the low-priority queue to the high-priority scheduler, use the delay or pipe objects to schedule a new, high-priority event. You might want to do this in order to force certain computations to happen at high priority, or to schedule an event for a specific time.

Event Backlog and Data Rate Reduction

With very rapid data streams, such as a high frequency metro, or the output of a snapshot~ object with a rapid interval like 1 millisecond, it is easy to generate more events than can be processed in real time. The high priority scheduler or low priority queue can fill up will events faster than Max can process them. Max will struggle to process new events and, eventually, the whole application will crash.

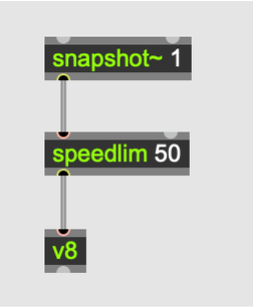
One common cause of event backlog is connecting an object that generates high-priority events to one that always execute in low priority. This could occur when connecting a snapshot~ object to a v8 object. Since v8 always executes at low priority, this will push an event onto the low priority queue every time that snapshot~ generates an event.



If the v8 object runs an expensive computation when it receives a float, this could cause backlog on the low-priority queue.

The speedlim, qlim, and onebang objects are useful at performing data rate reduction on these rapid streams to keep up with real time. In the above case, you could use speedlim to limit the rate

at which messages get sent from snapshot~ to v8.



The speedlim object limits the rate at which messages pass from snapshot~ to v8.

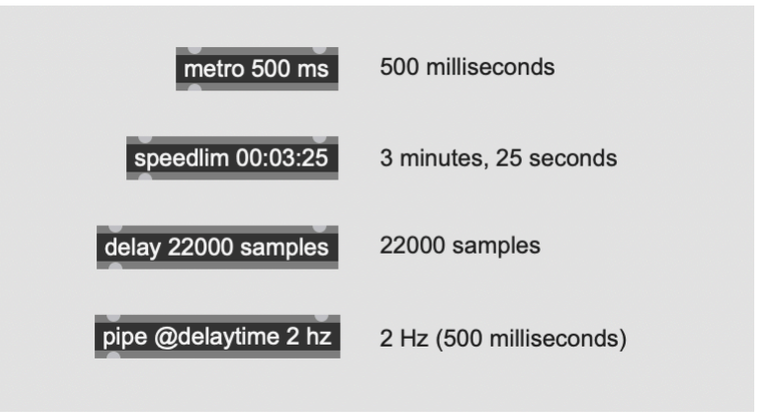
> There's nothing inherent wrong with connecting an object like snapshot~ to an object like v8. Rather, you should simply be aware of what's going on behind the scenes, so you know how to modify your patcher if it isn't behaving the way you want.

Time Value Syntax

Fixed Time Values 817
Tempo-relative Time Values 818
Note Values in Ticks 819
Positions vs Intervals 820

Most Max objects that deal with timed events, like metro, phasor~, and pipe, can represent time in multiple ways. In general these fall into two categories: **Fixed** time values and **Tempo-relative** time values. Fixed values express time in milliseconds or some other absolute units. Tempo-relative values depend on the current tempo and time signature, as set by a transport object (or the **Global Transport**).

Fixed Time Values

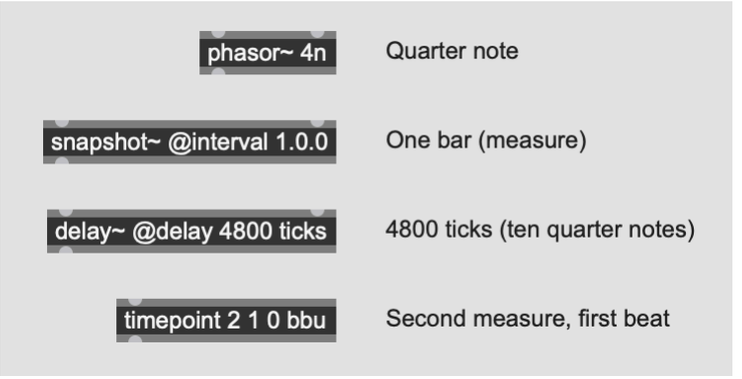


Some objects using Fixed time values

Unit	Format	Example	Notes
milliseconds	`ms` suffix	`100 ms`	The default for objects like metro
hours/minutes/seconds	`:` between each number	`01:03:45.250`	1 hour, 3 minutes, 45 seconds, and 250 milliseconds. The millisecond value after the decimal is optional.

hours/minutes/seconds	list of 3 or 4 numbers, followed by `hh:mm:ss`	`1 3 45 250 hh:mm:ss`	An equivalent option for representing time in terms of hours/minutes/seconds/milliseconds
samples	`samples` suffix	`1000 samples`	The actual duration will depend on the sample rate
frequency	`hz` suffix	`5 hz`	The inverse of milliseconds, so `2 hz` is equivalent to `500 ms`.

Tempo-relative Time Values



Some objects using Tempo-relative time values

All Tempo-relative Time Values can be expressed in terms of **ticks**, where one tick is $1/480$ of a quarter note (equivalently, there are 480 ticks in one quarter note).

Unit	Format	Example	Notes
ticks	`ticks` suffix	`100 ticks`	In places where only tempo-relative time values are allowed, such as the `@quantize` attribute of the metro object, values in ticks can be specified as a single number. In places where both fixed and tempo-relative units are accepted, such as the `@interval` attribute of a metro object, a value in ticks must be followed by `ticks` to be interpreted as ticks instead of milliseconds.

note values	see "Note Values" table	`4nt`	Symbols that abbreviate musical note time values—see the table below for recognized values.
bars/beats/units	`.` between each number	`2.4.240`	2 bars, 4 beats, 240 ticks. When you need to use a single value, bars/beats/units can be separated by periods.
bars/beats/units	three numbers	`2 4 240`	When you can pass a list of values, bars/beats/units can be specified by separate numbers like this. When an object will accept Fixed or Tempo-relative time values, you can add `bbu` to make sure the list is understood as bars/beats/units and not hours/minutes/seconds.

Note Values in Ticks

Note	Ticks	Interpretation
1nd	2880 ticks	Dotted whole note
1n	1920 ticks	Whole note
1nt	1280 ticks	Whole note triplet
2nd	1440 ticks	Dotted half note
2n	960 ticks	Half note
2nt	640 ticks	Half note triplet
4nd	720 ticks	Dotted quarter note
4n	480 ticks	Quarter note
4nt	320 ticks	Quarter note triplet
8nd	360 ticks	Dotted eighth note
8n	240 ticks	Eighth note
8nt	160 ticks	Eighth note triplet

16nd	180 ticks	Dotted sixteenth note
16n	120 ticks	Sixteenth note
16nt	80 ticks	Sixteenth note triplet
32nd	90 ticks	Dotted thirty-second note
32n	60 ticks	thirty-second note
32nt	40 ticks	thirty-second-note triplet
64nd	45 ticks	Dotted sixty-fourth note
64n	30 ticks	Sixty-fourth note
128n	15 ticks	One-hundred-twenty-eighth note

Positions vs Intervals

Some objects will interpret a Tempo-relative value in bars/beats/units as a *position*, while others will interpret the same value as an *interval*. For example, the timepoint object fires an event at a point in time, and would interpret the bars/beats/units value `1 1 0 bbu` as a point in time on the first beat of the first measure—in other words at time zero. Attributes like the `@quantize` attribute of metro on the other hand wil interpret the same value `1 1 0 bbu` as an *interval* of one bar and one beat, or 5 beats in 4/44/4 time (interval, quantization, and delay attributes of objects are generally time intervals).

The translate object has a `@mode` attribute that can convert time units as either intervals or positions.

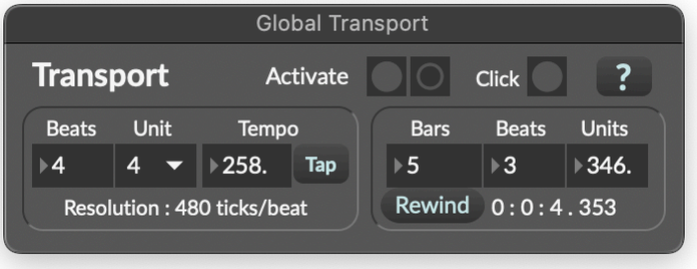
Transport

Accessing the Global Transport 821
Controlling the Transport 821
Named Transports 822
Max for Live 823
Transport Resolution 824

Most DAWs, especially those that organize audio and MIDI into tracks, have some notion of a *playhead*. When you tell the program to "play", the playhead represents the point in the current song or session at which play will resume. There are usually controls to play, stop, and loop playback, as well as some way to set the tempo and time signature. Max doesn't organize audio and MIDI into tracks and clips, but it does have a transport that you can use to organize events in time.

Accessing the Global Transport

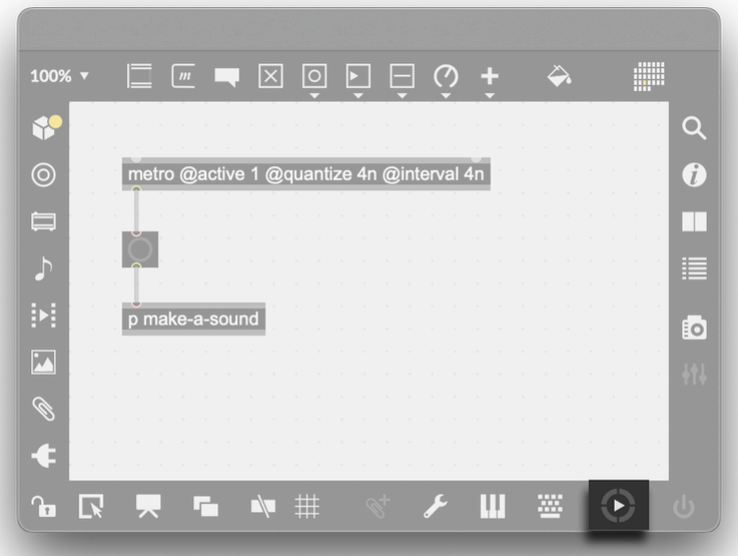
Open the **Global Transport** by choosing *Global Transport* from the *Extras* menu. This will bring up the Global Transport window, which is an interface to Max's shared global transport. All patches in Max can access this transport, and objects that work with the transport can react when the transport starts, stops, changes position, or changes tempo.



The global transport window

Controlling the Transport

From the *Global Transport* window, you can start and stop the transport by clicking the *Activate* button at the top of the window. It's also possible to start the global transport from the toolbar of any Max patcher. The *play* button on the right of the bottom toolbar can start and stop the global transport.



Transport controls in the patcher toolbar

To change the tempo of the global transport from the patcher toolbar, option- or alt-drag up or down on the play button control. The current tempo will be displayed in a caption above the control.

The transport object can also act as an interface to the global transport. This object lets you control all aspects of the transport, including play/pause state, tempo, and time signature, as well as retrieving the current play position.

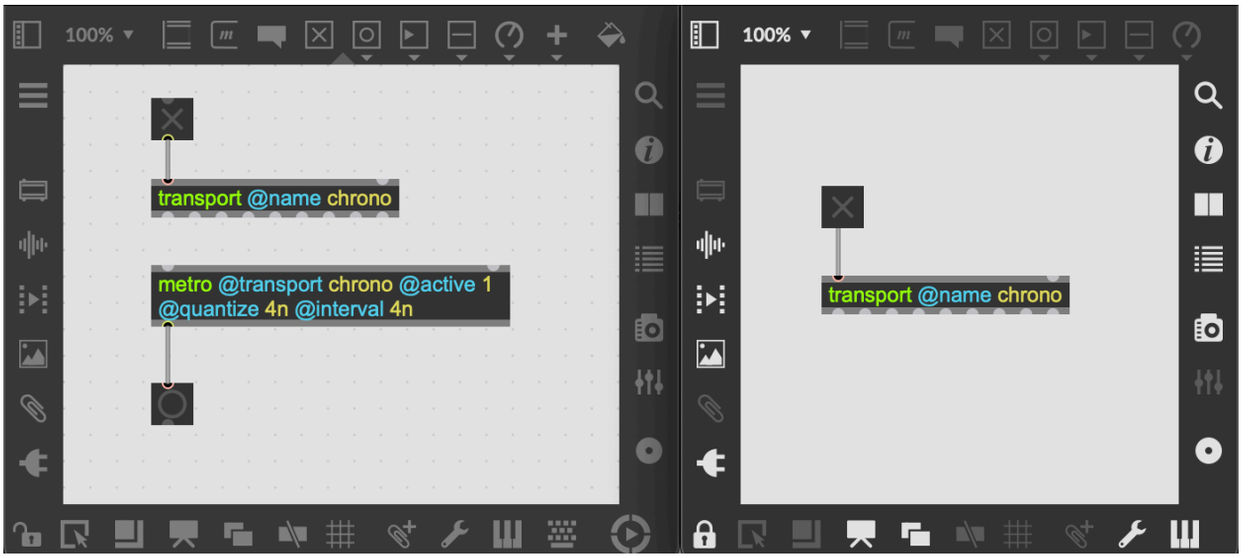
Named Transports

In addition to the global transport, it's also possible to create named transports using the `@name` attribute of a transport object. All transport objects with the same name essentially act as an

interface to the same, shared transport, and this includes transport objects in different Max patches.

> The Global Transport has the reserverd name `internal`. If you send a transport object the message `name internal`, that object will act as an interface to the global transport.

By default, objects like metro and timepoint refer to the Global Transport. To refer to a particular, named transport, use the `@transport` attribute of these objects.



Two {transport} objects with the same name are an interface to one shared transport. Starting one will start the other, and visa versa.

Max for Live

In a Max for Live device, the transport object will be bound to Live's transport. You can send the object a `bang` to get the current state of Live's transport, but you can't control Live's tranport this way. Instead, use the Max for Live API to interface with the Live application `link to the relevant doc here`. Other named transports—transports not bound to the Live transport—are not supported.

Transport Resolution

The resolution of the transport is always 480 PPQ (reference to the MIDI standard). When it comes to scheduling events, the transport uses the same **Scheduler** as all other events in Max, and so ultimately the resolution of the Max scheduler determines the resolution of transport events.

Max for Live

Max for Live

Example Devices 826
Learning Resources 827
Jitter and Max for Live 827
Max for Live Documentation 828

Max for Live allows you to harness the power and flexibility of Max inside Ableton Live. Using Max for Live, you can build your own devices and tools that enable you to completely control and shape how you work and create with Live. You can make:

- Audio Effects
- MIDI Effects
- Instruments
- MIDI Transformations and Generators

You'll be able to connect custom hardware controllers, design sequencers, process audio in novel and exciting ways, and much more.

Example Devices

Live Suite users have access to a wealth of packs on the packs store. From here you might get an idea of what Max for Live lets you build, including sequencers, custom note transformations, tools for controlling modular synthesisers, bespoke instruments and more.

There are also some parts of Ableton Live's software that are built using Max and it is a fully capable tool for making high-quality, professional devices. For example, Convolution Reverb, Granulator II and Granulator III are all devices made with a combination of Max and sometimes RNBO.

Lastly, one of the major strengths of Max and Max for Live is its community and the spirit of sharing that comes with it. There are lots of great devices that can help you feel inspired by other people,

often found at https://maxforlive.com.

Learning Resources

Max for Live extends how you can use Ableton Live through a combination of objects, abstractions and APIs. All of these different tools and capabilities are embedded in "devices". A device is file ending in the `.amxd` extension which can be loaded and used in Ableton Live, much like the existing library of devices that are shipped with Live. A Max for Live device can contain patchers, abstractions, media and anything that can be added to a Max collective. To begin putting all these building blocks together, we recommend the Building Max Devices pack. This pack guides you through building your own devices, while also offering over 90 devices that you can use as the basis of your own experimentation. You might also want to investigate both the MSP and Max tutorials to get a better handle on the mechanics of patching just in Max.

Another option to supplement this is to take a look at the Max sidebar and the Max for Live clippings and snippets. These include lots of helpful shortcuts for putting together your own work.

Jitter and Max for Live

You may find these tips useful if you want to create a Max for Live device using Jitter.

- If you are using Vizzie in a Max for Live device, you will need to use a PROJECTR module or a vs.projectr Vizzie abstraction in your device.
- If you are using jit.world in a Max for Live device, you will need an implicitly named context or a --- (triple-dash) named context.
- When you export a Max for Live project which includes a named context, a new instance of that context will be created during the export to the .amxd device. If the patch that includes that named context is open when you export the Project, you will see an error in the console that ob3d does not allow multiple bindings. The patch you're working on will no longer work as expected until you close it and then relaunch it after your export.
- Jitter operates in Ableton Live's interface update thread. This has two major effects:
    - Jitter frame updates will be interrupted by the Live application's interface redraws.

- The maximum frame rate is dependent on the Ableton application's audio buffer size.

Max for Live Documentation

Creating Devices

- Creating Devices Overview
- Creating Audio Devices
- Creating MIDI Effects
- Creating MIDI Tools
- User Interfaces
- Symbols
- Timing
- Limitations

Live API

- Live API Overview
- Creating Live API Devices

Parameters

- Automation
- Parameters
- Parameters Window
- Pattr

Sharing Devices

- Sharing
- Freezing

- Unfreezing
- Resolving Conflicts

Live API Overview

Live Object Model 830
Object Path 831
Root objects 831
Canonical Path 831
Canonical Parent 832
Object ids 832
Object Types 833
Children 833
Properties 833
Functions 833
Datatypes 834
Notifications 834
Max Objects 835
LiveAPI 837
Examples 837

Besides building new instruments and effects to be used in Live, Max For Live also allows to access Live itself, its tracks, clips, devices and hardware control surfaces. This chapter defines some basic terms used throughout the whole Live API and introduces the Max objects representing the Live API.

Live Object Model

The accessible parts of Live are represented by a hierarchy of objects called the *Live Object Model (LOM)*.

The model describes the hierarchy of objects inside Live, as seen from the Max devices. There are various types of objects in the model, like `Track` or `Clip`. For certain objects only a single instance exists, for other multiple instances are held in lists.

The Live Object Model reference shows how to navigate from a number of root objects down a path to the particular object of interest, and what to do with it. Not all of Live's parameters are

accessible via Live's API, the reference should give you an idea of what can and can't be done via Max for Live.

Object Path

Live objects are accessed using paths according to the Live Object Model. For example, the first clip in the third track can be accessed by the path `live_set tracks 2 clip_slots 0 clip`. Alternatively, it can be accessed via `live_set scenes 0 clip_slots 2 clip`. Or, if the clip is shown in the detail view, via `live_set view detail_clip`.

As you can see, different paths can point to the same Live object. Only one of these paths is the *canonical path* (see below).

When communicating with the Live API, no quotes are used in paths. List indexes start with 0.

When navigating through the object model, besides these *absolute* paths, *relative* paths can be used. These determine a subpath beginning at the current position in the object hierarchy.

Root objects

(Absolute) paths to all objects start with one of `live_app`, `live_set`, `control_surfaces N` or `this_device`. These are the *root* objects.

`live_app` : allows you to access controls of the Live application itself. This can be useful if you want to toggle the browser view, zoom or scroll features in Live. `live_set` : allows you to access various parameters within Live, for example Track Volume, Clip parameters (including launching Clips), etc. `control_surfaces` : allows you to access various control surface features (depending on your controller). `this_device` : allows you to construct API paths relative to the device you are in.

Canonical Path

Different paths can lead to the same object. `live_set view selected_track` and `live_set tracks 3` are the same object if the fourth track is selected.

Each object has a unique canonical path, `live_set tracks 3` in this case. The canonical path is sent out of live.object in reponse to `getpath`. In the Live Object Model, the canonical path is shown by bold connectors.

Canonical Parent

Additionally to what is described in the LOM, all objects have a `canonical_parent` child which is used by Live to determine the canonical path of an object. The canonical parents are get-only and useful for patching, too. For example, `goto this_device canonical_parent` is the perfect way to get the own track object.

Object ids

An object id identifies a particular object instance in Live like a track or a clip.

To get an id, a live.path object must be used to navigate to the Live object. When a live.path object sees this Live object the first time, an id is assigned to it.

The id is only valid inside the device with the live.path and remains unchanged as long the object exists. If the object is moved in Live, its id usually remains unchanged. There may be exceptions if the movement is implemented as a delete/create sequence, though. When an object is deleted and a new object is created at its place, it will get a new id.

An id is never reused in the scope of a Max device. Ids are not stored. Therefore, after loading a saved device, the live.path object must navigate to the object again.

An object id consists of the word `id` and a number, separated by a space, like `id 3`. `id 0` refers to *no object*. In Max terms it's a list of the symbol `id` and an integer.

Object Types

Each Live object is of a particular object type (or *class*), like `Track` or `Clip`. This object type determines what kind of object that is and what children, properties and functions it has. The object types are described in detail in the Live Object Model.

When live.object refers to a Live object, sending it `getinfo` will send all the Live object's children, properties and functions to its left outlet.

Children

Live objects have children identified by name. Some names, like `master_track` for the `Song` object type, point to single objects. Others, like `scenes`, point to a list of objects. The child name hints at which object type you can expect to find there.

List names are in plural, whereas single child names are in singular. Lists may be empty. Sending `getcount child_name` to live.path allows to find out how many children are in the list.

Single children names may point to no object, in which case you get `id 0` if you navigate there or send `get child_name` to live.object.

Most children can be monitored using live.observer.

Properties

Live objects have properties which describe its actual state. Properties are accessed by sending `get` and `set` messages to live.object. Not all properties can be set, though.

Many properties can be monitored using live.observer.

Functions

Many Live objects have functions which can be called by sending `call` and the function name to live.object, like `call create_scene` for a `Song` object. A function call may have parameters (a list of values). The return value will be sent out from the outlet of live.object.

Datatypes

Properties and function parameters or return values used in the Live Object Model and by the Max objects to access the Live API have one of the following data types:

Datatype	Description
bool	0 for false and 1 for true
symbol	a string with unicode character setUse double quotes in message boxes to create symbols with spaces: `set name "Smooth Synth"` Double quotes in symbols are to be *prefixed* by backslashes: `set name "Smooth \"Baby\" Synth"` Backslashes are to be included as double backslashes: `alpha beta \"gamma\" \\x\\` creates the symbol `alpha beta "gamma" \x\`.
int	a 32 bit signed integer
float	a 32 bit float value
double	a 64 bit float value (maily used for timing values)
beats	song beat time counted in quarter notes, represented as double
time	song time in seconds, represented as doubleTime is given in seconds: `time = beats * 60 / tempo_in_bpm`, or sometimes in milliseconds: `time = 1000 * beats * 60 / tempo_in_bpm`
list	a space separated list of the types above

Notifications

When Max devices need to know the state of the Live application and its objects, they can actively poll the state by navigating through the object hierarchy and getting object properties or calling functions.

But changes happen in Live while the Max device is passive. To allow the Max device to react on these changes in Live, notifications are sent from Live to the Max device. Notifications are spontaneous in the sense that messages are sent to outlets spontaneously, *not* in response to a message received at an inlet.

The notifications include object ids sent when the Live object at a certain path changes and values sent when a property changes.

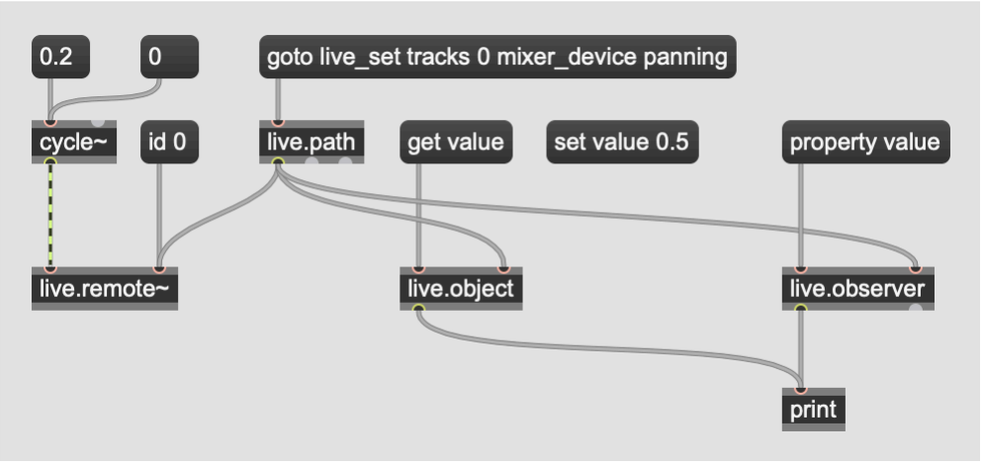
> Note: changes to a Live Set and its contents are not possible from a notification. The error message in the Max Console is 'Changes cannot betriggered by notifications'. In many cases putting a deferlow between the notification outlet and the actual change helps to resolve the issue.

Max Objects

Four Max objects interact in a certain way to allow Max devices to access the Live objects.

Max object	Purpose
live.path	select objects in the Live object hierarchy
live.object	get and set properties and children, call functions
live.observer	monitor properties and children
live.remote~	control Live device parameters in real time

The following patch shows the typical interconnections between the Live API objects. live.path is sending object ids out of its leftmost outlet connected to the rightmost inlet of live.object, live.observer and live.remote~. This causes these objects to operate on the object selected by live.path.



live.path

live.path objects are used to navigate to the Live objects on which live.object, live.observer and live.remote~ are supposed to operate. For this purpose, navigation messages like `goto live_set` are sent to live.path, which replies by sending an object id to the left outlet.

live.path can also observe the given path, and when the object at this path changes, its id is sent to the middle outlet. This is particularly useful for paths like `live_set view selected_track` which point to the currently selected track.

live.object

live.object is used to operate on a particular Live object which id has been received from live.path. It allows to get or set properties of the Live object and to call its functions with parameters.

live.observer

live.observer monitors the state of a particular Live object which id has been received from live.path. After telling live.observer which property to observe it recognizes all changes of the property and sends the current values to its left outlet.

live.remote~

live.remote~ receives the id of a DeviceParameter object from live.path and then allows to feed this parameter with new values by sending them into the left inlet, in realtime, without effects on the undo history or the parameter automation, which is deactivated.

DeviceParameter objects are children of Live devices, including Max devices, and also of tracks, like volume and pan.

LiveAPI

The Live API Javascript object is available in code written for the js object. It provides a succinct means of communicating with the Live API functions from JavaScript, incorporating the functionality provided by the live.path, live.object and live.observer objects.

Examples

Here are some examples of accessing Live from Max.

Controlling the volume slider of the selected track

Getting the volume of a track

To get the volume of a track in Live, we first need to find the path to the volume parameter of the track's mixer device.

In the Live Object Model reference, the graph can be navigated starting at the `live_set` root node. From the `Song` object, we use `tracks` to get the list of `Track` children. We select the second track with `1` (note that indexing starts at 0). We then use `mixer_device` to get its `MixerDevice` object, and finally by using `volume`, we get to the volume `DeviceParameter` object.

We can now send this path to a live.path with the following message: `path live_set tracks 1 mixer_device volume`. live.path will give us back `id n` (where n is an integer), which represents the `DeviceParameter` object we need. It gives us access to the properties of the mixer device's volume control, like its range (`min` and `max`), default value (`default_value`) or current value (`value`).

When we send this id to the right inlet of a live.object, we can get the properties of the DeviceParameter object with `get` messages. By sending `get value` to the live.object, it will send the current volume value to its left outlet as a number between 0 and 1.

Initializing a path when the device loads

If we want to make sure the live.object is set to the path we specified as soon as the device is loaded, we might be tempted to use loadbang to send the `path` message to live.path. It is important to know that we need to use live.thisdevice instead, to make sure the Live API is initialized before interacting with it.

Getting the volume of the *selected* track

In the previous example we hard-coded getting the volume value of the second track. However, if we would like to get the value of the *selected* track, we can use the `selected_track` child of the Song View instead.

After finding the `selected_track` keyword in the Live Object Model reference, we see that it is a child of `Song.View`. In the graph we see that the `Song.View` can be reached from the `Song` object type with the `view` keyword. So we send the following message to live.path : `path live_set view selected_track mixer_device volume`. This will send the id of the currently selected track to the left outlet, which we send to live.object.

Knowing when a different track is selected

In the previous example, we got the id of the selected track once. However, if we select a different track, this id is no longer up to date. If we want the id from a path containing a dynamic element like `selected_track` to stay up to date, we can use the second outlet of live.path.

Hovering the mouse cursor over the outlets over an object will show you what they send out. The second outlet of live.path reads `id: follows path`. When instead of the live.path 's left outlet we connect its second outlet to live.object, the live.object will be kept up to date, even after selecting a different track.

There is one catch, as mentioned under Notifications. Since the second outlet of live.path sends us notifications from Live, when we connect it to another Live API object like live.object, we need to add a deferlow in-between to make sure the next API interaction is not attempted before the first is finished.

Setting the volume of a track

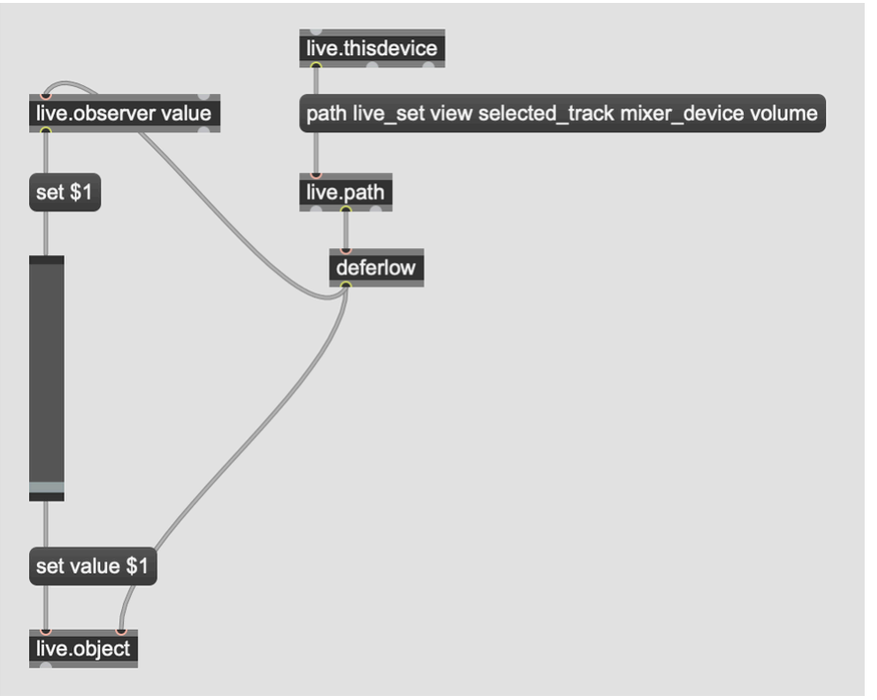
In much the same way as getting the current volume of a track like above, we can also *change* the value of a volume fader.

We can add a slider with a range between 0 and 1, connected to a message box saying `set value $1`. We can then connect this to the live.object as set up above, and now our slider will control the volume slider in Live.

Observing the volume of a track

We might want to get an update in our device whenever a Live user changes a volume slider. For this, we can send the id from the previously set up live.path to a live.observer that has its argument set to the property `value`. Now, the up to date volume values will be output from the live.observer 's left outlet.

Finally, we can set this value to the slider we created earlier. We now have a device with a slider that mimics and changes the volume slider of the selected track in Live.



Triggering a clip with MIDI notes

Navigating to a specific clip and triggering it

Apart from children and properties, Live objects also have *functions*. Looking at the Live Object Model reference, under `Clip`, in the Functions section, we can find the `fire` function. To call this function, we first need to refer a live.object to the clip we want to launch. For this, we need the clip's id.

Getting an id is done with live.path. Finding out what path to supply is most easily done by looking at the LOM graph. In the graph, we find the `Clip` object. To reach it, we start at the Song root node (`live_set`), go to its list of `Track` children and pick the first (`track 0`). Next we go to the list of `ClipSlot` children and pick, for example, the fourth (`clip_slots 3`), and finally we navigate to the `Clip` contained by this clip slot (`clip`). So the message we send to live.path will be `path live_set tracks 0 clip_slots 3 clip`.

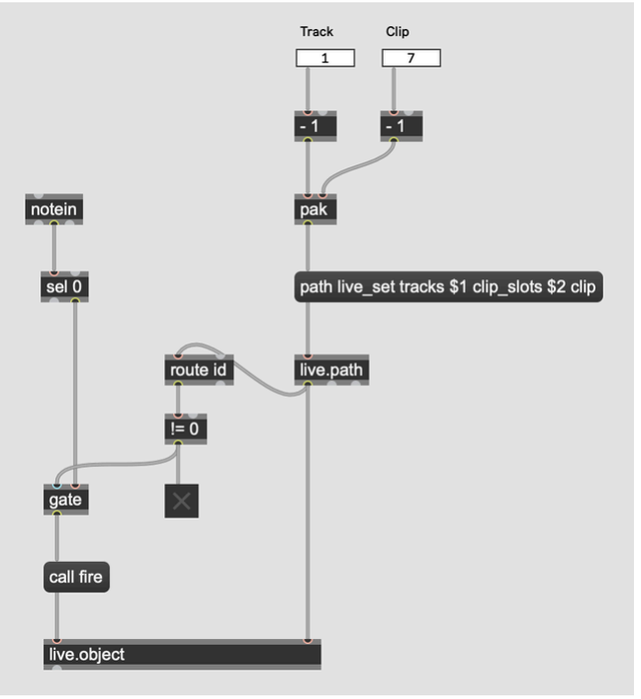
We send the id output from the left outlet of live.path to live.object 's second inlet. There are no dynamic elements in the path, so we don't need to use live.path 's second outlet. And now that we are ready to launch the clip, we simply send `call fire` to the left inlet of the live.object.

Adding control of which clip to fire and checking if it exists

We might want to use two live.numbox es to control the track number and the clip slot number that we want to launch. The outputs of the numboxes can be sent to pak and we can swap the hardcoded numbers in the path above with replaceable arguments:
`path live_set track $1 clip_slots $2 clip`.

Of course, with this approach, the user of this device will be able to select tracks or clip slots that don't exist in the Live set. To show a toggle that is on when the selection exists and off otherwise, we can use that `id 0` is output for paths that don't exist.

Finally, to make this a functional device, we can place this patch in a MIDI effect and use notein to trigger the selected clip. If the velocity is not 0, we know a note on comes in. So then, if the selected clip exists, we send `call fire` to it.


Track
Clip
1
7
- 1
- 1
notein
pak
sel 0
path live_set tracks $1 clip_slots $2 clip
route id
live.path
!= 0
gate
call fire
live.object

Creating Max for Live Devices

Creating a New Device 842
Saving a New Device 842
Copying a Device 843
See Also 843

When you use Max for Live to create new devices, you always begin from a basic *template* - a Max device that contains the necessary Max objects for receiving and transmitting audio or MIDI data to and from other devices in the Devices area of a Live track. When you edit a Max device, you launch the Max application, but you can continue to listen to your device in Live as you edit it thanks to a feature known as preview mode. When your Max for Live device is complete, you can save it (along with any presets you may wish to create) in a central library that allows you to load your patch or preset from the Live File browser at any time in the future.

Creating a New Device

- Click on the Devices button in Live's File browser to display the list of available devices.
- Click on the arrow to the left of the folder marked Instruments, MIDI Effects, or Audio Effects to show the available devices and their presets.
- Click on a Max for live device in Live's File Browser and drag it to the Devices Window for a Live Track.

Saving a New Device

- To save a device, click on the close button in the patcher's title bar or choose **Save** from the Max File menu. If you are saving a newly created device, a file dialog will appear and ask where you want to save the file.
- To see the current library of available devices, click on the arrow to the left of any of the three Max for Live device folders (the `Max Audio Effects` folder in the Audio Effects folder, the `Max MIDI Effects` folder in the MIDI Effects folder, or the `Max Instruments` folder in the Instruments folder). The folder will open and any saved devices will be displayed.

Copying a Device

- To save a copy of a Max for Live device you want to edit, choose **Save As...** from the File menu. A file dialog will appear so you can give your device a new name.
- When you save the device, the copy in Live will update automatically.

See Also

- Creating Audio Devices and Instruments
- Creating MIDI Effects
- Creating MIDI Tools
- Creating Live API Devices
- User Interfaces in Max for Live
- Working with Files
- Parameters

User Interfaces in Max for Live

Device Width and Height 844
Defining a Fixed Device Width 844
Using a Dynamic Device Width 844
Using Presentation Mode 844
Creating a Presentation 845

Device Width and Height

By default, the *width* of a device created in Max for Live is based on the contents of your device patcher. It will be slightly wider than the visible objects in your device. Alternatively, you can explicitly define the width of your device to any size. If you change your mind, you can always reset the device width to the default behavior. The *height* of all Live devices is fixed at 169 pixels.

Defining a Fixed Device Width

- While your Max for Live device patcher window is open for editing, resize the window to be desired width you would like to see in Live.
- Choose **Set Device Width** from the View Menu.
- If desired, you now can resize the patcher window to a size convenient for editing. You will see a vertical line indicating the fixed width of the device. The width will update in Live the next time you save the device.

Using a Dynamic Device Width

- While your Max for Live device patcher window is open for editing, choose **Clear Device Width** from the View Menu. The next time you save the device, the width in Live will be determined by the contents of your patcher.

Using Presentation Mode

Once you have your Max device up and running, you may want to present the user with only the user interface objects. Given the relatively small height of the Live Device view, it may be desirable to think of your interface separately from the logical position of the user interface objects in your patcher.

Creating a Presentation

- In the unlocked Patcher window for your Max for Live device patch. select the user interface objects you want to be visible in the Device Window by shift-clicking to select them.
- Choose **Add to Presentation** from the Max for Live Object menu to add the objects to the Presentation layer. A pink border will appear around the object(s) you have selected.
- Click the Presentation Mode button in the patcher toolbar to enter Presentation mode. When you switch to Presentation Mode, only objects you have added to the Presentation layer will be shown and the word **(presentation)** will appear in the title bar of the patcher window.
- While in Presentation Mode, you can reposition and change the color of user interface object, and also use resizing handles to change the size of some user interface objects (if an object can be resized, it will had resizing handles.
- When you are satisfied with the layout of your user interface, choose **Patcher Inspector** from the Max for Live View menu.
- Check the `Open in Presentation` attribute.

Automation

Modulating a Max for Live Device Parameter 846
Parameter Automation Data at Audio Rates 846
See Also 847

Parameters of a device whose `Parameter Visibility` attribute is set to Automated and Stored are available in Live's automation editor (visible in the Arrangement view). The automation editor permits you to draw curves or other control information that will change parameter values automatically as the arrangement plays. The parameter data type determines the type of automation editor shown. More on this topic soon. Another way parameter values can be changed automatically is by using clip envelopes to modulate parameter values. While automation sets the absolute value of a parameter, modulation adjusts the value up or down from its current value.

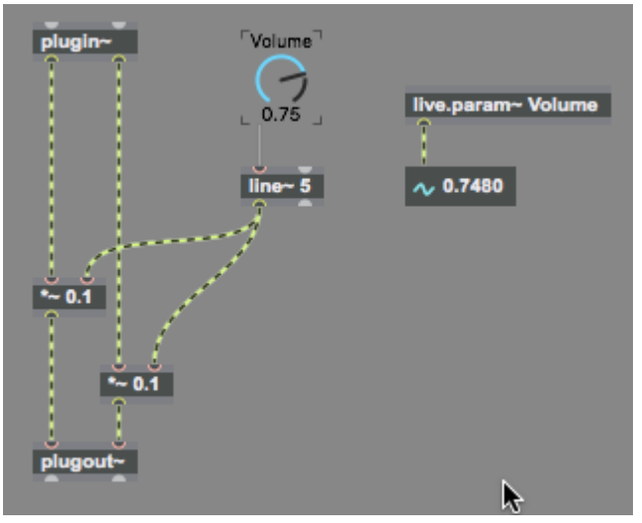
Modulating a Max for Live Device Parameter

- Click on a Max for Live user interface object to select it and click on the Inspector icon in the toolbar to show the Inspector.
- If you have not already done so, choose Int or Float from the `Type` attribute's pull-down menu to set the parameter type.
- Choose the modulation mode you want to use from the `Clip Modulation Mode` attribute's pull-down menu to set the parameter.
- In the Live application, select the parameter, choose a modulation source, and enable modulation as you would with any normal Live device parameter.

Parameter Automation Data at Audio Rates

While getting parameter automation data from the Live application at message rates for use in your Max for Live device is common, some devices require that you receive parameter automation data from the Live applcation at higher and more accurate rates. Any Max for Live device can receive its parameter automation data in the form of a sample-accurate audio-rate ramp using the live.param~ object.

The live.param~ object takes as an argument the name of an automatable parameter to which it is bound. Any change to the automatable parameter's output value will be send out the outlet of the live.param~ object at signal rate.



See Also

- Parameters
- Pattr

Sharing Max for Live Devices

To create Live devices that you can give to other people, you'll need to ensure the device is packaged with all the files it needs. We call this freezing.

> The use of the term *freezing* here should not be confused with the Max concept of frozen attributes or frozen tracks in Ableton Live.

Basic freezing is explained here. If your device opens files during its operation (such as a folder of samples or images), you may need to add these files as dependencies before freezing. Once you freeze a device, it cannot be edited. You'll need to unfreeze it first.

If you open a frozen device for editing and see this icon in the patcher toolbar



you'll need to resolve conflicts between the files in the frozen device and those you have on disk before you can unfreeze.

Timing and Synchronization in Max for Live

Implementation Notes 849

The Live transport is a clock source for the Max tempo-based time system.

Implementation Notes

- By default, all transport objects in Live devices (those without names) synchronize to the Live transport.
- Named transport objects do not synchronize to Live unless you set the `clocksource` attribute to the name `live`.
- If two device instances contain transport objects that share the same name, they will run independently.
- The tempo-based timing system synchronizes to Live even in preview mode. However, there may be disruptions in the continuity of timing when switching into or out of preview mode.

Max for Live Extended

Creating Audio Effect Devices

Defining Latency For A Device 851

> A great way to start making any kind of device is to check out the Building Max Devices pack. We highly recommend this as a tool for learning how to make specifically *devices*, both Audio and MIDI.

Max for Live lets you create audio effects devices which receive audio input from the Live application process it in some manner, and pass its audio output either back to the Live application, or to other downstream audio effects devices in the same audio track where the device resides. By convention, a Max for Live device gets all its audio from the Live application using the plugin~ object and sends its audio output using the plugout~ object. Audio input and output is limited to two channels.

> Sending audio to another Max for Live Audio Effect, Instrument, or MIDI Effect device using the Max send, receive, send~, or receive~, objects is not supported. While creating a Max for Live audio device can begin by using the Max for Live device templates, you can use some of the Max for Live Audio devices, MIDI effects, and Instruments that come with Max for Live as your starting point. Read more about Max for Live limitations.

Defining Latency For A Device

When Live sends audio or when MIDI triggers audio through an effect created using Max for Live, the events should all be time-aligned (e.g., if a MIDI note falls on the downbeat, the MIDI Instrument's audio should also end up in the mix on the downbeat). A device can provide latency

information to the Live host application so that the host can use *latency compensation* to adjust the relative timing of different audio tracks.

When you use Max for Live, there are situations where it is useful to set latency to counteract timing differences introduced in your signal processing. If your signal processing patch requires you perform some kind of analysis on a block of samples before any output is produced, you can enable latency to make sure that the output from your effect will be correctly aligned in the mix (whereas if you are creating a device in which signal delay is what you intend, there's no problem and you don't need to set any latency).

Setting Device Latency

- With a device window as the topmost window, choose **Patcher Inspector** from the View menu to show the Inspector.
- Double-click in the Value column for the Defined Latency attribute to show a cursor and text box. Type in a value for the latency value in *samples*, followed by a carriage return.

Creating Devices that use the Live API

Querying the Live API (getting data) using Max for Live objects 854
Querying the Live API 855
Setting a property in the Live API using Max for Live objects 856
Setting a Live API property 858
Observing a property in the Live API using Max for Live objects 859
Observing a Live API property 861
Calling a function in the Live API using Max for Live objects 863
Calling a function of a Live API property 864
Automating device parameters at signal rate 865
Controlling a property using the live.remote~ object 867

Max for Live provides two different ways to access the Live application directly through the Live API:

- You can use a trio of Max for Live objects - (live.object, live.observer, and live.path) to access, observe, and control the Live application.
- You can use the Max js object to write code using the Live API Javascript object that exposes the Live Object Model.

Regardless of which method you decide to use, online documentation for the Live Object Model describes the properties and functions of a Live session that can be queried and set and observed. You can get or set values, call functions, and observe the status of properties using the Live API from any kind of Max for Live device on any channel. The Live API is described in this overview.

The Live Object Model divides the Live application into several basic functional units (*properties*) associated with aspects of the Live application - the application itself, Songs, Tracks, Clip slots, Clips, Devices, Device Parameters, the Mixer Device, Scenes, Cue Points, And Control Surfaces. The Live API provides ways to access some properties of the application to control how the Live application displays them to you (Application.View, Song.View, and Track.View). Your use of the Live API involves one of four kinds of different operations:

- You can query (*get*) the current state of a property of your current Live session.
- You can *set* the state of some properties of your current Live session
- Some properties can be controlled using *functions* that perform various actions (e.g. firing a clip).
- Some properties can be *observed* (i.e. their current state is reported and updated automatically). The Live Object Model provides a complete reference to which objects can be queried, set, and observed, as well as a listing of the functions associated with them.

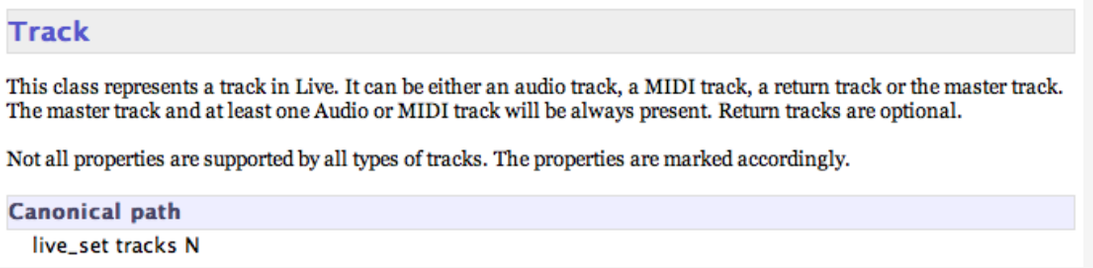
Querying the Live API (getting data) using Max for Live objects

You can use the Max for Live live.path and live.object objects to find out the current state of any property defined in the Live Object Model reference.

The live.path object is used to navigate to the Live object properties you wish to query. Each property of the Live object model in a session is associated with an id specific to that particular Song, Track, Clip, Clip Slot, Device, etc. Sending a message to a live.path object results in an object id being sent out the left outout (where the id follows the object) or middle outlet (where the id follows the path). In turn, the live.object object takes the id message from the live.path object and lets you use `get` messages to get information about the properties of the object. The result of the query is sent out the outlet of the live.object object, preceded by the name of the query.

Querying the Live API

- Using the Live Object Model, find the canonical path listing for the property (in this case, the Track properties). This part of the Live Object Model listing displays its canonical path - the syntax for queries about the Track in a Live session.



Track

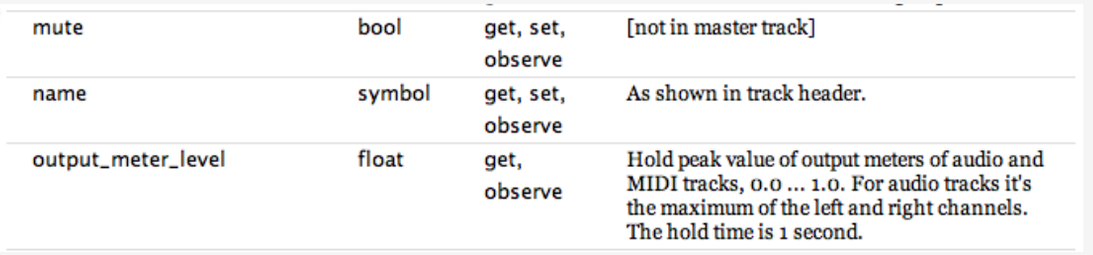
This class represents a track in Live. It can be either an audio track, a MIDI track, a return track or the master track. The master track and at least one Audio or MIDI track will be always present. Return tracks are optional.

Not all properties are supported by all types of tracks. The properties are marked accordingly.

Canonical path

live_set tracks N

- Find the listing in the Live Object Model page for the query you want to make (in this example, we want to to see whether or not Track 3 is muted).



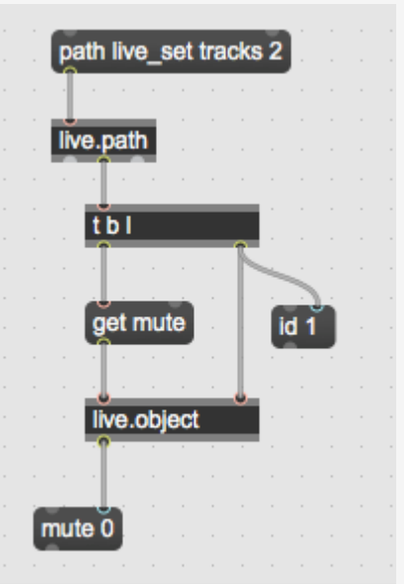
mute	bool	get, set, observe	[not in master track]
name	symbol	get, set, observe	As shown in track header.
output_meter_level	float	get, observe	Hold peak value of output meters of audio and MIDI tracks, 0.0 ... 1.0. For audio tracks it's the maximum of the left and right channels. The hold time is 1 second.

The listing from the Live Object Model listing for the `mute` property of a Live Track lists its Type as *bool* (boolean, 0 = off, 1 = on), and its Access as *get*, *set*, *observe*, which means that this property can be queried or set or observed (there's also an indication that Live Master Tracks cannot be queried or set).

- Use amessage boxto construct a message to be sent to the live.path object telling it the name of the property we want to query - Track 3 (by convention, track numbering in the Live API starts from 0). In this case, the message is `path live_set tracks 2` .

The live.path object responds with the message `id N`, where *N* is the id number associated with Track 3.

- use a Max trigger object to set up a sequence of operations. When the trigger object receives the list output, it sends the id message to the live.object object's right inlet, and then sends a bang message to themessage boxcontaining the symbol `get`, followed by the name of the property being queried. In response, the live.object object sends the message `mute 0` out the left outlet of the live.object object



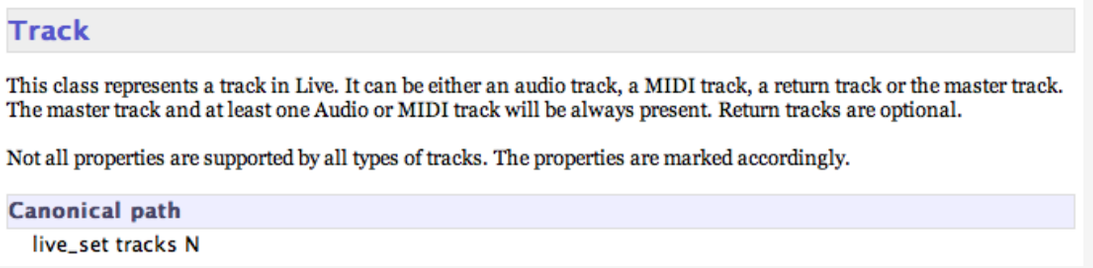
Setting a property in the Live API using Max for Live objects

You can also use the Max for Live live.path and live.object objects to set the current state of many properties defined in the Live Object Model reference. The live.path object is used to navigate to the Live objects you wish to query. Each property of the Live object model in a session is associated with an id specific to that particular Song, Track, Clip, Clip Slot, Device, etc. Sending a message to a live.path object results in an object id being sent out the left outout (where the id follows the object) or middle outlet (where the id follows the path). In turn, the live.object object takes the id message from the live.path object and lets you use `set` messages, followed by arguments that

specify the new setting to set the state of a property. In this example, we'll use a `set` message to mute Track 3 of a Live session.

Setting a Live API property

- Using the Live Object Model, find the canonical path listing for the property (in this case, the Track properties). This part of the Live Object Model listing displays its canonical path - the syntax for queries about the Track in a Live session.



Track

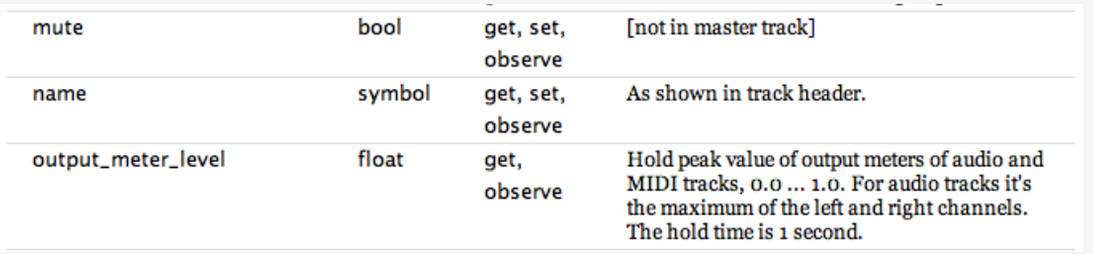
This class represents a track in Live. It can be either an audio track, a MIDI track, a return track or the master track. The master track and at least one Audio or MIDI track will be always present. Return tracks are optional.

Not all properties are supported by all types of tracks. The properties are marked accordingly.

Canonical path

live_set tracks N

- Find the listing in the Live Object Model page for the query you want to make (in this example, we want to to see whether or not Track 3 is muted).



mute	bool	get, set, observe	[not in master track]
name	symbol	get, set, observe	As shown in track header.
output_meter_level	float	get, observe	Hold peak value of output meters of audio and MIDI tracks, 0.0 ... 1.0. For audio tracks it's the maximum of the left and right channels. The hold time is 1 second.

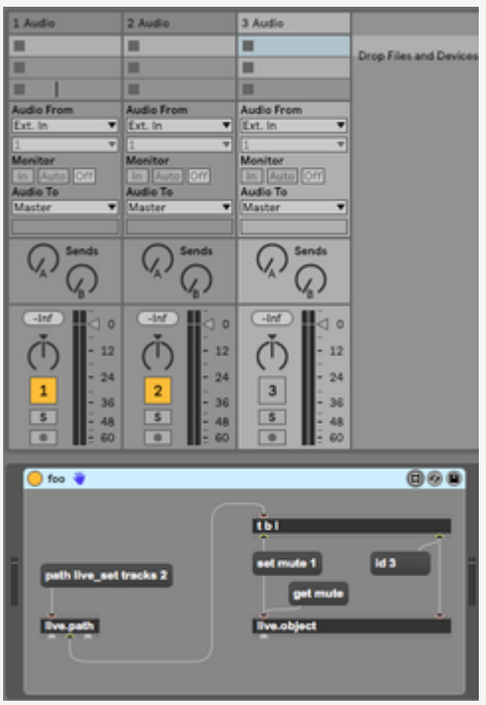
The listing from the Live Object Model listing for the `mute` property of a Live Track lists its Type as *bool* (boolean, 0 = off, 1 = on), and its Access as *get*, *set*, *observe*, which means that this property can be set as well as queried and observed (there's also an indication that Live Master Tracks cannot be queried or set).

- Use amessage boxto construct a message to be sent to the live.path object telling it the name of the property we want to query - Track 3 (by convention, track numbering in the Live API starts from 0). In this case, the message is `path live_set tracks 3` .

The live.path object responds with the message `id N`, where *N* is the id number associated with Track 3.

- Use a Max trigger object to set up a sequence of operations. When the trigger object receives the list output, it sends the id message to the live.object object's right inlet, and then sends a bang message to themessage boxcontaining the message `set`, followed by the name of the property we want to set and its new value - in this case, `set mute 1`.

When you save and close the device and click on the upper message box, Track 3 will be muted.



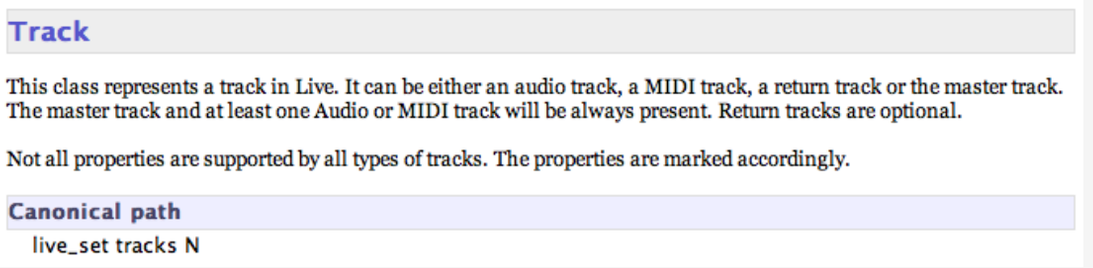
Observing a property in the Live API using Max for Live objects

Some properties in the Live API can be *observed*. The Live API not only reports the current state of a property in response to a query when it is observed, but also subsequently updates the state of that

property if it changes. Observing a property using the Live API uses the live.path object also used for getting and setting properties, but also uses the live.observer objects to perform the task. The live.path object is used to navigate to the Live objects whose functions you want to call. Each property of the Live object model in a session is associated with an id specific to that particular Song, Track, Clip, Clip Slot, Device, etc. Sending a message to a live.path object results in an object id being sent out the left outout (where the id follows the object) or middle outlet (where the id follows the path). In turn, the live.observer object takes the id message from the live.path object and lets you use `property` messages to define what property of the object you want to observe. In this example, we'll observe whether or not Track 3 of our session is muted or not.

Observing a Live API property

- Using the Live Object Model, find the canonical path listing for the property (in this case, the Track properties). This part of the Live Object Model listing displays its canonical path - the syntax for queries about the Track in a Live session.



Track

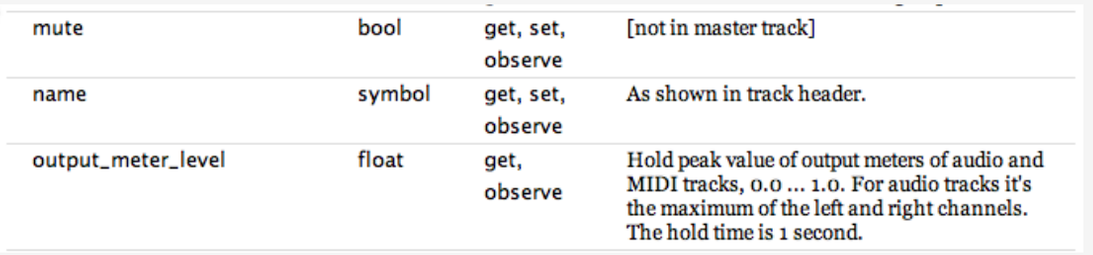
This class represents a track in Live. It can be either an audio track, a MIDI track, a return track or the master track. The master track and at least one Audio or MIDI track will be always present. Return tracks are optional.

Not all properties are supported by all types of tracks. The properties are marked accordingly.

Canonical path

live_set tracks N

- Find the listing in the Live Object Model page for the property you wish to observe. Not all properties may be observed via the Live API (in this example, we want to to observe the behavior of muting on Track 3).



mute	bool	get, set, observe	[not in master track]
name	symbol	get, set, observe	As shown in track header.
output_meter_level	float	get, observe	Hold peak value of output meters of audio and MIDI tracks, 0.0 ... 1.0. For audio tracks it's the maximum of the left and right channels. The hold time is 1 second.

The listing from the Live Object Model listing for the `mute` property of a Live Track lists its Type as *bool* (boolean, 0 = off, 1 = on), and its Access as *get*, *set*, *observe*, which means that this property can be observed (there's also an indication that Live Master Tracks cannot be observed).

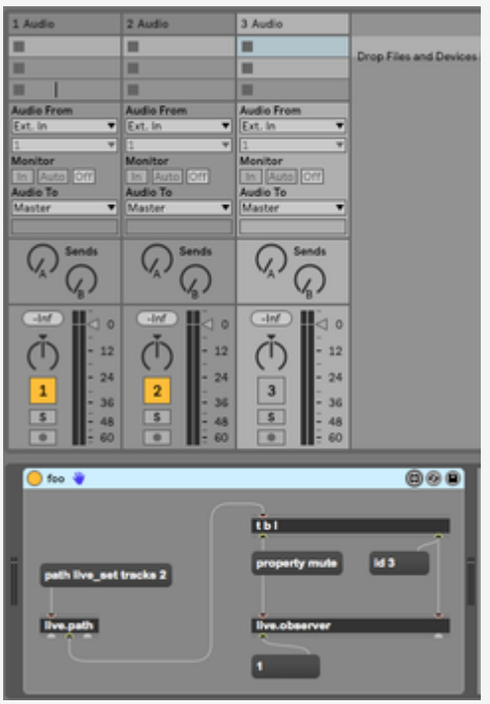
- Use amessage boxto construct a message to be sent to the live.path object telling it the name of the property we want to query - Track 3 (by convention,

track numbering in the Live API starts from 0). In this case, the message is `path live_set tracks 3` .

The live.path object responds with the message `id N` , where *N* is the id number associated with Track 3.

- Use a Max trigger object to set up a sequence of operations. When the trigger object receives the list output, it sends the id message to the live.observer object's right inlet, and then sends a bang message to themessage boxcontaining the message `property` , followed by the name of the property we want to observe - in this case, `property mute` .

When you save and close the device and click on the upper message box while Track 3 is playing, you'll notice that the patch displays a 0 (unmuted). If you mute the track by clicking on the Track Activator button, you'll see the output of the live.observer object change to a 1 (muted). As you unmute and mute the track, the value will change.



Calling a function in the Live API using Max for Live objects

The Live API also includes various kinds of functions that are used to perform activities such as changing various aspects of the Live application's interface display (View) or controlling the playing of clips or scenes. You can also use the Max for Live live.path and live.object objects to *call* (perform) these functions. The live.path object is used to navigate to the Live objects whose functions you want to call. Each property of the Live object model in a session is associated with an id specific to that particular Song, Track, Clip, Clip Slot, Device, etc. Sending a message to a live.path object results in an object id being sent out the left outout (where the id follows the object) or middle outlet (where the id follows the path). In turn, the live.object object takes the id message from the live.path object and lets you use `call` messages, followed by arguments that specify the name of the function you're calling and any data for the function in the form of arguments to execute the function. In this example, we'll use a `fire` function to play the clip in slot 2 or Track 2 of a Live session.

Calling a function of a Live API property

- Using the Live Object Model, find the canonical path listing for the property (in this case, the ClipSlot properties). This part of the Live Object Model listing displays its canonical path - the syntax for queries about the ClipSlot in a Live session.

ClipSlot

This class represents an entry in Lives Session view matrix.

The properties playing_status, is_playing and is_recording are useful for clip slots of group tracks. These are always empty and represent the state of the clips in their subtracks.

Canonical path

live_set tracks N clip_slots M

- Find the listing in the Live Object Model page for the property's functions (in this example, we want to fire (launch) the clip in clip slot 2 on Track 2).

Functions

Name	*Description*
fire	Fires the clip or triggers the stop button, if any. Starts recording if slot is empty and track is armed. Starts recording of armed and empty subtracks if Preferences->Launch->Start Recording on Scene Launch is ON.

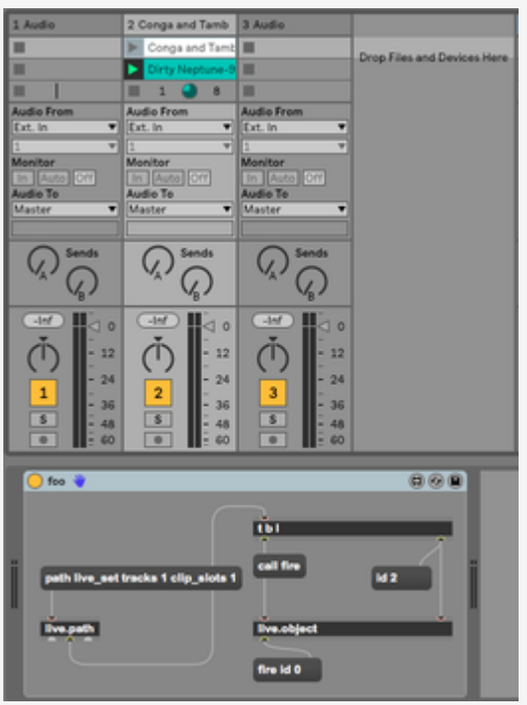
The listing from the Live Object Model listing for the `fire` function of the ClipSlot property indicates that it needs no other data arguments.

- Use amessage boxto construct a message to be sent to the live.path object telling it the name of the property we want to query - Clip Slot 2 or Track 2 (by convention, all numbering in the Live API starts from 0). In this case, the message is `path live_set tracks 1 clip_slots 1` .

The live.path object responds with the message `id N`, where *N* is the id number associated with Clip Slot 2 of Track 2.

- Use a Max trigger object to set up a sequence of operations. When the trigger object receives the list output, it sends the id message to the live.observer object's right inlet, and then sends a bang message to themessage boxcontaining the message `call`, followed by the name of the function - in this case, `call fire`.

When you save and close the device and click on the upper message box while Track any other clip in any other clip slot in Track 2 is playing, you'll notice that the clip in Clip Slot 2 is launched.

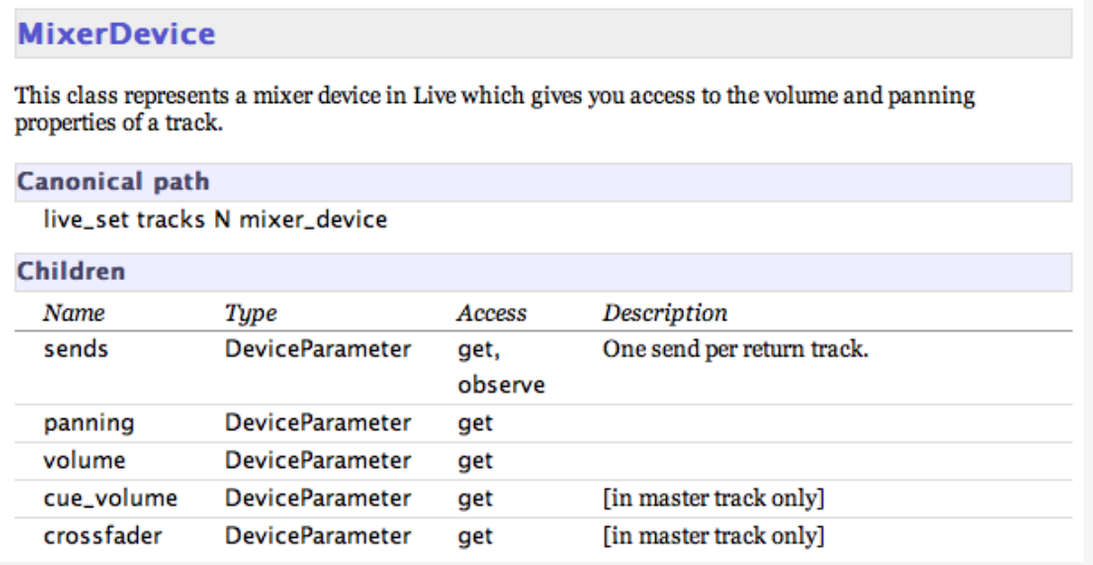


Automating device parameters at signal rate

The live.object is designed to mimic user interactions with the Live Session (and adds to undo history), so there are some situations that involve rapid modulation of device parameters where the object may not be appropriate. The live.remote~ object allows you to directly modulate the parameters of any "remotely mappable" control in Live at signal rate. As with setting values and calling functions, the live.path object is used to navigate to the Live objects whose functions you want to control. Each property of the Live object model in a session is associated with an id specific to that particular Song, Track, Clip, Clip Slot, Device, etc. Sending a message to a live.path object results in an object id being sent out the left outout (where the id follows the object) or middle outlet (where the id follows the path). In turn, the live.remote~ object takes the id message from the live.path object and accepts signal data in its left inlet which is used to modulate or control the Live API property. In the following example, we'll use the output of a cycle~ object to control the sends on an audio track.

Controlling a property using the live.remote~ object

- Using the Live Object Model, find the canonical path listing for the property (in this case, the MixerDevice properties). This part of the Live Object Model listing displays its canonical path - the syntax for queries about the Track in a Live session.



MixerDevice

This class represents a mixer device in Live which gives you access to the volume and panning properties of a track.

Canonical path

live_set tracks N mixer_device

Children

Name	*Type*	*Access*	*Description*
sends	DeviceParameter	get, observe	One send per return track.
panning	DeviceParameter	get	
volume	DeviceParameter	get	
cue_volume	DeviceParameter	get	[in master track only]
crossfader	DeviceParameter	get	[in master track only]

- Find the listing in the Live Object Model page for the property you wish to control (in this example, we want to to control the first send on Track 1). Remember that numbering in the Live API starts from zero by convention.
- Use amessage boxto construct a message to be sent to the live.path object telling it the name of the property we want to control. In this case, the message is `path live_set tracks 0 mixer_device sends 0` .

The live.path object responds with the message `id N`, where *N* is the id number associated with the first send on Track 1.

- Add some logic to produce signal-rate data to control the property. In this case, we're using a cycle~ device to use a sinusoidal waveform and then using the abs~ (absolute value) object to keep the output in the positive signal range.

MixerDevice

This class represents a mixer device in Live which gives you access to the volume and panning properties of a track.

Canonical path

live_set tracks N mixer_device

Children

Name	*Type*	*Access*	*Description*
sends	DeviceParameter	get, observe	One send per return track.
panning	DeviceParameter	get	
volume	DeviceParameter	get	
cue_volume	DeviceParameter	get	[in master track only]
crossfader	DeviceParameter	get	[in master track only]

When you save and close the device and click on the upper message box while Track 1 is playing, you'll notice that the dial for Send A on Track one moves at the rate you specify for the cycle~ object.

Creating MIDI Effects

Capturing a portion of the MIDI data stream in Max for Live 869
Redirecting a part of a MIDI data stream 870

> A great way to start making any kind of device is to check out the Building Max Devices pack. We highly recommend this as a tool for learning how to make specifically *devices*, both Audio and MIDI.

Max for Live lets you create MIDI effects devices which receive MIDI information from the Live application, perform some operation on that data (e.g. mapping incoming notes to a user-definied musical scale), and then pass MIDI data downstream to another MIDI effect or to a MIDI Instrument - the last device in a MIDI effects chain (although the audio output of the instrument may be further processed by any number of Audio effects devices). By convention, Max for Live gets all its MIDI data from the Live application using the midiin object, and all of that data must all be passed downstream to other MIDI devices by means of the midiout object. Max for Live provides a means to redirect portions of an incoming MIDI data stream for processing while leaving the rest of the stream unchanged, as described below.

Also by convention, all Max for Live MIDI devices receive audio on MIDI channel 1. You can receive receive MIDI data in your Max for Live MIDI Effects device on a channel number other than 1 by routing the MIDI input as described here. Max for Live provides several tutorials and example MIDI effects devices you can try, study, and modify:

Capturing a portion of the MIDI data stream in Max for Live

Max for Live requires that all MIDI data coming into a MIDI Effects device be passed downstream. There are situations in which you may want to use only a portion of the incoming MIDI data stream - to select only MIDI notes in a given range, or input from specific MIDI continuous controllers for use in your device. The Max midiselect object lets you specify portions of an incoming MIDI data

stream to filter out for further processing in your device, and passes any other MIDI data out unchanged.

Redirecting a part of a MIDI data stream

- In your unlocked patch, add a midiselect object and connect its inlet to the outlet of the midiin object in your MIDI Effect or MIDI Instrument template file.
- Following the name of the midiselect object enter the attributes corresponding to the MIDI data you want to filter out of the data stream, followed by any arguments that specify additional data (MIDI channel numbers, MIDI controller numbers, etc.). Attributes names begin with an at-sign (@), and there is no space following it and the type of data you want to filter out. The data types are:
- **ch:** MIDI channel (only used when routing MIDI data from a non-channel 1 source, as described below)
- **note:** MIDI note data
- **ctl:** MIDI controller data
- **bend:** pitchbend data
- **pgm:** MIDI program change messages
- **touch:** Aftertouch data
- **poly:** Polyphonic aftertouch data
- Connect the input of the midiout object in your MIDI Effect or MIDI Instrument template file to the right outlet of the midiselect object. Any data you do not specify when you instantiate the object or data you select by sending messages to the midiselect object will be passed downstream without being edited.

Device Parameters in Max for Live

Live UI Objects 871
Parameter Data Types 872
Setting an initial state for a Live UI object 873
Setting an initial state for a standard Max UI object 874
Parameter Modulation 874
Enabling Parameter Modulation 874
Parameter Names 875
Setting the display name for a parameter 875
Setting a custom unit style 876
Controlling a parameter's visibility 876

Parameters are settings of Max for Live devices you want to store and/or automate in Live. In some cases, a parameter may be set once and never change. In other cases, you'll want to use Max objects to interact with parameter values by clicking and moving the mouse, by receiving MIDI data mapped to a parameter, or via Live automation.

There are a few ways to add parameters to your device. The most straightforward method is to use Live UI objects. By default, each Live UI object and its corresponding value is stored in the Live set. Alternatively, you are able to configure Max UI objects to also have this behaviour and be stored in the Live document by setting Parameter Mode Enable to true. Another option is to use pattr.

The Parameters Window shows all parameters currently associated with a device, and permits you to change parameter attributes in a single place. You can also change parameter attributes for individual objects by using the Parameter tab of the Inspector.

Live UI Objects

Max for Live includes user interface objects designed to work with the parameter system whose names all begin with `live.`. These objects have some special abilities:

- They allow you to set an initial state that will be recalled automatically when a device is instantiated, saved, or edited.
- They work seamlessly with Live's MIDI and keyboard mapping capabilities.
- If you choose to make the UI object's parameter automatable, you can control it with Live's automation facility.

In other respects, the Live UI objects act like ordinary Max user interface objects.

Parameter Data Types

Parameters used in Max for Live can be one of four types:

- integer: integer values with a range of up to 256 values (default 0-255)
- floating-point: floating point values (no range restriction)
- enum: an enumerated list of items
- blob: parameters that cannot be automated but can be stored in presets. Non-automatable parameters may be any type of data you can store with a pattr object: single values, lists, or strings.

Working with Integer parameters requiring more than 256 values

The native integer representation is limited to the 0-255 range. If you need an integer type that exceeds this range then instead set the type to **Float**, and change the unit style to **Int**

- Select an object and click the Inspector icon on the Patcher toolbar to show the object's Inspector
- Choose **Float** from the pull-down menu in the Type attribute's Value column.
- Choose **Int** from the pull-down menu in the Unit Style attribute's Value column.

The native integer representation is limited to 256 values, with a default range of 0-255. When working with Live UI objects whose integer values will exceed this range, the Type attribute should be set to Float, and the Unit Style attribute should be set to Int:

Although the parameter value will be stored as a floating-point number, it will be displayed as an integer.

The native integer representation is limited to 256 values, with a default range of 0-255. When working with Live UI objects whose integer values will exceed this range, the Type attribute should be set to Float, and the Unit Style attribute should be set to Int:

Setting an initial state for a Live UI object

- Select the Max for Live UI object and click the Inspector icon in the Patcher toolbar to show the object's Inspector.
- Scroll down to the Parameter section to see the Parameter attributes.
- Check the Initial Enable checkbox.
- Enter a floating-point value for the Initial Value attribute.

Setting an initial state for a standard Max UI object

- Add a pattr object to your patch, typing the parameter name you want to use as an argument to the object. Connect the pattr object's middle (bindto) outlet to the Max UI object.
- Select the pattr object and open the Inspector. Scroll down to the Parameter section to see the Parameter attributes.
- Check the Parameter Mode Enable checkbox.
- Check the Initial Enable checkbox.
- Enter a floating-point value for the Initial Value attribute.

Parameter Modulation

Parameters can be modulated by clip envelopes in Live according to one of four modes:

- In *unipolar* mode, the parameter value is modulated between the minimum range value (set using the `Clip Modulation Range` attribute) and its current value.
- In *bipolar* mode, the full modulation range of a parameter is equal to twice the distance between the current value and nearest boundary set using the parameter's `Clip Modulation Range` attribute. If the current value is exactly halfway between the lower and upper ranges, the modulation range is equal to the total parameter range.
- In *additive* mode, the modulation range from the current value is equal to plus or minus one-half of the total range of the parameter. Values are truncated if they fall outside of the `Clip Modulation Range` attribute.
- In *absolute* mode, the current value is either the upper or lower bound of the modulation range. If the current value is less than half of the full parameter range, the modulation assumes a lower range of the current value minus the modulation range. If the current value is greater than half of the full parameter range, the modulation assumes the upper range is current value and the lower range is equal to the current value minus the modulation range value.

Enabling Parameter Modulation

- Select the parameter object (pattr or Live UI) and open the inspector. Click the Parameter tab to show the parameter attributes.
- Set the `Clip Modulation Mode` and, if applicable, the `Clip Modulation Range` attributes.

Parameter Names

Max for Live provides several ways to give a parameterised object a name using attributes. These attributes can be set using the guidelink[Inspector]#inspector.

- The `Scripting Name` attribute can be used to identify a UI object when used in conjunction with the Max pattr preset objects. When a UI object has a scripting name set, it will automatically appear in the pattrstorage object's inventory of parameter names when you add an autopattr object to your Max patch as described in the pattr Chapter 2 tutorial.
- The `Short Name` attribute can be used in conjunction with the `Display Parameter Name` attribute to label live.dial and live.slider object when you use them in a device.
- The `Long Name` attribute is be used to identify a parameter to the Live application's Parameter automation and MIDI mapping. You can use a single name for all three of these attributes by checking the `Link to Scripting Name` attribute in the object's Inspector. You may find this to be a simple approach to managing paramter naming.

Setting the display name for a parameter

- Add a Max for Live UI object to your device. When the UI object first appears, it displays the name of the object itself.



- Select a the object and click the Inspector icon on the Patcher toolbar to show the object's Inspector.
- Double-click in the Value column for the Short Name setting to show a cursor and text box. Type in a name for the parameter, followed by a carriage return. The Value column will be

de-selected, and your parameter name will appear in the object's display.



Note: If your short name is too long, it will be automatically truncated. If you are using a live.slider or live.gain~ object, you can use resize the object manually by clicking in the lower right-hand corner of the object and dragging. If you are using a live.dial object, you should enter a new horizontal value for the `Patching Rectangle` or `Presentation Rectangle` attributes using the object's Inspector.

Setting a custom unit style

- Select a Max for Live UI object and click the Inspector icon on the Patcher toolbar to show the object's Inspector.
- Choose **Custom** from the Unit Style pulldown menu.
- Double-click in the Value column of the Custom Units attribute to show a cursor and text box. Type in a string to be used for the custom unit style, followed by a carriage return. The Value column will be de-selected, and the name will be set.

  You can type in custom unit strings as symbols (e.g. "Harmonic(s)"), in which case the parameter's value will be displayed in its 'Native' display mode, followed by the symbol (e.g. "12 Harmonic(s)" for an Int-typed parameter or "12.54 Harmonic(s)" for a Float-typed parameter). If you would like to have additional control over the numerical component displayed, you can enter a sprintf-style string (e.g. "%0.2f Bogon(s)", which would display a value such as ".87 Bogons").

Controlling a parameter's visibility

- You can change the visibility of a parameter by changing the `Parameter Visibility` setting in the Inspector. If this attribute is set to Automated and Stored, the parameter will be stored in the Live Set and presets, and will be available for automation. If this attribute is set to Stored Only, the value will be stored, but it will not be visible to Live's automation

system. If this attribute is set to Hidden, it will neither be stored nor available for automation.

- You may want to have a parameter Hidden when it affects other Max for Live parameters. This will prevent problems with overloading Live's undo buffer, and will also limit issues with preset storage.

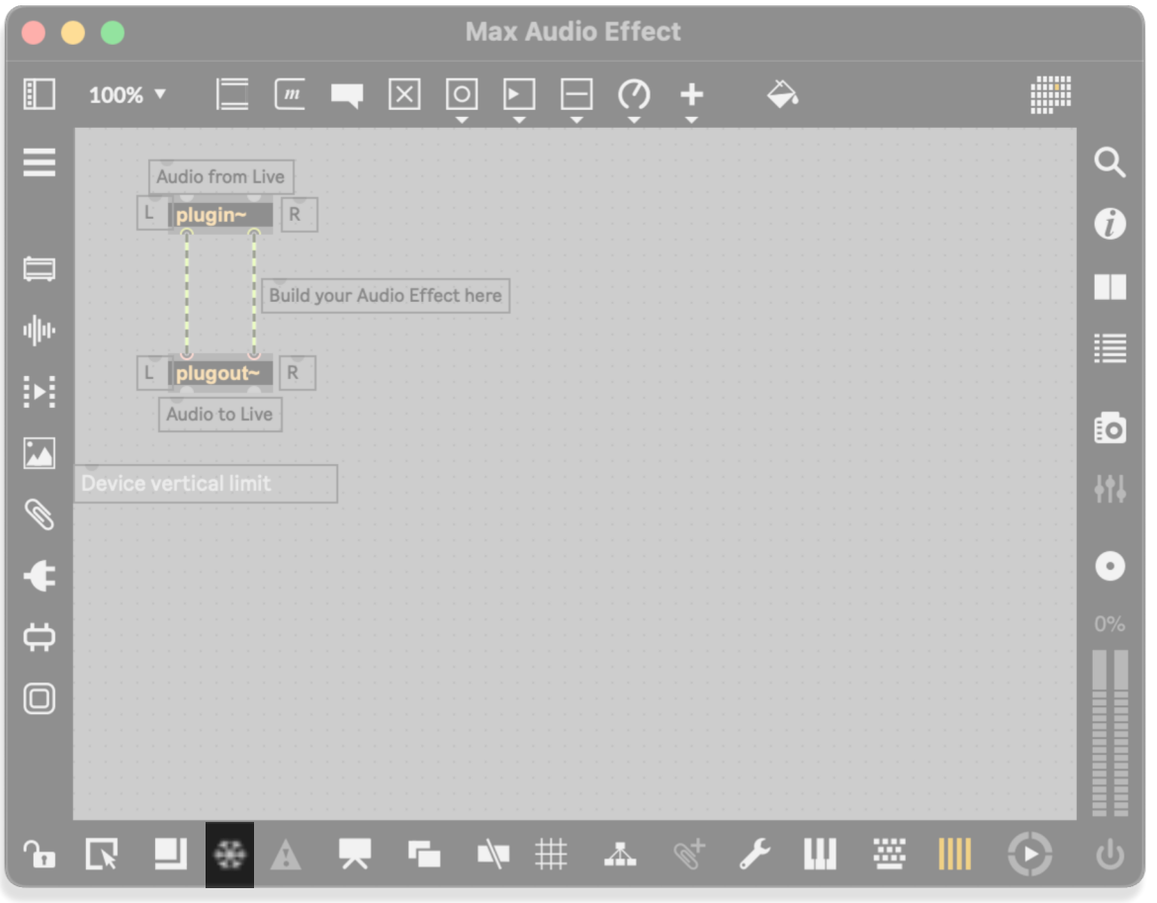
Freezing Max for Live Devices

Freezing a Device 878

Freezing a Max device prepares it for distribution. A frozen device contains files it needs to operate. These files might include subpatchers, audio files, image files, Javascript code, or third-party Max external objects. When you freeze a device that contains third-party external objects, you can include both Windows and Macintosh versions of the external in your Max search path, if you have them both. The frozen device will then will contain both versions, and will work on both platforms. Max analyzes your device to find any files it uses (called dependencies), and then combines these files with your device. When a frozen device is opened, the files inside the device are used, even if similarly named files reside on your disk in the Max search path.

Freezing a Device

- Click the Freeze button in the device window toolbar.



- Choose **Save** from the File menu to save the device. The device is now frozen and is reloaded in Live in its frozen state.

Note that the device is not frozen immediately when you click the freeze button. It is frozen when you save it. Before saving, you can make further changes to the device after clicking the Freeze button by clicking the Freeze button again to unfreeze. When your changes are finished, click the Freeze button again, then choose **Save** from the File menu.

Max for Live Limitations

Audio Limitations 881
MC Limitations 881
MIDI Driver Limitations 881
pattr and Max for Live Parameters 882
Other Limitations 882

Max and Max for Live are two separate products. The version of Max that comes bundled with Live is optimized for use with Live, but it is not the full Max 9 application, and is not intended to be used outside the context of Live. If you only have a Max for Live license, and not a full Max license, then the Max application will have certain limitations.

Limitation	Description
Save patches using Gen, MC, and Jitter Geometry	You will still be able to load and run Max for Live devices using these advanced Max features, but you will not be able to author your own Max patches using these faetures.
Gen code export	You will not be able to export C code from any of the Gen objects, like gen~ and jit.gl.pix.
Integrated OSC Support	Objects like udpreceive and udpsend will still work, but integrated OSC support will not be functional.
Direct Audio Driver Access (Multichannel Audio)	Audio input/output will be limited to the particular Max for Live device. See Audio Limitations for more details.
Unlimited MIDI Hardware Messages (sysex, program changes, etc.)	MIDI input and output will be limited to those messages that Live will recognize and pass between devices.
Direct MIDI Driver Access	MIDI input and output will be limited to those MIDI messages sent directly between Live devices. You will not have direct access to your system's MIDI Drivers. See MIDI Driver Limitations for more details.

Standalone Operation (Run Max without Live)	You will only be able to use Max inside of Live, and will not be able to run Max on its own.

Audio Limitations

When authorized only via Live, the Max application will not use its own audio drivers. Its audio input is the input to the Max for Live device you are editing, and its audio output is the output from that Max for Live device. Audio I/O works when using preview mode. If you turn preview mode off, all audio I/O for the Max application will stop.

If Max and MSP are authorized when editing a Live device, Max Consoles that are not a part of the Live device will use the regular Max audio drivers.

The use of the send~ and receive~ objects to pass audio between Max for Live devices is not supported.

MC Limitations

User initiated connections for MC patches are prohibited with only a Max for Live authorization. Runtime behavior and scripted connections will work, but in order to add new connections, a standalone Max authorization is required.

MIDI Driver Limitations

When authorized only via Live, the Max application will not use its own MIDI drivers. MIDI input arrives from Live and MIDI output is sent to Live. MIDI I/O works only when using preview mode. If you turn preview mode off, all MIDI I/O for the Max application will stop.

When you open a Max patcher file such as a help file containing Max MIDI objects, the MIDI output will be sent to the MIDI output of the device you are currently editing. If you open a file containing MIDI objects when you are not editing a device, there will be no MIDI I/O.

If Max is authorized when editing a Live device, Max Consoles that are not a part of the Live device will use the regular Max MIDI drivers for MIDI objects.

pattr and Max for Live Parameters

Although the pattr objectscan be used in the context of Max for Live, there are some differences compared to normal Max use.

The autopattr object cannot be used to batch-register objects with the Parameter system. You need to use individual pattr objects for this purpose.

The pattr object functions mostly identically under Max and Max for Live. However, some users might expect the value of a pattr object in a Max for Live device to be automatically maintained by the Live Set upon save and close, and to be correctly restored when the Set is re-opened. This is not the case. This behavior is available, but only if the pattr object's Parameter Mode Enable attribute is enabled in the object's Inspector and the Parameter Visibility attribute is set to '`Automated and Stored` *or* `Stored Only'`.

The pattrstorage object also functions mostly identically under Max for Live, but there are a few important distinctions to keep in mind, *if the object is in Parameter Mode*. First, the value of the pattrstorage object in Parameter Mode is its entire storage state (what is ordinarily saved to an external file), rather than the currently recalled slot. This means that devices using pattrstorage in Parameter Mode need not require an external file to recall the storage state of the object (it can be saved in presets, set as an initial value or stored in the Set). Use of an external file can be disabled using by setting the object's savemode attribute to 0. If the pattrstorage object has an Initial Value, the savemode and autorestore attributes are ignored and file-less use of the object is assumed. Finally, the pattrstorage object has an additional attribute when in Parameter Mode: Auto-update Parameter Initial Value. When this is enabled and Initial Enable is turned on, all changes to the object's storage state will cause the Initial Value to auto-update to the new state.

Other Limitations

The grab object cannot be used to communicate from a send to a receive between devices.

When authorized only via Live, Max cannot build standalones or collectives. Frozen devices, which Max for Live creates, are very similar to collectives.

Max for Live MIDI Tools

Types of MIDI Tools 884
Creating a New Max for Live MIDI Tool 885
Anatomy of a MIDI Tool 885
Creating A Transformation 886
Apply Cycle 895
Creating a Generator 895
Next Steps 900
Common Errors and How To Handle Them 900
Limitations 901

> Please note: Max for Live MIDI Tools can only be used in version 12.0 or greater of Live.

Max for Live MIDI Tools are a distinct type of Max for Live device designed to manipulate and generate MIDI data in a Live MIDI Clip. These tools offer versatility by allowing you to craft patches capable of generating MIDI data from the ground up or of reshaping existing MIDI data. For example, you could create a MIDI Tool that produces rhythms according to a specific generative algorithm, or use a Transformation to fine tune the properties of notes in a clip. Given this, MIDI Tools unlocks a myriad of creative possibilities and streamline the execution of repetitive algorithmic tasks on MIDI data within the Live environment.

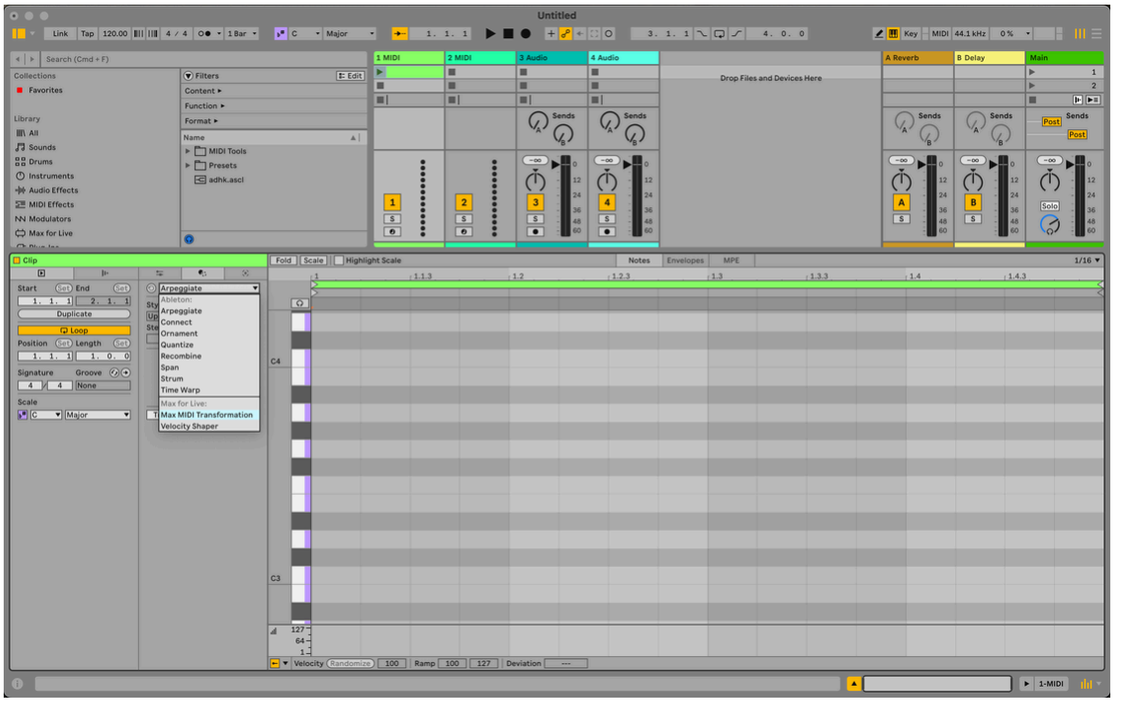
Types of MIDI Tools

Within the domain of Max for Live MIDI Tools, there are two distinct types of devices in the `.amxd` format: a "Generator" and a "Transformation". A Generator can be used to create new MIDI data from scratch while a Transformation can be used to transform existing MIDI data in a Clip. As you will come to understand later, the differences in patching between these two types of devices are slim and sometimes non-existent. However, there are subtle divergences in how data passes between Live and Max and the expectations with respect to user experience. At the end of the day, the takeaway message is that generation and transformation are two distinct MIDI Tool device

types that you need to be aware of. If your aim is to add new notes to a clip, you'll want to create a Generator. If you want to edit the properties of existing notes, you'll want to create a Transformation.

Creating a New Max for Live MIDI Tool

To create a new Max for Live MIDI Tool, you can load either the "MIDI Generator Template" or "MIDI Transformation Template" device that are both available in their respective dropdown menus of the Tool Tabs. Once loaded, you can save, which will prompt you to create a copy of it under a new name. Now, you should see a new MIDI Tool available in the Tool Tabs dropdown menu. Clicking its name will load the device.



Anatomy of a MIDI Tool

The basic anatomy of a MIDI Tool can be structured into three parts: an input, an output and some data manipulation between these two points.

live.miditool.in

In the same way that a MIDI Effect uses notein or midiin to receive MIDI data, or an Audio Effect uses plugin~ to receive audio input, a Max for Live MIDI Tool uses an object called live.miditool.in. This object only works inside of Max for Live MIDI Tools. This object, live.miditool.in, is a kind of "portal" between the currently focused Clip and a loaded Max for Live MIDI Tool. It outputs two valuable bits of information from its outlets: the note information and some contextual information about the Clip itself such as but not limited to the grid interval, selection time and selected scale.

live.miditool.out

Just as live.miditool.in is the only way to receive MIDI data from a Clip, live.miditool.out is the only way to send MIDI data back to a Clip in order to transform some existing notes, or generate new ones. Take note that live.miditool.out also only works inside of a Max for Live MIDI Tool.

MIDI Tool Logic

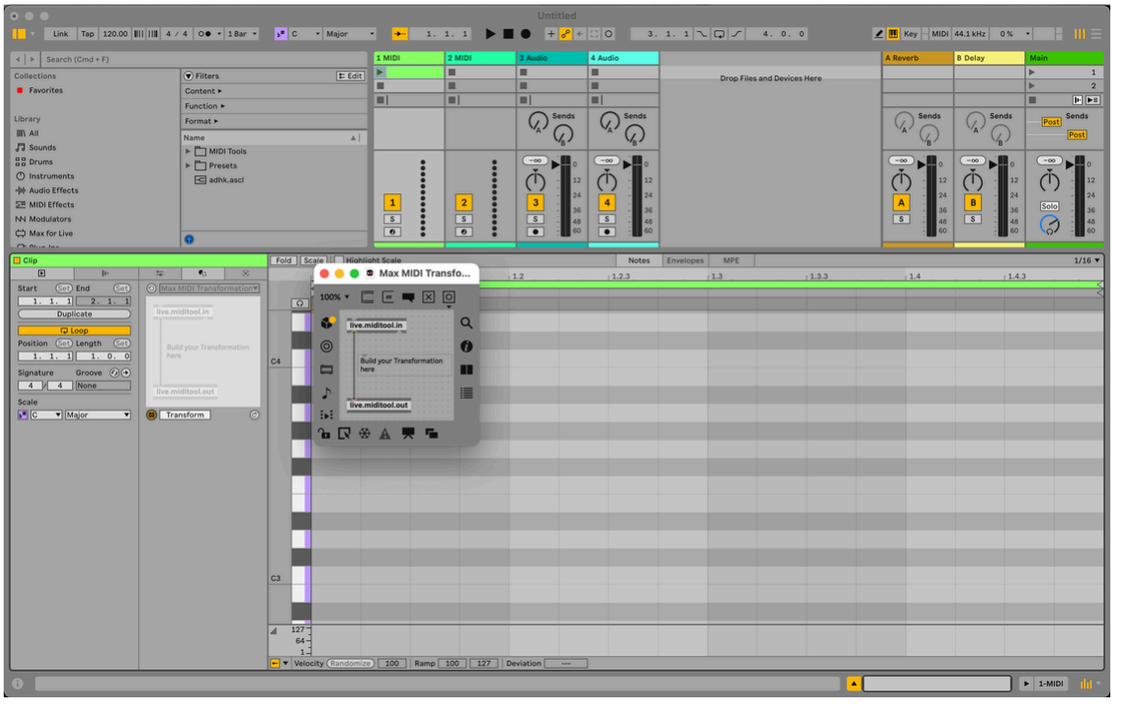
Between the live.miditool.in and live.miditool.out objects is where the logic of your Max for Live MIDI Tool is defined. In this part of the patch you will receive note and context data from the respective outlets of live.miditool.in as a dict. Using this information, you can then manipulate the note data and send it back to the live.miditool.out object which will update the corresponding Clip in Live. You can also ignore the data that comes out of live.miditool.in and generate your own data from scratch, for instance in a Generator.

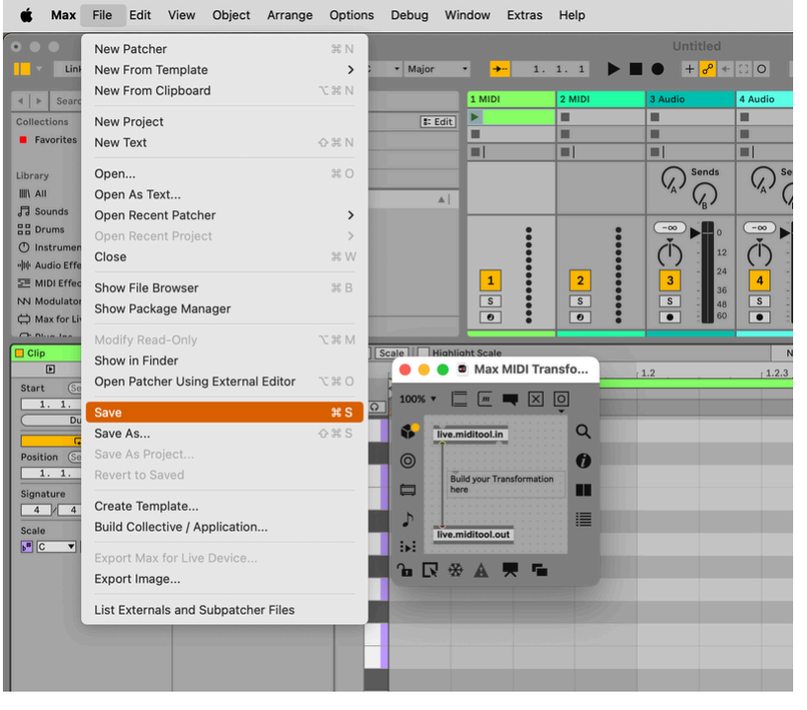
Creating A Transformation

Now that you understand what a MIDI Tool is *for* as well as the general anatomy, let's create a simple Max for Live MIDI Transformation that will transpose all the notes in a Clip by a user-defined amount.

Step 1: Create A New MIDI Tool Transformation

As described above, let's create a new Max for Live MIDI Tool by duplicating the existing "MIDI Transformation Template". Give it a unique name, and then open it in the Max editor by clicking the edit button which is located to the left below the MIDI Tool's interface.





Step 2: Receive Data from the Clip

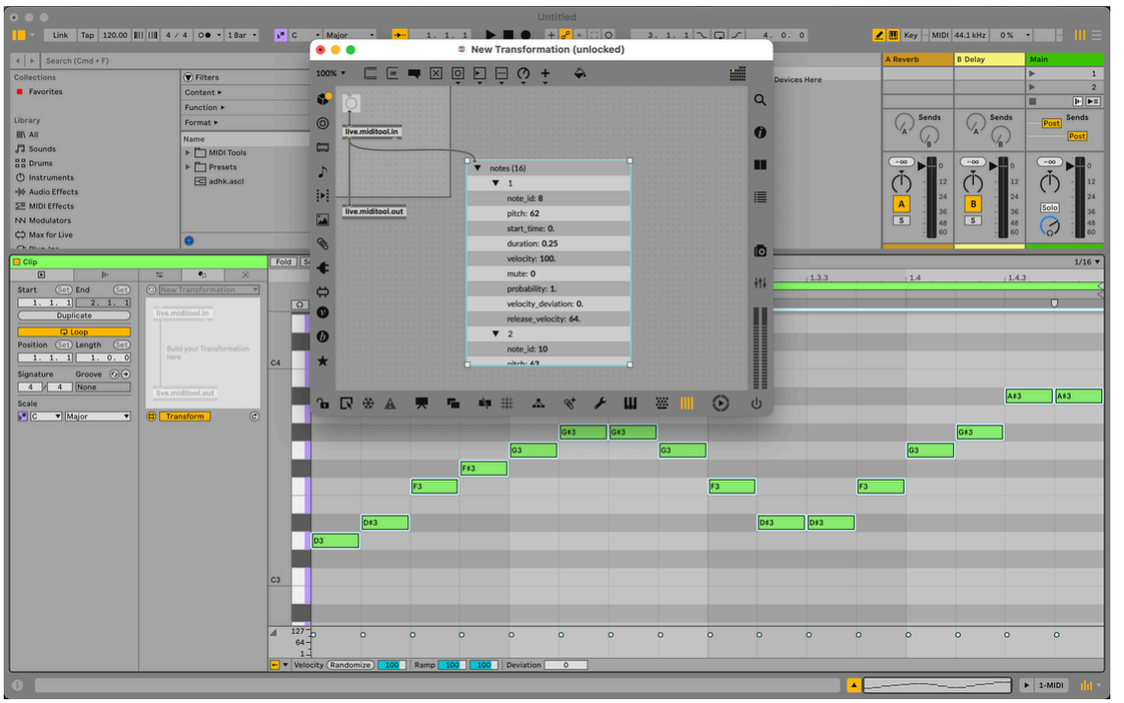
So now that we have a template MIDI Tool ready to go, let's dive into actually patching, and trying to understand the data output by the live.miditool.in object. Remember, it outputs the *note*

information and some *contextual information* about the Clip itself from the left and right outlets respectively.

> If you aren't too familiar with dictionaries or the dict object, visiting the help files is a good place to start.

It outputs both of these bits of information whenever you send a `bang` message to the inlet of live.miditool.in or when you press the "Apply" button underneath the MIDI Tool in Live's user interfaces. It is important to note now that there are differences between sending a `bang` message and pressing the "Apply" button, but we will discuss that later.

For now, we are mostly going to ignore the right outlet and focus instead on the left outlet which outputs note data. To understand the data better and how it arrives in Max, we can attach a dict.view object to the left outlet of live.miditool.in and then send a `bang` to the live.miditool.in object. This will cause the object to output the note data as a dictionary of the Clip that is currently in focus.



> Note that live.miditool.in won't output anything when it is loaded in a Transformation AMXD unless there are MIDI notes in the Clip.

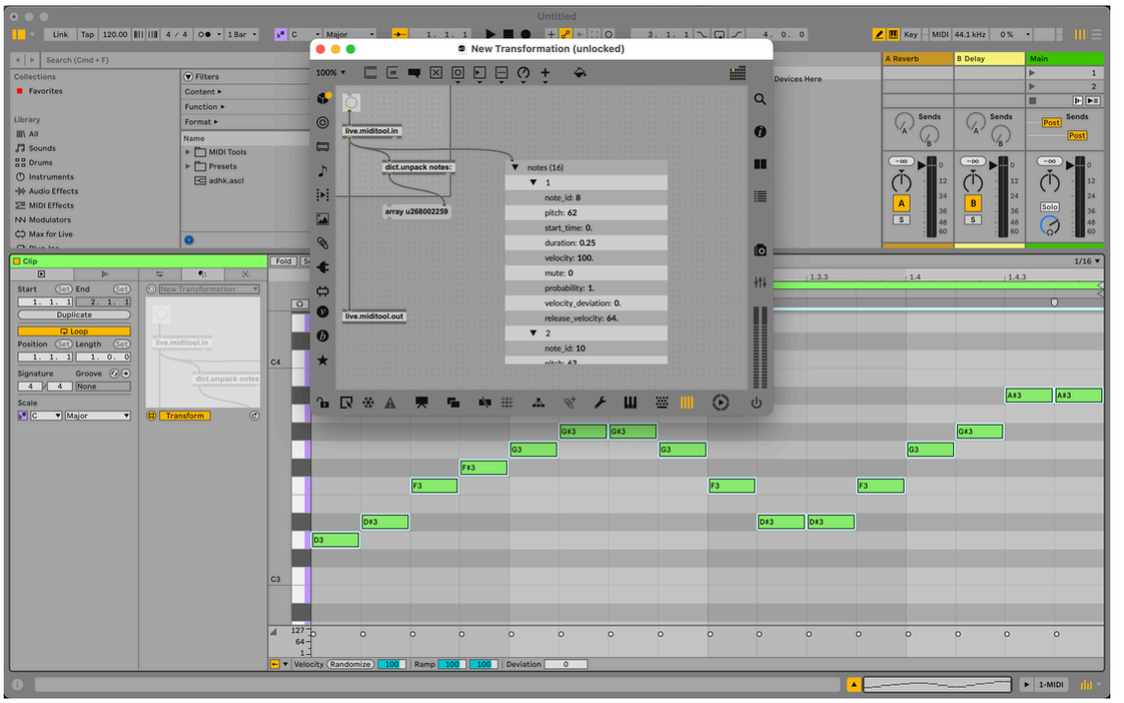
To summarise, the note dict that is output from live.miditool.in contains a `"notes"` key which is associated to an array of sub-dictionaries which each represent a single note. Below is a textual representation of this data structure using a Clip that has just two notes in it:

```
{
  "notes" : [ // the brackets denote the start of an array
    // this here is the first note
    {
      "note_id" : 9,
      "pitch" : 60,
      "start_time" : 0.0,
      "duration" : 0.25,
      "velocity" : 127,
      "mute" : 0,
      "probability" : 1,
      "velocity_deviation" : 0,
      "release_velocity" : 0
    }
    // this here is the second note
,               {
      "note_id" : 10,
      "pitch" : 61,
      "start_time" : 0.25,
      "duration" : 0.25,
      "velocity" : 127,
      "mute" : 0,
      "probability" : 1,
      "velocity_deviation" : 0,
      "release_velocity" : 0
    }
  ],
  }
```

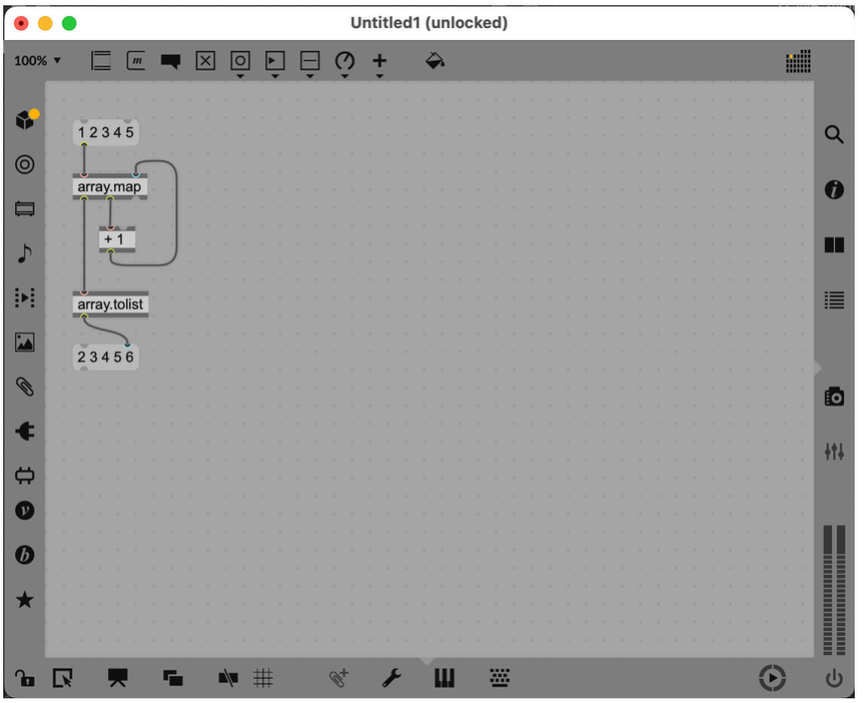
If we want to *transform* this Clip represented by a dictionary, we have to iterate through each sub-dictionary representing a note and modify the properties we are interested in mutating. For example, if we want to pitch shift all the notes up by 1 semitone, we must add 1 to the `"pitch"` property of each sub-dictionary.

Step 3: Transforming the Data

To coordinate the Transformation, we need to first extract the array of sub-dictionaries that represent each note in the Clip. By using dict.unpack and making the first argument `notes:`, we can extract this array. Then, we can then iterate over the sub-dictionaries and transform the pitch property of each one to realise the total pitch shift effect. Importantly, all the other properties need to be left intact, so as to not change, for example, the duration or start time.



Once we have just the array of sub-dictionaries, we are going to use the array.map object. A straightforward mapping example would be to take an array of numbers and add 1 to each element. In Max, this would look like the following:

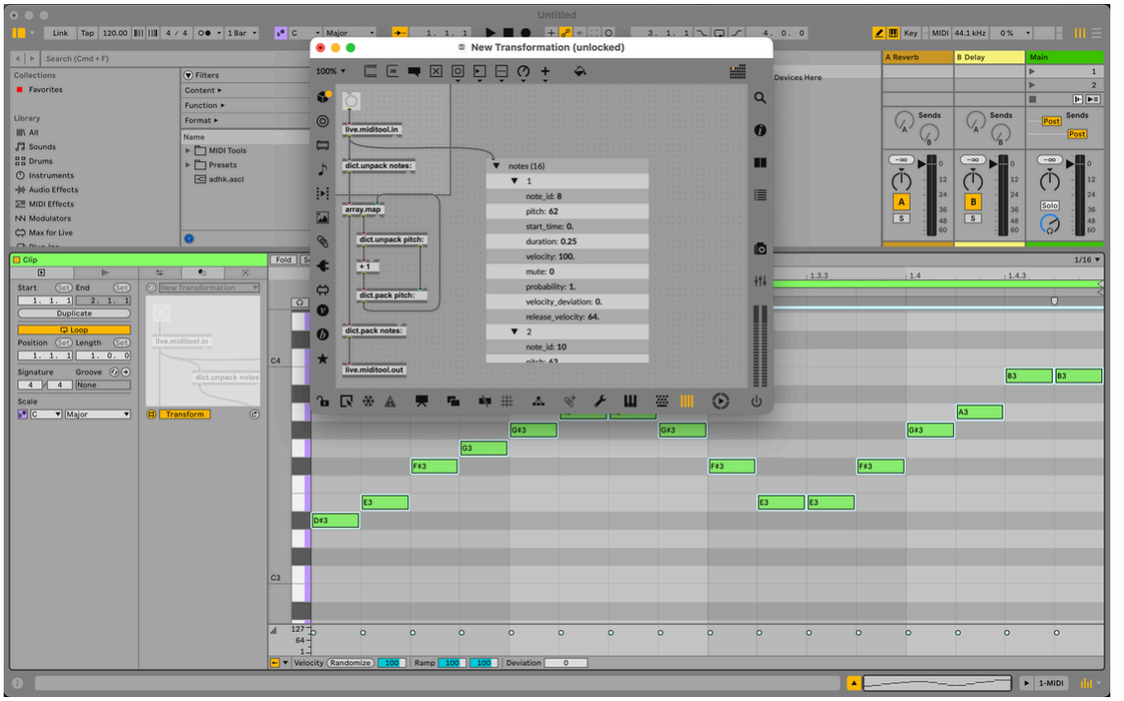


When you supply an array to array.map it iterates over each element, outputting it from the second outlet. Each time this happens, array.map will wait for some input from the second inlet. Whatever the object receives when it is waiting will *overwrite* the original value in the array. Once all the elements have been processed, the object will output the final result from the first outlet. In the case of this pitch shift example, we want to take each sub-dictionary from the array, and add a positive or negative offset to the "pitch" value. At each iteration we have to unpack the pitch value from the sub-dictionary, add the offset, and then pack it back together in order to preserve all the data we did not change (start_time, velocity, etc.). This can be done with the dict.unpack and dict.pack objects. To keep things simpler for now, we're just going to make the offset a fixed value of 1 in MIDI note numbers.

> The keys of a dictionary that aren't explicitly unpacked with dict.unpack are output by the right-most outlet. The right-most inlet of dict.pack can be connected to this outlet so that any unaltered keys are reassembled with the modified properties. This is useful for when you want to only transform a few properties of a dictionary and leave the rest untouched.

The only thing left to do is take the array of sub-dictionaries that we just transformed and wrap it back up in a dictionary with a `"notes"` key. This ensures that the array containing sub-dictionaries

is formed back into the same structure that we first saw from the output of live.miditool.in. To do this we use dict.pack and make the first argument `notes:`. Once you're at this stage your patch should look like this:



You can now transform notes in a Live Clip by pressing the apply button, or sending live.miditool.in a `bang`. You should see that the notes are shifted by 1 semitone upwards.

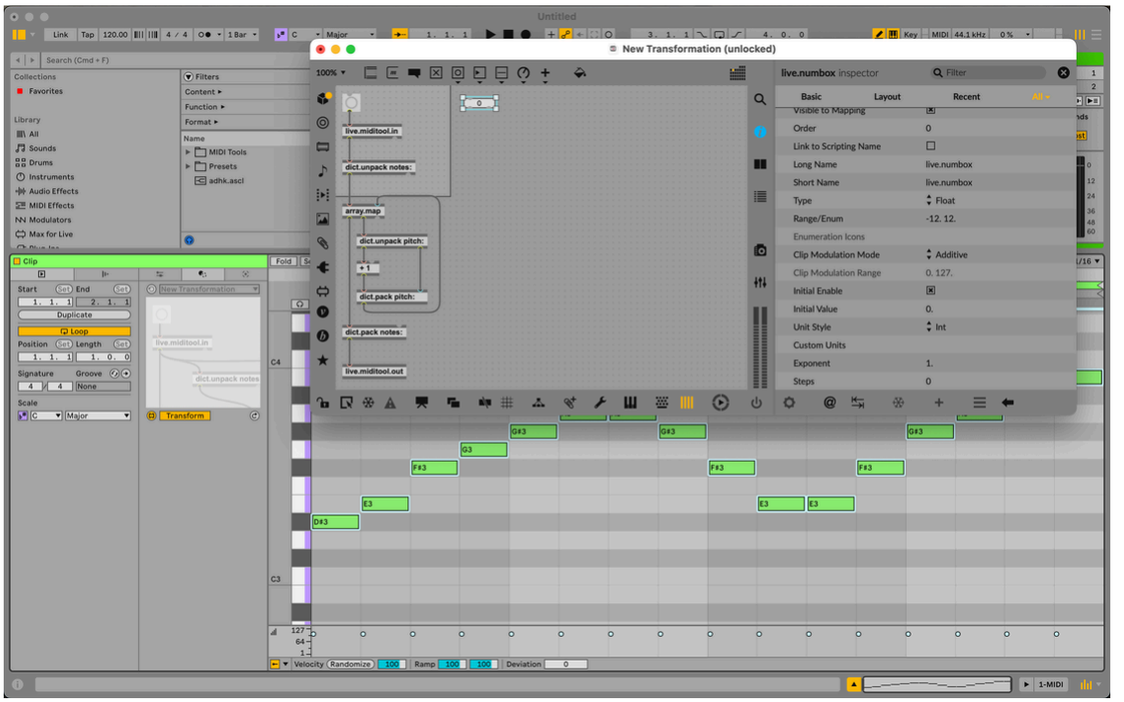
Summary

To quickly summarise, the patching workflow for a Transformation follows these steps each time:1) 1. Extract the array of note sub-dictionaries from the dict output from live.miditool.in

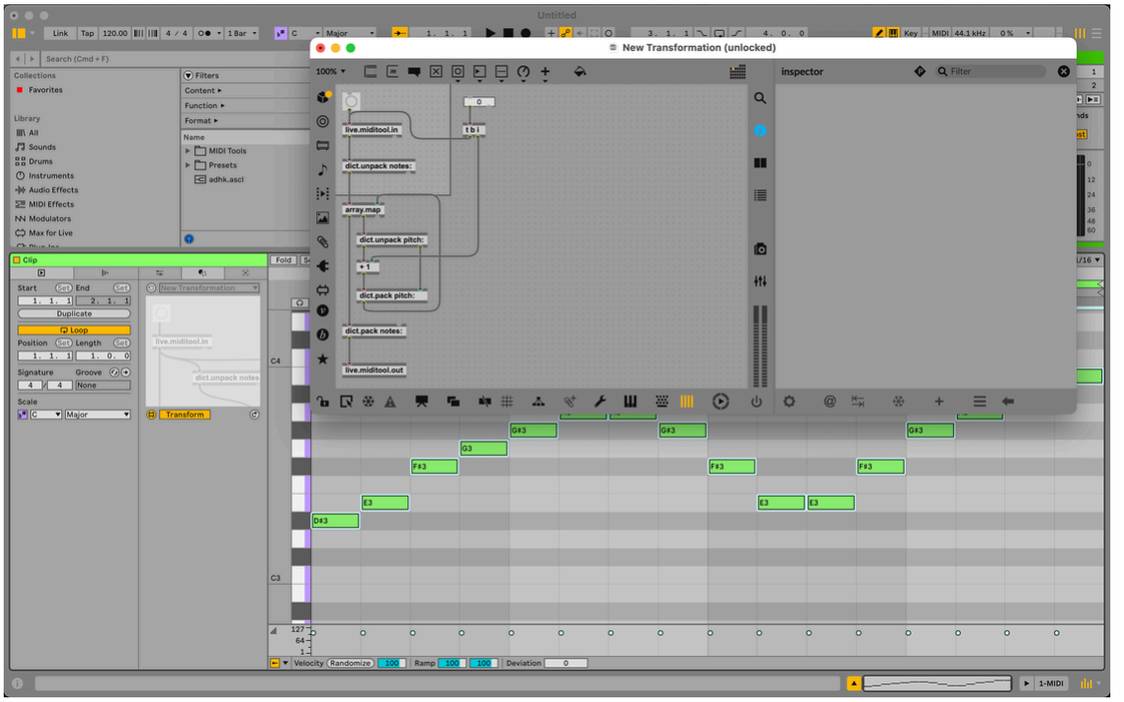
1. Extract the array of note sub-dictionaries from the dict output from live.miditool.in
2. Process the array of note sub-dictionaries
3. Pack the array of note sub-dictionaries back into a dict with a "notes" key
4. Send the dict to live.miditool.out

Step 4: Adding a User Interface

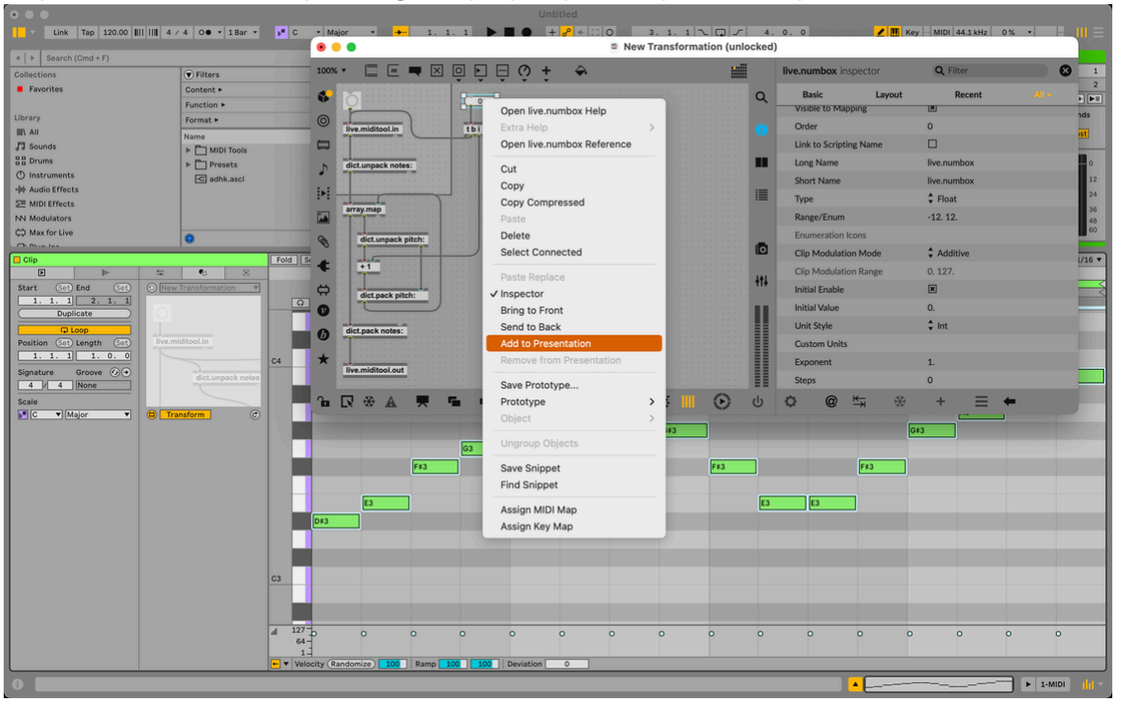
You will probably want to add some controls to the Transformation, in this case so that you can alter the amount of transposition. To do this, add a live.numbox to the patch. We can use the output of this live.numbox to set the amount of shift as well as trigger the Transformation. First though, let's set the range of the parameter to something useful, like -12 to 12, giving us an octave of shift in both a positive and negative direction.



Now that we've done that, we need to connect the output of the live.numbox to the + object. However, we still need to trigger the Transformation every time we modify this parameter value. To do this we also must send a `bang` to the live.miditool.in object. We should coordinate this with a trigger object to guarantee those things happen in the correct order: first set the offset and then trigger with a `bang`.



We can also tidy this up and put the parameter into presentation mode so that it is cleaner and simpler to use the Transformation while we are not editing it. We should also set the patcher to open in presentation mode, by editing this property in the patcher inspector.



Apply Cycle

Typically, part of using a Transformation or Generator involves a process where you adjust parameters to fine-tune the result you want. When you do this, and change the parameters of a MIDI Tool in Live, you might notice that the result of a transformation can always be reverted, and you are able to get back to the state of the Clip from before you started experimenting. Take our simple pitch shift MIDI Tool we just built as an example. If you had a Clip with a single MIDI note at C3, and you then set the live.dial to 4, you would see that the MIDI Note now appears as E3. If you then turn the dial back to 0, you would see the note return to C3. The reason that this behavior can happen is because of the apply cycle. The apply cycle is responsible for managing "snapshots" of the Clip that are sent as dictionaries to Max for processing. When you send a `bang` to live.miditool.in or press the apply button in Live's interface, an apply cycle starts. An apply cycle ends when you perform any interactions outside of the MIDI Tool. While you are inside an apply cycle, the dictionary output by live.miditool.in*will always contain the same note data*. So, when the logic of your patch adds 1 to the pitch of each note, it is always applying this change to the dictionary it received at the *start* of the apply cycle. This is to prevent a scenario in which the transformation result is then immediately reflected in the next output of live.miditool.in. If that were the case, a "feedback loop" would be created where you would not be able to return to the original state of the clip.

Creating a Generator

Creating a Generator is similar to creating a Transformation in terms of patching. The main difference is that you will be generating the note dictionary data from scratch rather than transforming existing note data output by the live.miditool.in object. In the following example, we will create a Generator that creates a randomised rhythm with 16th notes.

Step 1: Create A New MIDI Tool Generator

Create a new Max for Live MIDI Tool by opening the existing "MIDI Generator Template" and saving it. Give it a unique name, and then open it in the Max editor by clicking the edit button in the bottom left.

Step 2: Generating Data

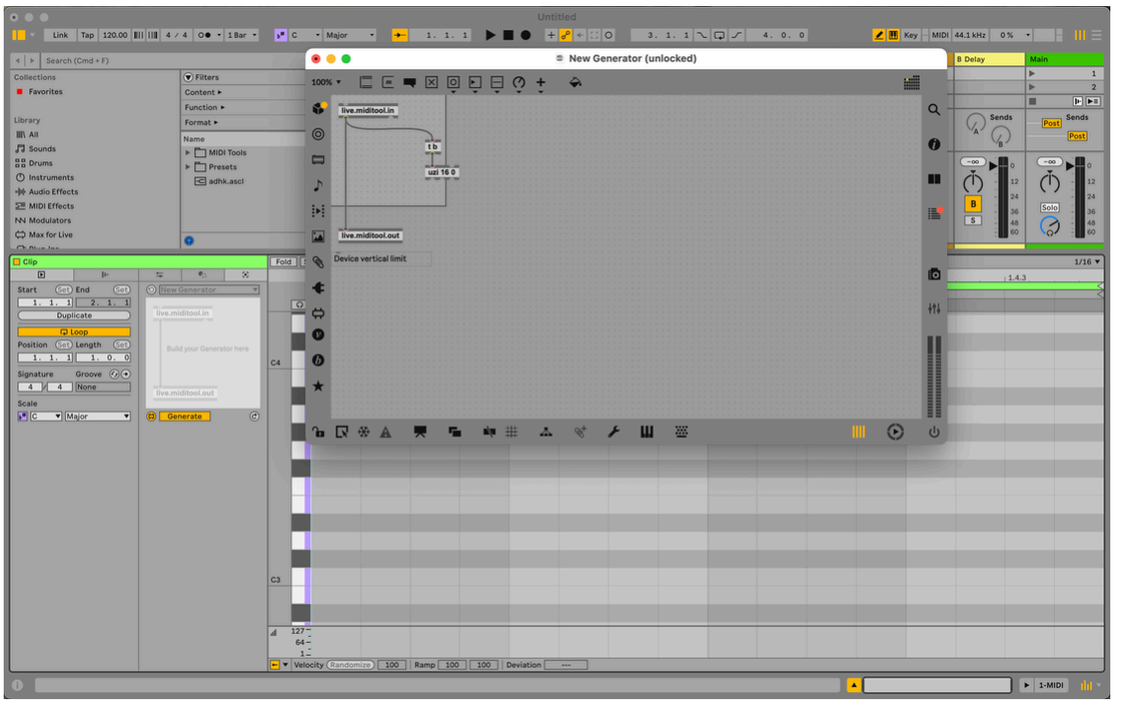
We've already had a glimpse of the shape of the data that represents a Clip and is output as a dict from live.miditool.in when making a Transformation. For a Generator, we can generate whatever

note data we would like to, as long as we pass it to Live via the live.miditool.out object and in the correct format.

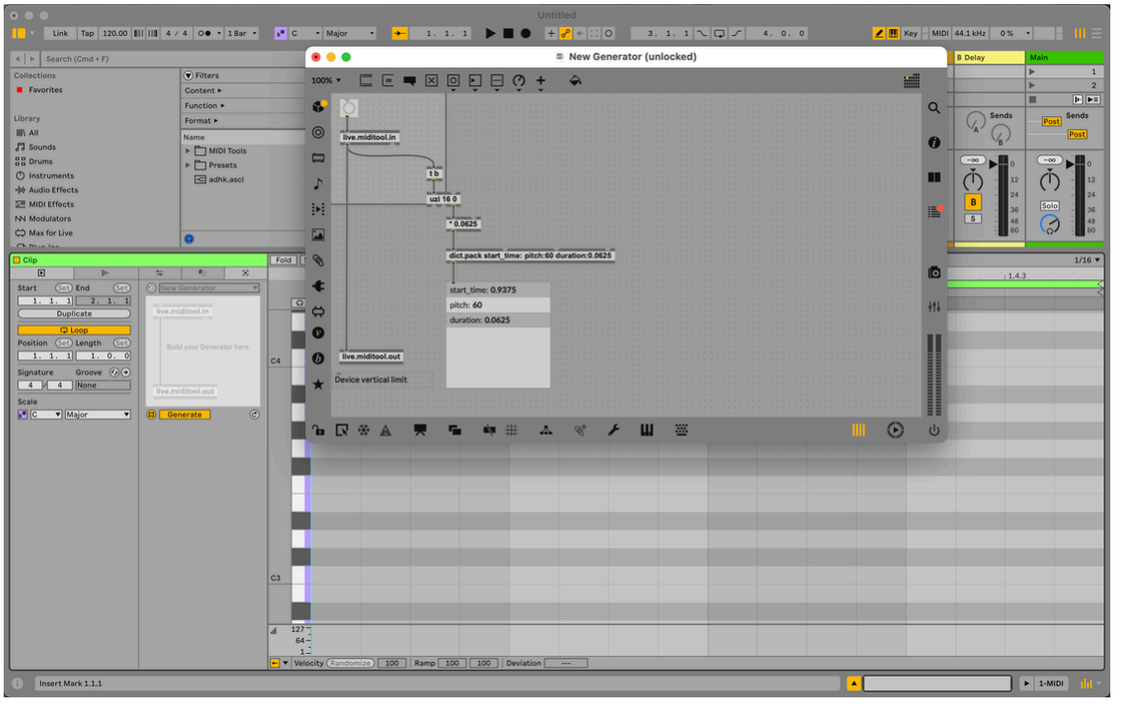
This code example shows all the properties that you can supply to create notes from scratch in a Generator. Notice that you don't have to supply the `"node_id"` property that can be seen when notes are passed from the Clip to the MIDI Tool. Only `"pitch"`, `"start_time"` and `"duration"` are required. The rest are optional.

```
{
  "notes" : [
    {
      "pitch" : 60,
      "start_time" : 0.0,
      "duration" : 0.25,
      "velocity" : 127,
      "mute" : 0,
      "probability" : 1,
      "velocity_deviation" : 0,
      "release_velocity" : 0
    }
,               {
      "pitch" : 61,
      "start_time" : 0.25,
      "duration" : 0.25,
      "velocity" : 127,
      "mute" : 0,
      "probability" : 1,
      "velocity_deviation" : 0,
      "release_velocity" : 0
    }
  ],
  }
```

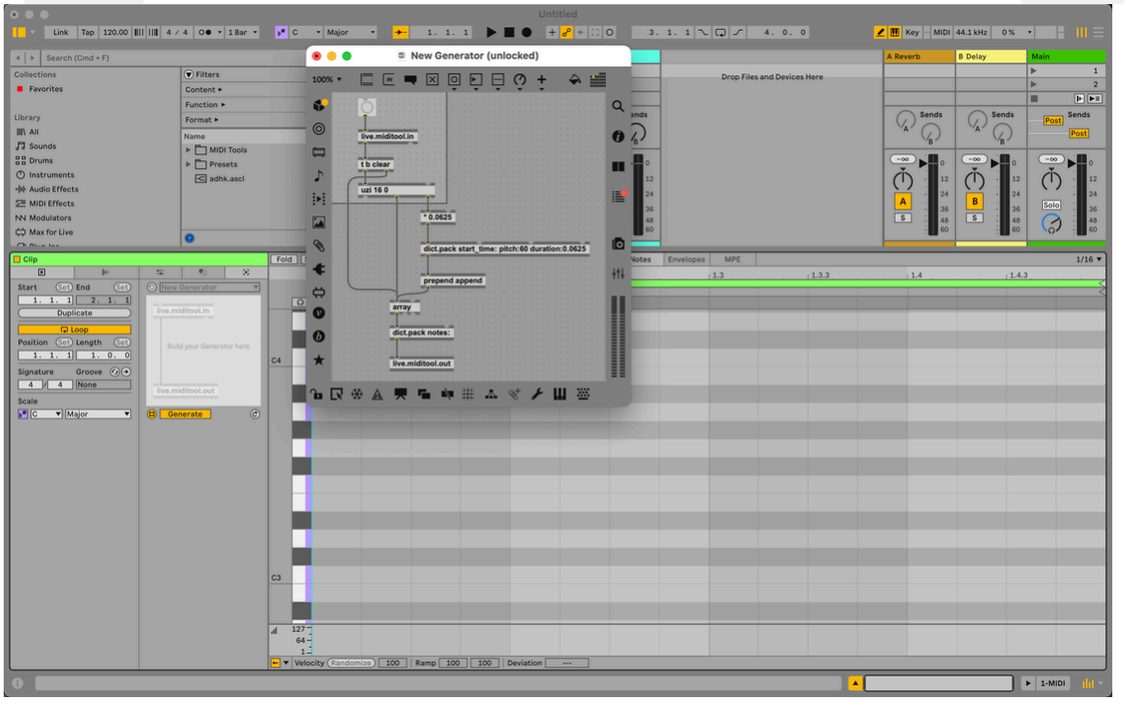
Because we are not transforming notes, we are going to take the output of live.miditool.in and use it to create a `bang`. This `bang` will initiate the logic of the patch that is responsible for generating the dict that represents the Clip. From here we can implement the generation logic, starting with uzi. In this specific case, we will always generate 16 notes, so we'll set the first argument to 16, and the second argument to 0, so that it counts up from 0 for the right outlet.



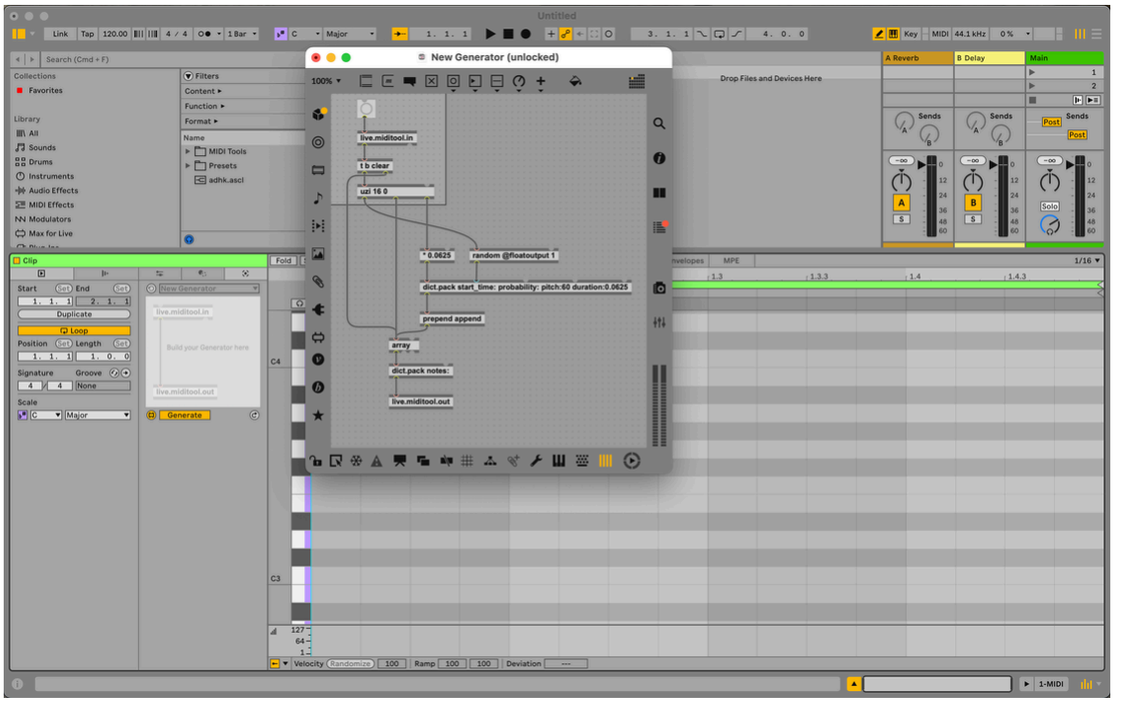
Now let's cobble together an array of note dictionaries using the uzi output. First we'll create a dict.pack which will be crucial in forming the keys and values that represent each note. We only need to generate three bits of information for each note: `"pitch"`, `"start_time"` and `"duration"`. We can set the `"pitch"` to a fixed value of 60, the `"start_time"` to the output of the uzi and the `"duration"` to a fixed value of 0.0625. The duration is always defined in beat time, and so a 16th note is represented by this floating-point value.



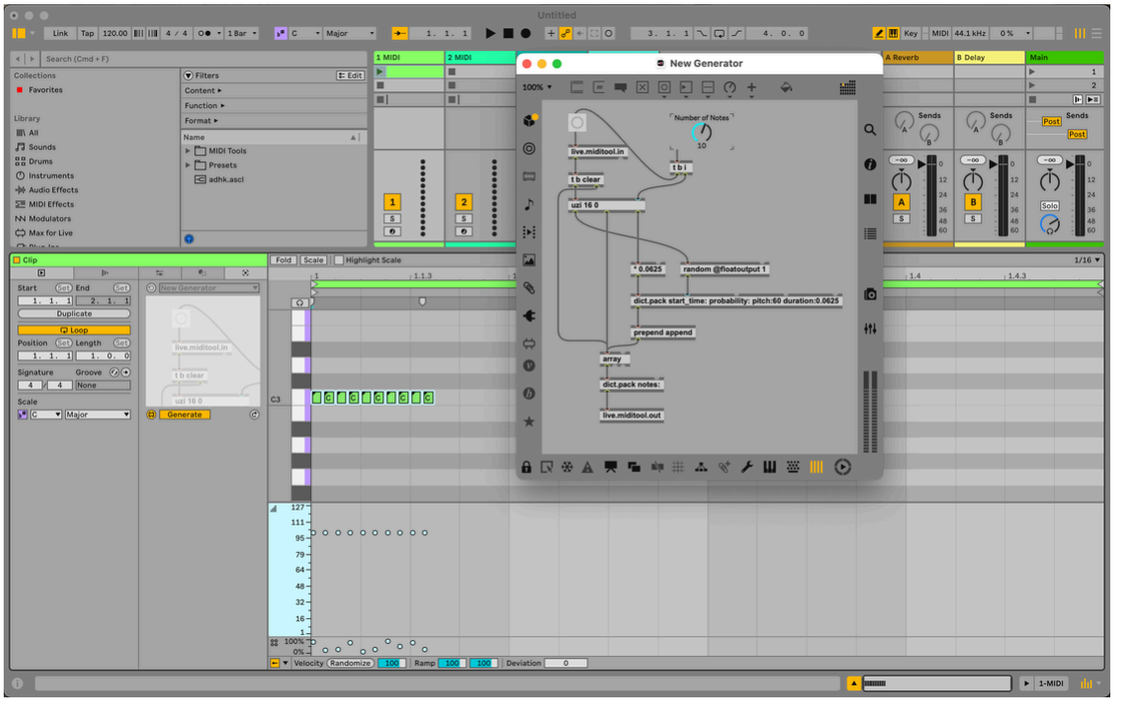
As each of these note dictionaries are generated, they need to be appended to an array containing *all* the notes. To do this we take the output of dict.pack and send it to an array prefixed by the `append` symbol. Once the uzi has finished outputting we will use its middle outlet to output the array of note dictionaries. Before sending that off to live.miditool.out we need to wrap it in a dict with a `"notes"` key. This can be done with dict.pack and by making the first argument `notes:`.



You can now generate 16th notes by pressing the bang or apply button in Live. To vary the rhythm that is generated each time, we can add a small randomisation to the chance that a note is generated at all. We can do this by adding a random object to the patch and connecting it to the `probability` key of each note dictionary. We will have to add the `probability:` argument to dict.pack to do this.



Lastly, we can also add a dial to control the number of notes generated. We can do this by adding a live.dial to the patch and connecting it to the uzi and live.miditool.in objects. As we did with the Transformation example, we can also add the control to the presentation view and set the patcher to "Open in presentation".



Next Steps

Now that you have the basic building blocks for creating both a Generator and Transformation MIDI Tool it's up to you to create more interesting, varied and explorative devices for yourself. Something that you can explore next is using the *context outlet* of live.miditool.in to create MIDI Tools that are more contextually aware of the Clip they are interacting with. You could also implement more sophisticated algorithms for melodic and rhythmic generation, as well as musical and complex transformations. Of course, you can also open the Euclidean and Velocity Shaper MIDI Tools in the editor and look at those patches for inspiration as well.

Common Errors and How To Handle Them

My Generator or Transformation doesn't do anything

In this scenario, it's always good to start by checking that the live.miditool.in object is outputting the data you expect. You can do this by attaching a dict.view object to the left outlet of live.miditool.in and sending a `bang` to the object. If you see the data you expect, then the problem is likely in the logic of your patch. If you see the correct data being emitted by live.miditool.in then check that the transformed or generated data is arriving at the live.miditool.out object in the correct format. Lastly, for a Transformation, ensure that there are notes in the MIDI Clip.

Remember, Transformations won't do anything if there are no notes available in the Clip to transform.

Synchronisation

In some cases, you may see an error message in the status bar of Live like this:



This is because the patch has initiated a sort of *transaction* between Live and the MIDI Tool, but the transaction has not been completed. This can happen for a number of reasons, but most likely happens if you send a `bang` to live.miditool.in and do not also pass a dictionary back to Live via live.miditool.out.

Limitations

MSP objects and audio processing

Objects that process audio do not work in MIDI Tools currently. MIDI Tools are not attached to the audio processing engine and therefore cannot process audio.

node.script

Max for Live MIDI Tools do not support the node.script object in a straightforward way because this object runs in its own process and can return values at an indeterminate time. Because MIDI Tools depend on a synchronous interaction with Live, the node.script object is not suitable for use in a MIDI Tool, unless the order of operations in a MIDI Tool is not contingent on the output of the node.script object.

Transport

The transport object cannot be used to control the transport. It currently reports values, however, these values are not necessarily correct and can be misleading.

Presets

Storing a Preset 902
Presets for a Devices in the Library 902
Presets for a Devices Outside the Library 903
Saving a Max Device in the Library 903

Presets are a feature of Live that permit you to store the current state of a device. Since it is Live, not Max, that does the saving, the state must be known to Live, which means a preset captures the state of all the parameters you have defined. This means that the value of a number box not connected to a parameter will not be saved -- Live presets are different from the Max preset object in this way.

Storing a Preset

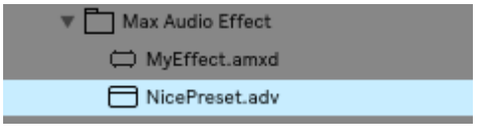
- Click the Save Preset icon in the title bar of the Max device. (The Save Preset icon is the one at the far right that resembles a *floppy disk*, for those of you who know what a floppy disk looks like.)



- In the File Browser, a new preset will be created and its name will be selected, ready for you to edit. Type in a name for the preset.

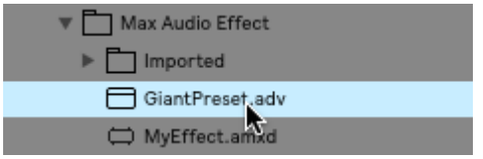
Presets for a Devices in the Library

When you store a preset for a Max device in the Library, it appears beneath the device hierarchically. In the example shown below, *NicePreset* is a preset that has been saved for the Max device *MyEffect*.



Presets for a Devices Outside the Library

When you store a preset for a Max device that is *not* located in the Library, the preset will have a special *preset-plus-device* icon and also show the device name in square brackets before the preset name. In the example below, we stored a preset called *GiantPreset* for an effect originally outside the Library called *MyGiantEffect*.



Saving a Max Device in the Library

If you want to save a device into the Live library, it's better to use the Save As... command within Max instead of moving the device file using your operating system. Using Save As... permits Live to keep track of your device and manage its presets.

- Insert the device you want to move. Click the edit button to launch Max to edit the device.
- In Max, choose **Save As...** from the File menu. Navigate the standard save file dialog to show the current Live Library folder. Save the device inside the Presets folder inside the Library folder, or a subfolder of the Presets folder.
- Return to Live and you will see the newly saved device. In the example below, we saved our device as *MyEffect*.



Preview Mode

Enabling and Disabling Preview Mode 904
Uses of Preview Mode 904
Preview Mode Caveats 904

After opening a Live device in Max for editing, there are two copies of the device -- the one you are editing and the one inserted into a track in Live. *Preview Mode* permits you to listen to the device you are editing while Live runs in the background. The copy of the device inserted in the Live track is disabled. Audio, MIDI, timing, and messages (including Live API commands) are sent between Max and Live in order to make this happen. Preview Mode is always enabled when you begin to edit a device in Max.

Enabling and Disabling Preview Mode

- To toggle preview mode, click the preview icon. When preview mode is disabled, the icon in the toolbar is gray.
- You can also turn preview mode off by clicking the edit button in the Live device's title bar. However, this also has the effect of closing the device in the editor.

Uses of Preview Mode

- You can turn preview mode off as you are editing a device to compare the edited version of a device with the original version.
- You can turn preview mode off after saving a device to see how its user interface appears in the Live device view.
- Preview mode will increase the latency of a Live device slightly. Turn preview mode off to hear the device with its original defined latency.

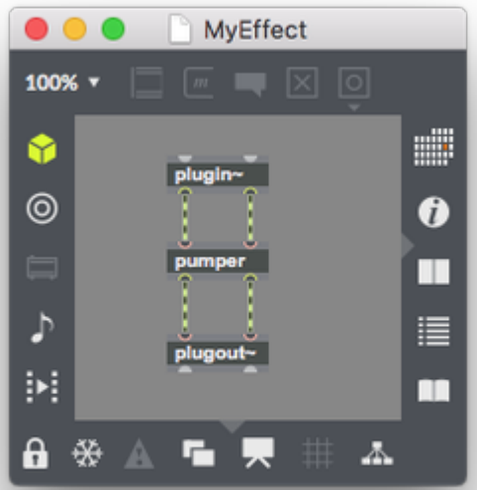
Preview Mode Caveats

- The memory state of Max objects is not made consistent between devices as you enable and disable preview mode. For instance, if you change a number box to 74 in a device in Live and then edit the device, the number box will be set to zero when the edited version is opened.
- However, parameter state will be updated when toggling preview mode. If you set a live.dial to 47 in Max, then disable preview mode, the dial will be updated to 47 in Live. If you then set the dial to 61 in Live, then enable preview mode again, the dial will be updated to 61 in Max.
- Changes to parameter values in Max while in preview mode are undoable in Live.
- Preview mode implements communicating between a send object in Live and a receive object in Max (or vice versa), but the performance of the communication will be reduced compared to the performance of send and receive within a single application.
- The grab object cannot communicate with a receive object unless both objects are in the same device, so you cannot use grab to communicate between devices in preview mode.
- Some Max objects monopolize a resource on your computer. The udpreceive object, for example, reserves a UDP port for itself so that no one else can access the port. This has the potential to create problems when you have two udpreceive objects in two copies of the same device. In this specific case, we have made the UDP objects release the port when they are disabled as a result of a preview mode switch. However, other third-party objects with similar resource-monopolizing issues may not work properly with preview mode.
- Max objects that share data via symbolic names, such as table, coll, and buffer~, do not share data across the boundary between Max and Live. For instance, if you have sample data loaded into a buffer~ object named `toto` in Live, and you edit another device with a buffer~ object named `toto`, the buffer~ object in the device in Max will not share data with the object in Live.

Resolving Conflicts in Frozen Devices

Overview of the Conflict Resolution and Unfreezing Process 906
Resolving Conflicts for a Frozen Device 907

The process of freezing adds copies of files in the Max search path to the device file. Consider the example shown below. A Max device called MyEffect contains an abstraction called *pumper*.



If we freeze this device and save it, a copy of the pumper patcher file will be added to the device. If we now open pumper, modify it, and save the changes, the frozen device is unaffected. However, what happens if we decide we want to work on our MyEffect device some more? At this point, there are two different versions of Pumper, the one in the device, and the one saved on disk. Which one should we use?

Overview of the Conflict Resolution and Unfreezing Process

Whenever you edit a frozen device, Max compares the files in the device with the versions on disk. If it spots differences, it disables the unfreeze icon and unlock icons in the patcher toolbar and enables the resolve conflicts icon, as shown below.



Before you can unfreeze a device with conflicts, you'll need to resolve them. Once you have decided which file(s) you wish to work with, both the version you want to keep *and* the version you want to discard will be written into a special folder located in your computer's Desktop folder called Unfrozen Max Device Files. However, only the version you wish to keep will be in the search path. The other version is put into a Discarded folder that is kept out of the search path.

Resolving Conflicts for a Frozen Device

- Click the Resolve Conflicts icon in the patcher toolbar. The Resolve Conflicts window will open.
- Use the Action pop-up menu for each listed file with a conflict to choose which version you wish to use.
- Once all conflicts have been resolved, the icon in the patcher window toolbar will turn gray, its caption will indicate No Conflicts, and the Unfreeze icon will become enabled.
- Click the Unfreeze icon in the patcher window toolbar to unfreeze the device.

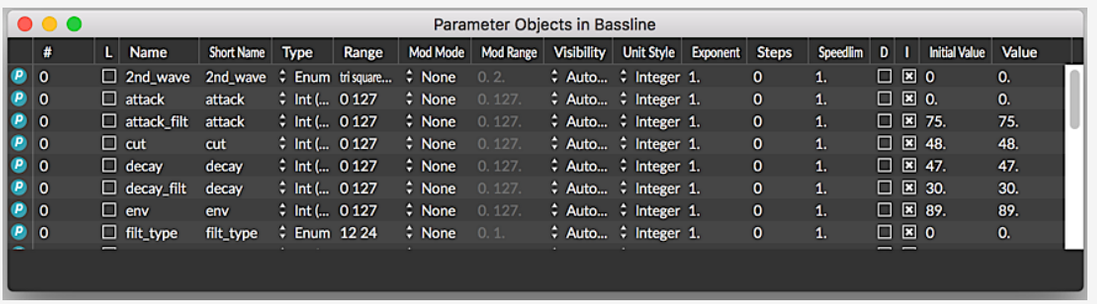
The Parameters Window

Displaying the Parameters Window 908
Locating a Parameter Object 911
Setting Parameter Object Values 912

The *Parameters Window* provides an overview of all the parameters currently associated with a Max for Live device, whether they are built into the UI objects or defined via pattr objects.

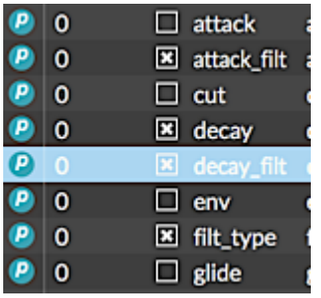
Displaying the Parameters Window

- Choose **Parameters** from the View menu when a device patcher window is the frontmost window. The Parameters window will appear.



- The **#** column displays the setting for the `Order` attribute, which can be used to set the order in which parameters are recalled when a device is loaded. By default, all attributes are recalled sequentially/at one time. By setting an attribute to have a lower order number, you can recall parameter values in complex patches whose initial values may need to be calculated based on initial or preset values so that the dependencies are preserved.

- The **L** column displays the setting for the `Link to Scripting Name` attribute, which can be used to link the Scripting Name used with the pattr family of objects and patcher scripting to the `Long Name` and `Short Name` attributes. If the attribute names are linked, the box is checked and the name of the parameter is shown in a different color text.



- The **Name** column displays either the `Scripting Name` attribute or the `Long Name` attribute, depending on whether or not they are linked. The `Long Name` attribute sets the name that appears in Live's automation view.
- The **Short Name** column displays either the `Scripting Name` attribute or the `Short Name` attribute, depending on whether or not they are linked. The `Short Name` attribute sets the name that appears as a part of the UI object's display.
- The **Type** column displays the `Type` attribute used to set the type of data associated with the object. The choices are:

  - Float: floating point number
  - Int: integer with a range of up to 256 values
  - Enum: enumerated list (used for objects such as multislider and umenu objects)

- The **Range** column displays the `Range/Enum` attribute that sets either a minimum/maximum or a list of possible values associated with the object's output.
- The **Mod Mode** column displays the `Clip Modulation Mode` attribute that sets the type of parameter modulation applied to the parameter. The clip modulation modes are described here.
- The **Mod Range** column displays the `Clip Modulation Range` attribute that sets the low and high range values within which a parameter is modulated. Some clip modulation modes use these parameter ranges for truncating modulation data.

- The **I** column displays the `Initial Enable` attribute checkbox that enables or disables setting an initial value as a part of a UI object's state.
- The **Initial Value** column displays the initial value of the parameter if the `Initial Enable` attribute is checked.
- The **Unit Style** column displays the `Unit Style` attribute used when adding labels to UI object displays. The unit style type for display can be set using the object's Inspector. The choices are:

    - Int: integer
    - Float: floating point
    - Time: output values will have a ms (milliseconds) label. Larger values automatically display seconds and milliseconds.
    - Hertz: output values are labeled in Hz (Hertz). Larger values automatically display in kHz.
    - deciBel: output values are labeled in dB
    - %: output values are displayed as an integer percentage value
    - Pan: output values are labeled as Left/Right pan integer values based on the range set for the object. When a range value between negative and positive integer values (e.g. -100. 100.), negative values will be labeled as left or right channel settings (-100L, 100R). If the range settings are have an upper range value of 0, the value will be reported as `value L`. only Left channel values will be displayed. If the range settings are have an lower range value of 0, the value will be reported as `value R`.
    - Semitones: displays values as negative/positive integer semitones for tuning displays
    - MIDI: displays integer values in the range 0 - 127 as MIDI note numbers (e.g. C#-2).
    - Custom: the label is user-definable, and is settable using the UI object inspector.
    - Native: displays the native unit style
- The **Exponent** column displays the exponential value used when calculating the object's output.
- The **Steps** column displays the the precision of the object's output when clicking and dragging. (The computer's keyboard can be used to enter any value.)
- The **Value** column displays the current value.

Locating a Parameter Object

- Click on the blue info icon in the left column for the parameter you want to locate and choose **Reveal in Patcher** from the pop-up menu. The object associated with the parameter will be highlighted.

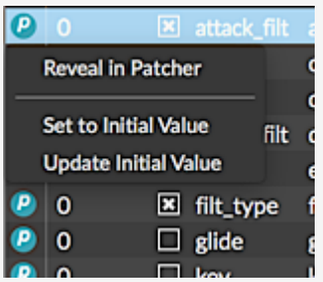


Setting Parameter Object Values

- Double-click in the Value column for a parameter setting to show a cursor and text box. Type in a new value or symbol for the initialization value, followed by a carriage return.

  You can also adjust the user interface object manually, click on the blue info icon in the left column for the parameter you want to set a value for and choose **Update Initial Value**.

- Click on the blue info icon in the left column for the parameter you want to set a value for and choose **Set to Initial Value** from the pop-up menu. The UI object will change to reflect the new initial value.



- Choose **Save** from the File menu to save the new initialization settings.

Unfreezing Devices

Rules for Unfreezing 913

Frozen devices cannot be edited. If you receive a frozen device from someone else that you'd like to edit, or you want to edit one of your own frozen devices, you'll need to unfreeze it first. To understand the unfreezing process, it helps to know what's in a frozen device.

- The main device patcher file
- Abstractions used by the device
- Third-party Max external objects
- Audio, image, and data files used by objects in the device
- Files added as explicit dependencies of the device By contrast, an *unfrozen* device contains only the main device patcher file. All the other files the device uses are expected to be somewhere in the Max search path.

The unfreezing process ensures that all files in the frozen device (other than the main patcher) exist in the search path. It is designed so that you will never lose data. If there are conflicts, you are forced to resolve those conflicts before you can unfreeze.

Rules for Unfreezing

Here is a description of what Max does during the unfreezing process.

- If a file exists in both the frozen device and the search path, and the file is identical (same modification date and size), no action is taken. This is typically what happens if you create a device, freeze it, and then unfreeze it.
- If a file exists in the frozen device but does not exist in the search path, it is written to a subfolder (named after the device) of the Unfrozen Max Device Files folder in the Desktop folder. This might happen if you receive a frozen device from someone else and want to unfreeze it in order to modify it.

- If a file exists in both the frozen device and the search path, but the file is different, then Max will take one of two actions depending on how you have chosen to resolve the conflict. An example file conflict is shown below.
- If you choose **Use Device Version**, the version in the search path will be moved to a Discarded Files subfolder of the device's subfolder of the Unfrozen Max Device Files folder. Then the version of the file in the device will be written to the device's subfolder of the Unfrozen Max Device Files folder.
- If you choose **Use Disk Version**, the version in the frozen device will be written to a Discarded Files subfolder of the device's subfolder of the Unfrozen Max Device Files folder. The version in the search path will be left untouched.

> Discarded Files folders are not in the search path. However, subfolders created when unfreezing devices are in the search path. When Max starts up, it looks for the Unfrozen Max Devices Folder, then adds any folders only one level below to the search path. This is different from the way the search path is typically constructed by recursively adding all subfolders of folders you have specified.

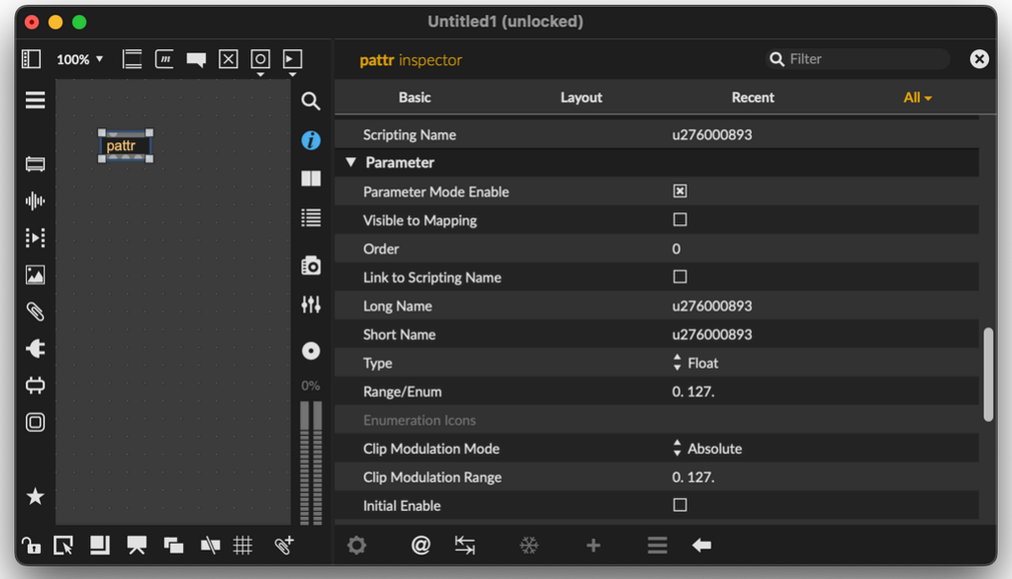
Using pattr in Live Devices

Enabling Parameter Mode 915
autopattr Considerations 916
Differences Between Max and Max for Live 916

Live-specific user interface objects such as live.dial save their state within Live documents and presets. If you want to use standard Max objects and have them interact with Live, you will have to enable the Parameter Mode Enable attribute in the pattr object. This ensures that the data internal to pattr is also stored (and recalled) using Live's document.

Enabling Parameter Mode

- Select a pattr object and click the Inspector button in the Patcher toolbar to show the object's
- Click the Parameter tab at the top of the inspector window to show the Parameter attributes.
- Check the Parameter Mode Enable checkbox.



autopattr Considerations

The autopattr object provides an easy to way manage the state of standard Max objects, but it will not work with the parameter system, so objects that are attached to an autopattr will not be seen by Live. If you want a pattr parameter to appear for modulation etc, you will need to add a pattr object for each instance.

Differences Between Max and Max for Live

Although the pattr objects can be used in the context of Max for Live, there are some differences when compared to Max. You can read more about various limitations here. To better understand pattr we encourage you to visit the pattr guide.

Using Symbols in Max for Live

Defining a unique symbol name 917

The "name space" in Max is global - when you have objects that have names associated with them such as coll, send, receive, table, or buffer~, you can share data between Max for Live devices. In these cases, the Max name space is shared, but the "signal processing space" is independent - each Max for Live device processes its audio or data separately.

Defining a unique symbol name

If you want a named object to be unique to a device, use three dashes (---) to start the name of your buffer~ or send / receive destination (e.g. `s ---filtercutoff`).

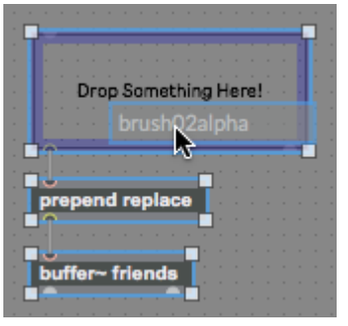
When your patch is initialized, it will replace the three dashes with a unique-to-Live number (e.g. `s 024filtercutoff`);

Working With Files in Max for Live

Setting Live's Preset and Export File Copying Behavior 919

Historically, Max developers have used the search path for convenient access to files by name, but this is not sufficient for managing files in Live presets or documents.

The live.drop object was designed to be added to Max devices that work with files. When you drag an audio file from either the Live file browser or your operating system's file manager, live.drop will output the complete file path as a single symbol. You can then pass this symbol to a buffer~ or sfplay~ object, as shown in the example below.



The live.drop object also *stores* the name and location of the most recently dropped file in a way that Live understands. By default, live.drop has its Parameter Mode Enable set to true, so any files dropped onto it will be saved in the current Live document. When the document is re-opened, the name will be sent out live.drop 's outlet. When saving a preset for a device containing a live.drop, Live will copy any referenced audio files to the Live library if they are not already there. When a user opens the preset, the file's path in the Library, not the file's original path, will be sent out live.drop's outlet.

If you perform a Collect All and Save on your Live Set, any audio files referenced by live.drop objects with Parameter Mode Enable set to true will be copied into a newly created folder. This permits Live users to move documents containing Max devices written with live.drop to different computers without losing the audio files the devices were using.

A Live user can, however, choose to turn off the program's default file-copying behavior.

Setting Live's Preset and Export File Copying Behavior

- Open the Live Preferences window.
- Click the **File Folder** tab.
- At the bottom of the File Folder preference pane, you will see an item called *Collect Files on Export*. *Always* is the default behavior. You can also choose *Never*, in which case files are never copied, or *Ask*, where you will be given the opportunity to decide whether files should be copied in each case that arises.

Appendix A: MC Extended

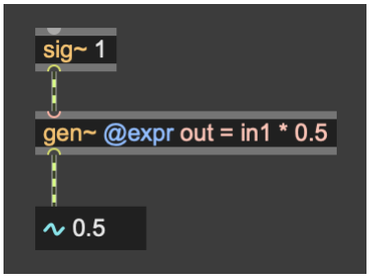
Gen Features for MC

Create Simple Gen Patchers with @expr 921
Control gen Execution with the @hot Attribute 921

New features have been added to Gen to support MC as well as to make integration of Gen easier with the remainder of your Max patching. The changes include:

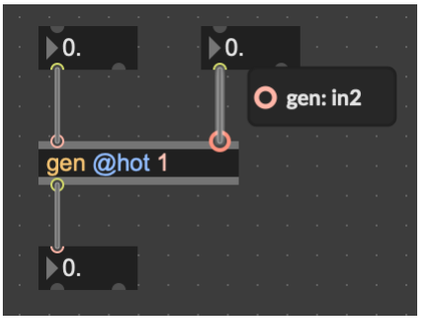
Create Simple Gen Patchers with `@expr`

You can use the `@expr` typed-in attribute with a gen, gen~, mc.gen~ or mcs.gen~ object to specify a single line of Genexpr code.



Control gen Execution with the `@hot` Attribute

By default, gen triggers processing only when receiving a value in its left inlet. The `@hot` attribute will set any other inlet to be "hot" and trigger processing.


0.
0.
gen: in2
gen @hot 1
0.

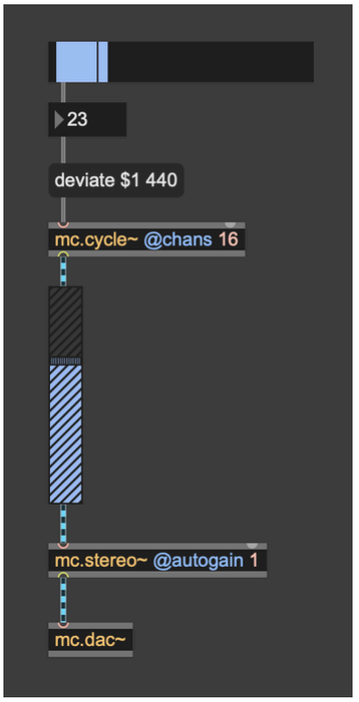
Generating Values for All MC Wrapper Instances

Generating Random Values 923
Generating a Range of Values 925
Generating an Exponential Series 926
Generating a Harmonic Series 926
Your Own Generating Algorithm 927

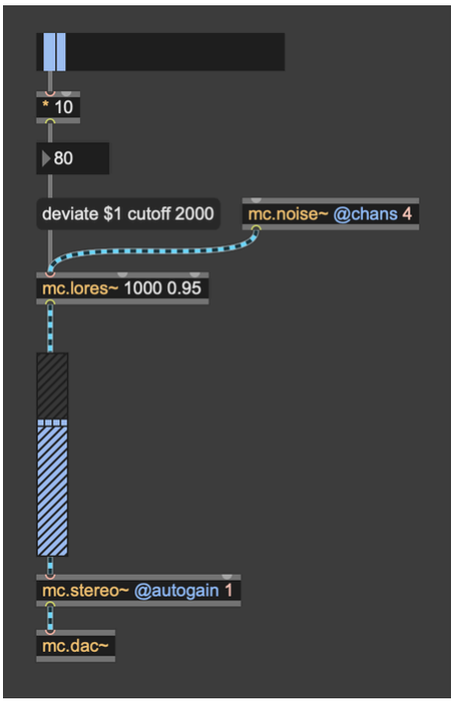
The MC Wrapper lets you treat all the objects within it as a "space" that can be changed globally with a single message. Here are some approaches using built-in wrapper functions. You can also generate your own control schemes using the lower-level `@applyvalues` message.

Generating Random Values

To generate a random space of values for all wrapper instances, use the `deviate` message. The first argument to the message is the "width" of the deviation. The second argument is the center value. Here is an example of changing the frequency of all oscillators in an mc.cycle~ object. As the slider value is increased, the range of values ("width") increases. When the slider is set to 0, values generated will be 440 because the width is 0.

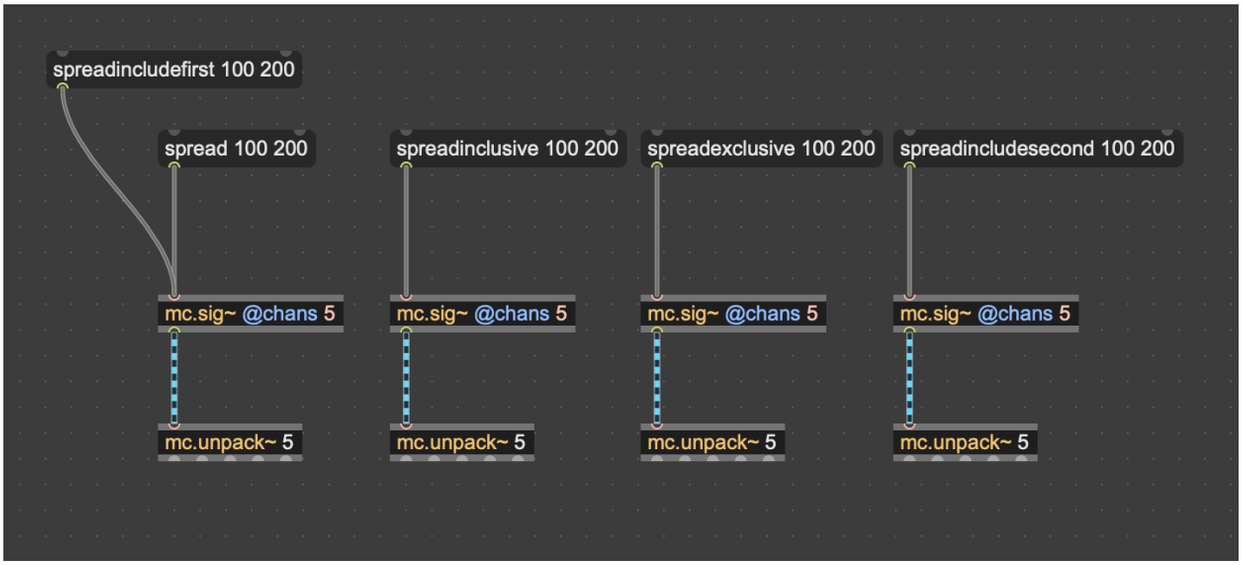


Optionally, you can specify an attribute or message that will be used instead of a float or int. Here is `deviate` applied to the `@cutoff` attribute of the lores~ object. It may help to think of the syntax of wrapper generation messages as follows: the first argument is the operating parameter (how, for example, `deviate` will do its job), and everything that follows is the message that it will use to change the value of an instance.



Generating a Range of Values

To set instances within the wrapper to a range of values, use the `spread` message. This message follows the same syntactical pattern as the `deviate` message. The first argument is the beginning of the range to generate, and the second argument is the end of the range. If the end is above the beginning, the first wrapper will receive the lowest value. If it's negative, the first instance will receive the highest value. The `spread` message has three variants that specify how the range is calculated. `spreadinclusive` includes both the first and second values in the range. `spreadexclusive` exclusives both the first and second values from the range. And `spreadincludesecond` includes the second value but not the first. By convention `spread` is the same as `spreadincludfirst`; it includes the first value but not the second value. The following examples show the differences in these variations:



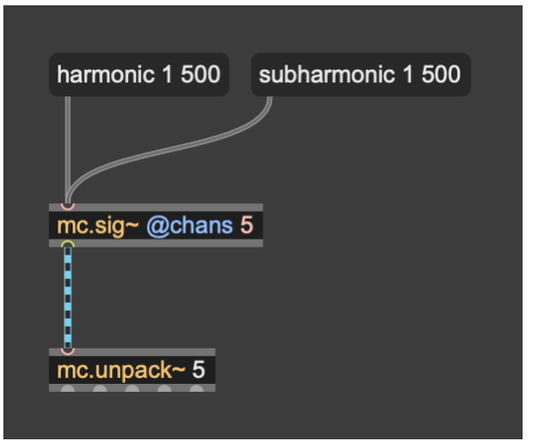
Generating an Exponential Series

To generate an exponentially increasing or decreating series of values applied to all instances in the wrapper, use the `exponential` or `scaledexponential` messages. The first argument is the value of the exponent in the series and the second argument is the base that is raised to the exponent. Negative exponents cause values to increase over the series while positive exponents cause values to approach zero. The `scaledexponential` variant divides values in the series by the number of instances in the wrapper, resulting in an overall range that is independent of the number of instances.

Generating a Harmonic Series

To generate a harmonic series of values applied to all instances in the wrapper, use the `harmonic` and `subharmonic` messages. Each message assigns the first instance in the wrapper to the first harmonic (or subharmonic), which is given by the second argument (the fundamental frequency) and subsequent harmonics or subharmonics to successive wrapper instances.

The first argument is a multiplier on the harmonic calculations, typically this is set to 1 for a harmonic series. As an example, here are the first harmonics and subharmonics for 500:



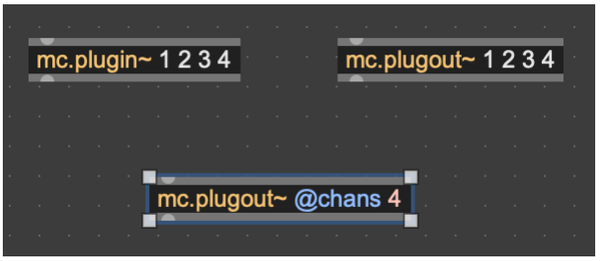
Your Own Generating Algorithm

In addition to the built-in MC Wrapper messages, you can define your own algorithms to control a space of channels:

- Javascript can produce an list of values to be used with the `applyvalues` message
- the mc.gen object can calculate distinct per-channel values with the **mc_channel** operator
- the mc.generate~ object can generate and update values for objects in the MC Wrapper at signal rate

MC and Max for Live

Max for Live devices receive audio input via plugin~ and send audio to Live via plugout~. The mc.plugin~ and mc.plugout~ versions of these objects accept multi-channel inputs and outputs to be routed to and from Max for Live.

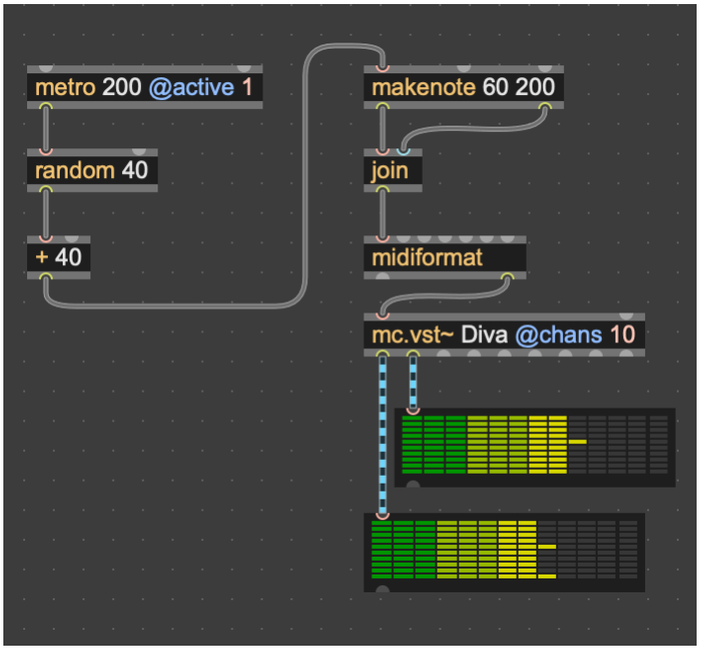


Devices created with multi-channel inputs and outputs will support multichannel routing within Live, as well as having the appropriate number of inlets and outlets when loaded into Max within the amxd~ object. The current maximum channel count is 64.

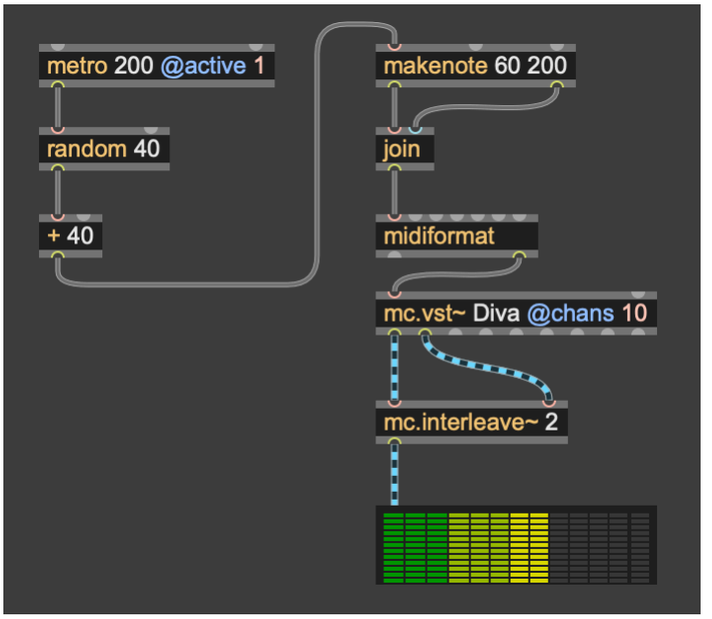
MC Channel Topology

Multichannel patch cords are useful as a way to organize audio signals. MC includes several objects that you can use to change how signals are organized.

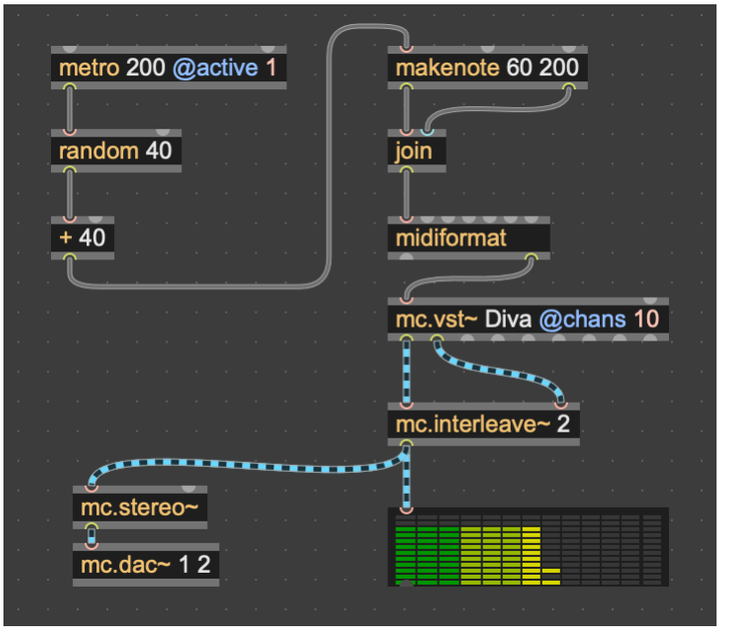
For example, if you are using mc.vst~ with a stereo synth plug-in that has 10 instances, that object will have two multichannel outputs with 10 channels each. The multichannel patch cord on the left has the "left" outputs for all 10 synths and the multichannel patch cord on the right has the "right" outputs for all 10 synths.



If want to mix all the synths to stereo, you will need to produce a two-channel signal routed to mc.dac~. To mix the signals as expected, use mc.interleave~ to produce a 20-channel signal that alternates the left and right channels.

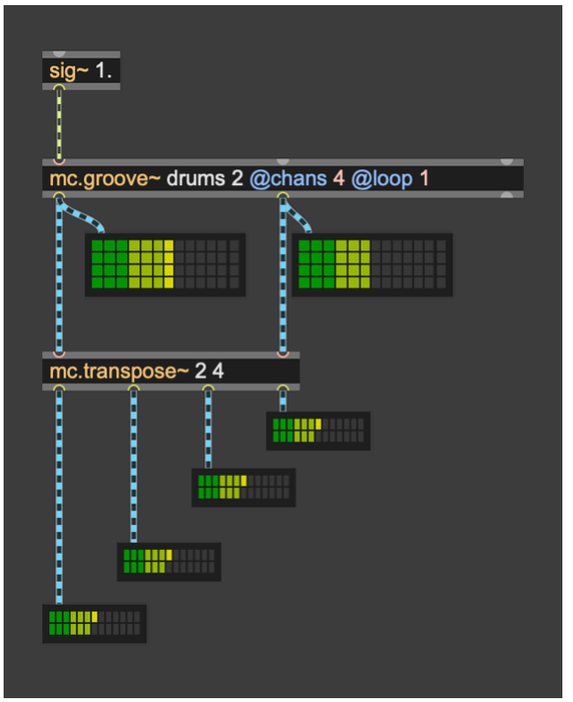


You can feed this 20-channel signal to mc.stereo~, which (by default) will add the odd-numbered channels (1, 3, 5...) to its first (or "left" output) and even-numbered channels (2, 4, 6...) to its second (or "right" output). The resulting signal can be connected directly to mc.dac~.

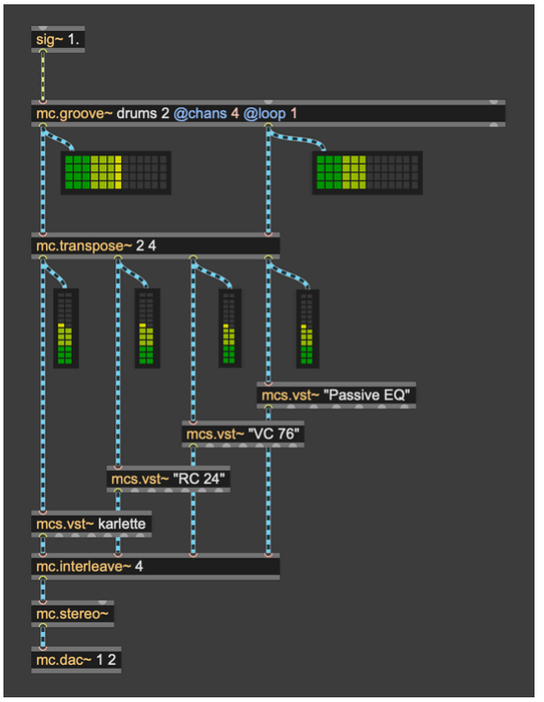


Use mc.deinterleave~ to transform an interleaved signal back to two non-interleaved signals. This is useful for separating signals for the inputs to mc.vst~, mc.poly~, or mc.gen~.

The mc.transpose~ object provides a more general way to reorganize multichannel signals. If you have four instances of groove~ in an mc.groove~ object playing back stereo samples, it provides you with two four-channel patch cords separating left and right channels. It may be more useful to have four two-channel patch cords, each of which has the left and right channels for the four instances. This can be accomplished with mc.transpose~:



Now you can add individual effects on a per-MC-instance basis, using mcs.vst~ as shown here:


sig~ 1.
mc.groove~ drums 2 @chans 4 @loop 1
mc.transpose~ 2 4
mcs.vst~ "Passive EQ"
mcs.vst~ "VC 76"
mcs.vst~ "RC 24"
mcs.vst~ karlette
mc.interleave~ 4
mc.stereo~
mc.dac~ 1 2

MC Dynamic Routing

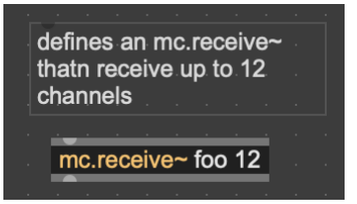
Sending Multi-Channel Audio Between Patchers 933
Dynamic Routing Using mc.send~ and mc.receive~ 933
Dynamic Multi-Channel Routing Using Max Messages 934

There are two basic methods of dynamic routing for multi-channel signals:

- Use mc.send~ and mc.receive~ to send multiple channels of audio between patchers
- Use mc.matrix~, mcs.matrix~, mc.selector~, and mc.gate~ to control routing of multichannel signals using Max messages

Sending Multi-Channel Audio Between Patchers

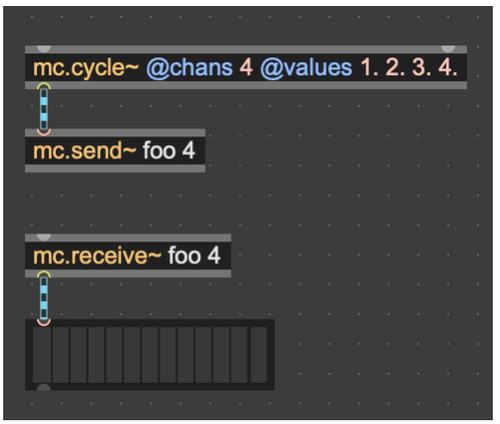
mc.send~ and mc.receive~ are multi-channel versions of send~ and receive~. An important limitation when using these multichannel versions is that you must expicitly define the number of channels you will be communicating between the objects via a typed-in argument that follows the receive name. The mc.receive~ cannot change its number of channels dynamically.



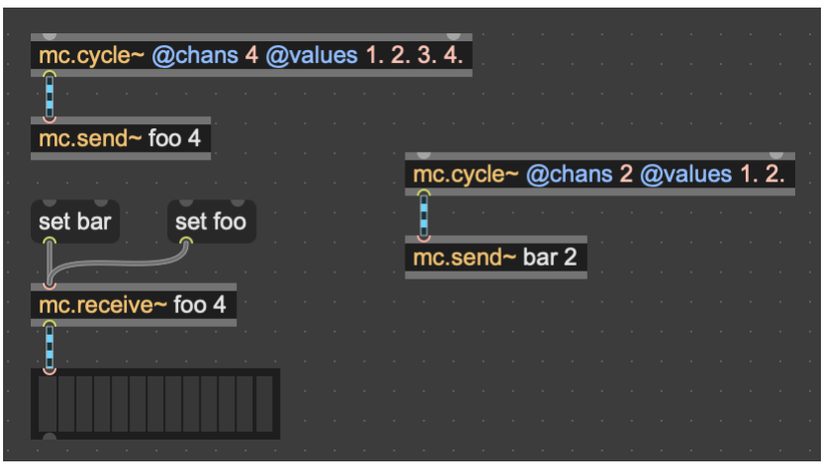
Any mc.send~ objects with a matching name can send up to the defined number of channels in the mc.receive~. Extra channels in a multi-channel signal connected to mc.send~ are ignored. If you supply too few channels to mc.send~, mc.receive~ will output these channels as zero signals. In other words, mc.receive~*always* produces a multi-channel signal with a set number of channels regardless of what you send it via mc.send~.

Dynamic Routing Using mc.send~ and mc.receive~

Like send~ and receive~, the mc.send~ and mc.receive~ objects accept the `set` message to change the source or destination name they use to communicate. However, the fixed channel limitation of mc.receive~ is tied to the object instance, not the symbol used for the destination name. Here is an example using the `set` message to mc.receive~. To start, the mc.receive~ uses the name `foo` and is communicating with the four-channel mc.send~ with name `foo`.

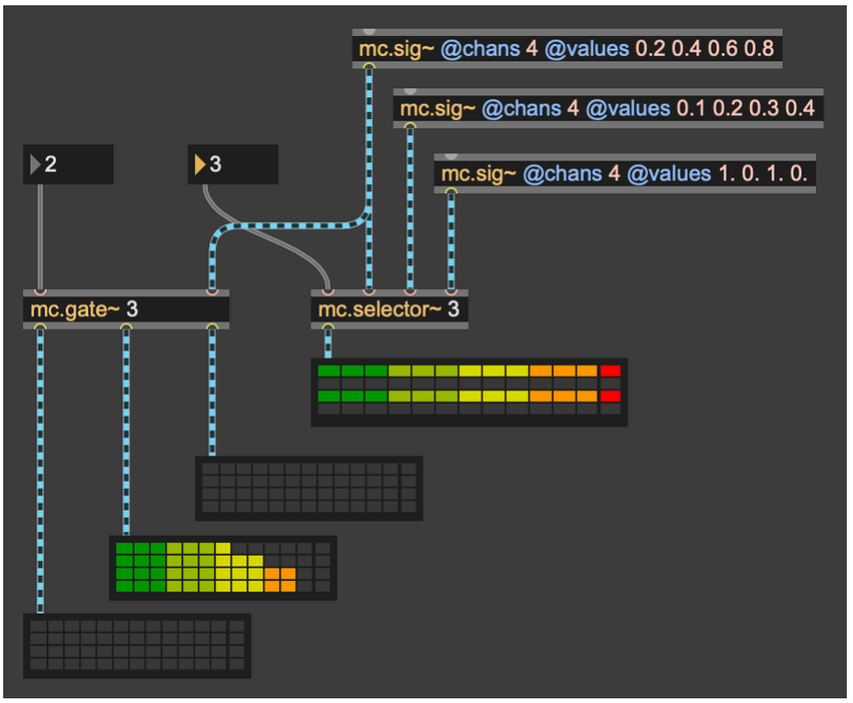


If we send `set bar` to the mc.receive~, the output is still four channels despite the fact that mc.send~ with name `bar` is only sending two channels.

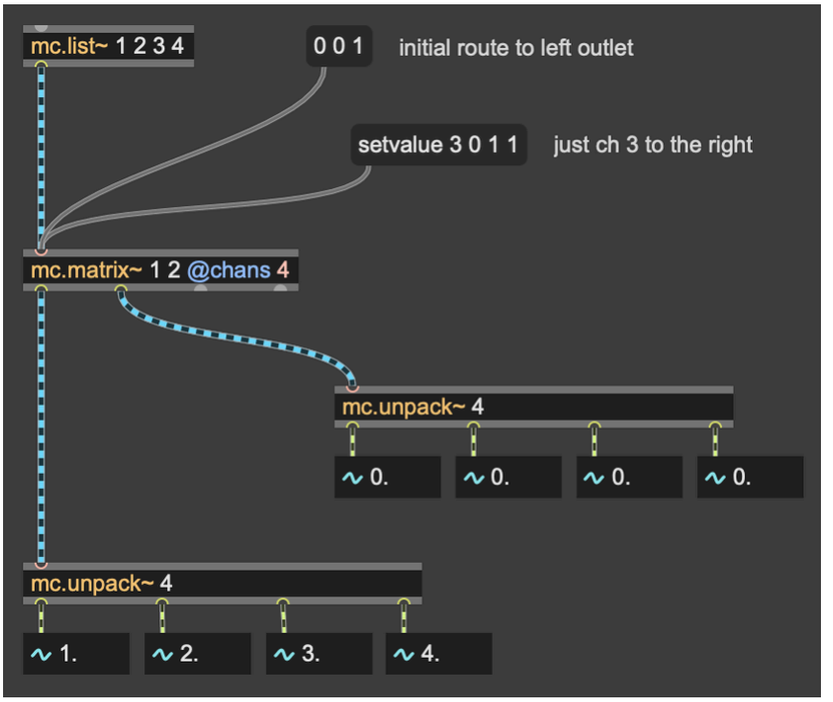


Dynamic Multi-Channel Routing Using Max Messages

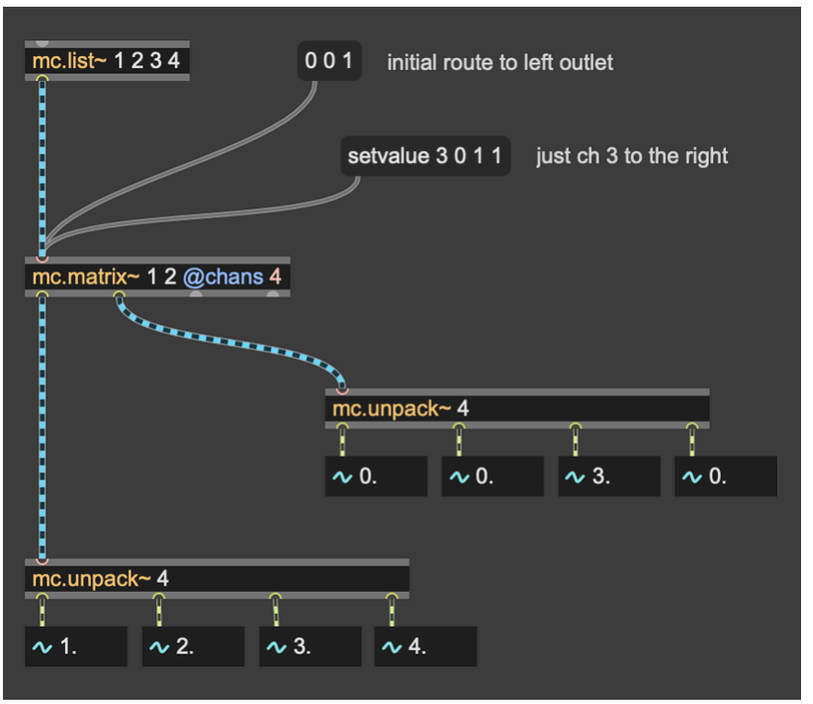
The mc.gate~ and mc.selector~ use the MC Wrapper and work similarly to their single-channel counterparts. The number or signal at the left inlet switches all channels of the multi-channel signal in the other inlets (mc.selector~) or outlets (mc.gate~).



The behavior of the two multichannel variants of the matrix~ warrants a more detailed explanation. The mc.matrix~ consists of multiple matrix~ objects in the MC Wrapper. However, because each matrix~ can be individually targeted, channels within a multichannel input can be routed to different multichannel outputs. In this example, there is an mc.matrix~ with one multichannel input and two multichannel outputs. By sending the message `0 0 1` (which goes to all matrix~ instances in the wrapper), we assign the inputs to the left outlet. The right outlet produces a multi-channel signal with zero in all channels.

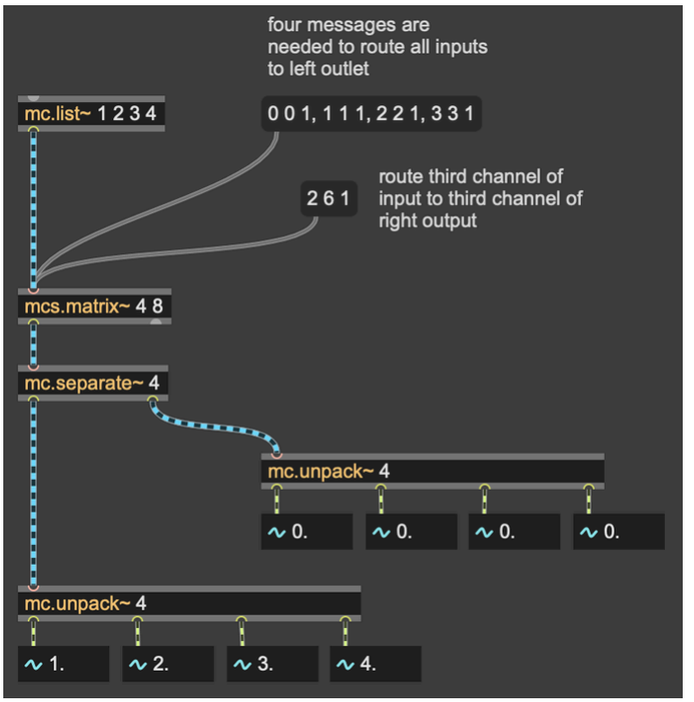


Using the wrapper's `setvalue` message to mc.matrix~, we can assign only the third channel of the input multi-channel signal to the right outlet using the message `setvalue 3 0 1 1`.

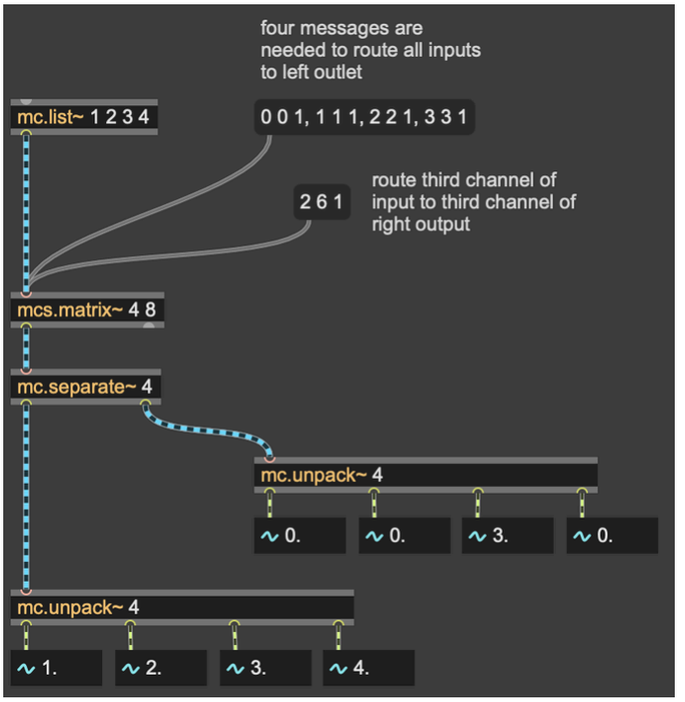


The matrix~ object numbers its channels starting at 0 whereas mc.target and the MC Wrapper `setvalue` message start channel numbering at 1. The special wrapper `setvalue` target of 0 before a message will address *all* instances inside a wrapped object, which is the same thing as sending the message without using `setvalue`.

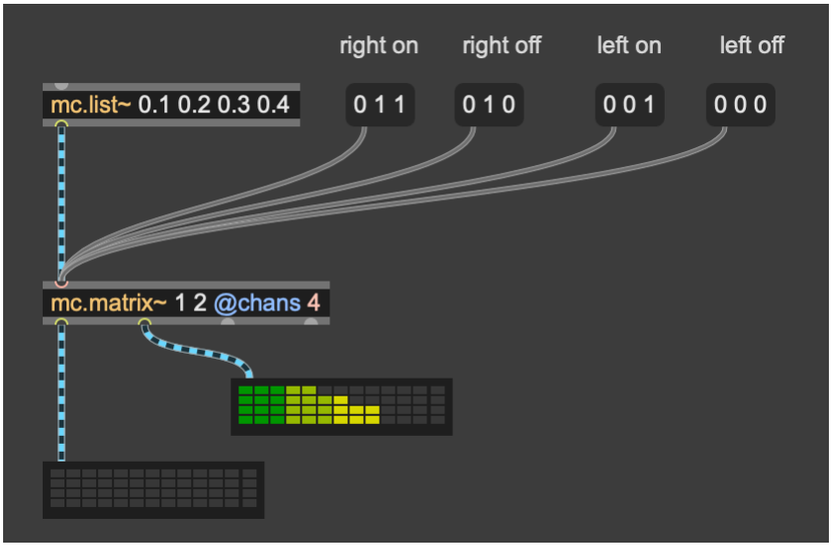
The mcs.matrix~ object is a single matrix~ object where all the signal inputs are combined into one multichannel input and all the signal outputs are combined into one multichannel output. To duplicate the above example with mc.matrix~ in mcs.matrix~, first observe that there are *four* total inputs and *eight* total outputs, despite the arguments `1 2` to mc.matrix~. To match this, we need an mcs.matrix~ with arguments `4 8`:



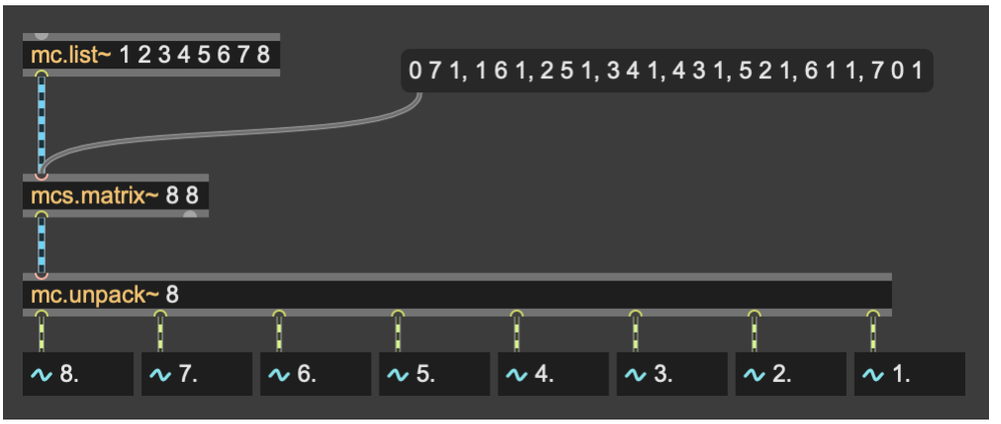
Since mcs.matrix~ provides only one multichannel output, we need to use an mc.separate to split its eight-channel multichannel signal into two four-channel multichannel signals. To route the third channel of the input in the same way, we send the message `2 6 1` (remember matrix~ channels are zero-relative):



Generally, you'll prefer mc.matrix when you want to route multichannel signals *as a whole* to different patch cords. This can be achived in one message with the mc.matrix example:



By contrast, mcs.matrix~ is an effective way to manipulate channels *within* a multichannel signal. For example, sending the messages `0 7 1, 1 6 1, 2 5 1, 3 4 1, 4 3 1, 5 2 1, 6 1 1, 7 0 1` in this mcs.matrix~ reverses the order of the eight input channels.


mc.list~ 1 2 3 4 5 6 7 8
0 7 1, 1 6 1, 2 5 1, 3 4 1, 4 3 1, 5 2 1, 6 1 1, 7 0 1
mcs.matrix~ 8 8
mc.unpack~ 8
8.
7.
6.
5.
4.
3.
2.
1.

MC Event Objects

- **multirange** - Two-dimensional function editor used to create definitions for mc.evolve~ and mc.gradient~
- **mc.gen** - Host multiple Gen patches for event processing
- **mc.target** - Format messages for individual voice targets
- **mc.targetlist** - Format messages for all defined voice targets
- **mc.line** - Produce multiple linear ramps with a voice identifier
- **mc.cell** - Format jit.cellblock selections into targeted event messages
- **mc.makelist** - Create lists using a target voice index
- **mc.route** - Route messages based on a target voice index
- **mc.function** - A version of the function object containing multiple functions
- **mc.input** - A version of the mc.target object with multiple inlets

See Also

Processing Events from MC Signals

Multichannel Function Generators

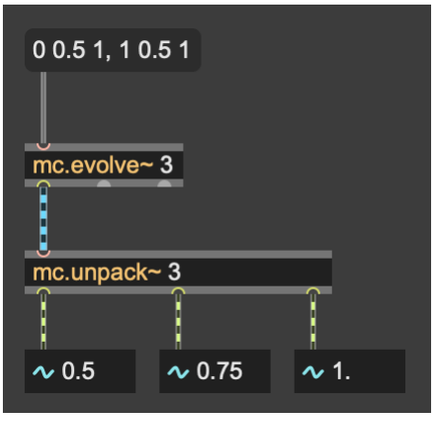
The mc.evolve~ Object 941
The mc.gradient~ Object 942
Creating Complexity With Multichannel Function Generators 944

Two MC objects, mc.evolve~ and mc.gradient~, generate repeating multi-channel control functions. The unique feature of these objects is that they allow you to specify function values as ranges; the objects uses these ranges to spread sample values within the range across the space of the output audio channels. The multi-channel control signals generated by these objects can be applied to oscillator or filter frequencies, audio levels, or pan positions.

By connecting the multirange user interface object, you can draw input functions for either mc.evolve~ or mc.gradient~ as they accept the same input format.

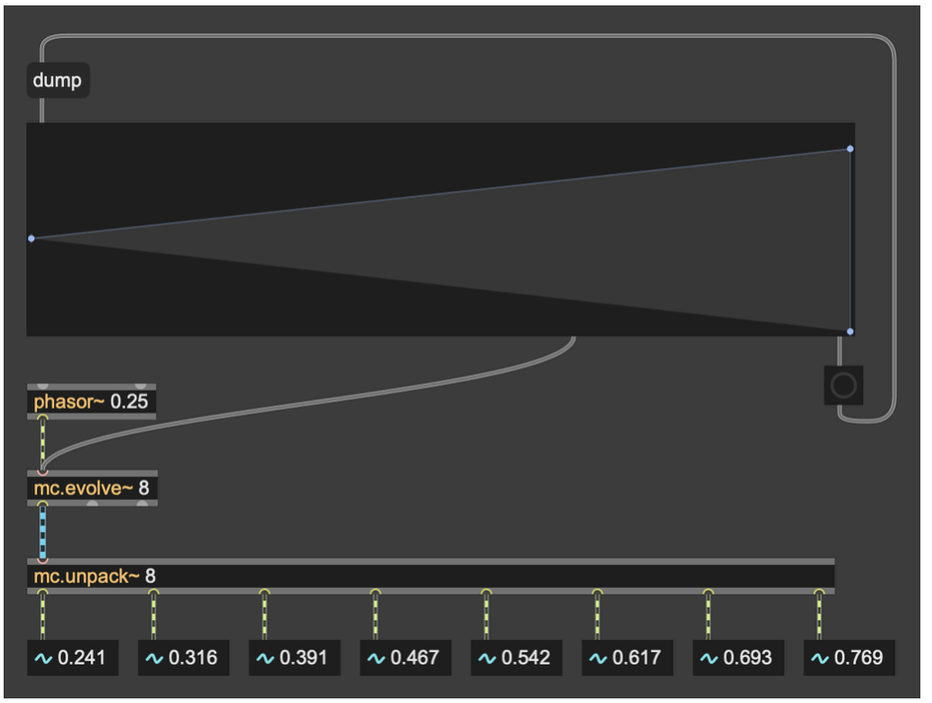
The mc.evolve~ Object

The mc.evolve~ object maps an input *driving function* from 0 to 1 over a series of ranges in a stored function. The value of the driving function represents an X position where 0 is the beginning and 1 is the end. Each output channel, starting at 1, fills out a range of Y values. As an example, sending two lists 0 0.5 1 and 1 0.5 1 establishes two ranges, one at X position 0 and one at X position 1. If there are three output channels, then the output, regardless of the "X" value of the input driving function, would be a constant multichannel signal containing 0.5, 0.75, and 1.



More interesting time-varying behavior begins when ranges specified at each X position are different. In this example all values at X position 0 are 0.5 because both the low and high range values are 0.5. But at X position 1, the range is from 0 to 1. If you drive mc.evolve~ with a phasor~, the effect of this range specification a diverging series of values that snap back to the middle as the phase resets.

We've used the multirange object to draw this range. Connect the third (dump) outlet of multirange to the left inlet of mc.evolve~. As you draw new functions, mc.evolve~ will update.

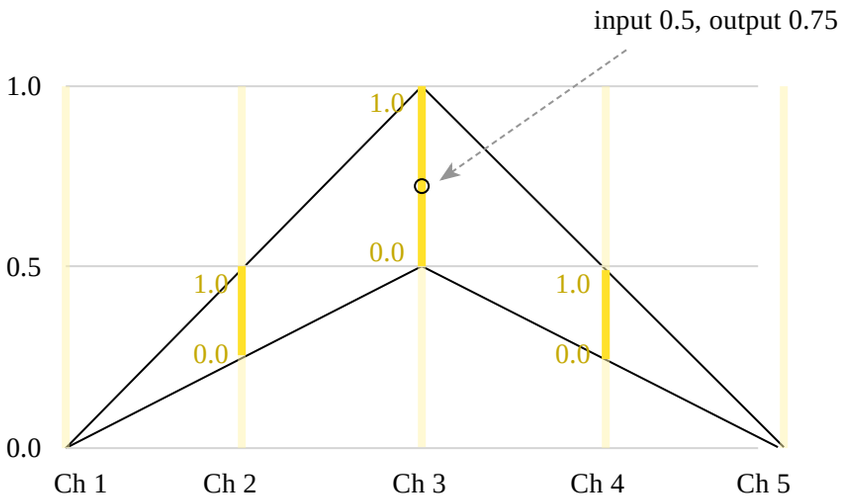


The mc.gradient~ Object

The mc.gradient~ object, like mc.evolve~, generates complex multichannel functions based on a stored set of ranges and an input driving signal.

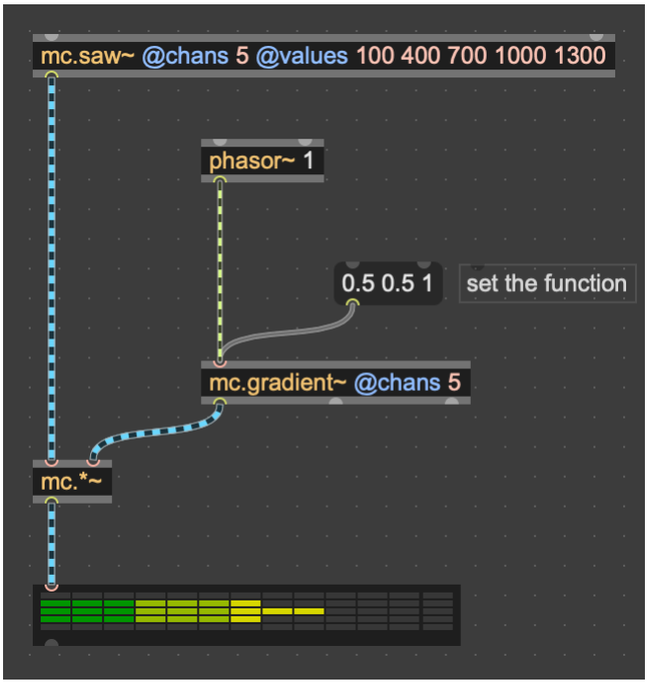
The mc.gradient~ object divides the X axis of a function into a number of evenly spaced "slices." The number and locations of the slices are determined by the object's `@chans` attribute. Each slice intersects a function of ranges. A channel's output value is determined by an input signal between 0 and 1 specifying a distance between the low and high range values at a given slice position. In the diagram below, mc.gradient~ is using five channels whose slices are located at X positions 0, 0.25,

0.5, 0.75 and 1. The output for channels 1 and 5 will always be zero, because there is no range at those X positions. However, at slice position 0.5 (channel 3), there is a range between 0.5 and 1.0. When the input signal is 0, channel 3's output will be 0.5, the low end of the range. When the input signal is 0.5, channel 3's output will be 0.75, the midpoint of the range. And then the input signal is 1.0, channel 3's output will be 1.0, the high end of the range.



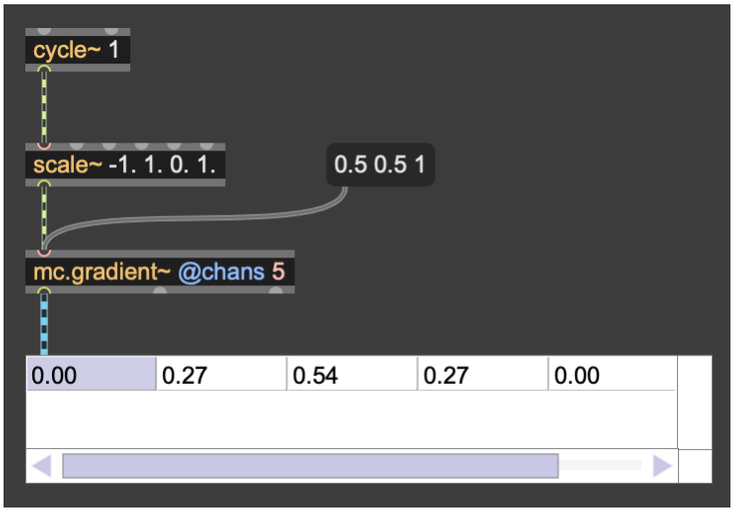
By default, mc.gradient~ has a function consisting of two points, one at X position 0 and one at X position 1. Both points have Y minimum and maximum values of 0. This means mc.gradient~ will output 0 for all channels given any input signal value. To create the function that corresponds to the above diagram, send mc.gradient~ the message `0.5 0.5 1.0`; this means: at X position 0.5, define a range between 0.5 and 1.0. Note that the X position need not correspond with a slice location; this is being done here to create more straightforward illustration of the object's behavior.

In this example, we apply the multichannel output signal produced by mc.gradient~ to the levels of five sawtooth oscillators. The mc.gradient~ object is driven by a phasor~ at 1 Hz.



The meter~ in the example shows that the level of channels 1 and 5 is always zero, while channel 3 has the highest level.

For a smoothly oscillating function output from mc.gradient~, map a cycle~ object to 0 - 1 using a scale~ as shown in this example:



Creating Complexity With Multichannel Function Generators

Both mc.evolve~ and mc.gradient~ are capable of generating complex evolving multichannel control functions. Here are a few ways to achieve complexity beyond the examples shown here:

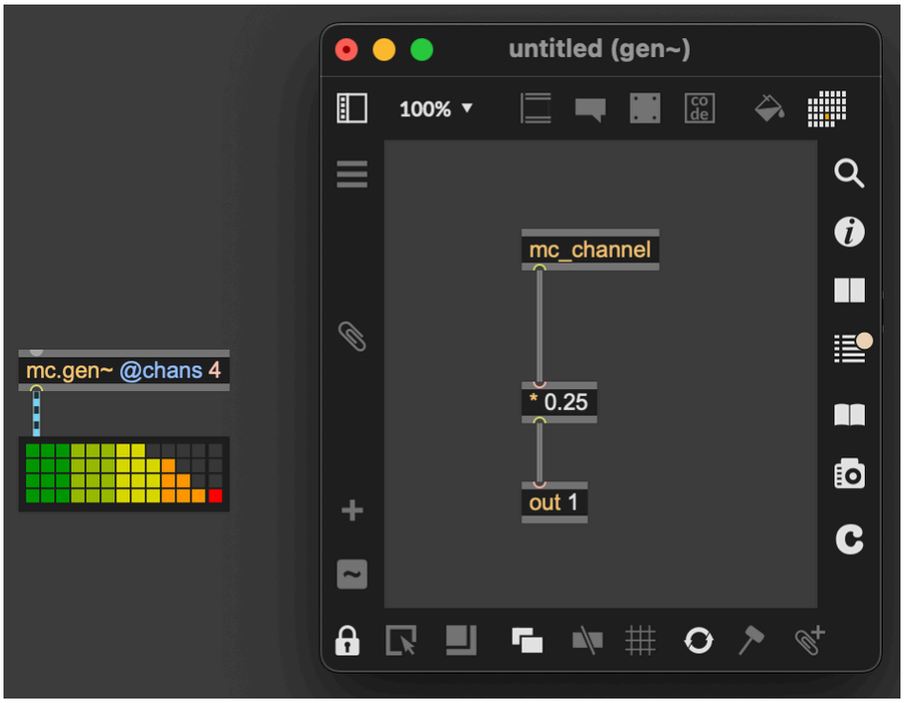
- Supply a multichannel signal to the input of mc.gradient~ - each input channel will drive its corresponding output channel indepdently.
- Each function point supplied to mc.gradient~ can have a fourth "phase" value that defines where the zero input value will start; this allows ranges to move in opposite directions as you move across the space of channel slices. By default the phase starts at the lowest point in the range.
- Use a more complex non-periodic input function such as one generated by a line~ object.

MC Gen Instances

Using One Gen DSP File 946
Using Multiple Gen DSP Files 947

When you create an mc.gen~ object, you are creating a hosting environment for multiple instances of a *Gen DSP* patch. There are several ways to manage the contents of the object, based on how the *Gen DSP* is loaded.

If you create an mc.gen~ object without a patcher name argument, you are working with the default, unnamed Gen DSP patch maintained by Gen. This patch is duplicated across all instances, and any changes to this patch will be propagated across all voices.



Using One Gen DSP File

When you load a single Gen DSP file into an mc.gen~ object, it is copied to all voices. If the patch is changed, the changes are immediately propogated to all voices as you edit.



Using Multiple Gen DSP Files

You can load each gen~ instance inside mc.gen~ with a different Gen DSP file. This is done using the `@values` wrapper message as a typed-in attribute. If fewer files are provided than channels are defined, the remaining channels will use the default Gen DSP patch.

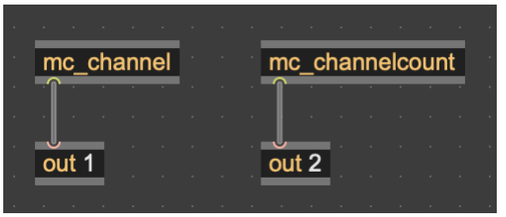


In this case, if you make a change to one of the patches, it *will not* propogate any changes to the other voices - each of the Gen DSP files is isolated from the others.

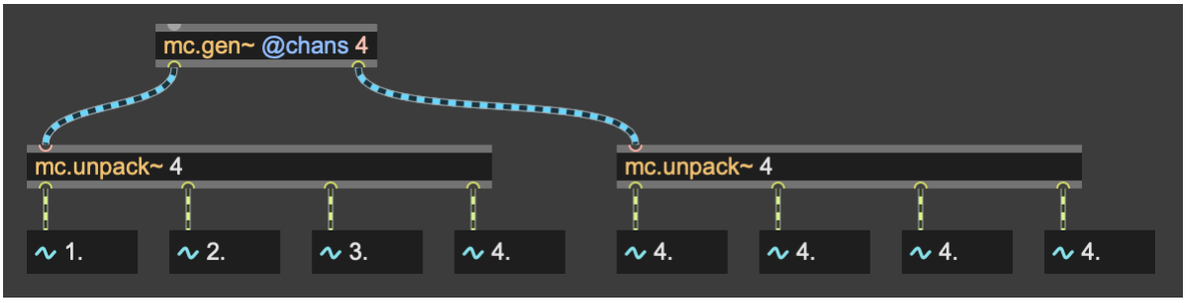
MC Gen Operators

Two Gen operators are useful with mc.gen~ and mc.gen:

- **mc_channel** reports the current channel of the Gen patcher within mc.gen~ or mc.gen (starting at 1).
- **mc_channelcount** reports the total number of channels (Gen instances) within mc.gen~ or mc.gen.



These Gen operators make it possible to do voice-specific calculations. You can also connect them to the Gen out operator to provide voice-specific identification signals or events outside the Gen context.



When used in a patcher inside gen~, mcs.gen~, or gen (in other words, outside the context of the MC Wrapper), both operators output 1.

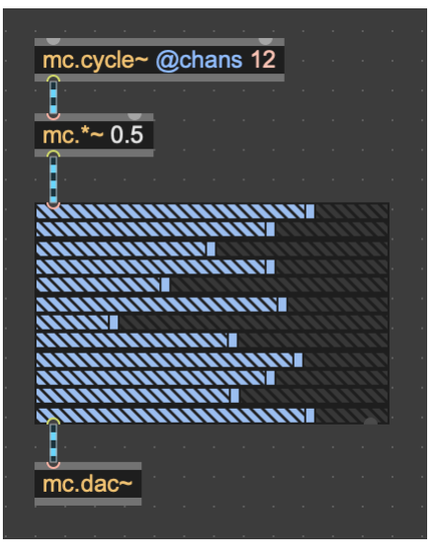
MC Managed Polyphony

The Basis for Polyphony 949
MC Polyphony using MIDI Notes 950
MC Polyphony using Max Messages 951
Managing DSP Resources 952
Enabling the Busy Map 953
Using Named Busy Maps 953

The MC Wrapper can manage polyphony without the use of the poly~ object. You can assign Max messages or MIDI events to specific MC channels and turn off processing for unused channels.

The Basis for Polyphony

Using the poly~ object, a polyphonic synth has a defined voice count. poly~ will load a separate patcher for each voice, which generates or processes sound independently from the other patchers. By contrast, with MC-based polyphony, there is a single patcher with MC wrapper objects set to a number of channels equal to the total voice count. For example, in this simple example, the mc.cycle~, mc.*~, and mc.gain~ objects have 12 channels. If you could control these objects independently, you could hear 12 independent voices.



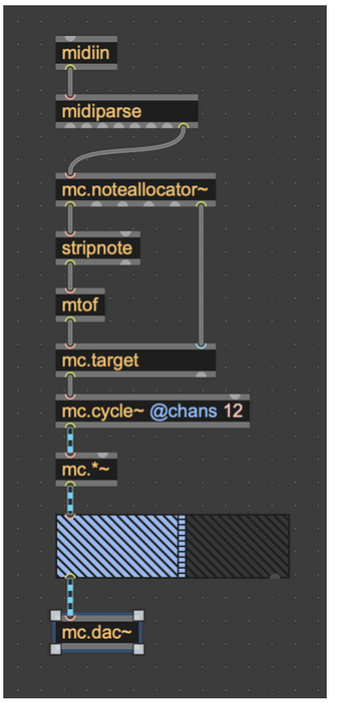
A good polyphony management system has three main benefits:

1. To the maximum extent possible given the fixed resources allocated to your synth, new notes will not cut off already-playing notes.
2. Voices allocated are spready cyclically to all channels, permitting pleasing panning or mixing effects based on channel number. For example you could spread channels across a stereo space knowing that on average, notes played will tend to fill the space.
3. A polyphony management system can help keep track of which voices are actually playing and manage DSP resources by shutting off process for voices not producing sound.

As with the Max poly~ object, the MC polyphony system provides all three benefits. DSP resource management is more involved with MC polyphony than with poly~; we'll cover that in the final section below.

MC Polyphony using MIDI Notes

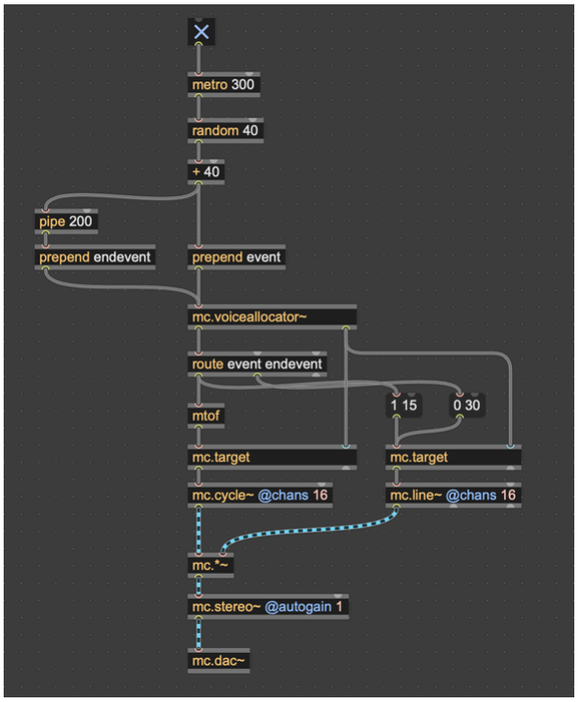
The mc.noteallocator~ object accepts MIDI messages and directs messages to a specific target voice based on its internal voice allocation mechanism. Voices can be allocated either on the basis of MIDI note-on and note-off messages, or, more typically, by connecting a multichannel audio signal to the left inlet of mc.noteallocator~. In that case, when a note-off for an allocated voice has been received *and* the corresponding audio signal in the multichannel input goes to zero, mc.noteallocator~ marks that channel as available to be allocated when future MIDI note-on messages are received.



To specify a maximum number of voice numbers for allocation, supply a value for the `@voices` attribute. By default the `@voices` attribute has a value of zero, which means that if a multichannel audio signal is connected to the left inlet of mc.noteallocator~, it will use the number of channels in the incoming audio signal. If no audio signal is connected, mc.noteallocator~ will use a default of 15 voices (mc.voiceallocator~ works similarly but defaults to 16 voices).

MC Polyphony using Max Messages

If you prefer to allocate and control MC channels using Max messages instead of MIDI notes, you can use the more general mc.voiceallocator~ object. Here is an example driven by a metro object that allocates and releases voices using mc.voiceallocator~.



The mc.voiceallocator~ object accepts the `endevent` and `release` messages to free an allocated voice, so if you want to write your own audio analysis algorithm to detect when a "note" has finished playing, that algorithm could trigger these messages.

Managing DSP Resources

MC polyphony manages DSP resources by turning off channels of processing within each object using the MC wrapper. To enable/disable this behavior you need to use the `@usebusymap` attribute for each MC object.

A *busy map* is maintained by mc.noteallocator~ and mc.voiceallocator~ to keep track of which voices are currently defined to be playing ("busy"). These objects use the map to decide which MC channels are available for use; effectively implementing the first benefit of polyphony management mentioned above (new notes don't cut off playing notes).

Keep in mind that an object such as mc.cycle~ in the MC Wrapper consists of a set of MSP cycle~ objects. When you use the busy map, wrapper-based objects will turn off processing in individual cycle~ objects within an mc.cycle~. If fewer than the maximum number of voices are playing, this can save CPU resources.

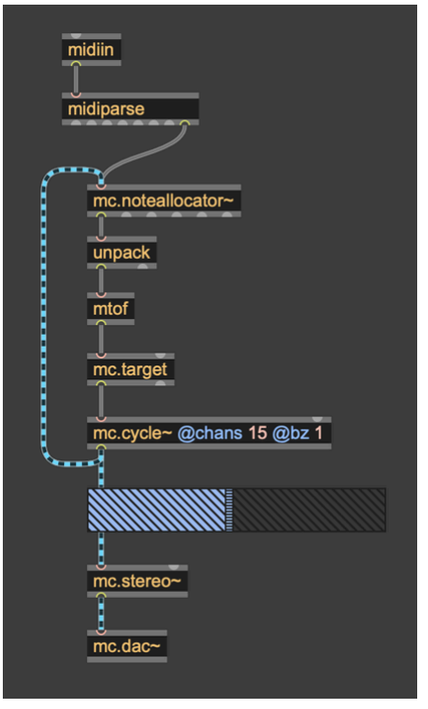
MC objects not based on the MC Wrapper do not implement busy maps because they have no way of turning off processing for individual channels.

Enabling the Busy Map

To enable the use of the busy map for any wrapper-based object:

- For all wrapper-based objects whose DSP resources you want to control, set the `@usebusymap` attribute to 1. For brevity the name `@bz` can also be used.

Here is the mc.noteallocator~ example above where all wrapper-based objects have `@bz 1` included.



The `@usebusymap` attribute uses a patcher-global busy map. If you want to control wrapper-based objects in several patchers, or you have more than one mc.noteallocator~ or mc.voiceallocator~ in the same patcher, you can use *named busy maps* described in the next section.

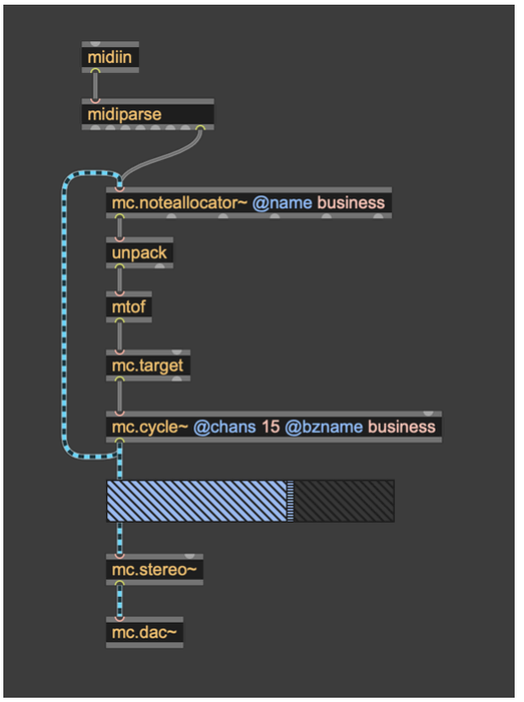
Using Named Busy Maps

The mc.noteallocator~ and mc.voiceallocator~ objects include a `@name` attribute that specifies a symbolic name to associate with their internal busy map.

To control DSP resources of a wrapper-based object using a named busy map:

- Create an mc.noteallocator~ or mc.voiceallocator~ object with a `@name` attribute
- Add the `busymapname` (`bzname` for short) attribute to each wrapper object you want to control

Here is the mc.noteallocator~ example above using named busy maps:



MC Mixing and Panning

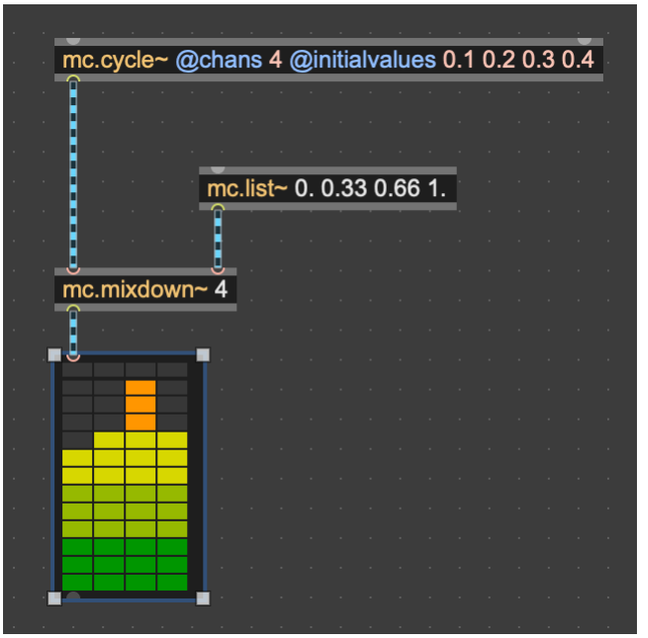
Mixing Multichannel Signals 955
Panning Modes 957
Auto-Adding Multichannel Signals 961
User Interface Objects for Multichannel Mixing 962

Mixing Multichannel Signals

The primary tool for mixing multichannel audio signals is the mc.mixdown~ object. This object mixes inputs in a multi-channel signal connected to its left inlet to an output multi=channel signal, optionally panning the inputs across the space of the outputs.

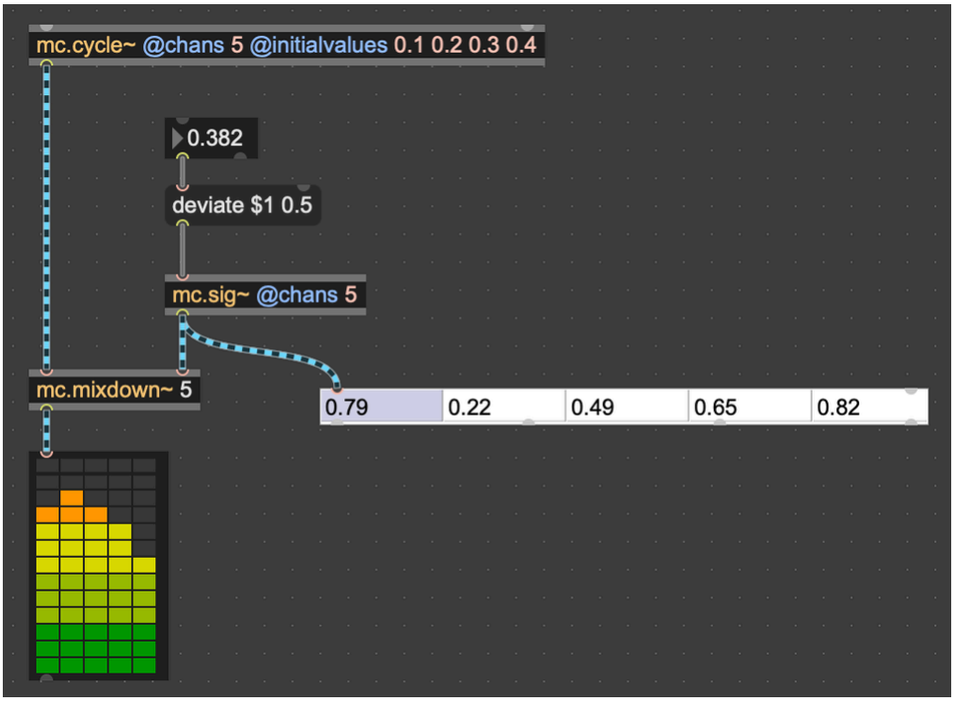
Static Pan Positions

There are two ways to specify a series of unchanging pan positions for inputs to mc.mixdown~. The most efficient way is by supplying a list of pan values to the `@pans` attribute, which can be a typed-in argument. Alternatively, you can use mc.list] or {mc.range to generate a multi-channel signal containing the pan values, then connect that multi-channel signal to the right inlet of mc.mixdown~.

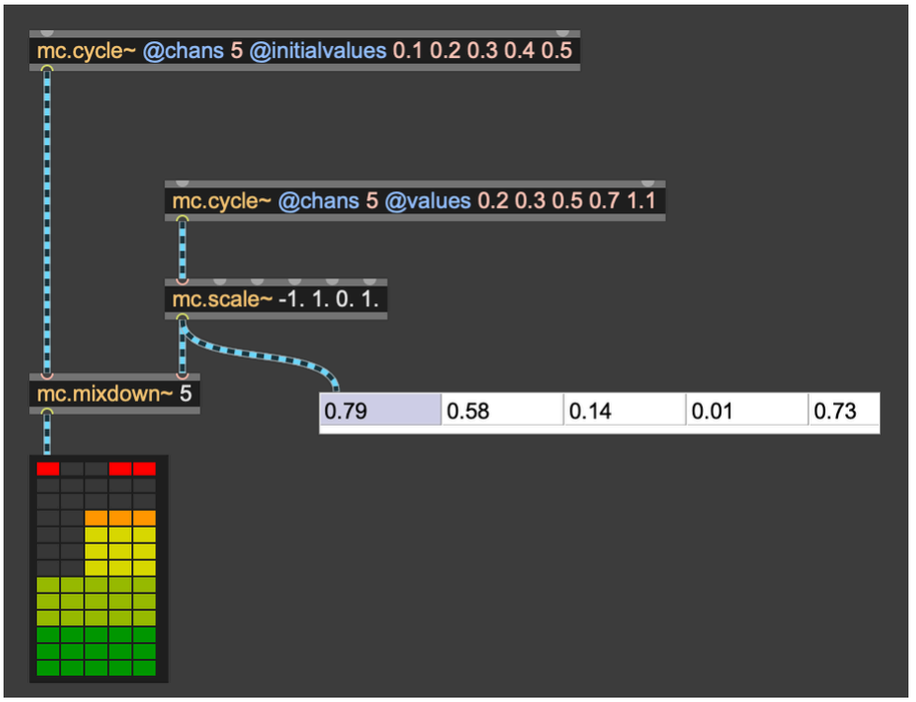


Dynamic Panning

For dynamic panning, you can use any multi-channel signal for the pan input of the mc.mixdown~ object. For example, we can use an mc.sig~ object with the `deviate` message to create a set of random values for channel panning.



Using oscillators for the panning position allows you to create evolving results with only a few objects.



Panning Modes

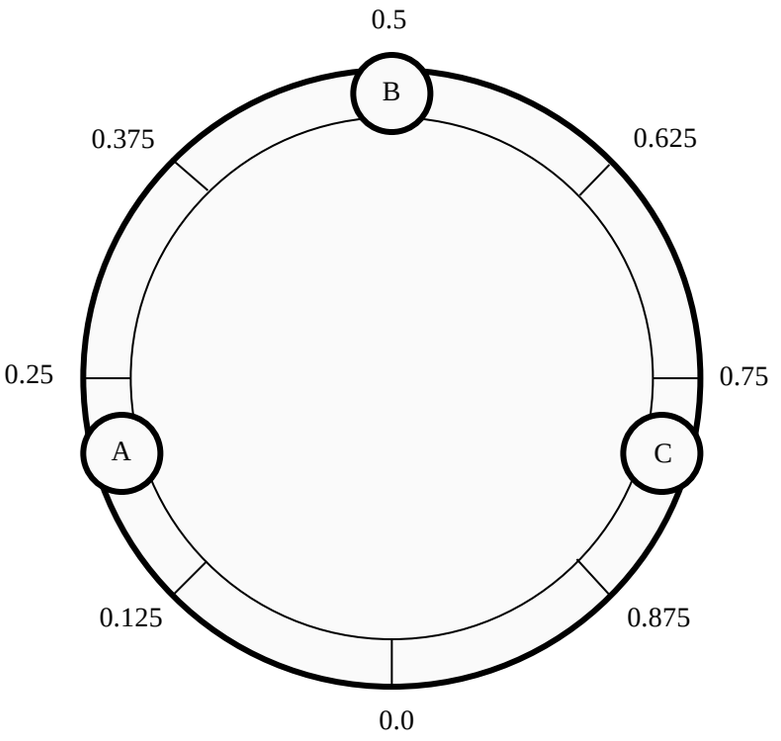
There are four panning modes available when using the mc.mixdown~ object: Mode 0 (Circular Panning), Mode 1 (Line Panning, ranged 0-1), Mode 2 (Line Panning, ranged 1-N) and Mode 3 (Circular Panning, ranged 0-N). Each uses a different method of determining where the outputs are located in relation to the number range available for panning.

Mode 0: Normalized Circular Panning (0-1)

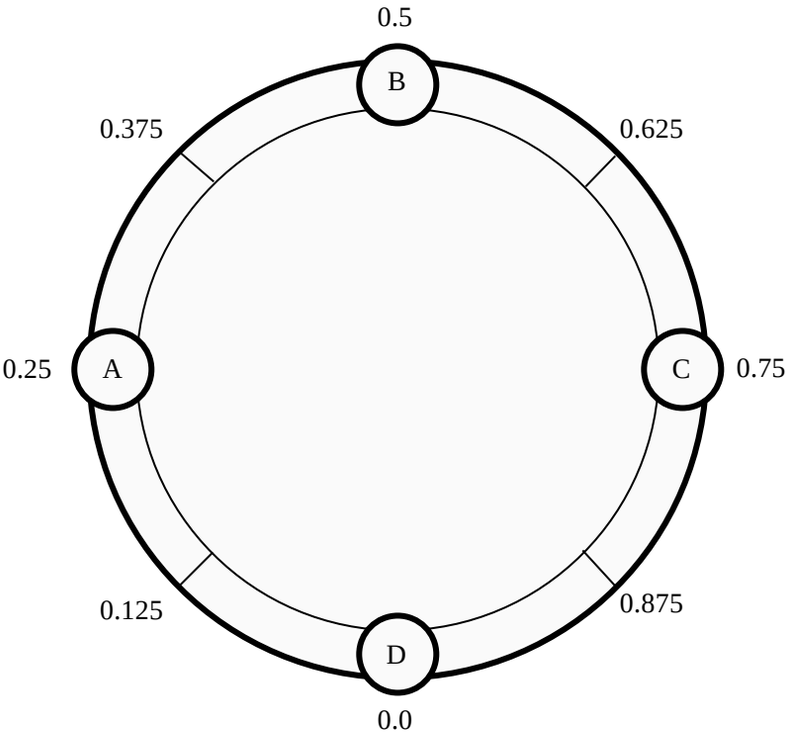
The pan position is a value between 0.0 and 1.0, and represents an even-power pan within a *circular* configuration of channels spread evenly across that range. For example, a two channel configuration has the two outputs at pan position 0.25 and 0.75.

Why not 0.0 and 1.0? Because 0.5 should represent a value evenly placed between the two channels, and 0.0 should also be halfway between the two channels - as if they were in a circle.

As you can see from the following diagram, if there are three outputs, they will be located at .167, .5 and .833.



Likewise, if there are four outputs, they will be located at 0.125, 0.375, 0.625 and 0.875.



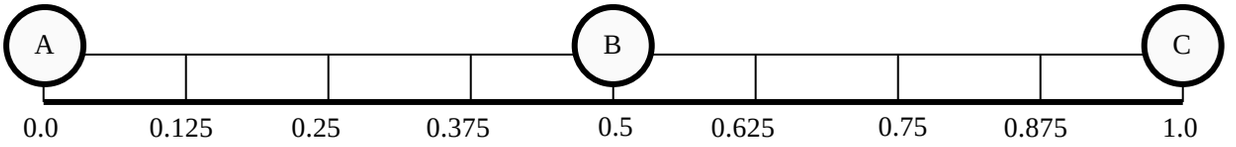
To determine the pan position of any single channel in a mc.mixdown~ scenario, divide 1.0 by the number of channels times 2, then use the odd-numbered values produced. For example, if you have 5 output channels, the pan positions for the outputs will be:

Calculate 1.0 / (5 channels * 2) = 0.1
The five output channels are 0.1, 0.3, 0.5, 0.7 and 0.9

Mode 1: Normalized Line Panning (0-1)

In this mode, the pan position is a value between 0.0 and 1.0, and represents an even-power pan across a linear configuration of channels spread evenly across that range, but with the first and last locations at positions 0.0 and 1.0.

This mode can be somewhat more easily understood, but the values are also clamped to the range of 0.0 to 1.0. If we have three outputs, their locations are at 0.0, 0.5 and 1.0:

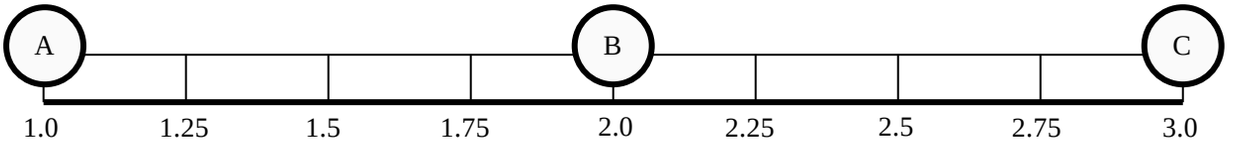


A four-output configuration would have the output locations at 0.0, 0.33, 0.66 and 1.0:

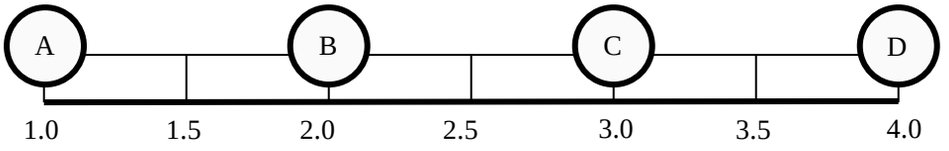


Mode 2: Channel-Based Line Panning (1-N)

Panning mode 2 is similar to mode 1, in that the values are placed on across a linear path with the values at the extreme ends. However, in this mode, each output is placed at its integer value, with the lowest output at 1.0, and the highest output located at its value. For example, a three-output configuration would have outputs at 1.0, 2.0 and 3.0:



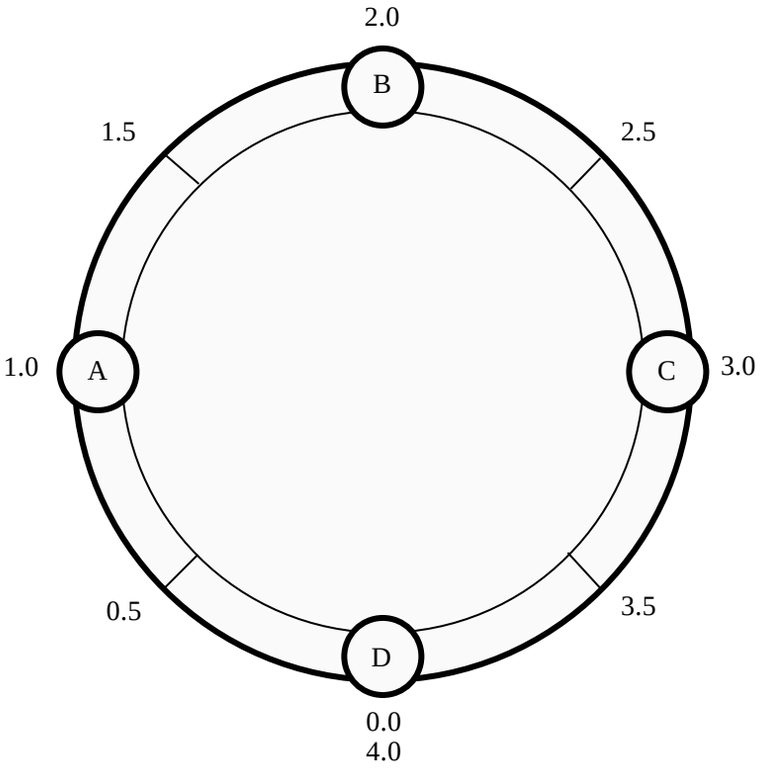
And a four-output object would have output locations at 1.0, 2.0, 3.0 and 4.0:



Like mode 1, the values are clamped at the top and the bottom of the range, so values under 1.0, or over the integer value of the highest output, will not produce expected results.

Mode 3: Channel-Based Circular Panning (0-N)

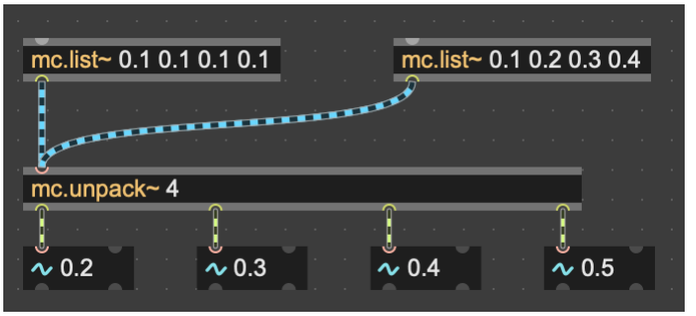
Mode 3 is similar to mode 0, in that the panning exists in a circular layout. But like mode 2, the outputs are located at their integer value locations. A four-channel configuration would have the outputs as follows:



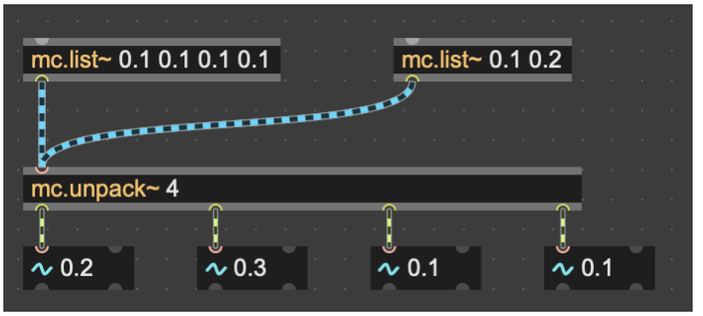
Note that location 0.0 is the same as the location of the last output, and the first output occupies location 1.0. This allows you to 'rotate' a sound around the outputs, but makes direct access to each output easier to manage.

Auto-Adding Multichannel Signals

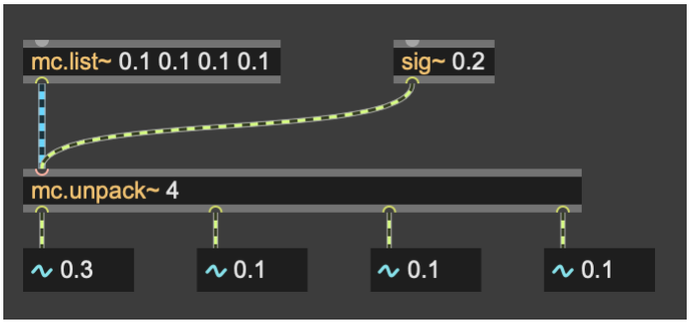
As with standard signal patchcords, combining two multichannel patchcords at an object inlet will mix the contents of prior to processing. When the two patchcords contain the same number of channels, the channels will be mixed together individually.



If one multichannel patchcord contains more channels than the other, the result will be the largest channel count of either of the patchcords.

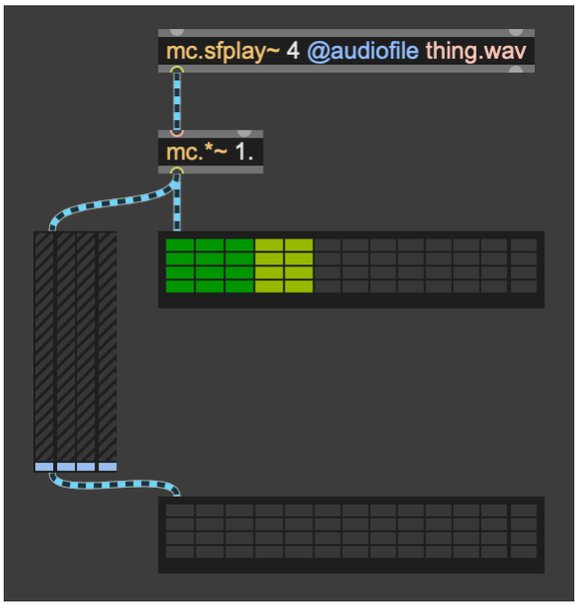


In many cases, you can also mix multichannel and standard audio signals, with the standard signal treated as the equivalent of a 1-channel MC signal.

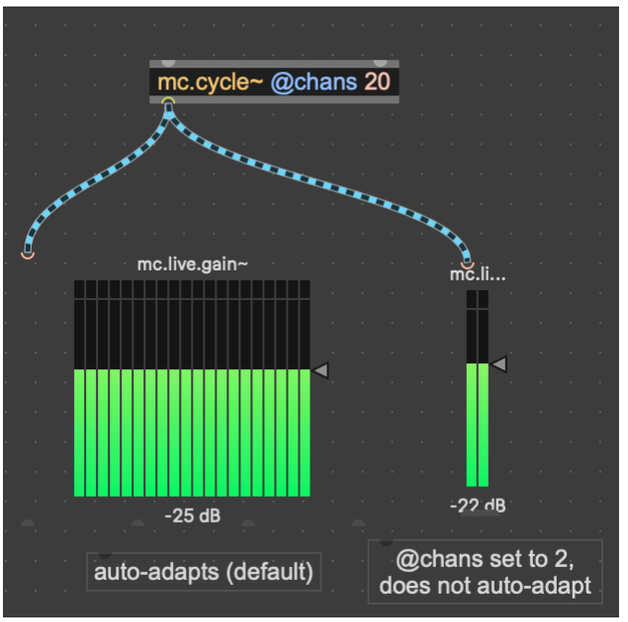


User Interface Objects for Multichannel Mixing

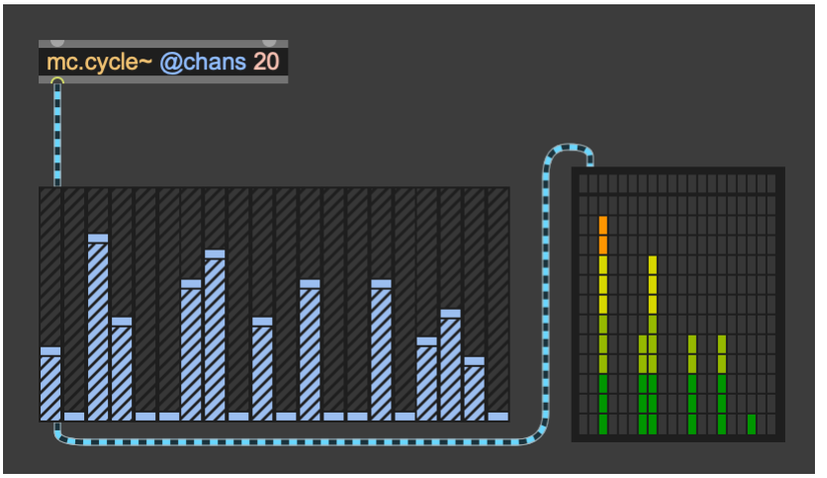
To set all levels of a multichannel signal, use the mc.gain~ object. By default, mc.gain~ auto-adapts to the number of channels in its input and produces the same number of channels, with the incoming signals scaled by the value of the gain slider.



To set levels of a multichannel signal individually, use the mc.live.gain~ or mc.multigain~ objects. mc.live.gain~ is a version of live.gain~ with a single multi-channel input and a single multi-channel output. Like mc.gain~ it auto-adapts to its input channel count by default. If the `@channels` attribute mc.live.gain~ is set to a non-zero value it outputs a fixed number of channels regardless of the input channel count.



The mc.multigain~ object is a version of mc.gain~ offering individual sliders for each input channel.



The mc.multigain~ object provides a way to set all sliders with modifier key gestures.

- To set a slider individually, click in the slider and drag to increase or decrease the level. You will not be able to change a different slider until you end the drag.
- To draw across all sliders setting them to the position of the cursor, hold down the option while dragging.
- To change the level of all sliders proportionally, hold down the shift key while dragging one slider.

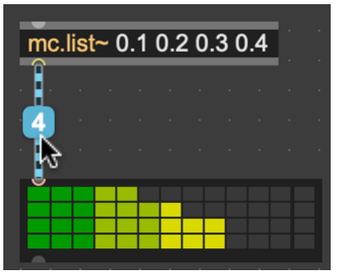
MC Patch Cords

Auto-Adding Multi-Channel Signals 965

MC patch cords carry multiple channels of audio simultaneously. This makes it easier to work with audio spatialization, complex synthesis voicings and explicitly polyphonic signals.

A patch cord will be multi-channel if it is connected to a multi-channel outlet of an MC object.

These patch cords have a distinctive appearance: they are blue and black (as opposed yellow and black single-channel patch cords). MC patch cords are also slightly thicker. You can quickly see how many channels are being used by hovering over the patch cord when the patcher is unlocked.

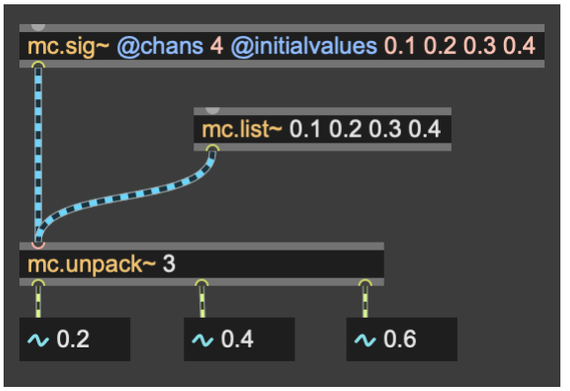


Auto-Adding Multi-Channel Signals

With single-channel patch cords, connecting two signals into a single object inlet will *add* the two signals before they are received by the object.

With multi-channel patch cords, this auto-adding also occurs, but it is more complex when the incoming multi-channel patchcords contain different numbers of channels.

When multi-channel signal patchcords are connected to a single inlet, the resulting signal will contain the number of channels from the patch cord with the greatest number of signals.


mc.sig~ @chans 4 @initialvalues 0.1 0.2 0.3 0.4
mc.list~ 0.1 0.2 0.3 0.4
mc.unpack~ 3
0.2
0.4
0.6

MC Polyphony

Topics 967
Objects 967
Examples 967

Topics

You can learn more about implementing polyphony with MC by reviewing the following:

Polyphony Using mc.poly~

Polyphony With Multiple Patchers

MC Managed Polyphony

Objects

MC objects that are particularly useful for polyphony include:

- mc.target
- mc.targetlist
- mc.noteallocator~
- mc.voiceallocator~

Examples

- mc.poly~ Outputs - An example of some of the MC features of poly~

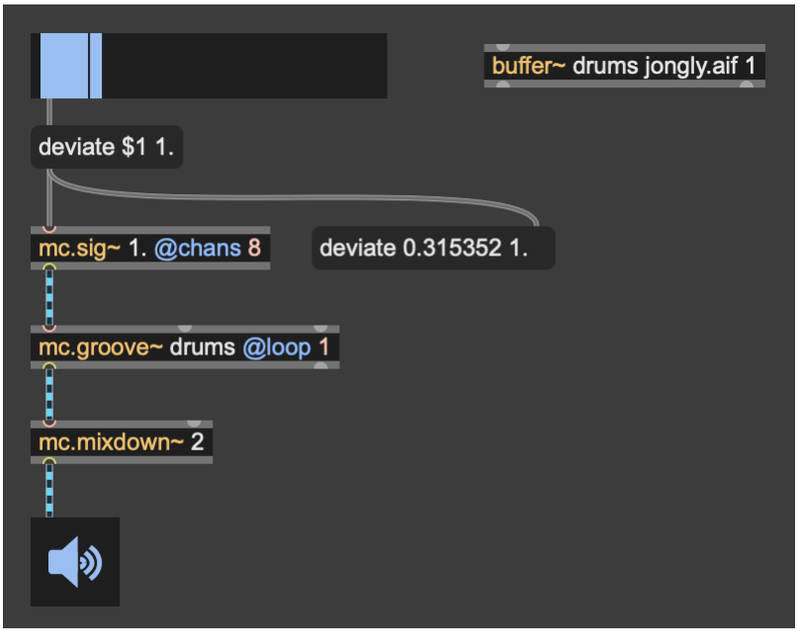
- MC Voice Numbers - An example targeting and receiving information from individual voices
- MC Polyphony Busymap - A example of the use of the busy map for voice allocation

MC Recording and Playback

Buffer Playback with mc.* Objects 969
Buffer Playback with mcs.* Objects 969
Recording 970

Buffer Playback with mc.* Objects

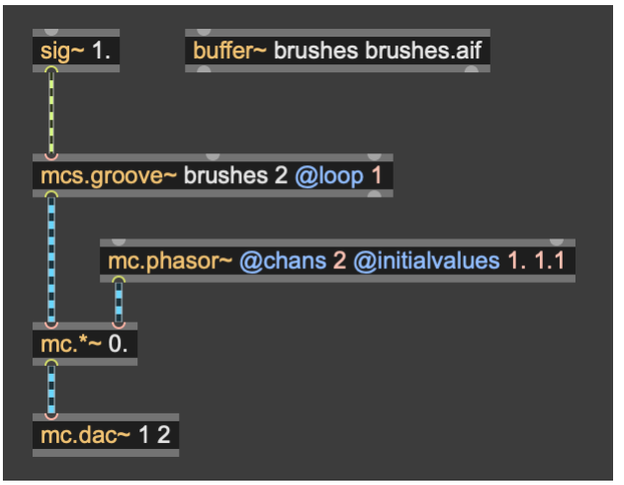
The **mc.*** versions of the objects create multiple parallel versions of the audio playback objects. By changing parameters to each of the devices, you can create complex results with relatively little patching. For example, you can use mc.groove~ modified by the `deviate` wrapper message to create a swarming effect.



Buffer Playback with mcs.* Objects

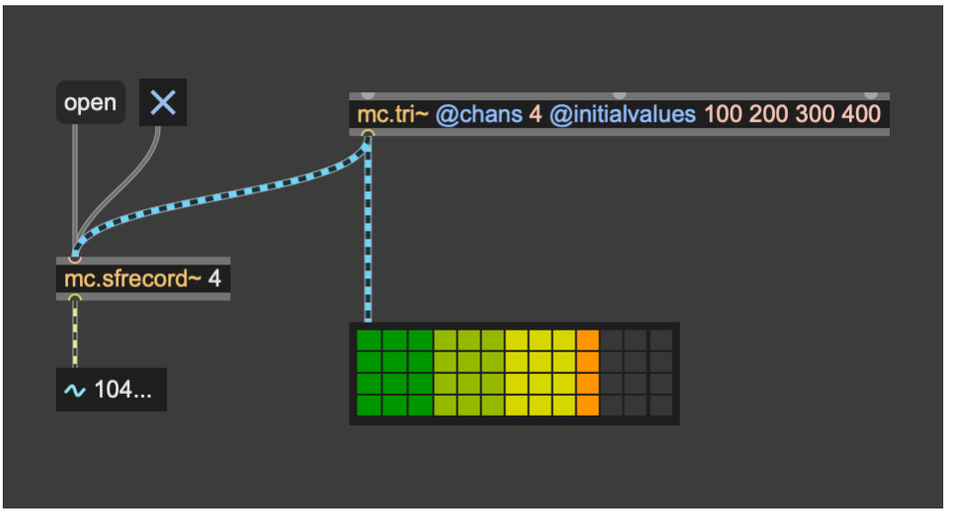
Using the **mcs.*** objects for buffer playback makes it easier to integrate audio streams with other MC objects. The mcs.* objects implement multichannel inlets and outlets that can be further manipulated with MC objects or routed to multi-speaker audio systems.

For example, we can use the multichannel output of the mcs.groove~ object as source material for a dual-channel fade-in patch. Since the multichannel audio is directly manipulated by the *mc.*~* object, a multichannel signal will control the output levels of our file player.



Recording

Use the mc.sfrecord~ object to record a multi-channel signal directly to disk.



You can also record into a multichannel buffer~ object using the mc.record~ object.

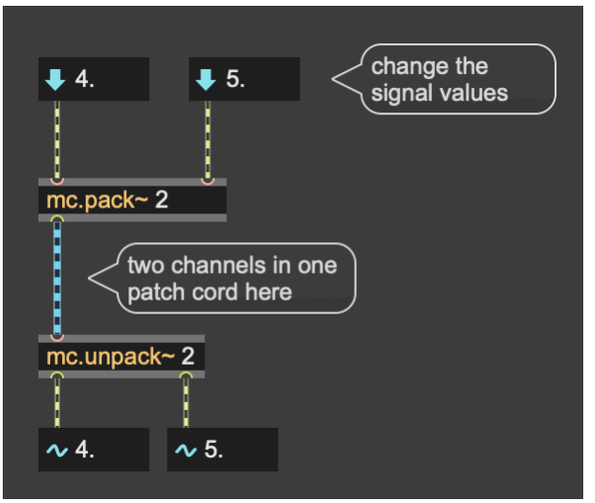
MC Signal Manipulation Objects

Creating Multi-Channel Signals 971
Separating Multi-Channel Signals into Single Channels 972
Combining and Separating Multi-Channel Signals 972
Adding and Removing Channels 973
Transforming Signals 974

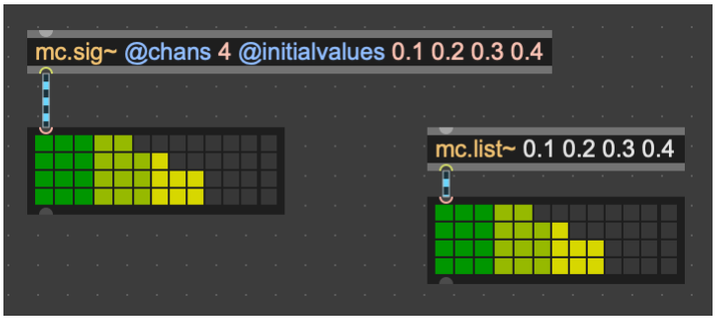
MC includes a number of objects that are useful for combining, separating, and transforming multichannel signals. These objects do not use the MC Wrapper but are useful in conjunction with wrapper-based objects.

Creating Multi-Channel Signals

mc.pack~ accepts numbers, single-channel signals, or multichannel signals and produces a multichannel signal with a designated number of outputs. This can be used to group separate single-channel sources into a single multichannel patch cord.



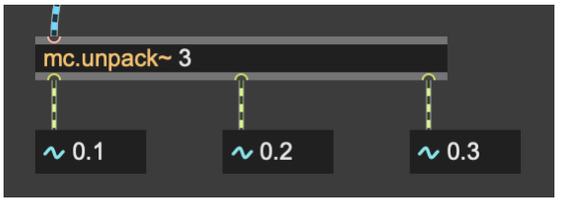
If you want to create a multichannel signal from numbers, you can also use the wrapper-based object mc.sig~ or the even simpler mc.list~.



Separating Multi-Channel Signals into Single Channels

To separate a multichannel signal into one or more individual signal outputs, use mc.unpack~. The argument to mc.unpack~ determines the number of individual signal outlets.

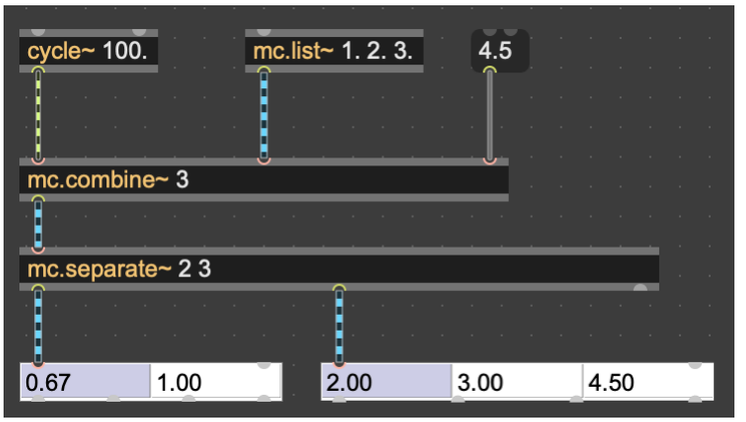
If the multi-channel input to mc.unpack~ contains fewer channels than the number of outlets, the extra outlets will produce a zero signal. If the multichannel input contains more channels than the number of outlets, the additional input channels are ignored.



Combining and Separating Multi-Channel Signals

If you have several multi-channel signals you would like to group into a single multichannel signal, use the mc.combine~ object. mc.combine~ produces an output multichannel signal containing the total number of input channels in the inputs. The argument to mc.combine~ specifies the number of inputs.

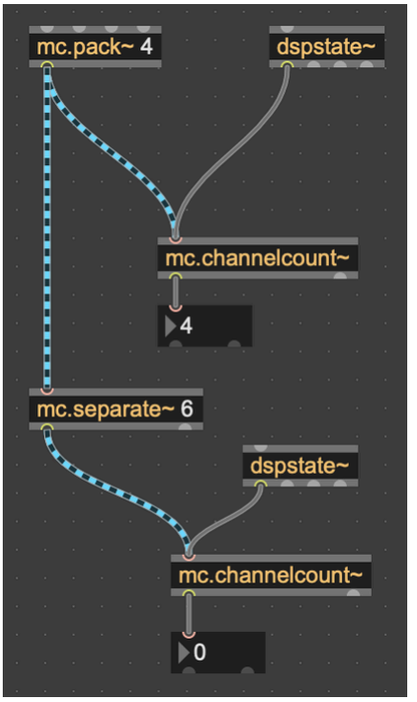
To separate a multichannel signal into two or more multichannel signals, use mc.separate~. Arguments to mc.separate~ specify the channel counts in its output signals.



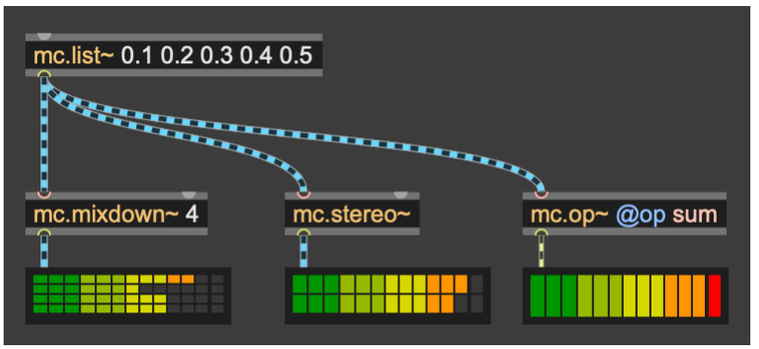
Adding and Removing Channels

To make a multi-channel signal that copies a single-channel input, use mc.dup~. The argument specifies the number of copies produced.

To force a multichannel output to have a set number of channels, use mc.separate~ with one argument specifying that number of channels. Additional channels are split off to the right outlet, but you don't need to connect that outlet to anything. If there are fewer channels in the input than you specify in the argument, the output will be padded with zero signals.



To mix all channels of a multichannel signal to fewer channels, use mc.mixdown~. mc.stereo~ is a stereo-specific version of the mc.mixdown~ object. A single mixed channel can be obtained by using mc.op~ with the `@op` attribute set to `sum`.



Transforming Signals

MC includes several objects that you can use to change how channels within multi-channel patch cords are organized, including mc.interleave~, mc.deinterleave~, and mc.transpose~. Some applications of these objects are described in MC Channel Topology.

MC Spatialization

Topics 975
Objects 975
Examples 976

One of the applications for which MC is especially adept is working with multi-speaker systems and spatialization. Multichannel source material can be transported to objects and manipulated as a single unit, greatly reducing the amount of patching required to work with large sets of audio routings and outputs.

Topics

You can learn more about issues related to spatialization by reviewing the following:

MC and Max for Live

MC Channel Topology

MC Recording and Playback

MC Dynamic Routing

Objects

In addition to the mc.* and mcs.* objects that perform multichannel audio manipulation, objects that are particularly useful for spatialization include:

- mc.dac~

- mc.adc~
- mc.plugin~
- mc.plugout~
- mc.combine~
- mc.separate~
- mc.interleave~
- mc.deinterleave~
- mc.transpose~
- mc.mixdown~

Examples

- MC Granular - A 16-channel granular synthesizer mixed to two channels
- Max for Live Channels - Output mapping for Max for Live

MC Visualization and Probing

Selecting a Display Channel 977
Signal Probing 978

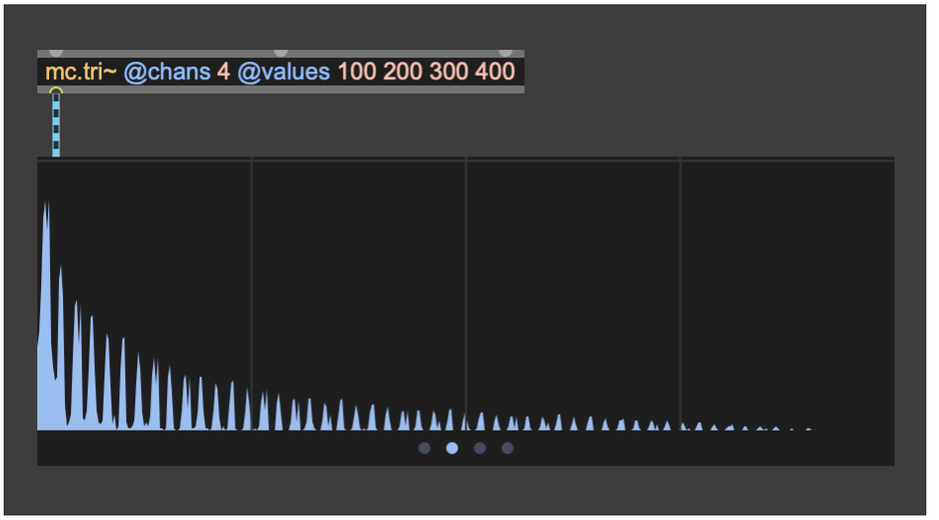
Multichannel signals can be visualized using the following objects:

- meter~ - standard LED-like metering
- levelmeter~ - VU metering
- number~ - displays the numerical value a signal
- scope~ - oscilliscope-like signal display
- spectroscope~ - displays the spectral content of a signal

Selecting a Display Channel

The number~, levelmeter~, scope~ and spectroscope~ objects will adapt to show all multichannel signals, but will only display or foreground one channel at a time. Use the *channel display selector* to bring one of the channels into focus.

- Click on one of the channel display selector indicators to switch to the chosen channel.



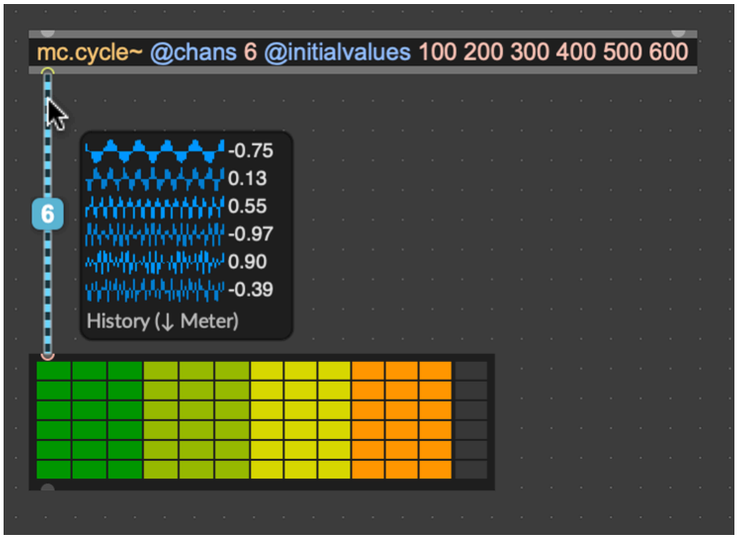
The scope~ and levelmeter~ objects have an *inactivealpha* attribute that controls the relative brightness of unselected channels.



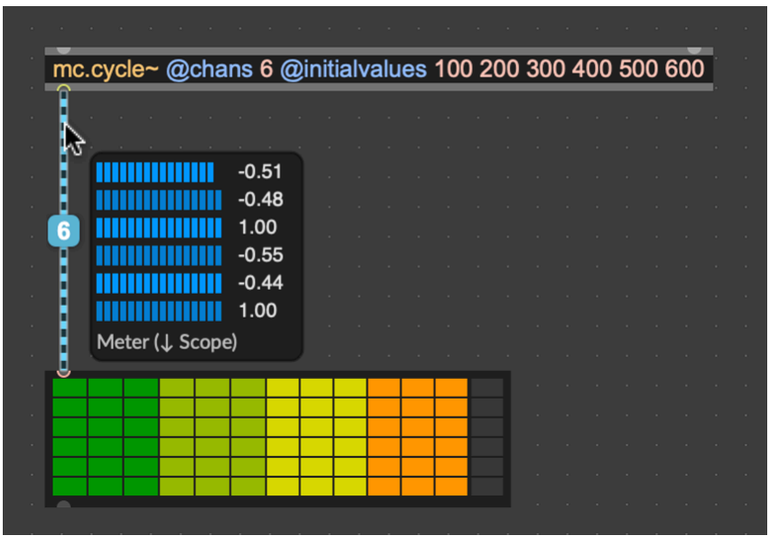
Signal Probing

The Signal Probe works with MC signals.

- Enable *Signal Probe* in the *Debug* menu
- Move the cursor over a multi-channel patch cord to view the signals it contains:



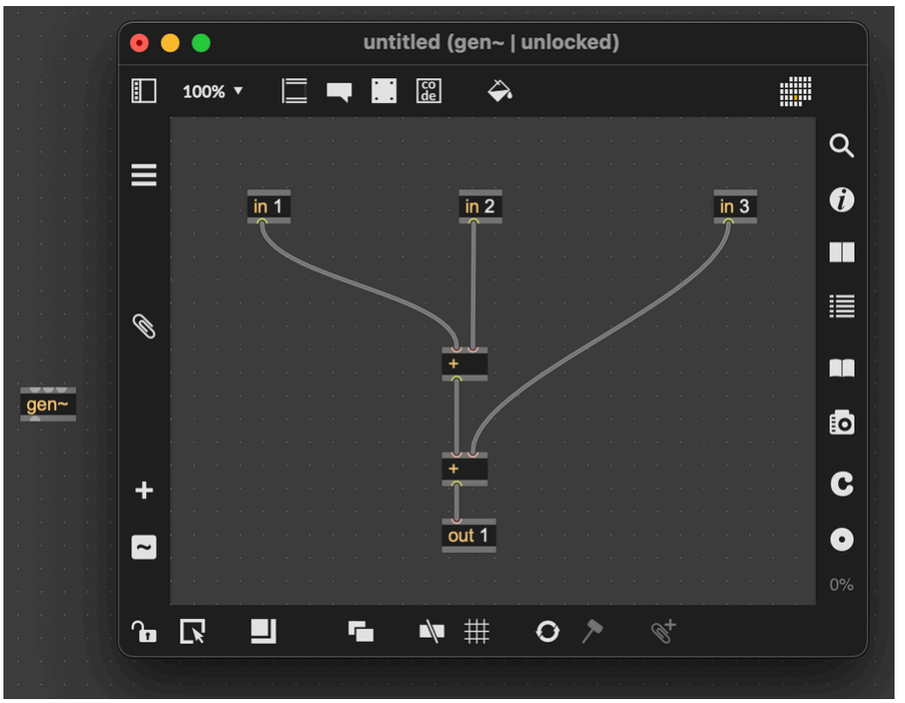
- Press the *down-arrow key* to switch to one of the alternative displays:



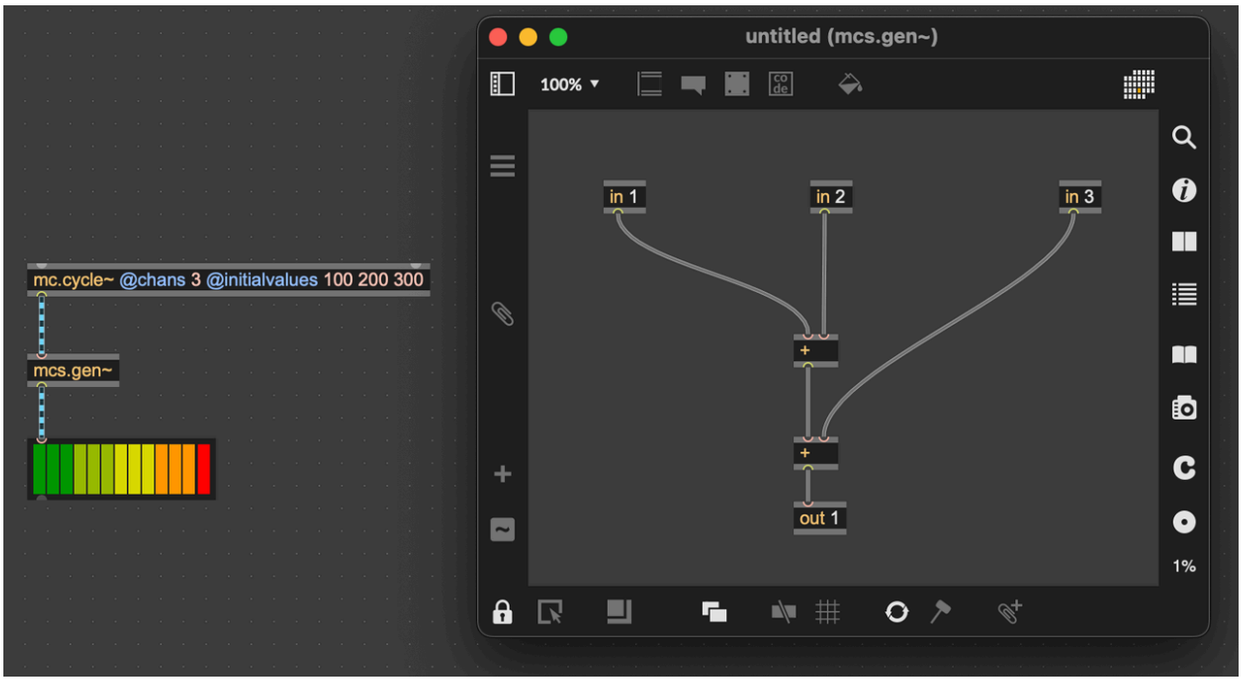
MC vs MCS Objects

As you work with MC you'll encounter a few objects beginning with **mcs**.

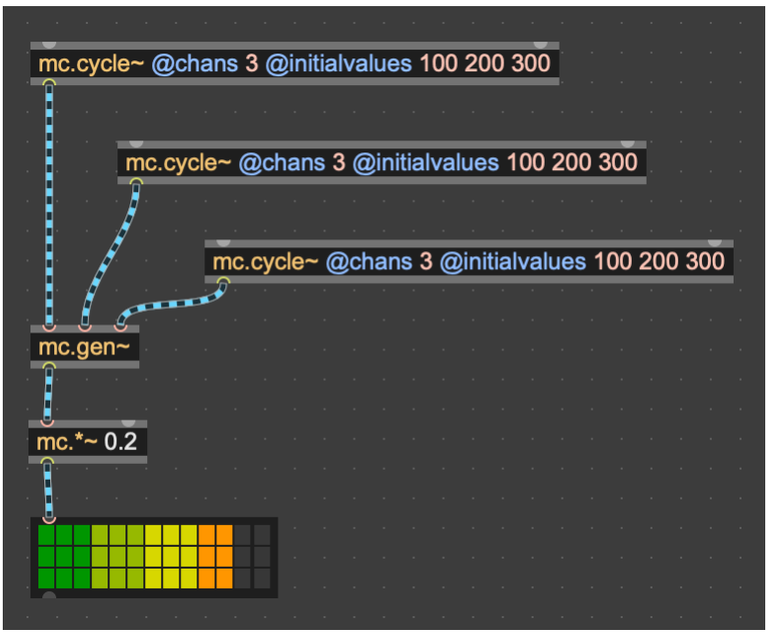
The "s" stands for **single**, meaning that an MCS object is a single instance with a single multi-channel input and a single multi-channel output. An example is mcs.gen~ : If you make a Gen patcher containing **in 1**, **in 2**, and **in 3** operators, a normal gen~ object would show three inlets:



The mcs.gen~ version of this patch would have only one multi-channel inlet. You can connect a multi-channel patch cord to this inlet and it will distribute the first three channels to the corresponding in operators. Likewise, multiple outlets are provided as a single multi-channel outlet.



By contrast, if you made an mc.gen~ with this same Gen patcher, you would see an object with three inlets and two outlets. Each of these inlets corresponds to the in operator within the Gen patcher, but with mc.gen~ there are multiple instances of the Gen patcher running inside the MC wrapper, so multichannel signals connected to these inlets are distributed to each Gen instance.



Another group of MCS objects deal with multichannel buffer~ data. If you create a buffer~ object with four channels of audio data, the mcs.play~ object will combine all four output channels of this audio data into one multichannel output. By contrast, mc.play~ lets you create multiple "players" of the same data in the MC Wrapper.

In deciding whether you want the MC or MCS version of play~, ask yourself whether you want to play one copy of a sound with multiple channels, or several copies of the same sound you can control independently.

Messages to the MC Wrapper

Setting a Number of Object Instances Within the Wrapper 983
Changing the @chans Attribute 983
Setting Initial Values of Instances Within the Wrapper 984
Sending Messages to All Instances Within the Wrapper 984
Sending Messages to a Specific Instance Within the Wrapper 985
Applying a List of Values to Successive Instances Within the Wrapper 986
Applying a Sequence of Values to Successive Instances Within the Wrapper 987

Most MC objects exist inside the MC Wrapper. The wrapper holds one or more instances of an object. Using features specific to the wrapper, you can address all instances at once or target specific instances.

Finally, a powerful set of wrapper messages permit simultaneous high-level control of all object instances.

Setting a Number of Object Instances Within the Wrapper

The `@chans` attribute sets the count of instances of an object `xxx~` within a wrapper object `mc.xxx~`. If the `chans` attribute has a value other than 0, the instance count is fixed and does not *auto-adapt* to the maximum number of channels in any multichannel signal connected to its inlets. Typically `@chans` is a typed-in argument to a wrapper object.

Changing the @chans Attribute

While the `@chans` attribute can be changed during the lifetime of an object, the actual multichannel outputs of the object will not update to reflect the new channel count until the audio is restarted.

In this example, if the `chans 4` message is sent to the mc.*~ object while the audio is running, the output will not change. But if you turn the audio off and on again after having sent the `@chans 4`

message, it will send a four-channel signal to the mc.unpack~ meaning the four right outputs of mc.unpack~ will change to 0.



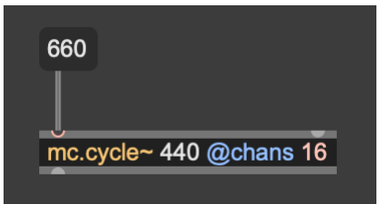
Setting Initial Values of Instances Within the Wrapper

If you simply type `mc.cycle~ 440` all cycle~ objects within the wrapper will have the initial frequency value of 440. To assign different initial values to wrapper instances, use the `@values` typed-in attribute. For example, `mc.cycle~ @chans 3 @values 440 660 880` would assign a frequency of 440 to the first cycle~ object, a frequency of 660 to the second cycle~ object, and a frequency of 880 to the third object.

You can combine both arguments if you want the first few objects to be set to different values but the rest set to a default value. For example, `mc.cycle~ 200 @chans 8 @values 440 660 880` would set the first three cycle~ objects to 440, 660, and 880 and the rest of the objects to 200.

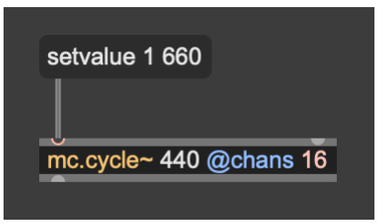
Sending Messages to All Instances Within the Wrapper

To send a message to all instances of an object within the wrapper, simply send the message. It will be sent to each instance. In this example, after clicking the message `660` connected to the mc.cycle~, all 16 instances will have the frequency 660.

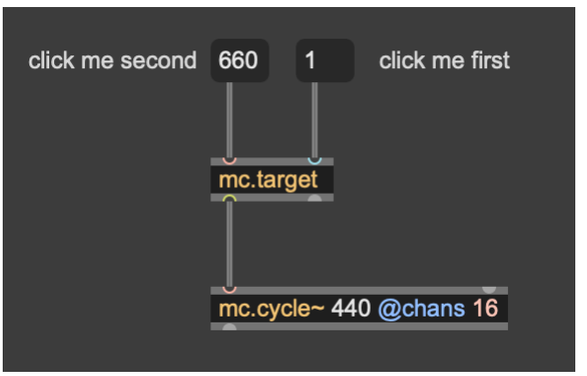


Sending Messages to a Specific Instance Within the Wrapper

To send a message to a specific instance of an object within the wrapper, precede the message by the word `setvalue` followed by the instance number starting at 1. For example, after clicking the message `setvalue 1 660` connected to the mc.cycle~ shown below, the first instance will have a frequency of 660 while the remaining 15 instances will retain their initial frequency of 440.



You can also target specific instances with the mc.target object. Sending any message into the left inlet of mc.target outputs that message preceded by `setvalue` and the current channel (voice) number. Here is the same example as above. First the message `1` is sent to the right inlet of mc.target, then the `660` is sent. The mc.target object produces the message `setvalue 1 660`.

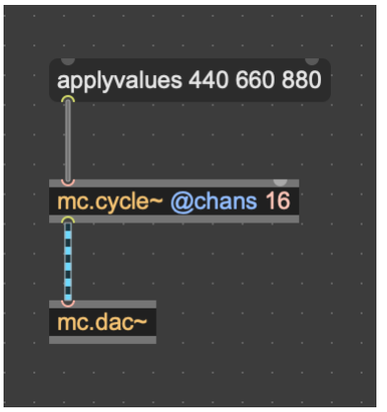


Many event-based objects in MC follow a convention of sending a voice number out a rightmost outlet immediately before a message out a different outlet, making it possible to use mc.target to

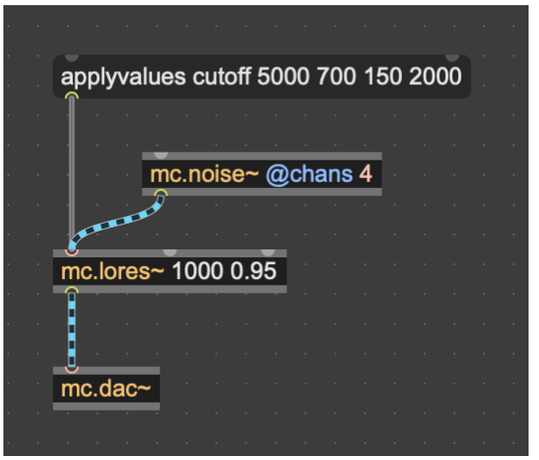
control specific object instances with these events.

Applying a List of Values to Successive Instances Within the Wrapper

Use the `applyvalues` message a send single value within a list to successive instances within the wrapper. In this example, after clicking the `applyvalues 440 660 880` message box, the first instance of cycle~ within the mc.cycle~ object will have a value of 440, the second will have a value of 660 and the third will have a value of 880.



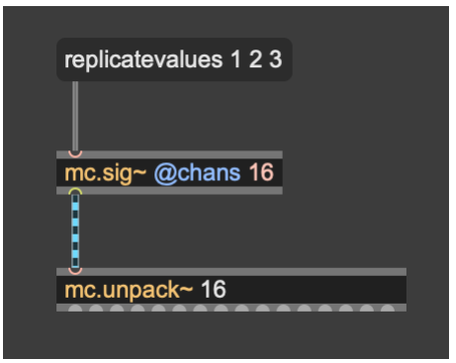
In addition to applying numerical values, `applyvalues` can set single-valued message arguments and attribute values. In this example, the `@cutoff` attribute of each lores~ instance is set to a different value based on the arguments to the `applyvalues` message sent to mc.lores~.



When showing the inspector an MC Wrapper object, the values shown for an object's attributes are for the first instance only along with wrapper-specific attributes such as `chans`.

Applying a Sequence of Values to Successive Instances Within the Wrapper

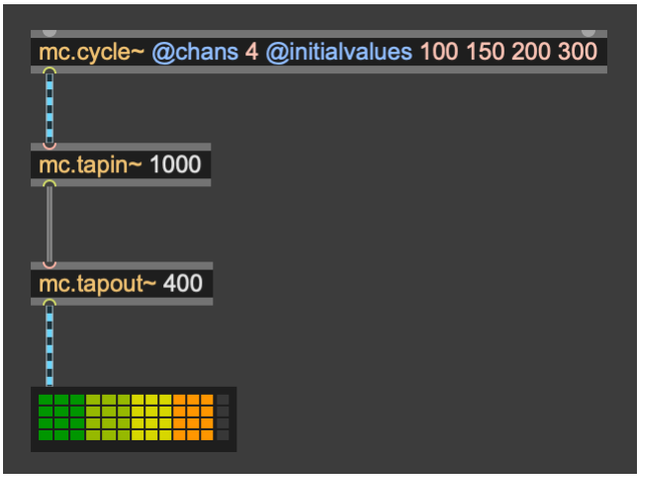
To apply a repeating sequence of values to all wrapper instances, use the `replicatevalues` message. With one argument, `replicatevalues` is the same as simply sending that value to all instances. With two or more arguments, `replicatevalues` cycles through the arguments applying them to each instance in succession. For example, clicking the `replicatevalues 1 2 3` message in this example will set all values of mc.sig~ to a repeating cycle of 1, 2, and 3.



Multichannel Delay Systems

Multichannel Delay Time Control 988
Multichannel Feedback Control 989

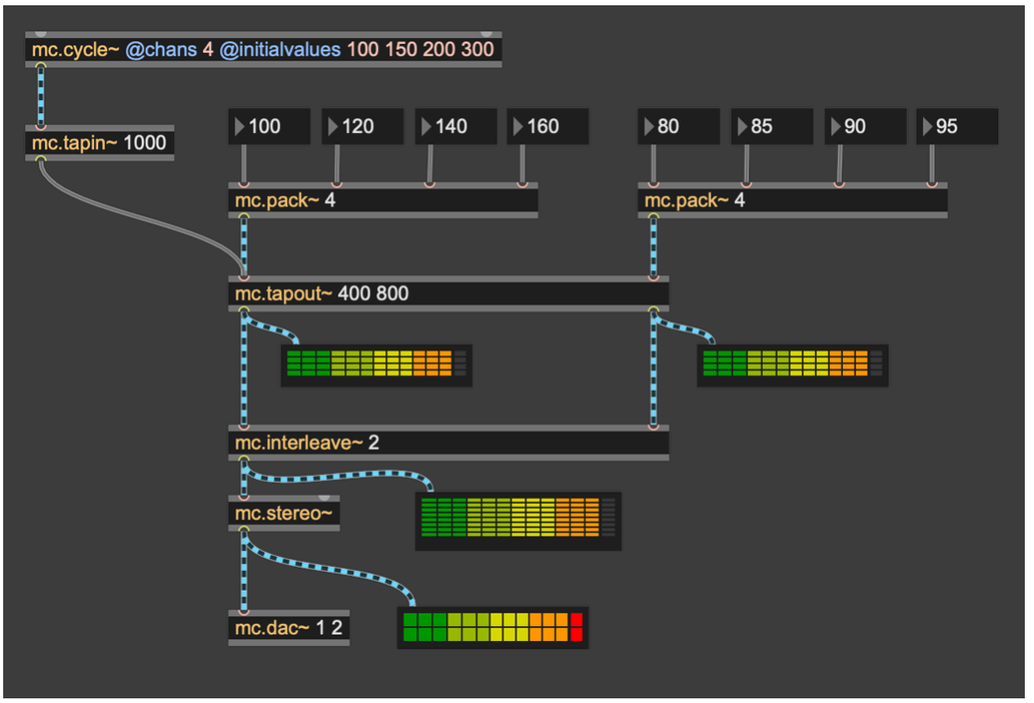
The mc.tapin~ and mc.tapout~ objects let you build networks of multichannel delay lines. The mc.tapin~ object auto-adapts to the number of channels in the multichannel patch cord connected to its input, creating individual delay memories for each input channel. Connected mc.tapout~ objects create one or more multichannel taps of this multichannel delay line.



Multichannel Delay Time Control

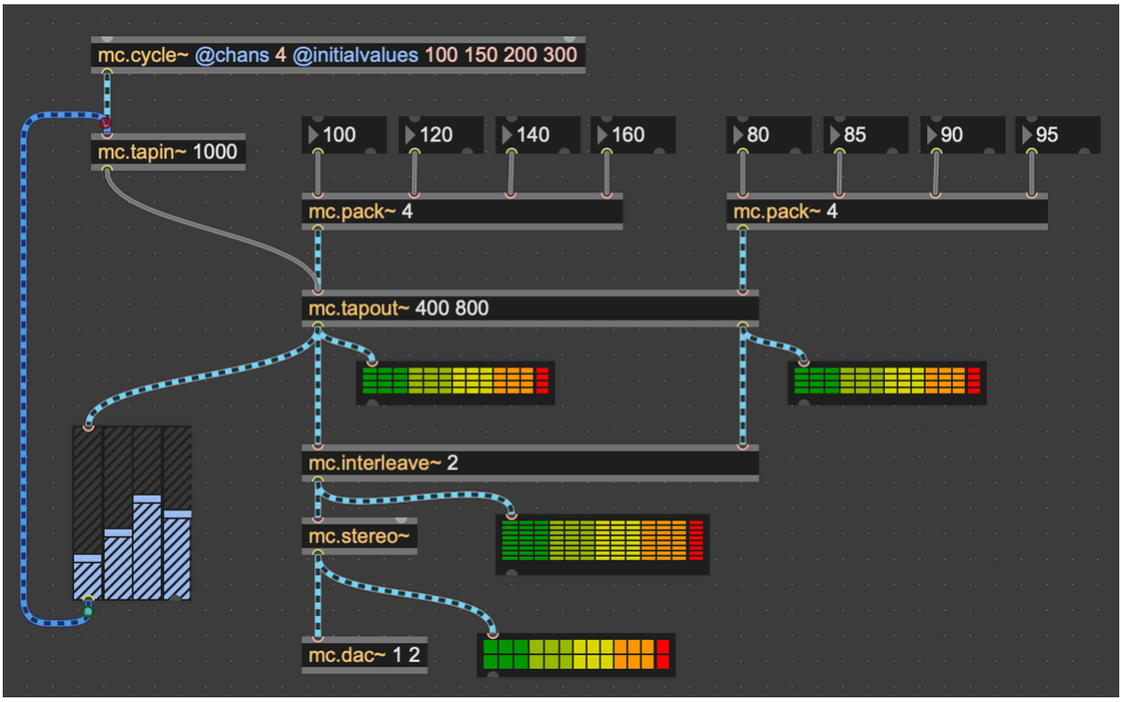
Using multichannel signals fed to the input(s) of mc.tapout~ that represent delay times, you can control the individual delay times of each of the channels in each multichannel tap.

Here is an example where we use a four-channel delay line with two output taps. Each tap has a four-channel multichannel signal controlling its delay times. The result is eight outputs, each with its own unique delay time.



Multichannel Feedback Control

You can feed the signals from mc.tapout~ back into the input of mc.tapin~ to create multichannel feedback delays. By multiplying the output gains with mc.multigain~ you can control the feedback levels of each channel individually.



Note that the multichannel inputs to mc.tapin~ are automatically added to the audio input coming from the mc.cycle~.

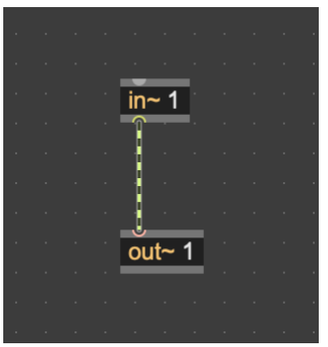
Polyphony Using mc.poly~

The poly~ object manages polyphony for multiple instances of a patcher. The poly~ object always *copies* audio inputs to each patcher instance, and *mixes* the output of each patcher instance together at the object's outlets.

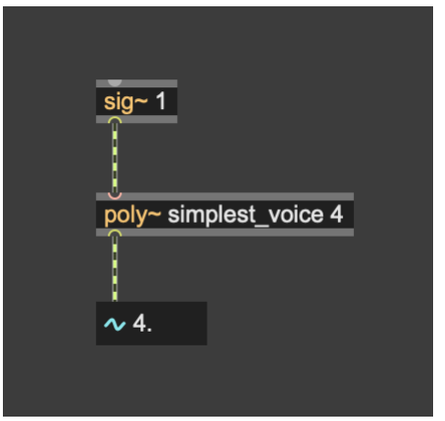
The mc.poly~ object uses multichannel signals to operate in a more flexible way. A multichannel input to mc.poly~ assigns the audio for each channel to its corresponding patcher instance (poly~ voice). The first audio input channel in the signal is passed to an in~ object within the first patcher instance, the second audio input channel in the signal is passed to an in~ object in the second patcher instance, and so on.

Similarly, the output fed to out~ objects in a patcher inside mc.poly~ is not mixed with the other patchers, it comes directly out to the corresponding channel of a multichannel signal.

Here is a patcher that simply passes audio input arriving at an in~ directly an out~:

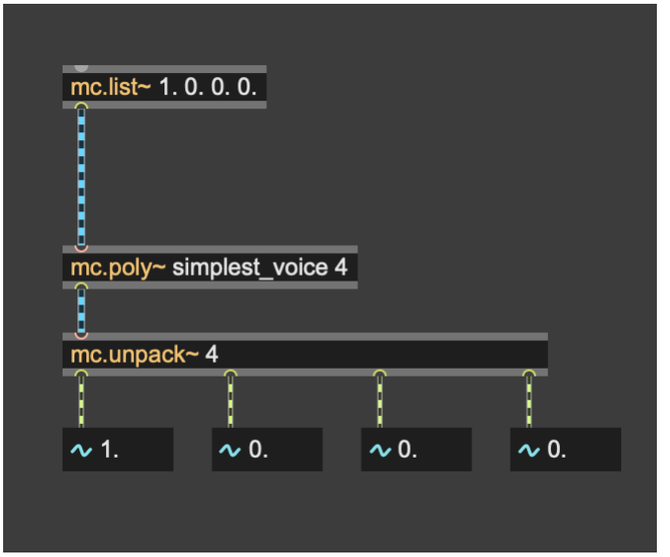


Here is a comparison of poly~ and mc.poly~ using simple numbers as input to four instances of this patcher loaded in each object. For the poly~ case, we feed a single-channel signal of 1 to the object and it produces a single-channel output:



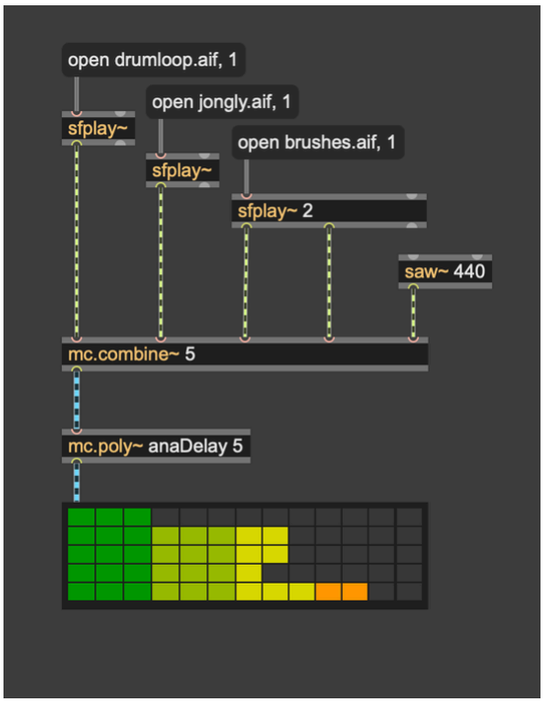
Note that the output is 4, representing the sum of all the patchers mixing their inputs together.

For the mc.poly~ case, we will feed in a four-channel multichannel signal consisting of 1 in the first channel and 0 in the other three channels.



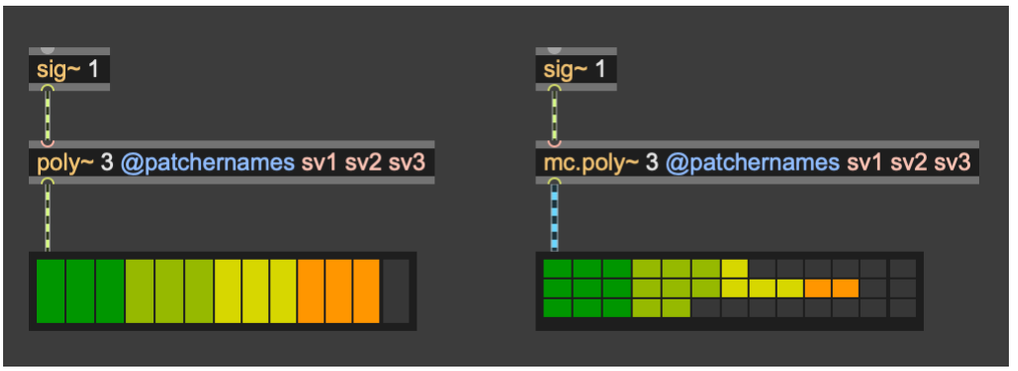
In this case, the output multichannel signal is equal to the input multichannel signal.

One use of mc.poly~ would be to load a simple patcher to use as an audio effect operating in parallel on each channel of a multichannel signal. If you want to mix the output of all the effects, you can do that later with mc.mixdown~ or mc.op~.

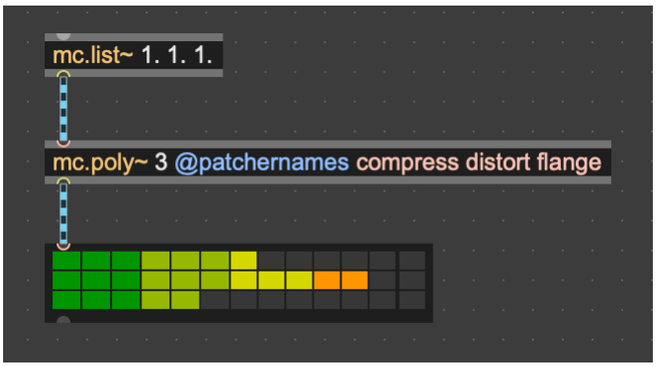

open drumloop.aif, 1
open jongly.aif, 1
sfplay~
open brushes.aif, 1
sfplay~
sfplay~ 2
saw~ 440
mc.combine~ 5
mc.poly~ anaDelay 5

Polyphony with Multiple Patchers

Both the poly~ and mc.poly~ objects can contain different patchers for each voice. The names of the voice patches are set using the `@patchernames` attribute.



You can use mc.poly~ to load a bank of patchers to use as audio effects operating in parallel on each channel of a multichannel signal. If you want to mix the output of all the effects, you can do that later with mc.mixdown~ or mc.op~.

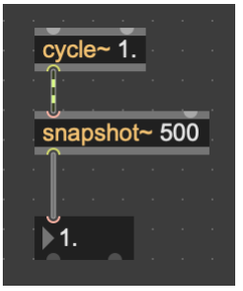


Processing Events from MC Objects

Events and MC Values 995
Voice outputs from poly~ and mc.poly~ 996
See Also 997

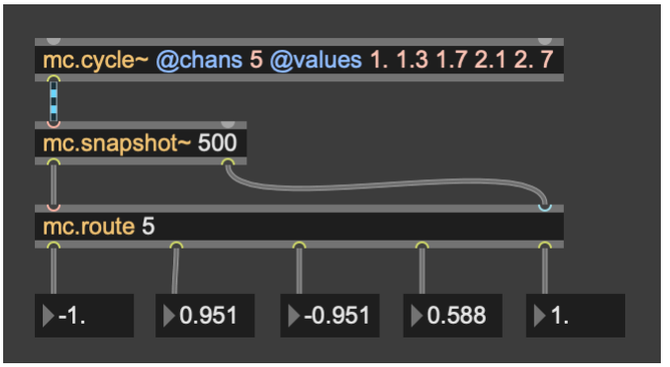
Events and MC Values

It is often useful to obtain values from an audio signal and use them in event-level processing. With standard MSP signals, you can use the snapshot~ object to get an instantaneous value from an audio signal at a regular interval.



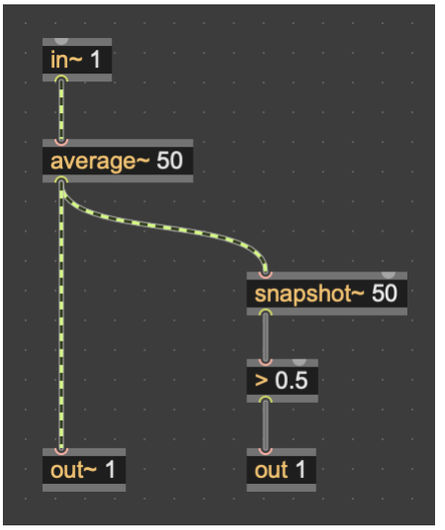
In the case of multi-channel signals, things are more complex: at each measurement, you would get a list with a value for each input channel. Instead of producing a list, the mc.snapshot~ object has an additional outlet that provides a **voice number**.

At each sampling interval, mc.snapshot~ outputs each of the values from each of the incoming samples, but outputs the voice number before the signal value. This combination of values can be used with a routing object (like mc.route) to send each value to a unique location. Alternatively, if you do want a list of all the snapshot values, mc.makelist can do that for you.

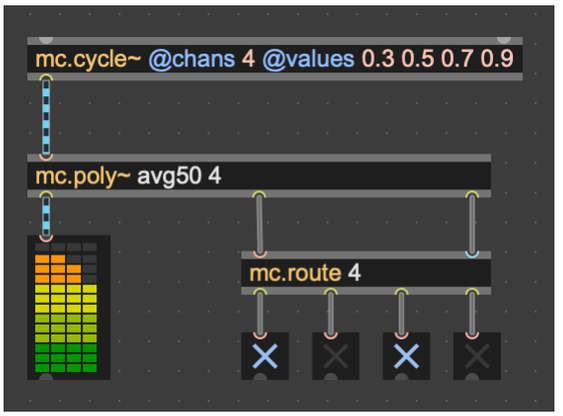


Voice outputs from poly~ and mc.poly~

Similar to the mc.snapshot~ object, both poly~ and mc.poly~ may have an additional voice output. This is only created when contained in a poly~ contains an event (non-signal) outlet.



You can use the voice output, along with a routing object such as mc.route to use the event output to control other parts of your patch.



See Also

MC Event-Based Objects

Using mc.gen with the MC Wrapper

mc.gen Connections 998
Using MC Operators 999
Target Connections 999

The MC Wrapper contains a set of messages that offer high-level algorithmic control over a set of individual objects. For example, using the `deviate` message with changing values for ranges you can generate and apply a set of random values for signal object parameters such as oscillator frequency.

For more possibilities than those offered by the basic wrapper messages, the mc.gen~ object offers a way to create algorithms for controlling objects in the wrapper. As an example, we will show an example using the `@expr` mode of Gen to create a simple range.

(Note: This feature is already available as the `spread` message to the wrapper but it illustrates a basic pattern you can customize.)

mc.gen Connections

The mc.gen object is not an audio object; it accepts and produces only Max messages. However, mc.gen uses the MC Wrapper; it contains multiple gen objects. Specify the number inputs using the @chans attribute.

Since mc.gen uses the MC Wrapper, you can connect an audio signal to one of its inputs and it will auto-adapt its channel count. (We will use this trick in our complete example below.)

The mc.gen object contains an extra rightmost outlet that outputs a voice number immediately before any values come out its other outlets. This permits identification of the gen instance that is sending output. The mc.target, mc.route, and mc.makelist objects make it simple to use voice outlet for further isolation of mc.gen output.

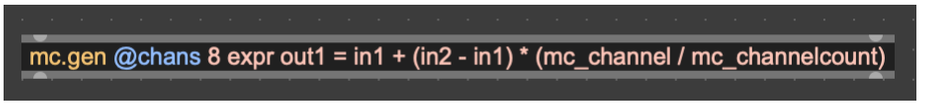
Using MC Operators

To generate a range of control values, one for each wrapper instance, we will need to use the MC-specific mc_channel and mc_channelcount operators in our Gen expression. A formula for generating the range is:

```
out_channel = min + (max - min) * (channel / number_of_channels)
```

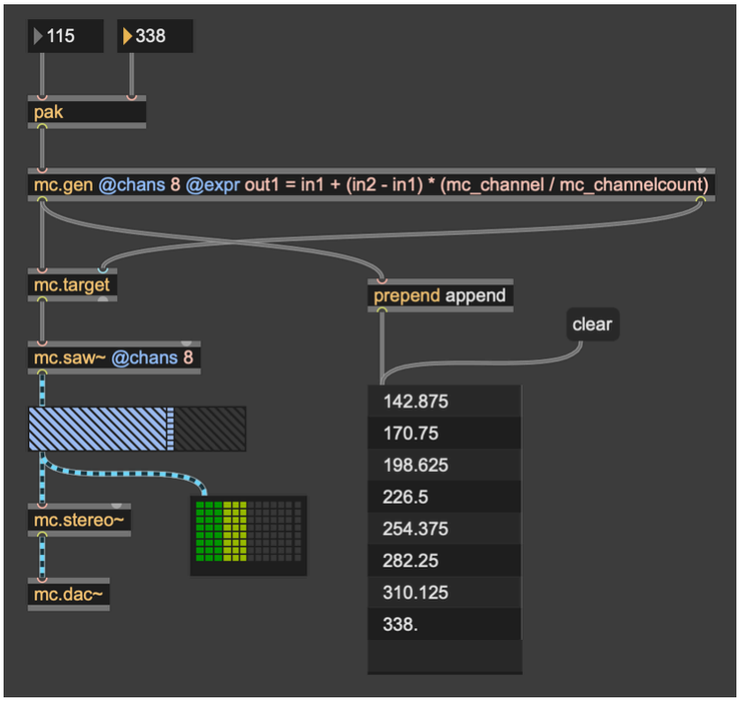
This will scale a range evenly over a space defined by min and max. We will use `in1` as our range minimum and `in2` as our range maximum. As a Gen expression, this would be:

```
out1 = in1 + (in2 - in1) \* (mc_channel / mc_channelcount)
```



Target Connections

Now we want to use the rightmost **voice** outlet of mc.gen in conjunction with mc.target to control an object in the MC Wrapper. In this example, we will control the frequencies of a bank of sawtooth oscillators in mc.saw~.



By connecting the rightmost outlet of mc.gen to mc.target, the per-voice range value is properly routed to the correct saw~ instance inside mc.saw~.

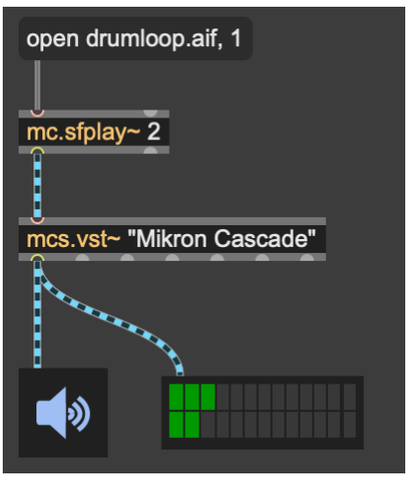
Using Plug-ins with MC

Single Plug-in, Multi-Channel Signals 1001
Multiple Copies of a Plug-in 1001
Accessing Parameters of Individual Plug-in Instances 1003
Working with Max for Live Devices 1004

In MC, you can work with multiple copies of a plug-in to produce multiple channels of audio, or use multi-channel inputs and output with a single copy of the plug-in.

Single Plug-in, Multi-Channel Signals

The mcs.vst~ object allows you to work with a single instance of a plug-in, but have multi-channel inputs and outputs. In the most basic case, you can create an mcs.vst~ with a favorite stereo VST effect, and have a 2-channel MC signal routed into and out of the effect.



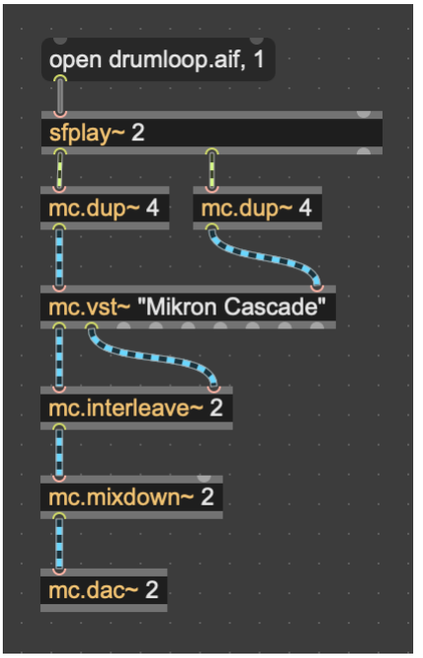
If the plug-in is capable of more than two input or output channels, you can define the input and output channel counts with the first two arguments to mcs.vst~.

Multiple Copies of a Plug-in

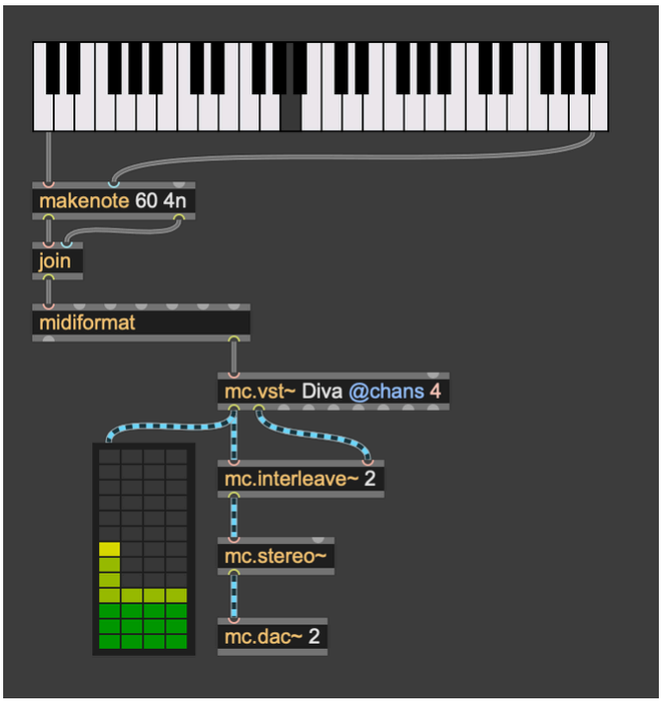
The mc.vst~ object works like other MC Wrapper objects: it contains multiple instances of a plug-in within a single Max object. The number of instances will auto-adapt to the number of inputs.

The number of inputs and outputs to the plug-in are defined by the channel argument(s) to mc.vst~, and each of the inlets and outlets will be a multichannel signal.

For example, a mc.vst~ defined to have two I/O channels and four instances will produce two four-channel signals containing the "left" and "right" inputs to the four plug-in instances.



When using a VST instrument plug-in, you will need to define the number of MC output channels to be used, preferably as a typed-in `@chans` argument, since you won't be sending the plug-in any audio inputs.

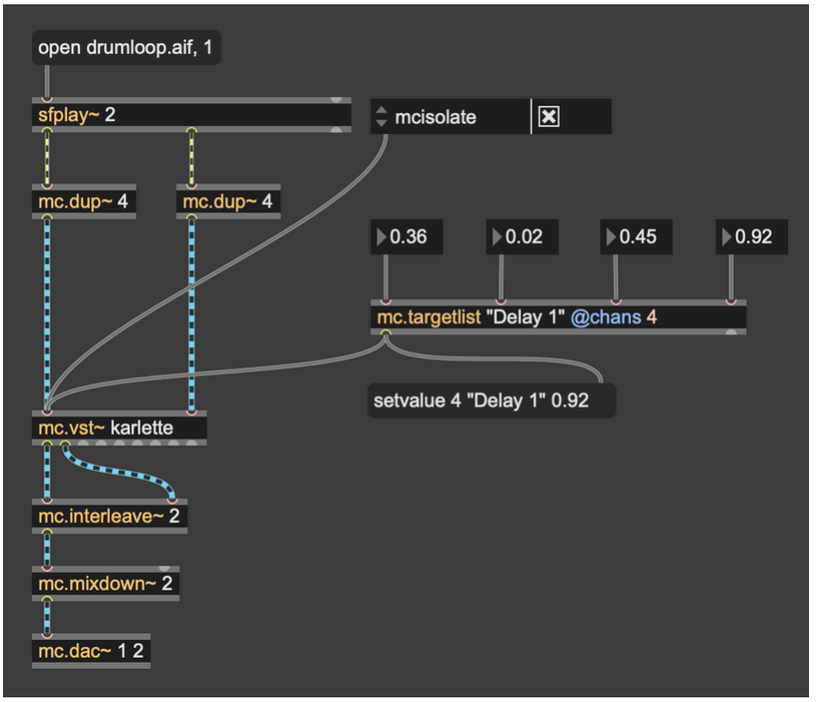


Accessing Parameters of Individual Plug-in Instances

When using mc.vst~ to host multiple copies of the same plug-in, you may want to change the parameters of each copy to different settings. This can be done using the `mcisolate` attribute of the mc.vst~ object, as well as the mc.target and mc.targetlist objects to route messages to a single instance of the plug-in.

To alter individual instances of a plug-in's parameters:

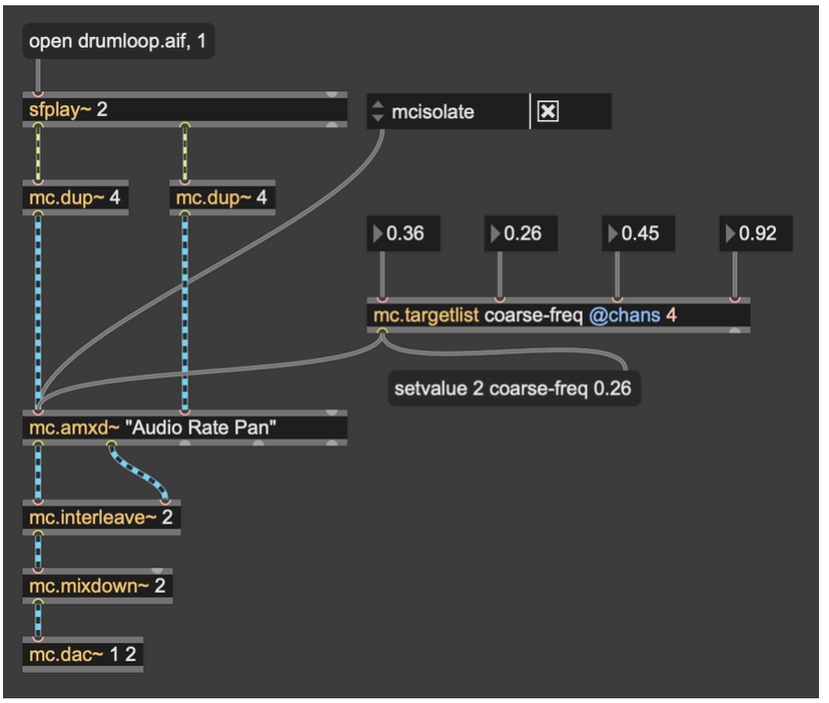
- Enable the `mcisolate` attribute of mc.vst~ using a typed-in argument, message box, or attrui.
- Using mc.target or mc.targetlist, send a `setvalue` message followed by the instance number and desired parameter name to mc.vst~. This will change the parameter for the targeted instance only.
- To target all instances of the mc.vst~ object, you can target using voice number 0 (zero), or disable the `mcisolate` attribute to distribute the change to all instances of the plug-in.



When `mcisolate` is off, any parameter change for any voice - or any change to the user interface of the plug-in - will result in that parameter being changed in all of the instances of the plug-in.

Working with Max for Live Devices

Max for Live devices are hosted similarly to VST plug-ins, using the mcs.amxd~ object for a single instance, and mc.amxd~ for multiple-instance applications.



If you are showing the device's interface in your patcher (by toggling on the *Show View In Patcher* attribute), you can choose the instance of the device that is displayed by using the `displaychan` attribute.

Appendix B: Lua Extended

jit.gl.lua Color Bindings

Color Names 1008

```
hue (HSLA, factor)
```

Shift the hue part of an HSL[A] color by a factor- `HSLA` A table of HSL and optionally an A value

- `factor` The shifting factor

```
saturate (HSLA, factor)
```

Scale the saturation part of an HSL[A] color by a factor- `HSLA` A table of HSL and optionally an A value

- `factor` The scaling factor

```
lighten (HSLA, factor)
```

Scale the lightness part of an HSL[A] color by a factor- `HSLA` A table of HSL and optionally an A value

- `factor` The scaling factor

```
RGBtoHSL (...)
```

Convert RGB[A] [0, 1] to HSL[A] [0, 1]If the input has an alpha channel, it will be appended to output

- `...` A table or unpacked list of RGB[A] values

```
HSLtoRGB (...)
```

Convert HSL[A] [0, 1] to RGB[A] [0, 1]If the input has an alpha channel, it will be appended to the output

- `...` A table or unpacked list of HSL[A] values

Color Names

	`aliceblue`
	`antiquewhite`
-	----------
	`aquamarine`
-	----------
	`azure`
-	-------
	`beige`
-	-------

	`bisque`
-	--------
	`black`
-	-------
	`blanchedalmond`
-	----------
	`blueviolet`
-	----------
	`brown`
-	-------
	`burlywood`
-	----------
	`chartreuse`
-	----------
	`chocolate`
-	----------
	`coral`
-	-------
	`cornflowerblue`
-	----------
	`cornsilk`
-	----------
	`crimson`
-	---------

	`darkgoldenrod`
-	--------------
	`darkgray`
-	----------
	`darkgrey`
-	----------
	`darkkhaki`
-	----------
	`darkmagenta`
-	----------
	`darkorange`
-	----------
	`darkorchid`
-	----------
	`darkred`
-	---------
	`darksalmon`
-	----------
	`darkseagreen`
-	----------
	`darkviolet`
-	----------
	`deeppink`
-	----------

	`dimgray`
-	---
	`dimgrey`
-	---
	`firebrick`
-	---
	`floralwhite`
-	---
	`fuchsia`
-	---
	`gainsboro`
-	---
	`ghostwhite`
-	---
	`gold`
-	---
	`goldenrod`
-	---
	`gray`
-	---
	`grey`
-	---
	`greenyellow`
-	---

	`honeydew`
-	----------
	`hotpink`
-	---------
	`indianred`
-	----------
	`ivory`
-	-------
	`khaki`
-	-------
	`lavender`
-	----------
	`lavenderblush`
-	--------------
	`lawngreen`
-	----------
	`lemonchiffon`
-	-------------
	`lightblue`
-	----------
	`lightcoral`
-	-----------
	`lightcyan`
-	----------

	`lightgoldenrodyellow`
-	----------
	`lightgray`
-	----------
	`lightgreen`
-	----------
	`lightgrey`
-	----------
	`lightpink`
-	----------
	`lightsalmon`
-	----------
	`lightskyblue`
-	----------
	`lightslategray`
-	----------
	`lightslategrey`
-	----------
	`lightsteelblue`
-	----------
	`lightyellow`
-	----------
	`linen`
-	----------

	`magenta`
-	---
	`maroon`
-	---
	`mediumaquamarine`
-	---
	`mediumorchid`
-	---
	`mediumpurple`
-	---
	`mediumslateblue`
-	---
	`mediumvioletred`
-	---
	`mintcream`
-	---
	`mistyrose`
-	---
	`moccasin`
-	---
	`navajowhite`
-	---
	`oldlace`
-	---

	`olive`
-	-------
	`olivedrab`
-	-----------
	`orange`
-	--------
	`orangered`
-	-----------
	`orchid`
-	--------
	`palegoldenrod`
-	----------------
	`palegreen`
-	-----------
	`paleturquoise`
-	----------------
	`palevioletred`
-	----------------
	`papayawhip`
-	------------
	`peachpuff`
-	-----------
	`peru`
-	------

	`pink`
-	------
	`plum`
-	------
	`powderblue`
-	------------
	`purple`
-	--------
	`red`
-	-----
	`rosybrown`
-	----------
	`saddlebrown`
-	------------
	`salmon`
-	--------
	`sandybrown`
-	------------
	`seashell`
-	----------
	`sienna`
-	--------
	`silver`
-	--------

	`skyblue`
-	---
	`slateblue`
-	---
	`slategray`
-	---
	`slategrey`
-	---
	`snow`
-	---
	`tan`
-	---
	`thistle`
-	---
	`tomato`
-	---
	`violet`
-	---
	`wheat`
-	---
	`white`
-	---
	`whitesmoke`
-	---

	`yellow`
-	--------
	`yellowgreen`
-	------------

jit.gl.lua OpenGL Bindings

The OpenGL bindings for jit.gl.lua are found in the `opengl` module, which is built-in to the object. To access the bindings, simpply use the built-in Lua function, `require`, to load the module as follows:

```
-- load in the module and set some aliases local  gl = require( "opengl" )
  local  GL = gl
```

The idiom above allows us to write code that more closely resembles C-style OpenGL code. For the OpenGL Utility (GLU) functions see the For the standard OpenGL bindings, see the jit.gl.lua OpenGL GLU Bindings.

```
Accum (op, value)
```

Operate on the accumulation bufferOpenGL Documentation: glAccum

- `op` Specifies the accumulation buffer operation. Symbolic constants GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN are accepted.
- `value` Specifies a floating-point value used in the accumulation buffer operation. op determines how value is used.

```
AlphaFunc (func, ref)
```

Specify the alpha test functionOpenGL Documentation: glAlphaFunc

- `func` Specifies the alpha comparison function. Symbolic constants GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, and GL_ALWAYS are accepted. The initial value is GL_ALWAYS
- `ref` Specifies the reference value that incoming alpha values are compared to. This value is clamped to the range 0 1 , where 0 represents the lowest possible alpha value and 1 the highest possible value. The initial reference value is 0

```
Begin (mode)
```

Delimit the vertices of a primitive or a group of like primitivesOpenGL Documentation: glBegin

- `mode` Specifies the primitive or primitives that will be created from vertices presented between glBegin and the subsequent glEnd. Ten symbolic constants are accepted: GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON.

```
BlendFunc (sfactor, dfactor)
```

Specify pixel arithmeticOpenGL Documentation: glBlendFunc

- `sfactor` Specifies how the red, green, blue, and alpha source blending factors are computed. The following symbolic constants are accepted: GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_CONSTANT_COLOR, GL_ONE_MINUS_CONSTANT_COLOR, GL_CONSTANT_ALPHA, GL_ONE_MINUS_CONSTANT_ALPHA, and GL_SRC_ALPHA_SATURATE. The initial value is GL_ONE.
- `dfactor` Specifies how the red, green, blue, and alpha destination blending factors are computed. The same constants are accepted as for sfactor.

```
CallList (list)
```

Execute a display listOpenGL Documentation: glCallList

- `list` Specifies the integer name of the display list to be executed.

```
Clear (mask)
```

Clear buffers to preset valuesOpenGL Documentation: glClear

- `mask` Bitwise OR of masks that indicate the buffers to be cleared. The four masks are GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER_BIT, and GL_STENCIL_BUFFER_BIT.

```
ClearAccum (...)
```

Specify clear values for the accumulation bufferOpenGL Documentation: glClearAccum

- `...` Either a 4 numbers or a table specifying the red, green, blue, and alpha values used when the accumulation buffer is cleared. The initial values are all 0.

```
ClearColor (...)
```

Specify clear values for the color buffersOpenGL Documentation: glClearColor

- `...` Either a 4 numbers or a table specifying the red, green, blue, and alpha values used when the color buffers are cleared. The initial values are all 0.

```
ClearDepth (depth)
```

Specify the clear value for the depth bufferOpenGL Documentation: glClearDepth

- `depth` Specifies the depth value used when the depth buffer is cleared. The initial value is 1.

```
ClearIndex (c)
```

Specify the clear value for the color index buffersOpenGL Documentation: glClearIndex

- `c` Specifies the index used when the color index buffers are cleared. The initial value is 0.

```
ClearStencil (c)
```

Specify the clear value for the stencil bufferOpenGL Documentation: glClearStencil

- `c` Specifies the index used when the stencil buffer is cleared. The initial value is 0.

```
ClipPlane (plane, equation)
```

Specify a plane against which all geometry is clippedOpenGL Documentation: glClipPlane

- `plane` Specifies which clipping plane is being positioned. Symbolic names of the form GL_CLIP_PLANEi, where i is an integer between 0 and GL_MAX_CLIP_PLANES -1 , are accepted.
- `equation` Specifies the address of an array of four double-precision floating-point values. These values are interpreted as a plane equation.

```
Color (...)
```

Set the current colorOpenGL Documentation: glColor

- `...` Specify either 3 or 4 numbers or a table for new red, green, blue and [alpha] values for the current color. The default value is (0, 0, 0, 1).

```
ColorMask (...)
```

Enable and disable writing of frame buffer color componentsOpenGL Documentation: glColorMask

- `...` Either 4 numbers or a table specifying whether red, green, blue, and alpha can or cannot be written into the frame buffer. The initial values are all GL_TRUE, indicating that the color components can be written.

```
ColorMaterial (face, mode)
```

Cause a material color to track the current colorOpenGL Documentation: glColorMaterial

- `face` Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are GL_FRONT, GL_BACK, and

GL_FRONT_AND_BACK. The initial value is GL_FRONT_AND_BACK.

- `mode` Specifies which of several material parameters track the current color. Accepted values are GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, and GL_AMBIENT_AND_DIFFUSE. The initial value is GL_AMBIENT_AND_DIFFUSE.

```
CopyPixels (x, y, width, height, type)
```

Copy pixels in the frame bufferOpenGL Documentation: glCopyPixels

- `x` Specify the x window coordinate of the lower left corner of the rectangular region of pixels to be copied.
- `y` Specify the y window coordinate of the lower left corner of the rectangular region of pixels to be copied.
- `width` Specify the horizontal dimensions of the rectangular region of pixels to be copied. Both must be nonnegative.
- `height` Specify the vertical dimensions of the rectangular region of pixels to be copied. Both must be nonnegative.
- `type` Specifies whether color values, depth values, or stencil values are to be copied. Symbolic constants GL_COLOR, GL_DEPTH, and GL_STENCIL are accepted.

```
CopyTexImage (target, level, internalformat, x, y, width, [height], border)
```

Copy pixels into a 1D or 2D texture imageIf no height argument is given, glCopyTexImage1D will be used, otherwise, glCopyTexImage2D be used. Example usage: 1D case: gl.CopyTexImage(GL.TEXTURE_1D, 0, GL.RGBA, 0, 0, 128, 0) 2D case: gl.CopyTexImage(GL.TEXTURE_2D, 0, GL.RGBA, 0, 0, 128, 128, 0) OpenGL Documentation: glCopyTexImage1D, glCopyTexImage2D

- `target` Only relevant in the 2D case. Specifies the target texture. Must be GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X,

GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

- `level` Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
- `internalformat` Specifies the internal format of the texture. See OpenGL documentation for accepted values.
- `x` Specify the x window coordinate of the left corner of the row of pixels to be copied.
- `y` Specify the y window coordinate of the left corner of the row of pixels to be copied.
- `width` Specifies the width of the texture image. Must be 0 or 2 n + 2 border for some integer n.
- `[height]` Specifies the height of the texture image. Must be 0 or 2 m + 2 border for some integer m. The height of the texture image in the 1D case is 1.
- `border` Specifies the width of the border. Must be either 0 or 1.

```
CopyTexSubImage (target, level, xoffset, [yoffset], x, y, width, [height])
```

Copy a 1D or 2D texture subimageIf no yoffset or height argument is given, glCopyTexSubImage1D will be used, otherwise, glCopyTexSubImage2D be used. Example usage: 1D case: gl.glCopyTexSubImage(GL.TEXTURE_1D, 0, 32, 0, 0, 64) 2D case: gl.glCopyTexSubImage(GL.TEXTURE_2D, 0, 16, 24, 0, 0, 128, 128) OpenGL Documentation: glCopyTexSubImage1D, glCopyTexSubImage2D

- `target` Only relevant in the 2D case. Specifies the target texture. Must be GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, or GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.
- `level` Specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image.
- `xoffset` Specifies a texel offset in the x direction within the texture array..

- `[yoffset]` Specifies a texel offset in the y direction within the texture array..
- `x` Specify the x window coordinate of the left corner of the row of pixels to be copied.
- `y` Specify the y window coordinate of the left corner of the row of pixels to be copied.
- `width` Specifies the width of the texture subimage.
- `[height]` Specifies the height of the texture subimage.

```
CullFace (mode)
```

Specify whether front- or back-facing facets can be culledOpenGL Documentation: glCullFace

- `mode` Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK are accepted. The initial value is GL_BACK.

```
DeleteLists (list, range)
```

Delete a contiguous group of display listsOpenGL Documentation: glDeleteLists

- `list` Specifies the integer name of the first display list to delete.
- `range` Specifies the number of display lists to delete.

```
DepthFunc (func)
```

Specify the value used for depth buffer comparisonsOpenGL Documentation: glDepthFunc

- `func` Specifies the depth comparison function. Symbolic constants GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, and GL_ALWAYS are accepted. The initial value is GL_LESS.

```
DepthMask (flag)
```

Enable or disable writing into the depth bufferOpenGL Documentation: glDepthMask

- `flag` Specifies whether the depth buffer is enabled for writing. If flag is GL_FALSE, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

```
DepthRange (near, far)
```

Specify mapping of depth values from normalized device coordinates to window coordinatesOpenGL Documentation: glDepthRange

- `near` Specifies the mapping of the near clipping plane to window coordinates. The initial value is 0.
- `far` Specifies the mapping of the far clipping plane to window coordinates. The initial value is 1.

```
Disable (cap)
```

Disable server-side GL capabilitiesOpenGL Documentation: glDisable

- `cap` Specifies a symbolic constant indicating a GL capability.

```
DrawBuffer (mode)
```

Specify which color buffers are to be drawn intoOpenGL Documentation: glDrawBuffer

- `mode` Specifies up to four color buffers to be drawn into. Symbolic constants GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_AND_BACK, and GL_AUXi, where i is between 0 and the value of GL_AUX_BUFFERS minus 1, are accepted. (GL_AUX_BUFFERS is not the upper limit; use glGet to query the number of available aux buffers.) The initial value is GL_FRONT for single-buffered contexts, and GL_BACK for double-buffered contexts

```
EdgeFlag (flag)
```

Flag edges as either boundary or nonboundaryOpenGL Documentation: glEdgeFlag

- `flag` Specifies the current edge flag value, either GL_TRUE or GL_FALSE. The initial value is GL_TRUE.

```
Enable (cap)
```

Enable server-side GL capabilitiesOpenGL Documentation: glEnable

- `cap` Specifies a symbolic constant indicating a GL capability.

```
End ()
```

Delimit the vertices of a primitive or a group of like primitivesOpenGL Documentation: glEnd

```
EndList ()
```

End the creation or replacement of a display listOpenGL Documentation: glEndList

```
Finish ()
```

Block until all GL execution is completeOpenGL Documentation: glFinish

```
Flush ()
```

Force execution of GL commands in finite timeOpenGL Documentation: glFlush

```
Fog (pname, ...)
```

Specify fog parametersOpenGL Documentation: glFog

- `pname` Specifies a single-valued fog parameter. GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, GL_FOG_INDEX, and GL_FOG_COORD_SRC are accepted.
- `...` Specifies the value that pname will be set to.

```
FrontFace (mode)
```

Define front- and back-facing polygonsOpenGL Documentation: glFrontFace

- `mode` Specifies the orientation of front-facing polygons. GL_CW and GL_CCW are accepted. The initial value is GL_CCW.

```
Frustum (left, right, bottom, top, near, far)
```

Multiply the current matrix by a perspective matrixOpenGL Documentation: glFrustum

- `left` Specify the coordinates for the left vertical clipping plane.
- `right` Specify the coordinates for the right vertical clipping plane.
- `bottom` Specify the coordinates for the top horizontal clipping plane.
- `top` Specify the coordinates for the bottom horizontal clipping plane.
- `near` Specify the distance to the near depth clipping plane. Must be positive.
- `far` Specify the distance to the far depth clipping plane. Must be positive.

```
GenLists (range)
```

Generate a contiguous set of empty display listsOpenGL Documentation: glGenLists

- `range` Specifies the number of contiguous empty display lists to be generated.

```
Get (pname)
```

Return the value or values of a selected parameterOpenGL Documentation: glGet

- `pname` Specifies the parameter value to be returned. See documentation for the list of accepted values.

```
GetClipPlane (plane)
```

Return the coefficients of the specified clipping planeOpenGL Documentation: glGetClipPlane

- `plane` Specifies a clipping plane. The number of clipping planes depends on the implementation, but at least six clipping planes are supported. They are identified by symbolic names of the form GL_CLIP_PLANE i where i ranges from 0 to the value of GL_MAX_CLIP_PLANES - 1.

```
GetError ()
```

Return error informationOpenGL Documentation: glGetError

```
GetLight (light, pname)
```

Return light source parameter valuesOpenGL Documentation: glGetLight

- `light` Specifies a light source. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHTi where i ranges from 0 to the value of GL_MAX_LIGHTS - 1.
- `pname` Specifies a light source parameter for light. Accepted symbolic names are GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, and GL_QUADRATIC_ATTENUATION.

```
GetMaterial (target, pname)
```

Return material parametersOpenGL Documentation: glGetMaterial

- `target` Specifies which of the two materials is being queried. GL_FRONT or GL_BACK are accepted, representing the front and back materials, respectively.
- `pname` Specifies the material parameter to return. GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, and GL_COLOR_INDEXES are accepted.

```
GetString (name)
```

Return a string describing the current GL connectionOpenGL Documentation: glGetString

- `name` Specifies a symbolic constant, one of GL_VENDOR, GL_RENDERER, GL_VERSION, GL_SHADING_LANGUAGE_VERSION, or GL_EXTENSIONS.

```
GetTexEnv (target, pname)
```

Return texture environment parametersOpenGL Documentation: glGetTexEnv

- `target` Specifies a texture environment. May be GL_TEXTURE_ENV, GL_TEXTURE_FILTER_CONTROL, or GL_POINT_SPRITE.
- `pname` Specifies the symbolic name of a texture environment parameter. Accepted values are GL_TEXTURE_ENV_MODE, GL_TEXTURE_ENV_COLOR, GL_TEXTURE_LOD_BIAS, GL_COMBINE_RGB, GL_COMBINE_ALPHA, GL_SRC0_RGB, GL_SRC1_RGB, GL_SRC2_RGB, GL_SRC0_ALPHA, GL_SRC1_ALPHA, GL_SRC2_ALPHA, GL_OPERAND0_RGB, GL_OPERAND1_RGB, GL_OPERAND2_RGB,

GL_OPERAND0_ALPHA, GL_OPERAND1_ALPHA, GL_OPERAND2_ALPHA, GL_RGB_SCALE, GL_ALPHA_SCALE, or GL_COORD_REPLACE.

```
GetTexGen (coord, pname)
```

Return texture coordinate generation parametersOpenGL Documentation: glGetTexGen

- `coord` Specifies a texture coordinate. Must be GL_S, GL_T, GL_R, or GL_Q.
- `pname` Specifies the symbolic name of the value(s) to be returned. Must be either GL_TEXTURE_GEN_MODE or the name of one of the texture generation plane equations: GL_OBJECT_PLANE or GL_EYE_PLANE.

```
GetTexLevelParameter (target, level, pname)
```

Return texture parameter values for a specific level of detailOpenGL Documentation: glGetTexLevelParameter

- `target` Specifies the symbolic name of the target texture, either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_PROXY_TEXTURE_CUBE_MAP.
- `level` Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level n is the nth mipmap reduction image.
- `pname` Specifies the symbolic name of a texture parameter. GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_BORDER, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMINANCE_SIZE, GL_TEXTURE_INTENSITY_SIZE, GL_TEXTURE_DEPTH_SIZE,

GL_TEXTURE_COMPRESSED, and GL_TEXTURE_COMPRESSED_IMAGE_SIZE are accepted.

```
GetTexParameter (target, pname)
```

Return texture parameter valuesOpenGL Documentation: glGetTexParameter

- `target` Specifies the symbolic name of the target texture. GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, and GL_TEXTURE_CUBE_MAP are accepted.
- `pname` Specifies the symbolic name of a texture parameter. GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R, GL_TEXTURE_BORDER_COLOR, GL_TEXTURE_PRIORITY, GL_TEXTURE_RESIDENT, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_COMPARE_FUNC, GL_DEPTH_TEXTURE_MODE, and GL_GENERATE_MIPMAP are accepted.

```
Hint (target, mode)
```

Specify implementation-specific hintsOpenGL Documentation: glHint

- `target` Specifies a symbolic constant indicating the behavior to be controlled. GL_FOG_HINT, GL_GENERATE_MIPMAP_HINT, GL_LINE_SMOOTH_HINT, GL_PERSPECTIVE_CORRECTION_HINT, GL_POINT_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT, GL_TEXTURE_COMPRESSION_HINT, and GL_FRAGMENT_SHADER_DERIVATIVE_HINT are accepted.
- `mode` Specifies a symbolic constant indicating the desired behavior. GL_FASTEST, GL_NICEST, and GL_DONT_CARE are accepted.

```
Index (c)
```

Set the current color indexOpenGL Documentation: glIndex

- `c` Specifies the new value for the current color index.

```
IndexMask (mask)
```

Control the writing of individual bits in the color index buffersOpenGL Documentation: glIndexMask

- `mask` Specifies a bit mask to enable and disable the writing of individual bits in the color index buffers. Initially, the mask is all 1's.

```
InitNames ()
```

Initialize the name stackOpenGL Documentation: glInitNames

```
IsEnabled (cap)
```

Test whether a capability is enabledOpenGL Documentation: glIsEnabled

- `cap` Specifies a symbolic constant indicating a GL capability.

```
IsList (list)
```

Determine if a name corresponds to a display listOpenGL Documentation: glIsList

- `list` Specifies a symbolic constant indicating a GL capability.

```
IsTexture (texture)
```

Determine if a name corresponds to a textureOpenGL Documentation: glIsTexture

- `texture` Specifies a value that may be the name of a texture.

```
Light (light, pname, params)
```

Set light source parametersOpenGL Documentation: glLight

- `light` Specifies a light. The number of lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHTi, where i ranges from 0 to the value of GL_MAX_LIGHTS - 1.
- `pname` Specifies a light source parameter for light. GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_CUTOFF, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, and GL_QUADRATIC_ATTENUATION are accepted.
- `params` Specifies the value that parameter pname of light source light will be set to.

```
LightModel (pname, params)
```

Set the lighting model parametersOpenGL Documentation: glLightModel

- `pname` Specifies a single-valued lighting model parameter. GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_COLOR_CONTROL, and

GL_LIGHT_MODEL_TWO_SIDE are accepted.

- `params` Specifies the value that param will be set to.

```
LineStipple (factor, pattern)
```

Specify the line stipple patternOpenGL Documentation: glLineStipple

- `factor` Specifies a multiplier for each bit in the line stipple pattern. If factor is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used. factor is clamped to the range [1, 256] and defaults to 1.
- `pattern` Specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized. Bit zero is used first; the default pattern is all 1's.

```
LineWidth (width)
```

Specify the width of rasterized linesOpenGL Documentation: glLineWidth

- `width` Specifies the width of rasterized lines. The initial value is 1.

```
ListBase (base)
```

Set the display-list base for glCallListsOpenGL Documentation: glListBase

- `base` Specifies an integer offset that will be added to glCallLists offsets to generate display-list names. The initial value is 0.

```
LoadIdentity ()
```

Replace the current matrix with the identity matrixOpenGL Documentation: glLoadIdentity

```
LoadMatrix (m)
```

Replace the current matrix with the specified matrixOpenGL Documentation: glLoadMatrix

- `m` Specifies 16 consecutive values, which are used as the elements of a 4×4 column-major matrix.

```
LoadName (name)
```

Load a name onto the name stackOpenGL Documentation: glLoadName

- `name` Specifies a name that will replace the top value on the name stack.

```
LogicOp (opcode)
```

Specify a logical pixel operation for color index renderingOpenGL Documentation: glLogicOp

- `opcode` Specifies a symbolic constant that selects a logical operation. The following symbols are accepted: GL_CLEAR, GL_SET, GL_COPY, GL_COPY_INVERTED, GL_NOOP, GL_INVERT, GL_AND, GL_NAND, GL_OR, GL_NOR, GL_XOR, GL_EQUIV, GL_AND_REVERSE, GL_AND_INVERTED, GL_OR_REVERSE, and GL_OR_INVERTED. The initial value is GL_COPY.

```
Material (face, pname, params)
```

Specify material parameters for the lighting modelOpenGL Documentation: glMaterial

- `face` Specifies which face or faces are being updated. Must be one of GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK.
- `pname` Specifies the material parameter of the face or faces that is being updated. Must be one of GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, GL_AMBIENT_AND_DIFFUSE, or GL_COLOR_INDEXES.
- `params` Specifies a pointer to the value or values that pname will be set to.

```
MatrixMode (mode)
```

Specify which matrix is the current matrixOpenGL Documentation: glMatrixMode

- `mode` Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE. The initial value is GL_MODELVIEW. Additionally, if the ARB_imaging extension is supported, GL_COLOR is also accepted.

```
MultMatrix (m)
```

Multiply the current matrix with the specified matrixOpenGL Documentation: glMultMatrix

- `m` Specifies 16 consecutive values, which are used as the elements of a 4×4 column-major matrix.

```
NewList (list, mode)
```

Create or replace a display listOpenGL Documentation: glNewList

- `list` Specifies the display-list name.
- `mode` Specifies the compilation mode, which can be GL_COMPILE or GL_COMPILE_AND_EXECUTE.

```
Normal (...)
```

Set the current normal vectorOpenGL Documentation: glNormal

- `...` Either a 3 numbers or a table specifying the x, y, and z coordinates of the new current normal. The initial value of the current normal is the unit vector, (0, 0, 1).

```
Ortho (left, right, bottom, top, near, far)
```

Multiply the current matrix with an orthographic matrixOpenGL Documentation: glOrtho

- `left` Specify the coordinates for the left vertical clipping plane.
- `right` Specify the coordinates for the right vertical clipping plane.
- `bottom` Specify the coordinates for the bottom horizontal clipping plane.
- `top` Specify the coordinates for the top horizontal clipping plane.
- `near` Specify the distances to the nearer depth clipping plane. This value is negative if the plane is to be behind the viewer.

- `far` Specify the distances to the farther depth clipping plane. This value is negative if the plane is to be behind the viewer.

```
PassThrough (token)
```

Place a marker in the feedback bufferOpenGL Documentation: glPassThrough

- `token` Specifies a marker value to be placed in the feedback buffer following a GL_PASS_THROUGH_TOKEN.

```
PixelStore (pname, param)
```

Set pixel storage modesOpenGL Documentation: glPixelStore

- `pname` Specifies the symbolic name of the parameter to be set. Six values affect the packing of pixel data into memory: GL_PACK_SWAP_BYTES, GL_PACK_LSB_FIRST, GL_PACK_ROW_LENGTH, GL_PACK_IMAGE_HEIGHT, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, GL_PACK_SKIP_IMAGES, and GL_PACK_ALIGNMENT. Six more affect the unpacking of pixel data from memory: GL_UNPACK_SWAP_BYTES, GL_UNPACK_LSB_FIRST, GL_UNPACK_ROW_LENGTH, GL_UNPACK_IMAGE_HEIGHT, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS, GL_UNPACK_SKIP_IMAGES, and GL_UNPACK_ALIGNMENT.
- `param` Specifies the value that pname is set to.

```
PixelTransfer (pname, param)
```

Set pixel transfer modesOpenGL Documentation: glPixelTransfer

- `pname` Specifies the symbolic name of the pixel transfer parameter to be set. Must be one of the following: GL_MAP_COLOR, GL_MAP_STENCIL, GL_INDEX_SHIFT, GL_INDEX_OFFSET, GL_RED_SCALE, GL_RED_BIAS, GL_GREEN_SCALE, GL_GREEN_BIAS, GL_BLUE_SCALE, GL_BLUE_BIAS, GL_ALPHA_SCALE, GL_ALPHA_BIAS, GL_DEPTH_SCALE, or GL_DEPTH_BIAS.
- `param` Specifies the value that pname is set to.

```
PixelZoom (xfactor, yfactor)
```

Specify the pixel zoom factorsOpenGL Documentation: glPixelZoom

- `xfactor` Specify the x zoom factor for pixel write operations.
- `yfactor` Specify the y zoom factor for pixel write operations.

```
PointSize (size)
```

Specify the diameter of rasterized pointsOpenGL Documentation: glPointSize

- `size` Specifies the diameter of rasterized points. The initial value is 1.

```
PolygonMode (face, mode)
```

Select a polygon rasterization modeOpenGL Documentation: glPolygonMode

- `face` Specifies the polygons that mode applies to. Must be GL_FRONT for front-facing polygons, GL_BACK for back-facing polygons, or GL_FRONT_AND_BACK for front- and

back-facing polygons.

- `mode` Specifies how polygons will be rasterized. Accepted values are GL_POINT, GL_LINE, and GL_FILL. The initial value is GL_FILL for both front- and back-facing polygons.

```
PolygonOffset (factor, units)
```

Set the scale and units used to calculate depth valuesOpenGL Documentation: glPolygonOffset

- `factor` Specifies a scale factor that is used to create a variable depth offset for each polygon. The initial value is 0.
- `units` Is multiplied by an implementation-specific value to create a constant depth offset. The initial value is 0.

```
PopAttrib ()
```

Pop the server attribute stackOpenGL Documentation: glPopAttrib

```
PopClientAttrib ()
```

Pop the client attribute stackOpenGL Documentation: glPopClientAttrib

```
PopMatrix ()
```

Pop the current matrix stackOpenGL Documentation: glPopMatrix

```
PopName ()
```

Pop the name stackOpenGL Documentation: glPopName

```
PushAttrib (mask)
```

Push the server attribute stackOpenGL Documentation: glPushAttrib

- `mask` Specifies a mask that indicates which attributes to save.

```
PushClientAttrib ()
```

Push the client attribute stackOpenGL Documentation: glPushClientAttrib

```
PushMatrix ()
```

Push the current matrix stackOpenGL Documentation: glPushMatrix

```
PushName (name)
```

Push the name stackOpenGL Documentation: glPushName

- `name` Specifies a name that will be pushed onto the name stack.

```
RasterPos (..)
```

Specify the raster position for pixel operationsOpenGL Documentation: glRasterPos

- `..` Specifies an array or list of values of two, three, or four elements, specifying x, y, z, and w coordinates, respectively.

```
ReadBuffer (mode)
```

Select a color buffer source for pixelsOpenGL Documentation: glReadBuffer

- `mode` Specifies a color buffer. Accepted values are GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_AUXi, where i is between 0 and the value of GL_AUX_BUFFERS minus 1.

```
Rect (...)
```

Draw a rectangleOpenGL Documentation: glRect

- `...` Specify the vertices of a rectangle as an array or list of 4 values.

```
RenderMode (mode)
```

Set rasterization modeOpenGL Documentation: glRenderMode

- `mode` Specifies the rasterization mode. Three values are accepted: GL_RENDER, GL_SELECT, and GL_FEEDBACK. The initial value is GL_RENDER.

```
Rotate (...)
```

Multiply the current matrix by a rotation matrixOpenGL Documentation: glRotate

- `...` Specify an array or list of 4 values with the order being the angle of rotation in degrees and then the the x, y, and z coordinates of a vector, respectively

```
Scale (...)
```

Multiply the current matrix by a general scaling matrixOpenGL Documentation: glScale

- `...` Specify an array or list of 3 values of scale factors along the x, y, and z axes, respectively

```
Scissor (x, y, width, height)
```

Define the scissor boxOpenGL Documentation: glScissor

- `x` Specify the x-coordinate of the lower left corner of the scissor box.
- `y` Specify the y-coordinate of the lower left corner of the scissor box.
- `width` Specify the width of the scissor box.
- `height` Specify the height of the scissor box.

```
ShadeModel (mode)
```

Select flat or smooth shadingOpenGL Documentation: glShadeModel

- `mode` Specifies a symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH. The initial value is GL_SMOOTH.

```
StencilFunc (func, ref, mask)
```

Set front and back function and reference value for stencil testingOpenGL Documentation: glStencilFunc

- `func` Specifies the test function. Eight symbolic constants are valid: GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, and GL_ALWAYS. The initial value is GL_ALWAYS.
- `ref` Specifies the reference value for the stencil test. ref is clamped to the range 0 2 n-1 , where n is the number of bitplanes in the stencil buffer. The initial value is 0.
- `mask` Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done. The initial value is all 1's.

```
StencilMask (mask)
```

Control the front and back writing of individual bits in the stencil planesOpenGL Documentation: glStencilMask

- `mask` Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all 1's.

```
StencilOp (sfail, dpfail, dppass)
```

Set front and back stencil test actionsOpenGL Documentation: glStencilOp

- `sfail` Specifies the action to take when the stencil test fails. Eight symbolic constants are accepted: GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP, and GL_INVERT. The initial value is GL_KEEP.
- `dpfail` Specifies the stencil action when the stencil test passes, but the depth test fails. dpfail accepts the same symbolic constants as sfail. The initial value is GL_KEEP.
- `dppass` Specifies the stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled. dppass accepts the same symbolic constants as sfail. The initial value is GL_KEEP.

```
TexCoord (..)
```

Set the current texture coordinatesOpenGL Documentation: glTexCoord

- `..` Specifies an array or list of one, two, three, or four elements, which in turn specify the s, t, r, and q texture coordinates.

```
TexEnv (target, pname, param)
```

Set texture environment parametersOpenGL Documentation: glTexEnv

- `target` Specifies a texture environment. May be GL_TEXTURE_ENV, GL_TEXTURE_FILTER_CONTROL or GL_POINT_SPRITE.
- `pname` Specifies the symbolic name of a single-valued texture environment parameter. May be either GL_TEXTURE_ENV_MODE, GL_TEXTURE_LOD_BIAS, GL_COMBINE_RGB,

GL_COMBINE_ALPHA, GL_SRC0_RGB, GL_SRC1_RGB, GL_SRC2_RGB, GL_SRC0_ALPHA, GL_SRC1_ALPHA, GL_SRC2_ALPHA, GL_OPERAND0_RGB, GL_OPERAND1_RGB, GL_OPERAND2_RGB, GL_OPERAND0_ALPHA, GL_OPERAND1_ALPHA, GL_OPERAND2_ALPHA, GL_RGB_SCALE, GL_ALPHA_SCALE, or GL_COORD_REPLACE.

- `param` Specifies a single symbolic constant, one of GL_ADD, GL_ADD_SIGNED, GL_INTERPOLATE, GL_MODULATE, GL_DECAL, GL_BLEND, GL_REPLACE, GL_SUBTRACT, GL_COMBINE, GL_TEXTURE, GL_CONSTANT, GL_PRIMARY_COLOR, GL_PREVIOUS, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, a single boolean value for the point sprite texture coordinate replacement, a single floating-point value for the texture level-of-detail bias, or 1.0, 2.0, or 4.0 when specifying the GL_RGB_SCALE or GL_ALPHA_SCALE.

```
TexGen (coord, pname, param)
```

Control the generation of texture coordinatesOpenGL Documentation: glTexGen

- `coord` Specifies a texture coordinate. Must be one of GL_S, GL_T, GL_R, or GL_Q.
- `pname` Specifies the symbolic name of the texture-coordinate generation function. Must be GL_TEXTURE_GEN_MODE.
- `param` Specifies a single-valued texture generation parameter, one of GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP, GL_NORMAL_MAP, or GL_REFLECTION_MAP.

```
TexParameter (target, pname, param)
```

Set texture parametersOpenGL Documentation: glTexParameter

- `target` Specifies the target texture, which must be either GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, or GL_TEXTURE_CUBE_MAP.
- `pname` Specifies the symbolic name of a single-valued texture parameter. pname can be one of the following: GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_BASE_LEVEL,

GL_TEXTURE_MAX_LEVEL, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R, GL_TEXTURE_PRIORITY, GL_TEXTURE_COMPARE_MODE, GL_TEXTURE_COMPARE_FUNC, GL_DEPTH_TEXTURE_MODE, or GL_GENERATE_MIPMAP.

- `param` Specifies the value of pname.

```
Translate (...)
```

Multiply the current matrix by a translation matrixOpenGL Documentation: glTranslate

- `...` Specify an array or list of 3 values of translation vector

```
Vertex (..)
```

Specify a vertexOpenGL Documentation: glVertex

- `..` Specifies an array or list of two, three, or four elements, which in turn specify the x, y, z, and w vertex coordinates.

```
Viewport (x, y, width, height)
```

Set the viewportOpenGL Documentation: glViewport

- `x` Specify the x-coordinate of the lower left corner of the viewport rectangle.
- `y` Specify the y-coordinate of the lower left corner of the viewport rectangle.
- `width` Specify the width of the viewport rectangle.
- `height` Specify the height of the viewport rectangle.

jit.gl.lua OpenGL GLU Bindings

The GLU (openGL Utility) bindings are located in the `opengl.glu` module. It is a sub-module of `opengl` module. To access them, use Lua's built-in `require` function as follows:

```
local  glu = require( "opengl.glu" )
```

For the standard OpenGL bindings, see the jit.gl.lua OpenGL Bindings.

```
LookAt (eye, center, up)
```

Define a viewing transformationOpenGL Documentation: gluLookAt If LookAt has nine arguments, it expects unpacked vectors. Otherwise, it will look for 3 vectors as arguments.

- `eye` Specifies the position of the eye point.
- `center` Specifies the position of the reference point.
- `up` Specifies the direction of the up vector.

```
Ortho2D (left, right, bottom, up)
```

Define a 2D orthographic projection matrixOpenGL Documentation: gluOrtho2D

- `left` Specify the coordinates for the left vertical clipping planes.
- `right` Specify the coordinates for the right vertical clipping planes.

- `bottom` Specify the coordinates for the bottom horizontal clipping planes.
- `up` Specify the coordinates for the up horizontal clipping planes.

```
Perspective (fovy, aspect, near, far)
```

Set up a perspective projection matrixOpenGL Documentation: gluPerspective

- `fovy` Specifies the field of view angle, in degrees, in the y direction.
- `aspect` Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
- `near` Specifies the distance from the viewer to the near clipping plane (always positive).
- `far` Specifies the distance from the viewer to the far clipping plane (always positive).

```
Project (...)
```

Map object coordinates to window coordinatesOpenGL Documentation: gluProject

- `...` Specify the object coordinates as either a table or unpacked values.

```
UnProject (...)
```

Map window coordinates to object coordinatesOpenGL Documentation: gluUnProject

- `...` Specify the window coordinates as either a table or unpacked values.

jit.gl.lua Vector Math

vec2 1053
vec3 1055
vec4 1059
quat 1061
mat3 1063
mat4 1064

The vector math functions in jit.gl.lua are located in the `vec` module and are organized into six categories. These categories are:- vec.vec2

- vec.vec3
- vec.vec4
- vec.quat
- vec.mat3
- vec.mat4 The arguments to all of the vector math functions are tables containing the appropriate number of values for the data type. For example a vec2 is a table with two elements, {1, 2}, while a mat3 has nine elements {1, 2, 3, 4, 5, 6, 7, 8, 9}. The matrices are specified in column-major order so for a mat3, the first three elements are the first column, the second three the second column and so on.

vec2

```
-- res is a 1 or 0 depending on equality
res = vec2.equal(v1, v2)
```

```
-- res is a 1 or 0 depending on equality
res = vec2.not_equal(v1, v2)
```

```
res = vec2.dot(v1, v2)
```

```
res = vec2.normalize(v)
```

```
res = vec2.mult(v1, v2)
```

```
res = vec2.scale(v, s)
```

```
res = vec2.div(v1, v2)
```

```
res = vec2.sub(v1, v2)
```

```
res = vec2.add(v1, v2)
```

```
res = vec2.mag_sqr(v)
```

```
res = vec2.mag(v)
```

```
res = vec2.negate(v)
```

```
res = vec2.max(v1, v2)
```

```
res = vec2.lerp(v1, v2, t)
```

vec3

```
-- res is a 1 or 0 depending on equality
res = vec3.equal(v1, v2)
```

```
-- res is a 1 or 0 depending on equality
res = vec3.not_equal(v1, v2)
```

```
res = vec3.mult(v1, v2)
```

```
res = vec3.scale(v, s)
```

```
res = vec3.div(v1, v2)
```

```
res = vec3.sub(v1, v2)
```

```
res = vec3.add(v1, v2)
```

```
res = vec3.mag_sqr(v)
```

```
res = vec3.mag(v)
```

```
res = vec3.negate(v)
```

```
res = vec3.cross(v1, v2)
```

```
res = vec3.dot(v1, v2)
```

```
res = vec3.reflect(v1, v2)
```

```
-- calculate the normal to a triangle defined by three points
res = vec3.normal(p1, p2, p3)
```

```
res = vec3.normalize(v)
```

```
res = vec3.max(v1, v2)
```

```
res = vec3.min(v1, v2)
```

```
res = vec3.lerp(v1, v2, t)
```

```
res = vec3.intersect_line_sphere(linepos1, linepos2, sphere_center,
sphere_radius)
```

```
res = vec3.axisx_from_quat(quat)
```

```
res = vec3.axisy_from_quat(quat)
```

```
res = vec3.axisz_from_quat(quat)
```

```
res = vec3.transform_axisangle(axis, angle, v)
```

```
res = vec3.mult_mat3(v, mat3)
```

```
res = vec3.centroid3(v1, v2, v3)
```

```
res = vec3.centroid4(v1, v2, v3, v4)
```

vec4

```
-- res is a 1 or 0 depending on equality
res = vec4.equal(v1, v2)
```

```
-- res is a 1 or 0 depending on equality
res = vec4.not_equal(v1, v2)
```

```
res = vec4.mult(v1, v2)
```

```
res = vec4.scale(v, s)
```

```
res = vec4.div(v1, v2)
```

```
res = vec4.sub(v1, v2)
```

```
res = vec4.add(v1, v2)
```

```
res = vec4.mag_sqr(v)
```

```
res = vec4.mag(v)
```

```
res = vec4.negate(v)
```

```
res = vec4.dot(v1, v2)
```

```
res = vec4.normalize(v)
```

```
res = vec4.max(v1, v2)
```

```
res = vec4.min(v1, v2)
```

```
res = vec4.lerp(v1, v2, t)
```

```
res = vec4.mult_mat4(v, mat4)
```

quat

```
-- res is a 1 or 0 depending on equality
res = quat.equal(q1, q2)
```

```
-- res is a 1 or 0 depending on equality
res = quat.not_equal(q1, q2)
```

```
res = quat.mult(q1, q2)
```

```
res = quat.scale(q, s)
```

```
res = quat.div(q1, q2)
```

```
res = quat.add(q1, q2)
```

```
res = quat.mag_sqr(q)
```

```
res = quat.mag(q)
```

```
res = quat.negate(q)
```

```
res = quat.dot(q1, q2)
```

```
res = quat.normalize(q)
```

```
res = quat.max(q1, q2)
```

```
res = quat.min(q1, q2)
```

```
res = quat.slerp(q1, q2, t)
```

```
res = quat.from_mat3(mat3)
```

```
res = quat.from_mat4(mat4)
```

```
res = quat.from_axis_angle(axis, angle)
```

```
res = quat.conj(q)
```

```
res = quat.from_euler(euler_angles)
```

mat3

```
res = mat3.add(m1, m2)
```

```
res = mat3.mult(m1, m2)
```

```
res = mat3.transpose(m)
```

```
res = mat3.mult_vec3(m, v)
```

```
res = mat3.from_axisangle(axis, angle)
```

```
res = mat3.from_uv(v1, v2)
```

```
res = mat3.determinant(m)
```

```
res = mat3.negate(m)
```

```
res = mat3.from_mat4(mat4)
```

```
res = mat3.from_quat(q)
```

mat4

```
res = mat4.add(m1, m2)
```

```
res = mat4.mult(m1, m2)
```

```
res = mat4.transpose(m)
```

```
res = mat4.mult_vec4(m, v)
```

```
res = mat4.from_axisangle(axis, angle)
```

```
res = mat4.from_uv(v1, v2)
```

```
res = mat4.determinant(m)
```

```
res = mat4.negate(m)
```

```
res = mat4.from_quat(q)
```

```
res = mat4.look_at(eye, center, up)
```

```
res = mat4.frustum(left, right, bottom, top, near, far)
```

```
res = mat4.perspective(fovy, aspect, near, far)
```

```
res = mat4.ortho(left, right, bottom, top, near, far)
```

Credits

Cycling '74
440 N Barranca Ave #4074
Covina, CA 91723 USA

Copyright 2026 Cycling '74. All rights reserved.

This document, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Cycling '74. Every effort has been made to ensure that the information in this document is accurate. Cycling '74 assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

Except as permitted by such license, no part of this publication may be reproduced, edited, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of Cycling '74.

Contact Support: https://cycling74.com/support/contact