Max and MSP

MSP is an addition to Max that provides signal generation and processing objects. It works entirely in the Macintosh, which gives you advantages and disadvantages.

Advantages:
• You can do whatever you want, not whatever marketing thought would sell.
• It doesn't cost a lot of money.
• It doesn't cost any money to change your mind about what kind of sound you want.
• When a new computer comes out, your patches won't have to be thrown away, they'll run better!

Disadvantages:
• Your typical MIDI instrument can run circles around a Mac (even a G3) when it comes to audio processing. So dedicated instruments will always have more channels, thicker chords, etc.
• There can be a noticeable delay when sounds pass in and out of the computer.
• You have to make things work yourself. An empty patcher won't make a sound.

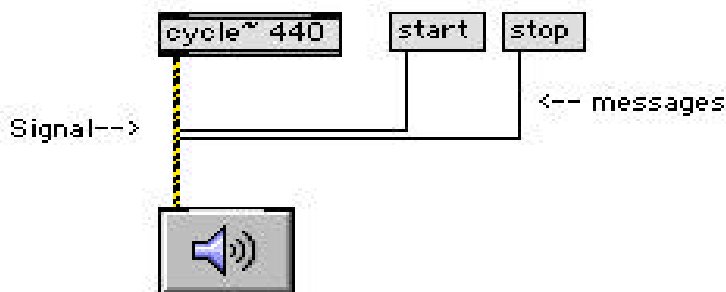
Warnings:
MSP is not entirely bug free. You are going to have some crashes and freezeups, especially when your patches get large. You must do three things to cope with this:
• Save often, especially when you are about to turn on audio
• If it starts to behave strangely when audio is not on, save and reboot.

You are going to have to study the tutorial (which is a PDF file on the computer- open the manuals alias to find it). But here are some basic concepts to get you started.

MSP Basics

MSP objects have a tilde (~) after their name. Many of them have the same name (except for the tilde) as regular Max objects. Usually, the function is similar to their namesake.

Max works with messages- int, float, list, bang, that are sent once from one object to another. MSP works with signals that are flowing continuously whenever audio is on.



Cords that pass signals are yellow with black stripes. The signals really consist of samples in batches called vectors. The number of samples in a vector can be changed for various reasons, but usually there are 256.

You can't watch a signal with a number box the way you can with the other messages. The best you can do is grab a single sample with sah~ (sample and hold), grab a bunch of numbers with capture~, or watch a display similar to an oscilloscope.

When a signal is captured, it looks like a lot of floats. The values will be between -1 and 1. A signal that swings all the way from –1 to 1 represents full scale and is very loud.

Audio Output

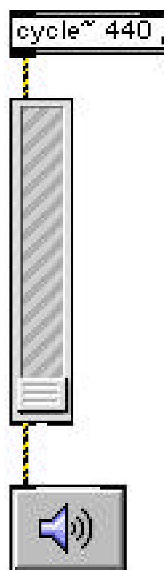
The output of an MSP patch is the dac~ or ezdac~ (shown above, it looks like a button with a picture of a speaker on it.) To start audio, send either a 1 or the message "start". (You can just click on the ezdac~) This starts all audio in all windows. To hear only audio from one window, send "startwindow". Audio will stop with the stop message, a 0, or if you edit the audio patch. You can add signal cords while audio is running, but they won't be heard until you stop and restart.

The left and right inlets of the dac~ correspond to left and right stereo outputs. dac~ can have arguments that determine the output channels, up to 16.

Audio input is handled by adc~ or ezadc~ (picture of a microphone). You can start and stop audio with the adc~ objects also.

Levels

To adjust the volume of a signal, you use either a multiplier [*~] (with a fractional value) or a gain~ slider. This looks very much like the Vslider when it's in the tool palette- it just has two little lines on the handle. When it's in the patch, it has colored stripes.

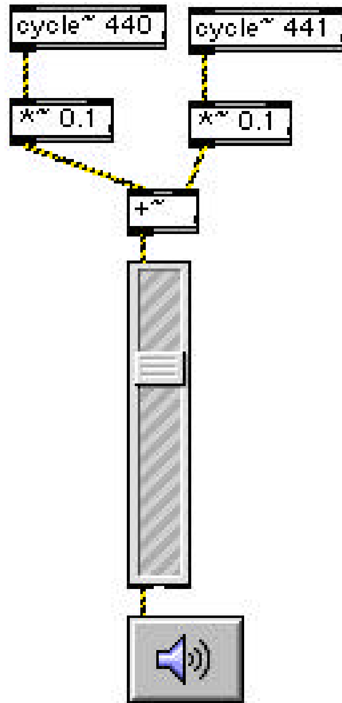


The gain slider has two outlets-
The left one gives you the attenuated signal. The right tells the position of the slider.

The signal is applied to the left inlet. Ints at this inlet will move the slider. A change of 10 is 6 dB. The gain~ slider does increase the signal level when it is all the way up. This will often produce distortion when used as shown.

The level does not change instantly when you move the slider. It ramps up or down so the signal doesn't pop. A float in the right inlet sets the ramp time (in milliseconds). It defaults to 10 ms.

A multiplier is often a better way to control gain, if you don't need a user control (you should have one gain~ slider per sound) if you think of the multiplier as equivalent to a VCA, you'll get the idea. I usually stick a multiplier after every signal producing element in order to get levels under control, like this:



Believe me, even attenuating those cycles~ to 0.1 gives a plenty strong signal. Remember your decibels?

Db = 20 log v/V, where V is the peak output of cycle~.

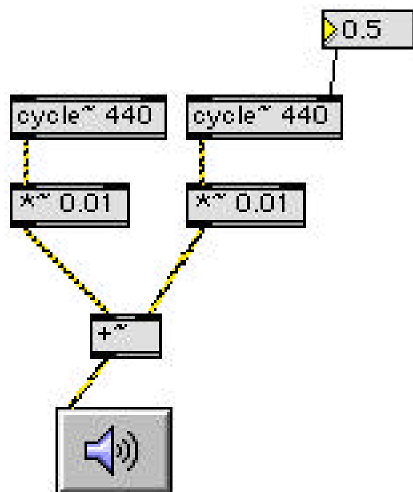
20 log 0.1/1 = -20 dB
20 log 0.01/1 = -40 dB
20 log 0.001/1 = -60 dB

20 dB down is noticeably quieter, but when we add two signals like this, the peak values of the sum will hit 0.2, which is only one fifth of distortion level.

Notice that the adder [+~] object is used to combine signals. Think of it as a mixer. There's no difference between the two inlets -- none of that right goes in first stuff, they are both active continuously. (In MSP2, adder objects are not as necessary. Any signal inlet will add signals.)

Oscillators

The cycle~ object is the basic oscillator. When audio is on it puts out a sine wave at he indicated frequency.

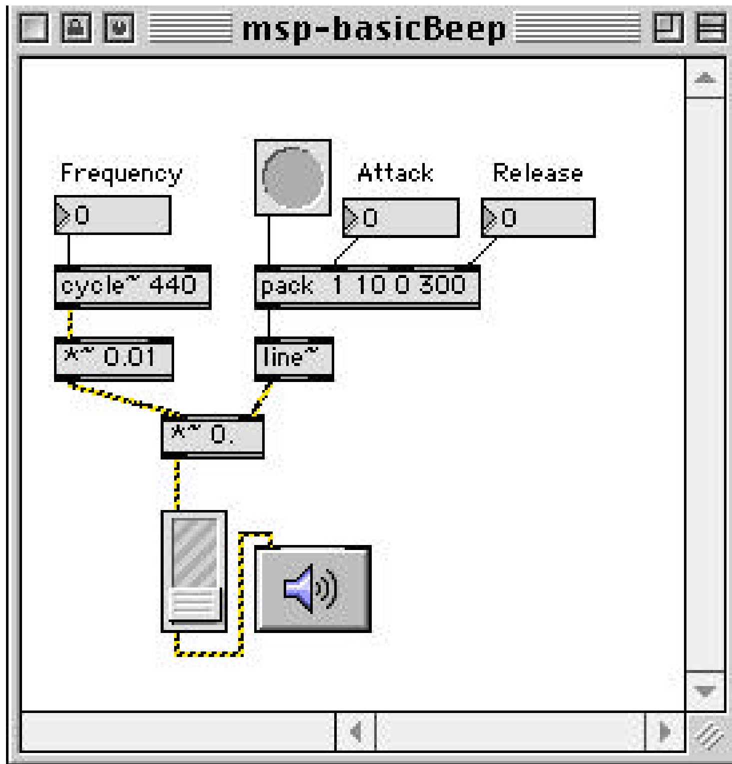


There are two inlets to cycle~ the left one controls the frequency, the right one controls phase. Frequency is in hertz (and there's a neat object called mtof that converts midi note numbers to frequency.)

Phase is a fraction of the wavetable, so 0.25 is 90 degrees, 1.0 is a full 360. The patch at the left doesn't make any sound, because 0.5 is 180 degrees out of phase, and you know what that does!

Basic Beep

The line~ object will give us the equivalent of the envelope generator. It can be used with cycle~ and another multiplier to give us this:



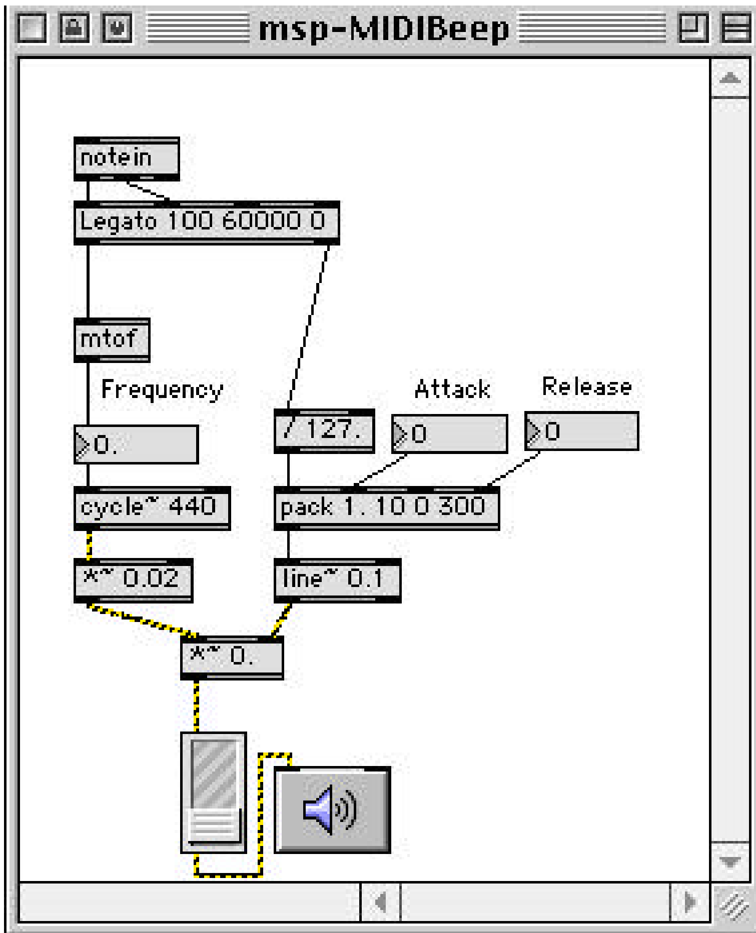
Note I still dedicate a multiplier to adjusting the initial level of the cycle~. This makes it easy to trim levels when the patcher gets complex.

Line~ is controlled by pairs of numbers in a message. The numbers are target, time; target, time; and so on, up to 46 pairs. When line~ receives the message it starts putting out numbers in a ramp from where it is now to the target. This is a signal that continues putting out the final target value until a new message comes in. You can also control line~ by ints; time in the right and target in the left.

The pack object contains a default envelope- notice the target for the attack is 1 and the final value is 0. Most envelopes will be similar, possibly with other line segments. The attack time is adjusted with the second inlet and the release time is adjusted via the last inlet. A bang to the pack object sends the envelope. If you set a long release and click the button quickly, you will hear only slight attacks, because when a new message is received, line starts at the current value

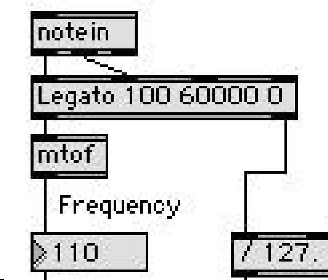
MIDI control

The next step is to trigger the sound from a MIDI keyboard. This uses objects you are already familiar with:



Here you see that all we need to do is take the pitch from notein and use the mtof object to convert it to a frequency, then divide the velocity by 127 and use that to trigger the pack with the envelope. Well, one more change: the default attack value in the pack has to be a float, or pack will change input in that inlet to ints, giving 0 most of the time.

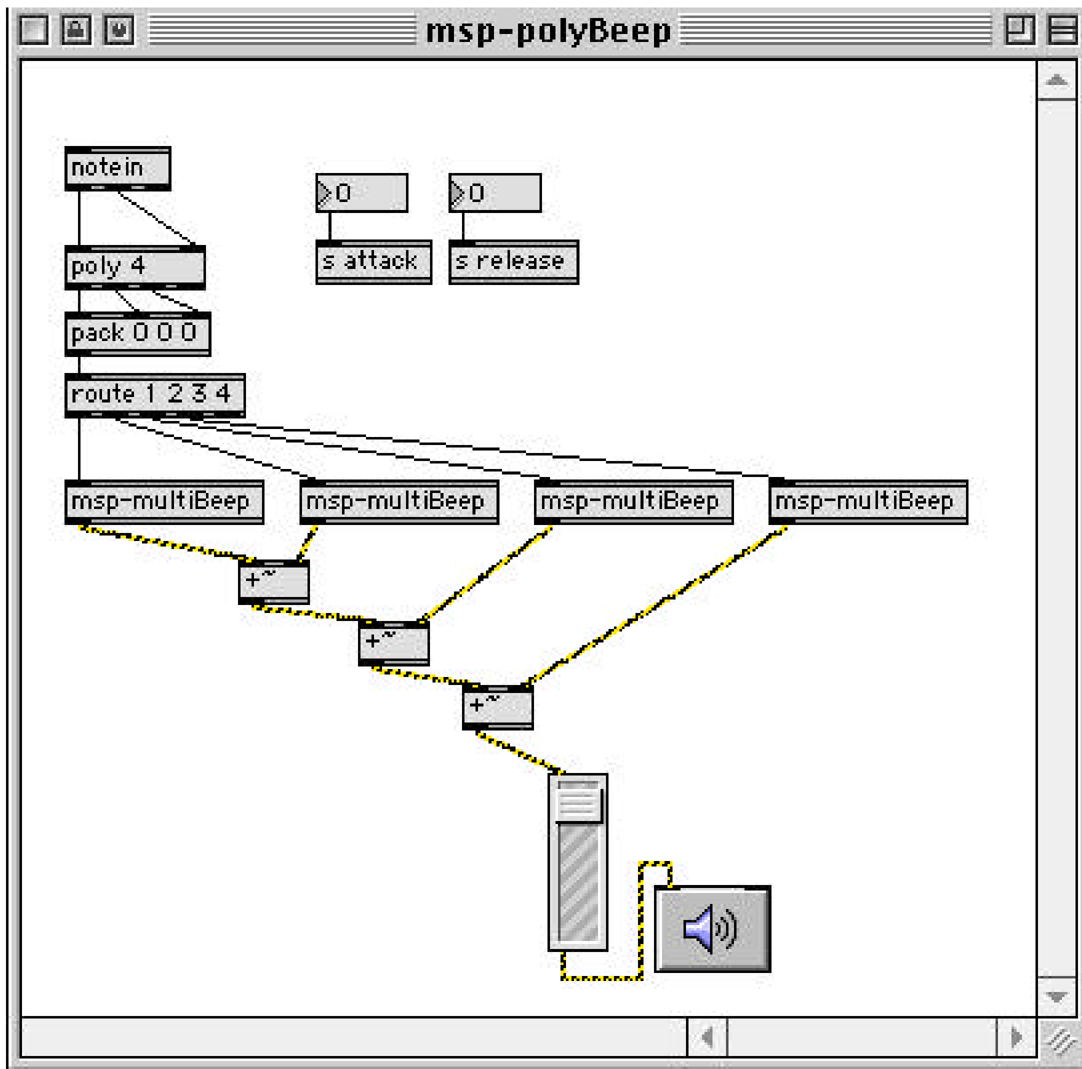
When you try this, you will find that it works, but that if you try to play too fast, you'll get chopped off notes. That's because if notes overlap, the note off for the first note will happen after the note on for the second note. There a variety of ways to fix this; the easiest being to stick a Legato (that's an Lobject) after the notein:



Legato is like makenote, but only allows one note to be active at a time. The arguments are velocity, duration and overlap. (Overlap can stretch the noteoff past the next note.) If we set the duration longer than we are likely to play, legato can be used as a kind of filter for notein turning off notes even if our keyboard technique is a bit sloppy.

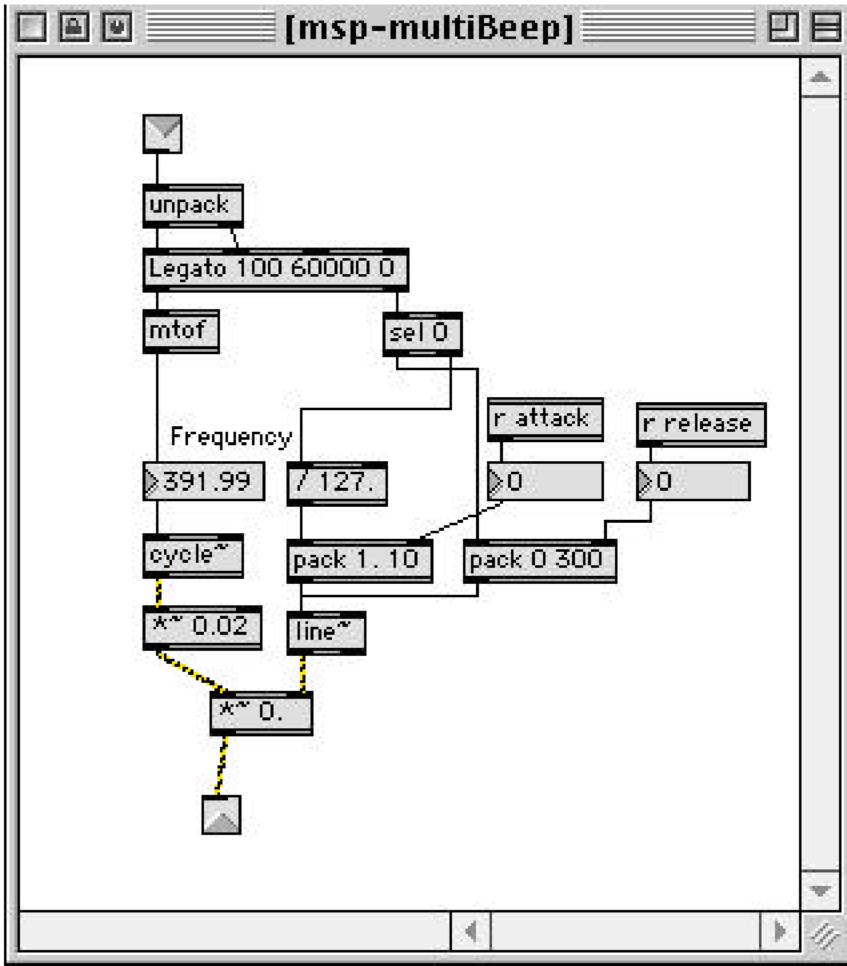
Polyphony

To get several notes to play at once we can embed several basic beeps in the same patcher:



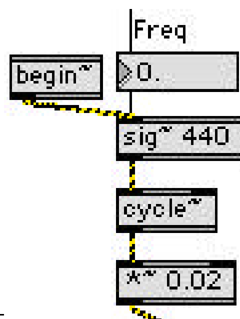
The poly object (part of normal Max) assigns a voice number to a note on message, and when the note off comes along, sends out the same voice number. If we put the voice number at the beginning of a message, the route object passes that message out the appropriate outlet, helpfully removing the voice number as it does so. Therefore our players get a list of note number and velocity. The other objects mix the notes into a single output.

The players look like this:



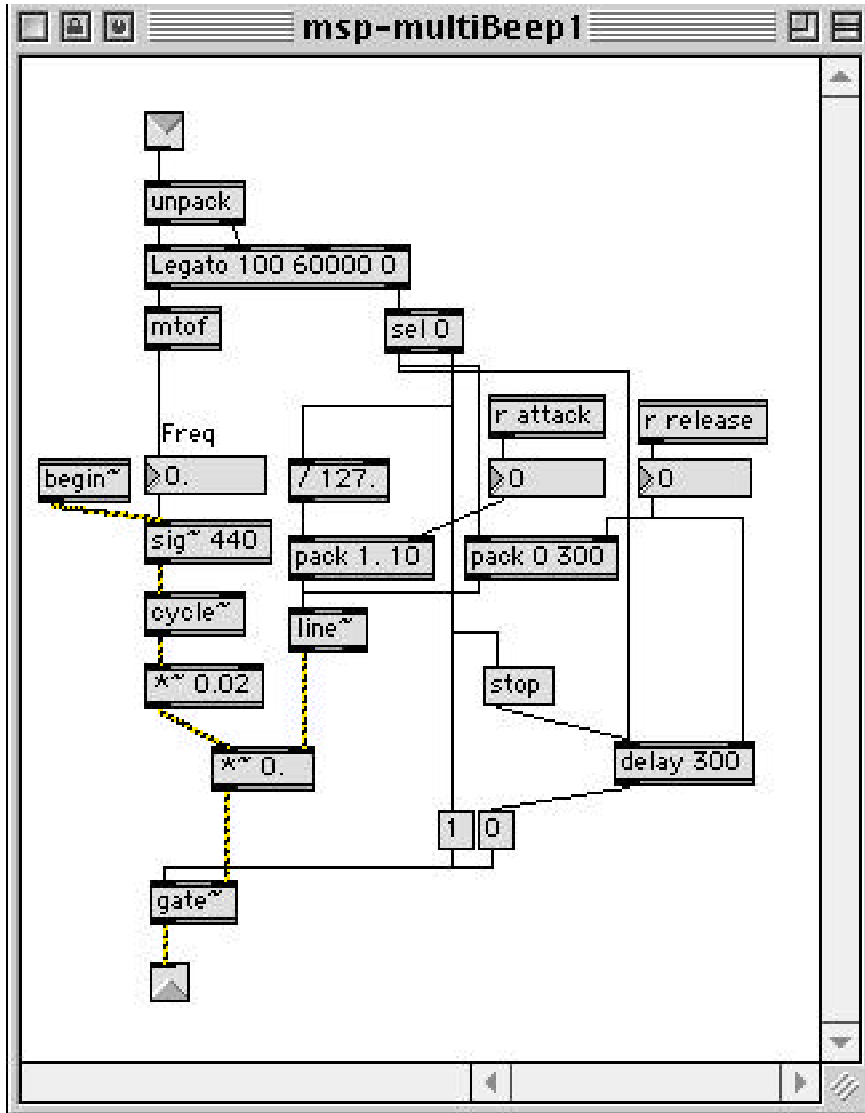
The only difference, besides using inlets and sends to get data into the subpatcher, is that I've broken the envelope into two packs so the note will sustain as long as the key is held down. Notice that signals can be sent through inlets and outlets, just like any other message.

There's one improvement that can be made on the player that is trivial at this point, but will become quite important when your patchers get large. When audio is active, all msp objects are doing their thing, taking up CPU time, whether they contribute anything to the final output or not. It's a good idea from the start to shut subpatcher down when they aren't needed. We do this with the begin~ and gate~ or selector~ objects. Begin goes at the start of the signal chain, like this:



We have to stick a sig~ object in between the begin~ and cycle in order to make the frequency control work. That's because begin~ puts out a stream of 0s when it's on. Sig~ will ignore this (It converts floats to steady signals) but cycle~ wouldn't.

The gate~ goes at the bottom of the signal chain. When it's open the signals are computed. When it's closed, everything between the begin~ and the gate is shut down. Here's the complete subpatch, with the logic needed to turn the gate on and off with the notes:



What does the delay object do here? And why must it occasionally be stopped?

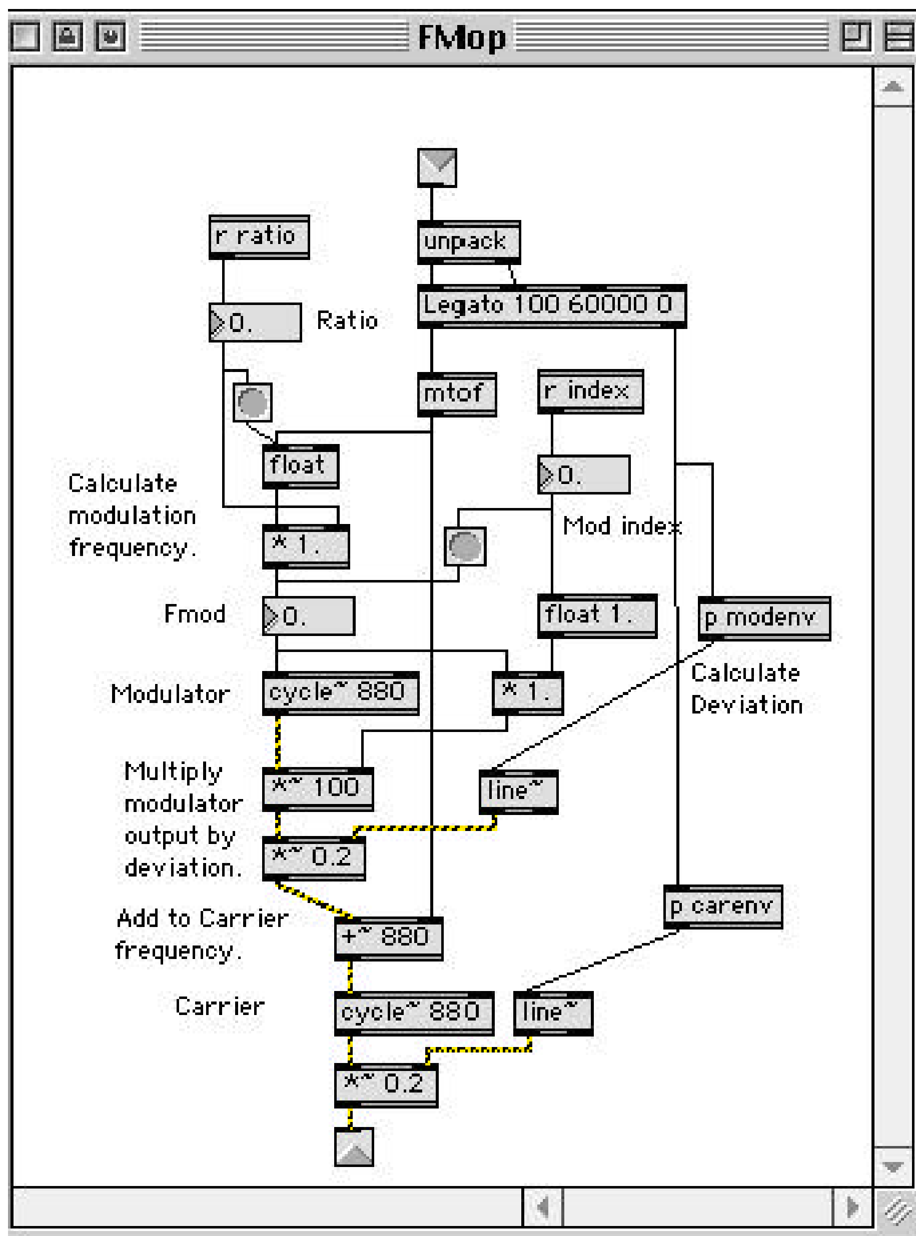
Getting Beyond Sine Tones

The waveform in cycle~ can be replaced with any wavetable you want, but there's two problems with that. One is that there aren't any other waves available (we'll have to get Csound out and make some), but the other is more serious. You can't produce things like sawtooh and square waves in a digital system with out exceeding the Nyquist limit and getting distortion. That's because these forms have very high harmonics that will exceed 22 kHz most of the time. Even filtering won't work, because the waveforms would have

to exist between the cycle~ and the filter. To get interesting sounds, we have to use other strategies.

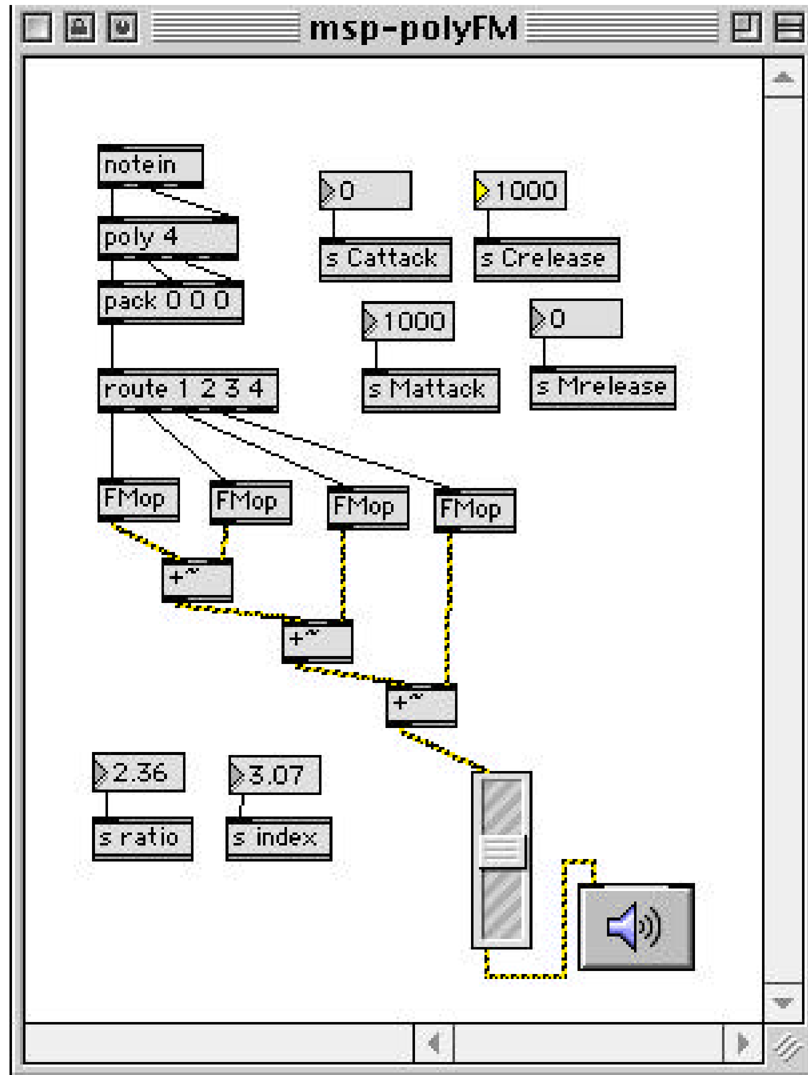
FM

The tutorial gives a pretty good demonstration of FM. Here's my version, which behaves much like the operators in the Yamaha synthesizers:



This gives direct control of the modulation ratio and the mod index and (via receive objects hidden in the modenv and carenv subpatchers) individual attack and decay for

carrier and modulator. This is designed to plug right into the polyphonic patcher, like this:

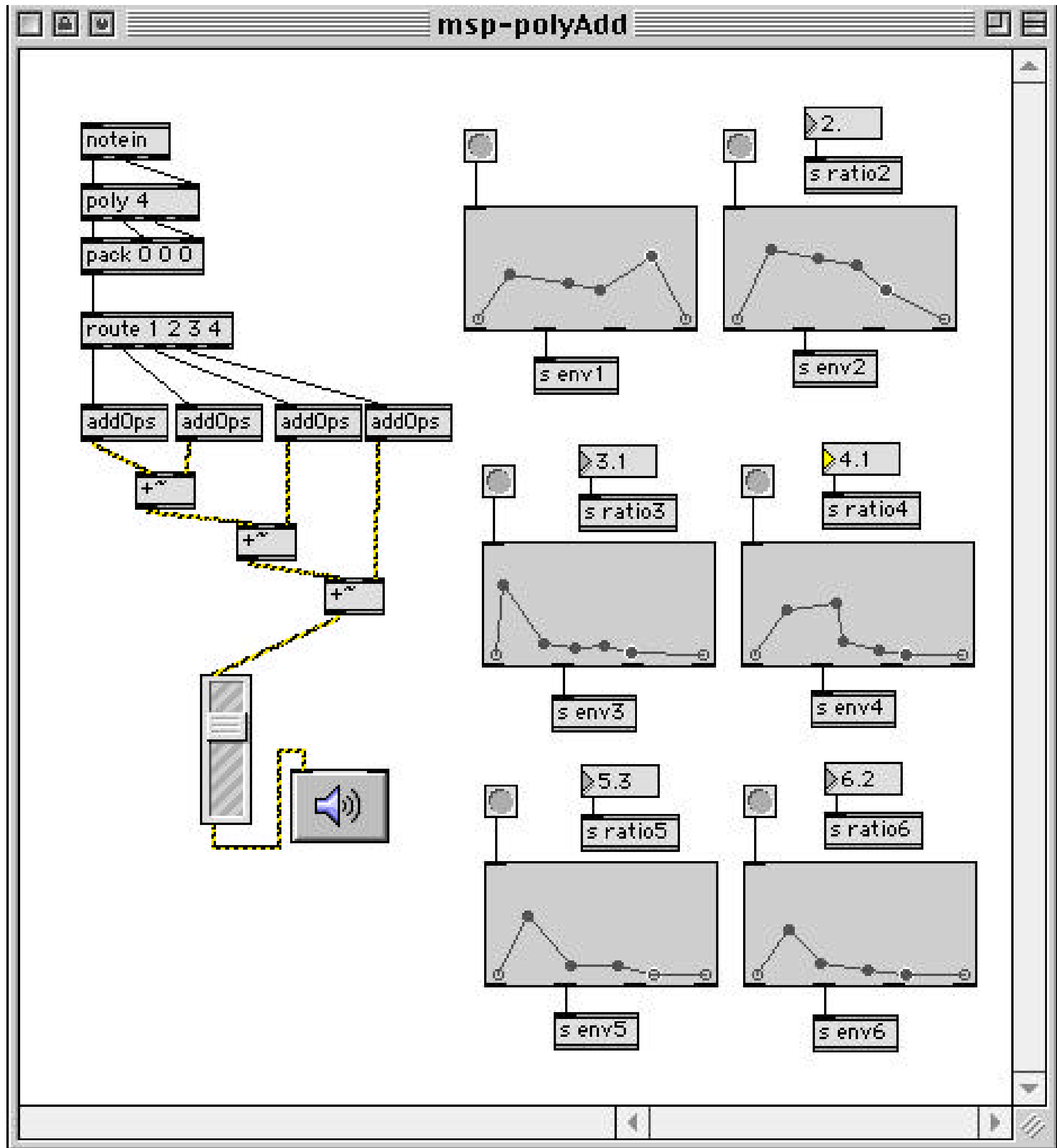


Polyphonic Additive Synthesis

The tutorial also explains additive synthesis very nicely, and uses it to introduce function~, an object that lets you draw envelopes directly on the screen. Unfortunately, function~ has some limitations. The way it's used in the tutorial patch, you have to bang directly on a function to get the sound, which makes polyphony awkward. Also, for some inexplicable reason, function~ sends the value of the first point separately before sending the rest as a list, usually producing a pop in the sound if envelopes over lap. Finally, although function~ allows you to fix the value of the first point (at for instance, 0) you can't fix the last point, because you don't know how many points there will be.

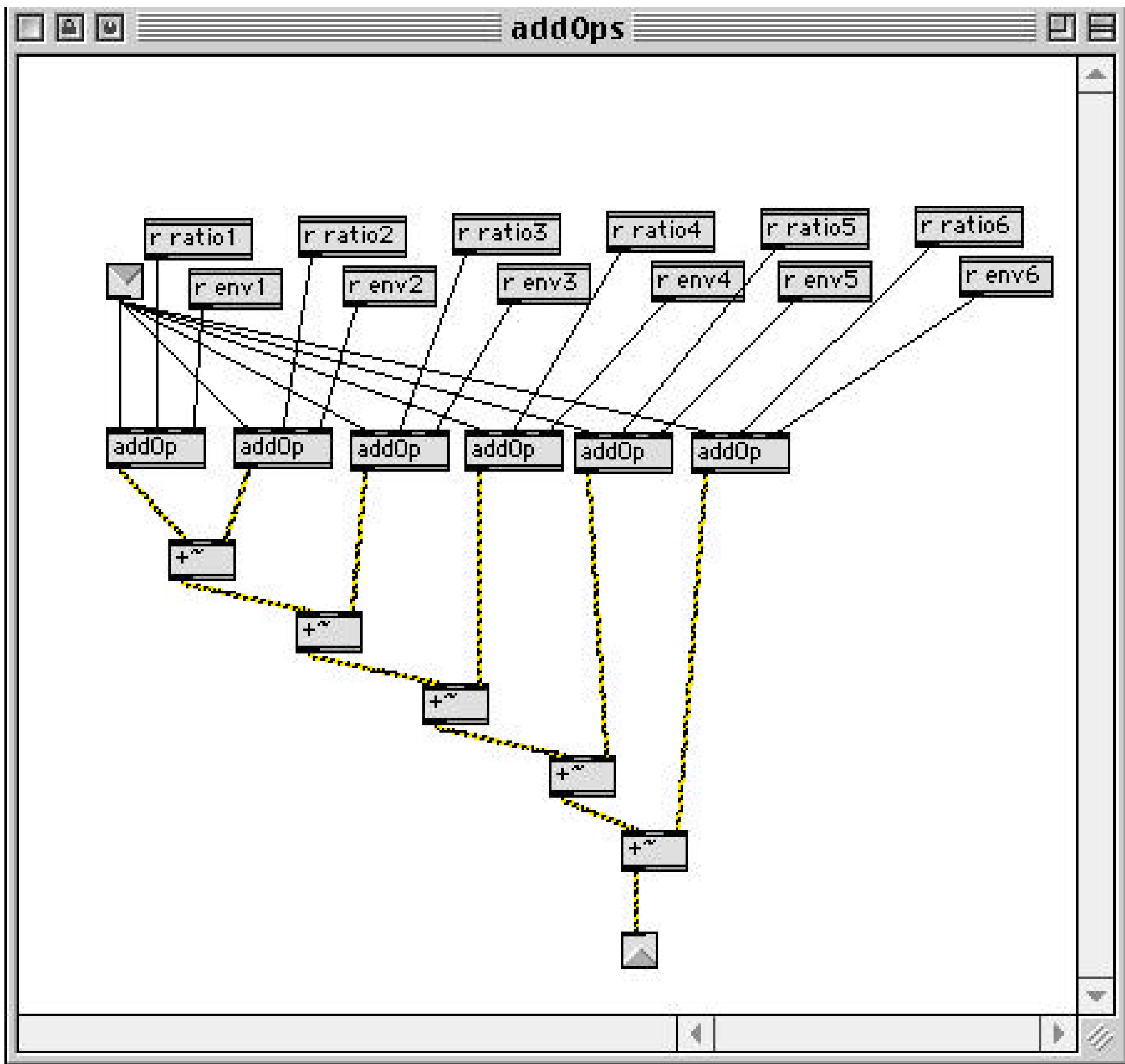
PolyAdd addresses these problems. The patchers are nested three deep:

The top:



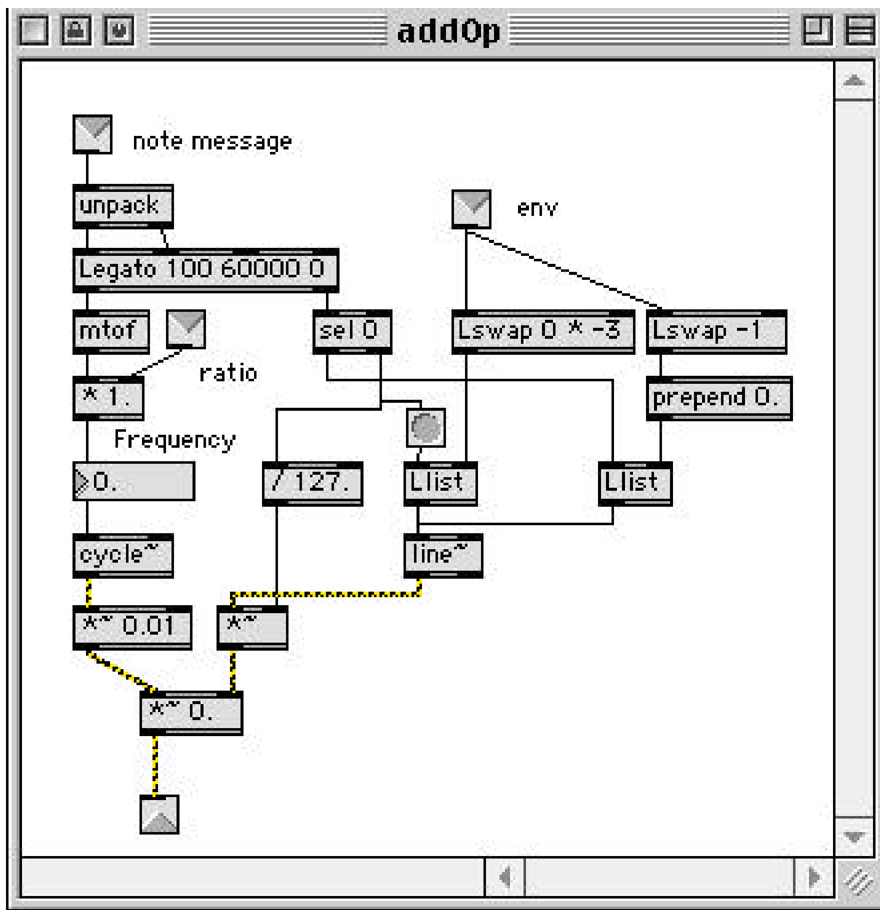
The function~ objects have a hidden connection from the right outlet to the button. This makes them send their envelope as a list each time a point is changed.

Next addOps:



The note message is passed to all of the subpatchers, and each gets their personal frequency ratio and envelope.

And finally, addOp



The envelop is broken into two parts by Lswap. All but the last two values are used to start the note. The final target value is discarded and replaced with a zero to make sure notes always shut off.

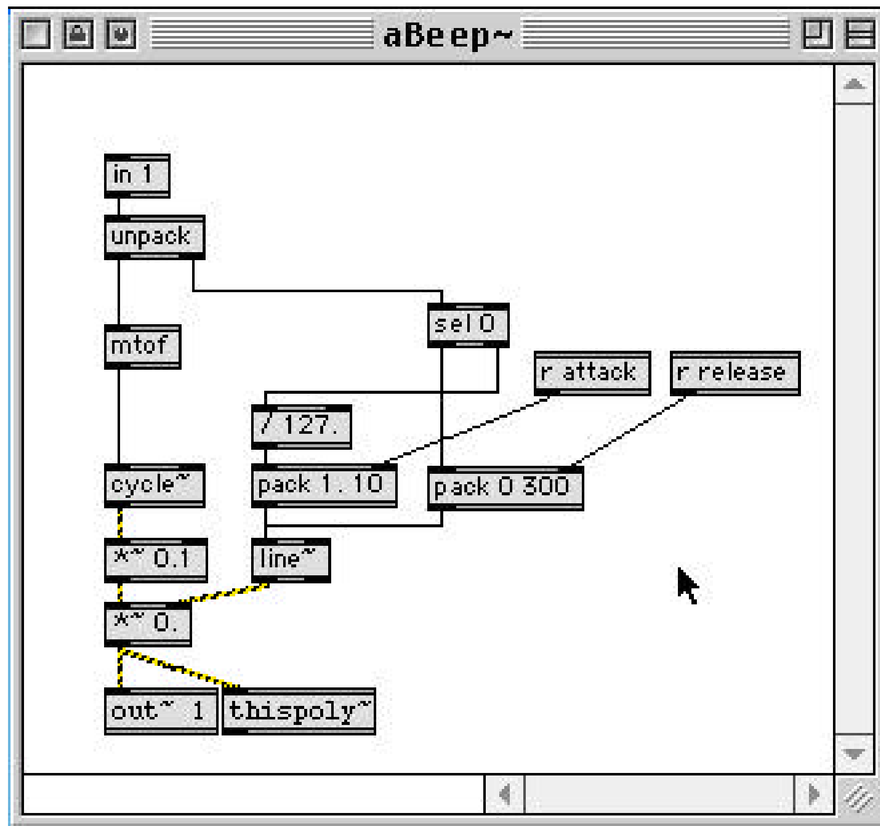
There are no begin~ and gate~ pairs in this example, but they would probably be a good idea.

Better Polyphony

MSP version 2 features the Poly~ object, which manages polyphonic instruments in a much nicer way. The Poly~ object is a "wrapper", which can include multiple copies of one of your patchers and control them in a polyphonic way. It works like this:

First, build an instrument.

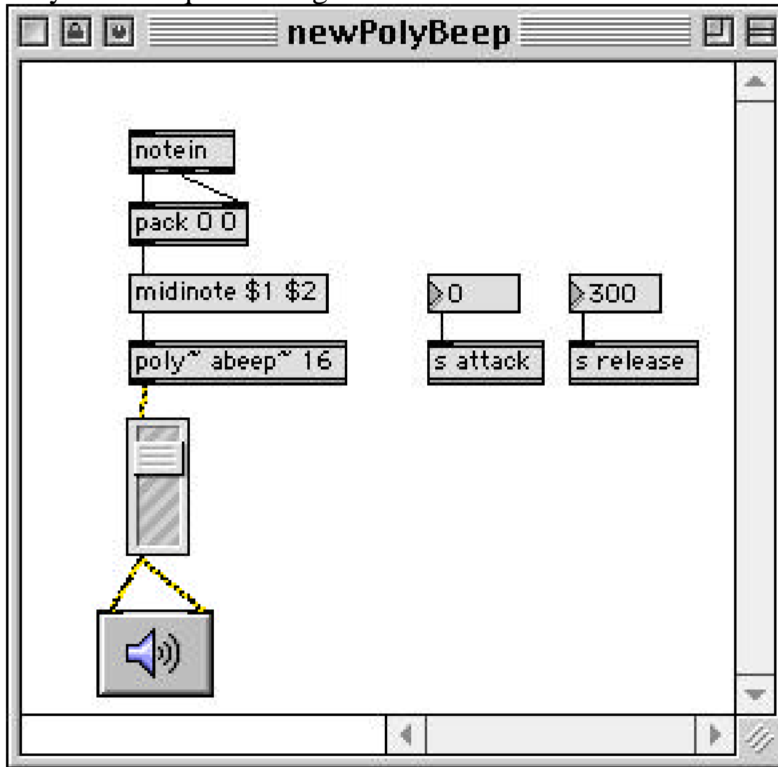
This is almost exactly like the subpatchers we used before. Here's one using the beep patcher:



The main difference is that in and out~ objects have replaced the inlets and outlets. The argument 1 for in means it will be connected to inlet 1 of Poly~. In accepts messages--use In~ to bring in a signal. Thispoly~ is the means the instrument sends information back to the poly~ wrapper. When thispoly~ receives a signal of 0, the poly~ will stop audio processing for the instrument. Thus there is no need for begin~ or mute~ functions.

Second, load the instrument into poly~

Poly~ takes two arguments: the name of your instrument file, and the number of voices you want available. A new instance of your instrument will be created for each voice. Poly~ will acquire enough inlets to match the ins and outs of the instrument.



To make a note sound, send the message **midinote** with the pitch and velocity. Poly~ will activate an unused instance of the instrument and send it the data. When the note off occurs, poly~ will send the pitch and a 0 velocity to the same instrument.

You could use the message **note** to start a note. This will send the arguments after note to the first unused instance. There is no tracking of sounding pitches with the note message, so it only works with instruments with predefined durations.

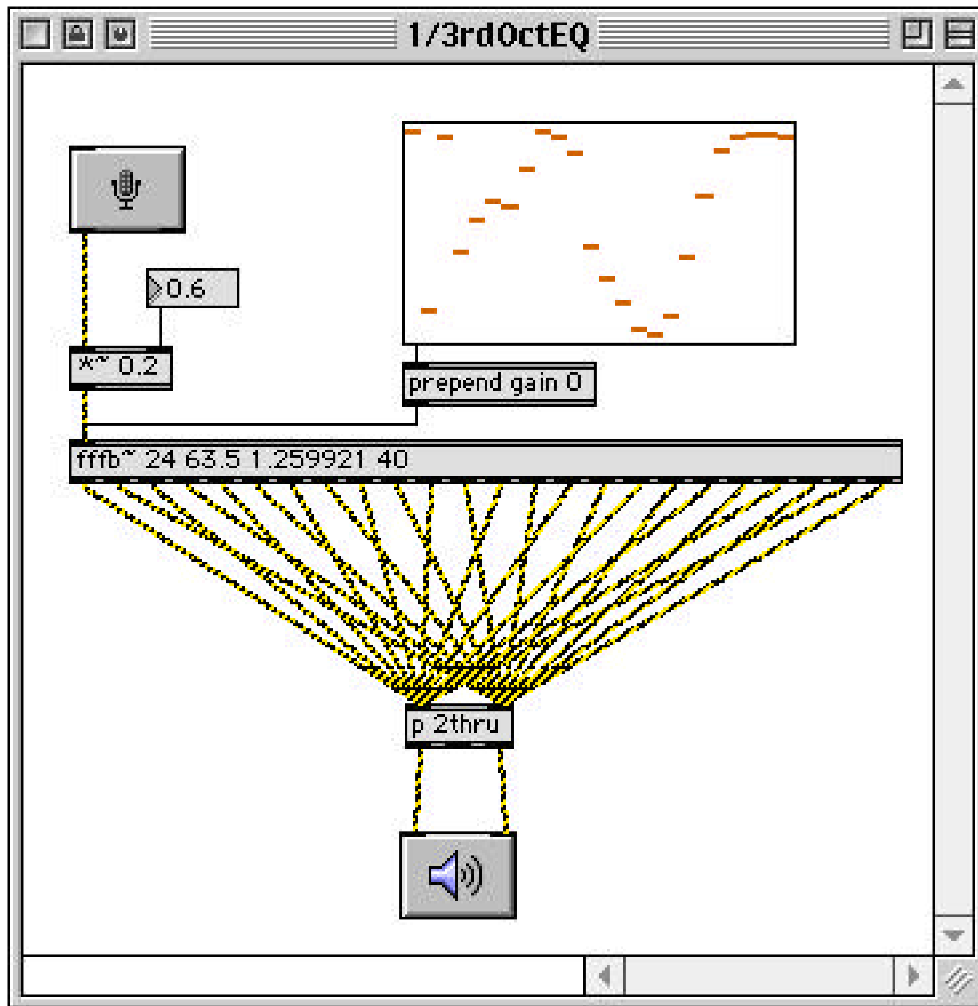
You can use send and receive objects to broadcast shared data to all of the poly instances or you can define inlets for the job. You can use the **target** message to route data to specific voices. For instance, once the message **target 4** is received, poly will route all data messages to instance 4. **Target 0** selects all instances.

You can send signals to instruments with in~. Signals are always sent to all instances. If an instrument patcher has both an in 2 and in~ 2 defined, the poly~ will route data sent to it second inlet to in 2 and signals at that inlet to in~ 2. If you have an out 2 and out~ 2 defined, poly will have 4 outlets, two for signals and 2 for data messages.

A Taste of Filters

MSP 2 features several nice filters. The most useful is probably lores~, which is a lowpass filter similar to those found on our modular machines. You will also find svf~ very familiar. This is a state variable filter, the circuit that gives simultaneous high, low bandpass and notch functions. Try adding these to the basic beep patches.

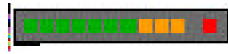
My favorite, however, is the fast fixed filter bank, the ffb~. This gives the sound of the old 1/3rd octave filter:



With MIDI control, if you like.
The arguments set the number of filters, the frequency of the first filter, the frequency ratio (here it's the cube root of 2), and the Q of the filters. Each filter has its own output, which you can sum as shown or use for independent processing. (The 2thru subpatch merely holds the outputs together so I don't have to draw 24 connections if I change the filter destination.)

Testing Things

Sometimes the hardest part of working with msp patches is to find out what is going on. The tutorial shows several common ways of measuring signals- these are nicely illustrated in the help files.



Meter shows level. It flashes red if the signal goes above 1.0.

Avg~ gives the level whenever it is banged. You usually see a metro hooked up to it.

Number~ periodically gives the value of the most recent sample at the right outlet. Like a sample and hold, it gives a funny pattern of numbers on audio, is most useful for slowly chaining signals, like the output of line~. When the arrow is showing, it creates a constant signal at the left outlet, which you set with the mouse.

Snapshot~ gives much the same thing, but reporting can be controlled by your patch (number~ is controlled in the get info window.)

Capture~ lets you see a chunk of signal as a series of numbers. Tedious to use, but enlightening when everything else fails.

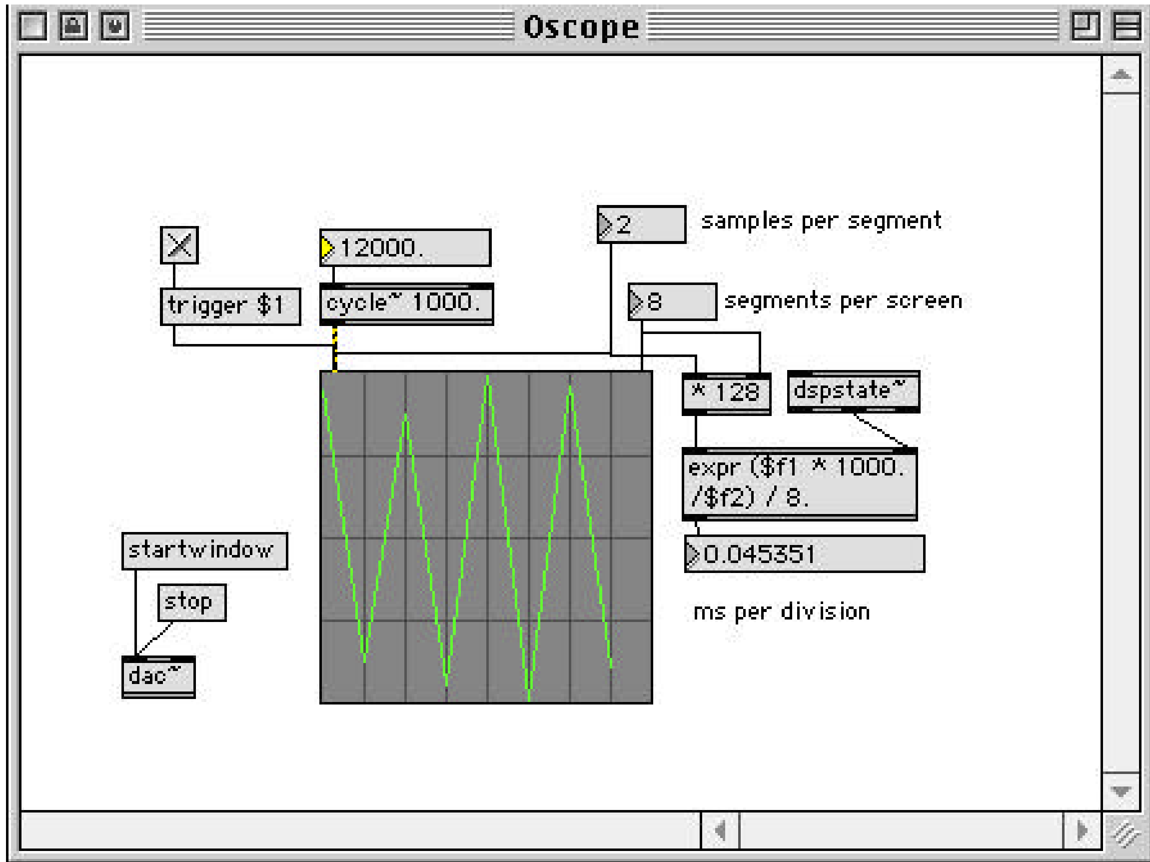
Scope~ Gives you pictures of signals. Its not much like an oscilloscope though, because what it really does is flash pictures on the screen at a rather slow rate. The help files and the tutorial don't really make the operation of scope~ very clear, so I'll have a stab at it.

The scope~ object captures the signal into a series of buffers that will be displayed on the screen. You can set the number of buffers by sending an int into the right inlet. You can set the number of samples per buffer at the left inlet. When the buffers are full, each is shown as a line segment from the value of the first to last sample in the buffer. (With the default display size, each buffer gets a single pixel in width, so the lines go straight up. If you stretch the display, you can see some slant.)

The total number of samples that will be on the screen is the product of number of segments per screen and samples per buffer. The defaults are 128 x 128 (16384) or about 1/3rd of a second at 44.1 kHz. This is suitable for showing low frequency waveforms, but the screen sort of jumps, and if you increase the frequency to the audio range, the display will probably just give you a line. To get an image of a high frequency tone, we need to display fewer samples. It's usually prettiest to keep 128 buffers and go to fewer samples per buffer. I usually start with around 8. If there are too many cycles showing, reduce the number. If you only see part of the wave, increase the number.

For high frequencies, reduce the number of segments per screen until you get a stable waveform. It may look strange, but then again the waveforms digital systems produce at high frequencies really do look like that.

With real oscilloscopes, you can set the sweep rate to a particular time per screen marking and use triggering circuitry to make the waveform sit still. The settings for scope~ are restricted to integer sample numbers, so we usually can't get convenient time settings. We can however, calculate the time per division we are seeing. This patcher shows how.



Dspstate~ gives the sample rate. With 8 divisions per screen and (samples per segment * segments per screen) samples on the screen, the expression object shown will give ms per division.

The trigger 1 message to the scope will stabilize the display if the other settings are in the right range.

Playing Sound Files

There are two basic ways to play sound files. With buffer~ (and its associated objects) which plays from memory, and with sfplay~ which plays files from the hard drive. There are advantages and disadvantages to each. Playback from memory is going to be instantaneous, but the total playing time you get is limited by the memory available in your computer. Remember that a minute of sound takes 5 megabytes of memory, twice that for stereo. There is no limit to the length of a file played by sfplay~ but there may be a slight delay in getting started, and the number of files you can have playing at one time is limited by the speed of your hard drive system. Most systems will easily do 8 tracks however.

Buffer~ and Friends

The buffer~ object is a holder for sound files. You have to give it a name and a size, then put some sound into it. The size is in milliseconds, and will be a two channel unless you specify how many in a third argument. Loading the sound is done with the read message. There are several variations:

Read	By itself read brings up a file dialog and you find the file in the usual manner.
Read filename	This will load in the file named… if the file is in the search path defined in the Max file Preferences option. Otherwise you have to specify the complete path, such as harddrive:myfolder:mysoundsfolder:filename. Note that folders are separated with colons. A colon at the beginning refers to the main drive.
Read filename offset	Offset is a number of milliseconds from the start of the file to begin reading.
Read filename offset duration	In addition, loads only part of the file.
Read filename offset duration channels	You can read only one channel of a two channel file if you want to.
Replace	reads a file and changes the buffer size to fit the file size.
Readagain	reads the last file, but with new options if desired.
Import	reads MP3 files.
Write	Write lets you save a buffer as a sound file.
Writeaiff, writesd2, writewave	These specify file type, saving a step.

If you double click on a buffer~ a little window pops up showing the sound file.

Play~

The play~ object plays what ever is in the buffer it points to (you have to specify a buffer name.) It's very primitive, rather like turning a record by hand-- you put a signal into it, and as the value of the signal changes, play~ produces sound. Usually the signal is derived from a line~ object, but you can use a phasor~ to play loops. All play actually does is convert the signal coming in, (which represents time in milliseconds) into a pointer to a sample to grab from buffer and sends that sample out. The help file gives a fine example.

Groove~

Groove~ is the most useful buffer~ player. It can be pretty confusing until you realize exactly what it does. It is also maintaining a pointer into the buffer~, but it has built in line functions to move the pointer for you. If you send a float to groove~ it will cue up the pointer, but nothing will happen until it gets a signal -- this signal determines the rate of play, so usually a sig~ 1.0 will do it. Changing the value from sig~ will change the rate of play-- it will even play backwards. After it plays once, you have to recue it to the beginning to get it to play again.

Groove~ has a loop mode, where it will recue itself. You can set the loop points. (The loop settings need to be floats.)

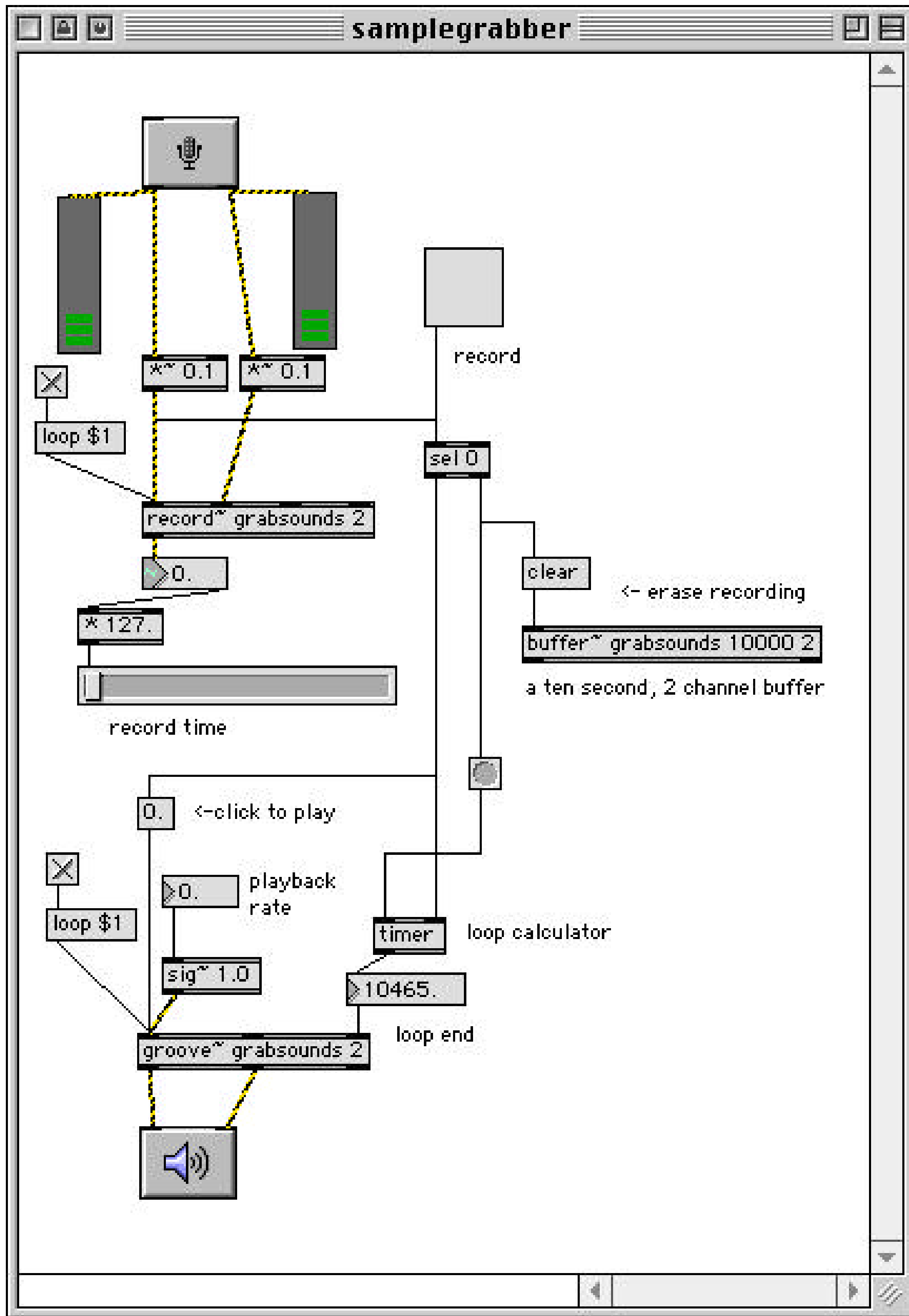
Record~

Record~ will put audio into the buffer~. You need to specify which buffer to use (the set message lets you change this) and number of channels if more than 1. Audio signals connected to the inlet(s) will be recorded when a 1 is received in the left. Recording is stopped with a 0. There are two mode flags for record:

Append	when off, recording always starts at the beginning of the buffer. When on, recording starts where it last left off.
Loop	when loop is off, recording stops at the end of the buffer. When on, when the end of the buffer is reached, recording moves to the beginning of the buffer, overwriting what is already there.

There are inlets to set the start point and end points in the buffer. These require floats that specify time in milliseconds. Record~ does not get a direct connection to its buffer~ It has a signal output that sends a location pointer. This could be connected to a play~ to synchronize playback from another buffer. This lets you patch multitrack features or set up a time remaining indicator.

Here is an example of a buffer in action:



Waveform~

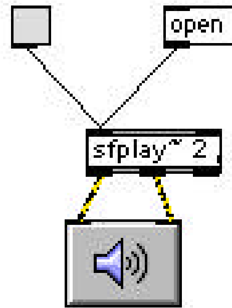
This is found in the tool bar. It shows the waveform that is in the buffer and lets you do some minor editing. Look at the help file for instructions. Note you have to use a message to set which buffer it is linked to.

Sfplay~

Sfplay~ plays sound files from the hard drive. The file can be in practically any format, although paired files (such as made by pro tools) will require two sfplay~s. Arguments to sfplay~ are:

Name of an sflist~ object ,	This is optional. If there is one, you can load several files and have them ready for instant playback.
Number of channels,	up to 8, apparently.
Size of play buffer	in milliseconds. A play buffer is necessary for smooth disk operations. If you put 0, the default size is used, which is usually fine. Adjust buffer size if you have slow disks that stutter when you play.
Number of position outlets	the first position outlet gives the time in milliseconds. This is rounded off from the actual sample locations. A second position outlet gives the round off error, so you add them to get the precise time. The help file shows how to convert this information into a time display.
Name	you can give the sfplay~ itself a name. Sfinfo~ can use this name to return information about the current file.

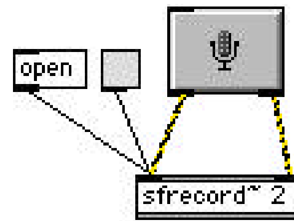
The reference on sfplay~ is 6 pages long, and it has extensive help files. The easy way to use sfplay~ is like this:



This will play a file for you. If you want to use the file like a bank of samples, you can define locations as cue points with the preload command. Once a location has been defined as cue 3 for instance, a 3 (as opposed to a 1 which starts at the beginning) will start playback there. You can create and save complicated lists of cues with the sflist~ object. Sflist~ lets you have cues from more than one file. Since each cue has a buffer to start playback instantly while the disk catches up, there is a memory cost for using cues. Figure about 40k per channel.

Other commands like speed, seek, pause and resume make playback interesting.

Sfrecord~



Recording can be just as simple as playback. With a patch like the one above:

- Open a file to record into
- Send in a 1 or the message [record length] to begin recording.
- Send a 0 to stop.

There are options for various file formats and so on.

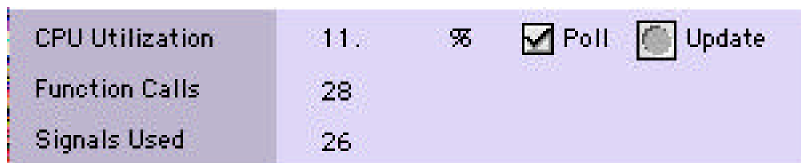
Sfinfo~

Sfinfo~ provides information about selected audio files. Usually you want to know how long a recording is, and confirm that its sample rate is appropriate for current settings.

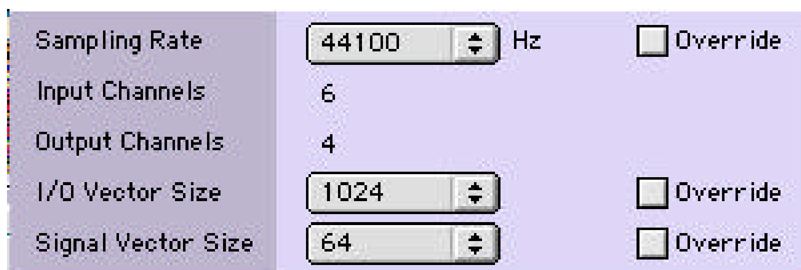
The DSP Status Window
MSP operations can be monitored by the DSP status window, which is found under the options menu. There are several important settings in this window.

This section allows you to select the sound card and synchronization source. You won't usually change the sound card, since the choice will only be Audiomedia 3 or sound manager. (You cannot route sound manager to the AM3 and then MSP to sound manager. The system will crash if you do this.) Soon I hope to replace the AM3s with something that works better.
An interesting choice for driver is "nonRealTime". Which lets you compute complex processes and listen later, a la Csound. You may use this when doing fft operations.

The clock source can be internal or, SPDIF if you are bringing sound in from the CD. If SPDIF is selected and the CD is turned off, you won't hear much.
Prioritize MIDI means deal with MIDI output before processing audio. You may want to set this if you notice odd timing in MIDI notes.

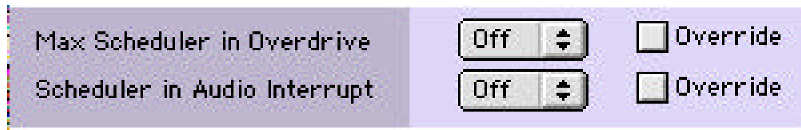


This sections tells how the computer is doing. Audio processing is pretty hungry, and if CPU usage gets above 70% or so, other functions of the computer, such as processing keyboard input will suffer.



You can change the sample rate here. The other settings will be automatically set for the chosen sound card, but you may want to increase signal vector size if the CPU is overloading. On the other hand, the in/out delay is directly related to this size, so smaller is better. (On the AM3 there isn't much choice.) If you check override when you change

any of these, the changes won't be remembered in the preferences file. That's the polite thing to do.

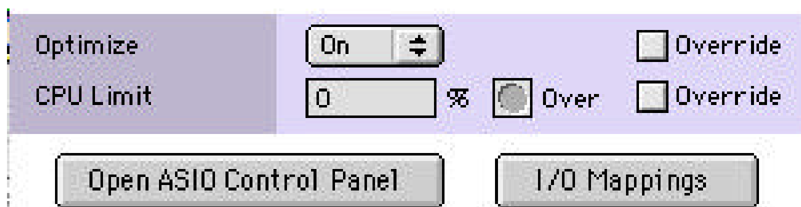


The Max scheduler is the master clock that processes all non audio activity. When it is placed in overdrive, all calculations happen at interrupt level, without stopping for slow processes like screen redraws. In the old days, when computers were pokey, we kept this on but it doesn't make a lot of difference on 300+ Mhz machines.

The Max scheduler is only accurate to within 1 millisecond. If you want timing tighter than that, putting the scheduler in audio interrupt will get it down to once per vector. (divide vector size into sample rate to figure out how often that is.)

Check override when you change these.

Here you can change inputs and outputs. Note that you can mix analog and SPDIF if you want to.



Optimize only works on G4s (it turns on altivec). Setting a CPU limit may help if your patcher is getting so big you lose control of the machine.
The AISO control panel lets you set extra parameters for the sound cards, like input level.
I/O mappings lets you define more than 2 input and output channels, and connect them to actual channels. If you want to call an output 42, you can hear it if you map 42 to something that exists.

How it all fits together

Although listening to audio and watching the screen gives the impression that everything is steadily chugging along, the system is really backing up and stretching out like cars on a crowded interstate.

There can only be one sample rate in effect at a time. This is determined by the sound card settings. Assume it's 44.1k for this exercise. That means the card needs a sample (X the number of channels) every 0.0226 ms. To make sure there's always something there, the program can run ahead of the card, up to whatever the output buffer size is, 1024 on the one I use.

The MSP audio interrupt is loosely locked to the audio card. The audio interrupt runs at (Sample rate)/ (signal vector size) . If the vector is 64, an audio interrupt happens approximately every 1.45 ms. The duration of the audio interrupt depends on the amount of processing necessary for every audio object to do its thing on one vector. Naturally, if it takes longer than 1.45 ms, the output buffer will start to drain. You can get away with a couple of long vector periods (like when something turns on or goes the long way around a branch), but they better be balanced by short ones. If the audio processes get ahead of the card, the card will say so, and an interrupt is skipped.

Note that there is a buffer for everything that is contributing signals to the system. The card input, any files being read, all put their data into a buffer, and on the audio interrupt, a vector's worth of samples are taken out. If the buffers are too big, input to output takes too long, but if they too small, audio processing is broken up. (The buffer~ object is an extreme example of this, as it contains the entire audio recording.)

Also note that for proper playback, any recording should be played at the sample rate in use when it was made. If the current sample rate is different, the program has to do some interpolation to compensate. MSP is doing this anyway for variable rate playback. When you specify sample rates for recordings, you are telling MSP the original SR so it knows what to do. Sample rates are marked in audio files, but it's easy to get them messed up in some programs.

When processing time gets tight, it would be nice to skip some calculations, especially if they are just giving the same number over and again. You have to do this in a closed off part of the system that is running at half the sample rate. Any signals entering this area have to be decimated (downsampled) on the way in, and resampled back up on the way out. The poly~ object is a nice enclosed area where this is possible, and often necessary. In a leisurely patch, some things may sound prettier if run at twice the sample rate. I'll be interested in finding out what these are.

That's the bottom layer- Next we find the old Max scheduler. It can be run by the Mac internal clock (which is a bit pokey and the source of years of complaints) or, for accuracy, can be tied to the audio interrupts. The scheduler ticks over once a millisecond.

Note that this is faster than the audio interrupt, but on most ticks it has nothing to do. When there is something, such as a metro to bang, everything attached starts happening in a long involved chain. If it's not finished by the next audio interrupt, the CPU puts a bookmark in and does the audio chores. That's why it's called an interrupt. When the audio chain is through, scheduled tasks resume. (The pecking order of the scheduler and audio can be reversed with the prioritize MIDI option)

There's an even slower layer. Every once in a while, after scheduled tasks are done, the program asks the event manager (part of the operating system) if the user has typed a key or clicked or something. If so, another long processing chain is kicked off, which can be interrupted by audio, or if overdrive is on, scheduler tasks and MIDI input. After this sort of thing is done, Max relaxes for a moment to let the OS catch up on outside work like sending you messages about your network. This is all interruptible, which is why the system will lock up if you pile on too much audio.

Max also keeps a list of things to do when the cows are milked and the chickens fed. These are things like file operations and graphics drawing. You have often seen the effects of this on number boxes and the like. You can move any part of a patch into this zone with the defer object.

This is beautiful system, but it makes some things difficult and unpredictable. Here are some questions to ponder:
What if a scheduler task needs to know the current signal value? Which of the 64 numbers in the current vector does this mean?
How is the operation of graphic sliders affected by audio load?
Why the sig~ object?
Why doesn't Signal Scope behave like the one on my bench?
How can midi controllers produce smooth audio effects?

Now Go Read the Tutorials

There's a lot more to MSP than explained here. The tutorials are pretty good, even if they are only pdf files. For general understanding of what is going on, get a copy of the Computer Music Tutorial by Curtis Roads.