Controlling Max with Commercial Devices

Max has a rich set of user interface objects, but they are necessarily limited to wha tcan be done with a keyboard and mouse. Many times we need better control, such as the ability to change two parameters at the same time. For this, hardware is the only solutions. There is an amazing variety of controllers available, ranging from iPhone apps to sophisticated musical instruments[1]. It may seem impossible to learn to take advantage of them all, but most of these funnel the control signals through one of three protocols: MIDI, HID (Human Interface Device) or OSC (Open Sound Control).

MIDI

Since Max was initially a music application, MIDI is its native language. You can review the nuts and bolts of MIDI in my tutor on Basic MIDI, but you don't need all of that just to get some controllers working. You do need to know these basic facts:

- MIDI is transmitted as a set of messages over specialized MIDI cables or USB.
- The computer refers to various MIDI inputs as "Ports" with names.
- Messages consist of a header byte that identifies the type of message and one or two bytes of data[2].
- The format limits the precision of most data to 7 bits or a range of 0 to127.
- Message headers include tags that identify them as belonging to one of 16 channels.

The message types are:

- Note On, with data for note number and velocity. This is sent when a key is pressed.
- Note Off, with data for note number and velocity. This is sent when a key is released. A note on with a velocity of 0 is also considered a note off.
- Control Change, with data for control number and value. These are sent as the control is changed, usually at a rate of 20 per second.
- Pitch Bend, with data for the bend amount of -8192 to 8191. (Not all devices use this much precision.)
- Program change, with a singe byte of data indicating a preset number.
- Channel pressure, with a single byte of data indicating excess pressure on any held key.
- Polyphonic aftertouch, with data for key number and excess pressure applied to that key.
- System Common messages, with various meanings like cue points in a song.

[1] If the commercial options are not enough for you, you can build your own with a bit of studuy and effort- see "Electronics for Contraptions", "Wiring Contraptions', and "Notes on Firmata".
[2] Actually, additional pairs of data bytes can be sent without repeating the header. Thus the header is often called "status".

- System Exclusive messages (Sysex) with meanings defined by individual manufacturers.

Note on and control change are the most useful for jitter interaction.

Catching Notes

The most important object in Max is notein.

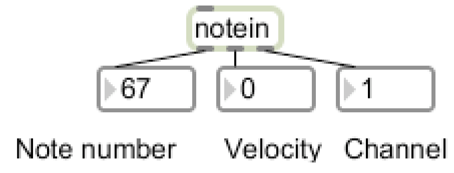


Figure 1.

Notein has three outlets providing the note number, velocity and channel of the note message. The note number identifies the key pressed. The range of note numbers is 0-127, but notes below 24 or above 108 are seldom available (or audible). Keyboards cover various ranges, but middle C is always note number 60. The velocity ranges from 1 to 127. It is almost impossible to play an exact velocity, so use the information for gradual processes like adjusting a sound volume.

Notein will report key number and a velocity of 0 for note offs. There is no explicit noteoff object. (There is xnotein if you need the velocity of note offs, but few keyboards actually send that.) If you double-click on the notein object while the patcher is locked, you can assign it to a particular port:

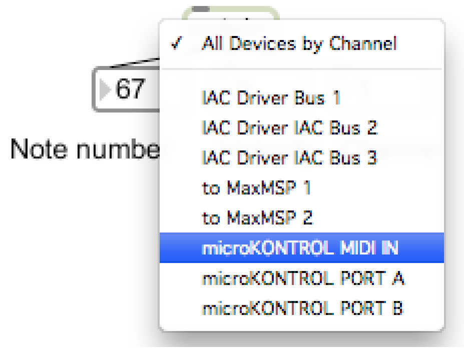


Figure 2.
The default is to follow all ports.

Much of the time, we don't care about the note off, so there is a stripnote object to isolate the note ons. (It should be called strip-note-offs.)

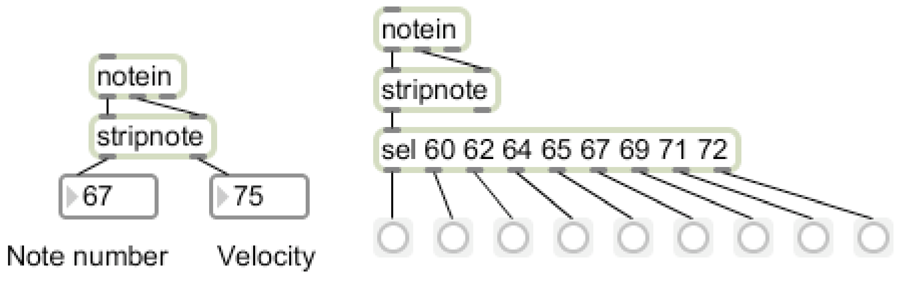


Figure 3.

Combine stripnote with select (sel) to notice particular keys.

If you want to know if a particular key is held down, you can pack the note messages into a list and filter them with route.

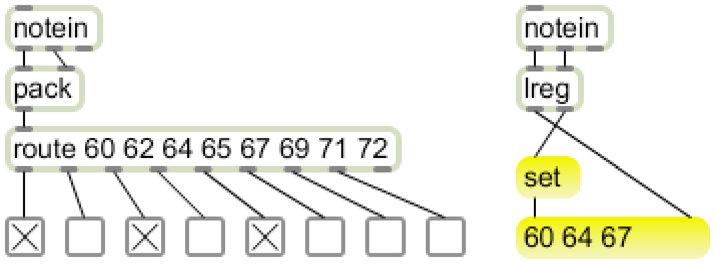
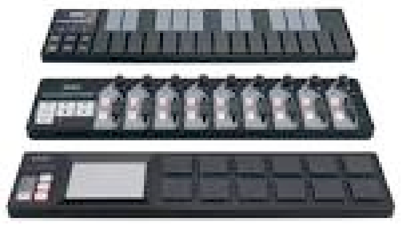


Figure 4.

If you want a running list of held notes, there is an Lobject called Lreg. The right outlet of Lreg bangs when all of the keys are released.

Note arrivals are ideal for triggering events such as starting movies. You aren't limited to interfaces that look like keyboards. There are MIDI trigger devices that work from contact microphones-- attach a piezoelectric transducer to practically anything, and a note will be produced when the object is tapped. A MIDI keyboard is mostly switches, so it is easy to hack. Open it up and pull out the circuit board (use one that is powered from USB) then wire switches to the key contact points. A student of mine once used this trick with some conductive fabric to make a playable pair of trousers. (Google DrumPants to see it in action.)

Knobs, Sliders and Buttons



There are a lot of MIDI control surfaces for sale right now, at prices ranging from $50-$5,000. These are mostly designed to control digital audio workstations, so they provide a combination of sliders, knobs and buttons. The essential object for reading these is ctlin.

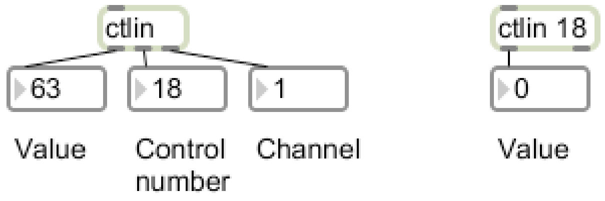


Figure 5.
Ctlin comes in two varieties. If there is no argument, the object will have three outlets providing value, control number and channel. Use this to discover the number of the widget you want to assign to a function. When you type a control number as an argument to ctlin, there will only be two outlets and only that control will be followed.

MIDI allows up to 14 bit precision on some controller numbers. The scheme pairs a controller numbered from 0 to 31 with one 32 higher. The low numbered controller transmits the most significant byte, and the other transmits the LSB.

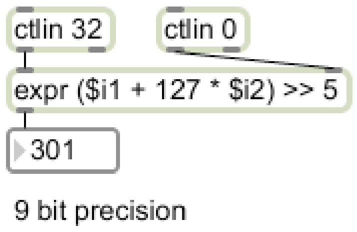


Figure 6.

Some assemblage like figure 6 is required to read the complete number. The general formula for getting a 14 bit number out of two data bytes is LSB+128*MSB. The controllers that do this generally provide 8 to 10 bits of precision, so some tweaking is required to get the proper range. This is provided by the >>5 in the expr object. That's a right shift 5 bits, equivalent to dividing by 32. (You can discover the precision of a device by looking at the output with number boxes in binary mode.)

Controls that are actually buttons or foot pedals will produce either 0 or 127.

There are about 120 control numbers. (Higher ones have special meanings and should be left alone.) Many of these have a standard definition-- for instance controller 7 is volume, controller 10 is pan and controller 11 is expression (sort of extra loudness). A controller for graphics does not have to pay attention to this of course, but it explains the odd selection of control numbers you may find.

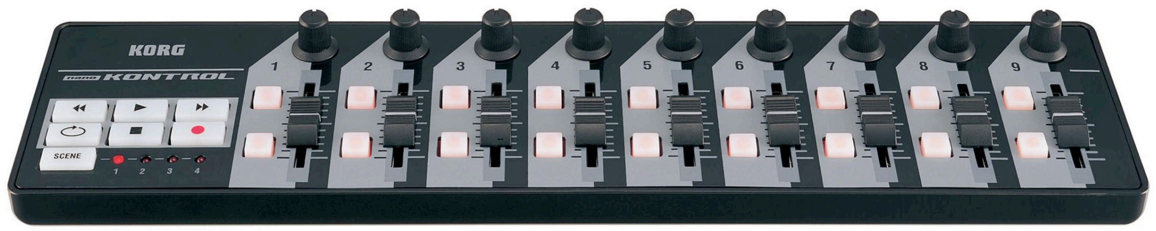
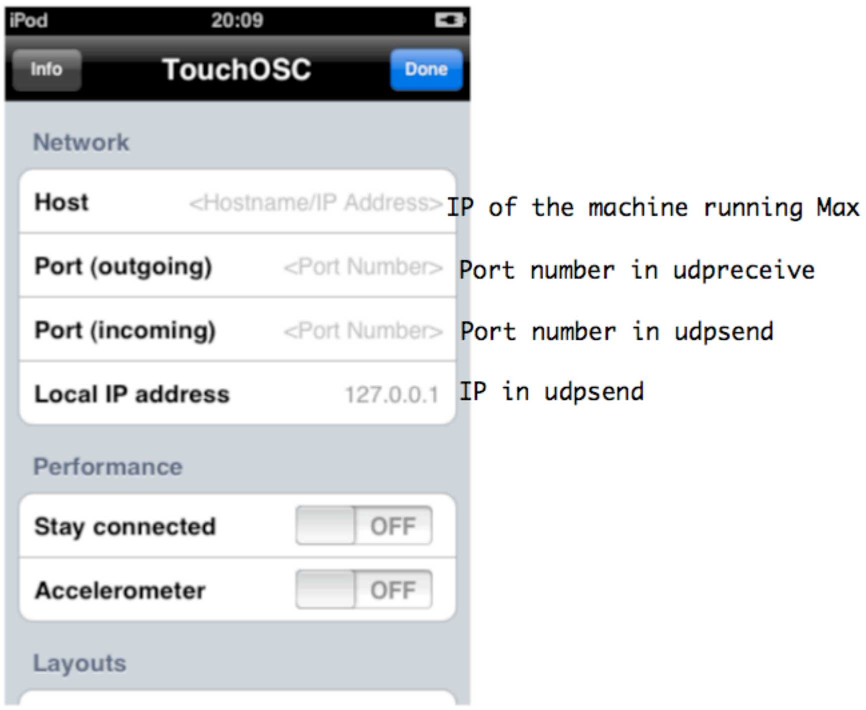


Figure 7.
Some controllers can be remapped to match the expectations of different software. For instance, if you explore the numbers of the sliders on the Korg nanoKontrol (pictured in figure 7) you will find they are numbered 2, 3, 4, 5 6, 8, 9,12 and 13. Pressing the scene button will change mappings-- on scene 4 every slider is control 7, but the channels change. You can often edit the mapping yourself, but since you are already writing a Max patch to interpret the numbers, why bother?

OSC

Open Sound Control is a protocol for sending control messages across networks. OSC can be used in many network schemes, but we usually use Ethernet or wireless systems running UDP/IP. Every computer has one or the other network connection, and normal network traffic will not interfere with OSC messages.

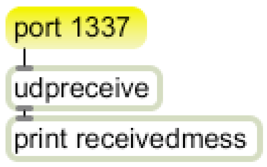
The first step to successful OSC use is getting connected. This is usually a matter of telling the OSC device the IP number of your computer. The IP number can be found in your network settings panel- it looks something like 128.114.11.139. When you do enter this in the device, you will notice that the device also has an IP number. Make note of that and the port number that shows up in the device menu.



Host	<Hostname/IP Address>	IP of the machine running Max
Port (outgoing)	<Port Number>	Port number in udpreceive
Port (incoming)	<Port Number>	Port number in udpsend
Local IP address	127.0.0.1	IP in udpsend
Stay connected	OFF	
Accelerometer	OFF	

Figure 8. Typical device setup menu

To receive data from the device, you need nothing more complicated than this:

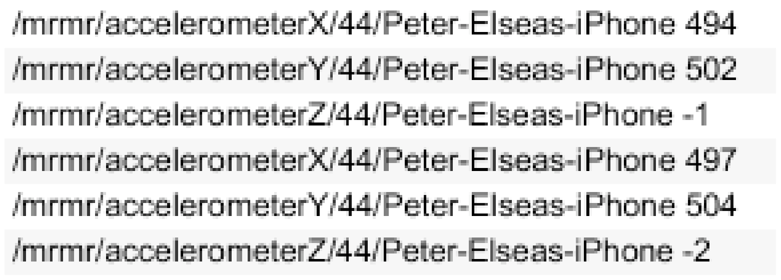


```
port 1337
udpreceive
print receivedmess
```

Figure 9.

Figure 9 prints the incoming messages to the max window. You will need to do this at the start to work out the format of the messages the device sends. The port message should match the outpoing port number of the device. If this never changes, it can be an argument to udpreceive.

The messages are formatted like URLs, such as:

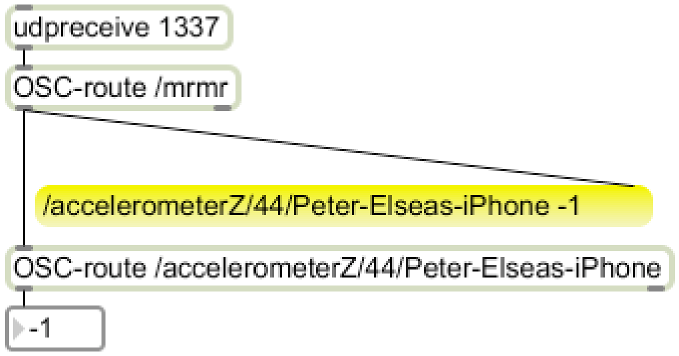


```
/mrmr/accelerometerX/44/Peter-Elseas-iPhone 494
/mrmr/accelerometerY/44/Peter-Elseas-iPhone 502
/mrmr/accelerometerZ/44/Peter-Elseas-iPhone -1
/mrmr/accelerometerX/44/Peter-Elseas-iPhone 497
/mrmr/accelerometerY/44/Peter-Elseas-iPhone 504
/mrmr/accelerometerZ/44/Peter-Elseas-iPhone -2
```

Every device or program has its own set of messages. To decode these, you will want the OSC objects from CNMAT at UC Berkley:

http://www.cnmat.berkeley.edu/OSC

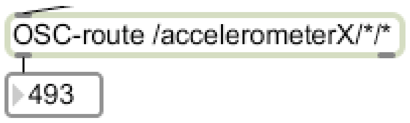
The OSC-route object can parse strings delimited by slashes. It works just like route--each string gets an outlet. If the input starts with the string, the outlet reports everything after the string. There is an extra outlet that reports anything that doesn't match.



```
udpreceive 1337
OSC-route /mrmr
/accelerometerZ/44/Peter-Elseas-iPhone -1
OSC-route /accelerometerZ/44/Peter-Elseas-iPhone
-1
```

Figure 10. Parsing OSC
I like to catch incoming messages so I can copy and paste them to the OSC-route. This saves a lot of typing.

OSC-route will recognize a * as a wildcard character. This allows us to abbreviate the string like figure 10. It's not only shorter to type, it will now work with any iPhone.



```
OSC-route /accelerometerX/*/*
493
```

Figure 11.

This is all that is needed for most OSC systems. A few (like Lemur) have oddball implementations that require their own Max externals.

You can send OSC messages too. The opensoundcontrol object from the CNMAT OSC page will create them, and you connect them to udpsend. This is the basis for Max patches that run on two or more computers. Simple messages may be sent directly. This patch will set the position of a fader in touchOSC on an iPhone:

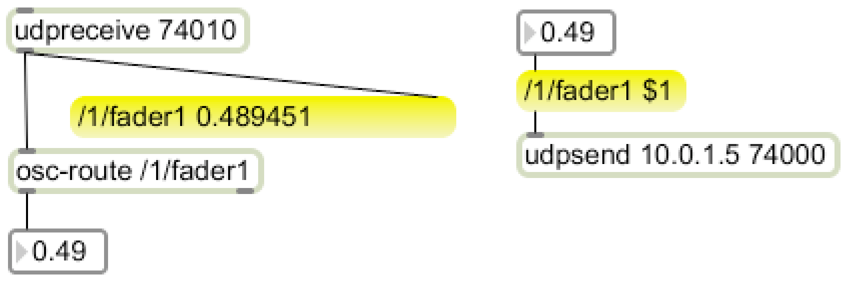


Figure 12.

HID

The Human Interface Device standard is a Microsoft protocol for devices such as joysticks. Quite a few gadgets use it, including Logitech wheels and gamepads, Wii remotes and Guitar Hero. The Max connection to the world of HID is the hi object.

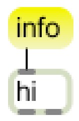
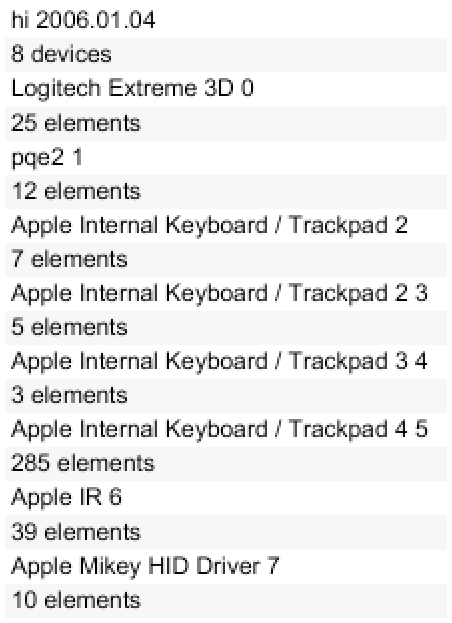


Figure 13. Probing for hi devces

Sending the info message to the hi object will print a list of connected hi devices to the Max window. (see Figure 14.) You will get the name of each device and the number of control elements (buttons, knobs, whatever) it has. The number that follows the name is an index number, not part of the name.



```
hi 2006.01.04
8 devices
Logitech Extreme 3D 0
25 elements
pqe2 1
12 elements
Apple Internal Keyboard / Trackpad 2
7 elements
Apple Internal Keyboard / Trackpad 2 3
5 elements
Apple Internal Keyboard / Trackpad 3 4
3 elements
Apple Internal Keyboard / Trackpad 4 5
285 elements
Apple IR 6
39 elements
Apple Mikey HID Driver 7
10 elements
```

Figure 14. An hi listing

If you add the name of the device (in quotes if the name includes spaces) to the hi object, the message poll xx will start reading the device every xx milliseconds. If any action has occurred, there will be a pair of numbers- element number followed by the new value. These are spaced closely enough to be converted into a list with the thresh object.

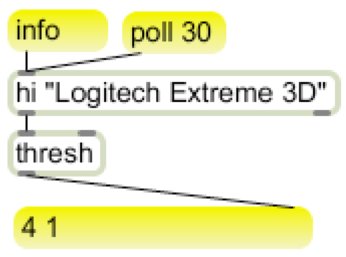


Figure 15. Basic hi patch.

From here in it is all detective work. You need to operate each control to find the associated element number and the range of data it provides. Once you know this, you can parse everything with route.

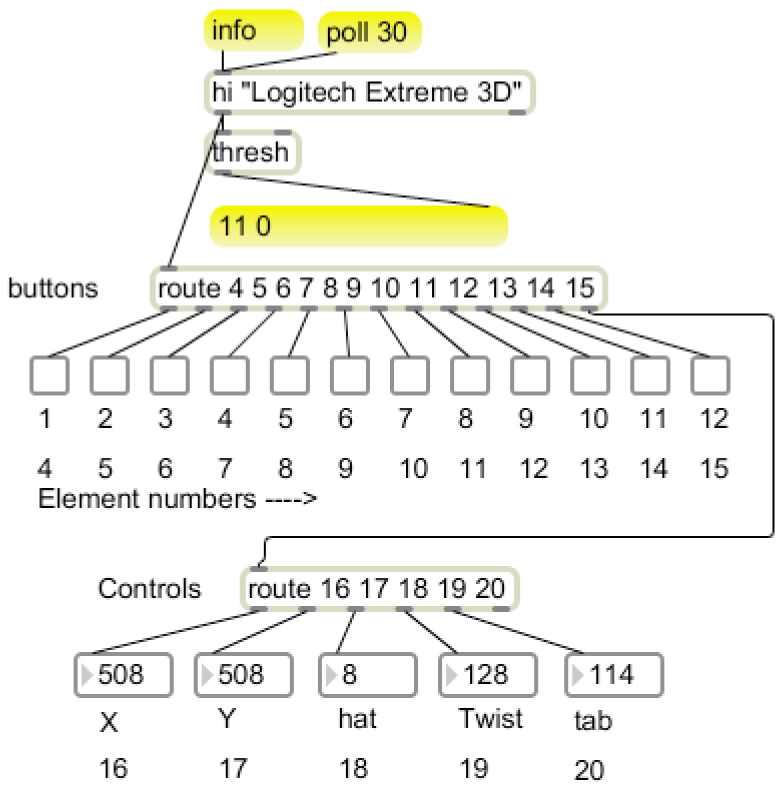


Figure 16.

Not all of the element numbers are always used.

Generalizing Patches with Send and Receive

One annoying feature of these controllers is that no two use the same set of control numbers or element numbers (to say nothing of what you are decoding in OSC.) When I need to use a patch with a choice of controllers, I include receive objects in the places I want remote control. Figure 17 shows these attached to a favorite patch of mine (see The Old Feedback Trick).

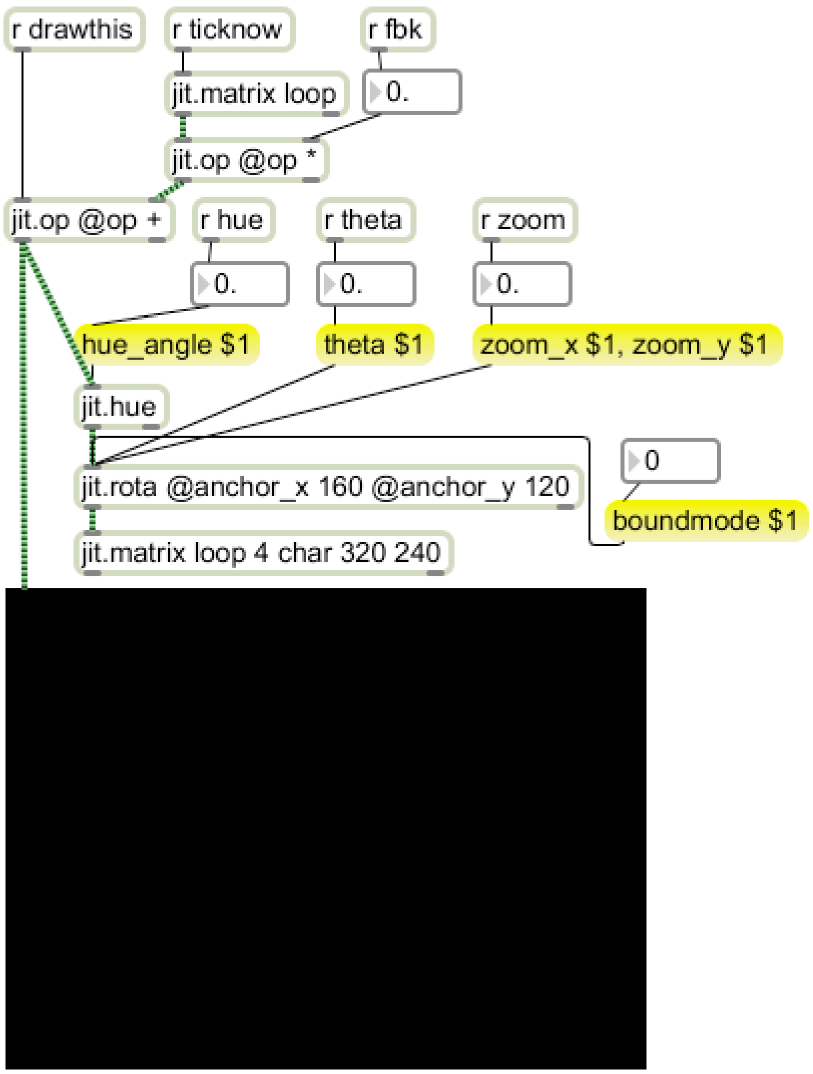


Figure 17. Feedback processor with receive object for remote control.

Figure 18 shows a second window that gives control to a Korg box.

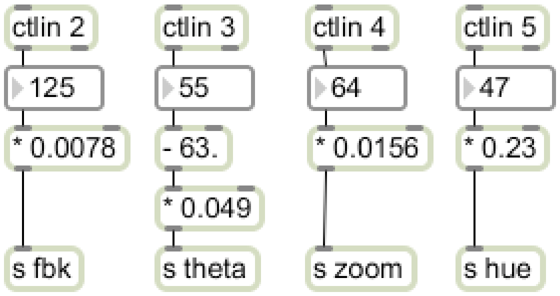


Figure 18. Midi based feedback control patch.
Putting the inputs in a separate window is useful on its own because it keeps the processing patch clean. Notice that the math required to convert from value range 0-127

is contained in the remote patch. A patch to use another MIDI controller would differ by the numbers in the ctl boxes. Figure 19 shows control of the same elements by a entirely different device.

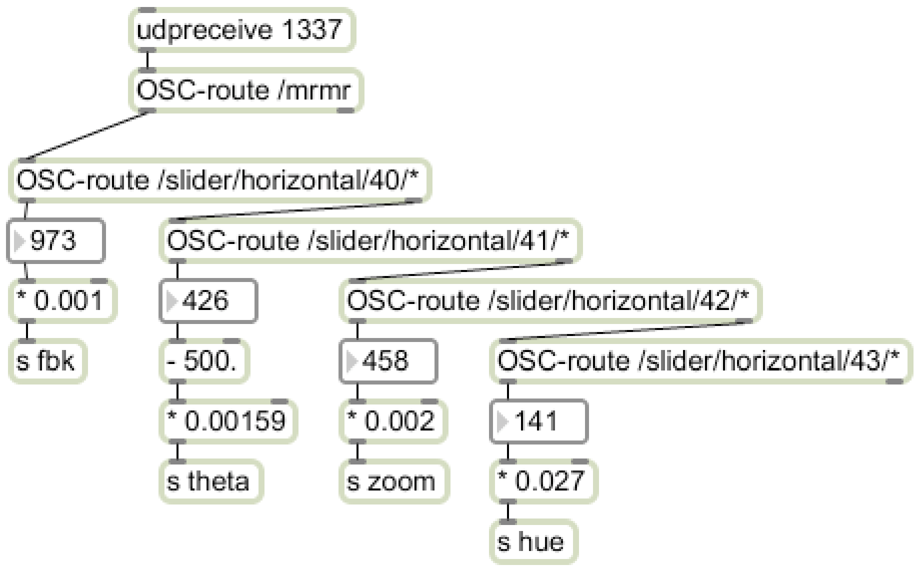

udpreceive 1337
OSC-route /mrmr
OSC-route /slider/horizontal/40/*
973
* 0.001
s fbk
OSC-route /slider/horizontal/41/*
426
- 500.
* 0.00159
s theta
OSC-route /slider/horizontal/42/*
458
* 0.002
s zoom
OSC-route /slider/horizontal/43/*
141
* 0.027
s hue

Figure 19. iPhone control of feedback via mrmr.

With this system, I can choose the control device I want to use for a show based on more practical reasons than what I had around when I first designed the patch.

Llatch

I like to have both local and remote control of performance patches. This is easily done by attaching appropriate send objects to sliders anywhere in the window. Of course this brings up the problem of synchronization. Unless the external controller has motorized faders (expensive) and you set up appropriate update messages, the slider in the patcher will be out of synch with the physical control, or vice versa. There is an Lobject that manages this problem.

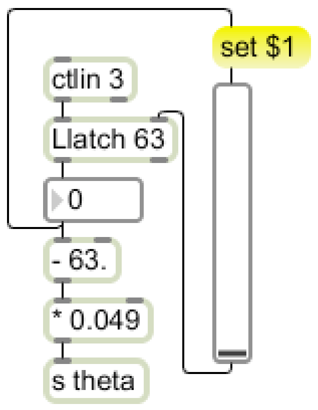


Figure 20.

Figure 20 shows the use of Llatch. Llatch has two inputs for data that is passed out the left outlet. The right inlet has priority over the left. If data is received in the right, data from the left is ignored until it has matched or "crossed" the right input. If the external control is connected to the left, it will have to be moved to match before anything happens. If the data output from Llatch is fed back to a set message to the graphic slider, it will always match the physical slider. This way there will be no sudden jumps in the data. Llatch is only available for Macs at this time-- figure 20 shows how to build an equivalent subpatch.

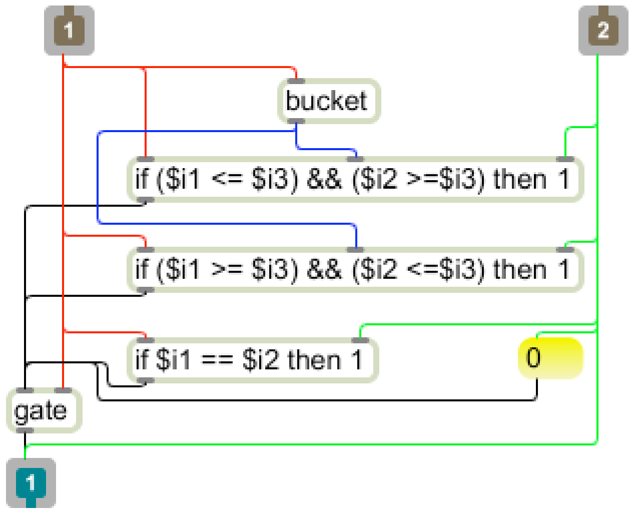

1
2
bucket
if ($i1 <= $i3) && ($i2 >=$i3) then 1
if ($i1 >= $i3) && ($i2 <=$i3) then 1
if $i1 == $i2 then 1
0
gate
1

Figure 21. Faking Llatch
The bucket object delays input and outputs old data when new data is received. This makes it easy to compare successive numbers. Any data in the right shuts data from the left off. Three if statements test for conditions that will allow it through again.

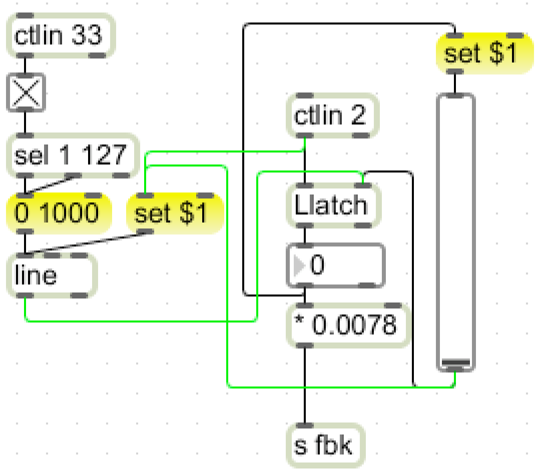


Figure 22. Adding automation to latched control.

Figure 22 shows how to keep automated control in synch. Any sort of automation should probably come in by way of a line object to slow transitions down. In this case it is triggered by a button on the controller[3]. The output of line can be applied to the right inlet of the latch-- the graphic slider will be updated to follow the changing value. However, line will not work at all until it is set to the current value. This requires a set message from any source that changes the current value. It cannot be set from the output of Llatch because that would cause a feedback loop.

[3] The sel object in this patch watches for both 1 and 127 because the toggle will pass the 127 that comes from a MIDI switch, but send 1 if it is clicked.