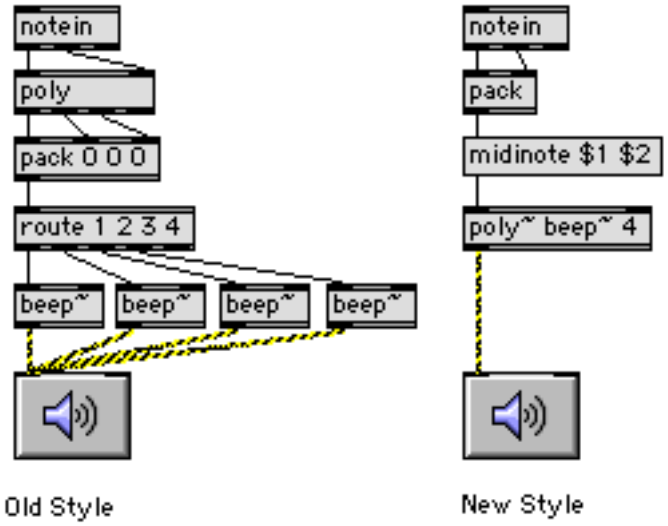
Working with Poly~ in Max

The poly~ object is the key to efficient use of the CPU in Max work. It is also useful for chores that need a lot of parallel processing of any kind.

What is poly~?

Think of it as a stack of embedded patchers that share inlets and outlets. These two arrangements are the same. Instead of having four explicit instances of the beep~ subpatcher, [poly~ beep~ 4] encapsulates four beeps for the space of one. four instances is trivial, but if you want 24 all you have to do is change the number in poly~



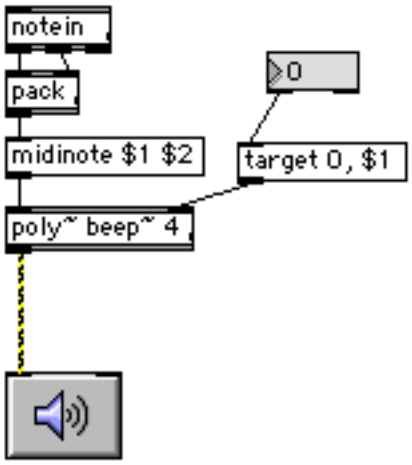
Note that the logic of selecting which instance of beep~ is playing a particular note is a bit different in the two. The problem of course, is getting the note off to the proper beep~.

The old style uses the poly (no tilde) object. This keeps track of what is playing, and assigns a voice number- the matching note off will be sent with that same voice number. Packing and routing gets everything where it needs to be.

To get the same behavior from poly~, you send it the message [midinote nn vv] where nn is the note number and vv is the velocity. Poly assigns voice numbers (the docs call these "instance numbers") and sends the note offs to find the proper voice.

If you send the message note (with whatever arguments you need), the behavior is simpler. The message goes to the next free voice, period.

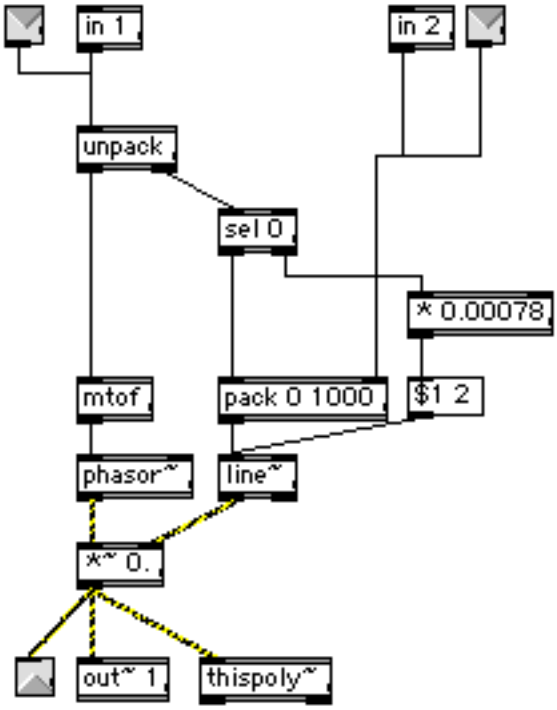
There is a third way to manage the messaging in poly~. It's a bit more complicated, but gives extra flexibility. The message "target n" sets voice n to be the recipient of whatever comes next. (not counting note or midinote of course). "Target 0" means send to all instances.



Thispoly~

So how does the Poly~ know what voices are available for note and midinote messages? The embedded patcher has to tell it. Communication from the patcher up to poly~ is via the thispoly~ object. The most important thing the voice patcher sends to thispoly~ is a 1 (int or signal) to mark the voice as busy. To free up the voice, send a 0. If you forget this, poly will be monophonic. The easiest way to use thispoly~ is simply hook up the signal output to it. Then when the signal goes to zero, the voice is marked as free. (Thispoly~ seems to be tolerant of the occasional 0s that go by in an active signal.)

Here's a simple voice that will work in a poly~:



The sound generator is phasor~ which makes a buzz. *~ adjusts output level as controlled by line~, The patch is expecting a note number and velocity to turn it on and a note number and 0 to turn it off. You can see how this is split by the unpack, and sel 0. These send messages to line~ to give a quick attack and an adjustable release. Note that the velocity from sel is multiplied by 0.00078 to translate from velocities 0-127 to amplitude 0.0-0.10. It's important to keep the sum of all signals coming out of the poly~ under 1.0.

In and Out

The inputs to a patcher in poly~ are handled differently than they are in ordinary subpatchers. This example has both the old style and poly~ style inlets, so it will work either way. There are four new objects to get in and out of poly~:

- in x makes an inlet that will accept messages. x is the inlet number. If you use 4 for x there will be at least four inlets. These are the messages that are affected by target and note, with messages only appearing in the voices you choice.
- in~ x is a signal inlet. Signals go to all voices always. If an in and in~ have the same number, they will be the same inlet looking from the outside.
- out~ x is a signal outlet. The signals from all the voices appear mixed at the outlet with no scaling, so it's easy to get distortion here.

· out x is a message outlet. The message outlets always appear to the right of the signal outlets, so if there are three out~s , out 1 will be the fourth outlet.

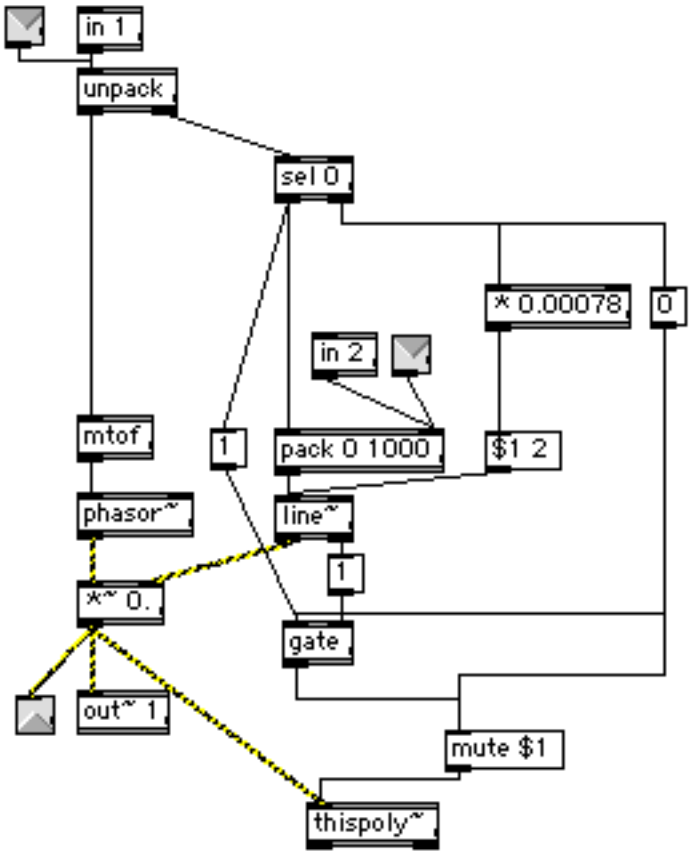
By the way- if you add these to a patcher that is already loaded in a poly~, (and save it) the new outlets and inlets won't magically appear as they do with embedded patchers. To get them to show, edit the poly~ by selecting the argument and retyping it. (Strangely, the code inside is updated, so you only need to do this if the number of inlets or outlets changes.)

Muting

One of the more important features of poly~ is muting. If you have 24 instances of a complex patch in a poly~, you are going to chew up CPU pretty fast. It's nice to be able to turn off unneeded signal processing, and the mute message to thispoly~ will do that exactly that. No matter what you may read in the manual:

· mute 1 turns muting on, therefore the voice off
· mute 0 turns the voice on again.

The trick is to get muting to happen when the voice is really done, not just when the note off arrives. Here's our voice modified to mute when unneeded.



When the note on comes in, the mute 0 happens unequivocally. We need to get the mute 1 from the bang at the end of line~, but this also bangs on the way up, so it has to be prevented by a gate until it's really needed. The right outlet of this poly~ shows the mute status, so its easy to test.

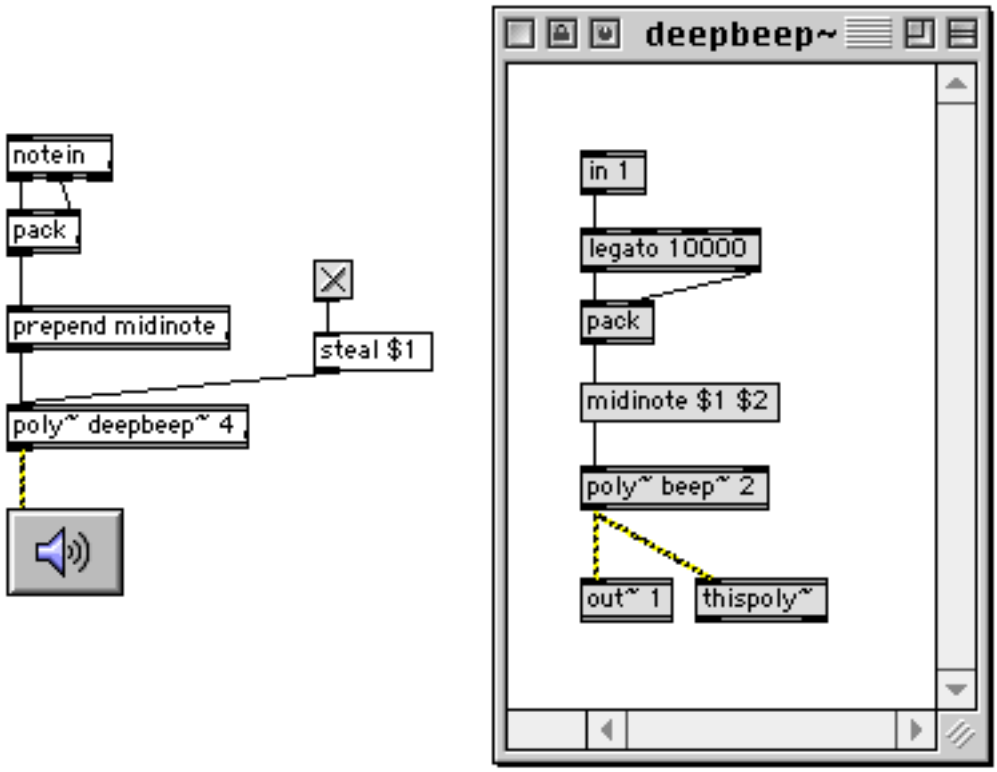
You can also mute voices from the outside, with the mute n 1/0 message to the poly~.

Ripoff

No matter how many layers of polyphony you specify, eventually someone is going to put down one key too many. What happens then is called ripoff. Normally, if all of the voices in poly~ are busy, any new messages will be ignored. If you give poly~ the message steal 1, something will be shut off unexpectedly, probably with a click. Your voice patchers should be able to handle this by one of these methods:

- Fade out the playing voice just before the new note comes in. Since this requires predicting the future and I haven't written that object yet, it's not very practical.
- Delay the new note a bit while the old one fades. This may be OK in some cases, but you should be sure to delay only when it's absolutely necessary.
- Crossfade between the old note and the new note. This is my preferred approach.

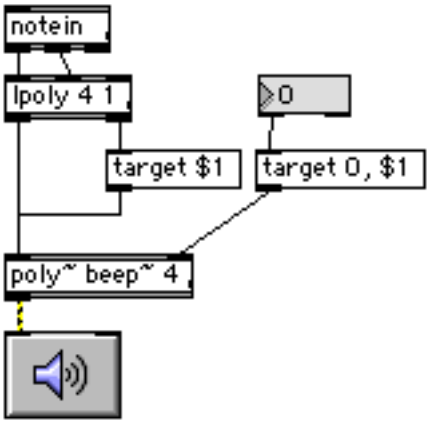
To make a crossfade, we'll put a poly~ in the poly~ by building an intermediate patcher:



The deepbeep~ patcher depends on the legato object (that's an Lobject) for the switching logic. Legato will provide a note off message for the playing note just before sending the new note. (Legato is a version of makenote. There's a duration parameter, which is set to 10 seconds in this example.)

Target

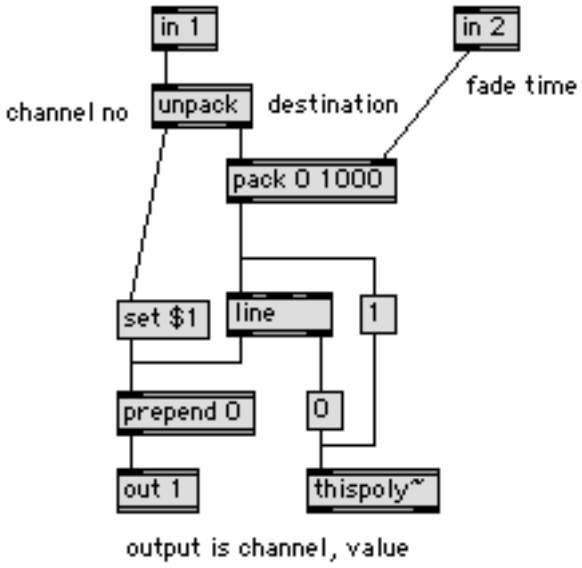
The target message allows you to control the action of poly more closely. Target n switches instance n to be the current recipient of all messages. (Signals go to all instances all the time, but you can use a targeted message to select a particular signal for a particular instance.



Here, I'm using the lpoly[1] object to manage the polyphony of poly~. In this configuration, the voice number chooses the target before the note message arrives. Lpoly uses a ripoff algorithm I prefer to the one in poly~[2]. You can easily roll your own in a patcher object. The right inlet controls the decay time of the beep~ patcher. (see page 3) The target 0 preface sets all instances to receive the message.

Other uses for poly~

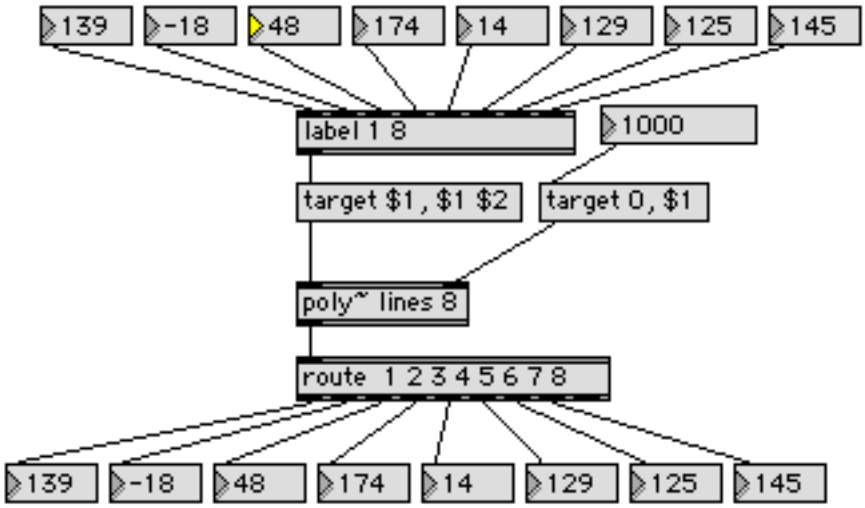
Poly is part of the MSP package, but its uses are not limited to audio. Here's an instance patcher that ramps from the current value to a new one:



Note that it always passes the channel number through. But as we need to keep the current state of line associated with a channel, the master patcher uses target messages:

[1] This is a new version of lpoly I created after thinking about this example a bit. The second argument does two things. It switches the operation so the voice number comes out its own right outlet, and sets the number of the lowest voice. With only one argument, the voice number starts with 0 and is at the start of the output list, appropriate for route.

[2] Poly just rotates through voices starting from 1. Lpoly dumps the oldest voice, unless it is the lowest pitch.



This could easily become a 128 channel light controller.

Special messages

There are many more useful features in poly~ You can tweak the audio performance by changing the sampling rate or vector size, trading quality for efficiency (or vice versa). These can be done with arguments or messages. If you set scheduling to internal with the local argument, events generated in the poly~ will have more accurate timing.

You can put #x type arguments in the instance patchers in the usual way. These are filled by what ever follows the word args in the poly~ itself.