Developing Plug-ins in Max/MSP for

pluggo

PRELIMINARY INFORMATION FOR PLUGGO 2.0

Cyding '74
1186 Folsom Street
San Francisco, CA 94103 USA
tel (415) 621-5743 fax (415) 621-6563
info@cycling74.com www.cycling74.com

revision 3 of 14 Dec 1999

Copyright and Trademark Notices

Credits

Runtime plug-in environment and Development Documentation: David Zicarelli

Plug-ins and Support Objects: jhno, Adam Schabtach, Joshua Kit Clayton, Les Stuck, Richard Dudas, David Zicarelli

Graphic Design: jhno, Lilli Wessling, Adam Schabtach

pluggo Character: Richard Dudas

Contents

Contents

Introduction

Plug-in Development

This manual describes how to use the audio plug-in construction tools in Max/MSP. These tools consist of nine Max/MSP objects, an application called *Plugmaker*, and a shell, which we call **pluggo** that loads plug-ins built in Max/MSP and creates an interface for them that appears to a host audio program as an audio plug-in. In this manual, we refer to the **pluggo** shell as the *runtime plug-in environment* and the audio program that can make use of **pluggo** as the *host mixer* or *host sequencer*.

pluggo can be downloaded from the Cycling '74 web site www.cycling74.com. When downloading **pluggo**, you need to choose the *Pluggo Installer for Max/MSP users*, which installs the special plug-in construction objects into your Max folder (as well as the development tutorials and other materials). You'll also need a host sequencer application for testing your plug-in, because the runtime plug-in environment does *not* run inside Max/MSP. That's right: even though MSP has a **vst~** object, this object cannot currently load VST plug-ins made with Max/MSP. Demo versions of the popular sequencing applications *Vision*, *Cubase*, *Logic Audio*, and *Digital Performer* can be obtained—with varying degrees of difficulty—from their publishers and distributors. These demo versions make ideal platforms for testing because they don't disable the Macintosh debugger Macsbug. So if you manage to create a plug-in that crashes, you might be able to get some idea of what's going on.

The runtime plug-in environment currently runs under MAS and VST host environments and provides a platform-independence so Max/MSP developers may write plug-ins that work under all supported environments. Support for other real-time audio plug-in host environments is currently under consideration.

How This Manual Is Organized

This manual assumes familiarity with **pluggo** as well as Max/MSP. Before developing your own plug-ins, we think it's useful to see what has already been done so if you haven't tried out the collection of plug-ins included with **pluggo**, we recommend doing so before reading any further.

After this brief introductory chapter, we present five Tutorials that will serve as an introduction to developing plug-ins using Max/MSP. These tutorials cover the process of constructing Max patches that will work as plug-ins, and they assume you're familiar with programming in Max and MSP. After the tutorials, we briefly discuss the Plugmaker application that transforms Max patcher files into VST plug-in files.

Next, you'll find Reference Pages that cover the plug-in construction objects, organized in a format that will be familiar to readers of the Max and MSP manuals. These reference pages are supplemented with object help files that are found in the plug-in development materials folder associated with this manual.

Following the reference pages, we present information about two evolving "conventions" adopted by the plug-ins included with **pluggo**, a synchronization scheme called *PluggoSync* and an inter plug-in audio communication scheme called the *PluggoBus*. By conforming to these

standards, you can synchronize your plug-in to the tempo of the music and allow it to send and receive audio signals.

The last chapter of the manual describes differences between Max and the runtime plug-in environment that you should take into account when designing your plug-in patcher.

About VST Plug-ins

The plug-in development model most closely follows the VST plug-in specification developed by Steinberg Soft- und Hardware GmbH. It is available from the Steinberg web site www.steinberg.net or www.steinberg.de. You won't need to know anything about the specification in order to develop plug-ins using Max/MSP, but it's helpful to be familiar with a few basic elements of plug-in jargon. VST defines a standard for plug-ins that process audio in real time. Most of the programs that currently host VST plug-ins are audio sequencers similar to Cubase, the first program that hosted the plug-ins, although this may not necessarily be true in the future.

VST plug-ins must have a process routine. This is the DSP code that transforms boring audio on input into audio output so wonderful you think people will pay for it. You can "construct" this routine using MSP audio objects, along with two special objects (**plugin**~ and **plugout**~) that define the audio signal interface between your MSP patch and the plug-in's host. We'll talk more about **plugin**~ and **plugout**~ in the first Tutorial chapter.

Parameters

Most VST plug-ins also have *parameters*. These are named values that the host mixer can change via automation or with its default interface that assigns a slider or knob to each parameter. If you make a VST plug-in that opens its own edit window, it is responsible for displaying and editing any parameters it has. Parameters are changed while the audio processing routine is running, so you get immediate feedback on the effect of a parameter change. When a VST plug-in loads, it tells the host mixer how many parameters it has. This value cannot change during the life of a plug-in.

Programs

VST plug-ins can store a collection of parameters called a *program* (we'll refer to these as *effect programs* in this manual to distinguish them from the many other meanings of the word "program"). Among popular host sequencer/mixer environments, *Cubase* and *Vision* provide users with the ability to use effect programs, but *Logic Audio* did not until version 4.1. Digital Performer versions 2.6 and higher (using MAS 2.0) allow for presets that are stored outside of the plug-in, unlike VST effect programs whose contents is the plug-in's responsibility.

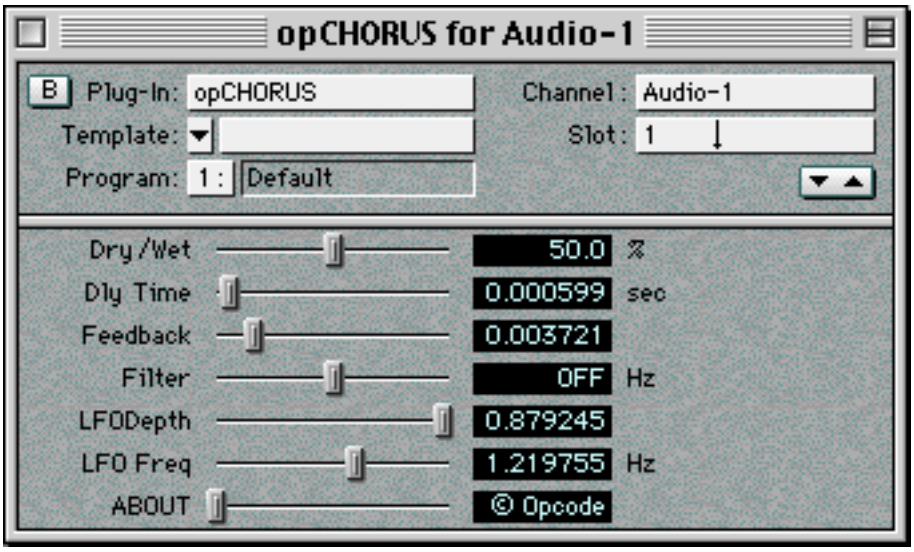
When a VST plug-in loads, it tells the host mixer how many effect programs it has. The plug-in itself is responsible for storing its own effect programs, and all of the plug-in's effect programs must have the same number of parameters. In MAS, the effect programs are exported to the host when the plug-in loads, and the host stores the information. Effect programs also have names. As you'll see in Tutorial P2, the **plugconfig** object allows you to store named effect programs full

of parameter values within your patcher. This provides a way give the user a taste of what your plug-in can do. There is no way for a user to modify permanently the contents of the effect program data within a plug-in. However, the parameter changes of all effect programs are typically saved in host sequencer documents. What's more, most hosts have the ability to load and save files of effect programs.

When a parameter in a plug-in is changed, the change is stored directly into the current effect program. Unlike a typical MIDI synth or effects unit, there is no "edit buffer." At least that's the way the original VST plug-ins behaved, so that's now what users expect. In any case, the storage and recall of programs and parameters is not something you need to worry about. The runtime plug-in environment handles it all for you as long as you use the parameter definition objects **pp** and **plugmultiparam**. In MAS, there is effectively an edit buffer the host writes over it when new effect programs are chosen.

User Interfaces

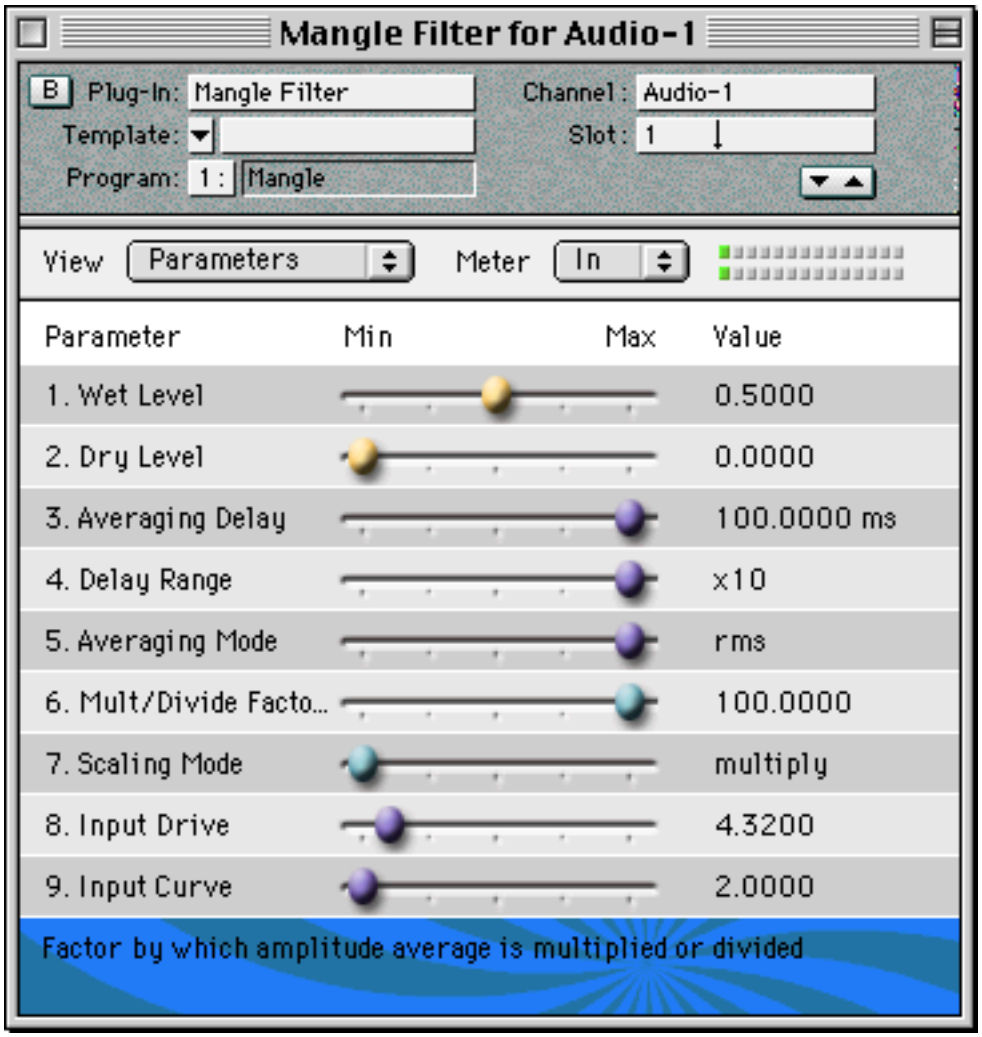
The plug-in developer has two choices regarding a user interface. You can count on the host mixer to display your parameters for you (using what we call the *default interface*), or you can create your own custom edit window. Below you can see a picture of a typical default interface for a VST plug-in—this one is from Vision.



Here is an example of a plug-in from Steinberg that has its own editing window.

Introduction

You can go either route with the runtime plug-in environment, although all the plug-ins included with **pluggo**—as well as the ones described in the Tutorials—use an edit window. If you decide to go the edit window route, you have an additional choice with the runtime plug-in environment: will you use the standard *egg slider interface* (also referred to in this manual as the *Parameters View*)?



Or will you create your user interface in Max?

The Tutorials will show you examples of both types of user interface. And by the way, it's possible to make a plug-in with both an egg slider interface and a custom Max interface. Users can switch between the two interfaces using the View menu.

How the Runtime Plug-in Environment Works

The runtime plug-in environment has two big pieces and a lot of small pieces. The small piece is a Macintosh "code resource" while the two big pieces are shared libraries called *MAXplugLib* and *Max Audio Library for Plugins*. You can think of the former as Max (or MAXplay) for plug-ins, and the latter as MSP for plug-ins. The MAXplugLib file also contains the user interface tools to display egg sliders for plug-in parameters as well as the interface linking a Max patcher to the host mixer environment. It's likely that there will be many revisions of MAXplugLib over the course of the **pluggo** life cycle, but there will probably be relatively few revisions of Max Audio Library for Plugins.

The *Pluggo* plug-in shows a standard file dialog when it is inserted. This plug-in is essential for development when testing with VST hosts. Among the files that Pluggo can open are Max patcher and collective files, so *Pluggo* allows you to test your patcher directly within the host sequencer. When you've finished developing your patcher and you want to make it into VST plug-in that will open when you choose its name from the host mixer's effect menu, you use the *Plugmaker* application.

Introduction

A plug-in made with Plugmaker contains the same code resource as the *Pluggo* plug-in (although it has been renamed from "Pluggo" to the name of the patcher or collective file that was given to Plugmaker). This resource is of the type that a VST plug-in host expects to load—it's Macintosh resource code is 'aEff' (for "audio effect"). In a Plugmade plug-in, the code in the aEff resource notices a patcher in the file and tells the runtime environment to load the patcher inside the plug-in file rather than ask the user for a file.

When testing under MAS, it is not necessary to use the Pluggo plug-in to load Max patcher files. Instead, you can simply drag these files to the VstPlugIns folder found within the MAS client application's folder. Then either restart the client application, or, in Digital Performer, choose MIDI Only from the Audio System submenu of the Basics menu. Then choose MOTU Audio System from this same menu, and the list of plug-ins will be updated.

You'll find that most everything about the runtime behavior of your plug-in patcher within Max will carry over into the plug-in environment. Your patcher interface will operate as you expect it to and most (but not all, see Appendix A) Max objects and capabilities are available to you. However, a plug-in is a guest in someone else's house, and there are definite restrictions in the area of user interface, file loading, and initialization that are mandated by the need to adhere to the VST standard.

One major difference between a patcher running in Max/MSP and one running in the runtime plug-in environment is that while you will probably want to maintain the individuality of your plug-in patcher, all the plug-in patchers are loaded within the same runtime space. As an example, if you want to send control data between one plug-in and another, you can use the **send** and **receive** objects. But if you want to use these objects and *not* make your data potentially available to other plug-ins, you'll have to use special symbols that create private communication channels within a plug-in patcher.

A major difference between MSP within Max and the runtime plug-in environment is that each plug-in has its own DSP chain—the sequence of calls made for each block of samples that define the signal processing algorithm. By contrast, in Max, all signal objects in all patchers currently in memory share a single DSP chain. One effect of this is that **send**~ and **receive**~ do not work to send audio between plug-ins. Instead, you have to use the new **plugsend**~ and **plugreceive**~ objects.

Please refer to the *Runtime Issues* chapter for more detailed information on these and other differences between Max and the runtime plug-in environment.

What Plug-ins Can and Can't Do

An audio plug-in has a highly restricted world view. It is passed input and output vectors of samples, and must process the input vectors and copy the result to the output vectors. Typically, plug-ins are used for filters, compressors, delays, and reverbs. Users probably expect that what you develop will be used to enhance some other source material that's already been recorded, so it's unclear whether sequencer users are going to appreciate a plug-in that ignores its input, producing unrelated output. But you may not care what others will appreciate. The VST 2.0 and

MAS standards allow plug-ins that can receive MIDI information and produce audio output from it (i.e., synthesizers). In Cubase 4.1, these plug-ins are referred to as VST Instruments.

But if audio is not what you want to process, it's OK to make a plug-in that contains no MSP signal objects at all. When the run-time plug-in environment notices there is no DSP chain after loading a patcher, it runs a default processing routine that simply echoes (or adds) the audio input to the audio output. You'll see why you would want to make a non-audio effect in Tutorial P5.

Among others, the MIDI objects in Max are not available in the runtime plug-in environment. In VST 2.0 and MAS, MIDI input and output to and from the host is possible and the special objects **plugmidiin** and **plugmidiout** are used for this purpose. These objects do nothing in a host environment not capable of sending MIDI to and from plug-ins. Appendix A lists the complete set of kernel objects and support functions that are not supported.

Another major restriction of the runtime plug-in environment is that since the VST 1.0 standard only allows a single fixed-size window for the user interface, opening more than one window is not allowed. It's even dangerous to open a dialog box. The environment does support scrolling the patcher window to set positions, called views, that the user can select using either a pop-up menu above the interface or via some control within the interface itself.

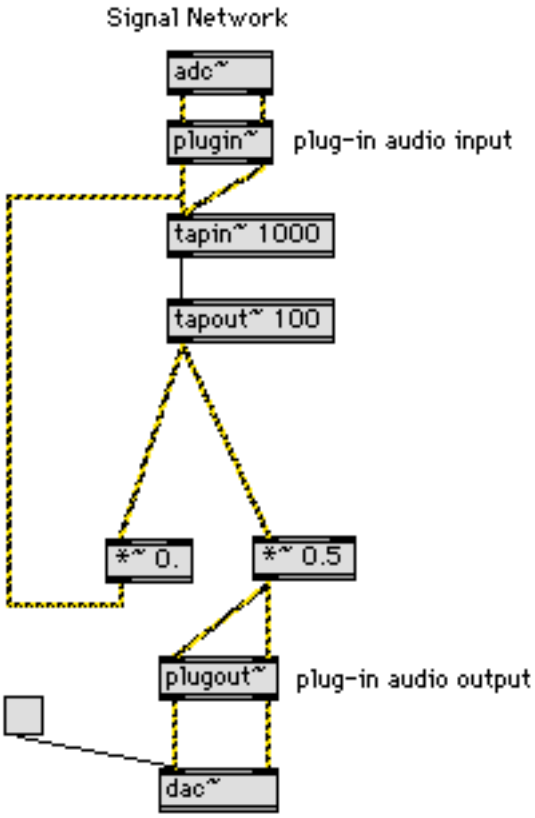
Tutorial P1 *A plug-in with an egg slider interface*

In this tutorial, we will look at a simple delay line that uses the runtime plug-in environment's Parameters view in its edit window.

Open the file called Tutorial P1 in your plug-in development materials folder.

Plug-in Inputs and Outputs

Let's look first at the part of the patcher labeled Signal Network. Turn on the audio by clicking on the check box near the **dac~** object. The patch will echo MSP's audio input to its output, adding a copy of the input signal delayed by 100 milliseconds.



As you examine the signal network, you'll notice two new objects, **plugin~** and **plugout~**. If you're suspicious that these two objects appear to be doing absolutely nothing to the sound, your suspicions are correct. **plugin~** and **plugout~** have a dual personality. In Max, they merely echo their audio input to their output. But within the runtime plug-in environment, **plugin~** serves to define the source of input to the MSP signal network, while **plugout~** defines the output of the MSP signal network that will be fed back to the host mixer.

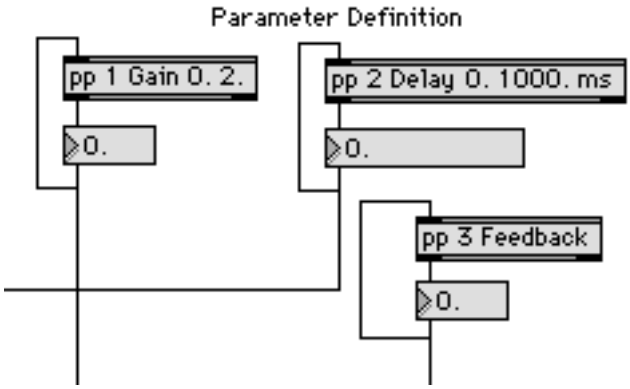
Why were these new objects created instead of using **adc~** to define the plug-in's input and **dac~** to define the plug-in's output? Because we anticipated that you'll want to try your plug-ins with a variety of test signals in addition to the audio input to the computer. If **adc~** were the plug-in's input and you wanted to develop the plug-in using a file played by an **sfplay~** object, there would be no way to run the same patch within both Max and the runtime plug-in environment. You can feed any signal to **plugin~** and hear it processed in Max, but when you load the plug-in

into a host mixer application, the signal will not be audible because **plugin~** ignores its input when loaded into the runtime plug-in environment.

plugin~ and **plugout~** have two inlets and outlets, representing the left and right plug-in inputs and outputs. Currently, all Max/MSP plug-ins within the runtime environment are stereo, but they can be used in a mono context. This may change in the future—**plugin~** and **plugout~** might acquire the ability to have more inputs, or optionally work as mono. For the moment, if you want your plug-in to be mono only, take both inputs from **plugin~** and add them together at the start of the processing chain, as we've done in this patch. You may also want to multiply the combined signal by 0.5 so that your input isn't doubled in volume.

Defining Plug-in Parameters

Now look at the part of the tutorial patch labeled *Parameter Definition*. Here you see several **pp** objects. **pp** is short for plug-in parameter. In developing plug-ins, you'll be using a lot of these objects, so we've spared you the inconvenience of having to type in a long object name (**pp** used to be called **plugparam**).

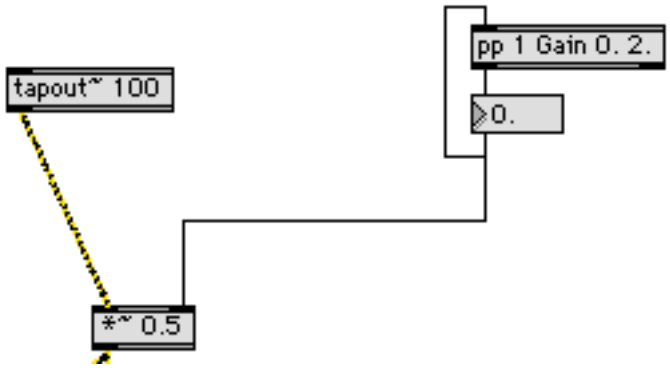


Each **pp** object defines the parameter number, name, minimum, maximum, and the units to be displayed next to the parameter value in the Parameters view of the plug-in edit window. For instance, look at the top left **pp** object shown above, `pp 1 Gain 0. 2.`. The first argument, `1`, says this is parameter number 1. The second argument, `Gain`, says that the parameter's name is Gain. The third argument, `0.`, says that the minimum value of the parameter is 0. Note that 0 has a decimal point after it, indicating it's been entered as a `float`. This is important, since the type of the minimum value sets the type of the parameter values. If the value had been 0 without a decimal point, only integers would be output from the **pp** object. The fourth argument, `2.`, sets the maximum value of the parameter to 2. Knowing how this **pp** object works, you can probably determine the details of the parameters defined by the other **pp** objects.

Now let's look at how this **pp** object is connected to other objects. First, you'll notice a curious circular connection between the float number box and the **pp** object. This is purely for the

convenience of the plug-in developer for creating effect programs. The user of this plug-in will never see the number boxes. By connecting the two objects together in this way, the number box always reflects the state of the parameter, and in turn, the parameter always stores the value entered into the number box. Those of you with extensive Max experience might suspect that this would create a feedback loop and result in a stack overflow. However, the **pp** object has an internal feedback loop avoidance mechanism—something that perhaps other Max objects should have had long ago—that allows you to connect objects to it in this circular fashion.

But what makes this **pp** object actually do something is not this connection with the number box, but its eventual connection to something in the signal network. In this case, we send the output of the number box (which echoes its input from **pp**) to the ***~** object, where it serves to scale the overall output before being sent to **plugout~**.



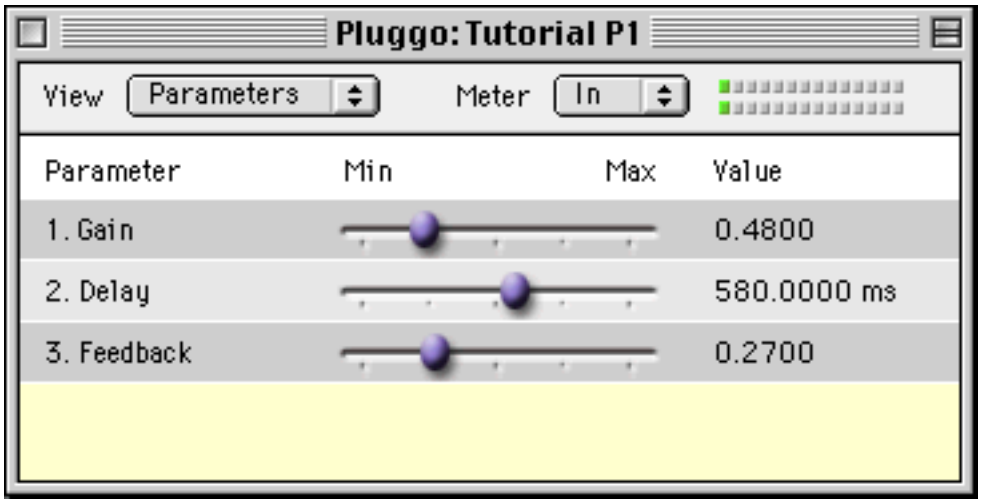
The other two **pp** objects serve different functions. pp 2 controls the delay time—its output is connected to the **tapout~** object. pp 3 is connected to another ***~**, serving as an internal gain control to determine how much of the output signal is fed back to the delay line input.

Unless the special keyword hidden appears in a **pp** object, there will be one egg slider for each **pp** object in the patcher. We've chosen not to hide any of our parameters here—you'll see an example of a hidden parameter in Tutorial P5.

Now let's quit Max and open the Tutorial P1 patcher—using the *Pluggo* plug-in—within your favorite sequencer.

> It's possible to run the sequencer and Max at the same time, but this requires that you have both a lot of memory and an audio card. You can assign one application to use the Sound Manager and the other to talk directly to the audio card. If you only have the Sound Manager available to you, one of the programs will not be able to access the audio input and may fail to initialize properly. In these tutorials, we assume that you'll have to keep quitting and restarting Max and your sequencer—if you can avoid it using the scheme suggested above, you can ignore the directions to quit and start up each application.

After you select the Tutorial P1 patcher from *Pluggo*'s standard open file dialog, open the plug-in's edit window and you'll see the following screen:

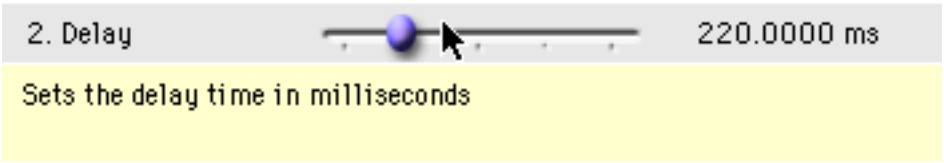


Test out the plug-in to verify that it does in fact work as a delay.

Then quit your sequencer and launch Max so you can examine the next tutorial.

Improving the User Experience

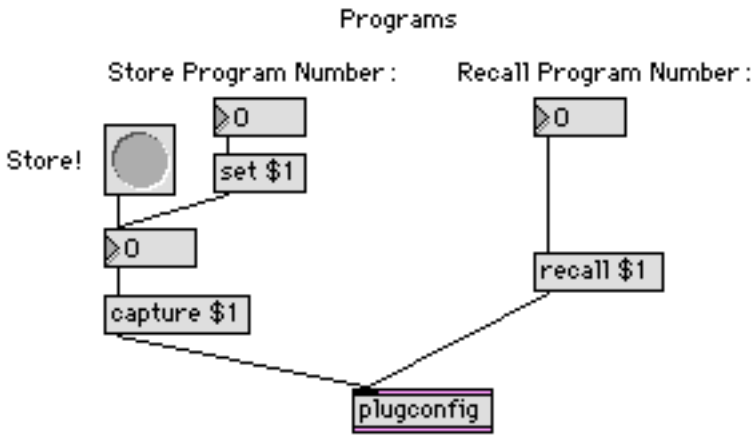
In this tutorial, we'll make some minor improvements to our delay line plug-in and introduce the somewhat hairy **plugconfig** object. We're going to make the following enhancements. First we want to add a collection of built-in effect programs to our delay to demonstrate the wide range of lovely effects it is capable of producing. Second, we want to add hints to our egg sliders describing in more detail what the parameters do. These hints will be visible when the user moves the mouse over a slider, as shown below.



Finally, we want to take advantage of the ability to brag about our plug-in and add a nice picture as an about box. Doing this will get us into the use of collectives and the Plugmaker application so we can package up our patcher as a VST plug-in file.

Making Effect Programs

Open the Tutorial P2 patcher. Examine the new section of the patcher labeled Programs.



Here you'll see a new object called **plugconfig**, plus message boxes containing the messages capture and recall. Double-click on the **plugconfig** object and a text window appears with a somewhat intimidating-looking script in it.

```
#C useviews 1 1 1 1;
#C numprograms 8;
#C preempt 1;
#C sigvschange 1;
#C sigvsdefault 32;
#C autosize;
#C defaultview Interface 0 0 0;
#C dragscroll 1;
#C noinfo;
#C setprogram 1 'Program 1' 0 0.25 0.1 0.;
```

```
#C setprogram 2 'Program 2' 0 0.25 0.25 0.39;
#C setprogram 3 'Program 3' 0 0.25 0.5 0.18;
#C uniqueid 128 157 227;
#C initialpgm 1;
```

The script is used to configure the plug-in's behavior within the runtime environment. The only thing we're interested in here at the moment are the lines that start with the setprogram message. These messages allow you to define presets that "ship" with your plug-in to demonstrate how wonderful it is. But it would be a pain if you had to type the presets in by hand, especially since all the values need to be entered between 0 and 1, regardless of the range the associated **pp** object. Fortunately, you don't; there's a much slicker way to do it. But before we close the script, look at the top line that defines the number of effect programs the plug-in will contain. By default, this number is a very generous 64. We've reduced it to 8 here.

One other important **plugconfig** script message when defining preset programs is initialpgm. In the vast majority of cases, you'll want to load the first program on startup, so give it in argument of 1. This message is not in a **plugconfig** script by default; you'll have to add it manually.

Now close the **plugconfig** text window and return to examining the patcher. If you change anything in the text window, you'll be asked if you want to store your changes back into the **plugconfig** object when you close the window. Avoid saving the window as a text file—this will not to you any good—just close the window and, if there are changes, click Save and the script will be updated.

Turn on the audio and, using some type of audio input as a test source, set the number boxes below each **pp** object to settings that you like. Move the number box labeled Store Program Number to 4. Now click on the **button** labeled Store! and the current settings are saved inside the **plugconfig** object. You can verify this by double-clicking on the **plugconfig** object and looking at the script. You'll see a setprogram message with a program name of Program 4. You can change this name to something more descriptive if you like, or leave it as is.

Now that your brilliant settings have been safely tucked away for posterity, try out one of the programs that was already stored in the **plugconfig** object. Move the number box labeled Recall Program to 1, 2, and 3. You can then return to the settings you stored by moving the number box to 4. If you want to tweak the settings further, make adjustments to the number boxes below the **pp** objects, then click the Store! **button** when you're happy.

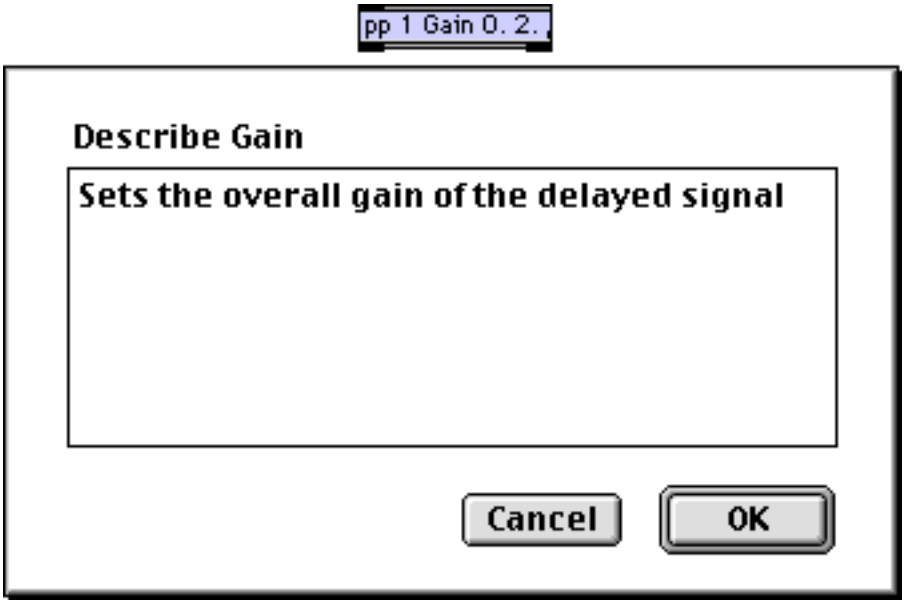
The capture message to **plugconfig** interrogates all **pp** objects in your patcher (and its subpatchers) for their current values, then creates a setprogram message in the object's script with all of these values. By using the float number box and **pp** object connected in a loop, you are saving the values you are using to affect the DSP algorithm inside the **pp** object where **plugconfig** can get them.

The recall message to **plugconfig** goes the other way—given a setprogram message inside the **plugconfig** script, it passes the values back to the **pp** objects in your patcher. You'll find these two messages invaluable for developing effect programs that you want to include in your plug-in.

Adding Descriptive Information to Parameters

You can add up to two lines of descriptive hint information to each **pp** object in your plug-in patcher. The information is displayed when the user moves the mouse over the egg slider associated with the parameter in the plug-in edit window.

To add the hint information, unlock the patcher window. Then select one of the **pp** objects (for example, the pp 1 Gain object) and choose Get Info… from the Max menu. The window shown below will appear.



If you think the description here is adequate, feel free to leave it alone. On the other hand, if you can think of something better to say, type it in. When you're finished, click OK.

Adding a Plug-in About Box

You may have noticed when using **pluggo** that all of the included plug-ins have either text or picture views that provide information about the author and/or function of the effect. For example, the *LFO* plug-in displays the following picture when you choose LFO Info from the plug-in edit window's View menu.

We've provided a picture for you to use as an about box. It's in a file called P2.pict in the plug-in development materials folder. If you want to see what it looks like, double-click on this file in the Finder and it will be displayed in SimpleText.

In order to link this picture into our plug-in, we first need to add a reference to it in our **plugconfig** script. At the end of the script, add the following line:

```
#C infopict P2.pict;
```

We've already specified with the useviews message near the top of the script that we want a plug-in info view. But until we added the infopict message, there was no information to display, so the view was not listed in the View pop-up menu in the plug-in edit window.

Close the text window and click Save to store the changes you made to the script.

Choose Save from Max's file menu to save the Tutorial P2 patcher with your changes. Then choose Save As Collective… from the File menu. You'll see the collective save dialog with the default script. We need to include the file P2.pict in our collective. An easy way to do this is to click Include File… and select P2.pict from the open file dialog. The collective script will now look something like this

```
open thispatcher
include Disk:Pluggo Stuff:Pluggo Development:Tutorials:P2.pict
```

(The exact path of the file P2.pict will of course be different for your computer.)

Click Make and the collective will be created. Call it P2.clct.

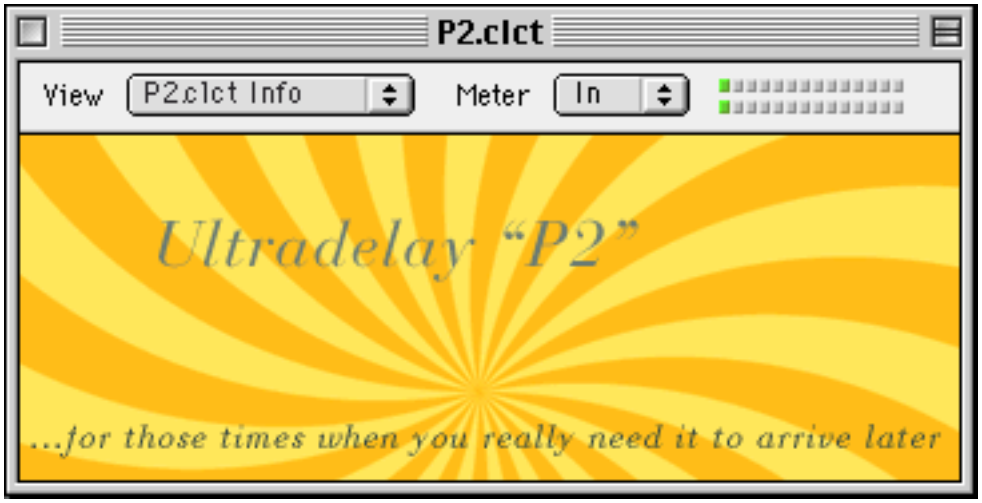
Now quit Max. In the Finder, drag the file P2.clct you just created onto the Plugmaker icon.

A new VST plug-in file will be created called P2.clct.pi in the same directory as P2.clct. You can move this file into your sequencer's VstPlugIns folder and it will be listed in the effect menu the next time the sequencer launches, or you can just open it where it is with the *Pluggo* plug-in.

Now let's look at the fruit of our labors. Launch your sequencer and insert the P2.clct plug-in. Open its edit window. First, notice the lovely hints when you move over the egg sliders.

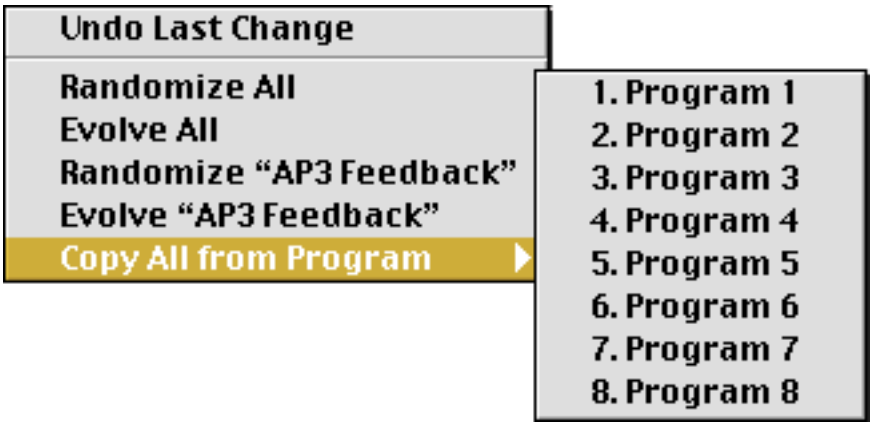
> The plug-in file is listed in the sequencer's menu as P2.clct even though the filename is P2.clct.pi. When displaying a plug-in's name, sequencers use the name associated with the aEff resource, not the filename. The Plugmaker application names the aEff based on the name of its input file. You can verify this by looking at the aEff resource inside the file generated by Plugmaker in ResEdit.

Now, choose P2.clct info from the View menu. There's the lovely picture.



Finally, try out the effect programs. Return to the Parameters view.

If you're using Vision or Cubase (or most other host applications), you can switch programs using the application's interface. If you're using Logic Audio, you will need to use the technique of holding down the command key and clicking somewhere in the egg slider interface, which will bring up a pop-up menu. One of the items in the menu is Copy All From Program. Choose some of the programs from the submenu.

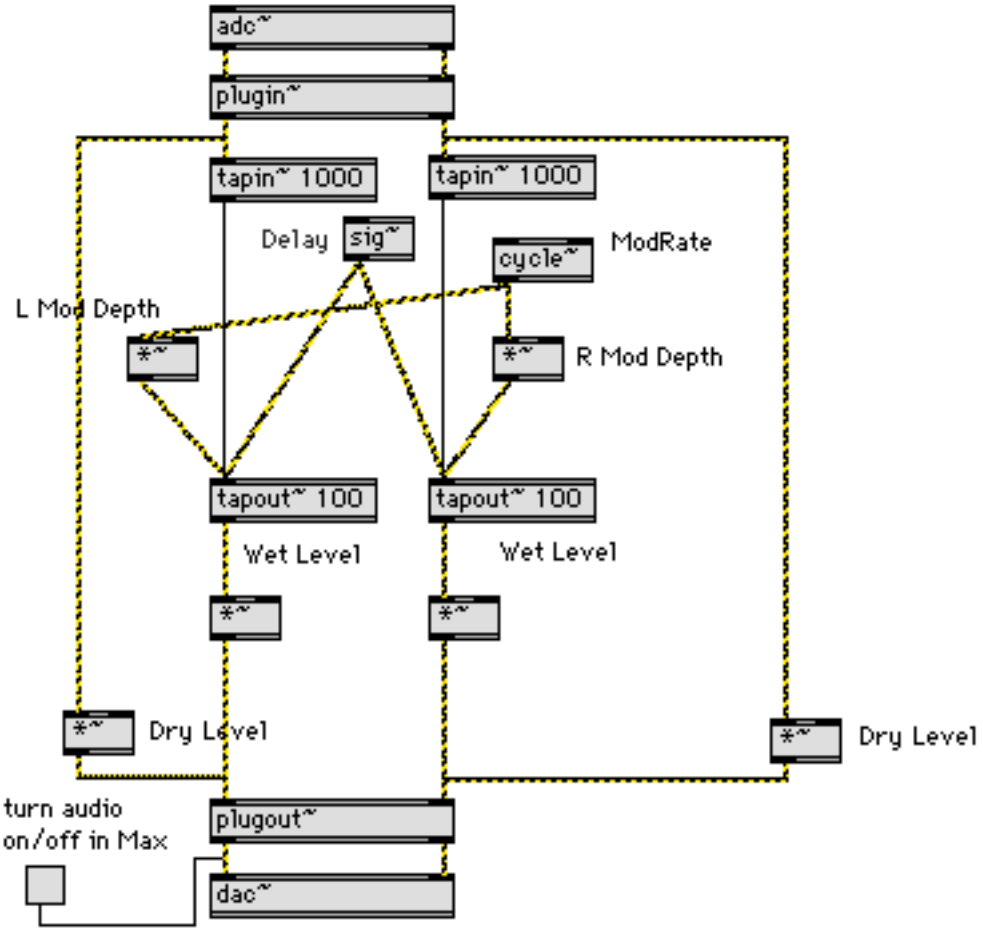


In this tutorial, we'll expand upon the previous examples to examine a patcher that implements a vibrato effect—also using a delay line—and has a Max patcher as an interface rather than making use of egg sliders. You'll see that things get quite a bit messier as the Max patch attempts to serve three functions simultaneously: signal network, parameter definition, and slick user interface.

The Wonders of Modulated Interpolating Delay Lines

You can do a lot of fun things with a time-varying signal connected to the inlet of a **tapout**~ object. In this example, we've connected a **cycle**~ object to the input of **tapout**~. The effect of the oscillator on the delay line is a pitch-changing effect that sounds pretty much like vibrato.

Another feature of this plug-in is that it is fully stereo, unlike the patchers in the first two tutorials. There are two **tapout**~ objects, each of which is being modulated by the same **cycle**~ object. In traditional jargon, the **cycle**~ modulator is functioning as an LFO (low frequency oscillator).

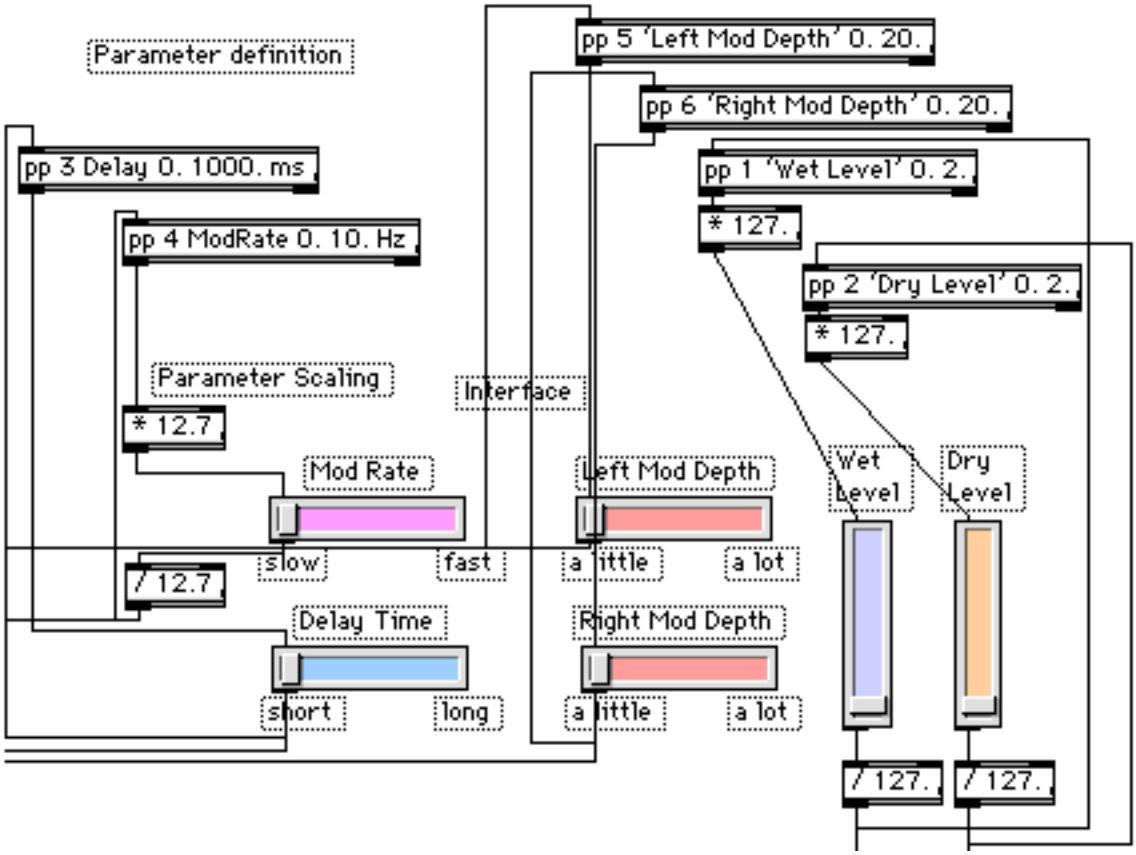


A ***~** scales the output of the **cycle**~ oscillator so it is appropriate for controlling delay time. In addition, a **sig**~ object is used to offset the oscillator with a fixed delay time. If the base delay time were 0, the delay time input to **tapout**~ would go negative—and since **tapout**~ would clip it

to 0 (no negative delays allowed in this universe) the vibrato effect would be somewhat non-standard. The ***~** and **sig~** objects are by the patcher's user interface and **pp** objects, giving the user control over the signal scaling and offset. Specifically, the Left Depth and Right Depth **pp** objects feed into the right inlets of ***~** objects scaling the output of the **cycle~** object. These are similar to the gain control parameters we saw in the first two tutorials, except in this case, the signal whose gain we are controlling is not an audible one.

A Plug-in User Interface in Max

Feast your eyes upon the collection of sliders and number boxes in the region of the patcher labeled Interface. It looks pretty nice until you unlock the patcher. Yikes!



Notice the multiply and divide objects used between the wet and dry level sliders and the **pp** objects. These are necessary because the sliders output only integers, so the range of values of the sliders needs to be scaled so that it is appropriate for the signal network. This is something that can be done automatically for you with the egg sliders in the Parameters view. With a Max interface however, you'll end up doing this translation yourself.

The user interface is restricted to a relatively small portion of the patcher window. In order to display only this portion of the window when we use our plug-in, we need to turn once again to the **plugconfig** object. It's located just below the interface objects. Double-click the object to see its script.

```
#C useviews 0 1 1 1;
#C numprograms 8;
#C preempt 1;
#C sigvschange 1;
#C sigvsdefault 32;
#C setsize 220 130;
#C defaultview Interface 550 220 0;
#C dragscroll 1;
#C noinfo;
#C setprogram 1 'Program 1' 0 0.232283 0.228346 0.103 0.1496 0.15 0.15;
#C setprogram 2 'Program 2' 0 0.232283 0. 0.292 0.677165 0.05 0.05;
#C setprogram 3 'Program 3' 0 0.232283 0.110236 0.143 0.07874 0.15 0.15;
#C uniqueid 128 221 227;
#C initialpgm 1;
```

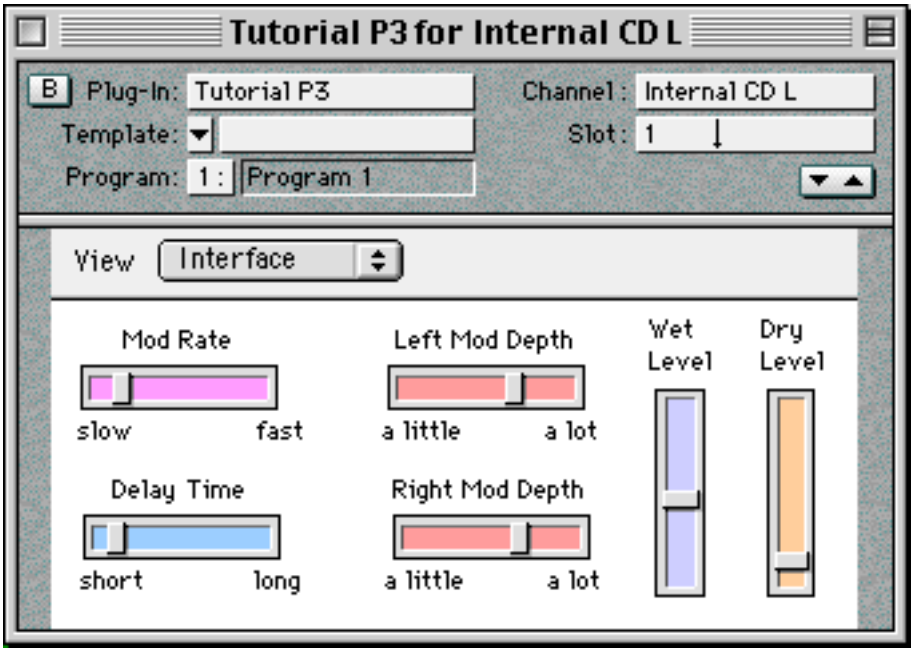
There are three relevant messages here: useviews, setsize, and defaultview.

The useviews message has been set so that we will not be seeing the Parameters view and its egg slider when we open this plug-in. Its arguments are `0 1 1 1`—the first argument says whether the Parameters view will be available or not. Note that we could have decided to make the Parameters view available *in addition* to our Max interface view, but we chose to aim for simplicity.

We've used the setsize message to determine the dimensions of our edit window. The setsize message takes two arguments for width and height. Unless you have some kind of pixel measuring tool, you may have to guess the width and height in pixels and refine them by trial and error. Or, take a picture of the screen in Max using command-shift-3 and use a graphics program that has a pixel ruler in it to make the measurements. Don't forget that the setsize message sets the size of the window including the 30 pixel area at the top where the View menu is located, so you'll want to add 30 to a pixel measurement you make of the height of the Max interface.

The defaultview message allows you to set a pixel offset to the left and top edge of your plug-in's user interface. Again, you could use trial and error to come up with the values you see in the script, but you can experiment with sending the **plugconfig** object the offset message, which attempts to scroll the patcher window to the specified X and Y values. While this feature might appear handy at first glance, it's made somewhat awkward by the fact that you are scrolling the interface that lets you change the scroll offset. If you measure the interface, remember to subtract 30 to the Y offset in order to account for the View menu area at the top of the edit window.

Close the script window and quit Max. Launch your sequencer and, using the *Pluggo* plug-in, insert the Tutorial P3 patcher into your sequencer's mixer. Open the plug-in edit window and try out the interface.



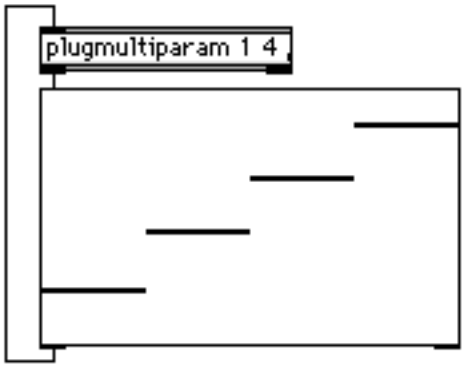
You might ask, given the somewhat tedious nature of building plug-in user interfaces out of Max objects, why it wouldn't be better to use the egg sliders wherever possible. This is probably not a bad idea, although occasionally there are specific reasons to make an interface that provides a greater variety of user interface elements than just sliders. We'll see that clearly in the next tutorial, which employs the **multislider** object.

Collections of Parameters

In this tutorial, we continue with our modulated delay line, but this time, we replace our simple oscillator with a function that steps through a sequence of delay times. A ramp between the delay times in the sequence creates a pitch shift effect. The patcher is similar in its structure to the *Cyclotron* and *Flange-o-tron* effects included with Pluggo, which apply repeating sequences to filter and flanger parameters.

The **multislider** object turns out to be an ideal object for setting a large number of similar parameters, and there's a plug-in parameter definition object that works with it called **plugmultiparam**. Yes, the name is a mouthful compared to **pp**, but **plugmultiparam** does the work of a lot of **pp** objects, so you'll probably not have to type its name very often.

Here's how **multislider** and **plugmultiparam** are typically connected to each other.

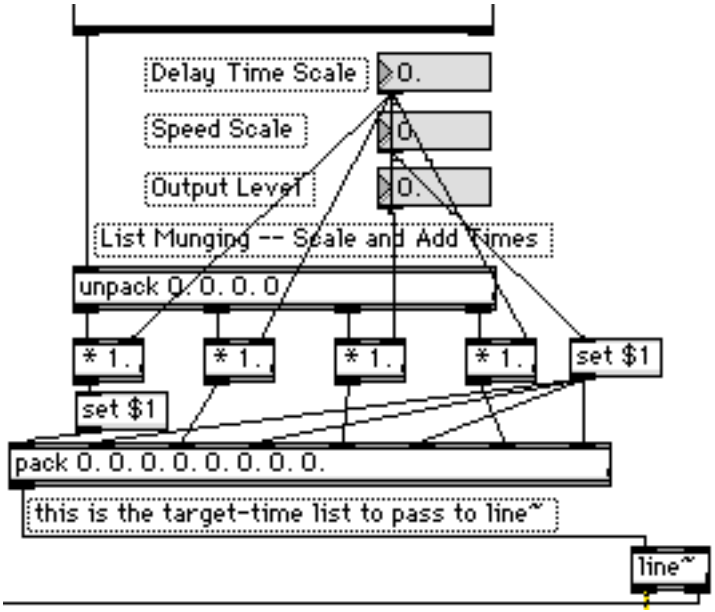


The **plugmultiparam** object defines a collection of up to 256 parameters. The first argument sets the index of the first parameter in the collection. The next argument sets the size of the collection. Here we've set our size to 4, corresponding to four steps in the repeating sequence.

Even though the **plugmultiparam** object defines a collection of parameters, it's possible that only one parameter within the collection will change at a time. As an example, consider the case where a single parameter in the collection is being automated within the host environment. **plugmultiparam** tries to avoid sending a giant list message with its entire collection of parameters whenever a single value changes. Instead, it uses the little-known select message to the **multislider** object that allows a single slider value to be changed. The select message also causes **multislider** to output its current set of values as a list.

Let's examine how the list of values coming from the **multislider** are communicated to the delay algorithm. Our goal is to use the **line**~ object to feed a sort of "envelope" of target-time pairs to the **tapout**~ object. But we have only the four "target" values in the **multislider**, so we need to insert millisecond values into the list output from the **multislider**. This requires a fair amount of what some computer types call *munging*. It's hard to give a precise definition of the word munge, but it has something to do with taking a collection of data, messing with its innards, and trying to put it back together without destroying its integrity.

Our munging procedure makes use of the **pack** and **unpack** objects as shown below.

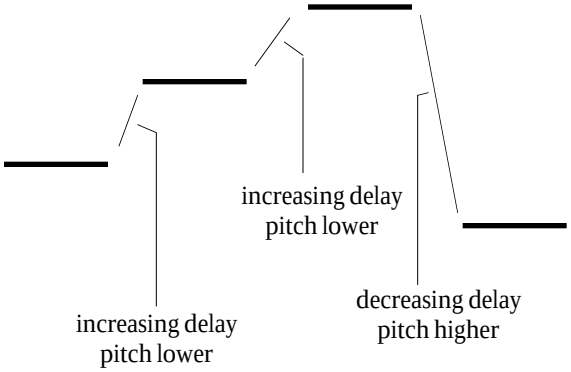


The list we want to assemble and send to the **line**~ object needs to contain a total of eight values because **line**~ expects a list of target-time pairs. We use the value of the Speed Scale parameter to be the time component of the pairs. We also want to be able to scale the **multislider**'s output, which ranges between 0 and 1, to a larger range of values that represent delay times. This is accomplished by multiplying the individual list values by the value of the Delay Time Scale parameter as they come out of **unpack** before reassembling the enlarged list.

The Speed Scale parameter sends its value to the four slots in the **pack** object that immediately follow the target values provided by the scaled list from the **multislider**.

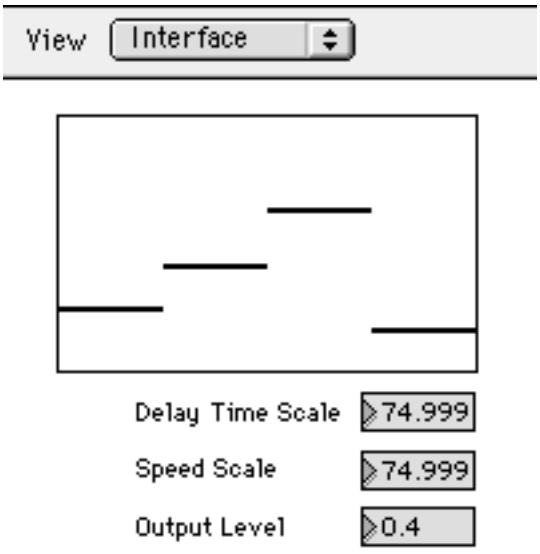
When the **line**~ object's sequence has completed, it sends a bang out its right outlet. We retrigger the sequence using the newest values from the **multislider**.

Turn on the audio by checking the box above the **dac~** object. Note that we've used a sine wave rather than the audio input to the computer as the plug-in's test signal. This will make the effect of the patcher quite obvious. But perhaps not *that* obvious. At first you might think that the steps of the sequence in the **multislider** represent pitches or transpositions, but they do not. It is the *direction* of the transition between one step and the next that determines whether the pitch is transposed up or down, as outlined in the diagram below.



Why does moving the delay time continuously from one value to another produce this pitch changing effect? One explanation of the use of modulating delay times for pitch effects is found in Tutorial 27 of the MSP manual—the short answer is that these modulations represent the same phenomenon as the Doppler effect except that they happen inside your computer, not out on the highway.

Quit Max, launch your sequencer and, using the *Pluggo* plug-in, insert the Tutorial P4 patcher into your sequencer's mixer. Open the plug-in edit window and try out the interface.



As an exercise, modify Tutorial P4 to use the **curve~** object and give it an additional parameter for curve factor. One solution to this problem is found in the patcher file Tutorial P4 Solution.

Parameter Modulation

In this tutorial, we'll look at a patcher for a plug-in that doesn't make or process sound at all. Instead, it's designed to change the parameter values of other plug-ins that handle sound. In **pluggo**, these patchers are called *Modulators*, and you'll find quite a number of them.

As you probably know, the basis of many audio effects is the way in which some aspect of the sound processing changes over time. By applying some type of modulating gesture or function to an effect parameter that was not originally thought to be something worth changing in a continuous way, you can expand the range of sonic results possible with many effects.

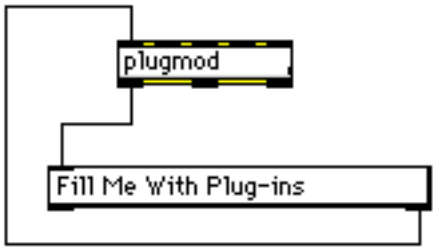
Open Tutorial P5. This patcher is a plug-in that uses a Max user interface for its edit window. It uses the **drunk** object to create a random process that you can apply to a parameter of another plug-in. Unlock the patcher, and focus on the object box in bright pink.

The plugmod Object

The key object needed to create patchers that change the parameters of other plug-ins in the runtime environment is called **plugmod**. It provides the core functions you'll need to create a Modulator plug-in, but you have to build a few things around it.

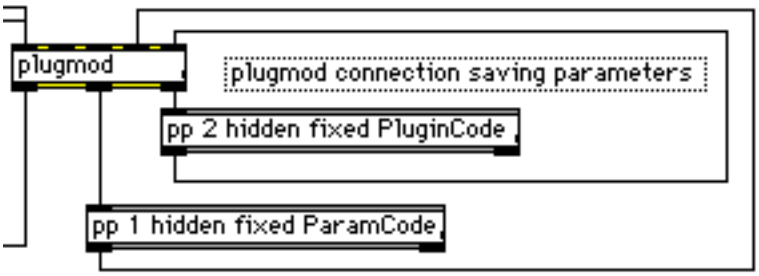
First, notice that the left outlet of **plugmod** is connected to a **menu** object—this outlet will set the **menu** object with a list of all the plug-ins and parameters that are currently loaded in the runtime environment. **plugmod** updates this list every time you insert or delete a plug-in. The only tricky thing is that the right outlet of the **menu** must be fed back into the **plugmod** object's left outlet. As you might remember, the right outlet of the **menu** object only functions if the Evaluate Item Text box is checked in its Get Info… dialog box, and this box is unchecked by default. While you're looking at the Get Info… dialog for the menu object, increase the maximum number of menu items to something like 256—you never know how many plug-ins and parameters will be around at one time, and you should be ready for a lot of them.

If you find that you can't get your **plugmod** plug-in to connect to a parameter when you choose it from the pop-up menu, verify that you've configured your **menu** object in the same way as shown here.



The pop-up menu used for assigning the modulation data to plug-in parameters necessitates that any Modulator plug-in have a Max-based interface, at least for the purpose of displaying the pop-up menu. You could use egg sliders for everything else.

Notice the two **pp** objects located below and to the right of the **plugmod** object. These **pp** objects are used for saving the connections between the **plugmod** object and the other plug-in and parameter the user has chosen for it to modulate. That way, when you save a document with modulation connections referencing other plug-ins in your host environment, the connections are preserved. And since the connection information is saved in the plug-in's parameters, the user can store and recall different modulator assignments using effect programs.

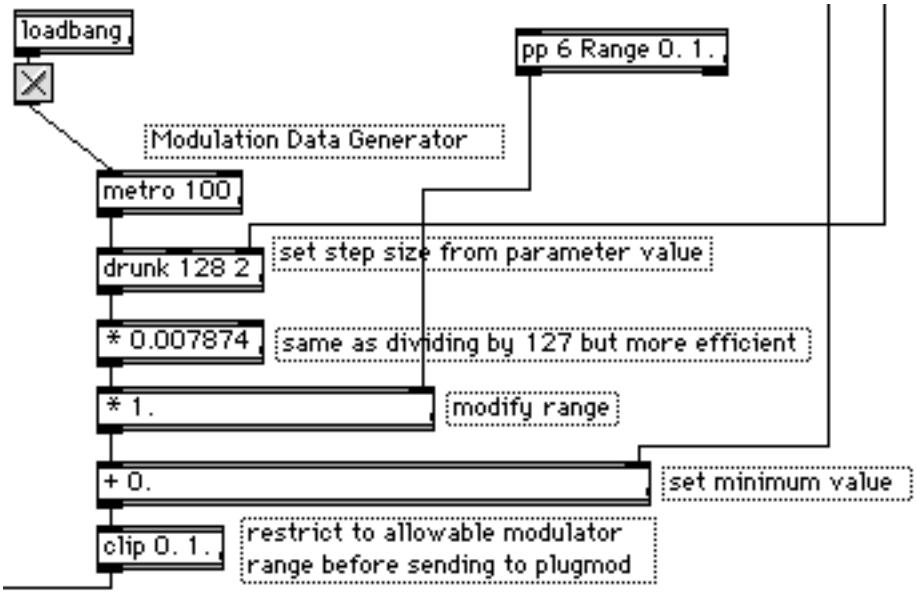


The two **pp** objects contain some new arguments we haven't seen before. The hidden argument means that the parameter defined in a **pp** object won't show up as an egg slider in an edit window. That's not an issue here, since this plug-in won't have an egg slider interface. More importantly for our purposes, hidden **pp** objects don't display their parameter names in the list of plug-ins and parameters generated by the **plugmod** object. That's important in this case because if the user were able to modulate the parameters that save the plug-in and parameter of the current modulation connection, the connection would immediately be broken. The other new argument in these two **pp** objects is fixed. The name is a little misleading, perhaps "protected" might have been a better choice. A fixed **pp** object will not be affected by the global randomization menu commands available when the user command-clicks in the plug-in edit window. Again, this is important because it allows the user to randomize the settings of this Modulator plug-in without affecting what parameter and plug-in are being modulated. Other parameters appropriate for having the fixed keyword are overall gain parameters. Usually, randomizing these parameters is at best uninteresting and at worse simply irritating.

The two **pp** objects need to be tied to specific inlets and outlets of the **plugmod** object. The fourth inlet and second outlet of **plugmod** handle what's called the plug-in or patcher code. This is a code generated from a plug-in identifier that you can set in the **plugconfig** object with the uniqueid message. If you don't have a **plugconfig** object, an identifier is constructed based on the name of your plug-in. The code is translated into a floating point number between 0 and 1, perfect for storage in a VST parameter. The fifth inlet and third outlet of the **plugmod** object handle a parameter code—again a floating-point number between 0 and 1 that is resolved into a parameter number by the **plugmod** object.

Generating Modulation Information

Now that we've managed to get through the required overhead associated with the **plugmod** object, lets examine how we actually do crazy things to the parameters of some unsuspecting plug-in.



In addition to the two parameters used for saving the modulation connection, the Tutorial P5 patcher contains three additional parameters that configure the random process used to generate modulation data. The Interval **pp** object just sets the interval of a **metro** object that is sending a bang to the **drunk** object. The Step Size **pp** object controls the size of the jumps the **drunk** object is allowed to make. A larger Step Size will produce more discontinuities in the **drunk** output. Finally, the Minimum and Range parameters let the user adjust the range over which the **drunk** object output produces values.

The **plugmod** object has three inlets that can be used for modulation data. In this tutorial, we're only concerned with the left inlet, where incoming data simply sets the value of the parameter. The second inlet takes incoming number and offsets the current value of the modulated parameter, and the third inlet scales the current value of the modulated parameter by the incoming number.

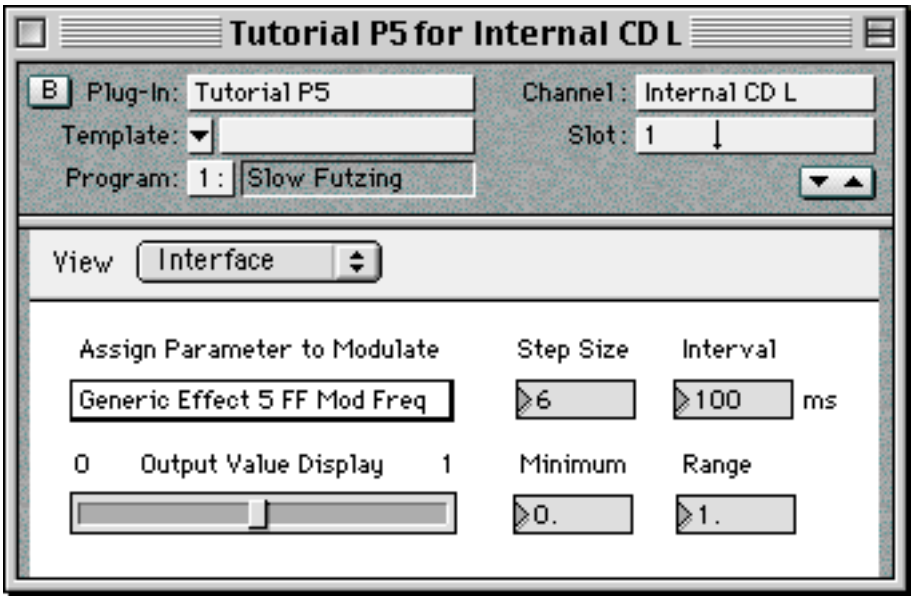
It's important to note that regardless of the actual range of a parameter being modulated, the number sent to the left inlet of **plugmod** must be within the 0 to 1 range. 1 will represent the maximum value of the parameter, and 0 will represent the minimum value of the parameter. As an example, suppose we choose to modulate an LFO Speed parameter that ranges from 3 Hz to 20 Hz. Sending a 0 into the left inlet of **plugmod** will set the LFO Speed parameter to 3. Sending a value of 0.5 into the left inlet of **plugmod** will set the parameter to the middle of its range, or 11.5. And sending a value of 1.0 into the left inlet of **plugmod** will set the parameter to its maximum value of 20.

The output of **drunk**, which is an integer between 0 and 127, is scaled so that it falls between 0 and 1. The Minimum and Range parameters provide further scaling along with an offset to constrain the final output range of the random process to a subset of the 0 to 1 range. We use a slider to give some indication to the user what the process is doing.

Unfortunately, it's not possible to test a **plugmod** connection within Max. You'll need to load the patcher into the runtime environment along with another plug-in created with Max/MSP. Or, you can also use the *Pluggo* plug-in to open any VST plug-in. Once hosted by *Pluggo*, the VST plug-in's parameters become available in the pop-up menu generated by **plugmod**.

In order to test this patcher, quit Max and launch your sequencer, inserting the *Pluggo* plug-in and choosing Tutorial P5 from the open file dialog.

Now insert another effect of your choice—we'll refer to this plug-in as the modulatee(!) If you're having trouble deciding, try *Generic Effect.*



Open both Tutorial P5's edit window and the modulatee's edit window. Using the pop-up menu in the Tutorial P5 window, choose one of the parameters of the modulatee. With the Tutorial P5 parameters set at their initial values, you should see the egg slider of the parameter you chose in the modulatee's edit window begin to move by itself.

Since Modulator plug-ins echo their input to their output, you may want to use them as Insert Effects rather than Send Effects, so that the volume of the Send Effects bus is not changed. Some Modulator plug-ins generate control information from their audio input. Since these plug-in's patchers contain audio objects (and therefore a DSP chain), the default echo audio thru behavior does not apply. If you create this type of plug-in, we suggest that for maximum flexibility, you provide an audio thru capability that you can turn on or off with a switch. The switch should also have an associated parameter.

Utilizing Host Syncronization Information

TEXT COMING SOON - refer to Tutorial P6 Max patcher

Audio Rate Pan

COMING SOON - refer to Tutorial P7 Max patcher

A Really Simple Synthesizer

COMING SOON - refer to Tutorial P8 Max patcher

A MIDI processor

COMING SOON - refer to Tutorial P9 Max patcher

Morphing

COMING SOON - refer to Tutorial P10 Max patcher

Conclusion

That concludes this short set of tutorials on plug-in building. To learn more about the details of the plug-in development tools, read through the reference pages on each of the objects. The **pp** and **plugconfig** objects sport a number of features we haven't discussed here. We hope the tools provided will allow you to create the plug-ins of your dreams. If there's something missing whose absence is preventing you from doing so, please let us know.

Plugmaker *Turn patches and collectives into plug-ins*

Introduction

The *Plugmaker* application transforms a Max patch or collective file into a VST plug-in. It creates a new file containing the aEff resource from the *Pluggo* plug-in, along with the patcher or collective data from the file you dragged. This file can be loaded directly by the sequencer, so the user will see your GranularLovePotion plug-in in the pop-up menu of available plug-ins rather than having to first choose *Pluggo* and then use a file open dialog box.

Using Plugmaker

To use Plugmaker, just drag the patcher or collective onto Plugmaker's icon. A new file will be created in the same folder as the file you dragged with .pi added to the filename. You'll probably want to get rid of the .pi extension after you move the file to the VstPlugIns folder of your favorite sequencer.

Plug-in Names

The name of your plug-in as displayed in the effect pop-up menu in a typical sequencer is not the filename of the plug-in (although, oddly enough, the first letter of the filename determines the order in which it appears in the menu in some applications). Rather, it is the name given to the aEff resource. Plugmaker uses the filename of its source—whether patcher or collective—as the name it gives to this resource.

This means that you should rename your patcher or collective to the name you desire for your plug-in *before* dragging it onto Plugmaker.

Plugmaker.nostrip

Plugmaker has the ability to remove selected external objects from the resulting plug-in file, saving disk space for a distribution of, like, 74 plug-ins or something. Create a text file called Plugmaker.nostrip in the same directory as Plugmaker. In this file, list the Max external objects you *don't* want removed from the resulting plug-in. Likely candidates for this list would be Max externals that you don't want to distribute by themselves to **pluggo** users who might also be Max/MSP users, for various reasons including greed and laziness. Of course, including externs within the plug-in file is a good way to make your plug-in easy to distribute.

If no Plugmaker.nostrip file is found, or if it contains no text, *all* external objects will be removed from the resulting plug-in file.

Introduction

The **plugconfig** object lets you configure your plug-in's behavior using a script that will be familiar to users of the **env**, **menubar**, and **lib** objects. The script can be accessed by double-clicking on a **plugconfig** object. You should only have one **plugconfig** object per plug-in patcher; if you have more than one, the object that loads last will be used by the runtime plug-in environment. Since it's not easy to determine which object that will be, just use one.

When you double-click on **plugconfig**, you'll see a short script already in place. These are the default settings, which are in fact identical to those you'd get if your patch contained no **plugconfig** object at all.

plugconfig is pretty much a read-only object when used within the runtime plug-in environment. The environment reads the settings from the object's script and is configured accordingly. You can send the messages view and offset to the object to scroll the patcher to a new location, but most plug-ins will allow the user to do this using the View menu that appears above the plug-in interface.

Input

Use the capture and recall messages to build a set of interesting presets that are embedded within your plug-in.

capture — The word capture, followed by a program number (1-based) and optional symbol, stores the current settings of all **pp** and **plugmultiparam** objects in the patcher containing the **plugconfig** object as well as its subpatchers. The settings are stored using a setprogram message added to the **plugconfig** object's script. The parameter numbers of the **pp** and **plugmultiparam** objects determine the order of the values in the setprogram message. capture does not work within the runtime plug-in environment.

recall — The word recall, followed by a program number (1-based), sets all **pp** and **plugmultiparam** objects to the values stored within a setprogram message in the **plugconfig** object's script. The parameter numbers of the **pp** and plugmultiparam objects determine the values they are assigned from the contents of the setprogram message.

read — The word read, followed by an optional symbol, imports a file of effect programs saved in Cubase format and loads as many as possible into the **plugconfig** object for saving as setprogram messages. No checking is done to verify that the file contains effect programs for a plug-in with the same unique ID code as the one in the **plugconfig** object, nor is there any checking to ensure that the number of parameters match. If the symbol is present, **plugconfig** looks for a file with that name. Otherwise, a standard open file dialog is displayed, allowing you select an effect program file.

view The word view, followed by a symbol that is the name of a view defined in the **plugconfig** object's script, scrolls the patcher containing the **plugconfig** object to the coordinate offset assigned to the view.

offset The word offset, followed by numbers for the X and Y coordinates, scrolls the patcher containing the **plugconfig** object to the specified coordinates.

Script Messages

Messages for View Configuration

A *View* is a particular configuration of the plug-in's edit window. **plugconfig** lets you control which views you'd like to see, and add views of the plug-in patcher at various pixel offsets that you can select with the menu. These might correspond to "pages" of controls you offer to the user.

usedefault Arguments: none

If this message appears in a script, there is no plug-in edit window. Instead, the parameter editing features of the host environment are used. By default, usedefault is not present in a script, and the plug-in's editing window appears.

useviews Arguments: 1/0 for showing views, as discussed below

useviews determines which plug-in edit window views are presented to the user. The views are specified in the following order: Parameters (the egg sliders), Interface (a Max patcher-based interface), Messages (a transcript of the Max window useful for plug-in development), and Plug-in Info (where you can brag about your plug-in). If the edit window is visible, the Pluggo Info view always appears.

For example, useviews 1 0 0 0 would place only the Parameters view in the plug-in edit window's View menu. The user would be unable to switch to another view.

defaultview Arguments: name, x offset, y offset, 1/0 for initial view

defaultview renames the Interface item in the plug-in's View menu to the name argument, scrolling the patcher to the specified x and y offsets when the view is made visible. If the third argument (optional) to defaultview is non-zero, the view is made the initial view shown when the plug-in editing window is opened. This will be true anyway if there is no Parameters view (as specified by the useviews message).

addview Arguments: name, x offset, y offset

addview adds an additional Interface view to the plug-in's View menu with a specified x and y offset. This allows you to scroll the patcher to a different location to expose a different part of the interface that might correspond to a "page" of parameter controls. If you send the view message to **plugconfig** with the name an added view as an argument, the patcher window will scroll to the view's x and y

offset. This works in Max as well as in the run-time plug-in environment, allowing you to test interface configurations.

dragscroll Arguments: allow (1), disallow (0)

This message is currently unimplemented.

meter Arguments: 1 (meter the input, default), 2 (meter the output), 3 (off)

The meter message sets the initial mode of the level meter at the top of the plug-in edit window. There is currently no way to permanently disable the meter, but it is disabled if there isn't enough space to display it fully because you've defined an edit window that is too narrow.

Messages for Window Configuration

autosize Arguments: none

autosize, which by default is enabled, sizes the plug-in edit window to be the height necessary to display all of the parameters, and the width of the parameter display.

setsize Arguments: width, height

setsize sets the plug-in edit window to be a specific size in pixels. If you use the Parameters view, this size may be overridden if you've specified a window too narrow to display the egg sliders properly. Note that you should add approximately 30 pixels to the size of the patcher window in order to account for the height of the View menu and level meter panel.

windowsize Arguments: none

windowsize sets the size of the plug-in edit window to the size of the patcher window.

Messages for Program Information

numprograms Arguments: number of programs

numprograms sets the number of stored programs for the plug-in. Programs are collections of values (between 0 and 1) for each of the parameters you've defined using **pp** and **plugmultiparam** objects. The default number of programs is 64, the minimum is 1, and the maximum is 128. By default, all programs are set to 0 for each parameter, but you can override this with the setprogram message.

setprogram Arguments: number, name, start index offset, list of values...

Normally, you won't be typing the setprogram message into a script yourself; you'll send capture messages to generate it automatically. You might end up editing it

though—for example to change the program's name—so it's useful to know a little about the message's format. setprogram lets you name a specific program and, optionally, set some initial values for it. Program numbers (for the first argument) start at 1. The name is a symbol, so if there are spaces in the name, it must be contained in single smart quotes. The start index offset argument sets a number added to 1 that determines the starting parameter number of the parameter values listed in the message. After this argument, one or more parameter values follow. If you don't supply enough values to set all the defined parameters, the additional ones are set to 0. You don't need to set the values at all if you want them to be 0. However, when you re-open the **plugconfig** script, the additional zero values will have been added. The start index offset argument is used to handle stored programs containing more than 256 parameters. 256 is the maximum size of a Max message.

initialpgm Arguments: program number

The initialpgm message specifies the program that should be loaded when the plug-in is initially opened. The default is 1, which loads the first program. An argument of 0 means no program will be loaded; instead in this case, you would use **loadbang** objects to set the initial values of plug-in parameters. This behavior, however, is not consistent with the majority of plug-ins that get set to the values in program 1 when they are loaded (since 1 is always the initial program, unless the plug-in is being restored as part of a document for the host application). Using an initialpgm 1 message has the added benefit of doing away with **loadbang** objects used to initialize your parameters. Any other program number (up to the number of programs in the plug-in specified by the numprograms message) can also be loaded, but the current program number as shown in the host sequencer's window cannot be changed by the plug-in, so given that all host sequencers are initially set to program 1, you'll end up confusing the user if you load another program number initially.

Messages for DSP Settings

accurate Arguments: none

The accurate message tells the runtime plug-in environment to run the Max event (or control) scheduler at the same number-of-samples interval as the signal vector size. At 32 samples this is slightly less than 1 ms but running the scheduler this often can have some impact on the overall CPU intensiveness of the plug-in.

By default, accurate mode is not enabled and the scheduler runs at the same interval as the I/O vector size of the host environment, typically 512 or 1024 samples. The only thing accurate mode affects is parameter updating to a plug-in, so for example if you have a control-rate "LFO" you may want to use this mode. The use of accurate mode will also increase the frequency of parameter updating from control-rate scheduled **plugmod** processes.

sigvsdefault Arguments: signal vector size

This message is currently ignored by the runtime plug-in environment. 32 is currently the only possible signal vector size.

oversampling Arguments: code number

This message is currently ignored by the runtime plug-in environment.

preempt Arguments: 1/0 sets priority of control messages.

This message is currently ignored by the runtime plug-in environment.

Messages for Descriptive Information

When configuring the plug-in's informational view, you choose between using text with infotext, a picture with infopict, or not having an info view at all with noinfo.

infotext Arguments: text as separate words and numbers

infotext allows you to describe the effect and have the text appear in the Plug-in Info view. There is a limit of about 256 words. A special symbol <P> produces a carriage return. Note that all commas and semicolons in the text must be preceded by a backslash. If you do not do this, you could wipe out the rest of your script when you save it.

infopict Arguments: file name of a PICT file in the Max search path

infopict allows you to include a picture to display in the Plug-in Info view. If you use infopict, you need to include the picture (manually) to your plug-in's collective script. The runtime plug-in environment will be able to find the picture within the collective.

noinfo Arguments: none

This is the default behavior for plug-in information. If neither text nor picture has been provided as information about the effect, the Plug-in Info item does not appear in the View menu, even if you've enabled it with the useviews command above. If noinfo and either infopict or infotext appear together in a script, noinfo "loses" and the info view is displayed.

welcome Arguments: text as separate words and numbers

The text arguments to the welcome message are displayed at the bottom hint area when the user opens the plug-in editing window for the first time and looks at the Parameters view, as well as when the cursor is moved into the top part of the window when the Parameters view is being used. If the nohintarea message is present

in the script, the lack of a hint area in the Parameters view will cause the welcome message not to be displayed.

nohintarea Arguments: none

If the nohintarea message appears in a script, the runtime plug-in environment does not provide additional space for a hint area at the bottom of the Parameters view. If however the number of egg sliders does not completely fill the edit window because its size was defined using windowsize or setsize, a hint area will be present.

swirl Arguments: none

The swirl message sets the hint area background to be drawn as a swirl inspired by the **pluggo** packaging (which was itself inspired by the publicity poster for the classic French film musical "Les Demoiselles de Rochefort"). The default appearance of the hint area is the pain, non-swirl background. To set the swirl colors, use hintfg and hintbg.

hintbg Arguments: red, green, and blue color components as 16-bit values

If you are offended by the yellow background color of the hint area, you can change it to something else. As an example, a medium gray would be specified with hintbg 40000 40000 40000, and a white background would be specified with hintbg 65535 65535 65535.

hintfg Arguments: red, green, and blue color components as 16-bit values

When using the swirl mode for the hint area, the hintfg message specifies the color of the dark part of the swirl. For best results, hintfg should be darker than hintbg.

uniqueid Arguments: id1 id2 id3 (between 0 and 255)

You'll find this message in your **plugconfig** script when you first open it. The arguments will be three randomly generated numbers between 0 and 255, something like three quarters of an IP address.

These numbers are used to build an ID code that will uniquely identify your plug-in. The code is used to identify a plug-in as a **pluggo**-based animal as well as to preserve **plugmod** connections between patchers.

You can either use the three randomly generated numbers or something intentional. There are about 16 million possibilities. 0 0 0 is reserved and cannot be used. 0 followed by two other numbers is reserved for use by Cycling '74 and its registered plug-in developers. You won't need to interact with this ID code, although you might want to know that part of it will be used as the basis for a floating-point "patcher code" output by the **plugmod** object. The floating-point value, however, will not in any way resemble the ID you choose.

Messages for Host Configuration

synth Arguments: none

This message tells the host that the plug-in should be considered a synthesizer and will receive MIDI and output audio only (i.e, it won't take audio as input). This directs the Cubase and Logic host applications to restrict the plug-in to VST Instrument contexts.

Arguments

None.

Output

None.

See Also

plugmod	Modify plug-in parameter values
Tutorial P2	Enhancing the plug-in interface
Tutorial P3	A plug-in with a Max interface

plugin~ *Define a plug-in's audio inputs*

Introduction

plugin~ and **plugout**~ define the signal inputs and outputs to a plug-in. You can use them within Max as simple thru objects, feeding **plugin**~ a test signal and routing the output of **plugout**~ to a **dac**~ object. When **plugin**~ and **plugout**~ are operating within the runtime environment however, they act differently. **plugin**~ ignores its input and instead outputs the plug-in's signal inputs fed to it by the host mixer. **plugout**~ does not output any type of signal out its outlets; instead it feeds its signal inputs to the plug-in's audio outputs to the host mixer.

Input

signal — In left and right inlets: When used in Max/MSP, the **plugin**~ object echoes its input to its output. When used in the runtime plug-in environment, signals sent to its inputs are ignored, and instead the audio inputs to the plug-in are copied to the **plugin**~ object's outlets.

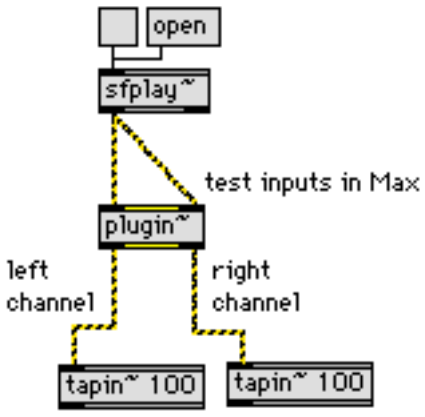
Arguments

None. **plugin**~ always has two inlets and two outlets.

Output

signal — When used in Max/MSP, the signal output of the **plugin**~ object is simply its signal input. When used in the runtime plug-in environment, the signal output will be the left and right channels of the audio input to the plug-in from the host. If the plug-in is inserted in a mono context, it's possible that only the left channel will contain the incoming audio signal and the right channel will be 0. The exact nature of the audio input to the plug-in is up to the host mixer.

Examples



See Also

plugout~ — Define a plug-in's audio outputs

Introduction

plugmidiin delivers any MIDI information targeted to the plug-in. It functions analogously to the Max **midiin** object, delivering raw MIDI as a sequential byte stream. You'll want to connect the **midiparse** object to its outlet. MIDI information is always delivered by **plugmidiin** at high-priority (interrupt) level. You may have more than one **plugmidiin** object in a patcher; each will output the same information.

Input

None.

Arguments

None.

Output

int MIDI message bytes in sequential order. For instance, a note-on message on chanel 1 for note number 60 with velocity of 64 would be output as 144 followed by 60 followed by 64.

See Also

midiparse (Max Reference manual) Interpret raw MIDI data

plugmidiout

Send MIDI to a plug-in host

Introduction

plugmidiout sends MIDI information to the host, where it is routed according to the host's current configuration. The plug-in has no control over the routing of its MIDI output. **plpugmidiout** is analogous to **midiout**; it expects raw MIDI bytes in sequential order. You can use **midiformat** to transform numbers into MIDI messages appropriate for **plugmidiout**.

Input

int — MIDI message bytes in sequential order. For instance, a note-on message on chanel 1 for note number 60 with velocity of 64 would be sent to **plugmidiout** as 144 followed by 60 followed by 64.

Arguments

None.

Output

None.

See Also

midiformat (Max Reference manual) Prepare data in the form of a MIDI message

Introduction

plugmod allows a plug-in to modify the parameter values of another plug-in. It generates a pop-up menu listing all the visible parameters of all currently loaded plug-ins. The output of this menu is fed back to the input of the object to tell it what parameter should be modified with the numeric input **plugmod** receives. Additional inlets and outlets interface with **pp** objects to save the object's connection to a particular plug-in and parameter in effect presets. This allows **plugmod** to reconnect to its target plug-in and parameter when a sequencer document is reloaded.

Input

anything — In left inlet: A plug-in name followed by a parameter index sets the parameter the **plugmod** object will modify with its numeric input. This plug-in and parameter are referred to as the object's *target*.

No Connection — In left inlet: When the word No Connection is received, the **plugmod** object breaks its connection (if any) with its current target and stops affecting the target parameter. The No Connection symbol is always the first item in the menu generated by **plugmod** 's left outlet when plug-ins are inserted or deleted in the runtime environment. plugmod:No Connection

int or float — In left inlet: The value received, which is constrained between 0 and 1, is assigned to the target plug-in and parameter.

In 2nd inlet: The value received is added to the base value of the parameter before **plugmod** began to modify it.

In 3rd inlet: The value received is multiplied by the base value of the parameter before **plugmod** began to modify it.

float — In 4th inlet: The value is interpreted as a code to assign a new plug-in as a target. The outlet of a **pp** object is normally connected to this inlet.

In right inlet: The value is interpreted as a code to assign a new parameter as a target. The outlet of a **pp** object is normally connected to this inlet.

Arguments

None.

Output

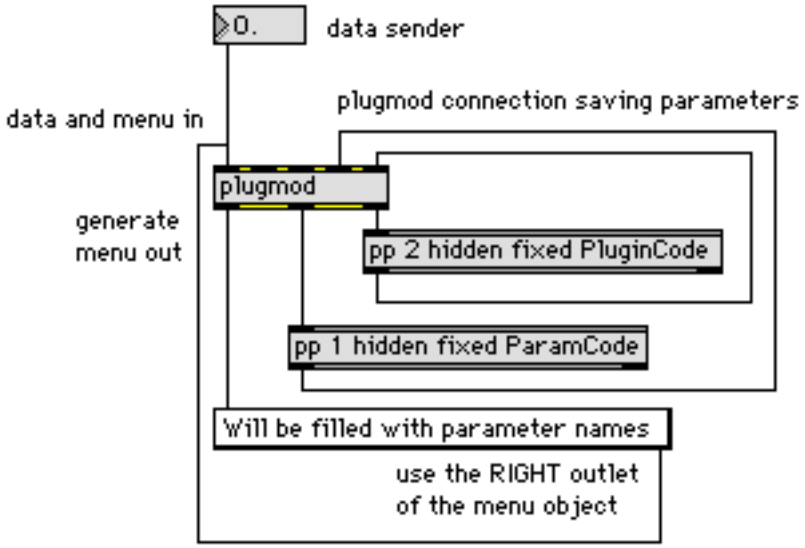
anything — Out left outlet: Output from this outlet of the **plugmod** object occurs when a new plug-in is either inserted or deleted. The messages update an attached **menu** object

with a new list of plug-ins and parameters that are potential targets for this object to modify.

float — Out 2nd outlet: The current plug-in code is output when the object's target changes via a message from the attached pop-up **menu** object sent to the object's left inlet, or when a new plug-in code is received in the 4th inlet.

Out right outlet: The current parameter code is output when the object's target changes via a message from the attached pop-up **menu** object sent to the object's left inlet, or when a new parameter code is received in the right inlet.

Examples



See Also

menu (Max Reference manual) Pop-up menu, to display and send commands
Tutorial P5 A modulator plug-in

Introduction

plugmorph allows a plug-in to modify the parameter values of another plug-in by creating a weighted average of two or more of its effect programs. Such an average is often known as a "morph" since it can often (but not always) create a continuous perceptual space between one effect program and another. **plugmorph** generates a pop-up menu listing all currently loaded plug-ins. The output of this menu is fed back to the input of the object, allowing the user to specify which plug-in should be modified according to the input **plugmorph** receives. An additional inlet and outlet interface with a **pp** object saves the object's connection to a particular plug-in. This allows **plugmorph** to reconnect to its target plug-in when a sequencer document is reloaded.

Input

anything In left inlet: A plug-in name sets what the **plugmorph** object will modify with its input. This plug-in is referred to as the object's *target*.

No Connection In left inlet: When the word No Connection is received, the **plugmorph** object breaks its connection (if any) with its current target and will no longer change a plug-in's parameters. The No Connection symbol is always the first item in the menu generated by **plugmorph** 's left outlet when plug-ins are inserted or deleted in the runtime environment. plugmorph:No Connection

list In left inlet: Causes **plugmorph** to calculate new values for the connected plug-in's parameters. The format of the list is an effect program number followed by a wieghting fraction. A maximum of 128 program numbers can be specified. If the fractions do not add up to 1, they are normalized to do so. As an example, the list 1 0.5 2 0.5 would set the target plug-in's pararmeters to values that were a simple average of effect programs 1 and 2. A list of 1 0.6 2 0.6 3 0.6 4 0.6 would perform a weighted averaging of the first four effect programs where the parameter values of each program were represented equally. In other words, each programs's parameter value contributes 25% to the morphed value. If the target plug-in's current effect program is among those being morphed, an attempt is made not to store the parameter values so the user can perform more than one morph. The generated parameter values can be stored later using the store message to **plugmorph**. However, some **multislider**-based plug-ins defer parameter changes in such a way that this storage prevention mechanism doesn't work, requiring that the user set the current effect program to a number that isn't involved in the morph.

morphfixed In left inlet: The word morphfixed, followed by a number, determines whether parameters marked as fixed are included in the morph. If the number is 0, fixed parameters are not included and their values are left unchanged. If the number not zero, fixed parameters are included. The default behavior of **plugmorph** is to include fixed parameters.

morphhidden In left inlet: The word morphhidden, followed by a number, determines whether parameters marked as hidden are included in the morph. If the number is 0, hidden parameters are not included and their values are left unchanged. If the number not zero, hidden parameters are included. The default behavior of **plugmorph** is to include hidden parameters.

store In left inlet: The word store copies the current values of the target plug-in's parameters to its effect program.

float In right inlet: The value is interpreted as a code to assign a new plug-in as a target. The outlet of a **pp** object is normally connected to this inlet.

Arguments

None.

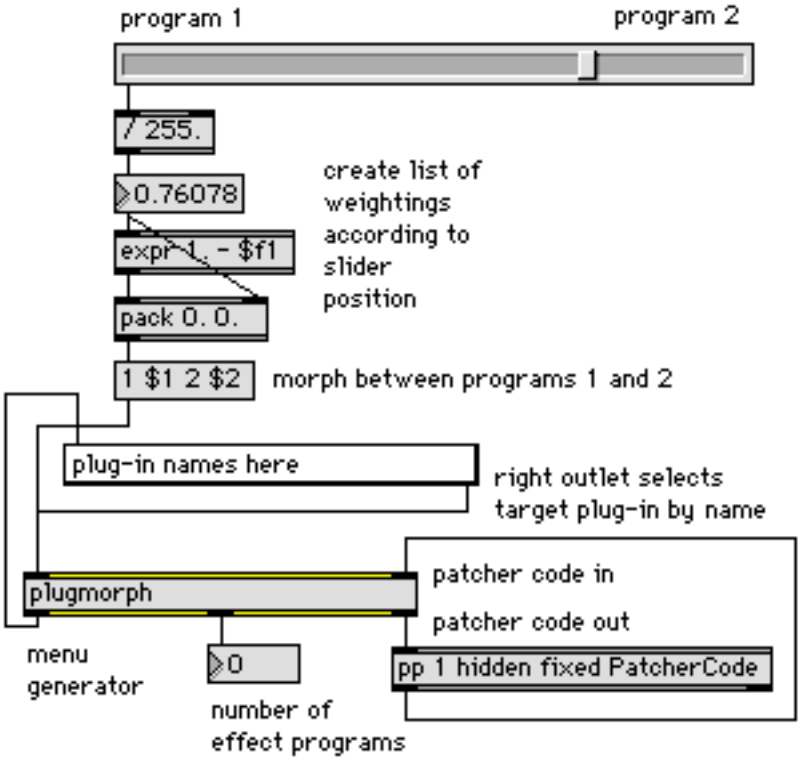
Output

anything Out left outlet: Output from this outlet of the **plugmorph** object occurs when a new plug-in is either inserted or deleted. The messages update an attached **menu** object with a new list of plug-ins that are potential targets.

float Out 2nd outlet: When a new plug-in is selected as a target, **plugmorph** outputs the number of effect programs it contains out this outlet.

Out right outlet: The current parameter code is output when the object's plug-in target changes via a message from the attached pop-up **menu** object sent to the object's left inlet, or when a new parameter code is received in the right inlet.

Examples



See Also

menu (Max Reference manual) Pop-up menu, to display and send commands

plugmultiparam *Define multiple plug-in parameters*

Introduction

The **plugmultiparam** object lets you define three or more parameters that are displayed and changed by a single object. However, these parameters will be hidden from the Parameters view in the plug-in window; they can only be changed by creating a Max user interface. Primarily, **plugmultiparam** was designed to be used in conjunction with the **multislider** object; it can also work with the **plugstore** object, or simply a set of cleverly organized **pack** and **unpack** objects.

Input

int — The value at the specified parameter index is sent out the object's right outlet.

list — Interpreted as a set of values to be assigned to the object's parameters, starting at the lowest numbered parameter. If the list is longer than the number of parameters defined by the object, the extra elements are ignored. The values of the list are constrained to be within the minimum and maximum arguments of the object.

bang — Sends the currently stored values out the object's left outlet.

setmessage — The word setmessage, followed by a symbol, changes the message that sets individual values when they change (for example, because the stored program was changed). The default select message is useful in conjunction with the **multislider** object.

Arguments

int — Obligatory. Defines the starting parameter index to be covered by the object.

int — Obligatory. Defines the number of parameter indices to be covered by the object.

float or int — Optional. Sets the minimum value of the input and output for all parameters. The default value is 0.

float or int — Optional. Sets the maximum value of the input and output for all parameters. The default value is 1.

Example: 32 parameters whose value ranges between 1 and 99 are stored starting at parameter index 13 with the following arguments to **plugmultiparam**:

```
plugmultiparam 13 32 1 99
```

Output

list — Out left outlet: The left outlet produces the current values as a list when the object receives a bang message.

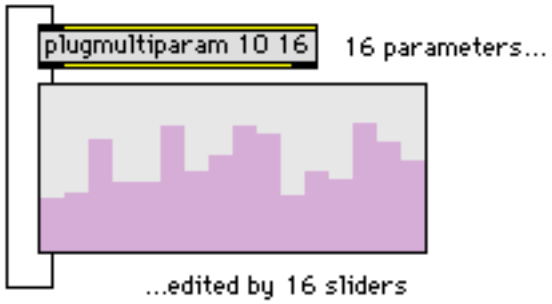
any message Out left outlet: The **plugmultiparam** object also produces a message to set individual values in the collection using the following format

<message name> <index> value

By default, the message name is select—this is appropriate for setting one value in a **multislider** object. You can change the name to something else with the setmessage message described above. The index argument starts at 0 for the first parameter and goes up by 1 for each subsequent parameter—it is not affected by the starting parameter index argument to **plugmultiparam**. The index argument is followed by the current parameter value.

float Out right outlet: When an int message is received, the value at the specified parameter index is output.

Examples



See Also

plugstore	Store multiple plug-in parameters
pp	Define a plug-in parameter
Tutorial P4	Using **multislider** and **plugmultiparam**

plugout~ *Define a plug-in's audio outputs*

Introduction

plugin~ and **plugout**~ define the signal inputs and outputs to a plug-in. You can use them within Max as simple thru objects, feeding **plugin**~ a test signal and routing the output of **plugout**~ to a **dac**~ object. When **plugin**~ and **plugout**~ are operating within the runtime environment however, they act differently. **plugin**~ ignores its input and instead outputs the plug-in's signal inputs fed to it by the host mixer. **plugout**~ does not output any type of signal out its outlets; instead it feeds its signal inputs to the plug-in's audio outputs to the host mixer.

Input

signal — In left and right inlets: When used in Max/MSP, the **plugout**~ object echoes its input to its output. When used in the runtime plug-in environment, the input to **plugout**~ is copied to the audio outputs of the plug-in.

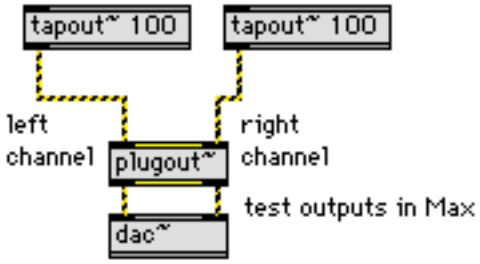
Arguments

None. **plugout**~ always has two inlets and two outlets.

Output

signal — When used in Max/MSP, the signal output of the **plugout**~ object is simply its signal input. When used in the runtime plug-in environment, the signal output to the outlets is undefined, and the input is copied to the audio outputs of the plug-in.

Examples



See Also

plugin~ — Define a plug-in's audio inputs

Introduction

plugphasor~ outputs an audio-rate sawtooth wave that is sample-synchronized to the beat of the host sequencer. The waveform can be fed to other audio objects to lock audio processes to the audio of the host.

Input

None.

Arguments

None.

Output

signal The output of **plugphasor~** is analogous to **phasor~**: it ramps from 0 to 1 over the period of a beat. If the current host environment does not support synchronization or the host's transport is stopped, the output of **plugphasor~** is a zero signal..

See Also

plugsync~	Report host synchronization information
Tutorial P7	Audio-rate synchronization

plugreceive~ *Receive audio from another plug-in*

Introduction

plugsend~ and **plugreceive**~ are used to send audio signals from one plug-in to another. They are used in the implementation of the PluggoBus feature of many of the plug-ins included with **pluggo**.

Input

signal — The input to the **plugreceive**~ object comes from a **plugsend**~ object to which it is currently connected. Initially, this will be a **plugsend**~ having the same name as the **plugreceive**~ object's argument.

set — The word set, followed by a symbol naming a **plugsend**~ object, connects the **plugreceive**~ object to a the specified **plugsend**~ object(s), and the **plugreceive** object's audio output becomes the input to the **plugsend**~. If the symbol doesn't name a **plugsend**~ object, the audio output becomes zero.

Arguments

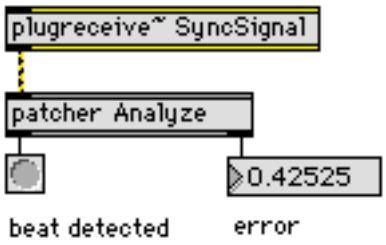
symbol — Obligatory. Gives the **plugreceive**~ object a name used for connecting with one or more **plugsend**~ objects.

Output

signal — The audio signal input to the **plugsend**~ objects connected to this object. If no **plugsend**~ objects are connected, the audio output is zero.

There may be a delay of one processing (I/O) vector size of the host mixer between the **plugreceive**~ output and the inputs to the plug-in in which the **plugreceive**~ is located. This occurs when a **plugsend**~ occurs later in the processing chain than the **plugreceive**~ to which it is sending audio.

Examples



See Also

plugsend~ — Send audio to another plug-in

Introduction

plugsend~ and **plugreceive**~ are used to send audio signals from one plug-in to another. They are used in the implementation of the PluggoBus feature of many of the plug-ins included with **pluggo**.

Input

signal — The input to the **plugsend**~ object is mixed with other **plugsend**~ objects, which can be in the same plug-in or a different plug-in, and is then sent out the signal outlets of any connected **plugreceive**~ objects.

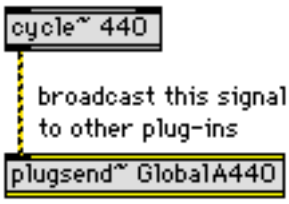
Arguments

symbol — Obligatory. Gives the **plugsend**~ object a name used for connecting with other **plugsend**~ and **plugreceive**~ objects.

Output

None.

Examples



See Also

plugreceive~ — Receive audio from another plug-in

plugstore

Store multiple plug-in parameter values

Introduction

The **plugstore** object works with **plugmultiparam** to allow you to get values into and out of **plugmultiparam** from multiple locations in a patcher.

Input

bang — Sends the stored list out the object's outlet.

list — Stores the elements of the list (up to the size of the object) and repeats them to the object's outlet.

select — The word select, followed by an index and value, stores the value at the specified index (starting at 1 for the first element) and sends the stored list out the object's outlet. plugstore:select

set — The word set, followed by an index and value, stores the value at the specified index (starting at 1 for the first element) but does not output the stored list. plugstore:set

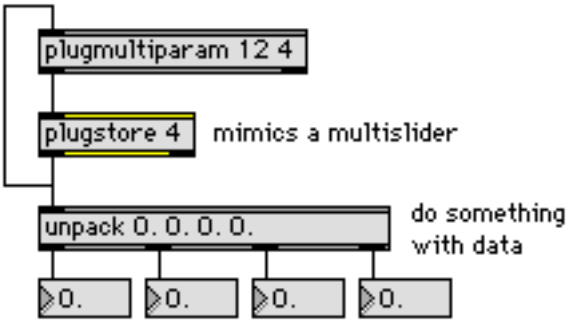
Arguments

int — Obligatory. Sets the number of elements stored in the **plugstore** object's list.

Output

list — The stored list is output whenever a list, bang, or select message is received.

Examples



See Also

plugmultiparam — Define multiple plug-in parameters

Introduction

The **plugsync~** object provides information about the current state of the host. Sample count information is available in any host; even Max. The validity of the other information output by the object is dependent upon what synchronization capabilties the host implements; the value from the flags (9th) outlet tells you what information is valid. Output from **plugsync~** is continuous when the scheduler is running./

Input

None.

Arguments

None.

Output

int — Out left outlet: 1 if the host's transport is currently running; 0 if it is stopped or paused.

int — Out 2nd outlet: The current bar count in the host sequence, starting at 1 for the first bar. If the host does not support synchronization, there is no output from this outlet.

int — Out 3rd outlet: The current beat count in the host sequence, starting at 1 for the first beat. If the host does not support synchronization, there is no output from this outlet.

float — Out 4nd outlet: The current beat fraction, from 0 to 1. If the host does not support synchronization, the output is 0. If the host does not support synchronization, there is no output from this outlet.

list — Out 5th outlet: The current time signature as a list containing numerator followed by denominator. For instance, 3/4 time would be output as the list 3 4. If the host does not support time signature information, there is no output from this outlet.

float — Out 6th outlet: The current tempo in samples per beat. This number can be converted to beats per minute using the following formula: (sampling-rate / samples-per-beat) * 60. If the host does not support synchronization, there is no output from this outlet.

float — Out 7th outlet: The current number of beats, expressed in 1 PPQ. This number will contain a fractional part between beats. If the host does not support synchronization, there is no output from this outlet.

float — Out 8th outlet: The current sample count, as defined by the host.

int Out 9th outlet: A number representing the validity of the other information coming from **plugsync~** . Mask with the following values to determine if the information from **plugsync~** will be valid.

Sample Count Valid	1 (always true)
Beats Valid	2 (2nd, 3rd, 4th, and 7th outlets valid)
Time Signature Valid	4 (5th outlet valid)
Tempo Valid	8 (6th outlet valid)
Transport Valid	16 (left outlet valid)

See Also

plugphasor~	Host-synchronized sawtooth wave
Tutorial P6	Synchronization with **plugsync~**
Tutorial P7	Audio-rate synchronization

Introduction

You can use the plugtell object to learn what capabilities are present in the host environment in which your plug-in is loaded. The symbols listed below are not the only things you can ask the host, but they are the ones currently supported by VST 2..0 and MAS as of the writing of this manual.

Input

sendVstEvents Returns whether the host sends events to the plug-in.

sendVstMidiEvent Returns whether the host sends MIDI events to the plug-in

sendVstTimeInfo Returns whether the host sends any form of time information to the plug-in

receiveVstEvents Returns whether the host does anything with events sent to it by the plug-in.

receiveVstMidiEvent Returns whether the host does anything with MIDI sent to it by the plug-in.

receiveVstTimeInfo Returns whether the host does anything with time information sent to it by the plug-in.

reportConnectionChanges Returns whether the sends information about changes in plug-in input and output connections.

acceptIOChanges Returns whether the plug-in can change its input and output connections with the host.

sizeWindow Returns whether the host supports the plug-in resizing its window.

asyncProcessing Returns whether the host's processing is asynchronous (such as on an external DSP card).

offline Returns whether the host supports the VST offline specification (the runtime plug-in environment currently doesn't support it).

supplyIdle Returns whether the host can supply an idle callback to a plug-in even if the plug-in's window isn't open.

supportShell Returns whether the host supports a shell that can open more than one plug-in.

Arguments

None.

Output

int — 1 if the capability is supported by the host environment, 0 if it is either not understood or not supported.

Introduction

The **pp** object (an abbreviation for *plug-in parameter*) defines plug-in parameters. It has a number of optional arguments that let you define the parameter minimum and maximum, hide the parameter from display, set the color of the egg slider associated with it, etc. You connect the output of the **pp** object to something you want to control with a stored parameter. If your plug-in will use a Max patcher interface, you need to connect the interface element that will change the parameter's value to the inlet of the **pp** object. The **pp** object will send new parameter values out its outlet at various times: when you move an egg slider, when the user switches to a new effect program, and when the host mixer is automating the parameter changes of your plug-in.

Internally, the **pp** object and the runtime plug-in environment store values between 0 and 1. By giving the **pp** object optional arguments for minimum and maximum, you can store and receive any range of values and the object will convert between the range you want and the internal representation. If for some reason you want to know the internal 0-1 representation, you can get it from the object's right outlet. If you want to send a value that is based on the internal 0-1 representation, use the rawfloat message.

Input

bang — Sends the current value of the parameter out the object's right outlet in its internal (unscaled) form between 0 and 1, then out the object's left outlet scaled by the object's minimum and maximum.

float or int — Sets the current value of the parameter and then sends the new value out the right and left outlets as described above for the bang message. The incoming number is constrained between the minimum and maximum values of the object.

float or int — Sets the current value of the parameter without any output. The incoming number is constrained between the minimum and maximum values of the object.

(Get Info...) — Choosing Get Info… from the Max menu opens a dialog for editing a description of the parameter that is displayed in the Parameters view of the plug-in edit window when the user moves the cursor over the egg slider corresponding to the parameter. This command is not available in the runtime plug-in environment.

open — Same as choosing Get Info… from the Max menu.

text — The word text, followed by a single symbol, allows you to set the text displayed in the Parameters view of the plug-in edit window when the user moves the mouse over the egg slider corresponding to the parameter. pp:text

rawfloat — The word rawfloat, followed by a number between 0 and 1, sets the current parameter value to the number without scaling it by the object's minimum and

maximum. The value is then send out the right and left outlets of the object as described above for the bang message.

Arguments

The **pp** object takes a number of arguments. They are listed in the order that they need to appear.

int — Obligatory. The first argument sets the parameter number. The first parameter is 1. Parameter numbers should be consecutive (but they need not be), and two **pp** objects should not have the same parameter number. An error will be reported in the Messages view of the runtime plug-in environment if duplicate parameter numbers are encountered.

hidden — Optional. If the word hidden appears as an argument, the parameter will not be given an egg slider in the plug-in edit window and will not appear in the pop-up menu generated by the **plugmod** object. pp:hidden

fixed — Optional. If the word fixed appears as an argument, the parameter will not be affected by the Randomize and Evolve commands in the parameter pop-up menu available in the plug-in edit window when the user holds down the command key and clicks in the interface. This is appropriate for gain parameters, where randomization usually produces irritating results.

c2-c4 — Optional. If c2, c3, or c4 appears as argument, the color of the egg slider is set to something other than the usual purple. Currently c2 is Wild Cherry, c3 is Turquoise, and c4 is Harvest Gold. pp:color messages

symbol — Optional. The next symbol after any of the optional keywords names the parameter. This name appears in the Name column of the Parameters view and in the pop-up menu generated by the **plugmod** object.

float or int — Optional. After the parameter name, a number sets the minimum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pp** object's outlets, as well as the displayed value in the Parameters view. The type of the minimum value determines the type of the parameter values the object accepts and outputs. If the minimum value is an integer, the parameters will interpreted and output as integers. If the minimum value is a float, the parameters will be interpreted and output as floats.

float or int — Optional. After the minimum value, a number sets the maximum value of the parameter. The minimum and maximum values determine the range of values that are sent into and out of the **pp** object's outlets, as well as the displayed value in the Parameters view.

symbol — Optional. After the minimum and maximum values, a symbol sets the label used to display the units of the parameter. Examples include Hz for frequency, dB for amplitude, and ms for milliseconds.

choices — Optional. If the word choices appears after the minimum and maximum values, subsequent symbol arguments are taken as a list of discrete settings for the object and are displayed as such in the Parameters view. As an example pp 1 Mode 0 3 choices Thin Medium Fat would divide the parameter space into three values. 0 (anything less than 0.33) would correspond to Thin, 0.5 (and anything between 0.33 and 0.67) would correspond to Medium, and 1 (and anything between 0.67 and 1.0) would correspond to Fat. Only the name of the choice, rather than the actual value of the parameter, is displayed in the Parameters view.

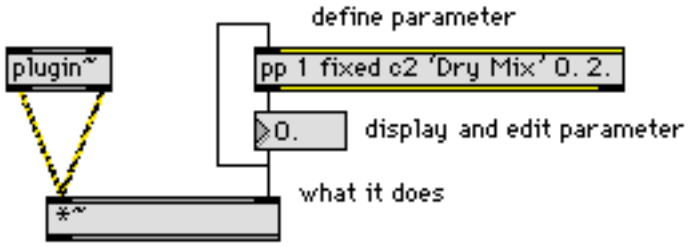
dB — Optional. If the word choices does not appear as argument, the word dB can be used to specify that the value of the parameter be displayed in decibel notation, where 1.0 is 0 dB and 0.0 is negative infinity dB.

Output

int or float — Out left outlet: The scaled value of the parameter is output when it is changed within the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlet. The parameter value can be changed in the runtime environment in the following ways: the user moves an egg slider, the parameter is being automated by the host mixer, or the user has selected a new effect program for the plug-in within the host mixer.

float — Out right outlet: The unscaled value of the parameter is output when it is changed by the runtime environment or when a bang, int, float, or rawfloat message is received in the object's inlet. You might use this value if you want to use a different value in your plug-in's computation than you display to the user.

Examples



See Also

plugmultiparam Define multiple plug-in parameters
Tutorial P1 A plug-in with an egg slider interface

Introduction

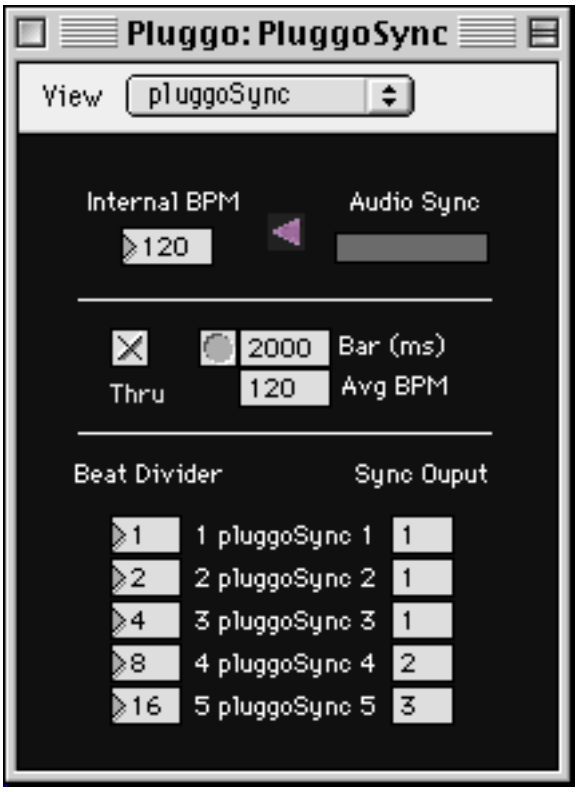
COMING SOON - refer to Tutorial P7 Max patcher for an example

Introduction

COMING SOON - refer to Tutorial P7 Max patcher for an example

PluggoSync

The *PluggoSync* plug-in outputs Max messages that you can use to keep your plug-ins in sync with each other and/or with the host sequencer environment.



The *PluggoSync* plug-in can run on its own internal clock, or it can derive its tempo from the audio signal it is receiving. To synchronize to audio, *PluggoSync* expects a short pulse at the beginning of each bar. We distribute a file called sync.aiff for this purpose that can be imported into an audio track in the sequencer. Simply loop the track or copy the file so that it repeats every bar. *PluggoSync* has the ability to chase a changing tempo.

PluggoSync uses the following Max **send** objects to broadcast messages about the current synchronization state.

plugsync_ms - current number of milliseconds per bar
plugsync_click - a bang at the top of each bar
plugsync_bpm - current tempo in beats per minute
plugsync_1 - user-defined sync code (see below)
plugsync_2 - user-defined sync code (see below)
plugsync_3 - user-defined sync code (see below)
plugsync_4 - user-defined sync code (see below)

To receive these messages in your own plug-in, use a **receive** object with the name corresponding to the information you want. For example, if you want to know when each bar starts, use receive plugsync_click.

Note that the plugsync_5 output available in version 1.0 has been removed.

In the *PluggoSync* edit window, the user can select a beat division factor for each of the five plugsync signals. They use this to factor to broadcast a numbered beat count on each subdivision of the bar.

For example, if *PluggoSync* is set to internal clock running at 120 bpm, it takes two seconds for each bar to go by. Therefore *PluggoSync* will send the number 2000 to plugsync_ms. If the user has selected a beat division of 4 for plugsync_1, the following messages are sent:

- at the top of the bar, plugsync_click sends a bang and plugsync_1 sends a 1

- at 500 ms past the top of the bar, plugsync_1 sends a 2

- at 1000 ms past the top of the bar, plugsync_1 sends a 3

- at 1500 ms past the top of the bar, plugsync_1 sends a 4

The incoming audio sync signal is echoed to a **plugsend**~ object named pluggoSyncAudio. You can use a **plugreceive**~ with this name to pick it up if you'd like to do something with it.

PluggoSync outputs a PluggoSync now outputs an audio signal the moves from 0 to 1 over the duration of a beat. Use **plugreceive~** named plugsync_phasor to receive this information. *PluggoSync* assumes 4/4 time - you will get 4 phasor cycles for each *PluggoSync* bar

PluggoBus

Audio signals on the PluggoBus can be found on eight different **plugsend**~/**plugreceive**~ "channels." You can pick up a signal off the bus with a **plugreceive**~ object, and put a signal onto the bus with a **plugsend**~ object.

The names are as follows:

PluggoBus1L left channel of PluggoBus 1

PluggoBus1R right channel of PluggoBus 1

PluggoBus2L left channel of PluggoBus 2

PluggoBus2R right channel of PluggoBus 2

PluggoBus3L left channel of PluggoBus 3

PluggoBus3R right channel of PluggoBus 3

PluggoBus4L left channel of PluggoBus 4

PluggoBus4R right channel of PluggoBus 4

The source patchers for both *PluggoBus Send* and *PluggoBus Rcv* are included in the development materials. You can copy these as necessary if you wish to add either selectable PluggoBus send or receive capability into your plug-in.

User Interface Limitations

There are some differences between the runtime Max environment as it exists in MAXplay and in the plug-in environment. Perhaps the most important one is that the runtime plug-in environment lets you load a single patcher (with subpatchers), but you can't view what's in the subpatchers. Instead, you have a fixed size "view" into a portion of a single patcher window. In the VST specification there is no provision for opening anything beyond a single editing window, so this is a relatively permanent limitation. As a part of this limitation, double-clicking on any Max object within the interface view is disabled. We can't, however, prevent objects from opening windows in other ways, but it is your responsibility to avoid doing this (and prevent the user from doing this), since most host environments will crash if a foreign window enters their space.

Audio Processing

The **adc**~ and **dac**~ objects are completely non-functional in the runtime plug-in environment. To get any kind of audio input or output, you will need to use **plugin**~ and **plugout**~. In addition, the I/O vector size, signal vector size, and sampling rate are determined by the host environment. For older versions of the Sound Manager, many sequencers uses an I/O vector size of 1056, which corresponds to some magic value that an FFT-ignorant hardware engineer at Apple thought was, in the immortal words of William Casey as reported by Oliver North, "a neat idea." Luckily, 32, a power of 2, will divide into this number (1056 is 32 x 33), so 32 has become the one and only signal vector size used in the runtime environment. The reason for this is that it allows all plug-ins to share the same set of signal vectors, which is a gigantic win in terms of CPU utilization since it allows memory that is already in the data cache to be reused. The processor operates much more efficiently when it can access data that is already in the cache.

Note that it is impossible to know the host environment's "real" audio input or output device. Your patcher, when it works as a plug-in, only has the input and output audio vectors to communicate with the outside world.

Initialization

Often people write initialization schemes in their audio patchers that employ **delay** or **pipe** objects triggered from **loadbang** objects. Usually, this is an attempt to get something to initialize in the proper order. The problem is that these schemes will not initialize the patcher before the audio is turned on. The execution of events connected to **loadbang** objects is the only thing that happens in the runtime plug-in environment after a patcher and before the DSP chain is compiled and executed.

This has specific implications for the cycle~ object when you supply it with a **buffer**~ object name as an argument. The cycle~ object *copies* its wavetable data from the associated **buffer**~ when the DSP chain is compiled. This means that if the contents of the **buffer**~ have not already been prepared, the **cycle**~ object will not have the right data in its wavetable, and nothing short of sending the **cycle**~ object a set message will correct the problem. The solution is to initialize the **buffer**~ entirely from the output of the **loadbang** without depending on the scheduler, either

by using the **Uzi** object or by reading a file into the **buffer**~. Do not use objects that use the scheduler such as **line** and **metro** to do this sort of initialization.

The Max Window

In the Messages view of a plug-in edit window, you can see the last few lines of the Max window. This will help you reveal errors in your code or report certain types of diagnostic information. This is more of a developer than end-user feature, so you may want to disable the Messages view if slickness is your main concern. Note that the Max window is global to all plug-ins, so you may see error messages and posts generated by other plug-ins in your own plug-in's Messages view.

Multiple Plug-in Issues

You might ask, could I create more than one plug-in and then have them communicate using **send** and **receive**, or **send**~ and **receive**~? The answer is yes for **send** and **receive**, but no for **send**~ and **receive**~. For communicating audio signals, you need to use **plugsend**~ and **plugreceive**~. When there are several plug-ins running in the environment, the Max "name space" is shared but the signal processing "space" is independent; each plug-in processes audio independently.

If you want to send data from one plug-in to another, then you can leave the names associated with objects such as **send**, **receive**, **table**, and **buffer**~ alone. But you cannot count on communicating data within a plug-in using these objects without using special symbols that are not "global" in the Max name space. This is due to the fact that the user can load more than one copy of your plug-in, causing anything you put into a **send** object in one plug-in to come out the **receive** objects of all similar plug-in instances.

The solution—which works only in the runtime plug-in environment—is to start any name you want protected from other plug-ins in the environment with three dashes (---). Examples are shown below.

The transformation of symbols starting with three dashes is only guaranteed to occur correctly when the plug-in is being loaded. You cannot currently generate a new symbol on the fly using an object such as **sprintf** that would be guaranteed to be local to the plug-in context in which it was created.

Priority Level Concerns

This section presents some more in-depth information discussing the internal architecture of the runtime plug-in environment that may be of use to you in understanding some of the issues involved in its priority levels for control messages that are subtly different from those in Max.

The runtime plug-in environment has two basic priority levels, just like Max. Its high priority, or interrupt, level is executed within the audio processing callback known as the process routine. The low priority level is generally executed within an idle-time callback. Currently, the idle-time callback only happens when a plug-in window—and this can be *any* plug-in window owned by the runtime plug-in environment—is open. There are promises that in future versions of VST, idle-time callbacks can be run when a plug-in window is not open, but this has not yet been implemented. So, thus far, with this minor exception, the story is similar to the two levels in Max when Overdrive mode is enabled. However, the story gets more complicated when the host tells the plug-in to change a parameter value or its current effect program.

Parameters can be changed by the host as well as by the user of the plug-in. When parameter automation is employed by the host mixer, parameter changes may be sent from the host to the plug-in at interrupt level. The same holds true if the host automates program changes to the plug-in. Since there is no way in the current VST standard to know the priority level of the communication from the host to the plug-in, the runtime plug-in environment has to assume that all parameter and program change messages from the host occur at interrupt level, since this is the most restrictive case in terms of how the plug-in must behave. Therefore, it simulates this condition by setting a flag during the handling of program and parameter changes sent from the host declaring that processing is occurring at interrupt level. Objects that behave correctly at interrupt level in Max should therefore behave correctly in the context of a parameter or program change. There might be potential problems with a fake "interrupt level" that itself can be interrupted, but we haven't found any so far.

The runtime plug-in environment *can* tell the difference between parameters as changed by the host and parameters as changed by the user moving an egg slider or within the plug-in edit window's Max-based interface, and it does not simulate the interrupt-level condition in these two situations.

How does any of this information affect your plug-in? Here are some problems that came up during the development of the *PlugLoop* and *Very Long Delay* plug-ins. In both of these plug-ins, there is a parameter that sets the maximum time of a delay line buffer. The output of a **pp** object is ultimately connected to a message that causes a sample buffer to be resized. In the case of both the **buffer**~ and **tapin**~ objects, resizing a sample buffer is not something that can be done "on the fly"—it must be deferred to low priority since on the Macintosh, one cannot allocate memory at interrupt level. When the host sends a program change message (which results in all **pp** objects sending out new parameter values), the runtime plug-in environment sets a flag telling the Max environment that it is running at interrupt level. This causes the buffer resizing to be deferred until after the program change message. When the host gives the runtime plug-in environment additional idle time, the resizing actually occurs. In this case at least, you can see where parameter automation will not occur properly if the plug-in edit window is closed, since the sample buffer will not be resized until the edit window's idle time callback is run. Apparently Waves plug-ins have a similar problem. You should note however that the vast majority of plug-ins made with Max/MSP will not have a parameter updating problem since almost all parameter changes can be handled immediately.

Runtime Issues

In implementing *PlugLoop* and *Very Long Delay*, we did a few things to minimize some of the potential problems created by the need to defer a time- and memory-consuming operation:

- We used a **change** object on the output of **pp** so that the buffers will not be resized unless the parameter value actually changes
- All preset effects programs have the same maximum delay time, so switching among the presets will not cause buffers to be resized.
- The number box that changes the maximum time in *PlugLoop* is set to output only on mouse up, so that the buffers won't resize as the user is deciding how large they should be.

Another problem in *PlugLoop* was that the clear message was being sent to the **buffer**~ object after a resize. Since *PlugLoop* uses three **buffer**~ objects, this was a lot of clearing, and, unlike the resize action, a clear to a **buffer**~ will be executed immediately. If the program change arrived at interrupt level, this means the clear would have been executed at interrupt level. It was worse, since there was a **delay** object before the clear message, causing the message always to be sent (and handled) at interrupt level. The effect of the interrupt-level clearing was to peg the CPU utilization meters of host sequencers. The clear message was unnecessary, since the **buffer**~ now clears its memory after resizing, so it was removed. This is the sort of thing you could get away with in Max, where no attempt to protect the user from massive CPU overload occurs (one could argue that a click in the audio output is a lot easier for the user to deal with than a window popping up and the sound stopping), but not in a sequencer, where apparently, protecting the user from hearing any clicks is of prime importance.

Discontinuous DSP Networks

Consider a plug-in that takes input from **plugin**~ and sends it to **record**~ to write the data into a buffer. It then reads from the **buffer**~ using **play**~ and sends the output to **plugout**~. The *Slice-n-Dice* plug-in worked like this, until we discovered it would not work as an insert effect in Cubase. As a send effect it was fine. What was happening?

Insert effects in Cubase pass the same signal vectors for their input and output. This means that you cannot write a sample to the output vector before reading it from the input vector, otherwise you will write over your input. When you give it a configuration of DSP objects and connections, the MSP signal compiler—which determines the order of processing in order to execute your algorithm—is not always that smart. In the *Slice-n-Dice* case, it had decided it was OK to perform the **play**~ to **plugout**~ section of the algorithm before the **plugin**~ to **record**~ section. This caused the output vector to be written before the corresponding input vector was read. This does not cause problems (other than a signal vector size delay) in Max, nor in the send effect case (nor in Vision, where the input and output vectors of an insert effect are never the same). But it was a big problem in the Cubase insert effect case since, due to the fact that the **buffer**~ was initially empty, the **play**~ to **plugout**~ code would zero the output (which was also the input) before it could be written into the **buffer**~. The cycle would repeat itself as the zeroed input continued to be written to the **buffer**~, and no sound would be produced from the plug-in at all.

There are a number of solutions to this problem. The one we adopted was to add a direct signal path from the input to the output along with a gain control. This was needed to the make the effect more useful as an insert effect anyway, and it caused the proper sorting of the DSP network by the signal compiler, since there was no longer an ambiguity about which of two discontinuous "pieces" of the network should be ordered first.

The moral of the story is that if you want to avoid these types of problems you should always include a continuous path of signal objects from **plugin~** to **plugout~** in your plug-in.

Collectives

The Max collective file has a feature that allows you to open more one patcher when the collective is opened. You do this by including multiple open commands in the collective script. The runtime plug-in environment will only open one patcher when your collective is loaded—the first one it finds. Since it's not easy to guarantee what patcher is the first one, you should avoid using this feature. You can safely include abstractions (separate patcher files that are loaded as subpatchers). These should be added automatically when the collective builder analyzes the dependencies of your patcher.

Appendix A *What's not in the plug-in environment*

Objects Not Included

bendin
benout
ctlin
ctlout
follow
midiformat
midin
midiout
midiparse
notein
noteout
polyin
polyout
seq
sysexin
timeline
touchin
touchout

External Object Support Functions Not Available

Note: This list is only of interest to developers who might be writing their own Max/MSP objects in C. All MSP support functions are available in the runtime plug-in environment.

`editor_register`
`event_avoidRect`
`event_box`
`event_clock`
`event_new`
`event_offsetRect`
`event_save`
`event_schedule`
`event_spool`
`gwind_get`
`gwind_new`
`gwind_offscreen`
`gwind_setport`
`message_patcherRegister` (exported but does nothing)

`message_patcherUnregister` (exported but does nothing)

`message_register` (exported but does nothing)

`message_unregister` (exported but does nothing)

`midiinfo`

`off_copy`

`off_copyrect`

`off_free`

`off_maxrect`

`off_new`

`off_size`

`off_tooff`

`sprite_draw`

`sprite_erase`

`sprite_move`

`sprite_moveto`

`sprite_new`

`sprite_newpriority`

`sprite_rect`

`sprite_redraw`

`track_clipBegin`

`track_clipEnd`

`track_drawDragParam`

`track_drawDragTime`

`track_drawTime`

`track_eraseDragTime`

`track_MSToPix`

`track_MSToPos,`

`track_pixToMS`

`track_posToMS`

`track_setport`

Index

Index