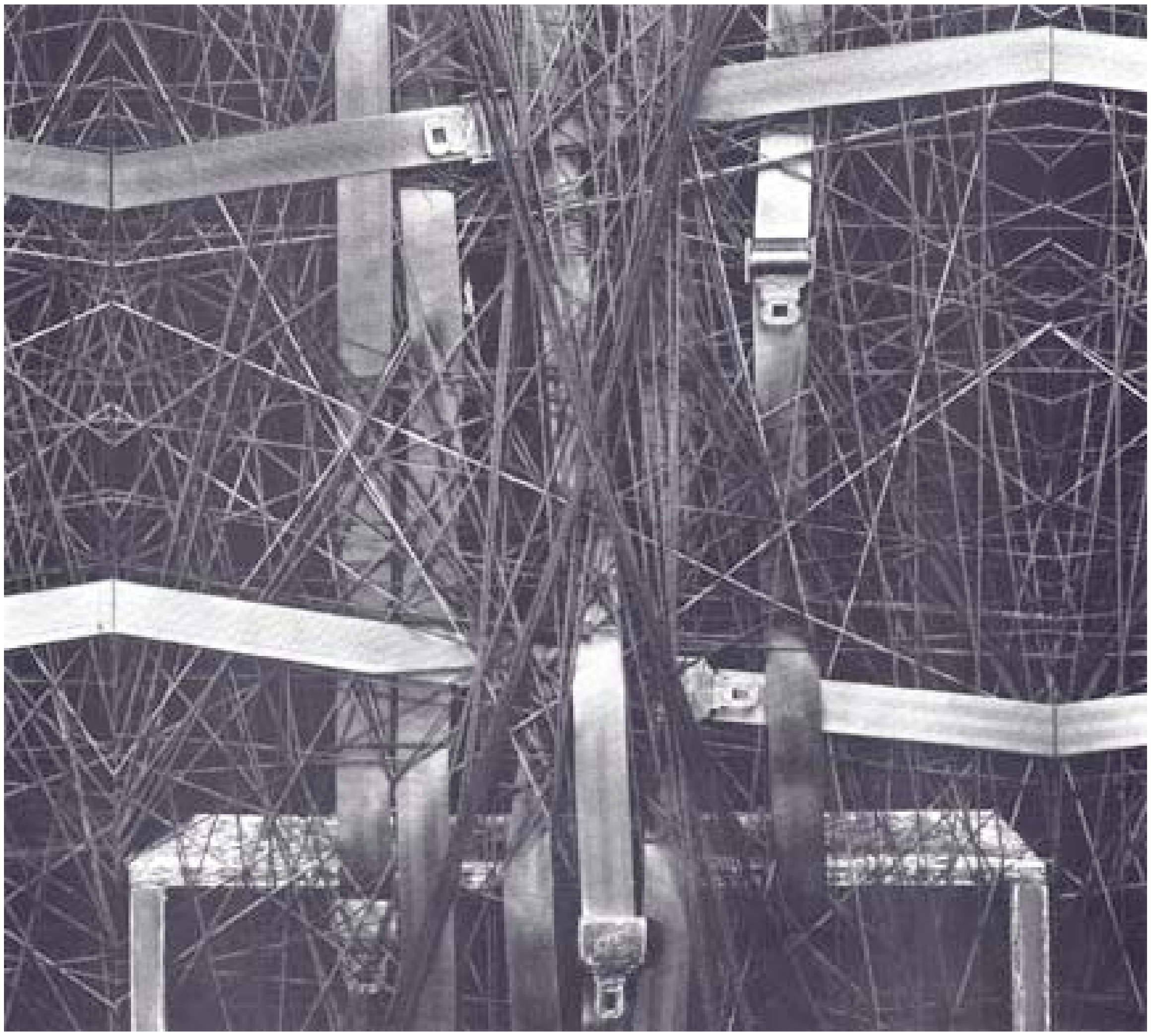
MAX



Tutorial

Version 4.6/7 August 2006

Copyright and Trademark Notices

Credits

Original Max Documentation: Chris Dobrian

Max 4.6 Reference Manual: David Zicarelli, Gregory Taylor, Joshua Kit Clayton, jhno, Richard Dudas, Ben Nevile

Max 4.6 Tutorial: David Zicarelli, Gregory Taylor, Jeremy Bernstein, Adam Schabtach, Richard Dudas, R. Luke DuBois

Max 4.6 Topics Manual: David Zicarelli, Gregory Taylor

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Table of Contents

Table of Contents

Introduction

This manual is a step-by-step course designed to teach you all about Max, beginning with the simplest concepts and building upon those concepts as you learn new ones. The course is primarily for new Max users who don't have prior programming experience, but even if you have some knowledge of programming, the Tutorial is a good way to learn Max.

The tutorials are designed to be read in order. Each Tutorial is accompanied by a sample Max program (document) in the *Max Tutorial* folder. The document is a working illustration of the concepts in the chapter text—it lets you see Max in action and try things yourself. We feel this hands- on approach is a more efficient way to learn about Max than just reading the manual by itself.

By the time you have completed the tutorials, you will have a good understanding of Max and its capabilities, and will probably also have many ideas for your own Max applications.

As you read each tutorial, you can open the corresponding Max document in the *Max Tutorial* folder. Some of the tutorials take the form of "quizzes" so you can be sure you understand the material before proceeding. At the end of each Tutorial are suggestions—labeled **See Also**—of other sections of the Max documentation you can investigate in order to learn more.

If you are new to Max, we suggest you begin by reading the *Setup and Overview* sections of the Fundamentals manual, then trying a few of the Tutorials. You can also learn by looking at the help files in the *max-help* folder, and by browsing the *Max Object Thesaurus* in the Max Reference Manual. The sample patches show some of the things others have done with Max.

Manual Conventions

The central building block of Max is the *object*. Names of objects are always displayed in bold type, **like this**.

Messages (the arguments that are passed to and from objects) are displayed in plain type, like this.

In the **See Also** sections, anything in regular type is a reference to a section of either this manual or the Reference manual.

Introduction

MIDI Equipment

The first few tutorials in this manual do not deal with MIDI directly, but simply teach you about some of the elements of Max. Later tutorials do involve MIDI quite extensively, though, and in the sample programs we make certain assumptions about what MIDI equipment you are using and how it is connected to the computer. In order to benefit the most from the Tutorial, keep in mind these assumptions:

1. You are using a a 61-key velocity-sensitive keyboard with pitch bend and modulation wheels and a polyphonic synthesizer or sampler. Your keyboard should ideally be set to send on MIDI channel 1, and the synthesizer set to receive in Omni On mode.

2. You have connected the MIDI Out of your MIDI keyboard to the MIDI In of your MIDI interface, and connected the MIDI Out of your interface to the MIDI In of your synthesizer or sampler.

3. For the purpose of this Tutorial, your MIDI interface should be connected to the modem port or the primary USB interface of your computer.

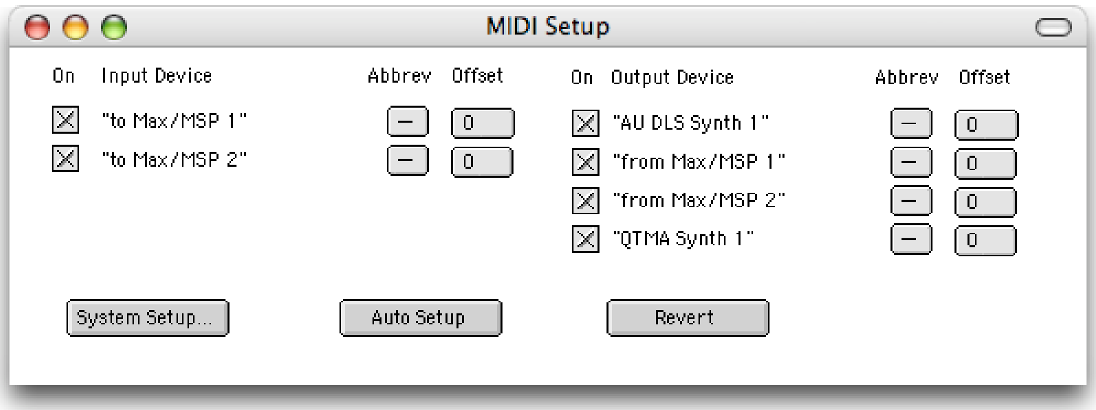
Even if your equipment doesn't exactly match that assumed by the Tutorial, try to emulate the assumed setup as much as possible. You may want to read the user's manual of your synthesizer, to be sure you understand its MIDI capabilities.

The Tutorial patches are designed for a keyboard synth with local control on—one that makes sounds when you play, without receiving any additional MIDI in—rather than for a keyboard controller with no built-in synth. If the keyboard and tone generator you are using are separate, you should open the patch called *thru* in the Max Tutorial folder, specify your input and output ports with the pop-up menus, and leave it open as you run the Tutorial patches. This will route the MIDI output of your keyboard directly through Max to your tone generator, emulating a keyboard synth.

Open Tutorial 1

- If you have not already started up the Max application, do so now by double-clicking on the Max application icon.

- Once Max is launched, open the MIDI Setup window by choosing **Midi Setup…** from the File Menu and assign the input and output devices you want to use to port abbreviation a with a channel offset of 0.



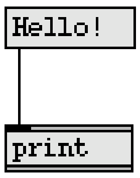
On	Input Device	Abbrev	Offset	On	Output Device	Abbrev	Offset
☒	"to Max/MSP 1"	–	0	☒	"AU DLS Synth 1"	–	0
☒	"to Max/MSP 2"	–	0	☒	"from Max/MSP 1"	–	0
				☒	"from Max/MSP 2"	–	0
				☒	"QTMA Synth 1"	–	0

Both the Macintosh and Windows XP provide a DLS (Downloadable Soundfont) synthesizer for MIDI playback. If you don't want to use any external MIDI gear, you can drive the DLS synth directly from Max via MIDI. For more information about working with DLS synths, see the section "Using Max with MIDI" chapter in the **Fundamentals** manual.

To open the sample program for each chapter of the Tutorial, choose **Open…** from the File menu and find the document in the *Max Tutorial* folder with the same number as the chapter you are reading.

Tutorial 1

- Open the file called *1. Saying "Hello!"*



Objects and Messages

- Click in the box marked Hello!. Notice what happens in the Max window each time you click on Hello!.

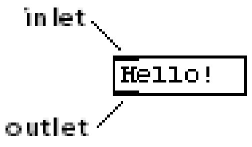
The basic operation of a patcher program is simple. Different types of boxes, called *objects*, send *messages* to each other through *patch cords*.

This program contains two different objects:

The box containing the word print is a **print** object. A **print** object prints whatever message it receives in the Max window.

The word Hello! is a *message* contained in a **message** box, which can contain anything that can be typed. Often a message will contain numbers.

Different kinds of objects have different numbers of *inlets* and *outlets*. The **message** box always has one inlet and one outlet.



Inlets are always at the top, indicated by blackened areas at the top of an object. Outlets are always at the bottom of an object.

The **print** object has no outlet—its output is always just printed in the Max window. Usually, an object will have *both* inlets and outlets; it receives messages, performs some task, then sends out messages. The **print** object just prints whatever it receives.

The **message** box is connected to the **print** object by means of a *patch cord*. Just like components of a stereo system, the outlet of one object is connected to the inlet of another object. You can't connect an inlet to another inlet, or an outlet to another outlet.

The program operates as follows:

1. When you click on the **message** box object, the message Hello! is sent out the **message** box's outlet and through the patch cord.

2. The message reaches the inlet of the **print** object, which prints the message print: Hello! in the Max window.

Locking and Unlocking a Patcher Window

A Patcher window can be in one of two states: *locked* or *unlocked*. When a Patcher window is *locked*, it is a program ready to run, and you can operate the patcher by clicking on objects (such as sliders) that do things. When it is unlocked, you can edit the patcher by moving objects around, creating new ones, and connecting objects together.

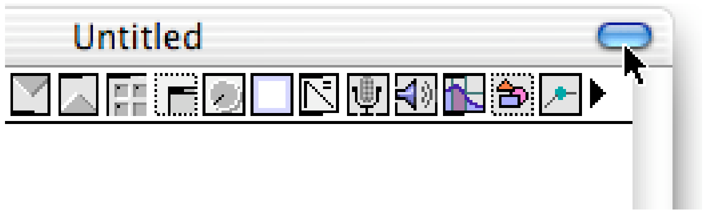
The lock/unlock state of the window is indicated by the presence of the patcher palette at the top of the window. If you see the palette, then the window is unlocked.

There are several other ways you can lock or unlock a patcher.

- Choose **Edit** from the View menu, or type Command-E on Macintosh or Control-E on Windows.

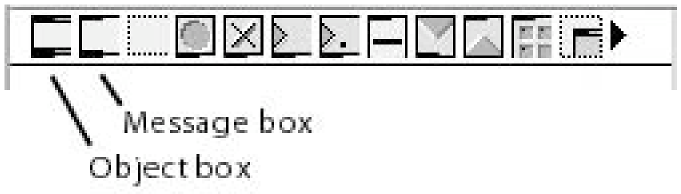
- Command-click on Macintosh or Control-click on Windows on the "white space" in the Patcher window.

- On Macintosh, there is a transparent rectangular pill on the right side of the window that can be used for toggling between locked and unlocked state.



- Using one of these methods, unlock the Patcher window. You should be able to see the palette. You can now modify the program.

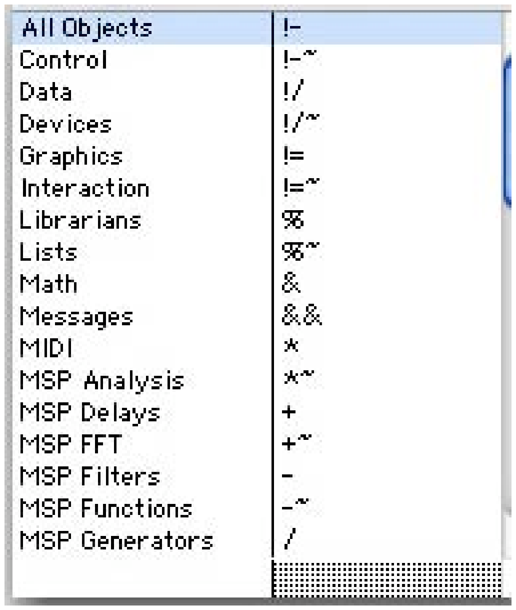
The first two items in the palette are the object box and the message box.



Modifying the Patch

Now we'll produce a program that prints "Good-bye!"

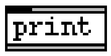
- Click on the object box in the palette. The cursor turns into an object icon. Click inside the Patcher window, near the bottom-right corner. A list of pre-defined Max objects called the *New Object List* will appear. (If the list does not appear, it's because **New Object List** is not checked in the Options menu. You can bring up the Object List by Option-clicking the empty object box on Macintosh or Alt-clicking the empty object box in Windows. Or, if you want the list to always appear, check **New Object List** in the Options menu.)



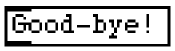
- Scroll down through the right-hand column of the New Object List until you see **print**, click on the word to select it, and type the Return or Enter keys on Macintosh or the Enter key on Windows. Alternatively, you could just type the first few letters of the word "print" until it is selected in the list.

Note: If you want to type in the name of an object without using the New Object List, type the Delete (Backspace) key or click anywhere outside the New Object List and it will go away. You can also hold down the Option key on Macintosh or the Alt key on Windows as you place the object box in the window if you want to temporarily toggle the **New Object List** option on or off.

- You now have an object box with the word print in it. Type the Enter key on Macintosh or the Shift and Enter keys on Windows, or click anywhere else in the window (outside that object box), and a **print** object is created with an inlet at the top.

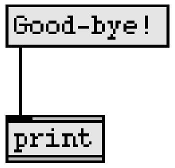


- Next, click on the message box icon in the palette, and click just above your **print** object, to place a new **message** object in the window. Type Good-bye! into your **message** object.



To connect the **message** object to the **print** object, drag from the outlet of the **message** object to any place inside the **print** object:

- Position the cursor on the outlet of your **message** box. When the cursor is over an outlet, the outlet expands. Click on the expanded outlet and drag until the cursor is inside your **print** object and you see the *inlet* of the **print** object expand. Then release the mouse button. This will create a patch cord connection between the two objects.



Note: If you are unable to connect a patch cord according to the method described in the preceding paragraph, it's probably because **Segmented Patch Cords** is checked in the Options menu. For the moment, that option should be unchecked.

If your **message** object and your **print** object are not perfectly aligned vertically, the patch cord will appear jagged. This has no effect on the functioning of the patch. However, if you're a fastidious person and want to clean up the appearance of your patch, select both objects just as you'd select multiple icons in the Finder (by Shift-clicking on each of them or by dragging across both of them with the mouse). Then choose **Align** from the Object menu.

You can also move objects by dragging them to the desired location. Objects and patch cords can be removed entirely by clicking on them to select them, then pressing the Delete key on Macintosh or the Backspace key on Windows or choosing **Cut** or **Clear** from the Edit menu.

- Type Command-E on Macintosh or Control-E on Windows to lock the Patcher window. Your program is now ready to run. Click on the **message** box containing Good-bye! and you should see print: Good-bye! in the Max window.

Summary

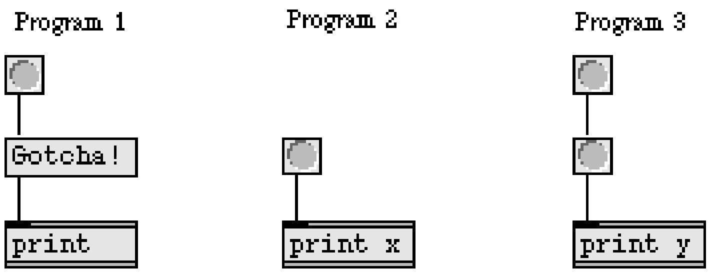
When a Patcher window is *unlocked*, it is in **Edit** mode, and can be modified. When the window is locked, the program is ready to run. You can also run the program by holding down the Command key on Macintosh or Control key on Windows and clicking in the Patcher window.

A *message* is sent through a *patch cord* from the *outlet* of one object to the *inlet* of another. A **message** box contains any message you type into it. When you click on a **message** box, it sends its message out the outlet. A **print** object prints in the Max window whatever message it receives in its inlet.

See Also

message	Send any message
print	Print any message in the Max window
Objects	Creating a new object in the Patcher window

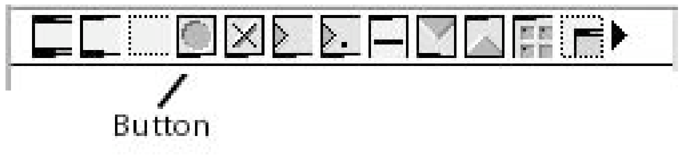
The bang **Message**



- This program is actually three separate printing programs. Click on the **button** icons in each program and notice what gets printed in the Max window.

The first thing to observe is that two of the **print** objects have names: x and y. Since there can be any number of **print** objects in a Patcher window, you will often want to make it clear which one is actually printing a message. You do this by putting a name after the word **print** in the box. When there's no name, the message is preceded by print:, as in Program 1. When there is a name, it precedes the message, as in Programs 2 and 3.

The second important new thing in this window is the **button** object. It appears as a separate item in the palette, and is really very much like a **message** box that contains the message bang.



You see, bang is a magic word in Max. It's a special message that means, "Do it!", which causes an object to do whatever it's supposed to do. For example, a **message** box sends out the message it contains in response to a bang or a mouse click.

- In Program 1, you can click on the "Gotcha!" **message** box to print it, or you can click on the **button**, which sends a bang message to the inlet of the **message** box. The effect is equivalent, since in either case the **message** box is "triggered" and sends out the message it contains.

- Program 2 not only proves that the **button** quite literally sends the message bang; it also proves that bang has no special effect on the **print** object. That's because the **print** object doesn't try to understand the message it receives. Its only purpose in life is to print out what arrives in its inlet.

- Program 3 is sort of a puzzler. Clicking on either **button** produces a printout of y: bang.

When you click on the upper **button**, which **button** actually supplies the message to the **print** object?

The answer is the lower **button**.

The upper **button** sends a bang message to the lower **button**. The lower **button** interprets the bang message as "Do it!", and performs its expected function, which is to send a bang message. The **print** object simply prints out what it receives.

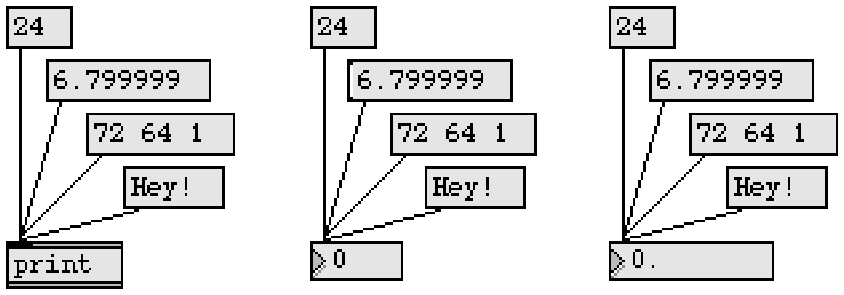
Summary

bang is a special triggering message that causes an object to perform its task. The **button** object's task is to send out the message bang, thus triggering other objects.

See Also

button	Flash on any message, send a bang

int, float, and list



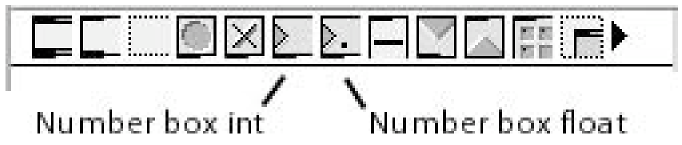
We have seen that a message can consist of text, and that some words have a special meaning to certain objects, such as the word bang. Commonly, a message will consist of one or more numbers.
Max distinguishes integer numbers from decimal numbers (with a fractional part). Integer numbers are stored in Max in a data type called *int*, and decimal numbers are stored in a data type called *float*.
Most of the time you won't really need to worry about this distinction in how numbers are stored, because Max will take care of it for you, and will even convert an int into a float or vice versa if it needs to (for instance, if a float is received by an object that expects to receive an int). The main thing you need to know is that when a float is converted to an int, its fractional part is not rounded off, but is *truncated*. (The fractional part is just chopped off.) For example, the number 6.799999 does not become 7, it becomes 6.

A message can also consist of several numbers, separated by spaces, which are all sent together. This is known as a *list*. A list can consist of both ints and floats. You'll encounter lists in later chapters of the Tutorial.

Number box

If you want to show a number in a Patcher window, use a **number box**. There are two **number box** icons available in the object palette, one for showing ints and one for showing floats.

Tutorial 3



A number received in the inlet of a **number box** is displayed and passed on out the outlet. This is an effective object to use as a "wiretap" to see what is the most recent number to have passed through a patch cord.

- Click on the different **message** boxes, and notice what is displayed, either in the **number box** objects or in the Max window.

Notice a couple of important differences between printing messages with a **print** object, and displaying them with a **number box**.

1. The **print** object will print any message it receives, regardless of the content of the message. The **number box**, on the other hand, can display only one number at a time. If it receives a list, it displays (and passes on) only the first number in the list. If it receives an arbitrary text message, it does nothing except complain that it doesn't understand that message.

2. A **number box** can show only an int or a float. If an int **number box** receives a float, it converts the number to int, and vice versa.

The **number box** has other features not described here. This patch does show one of its most common uses, though—to display the number that has most recently passed through a patch cord. You will learn more in the *Number box* Tutorial.

Summary

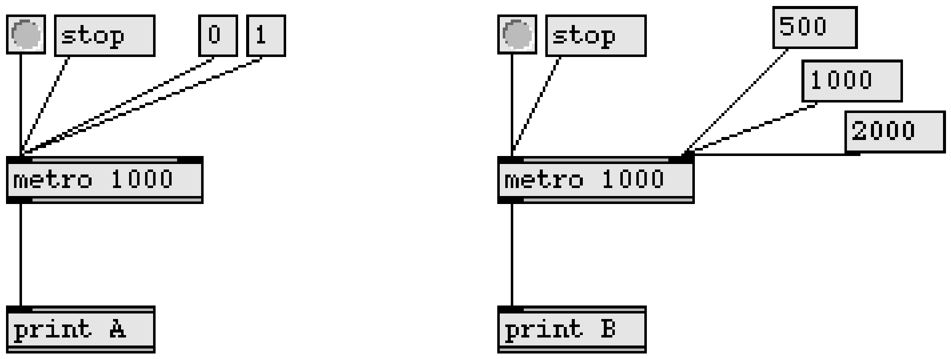
A Max message can consist of a single number, of type *int* (for integers) or *float* (for decimals). Many numbers used in Max (such as MIDI data and millisecond time values) are ints. A message can also consist of a space-separated *list* of numbers, which are all sent together in one message.

A **number box** shows the most recent number it has received, and passes that number on out the outlet. A **number box** is either of type *int* or *float*, and will convert numbers to that type.

See Also

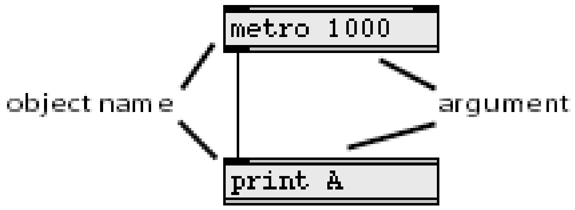
number box	Display and output a number

Object Names and Arguments



In this chapter, we introduce a new object called **metro**, which functions as a metronome. You will notice that we have typed in a number after the word **metro** in the object box. This is the number of milliseconds between ticks of the metronome.

The number after the word **metro** is called an *argument*. We have already seen arguments used to give names to **print** objects. Arguments typically give objects information necessary to do their job.



Some objects require typed-in arguments in order to function. More commonly, an argument is optional, to supply some starting value, as in the case of **metro** where the argument determines the initial speed of the metronome. When **metro** is started, it sends out a bang message every *n* milliseconds (where *n* is the argument) until the metronome is stopped. If no argument is typed in, metro has a *default* value of 5, and sends out a bang every 5 milliseconds.

The **metro** object has two inlets. A message received in the left inlet can start or stop the metronome. The metronome will start when it receives any non-zero number in its left inlet, and it will stop when it receives a 0. Alternatively, you can send it a bang message to start, and stop to stop. A number received in the right inlet will change the number of milliseconds between bang outputs that was initially set by the argument.

- Try turning the **metro** objects on and off, and watch what is printed in the Max window.

- Try sending different numbers to the right inlet of **metro**, and notice the change in the speed with which messages are printed. The speed can be changed while the metronome is running, but the change does not take effect until the next bang is sent out.

Because it sends out the message bang, **metro** is a useful object for triggering other objects repeatedly at a specific speed.

Summary

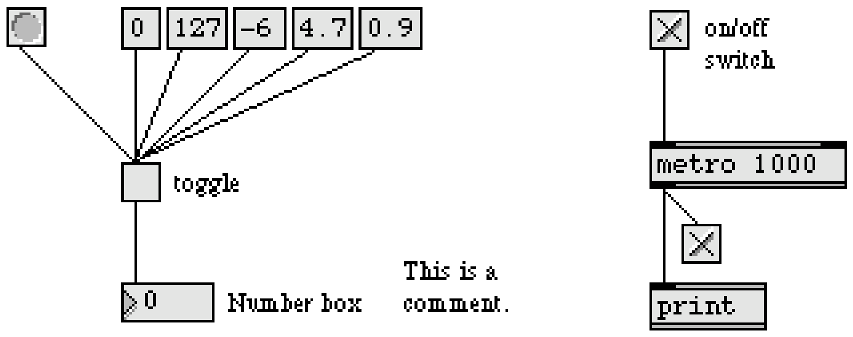
After you type a name into an object box, you can supply additional information by typing in *arguments* after the object name. Arguments are usually *optional*, but some objects have *obligatory* arguments. If optional arguments are not typed in, Max usually supplies a *default* value.

A **metro** sends out bang messages repeatedly at regular intervals of time, until it is stopped. The number of milliseconds between bang messages is specified by the argument or by a number received in the right inlet.

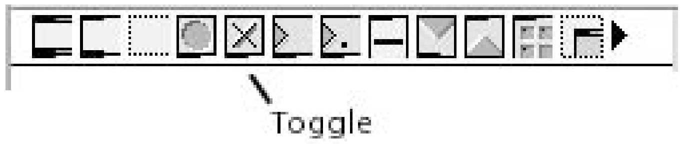
See Also

metro	Output a bang, at regular intervals

toggle



The **toggle** object is the box with an X in it in the object palette. It functions as an indicator or a switch between two states: zero and non-zero.



- Click on the different message boxes containing numbers, and notice what happens to the **toggle** and the **number box**.

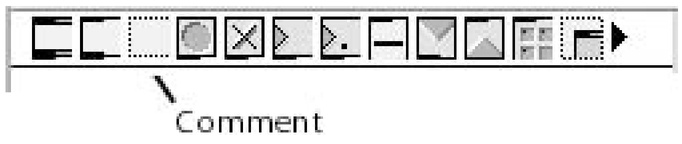
The **toggle** object can receive a number or a bang in its inlet. If the number is non-zero, **toggle** will show an X and send out the number. If the number is 0, the box will be blank and 0 is sent out the outlet. The **toggle** expects to receive an int, so when it receives a float it converts it to int. That is why the number 0.9 is understood as 0 by **toggle**.

The **toggle** alternately sends out the values 1 and 0 each time it is clicked with the mouse or receives a bang in its inlet. When it receives a bang or a mouse click, it reverses its state and sends out the new value. This distinction between zero and non-zero is Max's way of turning things on and off, or distinguishing between true and false.

- Thus, you can use a **toggle** as an on/off switch. In our example, the **metro** object can be turned on and off by clicking on the **toggle**. Try it. This works because **metro** starts when it receives a non-zero number (like 1) and stops when it receives a 0.

comment

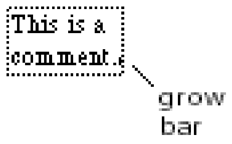
The dotted box in the palette, to the right of the message box, is a **comment**.



A **comment** has no effect on the functioning of a program. It's simply a way of putting text into a Patcher window. The main reasons to add a **comment** are:

1. To label objects in the patch, such as "on/off switch".
2. To give instructions to the user, such as "Click here".
3. To explain the way a program works, or how a particular item in a program functions. This is not only helpful to the user of the program, but is also very helpful to you, the programmer. You'd be amazed how quickly you can forget how your own program works. Get in the habit of adding many explanatory **comments** as you build programs.

A **comment** box (or almost any other object) can be resized by dragging on the grow bar in the lower-right corner of the box.



- You can also change the size of the text in a **comment** (or any other object). Click on the **comment** box to select it, then choose a different font or size from the Font menu. Try changing the font characteristics and the size of the **comment** that says "This is a comment."

When you specify font characteristics with *no* objects selected, you set the characteristics for any new objects you subsequently create in the active window. When you specify font characteristics with the Max window in the foreground, you set the characteristics for *all* new Patchers you subsequently create. Max stores these font characteristics in the Max *Preferences* file, and recalls them each time you use Max.

Tutorial 5

toggle and comment

Summary

A **toggle** can be used to generate the numbers 1 and 0, for turning other objects (such as **metro**) on and off. It can also be used as an indicator of numbers passing through it, telling whether the most recent number was zero or non-zero (although any floats passing through will be converted to int.) A **comment** doesn't do anything, but is useful for putting text in a Patcher window.

See Also

comment	Put explanatory notes or labels in a patch
led	Display on/off status, in color
togedge	Report a change in zero/non-zero values
toggle	Switch between on and off (1 and 0)
ubutton	Transparent button, sends a bang

Tutorial 6: Test 1—Printing

Make a Printing Program

Here is an exercise to make sure that you understand what has been explained so far.

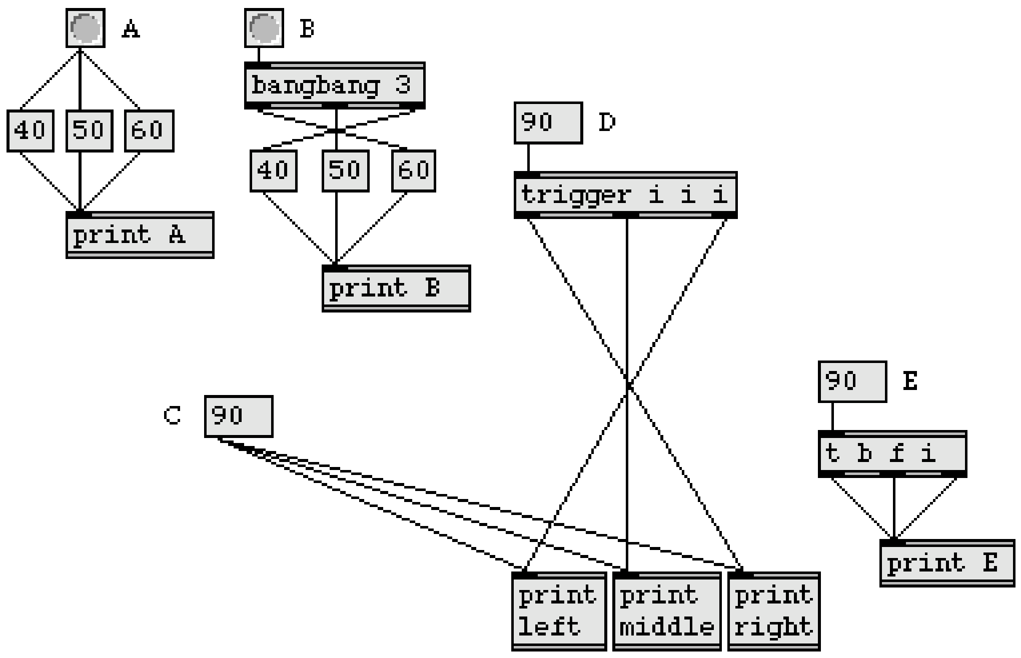
- Create a patcher program which, when turned on, prints the phrase…

```
test: 1
```

…in the Max window every two seconds until it is turned off. Include a way of turning the program on and off.

The answer has been hidden in the right side of the Patcher window. Scroll to the right or enlarge the Patcher window to see the answer.

Message Order



This lesson illustrates that messages in Max are always sent in *right-to-left* order. And, if a message triggers another object, that object will send *its* message(s) before anything else is done. Knowing these two principles can help you figure out exactly how a patcher program is operating.

For example:

- Click on the **button** marked A. The bang message is first sent to the **message** box containing the number 60, that message is sent to the **print** object, and A: 60 is printed in the Max window. Then the bang message is sent to the **message** box containing the number 50, that message is sent to the **print** object, and A: 50 is printed in the Max window. Finally, the bang is sent to the **message** box containing the number 40, that message is sent to the **print** object, and A: 40 is printed in the Max window.

This illustrates the right-to-left order in which bang messages are sent from the outlet of **button** to other objects, and also illustrates that the order of messages continues down the line until no more objects are triggered (in this case, until the **print** object does its job), then goes back to the next patch cord coming out of the **button**, and the next bang is sent.

Tutorial 7

bangbang

The **bangbang** object sends a bang out *each* of its outlets when it receives any message. The number of outlets is specified by the typed-in argument. The order in which the messages are sent out the outlets is still right-to-left: the rightmost outlet sends first and the leftmost outlet sends last.

- Click on the **button** marked "B", and you will see that when an object (such as **bangbang**) has more than one outlet, messages are sent out the outlets in right-to-left order.

When multiple patch cords are connected to a single outlet, as in examples A and C, messages are sent to the receiving objects in order of their right-to-left position, but when a single object has more than one outlet, as in examples B, D, and E, messages are sent out the outlets in right-to-left order, regardless of the destination.

trigger

The **trigger** object is very similar to **bangbang**, but deals with numbers as well as bang messages. Instead of a single argument telling how many outlets there are, the number of outlets a **trigger** object has depends on how many arguments are typed in. Each argument in a **trigger** specifies what the output of an outlet will be: i for int, f for float, b for bang, or l for list (not shown in the example).

- Click on the **message** box 90, marked C. The **print** objects receive the number in right-to-left order, depending on their position.

- Click on the **message** box 90 marked D. Each outlet of the **trigger** has been assigned to send an int, so the number 90 will be sent out each outlet, in order from right-to-left.

- Click on the **message** box 90 marked E. In this example, each outlet has been assigned to send something different. The right outlet sends an int, the middle outlet sends a float, and the left outlet sends a bang.

Note: The names of **bangbang** and **trigger** can be shortened to **b** and **t** (as in example E). Max will still understand these object names.

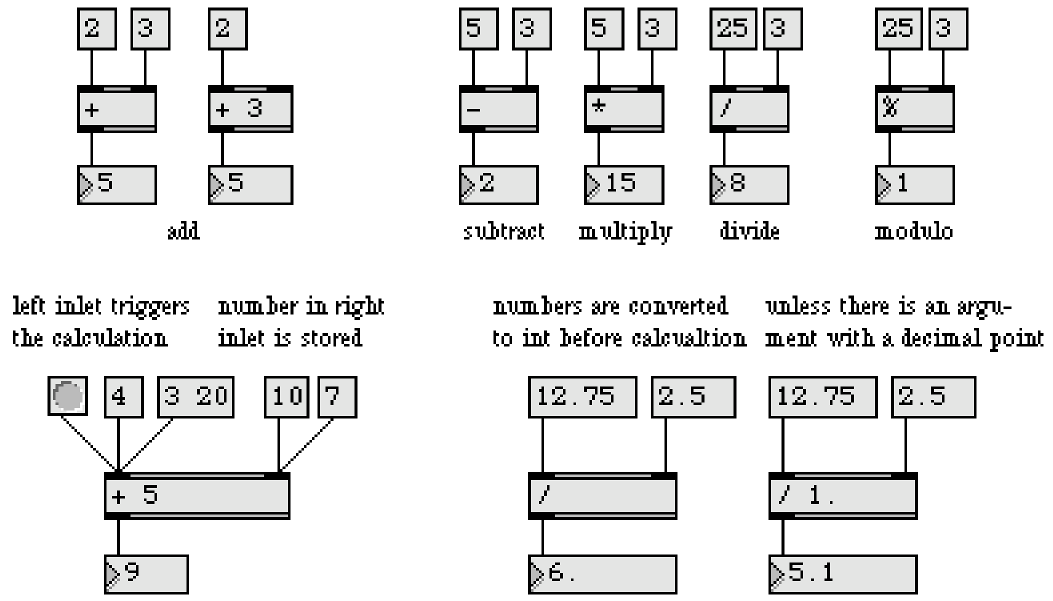
Summary

An object with multiple outlets sends messages out its outlets in order from *right-to-left*. When multiple patch cords are connected to a single outlet, the messages are sent in *right-to-left* order, depending on the position of the receiving objects. (If the receiving objects are perfectly aligned vertically, the order is *bottom-to-top*.) When the **bangbang** object receives any message, it sends a bang out each outlet. When **trigger** receives a number, a list, or a bang, it converts the message into the type assigned to each outlet before sending it out.

See Also

bangbang	Send a bang to many places, in order
buddy	Synchronize arriving numbers, output them together
fswap	Reverse the sequential order of two decimal numbers
swap	Reverse the sequential order of two numbers
trigger	Send a number to many places, in order

Arithmetic Operators


2
3
+
5
2
+ 3
5
add
5
3
-
2
subtract
5
3
*
15
multiply
25
3
/
8
divide
25
3
%
1
modulo
left inlet triggers the calculation
number in right inlet is stored
4
3 20
10
7
+ 5
9
numbers are converted to int before calcualtion
unless there is an argu- ment with a decimal point
12.75
2.5
/
6.
12.75
2.5
/ 1.
5.1

Max has an object for each of the basic arithmetic operations, plus a modulo operator (which gives the remainder when two integers are divided).

We call these objects *operators*—and the numbers they operate upon are called *operands*. Each operator object expects one operand in its right inlet (which it stores) and then the other in its left inlet (which triggers the calculation and the output). An initial value for the right operand can be typed in as an argument. In the upper-left example, you see both methods. Be aware, however, that as soon as a different number is received in the right inlet, it will be stored in place of the initial value, even though that initial value continues to show as the argument.

Left Inlet Triggers the Object

Note that just connecting to an object's inlet does not perform any calculation. You have to *trigger* the calculation by sending a number (or bang) into the *left* inlet. The vast majority of objects are triggered by input received in the left inlet. Input received in the other inlets is usually stored for later use.

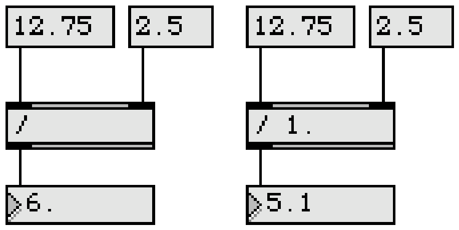
- In the upper examples, click on the **message** boxes above the operators.

Notice that the number coming in the right inlet has to be received *before* the number in the left inlet is received. That is because the message received in the left inlet triggers the calculation with the *most recently received* numbers. If you haven't supplied a number as a typed-in argument (and no number has been received in the right inlet), 0 is the default argument for the **+**, **-**, and ***** objects, and 1 is the default for **/** and **%**.

Int or Float Output

You may have noticed that the / object sends out 8 as the result of 25 ÷ 3. That's because the output is an int, and is truncated before being sent out.

All the arithmetic operators send out an int as the result, unless they have a typed-in argument that contains a decimal point, in which case they are converted to float.



The two division programs at the bottom-right corner of the Patcher window demonstrate converting from one type to another. The first program removes the decimal part of any float numbers it receives. It performs the operation 12 ÷ 2 and outputs a result of 6. The second program divides the numbers 12.75 and 2.5 as floats and gives a full float output, because its typed-in argument contains a decimal point.

If you want an operator always to do float arithmetic operations, give the object an initial argument of a number with a decimal point, and then send the numbers you want it to use in through the left and right inlets.

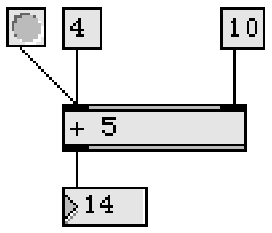
bang **Message in Left Inlet**

The program in the bottom-left corner illustrates a couple of other features of operators.

- First, send the number 4 to the left inlet of the **+** object by clicking on the **message** containing 4. The object performs the calculation 4 + 5 and outputs the result, 9.

- Next, send the number 10 to the right inlet. The number 5 is replaced by the number 10, but no output is sent. Only the left inlet triggers output.

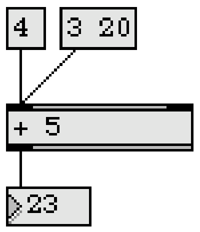
- Now click on the **button** to send a bang to **+**. What happens? The bang causes **+** to "Do It!"—in this case, to do the calculation with the numbers it has most recently received.



List in Left Inlet

Both operands can be sent to an operator *together*, as a list received in the left inlet. The operator will function exactly as if it had received the second number in the right inlet and the first number in the left inlet. The numbers are stored, the calculation is performed, and the result is sent out.

- Click on the **message** box containing 3 20 to see the effect of sending a list to the left inlet.



- Then send the number 4 to the left inlet, and you will see that the number 20 has been stored just as if it had been received in the right inlet.

This demonstrates that when you send a list of numbers to an object with more than one inlet, the numbers are generally distributed to the object's inlets, one number per inlet. You will see other examples of this in future chapters.

Summary

Mathematical calculations are performed by *arithmetic operator* objects: **+**, **-**, *****, **/**, and **%**. The *operands* are received in the two inlets, but only the *left* inlet triggers output. A bang or a *list* in the left inlet can also trigger output. The operators send out an int, unless they have a float argument, in which case they send out a float.

Tutorial 8

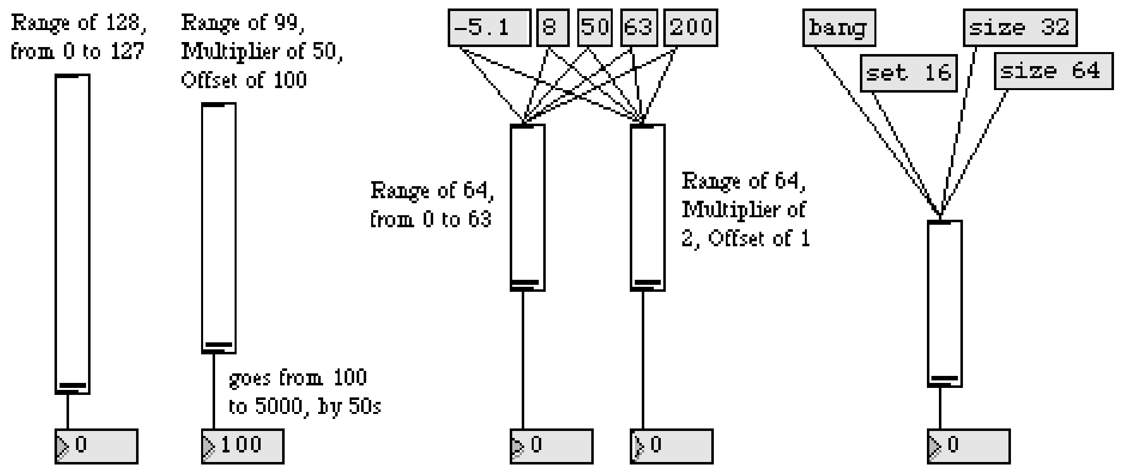
Most objects in Max are triggered by input received in the *left* inlet. A *list* can be received in the left inlet, supplying values to more than one inlet at the same time.

Arithmetic operators are essential for any algorithm involving numerical calculation. Their use will be shown in future programs.

See Also

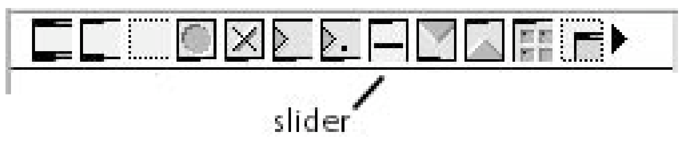
expr	Evaluate a mathematical expression
Tutorial 38	**expr** and **if**

Onscreen Controller


Range of 128, from 0 to 127
Range of 99, Multiplier of 50, Offset of 100
goes from 100 to 5000, by 50s
-5.1
8
50
63
200
Range of 64, from 0 to 63
Range of 64, Multiplier of 2, Offset of 1
bang
set 16
size 32
size 64
0
100
0
0
0

Clicking on a **message** box is one way of sending a number through a patch cord. Another object, the **slider**, lets you send any of a whole range of numbers by dragging with the mouse.

The **slider** object looks like this in the palette…

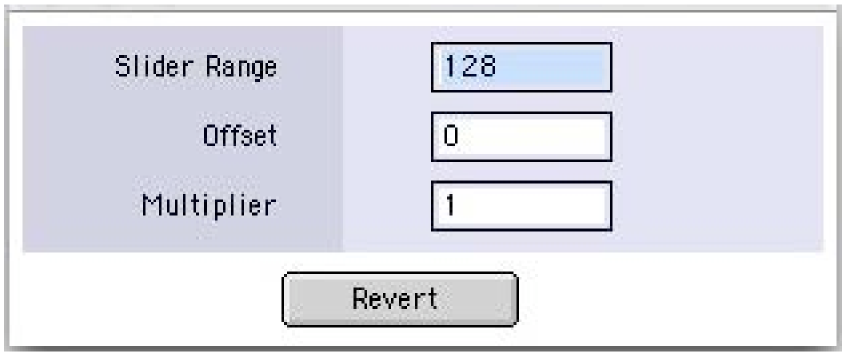


When it is placed in the Patcher window it resembles a slider on a mixing console. Dragging on the **slider** sends out numbers as the mouse is moved.

- Click and drag on the first **slider** in the Patcher window, and see the output in the **number box**.

When you create a new **slider**, its output ranges from 0 to 127. You can change the *Slider Range* by selecting the **slider** (when the Patcher window is unlocked) and choosing **Get Info…** from the Object menu. The **slider** automatically resizes itself to accommodate the specified range.

The **Get Info** dialog box (also called the Inspector) has two other values you can set: a *Multiplier*, by which all numbers will be multiplied before being sent out, and an *Offset*, which will be added to the number, after multiplication.


Slider Range 128
Offset 0
Multiplier 1

- The second **slider** in the Patcher window has a range of 99 (from 0 to 98), but before a number is sent out it is multiplied by 50, then has 100 added to it. So, when the slider is in the lowest position, it will output (0 * 50) + 100, which equals 100. When the slider is in the top position, it will output (98 * 50) + 100, which is 5000.

Many objects let you set options like this with the Inspector.

Graphic Display of Numerical Values

In addition to responding to the mouse, the **slider** will move to whatever number it receives in its inlet. This makes it useful for graphically displaying the numbers passing through it. The *Multiplier* and the *Offset* are also applied to numbers received in the inlet, so the **slider** can actually change values as they pass through.

- Click on the **message** boxes containing numbers, above the middle **slider** objects.

Notice that both **slider** objects move to display the value they have received, but the number that each one sends is different. The **slider** on the left has an *Offset* of 0 and a *Multiplier* of 1, so it doesn't change the number it receives, but the other **slider** multiplies the incoming number by 2 and adds 1 to it.

Notice also that the numbers that are received and sent out can exceed the specified range of the **slider**, and that a float gets converted to int.

Other Inputs

A **slider** can receive other messages in its inlet. When it receives bang, it sends out whatever number it currently is displaying (with the *Multiplier* and *Offset* effects). The word set, followed by a number, sets the value of the **slider** without sending any output. The word size, followed by a number, changes the *Range* of the **slider** to that number.

Summary

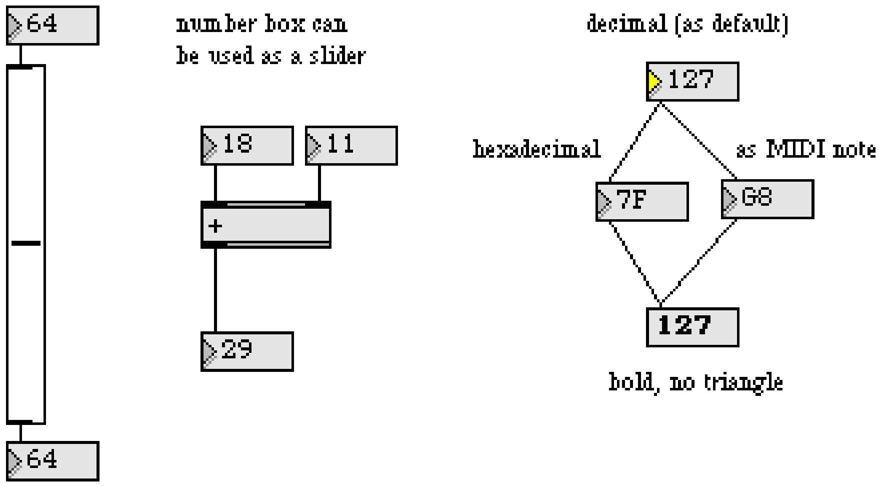
A **slider** lets you output a continuous stream of numbers within a specified range by dragging on it with the mouse. It will also show and send out numbers received in its inlet, making it useful for graphically displaying the numbers passing through it.

By choosing **Get Info…** from the Object menu, you can change the *Slider Range*, and can also specify a *Multiplier*, by which all numbers will be multiplied before being sent out, and an *Offset*, which will be added to the number, after multiplication.

See Also

hslider	Output numbers by moving a slider onscreen
kslider	Output numbers from a keyboard onscreen
rslider	Display the range between two values
slider	Output numbers by moving a slider
uslider	Output numbers by moving a slider onscreen
Tutorial 14	Sliders and dials

Onscreen Controller



In the previous chapter we saw that a **slider** graphically displays the numbers passing through it, and can also send out numbers when you drag on it with the mouse. The **number box** has these same capabilities.

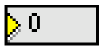
- Try dragging on the **number box** at the top of the Patcher window, and you will see that it can be used as an onscreen controller much like the **slider**.

Unlike the **slider**, the **number box** can have an unlimited range. You can produce virtually any number with the **number box** if you keep dragging.

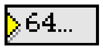
Type In Numbers

You can also type numbers into a **number box** from the computer's keyboard.

- Click on the **number box** at the top of the screen, without dragging. Notice that the triangle in the left edge of the **number box** becomes highlighted, showing that it has been selected.



- Type the number 64 on the computer's keyboard. The number will be followed by an ellipsis, indicating that the number has not yet been sent out the outlet.



- When you have finished typing in the number, you can send it out the outlet with any one of three actions: type the Return or Enter keys on Macintosh or the Enter key on Windows, type the Enter key on Macintosh or the Shift and Enter keys on Windows, or click anywhere in the Patcher window outside of the **number box**.

- While a **number box** is selected in a locked Patcher, you can also raise and lower the number in it by pressing the up and down arrow keys. Holding down one of these arrow keys moves the number up or down continuously, just as if you were dragging on the **number box** with the mouse.

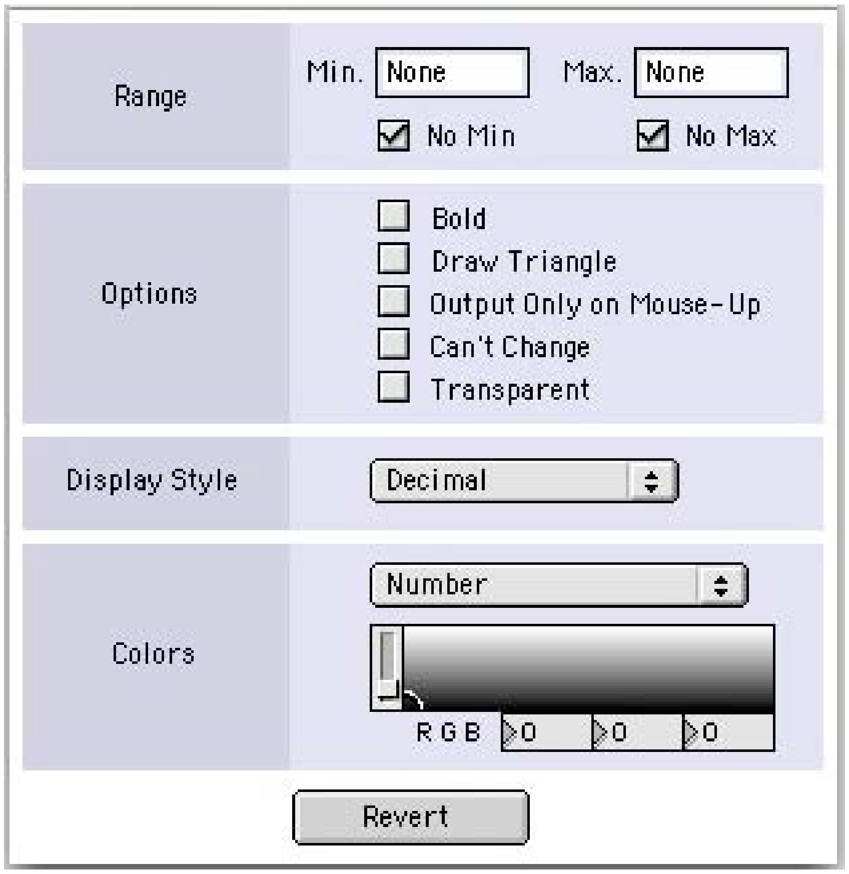
You can see that the **number box** is useful both for displaying the numbers received in the inlet (as in the case of the **number box** below the **slider**), and for allowing you to send numbers by typing them in or dragging with the mouse. The second patch shows the **number box** in both uses—for sending numbers to the **+** object, and for displaying the result.

- Send a number to the right inlet of the + object, either by dragging on the **number box** or by clicking on it and typing in a number. Remember, we want to send a number to the right inlet first, because the left inlet is the one that triggers the addition.

- Now send a number to the left inlet of the + object, and you will see the result of the addition in the bottom **number box**.

Number box Range

You can set many characteristics of a **number box**—how it functions and how it looks—by selecting a **number box** and choosing **Get Info…** from the Object menu to display the **number box** Inspector.


Range
Min. None Max. None
☑ No Min ☑ No Max

Options
☐ Bold
☐ Draw Triangle
☐ Output Only on Mouse-Up
☐ Can't Change
☐ Transparent

Display Style: Decimal

Colors: Number
R G B 0 0 0

Inspector for a number box

When you create a new **number box**, it has an unlimited range. You can limit the range by typing a number into the *Minimum* and *Maximum* boxes in the Inspector.

- Unlock the Patcher window, select the **number box** located above the **slider**, then choose **Get Info…** from the Object menu.
- Click on the checkboxes for *No Min* and *No Max* to disable them. Type the number 0 into the *Minimum* box, and type the number 127 into the *Maximum* box. (You can move from one box to the other by typing the Tab key.) Click OK, then re-lock the Patcher window.
- Now when you drag on the **number box**, it will not exceed the range of 0 to 127.

The *Minimum* and *Maximum* settings of a **number box** limit the range of numbers that can be sent out by dragging on it or by typing in a number, and also limit the range of numbers *passing through* it. Incoming numbers that exceed the specified Minimum and Maximum will be changed to stay within the limits.

Tutorial 10

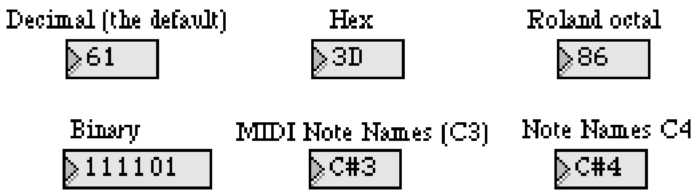
Display Options

The Inspector has check boxes for toggling on and off various features. Some of the options affect the way the **number box** functions, while others only affect the way it looks.

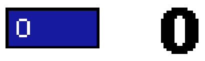
The *Draw Triangle* option is already checked, so that the triangle in the left edge of the **number box** will make it visually distinct from the **message** box. Also, the triangle shows when a **number box** has been clicked on, by becoming highlighted. The presence or absence of the triangle has no effect on the way the object functions, but it lets you change the appearance.

Draw in Bold displays the number in bold typeface. These aesthetic options can be used to emphasize certain **number box** objects, or to show the user of your program which ones to drag on.

The *Display* pop-up menu lets you select the format of the displayed data. (These options are available only in the int **number box**.)



Although we won't be using these options in the tutorial, the Inspector will also let you assign colors to both the numbers and the box they are in, or to make the box transparent. You can also choose fonts and font sizes for numbers from the Font menu.



Note: Numbers entered by typing into a **number box** must be typed in the same format as that in which the number is being displayed.

Mouse Options

Normally the **number box** sends out a continuous stream of numbers as it is being dragged upon with the mouse. The *Output only on Mouse-Up* option causes the **number box** to send out only the *last* number, the number that is showing when the mouse button is released. This lets you see the numbers as you drag, but only send out the single number that you choose.

When *Can't Change* is checked, numbers cannot be entered by dragging or typing. This is useful when you want a **number box** to be for display only, without being an onscreen controller.

The third patch shows some of these options in use. The patch is for converting decimal numbers to their hexadecimal or note name equivalents, or vice versa.

- Drag on the top **number box**, and you will see the numbers displayed in different formats.

Summary

The **number box** can be used to display numbers passing through it, and/or as an onscreen controller for sending out numbers. Numbers can be sent out by dragging on the **number box** with the mouse, or by clicking on the **number box** and then typing in a number (or pressing the up or down arrow keys).

The range of numbers a **number box** can send out can be specified by choosing **Get Info…** from the Object menu. With the Inspector you can also change how the numbers are displayed, and how the **number box** responds to the mouse.

See Also

number box	Display and output a number
Tutorial 3	About numbers

Tutorial 11: Test 2—Temperature Conversion

Using Arithmetic Operators

To be sure you understand how to use arithmetic operators and the **number box**, try this exercise:

1. Make a patch that converts a temperature expressed in degrees Fahrenheit into one expressed in degrees Celsius. Use a **number box** to enter the Fahrenheit temperature, send the number to arithmetic operator objects to convert it, and use another **number box** to display the result as a Celsius temperature.

Hints

The formula for converting Fahrenheit to Celsius is:

$$^{\circ}C = (^{\circ}F - 32) * 5/9$$

(The * is the multiplication operator.) You will first want to subtract 32 from the Fahrenheit temperature, then multiply the result by 5, then divide that result by 9.

Using Sliders

Here is a second exercise, a bit more difficult than the first one.

2. Make a patch that converts a temperature expressed in degrees Celsius into one expressed in degrees Fahrenheit. Limit the temperatures between the freezing point and the boiling point.

   In addition to using **number box** objects to show the temperatures, use **slider** objects as "thermometers" to show the temperatures graphically.

   Since the *Offset* and *Multiplier* features of the **slider** objects can do addition and multiplication, try using these features to do some of the arithmetic work. Use as few arithmetic operator objects as possible.

Hints

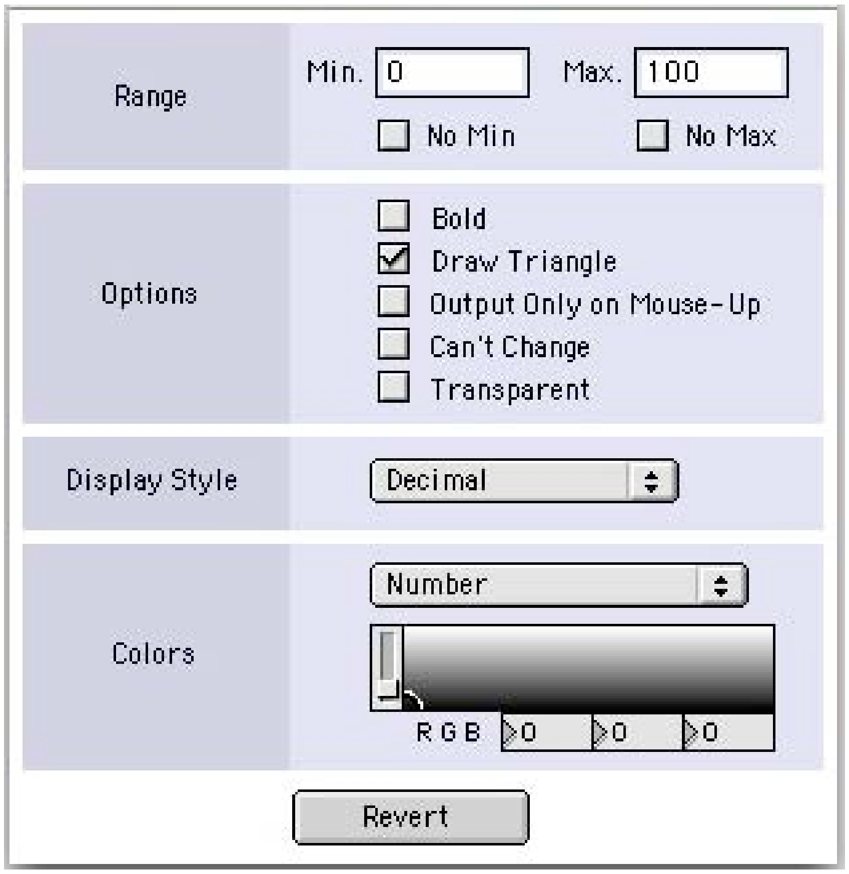
The formula for converting Celsius to Fahrenheit is:

$$˚F = (˚F * 9/5) +32$$

You will first want to multiply the Celsius temperature by 9, then divide the result by 5, then add 32 to that result.

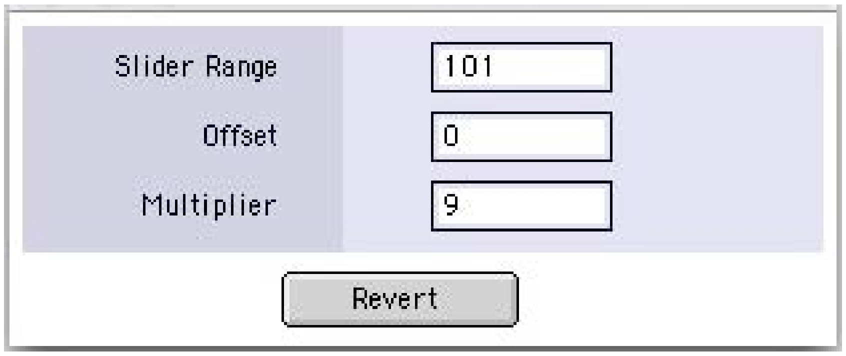
In degrees Celsius, 0 is the freezing point and 100 is the boiling point. In degrees Fahrenheit, 32 is the freezing point and 212 is the boiling point.

- Use the *Minimum* and *Maximum* features of the **number box** to limit the input (Celsius temperature) between 0 and 100.



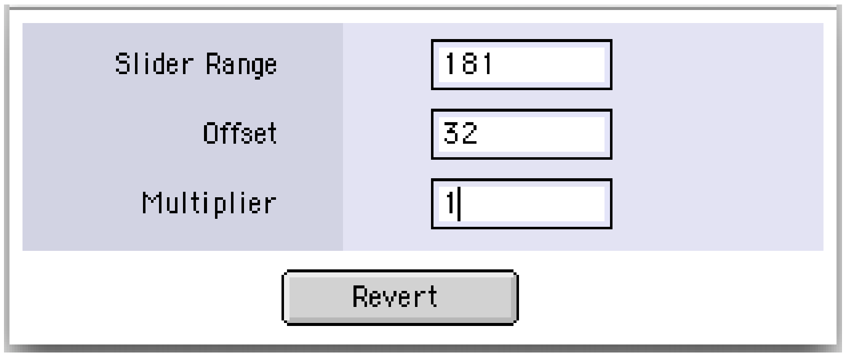
Range	Min. 0 Max. 100
	☐ No Min ☐ No Max
Options	☐ Bold
	✓ Draw Triangle
	☐ Output Only on Mouse-Up
	☐ Can't Change
	☐ Transparent
Display Style	Decimal
Colors	Number
	RGB 0 0 0

- Set the *Slider Range* of the **slider** which is depicting your Celsius "thermometer" to 101 so that it will display values from 0 to 100. (You can use the *Multiplier* feature of this **slider** to multiply the Celsius temperature by 9.)



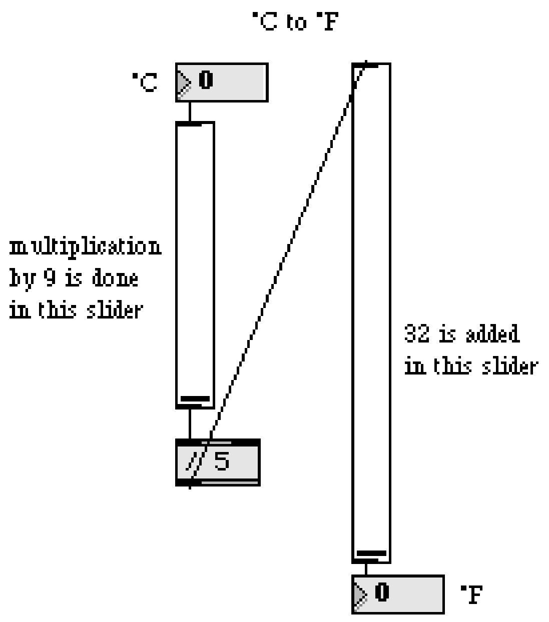
Slider Range	101
Offset	0
Multiplier	9

- Use a / object to divide the Celsius temperature by 5. Then use the *Offset* feature of the **slider** that's depicting your Fahrenheit "thermometer" to add 32, and you will have the result. (Set the *Slider Range* to 181 so that it will range from 32 to 212.



Slider Range	181
Offset	32
Multiplier	1

Your objects will be connected something like this:


°C to °F
°C
0
multiplication by 9 is done in this slider
// 5
32 is added in this slider
0
°F

Scroll the Patcher window to the right to see solutions to these two exercises. Although temperature conversion is not a very useful musical function, these exercises exemplify how to solve a mathematical problem using operator objects.

In the subsequent chapters you will use these operators to manipulate MIDI data.

Summary

Arithmetic operators can be linked together to form a complete mathematical expression. The order in which the objects are linked is important for performing each operation in the proper order.

In some instances, the *Offset* and *Multiplier* features of the **slider** object can be used to perform an arithmetic operation.

Tutorial 12: Using MIDI Data

Verify your MIDI Setup

Now that you've gotten a feel for how applications are constructed by connecting objects, we'll begin using MIDI data in our patches so that the examples have a more direct musical application.

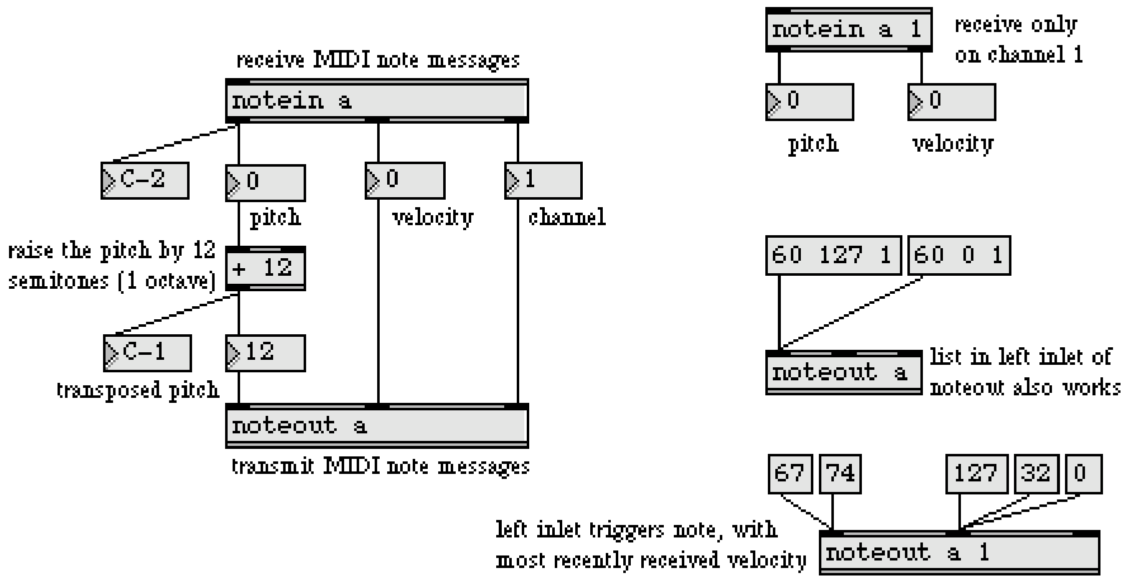
Make sure that your MIDI equipment is connected properly. If you have any doubts, review the section of the Fundamentals manual titled *Setup*, and review the first page of the Tutorial 1 for a discussion of MIDI equipment and connections.

MIDI Objects

There are many objects for transmitting and receiving data to and from your MIDI equipment. Objects that receive MIDI messages from your synth don't receive that data in through their inlet. Their MIDI input comes directly from the virtual MIDI ports (see the Ports chapter in the Fundamentals manual) rather than from other Max objects. Objects that transmit MIDI messages have no outlets, since they transmit their messages out from Max.

The most basic MIDI objects are **midiin** and **midiout**, which receive and transmit raw MIDI data byte-by-byte, without analyzing the MIDI messages at all. More commonly, though, you will use more specialized MIDI objects, which filter the raw MIDI data coming into Max, and output only the information you need.

For example, the **notein** object looks only for MIDI note messages, and when a note arrives, **notein** outputs the key number, the velocity, and the channel number. Similarly, the **bendin** object looks only for incoming pitch bend messages, and sends out the amount of pitch bend and the channel number.


receive MIDI note messages
notein a
C-2
0
pitch
0
velocity
1
channel
raise the pitch by 12 semitones (1 octave)
+ 12
C-1
12
transposed pitch
noteout a
transmit MIDI note messages
notein a 1
receive only on channel 1
0
pitch
0
velocity
60 127 1
60 0 1
noteout a
list in left inlet of noteout also works
67
74
127
32
0
left inlet triggers note, with most recently received velocity
noteout a 1

The notein and noteout Objects

For the moment we will concern ourselves only with MIDI note data—receiving information about notes played on the synth, and transmitting messages to play notes on the synth.

- Play a few notes on your synth. You should see the note data in the **number box** objects.

If you don't see anything happen, re-check your connections.

For the purpose of this Tutorial, your MIDI interface should be connected to your computer. If you have not already done so, you should use the **MIDI Setup** dialog to assign port a—with a channel offset of 0— to the input and output devices you want to use for the Tutorial.

The letter argument to **notein** indicates the port in which it receives MIDI note messages. If no argument is present, it receives from all ports.

- Play a few notes and hold the keys down for a moment before releasing them. You can see that a message is sent for each note both when you press the key and when you release it. A note message with a velocity of 0 indicates a note-off.

The patch on the left is for playing in parallel octaves. Every note you play on the synth is received by the **notein** object. The pitch information is sent to a + object which adds 12 to the pitch value. This, of course, raises the pitch by 12 semitones (1 octave). We've included extra **number box** objects so that you can see the pitch values both as numbers and as MIDI note names.

The velocity and channel information is passed on unchanged, and is reunited with the transposed pitch information in the **noteout** object. The new note, an octave higher than the one you played, and with the same velocity as you played, is sent back out to the synth by **noteout**.

Message Order

Although the pitch, velocity, and channel information appear to come out of **notein** at the same time, the numbers come out in right-to-left order (channel, then velocity, then pitch) just like any other object.

Like most objects, **noteout** is triggered by a message received in its left inlet—the pitch number. The pitch is combined with whatever velocity and channel values were most recently received in the other inlets, and a MIDI message is sent out to the synth.

This consistency of message ordering—outlets always send right-to-left, and objects are triggered by the left inlet—allows different objects such as **notein** and **noteout** to communicate easily. Because the velocity and channel numbers come out of **notein** before the pitch does, they arrive at **noteout** before the pitch does, keeping all the data properly synchronized.

Receiving On One Channel

A **notein** object with no arguments, or with only a letter argument, receives incoming note messages on *all* channels. (This is known as *omni mode* in MIDI terminology.) You can set **notein** to receive on only one channel by typing in a channel number argument. When there is a channel argument, **notein** has only two outlets—for pitch and velocity—because the channel number is already known. Both the port argument and the channel argument are optional.

- If your MIDI keyboard can transmit on different channels, set it to transmit on some channel other than 1. Now when you play notes the **notein** on the left still receives them, but the **notein** on the right ignores them.

Transmitting Note Messages

You don't necessarily need to play notes into Max to send notes out. You can transmit notes to the synth that are produced within Max.

One way to do this is to send a list—consisting of pitch, velocity, and channel—to the left inlet of **noteout**. You may remember this use of lists with the arithmetic operator objects in *Tutorial 8*.

- Click on the **message** boxes containing lists. One list sends a note-on, and the other sends a note-off (a note with a velocity of 0). It is necessary (or at least polite) to follow a note-on with a note-off, otherwise the note will continue to play. Try this with a sustained sound on your synth.

The last patch demonstrates that you can type in an argument for the channel on which **noteout** will transmit. The channel inlet is still present, however, and you can change the channel by sending in a new number.

The patch also shows that **noteout** combines pitches with whatever velocity was most recently received.

- Try sending different velocities to **noteout**. The velocity is just stored until a pitch number is received to trigger a MIDI note message.

Summary

The **notein** object looks for incoming MIDI note messages, and outputs pitch, velocity, and channel data. You can type in a specific port letter as an argument, which causes **notein** to output only the note data received in that one port. You can also type in a specific channel number, causing **notein** to output only the note data received on that one channel.

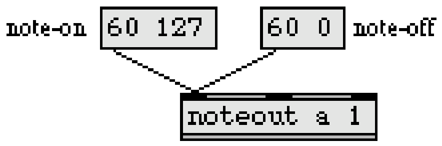
When the **noteout** object receives a number in its left inlet, it uses that number as a pitch value, combines the pitch with a velocity and a channel number, and transmits a MIDI note message. The pitch, velocity, and channel can also be received together as a list in the left inlet.

See Also

notein	Output incoming MIDI note messages
noteout	Transmit MIDI note messages

Note-On and Note-Off Messages

One of the main problems that you encounter when sending note messages to a synth from Max is the need to follow every note-on message with a corresponding note-off message. For example, just sending a pitch and a velocity to **noteout** plays a note on the synth, but that note will not be turned off until you also send the same pitch with a velocity of 0.

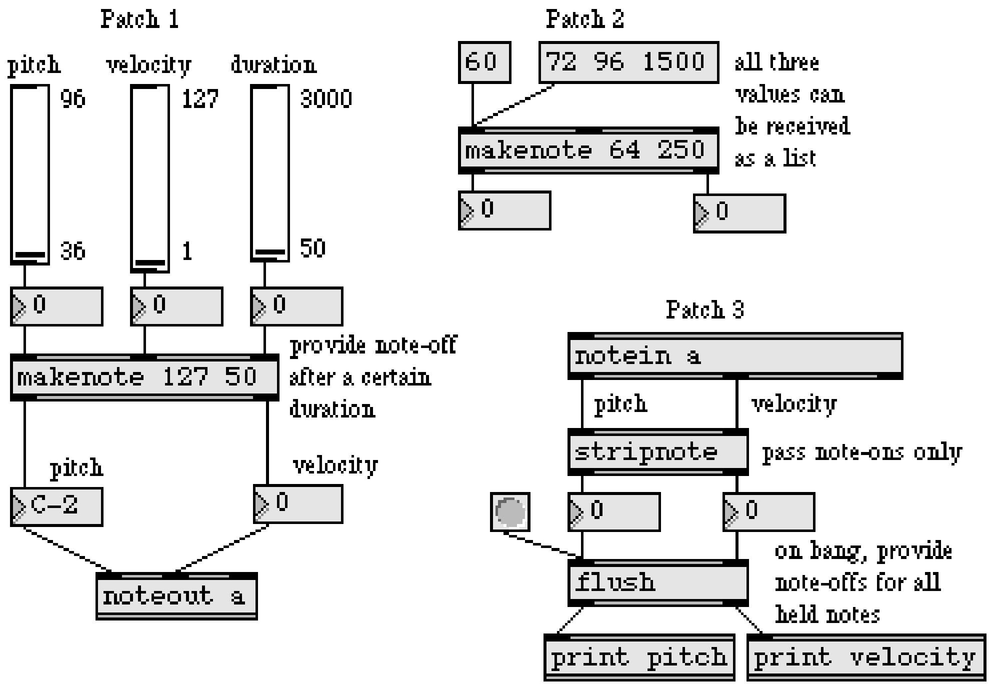


makenote

Max has objects that generate note-off messages, for turning off notes that have been sent to the synth. One such object is **makenote**.

When **makenote** receives a number in its left inlet, it uses the number as a pitch value, combines that pitch with a velocity, and sends the numbers out its two outlets. Then after a specified delay (or *duration*), **makenote** automatically sends the same pitch number, but with a velocity value of 0.

The synth interprets a note-on message with a velocity of 0 as a note-off. So, when the output of **makenote** is sent to **noteout**, both a note-on and a note-off get transmitted to the synth.



- Drag on the **slider** marked pitch in Patch 1. Each number that comes out of the **slider** is combined with a velocity by **makenote** (in this case, the velocity is 127, specified in the first argument), and the pitch and velocity are sent to **noteout**.

  50 milliseconds after each pitch is received (the duration specified in the second argument) **makenote** sends the same pitch out again, with a velocity of 0. The result is that every note has a duration of 50ms.

The velocity and the duration can be changed by numbers received in the middle and right inlets. The most recent values received in these inlets are used the next time a pitch is received in the left inlet.

- Try changing the velocity and duration by dragging on the **slider** objects, then play more notes by dragging on the pitch **slider**. The notes now have the velocity and duration you specified.

Note: When no channel number has been specified to **noteout**, either as a typed-in argument or in the right inlet, it is set to channel 1 by default.

Patch 2 demonstrates that the pitch, velocity, and duration values can all be received in the left inlet as a list.

- Click on the **message** box containing the number 60. You can see that it is combined with a velocity of 64, then combined with a velocity of 0 after 250 milliseconds.
- Click on the **message** box containing the list. The pitch 72 is sent out with a velocity of 96, and after 1.5 seconds it is sent out again with a velocity of 0.
- Now click again on the number 60. You can see that the velocity and duration values (96 and 1500) have been stored in **makenote**, and are applied to the pitch received in the left inlet.

stripnote

The **stripnote** object is sort of like **makenote** in reverse. It receives a pitch and a velocity in its inlet, and passes them on only if the velocity is not 0. In this way, it filters out note-off messages, and passes only note-on messages.

This is useful if you want to get data only when a key on your keyboard is pressed down, but not when the key is released. For example, you might want to use a pitch value from the keyboard to send a number to some object in Max, but you wouldn't want to receive the number both from the key being pressed *and* from the key being released.

flush

The **flush** object is another object for generating note-off messages. Unlike **makenote**, however, it does not generate them automatically after a certain duration. Instead, **flush** keeps track of the notes that have passed through it. When it receives a bang in its left inlet it provides note-offs for any notes that have not yet been turned off.

Both **flush** and **stripnote** receive velocity values in the right inlet and pitch values in the left inlet, and pass the same type of values out the outlets. They are triggered by a pitch value received in the left inlet, and use the velocity value that was most recently received in the right inlet. Both objects can also receive the pitch and velocity values *together* as a list in the left inlet.

- Play a few notes on your MIDI keyboard. You can see that **stripnote** passes only the note-on messages and suppresses the note-offs. The note-ons get passed through **flush**, and are received by the **print** objects. (A **flush** object will also pass on any note-offs it receives, but in this case **stripnote** has filtered them out.)

- Now click on the **button** to send a bang to the left inlet of **flush**. The **flush** object keeps track of all the note-ons it has received that have not been followed by note-offs, and when a bang is received, **flush** provides note-offs for those held notes.

The advantage of sending pitch and velocity pairs through **flush** before sending them to **noteout** is that **flush** has no noticeable effect until it receives a bang, then it turns off any notes that are still on. This is useful for turning off stuck notes.

Summary

A MIDI note-on message transmitted by **noteout** should be followed by a corresponding note-off message, so that the note played by the synthesizer gets turned off.

The **makenote** object combines pitch values with velocity values, to be sent to **noteout**. After a certain duration, the same pitch is sent with a velocity of 0, to turn off the note. The **stripnote** object is the opposite of **makenote**. It filters out note-off messages (pitch-velocity pairs in which the velocity is 0), and passes on only note-on messages (messages with a non-zero velocity).

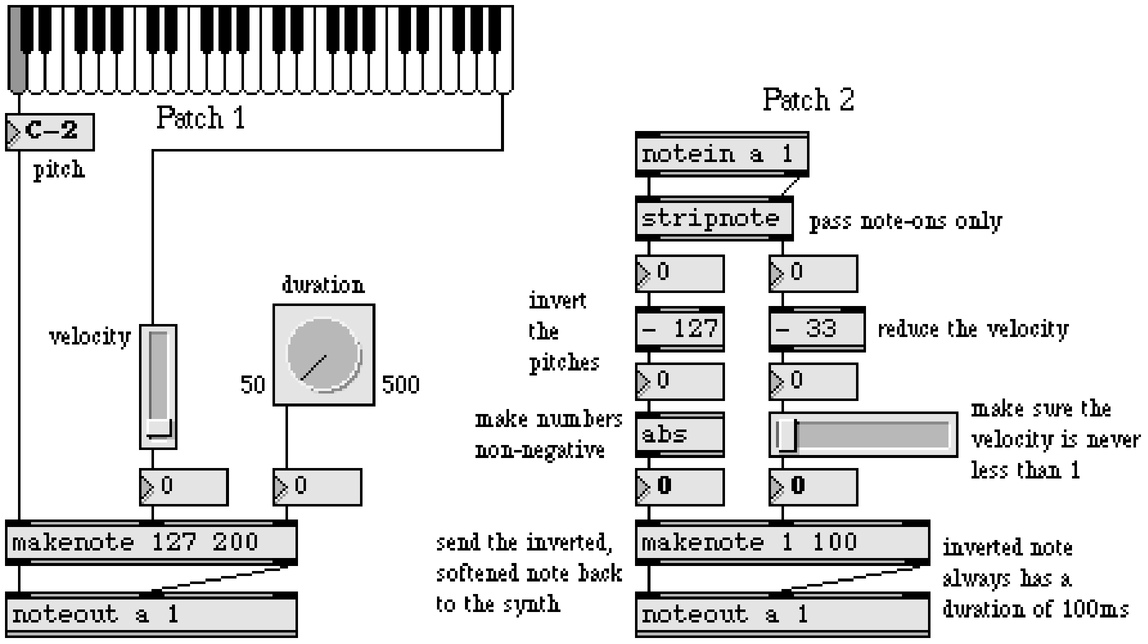
The **flush** object keeps track of the notes that have passed through it, and when it receives a bang it sends out a note-off for any notes which are still on.

See Also

flush	Provide note-offs for held notes
makenote	Generate a note-off message following each note-on
poly	Allocate notes to different voices
stripnote	Filter out note-off messages, pass only note-on messages
sustain	Hold note-off messages, output them on command

Diverse Onscreen Controllers

In this tutorial, we'll introduce some objects that function similarly to the **slider**, but differ somewhat in appearance and behavior.



kslider

Patch No. 1 is similar to the patch in the previous chapter. It allows you to play notes with the mouse. However, this patch uses a keyboard slider, **kslider**.

- Try playing notes by clicking and/or dragging on the **kslider**. It has been set to output numbers from 36 to 96 (MIDI notes C1 to C6) out its left outlet. The numbers are then sent to the left inlet of **makenote**, where they are paired with a velocity (from the right outlet of **kslider**), and the notes are sent to **noteout**.
- When you drag along the lower half of **kslider**, it outputs only the numbers associated with the white keys. When you drag along the upper half, it plays both white and black keys.
- The velocity that is sent out the right outlet depends on how high the mouse is placed on the key you are playing.

The *Range* and *Offset* of the notes displayed by **kslider** can be changed by choosing **Get Info…** from the Object menu. The *Offset* is the value that will be output by clicking on the lowest note of the **kslider**, and is specified as a MIDI note name. The default is `C1` (`36`). If you want an offset of `0`, set it to `C-2`. The *Range* is specified as the number of octaves you want the **kslider** to have. The Inspector also lets you select one of two sizes for **kslider**, *Large* or *Small.*

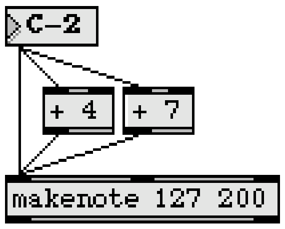
Range and *Offset* refer only to the numbers displayed by **kslider**, or sent out its outlet by clicking and dragging with the mouse. Numbers received in the inlet of **kslider** are unaffected by the *Offset*, and are passed through unchanged.

Playing Parallel Chords

Suppose you wanted **kslider** to play parallel major triads. How would you go about it?

In addition to sending the numbers directly to the left inlet of **makenote**, you can also send them to two different **+** objects. One **+** object can add 4 to the number (raising the pitch a major third), and the other can add 7 (raising the pitch a perfect fifth). These transposed pitches are then sent to **makenote**, along with the original pitch.

- Try it yourself. Unlock the Patcher window and create two new **+** objects just above **makenote**. Then connect the outlet of the **number box** to the inlets of the **+** objects, and connect the outlets of the **+** objects to the left inlet of **makenote**.

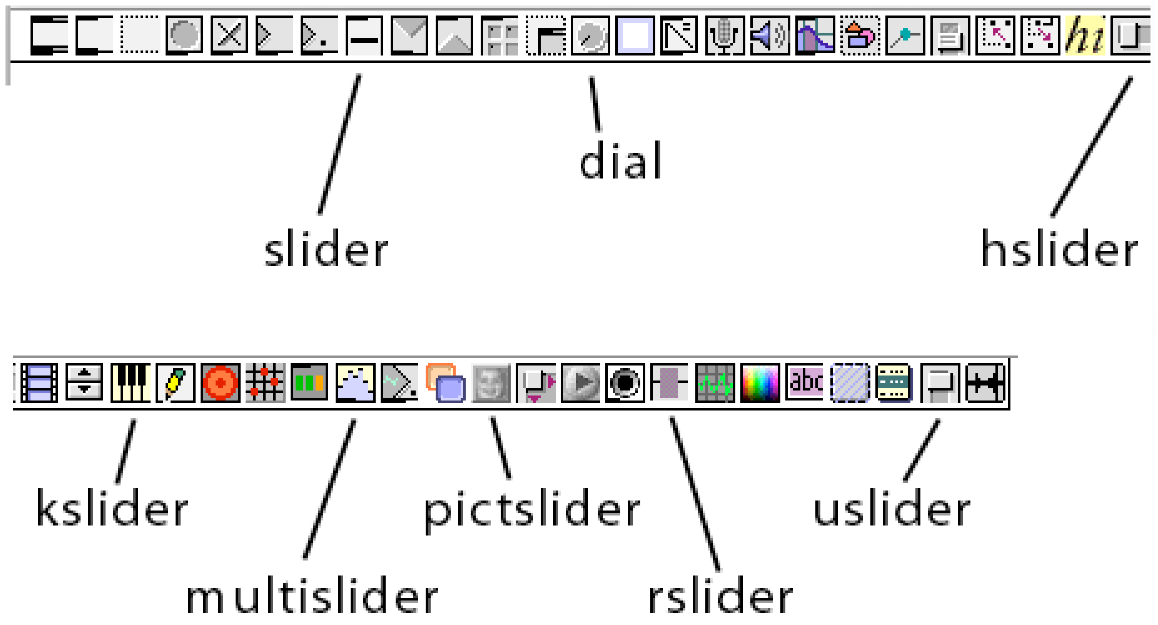


- Now lock the Patcher window and click on a note of the **kslider** to hear the results. You can also try changing the numbers you add with the **+** objects, to create other types of triads.

Tutorial 14

dial, hslider, and uslider

In Patch 1 the velocity values are displayed by a slider object named **uslider**, and the durations are supplied by a **dial**. Patch 2 contains the horizontal slider, **hslider**.



Various sliders and dials in the palette

There are a few important differences between these objects and the **slider** and **number box** objects seen in previous chapters.

1. The **slider** and the **number box** send out numbers when you drag them with the mouse. You can drag on the other sliders and dials, but you can also change them with a single mouse click.

- Click on the sliders and dials, and notice how they jump to the new position and send out a number, even without dragging the mouse.

2. The **slider** and **kslider** objects resize themselves automatically depending on their range. The **hslider** and **uslider** can be shrunk or enlarged to virtually any size with the grow bar, regardless of the range of numbers they send out. The **dial** has only one possible size, regardless of its range.

3. Although the **slider** and **kslider** may have a limited range of numbers that can be output by dragging, they do not limit the range of numbers that can pass *through* them. The

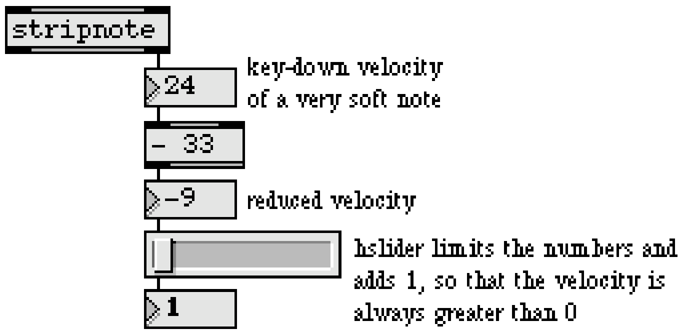
dial, **hslider**, **number box**, and **uslider** *do* limit the numbers received in their inlets. Any incoming number that is less than 0 (or the specified minimum, in the case of **number box**), or that exceeds the specified range, will be automatically restricted within those limits.

The limiting feature can be put to use, as is shown in Patch 2. Let's analyze what the patch does.

Analyzing Patch 2

- Play a scale on your MIDI keyboard. Notice that as you play you also hear a scale of short notes moving in the opposite direction.

When you play notes on the synth, the pitch and velocity are sent through **stripnote**, which filters out all the note-off messages, passing only the note-ons. Then 33 is subtracted from the velocity.


stripnote
24
key-down velocity of a very soft note
- 33
-9
reduced velocity
hslider limits the numbers and adds 1, so that the velocity is always greater than 0
1

- Play some notes very, very softly so that your key-down velocity is less than 33.

This results in negative numbers coming out of the - (minus) object. The **hslider** limits the numbers it receives in its inlet, so that none of them is less than 0, and the **hslider** object's *Offset* of 1 ensures that all velocities are at least 1. The reduced velocity finally arrives in the middle inlet of **makenote** and is stored there.

Next, the pitch value comes out of **stripnote**, and has 127 subtracted from it. This means that pitches, which usually range from 0 up to 127, will range from -127 up to 0. If you have a 61-note keyboard, your pitches range from 36 up to 96, and subtracting 127 from them causes them to range from -91 up to -31.

This number is then sent to an **abs** object, which sends out the absolute (non-negative) value of whatever number it receives. So now, instead of your pitches ranging from -91 up to -31, they range from 91 *down* to 31. As you play higher on the keyboard, the numbers being sent to **makenote** become lower, and vice versa.

The inverted pitches are paired with the reduced velocity in **makenote**, and the notes are sent out, then are turned off after 100 milliseconds ($^1/_{10}$ of a second).

Summary

The **hslider** and **uslider** objects are similar to **slider**, but can be made any size. **kslider** is a keyboard- like slider, the *Range* of which is specified as a number of octaves. You can perform both chromatic glissandi and diatonic glissandi (white-keys only) on **kslider**.

The **dial**, **hslider**, and **uslider** objects all limit the numbers they receive in their inlet. Numbers that exceed the range of these sliders are set to the minimum or maximum value of the slider. Unlike **slider**, these other sliders respond to a single mouse click, without dragging.

The **abs** object sends out the absolute value of whatever number it receives in its inlet. The limiting sliders and **abs** represent two different ways to avoid negative numbers. (Other objects that can serve this purpose are **maximum** and **split**.)

See Also

abs	Output the absolute value of the input
dial	Output numbers by moving a dial onscreen
hslider	Output numbers by moving a slider onscreen
kslider	Output numbers from a keyboard onscreen
split	Look for a range of numbers
uslider	Output numbers by moving a slider onscreen

Tutorial 15: Making Decisions with Comparisons

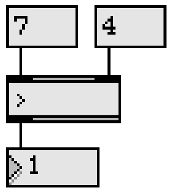
Relational Operators

One of the most basic things a computer program does is perform some kind of a test, then make a decision based on the result of that test. The test is usually some kind of *comparison*, such as seeing if two numbers are equal. The answer to this test can be used to determine what the computer does next.

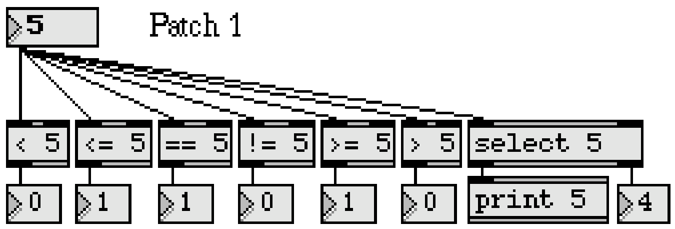
Numbers are compared using *relational operators* which characterize the relationship of one number to another with such terms as *is less than*, *is greater than*, is *equal to*, etc. Max has several relational operator objects, for comparing one number to another:

< means is less than	**<=** means is less than or equal to	**==** means is equal to
> means is greater than	**>=** means is greater than or equal to	**!=** means is not equal to

Max's relational operator objects send out the number 1 if the statement is true, and 0 if the statement is false. So, for example, to test the statement *7 is greater than 4*, you would send the number 4 to the right inlet of a > object, then trigger the object by sending the number 7 in the left inlet. Since the statement *7 is greater than 4* is true, the objects sends out the number 1.



The right operand can also be provided as an argument typed into the object box.



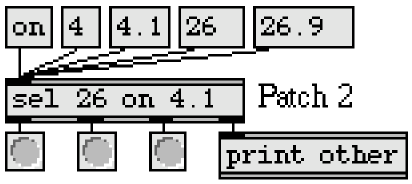
- Drag on the **number box** at the top of Patch 1. Notice especially the output of each object as you pass by the number 5.

The relational operators normally expect to receive ints in the inlets. Floats are converted to int before the comparison is made. Like the arithmetic operators, however, the relational operators can compare floats if there is a float argument typed in.

select

The **select** object is a special relational operator. If the left operand is equal to the right operand, a bang is sent out the left outlet. Otherwise, the left operand is passed out the right outlet. The effect is that every number received in the left inlet gets passed on out the right outlet except the one **select** is looking for. When **select** receives the number it's looking for, it sends a bang out the left outlet.

Patch 2 shows that the **select** object (whose name can be shortened to **sel**) can actually be given *several* arguments, and each argument can be an int, a float or a *symbol* (i.e., a word). The input is converted to the proper type (int, float, or symbol) before being compared to each argument. Notice that the right inlet is not present if there is more than one argument.



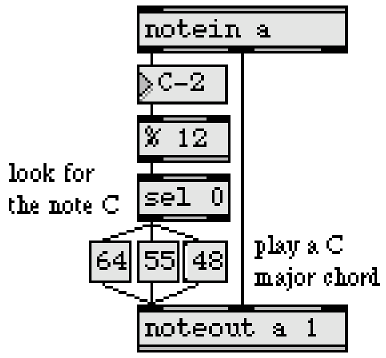
- Click on the different messages. Notice that if the input matches one of the arguments, a bang is sent out the outlet that corresponds to that argument. If there is no match, the input is passed out the right outlet.

When the input is an int (such as 4) it is converted to float before being compared with a float argument. A float input (such as 26.9) is truncated before being compared to an int argument.

Combining Comparisons with the select object

The **select** object sends out a bang, which can be used to trigger other objects, and relational operators send out the numbers 1 and 0, which can be used to toggle something on and off (such as a **metro**). So you can see that comparisons can be used in a patch to decide when to trigger another object.

Patch 3 shows the use of **sel** to look for a certain pitch being played on your MIDI keyboard.



The pitch is first sent to an **%** object, which divides it by 12 and sends out the remainder. Since the note C always has a MIDI key number which is a multiple of 12 (such as 36, 48, 60, etc.), the output of the **% 12** object will be 0 whenever the note C is played.

Each time the **sel** object receives the number 0 from **%**, it sends a bang to the **message** boxes, which send the notes C2, G2, and E3 (48, 55, and 64) to **noteout**. These pitches are combined with the velocity of the note C that is being played on the synth, so the chord has the same velocity and duration as the note being played.

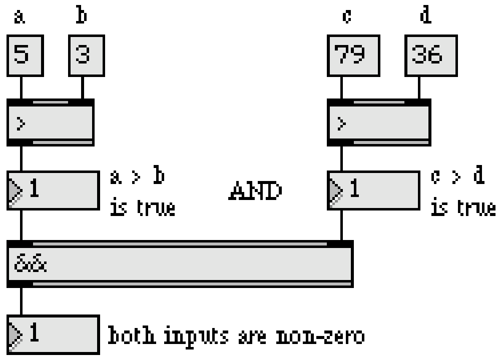
In this example, we test to see if the pitch being played is equal to C. When this is true, the chord is triggered.

Combining Comparisons with "Or" or "And"

The object **||** means *or*. If either the left operator or the right operator is non-zero (*true*), **||** sends out the number 1. If both operators are 0, it sends out 0.

The object **&&** means *and*. If the left operator *and* the right operator are both non-zero, **&&** sends out the number 1. Otherwise, it sends out 0.

|| and **&&** are used to combine two comparisons into a single statement, such as: *a is greater than b AND c is greater than d*.



Note that in the example above, the number 5 (*a*) must be sent *last*, so that all the other values will have arrived when **&&** is triggered.

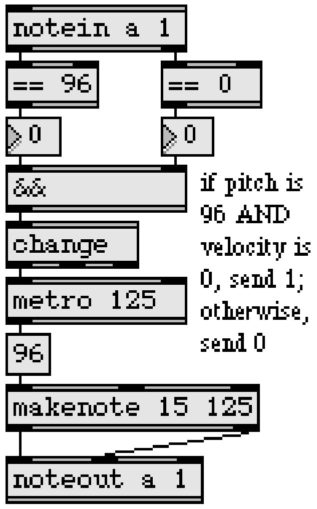
Patch 4 is similar to Patch 3, but it uses **||** to look for two pitches instead of one. The patch says, *if the pitch played is B OR it is D, then play the notes G1 and F3*. The effect, of course, is to accompany the notes B and D with an incomplete dominant seventh chord.

- Play a melody in the key of C on your synth. Patches 3 and 4 provide you with an annoyingly Haydnesque accompaniment.

Using Comparisons to Toggle an Object On and Off

Patches 3 and 4 demonstrate that any number—and thus any key or combination of keys on the synth—can be used to trigger something in Max. Similarly, the 1 and 0 sent out by relational operator objects can be used to turn an object such as **metro** on and off.

Patch 5 demonstrates this idea.



- Play the note C6 (high C) on the synth. As soon as you release the key, Patch 5 begins playing the note repeatedly until the next time you play a note.

The patch is looking for the condition when the pitch is equal to 96 and the velocity is equal to 0. When both conditions are true, **&&** sends out 1, otherwise **&&** sends out 0. Obviously, the vast majority of note messages will cause **&&** to send out 0. In order to avoid sending the number 0 to **metro** over and over unnecessarily, the output of **&&** is first sent to a **change** object. The purpose of **change** is to filter out repetitions of a number. The number received in the inlet is sent out the left outlet only if it is different from the preceding number.

When **metro** is turned on it sends the number 96 to **makenote** at the rate of 8 notes per second (once every 125 ms).

Summary

Relational operators—**<**, **<=**, **==**, **!=**, **>=**, and **>**—compare two numbers, and report the result of the comparison by outputting 1 or 0. The **&&** and **||** objects test whether their inputs are 0 or non- zero, making them useful for combining two comparisons into a single test.

The **select** object (also known as **sel**) looks for certain numbers (or *symbols*). If the input matches what it is looking for, it sends a bang out one of its outlets. Otherwise, it passes the input out its right outlet.

The results of any of these comparisons can be used by the program to *make a decisio*n whether to trigger other objects.

The **change** objects passes on a number received in its inlet, only if the number is different from the preceding one.

See Also

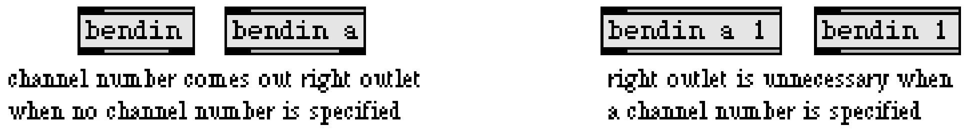
change	Filter out repetitions of a number
select	Select certain inputs, pass the rest on
<	*Is less than*, comparison of two numbers
<=	*Is less than or equal to*, comparison of two numbers
==	Compare two numbers, output 1 if they are equal
!=	Compare two numbers, output 1 if they are not equal
>=	*Is greater than or equal to*, comparison of two numbers
>	*Is greater than*, comparison of two numbers

Tutorial 16: More MIDI Ins and Outs

Introduction

There are many MIDI objects besides **notein** and **noteout**. Objects exist for receiving and transmitting any kind of MIDI message. In this chapter, we introduce a few of these objects: **bendin** and **bendout** for pitchbend messages, **pgmin** and **pgmout** for program change messages, and **ctlin** and **ctlout** for continuous controller messages.

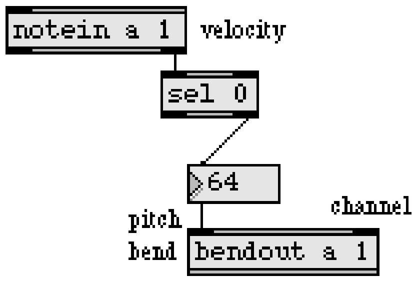
Like **notein** and **noteout**, these other objects can be given optional arguments to specify the port and MIDI channel on which they will operate. When a channel number is specified as an argument in a MIDI receiving object, the outlet for sending the channel number disappears.


bendin
bendin a
channel number comes out right outlet when no channel number is specified
bendin a 1
bendin 1
right outlet is unnecessary when a channel number is specified

bendin and bendout

bendin receives data from the pitch bend wheel of your MIDI keyboard. The channel is sent out the right outlet, and the pitch bend data (the amount of pitch bend) is sent out the left outlet. Pitch bend data ranges from 0 to 127, with 64 meaning no bend at all.

The first patch demonstrates how easily one kind of MIDI data can be given a different meaning. In this case, the velocity of the notes played on the synth is sent to **bendout** to control the pitch bend.



- Play a single note repeatedly on the synth. The pitch is bent upward when you play hard (when the velocity is greater than 64), and is bent downward when you play softly.

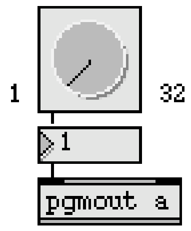
Notice the way that **sel** is used to filter out note-off velocities. If this were not done, the pitch would be bent down to 0 each time a key is released, which might be bothersome in some cases. (On the other hand, triggering pitch bends with note-offs could be an interesting effect.)

pgmin and pgmout

MIDI program change messages change the sound a synthesizer uses to play notes. Almost all synths can receive program change messages, and many can send them as well.

Different synths have different numbers of possible sounds, and have different ways of numbering their sounds. Some synths start numbering sounds from 0, while others start from 1. Others use unique numbering systems, such as a letter-number combination simulating base-8 arithmetic, etc.

- Use the **dial** in the second patch to send program change messages to the synth.

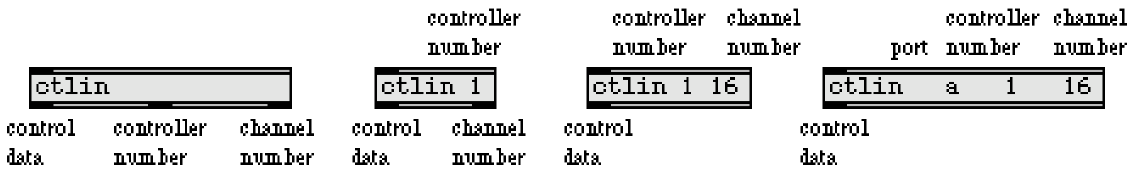


- The **dial** sends out numbers from 1 to 32. If this is not appropriate for your synth, you can change the *Dial Range* and *Offset*. Unlock the Patcher window, select the **dial**, and choose **Get Info…** from the Object menu.

ctlin and ctlout

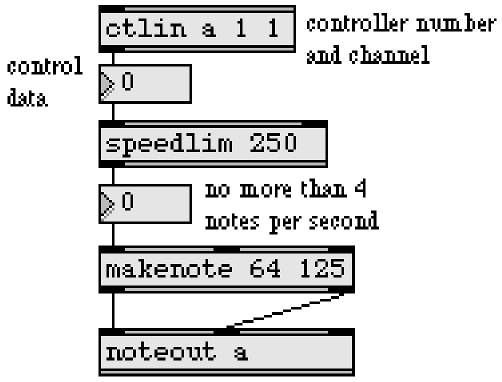
A control change message contains three vital items of information: the channel, the controller number, and the control data. The meaning of the data is dependent on the controller number. For example, controller number 1 is usually assigned to the modulation wheel, controller 7 to volume, etc.

Therefore, in addition to port and channel arguments, **ctlin** and **ctlout** can be given a specific controller number as an argument, immediately after the port argument (if present). When a specific controller number is given as an argument to **ctlin**, the controller number outlet disappears. For more about the arguments and their default values, look under **ctlin** and **ctlout** in the Max Reference Manual.



Reassigning Control Data

You can use a continuous MIDI controller to send a stream of numbers to Max, then use those numbers in any way you like. In this patch, the data from the mod wheel of the synth is used to send pitch values back to the synth.



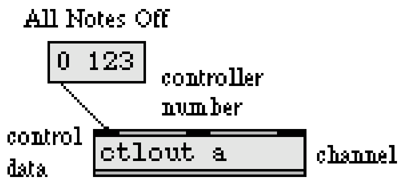
- Move the modulation wheel on your MIDI keyboard, and you should hear notes play.

The **speedlim** object limits the speed with which numbers can pass through it. When **speedlim** receives a number, it sends the number out the outlet, then waits a certain number of milliseconds before it will receive another number. The number of milliseconds between numbers can be a typed-in argument and/or supplied in the right inlet.

Channel Mode Messages

Controller numbers 122 to 127 are reserved for special MIDI commands known as *channel mode messages*. Channel mode messages can be received and transmitted with **ctlin** and **ctlout**, just like any other control message.

The last patch shows an example of **ctlout** used to transmit a channel mode message meaning *All Notes Off* (controller number 123 with a value of 0). Many synths (but not all) recognize this message and turn off all notes currently being played. For turning off notes within Max, it's more reliable to use an object such as **flush** or **poly**.



The patch also demonstrates that **ctlout** (and the other transmitting objects) can receive values for all inlets in the form of a list in the left inlet. When there are no arguments, **ctlout** transmits on channel 1 out port a.

Summary

Pitch bend messages are received and transmitted with **bendin** and **bendout**, program changes with **pgmin** and **pgmout**, and continuous control messages (and *channel mode* messages) with **ctlin** and **ctlout**.

MIDI data can be altered and reassigned in any way within Max.

A stream of numbers can be "slowed down" by filtering them with **speedlim**, which ignores some of the numbers if they arrive too fast. This is a good method of converting a continuous stream of numbers into regular, discrete steps.

See Also

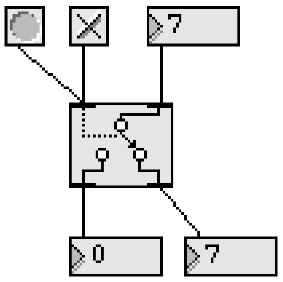
bendin	Output incoming MIDI pitch bend values
bendout	Transmit MIDI pitch bend messages
ctlin	Output incoming MIDI control values
ctlout	Transmit MIDI control messages
pgmin	Output incoming MIDI program change values
pgmout	Transmit MIDI program change messages

speedlim	Limit the speed with which numbers can pass through

Ggate

In Tutorial 15 we saw examples of how to use comparisons to make a decision whether to send a message. It is also possible for objects to make decisions about where to send a message.

The patch in the upper-left corner shows a graphical object, **Ggate**, for routing incoming messages out one outlet or the other. Messages received in the right inlet are sent out whichever outlet is pointed at by the arrow. The direction of the arrow can be changed by clicking on **Ggate** with the mouse, sending a bang to the left inlet, or sending a zero or non-zero number to the left inlet.

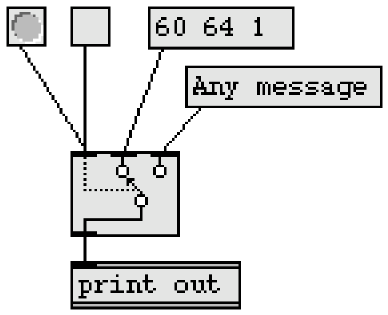


When the number in the left inlet is 0, the arrow points to the left outlet. A non-zero number in the left inlet causes the arrow to point to the right outlet. In the example above, the **toggle** has sent the number 1 to the left inlet of **Ggate**, causing the arrow to point to the right outlet. Consequently, any message received in the right inlet is passed out the right outlet.

- Try the various methods of changing the direction of the arrow, then drag on the **number box** to send numbers to the right inlet of **Ggate**.

Gswitch

The second patch shows a similar object, **Gswitch**, which can open one of two inlets. Whichever inlet the arrow points to is the *open* inlet, and messages received in that inlet are passed out the outlet. Messages received in the closed inlet are ignored. The leftmost inlet is the *control inlet*, for switching the arrow back and forth. It functions like the left inlet of **Ggate**.



- Click on the two **message** boxes to send messages to **Gswitch**. Only the message received in the open inlet is sent out the outlet. Change the direction of the arrow and try again.

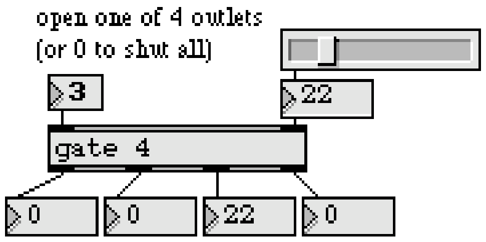
Ggate and **Gswitch** will pass on any type of message—numbers, lists, and text.

gate

The **gate** object is like **Ggate**, with a few important differences:

1. The number of outlets is determined by the argument to **gate**. A single outlet is opened when its number is received in the left inlet. All other outlets are closed.
2. When the number 0 is received in the left inlet, *all* outlets are closed.
3. A **gate** does not respond to a mouse click the way **Ggate** does.

- Send a number to the left inlet of **gate** to open one of the outlets (or 0 to close all outlets), then send numbers to the right inlet.

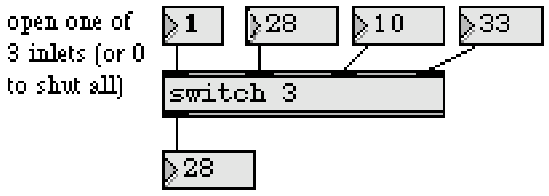


You can see that **gate** can be used to route messages to any of a number of destinations, just by specifying which outlet the messages are to be sent out.

switch

The **switch** object opens one of several inlets to receive messages, and ignores messages received in the other inlets. The leftmost inlet is the control inlet, as with **gate**, and the remaining inlets can receive any message to be sent out the outlet if the inlet is open. The number of inlets—in addition to the control inlet—is specified by the argument. (There can be as many as 9.)

- Send a number to the leftmost inlet of **switch** to open one of the inlets (or 0 to close all inlets). Then send numbers to the other inlets, and you will see that only the one inlet you specified is open.



Using **switch**, you can have messages coming into individual inlets from several different objects, but **switch** "listens to" only one of its inlets.

Any type of message can be passed through **gate** or **switch**.

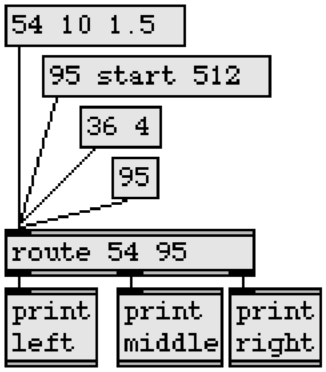
Left Inlet Is Control Inlet

Unlike most objects—which are triggered by a message received in the left inlet—**gate**, **switch**, **Ggate**, and **Gswitch** all use the left inlet as a *control inlet*, for telling which inlet or outlet is to be open. Messages received in the other inlet(s) are then sent out the appropriate outlet.

route

One other valuable traffic controller is **route**, sort of a cross between **sel** and **gate**. When **route** receives a message in its inlet, it compares the first item in the message to each of its arguments. If it finds a match, it sends *the rest* of the message out the corresponding outlet. If there is no match, the entire message is sent out the rightmost outlet. **route** is especially useful when it is sent a list (several items separated by spaces).

- Click on the different **message** boxes and observe what is printed in the Max window.



The number of outlets **route** has depends on how many arguments are typed in. Each argument is an identifier for an outlet, and an additional outlet on the right sends out any messages not matched by **route**.

Like **select**, **route** looks only for certain inputs, and if there is no match it passes the message on unchanged. One important difference between **select** and **route** is that when **select** receives a match for one of its arguments, it sends a bang out the corresponding outlet. When **route** receives a match, it sends out *the rest of the message*. (Unless there is no rest of the message, in which case it sends out bang.)

Summary

The **gate** and **Ggate** objects receive messages in their right inlet, and send the messages out only one of their outlets, depending on which outlet has been specified as *open*. The **switch** and **Gswitch** objects receive messages in only one of their inlets, depending on which inlet has been specified as open.

Tutorial 17

The **route** object tries to match the first item of each incoming message to one of its arguments. If a match is found, **route** sends *the rest* of the message out the appropriate outlet (if there is no “rest of message”, **route** sends a bang to the appropriate outlet). If there is no match, the entire message is sent out the rightmost outlet.

Routing objects can be used to filter messages coming from many different objects—passing on only those messages which arrive in a particular inlet—or they can be used to send incoming messages to one of many destinations.

See Also

gate	Pass the input out a specific outlet
Ggate	Pass the input out one of two outlets
Gswitch	Receive the input in one of two inlets
route	Selectively pass the input out a specific outlet
switch	Receive the input in a specific inlet

Tutorial 18: Test 3—Comparisons and Decisions

Comparisons and Decisions

You can write a patch that examines the notes being played on your MIDI keyboard, and makes a decision about what to do, based on the note information. Here is an exercise to test your understanding of the use of comparisons to make decisions.

- Make a patch that receives the notes being played on your MIDI keyboard and, if the note is Middle C (MIDI note 60) or higher, plays the notes an octave lower and an octave higher than the played pitch.

Hints

A comparison is needed to find out if the pitch being played *is greater than or equal to* 60.

Based on that comparison, a decision is made to play or not to play the notes an octave lower (12 semitones less) and an octave higher (12 semitones more) than the original pitch.

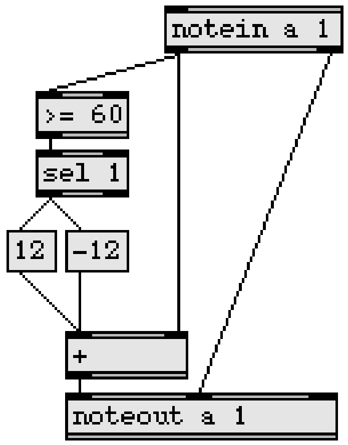
If the original pitch is less than 60, nothing needs to be done, but if the pitch is greater than or equal to 60, two new pitches need to be generated. The original pitch must have 12 subtracted from it to get the lower pitch, and it must have 12 added to it to get the higher pitch.

The new pitches must be combined with a velocity (perhaps the velocity of the original played note) to send note-on and note-off messages to the synth.

- Scroll the Patcher window to the right to see two possible solutions to the problem.

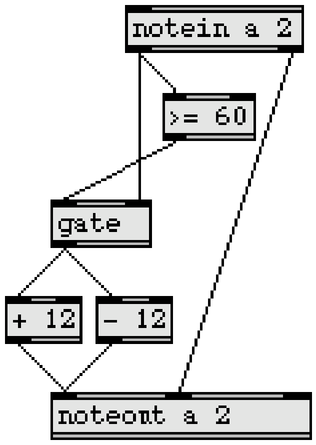
Solution 1

Both solutions use the relational operator **>=** to find out if the played pitch is greater than or equal to 60. Each solution uses the result of that comparison in slightly different ways.



In Solution 1, a successful comparison results in a decision to trigger the numbers -12 and 12. The pitch is stored in the right inlet of a **+** object before the comparison is made. Then, if the pitch is greater than or equal to 60, the **>=** object sends out 1, causing the **sel** object to trigger the messages -12 and 12. These two numbers are added to the original pitch, and the sums are sent to **noteout**, where they are combined with the velocity of the played note, triggering note messages to the synth. This process occurs both for note-ons and note-offs, so the transposed pitches are successfully played and turned off.

In Solution 2, a successful comparison opens a **gate**, letting the pitch through, so that only pitches greater than or equal to 60 will be passed on.



The placement of the **>=** object is of utmost importance. Because it is to the *right* of the **gate** object, it receives the pitch first, and either opens or shuts the **gate** before the pitch arrives at the **gate**.

When a pitch is let through, it has 12 subtracted from it in one object, and has 12 added to it in another object. The results are sent to **noteout** as pitches, where they are combined with the original played velocity (both for note-ons and note-offs).

Note: The **notein** and **noteout** objects in Solution 2 are set to receive and transmit on MIDI channel 2 only so that a note from your MIDI keyboard won't cause *both* patches to react.

Summary

In this exercise, the comparison **>=** 60 was used to trigger messages in one case, and to open a **gate** in another case. Either method could be incorporated in a solution to the problem.

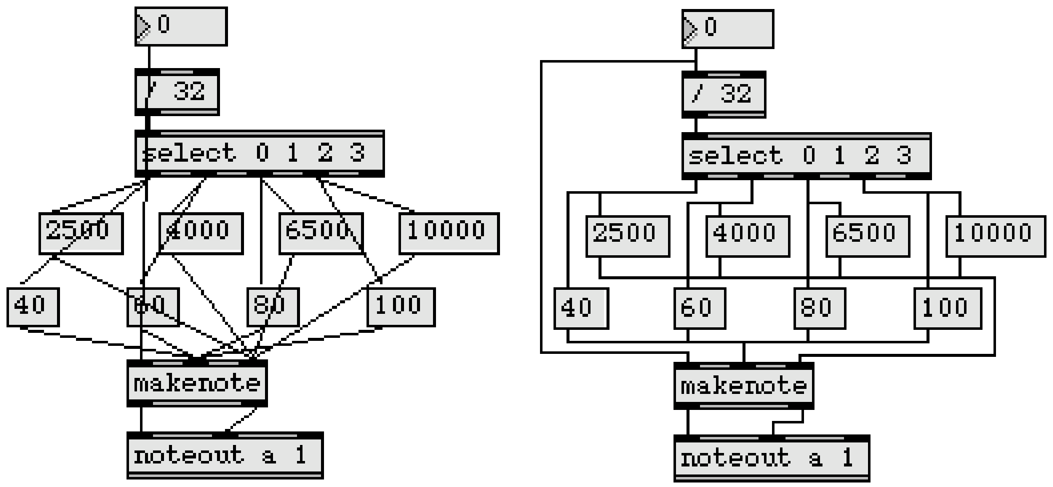
To perform a test and make a decision, ask yourself these questions:

1. What do I need to know to make the decision? (What will be tested?)
2. What action will be taken if the test is successful (true)?
3. What action will be taken if the test is unsuccessful (false)?

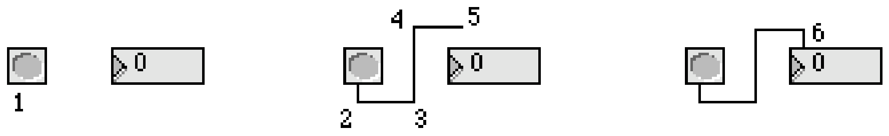
Segmented Patch Cords

So far we have been making patch cord connections with straight patch cords, by dragging from the outlet of one object to the inlet of another object. In complicated patches, though, it would be nice if we could bend patch cords around objects, to keep things from getting too messy.

When **Segmented Patch Cords** is checked in the Options menu, patch cords can be made up of as many as 10 line segments, allowing them to bend around objects. Segmented patch cords function no differently from straight patch cords, and they can often make a patch neater and more comprehensible.

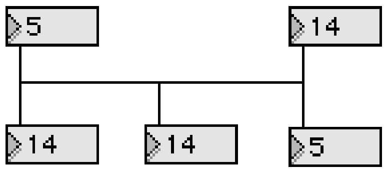


The method of connecting objects is a little different when the **Segmented Patch Cords** option is in effect. Instead of dragging from outlet to inlet, the method is to click on the outlet, then click at each of the points where you want the patch cord to bend, then click on the inlet of the receiving object.



The patch in the left part of the Patcher window shows how segmented patch cords can be used to give the patch a neater look, making it easier to understand how the patch functions.

Of course, sometimes segmented patch cords can make a patch less clear to the eye. In the following example, the user cannot be certain where the top-left **number box** is connected, without actually changing the number to see which of the lower three values changes, too:



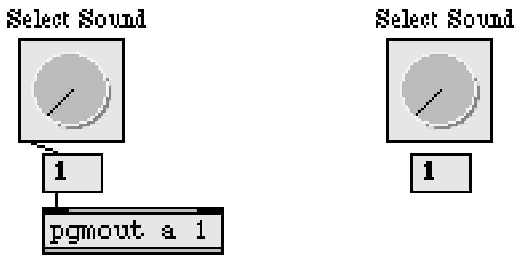
Note that you don't need to turn the **Segmented Patch Cords** option on in order to make patch cords that turn corners. Shift-clicking on an outlet will allow you to connect a patch cord in "segmented" mode.

Hide On Lock

- Unlock the Patcher window. You will see several objects and patch cords that were not visible before.

When editing a patch, you can select objects and/or patch cords and choose **Hide On Lock** from the Object menu. This sets those objects to be invisible when the Patcher window is locked. **Show On Lock** makes objects visible which were previously made invisible with **Hide On Lock**.

Hide On Lock is an invaluable feature for making patches with a good user interface. For example, in the patch in the right part of the Patcher window, there is no reason that the user needs to see the **pgmout** object and the patch cords connecting the various objects. All that the user of your patch needs or wants to see is the **dial**, the label (**comment**), and the number being sent out by the **dial**.



The Hide On Lock command lets you hide unsightly parts of a patch

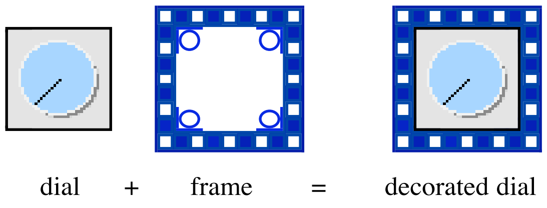
Note: If you select a region of the Patcher window and hide several objects at once with Hide On Lock, the objects will be hidden but the patch cords will still be visible. To hide patch cords, you must select them before choosing Hide On Lock. You can select patch cords by holding down the Option key on Macintosh or the Alt key on Windows while dragging a box around a group of objects. Alternatively, choosing Select All from the Edit menu selects all boxes and patch cords, or you can just click on patch cords individually, using the shift key to select multiple objects.

Graphical user interface objects (such as **button**, **toggle**, **dial**, etc.) that have been hidden with **Hide On Lock** do not respond to clicking and dragging with the mouse.

Paste Picture

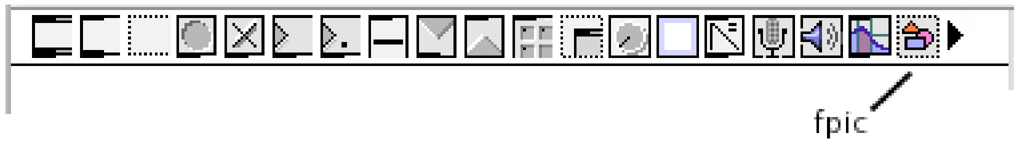
Your completed patch can actually have any visual appearance you choose, because you can import pictures from other applications such as painting and drawing programs. Cut or Copy the picture from another application, then in Max, choose **Paste Picture** from the Edit menu to paste the picture into a Patcher window.

For example, the decorative border around the **dial** in this patch is actually just a frame drawn in a painting program. The **dial** was then placed on top of the frame to give the illusion of a different kind of **dial**.



Similarly, the two buttons marked OFF and ON are not actual Max objects. They are pictures that were drawn in another program and pasted in with the **Paste Picture** command.

The Max icon was placed in the Patcher window by a different method; it is contained inside an **fpic** object. If you have designed a picture and saved it as a graphics file, **fpic** can load that external file into memory when the patch is opened.



After you place the **fpic** object in the Patcher window, while it is still selected, choose **Get Info…** from the Object menu. The **fpic** Inspector will appear. Click the *Choose...* button and a standard open file dialog appears. After you choose the desired graphics file, you can resize the **fpic** to display as much or as little of the picture as you want.

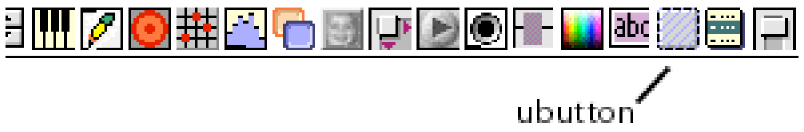
When you save the patch, **fpic** saves a reference to the graphics file so that can will load that picture automatically the next time the patch is opened. With this method, the graphic is not saved as part of the patch. If you are using the same graphic in several patches, you can save memory and disk space by using **fpic** objects that all reference the same file. You must take care, of course, that the file is located where Max can find it. See the discussion of the file search path in the Overview section of the Fundamentals manual.

(The Max icon in this particular patch doesn't do anything. It's just there to demonstrate the **fpic** object, which loads in a small graphics file containing a picture of the Max icon.)

Clicking on a Picture

- Lock the Patcher window and click on the picture of a button marked ON. The **metro** object is started, just as if you had clicked on the **toggle**. Click on the picture of an OFF button to stop the **metro**.

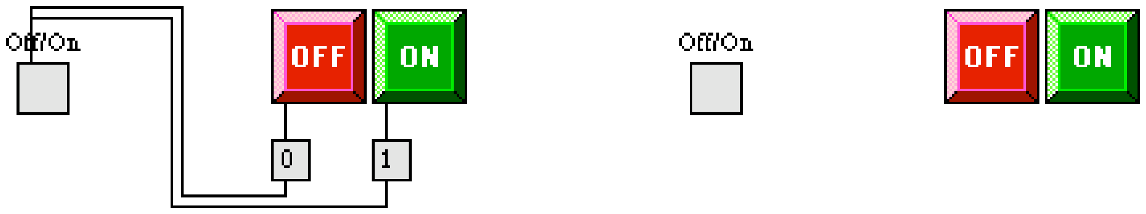
We know that the OFF and ON buttons are just pictures, so how do they turn the **metro** off and on? The pictures seem to respond to a mouse click because transparent buttons—**ubutton** objects—have been placed on top of them.



The **ubutton** object is a rectangular button that becomes invisible when the Patcher window is locked. When you click down on a **ubutton**, it sends a bang out its second outlet and inverts the part of the screen it covers. When the mouse button is released, **ubutton** sends a bang out its left outlet and becomes transparent again.

A **ubutton** can be placed over a picture or a **comment** (or over nothing, for that matter, if you just want an invisible button) to make that portion of the screen respond to a mouse click. The pixel- inverting feature of **ubutton** can also be used to highlight a spot on the screen. Look under **ubutton** in the Max Reference Manual for details.

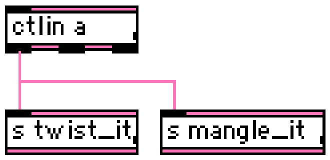
The connections between the **ubutton** objects and the **toggle** are hidden from sight with the **Hide On Lock** command.



Coloring and Resizing Objects

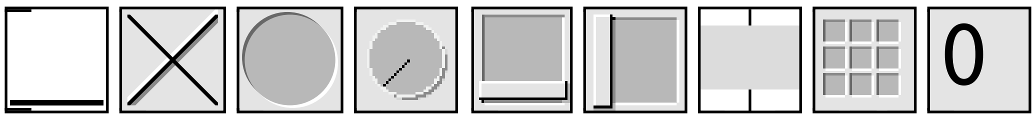
In addition to adding color to your patches by pasting in pictures, you can set certain user interface objects to a color other than gray by selecting them and choosing a color from the **Color** submenu of the Object menu. Objects that can be colored in this way include **button**, **dial**, **hslider**, **rslider**, and **uslider**. (The color of the **led** object is set by selecting it and choosing **Get Info...** from the Object menu.) If you have selected an object that cannot be colored, the **Color** submenu will be disabled in the Object menu.

You can color the top and bottom edges of an object box by using the **Color** submenu of the Object menu.



Many objects allow you to change their colors with RGB values; this is normally done using the object's Inspector window. These patchers are opened when you choose **Get Info...** from the Object menu with a single object selected.

The size of objects can be altered by dragging on the *grow bar* in the bottom-right corner of the object. This lets you customize the look of the graphical objects in your Max patches.



Different objects, resized to be the same size and shape

Tutorial 19

Summary

The **Segmented Patch Cords** option lets you create patch cords that bend around objects, making your patches easier to understand. You can also create segmented patch cords without this option turned on by shift-clicking on an outlet.

Objects and patch cords can be hidden from the sight of the user with the **Hide On Lock** command, so that the user sees only those things you want seen. Objects which have been hidden with **Hide On Lock** do not respond to clicking and dragging with the mouse.

Pictures can be imported from graphics applications and placed in a Patcher window with the **Paste Picture** command. A picture can be loaded from a graphics file on disk and displayed in a Patcher with the **fpic** object.

Graphical objects can be resized by dragging on the grow bar in the bottom-right corner of the object box. The color of some objects can be changed by selecting them and choosing the **Color** command from the Object menu. The appearance of a graphical object can also be altered by pasting a picture around it to serve as a frame.

The transparent button object, **ubutton**, lets you make any portion of the screen respond to clicks from the mouse.

The combination of these features lets you design the screen to have almost any imaginable appearance and respond to the mouse in a variety of ways.

See Also

fpic	Display a picture from a graphics file
ubutton	Transparent button, sends a bang
Menus	(**Fundamentals**) Explanation of commands

Tutorial 20: Using the Computer Keyboard

ASCII Objects

When you type a key on your computer keyboard, a message is sent to the computer telling it which key you typed. Max has objects for receiving and interpreting this information.

The *American Standard Code for Information Interchange* (ASCII) is the standard system of key numbering. The **key** object receives *key down* messages from the computer keyboard and sends the ASCII number of the typed key out its left outlet. (Because **key** receives its input directly from the computer keyboard, it has no inlet.) The ASCII number can then be used in a patch just like any other number.

The **keyup** object is similar to **key**, but it sends out the ACSII number of a key when it is *released* (when a *key up* message is received from the keyboard). The **numkey** object receives ASCII numbers from a **key** or **keyup** object, and deciphers them to determine if a *number* is being typed in on the keyboard. It reports the value of the number the user is entering.

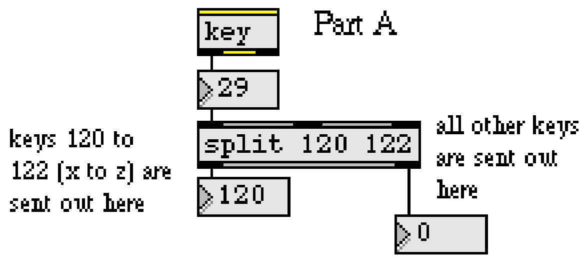
key

This example contains a single, rather complex patch. At the top of Part A is a **key** object. Every time you press a key on the computer keyboard, **key** sends the ASCII number of that key out its left outlet. The ACSII number is sent to an object called **split**.

split

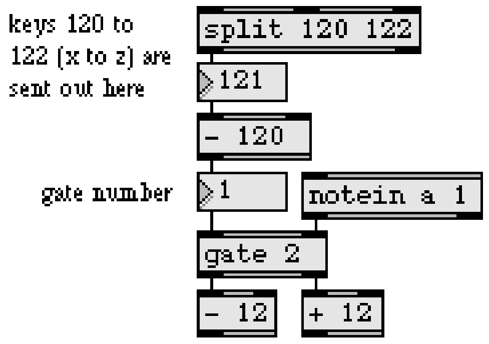
The **split** object is a combination of relational operator and gate. It looks for a specific range of numbers. If the number received in the left inlet is within the specified range, it is sent out the left outlet. Otherwise it is sent out the right outlet.

The minimum and maximum values of a **split** object's range can be typed in as arguments and/or they can be supplied in the middle and right inlets. In this case, ASCII numbers 120 through 122 (keys *x*, *y*, and *z*) are sent out the left outlet, and all other numbers are sent out the right outlet.


key
Part A
29
keys 120 to 122 (x to z) are sent out here
split 120 122
all other keys are sent out here
120
0

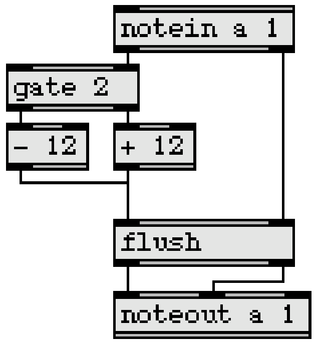
Using Key Commands to Control a gate

Let's see what happens when the keys *x*, *y*, and *z* (120, 121, and 122) are typed. First of all, 120 is subtracted from the key number, resulting in the numbers 0, 1, and 2. These numbers are used to control a **gate**. The letter x closes the **gate**, the letter y opens the left outlet, and the letter z opens the right outlet.



Pitch information from **notein** is passed through the open outlet of the **gate**. So, if the letter *y* is typed, pitches are passed to the **- 12** object and are transposed down an octave. If the letter *z* is typed, the pitches are passed to the **+ 12** object and are transposed up an octave. If *x* is typed, the **gate** is closed and no pitches are passed through.

The transposed pitches are combined with velocities in the **flush** object, and are sent to **noteout**.



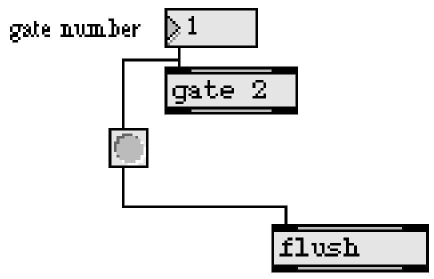
- Type the different key commands *x*, y, and *z*, and listen to the change in effect for each command when you play on your MIDI keyboard.

Turning Off Transposed Notes

At first glance, the **flush** object may seem to be unnecessary in this patch. Why can't the pitches be combined with velocities right in **noteout**? They can, but this would leave open the possibility that some note-off messages would not be received by the synth. Consider the following scenario.

Suppose you type the letter *y* to transpose pitches down an octave. Then you play and hold down the note C3 (60) on your MIDI keyboard. This will cause the note 48 to be sent to **noteout**. Before you release the note 60 on your keyboard, you type *z* to transpose pitches up an octave. Now when you release the note 60, a note-off for note 72 will be sent to the synth. The note 48 will not get turned off.

To solve this potential problem, the notes are sent first to a **flush** object. Each time a number is received in the left inlet of **gate**, a bang is also sent to **flush** to turn off any held notes. In this way, note-offs are always provided for any notes that are being held when the status of **gate** is changed.

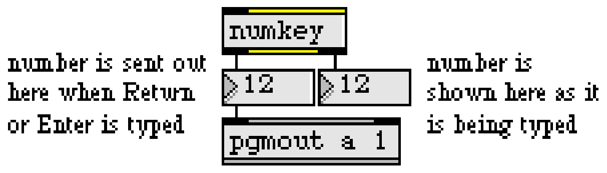


Whenever a gate control number is received, a bang is sent to flush

Of course, if you never play notes and give commands at the same time, this precaution is unnecessary. As a general rule, though, whenever you are processing notes (for example, transposing them) it's good to make sure that a note-on message is always followed by a corresponding note- off message. Changing the transposition, closing a gate, etc. while a note is being played can often cause this sort of problem.

numkey

Part B of the patch shows how **numkey** interprets numbers typed on the computer keyboard. ASCII values from the **key** object are sent to **numkey** (except for the keys *x*, *y*, and *z*, which **numkey** ignores anyway because they are not numerical digit keys).


numkey
number is sent out here when Return or Enter is typed
12
12
number is shown here as it is being typed
pgmout a 1

As digits are typed, **numkey** sends the current number out its right outlet. The Delete (Backspace) key can be typed to erase the last digit entered. When you have typed in the complete number, you can type the Return or Enter keys on Macintosh or the Enter kay on Windows to send the number out the left outlet. In general, the right outlet is used for showing what number is being typed, and the left outlet is used for actually sending it.

- Try changing the sound on your synth by typing in the program number on the computer keyboard.

Using a combination of **key** and **numkey** to type in numbers is different from typing numbers directly into a **number box**, because you have to click on the **number box** before typing into it. The **key** and **keyup** objects receive *all* typed keys, so there is no need to select any object before numbers are typed in via **numkey**.

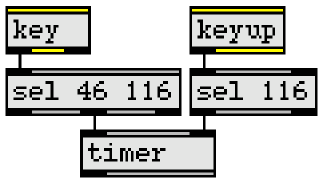
keyup

Typing a key on the computer's keyboard, just like playing a note on your MIDI keyboard, sends two messages to the computer—one when the key is pressed down, and another when the key is released. The **keyup** objects sends the ASCII number of any key that is released.

In Part C of the patch, we measure the time that a key is held down by measuring the time between arrival of a number sent by **key** and a number sent by **keyup**.

timer

In Part C, we use **sel** objects to look for specific ASCII values. Both **sel** objects look for the number 116, which is the key *t*. (We chose t as a mnemonic for *tempo*.) When *t* is pressed down a bang is sent to the left inlet of a **timer** object. When the key *t* is released a bang is sent to the right inlet of the **timer**.



The **timer** object outputs the number of milliseconds between a bang received in its left inlet and a bang received in its right inlet.

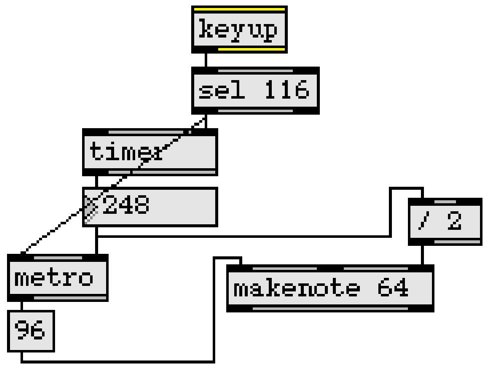
Note: **timer** is an exception to the general rule of the left inlet being the one that triggers output. In this case, a bang in the left inlet starts the timing process, but the right inlet is the one that causes the elapsed time to be sent out.

Using Duration to Set Tempo

This patch uses the duration that a key is held down to set the speed of a metronome. When the key *t* is released, a bang is sent to the right outlet of **timer**, which reports the time that the key was held down. (We used the message from **key** to start the **timer**.)

The time is sent to the right inlet of **metro**, and is also divided by 2 and sent to the right inlet of **makenote**. In this way, the duration of the notes played will be 1/2 the amount of time between notes, giving a *staccato* effect.

The release of the *t* key also starts the **metro**. Notice that the **timer** is triggered *before* the **metro**, so that the time values will arrive in **metro** and **makenote** before **metro** is started.



The metro sends the pitch 96 to makenote until the period key (.) is typed to stop the metro.

- Hold the *t* key down for various lengths of time and listen to the change in the tempo of the **metro** (and the duration of the notes). Type the period key (.) to stop the **metro**.

Summary

The **key** object reports the ASCII code of keys typed on the computer's keyboard. The **keyup** object reports the ASCII code of keys when they are *released*. The **numkey** object interprets ASCII received from **key** or **keyup**, and reports any numerical values being typed.

ASCII values from **key** or **keyup** can be used to send commands to a patch, opening a **gate** or triggering processes. A relational operator such as **sel** can be used to look for certain keys being typed.

The **split** object looks for numbers within a certain range. If an incoming number is within range, it is sent out the left outlet, otherwise it is sent out the right outlet.

The elapsed time between any two events can be reported with **timer**. The **timer** is started by a bang received in its left inlet, and the elapsed time is sent out when a bang is received in the right inlet.

See Also

key	Report keys typed on the computer's keyboard
keyup	Report keys released on the computer's keyboard
numkey	Interpret numbers typed on the computer's keyboard
split	Look for a range of numbers
timer	Report elapsed time between two events

Variables

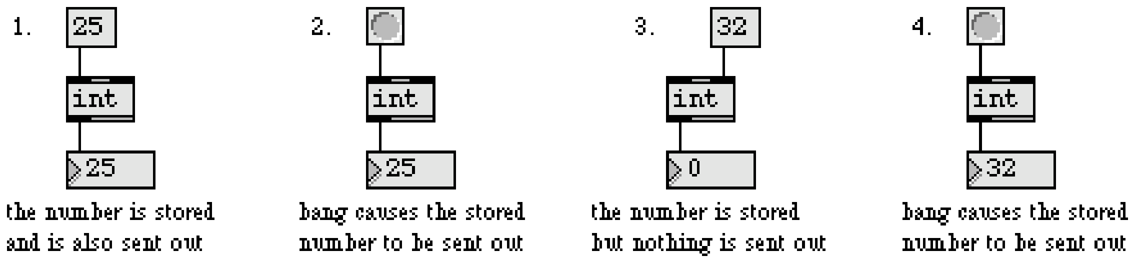
In traditional programming languages, *variables* are places in memory used by a program to store numbers so they can be recalled at a later time.

Many objects in Max are capable of storing a number and recalling it later. For example, the **number box** will send out the number stored in it when it receives a bang message.

In this tutorial, we'll use objects that do nothing but store a number and send it out when a bang is received in the left inlet. These objects are **int** for storing integer numbers, and **float** for storing decimal numbers.

int and float

When a number is received in the left inlet of an **int** or a **float** object, it is stored and sent out the outlet. Whenever a bang is received in the left inlet, the stored number is sent out again.



When a number is received in the right inlet, it is stored without triggering any output (replacing the previously stored value).

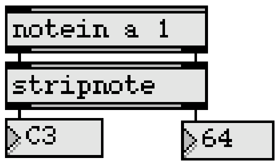
The **int** and **float** objects both function in exactly the same way. The only difference is the type of number they store. When a number with a decimal point (float) is received by an **int** object, the number is converted to an int before being stored, and vice versa.

An initial value to be stored in **int** or **float** can be typed in as an argument. If there is no argument, the objects initially store the number 0.

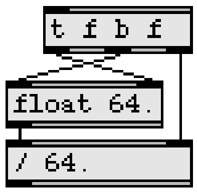
Using float

The patch in the left part of the Patcher window uses both **int** and **float**. Once we understand what the patch does, the need for these objects should be clear.

The combination of **notein** and **stripnote** should be familiar to you. It's the easiest method of getting note-on data from a MIDI keyboard (without getting note-off data).



The velocity is converted to three separate messages by the **t** (**trigger**) object: a float, a bang, and another float. The first float (from the right outlet of **t**) is sent to the right inlet of the / object, where it is stored. The bang then causes the **float** object to send out its stored number.

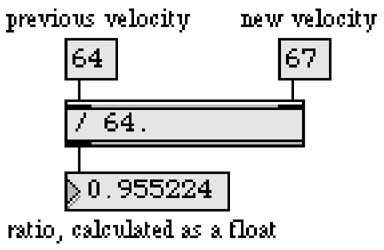


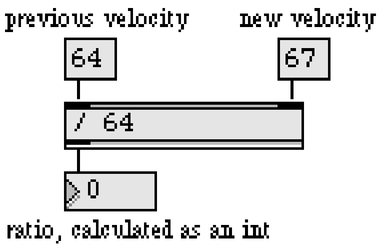
This triggers a series of calculations, finally resulting in a number being sent to the right inlets of **makenote** and **metro**. Then the last float (from the left outlet of **t**) is stored in the **float** object.

What this means is that each time a new velocity is received, the *previous* velocity is divided by the *new* velocity, then the new velocity is stored as the "previous" one. The result is the ratio of the old velocity to the new velocity.

Note: The velocity could just as well be stored in an **int** object, but it is converted to float by the **/** object in any case. Since the conversion from int to float has to occur somewhere, we made the conversion with the **t** object.

Why do we need to divide the two velocities as floats? Consider the possible cases. The range of possible ratios between two note-on velocities is from $^{1}/_{127}$ to $^{127}/_{1}$, (i.e., from 0.007874 to 127.0). Whenever the previous velocity is less than the new one, the ratio will be some fraction between 0 and 1. But if we performed an integer division, the result would always be 0 when the velocity is increasing.





Float division is needed to get the precise ratio between two velocities

The ratio is then multiplied by 250 (again the numbers are calculated as floats, so that ratios less than 1 are not converted to 0 before the multiplication), and the result is used as the new note duration for **makenote** and the new tempo for **metro**.

However, not all the numbers we get this way are really suitable as musical values. Extreme changes in velocity result in very small or very large numbers. The range of possible values in this calculation can be as small as 1.9685 (which will be truncated to 1 when it is converted to int) and as great as 31750. So, we use **split** to limit the values between 40 milliseconds (25 notes per second) and 2000 milliseconds (one note every 2 seconds). Numbers that exceed these limits will be ignored. The **split** object automatically converts the floats back to ints.

- Play a few notes on your MIDI keyboard, and observe how changes in velocity affect the numbers being sent out of **split**. When the velocity is increasing, the numbers are less than 250. When the velocity is decreasing the numbers are greater than 250. Extreme changes in velocity result in extremely large or small numbers, which are ignored by **split**.

Using int

What happens next in our patch? The velocity is sent to the middle inlet of **makenote**, where it is stored. Then the pitch value is stored in the **int** object.

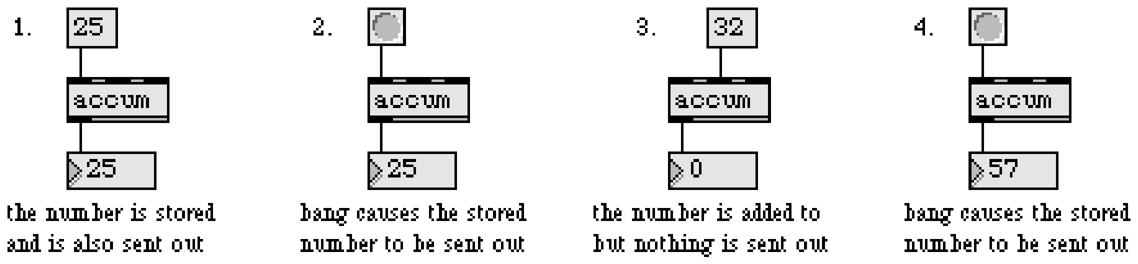
Well, so far we've seen that a lot of numbers get stored, calculated, and changed by playing on the synth, but nothing else happens...until we turn on the **metro**. The bang messages from the **metro** trigger the **int** object, which sends out its number—whatever pitch was most recently played—to **makenote**. The speed of the **metro** is dependent on the change in velocity between successive notes played on the synth.

- Turn on the **metro** and play on your MIDI keyboard. Notice how you can affect the speed, velocity, and duration of the repeated notes by changing the velocity with which you play.

accum

Another storage object, **accum**, performs internal additions and multiplications to change its stored value.

The left inlet of **accum** functions just like that of the **int** and **float** objects: a number received in the left inlet is stored and sent out the outlet, and a bang sends the stored number out again. However, the middle and right inlets of **accum** are used to *add* to the stored number or *multiply* the stored number, respectively. The number is changed without anything being sent out the outlet.



An initial value to be stored in **accum** can be typed in as an argument. If there is no argument, the object initially stores the number 0. The value stored in **accum** is normally an int, but if the typed- in argument contains a decimal point **accum** stores a float. Multiplication in **accum** is always done with floats, even if the stored number is an int.

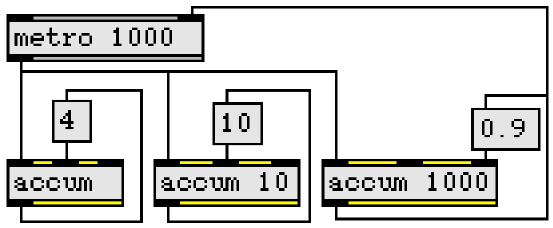
Using accum

The **accum** object is most useful for storing a value that you wish to change often by adding to it or multiplying it. For example, you may want to continually increment a number by adding some amount to it over and over. The second patch in the Patcher window shows an example of incrementing.

- Click on the **toggle** to start the **metro**. Notice how the pitch, velocity, and duration values sent to **makenote** change continually. The amount of change is directly related to the numbers being added or multiplied in the **accum** objects.

Each time the **accum** objects receive a bang from the **metro**, they send their stored values to **makenote**, and they also trigger **message** boxes which add some number back into the stored value or multiply the stored value by some amount. (Note: Until now we've usually triggered the **message** box with a bang, but it can also be triggered with a number.)

The result is that the **accum** objects change their own stored value each time they receive a bang.



These values would soon exceed reasonable ranges unless we place some kind of restriction on them. In this patch, numbers sent to **makenote** loop repeatedly through cycles of different lengths. Two methods of looping are shown.

The **accum** that sends durations to **makenote** (and tempos to **metro**) starts at 1000, and multiplies itself by 0.9 every time it sends out a number. Eventually the number is reduced to be less than 40. When this happens, the message set 1000 is sent to the left inlet of **accum**, resetting its stored value.

We have already seen a set message used to set the value of a **slider** without triggering output. It has the same effect when sent to the left inlet of **accum**. Every time the value of the **accum** goes below 40, it is reset to 1000 and the cycle begins again.

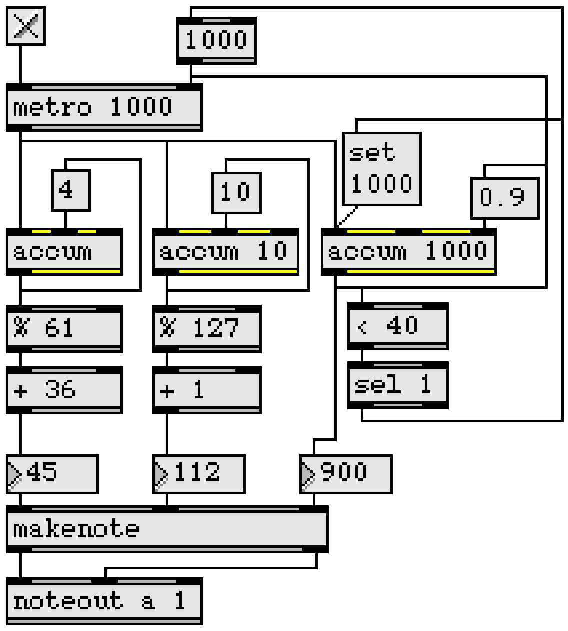
The tempo of the **metro** is also set back to 1000 at the same time, by sending a bang to the **1000** object. An object box that contains only a number is actually an **int** object (or a **float** object if the number contains a decimal point) with its initial value set to that number.

This shows the basic method of looping: change a value continually until some condition is met (for example, until it exceeds some limit), then reset the value and begin again.

The values stored in the other two **accum** objects continue to increase without being reset, but the modulo operator **%** limits the numbers so that they always cycle within a limited range. Using the **%** object is another good way of looping.

Loops Can Create Cyclical Patterns

Using loops in a program is a way of creating *periodicities*. In this patch, the parameters of duration, velocity, and pitch all have a different periodicity of recurrence, which makes the music seem to repeat itself while always changing slightly.



Actually, the duration changes according to a 32-note cycle, the velocity changes according to a 118-note cycle, and the pitch changes according to a 37-note cycle. Thus, the entire pattern is repeated every 69,856 notes—about every 6 hours. Not a melody likely to get stuck in your head.

Overdrive

Max can get rather busy playing music, running metronomes, doing mathematical calculations and printing numbers on the screen. Checking **Overdrive** in the Options menu causes Max to give priority to music-making tasks, and results in more accurate musical timing. If you hear improper delays on notes, or erratic performance, try enabling Overdrive.

Summary

Integer numbers can be stored in an **int** object, and sent out later by triggering **int** with a bang in its left inlet. Numbers with a decimal point can be similarly stored and recalled with a **float** object. The **accum** object can store and recall either an int or a float value, and can add to or multiply the stored value without sending it out.

Floats are useful for calculations that involve numbers between 0 and 1, or for any calculation that requires additional precision.

The **split** object is useful for limiting numbers to a specific range. All numbers it receives that are within its specified range are sent out its left outlet. Otherwise, they are sent out its right outlet.

Cycles of numbers can be produced by *looping*. A loop is created by continually changing a number, then resetting it when the number meets a certain condition. The **accum** object is well suited for such looping schemes.

See Also

accum	Store, add to, and multiply a number
float	Store a decimal number
int	Store an integer number
Loops	(**Max Topics**) Using loops to perform repeated operations

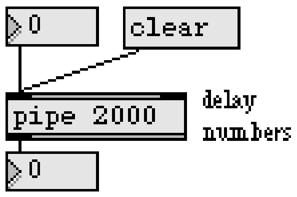
Delaying Numbers and bang messages

Messages normally pass through patch cords as fast as the computer can send them. However, you can also delay numbers, lists, or bang messages for a certain amount of time before sending them on their way. This is useful for creating a specific time lag between messages, or delaying notes to create echo effects.

The **pipe** object delays the numbers it receives for a certain amount of time before sending them on. The **delay** object (also called **del**) delays a bang for a certain amount of time before sending it on.

pipe

- Drag on the **number box** at the top of Patch 1. The numbers are all delayed for 2000 milliseconds by the **pipe** object.

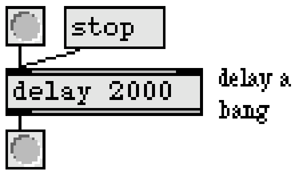


Each number is sent out the outlet a certain amount of time after it is received, so **pipe** can delay many numbers and send them out later in the same rhythm in which they were received. A number received in the right inlet sets the delay time, in milliseconds, to be applied to all numbers subsequently received in the left inlet. A clear message received in the left inlet erases any numbers currently being delayed in the **pipe**.

- Send some more numbers to the left inlet of the **pipe** in Patch 1, then quickly click on the word clear. Any numbers not yet passed through the **pipe** are forgotten.

delay

- Click on the **button** at the top of Patch 2. The bang is delayed for 2000 milliseconds before being sent out the outlet of the **delay** object.



Unlike **pipe**, which can keep track of many numbers at a time, **delay** can keep track of only *one* bang at a time. If **delay** receives a new bang while it's already delaying a bang, the old bang is lost and the new bang is delayed instead.

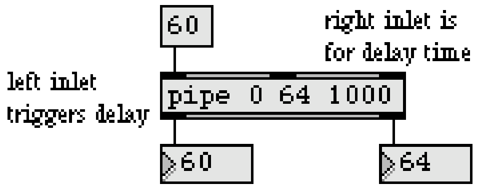
- Send many bang messages to **delay** in quick succession. Each bang received within 2 seconds of the previous one *erases* the previous one, so only the last bang gets sent out the output.

A number received in the right inlet of **delay** sets the delay time to be applied to any bang subsequently received in the left inlet. A stop message received in the left inlet stops any bang currently being delayed.

- Send another bang to **delay**, then quickly click on the word stop. The bang is not sent out.

Delaying Groups of Numbers

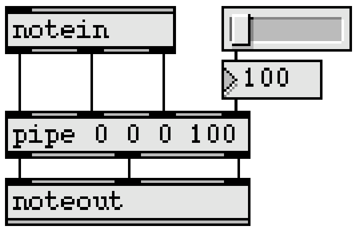
A single **pipe** object can actually delay several parallel streams of numbers—such as pitch, velocity, and channel information from **notein**—with a separate pair of inlets and outlets for each stream. To make a **pipe** with more than one outlet, type in one number argument for each stream of numbers you want to delay, plus an argument for the delay time. The last argument is always the delay time.



Arguments set initial value for each delay line.
Last argument is the delay time.

As with most objects, it's the left inlet that triggers the **pipe**. When a number is received in the left inlet, it is delayed along with whatever number was most recently received in the other delay line inlets. If no number has been received in the other inlets, **pipe** uses the initial value named in the argument, as in the example above. The numbers can also be received together as a list in the left inlet, with an additional number included at the end of the list to specify the delay time.

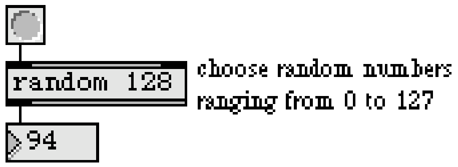
Patch 3 shows a **pipe** that delays three streams of numbers. The channel and velocity values from **notein** are received, and then the pitch value triggers the delay of all three numbers. The delay time can be changed by sending a new number to the right inlet of **pipe**.



- Try playing on your MIDI keyboard using different delay times.

random

Each time the **random** object receives a bang in its left inlet, it chooses a number at random and sends the number out the outlet. The range of numbers from which **random** chooses is determined by typing in an argument or by sending a number in the right inlet. The random values will always be between 0 and one less than the argument.



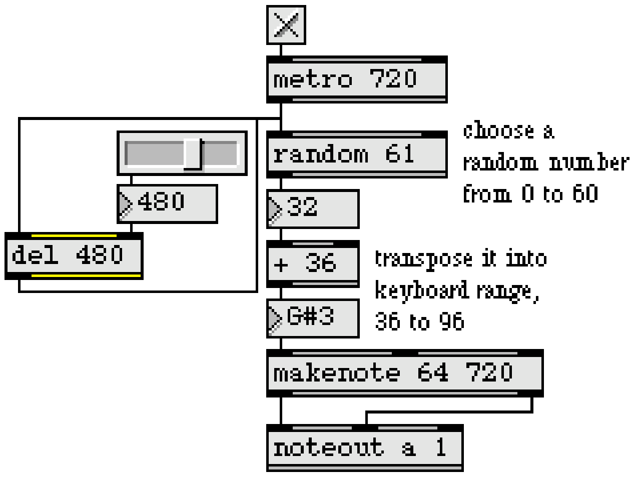
Different uses of random numbers will be seen in the course of the Tutorial.

Using Delayed Triggers

In Patch 4 the **metro** triggers **random** to send out a random number between 0 and 60 once every 720 milliseconds. 36 is added to the number to bring it up into the range of a 61-key MIDI keyboard, and it is then transmitted as a pitch to be played on the synth.

Tutorial 22

The bang from **metro** is also sent to the **del** object, where it is delayed a certain time before being sent to **random**. A new randomly chosen number is then sent out, so actually two notes are played every 720ms. The rhythm between the two notes depends on the delay time sent to the right inlet of **del** from the **hslider**.


metro 720
random 61
choose a random number from 0 to 60
32
480
del 480
+ 36
transpose it into keyboard range, 36 to 96
G#3
makenote 64 720
noteout a 1

- Turn on the **metro** and experiment with creating different rhythms by changing the delay time of the **del** object.

Summary

A single bang can be delayed for a specific amount of time by the **delay** object, also called **del**. If a second bang is received while the first bang is being delayed, the first bang is forgotten and the second bang is delayed.

Numbers or lists can be delayed by the **pipe** object. A **pipe** can delay a series of numbers, and output them later in the same rhythm in which they were received.

A **pipe** can also delay a list of numbers (or numbers received together, such as pitch-velocity pairs) when arguments are typed in to indicate how many numbers are to be delayed.

The delay time is specified in milliseconds, by a number received in the right inlet (or typed in as an argument).

Delays can be used to create echo effects or rhythms.

Each time the **random** object receives a bang in its left inlet, it generates a random number between 0 and one less than its argument and sends the number out its outlet.

See Also

delay	Delay a bang before passing it on
pipe	Delay numbers or lists
random	Output a random number

Tutorial 23: Test 4—Imitating a Performance

Creating Imitation

In previous chapters you have seen how to transpose notes played on your MIDI keyboard, and how to delay notes. Try making a patch of your own that imitates what you play, starting on a different note.

1. Make a patch that imitates whatever you play, 3 seconds after you play it, transposed up a perfect fifth, *and* also imitates whatever you play 6 seconds later, transposed up an octave.

Hints

The notes you play on your keyboard will have to be sent to two different places. In one, the pitch will be transposed up by 7 semitones and all the note data will be delayed by 3000 milliseconds. In the other, the pitch will be transposed up by 12 semitones and the note data will all be delayed by 6000 milliseconds.

Each imitation should use one **pipe** object to delay velocity and pitch data together.

- Page once to the right in the Patcher window to see the solution to the problem, labeled Patch†1. Note that the order of the **+** object and the **pipe** object could be reversed; the transposition could take place *after* the delay.

Let the User Type In a New Delay Time

If that exercise was too easy for you, try this more difficult one.

2. In Patch 1 each imitation of the melody comes 3 seconds later than the previous one. Make a version of Patch 1 which lets the user type in a new delay time between imitations.

This presents two potential problems:

- What happens if the user types in a ridiculous delay time such as -1000 or 3600000?

- What happens if the user types in a much shorter delay time while holding down a note, and the note-off gets delayed less than the note-on and is played *before* the note-on?

These problems represent the sort of extreme or unlikely cases you must take into consideration to protect against your program malfunctioning or producing unwanted results.

Dealing With Potential Problems

The problem of negative delay times is not serious because the **pipe** object will set any negative delay time it receives to 0. The problem of extremely large delay times, on the other hand, can be more serious.

If the delay time is very long and you are playing a lot of notes, the number of notes being stored could cause **pipe** to run out of memory. This would cause some notes to be lost, and could conceivably even cause the computer to crash.

The way to deal with this problem is to limit the numbers the user types in as a delay time, and only send them to **pipe** if they are reasonable. Try using a **split** object or **hslider** to limit the numbers between 0 and 15000.

The problem of note-offs being played before note-ons could occur if the user types in a much smaller delay time while holding down a note on the synth. It would result in stuck notes on the synth.

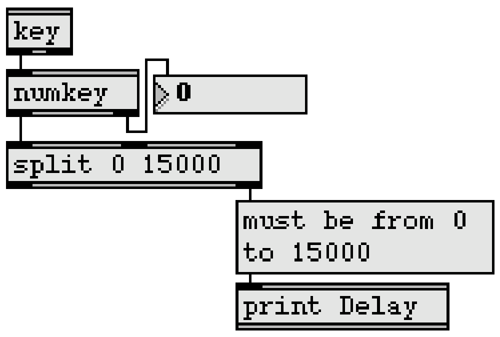
One solution is to compare each new delay time to the previous one. If the new one is smaller, send a note-off to **pipe** for any pitch being held down on the synth. This requires running the notes through a **flush** object before sending them to the **pipe** objects, and also requires comparing the delay times and sending a bang to the **flush** objects if a smaller delay time is typed in.

Solution to Exercise 2

- Scroll the Patcher window all the way to the right to see Patch No. 2, a possible solution to the exercise.

Note: We have set Patch 2 to receive notes only on MIDI channel 2 so that it will not play while you are trying out Patch 1. To hear Patch 2, set your keyboard to transmit on MIDI channel 2.

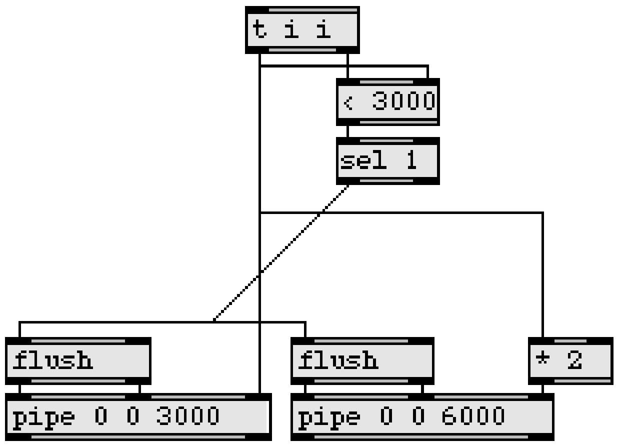
We have used a combination of **key** and **numkey** to get the numbers typed on the computer's keyboard. The typed numbers are sent to **split**, and any numbers less than 0 or greater than 15000 will cause an error message to be printed in the Max window.



Invalid numbers cause an error message to be printed.

The typed delay time is first sent to the relational operator **<** to compare it with the current delay. If it is less than the current delay, a bang is sent to the **flush** objects, causing them to send out note-offs for any notes that may be held down on your MIDI keyboard.

The new delay time is then sent to the right inlet of **<**, to be stored as the current delay time. It is also sent to the right inlet of the two **pipe** objects. Note that it is doubled before being sent to the **pipe** on the right, so that the right pipe will delay twice as long as the left **pipe**.



If the new delay time is less than the current one flush any held notes before changing the delay time.

Tutorial 23

Summary

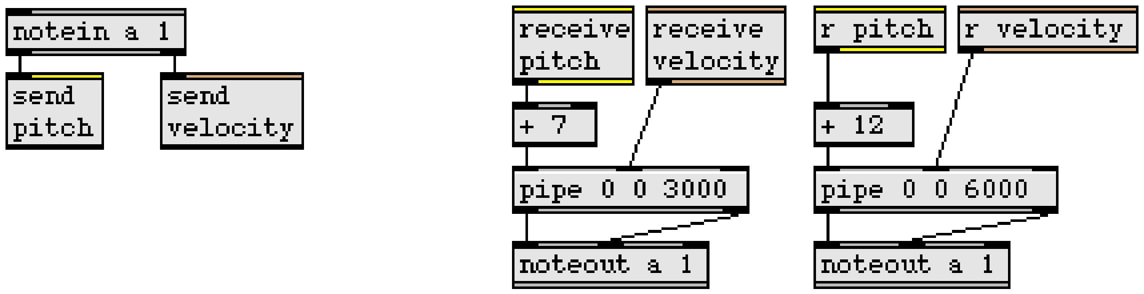
Delay and transposition can be combined to create imitation.

Always consider unlikely possibilities. For example, whenever you ask the user to supply a value, check to make sure it is a valid value before using it. (You can print an error message when an invalid number is received, or you can just change it to some valid value.) Whenever you are processing note data, make sure that note-ons are always followed by note-offs.

Sending Messages Without Patch Cords

It's possible to send any type of message without using a patch cord with the **send** and **receive** objects. A message in the inlet of a **send** object comes out the outlet of any **receive** object that has the same argument.

In this patch we have redone the imitating patch from the previous chapter using **send** and **receive** objects (also called **s** and **r**).



- Play on your MIDI keyboard. The note data is sent to the **receive** objects (and **r** objects) that have the same name (argument) as the **send** object.

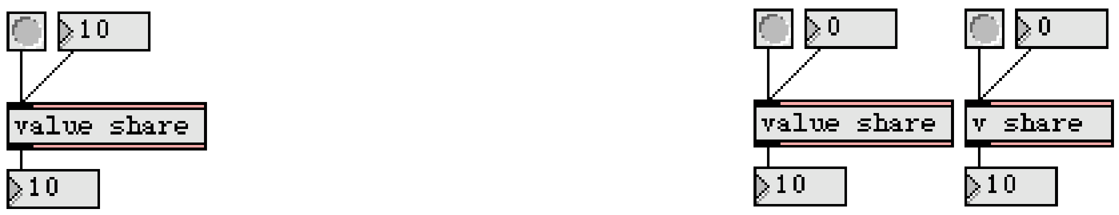
The name argument of a **send** object is like a unique radio frequency, and any **receive** object with the same name is "tuned in" to that frequency. Any type of message can be sent with **s** and **r**: ints, floats, lists, symbols, etc.

Communication Between Patcher Windows

The **s** and **r** objects have one particular advantage over patch cords, in that they can communicate *even if the objects are not in the same Patcher window*. This is a very valuable feature, enabling different patches to communicate with each other. You must take care when naming your **send** and **receive** objects, though, so you won't send a message to another Patcher window unintentionally.

value

The **value** object (also called **v**) stores any message received in its inlet. The message is sent out when a bang is received. All **value** objects with the same argument share the same storage location in the computer's memory, so the number can be stored and recalled by any one of the objects. When a new message is stored in one **value** object, all others that share the same name will also contain the new message.



A message stored in one location can be recalled in another location.

- Use one of the **number box** objects to store a number in the **value** object named share. The number can be recalled from any of the **value** share objects by sending a bang to its inlet.

All **value** objects with the same name share the same value, even if they are located in different Patcher windows.

Summary

Any message received in the inlet of a **send** object comes out the outlet of all **receive** objects with the same name (argument), even if they are in different Patcher windows. This is valuable for communicating between Patchers.

A message stored in a **value** object is shared by all **value** objects with the same name, even if they are in different Patcher windows. When a **value** object receives a bang in its inlet, it sends the message out the outlet (even if the message was received in another **value** object with the same name).

See Also

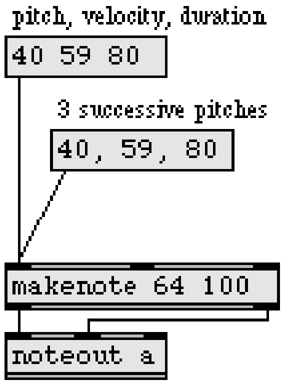
pv	Share variables specific to a patch and its subpatches
receive	Receive messages, without patch cords
send	Send messages, without patch cords
value	Share a stored message with other objects

Using the message box

So far we have used the **message** box to send a single message, triggering it either with a mouse click or with a bang, a number, or a list in its inlet. The **message** box has many additional features for constructing and changing messages, some of which are displayed in this patch.

comma

If a **message** box contains a comma, messages are sent out one after another. In this way, messages can be sent in rapid succession in response to a single trigger.



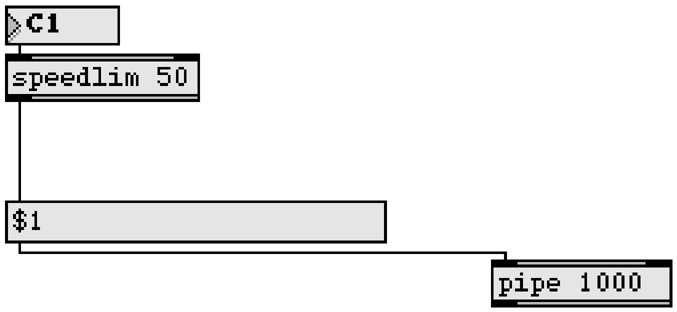
- Click on the two **message** boxes (marked A) in the bottom-left portion of the Patcher window. One **message** box contains a list of three numbers, 40 59 80. When **makenote** receives the list, it interprets the third number as a duration, the second number as a note-on velocity, and the first number as a pitch. The other **message** box contains three separate messages. It sends 40, then 59, then 80, and each number is interpreted as a pitch by **makenote**. You can see the messages printed in the Max window, and you can hear the difference in result.

This is one way to play a chord.

The Changeable Argument: $

The dollar sign ($) is a special character in a **message** box. It is a *changeable argument*, an argument that is replaced by an item from the incoming message. For example, if a **message** box contains The pitch is $1 and the velocity is $2, and receives the message 60 64 in its inlet, it will send out The pitch is 60 and the velocity is 64. The numbers 60 and 64 are stored in place of $1 and $2 until they are replaced by other values received in the inlet. The dollar sign can be followed by numbers in the range 1-9.

- Drag on the **number box** (marked B) in the top-left corner of the Patcher window. After being limited by **speedlim**, each of the numbers triggers the **message** box. Because the **message** box contains the changeable argument $1, the $1 is replaced by the incoming number before the message is sent out.



The incoming number is stored in the changeable argument $1 before the message is sent out.

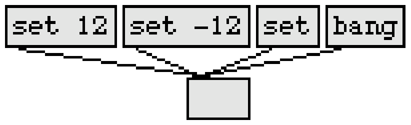
The number is then sent to **pipe**, and 1000ms later it is sent to **makenote** and on to the synth.

- The last number to trigger the **message** box is still stored in place of the $1 argument. Now if you trigger the **message** box with a bang, the stored number will be sent out again.

The set Message

We have already seen that the message set, followed by a number, can specify or replace what is stored in many objects without triggering any output. The word set, followed by *any message* can replace the contents of a **message** box without triggering output. The word set by itself clears the message. (When an empty **message** is triggered, nothing is sent out.)

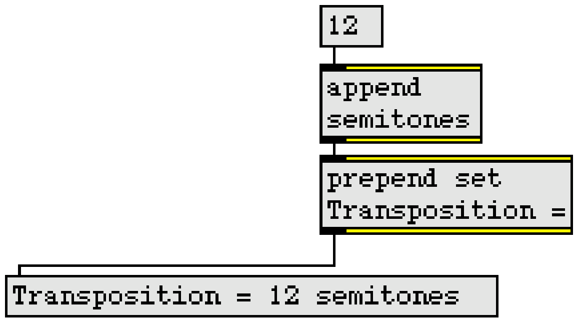
- Click on the different set messages in the portion of the patch marked C.



- Although the text in the **message** box changes, nothing is sent out until it is triggered with the bang message.

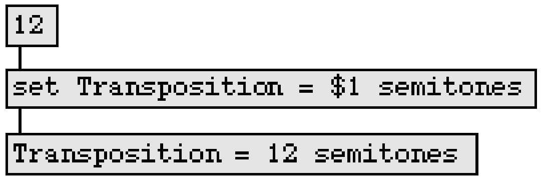
append and prepend

The **append** and **prepend** objects are for constructing complex messages. The **append** object *appends* its arguments (preceded by a space) at the end of whatever message it receives, and sends the combined message out its outlet. The **prepend** object places its arguments (followed by a space) *before* the message it receives, and sends the combined message out its outlet. An example of these objects is in the bottom-right part of the Patcher window.



When the **append** object receives a message—for example, the number 12— it places the word semitones after it and sends out 12 semitones. The **prepend** object then puts set Transposition = before it and sends out set Transposition = 12 semitones, which changes the contents of the **message** box to Transposition = 12 semitones.

The same result could be obtained using only **message** boxes, in the following manner:


```
12
set Transposition = $1 semitones
Transposition = 12 semitones
```

backslash

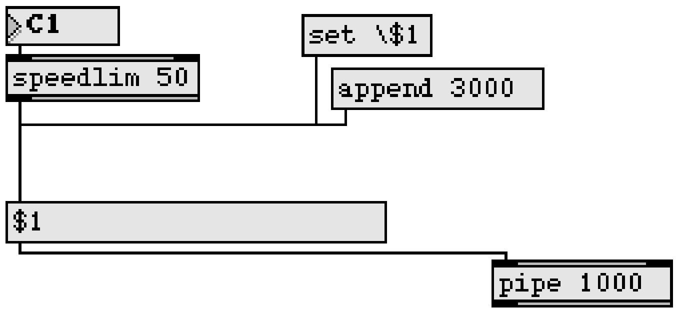
The backslash (\) is a special character for negating *other* special characters. A special character that is preceded by a backslash loses its special characteristics and is treated like any other character. This is necessary if you want to include a character such as a comma or a dollar sign in a message without its being interpreted to have a special meaning.

The append Message

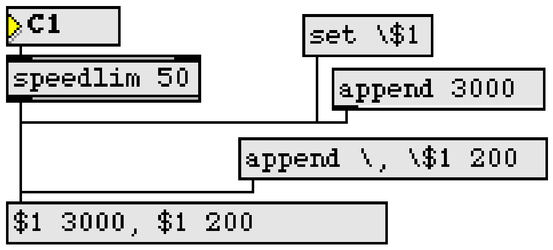
When append, followed by any message, is sent to a **message** box, the message following append will be added to the contents of the **message** box.

- In the part of the patch marked D, click on the messages set \$1 and append 3000 to construct the message $1 3000. (Notice that we had to precede the dollar sign with a backslash. Otherwise, the **message** box would have tried to interpret $1 as a changeable argument, and the message would have been set 0). Then drag on the **number box** marked B.

The $1 argument is replaced by the incoming number and is sent out as a list with the number 3000. The list is received by **pipe**, 3000 is stored as the new delay time, and the numbers are delayed for 3 seconds before being sent on.



- Next, click on all three **message** boxes in part D, to construct the message $1 3000, $1 200.

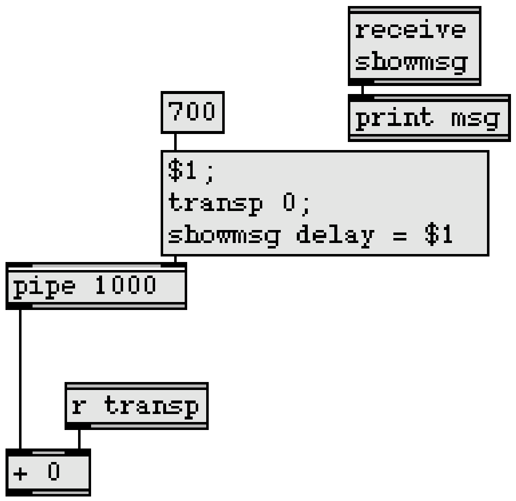


- Now when you send numbers to the **message** box, it sends out two lists, resulting in each number being delayed both 3000ms *and* 200ms.

semicolon

When a semicolon (;) appears in a **message** box, the first word after the semicolon is interpreted as the name of a **receive** object. The rest of the message (or up to the next semicolon) is sent to all **receive** objects with that name, instead of out the **message** box's own outlet.

- Click on the **message** box marked E, containing the number 700.



The number 700 is sent out the outlet to the right inlet of **pipe**, the number 0 is sent out the outlet of the **r** transp object, and the message delay = 700 is printed in the Max window. This is a way of sending many different messages to different places with a single trigger.

Summary

In addition to simply being able to send any message out its outlet, the **message** box can be used to construct messages, and to send them to different places.

The *comma* is used to separate different messages within a **message** box, and send them out one after the other. When a message is preceded by a *semicolon* in a **message** box, the first word after the semicolon is the name of a **receive** object, and the rest of the message (or up to the next semicolon) is sent to all **receive** objects with that name, instead of out the **message** box's outlet. The comma and the semicolon enable a **message** box to send many different messages with a single trigger.

The *dollar sign*, followed immediately by a number (such as $1) is a *changeable argument*. When the **message** box receives a triggering message in its inlet, each changeable argument is replaced by the corresponding item from the triggering message. ($1 is replaced by the first item, $2 is replaced by the second item, etc.) If no item is present in the incoming message to replace the value of a changeable argument, the previously stored value is used. If no value has yet been stored in a changeable argument, its value is 0 by default.

A *backslash*, used before a special character such as a comma, a semicolon, or a dollar sign, negates the special characteristics of that character.

A set message can be used to change the contents of a **message** box without triggering any output. An append message can be used to add things to the end of the message in a **message** box.

The **prepend** and **append** objects attach their typed-in arguments to the beginning or end of incoming messages, then send out the combined message.

See Also

append	Append arguments at the end of a message
message	Send any message
prepend	Put one message at the beginning of another
receive	Receive messages, without patch cords
send	Send messages, without patch cords
Arguments	$ and #, changeable arguments to objects
Punctuation	Special characters in objects and messages

Tutorial 26: The **patcher** *Object*

Subpatches

A Patcher program can contain other Patcher programs as *subpatches*. The **patcher** object lets you create a patch within a patch.

A new Patcher window opens when you type **patcher** into an object box. You can edit a patch in the newly opened *subpatch* window, then when you save your *main patch*, the subpatch is saved as part of the same document. If the subpatch window is open when the document is saved, it will be automatically reopened the next time you open the document. The subpatch window can be brought to the foreground at any time by double-clicking on the **patcher** object. You can even nest **patcher** objects; that is, put **patcher** objects within **patcher** objects, within **patcher** objects, etc.

A **patcher** object can be given an argument specifying the name to be shown at the top of the subpatch window. If there is no argument, the window is named *sub patch*. The name is enclosed in brackets to show that it's part of another patch.

This patch contains two **patcher** objects, named *modwheel* and *keyboard*, and their contents are shown in the subpatch windows. For aesthetic reasons we have hidden most of the objects in the subpatches with **Hide On Lock**, but we will examine them shortly.

- Play a few notes on your MIDI keyboard and move the modulation wheel. You will see the **dial** and **kslider** display your actions in the two subpatch windows.

All Windows Active

In computer applications, the front window is the *active window*, where you apply menu commands such as Save and Close, and click and drag on objects. To make a background window active you have to click on it first to bring it to the foreground.

The **All Windows Active** option lets you use background windows without bringing them to the front. To bring any window to the front, click on its title bar or choose its name from the Windows menu. You can also Command-click on any visible part of the window in Macintosh, or Control- click on Windows.

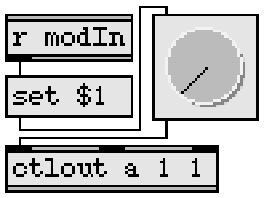
- Check **All Windows Active** in the Options menu. This will let you click and drag on the **dial** and **kslider** in the background windows without bringing the windows to the foreground.

- Drag on the **dial** and the **kslider**. They can send data to the synth as well as display data received *from* your keyboard.

You can close a subpatch window by clicking in its close box, and you can reopen it by double- clicking on the (locked) **patcher** object. Now let's examine the contents of the **patcher** objects.

The modwheel Subpatch

- Bring the *modwheel* window to the foreground and unlock it.

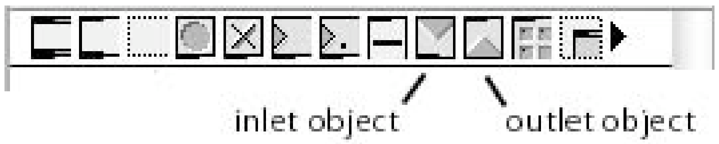


Now you can see the hidden objects. Modulation wheel data received from your MIDI keyboard with **ctlin** is sent to the **r modIn** object in the subpatch. The control data replaces the $1 argument and sets the **dial** without triggering any output (so the data won't be echoed back to the synth). When you change the **dial**, the data is transmitted to the synth with **ctlout**.

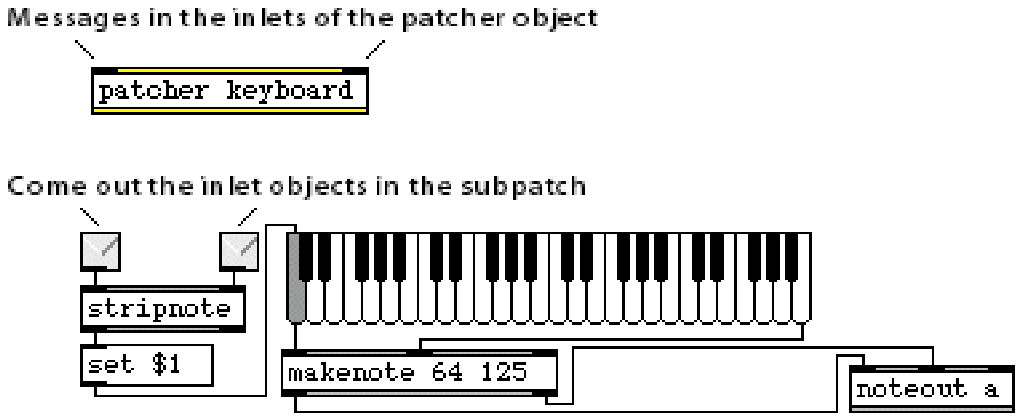
Because they can communicate from one Patcher window to another, **send** and **receive** objects allow you to send messages back and forth between a patch and an embedded subpatch.

inlet and outlet Objects

- Now bring the *keyboard* window to the foreground and unlock it to see the hidden objects. At the top of the subpatch you see two **inlet** objects.



When you include an **inlet** or **outlet** object in a subpatch, a corresponding inlet or outlet is created in the **patcher** object in the main Patcher window. This is usually the most efficient way to send messages to and from a subpatch.


Messages in the inlets of the patcher object
patcher keyboard
Come out the inlet objects in the subpatch
stripnote
set $1
makenote 64 125
noteout a

Assistance

When **Assistance** is checked in the Options menu, Max gives you information about the inlets and outlets of objects while you are editing a patch. Every time you place the mouse on an inlet or an outlet, a brief description of that inlet or outlet is printed in the bottom bar of the Patcher window.

You can give Assistance descriptions to the inlets and outlets of your **patcher** object. To do so, select the **inlet** or **outlet** object in your subpatch and choose **Get Info…** from the Object menu. You can type in a description which will show up as an Assistance message when you are working in the main Patcher window.

- Unlock the main Patcher window and pass the mouse over the inlets of the **patcher** keyboard object to see the Assistance messages.

Although writing Assistance messages to yourself may seem like a waste of time, it can be very helpful in reminding you later what type of message a subpatch object expects to receive in its inlet and what type of message will come out of its outlet.

Summary

The **patcher** object creates a *subpatch* within a patch. The subpatch is saved as part of the document that contains the **patcher** object. If the subpatch window is open when the patch is saved, it will be opened automatically when the document is reopened. You can even *nest* a **patcher** object within another **patcher** object.

Messages can be sent between the main patch and the subpatch with **send** and **receive** objects, or with **inlet** and **outlet** objects. When **inlet** or **outlet** objects are placed in a subpatch, corresponding inlets or outlets are automatically created in the **patcher** object.

When **Assistance** is checked in the Options menu, Max prints a description of inlets and outlets in the bottom bar of the Patcher window while you are editing a patch. You can assign Assistance messages to the inlets and outlets of a **patcher** object by selecting the **inlet** or **outlet** object in the subpatch and choosing **Get Info…** from the Object menu.

When **All Windows Active** is checked in the Options menu, you can click and drag on objects in a background window without first bringing the window to the front.

See Also

inlet	Receive messages from outside a patcher
outlet	Send messages out of a patcher
patcher	Create a subpatch within a patch
Menus	(**Fundamentals**) Explanation of commands

Use Your Patch as an Object

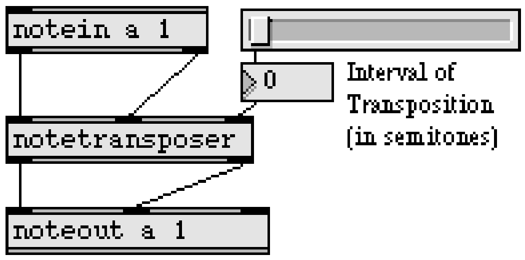
As you get involved with writing your own patches, you will probably find that you are using certain configurations of objects very frequently, or that there are certain computational tasks that you need to do very often. It would be nice if you could just make an object to do that task, then plug in the object wherever necessary.

Actually, any patch you have created and saved can be used as an object in another patch, just by typing the filename of your patch into an object box as if it were an object name. Many Max users refer to patches used in this way as *abstractions*.

As we saw with the **patcher** object, when you use a patch within a patch you usually want to be able to communicate with the subpatch. Therefore, when you are making a patch that you plan to use as an object inside another patch, you will usually want to include **inlet** and **outlet** objects (or **send** and **receive** objects) so that you can send messages to your object and it can send messages out.

The notetransposer Object

In this patch you see a **notetransposer** object, for transposing incoming pitches and sending out the transposed pitch. The interval of transposition (the number of semitones up or down) is supplied in the right inlet.



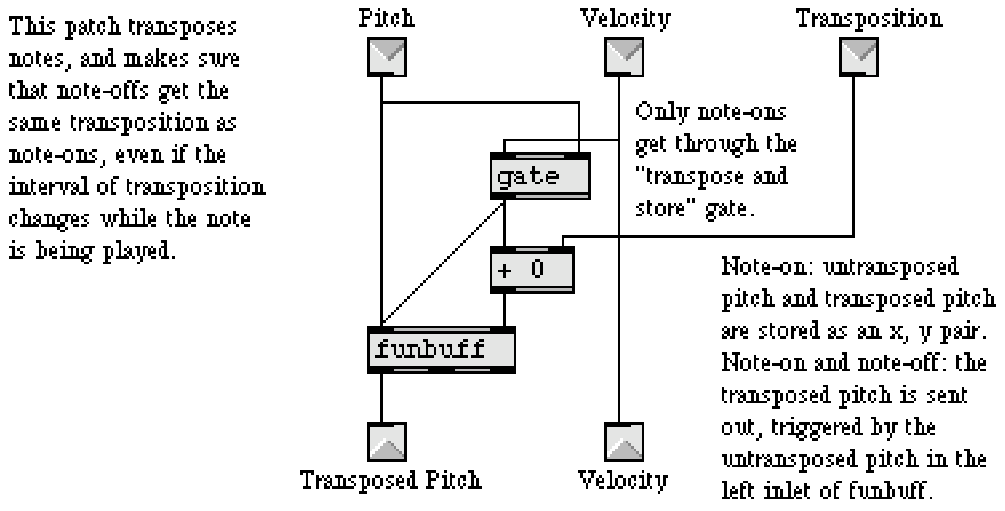
The **notetransposer** is not a built-in Max object. It’s a patch that we created and stored in a file named *notetransposer*.

- Double-click on the **notetransposer** object to see its contents.

In previous patches we have simply sent the pitch to a + or - object to transpose a note. Why do we need a subpatch like this just to transpose notes? The advantage of the

notetransposer over a simple **+** operator is that the **notetransposer** ensures that note-offs are transposed by the same interval as their corresponding note-ons, even if the interval of transposition changes while the note is being held down.

If a note-off message is transposed by a different interval than its note-on was transposed, the note-on will never get turned off and the note will be stuck on the synth. The **notetransposer** solves this problem by keeping a list of the note-ons and their transpositions in an object called **funbuff**, then looking up the transposition when the note-off is played.


This patch transposes notes, and makes sure that note-offs get the same transposition as note-ons, even if the interval of transposition changes while the note is being played.
Pitch
Velocity
Transposition
Only note-ons get through the "transpose and store" gate.
gate
+ 0
funbuff
Note-on: untransposed pitch and transposed pitch are stored as an x, y pair. Note-on and note-off: the transposed pitch is sent out, triggered by the untransposed pitch in the left inlet of funbuff.
Transposed Pitch
Velocity

funbuff

An *array* is an indexed list of numbers. Each number in the list has a unique index number or *address*. We'll call the address *x*, and the value stored at that address *y*. The **funbuff** object stores an array of numbers as *x,y* pairs.

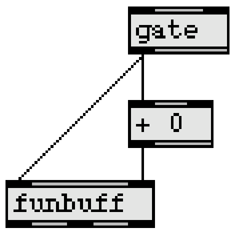
When a number is received in the right inlet followed by a number in the left inlet, the number in the right inlet (*y*) is stored at the address specified in the left inlet (*x*). Then, when an address number is received by itself in the left inlet (*x*), **funbuff** sends the corresponding *y* value out its left outlet.

The numbers can also be stored in **funbuff** as a list: an *x* address and a *y* value. For more information, look under **funbuff** in the Max Reference Manual.

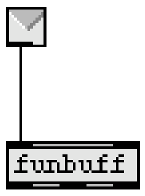
Storing and Recalling Transpositions

The **gate** object in **notetransposer** is used to pass only the pitch of *note-on* messages. Before the pitch reaches the right inlet of **gate**, the velocity value goes to the control inlet of **gate** and either opens it or closes it. If the velocity is 0 (note-off) the **gate** will be closed and the pitch will not get through.

The note-on pitch goes first to the **+** object to be transposed, then to the right inlet of **funbuff**. The untransposed pitch then goes to the left inlet of **funbuff**, so the transposed pitch is stored as the y value, with the untransposed pitch as the address (*x*).



As soon as the *x*,*y* pair is stored, the untransposed pitch (*x*) is sent by itself, causing the transposed pitch (*y*) to be sent out.



When the note-off message comes later, nothing goes through the **gate**, and the untransposed pitch is sent by itself to **funbuff**, causing the transposed pitch to be sent out again. Since the note- off messages get their transposition from **funbuff** rather than from the **+** object, the value in the **+** object can change without affecting the note-off transpositions.

- Close the subpatch window. Play on your MIDI keyboard and drag on the slider at the same time to change the transposition of what you are playing.

Storing transpositions in this manner is essential whenever the interval of transposition is to be changed *while* the notes are being transposed. For example, the transposition might be changed automatically by numbers generated in some other part of the patch.

Differences Between the Patcher Object and Your Object

What are the differences between a subpatch in a **patcher** object and a subpatch you created earlier and saved in a separate file?

One difference is in the way they are saved. The subpatch in a **patcher** object is saved as part of the file that contains the **patcher** object. As a result of this, you can edit a **patcher** object subpatch just by double clicking on the **patcher** object and unlocking the subpatch window. When the subpatch is saved as a separate file, however, you can see its contents by double-clicking on the object, but you can't edit the contents of the subpatch window. (Max will not let you unlock it.) To edit the object, you have to open the separate file in which it was created.

The separate file containing your object must be in a folder where the patch that uses it can find it. Max looks for files in the following places:

- The same folder as the patch that is using the subpatch,
- The same folder as the Max application
- Any other folder you have specified in the File Preferences dialog, under *Look for files in:*. For more information about **File Preferences…**, see the Menus chapter of the Fundamentals manual.

The other main difference is that if you save your patch while the subpatch window of the **patcher** object is open, it will be opened automatically each time you open the main patch. This is not true of a subpatch that is saved as a separate file.

Beware of Recursion

A patch that is used as an object in another patch can itself contain subpatches. For example, our **notetransposer** object could have been written to contain a subpatch object called **splitnote** which separated note-on messages from note-off messages.

A subpatch object may not contain itself, however, since this would put Max into an endless loop of trying to load a patch within itself ad infinitum. For example, our **notetransposer** object could not itself contain a **notetransposer** object, or any subpatch that contains a **notetransposer** object.

Tutorial 27

Documenting Your Object

You can see that the **notetransposer** object has been copiously commented, and all of its inlets and outlets have been given Assistance messages. Such thorough documentation makes it more likely that others will understand your patch and be able to use it, and also helps to remind you how your patch works. Note: If your comments are extensive and you want to include carriage returns in your comment text, use the Inspector to set two-byte compatibility mode for the **comment** box.

Summary

Any patch you create and save can be used as an object in another patch. When you are making a patch that will be used as a subpatch in another patch, you will usually want to include **inlet** and **outlet** objects (or **send** and **receive** objects) so that you can send messages to your object and it can send messages out.

The **funbuff** object stores an *array* of numbers: *x*,*y* pairs of *addresses* and *values*. When an address number (*x*) is received in the left inlet, the value stored at that address (*y*) is sent out the left outlet. This type of array is useful as a lookup table, for storing values in an indexed list and looking them up later. One use of arrays is to pair note-on pitches with their transposition so that the transposition can be looked up again when the corresponding note-off is played.

The window of a subpatch object that is saved as a separate file is not opened automatically when the Patcher window that contains it is opened (unlike the **patcher** object). A patch that was saved as a file and used as a subpatch object can be edited only by opening the file in which it is saved.

Explanatory notes in the form of **comment** boxes and Assistance messages are helpful to you and to others who may use your patch.

See Also

funbuff	Store *x*,*y* pairs of numbers together
Encapsulation	(**Max Topics**) How much should a patch do?

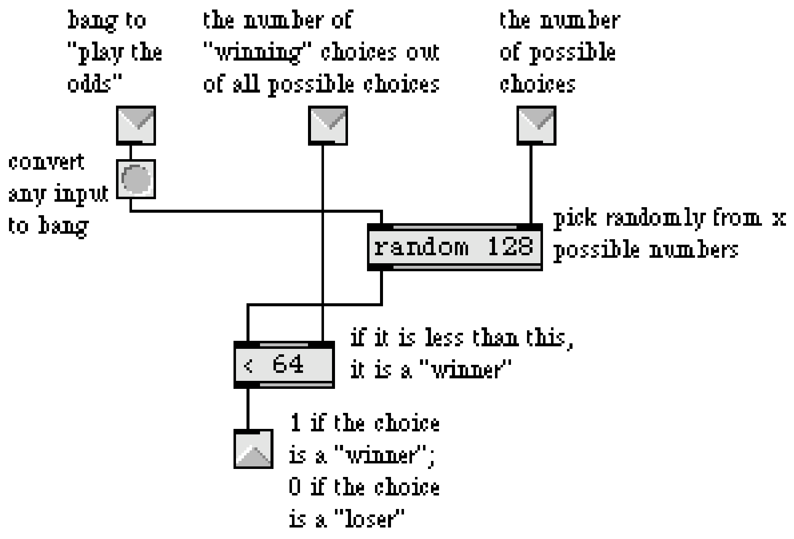
Supplying Initial Values to Your Abstraction

Many Max objects take *arguments*, typed in after the object name, to supply some information to the object such as a starting value. You can design your own object to get information from typed- in arguments, too.

The gamble Object

In the Patcher window you can see several instances of an object called **gamble**. It's not a Max object; it's a patch we created and saved in a document named *gamble*.

- Double-click on the **gamble** 64 128 object in the right part of the Patcher window to see the contents of the subpatch.


bang to "play the odds"
the number of "winning" choices out of all possible choices
the number of possible choices
convert any input to bang
random 128
pick randomly from x possible numbers
< 64
if it is less than this, it is a "winner"
1 if the choice is a "winner"; 0 if the choice is a "loser"

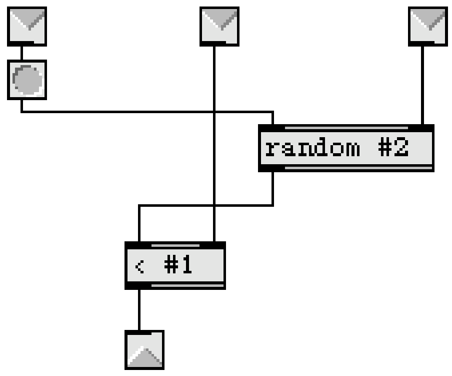
The **gamble** object functions as an electronic gaming table. When it receives a bang in its left inlet (or anything else, since the **button** inside **gamble** converts all incoming messages to bang), it chooses a random number (limited by the 2nd argument or the number received in the right inlet). If the random number is less than a certain other number (specified by the 1st argument or received in the middle inlet), **gamble** sends out 1. Otherwise **gamble** sends out 0.

In effect, the arguments to **gamble** state the odds of a 1 being output each time a bang is received in the left inlet. In this case the odds are 64 in 128 (even up).

- Close the subpatch window, and double-click on the **gamble** 2 5 object to see the contents of the subpatch. The odds are different in this subpatch, because the arguments are different.

The # argument

- Now open the document named *gamble* in the Max Tutorial folder. You can see that in the original *gamble* patch, the odds are specified with *changeable arguments*.



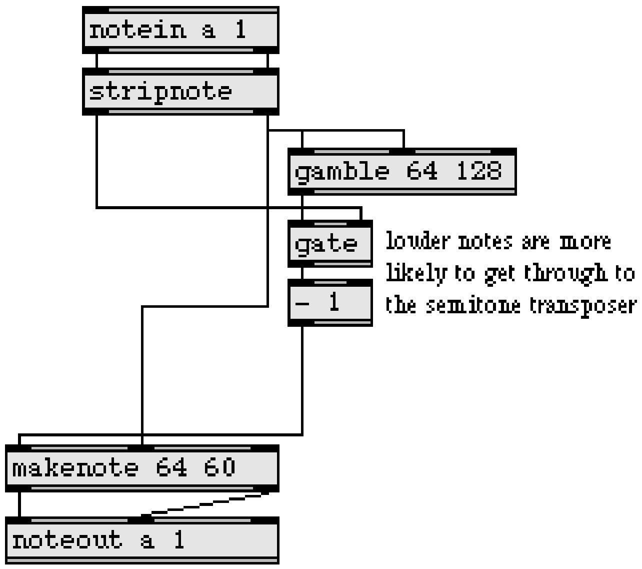
When the *gamble* patch is used as a subpatch in other patches, the changeable #1 and #2 arguments are replaced by the 1st and 2nd arguments typed into the **gamble** object. If no argument is typed in, the # arguments are replaced by 0.

The # argument can be used with most Max objects inside your object, and can be replaced by a symbol as well as a number. For examples of its usage, look in the *Arguments* section of this manual.

Using Weighted Randomness

Now that you have seen how arguments are used to set initial values for a subpatch object, let's see how **gamble** is actually used in this patch. Each time **gamble** receives a bang in its left inlet, it makes a *probabilistic decision* whether to send out 1 or 0, depending on the specified odds.

In the right portion of the Patcher window **gamble** is used to decide whether to open or shut a **gate**.


notein a 1
stripnote
gamble 64 128
gate
- 1
louder notes are more likely to get through to the semitone transposer
makenote 64 60
noteout a 1

The velocity of each played note sets the odds of the **gate** being opened, then **gamble** is triggered to open or shut the **gate** based on those odds. If the **gate** is open, the pitch will get through and will be transposed down a semitone and transmitted back to the synth.

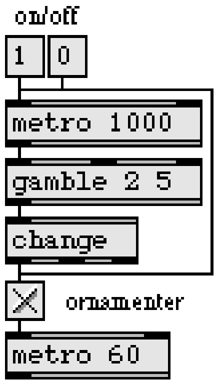
Let's say you play a note with a velocity of 93. The odds of the gate being open are 93 in 128, a little less than 3 in 4, so it is likely that the note you play will be transposed. If you play a note with velocity of 3, however, the odds are only 3 in 128 of the **gate** being open, so the note will probably not be transposed.

The result is a probabilistic "Thelonius Monk effect" of adding lower grace notes to more and more pitches as the velocity increases. Notice that we don't need to use the **notetransposer** object shown in the previous chapter, because we are transposing only note-on pitches and **makenote** provides the note-offs.

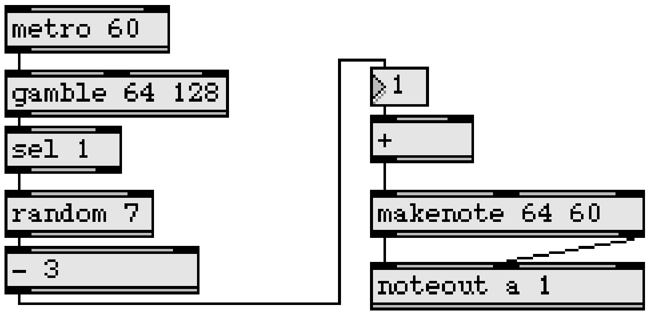
- Play with different extreme velocities on your MIDI keyboard and notice the difference in likelihood of a grace note being added to what you play.

Tutorial 28

In the left side of the Patcher window **gamble** is again used to make *weighted random decisions*, with two slightly different implementations. When the **metro 1000** object is turned on, it triggers **gamble** every second, and **gamble** turns the **metro 60** object on or off (it will be turned on approximately 40% of the time).



Every 60ms the lower **gamble** will send out either a 1 or a 0, with the odds depending again on the velocity of the played note. When it is 1, **sel** triggers **random** to choose a random ornamentation interval which is added to the played note and transmitted to the synth.



A little bit of additional calculation is performed to make the range of the ornamentation interval also depend on the played velocity. When the velocity is at a maximum, the range of the ornamentation will vary from -7 to 7 semitones (up or down as much as a perfect fifth). When the velocity is at a minimum, the ornamentation will only be 0 (unison).

For example, when the velocity is 127, a random number is chosen between 0 and ((127+8)÷9)- 1, that is, 14. That number will then have ((127+8)÷9)÷2) subtracted from it, i.e. 7, setting the range of possible ornamentations from -7 to 7.

- Turn the **metro** 1000 object on, and play on your MIDI keyboard with extreme changes of pitch and velocity. Notice that the ornamentation is wider and more dense when you play harder. The effects of the ornamenter are most comprehensible when you play very sparsely on the keyboard.

When to Use Arguments

The reason for supplying values to an object is to modify some characteristic of the subpatch. If you always want the subpatch to do exactly the same thing, you probably don't need to change the values inside it in any way. If, however, you want your object to do something slightly differently depending on some value it receives, the value will have to be supplied using an inlet or a typed-in argument.

There's no hard and fast rule about when to supply values to a subpatch by using arguments, and when to supply values via inlets. Generally speaking, if you will just want to supply the value once it can be most easily given as an argument, but if you want to change the value of a single object often you will need to use an inlet.

One solution is to make both ways possible, as we have done with **gamble**. The arguments are used to set initial values inside the subpatch, but the values can be changed by numbers received in the middle and right outlets.

Summary

You can enable your object to accept information from typed-in arguments by including changeable # arguments in the subpatch. A changeable argument of #1 in the subpatch is replaced by the first typed-in argument in the object box, #2 is replaced by the second argument, and so on. If no argument is typed into the object box, the changeable argument is set to 0.

Your patch can make *weighted random* (*probabilistic*) decisions by choosing a random number, then testing to see if the number meets certain conditions.

See Also

Arguments	$ and #, changeable arguments to objects

Tutorial 29: Test 5—Probability Abstraction

Create Your Own Abstraction

This is an exercise in the creation and use of your own abstraction—one object that passes on a certain percentage of the bang messages it receives, then use that object in a patch. First we must create the object.

1. Create an object called **passpct** that receives bang messages in its left inlet and passes a certain percentage of them out its outlet. The percentage should be specified by a typed-in argument or by a number received in the right inlet.

Hints

A *percentage* of probability is the number of times an event is likely to occur in 100 tries. For example, a 33% chance means the odds are 33 in 100 that the event will occur.

Use the **gamble** object from the previous chapter as a model to give you an idea how to proceed. The **passpct** object will be similar except:

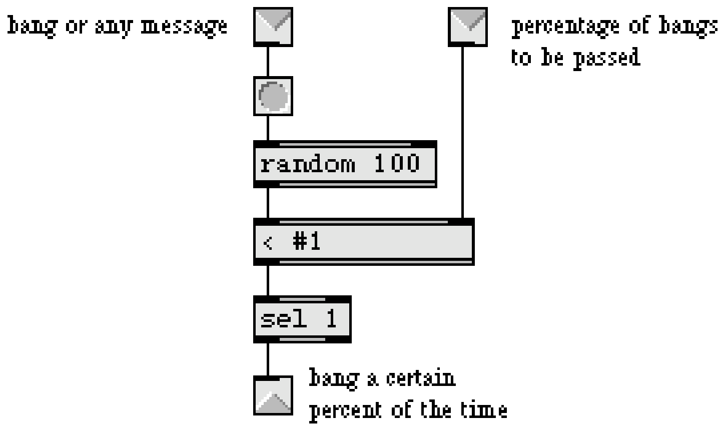
a) The number of possible random choices will always be 100.

b) Instead of sending out a 1 or a 0, you want your object to send out bang (whenever the condition is met).

Solution to Exercise 1

We have saved our solution to Exercise 1 in a file called *PassPct* in the Max Tutorial folder.

When a bang or any other message is received in the left inlet, the **random 100** object chooses a number from 0 to 99. If it is less than the number specified as an argument (or received in the right inlet), **sel** sends a bang out the outlet.


bang or any message
percentage of bangs to be passed
random 100
< #1
sel 1
bang a certain percent of the time

Next we will use the **PassPct** object in a patch to make probabilistic decisions.

Pass a Percentage of bang messages from a metro

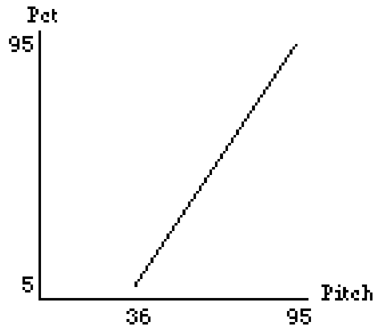
2. Use a **metro** to send bang messages at a constant speed and use **PassPct** to pass only a certain percentage of those bang messages. Use the bang messages passed by **PassPct** to trigger notes sent to the synth. Use a 5-octave **kslider** to choose which pitch will be transmitted.

   Make the percentage value of **PassPct** depend directly on the pitch selected with **kslider**. As the pitch increases from 36 to 95, the percentage should increase from 5 to 95.

Hints

The pitch value sent out by **kslider** should be stored in some type of storage object (an **int**, a **value**, a **number box**, etc.—an **int** is the most efficient). The bang messages from **PassPct** can then trigger the storage object to send its number to **makenote** and play a note.

The hard part of this exercise is using the range of pitches sent out by **kslider** (from 36 to 95) to provide a different range of percentages (from 5 to 95) to **PassPct**. This is known as mapping one range to another. A direct correspondence such as this is a *linear* map: the relationship between the two ranges can be graphed as a straight line.



As the pitch changes from 36 to 95, the percentage changes from 5 to 95

Calculating a Linear Map

The problem of linear mapping is: given one range of numbers from *xmin* to *xmax* and another range of numbers from *ymin* to *ymax*, and given some number x within the first range, find the number y that occupies the same position in the second range.

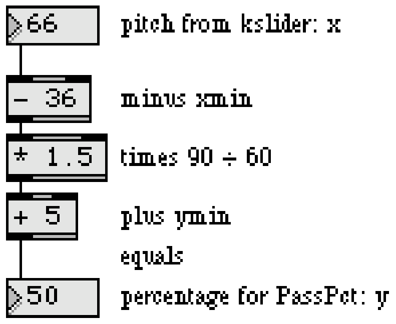
Here's a formula for finding the *y* value that corresponds to any given *x* value.

$$y = ((x - xmin) * (ymax - ymin) \div (xmax - xmin)) + ymin$$

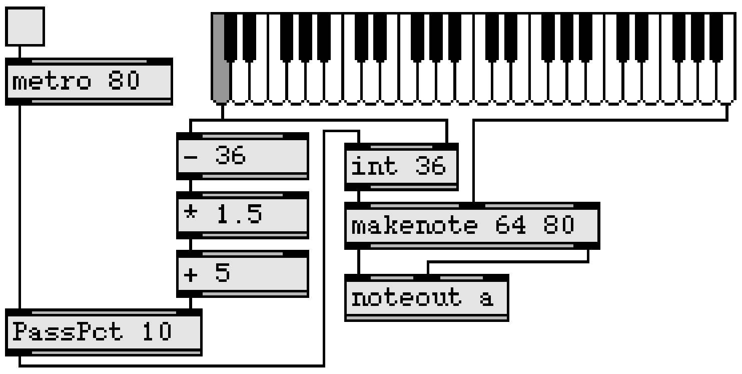
When we plug our ranges into the formula, we get

$$y = ((x - 36) * 90 \div 60) + 5$$

How do we translate this into objects?



The pitch from **kslider** is sent into the formula, and the percentage is sent to **PassPct**. Your patch might look something like this.



Use PassPct to make Random Choices

Let's add one more element to the exercise.

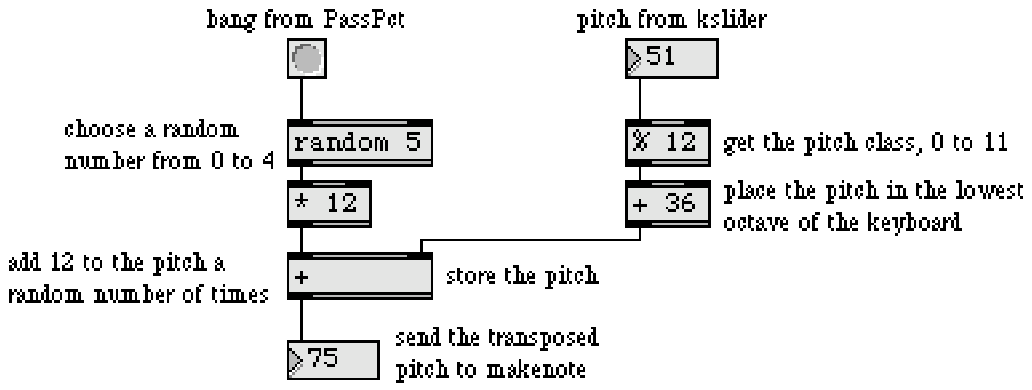
3. Add another **PassPct** object that receives **bang** messages from the same **metro**, but triggers random octave transpositions of the selected pitch. Make the percentage of this **PassPct** object *decrease* from 95 to 5 as the selected pitch increases.

This part of the exercise presents two new problems: how to create random octave transpositions of a pitch, and how to express an inverse linear relationship. Try to find a solution to these problems yourself before reading further.

Random Octave Transpositions of the Pitch

To make a random octave transposition of a note, you need to calculate the *pitch class* of the note (C, C#, D, etc.), then add 12 to that pitch class some random number of times.

You will want to limit the random numbers so that they keep the transpositions within the range of the keyboard. The solution might look something like this:


bang from PassPct
pitch from kslider
51
choose a random number from 0 to 4
random 5
% 12
get the pitch class, 0 to 11
* 12
+ 36
place the pitch in the lowest octave of the keyboard
add 12 to the pitch a random number of times
+
store the pitch
75
send the transposed pitch to makenote

Calculating an Inverse Linear Map

You may remember from an earlier patch in which we inverted pitches that we subtracted the maximum possible pitch from whatever pitch was played, then took the absolute value of the result. (See Patch 2 in Tutorial 14.)

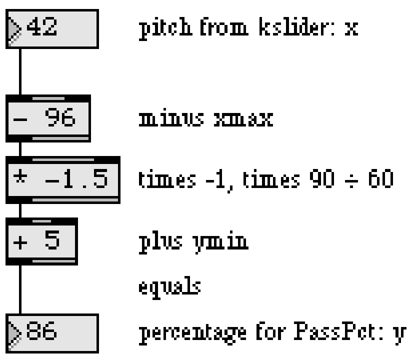
The formula for an *inverse* linear map, then, looks like this:

$$y = (-(x - xmax) * (ymax - ymin) \div (xmax - xmin)) + ymin$$

When we plug our ranges into the formula, we get

$$y = (-(x - 96) * 90 \div 60) + 5$$

We can translate this into Patcher objects as



Scroll to the right in the Patcher window to see Exercise 2 and Exercise 3 combined in a single patch. (We've used our **PassPct** object).

Tutorial 29

Summary

To create your own object, make a patch that includes **inlet** and **outlet** objects (and changeable **#** arguments if appropriate), save the patch, then use your object by typing the name of the file into an object box in some other patch.

The **PassPct** object is similar to the **gamble** object from the previous chapter. It passes or suppresses the bang messages it receives, according to some percentage of probability.

You can transpose a note by an arbitrary number of octaves by first calculating its *pitch class* (with a **%** 12 object), and then adding some multiple of 12 to the pitch class.

You can create a direct or inverse linear relationship between two ranges of numbers using the *linear mapping* procedure described in this chapter.

Tutorial 30: Number Groups

Use of Lists

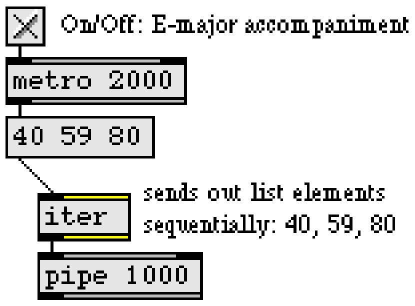
We have seen that a message can consist of a single number or a *list* of numbers separated by spaces. The list is an effective way of sending numbers together, ensuring that they are received at the same time by an object.

For example, we usually want to keep pitch values and velocity values synchronized so that they are received in the proper order by **noteout**. When **noteout** receives a list in its left inlet, it interprets the third element (if present) as the channel number, the second element as the velocity, and the first element as the pitch.

There are objects specifically for combining numbers into a list, and objects for breaking lists up into individual numbers. So, you can choose the most appropriate way to send groups of numbers between objects. A list even can include symbols (words) as well as numbers, which may be useful in some cases. As long as the first element is a number, Max objects will recognize the message as a *list*.

iter

When the **iter** object receives a list of numbers in its inlet, it breaks the list up into its individual elements and sends the numbers out in sequential order rather than all at the same time. It's as if **iter** puts commas between the elements, to make them into separate messages.


On/Off: E-major accompaniment
metro 2000
40 59 80
iter
sends out list elements sequentially: 40, 59, 80
pipe 1000

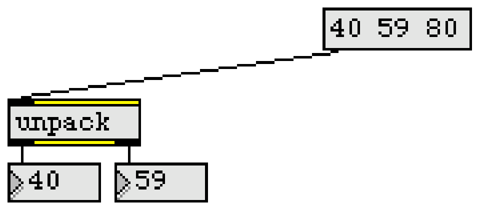
In the right part of the Patcher window you can see **iter** at work. When the **metro** triggers the list of numbers, it is sent to **iter**, which breaks up the list and sends each of the numbers on in order, as rapidly as possible. The numbers are delayed by the **pipe**, then are sent on as (virtually simultaneous) pitches to **makenote**.

unpack

When a list is received by **unpack**, each element of the list is sent out a different outlet. The number of outlets **unpack** has is determined by the number of arguments you type in. (The arguments also set an initial value for each outlet.) If there are no typed-in arguments, **unpack** has two outlets, both with an initial value of 0.

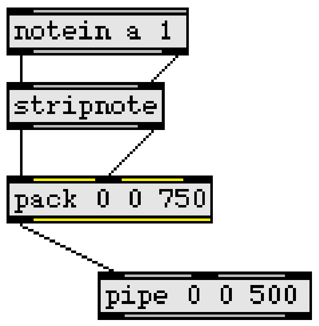
If there are more items in the incoming list than **unpack** has outlets for, the extra items are ignored. If a list is received that has fewer items than there are outlets, **unpack** sends those items out the appropriate outlets but sends nothing out the remaining outlets.

In the example patch, when a list is received by **unpack**, the second item in the list is sent out the right outlet, then the first item in the list is sent out the left outlet (output order is always right to left).



pack

The **pack** object combines separate items into a list. It stores the message most recently received in each of its inlets, then when it receives a message in the *left* inlet it sends out all the stored items together as a list. The number of inlets—and the initial value stored in each one—is specified by the typed-in arguments.

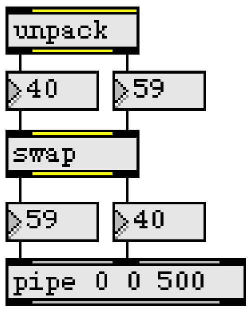


In the left part of the Patcher window, note-on pitch and velocity values from your MIDI keyboard are packed in a list along with the number 750, and the list of pitch-velocity-delay is sent to the **pipe**. Every note from the keyboard will be delayed 750ms, even if the delay time of the **pipe** is changed by some other part of the patch, because the delay time is sent in the same list as the note- on data.

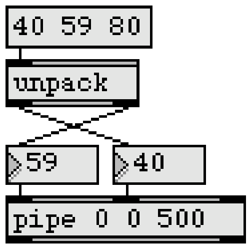
swap

The **swap** object reverses the sequential order of numbers it receives. It is triggered by a number in its left inlet, just like other objects, but it sends that number out its right outlet *first*, then sends the number that was received earlier in the *right* inlet out its left outlet.

In the example patch, **swap** reverses the order of the first two list items, received from **unpack**, and uses the first number in the list, 40, as a velocity and the second number, 59, as a pitch.



It would not be sufficient just to cross the patch cords from **unpack**, because the number 59 would arrive at the left inlet of **pipe** and trigger it before the number 40 got there.



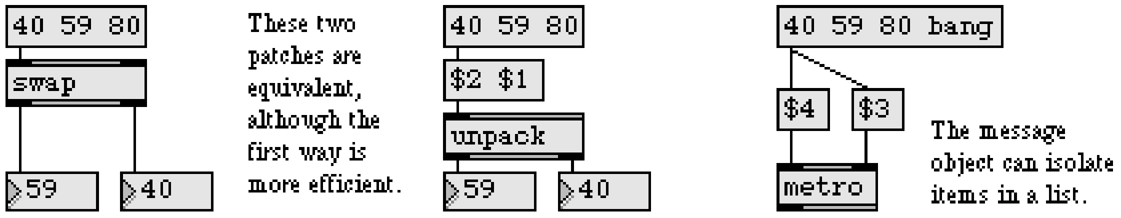
This patch is not equivalent to the one shown above.

When **swap** receives a bang in its left inlet it sends out the same numbers again. The two numbers to be swapped can also be received in the left inlet as a list. In fact, the **unpack** object in this patch is not strictly necessary, because **swap** would understand the list and swap the first two items, but we included **unpack** to make the number-swapping more evident visually. There is also an object for swapping floats, called **fswap**, not demonstrated here.

Tutorial 30

Lists Can Be Managed with Message Boxes

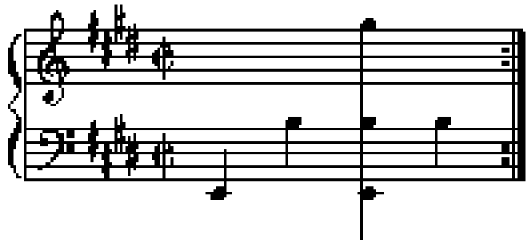
As was shown in Tutorial 25, a **message** box can also be used to isolate and rearrange items in a list. Here are a couple of examples showing possible uses of **message** boxes for selecting individual items from a list.


40 59 80
swap
59
40
These two patches are equivalent, although the first way is more efficient.
40 59 80
$2 $1
unpack
59
40
40 59 80 bang
$4
$3
metro
The message object can isolate items in a list.

An Automatic Accompanist

Now that we have seen how the list management objects work, let's see how they are used in the example patch. Elements of the list 40 59 80 are rearranged and delayed in different ways to send different messages to **makenote** at different times.

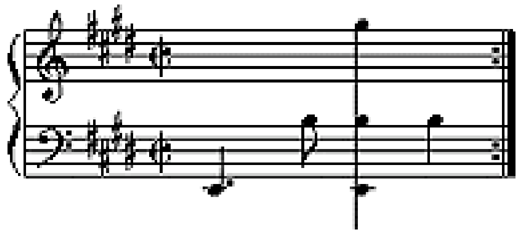
When the **metro** is turned on, the entire list is sent to **makenote** immediately, playing the note 40 (E1) with a velocity of 59 and a duration of 80ms. The pitch and velocity are reversed by **swap**, and delayed 500ms before being sent to **makenote**, playing the note 59 (B2) with a velocity of 40. One second after the **metro** was turned on, the numbers are all sent to **makenote** as a chord—E1, B2, and G#4—with the velocity of 40 from the previous note. At the same time, the bang that was delayed by the **del** object retriggers the note B2 from **swap**, and it is delayed another 500ms before being played. After a total of 2 seconds, the entire process is repeated. The result sounds like this:



Automatic E-major accompaniment figure

Note-on pitches and velocities from your MIDI keyboard are packed into a list along with a delay time and sent to **pipe** with a delay of 750ms. This causes a short-note echo of every played note 750ms later.

The played notes also have an effect on the accompaniment. If a played note arrives at **pipe** in between the first and second notes of the accompaniment figure, the delay of the second note of the accompaniment will be 750ms, causing this rhythmic change:



Also, if a delayed played note reaches **makenote** between the second and third notes of the accompaniment, the velocity of the chord will be altered.

- Turn on the accompaniment and play a melody along with it.

Summary

A *list* is any message that begins with a number and contains additional items as *arguments*. Usually the arguments are all numbers, but they may also be symbols.

Sending numbers together as a list ensures that they will be received together. Many objects, such as **pipe**, **makenote**, and **noteout**, interpret a list of numbers received in the left inlet as if the numbers had been received separately in different inlets.

The **pack** object combines the messages it receives in each inlet into a single list. The **unpack** object breaks a list up into its individual items, and sends each item out a different outlet, in order from right to left. The **iter** object sends each number of a list individually, in order from left to right, out a single outlet.

The changeable **$** argument in a **message** box can be used to isolate individual elements of a list. This is especially effective if the list contains symbols in addition to numbers.

The **swap** object reverses the sequential order of two numbers. When a number is received in the left inlet, it is sent out the right outlet, then the number that was received earlier in the right inlet is sent out the left outlet.

See Also

buddy	Synchronize arriving numbers, output them together
fswap	Reverse the sequential order of two decimal numbers

Tutorial 30

iter	Break a list up into a series of numbers
pack	Combine numbers and symbols into a list
swap	Reverse the sequential order of two numbers
thresh	Combine numbers into a list, when received close together
unpack	Break a list up into individual messages

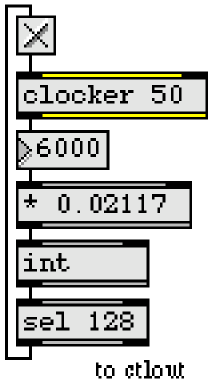
Timed Processes

So far we have used two different timing objects: **metro** for sending a bang at regular intervals, and **timer** for reporting the elapsed time between two events. In this chapter we introduce some objects for producing timed progressions of numbers.

clocker

The **clocker** object is the same as **metro**, except that instead of sending out bang at regular intervals it sends out the time elapsed since it was turned on. With this information you can cause values to change in some manner correlated with the passing of time.

In the part of the Patcher window marked A, a **clocker** reports the elapsed time, and that information is mapped to send increasing values to the mod wheel of the synth. Over the course of 6 seconds the time progresses from 0 to 6000, causing the control values to increase from 0 to 127. When the value reaches 128, the **clocker** is turned off by **sel**. The result is a 6-second linear *fade-in* of the modulation effect on the synth.

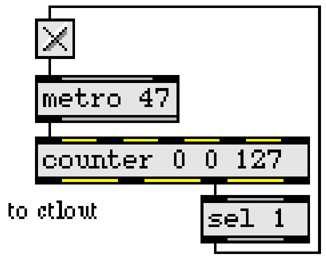


The **int** object is included to truncate the float output of the * object so that **sel** will make an accurate comparison.

counter

The **counter** is not itself a timing object, but it is frequently used in conjunction with **metro**, because **counter** counts the number of bang messages it has received. The **metro**-**counter** combination is an effective way to increment or decrement a value repeatedly.

In the part of the Patcher window marked B, the first argument to **counter** specifies the direction of the count: 0 for upward. (1 is for downward, and 2 is to go back and forth between up and down.) The second argument sets the minimum value of the count, and the third argument sets the maximum value.



Note: The meaning of the arguments to **counter** changes depending on how many arguments there are. Look under **counter** in the Max Reference Manual for details.

The count is sent out the left outlet. When the maximum (127) is reached, **counter** sends a 1 out its right-middle outlet. This 1 is detected by **sel**, which toggles the **metro** off. This is another way to get the same effect as we did using **clocker**. With **counter**, however, the numbers can be easily placed in the desired range (0 to 127 in this case) without a multiplication being performed each time. Multiplication takes longer for a computer to perform than incrementing a count.

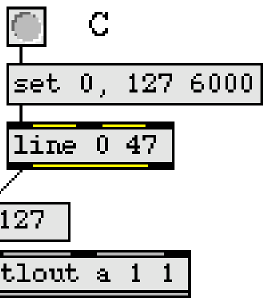
The **metro** is set to a speed of 47ms so that the progression from 0 to 127 will be completed in 5.969 seconds—as close as possible to 6 seconds (using this method).

line

The **line** object also outputs numbers in a linear ramp from some starting value to some ending value over a specific period of time. The first argument sets the starting value and the second argument sets the *grain*—the time interval at which numbers will be sent out. When a time period is received in its middle inlet and an ending value is received in its left inlet, **line** outputs numbers in a linear progression from the starting value to the ending value over the specified time period.

The numbers in the inlets can also be received together as a *list* in the left inlet. If a number is received by itself in the left inlet, without a time period being received at the same time, **line** jumps to (and outputs) the new value immediately.

A starting value can be sent to **line** without triggering any output by sending it a set message (the word set, followed by a number).



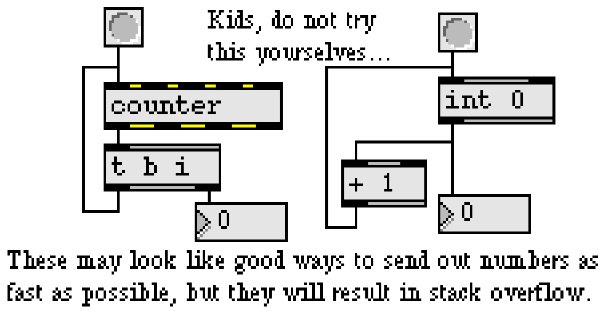
Set starting value to 0, then progress to 127 in 6 seconds, outputting a number every 47ms

Stack Overflow

Have you ever been in the position of feeling like the list of things you have to do is growing faster than you can get them done? Well, it's possible to overload Max in a similar way, so that the list of things Max has to do eventually overflows the amount of memory space available for its *stack* of things to do. This is known as a *stack overflow,* and it causes Max to shut down its internal scheduler and stop performing timed operations until you fix whatever is causing the overflow.

One way to cause a stack overflow is to feed an object's output back into its input. For example, when you want to increment numbers as fast as possible, you might be tempted to feed the output of an object like **counter** right back into itself, repeatedly incrementing the count. But such automatic repetitions must be separated by at least a millisecond or two, otherwise Max will generate repetitions too fast for itself to keep track of, and you will get a *Stack Overflow* error dialog. When this happens, you must choose **Resume** from the Edit menu to restart Max's scheduler.

- Patches D and E show two examples of situations that result in stack overflow. Click on the buttons if you want to make Max very unhappy. (Go ahead, you won't break anything.) Remember to choose the **Resume** command to start Max up again.


Kids, do not try
this yourselves...
counter
t b i
0
int 0
+ 1
0
These may look like good ways to send out numbers as
fast as possible, but they will result in stack overflow.

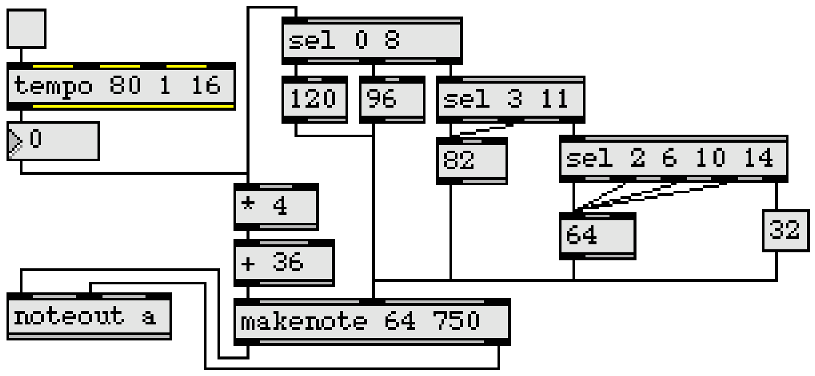
tempo

The **tempo** object is another metronome, but it operates in somewhat more traditional musical terms than the millisecond specifications necessary with **clocker**, **metro**, and **line**. The first argument to **tempo** (or a number received in the left-middle inlet) sets a metronomic tempo in terms of *beats per minute*—that is, quarter notes per minute—just like a traditional metronome.

The second and third arguments (or numbers supplied in the right-middle and right inlets) specify what fraction of a whole note **tempo** will use to send out ticks of the metronome. For example, if the second and third arguments are 1 and 16, the fraction is $^1/_{16}$ of a whole note and **tempo** sends out a number from 0 to 15 for every sixteenth note, based on the specified quarter note tempo. A fraction of $^2/_3$ would send out half note triplet ticks (a tick every $^2/_3$ of a whole note), and so on.

The numbers sent out by **tempo** always go from 0 to the number 1 less than the pulse division (the third argument). The greatest allowable division is 96 (sixty-fourth note triplets). The fact that **tempo** sends out a number (a sort of pulse index), lets you assign different things to happen on different pulses in a measure. In this way you can generate metrically-based automated processes.

In Patch F, **tempo** sends out a number for each sixteenth note at a tempo of 80, and triggers a different pitch and velocity for each pulse of the measure. The pitch ascends in an arpeggiated augmented triad, and the velocities are greater on the strong beats of the 4/4 measure, and smaller on the weaker pulses.



- Select a velocity-sensitive sound on your synth and turn on the **tempo** object.

External Timing

The **metro**, **line**, **clocker**, and **tempo** objects can be synchronized to some timing source other than Max's internal millisecond timer, such as a time-code generator, an external sequencer, or even some other software sequencer. For details, look under those object names, as well as the **setclock** object, in the *Objects* section of this manual.

Summary

When **clocker** is turned on it sends out the elapsed time at regular intervals. The time value can be mapped to other ranges to make them depend on the passing of time.

The **counter** keeps track of how many bang messages it has received and sends out the count. The count can be restricted to a specific range, and the bang messages can be supplied repeatedly by a **metro** to increment and/or decrement the **counter** at a specific speed. This is another way of creating a particular progression of numbers over time.

The **line** object is a third way of generating a linear progression of numbers. **line** outputs numbers in a ramp from some starting value to some ending value, arriving at the new value in a specific amount of time.

Tutorial 31

Incrementing numbers by means of recursive loops, without some type of delay between repetitions, can result in a *stack overflow* error, which causes Max to stop its internal scheduler. Choosing **Resume** from the Edit menu restarts the scheduler.

The **tempo** object is a metronome that lets you specify timing in traditional musical terms of beats per minute and beat divisions. It sends out a different number for every pulse in a measure, so each pulse number can trigger a different action.

The **metro**, **line**, **clocker**, and **tempo** objects can be synchronized to an external timing source such as a sequencer or a time-code generator.

See Also

clocker	Output the elapsed time, at regular intervals
counter	Count the bang messages received, output the count
line	Output numbers in a ramp from one value to another
metro	Send a **bang**, at regular intervals
setclock	Control the clock speed of timing objects remotely
tempo	Output numbers at a metronomic tempo
timein	Report time from external time code source

An Indexed Array of Numbers

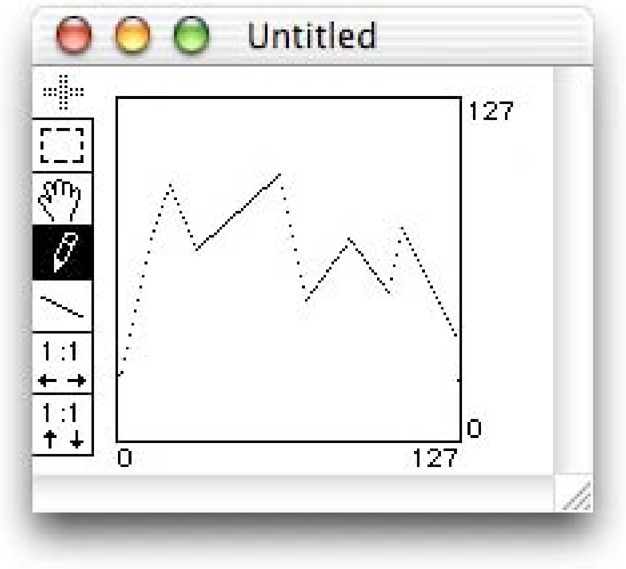
In Tutorial 27 we introduced the **funbuff** object for storing an indexed array of numbers. Number values are stored with an index number (address), then when you want to recall a value you just specify the address where it is stored. The **table** object stores and recalls numbers similarly, but has many more features.

Graphic Editing

The most notable feature of the **table** object is that it allows you view and edit the stored numbers in a graphic editing window.

- Check **All Windows Active** in the Options menu so that you can view **table** objects and click in the Patcher window at the same time.

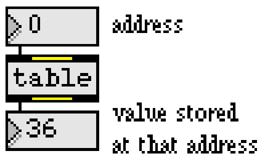
- Double-click on the **table** object at the bottom of Patch 1. A Table window will open to show a graph of some numbers that we have stored there.



This **table** contains 128 numbers, with addresses from 0 to 127. Addresses always go from 0 to the number 1 less than the size of the **table**. This **table** shows a range of possible *values* from 0 to 127, and the values we have stored range from 36 to 96.

- Turn on the **metro** at the top of Patch 1. The **counter** counts up and down between 0 and 127. The numbers are sent through a **uslider** just to show their progression graphically, then they are sent to the left inlet of **table**.

A number by itself in the left inlet of **table** specifies an address, and the value stored at that address is sent out the left outlet. The output of the **table** is displayed graphically by the second **uslider**. You can see and hear the numbers in the **table** as **counter** steps through them.



Get Info…

- With the *Untitled* table editing window still in the foreground, choose **Get Info…** from the Object menu to open the table Inspector.

The table Inspector shows you the *Size* of the **table** (the number of storage addresses) and the *Range* of displayed values. It also has two options for viewing the numbers. Checking *Signed* causes the Table window to display negative values as well as positive, and checking *Note Name Legend* shows the *y* axis values as MIDI note names instead of numbers.

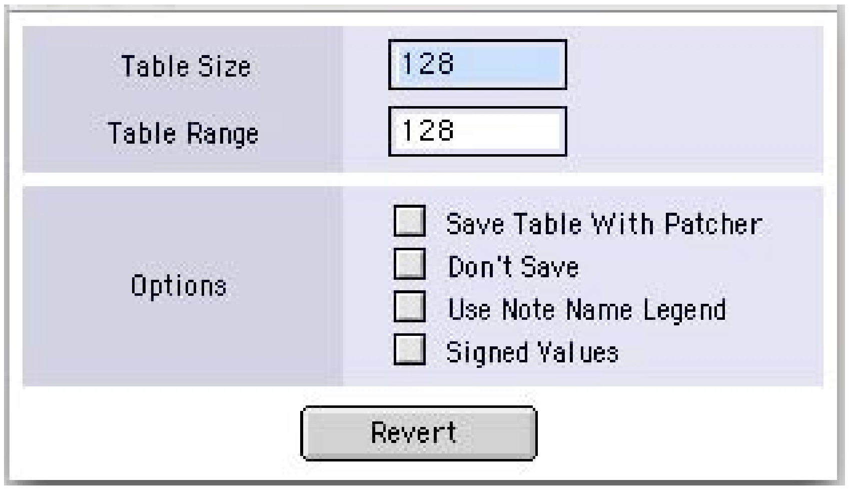


Table Size: 128
Table Range: 128
Options:
☐ Save Table With Patcher
☐ Don't Save
☐ Use Note Name Legend
☐ Signed Values

Saving the Values in a table

It's important to understand the different options for saving a **table**, so you don't lose numbers you've carefully entered. Normally, the values you store in a **table** object are lost when you close the Patcher window. If you check *Save Table with Patcher*, however, the numbers in the **table** will be saved as part of your Patcher document. Then, when you reopen the patch the **table** will still contain the numbers. We have checked *Save Table with Patcher* for this **table** so that our masterpiece will be preserved.

If you change the values in a **table**, Max will ask you if you want to save the changes you made to that **table** when you close the Patcher window. If you don't want Max to ask you that every time you close the Patcher window, check *Don't Save*. *Don't Save* does not cause you to lose any values you have explicitly saved with *Save with Patcher*; it just doesn't remind you to save any subsequent *changes*.

Any time the table editing window is in the foreground you can save its contents to a separate file by choosing **Save** from the Max menu. Then, whenever you want to use the file in a patch, just create a **table** object and type in that file name as an argument. The contents of the file will be loaded into the **table** object.

An example of a **table** that's stored as a separate file can be seen in Patch 3. The file *Cmajor.t* is loaded into the **table** object whenever the Patcher window is opened. You might want to give your Table files names that include some distinguishing characteristic, such as *.t*, so that you can tell Table files and Patcher files apart.

- Double-click on the **table** Cmajor.t object to see its contents. With the *Cmajor.t* table editing window in the front, choose **Get Info…** from the Object menu to open the table Inspector.

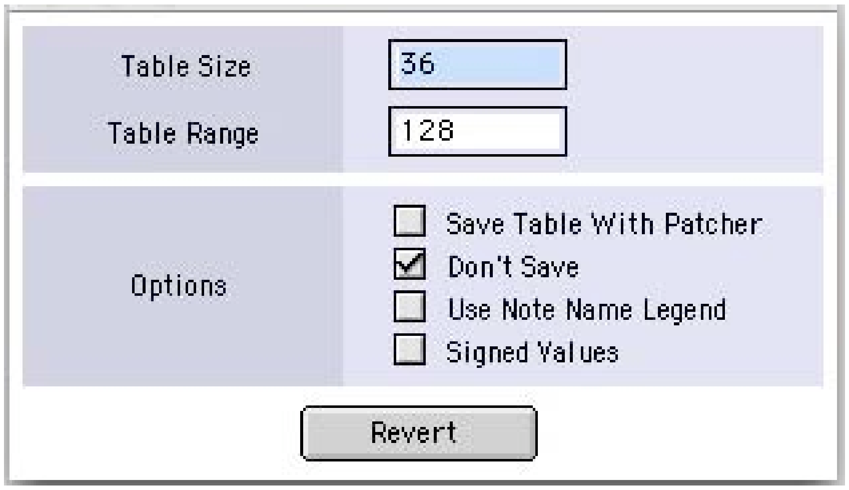
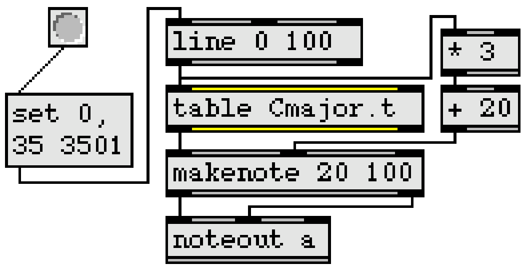


Table Size: 36
Table Range: 128
Options:
- [ ] Save Table With Patcher
- [✓] Don't Save
- [ ] Use Note Name Legend
- [ ] Signed Values

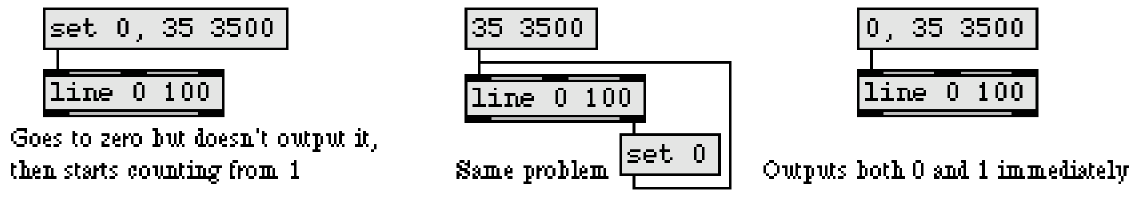
You can see that the Size of the **table** is 36 (the number of notes in C-major that are on a 61-note keyboard). *Don't Save* is checked because we don't anticipate wanting to save changes to this file, and *Save Table with Patcher* option is not checked because it's sufficient to have the **table** stored in a separate file and read it in when we open the patch.

- Close the table Inspector and click on the **button** in Patch 3 to hear the use of **line** and **table** for reading through a predetermined set of pitches. Notice that **line** is also used to create a velocity crescendo from 20 to 125.



In order to go from 0 to 35 in exactly 3.5 seconds at a rate of exactly 10 notes per second, we had to play a small trick on the **line** object by giving it a time slightly longer than desired (3501ms). Here's why. To produce a perfectly timed ramp of *all* values from one number to another with **line**, you need to be aware of two details. The first detail is that **line** sets out interpolating from its starting point immediately, without pausing and without necessarily first outputting its starting point value.

So, specifying a 3.5 second line from 0 to 35 in one of the ways shown in the following example will not give us quite the desired results.


set 0, 35 3500
line 0 100
Goes to zero but doesn't output it, then starts counting from 1
35 3500
line 0 100
set 0
Same problem
0, 35 3500
line 0 100
Outputs both 0 and 1 immediately

A second detail worth knowing about **line** is that it actually travels to its destination in less than the specified time. It will output numbers at the rate specified by its "grain of resolution" (the rate specified by its second argument or by a number received in its right inlet) as long as the total time elapsed is less than that specified in its middle inlet. So, in the preceding example, it will actually arrive at its destination value of 35 in 3400ms. By giving it a slightly longer time, we allow it to take 3500ms, in 36 steps (including the first, immediate output), so the first step starts at 0.

Creating a New table

There are three easy ways to create a new Table file.

1. Choose **Table** from the New submenu of the File menu to open a new table editing window. Draw in the values you want, then choose **Save** from the File menu to save the table values.

2. Create a new **table** object, which will automatically open a new table editing window for you. The new **table** can be saved as a separate file before closing the table editing window, or you can check *Save with Patcher* so that it will be saved as part of your patch.

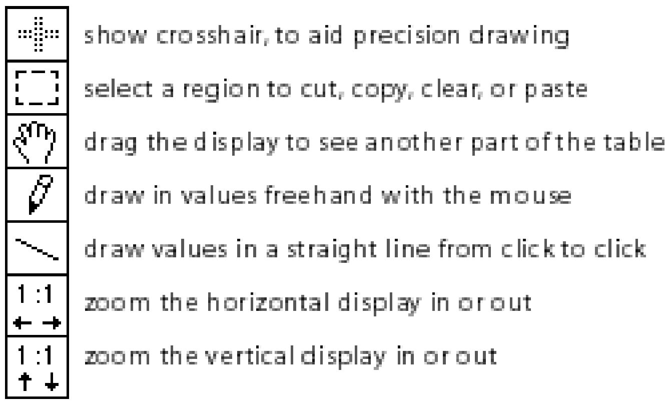
3. Choose **Text** from the New submenu of the File menu to open a new Text window. Type in the word table, followed by a list of numbers, then save the file.

Once you have saved a Table file you can use it in a Patcher window by creating a new **table** object and typing in the text file name as an argument.

Tutorial 32

Drawing in a Table Window

You can *draw* numbers into in a table editing window with the *Pencil tool*, or use the *Straight Line tool* which automatically draws a straight line between the points where you click.


show crosshair, to aid precision drawing
select a region to cut, copy, clear, or paste
drag the display to see another part of the table
draw in values freehand with the mouse
draw values in a straight line from click to click
1:1
zoom the horizontal display in or out
1:1
zoom the vertical display in or out

You can select a region of values with the Selection tool, cut or copy the values, then select another region and paste the first region in its place. You can even copy numbers from a Text window—or from any word processing application—then paste them into the graphic table window in a region specified with the *Selection tool.*

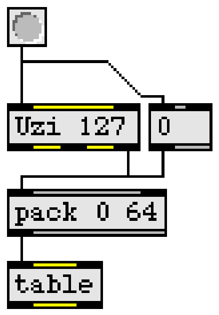
- Double-click on the **table** object at the bottom of Patch 1 and draw a new melodic curve, then listen to it by turning on the **metro**.

Other Ways to Alter a table

The **table** object can understand a number of messages in its left inlet. For a complete list, look under **table** in the Max Reference Manual. Patch 2 shows a few of the different messages, demonstrating ways to alter a **table** without opening its graphic editing window.

To store values in a **table**, send the value in the right inlet, then send the address where you want it stored into the left inlet. You can also send the address and the value to the left inlet together as a list.

In Patch 2, we use an **uzi** object to send lists to **table** automatically, filling all its addresses with the value 64.

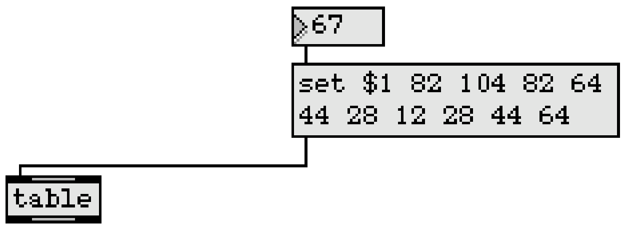


- Double-click on the **table** object in Patch 2, then click on the **button** to see the value 64 being stored at all the addresses.

When the **uzi** object receives a bang or a number, it sends a specific number of bang messages out its left outlet *as fast as possible*, all within a single tick of Max's internal clock. It also counts the bang messages as it goes and sends the count out its right outlet. It is particularly useful for sending out a series of messages "at the same time", such as a series of addresses and values for initializing a **table**. Since **uzi** starts counting from 1, we send the 0 separately, triggered by the same bang.

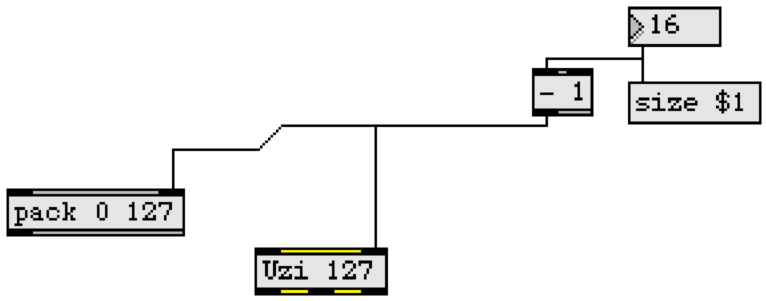
The word set, followed by an address and one or more values, stores the values starting at the specified address. For example, the message set 23 65 68 79 stores the number 65 at address 23, 68 at address 24, and 79 at address 25.

- Send address numbers from the **number box** to trigger the **message** containing set $1. Watch the results in the table editing window.



- The word size, followed by a number, sets the size of the **table** (the number of addresses). Trigger the **message** containing size $1 by sending it a number from the **number box**. Notice the change in the table editing window.

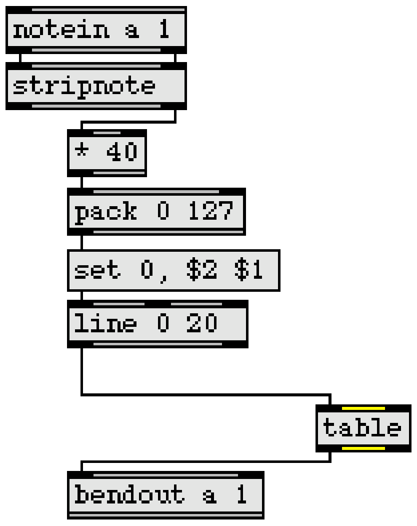
You'll also notice that we've included additional connections so that a new size setting will cause corresponding changes in other objects, so they interact properly with the **table**.



Using a table for MIDI Values

In Patch 1 we used the values in a **table** to provide pitches to **noteout**. In Patch 2 we use a **line** object to step through the **table** at different speeds, outputting different pitch bend curves.

Each note-on velocity is multiplied by 40 (yielding potential values from 40 to 5080). This value is used as the amount of time the **line** object will take to read through the **table**. The louder a note is played, the more slowly **line** reads through the table, sending out pitch bend values.



Notice how the **message** box is used to rearrange the incoming numbers and send out two different messages. We are not as picky about the timing of the **line** object here as we were in Patch 3 because the number of values sent out by **line** is quite unpredictable due to possible variations in played velocity.

- Alter the values in the **table** in the ways discussed, or by drawing in a curve yourself. Play a melody on your MIDI keyboard with long notes and a variety of velocities in order to hear the different pitch bend speeds.

Summary

The **table** object stores and recalls an indexed array of numbers. You can graphically view and edit the stored numbers by double-clicking on the **table** object.

The values in a **table** are normally discarded when the Patcher window is closed, but you can save them as part of the patch by selecting the **table**, choosing **Get Info…** from the Object menu, and checking *Save with Patcher*. You can also save a **table** as a separate file, and can then use it in a patch by creating a **table** object and typing in the file name as an argument.

To open a new Table window, choose **Table** from the New menu, or create a new **table** object in a Patcher window. You can also just type the word table, followed by a list of numbers, into a Text window and save it as a file.

To store values in a **table** object without opening its graphic editing window, send the *value* in the right inlet then send the *address* where you want to store it into the left inlet. Alternatively, you can send the address and value in the left inlet together as a list.

A set message changes certain values in the **table**, and a size message changes the number of values the **table** can hold.

The **uzi** object sends out a specific number of bang messages as fast as possible, in a single tick of Max's internal clock. It also counts the bang messages and sends out the count, so it can be used to send a whole series of messages in a single instant.

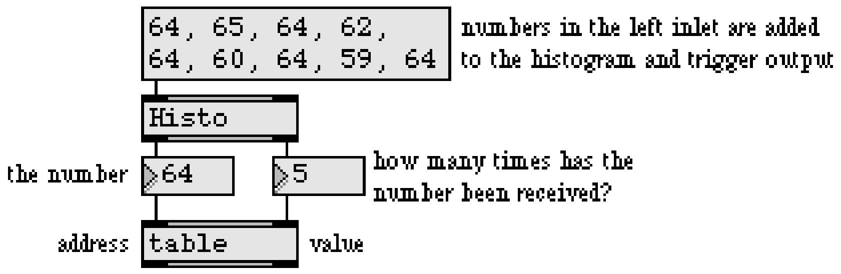
See Also

table	Store and graphically edit an array of numbers
uzi	Send a specific number of bang messages
Tables	Using the **table** graphic editing window

Making a Histogram

A *histogram* is a graph of frequency distribution, showing the relative occurrence of different events. The **histo** object keeps track of all the numbers it receives, as well as how many times it has received each number, in an internal histogram.

Each time **histo** receives a number from 0 to 127 in its left inlet, it adds that number to its internal histogram, then sends the number of times it has received that number out the right outlet, and the number itself out the left outlet. This output can be sent directly to a **table** to keep a graphic representation of the histogram. The *addresses* in the **table** correspond to the numbers received by **histo**, and the *values* in the **table** tell the frequency of occurrence of each number.


64, 65, 64, 62, 64, 60, 64, 59, 64
numbers in the left inlet are added to the histogram and trigger output
Histo
the number
64
5
how many times has the number been received?
address
table
value

The frequency distribution of different numbers—a comparison showing which numbers occurred most frequently—is displayed in the graphic window of the **table**.

Probability Distribution

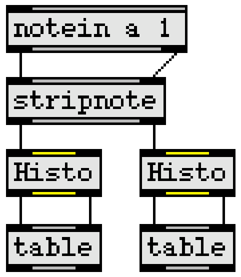
The bang message in the left inlet of a **table** has a special function. Instead of sending out a stored value, the **table** sends out an *address*. The probability of a particular address being sent out is in direct proportion to its stored value, as compared to the other values in the **table**. If the value stored in an address is greater than in other addresses, that address is more likely to be sent out when a bang is received. For a more detailed description of the effect of bang on a **table**, look under *Quantile* in this manual.

This feature of the **table** makes it perfect for storing a probability distribution. Each address can be assigned a different likelihood of being sent out when a bang is received. If the values in the **table** have been supplied by **histo**, as described above, the likelihood of a number being sent out of the **table** depends on how many times it was received by **histo**.

With this combination you can base the probability of a number's occurrence on the past history of how many times it has already occurred. The more it has occurred in the past, the more likely it is to occur in the future.

Keeping a History of What is Played

In our example patch, we have used **histo** and **table** to keep a frequency distribution of the pitches and velocities of notes played on the synth. These **table** objects store histograms of the pitches and velocities played.



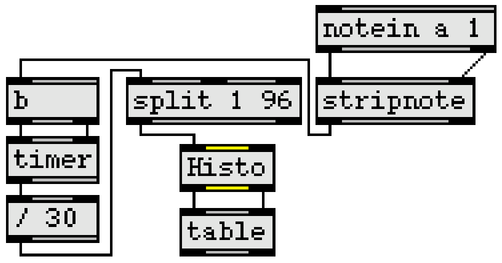
The **stripnote** object is very important here because without it note-off messages would cause each pitch to be counted twice, and the velocity 0 would be by far the most common velocity.

- Open the table editing windows containing the histograms of pitches and velocities, and play on the synth to see how the distributions are stored.

Rhythm Analysis

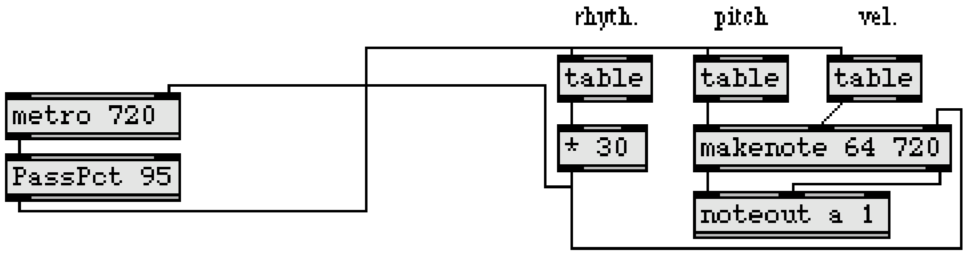
In the patch we use a simple method of rhythmic analysis to keep a histogram of the rhythms played. We use a **timer** to get the time between note-ons, and divide the time by 30 to get the *rhythm*—the number of 30ms pulses that elapse between notes.

If the time between any two notes is less than 1 pulse (30ms), we assume the second note is virtually simultaneous with the previous note and should therefore not be included in our analysis. If the time between notes is greater than 96 pulses (2880ms), we assume that the performer has stopped playing momentarily, or is holding an extremely long note. In either case, we don't want to include it in our histogram. So the **split** object passes only rhythms that are between 1 and 96 pulses in length, and a histogram of these rhythms is stored in the **table**.



An Improviser with a Memory

When the **metro** in the bottom left corner of the Patcher window gets turned on, it sends bang messages to **PassPct** (the patch we wrote in Tutorial 29), and 95% of the bang messages get passed on to the three **table** objects. A velocity, a pitch, and a rhythm are sent out, with the choice of each being based on the stored probability distributions (the histograms of what has been played by the performer). The velocity and the pitch are sent immediately to the synth. The rhythm is translated back into milliseconds by multiplying it by 30, then it is sent to the **metro** to set a new speed (and to **makenote** to set a new duration for the subsequent note).



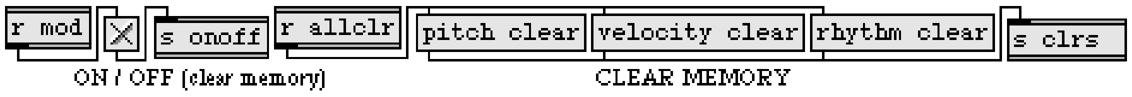
The resulting "improvisation" bears some resemblance to what you played on your MIDI keyboard, because it uses the same pitches, velocities, and rhythms, but the improviser patch recombines these parameters randomly. Because of the **PassPct** object, the improviser also rests about 5% of the time.

The User Interface

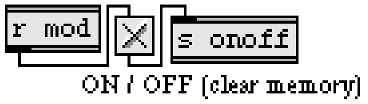
We had to decide how much control the performer should have over the improvising patch, and how the control should be implemented. We decided that the improviser would be turned on by moving the modulation wheel to any position other than 0, or by clicking on a **toggle** object.

We also wanted the performer to be able to erase the improviser's memory, either all parameters or just one parameter, so that its memory can be filled with new information. This requires sending clear messages to the **histo** and **table** objects, to set all their values to 0.

We decided to have all mouse controls located in a separate window, and have automatic on/off control from the mod wheel as well. We have hidden most of the objects and patch cords in the *[controls]* subpatch window, so if you want to see how the main patch communicates with the subpatch you'll need to unlock the *[controls]* window.

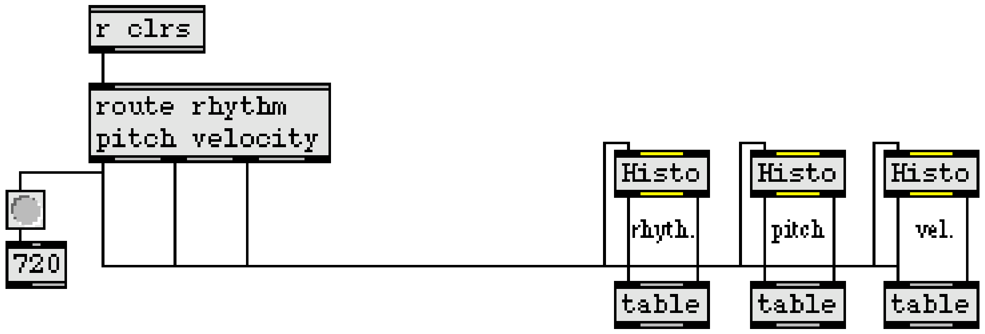


The data from **ctlin** is sent to the **toggle** in the subpatch, then back to the main patch. This lets us use the **toggle** both for displaying the on/off state received from the mod wheel and for actually *sending* on/off commands with the mouse.



The on/off state (0 or non-zero) is sent to **togedge**. **togedge** sends a bang out one of its outlets only when the number it receives represents a change from 0 to non-zero or vice versa. The left outlet is for changes from 0 to non-zero and the right outlet is for changes from non-zero to 0. If we sent the control data directly to the **metro**, the **metro** would get restarted with every non-zero number from the mod wheel. **togedge** lets us detect only the essential control data: changes to and from 0.

When **togedge** receives the *on* status from the **toggle**, it turns on the **metro**. When it receives a 0, it turns off the **metro** and sends a bang to all the **message** boxes in the controls window. Each of the clear messages is routed to the proper **histo** and **table** objects with **route**. Clearing the rhythm also resets the time of **metro** and **makenote** to 720.



The reason we used three different **message** boxes to send the clear messages separately is because it also gives the user the option of clearing the memory of only *one* parameter by clicking on a specific **message** box. Turning off the improviser clears *all* memories at once.

If we really wanted to make this improviser patch into a completed Max program for someone else to use, we would probably hide everything except the controls (plus a few comments to tell the user what to do). We left most things visible here so you could examine the patch.

Summary

histo keeps an internal *histogram* of the numbers it has received. When it receives a number in its left inlet it adds the number to its internal histogram, sends a report of how many times it has received that number out the right outlet, and sends the number itself out the left outlet.

The output of **histo** can be sent directly to a **table**, so that the frequency of occurrence of each number, as reported by **histo**, is stored as a value in the **table**. You can open the graphic window of the **table** to see the histogram.

A clear message in the left inlet of **histo** or **table** sets all values to 0. A bang in the left inlet of **table** causes it to send out an *address* rather than a value. The probability of a specific address being sent out depends on the value it stores, compared to the other values in the **table**. The greater the stored value of an address, the more likely that address is to be sent out when a bang is received. This feature of **table** allows you to use it for probability distributions.

By sending bang to a **table** that contains a histogram (a frequency distribution of past numbers, received from **histo**), you can cause numbers to be sent out of the **table**, with the likelihood of getting a number based on how frequently it has occurred in the past.

togedge is used to detect a *change* in the zero/non-zero status of incoming numbers. When the numbers change from 0 to non-zero, a bang is sent out its left outlet; when the numbers change from non-zero to 0, a bang is sent out its right outlet.

Using **route** to detect specific *selectors* (the first item in a message), messages can be routed to different destinations.

See Also

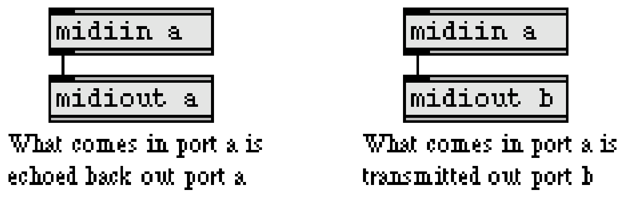
histo	Make a histogram of the numbers received
table	Store and graphically edit an array of numbers
togedge	Report a change in zero/non-zero values
Quantile	(**Max Topics**) Using **table** for probability distribution
Tables	Using the **table** graphic editing window

midiin and midiout

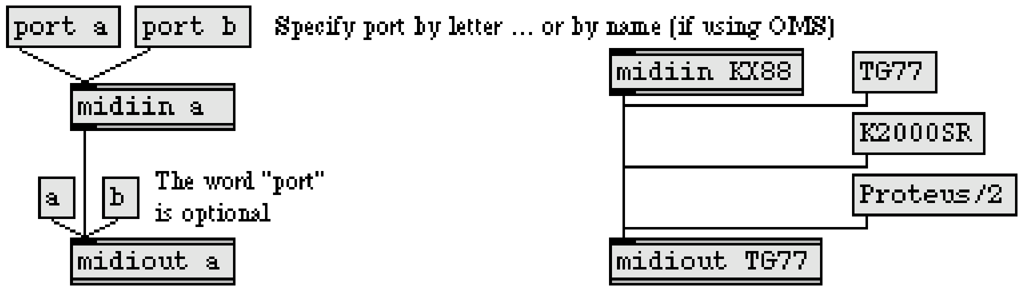
MIDI objects such as **notein**, **noteout**, **bendin** and **bendout**, transmit and receive specific types of MIDI data. If you want to transmit or receive *all* types of MIDI data as individual bytes (including status bytes), use **midiin** and **midiout**.

The **midiin** object is useful for examining every incoming MIDI byte. As we will see in Tutorial 35, it is also used for recording MIDI from your gear into the sequencer object, **seq**. The **midiout** object is used for sending any type of MIDI message to the synth, including system exclusive messages. It is also used to send MIDI data that is played back from the **seq** object.

In the simplest possible situation, Max can turn your computer into a very expensive MIDI thru box, by simply connecting the outlet of **midiin** to the inlet of **midiout**. These two objects—in fact, all MIDI objects—can be given a letter argument specifying a single port through which to receive or transmit, so you can use the arguments to route MIDI data from one port to another.


midiin a
midiout a
What comes in port a is echoed back out port a
midiin a
midiout b
What comes in port a is transmitted out port b

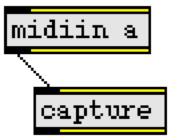
You can also change the input or output port of any MIDI object dynamically by sending the name of a port in the inlet. Beware of the possibility of stuck notes if you change ports while notes are being played.


port a
port b
Specify port by letter ... or by name (if using OMS)
midiin a
a
b
The word "port" is optional
midiout a
midiin KX88
TG77
K2000SR
Proteus/2
midiout TG77

If there is no port specified for **midiin** or **midiout**, either by an argument or by a port message in the inlet, port a is assumed by default. For more information about port assignment, see the *Ports* section of this manual.

capture

If you just want to examine the MIDI bytes that your equipment is sending out, you can connect the outlet of **midiin** to a **capture** object, as we have done in this Tutorial patch.

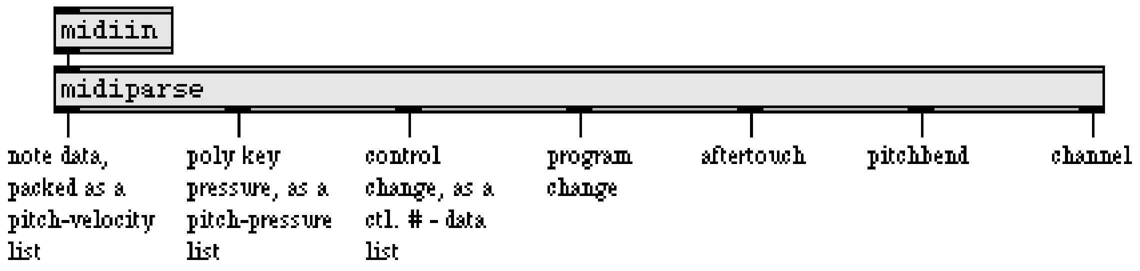


The **capture** object is a good all-purpose debugging tool. It collects the numbers it receives, and when you double-click on it, it opens a Text window for you to view the numbers. The numbers stored in **capture** are not saved when the patch is closed, but you can save the Text window as a separate file or copy the numbers and paste them somewhere else—even into a graphic Table window. Whenever you want to see what numbers are being sent from an outlet, just connect the outlet to a **capture** object, run the patch, then view the contents of **capture**.

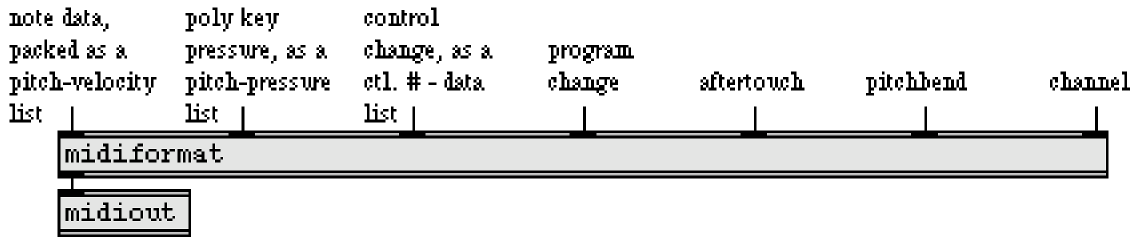
- Send out various types of MIDI messages from your keyboard: pitch bend, modulation, notes, program changes, etc. Every byte is received by **midiin** and stored in **capture**. Double click on the **capture** object to see the MIDI data.

midiparse and midiformat

The **midiparse** object sorts the raw MIDI data it receives from **midiin** or from **seq**, and sends the vital sorted data out its outlets. The combination of **midiin** and **midiparse** is like having all of the specialized MIDI receiving objects in one place.


midiin
midiparse
note data, packed as a pitch-velocity list
poly key pressure, as a pitch-pressure list
control change, as a ctl. # - data list
program change
aftertouch
pitchbend
channel

The **midiformat** object performs exactly the reverse function of **midiparse**. It prepares data into well-formatted MIDI messages with the appropriate status byte, and sends each byte to **midiout** for transmission to the synth.

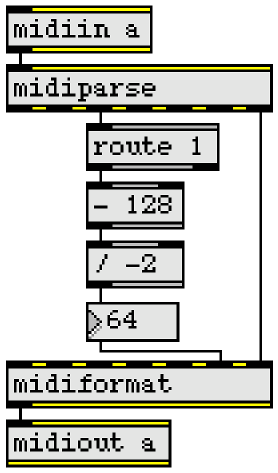

note data, packed as a pitch-velocity list
poly key pressure, as a pitch-pressure list
control change, as a ctl. # - data list
program change
aftertouch
pitchbend
channel
midiformat
midiout

Parsing and Formatting MIDI Data

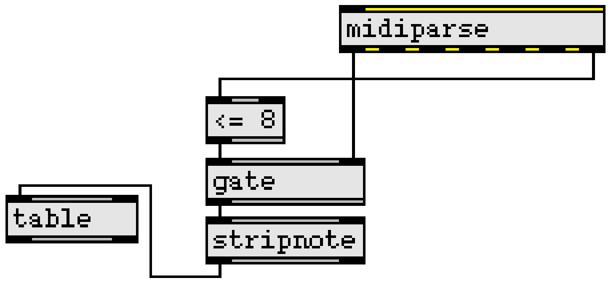
In the example patch, we have shown a couple of ways in which diverse MIDI data from **midiparse** may be used to control objects in Max, or may be given another meaning and transmitted with **midiformat** and **midiout**.

The controller data from the third outlet of **midiparse** is sent to **route**, which selects only data from controller 1, the mod wheel. The mod wheel data, from 0 to 127, is mapped to the range 64 to 0, then it is reassigned as pitch bend data by **midiformat** and transmitted to the synth. The resulting effect is that the mod wheel of the keyboard also controls the pitch bend. As the modulation increases from 0 to 127, the pitch is bent downward from 64 to 0.

This type of reassignment is a convenient way of correlating two different kinds of control data.



Another part of the patch shows how you can select data from a *group* of MIDI channels. The channel number is used to open or shut a **gate** for the note data. Only note data on channels 1 through 8 is sent on, and the pitch data triggers a number from the **table**.



- Play notes on your MIDI keyboard and you will hear that each note-on pitch is also used as an address to trigger a value from the **table**. If you set your keyboard to transmit on a channel between 9 and 16, the notes will not be passed by the **gate**.

Copying Captured Values into a table

The incoming pitch bend data is sent out of **midiparse** to a **capture** 128 object. The argument to **capture** sets the quantity of numbers it will store. This is one way to produce values for a **table** quickly and easily. It's a good way to preserve something you have done, such as a nice pitch bend, and save it in a **table** for future use.

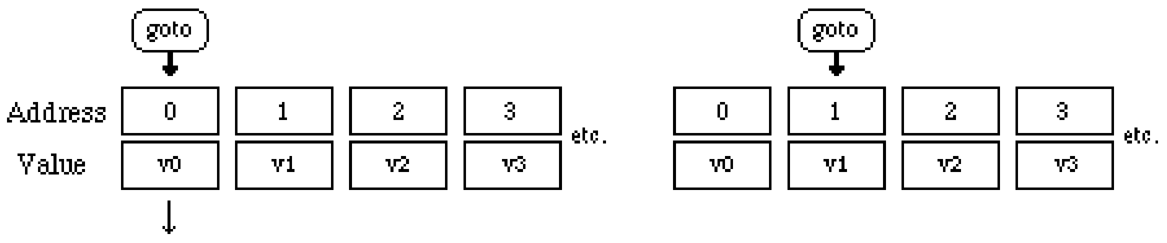
To copy captured pitch bend values into a **table**:

1. Click on the clear message to clear the **capture** objects.
2. Move the pitch bend wheel for at least 3.2 seconds. (The **speedlim** object limits the incoming pitch bend values to 40 per second.) After 3.2 seconds, the earliest values received by **capture** will be lost as new ones are received.
3. Double-click on the **capture** 128 object to open its Text window.
4. Select all the numbers in the Capture window.
5. Choose Copy from the Edit menu.
6. Close the Capture window.
7. Double-click on the **table** object to open its graphic editing window.
8. Choose the *Selection tool* from the Table window palette.
9. Choose **Select All** from the Edit menu.
10. Choose **Paste** from the Edit menu.
11. If you want to, you can save the Table window as a separate file, for future use in patches.

Stepping Through a table

In the left part of the Patcher window we introduce another way to step through the values in a **table**. A **table** object has a pointer—a place in memory where it stores an address. You can set the pointer to point at any address in the **table** with the word goto, followed by the address number in the left inlet.

For example, the message goto 0 sets the pointer at address 0 in the **table**, the first address.

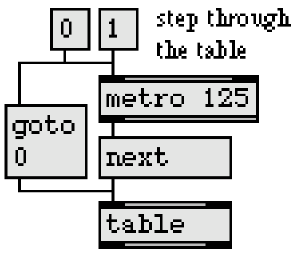


A next message sends out the value in the address at the pointer…

…then sets the pointer to the next address

When the message next is received in the left inlet, **table** sends out the value stored at the address at the pointer, then increments the pointer to the next address. When the pointer reaches the last address in the **table**, a next message will cause it to *wrap around* and point to the first address again, so you can use next to cycle continuously through a **table**. (You can also cycle backward through a **table** with the prev message, not shown in this patch.)

Thus, in our patch values are sent out of the **table** each time a pitch is played on your keyboard (on channels 1 through 8), and values can also be sent out automatically by turning on the **metro** to send repeated next messages to the **table**.



System Exclusive Messages

MIDI *system exclusive* (*sysex*) messages are used to send information other than that which is established as standard by the MIDI specification. Sysex commands are implemented by manufacturers as a way of modifying settings on their gear via MIDI.

Max has a **sysexin** object for receiving system exclusive messages, but to *send sysex* messages you need to format them yourself and then send them using **midiout**.

There is an object to help you format sysex messages, called **sxformat**. The **sxformat** object lets you specify some bytes of a MIDI message as arguments, and other bytes as *changeable arguments* to be replaced by numbers received in the inlet(s).

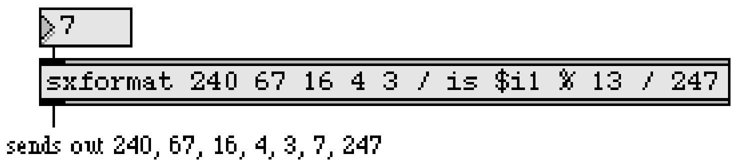
The format of changeable arguments in an **sxformat** object is different from that of changeable arguments in a **message** object. Changeable arguments in **sxformat**:

- contain the letter i to indicate that they are integer arguments (as in $i1)
- are preceded by the word is
- are bounded on either side by a slash (/)

For example, the changeable argument / is $i2 / will be replaced by the number received in the second inlet (or the second number in a list received in the left inlet).

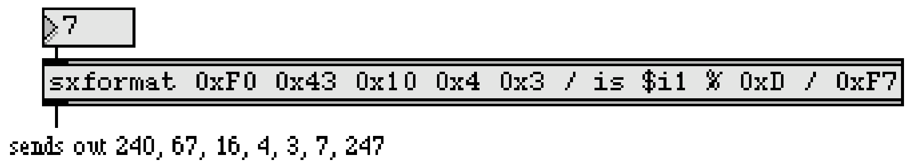
Calculations can even be performed on incoming numbers using the changeable argument. For example, the changeable argument / is $i1 + 1 / adds 1 to the number received in the left inlet before sending it out.

When **sxformat** receives a number or a list in its left inlet, it uses the number(s) to replace any changeable arguments, then sends each of the arguments out the outlet in sequential order.


7
sxformat 240 67 16 4 3 / is $i1 % 13 / 247
sends out 240, 67, 16, 4, 3, 7, 247

Programmers often express bytes of a sysex message in *hexadecimal* format, rather than decimal. If you prefer to type hexadecimal numbers, you can do so in Max by preceding the hexadecimal number with 0x (zero-x).

Here is an example of the same sysex message expressed in hexadecimal:


7
sxformat 0xF0 0x43 0x10 0x4 0x3 / is $i1 % 0xD / 0xF7
sends out 240, 67, 16, 4, 3, 7, 247

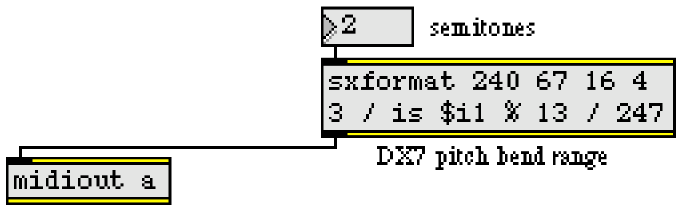
An Example Sysex Message

The status (beginning) byte of any sysex message is always 240. The second byte is the Manufacturer ID; each major synthesizer manufacturer has a unique number assigned to the brand name. The next bytes are established by the manufacturer—as many as are needed to express whatever is being expressed. A sysex message always ends with the "end-of-sysex-message" byte, 247.

When a synthesizer receives the sysex status byte, 240, it looks at the second byte. If the second byte is the ID of some *other* manufacturer, the synth ignores all the subsequent bytes until it sees 247. Then it begins to pay attention to incoming MIDI messages again.

In the bottom-right corner of the Patcher window is an example of the use of **sxformat**. It is designed to change the effective pitch bend range on a Yamaha DX7 synthesizer (or TX sound module). The first argument is the sysex status byte, 240, and the second argument is the Manufacturer ID for Yamaha, 67. Yamaha decided that the next byte would tell the synth what kind of message it's going to receive; in this case, 16 means "parameter change on channel 1."

The fourth byte specifies that the sysex message is a *performance parameter* change. The next byte is the parameter number—3 is for *pitch bend range.*



The next byte specifies the setting for the pitch bend range—how many semitones up and down we can bend the pitch. This is the value we want to be able to change, so we've made this byte a changeable argument in **sxformat**. The pitch bend range value must be from 0 to 12 semitones, so we've included a % 13 calculation to limit incoming numbers between 0 and 12: / is $i1 % 13 /. That's the end of the data portion of the message, so the ending byte, 247, comes next.

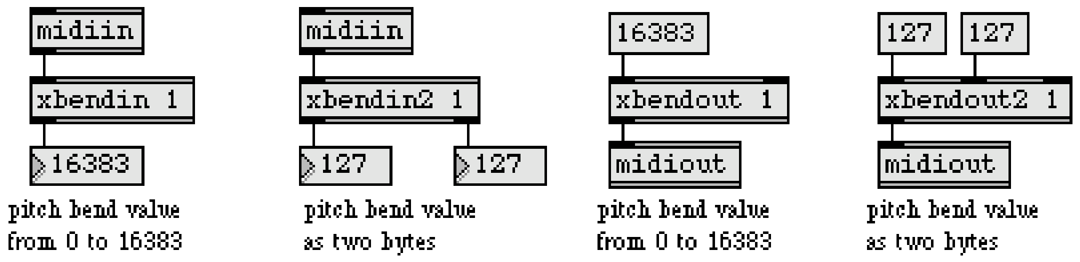
When a number is received in the inlet, the entire message is sent out, one number at a time, using the incoming number as the *pitch bend range* value.

- If you have a Yamaha DX7, you can change the pitch bend range by dragging on the **number box**. If you don't, your synth will ignore this message.

Extra Precision Pitch Bend Data

Most MIDI keyboards transmit and receive 128 different pitch bend values, and Max's MIDI objects do the same. However, a MIDI pitch bend message actually contains another byte for additional precision in expressing the pitch bend amount, and some synthesizers take advantage of this capability. If a synth does *not* have the extra precision capability, it always transmits a value of 0 in the extra precision byte, and ignores the extra byte when it is receiving pitch bend messages.

For MIDI keyboards that do have the extra precision capability, Max has objects for interpreting incoming extra precision pitch bend data received from **midiin**, and for formatting extra precision pitch bend messages to be transmitted by **midiout**.

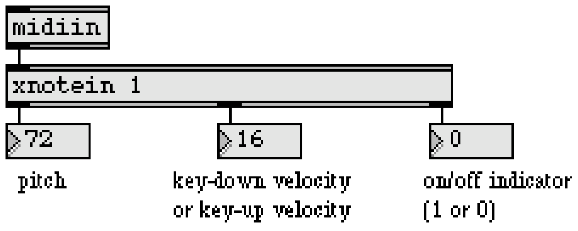

midiin
xbendin 1
16383
pitch bend value from 0 to 16383
midiin
xbendin2 1
127
127
pitch bend value as two bytes
16383
xbendout 1
midiout
pitch bend value from 0 to 16383
127
127
xbendout2 1
midiout
pitch bend value as two bytes

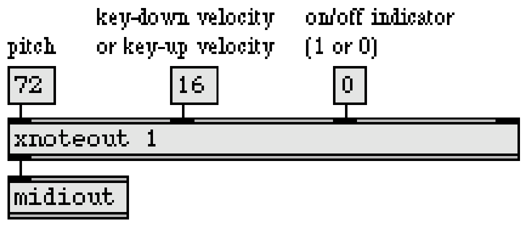
Because relatively few MIDI instruments have this capability, we don't discuss the matter in detail in this Tutorial. For more information, look under **xbendin** and **xbendout** in the Max Reference Manual.

Note-Off Messages with Release Velocity

In MIDI there are two ways to express a note-off. One way is as a *note-off* message with a release (key-up) velocity, and the other way is as a *note-on* message with a key-down velocity of 0. Since most synths are not sensitive to key-up velocities, **noteout** uses the latter method for note-offs.

For synths that are sensitive to key-up velocity, however, Max has objects for interpreting and formatting note-off messages with release velocity. To read more about these objects, look under **xnotein** and **xnoteout** in the Max Reference Manual.





Summary

The **midiin** object outputs each byte of MIDI data it receives. The **midiout** object transmits any number it receives in its inlet. You can set these objects to transmit or receive through a specific port by typing in a letter argument, a device name, or by sending a port message in the inlet.

The **midiparse** object interprets raw MIDI data from **midiin** and sends each type of data out a different outlet. The counterpart to **midiparse** is **midiformat**, which receives data in its various inlets and prepares different types of complete MIDI messages, which are sent by **midiout**.

To aid you in formatting system exclusive messages to be sent by **midiout**, **sxformat** lets you type in arguments which it sends out one at a time as individual bytes. You can include changeable arguments in **sxformat** which will be replaced by incoming numbers before the message is sent out.

The **capture** object stores a list of all the numbers it receives. You can view the list in a Text window by double-clicking on the **capture** object, and you can copy the contents of that Text window into a Table window. The **capture** object is good for viewing any stream of numbers when you are trying to figure out exactly what numbers are coming out of an outlet.

See Also

midiformat	Prepare data in the form of a MIDI message
midiin	Output incoming raw MIDI data
midiout	Transmit raw MIDI data
midiparse	Interpret raw MIDI data
sxformat	Prepare MIDI system exclusive messages

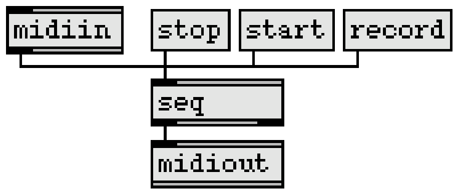
xbendin	Interpret extra-precision MIDI pitch bend messages
xbendout	Format extra precision MIDI pitch bend messages
xnotein	Interpret MIDI note messages with release velocity
xnoteout	Format MIDI note messages with release velocity
Ports	(**Fundamentals**) How MIDI ports are specified

Tutorial 35: **seq** *and* **follow**

seq

Max has four objects for recording and playing back MIDI performances: **seq**, **follow**, **mtr**, and **detonate**. In this chapter of the Tutorial we will discuss how to record a single track of MIDI data with the basic sequencing object **seq**, and how to compare a live performance to a previously recorded performance—in order to follow along with a performer—using **follow**.

The **seq** object records and plays back raw MIDI data in conjunction with **midiin** and **midiout**. It understands various text messages to control its operation, such as stop, start, and record.



Patch 1 contains the basic **seq** configuration shown above, plus a few other useful messages.

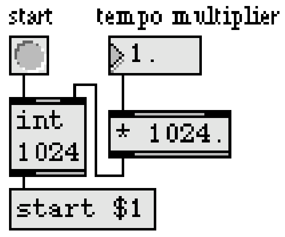
- Click on the record **message** box in Patch 1. Play notes, pitch bends, and modulation on your MIDI keyboard. Click on the word start to hear your performance played back. (You don't need to click on stop first, because start automatically stops the recorder before playing back.)

There is probably a delay before you what you played, because you didn't start playing at exactly the same moment you sent the word record to **seq**. The delay message can be used to change the starting time of the sequence.

- Click on the **message** containing delay 0 to set the starting time of the sequence to 0. Now when you click on start again, the sequence starts playing immediately.

The start message can be followed by a number argument specifying the tempo at which you want the sequence to be played back. The start argument divided by 1024 determines the factor by which the tempo will be increased or decreased. So, for example, the message start 1024 indicates the original (recorded) tempo, the message start 1536 plays the sequence back 1.5 times as fast, start 512 plays it back half as fast, and so on.

In the upper-left corner of Patch 1 we've devised a way to calculate the tempo ratio, letting you specify the tempo in terms of a multiplier, with 1 being the normal tempo.



- Drag on the **number box** to choose a tempo multiplier. (You can change the fractional part of the number by dragging with the mouse positioned to the right of the decimal point.) Then click on the **button** to play your sequence at the new tempo.

Changing the playback speed in this manner does not actually change the times recorded in the sequence, it merely changes the speed at which **seq** reads through it. Another message, called hook (not shown here), alters the times in a sequence. Look under **seq** in the Max Reference Manual for details.

Saving and Recalling Files

If you like, you can save the sequence you have recorded in a separate file, to be used later. The write message opens a standard Save As dialog box for you to name the file where you want to store the sequence. You may want to give the names of your sequence files some unique characteristic so you can distinguish them from Patcher files and Table files. (We use *.sc* at the end of the name to identify the file as a musical score).

If you check *Save as Text* in the dialog box when you save the file, you can view the sequence in a Text window by choosing **Open As Text…** from the File menu. Otherwise, the file is stored as a standard MIDI file.

To load a saved file into a **seq** object, send the read message to **seq**, and a standard Open Document dialog box will appear so you can choose the file you want to load in. If the read message is followed by a file name argument, **seq** loads that file automatically (provided it's located where Max can find it). You can set **seq** to load a file automatically when the patch is opened by typing the name of the sequence file as an argument to the **seq** object.

- Click on the **message** containing read bourrée.sc to load in a brief melodic excerpt from a Bach bourrée in E-minor. Send a start message to **seq** to hear the melody.

Processing a MIDI Sequence

The output of **seq** is in the form of individual bytes of MIDI messages, and can be transmitted directly to the synth with **midiout**. It can also be sent to **midiparse**, however, and the parsed data can then be processed by other Max objects before it is sent to the synth.

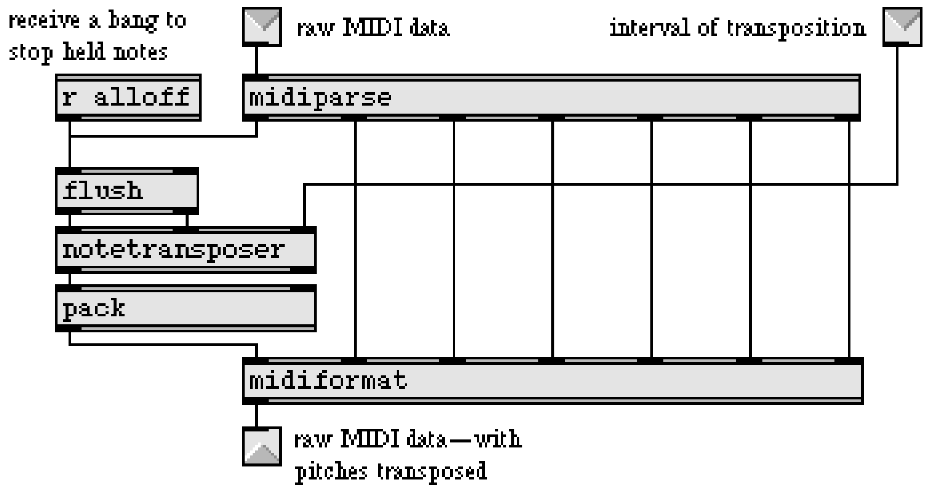
In the patcher **transpose** object we parse the raw MIDI data received from **seq**, transpose the pitch of the notes by some amount, then reformat the MIDI messages and send them to **midiout**.

- Double-click on the patcher **transpose** object to see its contents.

The patcher **transpose** subpatch contains a *nested subpatch*, the **notetransposer** patch that we made in Tutorial 27. Subpatches can be nested in this manner so that each task of a patch is *encapsulated* and is easily modified. (For more on this subject, look under *Encapsulation* in this manual.)

Notice that we have included an additional handy feature inside the **patcher** transpose subpatch: a **flush** object to turn off held notes. When **seq** is playing a sequence and gets stopped by a stop message, it may be in the middle of playing a note, and the note-off message will not be sent out.

In Patch 1, we made the stop message also trigger a **button** which sends a bang to the **flush** object in the subpatch to turn off any such stuck notes.


receive a bang to stop held notes
raw MIDI data
interval of transposition
r alloff
midiparse
flush
notetransposer
pack
midiformat
raw MIDI data—with pitches transposed

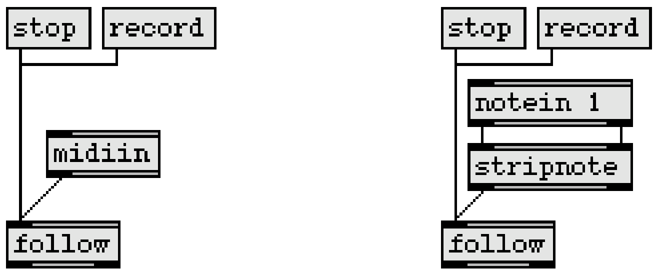
Tutorial 35

In general, whenever your patch is capable of stopping **seq** while notes are being recorded or played back, there is the potential for vital note-off messages to be lost. This is especially true if your patch sends stop, record, or play messages by some automatically generated means. Bear this potential danger in mind when constructing your patch, and include an object such as **flush**, **midiflush**, **poly**, or **makenote**—whichever is appropriate—to provide missing note-offs. Examples are shown in Tutorial 13.

- Record a sequence (or use the bourrée excerpt), and play the sequence with a start message. Try changing the transposition with the **hslider** while the sequence is playing.

follow

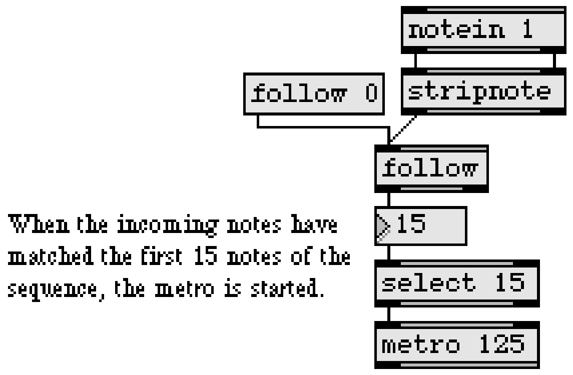
The **follow** object is very similar to **seq** in its ability to record MIDI data. But whereas **seq** only records MIDI messages, **follow** can also record a sequence of single numbers that are not in the form of complete MIDI messages (such as the pitches from MIDI note-ons).



follow can record MIDI messages, or single numbers (e.g., just note-on pitches)

A sequence can be stored in **follow** by recording MIDI data, by recording a series of single numbers, by reading in a file with a read message, or by typing in a file name argument. Once it has a stored sequence, **follow** can use that sequence as a musical score, and follow along while a performer plays the music. Each time the performer plays a pitch that matches the next note-on in the stored sequence, **follow** sends the pitch out its right outlet and sends the index number of that note's position in the sequence (1, 2, 3, etc.) out its left outlet.

The particular utility of this score-following feature is that the index numbers can be used to trigger other notes, or any other process such as, say, turning on a **metro** when the 15th note is matched.


notein 1
follow 0
stripnote
follow
When the incoming notes have matched the first 15 notes of the sequence, the metro is started.
15
select 15
metro 125

How follow Follows

When **follow** receives the message follow with a number argument, it begins to look for incoming pitches which match the notes in the score, starting at the index specified in the argument. For example, follow 10 causes the object to look for incoming pitches that match the 10th note in the score. When the matching pitch is received, **follow** sends that pitch out its right outlet, and sends the index out its left outlet.

The **follow** object even allows for wrong notes, so if the performer plays a couple of spurious notes, or skips a note or two, **follow** will still be able to keep track of the performer's progress through the score.

One can also step through the score with repeated next messages. After a follow message has been received, the message next triggers the pitch at the specified index and increments the pointer to the next index.

An Attentive Accompanist

When we use the index numbers from the left outlet of **follow** as addresses of a **table**, or addresses of some other array object like **funbuff**, the index numbers can trigger other values. In this way, we can create an accompanist who "knows the score" and follows along with the performer. Each time the performer plays a note of the score, the accompanist has a specific reaction—play a simultaneous note or notes, play some independent melody, rest, whatever—and seems to follow along with the performer.

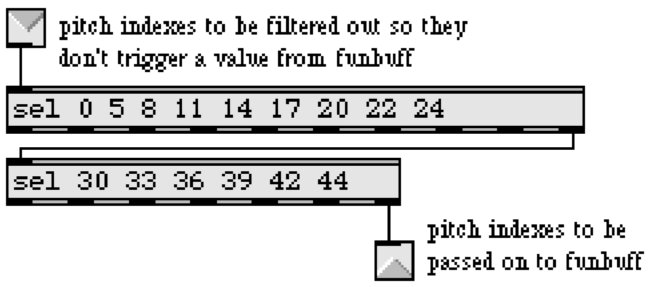
We've made such an accompanist in Patch 2. The accompanist plays the left hand part of the Bach E-minor bourrée while you play the right hand part. The **follow** object has loaded the sequence file *bourrée.sc* to use as the score. Each time a note of the score is played, an index number is sent out that triggers some sort of reaction.

- Click on the follow 0 message to start the score-follower at the beginning of the score. Play the right hand part of the bourrée excerpt and Patch 2 will play the left hand part along with you.
- If you've forgotten how the melody goes, read the *bourrée.sc* sequence into the **seq** object in Patch 1 and listen to it.
- Click on follow 0 again, and play the melody with an occasional wrong note or skipped note. If you don't mess up too much, **follow** manages to account for your mistakes and continues following the score.
- Try the melody again, with ritards at the end of the phrases. The extra notes that the accompanist plays match your tempo.

Analysis of Patch 2

Sometimes we want the left hand to play a note along with the right hand, other times we want the left hand to do nothing new (when the right hand is playing the second of a pair of eighth notes and the left hand is just holding a quarter note), and occasionally we want the left hand to play a note in between notes played by the right hand. How do we accomplish each reaction?

The index numbers are first sent to a subpatch called **patcher** silencer. This subpatch simply filters out the index numbers which we don't want to trigger a note of the accompaniment. The **sel** objects select those index numbers and pass the rest on.


pitch indexes to be filtered out so they don't trigger a value from funbuff
sel 0 5 8 11 14 17 20 22 24
sel 30 33 36 39 42 44
pitch indexes to be passed on to funbuff

Contents of the patcher silencer subpatch

Notice that **sel** objects can be linked together to select more than 10 numbers, since the numbers that are not matched by the first **sel** object are passed out the rightmost outlet to the second **sel** object.

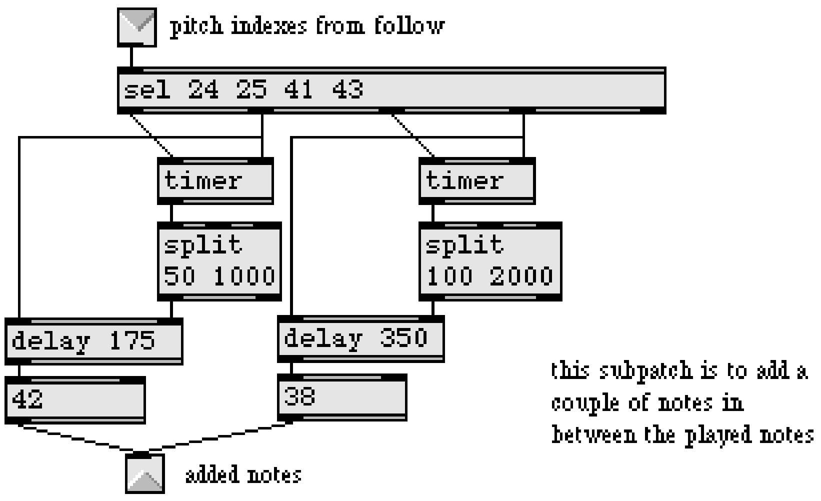
The remaining index numbers are sent as addresses to **funbuff**, which sends out an appropriate accompanying pitch value. To make **funbuff** respond properly, we simply made a list of *addresses* and *values* and saved the list as a **funbuff** file named *bourrée.fb*.

- If you want to see the contents of **funbuff**, choose **Open As Text…** from the File menu and open the file named *bourrée.fb.*

We could have also stored the accompaniment pitches in a **table**—or in a **coll** object, which will be explained in Tutorial 37.

So far we have made the accompanist play some notes that are simultaneous with the melody notes, and we've made the accompanist rest on melody notes that are unaccompanied, but how about when the accompanist has to play notes on its own, in between melody notes? This occurs twice in the score, once at the end of each phrase.

To help the accompanist play notes on its own, the **patcher** addnotes object measures the tempo of the performance and plays notes with a delay time based on its perception of the performer's tempo.


pitch indexes from follow
sel 24 25 41 43
timer
timer
split 50 1000
split 100 2000
delay 175
delay 350
42
38
this subpatch is to add a couple of notes in between the played notes
added notes

Contents of the patcher addnotes subpatch

For example, the subpatch measures the amount of time between notes 24 and 25 of the melody (the speed of an eighth note), then delays for that amount of time before triggering the pitch 42. Likewise, the time between the 41st and 43rd melody notes (the speed of a quarter note) is used as a delay time before sending out the pitch 38. This is a simple (but fairly effective) method of analyzing the performer's tempo and playing notes in that tempo.

It's always a good idea in programming (and elsewhere, for that matter) to prepare for the unexpected. What happens if the performer accidentally misses one of these notes that we need for analyzing the tempo and triggering added accompaniment notes?

If the performer misses the first note of a pair, for example, the second note will trigger a ridiculously large value from the **timer** and the accompaniment note will get delayed far too long. To protect against this eventuality, we have used **split** objects to limit the time values that can be sent to **delay** within certain (only *moderately* ridiculous) extremes. If the value from **timer** exceeds these limits, the **delay** object will use the delay time in its argument. If the performer misses the second note of a pair but continues on, the added note will never get played, but by then the performer will have passed that point anyway, and **follow** will keep up with the performer.

The pitches from **patcher** addnotes and from **funbuff** are sent to **makenote** where they are paired with the velocity of the right hand melody notes, so the accompanist is sensitive to the performer's dynamics, as well. Rather than use an algorithm or a lookup table to provide durations for the accompaniment notes, we just picked a duration that seems to work both as an eighth note duration and as a stylistically staccato quarter note.

Summary

A single track of raw MIDI data can be recorded and played back (at any speed) with the **seq** object. The MIDI data is received from **midiin** and is transmitted by **midiout**. You can also parse the data from **seq** using **midiparse**, and process the numbers with other Max objects before transmitting them.

A recorded sequence can be saved as a separate file by sending a write message to **seq**. If you check the *Save as Text* option in the Save As dialog box, you can open and edit the file later with **Open As Text…**. A MIDI file can be read into **seq** by sending a read message, or by typing in the file name as an argument.

The **follow** object allows you to record or read in a sequence, then use that sequence as a musical score to follow along with a live performance. As the pitches received in the inlet are matched with notes in the score, the index number for each note is sent out, and can be used to trigger other notes or processes.

Tutorial 35

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
Sequencing	(**Max Topics**) Recording and playing back MIDI performances

mtr

The **mtr** object is Max's most versatile sequencer. It can record and play back up to 32 different tracks of messages: numbers, lists, or symbols. The tracks can be recorded and played back either separately or all together. With this versatility, you can record and play back not only MIDI bytes, but numbers from any object such as a **slider** or a **dial**, sequences of text messages to be displayed to the user, pitch-velocity lists, etc.

We'll show how **mtr** is used to record and play back MIDI data.

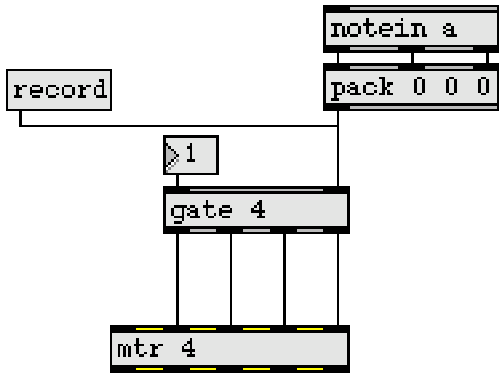
The number of tracks in an **mtr** object is specified by a typed-in argument. The leftmost inlet is a *control inlet* for receiving commands, and the other inlets are for messages you want to record. The command messages for **mtr** are similar to those for **seq**, but not identical. Notably, **mtr** understands the message play instead of start, and the play message does not take a tempo argument.

When command messages such as stop, play, record, mute, and unmute are received in the left inlet they apply to *all* tracks of **mtr**. These commands can be followed by a number argument, however, specifying a unique track to which the message applies. Alternatively, these messages can be received in an individual track's inlet, to give a command to just that track.

A 4-Track "Simul-Sync" Recorder

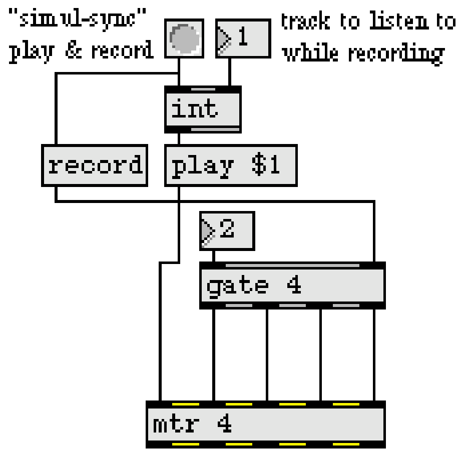
Patch 1 shows a configuration to record four separate tracks of MIDI note data separately, then play them all back together. **number box** objects let you specify the track you want to record on and, if you wish, a track to listen to while you are recording.

When you choose a track to record, the **gate** opens that outlet to let the record message and the note data go only to that track.



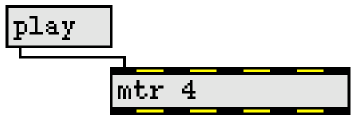
- Set the open **gate** outlet to 1 (to record on track 1), and click on the record message. Play some notes on your MIDI keyboard. When you are finished recording, click the play message to hear what you have recorded.

- Now open gate outlet 2, and enter the number 1 in the **number box** at the top of the patch so that you can listen to track 1 while you record track 2.



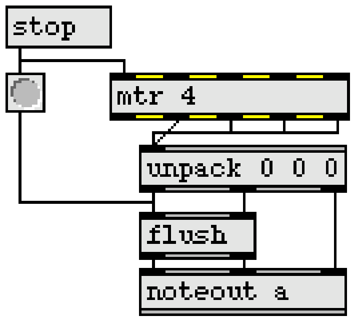
When you click the "simul-sync" **button**, the message play 1 will be sent to the left inlet of **mtr**, and the message record will be sent to the inlet of track 2.

- Click the **button** and record track 2. When you have finished, click on play to hear both tracks.



You can continue in this manner to record all four tracks. If there is some delay between the time you click play and the time the sequence starts to play, it's because you took some time to begin recording notes after you clicked record. To eliminate this delay, and cause the first event in **mtr** to begin at time 0, click the message first 0.

Notice that once again we have included a **flush** object to guard against stuck notes. Every time a stop message is sent to **mtr**, a bang is also sent to **flush** to turn off any notes currently being held.



Recording Messages from Different Sources

Note data is not the only thing that can be recorded with **mtr**; messages from virtually any combination of objects can be recorded and played back by the same **mtr** object. In Patch 2 we record numbers from **pgmin**, **bendin**, **ctlin**, and a **dial**, each on a different track.

- Click on the record message in Patch 2 and send pitch bend, modulation, and program change messages from your MIDI keyboard for several seconds. You can also move the **dial** with the mouse. When you have finished, click play and you will see your performance played back, controlling other objects.

The first 0 message can be used to eliminate the delay between the time you clicked record and the time you started to transmit MIDI messages from the keyboard. The message delay 0 causes *every* track to start at time 0, even if you *started* sending data to the tracks at different times.

- To see the difference between first 0 and delay 0, click on record and send about 5 seconds of pitch bend data, then 5 seconds of mod wheel data, and so on. When you have finished, click stop.

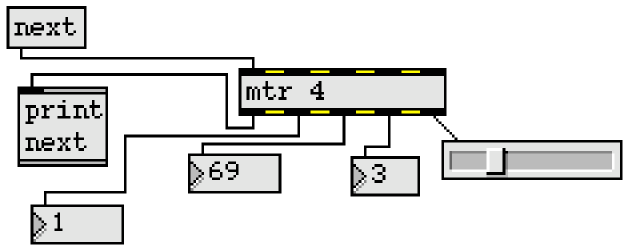
- Next, click first 0 to eliminate the initial delay before any data was recorded. Click play to see your performance replayed.

- Now click delay 0 and play your sequence again. This time all tracks start at time 0, even though you started recording data on one track before the others.

When you send **mtr** a mute message while it is playing, it continues to play its stored sequence, but it *suppresses* the actual output. Use the unmute message to restore output. Individual tracks can be muted and unmuted by following the mute or unmute message with a track number argument, or by sending the messages into a specific track's inlet.

You can *step through* the messages stored in **mtr** by sending repeated next messages to the control inlet. When **mtr** receives next, it sends out the next message stored in each track. It also sends a two- item list out the leftmost outlet once for each track, reporting the track number and the duration (the time between that message and the following one in the track).

- Check **All Windows Active** in the Options menu, and bring the Max window to the front so you can see what gets printed in it. Then click on the next message. The next value stored in each track is sent out the track outlets, and a list for each track, consisting of the track number and the duration between messages, is sent out the left outlet.



The rewind message is used in conjunction with next. It sets the pointer back to the beginning of the sequence, so that the message next will start at the beginning again.

Tutorial 36

Summary

The **mtr** object records and plays back up to 32 tracks of any message type—numbers, lists, or symbols. Tracks can be recorded, played, stopped, muted, and unmuted—either individually or all tracks at the same time.

The next message can be used to *step through* the recorded messages instead of playing them back at their original recorded speed.

See Also

detonate	Graphic score of note events
follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
Detonate	(**Max Topics**) Graphic editing of a MIDI sequence
Sequencing	(**Max Topics**) Recording and playing back MIDI performances

Tutorial 37: Data Structures

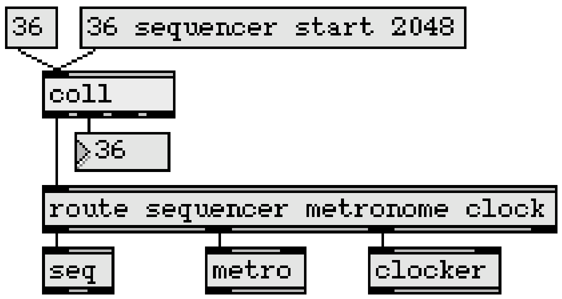
What Is a Data Structure?

A *data structure* is any collection of data that is stored in some arrangement that allows individual items to be found easily. In Tutorial 32, we used the **table** object, a data structure called an *array*, where we used an index address to access the stored values. In this chapter, we'll use some objects that allow you to create your own collections of data and retrieve them with whatever addresses you wish.

coll

The most versatile data structure object in Max is the **coll** (short for *collection*). A **coll** object stores a collection of many different messages, of any type and length (up to 256 items long), and can give each message either a number address or a *symbol* (word) address.

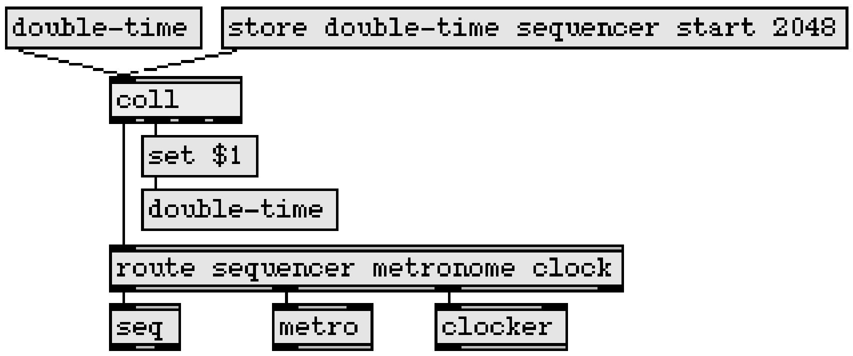
Any time **coll** receives a list in its inlet it uses the first number in the list as an address, and stores the *remaining* items in the list at that address. (You'll recall that a list is any space-separated set of items beginning with a number.) For example, when **coll** receives the message 36 sequencer start 2048, it stores the message sequencer start 2048 at address 36. After that, whenever **coll** receives the number 36 alone, it sends the address (36) out the second outlet, and sends the message sequencer start 2048 out the left outlet.



You can also store messages with a *symbol* as an address instead of a number. If you just send it a message beginning with a symbol, **coll** will try to interpret the symbol as another kind of command, and won't store the rest of the message. So, to store messages with a symbol as the address, you must precede the symbol with the word store.

When **coll** receives a message beginning with the word store, it uses the first item after the word store as its address, and stores the rest of the message at that address. When **coll** receives that address alone in the inlet, it sends it out the right outlet (preceded by the word symbol), and sends the stored message out the left outlet.

Here's the same patch, using a *symbol* as an address for the message stored in **coll**, instead of a number.


double-time
store double-time sequencer start 2048
coll
set $1
double-time
route sequencer metronome clock
seq
metro
clocker

coll precedes the symbol address it sends out its second outlet with the word symbol so that the address will not be interpreted as a command by other objects. For example, a **message** box will not be triggered by a word, because it will try to understand the word as some kind of command. However, if the word is preceded by symbol, the **message** box will be triggered and the word will replace a $1 changeable argument in the box.

Editing the Contents of coll

To view and edit the contents of a **coll**, double-click on the object and a Text window will open. If you make any changes to the Text window, you will be asked whether you want to keep those changes in the **coll** when you close the Text window.

The contents of a **coll** are written in a specific format. For details, look up **coll** in the Max Reference Manual.

Saving the Contents of coll

Once you have stored messages in **coll**, you can set it to save its contents as part of the patch. You unlock the Patcher window, select the **coll** object, choose **Get Info…** from the Object menu, and check *Save coll with Patcher* in the Inspector window.

Alternatively, you can save the contents of the **coll** as a separate file (so the contents can be used by more than one patch). To do this, open the **coll** object's Text window and choose **Save As…** from the File menu. Another way to save the contents as a separate file is to send a write message to the **coll** object, which opens a Save As dialog box.

To *load* a file into a **coll** object, type the name of the file in as an argument, or send **coll** a read message, which will cause the Open Document dialog box to appear.

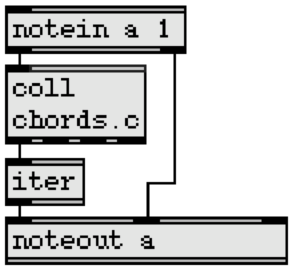
Storing Chords in coll

- Double-click on the patcher **coll_examples** object to open the subpatch window.
- Play some long notes on your MIDI keyboard. Every key on the keyboard has a unique 3-note chord assigned to it.

The chords are stored in a **coll** object, using the key number (the pitch of the played note) as the address.

- Double click on the **coll** object in Patch 1 to see how the chords are stored. If you want to change some of the chords, edit the numbers in the Text window, then close the window to update the contents of the **coll**.

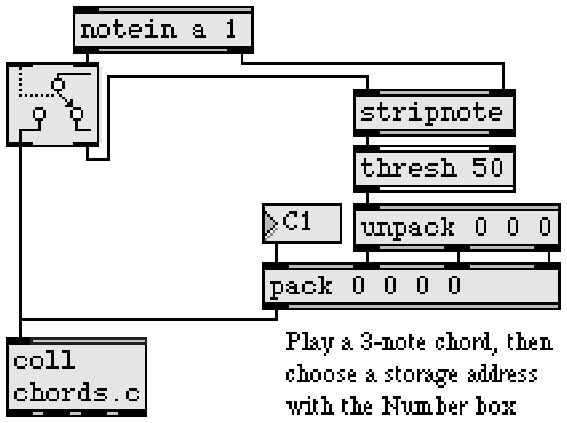
When a pitch value is received in the inlet, **coll** sends out the 3-item list stored at that address. The list is broken up into a series of numbers by **iter**, and the numbers are sent (virtually simultaneously) to **noteout**, where they are combined with the note-on or note-off velocity being played on a MIDI keyboard.



The rest of Patch 1 is for storing your own chords in a **coll**. When pitch data is routed out the right outlet of the **Ggate**, the note-on pitches are sent to a **thresh** object.

The **thresh** object is like **iter** in reverse. Numbers which are received within a certain *threshold* of time are packed together in order. The threshold is the maximum number of milliseconds between any given number and the previous one. When no new number is received within a certain period, the numbers are sent out as a list.

So, when you play notes of a chord simultaneously (within 50ms of each other) they are packed as a list, and after 50ms they are sent out. The **unpack** object selects the first three numbers and stores them in **pack**. Then, when you select an address by entering a number in the **number box**, the address and the accompanying chord notes are all sent to **coll** as a list for storage. The **number box** has been set to send only on mouse-up so that you can use it as a slider to enter an address. Otherwise, the chord would be stored in the address of every number you dragged through.


notein a 1
stripnote
thresh 50
C1
unpack 0 0 0
pack 0 0 0 0
coll chords.c
Play a 3-note chord, then choose a storage address with the Number box

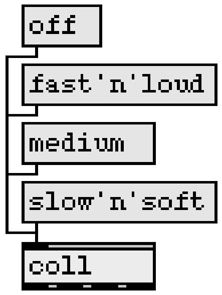
- Click on the **Ggate** to point its arrow to the right outlet.
- Play a 3-note chord that you want to store in **coll**. Play it as a 4-note chord first, to hear it along with the address note that will eventually trigger it, then play it as a 3-note chord to store it in **pack**.
- Use the **number box** to select the address where you want to store your chord.
- Repeat the above steps until you have stored all the chords you want, then click on **Ggate** again to direct the played pitches back to **coll**. Play the address notes to hear the results.

If you want to save your new chords, you must open the **coll** object's Text window again and choose **Save As…** from the File menu.

Parsing the Data Structures in a coll

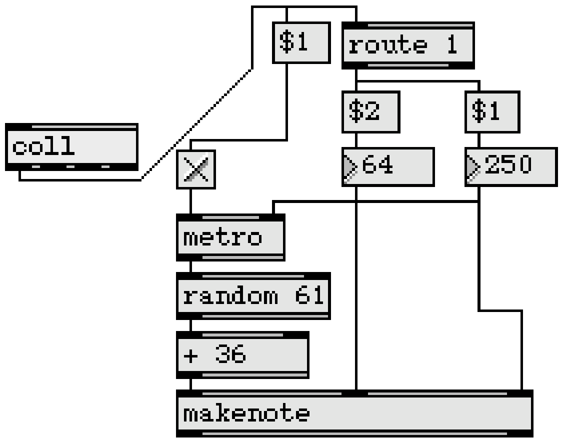
Patch 2 shows how **coll** can be used to store messages with symbol addresses, and it also shows how complex messages can be stored in **coll** and then parsed when they are sent out.

- Click on the different messages in Patch 2.



- Double-click on the **coll** object in Patch 2 to view its contents.

The format we have chosen for our data structure in this **coll** is: m*etro command, tempo (duration)*, and *note velocity*. Because each message stores the data in the same order, we can access individual items in the data structure and use the item in a specific way.



The data structure is parsed as it comes out of the **coll**. The message is first sent to **route**. If the first item of the message is 1 (meaning *metro on*), we use the remaining items in the message to supply a duration to **makenote**, a tempo to **metro**, and a velocity to **makenote**. If the first item in the message is not 1 (in this case, **0** is the only other possibility), nothing needs to be sent, so **route** ignores the message. After the essential data is supplied to **metro** and **makenote**, the first item is used to turn the **metro** on or off.

Other Features of coll

These examples should give you a taste of what **coll** can be used for. There many other command messages which **coll** understands, too numerous to cover in detail in this

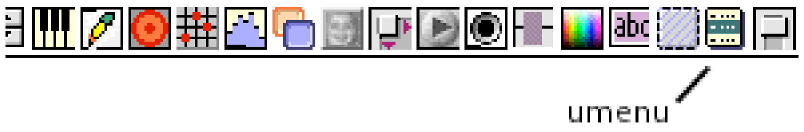
Tutorial. For example, you can step through the different messages in **coll** with goto, next, and prev commands. And you can select or alter individual items of stored messages with commands such as nth (to get the *nth* item within a message), sub (to substitute an item in a message), and merge (to append items at the end of a message). For details about these commands, look under **coll** in the Max Reference Manual.

- Before you go on and look at the other subpatches, you will want to disable the chord-playing patch in the *[coll_examples]* window. Point the arrow of **Ggate** to the right outlet, or disable MIDI in the window by clicking on the MIDI enable/disable icon in the title bar. Close the *[coll_examples]* subpatch window.

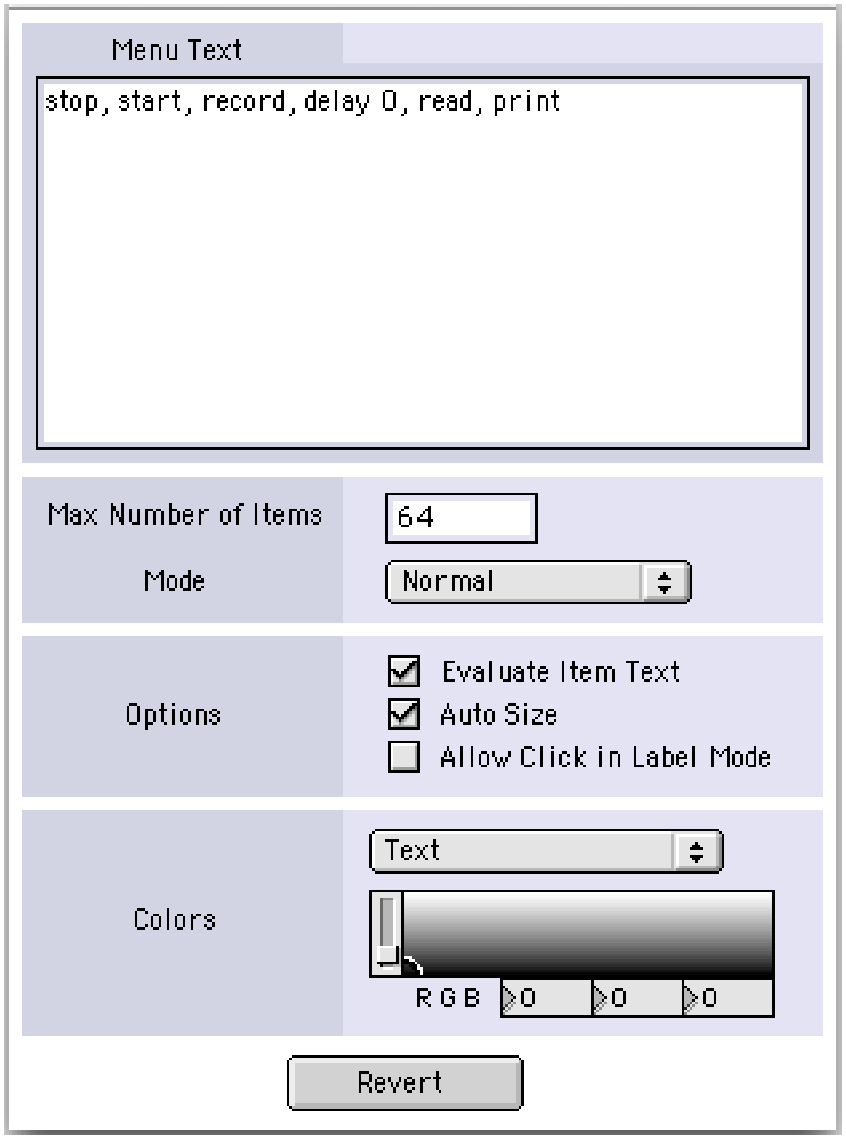
menu

The **menu** object creates a pop-up menu in a Patcher window. It can be used to choose commands with the mouse, just like any other menu, and it can also be used to display messages when the number of a **menu** item is received in the inlet.

When an item in the **menu** is selected with the mouse (or by a number received in the inlet), the number of the **menu** item is sent out the left outlet. The items in the **menu** are numbered beginning with 0.



After you create a new menu object, choose **Get Info...** from the Object menu to open the menu Inspector. You type the menu items into the large text field in the Inspector window, separating them by commas. The menu items can be any type of message: numbers, lists, words, sentences, whatever. If you want to include a comma *within* a **menu** command, you must precede it with a backslash (\).


Menu Text

stop, start, record, delay 0, read, print

Max Number of Items: 64

Mode: Normal

Options: ✓ Evaluate Item Text, ✓ Auto Size, ☐ Allow Click in Label Mode

Colors: Text

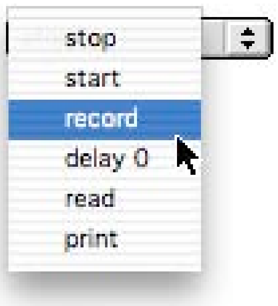
RGB 0 0 0

If *Evaluate Item Text* is checked in the Inspector window, **menu** will send the text of the item out the right outlet. If you check the *Auto Size* option, the width of the **menu** will automatically adjust according to the length of the text in **menu** commands.

- Double click on the patcher **menu_examples** object to open the subpatch window.

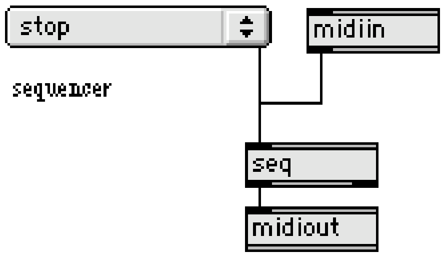
We've hidden many of the objects in this subpatch, to give you a visual idea of how menus may be used to enhance the user interface of a patcher program.

- Record and play back a MIDI sequence using the *Sequencer* **menu** in the left part of the window.



It's easier to use a single **menu** than it is to click on a bunch of **message** boxes, it's more aesthetically pleasing, and it has the advantage of displaying the most recent command.

- Unlock the *[menu_examples]* window to see how the **menu** is connected in the patch.



The right outlet sends the actual text messages to **seq**.

In the other patch, the **menu** has a dual purpose of sending values and displaying the values it receives. If you have a lot of different sounds available on your synth, you may not be able to memorize all the program change numbers. A **menu** can help you associate the name of a sound (the text of a **menu** item) with a program change value (the item number).

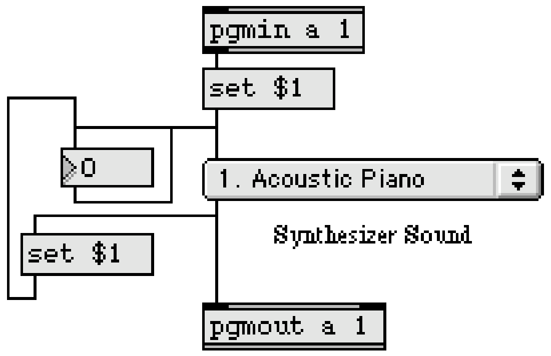
When you select a **menu** item with the mouse, the item number is sent to **pgmout** as a program change value. Just as the names of sounds are specific to a given synthesizer, so may be the numbering system used by the synthesizer manufacturer. You'll need to figure out exactly how MIDI program change values correspond to the sound numbers on your synth. In this example we left menu item 0 empty, and used menu items 1 to 32 to store the names of sounds, so selecting a sound will transmit a program change value from 1 to 32.

- Look at the **menu** Inspector for the *Synthesizer Sound* **menu**. Notice how we left the first **menu** item empty (by starting the text with a comma) so that we could use items 1 to 32.

How did we get more items in the **menu** than will fit into the dialog box? We typed the items in a Text window, then copied them and pasted them into the **menu** Inspector.

- You may want to replace our list of sounds with one that corresponds to your equipment.

We also wanted to display *incoming* program change values, so we directed them to the inlet of our **menu**. However, we don't want the program change to be sent *out* again, so we use the set message to set the **menu** to the specified item without causing any output. Likewise, we want the **number box** to reflect numbers from the **menu**, but we don't want it to send the number *back* to the **menu** because that would cause a *stack overflow*. Once again, the set message is the solution.



- Close the *[menu_examples]* window and double-click on the patcher **preset_examples** object to open the subpatch window.

preset

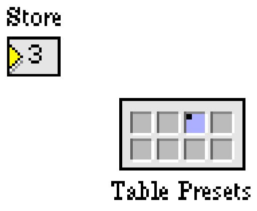
The **preset** object can store and recall the settings of other user interface objects in the same window such as **slider**, **dial**, **number box**, and **toggle** objects. When you recall a stored setting, **preset** restores all these objects back the way they were at the moment the settings were stored. You can even connect the outlet of a **preset** to a **table** to store and recall various versions of the **table** object's contents.

The **preset** can operate in one of three ways. If the left outlet of **preset** is connected to the inlet of other user interface objects, it stores and recalls the settings of *only* those objects. Or, if the right outlet of **preset** is connected to the inlet of other objects, **preset** stores and recalls the settings of all user interface objects in the window except those objects. If neither the left nor right outlet of **preset** is connected to anything, **preset** stores the settings of *every* user interface object in the window (except **table** objects, which can *only* be stored by being connected to the left outlet of a **preset**).

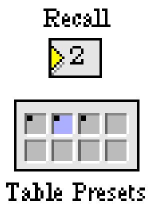
In the *[preset_examples]* window, the **preset** object is actually connected to the **table** with a patch cord, but we have hidden the patch cord for aesthetic reasons.

- Before you use the patch, enable **All Windows Active** in the Options menu. Then double- click on the **table** object to open its graphic editing window. You can draw in pitch values from 0 to 60 (which will be transposed up into the keyboard range by the *Offset* of the lower **hslider**), and then play those pitches by dragging on the upper **hslider**.

- When you have drawn a pitch curve that you like in the Table window, enter a number in the **number box** marked Store, and the **table** values will be stored in that **preset** location.

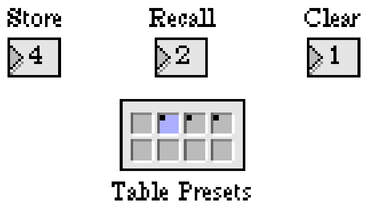


- Repeat the process until you have stored several **table** presets. Then you can recall different ones by entering a number with the **number box** marked *Recall*.



- Now unlock the *[preset_examples]* window to see what's going on behind the scenes.

The **number box** objects labeled *Store*, *Recall*, and *Clear* are actually sending messages to the **preset**.



To store settings in a preset location, you send the message store, followed by the number of the preset location. To recall a preset, just send the number of that preset alone. To clear a preset, send the message clear, followed by the preset number. clearall will clear all stored presets.

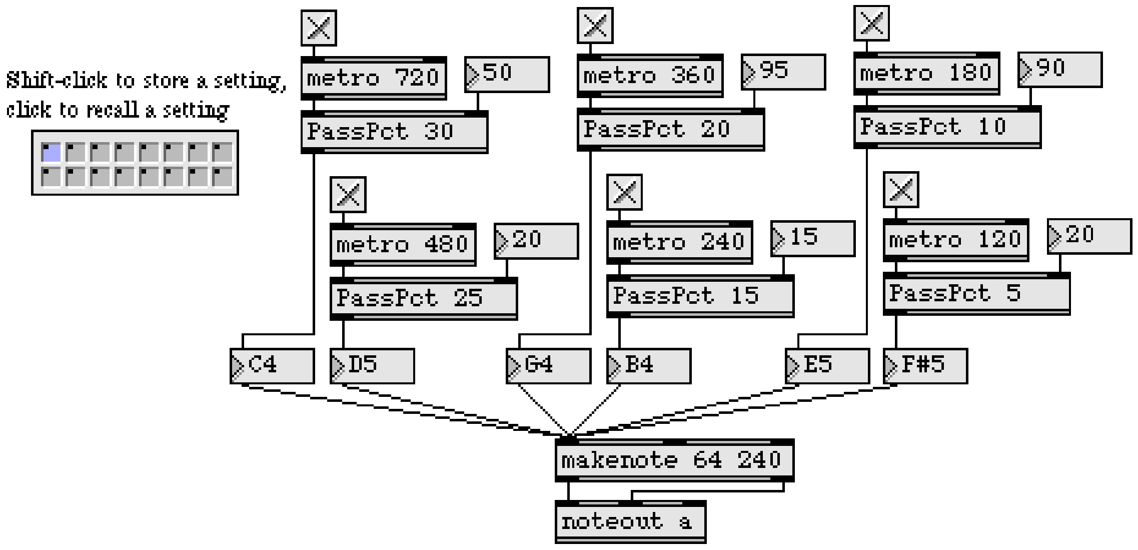
You can save the contents of **preset** in a separate file with the write message, and load a file in with the read message.

- Double-click on the patcher **another_example** object to see the sub-subpatch.

The **preset** object in the *[another_example]* window already has 16 presets stored in it, as part of the patch. To store the contents of a **preset** along with a patch, rather than as a separate file, you select the **preset** object, choose **Get Info…** from the Object menu, and check *Save Presets with Patcher.*

This patch shows you another way to store and recall presets: you Shift-click on a preset to store the current settings, and you can then recall the settings by just clicking on that preset. The number of preset locations in a **preset** object is not dependent on the object's physical size. Each **preset** object holds 256 preset locations, even if they aren't shown within its object box.

- Click on different preset buttons to recall different **toggle** and **number box** settings. Try creating your own repeated note patterns and storing your settings in the **preset** object.



There are no hidden patch cords in this window. When a **preset** is not connected to anything, it stores the setting of *every* user interface object in the window.

Summary

A *data structure* is used to store data so that individual items can be easily accessed.

The **coll** object stores any kind of message, with either a number or a symbol as the address. Data to be stored can be received as messages in the inlet or typed into a **coll** object's text editor window. When **coll** receives an address in its inlet, it sends the address out its second outlet, and sends the message stored at that address out its left outlet.

The contents of a **coll** object can be stored as part of the patch that contains it, or as a separate file. A file can be loaded into a **coll** object with the read message, or by typing the file name in as an argument.

The message sent out by **coll** can be parsed by other objects to select particular items from the data structure. Also, individual data items can be sent out or altered by certain commands in a **coll** object's inlet.

The **menu** object is a pop-up menu in a Patcher window, and the **menu** items (commands) can be any kind of message. The **menu** may be used for selecting commands with the mouse and/or for displaying messages. When a **menu** command is selected, either with the mouse or by a **menu** item number received in the inlet, **menu** displays the command, (optionally) sends the stored message out the right outlet, and (always) sends the item number out the left outlet.

The **preset** object lets you store the settings of every other user interface object in the window at a certain point in time, then recall those settings at some later time. If the left outlet of **preset** is connected with patch cords, to certain objects, **preset** stores and recalls the settings of only those objects. The contents of a **table** can also be remembered by **preset**, but the **table** must be connected to **preset**. The **preset** object can store and recall up to 256 different collections of the settings of all user interface objects.

See Also

coll	Store and edit a collection of different messages
menu	Pop-up menu, to display and send commands
preset	Store and recall the settings of other objects
Data Structures	(**Max Topics**) Ways of storing data in Max

Tutorial 38: **exp** *and* **if**

C Language Expressions

The Max application itself is written in the C programming language, and many of the terms and object names (such as **&&** and **||** for *and* and *or*) in Max have a basis in C. For programmers who have some experience with C or Pascal, and who feel comfortable using traditional programming language syntax, Max provides objects for evaluating mathematical expressions and conditional statements that are expressed in a C-like way.

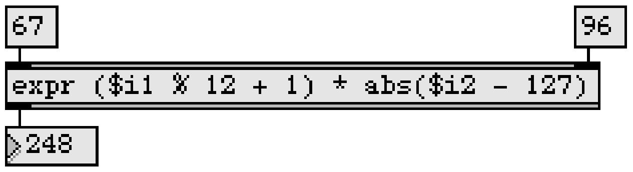
Even if you don't know a programming language, you can understand and use these objects. Often a complex comparison or mathematical calculation that would require several Patcher objects can be expressed in a single phrase, in a single object. Also, you can do a few calculations with these objects that you can't do with any of the other arithmetic operators.

expr

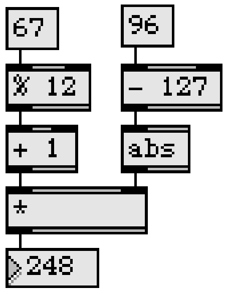
The arguments to the **expr** object make up a mathematical expression (formula) in a format that is similar to C programming language. For example, the C expressions…

*x = 67; y = 96; z = (x % 12 + 1) * abs(y - 127);*

can be expressed in an **expr** object as

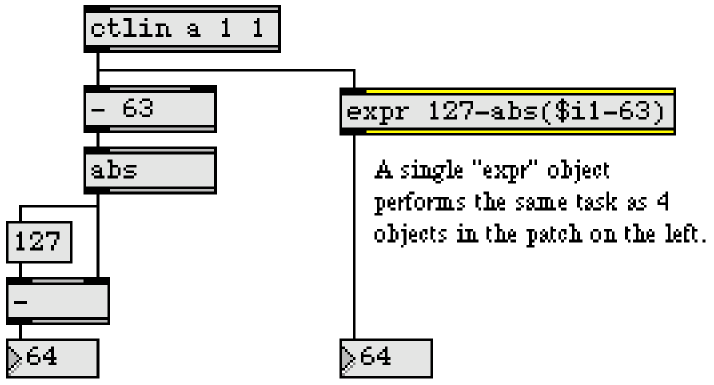


Without using **expr**, you would perform the calculation with a patch with many objects that looks like this:



- To see an example of an object-based solution to a programming problem, and a comparable solution using **expr**, double-click on the **patcher expr_example** object.

For this example, we want to solve the following problem: as the modulation wheel progresses from 0 to 127, send pitch bend values from 64 to 127 and back down to 64. The patch on the left shows a standard Patcher way of doing this. The patch on the right shows the different tasks all combined into a single mathematical expression in **expr**.


ctlin a 1 1
- 63
abs
127
-
64
expr 127-abs($i1-63)
A single "expr" object performs the same task as 4 objects in the patch on the left.
64

Notice that the changeable arguments in an **expr** object include a letter, as in $i1, to tell **expr** what data type to use for that argument (i for int, f for float).

- Move the modulation wheel on your MIDI keyboard from 0 to 127, and you'll see that both methods of stating the mathematical expression work equally well. However, it's a bit more memory-efficient to use a single object instead of four.

Tutorial 38

if

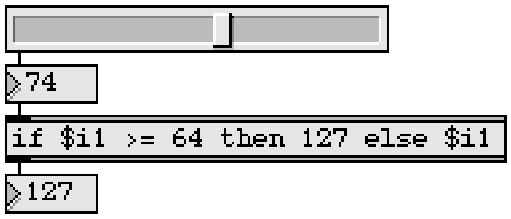
Another staple of C programming is the *if () ; else ;* combination. In Pascal, this is expressed as *IF condition THEN statement ELSE statement*. In plain English this means: if a certain condition is met, do one thing, otherwise do another thing. Sometimes this way of thinking about the world just seems to make a lot more sense than a bunch of boxes connected together with wires!

Max has an object called **if** which lets you express programming problems in an *if/then/else* format. If the comparison in the arguments is true (does not equal 0), then the message after the word then is sent out the outlet, otherwise, the message after the (optional) word else is sent out.

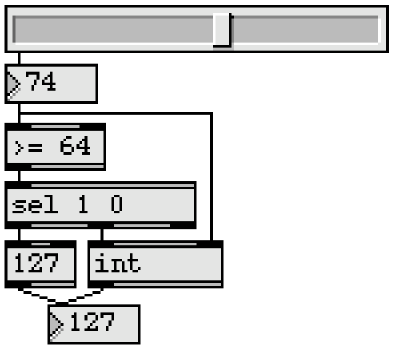
So, the conditional statement

> *if the received number is greater than or equal to 64,*
> *send out 127,*
> *otherwise send out the received number*

would be expressed as

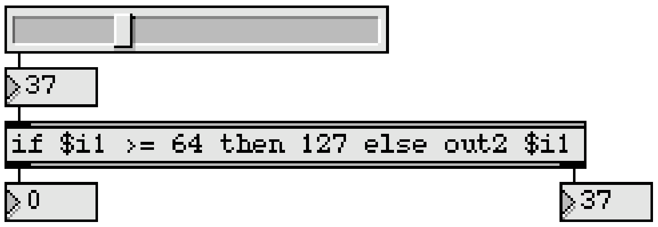


An object-based way of saying the same thing might be:



The then and else portions of the **if** object contain a message similar to that you would type into a **message** box. You can include changeable arguments, but not mathematical expressions as you can in the portion of the message after the if.

If the then or else portions of the **if** object begins with the argument out2, then the object has a second outlet on the right, and the message is sent out the right outlet.



The then portion and else portions can also begin with send, followed by the name of a **receive** object. In that case, the output is sent to all **receive** objects with that name, instead of out the outlet.

- Double-click on the **patcher if_example** object to see the usefulness of **if**.

The problem in this example was, "If the note G1 is played with a velocity between 16 and 95, start a sequence, otherwise increment a **counter** somewhere else." The example shows that a great many tasks can be combined into a single *if/then/else* expression. In this instance, one **if** object does the work of nine other objects.

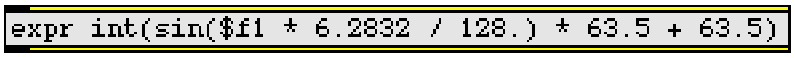
C Math Functions

The C math library has many functions for such calculations as logarithms, trigonometric ratios, *x* to the power of *y*, and so on. Max does not have specific objects for these functions, but they can be included in the arguments of an **expr** object. This is a real strength of **expr**, because it lets you make calculations you would not otherwise be able to make in a Max program.

In the main patch of this example, we use two different math functions, sin() and pow() to calculate pitch bend curves to be stored in the **table**. One formula makes a single cycle of a sine wave with a range from 0 to 127. The other formula draws exponential curves from 64 to 127.

- Check the **All Windows Active** option and double-click on the **table** object to open its graphic window. Click on the **button** at the top of the patch to draw a cycle of a sine wave in the Table window.

The expression in **expr** converts the input to a float by using the $f1 argument (instead of $i1), in order to do a floating point calculation. It divides the input by 128, (so as the input progresses from 0 to 127 it will produce a progression from 0 to almost 1), multiplies the input by 2 (approximately 6.2832), and calculates the sine of that amount. The resulting sine wave values are multiplied by 63.5 and offset by 63.5 to expand them to the proper range, and the final result is converted back to an int before being sent out.



```
expr int(sin($f1 * 6.2832 / 128.) * 63.5 + 63.5)
```

- The expression in the other **expr** is a simple exponential mapping function. Click on **Ggate** to point it to the right outlet. Drag on the **number box** to select an exponent for the curve to be calculated by **expr**. An exponent of 1 produces a straight line, an exponent greater than 1 yields an exponential curve, and an exponent less than 1 yields an inverse exponential curve.

- Click on the **button** to draw the curve in the Table window. Try different exponent values and redraw the curve.

Large numbers of exponential calculations—especially with a large exponent—require fairly intensive processing for the computer to calculate. For this reason it's often better to perform such calculations in advance and store the values in a **table** to be accessed later, rather than to calculate the values on the fly while the computer is performing music.

Once the curve is stored in the **table**, it is read through by a **line** object each time you play a note on your keyboard. The values are sent to **bendout** to be transmitted to the synth as pitch bend. The speed with which **line** reads through the **table** depends on the velocity of the note you play.

- Play long notes on your keyboard with widely varying velocities, and listen to the different speeds with which **line** reads through the curve in the **table**. Draw different curves in the **table** to hear their sonic effect.

Summary

The **expr** object takes a C-like mathematical expression as its argument, including changeable arguments. When a number is received in the left inlet, **expr** replaces the changeable arguments, evaluates the expression, and sends out the result.

The **if** object evaluates a conditional statement in the form "**if** x is true then output y else output z". The conditional statement can contain changeable arguments. The output can be sent to **receive** objects instead of out the outlet.

Tutorial 38

Both **if** and **expr** are capable of combining the computations of several Patcher objects into a single object, which is usually more memory-efficient.

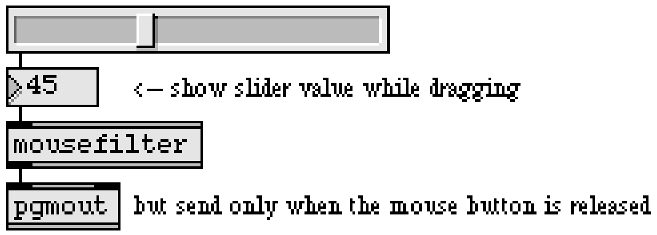
The expression in the argument of **expr** can contain C math functions such as pow() and sin(), and can also contain relational operators. For details on the operators and functions you can use, look under **expr** in the Max Reference Manual.

See Also

expr	Evaluate a mathematical expression
if	Conditional statement in if/then/else form

mousefilter

There may be times when you want to see the exact value that is going to be sent out of a **slider** or **dial** *before* it is actually sent. The **mousefilter** object helps you do that. It receives numbers in its inlet, but passes them on only when the mouse button is up. Consider the example below.


45
<-- show slider value while dragging
mousefilter
pgmout
but send only when the mouse button is released

While you are dragging on the **hslider**, the numbers are sent to the **number box** for display, but **mousefilter** does not pass them on because the mouse button is down. When you release the mouse button, the last number is sent out the outlet of **mousefilter**.

The patch in the left part of the Patcher window is very similar to this example, except that we have hidden the **mousefilter** object and the patch cords. When you drag on the **dial**, the upper **number box** shows the output of **dial**, but no number is sent to the lower **number box** (and to **pgmout**, also hidden) until the mouse button is released.

- Drag on the **dial** to select a new program change number. Nothing is sent to your synth until you release the mouse button.

The **button** at the top-left corner of the patch triggers a **line** object to step through the program changes automatically over the course of 16 seconds.

- Click on the **button**. While the **dial** is being automatically controlled, you can use the mouse to suppress certain program change numbers. Whenever you hold the mouse button down, **mousefilter** acts as a gate to shut off the flow of numbers to **pgmout**.

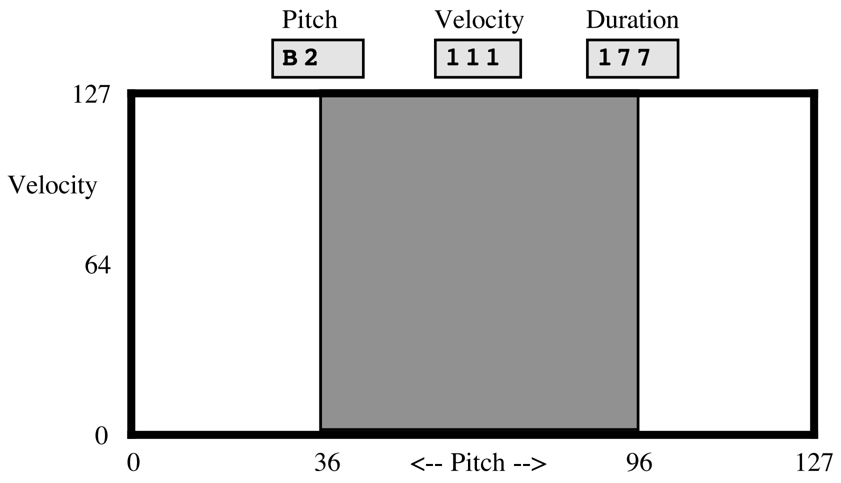
- Unlock the Patcher window to see how the connections are made. Lock the window again before trying the rest of the patch.

Tutorial 39

Using the Mouse Position to Provide Values

The large box in the example Patcher window was imported from a graphics application and pasted into the window using **Paste Picture** from the Edit menu. It delineates a pitch-velocity grid in which you can drag with the mouse to play notes. The gray area shows the pitch space that corresponds to the range of a 61-key keyboard (C1 to C6).

- Before using the pitch-velocity grid, you must click on the most extreme bottom-left corner of the box. This tells Max where the 0,0 point is.
- After you have done that, click and/or drag inside the box to play notes with the mouse.



When you hold down the mouse button inside the box, notes are played continuously. Moving the mouse from left to right in the box increases the pitch. Moving from bottom to top increases the velocity. Large changes upward or downward cause the tempo of the notes to increase or decrease.

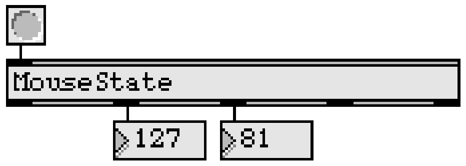
The mouse is not moving any kind of slider or other type of user interface object, so how does it send out notes?

- Unlock the Patcher window and scroll to the right to see what's going on.

mousestate

The generator of numbers in this patch is the **mousestate** object. When it receives a bang in its inlet, **mousestate** report the current horizontal and vertical location of the mouse out its left-middle and middle outlets.

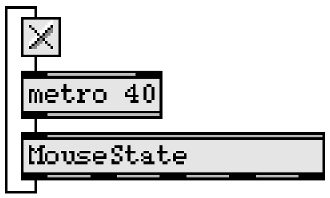
A location on the screen is expressed as a horizontal-vertical pair of numbers, normally measured as the number of pixels away from the upper-left corner of the screen. Horizontal location is measured from left to right, and vertical location is measure from top to bottom.



The cursor is 127 pixels to the right of, and 81 pixels down from, the upper-left corner of the screen

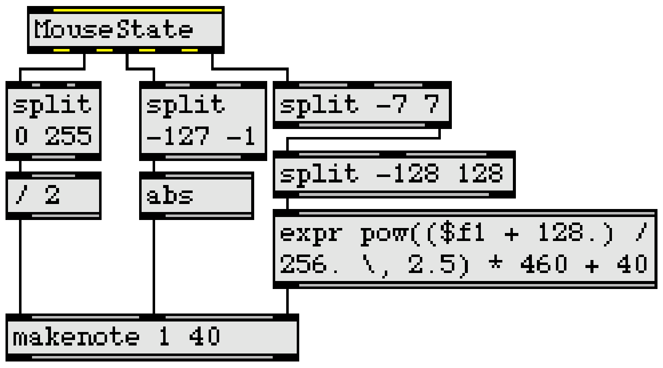
When **mousestate** receives the message zero, it uses the current mouse location as the new 0, 0 point, and makes all subsequent measurements in terms of that new point. That's why you click on the bottom-left corner of the pitch-velocity box before starting. We've situated a tiny **ubutton** object on the corner of the box, so when you click there it triggers a zero message to **mousestate** and sets that point as the new 0, 0 point.

mousestate also reports the status of the mouse button. It sends out 1 when the button is pressed and 0 when it is released. We use this feature in our patch to control the **metro** object which sends bang messages to **mousestate**. Since **metro** only sends a bang when the mouse button is down, **mousestate** only sends out location values while the mouse is down.



The two right outlets report the *change* in location since the previous report. The horizontal location, the vertical location, and the change in vertical location are used in this patch to supply values for pitch, velocity, and tempo. Each range of values has to be limited and processed slightly differently to place the values in an appropriate range.

For example, the pitch-velocity box is 256 pixels wide, so we limit the horizontal location values between 0 and 255 with a **split** object, then divide them by 2 to get a range of pitches from 0 to 127. The box is 128 pixels high, but remember that vertical location is measured from top to bottom, so when the mouse is in the box the vertical values will range from 0 to -127. We therefore limit the vertical values between -127 and -1 and use the **abs** object to make the values positive. (We don't want any 0 velocities because they'll be supplied by **makenote**.)



To get the tempo, we use the change in vertical location of the mouse. But we only want to detect *substantial* change, so we first filter out slight changes (±7 pixels). Then we limit the values between -128 and 128 and use **expr** to map that range onto an exponential curve from 40 to 500. Thus, a large increase in velocity causes a fast tempo, while a large decrease in velocity causes a slower tempo. A gradual change in velocity does not change the tempo.

Summary

When the mouse button is down, **mousefilter** suppresses all numbers it receives until the button is released, then it sends out the last number it received. **mousefilter** can be used as a mouse-dependent gate, especially to allow you to view many numbers but only send on the ones you want.

Every time **mousestate** receives a bang, it sends out the location of the mouse and the *change* in location since the last report. These values can be used to provide continuous musical control, giving you the ability to use the entire screen as a field in which to produce values in two dimensions by moving the mouse.

When the mouse button is pressed **mousestate** sends 1 out its left outlet, and when it is released **mousestate** sends out 0. These values can be used to turn a process on and off, or to open and shut a **gate**.

See Also

mousefilter	Pass numbers only when the mouse button is up
mousestate	Report the status and location of the mouse

Tutorial 40: Automatic Actions

Opening a Subpatch Window

Your programs can automatically open and close Patcher windows and detect when a window is opened or closed, triggering some action.

- When you open the example patch for this chapter of the Tutorial, the window of the subpatch object **stopwatch** is opened immediately and begins to display the time elapsed since the window was opened.

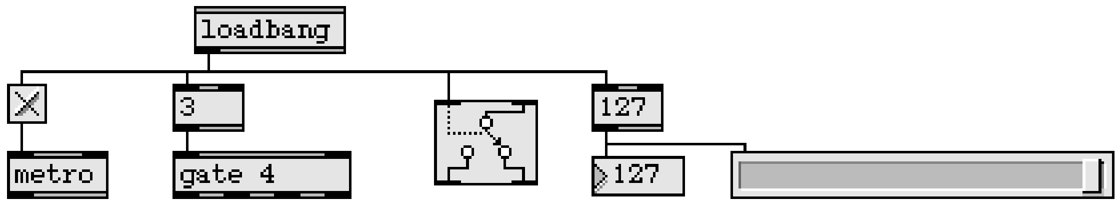
As we explained earlier (in Tutorial 26), a **patcher** object will open automatically if you leave its subpatch window open when you save the Patcher that contains it. A subpatch saved as a *separate file*, however, (such as **stopwatch**) will always have its window closed when the main patch is opened.

To open the *[stopwatch]* window, which would normally not be open, we used two objects, **loadbang** and **pcontrol**.

loadbang and closebang

The **loadbang** object sends out a bang once when the Patcher that contains it is opened (loaded into memory). This allows you to trigger certain actions *immediately* when a patcher is loaded. You can use **loadbang** to open **gate** and **switch** objects (which are closed when a patch is opened), start timing objects such as **metro**, or supply initial number values to an object such as **number box.**

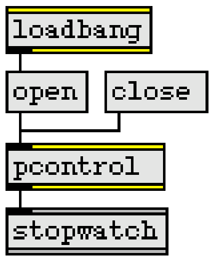
The counterpart to **loadbang** is **closebang** (not shown here) which can be used to trigger actions— such as turning off a **metro** or resetting the contents of a **table**—when a patcher is closed.



pcontrol

Subpatch windows can be opened and closed by the **pcontrol** object. When **pcontrol** receives an open or close message in its inlet, it opens or closes the window of any subpatch objects connected to its outlet.

In the left part of the main Patcher window you can see how the *[stopwatch]* window was opened automatically. The bang from **loadbang** triggered an open message to **pcontrol**, which opened the window of the **stopwatch** object.



Using **pcontrol**, you can produce multi-window patches with each window displaying something different, and you can make **pcontrol** show or hide windows when appropriate.

Note: Because opening and closing windows takes some time, it's not advisable to do it while Max is playing music, unless you're in Overdrive mode.

- You can stop and restart the **stopwatch** by clicking on the **toggle** in the *[stopwatch]* window. To open and close the *[stopwatch]* window, send open and close messages to **pcontrol**.
- Close the *[stopwatch]* window, and open the *[clicktrack]* window by sending an open message to the other **pcontrol** object. When the *[clicktrack]* window is opened, a 4-note

click track automatically begins to play, at the metronomic tempo shown in the **number box**.

The **pcontrol** object can also enable and disable MIDI objects in the subpatch windows it controls. The message enable 0 in the inlet of **pcontrol** disables the MIDI objects in the subpatch, and enable 1 (or any number other than 0) re-enables them. Bear in mind, if you make a patch that automatically disables the MIDI objects in a subpatch, that you run the risk of causing stuck notes on the synth if you cause a note-off message to be lost.

- Enable **All Windows Active** so that you can click in the main Patcher window without bringing it to the foreground. Then click on the toggle in the main Patcher window to disable MIDI. The sound stops, but the **led** continues to flash.

led

The **led** object is an *on/off* indicator similar to **toggle**, but not identical. Whereas **toggle** passes on any number it receives, **led** outputs only 0 or 1 indicating the zero/non-zero status of the number it receives. When **led** receives a bang, it flashes and outputs 0. You can change the color and flash time of an **led** object by selecting it and choosing **Get Info…** from the Object menu.

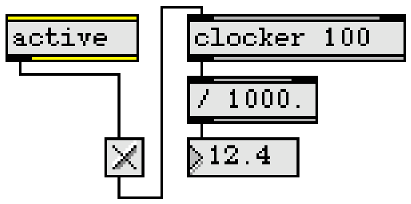
active

What makes the **clicktrack** and **stopwatch** objects run automatically when their windows are brought to the foreground? They are controlled by another automatic control object, **active**.

- To see the hidden objects in the subpatches, you must open the actual file in which the subpatch is saved. Choose **Open…** from the File menu to open the file named *stopwatch*.

- Stop the time display by clicking on the **toggle** in the *stopwatch* window, then unlock the window to see its hidden objects.

When a window is made active (i.e., brought to the front), the **active** object in that window sends out 1. When the window made inactive (is no longer in front) **active** sends out 0. We have used **active** to turn a **clocker** object on automatically whenever the window is brought to the foreground.

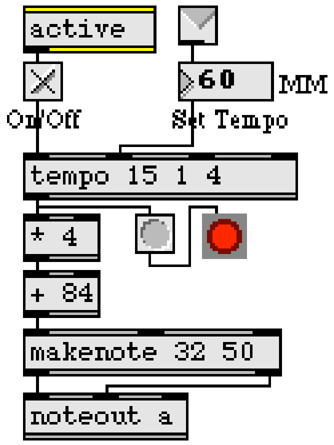


The **active** object sends out a number *only* in response to a change in its foreground/background status, and is not affected by the setting of **All Windows Active**. When **All Windows Active** is checked, you can click in any window without first bringing it to the foreground, but only the foreground window is technically *active*. When you move a window to the background, an **active** object in that window sends out 0, but when you close the window **active** does not send a 0, because it's not actually being sent to the background.

Even though the **stopwatch** object doesn't get any messages from other objects, it needs to have an inlet so that it can be controlled with **pcontrol**. You can include a *dummy* **inlet** object in a patch for this purpose.

- Close the *stopwatch* patch and open the file named *clicktrack*. Turn off the **toggle** in the *clicktrack* window, and unlock the window to see its hidden objects. You can see that it contains an **active** object to turn on **tempo** whenever the window is made active.

The numbers 0 to 3 sent out by **tempo** are multiplied and transposed to play the pitches C5, E5, G#5, and C6.



The **clicktrack** object has one inlet for receiving new tempo values, but this same inlet can be used by **pcontrol** in the main patch to control the *[clicktrack]* window.

Summary

The **loadbang** object sends a bang whenever the patch that contains it is loaded into memory. The **closebang** object sends a bang when the patch that contains it is closed. These bang messages can be used to start processes, open or close a **gate**, send a message, etc.

When the **pcontrol** object receives an open or close message, it opens or closes all subpatch objects connected to its outlet. You can also enable or disable MIDI using the enable 1 and enable 0 messages to the **pcontrol** object. Disabling MIDI objects while a note is being played, however, may cause a note-off message to be lost, leaving a stuck note on the synth.

The **active** object sends out 1 when the window that contains it is brought to the front and 0 when some other window is brought to the front.

See Also

active	Send 1 when window is active, 0 when window is inactive
closebang	Send a bang automatically when patch is closed
loadbang	Send a bang automatically when patch is loaded
pcontrol	Open and close subwindows within a patcher

Tutorial 41: Timeline of Max Messages

Writing a score

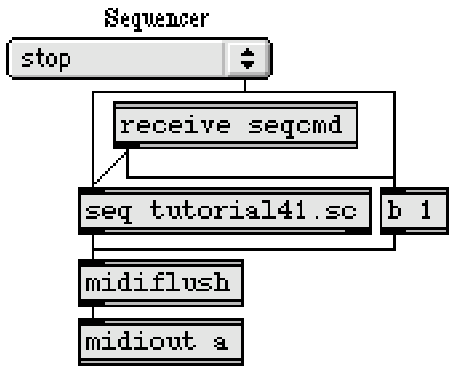
Composers of orchestral music write the activities of all the players out together in a single score, so that all the predetermined events can be seen together, organized in time. Composers of computer music often use a MIDI sequencing program for a similar purpose. In Max, the **timeline** object exists as a combination of score and sequencer.

A timeline in Max is a multi-track sequencer of Max messages. Each *track* in the sequence is a Max Patcher (referred to as an *action* patch), and the *events* that are placed in each track are messages which will be sent to specific objects in the action patch at the desired moment. And just as a prerecorded sequence (or imported MIDI file) can be read into a **seq** object and played from within a patcher, a prerecorded timeline can be read into a **timeline** object and played back from within a patcher.

- In order for the timeline in this Tutorial to work correctly, you should make sure that the Overdrive setting in the Options menu is checked.
- When you open the example patch for this chapter of the Tutorial, two other windows are opened, as well, although they may be hidden behind the Patcher window. One is the graphic editor window for a timeline, and the other is a QuickTime movie window.

(Note: If you don't have the QuickTime extension installed in your system, the QuickTime movie window will not appear, and you should disregard references to the movie when reading this chapter of the Max Tutorial.)

This patch has two main components. On the right side of the window is a **seq** object containing a prerecorded sequence (which was read in automatically from a file named tutorial41.sc). It can be controlled by messages sent from the **menu** object, or by messages received remotely from a **send†seqcmd** object somewhere else. Notice that any message sent to **seq** also sends a bang to **midiflush**, to turn off any notes that may be held at the moment when **seq** is stopped or restarted.



A bang received by midiflush turns off any held notes

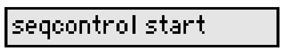
On the left side of the window is a **timeline** object containing a prerecorded timeline (which was read in automatically from a file named *tutorial41.ti*). The other objects around it are for sending it control messages or for handling its output. We'll come back to this portion of the patch presently.

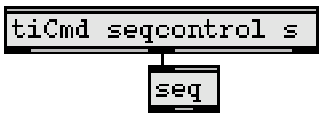
The Timeline Window

- To see the contents of the timeline, bring the timeline graphic editor window to the foreground by double-clicking on the **timeline** object (or by choosing *tutorial41.ti* from the Windows menu).

This timeline has two tracks. Track 1 contains events to be sent to an action patch; track 2 contains only *markers*, which mark specific important points in the timeline. The first track contains a variety of *event editors*, each of which contains one or more events (messages) to be sent to the track's action patch at a specific time. The action patch contains **ticmd** objects, which receive these messages (as if they had come in through inlets) and use them in the patch.

For example, the event editor containing the text seqcontrol start is called a **messenger**; it sends the message start to a **ticmd** object named seqcontrol in the action patch. The **ticmd** is connected to a **seq** object, which will receive the start message from **ticmd**. So, four seconds after the timeline begins to play, a sequence will be started by the seqcontrol start event.





This event in the timeline sends its message... ...out the outlet of the named **ticmd** object in the track's action patch

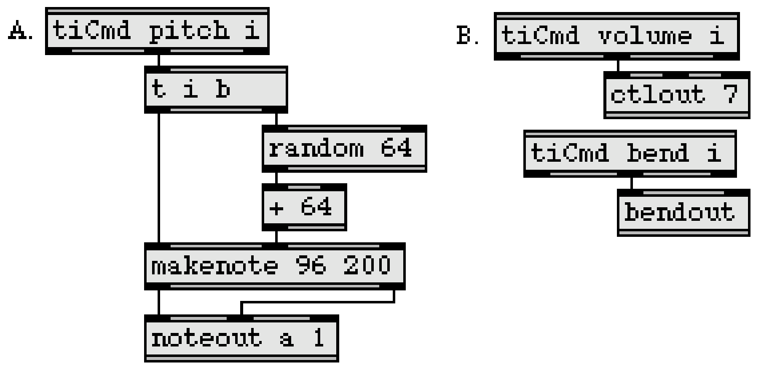
But where is the action patch that will receive these messages and do things with them? The action patch is a Max document on the hard disk, like any other patch you have created and saved. It can be anywhere in Max's file search path. In this case, it's in the same folder as the tutorial patch (the *Max Tutorial* folder). There is also a special folder called *tiAction* in the *Max* folder, where you can keep action patches that the timeline will display in its Track pop-up menu. In any case, the timeline finds the file and loads it into memory to be used by a single track of events. You can view (and even edit) the action patch from within the timeline.

Actions and ticmd

- To see the action patch, double-click on the small Max icon at the left end of track 1.

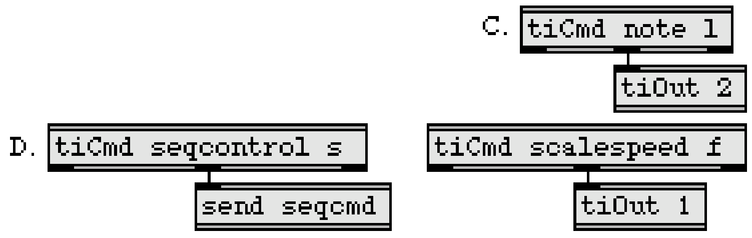
An action patch contains one or more **ticmd** objects, for receiving messages from the timeline. Each **ticmd** object has a name (its first argument), and specifies the type of message(s) it expects to receive. The name of each **ticmd** object in the action will appear as a possible event in the timeline track.

For example, just by looking at portions *A* and *B* of the action, we can see that the timeline may contain events named pitch, volume, and bend which would send int values to their respective **ticmd** objects.



Any volume event in the timeline will be sent out (via the **ticmd** volume i object) as the value of a MIDI controller 7 message, to modify the volume of the synth. Similarly, any bend event in the timeline will be sent out as a MIDI pitchbend message. A pitch event will be played as a 200ms note, with a randomly chosen velocity somewhere between 64 and 127. (A random number from 0 to 63 is chosen, then 64 is added to that number before it is sent to the velocity inlet of **makenote**.)

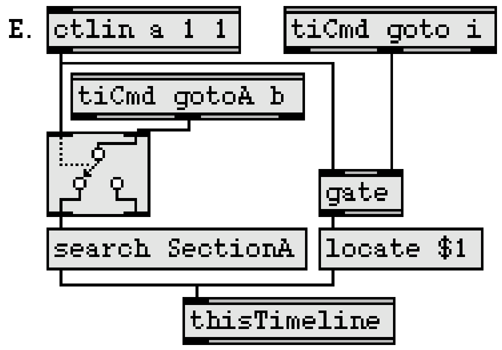
An action patch doesn't need to handle all the events itself. It can simply send them somewhere else, by connecting the outlet of **ticmd** to a **send** object or a **tiout** object, as is done in portions *C* and *D* of this action.



The **tiout** object passes any messages it receives in its inlet out the specified outlet of the **timeline** object itself. So, in this case, note or scalespeed events from the timeline get sent out the outlet of the **timeline** object in the *41. Timeline* patch. (You might find it useful to think of **tiout** objects in an action as analogous to **outlet** objects in a subpatch. They send messages out the outlets of the **timeline** that contains them.) We also see that seqcontrol messages do *not* go directly to a **seq** object in the action; rather, they go to a **send seqcmd** object, so in fact they will come out anywhere that there is an existing **receive seqcmd** object. This is another way that the timeline can communicate with patches other than one of its own action patches.

thistimeline

Let's look at one more feature that's available in an action: the **thistimeline** object. Any message received by a **thistimeline** object in an action gets transmitted to the timeline that contains that action. In this way, a timeline can actually send control messages to itself! In portion *E* of this action, there are two **ticmd** objects, for handling goto and gotoA events from the timeline.



When a gotoA event is reached in the timeline (and the **Ggate** is pointing to the proper outlet), it bangs the search SectionA **message** box, sending that message to the timeline. The timeline will then look for a marker called SectionA, and relocate itself to that marker if it finds it. When a goto event is reached in the timeline (and the **gate** is open), it sends a number (specifying a point on the timeline, in milliseconds) to the locate $1 **message** box, which causes the timeline to relocate to that point. In either case, the timeline will continue to play after it has relocated itself to the new point.

In order to give the user some control over the timeline's behavior, the mod wheel of the synth (controller 1) is used in this action to block or let pass the gotoA and goto messages. Notice that a gotoA message will be passed out the proper outlet of **Ggate** only if the most recently received mod wheel value is 0, and a goto message will pass through the **gate** only if the mod wheel is at some *non-zero* position.

Reading the timeline Score

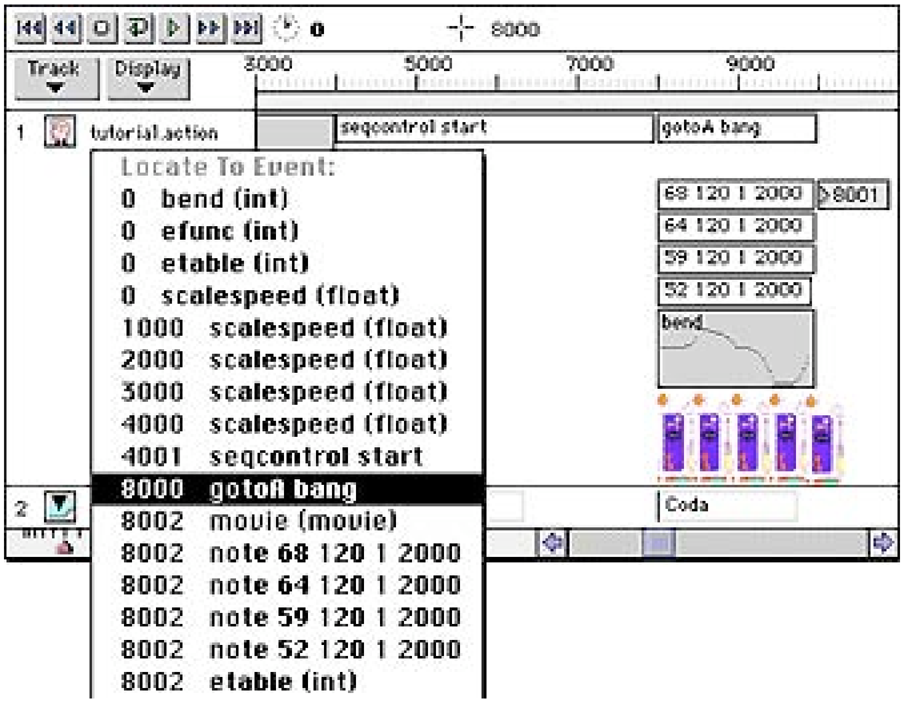
- Close the *tutorial.ac* window so that you can see the timeline editor window again.

Now that you have seen what's going on in the action patch, you can figure out what will happen when the timeline is played. In the first four seconds, there is a whole table full of pitch events, which will be sent out one-by-one over the course of those four seconds. (A table of values is placed as an event in a timeline with the **etable** event editor.) There is also a graph of volume events, which will likewise be sent out continuously over the span of time covered by the event editor (known as an **efunc**).

Four seconds into the timeline, a seqcontrol event will send the message start. We have already seen that this start message will go from the timeline to the **ticmd** seqcontrol s object in the action, to a **send** seqcmd object in the action, to a **receive** seqcmd object in the *41. Timeline* patch, and from there to the **seq** tutorial41.sc object, starting the sequence.

- Scroll to the right in the window to see the remainder of the timeline.

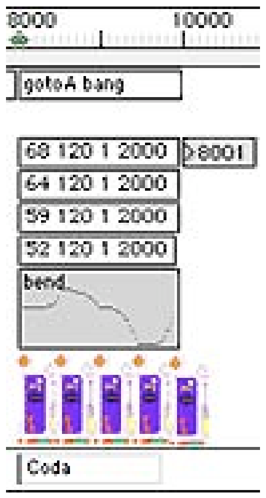
At the 8000 milliseconds (8 seconds) point on the timeline, there appear to be several simultaneous events. You can examine a pop-up menu containing their exact times by holding down the mouse in the left portion of the track, just under the track name.



Locate To Event:
0 bend (int)
0 efunc (int)
0 etable (int)
0 scalespeed (float)
1000 scalespeed (float)
2000 scalespeed (float)
3000 scalespeed (float)
4000 scalespeed (float)
4001 seqcontrol start
8000 gotoA bang
8002 movie (movie)
8002 note 68 120 1 2000
8002 note 64 120 1 2000
8002 note 59 120 1 2000
8002 note 52 120 1 2000
8002 etable (int)

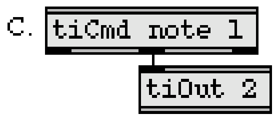
From this list of events you can see that the gotoA bang event occurs just before the other events. You know from examining the action patch that this will cause the timeline to relocate to the SectionA marker (located at time 4000), provided that the mod wheel of the synth is in the 0 position. The timeline will continue to loop from 4000 to 8000 until the mod wheel has been moved to a new position.

When the gotoA bang event is reached, *and* the mod wheel is in a non-zero position, the message will not go out the left outlet of the **Ggate** in the action, so the timeline will be permitted to continue on its normal course. It will then send the note events (from the **messenger** objects), an **emovie** event (a start message that is transmitted directly to the **movie** object in the action), and an **etable** full of bend events (a series of ints sent out one-by-one). At time 10000, it will send a goto 8001 event, thus relocating itself to that point in the timeline (provided that the mod wheel has not been returned to its 0 position).



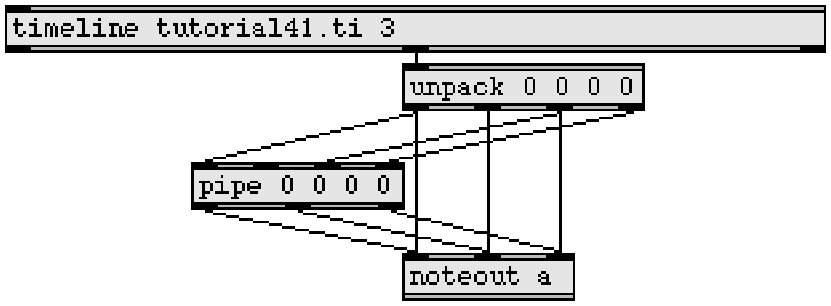
So, at time 8002 the timeline will start the movie, play a four-note chord, and begin bending the pitch; then at time 10000 it will relocate itself to time 8001 and continue playing until the mod wheel is at 0 at time 10000. (Note: because you have the Overdrive option checked—which is necessary for the MIDI data to be sent out with the proper timing—the QuickTime movie may move jerkily or intermittently, depending on the speed of your CPU.)

You may recall that in the action patch the note messages received from the timeline (the four-item lists in the above example) get passed out the second outlet of the **timeline** object.



- Bring the *41. Timeline Patcher* window back to the foreground to see what happens to those note messages.

The lists that are sent from the timeline as note messages come out the second outlet of the **timeline** object, where they are broken up into individual numbers by an **unpack** object. The first three numbers in each note message go directly to a **noteout** object to be used as the pitch, velocity, and channel information of a MIDI note-on message. The 1st, 3rd, and 4th numbers of the note message also go to a **pipe** object, where the 4th number is used as the number of milliseconds to delay before sending on the other numbers.

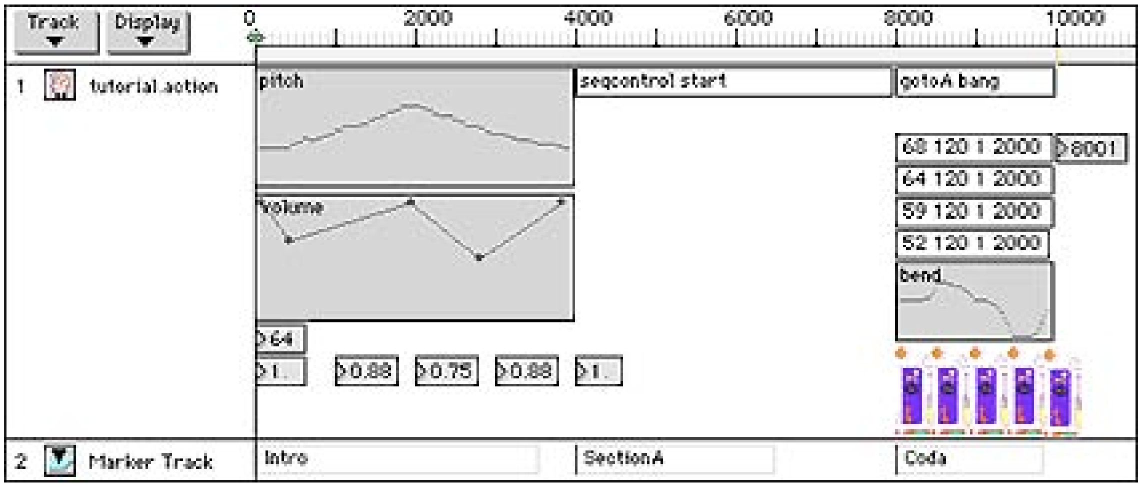


Note that since nothing ever gets sent into the second inlet of **pipe**, the number coming out the second outlet will always be 0. These delayed numbers go to **noteout**, and provide the note-off message. This is a convenient method of providing note-offs, by specifying a note *duration* and using that number to delay a second note message with a velocity of 0. It's similar to using a **makenote** object, but allows you to delay the channel number, as well (which **makenote** does not do).

Playing the timeline

Now that you understand what the different events in the timeline do, you have a pretty good idea what will happen when you play it. In the first four seconds, the notes in the pitch table will be played and the volume will be adjusted by the volume graph. From time 4000 to 8000 the *tutorial41.sc* sequence will be played repeatedly until you move the mod wheel of your synth. Then at time 8002 a chord will be played, pitchbend messages will be sent out from the bend table, and the QuickTime movie will be played.

The time from 8001 to 10000 will repeat until you move the mod wheel back to 0.



- Use the Windows menu to bring the QuickTime movie window to the foreground. (You can drag it to the upper-right corner of the screen so that you can still see the Patcher window.) Check the **All Windows Active** command in the Options menu so that you can leave the movie window in the foreground and still click on objects in the Patcher window.

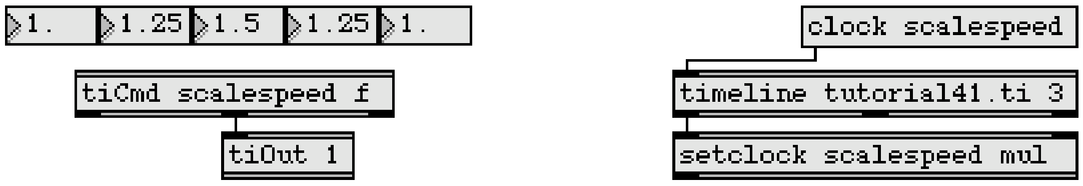
- To play the timeline, just click on the **message** box that says play in the Patcher window. When you get bored with the repeating sequence in *SectionA*, move the mod wheel and the timeline will progress on to the *Coda* section. To stop the timeline, click on the stop message in the Patcher window. To go back to the beginning, click on the locate 0 **message** box, or choose Intro from the *Go To* pop-up **menu** in the Patcher window.

Controlling the timeline's Tempo

Like the **clocker**, **line**, **metro**, **pipe**, and **tempo** objects, a **timeline** object can be synced to a **setclock** object, and its tempo will then be controlled by that **setclock** object rather than by Max's regular millisecond clock.

- Click on the **message** box that says clock scalespeed. That instructs the **timeline** object to sync to the **setclock** scalespeed mul object. Click on the locate 0 **message** box, to "rewind" the timeline, then click play. You will notice some changes in the tempo of the Intro section.

Whenever a **setclock** object with a mul argument receives a number in its left inlet, it multiplies its clock values (i.e., divides its tempo) by that number. In this case, the tempo changes come from the **timeline** itself. Specifically, scalespeed event editors (**float** objects) in the *Intro* section of the timeline transmit numbers to the **ticmd** scalespeed f object in the action patch, which sends them (via a **tiout** 1 object) out the first outlet of the **timeline** object.

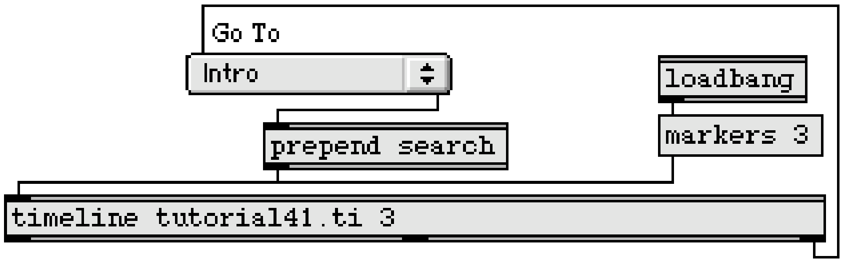


scalespeed events in the timeline track are received by the ticmd object in the action patch, and are sent out the first outlet of the timeline object to setclock which changes the timeline object's tempo

The above example is a rather complex situation, which is included here primarily in order to demonstrate **timeline** object's ability to control itself, and to demonstrate the operation of the **tiout** and **setclock** objects. However, the numbers that go into **setclock** to change its tempo could come from any source, such as a **slider** or a MIDI controller (with the proper arithmetic to map the numbers to an appropriate range of floats).

- If you want to revert **timeline** to following Max's regular millisecond clock, click on the **message** box that says clock.

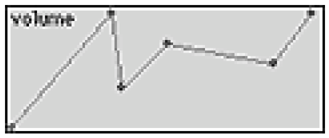
There is only one remaining part of the patch that has not yet been explained. When the **timeline** receives the message markers 3 (as it does when the patch is loaded), it sends the first word from each of its markers out its third outlet, to be stored in the **menu** object. This **menu** can then be used to cause **timeline** to go immediately to any of the markers.



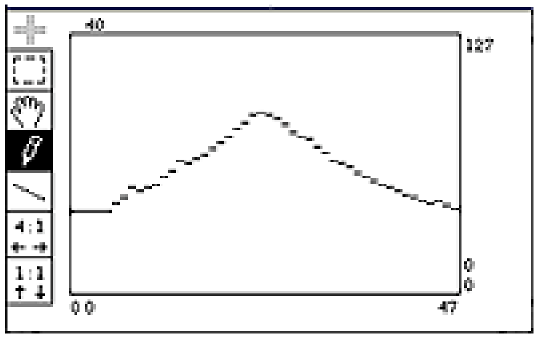
- Try using the **menu** to jump to a specific section in the timeline. You can do this while the timeline is playing.

Editing the timeline

The timeline in this tutorial example has already been arranged and saved in a file named *tutorial41.ti*. You can make changes to the timeline by bringing the graphic editor window to the foreground. For example, you can change the volume graph in the *Intro* section just by clicking and dragging on the control points in the graph, or by clicking where no control point exists to create a new one.



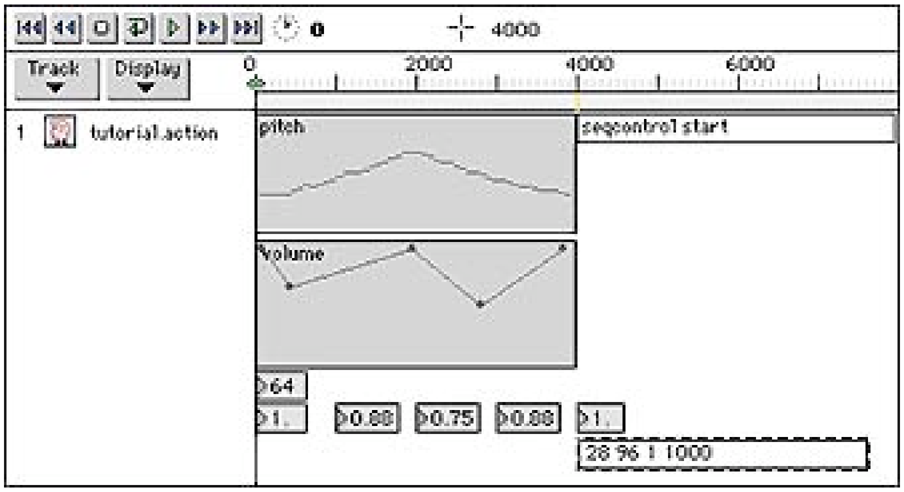
- Double-click on the **etable** editor of pitches in the *Intro* section. You will be presented with a table editing window, and you can change the values in the pitch table.



Notice that the table editing window has a title: *intropitches*. That's because this particular **etable** has been *linked* to the **table** intropitches object in the action patch. When you create an **etable** (or **efunc**) event editor in a timeline track, you can link it to an existing **table** object (or **funbuff** object in the case of **efunc**) by selecting it, choosing the **Get Info...** command from the Object menu, and typing in the name of the object as the *Table Label* for your editor. From then on, any changes you make in the **etable** will affect the **table** object to which it is linked, and vice versa.

To place new events in a timeline track, you hold down the Option key on Macintosh or the Alt key on Windows and click on the mouse in the event portion of the track at the point where you want to place the event. You will be presented with a pop-up menu of all the possible events you can place in that track, based on the **ticmd** objects in the action that the track is using. If there is more than one possible editor for a particular event (for example, an event of type int can be placed using an **int**, **etable**, or **efunc** editor), the editors are presented in a submenu. You choose the event you want from the pop-up menu, then enter the message you want that event to send to the **ticmd** object.

- Try placing a note event in track 1 at time 4000. Move the mouse in the event portion of the track until the indicator at the top of the window tells you that your cursor is at time 4000. Option-click on Macintosh or Alt-click on Windows in the track (in some white space where there are no other events in the way) and choose a note event from the pop-up menu. You will get a **messenger** event editor, into which you can type the note information. As we have seen, a note event should be a four-item list in the format pitch-velocity-channel-duration, so type in 28 96 1 1000 to play a low E on channel 1 for 1 second.



You can play your timeline without leaving the graphic editing window, by using the tape recorder style controls at the top of the window.

Summary

A timeline is a multi-track sequencer, each track of which sends messages to **ticmd** objects in a specified *action* patch. You place *events* (messages) in a track in non-real time by Option-clicking on Macintosh or Alt-clicking on Windows at the desired location on the timeline. The messages come out the outlet of the **ticmd** object in the action patch, and can either be used inside the action patch or sent elsewhere via a **send** object or a **tiout** object.

Once the "score" of Max messages has been composed on the timeline, it can be saved in a file, and then can be accessed from a patcher by reading the file into a **timeline** object. You play the timeline by simply sending a play message in the inlet of the **timeline** object. You can also move to a specific time location in the timeline with the locate message, or by searching for a *marker* event with the search message.

An action patch can send messages out the outlet of the **timeline** object that contains it, via the **tiout** object. An action can also control the timeline that contains it, via the **thistimeline** object. The tempo of a **timeline** can be controlled in real time by syncing it to a **setclock** object and sending messages to the **setclock** (possibly even from the **timeline** itself).

See Also

setclock	Modify clock rate of timing objects
thistimeline	Send messages from a timeline to itself
ticmd	Receive messages from a timeline
timeline	Time-based score of Max messages
tiout	Send messages out of a **timeline** object
Timeline	(**Max Topics**) Creating a graphic score of Max messages

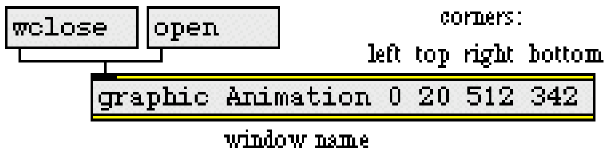
Tutorial 42: Graphics

The Graphics Window

In Max you can state the vital information about a musical note in terms of integers specifying key number, velocity, channel, and (with **makenote**, for example) duration in milliseconds. Max also allows you to place pictures and geometric shapes of color onscreen, using integers to state the position, size, priority (foreground-background level), and color of the images. Since both sounds and images are described with integers, it's a simple matter to write patches that correlate the two.

In order to display animated graphics, you need to include at least one **graphic** object in your patch. Each **graphic** object opens a graphics window automatically when the patch is opened.

- When you open the example patch for this chapter of the Tutorial, a graphics window titled is opened by the **graphic** object.



The first argument gives the graphics window a name, which appears in the title bar of the graphics window. (In this case, the graphics window's title bar is hidden behind the menu bar.) Other objects will use the window name to refer to the window in which they are going to draw.

The four number arguments following the window name specify the four corners of the drawing area of the window—top, left, right, and bottom—in terms of pixels from the top left corner of your screen. We have made the window precisely fill a 9" screen, leaving twenty pixels at the top for the menu bar.

The **graphic** object can receive open and wclose messages. The wclose message is particularly helpful in a case like this, where the close box is hidden behind the menu bar. Obviously, the open message is necessary to reopen the window once it has been closed, and it can also be used to bring the window to the foreground. We have also used the **key** object to include keyboard shortcuts o and w for open and wclose, since the graphics window completely covers the Patcher window once it has been brought to the foreground.

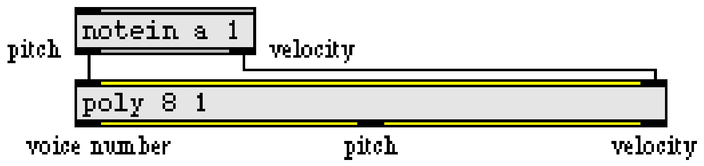
Tutorial 42

Drawing Shapes

- Use the open message, or the o key on your keyboard, to bring the *Animation* window to the foreground. Play some notes on your keyboard and watch what happens in the graphics window. Analyze the correlation between your actions and the graphics onscreen.

- Choose *42. Graphics* from the Windows menu, or use the *w* key on your keyboard to close the *Animation* window. Double-click on the **patcher** Eight Rectangles object to examine its contents.

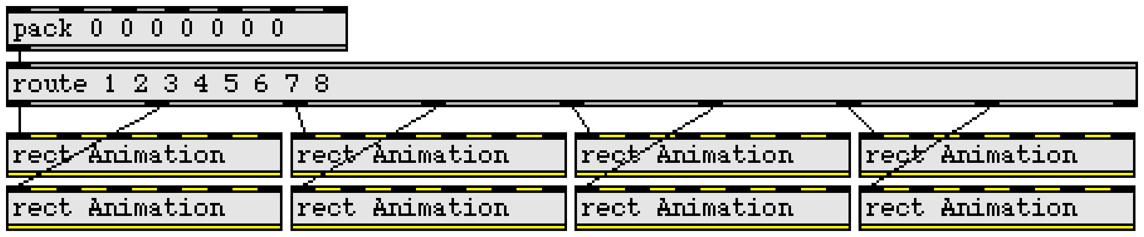
The played pitches and velocities are passed through a **poly** object, which assigns a unique *voice number*, 1 through 8, to each note currently being held. The pitch and velocity are passed out the middle and right outlets, and the voice number is sent out the left outlet. If more than 8 notes are held down at a time on the keyboard, **poly** sends out a note-off message for the oldest note to make room for the newest note. This is known as *voice-stealing*. The first argument tells **poly** how many notes to hold, and the second argument (if non-zero) tells **poly** to steal voices.



The pitch and velocity of the note are used to determine characteristics of the rectangles to be drawn in the graphics window. The voice number is used to route messages to one of eight different **rect** objects.

Shapes and pictures are animated in a graphics window as *sprites*, objects that draw themselves in a single place and erase themselves from their old location when they are drawn somewhere else. Each shape-drawing object such as **rect** controls a single sprite, so multiple objects are needed if you want to display more than one shape at a time. We chose eight **rect** objects as a reasonable number to take care of most keyboard playing styles.

The **rect** object requires an argument telling it which graphics window to draw in. It has inlets for specifying the coordinates of its four corners—left, top, right, and bottom—relative to the top left corner of the graphics window's drawing area. It also has inlets for the sprite's pen mode (for a list of pen modes, see **oval** in the Max Reference Manual) and color. We use the incoming MIDI data to calculate these characteristics of the shapes to be drawn, and we pack the numbers for all six inlets as a list, combined with the voice number at the beginning of the list, so that we can route an entire rectangle description to the appropriate **rect** object.

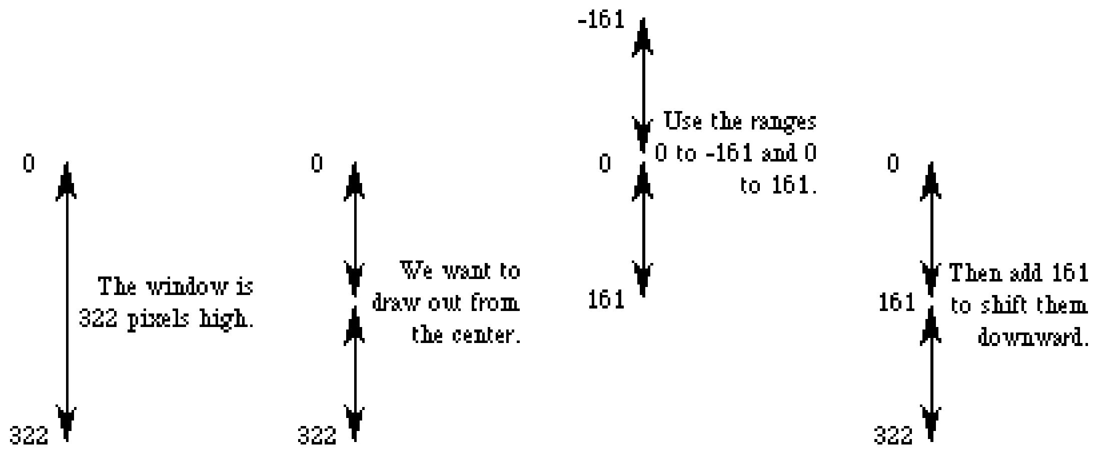


We use only **rect** objects for drawing shapes in this patch, but the inlets of the **frame**, **oval**, and **ring** objects are exactly the same.

Correlating Graphics and MIDI

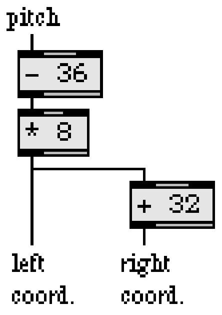
You can make any correspondence you like between MIDI data and graphics data. The most straightforward solution of the matter is simply to map one range of values to another. In this patch we use velocity to calculate the height of the rectangle, pitch to calculate the rectangle's placement from left to right, and pitch class (C, C#, D, etc.) to determine its color.

Velocities range from 0 to 127, and the vertical range of pixels in the drawing area is from 0 to 322 (342 - 20 = 322). We made the decision to center all the rectangles vertically in the drawing area, so we want to calculate the height of the rectangle as a distance up and down from the center. This means that in fact we want to use the vertical range 0 to 161 (322 ÷ 2 = 161) and 0 to -161, then offset the rectangle downward by 161 pixels.

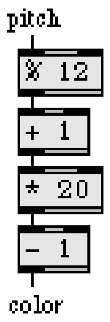

0
322
The window is 322 pixels high.
0
322
We want to draw out from the center.
-161
0
161
Use the ranges 0 to -161 and 0 to 161.
0
161
322
Then add 161 to shift them downward.

To convert the velocities to the proper range—0 to -161 or 0 to 161—we multiply by -1.27 or 1.27, then add 161. The resulting values are sent to **pack** to be stored in the locations for the top and bottom coordinates of the rectangle. Note that when the velocity is 0, the height of the rectangle will be 0; both the top and the bottom coordinates will be 161. This causes the rectangle to disappear when the note is released, because it's drawn with a height of 0.

The effect of pitch on the horizontal coordinates of the rectangle is calculated in a similar manner. The played pitches will range from 36 to 96, and the horizontal range of pixels is from 0 to 512. We first subtract 36 from the pitch to bring it into the range 0 to 60. Then if we offset each key of the ascending scale by 8 pixels to the right, and make each rectangle 32 pixels wide, the notes of the keyboard will precisely span the graphics window.



There are 256 available colors available to the shape-drawing objects, numbered 0 to 255. Using the modulo operator **%**, we can determine the pitch class as a number from 0 to 11. We add 1 to each pitch class value, to put it in the range 1 to 12, then we assign each pitch class a color by multiplying it by 20 to distribute it in the range 20 to 240. Finally, we subtract 1 from it, since the odd numbered colors show up as black on monochrome monitors. (If we don't do that, they will all be drawn in white on a monochrome monitor, and will be invisible.)



Because the pitch and velocity values come out of **poly** before the voice number, the rectangle characteristics can all be calculated and stored in **pack** before the voice number triggers the message and sends it to **route** to pass it to the correct **rect**.

- Close the *Eight Rectangles* window and open the graphics window again, then play some notes to verify that the rectangles behave as described.

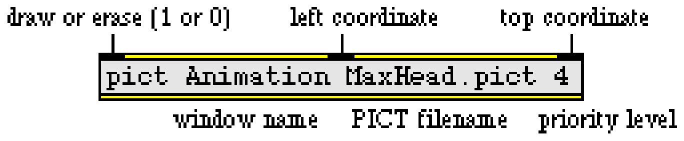
Animating Pictures and Shapes

To give the illusion that a sprite is moving, we simply draw it several different places in rapid succession, progressing along a particular trajectory. Any source of a continuous stream of numbers can therefore be used to control an animation—the pitchbend wheel, the mod wheel, a volume pedal, a **counter**, a **clocker**, a **line**, etc. In this patch we use a **line** object to move a picture along a straight line. The same principle can be applied for moving shapes.

- Close the graphics window again, and double-click on the **patcher** Moving Picture object.

A **pict** object loads an entire graphics file and displays it in a graphics window. Since it loads and displays the entire file, you will usually want to make sure that your image is tucked as far as possible into the top left corner of your graphics file, so that the file is no bigger than it needs to be and has no superfluous white space around the edges.

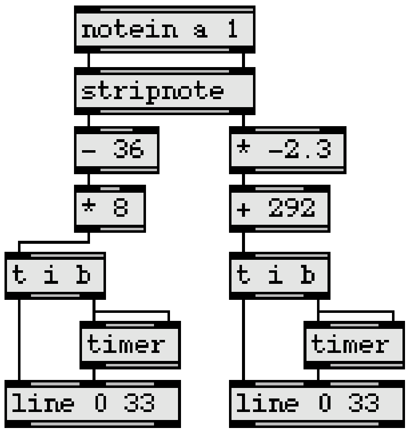
The first argument of a **pict** object is the name of the graphics window in which you want the picture to be shown. The second argument is the name of a graphics file to show. The file must be located in Max's search path; if Max can't find the file, it just prints an error message in the Max window and displays nothing.



The third argument is the sprite's *priority*. The higher a sprite's priority, the closer to the foreground it is considered, and it will be shown in front of sprites that have a lower priority. The default priority of a **pict** object is 0, while the default priority of a **rect** is 3, so by default a **rect** will cover a **pict**. We give our **pict** a higher priority so that the picture will be drawn in front of the rectangles.

Because the size of a picture is predetermined by the dimensions of the graphics file, you only need to give **pict** two coordinates to situate the picture—the coordinates of the left top corner. A non- zero number or a bang in the left inlet draws the picture at the specified spot.

In the example patch, **line** objects are used to change the left and top coordinates continuously on a trajectory toward a specified goal. The left coordinate goal of the picture is calculated from the pitch of the played note, just as in the case of the rectangles. The picture's distance from the bottom of the window is determined by mapping the range of note-on velocities (1 to 127) to the range of vertical pixels (going up, 322 to 0). Because the picture is 32 pixels high, the effective vertical pixel range is 290 to 0. Multiplying the velocities by -2.3 causes them to range from 2 to -290, and adding 292 to that gives us the desired pixel range.



The amount of time that each **line** object takes to move the picture to the target coordinates is determined using **timer** objects that measure the elapsed time since the previous note-on. The interpolation resolution of 33ms was chosen to animate the picture at a potential rate of 30 "frames" per second. The actual rate at which the image is redrawn will depend on the speed of your computer.

Summary

Pictures and colored shapes can be drawn in a graphics window, which is created by placing a **graphic** object in your patch. The name argument given to the **graphic** object is also given to any object that draws in its window. The objects **frame**, **oval**, **rect**, and **ring** are used to draw geometric shapes into a graphics window. The **pict** object loads an entire graphics file into memory and displays the picture at any specified location in a graphics window.

Each image in a graphics window is a *sprite*, which you can move around by redefining its coordinates, and which is assigned a *priority* that determines whether it will be drawn in front of or behind other sprites. You can animate sprites in such a way as to give the illusion of continuous movement by redrawing them in rapid succession in different locations along a chosen trajectory. Any continuous stream of numbers may potentially be used to describe such a trajectory.

Tutorial 42

The parameters and location of the shapes and pictures drawn in the graphics window can be easily correlated to MIDI data to create the desired correspondence between sound and images. This is usually achieved by multiplying a range of values by some factor to make them appropriate for use both as MIDI data and as pixel locations.

The **poly** object assigns a unique voice number to each note currently being held. This voice number can be used to route the note information to different locations, such as different drawing objects.

See Also

frame	Draw framed rectangle in graphics window
graphic	Open a graphics window
oval	Draw solid oval in graphics window
pics	Animation in graphics window
pict	Draw picture in graphics window
rect	Draw solid rectangle in graphics window
ring	Draw framed oval in graphics window
Tutorial 43	Graphics in a patcher
Graphics	(**Max Topics**) Overview of graphics windows and objects

Tutorial 43: Graphics in a Patcher

Animation in a Patcher Window

In order for this Tutorial patch to function correctly you need to make sure QuickTime is installed in your system. You should also disable Max's Overdrive option to give more of the computer's attention to screen drawing activities.

In Tutorial 19 it was pointed out that you can customize the user interface of your patch by importing pictures from other programs. In this chapter we demonstrate various ways you can change the contents of a Patcher window dynamically, and even include animation right in the Patcher window.

In the patch *43. Graphics in a Patcher* you see two new large object boxes in the bottom of the screen. One is the object **imovie** for playing a QuickTime movie in a Patcher window, and the other is **lcd** for drawing lines, shapes, and text. The **patcher** objects contain subpatches that control these objects.

There are a few other objects that are invisible to you in this patch, not because they have been hidden with the **Hide On Lock** command, but because they have no visible borders. These objects are **bpatcher** and **menu** (in *Label* mode), which are discussed later in this chapter.

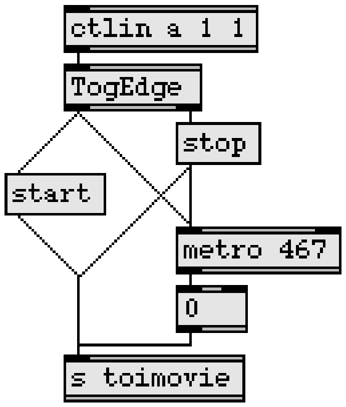
Playing a QuickTime Movie

- Move the modulation wheel on your synth to a non-zero position.

While the mod wheel is in a non-zero position, the movie in **imovie** plays in a loop. This particular movie is only fourteen frames long, so it lasts a little less than half a second. In those fourteen frames there are only four *different* frames, so the effective frame rate is only about eight frames per second, which explains why the motion is rather jerky.

By selecting the object and choosing **Get Info…** from the Object menu, then choosing a QuickTime movie file from the dialog box, you tell **imovie** what movie to read in when the patch is loaded. **imovie** responds to various control messages, most notably start and stop, which are the only messages we use in this example.

- Stop the movie by returning the mod wheel to its zero position. Double-click on the patcher **playmovie** object to see how the movie is being controlled.



Contents of the patcher playmovie object

A **togedge** object is used to detect changes in the zero and non-zero status of the mod wheel. It filters out the numerous non-zero numbers the mod wheel might generate, and reacts only to a change in its zero/non-zero status. It starts the movie and uses the **metro** to rewind it to time 0 every 467 milliseconds. 467 milliseconds = $^{14}/_{30}$ second (14 frames at 30fps). Setting the **imovie** object's time location with a number while the movie is playing, as is done here, causes the movie to continue playing from that point.

The control messages are sent to **imovie** via a **send** and **receive** pair. The **r** toimovie object is hidden in the main patch.

Drawing with the lcd Object

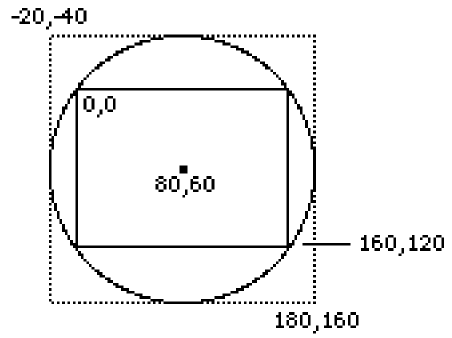
In Tutorial 42 you learned how to draw colored shapes with sprites in a graphics window. The **lcd** object lets you paint shapes, lines, and text in a Patcher window, not with sprites but with commands. The principles of specifying the colors and coordinates of the shapes are very similar in these two cases.

- Close the subpatch window *[playmovie]*, and double-click on the **patcher** concentrics object to see its contents. Play the low C on your keyboard (key 36) once to set the *[concentrics]* subpatch into action.

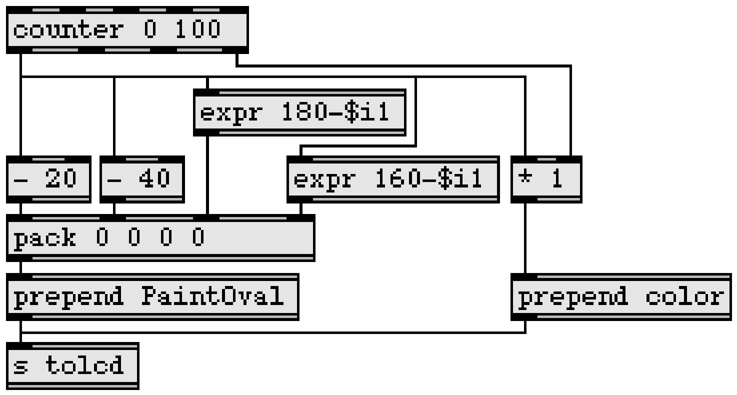
The note toggles a **metro**, which increments a **counter** cycling from 0 to 100 about every two seconds. The numbers from the **counter** are used to calculate the color and coordinates of concentric circles to be painted with the PaintOval message to **lcd**.

Let's examine how to calculate the coordinates for these concentric circles which are precisely centered within the **lcd**. This particular **lcd** object has been sized to be 160 pixels wide by 120 pixels high. A little trigonometry reveals to us that the distance from the center of this **lcd** to one of its corners is equivalent to 100 pixels, so the entire **lcd** can be circumscribed by a circle with a radius of 100.

Since we know that the dimensions of the **lcd** are 160x120, we can easily calculate that the center point is at the coordinates "80, 60" relative to the left top corner of the **lcd**. We can then calculate that a perfectly centered circle with a radius of 100 would be bounded by a square with the coordinates -20, -40, 180, 160.



So, to create a progression of diminishing concentric circles, we want the coordinates of the circles' bounding square to progress from -20, -40, 180, 160 (a circle of radius 100) to 80,†60,†80,†60 (a circle of radius 0) as the counter progresses from 0 to 100.



The calculated coordinates are packed as a list, the word PaintOval is prepended to that list, and the entire message is sent to **lcd** via **s** tolcd and **r** tolcd (hidden in the main patch).

The color with which the **lcd** will paint is specified by the word color followed by a number from 0 to 255. If a color number greater than 255 is received, it is automatically "wrapped around" with a modulus operation to keep it in the correct range. This modulus feature is taken advantage of in the [concentrics] patch. The numbers from the left outlet of **counter** are multiplied by the carry count from its right outlet (the number of times the **counter** has reached its maximum). The result is that as each circle is painted, the **lcd** object's pen color is incremented first by 1's, then by 2's, then by 3's, and so on. Even though these numbers quickly exceed the range of acceptable colors, **lcd** keeps them within range automatically. The fact that the color value is always being incremented by a different amount makes the color pattern of the circles constantly change.

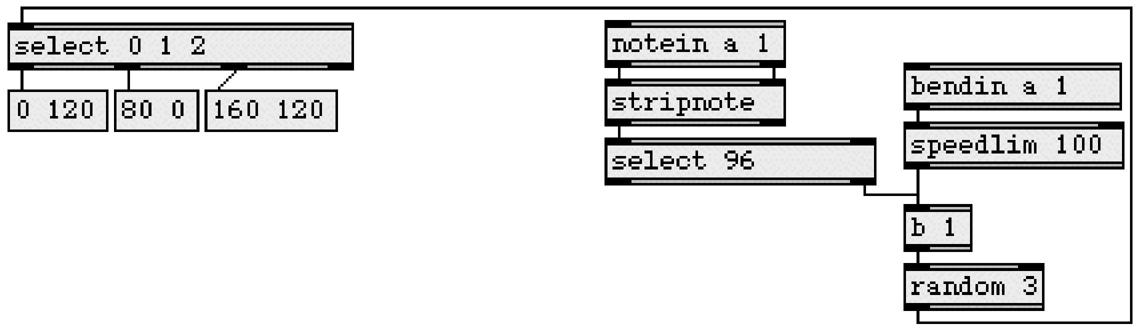
Drawing a Chaotic Image

- To see another example of drawing in **lcd**, play notes on your keyboard and/or move the pitchbend wheel.

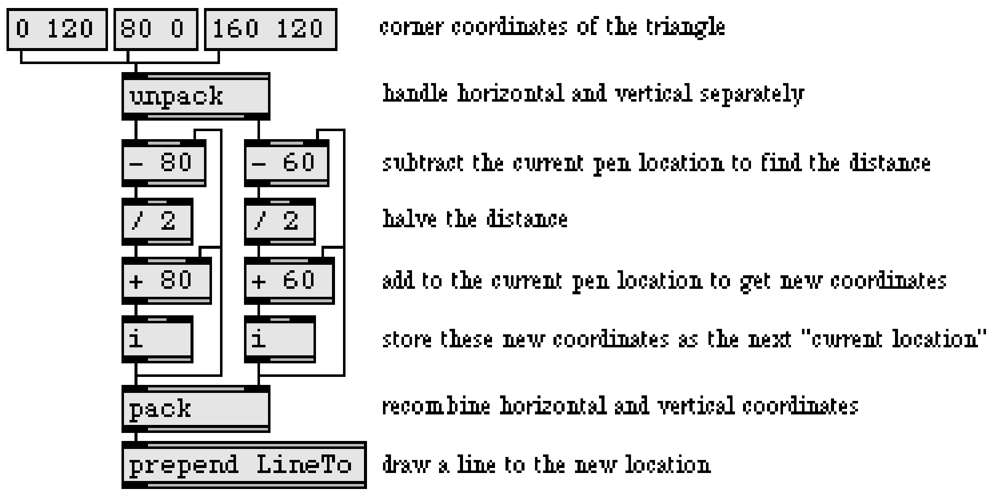
The MIDI notes and pitchbends draw lines in the **lcd**. As you draw more and more lines, you will notice that they are filling in an isosceles triangle in an unpredictable but fairly coherent pattern. Each line segment is drawn by moving the pen exactly half the distance from its current position towards a randomly chosen corner of the triangle. This is one of many interesting algorithmic patterns proposed by the mathematician Waclaw Sierpinski.

- Close the *[concentrics]* window and double-click on the **patcher** Sierpinski object to open it.

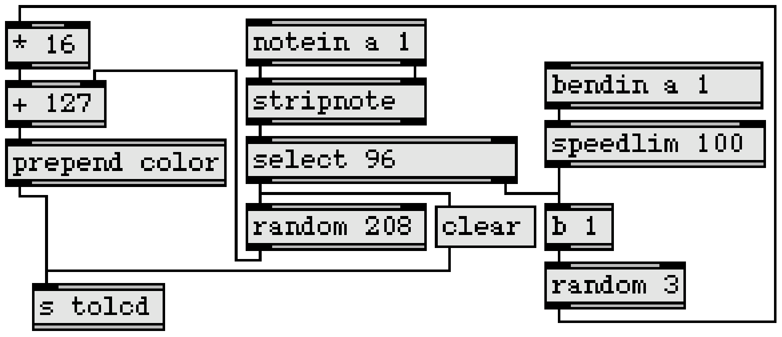
The note-on messages and speed-limited pitchbend messages are converted to bang messages with a **b** object (shorthand for **bangbang**), and trigger one of three random numbers which designate the coordinates of the corners of an isosceles triangle.



The current pen coordinates are subtracted from the coordinates of the chosen corner, and that distance is divided by 2 to determine the length of the line segment. The length is added to the current pen location to determine the endpoint of the line segment. A line is drawn from the current location to that endpoint, and the endpoint is stored as the new "current location".


0 120
80 0
160 120
corner coordinates of the triangle
unpack
handle horizontal and vertical separately
- 80
- 60
subtract the current pen location to find the distance
/ 2
/ 2
halve the distance
+ 80
+ 60
add to the current pen location to get new coordinates
i
i
store these new coordinates as the next "current location"
pack
recombine horizontal and vertical coordinates
prepend LineTo
draw a line to the new location

The random number is also used to designate a color for the **lcd** object's pen, so each line is one of three colors, depending on which corner of the triangle it is drawing toward. When the note 96 (the highest C on the keyboard) is received, the contents of the **lcd** are erased with a clear message and three new colors are chosen by putting a new random number into the right inlet of the **+** object.

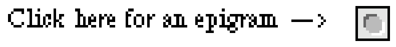


Displaying and Hiding Text

It is possible to display changing text messages that don't seem to be contained in Max objects, by using a containing object that has no borders.

One method is to display messages in a **menu** object that is in *Label* mode. A **menu** is put into *Label* mode by sending it the message mode 3, or by selecting it and choosing **Get Info…** from the Object menu and setting its mode to *Label*. Once this has been done, the menu displays no borders and does not respond to the mouse. Sending the **menu** an item number displays a new text message, and if you leave an empty item in the **menu** you can hide it entirely by sending it the number of that item. There are actually three such **menu** objects in the lower left corner of the Tutorial patch.

- If you have not already done so, close the *[Sierpinski]* window. Click on the **button** at the right edge of the Patcher window.



The **button** triggers numbers and sends them (via hidden **send** and **receive** objects) to the borderless **menu** objects in the lower left corner.

Window into a Subpatch

The **button** certainly appears to be in the Patcher window, but it is actually part of a subpatch contained inside a **bpatcher** object. A **bpatcher** is like a window into a subpatch. You can load any previously saved patch into a **bpatcher** object, and its contents are then visible through the **bpatcher**. You can resize the **bpatcher** to control just how much of the subpatch is visible, and user interface objects inside the **bpatcher**—such as the **button** in this example—respond to the mouse just as if they were in the main patch.

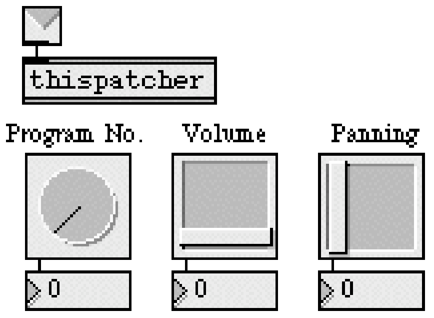
- Unlock the Patcher window and you will see that it contains two **bpatcher** objects, one that contains the **button** and a long thin one at the top of the window that (apparently) contains nothing.

Not only can you control how much of the subpatch is visible by resizing the **bpatcher**, you can also control *what portion* of the subpatch shows through it. Holding down the Shift and Command keys on Macintosh or Shift and Control keys on Windows while dragging on a **bpatcher** moves the subpatch around within it, allowing you to offset the subpatch's position. The amount of the offset shows up in the Assistance portion of the main Patcher window.

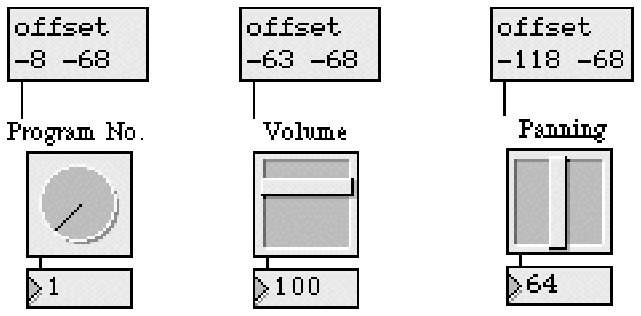
- Lock the Patcher window and hold down the Shift and Command keys on Macintosh or the Alt and Control keys on Windows as you drag on the **button** to move it around within the **bpatcher**. Notice the coordinate information shown in the Assistance area.

You can even make the subpatch inside the **bpatcher** reposition itself, by sending an offset message to a **thispatcher** object inside the subpatch. An offset message consists of the word offset followed by two numbers representing the number of pixels to offset the subpatch horizontally and vertically. So, by carefully designing the patch you want to use as a **bpatcher** subpatch, by carefully sizing your **bpatcher** object, and by sending the proper offset messages to a **thispatcher** object in the **bpatcher** subpatch, you can cause an entirely different image to show through the visible portion of the **bpatcher**.

In the following example, different objects in the subpatch shown on the left can be windowed inside a carefully sized **bpatcher** by sending the correct offset messages, as shown on the right



The contents of this subpatch...



...can be windowed three different ways inside a single **bpatcher** in the main patch.

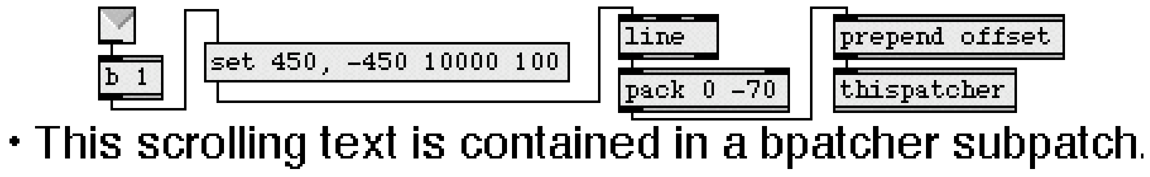
This is the most obvious use of the offset message to a **thispatcher** object in a **bpatcher**. However, as with any image that is positioned by specifying its pixel coordinates, the contents of a **bpatcher** can be animated with a continuous stream of different positioning messages.

Animating a bpatcher

- To see a demonstration of an animated **bpatcher**, click on the words Roll Credits in the lower left corner of the window.

The use of offset messages to a **thispatcher** object in a **bpatcher** is another method of displaying and hiding different text messages. In this case, the **message** box containing the words Roll Credits is connected by a hidden patch cord to the inlet of the **bpatcher**, and its output triggers a progression of different offset messages, causing the appearance of scrolling text.

- To see the contents of the scrolling text **bpatcher**, open the file named *scrollingtext* in the *Max Tutorial* folder and unlock it.



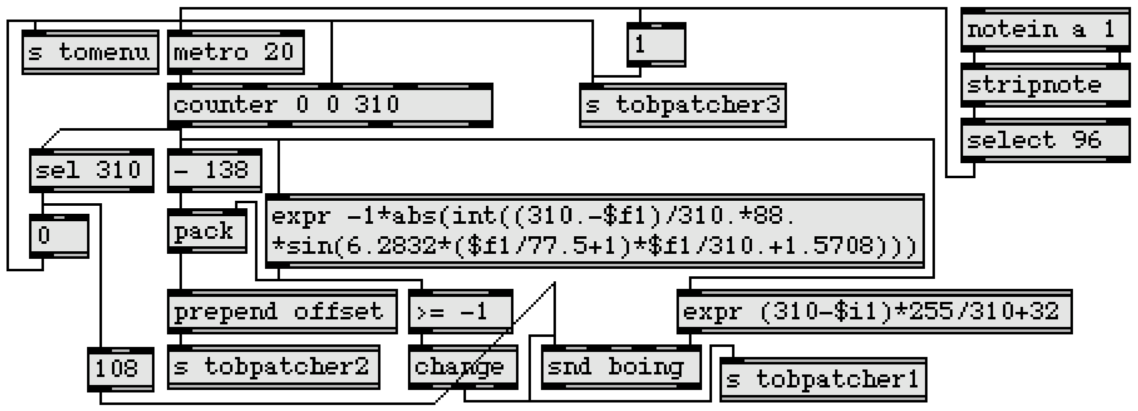
When a message is received in the inlet of a **bpatcher**, it is converted to a bang by the **b** object, and triggers a **line** object which sends out a stream of numbers progressing from 450 to -450 over the course of ten seconds. The number is used as the horizontal coordinate—with the number -70 appended as the vertical coordinate and with the word offset prepended—in an offset message to **thispatcher**.

The other **bpatcher**, containing the **button**, is just for fun, to demonstrate an extreme example of animating the contents of a **bpatcher**. It also provides an opportunity to introduce some new useful objects.

- Play the high C on your keyboard (key 96) once to trigger the animation of a bouncing **button**. If the animation is extremely jerky or you don't hear any sound, check to make sure that the Overdrive option is disabled and the computer's output sound is turned on in the Sound control panel.

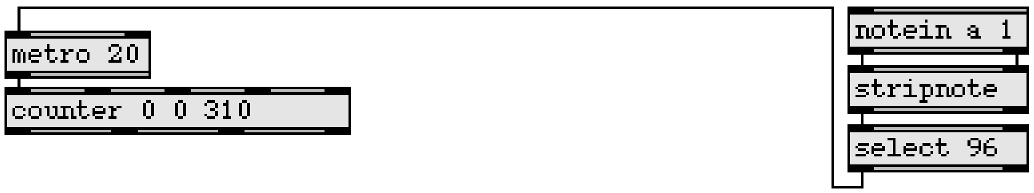
Although it appears that the **button** is moving, you know by now that the effect is actually being achieved by continuously changing the offset of the subpatch inside the **bpatcher**.

- To see how this is achieved, first double-click on the patcher **bouncing** object to see its contents.

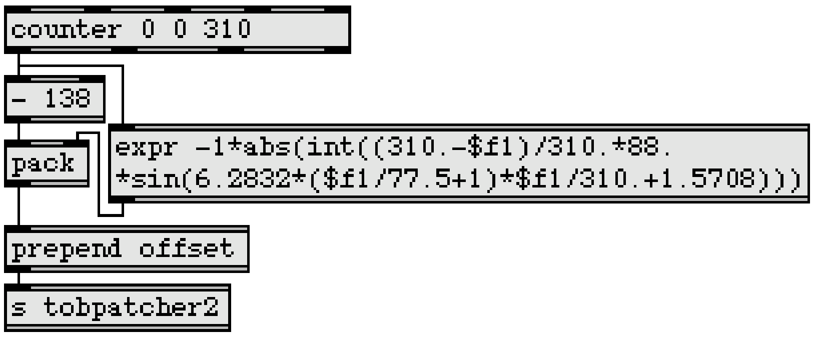


Contents of the patcher bouncing object

When a note-on from key 96 is received, it turns on a **metro** which causes a **counter** to send out numbers from 0 to 310 at a rate of 50 numbers per second.



These numbers are used to calculate the horizontal and vertical offset of the **bpatcher** subpatch, which gets sent to a **thispatcher** object in the **bpatcher** via the **s** tobpatcher2 object.

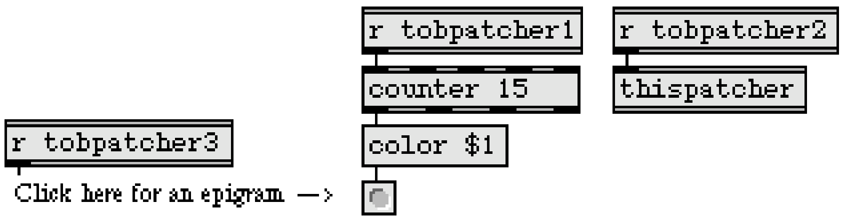

counter 0 0 310
- 138
pack
expr -1*abs(int((310.-$f1)/310.*88.
sin(6.2832($f1/77.5+1)*$f1/310.+1.5708)))
prepend offset
s tobpatcher2

The rather complicated equation in the **expr** object calculates the vertical offset values, using a cosine wave of decreasing amplitude and increasing frequency. Since a cosine wave can represent harmonic physical motion, the absolute value of this diminishing cosine wave is used to imitate a hard bouncing object being affected by gravity.

The value of the cosine wave over the course of time is calculated as the sine of: 2 (6.2832, a complete 360˚ arc) times an increasing frequency ($f1/77.5+1, progressing from 1Hz to 5Hz) times “time” ($f1/310., with “time” being considered the progression of input numbers from 0 to 310) plus a phase offset of / $_2$ (1.5708, a 90˚ phase offset to change the sine wave into a cosine wave). The amplitude of that cosine wave is scaled by a continuously changing amplitude: (310.-$f1)/310.*88. The entire result is converted to an int and its absolute value is used. The multiplication by -1 at the beginning of the equation is there because we need to move the contents of the **bpatcher** with a vertical pixel value between -88 and 0 in order to give the appearance of the button coming to rest at a 0 vertical offset.

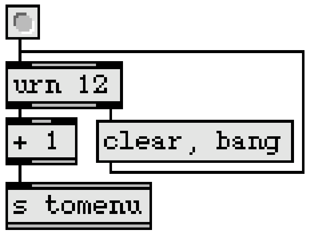
Accessing Text Messages

- Close the *[bouncing]* window. Open the *bouncingbutton* file in the *Max Tutorial* folder and unlock the window to see the contents of the bouncing button **bpatcher**.



You can see the **r** objects that receive messages from inside the **patcher** bouncing object. The color messages for the **button** are received by the **r** tobpatcher1 object, and the offset messages are received by the **r** tobpatcher2 object. What appears to be a **comment** next to the **button** is actually a **menu** in *Label* mode. The **menu** contains text in menu item 0, and nothing in menu item 1. Thus, the text can be hidden by the number 1 being received from the **patcher** bouncing object via the **r** tobpatcher3 object.

You've already seen that when you click on the **button**, text is displayed in the **menu** objects in the lower left corner of the Tutorial patch. In the following example you can see how that's accomplished inside the **bpatcher**.



The **urn** object is very similar to **random**; when it receives a bang it outputs a random number from 0 to the number one less than its argument. Unlike **random**, however, **urn** keeps track of the numbers it has sent out, and will not output the same number twice. The **urn** object is used any time you want to generate all the elements of a set without repetition. In this case, it outputs numbers from 0 to 11, which have 1 added to them to select items 1 to 12 of the **menu** objects. Since all three **menu** objects receive the same number, their messages can be correlated and be guaranteed to be displayed together.

When **urn** has output all the possible numbers in its range, it does not send any more numbers, and instead sends a bang out its right outlet. This bang can be used to send a clear message back to **urn**, clearing its memory, preparing it to output numbers once again. In this example, **urn** always clears its own list and re-bangs itself whenever it has run out of numbers to send; so it always sends out a number, but it minimizes the repetitions that occur.

Summary

There are several ways to create animation within a Patcher window. You can play a QuickTime movie in a Patcher window with the **imovie** object; you can paint colored lines, shapes, and text in a Patcher window with QuickDraw-like messages to an **lcd** object; and you can make text messages appear, change, or disappear in the Patcher window by sending a menu item number to a borderless **menu** object.

Graphic images can be animated algorithmically using controlled randomness, fractal formulae, or any formula into which you send a progression of different input values to calculate the coordinates of graphic objects, lines, or shapes.

The **bpatcher** object allows you to create a window into the contents of a subpatch. User interface objects that are visible in a **bpatcher** can respond to the mouse just as if they were in the main patch. You can display different parts of the subpatch in a single **bpatcher** box by sending an offset message to a **thispatcher** object inside the subpatch. By sending a progressive series of offset values to a **bpatcher**, you can scroll text or give the impression of moving objects.

The **urn** object functions like **random**—when it receives a bang it outputs a random number within a specified range—but it keeps track of the numbers it has sent out, and does not send out the same number twice until its memory has been cleared. Thus, **urn** is useful for generating a random, non-repeating sequence of any set of messages or events.

See Also

bpatcher	Embed a visible subpatch inside a box
Graphics	(**Max Topics**) Overview of graphics windows and objects
imovie	Play a QuickTime movie in a Patcher window
lcd	Draw QuickDraw graphics in a Patcher window
menu	Pop-up menu, to display and send commands
urn	Generate random numbers without duplicates
Tutorial 42	Graphics

Tutorial 44: Sequencing with **detonate**

Extended Sequencing Capabilities

In this chapter we demonstrate the use of the **detonate** object for sequencing MIDI note events, and we show how **detonate** can be used to implement more advanced sequencing capabilities such as non-realtime "step" recording, continuously variable playback tempo, and triggering individual notes on command. Because this is a fairly complex patch, it is also instructive as an example of how to organize a maze of communications between objects by encapsulating the various tasks into separate subpatches.

The functions of this patch are:

1. to record incoming MIDI notes
2. to play them back while varying the tempo, or
3. to step through the recorded sequence one note at a time by triggering each note from the computer keyboard or the MIDI keyboard.

You can switch from one function to another by clicking on buttons onscreen (actually **message** boxes) or by typing key commands on the computer's keyboard.

Using the Patch

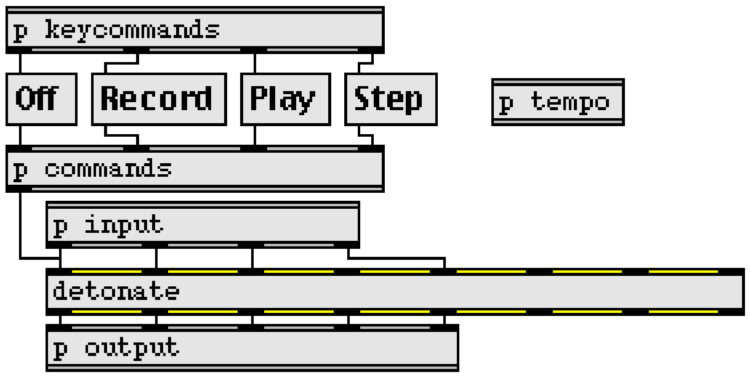
Before examining the construction of this patch, you may want to use it to get an idea of what it does.

- Click on the *Record* **message** box—or type *r*—and play a melody or some arpeggiated chords on your MIDI keyboard for at least fifteen seconds or so.
- When you have finished playing, you can hear your performance played back by clicking on the *Play* **message** box or typing *p*. You can vary the tempo of the playback—from $^1/_2$ to 2 times the original tempo—by dragging on the horizontal slider in the *[tempo]* window.
- Return the tempo to 1, then click on the *Step* **message** or type *s*. You can now play each of the recorded notes one at a time by playing any key on your MIDI keyboard or by typing the Enter or Return keys on Macintosh or the Enter key on Windows on your computer keyboard.

- When you have finished, click Off or type *o*. You can edit the recorded notes by double-clicking on the **detonate** object.

Encapsulation of Tasks

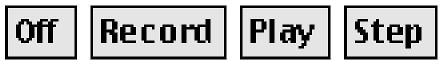
To keep this patch neat and comprehensible, it was necessary to think of it in terms of the different tasks to be performed—as outlined above—and then try to enclose each task in its own subpatch. So, there is one **patcher** for capturing key commands from the computer's keyboard, another for actually performing the commands, one for getting MIDI input and sending it to **detonate**, one for sending the data from **detonate** to the MIDI output, and one for varying the tempo of the notes played by **detonate**.



A subpatch such as **p** commands needs to communicate to *all* of the other subpatches, which would cause a tangled net of patch cords. So we had to decide which are the *direct* communications to be made via patch cords with inlets and outlets—commands coming in from the **message** boxes that the user clicks on, and going out to the **detonate** object—and which are the indirect ones to be made remotely via **send** and **receive** objects—such as supplying values to other subpatches or controlling the flow of MIDI messages.

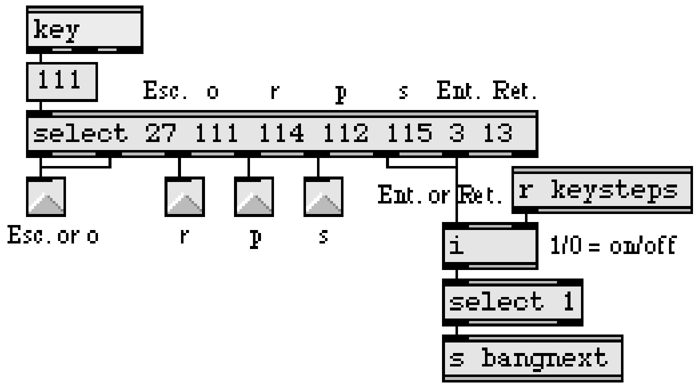
Receiving Commands from the User

Most patches require some kind of controlling command input from the user. In this case we want to choose one of three mutually exclusive actions—*record*, *play*, and *step* through the recorded notes—plus a fourth action, *off*. This is accomplished easily enough with four clickable commands in **message** boxes.



For quick access to the commands, we can make keyboard equivalents by looking for specific ASCII values and banging the **message** boxes when the keys are pressed. Detecting key presses on the computer's keyboard has already been demonstrated in Tutorial 20. The key detection is very simple, and is a very specific task, so it is easily encapsulated in the subpatch **p** keycommands, the outlets of which are connected directly to the command **message** boxes.

- Double-click on the **p** keycommands object to see its contents.



In addition to the mnemonic key commands *o*, *r*, *p*, and *s* for triggering the **message** boxes, the Escape key is used as a synonym for *off*, and the Return and Enter keys on Macintosh or the Enter kay on Windows can be used to step through the score. If the number 1 has been received from the **r** keysteps object, then Return or Enter on Macintosh or Enter on Windows will trigger a next message in another subpatch via the **s** bangnext object.

The Central Command Post

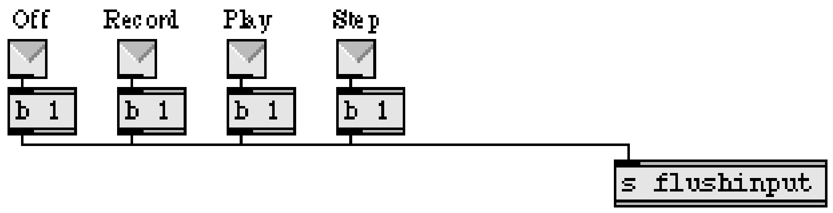
Analysis of the different functions of the patch revealed that the user interface could really be very simple: four clickable commands with keyboard equivalents. However, each of those commands must actually trigger a variety of actions throughout the whole patch.

The **p** commands subpatch is for ensuring that all of those actions are carried out in the proper order when a command is received.

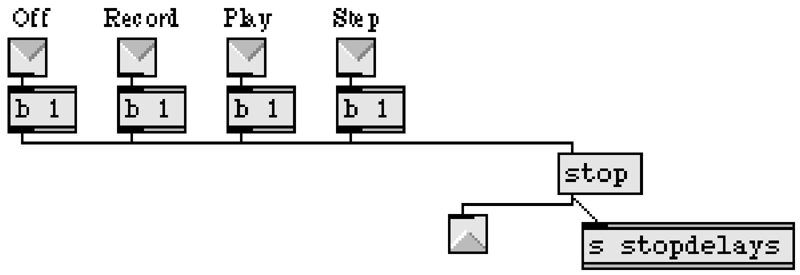
- Close the *[keycommands]* window and double-click on the **p** commands object to see its contents.

The output of each of the clickable **message** boxes comes in the one of the inlets of **p** commands and is converted to a bang with a **b** 1 object, and that bang triggers everything that needs to happen for each command.

Each new command that comes in could *potentially* cause **detonate** to stop recording while a note is in the process of being recorded, so the first thing each command does is bang a **flush** object in the **p** input subpatch to turn off any incoming MIDI notes.



Although it's not strictly necessary, each incoming command also stops **detonate** before giving it a new command, and stops any delayed bang messages that may exist in the **p** output subpatch if **detonate** were playing.



Then finally each of the incoming commands opens or closes the appropriate **gate** objects in the **p** input and **p** output subpatches, and sends the appropriate command to **detonate**. So, for example, an *Off* command will:

1. flush any held notes in the **p** input subpatch

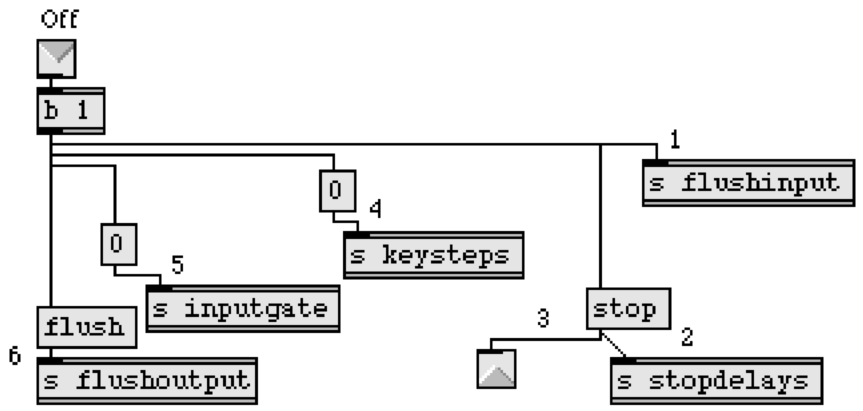
2. stop any delayed bang messages in the **p** output subpatch

3. stop **detonate**

4. send 0 to the **p** keycommands subpatch so that the Return and Enter keys will no longer have any effect

5. close a **gate** in the **p** input subpatch to stop incoming MIDI notes

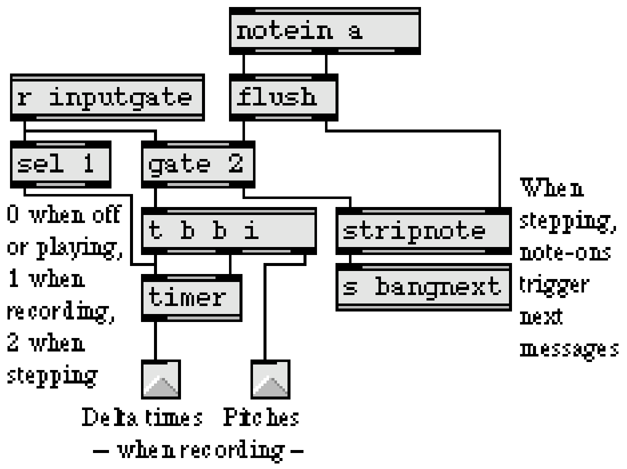
6. flush any held notes in the **p** output subpatch.



MIDI Input to detonate

- Close the *[commands]* window, and double-click on the **p** input object.

A **gate** object is used to route the incoming MIDI pitch numbers to the proper place. When **detonate** is stopped or playing, we want it to ignore incoming MIDI information, so the **gate** is closed. When recording, the pitches are sent out the left outlet of the **gate**, and when stepping through notes the pitches are sent out the right outlet of **gate**.



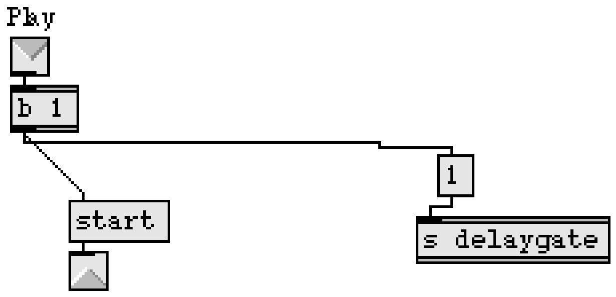
When **detonate** is recording, we need to send it not only the note information, but also the time elapsed since the previous message. Therefore, we use the **sel** 1 object to start a **timer** when recording is turned on. During recording, the pitch value goes directly to **detonate**, and also bangs the **timer** to report the elapsed time; then it restarts the **timer** for the next incoming note message. The time reported by **timer** is used as the delta time, and is combined with the pitch, velocity, and channel numbers to record a note event in **detonate**.

When the *Step* command is chosen, the number 2 is sent to **gate** to open its right outlet. Instead of going to the **timer** and to **detonate**, the pitch numbers go to **stripnote**. The **stripnote** object filters out the note-off messages, and only the note-on pitches are used to trigger a next message to **detonate** (back in the **p** commands subpatch).

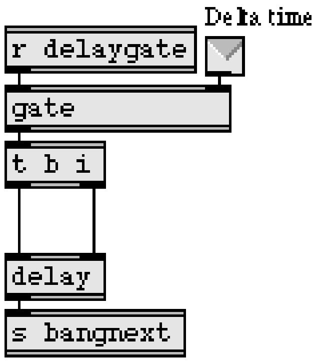
Note Events from detonate

- Close the *[input]* window, and double-click on the **p** commands object again.

When the user clicks on *Play*, it sends 1 from the **s** delaygate object to the **r** delaygate object in **p** output, and then it sends a start message to **detonate**.

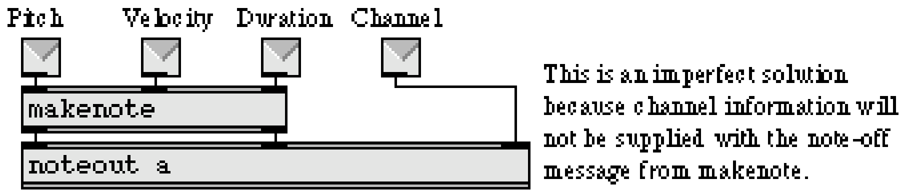


- To see where those messages will go, close the *[commands]* window, and double-click on the **p** output object to open it.

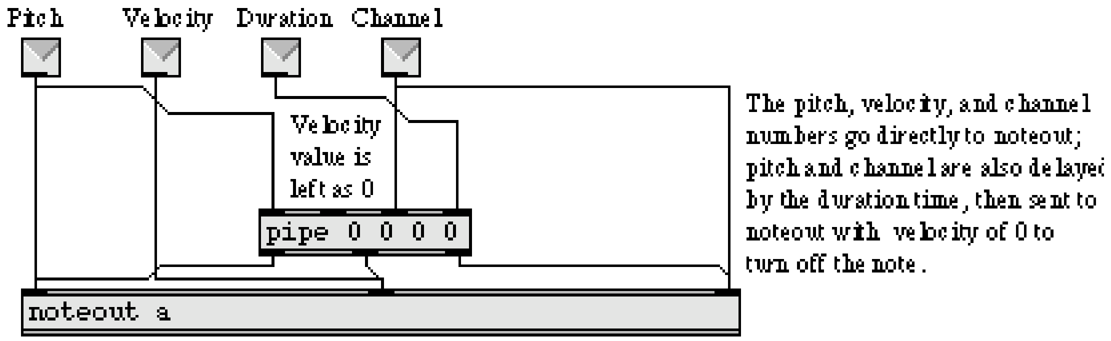


The number 1 from **r** delaygate opens a **gate** to let the numbers received in the left inlet go through. Then the start message sent to **detonate** causes it to report the first delta time, which comes in the left inlet of **p** output and passes through the **gate**. The number goes to the right inlet of **delay** and is used as the delay time before banging a next message to **detonate** to trigger the event information for the first note. As **detonate** sends out event information in response to the next message, it also sends out the delta time of the *next* note event, so the process continues until **detonate** is stopped or runs out of notes.

The other items of event information that come from **detonate** are pitch, note-on velocity, duration, and channel. Using **makenote** to supply note-off messages seems reasonable, but in this case doing so would unfortunately separate the channel information from the pitch and velocity, making it possible that note-offs could be transmitted on the wrong channel (if, for example, a note message on channel 2 occurs just before the note-off for a note on channel 1).


Pitch
Velocity
Duration
Channel
makenote
noteout a
This is an imperfect solution because channel information will not be supplied with the note-off message from makenote.

Therefore, it's preferable to create note-off messages by using a **pipe** object to delay the pitch and channel information together, which will send those values out with a velocity of 0 after waiting for the number of milliseconds specified by the duration value. So in this patch the note-on message goes directly to **noteout**, and **pipe** supplies a later note-off message on the same key and channel.

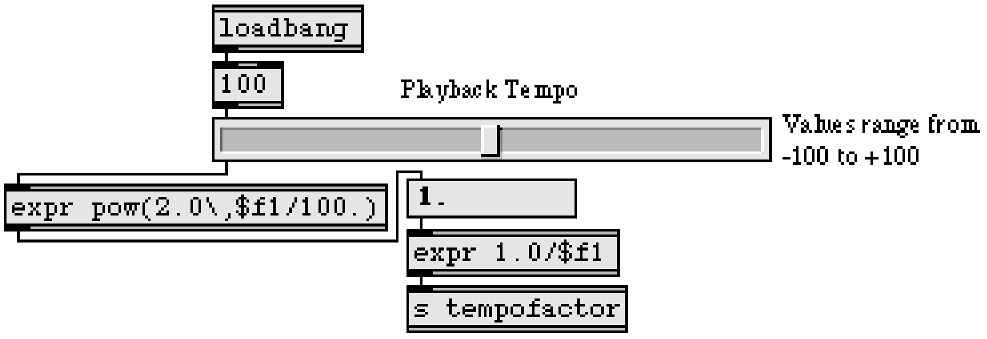

Pitch
Velocity
Duration
Channel
Velocity value is left as 0
pipe 0 0 0 0
noteout a
The pitch, velocity, and channel numbers go directly to noteout; pitch and channel are also delayed by the duration time, then sent to noteout with velocity of 0 to turn off the note.

Modifying the Playback Tempo

You have no doubt noticed that the duration values and the delta time values each pass through a ***1.** object. As they go through, they are multiplied by a scaling factor received from the **r** tempofactor objects. This tempo scaling factor is produced in the **p** tempo subpatch.

- To see how the scaling factor is produced, close the *[output]* window and bring the *[tempo]* window to the foreground.

- Since the *[tempo]* window contains hidden objects, you'll need to unlock it and click on the zoom box in the right corner of the title bar to see its contents (or you can simply consult the picture of it shown here).



The contents of the p tempo subpatch

We decided to use the **hslider** object to permit the user to give tempo scaling values from half the original tempo to twice the original tempo (from 0.5 to 2.0). This presents a small problem, because the factor we want to use is a multiplier, while **hslider** is on an additive (linearly increasing) scale. However, if we recognize that $0.5 = 2^{-1}$, $1 = 2^{0}$, and $2.0 = 2^{1}$, then we see that we can use the **hslider** to provide the *exponent* ranging from -1 to +1. By selecting the **hslider** and choosing **Get Info...** from the Object menu, we set the *Slider Range* to 201 values, and the *Offset* to -100, so that it sends out values from -100 to +100. In the **expr** object, we divide that number by 100., and use the result as the exponent in the pow() function, to get 2^{x}.

As a matter of fact, though, in order to double the tempo, we need to *halve* the delta times and durations; conversely, to halve the tempo we need to double the delta times and durations. This means that we want to show the user numbers ranging from 0.5 up to 2.0, but actually send numbers ranging from 2.0 down to 0.5 to **r** tempofactor in the **p** output subpatch. The value we want to send is the reciprocal of the value we want to show, so we actually send one over the tempo factor.

Although the **seq** object permits playback at different constant tempi, the use of **detonate** shown here is the best way to vary **continuously** the playback tempo of a MIDI file or other stored sequence of note events.

Non-Realtime Recording

The rhythm of a sequence recorded in **detonate** is determined by the event starting times given to **detonate** (that is, the delta time received for each note event), rather than by the actual time **detonate** receives the events. For this reason, a sequence can be recorded over any period of time, or even in a single instant. This is demonstrated in the subpatch **p** 'Another Example', which is a completely separate program from the rest of this patch.

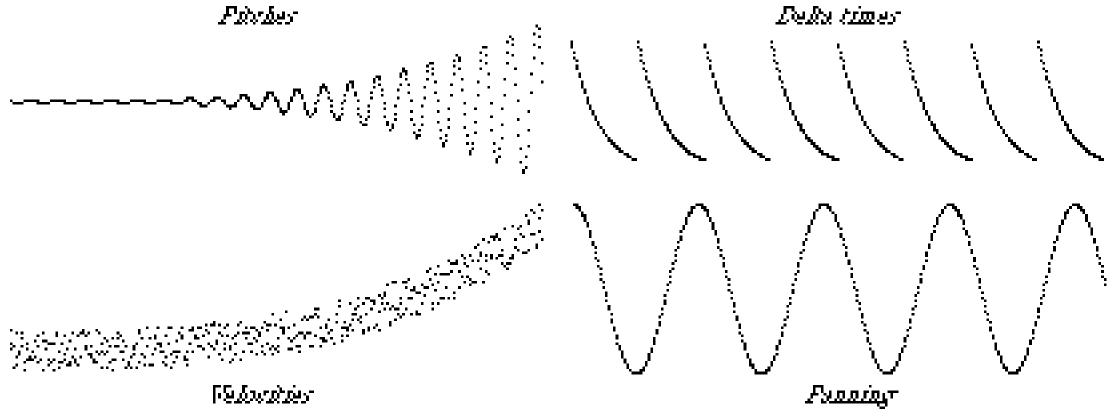
- Double-click on the **p** 'Another Example' object to open it.

Although some of the arithmetic in the **expr** objects may appear daunting, the basic operation of this patch is extremely simple. When you click on the **button**:

1. A record message is sent to **detonate**.
2. **uzi** sends out 1000 numbers ascending from 1 to 1000 (effectively from 0 to 999, since the numbers go immediately to a **-** 1 object).
3. Each of those numbers is used to calculate the different parameters of a note event.
4. When **uzi** is done, a start message is sent to **detonate**, followed immediately by a next message to send out the first note event.
5. The event parameters are converted to MIDI messages by **makenote** and **noteout** (and **ctlout** for panning messages), and the delta time is used to determine when the next note should be triggered.

In a single tick of Max's clock, a melody approximately 78 seconds long is composed and recorded.

Each of the event parameters is calculated according to a unique formula describing a particular curve from the beginning to the end of the melody's duration.



When these individual curves of progression for each of the parameters are combined, they create a constantly changing yet still quite predictable melody. Panning moves according to 4$^{1}/_{4}$ cycles of a cosine wave, beginning panned to one side, then moving slowly from side to side and ending in the center of the stereo field. Velocity is random within a restricted range that begins from 1-32 and increases according to an exponential curve ending in the range 96-127. Pitch moves in 480 cycles of a sinusoidal wave centered around key 66, beginning with an amplitude of 0 semitones and ending with an amplitude of ±30 semitones, from 36 to 96. Delta time between notes changes according to 8 exponential curves of acceleration, repeatedly accelerating from 5 notes per second to 50 notes per second. Duration is always 5 times as long as the delta time of the next note, so that even the fastest notes last at least 100 milliseconds.

- Click on the **button** to compose, record, and play the melody.

Summary

The **detonate** object is useful for recording and playing sequences of notes, and can read and write standard MIDI files. It is also useful for less commonplace sequencing tasks such as non-realtime recording, continuously variable playback speed, and playing back the recorded notes in a new rhythm.

To record MIDI note messages in **detonate**, a **timer** should be used to report the time elapsed between messages, which **detonate** will record as the *delta time* parameter of each note event. On playback, the delta time should be used to determine how long to wait before playing the next note. Multiplying the delta times and durations by some number other than 1 changes the tempo of the playback. When supplying note-offs for notes on different channels, **pipe** can be a useful substitute for **makenote**.

See Also

detonate	Graphic score of note events
Detonate	(**Max Topics**) Graphic editing of a MIDI sequence
Sequencing	(**Max Topics**) Recording and playing back MIDI performances

Tutorial 45: Designing the User Interface

Making an Application for Others

When you have written an interesting Max program, you may want to give it to other people to use. If your program consists of many different files—your own objects, graphics files, etc.—you will probably want to use the **Save As Collective...** command in the File menu to save all the necessary files together as a single *collective*. You can even save your collective as a standalone application for people who don't have Max or Max/MSP Runtime. For more information about saving your program as a collective or standalone application, see the chapter on *Collectives* in the *Topics* section of this manual.

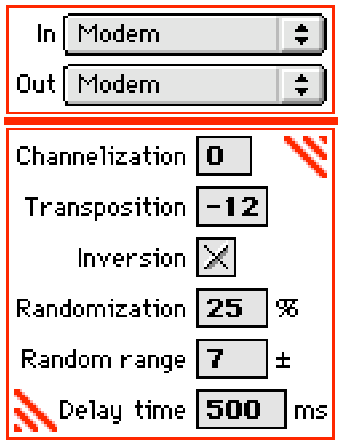
If you're going to give your program to others to use, you will probably also want to spend some time planning and designing the user interface, to make it as well-organized, attractive, intuitive, clear, and user-friendly as possible. This chapter presents a complete application written in Max, and discusses a variety of issues to consider when planning your application and designing its user interface.

Because this patch is considerably more complex than any of the other examples in this Tutorial, we won't go into extensive detail trying to explain how it works. We'll leave that for you to investigate on your own if you're curious. Rather, we'll try to point out some of the visual design decisions that were made and some ways of implementing certain user interface features. This chapter will show how to plan the layout of your program, how to modify windows and the menu bar to your liking, how to add graphics to customize the look of your program, and how to decide the best way to present information to, and get input from, the user.

The Note Modifier Program

The example application, called *Note Modifier*, is a four-track router-channelizer-transposer-inverter-randomizer-delayer of MIDI note messages. The four tracks of modification work in parallel—separately and simultaneously—and can be turned on and off individually. The actual modifications performed in each of the four tracks are in series—the output of one goes into the input of the next—and can be turned on or bypassed individually. In addition the program provides an onscreen imitation keyboard, so that notes can be played with the mouse and fed into the *Note Modifier.*

- To begin modifying MIDI notes, turn on one track by clicking on the *Track A* button. (You can also turn the track on or off by choosing **Track A** from the Modify menu, or by typing Command-1 on Macintosh or Control-1 on Macintosh.)



- Use the *In* and *Out* pop-up menus to choose the input port from which you wish to receive the MIDI notes and the output port to which you wish to transmit the modified notes. The pop- up menus should contain the list of devices from your current MIDI setup.

- As long as a **number box** in the *Track* A window shows 0, that particular modification will be bypassed and the note will be sent on unchanged. Drag on the **number box** objects with the mouse (and/or click on the *Inversion* **toggle**) to set the desired modifications. Then begin playing on your MIDI keyboard. Try different combinations of modifications.

- If you want more streams of modified notes, turn on additional tracks and set different values for the parameters of those tracks.

- By choosing **Keyboard** from the Modify menu, you can use an onscreen imitation MIDI keyboard which sends its notes to the *Note Modifier* tracks as well as directly to the output port you select. This allows you to use the application even when you don't have a MIDI keyboard available.

The *Note Modifier* program is modeled after the PCL software originated by Richard Teitelbaum and coded in 68000 assembly language by Mark Bernard in 1983 (see Richard Teitelbaum, "The Digital Piano and the Patch Control Language," *The Proceedings of the ICMC*, Paris 1984), and later re-implemented in Max to Teitelbaum's specifications by Christopher Dobrian in 1990.

Tutorial 45

Planning Your Application

In order to design a good program and a good interface, it pays to do some planning before you begin programming, to make sure that you know *a)* what things you want the program to do, and how you plan to do them, *b)* what information you'll need to give the user, and how you plan to display it, and *c)* what information you need to get from the user and how the user can best provide it. Once we decide what our application will do (four tracks of MIDI routing, channelizing, etc.) and how that can be accomplished, the next thing to consider is "What do we need to tell the user?"

The user needs to be told which tracks are currently turned on (an on/off indication for each track), and what the settings are for each track (a set of parameter names and their values). Four tracks, with eight modifiable parameters on each track plus an on/off indicator, makes 36 different items of information we need to show the user, plus labels to identify the items. Some information is numerical, some is a simple on/off indication, and some (the port names and labels) is text. All of it will potentially need to be visible at one time, and there should be a way for the user to change any of the values at any time.

All of the above considerations will affect your decisions of screen layout, which user interface objects to use, and what combination of typing and mousing—menus, dialogs, pop-up menus, buttons, toggles, sliders, etc.**—**is best for getting information from the user. You can be guided in these decisions by observing other effective applications, and by considering any real-world models that might provide a good example.

Designing Your Own Buttons

As was demonstrated in Tutorial 19, you are not restricted to using Max's **button** object or **message** box for responding to mouse clicks. You can design your own button in a painting or drawing program, place it in a Patcher window—with the **fpic** object or by copying it and using the **Paste Picture** command in the Edit menu—and then cover it with a transparent **ubutton**. That's the method used for the track on/off buttons in this program. We drew a picture of four buttons, used **fpic** to display the picture, and placed four **ubutton** objects on top of it. To be sure that the **fpic** is *behind* the transparent **ubutton** objects—so that the mouse clicks will go to the **ubutton** objects and not the **fpic**—we simply selected the **fpic** and chose **Send to Back** from the Object menu.

If you select a **ubutton** and choose **Get Info...** from the Object menu, you will see that it has an optional setting called *Toggle Mode*. When a **ubutton** is in *Toggle Mode*, the first bang or mouse click it receives highlights it and sends bang out its right outlet. The next bang or mouse click unhighlights it and sends bang out the left outlet. This makes it very versatile as an on/off switch *and* an on/off indicator. When the display monitor is black and white, **ubutton** reverses the color of whatever picture is underneath it. When the monitor is color, **ubutton** reverses only the black or white portions of the picture.

For this application we chose to use solid dark colors rather than light colors or gradients, so that they would work well on any monitor. We also chose four different basic colors, one for each of the four tracks, so that the color scheme plays a functional role as well as a decorative one, helping the eye separate the windows.

Another issue that involves color versus black and white is the use of anti-aliasing for text and graphics. Anti-aliased text, which you can produce in most painting programs, looks much better onscreen than plain text, but on a black and white monitor it can look very jagged and unattractive. Therefore, in most cases it’s wise to choose a font, size, and style that is clearly legible without anti-aliasing— especially when the text is small. Of course, the majority of people using this version of Max have a color monitor, so you might decide to design the look of your program with color users in mind and accommodate people with black and white monitors to whatever extent you see fit.

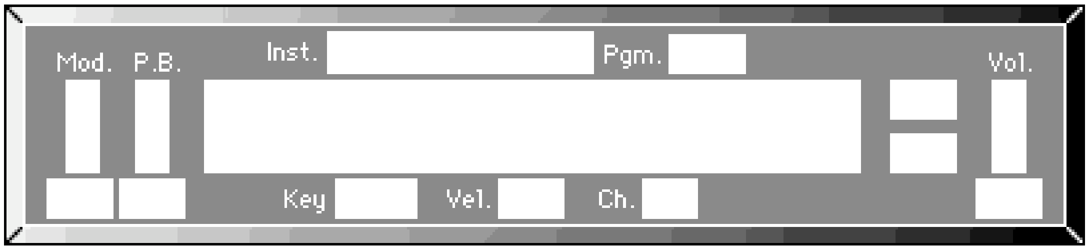
TRACK A
TRACK A

Anti-aliased text looks better on a greyscale or color monitor than on a black and white monitor.

Combining Max Objects and Graphics

Once you have decided what objects you want to show to the user, and have laid them out the way you want them, you can copy them from your Max patch, paste them into a drawing or painting program, and then draw around them to make a picture that seems to include the Max objects. If your graphics program supports multiple layers—as does Adobe Photoshop, for instance—you can put the Max objects in a separate layer from the rest of your picture. Once your picture is the way you want it, delete the Max objects from the picture, copy the rest, and paste it into your Max patch. It will fit perfectly with the original Max objects that you copied in the first place.

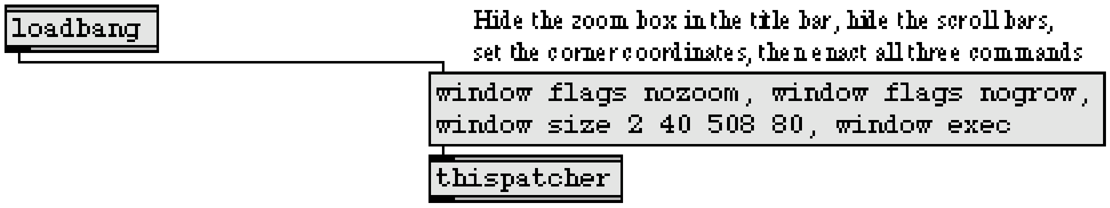
The track windows and the keyboard window of this application were done this way.



This picture was painted around some Max objects, leaving perfectly sized holes for them in Max

Window Size and Placement

You can open and close the window of a subpatch automatically with the **pcontrol** object, and you can open, close, move, resize, and alter the appearance of a subpatch window with **thispatcher**. A **thispatcher** object sends messages to the Patcher that contains it. Each of the windows in this application contains a hidden **thispatcher** object to set the window up with exactly the desired size, location, and characteristics. When the application is opened, a **loadbang** object triggers the messages to each **thispatcher** to set up each window.



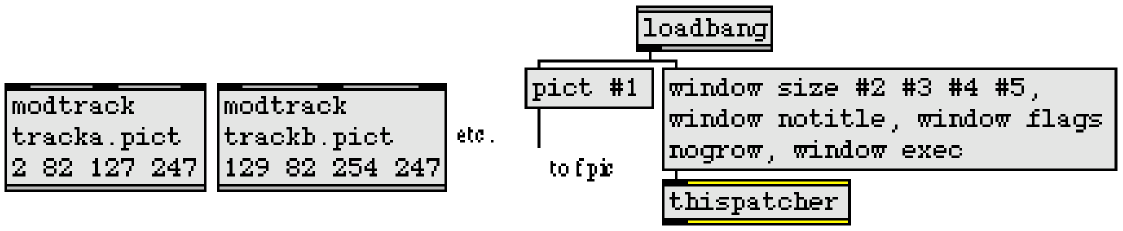
```
loadbang
```

Hide the zoom box in the title bar, hide the scroll bars, set the corner coordinates, then enact all three commands

```
window flags nozoom, window flags nogrow,
window size 2 40 508 80, window exec
thispatcher
```

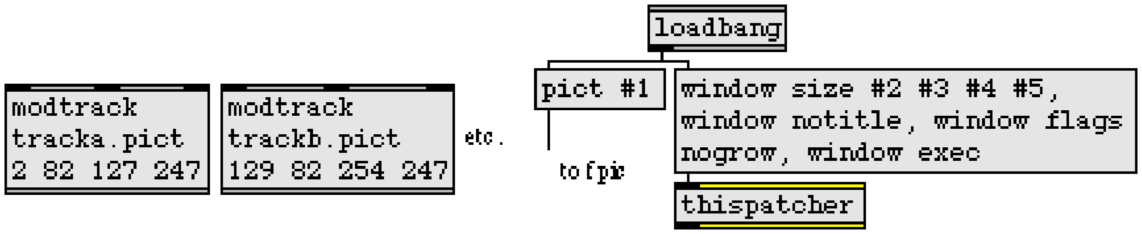
Set the characteristics, size, and location of a window with thispatcher

The windows for the four tracks are four instances of the same subpatch, a separate file called *modtrack*. Yet, each instance can have a unique picture and a unique window placement because that information is supplied to the **fpic** object and the **thispatcher** object as arguments to the **modtrack** object in the main patch.


modtrack tracka.pict 2 82 127 247
modtrack trackb.pict 129 82 254 247
etc.
loadbang
pict #1
window size #2 #3 #4 #5, window notitle, window flags nogrow, window exec
to fpic
thispatcher

Arguments to modtrack in the main patch... provide unique attributes for each modtrack subpatch

Some caution is advised when changing windows with **thispatcher**. For example, it's possible to give window size coordinates that are entirely outside the bounds of your screen, making it invisible to you (but still open). Also, once you hide the title bar you can no longer drag the window to a new location, and once you hide the scroll bars you may be unable to get to the proper place in the patch to make some necessary changes. A good safeguard against these problems is to connect a **receive** object to the inlet of **thispatcher** so that you can send it messages from another Patcher if necessary.


modtrack tracka.pict 2 82 127 247
modtrack trackb.pict 129 82 254 247
etc.
loadbang
pict #1
window size #2 #3 #4 #5, window notitle, window flags nogrow, window exec
to fpic
thispatcher

A message in one patch...can change the window characteristics of another patch

Customizing the Menu Bar

The standard way for a user to give commands to an application is by choosing a command from the menu bar. In our application we want menu commands for turning each track on or off, for opening and closing the keyboard window, and for sending an *all notes off* message out on all channels in case there are stuck notes on the synth.

With the **menubar** object you can add your own menus and commands to the menu bar. The argument to **menubar** tells it how many menus you want there to be. (There must be four menus. These are the Help, File, Edit, and Windows menus.) Then you type in a script that explains to **menubar** where you want it to put additional menus and commands. (See **menubar** in the Max Reference Manual for details on writing the script.) In our case, we want to change the first item in the Help menu from **About Max...** to **About Note Modifier...,** and we want to add a new menu called Modify that contains the new commands we want.

The script is as follows:

```
#X about About Note Modifier...;
#X menutitle 5 Modify;
#X item 5 1 Track A/1;
#X item 5 2 Track B/2;
#X item 5 3 Track C/3;
#X item 5 4 Track D/4;
#X item 5 5 -;
#X item 5 6 Keyboard/K;
#X item 5 7 -;
#X item 5 8 Panic/P;
#X end;
```

The / character is special, indicating that the character the follows it should be the keyboard shortcut associated with that menu item (Command + <*key*> on Macintosh or Ctrl + <*key*> on Windows). The – character is also special, indicating that a gray line should be substituted for an actual menu item at that point in the menu, which is useful for dividing the menu into sections.

Once our menu bar is in place, we have three ways to turn tracks on or off: a button, a menu command, and a key command. This introduces a bit more complexity to our programming task, however, because each of the three methods needs to: highlight or unhighlight the button, check or uncheck the menu item, open or close the track window, and enable or disable MIDI in that track window.

- To see how this is done, you'll have to resort to a trick to see the contents of the main patch. Close the *45. Note Modifier* window, then re-open it and hold down the Command and Shift keys on Macintosh or Control and Shift keys on Windows as it is opening. This will stop all **loadbang** objects from sending out their bang messages, and will open the window without hiding the scroll bars and zoom box. Now you can unlock the Patcher and enlarge the window to see how the **menubar** object triggers the

ubutton objects, which in turn trigger all the other necessary actions for turning a track on or off.

Changing Text Labels

When you want a text label to change in a patch, the **menu** object is a good substitute for a **comment**. In the **menu** Inspector window, you can set the menu's *Mode* to *Label*. In this mode, **menu** appears as a borderless text label that does not respond to a mouse click, very much like a **comment**. Unlike with a **comment**, however, you can type in a series of different text messages as menu items, and recall them by sending the item number in the inlet. In this way, you can cause a label to change to fit the number it is describing.

For example, in the track window, when the channel of the MIDI note is to be left unchanged, the *Channelization* value is at 0. As soon as a number from 1 to 16 is entered as the *Channelization* value, though, the label changes to *Out Channel*, to show that that is the new output channel. When the value is changed back to 0, the label changes back to *Channelization* to show that there is currently no channelization occurring.

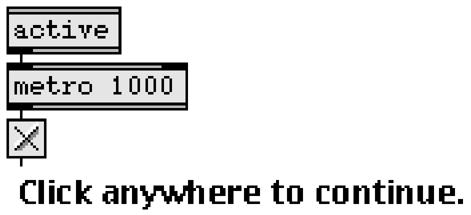
Channelization
0
Out Channel
10

Changing the value in the number box changes the label

Another use of **menu** for changing text can be found in the **About Note Modifier...** screen.

- Choose **About Note Modifier...** from the Apple or Help menus.

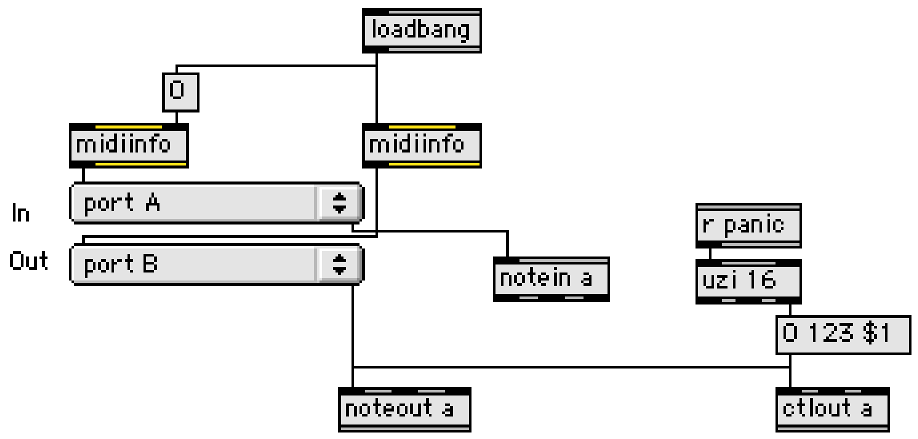
The text "Click anywhere to continue." blinks on and off. This is really just a **menu** in *Label* mode that is being switched between two menu items. One item contains the text, and the other is empty. When the window is brought to the front, an **active** object starts a **metro** which toggles the **menu** back and forth between the two items once each second.



Blinking text by switching between items of a menu in Label mode

Input and Output Ports

Each of the track windows contains pop-up menus for setting the desired input and output ports for MIDI note messages. These pop-up menus contain all the devices in the current MIDI setup, as retrieved by the **midiinfo** object, and they are used to reset the port of **notein**, **noteout** and **ctlout** objects.



midiinfo reports all devices in the current MIDI setup

When the patch is loaded, **loadbang** sends a number in the right inlet of one **midiinfo** object to report the input devices, and sends a bang to the left inlet of another **midiinfo** object to report the output devices. When **midiinfo** gets one of these inputs, it first sends out a clear message to empty the **menu**, and then it sends out a series of append messages to add each of the appropriate device names to the **menu**. This configuration of objects is the way to get information about the current MIDI setup into a patch. The desired port can then be chosen from the pop-up menu.

When the Panic command is chosen from the Modify menu, Command-P on Macintosh or Control-P on Windows is typed, a bang is sent to the **uzi** 16 object in each track, which proceeds to send out an *all notes off* message (continuous controller 123 with a value of 0) on all 16 channels to the output port of that track. This is the best way to implement a quick panic command for stopping stuck notes on the synth.

Summary

With some attention to programming and designing the user interface, a Max patch can be made into a finished application for distribution to others. The menu bar can be customized with new menus and commands using the **menubar** object. The windows of all constituent patches and subpatches can be sized, placed, and customized precisely and automatically using the **thispatcher** object. New onscreen buttons can be designed in a graphics program, placed in a patcher, and made clickable using the **ubutton** object. And Max's user interface objects can be nested in a picture that was designed in a graphics program, making them look like part of the picture. You should choose the colors and fonts in the graphics you design not only for attractiveness, but also for functionality and clarity.

The picture in a patcher can be changed using pict messages to **fpic**. Text labels can be changed by sending item numbers to a **menu** in *Label* mode. Device names in the current MIDI setup can be obtained using the **midiinfo** object and placed in a **menu** object. The names can then be sent to MIDI objects to change their port assignment.

See Also

fpic	Display a picture from a graphics file
menubar	Put up a custom menu bar
pcontrol	Open and close subwindows within a patcher
thispatcher	Send messages to a patcher
Tutorial 19	Screen aesthetics
Tutorial 43	Graphics in a patcher
Collectives	(**Max Topics**) Grouping files to create a single application

Tutorial 46: Basic Scripting

Introduction

Max 4 offers a new way of working with objects and patchcords within a patcher: scripting. Scripting permits you to perform numerous operations on Max objects by sending simple text messages to the **thispatcher** object. Scripting commands are available which create and delete objects and patchcords, send values to objects and change object properties such as visibility, size or position. With scripting, Max programmers may change objects, connections and patcher layout even when the Patcher window is locked.

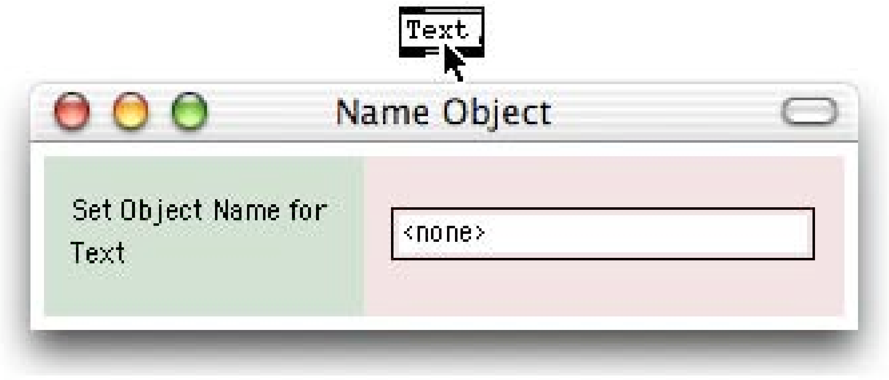
Scripting might be useful for any number of purposes:

- Instantiating and deleting elements of a patcher as you need them.
- Creating, altering and deleting connections between objects.
- Replacing embedded objects, such as patchers inside a **bpatcher** object.
- Controlling the visual arrangement of patches. You can change object sizes and arrangements, even in response to user input.

Give It A Name

In order for scripting to work, objects must have *names*. All scripting commands refer to object names in order to properly assign actions to them. Names can be assigned in one of several ways:

1. Select an object, then choose **Name...** from the Object menu. The Name Object window will open:



Set Object Name for Text

<none>

The Name Object window

By default, Max objects do not have names, so <none> will appear in the Name Object window when you first open it for an object. Type any non-reserved term into the Name Object Inspector, and you've named the object (reserved terms include bang, int, float and list—and errors may result if you use these names). Objects must have unique names within a patcher —Max will warn you that a name is already in use if you try to assign duplicate names to objects. Since this restriction only applies to objects within a Patcher window, identically named objects inside duplicate subpatchers or **bpatcher** objects are not a problem.

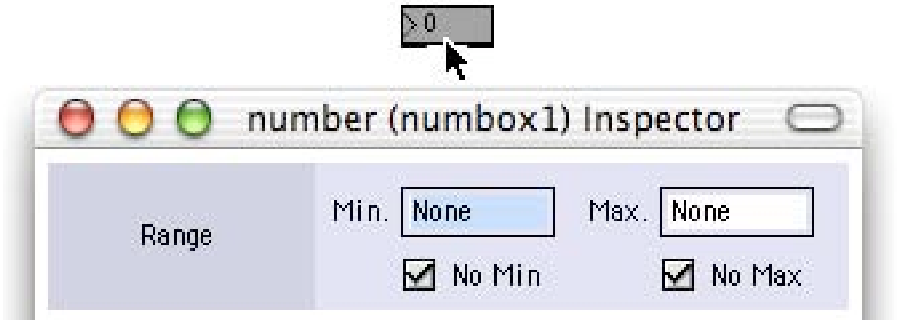
2. Create a new object with scripting: If you create an object using scripting, your new object is given a name as part of the act of creating it. If the name you assign to an object is already in use, the newly created object takes the name away from the object that owns it.

3. Two scripting commands permit you to assign names to objects based on certain criteria (script class and script nth). Please refer to the reference page for the **thispatcher** object for more details about these commands.

To check whether an object is named, you can:

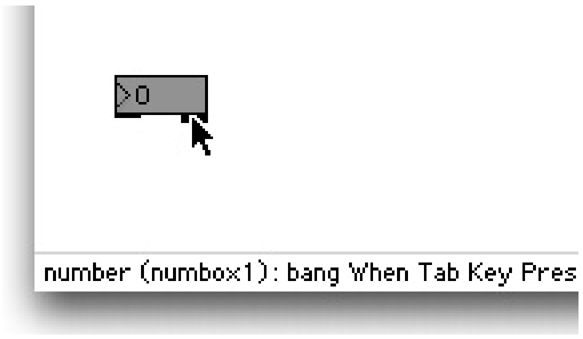
- Select the object and choose **Name…** from the Object menu. If the object is named, the Name Object window will display it. Otherwise you'll see <none>.

- Select the object and, if possible, choose **Get Info…** from the Max menu. The name of the object appears in the title bar of an object's Inspector window.



Object name in its Inspector window's title bar

- Moving the cursor over an inlet or outlet of a named object will show the name of the object in the Assistance field of the Patcher window, if Assistance is checked in the Options menu.



number (numbox1): bang When Tab Key Pres

Object name in Assistance

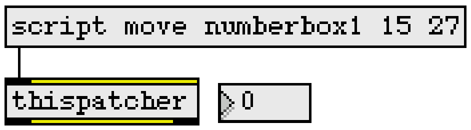
Basic Scripting

Scripting commands take the following form:

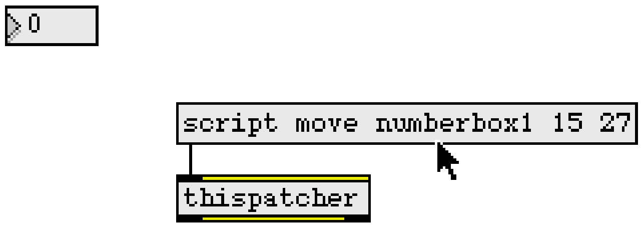
```
script <action> <arguments...>
```

Scripting commands are sent as messages to a **thispatcher** object contained inside the Patcher window where you want something to happen. For instance, if you wanted to move the **number box** named numberbox1 to the Patcher window coordinates (15, 27), the scripting command to do so is script move numberbox1 15 27.

A before-and-after illustration is shown below:



Before sending the script move... message



After sending the script move... message

Making Connections

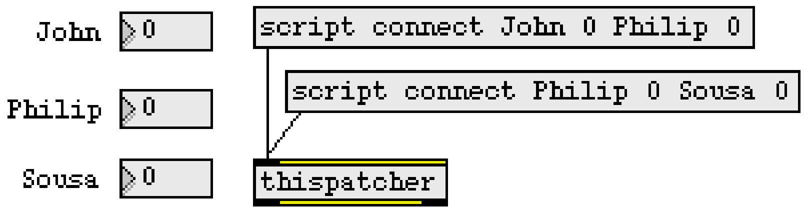
The scripting commands script connect and script disconnect are used to connect and disconnect Max objects. They both use the same format:

script connect <outlet-variable-name> <outlet-index> <inlet-variable-name> <inlet-index>

Inlets and outlets are counted beginning at 0, from left to right. To disconnect objects, the word connect is changed to disconnect:

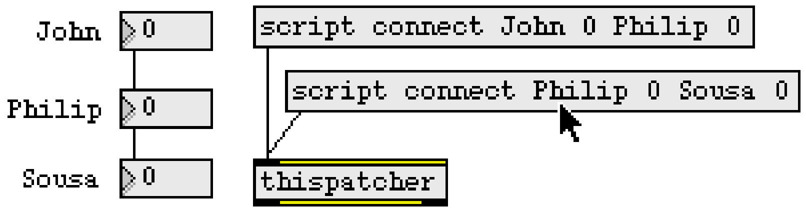
script disconnect <outlet-variable-name> <outlet-index> <inlet-variable-name> <inlet-index>

Here's a before-and-after illustration of these messages.


John
0
Philip
0
Sousa
0
script connect John 0 Philip 0
script connect Philip 0 Sousa 0
thispatcher

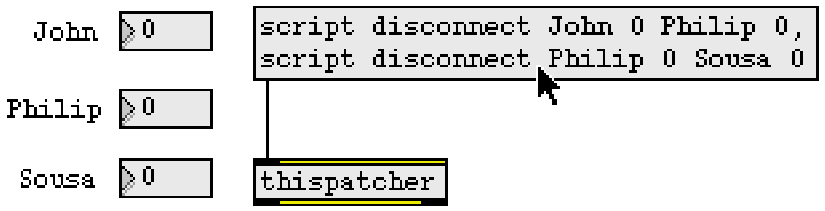
Before sending the script connect message

In the above example, we have three **number box** objects, named *John, Philip* and *Sousa*. The script connect messages to the right can be used to connect them to each other:


John
0
Philip
0
Sousa
0
script connect John 0 Philip 0
script connect Philip 0 Sousa 0
thispatcher

After sending the script connect message

To disconnect, we simply change the connect to disconnect:


John
0
Philip
0
Sousa
0
script disconnect John 0 Philip 0,
script disconnect Philip 0 Sousa 0
thispatcher

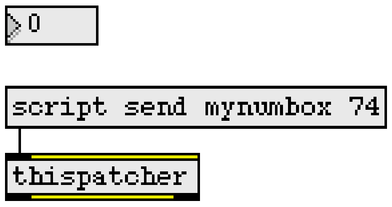
After sending the script disconnect message

Tutorial 46

Sending Messages

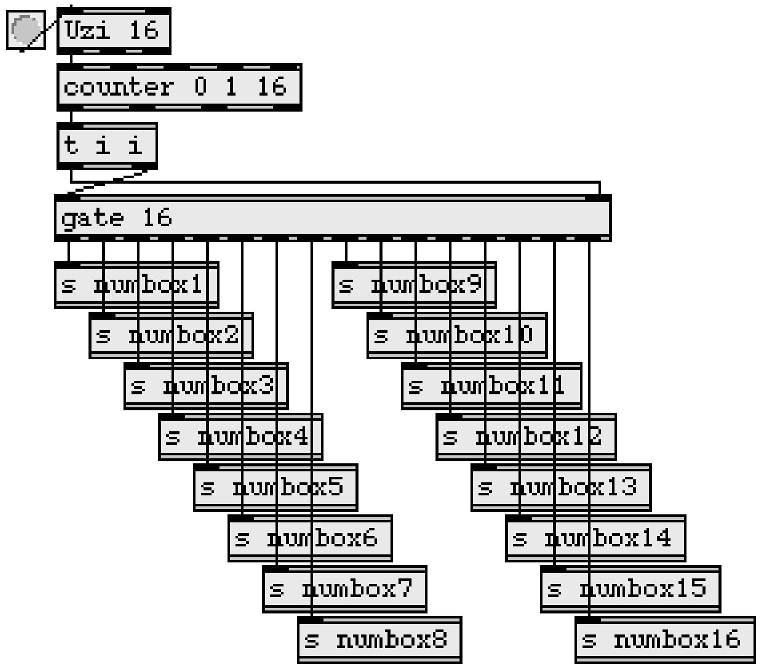
You can use scripting to send values or messages to any named object. The command to do this is:

script send <variable-name>



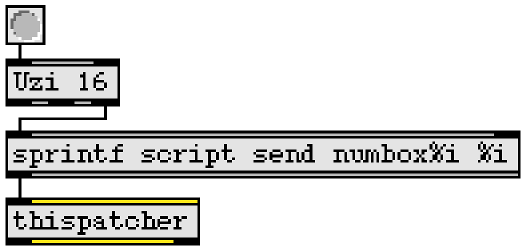
The script send message

This is particularly useful when working with large groups of named objects, where **gate** or **send** objects might be unwieldy. Consider this patch:



Sending to many objects using gate and send

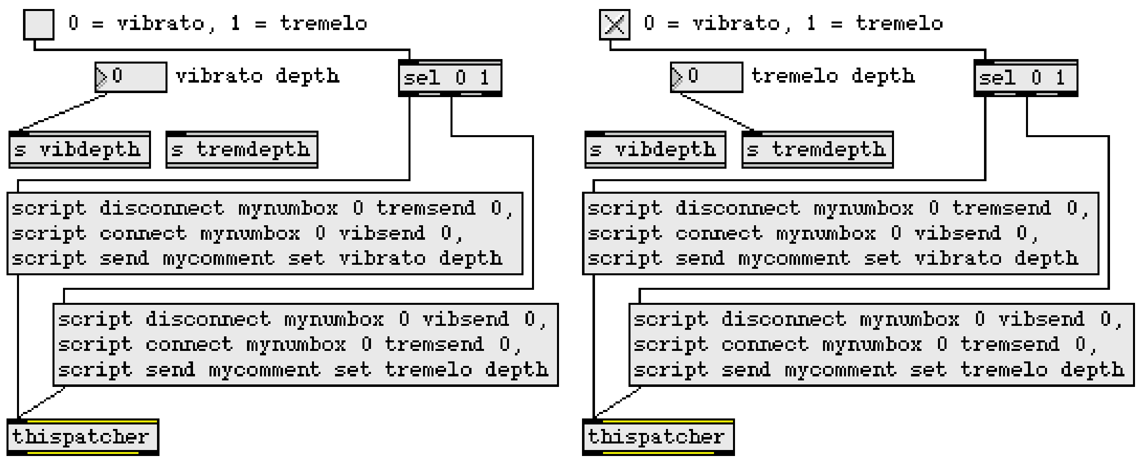
Compare the previous patch to this version:



Sending to many objects using script send

Not only does the second patch eliminate the **gate** and the **send** objects, but there is no need for **receive** objects on the other end. The receiving **number box** objects simply have to be named. In this case, each **number box** has a name starting with numbox and ending with a number. These names can easily be generated by the **sprintf** object.

Another important use of the script send message is to send messages to objects that don't have inlets, such as **comment**. For instance, in the following example, we repatch objects and update the text of the **comment** located to the right of the number:


0 = vibrato, 1 = tremelo
vibrato depth
sel 0 1
s vibdepth
s tremdepth
script disconnect mynumbox 0 tremsend 0, script connect mynumbox 0 vibsend 0, script send mycomment set vibrato depth
script disconnect mynumbox 0 vibsend 0, script connect mynumbox 0 tremsend 0, script send mycomment set tremelo depth
thispatcher
0 = vibrato, 1 = tremelo
tremelo depth
sel 0 1
s vibdepth
s tremdepth
script disconnect mynumbox 0 tremsend 0, script connect mynumbox 0 vibsend 0, script send mycomment set vibrato depth
script disconnect mynumbox 0 vibsend 0, script connect mynumbox 0 tremsend 0, script send mycomment set tremelo depth
thispatcher

Changing comment text using scripting and repatching objects using script connect/ script disconnect

You could also use this method to send offset messages to **bpatcher** objects that lack an inlet.

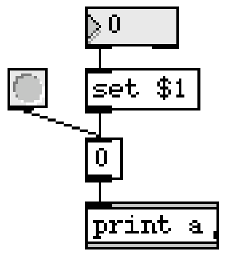
Creating Objects

The most powerful feature of scripting is the ability to create new objects. The form of the scripting command is:

script new <variable-name> <creation message>

As mentioned above, the <variable-name> field is a new object name, which is assigned to the object being created.

The message part of script new is not straightforward. You want to send a message that is identical to the format of Max text patch files. In order to understand this, let's take a look at this simple patch in its text format:



A simple patch

(You can look at the text version of any Max patch by choosing Open As Text… from the File menu.)

max v2;	header information
#N vpatcher 40 55 299 300;	patcher window definition
#P button 65 98 15 0;	object definition for **button** object
#P number 94 75 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0;	object definition for **number box** object
#P message 94 124 14 1441802 0;	object definition for lower **message** box
#P message 94 98 43 1441802 set \$1;	object definition for upper **message** box
#P newex 94 148 50 1441802 print a;	object definition for **print** object
#P connect 3 0 1 0;	patchcord
#P connect 4 0 2 0;	patchcord
#P connect 1 0 2 0;	patchcord
#P connect 2 0 0 0;	patchcord
#P pop;	create patcher window

In order to create any object in Max using scripting, use the portion of the object definition (found in the Max text file) after the #P and before the semicolon.

To create the button object shown above, the scripting command is:

```
script new mybutton button 65 98 15 0
```

To create the **number box** shown above, the scripting command is:

```
script new mynumber number 94 75 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0
```

Without going into great detail about each object, it's impossible to explain what all of the numbers after the name (or class) of the object (button, number, message, etc.) mean. In most cases, the first two numbers refer to the horizontal and vertical position relative to the top left corner of the Patcher window. Note that if you have set a new Origin for your Patcher window by choosing **Set Origin** from the View menu, the script new message doesn't take it into account when placing objects at window coordinates.

The wide variation in object creation messages means that the most effective way to create objects using scripting is often to simply create the object desired using conventional means, and then copy the message used to recreate it from a saved patch edited as text. Once you have the correct message for creating the object, try varying some of the numbers to see what changes.

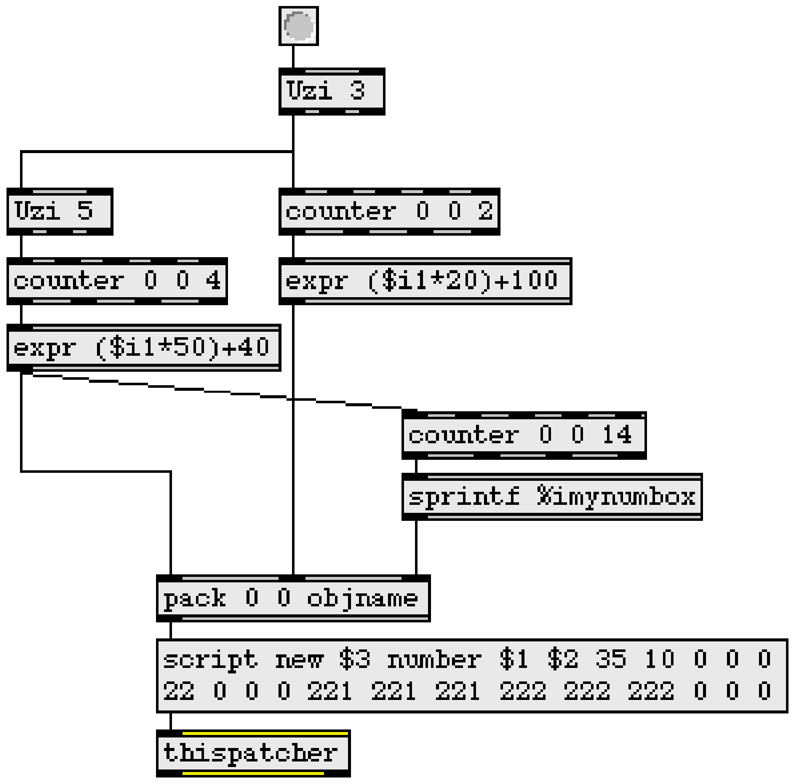
For reference when scanning Max text files, the most common object types are:

object type	*object*
newex	object box
message	**message** box
number	**number box**
flonum	float **number box**
button	**button**
toggle	**toggle**
bpatcher	**bpatcher**

Armed with this information, we can use object creation scripting to automate the task of creating of multiple instances of a similar object. For instance, let's use the **number box** object we saw above. The object definition string for the specific **number box** was

```
#P number 94 75 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0;
```

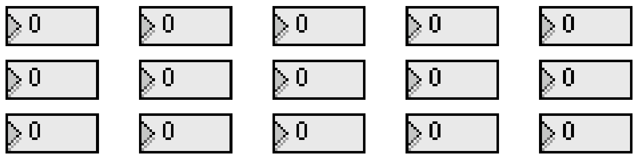
We begin by stripping off the #P and the semicolon. We also know that the first and second numbers following the word number refer to the object's horizontal and vertical positions in the Patcher window. The following patch illustrates an approach to mass-producing a flock of **number box** objects:



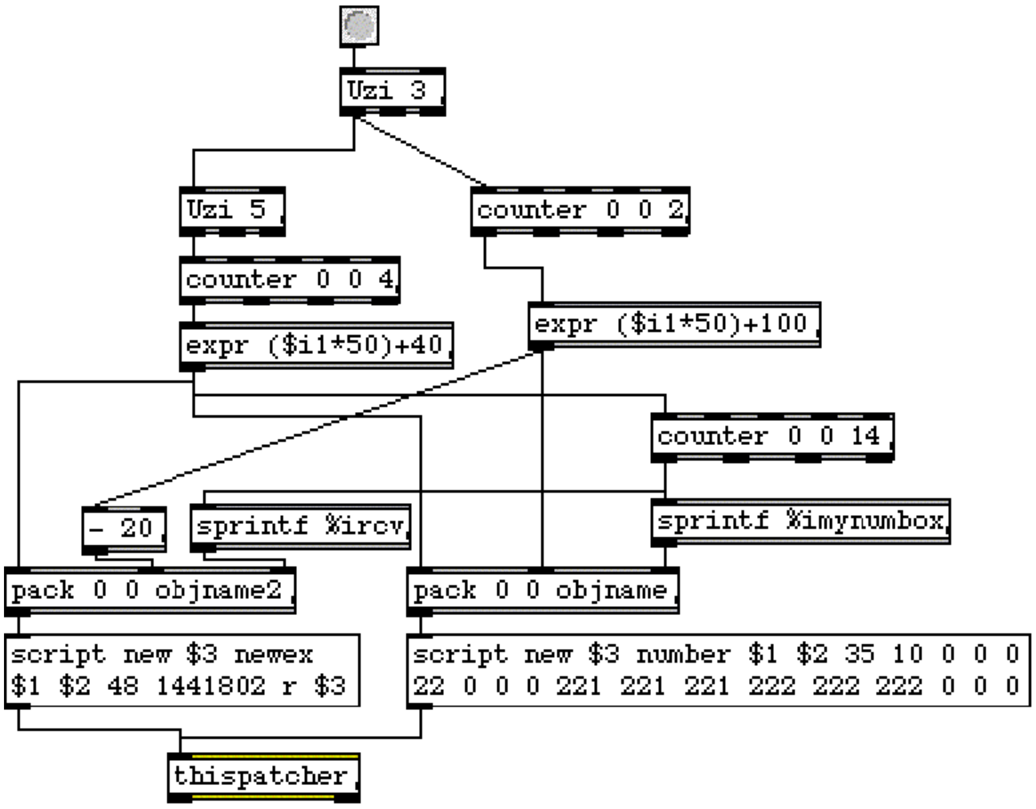
Making 15 number box objects automatically

Tutorial 46

Here it the result: an orderly series of 15 **number box** objects, uniquely named from *0mynumbox* to *14mynumbox*:



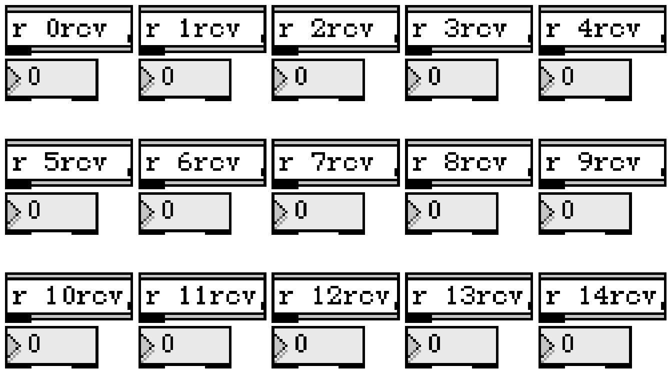
Why would you want to do this? Let's expand the patch...



Making number box and receive objects

Now we've added the ability to create **receive** objects to the patch by copying the line that created the print objects in the patch we examined above.In the case of the object box definition, the first and second numbers after **newex** refer to the horizontal and vertical coordinates, and the third number refers to the object's width. (The fourth number represents the font and font size information.).

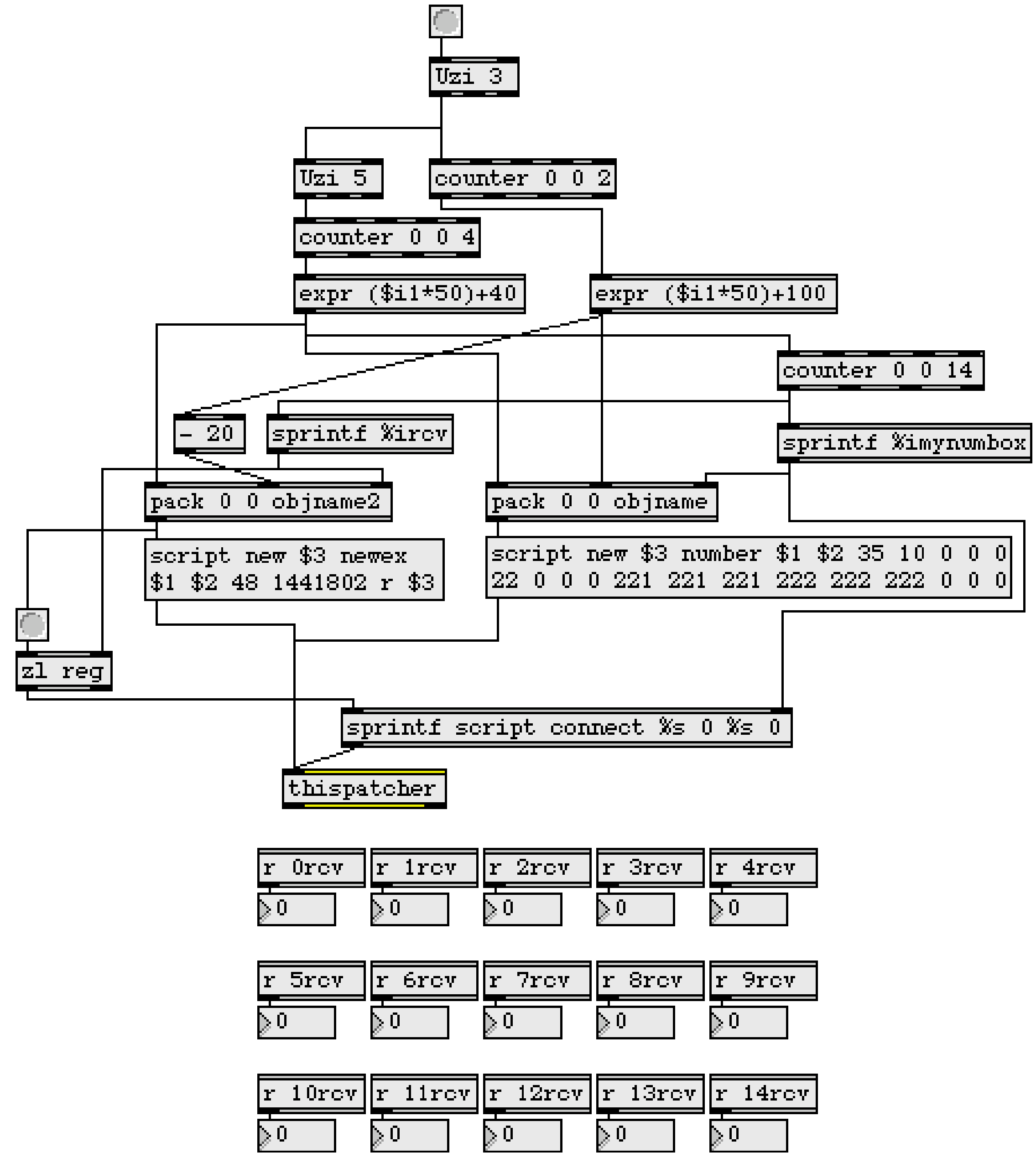
After executing the scripting commands, we obtain the result shown below:



The result

Now we'll use scripting to automate connecting the **receive** objects to the **number box** objects.

Let's finish our patch and connect everything up:



Connecting number box objects to receive objects

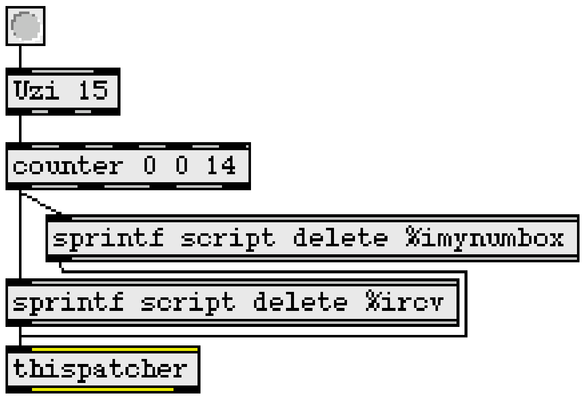
This may seem like a lot of trouble to go to just to create and hook up 30 objects, but now each object is uniquely named (with a patcher-specific scripting name, and, in the case of the **receive** objects, a global symbolic name). This means we can continue to manipulate them for the life of the patch. Using the basic technique shown above, we can create thousands of connected objects from a prototype.

Deleting Objects

To delete objects, use the script delete message:

script delete <variable-name>

This example destroys everything we worked so hard to create above:



Deleting objects

Summary

Scripting is performed by sending messages beginning with the word script to the thispatcher object. Objects must be given names in order to be scriptable. You can perform a number of tasks with scripting, including creating objects, connecting them together, sending messages, and, finally disconnecting and deleting them.

In the next Tutorial, we'll explore some more advanced uses of scripting including replacing objects, moving them around, and hiding them.

Tutorial 47: Advanced Scripting

Replacing replace

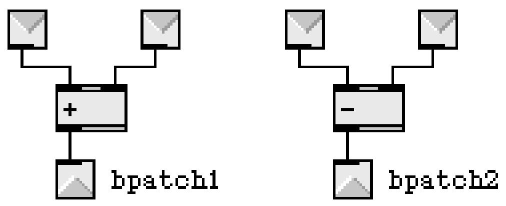
Scripting allows you to replace objects in patches. To do this, you first delete an object, then create a new object in its place and make all of the appropriate connections to and from the new object. This feature is particularly useful when you need to replace subpatchers and **bpatcher** objects within a patch where you need a part of the patch to use an algorithm that can vary—a sort of "plug-in." The previous method for replacing **bpatcher** objects dynamically (using a replace message to a **thispatcher** object inside of a **bpatcher**) has been replaced with scripting in Max 4, and offers the following improvements:

- Control is assigned to the top-level of the patch, instead of depending on a mechanism internal to a patch contained in the **bpatcher**.
- It's simple to repatch **bpatcher** objects once they are created, even if the new **bpatcher** contains a different number of inlets or outlets.

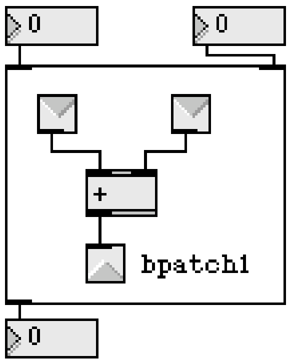
To implement a replace feature, we need to take the following steps:

1. The original **bpatcher** object must be named. Any objects connected to it via inlets or outlets should also be named.
2. To replace the object, we first delete it, using the script delete message.
3. Using the script new message, we then create a new **bpatcher** with the same object name as the previous one that refers to a new patch file.
4. Finally, we reconnect objects to the inlets and outlets of the new **bpatcher** as necessary.

Let's begin with two simple patches, bpatch1 and bpatch2 that we'd like to use inside a **bpatcher**:



Our main patch looks like this, and already contains a **bpatcher** named mybpatcher containing the bpatch1 patch:



example 2: 'replace' master patch

The Max text file for this patch looks like:

```
max v2;
#N vpatcher 31 53 416 384;
#P number 8 166 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0;
#P objectname num_bottom;
#P number 78 49 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0;
#P objectname num_topright;
#P number 8 49 35 10 0 0 0 22 0 0 0 221 221 221 222 222 222 0 0 0;
#P objectname num_topleft;
#P bpatcher 8 71 105 90 0 0 bpatch1 1;
#P objectname mybpatcher;
#P connect 1 0 0 0;
#P connect 0 0 3 0;
#P fasten 2 0 0 1 83 67 108 67;
#P pop;
```

In the text file listing above, items with names are immediately followed by a line starting with #P objectname. This is a good way to determine exactly *which* object definition string you want to grab, when you are looking at complicated Max text files. In this example, we have four named objects, *num_bottom*, *num_topright*, *num_topleft* and *mybpatcher*.

The object definition string of **bpatcher** looks like this:

```
#P bpatcher <horizontal pos> <vertical pos> <width> <height> <h-offset> <v-offset> <patchname> <border on/off [1/0]> <argument 1> <argument N...>
```

When we create a new **bpatcher** object that contains a different patcher, we'll leave everything the same except for the name. Our script sequence goes like this:

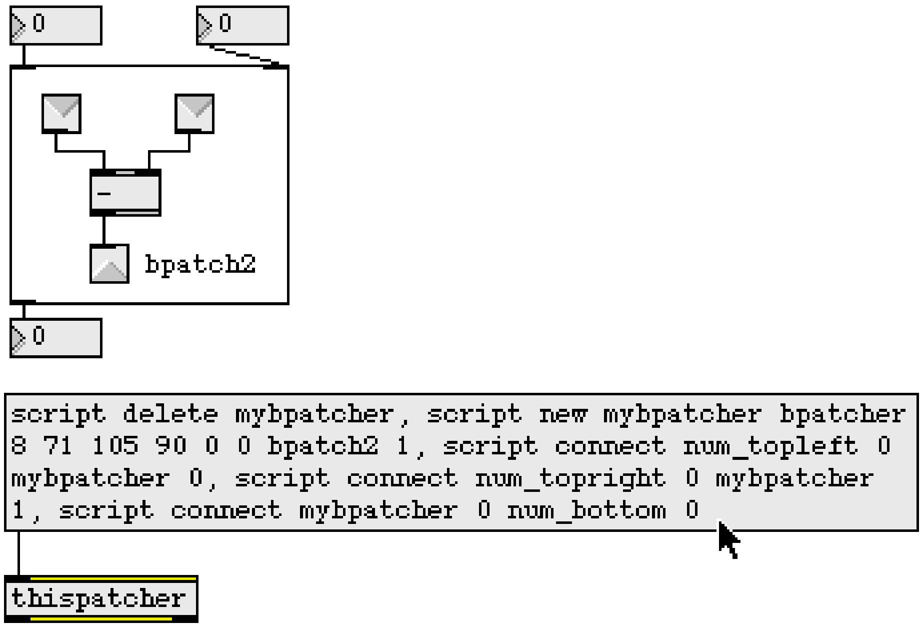
1. script delete mybpatcher

2. script new mybpatcher bpatcher 8 71 105 90 0 0 bpatch2 1

3. script connect num_topleft 0 mybpatcher 0

4. script connect num_topright 0 mybpatcher 1

5 script connect mybpatcher 0 num_bottom 0



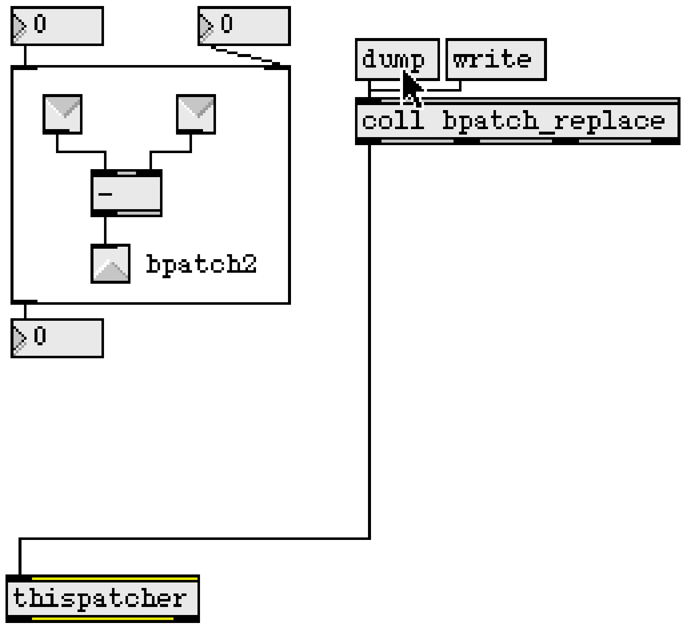
```
script delete mybpatcher, script new mybpatcher bpatcher
8 71 105 90 0 0 bpatch2 1, script connect num_topleft 0
mybpatcher 0, script connect num_topright 0 mybpatcher
1, script connect mybpatcher 0 num_bottom 0
```

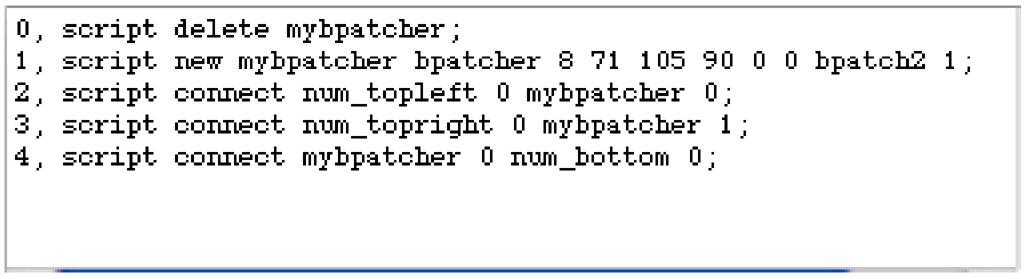
A replaced and repatched bpatcher

Where to Put a Script

As the above example makes clear, even simple scripts can become rather long. Managing your script text can be as challenging as writing it. If you find yourself routinely working on long scripts, you might consider writing them inside of a **coll** object.

Using the previous example as a model:





```
0, script delete mybpatcher;
1, script new mybpatcher bpatcher 8 71 105 90 0 0 bpatch2 1;
2, script connect num_topleft 0 mybpatcher 0;
3, script connect num_topright 0 mybpatcher 1;
4, script connect mybpatcher 0 num_bottom 0;
```

There are several advantages to this method:

- A **coll** object takes up virtually no screen space.
- You can save scripts to files, or read them in as necessary.
- You can manage multiple scripts inside of a single object.

Moving and Resizing Objects

Some of the most exciting features of scripting are the commands to dynamically move, resize and hide elements of Max patches. Using these features, flexible interface designs are straightforward to implement.

Tutorial 47

The main scripting command for moving objects is:

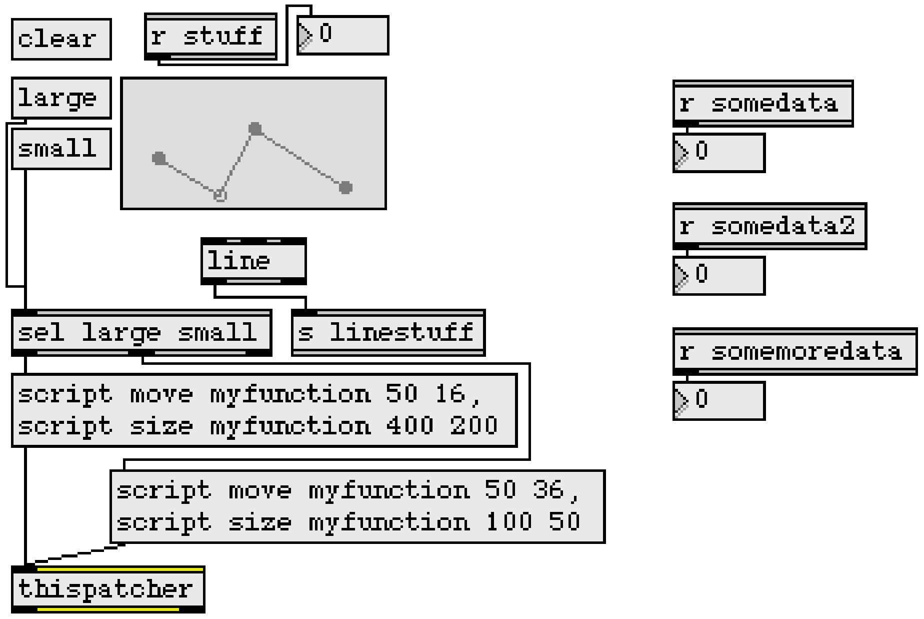
script move <variable-name> <top> <left>

The horizontal and vertical coordinates refer to the pixel location of the top left corner of the object *inside* the window.

To resize objects:

script size <variable-name> <horizontal size> <vertical size>

Again, values are in pixels. Consider the following patch:

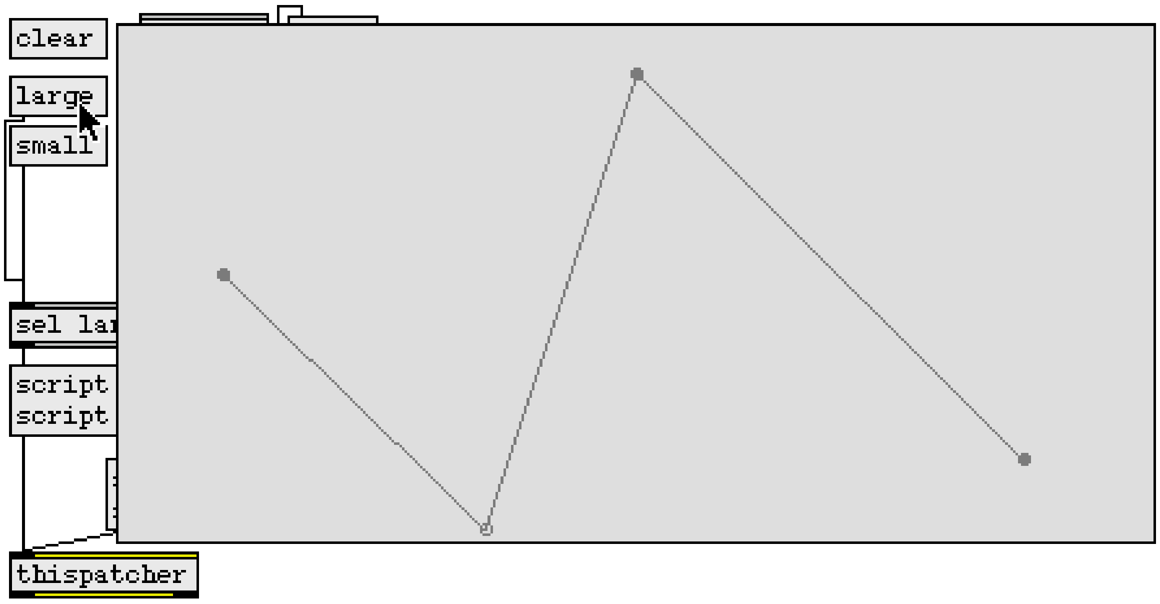


```
script move myfunction 50 16,
script size myfunction 400 200
```

```
script move myfunction 50 36,
script size myfunction 100 50
```

Using the script move and script size messages

Click on the large and small **message** box objects to trigger script commands to move and resize the function object:



The result of sending the large message

Additional commands

script messages are available for advanced object moving operations. The command script offset message permits you to specify a change in an object's location relative to its current position:

script offset <variable-name> <delta-x> <delta-y>

The script command script offsetfrom message allows you to move an object relative to the position of another object:

script offsetfrom <variable-name-to-move> <target-variable-name> <top-left-flag> <delta-x> <delta-y>

The variable <variable-name-to-move> is the name of the object you want to move, and <target-variable-name> is the name of the object being used to determine the relative position. Set the <top-left-flag> flag to 0 if you want the new position to be relative to the top left corner of <target-variable-name>, or set it to 1, and the new position will be relative to the bottom right corner of <target-variable-name>.

Hiding and Showing, and Clicking

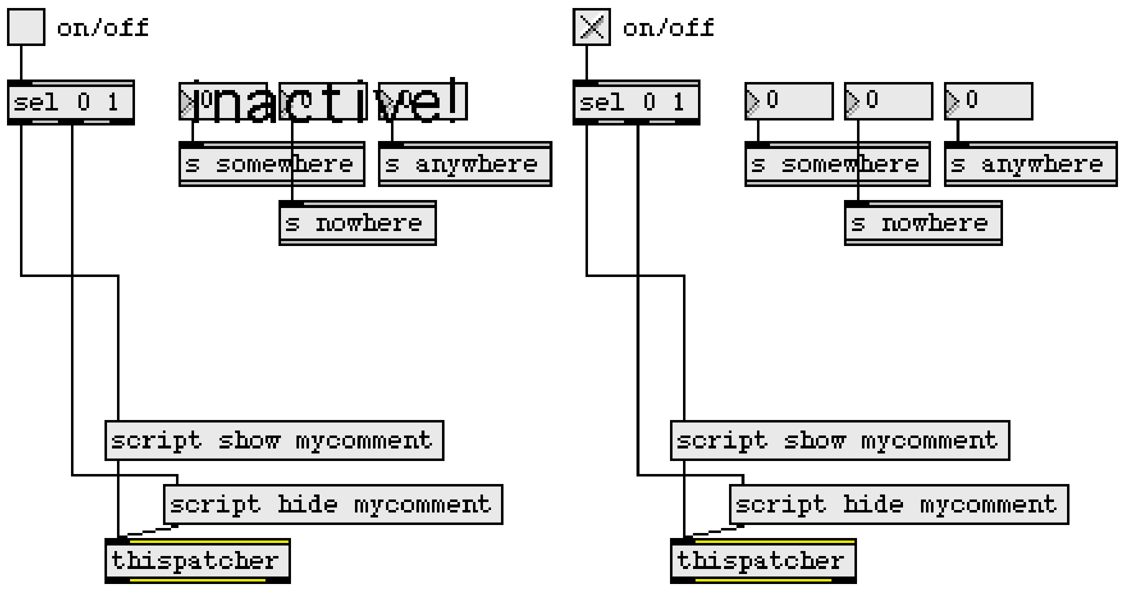
You can use scripting to hide objects using the following command:

script hide <variable-name>

To show them again:

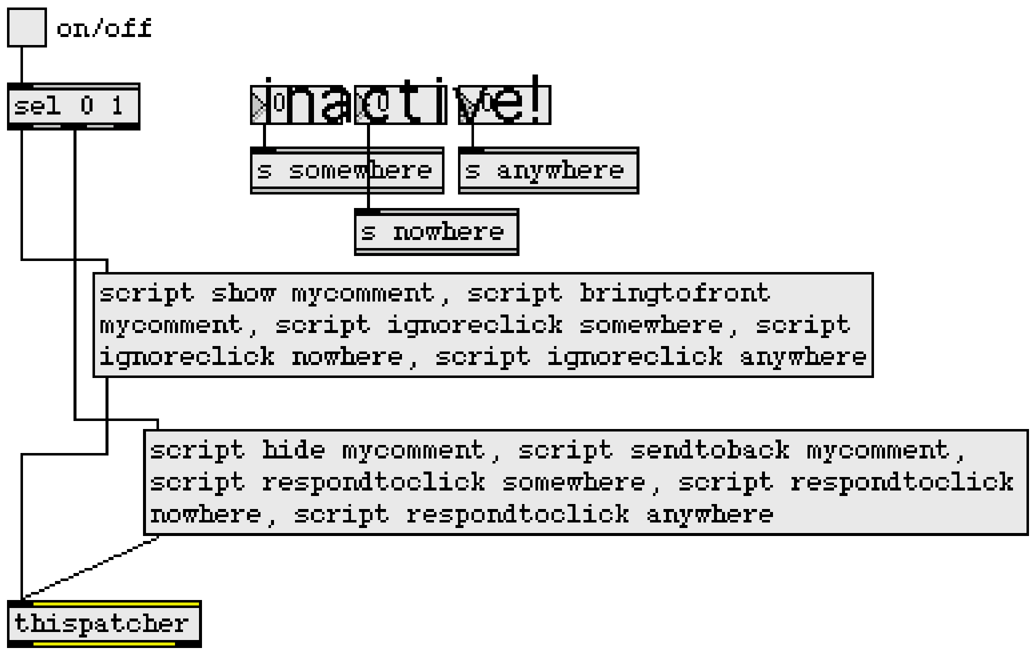
script show <variable-name>

The following patch demonstrates a simple application of script show and script hide, in which a **comment** box (named *mycomment*) is used to clearly indicate the inactive status of the interface objects.



The script show and script hide messages

The above patch makes some assumptions, however. It assumes that *mycomment* is in front of the three number boxes (otherwise, it would appear behind them). It also assumes that the user takes a simple "inactive!" sign to heart, and doesn't try to change the number box values anyway. Take a look at this variation:


on/off
sel 0 1
inactive!
s somewhere
s anywhere
s nowhere
script show mycomment, script bringtofront mycomment, script ignoreclick somewhere, script ignoreclick nowhere, script ignoreclick anywhere
script hide mycomment, script sendtoback mycomment, script respondtoclick somewhere, script respondtoclick nowhere, script respondtoclick anywhere
thispatcher

A better approach

We've added several messages. The script bringtofront and script sendtoback messages are used in the same manner as the Max menu commands **Bring to Front** and **Send to Back** to adjust the visual priority of the **comment** object. The format of those messages is:

script bringtofront <variable-name>

script sendtoback <variable-name>

To *really* deactivate those number boxes (named *somewhere, nowhere* and *anywhere*), we've also added scripting messages that enable and disable object response to mouse clicks:

script respondtoclick <variable-name>

script ignoreclick <variable-name>

Tutorial 47

Summary

You can use scripting to replace objects in a patcher and re-establish their previous connections. One important step in doing this is that all objects involved should be named prior to executing the script messages. When performing scripting operations such as replacing an object, placing all the script messages inside a **message** box can become unwieldy, and placing script messages as lines in a **coll** object is a good solution.

Tutorial 48: Basic JavaScript

What is JavaScript?

JavaScript is an object-oriented scripting language originally developed to facilitate the use of embedded software in websites. The Max **js** object allows you to use JavaScript as a language within the Max environment, writing program code for your own object without the need for external developer tools (compilers, debuggers, etc.). In this Tutorial, we'll show a simple example of how you would use the **js** object to create an object in JavaScript to respond to simple numerical input from Max and return a number to the patcher.

The **js** object gives Max users the powerful ability to write an object using an embedded programming language directly within Max. In a nutshell, it allows us to:

- Design and program procedural operations that may be difficult or impossible to implement using Max objects by themselves. These may include operations that require recursion or respond to messages with an unknown number of arguments, to give two examples.
- Create objects that respond to customizable messages and rely on their own internal data structures.
- Schedule timed events in response to messages.
- Manage global variables for use among multiple **js** objects or between **js** objects and the Max patcher.
- Interface to Max's powerful scripting architecture.
- Access the file system of your computer to look for files by name and types.

In previous versions of Max, in order to develop a Max object using a programming language other than Max itself (i.e. subpatchers) you had to develop your own external object in C using the Max Software Development Kit. You can also develop objects in Java using the **mxj** object. Both of these solutions require that the code you write be compiled into a format that Max can execute (either directly by loading a shared library or by executing Java code through the Java Virtual Machine). With the **js** object (and its graphical cousin, **jsui**), code is evaluated as a script directly by the computer as you run your patch, allowing for more immediate feedback on how the mini-program written for your **js** object will behave. That isn't to say that js doesn't check for mistakes first; it does, and we'll look at some of the ways in which the **js** object can help you catch mistakes in your program.

The Max **js** object uses version 1.5 of the JavaScript language developed by Netscape. While we don't presume specific knowledge of JavaScript in these tutorials, the better you understand the language, the easier it will be for you to develop code. The definitive reference for the version of JavaScript we use in Max can be found at the following URL:

http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/.

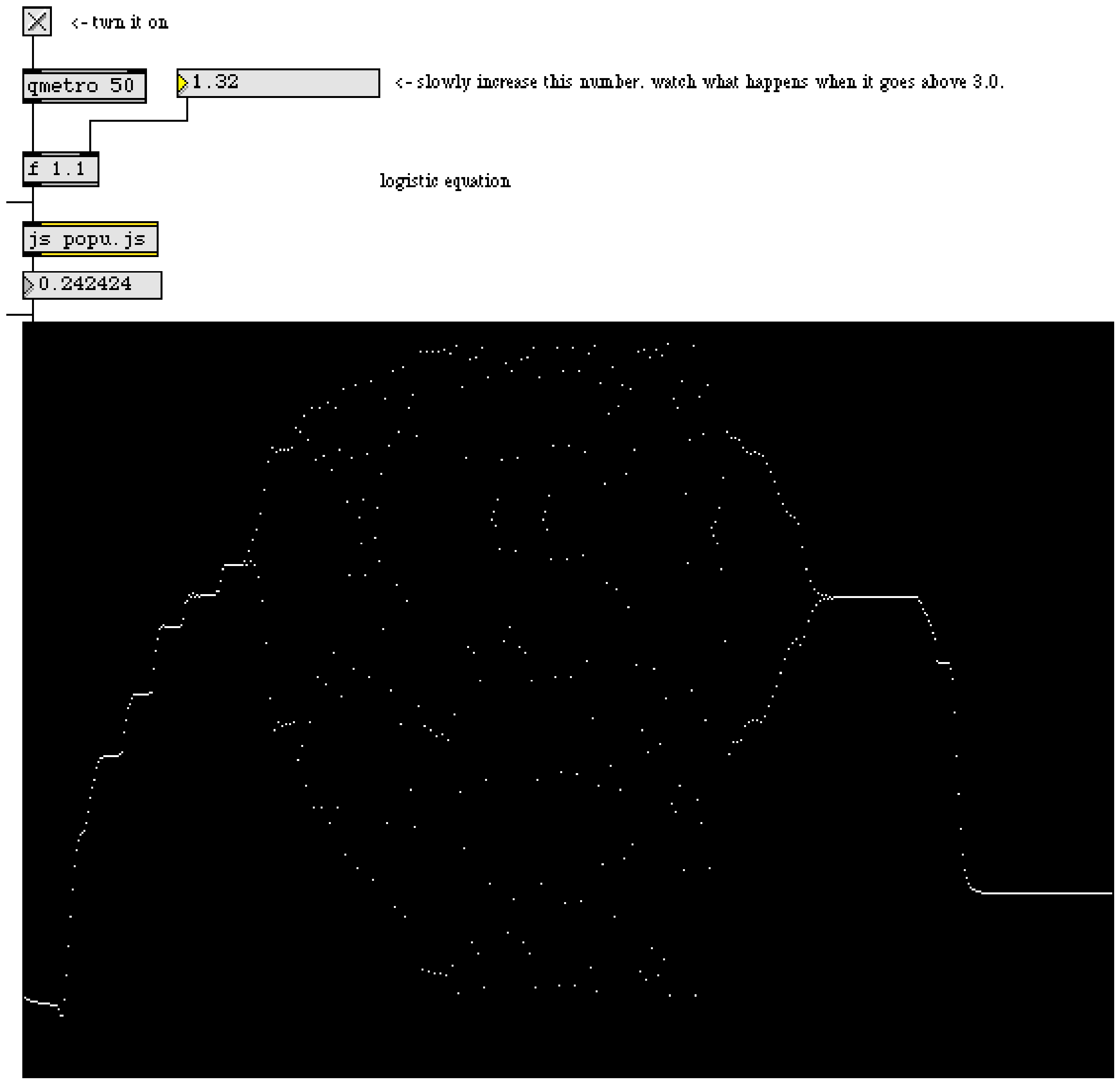
Note: There are many variations on the JavaScript language, as well as a number of extensions to the language specifically designed for its use with web browsers. The Max **js** object supports *only* the core JavaScript language (as outlined at the URL above) as well as some extensions added to the language to support interfacing with Max

JavaScript in Action

- Open the tutorial patch *48.Basic Javascript.pat*. If you want to hear the MIDI playback from the patch, make sure you have an output synthesizer configured to listen to MIDI output from Max (see Tutorial 12: Sending and receiving MIDI notes if you need review on this topic).

- Click on the **toggle** at the top of the patch to start the **qmetro** object. The **float** object sends a value into the **js** object at every tick of the metro. Slowly increment the floating-point **number box** attached to the right inlet of the **float** object. The output of the **js** object will change in response. The output number is displayed using the **multislider** object in the patch and generates MIDI notes.

As you increase the value going into the **js** object above 3.0, the value generated by our **js** object will begin to oscillate between a high and a low value. At higher values (e.g. above 3.5), the output of will become chaotic.



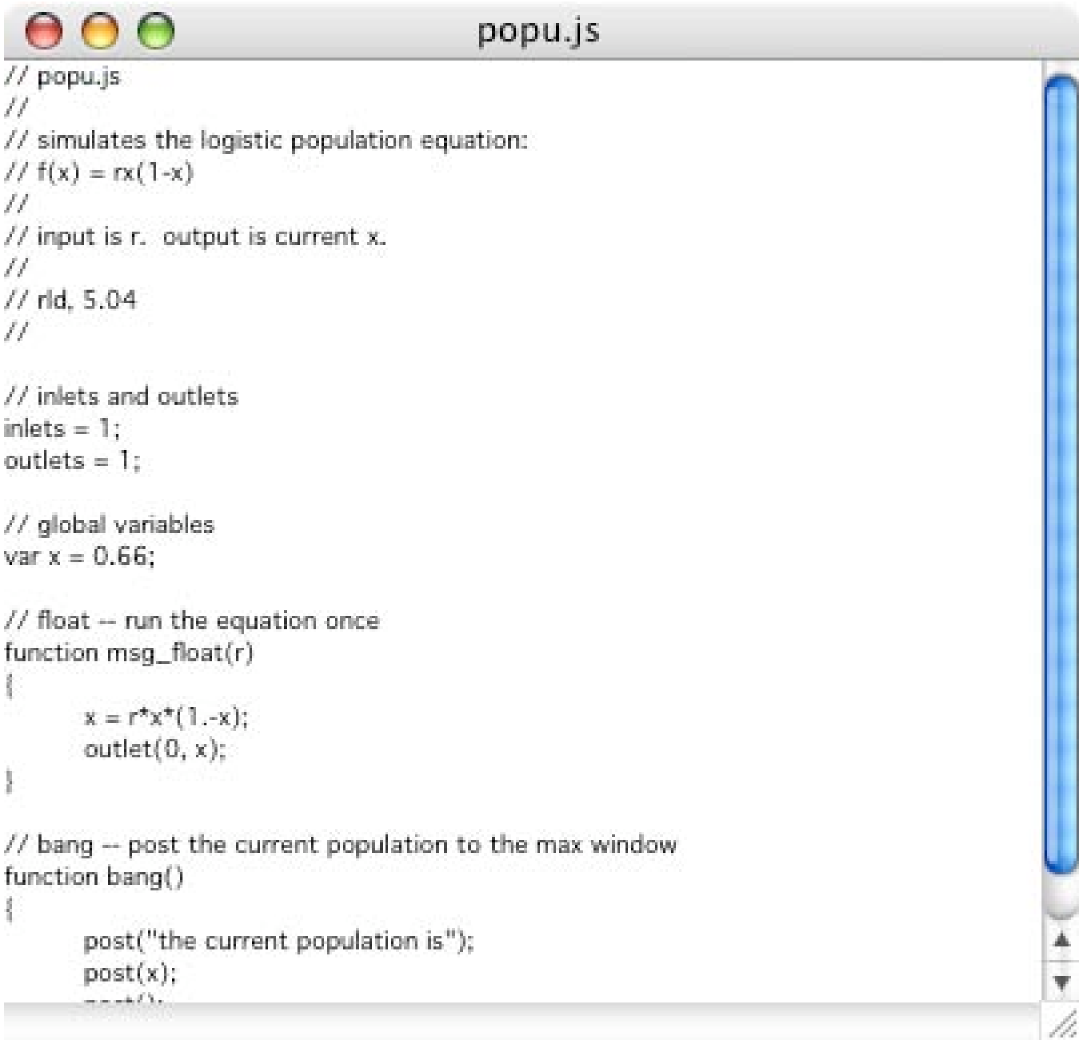
<- turn it on
qmetro 50
1.32
<- slowly increase this number. watch what happens when it goes above 3.0.
f 1.1
logistic equation
js popu.js
0.242424

The **js** object in this patch simulates a simple chaotic function called the logistic population equation. The basic formula for the function is:

$$f(x) = rx(1-x)$$

where *r* is the current input value, and *x* is the previous output value.

- Double-click the **js** object in the Tutorial patch. A text editor will appear, containing the source code for the **js** object in the patch. This code is saved as a file called 'popu.js' located in the same folder as the Tutorial patch.


```
// popu.js
//
// simulates the logistic population equation:
// f(x) = rx(1-x)
//
// input is r.  output is current x.
//
// rld, 5.04
//

// inlets and outlets
inlets = 1;
outlets = 1;

// global variables
var x = 0.66;

// float -- run the equation once
function msg_float(r)
{
        x = r*x*(1.-x);
        outlet(0, x);
}

// bang -- post the current population to the max window
function bang()
{
        post("the current population is");
        post(x);
```

The Maxjs object allows you to edit JavaScript code directly within Max through a basic text editor (the same text editor, in fact, that you use when editing the contents of a **coll** or **text** object). The source code that the **js** object loads is determined by the first argument to the object, which specifies a text file in the Max search path. If you don't give **js** an argument, you can still write a Java- script program from scratch in the editor, but you'll have to save the file in order to use it.

The Global Code

When you open the **js** object's editor you'll see some JavaScript code. The code begins with a comment block that tells us the name of the file, what it does, and who wrote it. The **js** object ignores these lines because they are prefaced with a double slash ('//'), commonly used in C++ (and other programming languages) to define a comment. C-style comments ('/*' to start, and '*/' to end) are also allowed in JavaScript.

The code after the initial comment block defines some global code for the **js** object. This is code that will let us define variables and run any part of the program we need to execute before anything happens to the object within the Max environment. In our example, we use the global code to tell Max how many inlets and outlets we need for our **js** object and to define and initialize a variable (called x):

```
// inlets and outlets
inlets = 1;
outlets = 1;
// global variables
var x = 0.66;
```

The keywords 'inlets' and 'outlets' tell Max how many of each we'll need our **js** object to have. Unlike most other changes we make to our JavaScript code, if you change the number of inlets and outlets, you will need to manually recreate the **js** object in order for it to reflect those changes. You can do this by closing and reopening the Max patch containing the **js** object, or by retyping the object box.

The 'var' keyword tells JavaScript that the label following it is to be declared as a variable, which we then assign to any value either in the declaration itself (as we do here, by saying that *x* equals 0.66) or later on in the code. Our new variable, *x*, is global in scope; we'll investigate exactly what that means later on.

In order to make changes in our JavaScript code permanent, we need to save the code in the text editor. When you save changes (by selecting **Save** from the File menu with the text editor in the front), Max will tell you that it has updated the **js** object, and will report any problems it may have had with your code.

- Type the following line underneath the 'outlets = 1;' statement in the code:
  `post("Hi There!!!");`

- Save the code in the text editor. The Max window should have printed the following:

```
js: Compiling Functions and Executing Global Code...

Hi There!!!
```

The first message tells us that the **js** object has reloaded our (changed) JavaScript code. The second line shows the output from the `post()` statement we put in. The `post()` statement prints its arguments into the Max window, letting you do exactly what the **print** object does from within **js**.

- Add the following line right underneath where we initialize *x* to equal 0.66:
  `post(x);`

When we save the code, our Max window now tells us:

```
js: Compiling Functions and Executing Global Code...

Hi There!!! 0.66
```

If you give `post()` a text string (enclosed in double-quotes) it will print it. Other letters are interpreted as variable names. In this case, we asked js to tell us the value of the variable *x*, which we had initialized on the previous line to 0.66. Note that `post()` does not put a carriage return at the end of the line; to do this, we can place a `post()` statement with no arguments.

- Add this line somewhere in between the two `post()` statements we've added:
  `post();`

Our Max window now tells us this when we save the code:

```
js: Compiling Functions and Executing Global Code...

Hi There!!!

0.66
```

Handling Mistakes

- Remove the closing parenthesis (')') from any of the `post()` statements we've added so far. Save the JavaScript code. The Max window should print an error:

```
js: Compiling Functions and Executing Global Code...

• error: js: Javascript: SyntaxError: syntax error, line 15

source line: post(;
```

This tells us that we've made a type of mistake called a *syntax error*. This means that we wrote something illegal from the point of view of JavaScript syntax; in this case, we broke the rule that says that parentheses must be balanced. Mismatched parentheses, brackets, and braces are common causes of syntax errors. Helpfully, **js** attempts to isolate which line the error occurred on (the line cited to you may be slightly different, depending on where you placed your `post()` statements).

- Go to the offending line in your JavaScript code and close the parentheses correctly. When you save the code, all should be well again.

Misspellings are another common cause of mistakes in JavaScript.

- Rewrite one of your `post()` statements so that the word 'post' is spelled wrong (feel free to be creative here). Save your script. Something like this will appear:

```
js: Compiling Functions and Executing Global Code...

Hi There!!!

• error: js: Javascript: ReferenceError: pst is not defined, line 15
```

A *reference error* means that we told JavaScript to execute a command that it simply didn't understand. While `post()` is in its vocabulary of legitimate commands, `pst()` is not.

Note: JavaScript stopped executing our global code at the point where the error occurred. In the case just cited, the words 'Hi There!!!' were printed in the Max window, but the value of *x* (0.66) was not. This gives us an important clue as to where the error lies (i.e. somewhere between the two). Using `post()` statements liberally in the development phase of your **js** code is just as important as using print objects for debugging in Max. You can always take them out later.

- Correct your spelling and save your code. Let's move on to the rest of the script.

Defining Functions

Max objects interface with one another through the use of *methods*, which are routines of code that are executed in response to messages received at the inlets of the object. In JavaScript, these methods are defined as functions within our code, each of which responds to a different type of incoming Max message. In our example **js** object, we have two functions defined. These functions (`msg_float()` and `bang()`) contain the code that **js** executes when our object receives in its inlet a floating-point number and a bang, respectively.

- Take a look at the `bang()` function first. The code looks like this:

```
function bang()
{
   post("the current population is");
   post(x);
   post();
}
```

This function tells **js** what to do if we send it a bang from our Max patch. It will print out the current value of *x* in the Max window with an explanatory preface. The `post()` statement at the end of the function tells Max to put a carriage return in the Max window so that the next message printed there will begin on a new line. Note that our function is enclosed in curly braces. Removing one of these braces will result in a syntax error.

- In the Tutorial patch, click on the **button** object connected to the **js** object. Our bang method should be working correctly.

- Look at the `msg_float()` function. Here is the code:

```
function msg_float(r)
{
   x = r*x*(1.-x);
   outlet(0, x);
}
```

Our `msg_float()` function executes the most important part of our JavaScript code, which is to run a single iteration of our cool formula. Note that, unlike our `bang()`

function, our `msg_float()` function has an argument (stated within the parentheses as *r*). This argument tells the **js** object to assign the floating-point value coming into our object's inlet to the variable *r*. We can then use this value in the rest of the function.

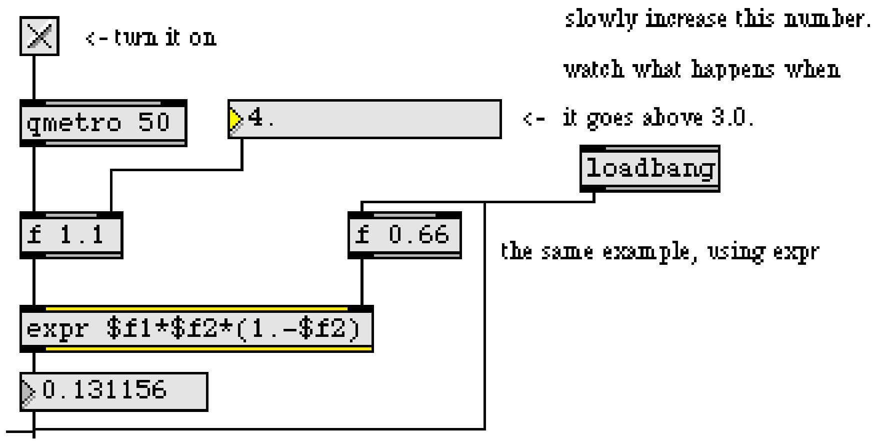
Note: generally speaking, the name of the function in JavaScript will correspond directly to the name of the message that you want to call it. For example, we respond to a bang message with the `bang()` function in our **js** object. A function called `beep()` would respond to a Max message that began with the word beep. Since float and int are reserved words in JavaScript, however, we use `msg_float()` and `msg_int()`, respectively, to define the functions which respond to floats and ints.

The main body of our float method sets *x* to the result of the multiplication of r (the value coming in the inlet), the old value of *x*, and the old value of *x* subtracted from 1.0. This statement:

```
x = r*x*(1.-x);
```

is an example of a powerful feature of using an embedded programming language within Max. To accomplish this with the **expr** object, for example, we would have to take the output of the object and feed it back around to an inlet in such a way as to prevent a feedback loop in Max, probably through the use of a **number box**.

- Double-click the subpatcher **p done_with_expr** to see how you would do this using normal Max objects. Note that because the **expr** object has no knowledge of it's previous output value, we have to store it manually and enter it again using a second inlet to the object.



The second line of code in our float method takes our new value of *x* and sends it out the **js** object's outlet to Max. The `outlet()` function takes as its arguments the number of the outlet to use and the information to send out. The outlet numbering starts at 0 for the leftmost (and, in our case, only) outlet.

- Try running the patch again, knowing how the code works. See if you can work out at what point the equation becomes chaotic, and why.

Variable Scope

The key to the success of our JavaScript code resides in the use of *x* as a global variable. JavaScript, like many scripting languages, will dynamically allocate variables as you use them, allowing you to use new variable names as you go without having to predefine them all ahead of time. These dynamic variables, however, are local to the functions within which they are used. For example, if we had a variable *i* in our `msg_float()` function, `our bang()` function would not be able to use it. Similarly, we could use *i* as a variable in both functions independently of one another. Because we explicitly defined *x* as a variable in our global code, both `msg_float()` (which evaluates *x*, sets it to a new value, and sends it out to our patch) and `bang()` (which prints the current value of *x* in the Max window) are talking about the same thing when they refer to *x*.

- Comment out the line:

```
var x = 0.66;
```

by placing two slashes ('//') at the beginning of it. Save your script, and either recreate the **js** object or reopen the Tutorial patch. Try to run the patch.

With *x* undefined, our **js** object reports reference errors when you send it a float or a bang. This is because it is trying to access a variable that has never been initialized. We could remove these errors by setting *x* to some value within each of our functions, but we would then be using *x* locally. Not only would this prevent us from sharing the value of *x* between our two functions, it would also reinitialize *x* every time we sent a float into **js**, preventing our object from maintaining *x* over multiple iterations of the function.

- Uncomment the variable declaration for x. All should be well again when you save your script.

Summary

The **js** object is a powerful tool that lets you write code using a standard programming language within Max. You define inlets, outlets, and global variables in the global code section of the script (which resides outside of any specific function). Methods that respond to particular messages (floats, **bang** messages, etc.) are written as functions below the global code, and are executed in response to the relevant messages from the Max environment. You can use the `post()` function to print information in the Max window (much as you'd use the **print** object in a patch), and you can use the `outlet()` function to send messages out of your **js** object back to the Max patcher containing it. You can write JavaScript code directly into the text editor for the **js** object; when you save a modified script, the **js** object will scan your code for programming mistakes such as syntax and reference errors, allowing you to program and debug your code from within Max.

In the next two Tutorials, we'll examine more features of JavaScript as relevant to its use in Max.

Code Listing

```
// popu.js
//
// simulates the logistic population equation:
// f(x) = rx(1-x)
//
// input is r. output is current x.
//
// rld, 5.04
//
// inlets and outlets
inlets = 1;
outlets = 1;
// global variables
var x = 0.66;
// float -- run the equation once
function msg_float(r)
{
   x = r*x*(1.-x);
   outlet(0, x);
}
// bang -- post the current population to the max window
function bang()
{
   post("the current population is");
   post(x);
   post();
}
```

Tutorial 48

See Also

js	Max JavaScript object
expr	Evaluate a mathematical expression
print	Print any message in the Max window

Tutorial 49: Scripting and Custom Methods in JavaScript

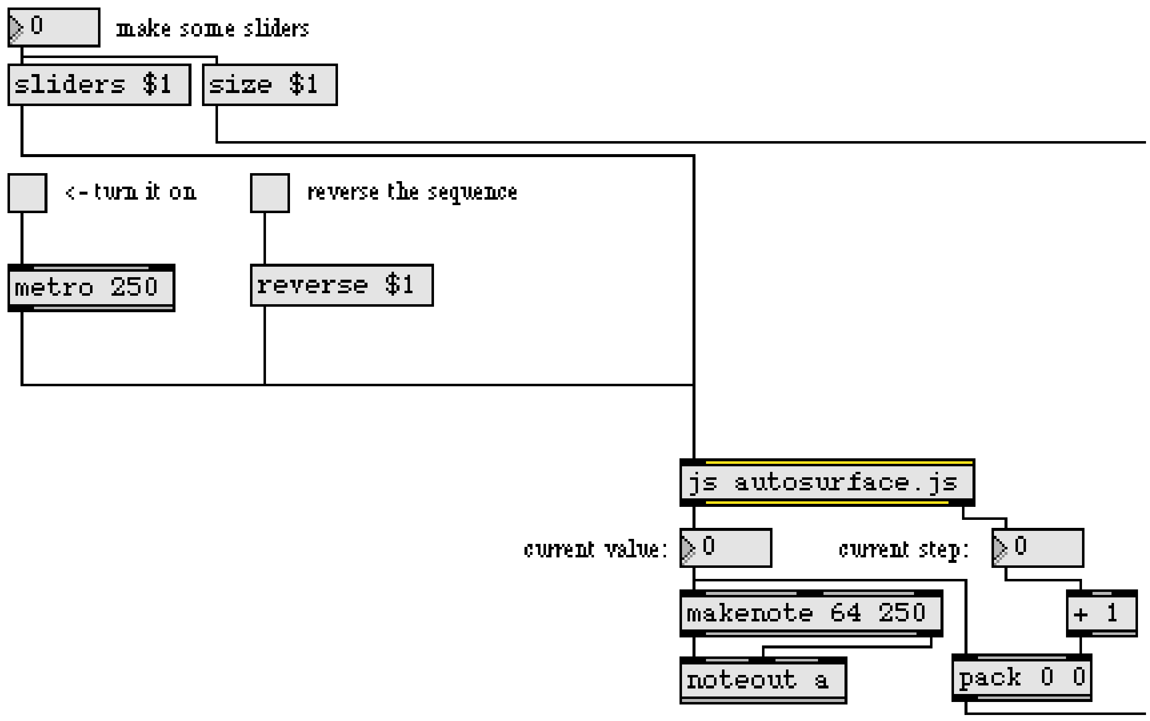
Patcher Scripting in JavaScript

With the Max **js** object, it's possible to use JavaScript code to perform patcher scripting, where you can create Max objects in a patcher dynamically, setting their properties, sending them messages, and making connections between them. JavaScript allows you to use procedural code to generate patcher elements in ways that may be more difficult to do through messages to the **thispatcher** object. This Tutorial covers how to create and delete objects and connections in a Max patch through custom methods written in JavaScript, as well as to show how to use methods to handle custom messages coming from the patcher.

Much of the basic information regarding scripting is covered in Tutorial 46: Basic scripting. You may wish to review the information presented there before proceeding with this Tutorial.

As with the previous JavaScript Tutorial, you may wish to review how to connect your MIDI synthesizer equipment in order to hear MIDI generated by the Tutorial patch. Tutorial 12: Sending and receiving MIDI notes explains how to configure and test these connections.

- When you initially open the tutorial patch *49.Javascript Scripting.pat*, you will see a largely empty patcher with a **js** object in the lower part of the patcher window. The **js** object has loaded a JavaScript source file called 'autosurface.js', which is located in the same folder as the Tutorial patch.



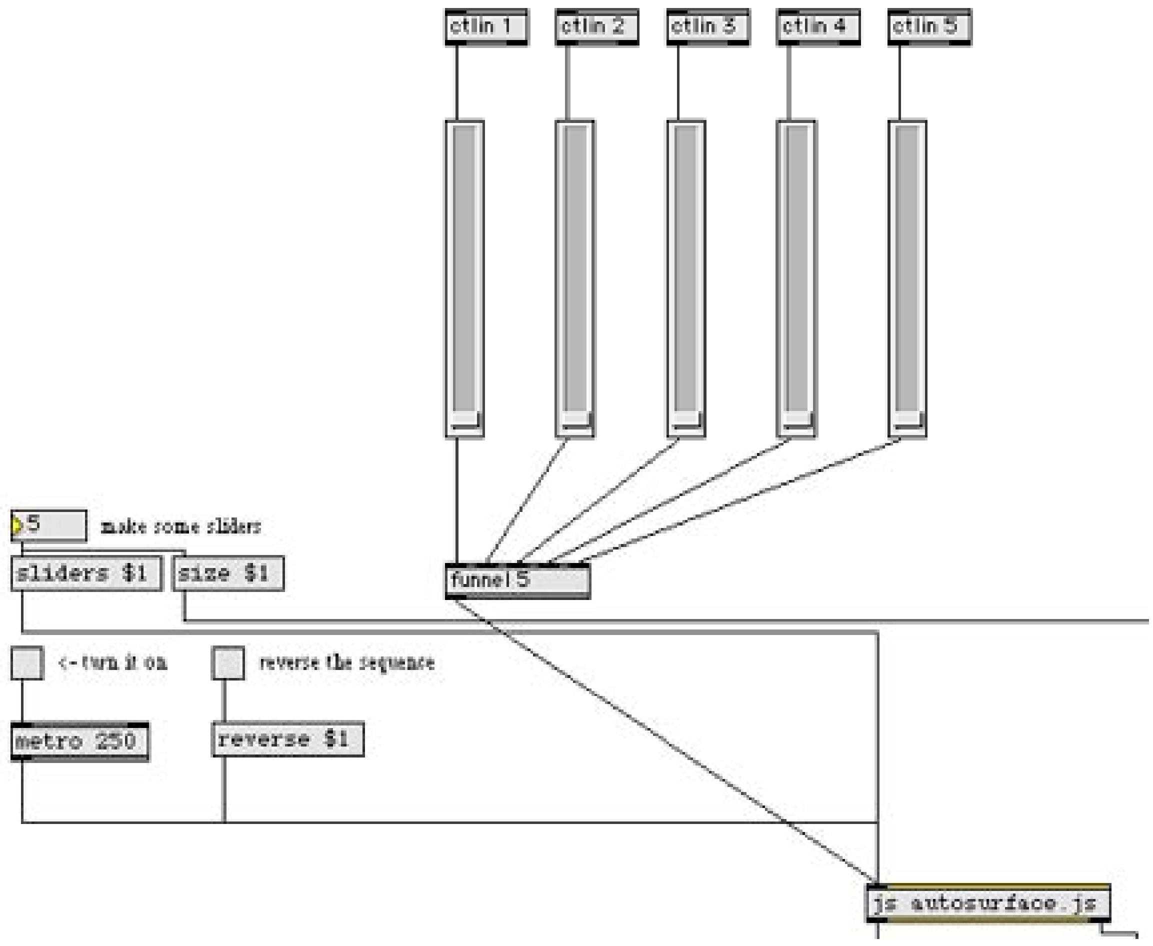
The **js** object is configured to send numbers to a MIDI output device (using the **makenote** and **noteout** objects). It also has a right outlet sending values to the right inlet of the **pack** object driving messages to a **multislider** object. In addition, our **js** object has a number of objects connected to its inlet. A **metro** object is connected to our **js** object, as are two message boxes that will send the messages sliders *n* and reverse *n*, where *n* in each case is the value present in the **number box** connected to them.

From the patch layout, we can infer that the JavaScript code in our **js** object should have at least three methods, for bang, sliders, and reverse. It actually has one more, which will become apparent when we use the patch.

Patch Auto-Generation

- Select the **number box** attached to the message box containing the sliders $1 message. Type in or scroll to the number 5, and watch what happens. Change the value in the **number box**. Try setting it to a large number (like 50).

Set it to 0, and see what happens.

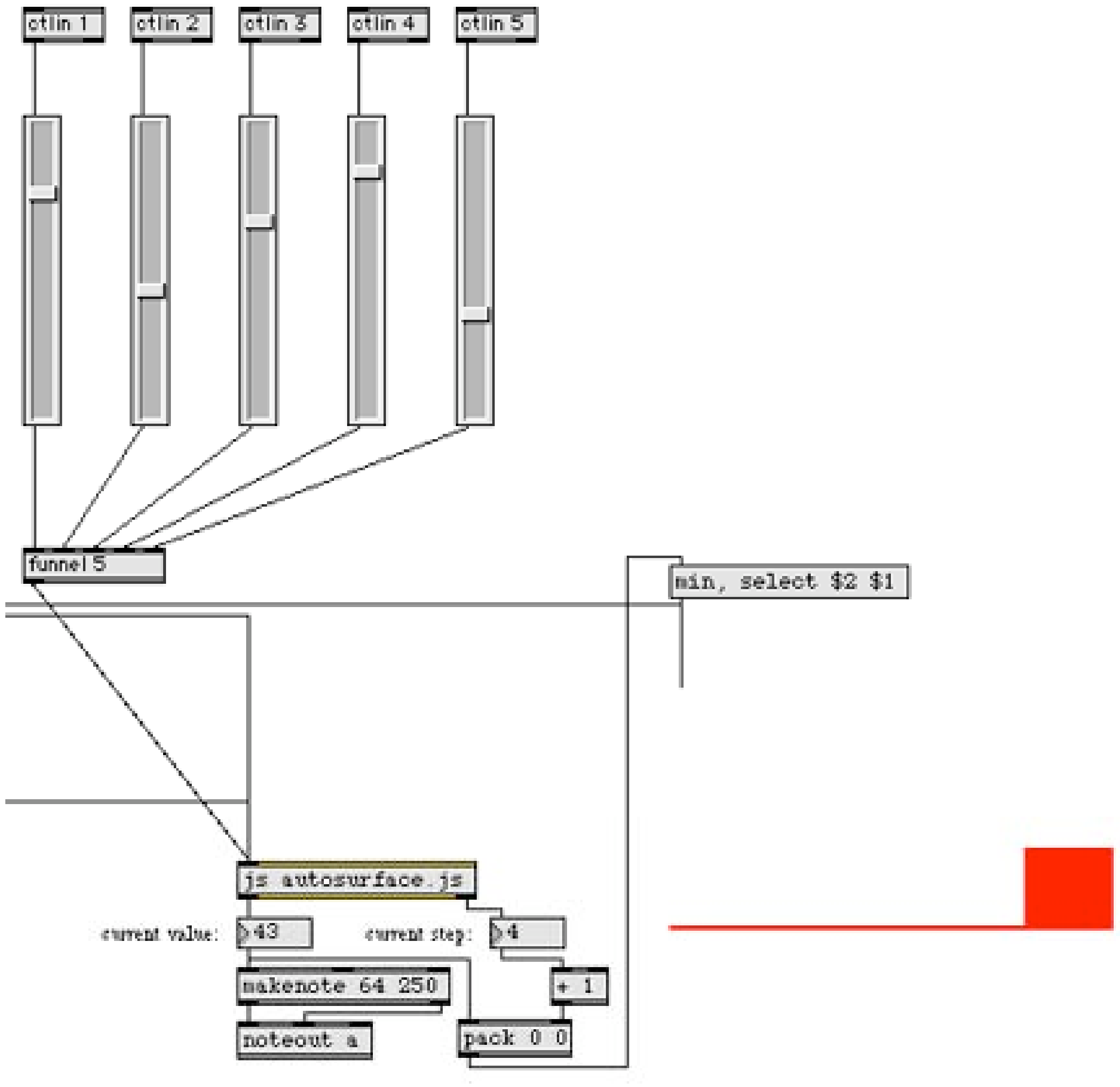


In response to our sliders $1 message, our **js** object dynamically creates Max objects and connections through scripting. It creates pairs of **ctlin** and **uslider** objects to match the number of sliders you request through the message to the **js** object. Furthermore, it creates a **funnel** object with the appropriate number of inlets for the **uslider** objects and makes the appropriate connections between them. The **funnel** object is then connected to our **js** object, allowing the values generated by the sliders to be used by our JavaScript code as well.

- As you create sliders, note that the **ctlin** objects are automatically numbered to listen to incrementing MIDI controller numbers. As a result, a MIDI control surface that sends MIDI continuous control values on multiple controller numbers will send values to multiple **uslider** objects. Also, note than when you decrement the number of sliders, the excess objects will disappear (actually, everything disappears and is recreated again). If you set the number of sliders to 0, all the script-created objects (including the funnel) will be deleted from the patch.

- Set the number of sliders to something modest, such as 5. Change the values in the **uslider** objects, either by clicking on them or by using a MIDI controller input. Turn on the **metro** object by clicking the **toggle** attached to it. The values in the **uslider** objects should come out of the **js** object in turn, creating a sequence of MIDI notes.

The **multislider** object to the right of the patch will give you a running display of the current note out of our sequencer, set at its appropriate position in the sequence.



- Click the **toggle** attached to the message box containing the message reverse $1. Note that the order in which the uslider values are sequenced is now backwards. Our **multislider** display runs backwards as well.

In brief, our **js** object dynamically creates a scalable MIDI control surface (with **ctlin** and **uslider** objects), and uses those objects' values to create a simple MIDI step sequencer. The number of sliders created by our JavaScript code determines the length of the sequence.

- Turn off both **toggle** objects, stopping the sequence and putting the sequencer transport back into 'forward' mode. Let's look at the code for our **js** object.

The Global Block: Arrays and Maxobjs

- Double-click the **js** object in the Tutorial patch. The code for 'autosurface.js' should appear.

At the top of the code should be the familiar comment block, explaining what the script does. Below that we should see our global code statements:

```
// inlets and outlets
inlets = 1;
outlets = 2;
// global variables and arrays
var numsliders = 0;
var seqcounter = 0;
var thereverse = 0;
var thevalues = new Array(128);
// Maxobj variables for scripting
var controlin = new Array(128);
var thesliders = new Array(128);
var thefunnel;
```

As we saw in the previous tutorial, our 'inlets' and 'outlets' at the top of the code tell **js** how many inlets and outlets we want in our object.

The following block of code defines some variables that our JavaScript code will need to use globally. These variables include:

numsliders	Stores how many 'sliders' (**ctlin** and **uslider** pairs) we have in our patch. This is set by the sliders message to our **js** object.
seqcounter	Stores the current point in our sequence. This is driven by the **metro** object in our patch, and therefore is changed by a bang method in our code.
thereverse	Sets whether or not our sequencer is running backwards. This is set by the reverse message to our **js** object.
thevalues	An array (see below) of values reflecting the state of the **uslider** objects in our patch. The **funnel** object in our patch sets these values by sending lists to our object.

The `new Array()` constructor creates arrays in JavaScript. The array variable thevalues, above, has 128 elements, which are accessed by bracket notation following the array name, e.g.:

```
        k = thevalues[5];
```

will set the variable *k* to the value of the sixth element (starting from 0) of the array thevalues.

```
        thevalues[n] = 55;
```

will set the element *n* of the our array thevalues to *55*.

Note that JavaScript treats Arrays as objects, so that:

```
        k = thevalues.length;
```

will set the variable *k* to the number of elements in the array thevalues. For more information on this, consult any good JavaScript reference.

After our variable declarations, we have variables that we will use to reference dynamically created objects in our Max patch. These variable names are used internally in our JavaScript code so that we can create, connect, delete, and modify objects all through properties to these objects. Objects in **js** that refer to Max objects in a patcher are referred to as *Maxobjs*. We have the following Maxobj variables in our script:

controlin	An array of Maxobjs that refer to the **ctlin** objects in our patch.
thesliders	An array of Maxobjs that refer to the **uslider** objects in our patch.
thefunnel	A Maxobj which references the **funnel** object in our patch.

Note that there is no difference in JavaScript variable declaration with relation to the *type* of value that the variable stores; integers, floats, strings, and objects are all considered equivalent when declaring a variable. Similarly, arrays are defined simply to refer to quantity of information, rather than what type of information will be stored in them. Similarly, JavaScript will correctly type variables following a calculation, e.g.:

```
        x = 4/2;
```

will set the variable *x* to 2 (an integer), whereas:

```
        x = 3/2;
```

will set the variable *x* to 1.5 (a floating-point value). Variables can switch types dynamically throughout their existence. This use of *untyped* variables only exists *within* the JavaScript environment, however, which is why we still need independent methods (`msg_int()` and `msg_float()`) to deal with differently typed numbers coming in from Max.

We will use various properties of the Maxobj object class to perform our scripting, all of which is accomplished by a single function: our `sliders()` method.

Arguments, Agreements…

Our **js** object responds to the sliders message via a method contained in the `sliders()` function (remember that the function name typically matches the message you want to trigger that function).

- Examine the code for the `sliders()` function. The comments at the top of each section explain what's happening at each step in the process:

```
// sliders -- generates and binds the sliders in the max patch
function sliders(val)
{
   if(arguments.length) // bail if no arguments
   {
      // parse arguments
      a = arguments[0];
      // safety check for number of sliders
      if(a<0) a = 0; // too few sliders, set to 0
      if(a>128) a = 128; // too many sliders, set to 128
      // out with the old...
      if(numsliders) this.patcher.remove(thefunnel); // if we've
done this before, get rid of the funnel
      for(i=0;i<numsliders;i++) // get rid of the ctlin and
uslider objects using the old number of sliders
      {
         this.patcher.remove(controlin[i]);
         this.patcher.remove(thesliders[i]);
      }
```

```
            // ...in with the new
            numsliders = a; // update our global number of sliders to
      the new value
            if(numsliders) thefunnel = this.patcher.newdefault(300,
      300, "funnel", a); // make the funnel
            for(k=0;k<a;k++) // create the new ctlin and uslider
      objects, connect them to one another and to the funnel
            {
               controlin[k] = this.patcher.newdefault(300+(k*50), 50,
      “ctlin”, k+1);
               thesliders[k] = this.patcher.newdefault(300+(k*50), 100,
      “uslider”);
               this.patcher.connect(controlin[k], 0, thesliders[k], 0);
               this.patcher.connect(thesliders[k], 0, thefunnel, k);
            }

         // connect new objects to this js object's inlet
            ourself = this.box; // assign a Maxobj to our js object
            if (numsliders) this.patcher.connect(thefunnel, 0, ourself,
      0); // connect the funnel to us
         }
         else // complain about arguments
         {
            post(“sliders message needs arguments”);
            post();
         }
      }
```

In pseudo-code, our `sliders()` function performs the following steps:

```
Check to see if the arguments for the sliders method are valid.
      If true…
      Make sure the number of sliders requested are in a reasonable range (0-128)
      Delete any objects previously created by our js object.
      Make the new objects and connect them to one another.
      Find our js object (see below) and connect our new funnel to it.
      If false…
      Post an error message in the Max window and exit the function.
```

Our JavaScript code takes advantage of two important features of procedural programming, namely conditional statements (if…else…) and iteration (for() loops). If you’ve used another programming language such as C or Java, you should find these constructions familiar. A Java- Script reference will help you with the specifics.

One of the first things we do in our `sliders()` function is check to see what the arguments were to the **sliders** message sent in from the patcher. We do this by checking the arguments property of the function itself, e.g.:

```
if(arguments.length) {
   // some code here
}
```

will execute the code between the braces only if there are a non-zero number of arguments to the message that called the function. Otherwise, that part of the code will be ignored. Similarly, you can access the arguments by number as an array:

```
a = arguments[0];
```

will assign the variable *a* to the value stored in the first argument of the message. In our case, this refers to the number of sliders we want to create.

Object Creation and Maintenance

From the perspective of using **js** for object creation in Max, the Maxobj class allows us to use our object variables to create, connect, and destroy objects. This is done by first accessing the Patcher object, which is a JavaScript representation of our Max patch. The statement:

```
this.patcher.remove(thefunnel)
```

tells **js** to find a Maxobj called *thefunnel* in the Patcher called *this* (which is always the patcher containing the **js** object) and delete it. The 'this' in the statement is actually optional, but it's worth noting that you can use JavaScript to control objects in patches other than the one in which the **js** object resides.

To create an object, we assign a variable to a new Maxobj created by the Patcher:

```
thefunnel = this.patcher.newdefault(300, 300, "funnel", a);
```

In this case, the Maxobj *thefunnel* is created to be a default object at coordinates 300 by 300 on the patcher window. The object's type is set to **funnel**, with the object's arguments set to whatever is contained in the variable *a*.

Note: the `newdefault()` method to the Patcher object creates a new object just as if you had created it manually from the palette or patcher contextual menu. This simplifies scripting substantially over using messages to the **thispatcher** object. If you wish to specify all the object parameters (object width, flags, etc.) you can use the `newobject()` method instead.

Connections are made by taking two Maxobjs and linking them using the `connect()` method to a Patcher object, e.g.:

```
this.patcher.connect(thesliders[5], 0, thefunnel, 5)
```

will connect the leftmost (0) outlet of the sixth Maxobj in the array *thesliders* to the sixth inlet of the Maxobject *thefunnel*. Remember that numbering starts at 0 for both arrays and inlet/outlet numbers.

We use iteration and arrays to create multiple objects at once, for example:

```
for(k=0;k<8;k++)
   {
   controlin[k] = this.patcher.newdefault(300+(k*50), 50,
"ctlin", k+1);
      thesliders[k] = this.patcher.newdefault(300+(k*50), 100,
"uslider");
      this.patcher.connect(controlin[k], 0, thesliders[k], 0);
      this.patcher.connect(thesliders[k], 0, thefunnel, k);
   }
```

will automatically generate 8 **ctlin** and **uslider** objects spaced 50 pixels apart on the patcher window (starting at horizontal coordinate 300), connect them to one another, and then connect them to the **funnel** object referenced by *thefunnel*. Note that the variable *k* in our JavaScript code is never declared, since we only use it as a local variable (in the `sliders()` function) and re-initialize it every time that function is called. In our actual JavaScript code in the Tutorial patch, the number 8 is replaced by the local variable *a*, which represents the number of sliders we want to create.

Finding Ourself in All of This

One important thing we accomplish in our `sliders()` method is the connection of the Java- Script-created **funnel** object to our **js** object's inlet. However, our **js** object was created by hand, not by our JavaScript code (this would be impossible, if you think about it). How do we bind a Maxobj to an object that was created independently of a JavaScript program?

```
ourself = this.box; // assign a Maxobj to our js object
```

The 'box' property of our patcher returns a Maxobj referring to our **js** object itself! We then take the variable *ourself* and assign it to our **js** object. This allows us to make connections to the object containing our JavaScript code.

We then connect our **funnel** object to our **js** object using our newly assigned Maxobj *ourself*:

```
this.patcher.connect(thefunnel, 0, ourself, 0);
```

Other Methods

The **js** object in this Tutorial doesn't just create and connect a MIDI control surface; it also reacts to messages from the control surface as well as other messages from the Max patcher.

- Open up the source code for the **js** object in the Tutorial patch again, and look for the function called `list()`:

```
// list -- read from the created funnel object
function list(val)
{
   if(arguments.length==2)
   {
      thevalues[arguments[0]] = arguments[1];
   }
}
```

As with our `sliders()` function, our `list()` function first checks out how many values we've sent in from Max, e.g.:

```
if(arguments.length==2) {}
```

The **funnel** object puts out a list corresponding to the number of the inlet receiving the value followed by the value received. For example, the number 55 arriving at the second inlet (which is really inlet number 1) will trigger the list 1 55 from the **funnel** object. We check to make sure we have two arguments in our message before we proceed in our `list()` method, as we use both the values in the list in the function. We use the first argument (which slider we moved) to determine which element of the array *thevalues* we set to the second argument (the value).

- Look at the `bang()` and `reverse()` functions in the JavaScript code.

```
// bang -- steps through sequencer
function bang()
{
   if(seqcounter>=numsliders) // reset sequencer
   {
      seqcounter = 0;
   }
   if(thereverse) // read from the array backwards
   {
      outlet(1, numsliders-seqcounter-1); // send out our
location in the sequence
      outlet(0, thevalues[numsliders-seqcounter-1]); // send out
the current note
   }
   else // read from the array forwards
   {
      outlet(1, seqcounter); // sound out our location in the
sequence
      outlet(0, thevalues[seqcounter]); // send out the current
note
   }
   seqcounter++; // increment the sequence
}
// reverse -- changes sequence direction
function reverse(val)
{
   if(arguments.length)
   {
      thereverse = arguments[0]; // flip it
   }
}
```

Our `bang()` method (which in our patch is triggered by a **metro** object) steps through a sequence of values in a manner analogous to the **counter** object. The maximum count is set by the number of sliders we have in our patch (defined by *numsliders*). The direction of the counting is always upwards, snapping back to 0 when we exceed the number of sliders. The `reverse()` function sets a variable (*thereverse*) based on the arguments for a reverse message sent in from Max. This changes the way in which the bang method reads from the array (*thevalues*) storing the numbers from our control surface of **uslider** objects. Our two `outlet()` functions send our current index value out our **js** object's right (1) outlet, followed by the value at that index in the sequence out our js object's left (0) outlet.

Note that we follow the important Max convention of outputting values from outlets in a *right-to-left order*. Otherwise, our **pack** object would be triggered by its left inlet before it receives the value it needs in its right inlet.

The `outlet()` function outputs the value at the current index in the sequence out our **js** object's left (0) outlet.

- Now that you know how the JavaScript code is working, play with the patch some more. Think about how you would recreate the sequencer using the normal Max **table** and **counter** objects.

Summary

The **js** object offers you a powerful way to create Max patches dynamically in JavaScript. Object creation is accomplished through the assignment of variables to Maxobj objects created by a Patcher object, which represents the patch in JavaScript. The `newdefault()` and `newobject()` methods allow you to create objects, which can be destroyed by a `remove()` method. The `connect()` method lets you make patcher connections between Maxobjs in your script. A Maxobj can be assigned to the **js** object itself through the 'box' property to the patcher. When designing JavaScript functions to act as methods for Max messages, the arguments passed with the messages are available through the *arguments* array from within the function.

The next Tutorial will look at how to schedule timed events in JavaScript, how to parse arguments to the **js** object itself, and how to share a variable with the Max environment.

Code Listing

```
// autosurface.js
//
// automatically generate a MIDI control surface with
// visual feedback (sliders), hook it up to a funnel
// object, and use it to drive a simple sequencer.
//
// rld, 5.04
//
// inlets and outlets
inlets = 1;
outlets = 2;
// global variables and arrays
var numsliders = 0;
var seqcounter = 0;
var thereverse = 0;
var thevalues = new Array(128);
// Maxobj variables for scripting
var controlin = new Array(128);
var thesliders = new Array(128);
var thefunnel;

// methods start here
// sliders -- generates and binds the sliders in the max patch
function sliders(val)
{
   if(arguments.length) // bail if no arguments
   {
      // parse arguments
      a = arguments[0];
      // safety check for number of sliders
      if(a<0) a = 0; // too few sliders, set to 0
      if(a>128) a = 128; // too many sliders, set to 128
      // out with the old...
      if(numsliders) this.patcher.remove(thefunnel); // if we've
done this before, get rid of the funnel
      for(i=0;i<numsliders;i++) // get rid of the ctlin and
uslider objects using the old number of sliders
      {
         this.patcher.remove(controlin[i]);
         this.patcher.remove(thesliders[i]);
      }
      // ...in with the new
      numsliders = a; // update our global number of sliders to
the new value
      if(numsliders) thefunnel = this.patcher.newdefault(300,
300, "funnel", a); // make the funnel
      for(k=0;k<a;k++) // create the new ctlin and uslider
objects, connect them to one another and to the funnel
      {
         controlin[k] = this.patcher.newdefault(300+(k*50), 50,
“ctlin”, k+1);
         thesliders[k] = this.patcher.newdefault(300+(k*50), 100,
“uslider”);
         this.patcher.connect(controlin[k], 0, thesliders[k], 0);
         this.patcher.connect(thesliders[k], 0, thefunnel, k);
      }
      // connect new objects to this js object's inlet
      ourself = this.box; // assign a Maxobj to our js object
      if (numsliders) this.patcher.connect(thefunnel, 0, ourself,
0); // connect the funnel to us
   }
   else // complain about arguments
   {
      post(“sliders message needs arguments”);
      post();
   }
}
// list -- read from the created funnel object
function list(val)
{
   if(arguments.length==2)
   {
      thevalues[arguments[0]] = arguments[1];
   }
// bang -- steps through sequencer
function bang()
{
   if(seqcounter>=numsliders) // reset sequencer
   {
      seqcounter = 0;
   }
   if(thereverse) // read from the array backwards
   {
      outlet(1, numsliders-seqcounter-1); // send out our
location in the sequence
      outlet(0, thevalues[numsliders-seqcounter-1]); // send out
the current note
   }
   else // read from the array forwards
   {
      outlet(1, seqcounter); // sound out our location in the
sequence
      outlet(0, thevalues[seqcounter]); // send out the current
note
   }
   seqcounter++; // increment the sequence
}
```

```
// reverse -- changes sequence direction
function reverse(val)
{
   if(arguments.length)
   {
      thereverse = arguments[0]; // flip it
   }
}
```

See also

js	Max JavaScript object
Data Structures	(**Max Topics**) Ways of storing data in Max
thispatcher	Send messages to a patcher
counter	Count the bang messages received, output the count
table	Store and graphically edit an array of numbers

Tutorial 50: Tasks, Arguments and Global Objects in JavaScript

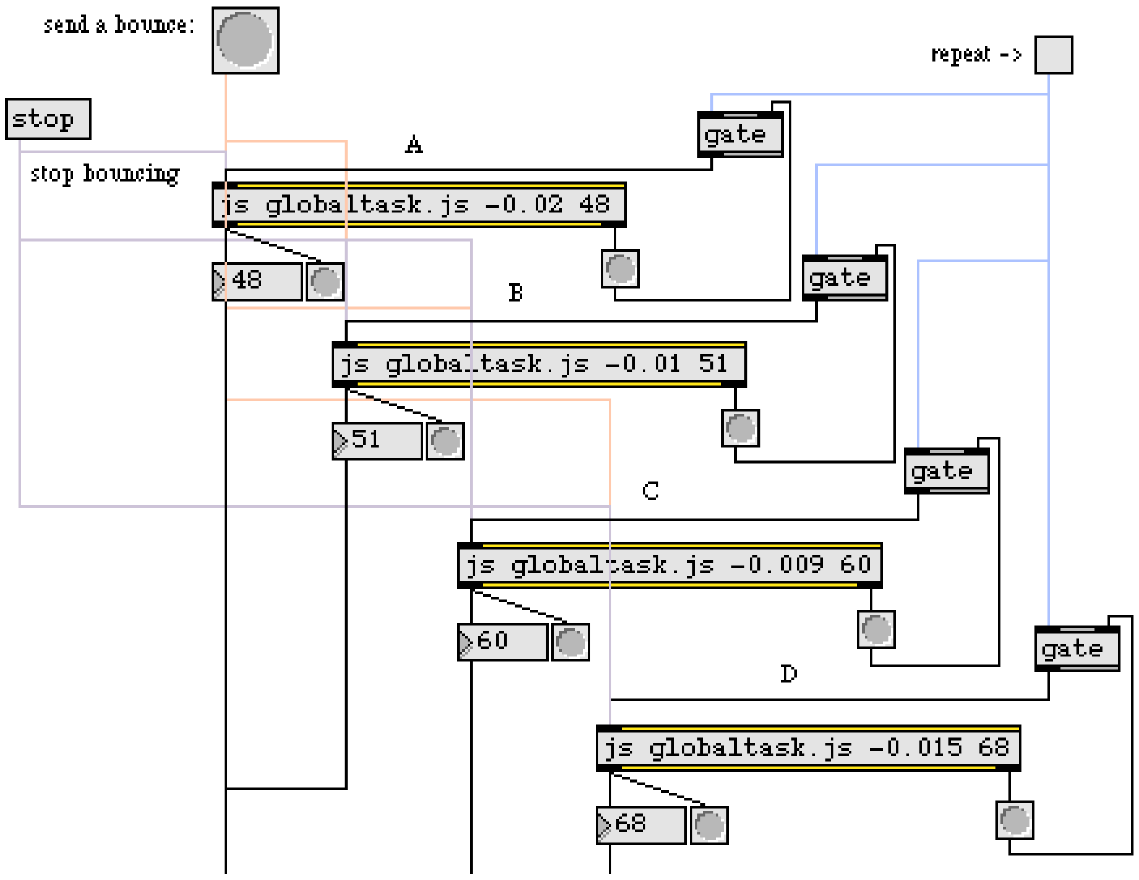
Scheduling in JavaScript

The **js** object allows you create JavaScript functions that use the Max scheduler. These functions can be triggered by Max messages to the **js** object. The timing interval at which the function is called, how many times it repeats (including whether it repeats indefinitely), and whether it begins executing immediately or at some point in the future can all be determined by your code.

In this Tutorial, we'll look at how scheduling works in JavaScript. Along the way, we'll look at two other important features of the JavaScript implementation in Max: **js** object arguments (which allow you to pass arguments directly to your JavaScript from the object box) and Global objects (which allow you to share data between internal **js** data structures and Max).

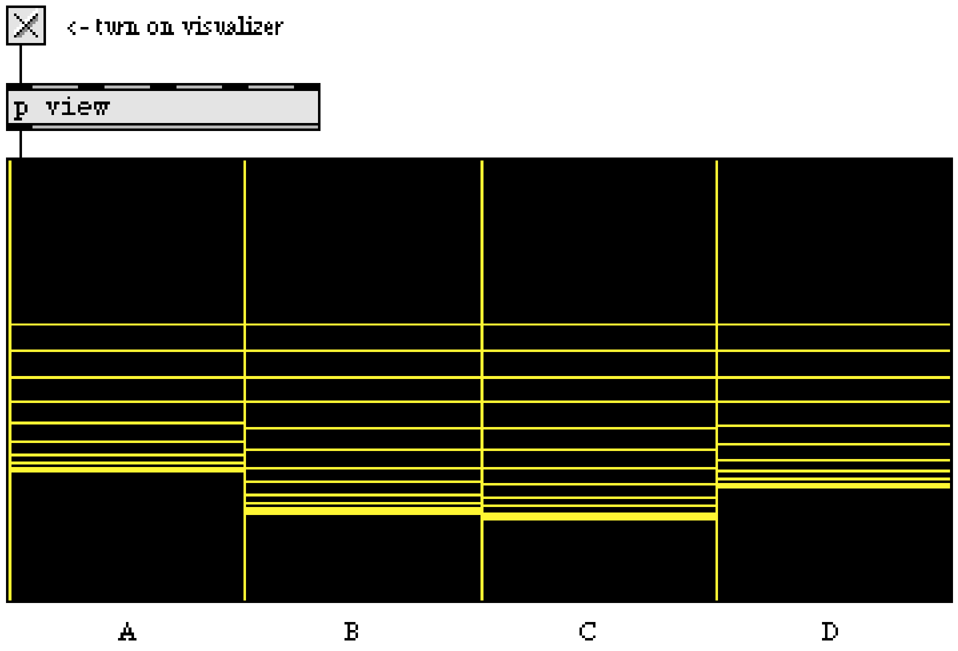
As with the previous JavaScript Tutorials, you may wish to review how to connect your MIDI synthesizer equipment in order to hear MIDI generated by the Tutorial patch. Tutorial 12: Sending and receiving MIDI notes explains how to configure and test these connections.

- Open the tutorial patch *50.Javascript Globals.pat*. You'll see four **js** objects, all of which use the same JavaScript source file (called 'globaltask.js'), which is saved on disk in the same folder as the Tutorial patch.


send a bounce:
repeat ->
stop
stop bouncing
A
gate
js globaltask.js -0.02 48
48
B
gate
js globaltask.js -0.01 51
51
C
gate
js globaltask.js -0.009 60
60
D
gate
js globaltask.js -0.015 68
68

- Click the **button** object at the top of the patch labeled 'send a bounce.' The four **js** objects should begin to generate numbers that are then sent downstream to your MIDI synthesizer output (via the **makenote** and **noteout** objects). Additional **button** objects are connected to the left outlets of the **js** objects to provide visual feedback on when a number is being sent. In addition, a patcher object called **view** receives the bang messages from the **button** object and creates a scrolling visual feedback of the generated rhythm with a **multislider** object.

Note that the specific timing of the four objects, as well as what pitches they generate, are different; these are determined by the arguments to the **js** objects (more on this later).



The JavaScript code used in the **js** objects is a simple mock-up of an exponentially decaying timing function (analogous to a rubber ball being dropped onto a hard floor). Notes are sent at an exponentially increasing rate, until the speed at which they are sending exceeds a threshold value (five milliseconds, in the case of our script). Upon exceeding that threshold, the function stops and a bang is sent out of the right outlet of our **js** objects to notify that the timing function has ceased. The use of a 'done' bang is a common convention among Max objects to signify the completion of a task (c.f. **line**, **uzi**, **coll**, etc.).

- Click on the **toggle** labeled 'repeat' and click the top **button** again. Notice that the cycle of bouncing notes repeats, as the bang from the **js** objects cause them to retrigger themselves. The difference in the timing acceleration of the four **js** objects will cause them to phase over multiple iterations of the cycle (note how this is displayed in the **multislider**). If you click the **toggle** again, the current cycle of bounces will complete in each object and then stop. Double-click any of the **js** objects in the patch; let's examine their code.

Right on Schedule

Examine the global code for our script:

```
// inlets and outlets
inlets = 1;
outlets = 2;
// define global variables and set defaults
var tsk = new Task(mytask, this); // our main task
var count = 0;
var decay = 1.0;
// defaults for arguments
var dcoeff = -0.0002; // decay coefficient
var note = 60; // note to trigger upon bounce
// process arguments (decay coefficient, note to trigger)
if(jsarguments.length>1) // argument 0 is the name of the js file
{
   dcoeff = jsarguments[1];
}
if(jsarguments.length>2)
{
   note = jsarguments[2];
}
// Global (Max namespace) variables
glob = new Global(“bounce”);
glob.starttime = 500;
```

The global code section of our JavaScript file, which defines the familiar inlets, outlets (we have two this time), and variables, has a number of things that we haven’t encountered before. The first is a variable assigned to a special object called Task:

```
var tsk = new Task(mytask, this); // our main task
```

This creates a new Task object in JavaScript referred to by the name *tsk*. When we call methods for *tsk*, they will relate to the scheduling of a function called `mytask()`. The controlling object for the Task will be our **js** object (which we refer to as ‘this’). If we would like to execute our task once, we would write:

```
tsk.execute(); // run our task function once
```

If we would like to make our task repeat every 250 milliseconds for 20 repetitions, we would write:

```
tsk.interval = 250;
tsk.repeat(20);
```

If no arguments are given to the `repeat()` method, the Task will be scheduled indefinitely, until we cancel it as follows:

```
tsk.cancel(); // cancel our task
```

The `execute()`, `repeat()`, and `cancel()` methods give us all of the flexibility we need to schedule repeating events in JavaScript. In addition to the 'interval' property of the Task object, we can also find out whether a task is running or not ("running") and how many times it has been called ("iterations"), for example.

> One important thing to keep in mind is that all methods in the **js** object (whether triggered by Max messages or scheduled internally through tasks) are executed at *low priority* in the Max scheduler. This means that, while they will always execute and send data to Max in the correct order, they cannot be relied on for critically accurate timing if the scheduler is overloaded with other actions.

Once we've defined our Task *tsk*, we trigger it through the `bang()` method to our **js** object:

```
function bang()
{
   tsk.cancel(); // cancel the bounce, if it's going already
   count = 0; // reset the number of bounces
   decay = 1.0; // reset the initial decay
   tsk.interval = glob.starttime; // set the initial task interval
   tsk.repeat(); // start the bouncing
}
```

When any of our **js** objects receive a **bang**, they cancel any previously scheduled *tsk* tasks, reset some variables that are relevant to the task function, set an initial timing interval for the task, and then start it going again.

- Start a 'bounce' in the Tutorial patch by clicking the **button** at the top. Click the **message** box labeled stop. The notes should cease.

Our `stop()` function will cancel our previously scheduled task simply by calling the `cancel()` method to *tsk*:

```
function stop()
{
   tsk.cancel(); // cancel our task
}
```

The Task at Hand

Our Task object, once set in motion by our `bang()` method, calls the function defined in its initial declaration.

- Peruse the code for the `mytask()` function:

```
// mytask -- the scheduled task - output number and reschedule next
task
function mytask()
{
   if(arguments.callee.task.interval>5) // keep bouncing
   {
      outlet(0, note); // send a note value
      decay = decay*Math.exp(++count*dcoeff); // increment the decay
variable
      arguments.callee.task.interval=arguments.callee.task.inter-
val*decay; // update the task interval
   }
   else // bounce interval is too small, so consider it 'floored'
   {
      arguments.callee.task.cancel(); // cancel the task
      outlet(1, bang); // send a bang out the right outlet to sig-
nify that we're done bouncing
   }
}
```

Unlike the other functions we've used in our JavaScript tutorials, we don't intend our `mytask()` function to be triggered by a Max message from outside the **js** object. By default, any declared function in a **js** object will respond to an appropriately named message from the Max environment. Since we don't want `mytask()` triggerable by a mytask message from our patcher, we place the following line of code after the function ends:

```
mytask.local = 1;
```

This statement makes `mytask()` local to the **js** environment, and inaccessible from outside.

Our task function accomplishes two things: it sends out an integer to Max (triggering a MIDI note), and increments its own timing interval so that the next run of `mytask()` will happen a little bit sooner. Outside of the task function, we can change our timing interval by setting the interval property to the task (e.g. tsk.interval = 250). Properties and methods of a Task object can be modified within the task function by referring to the Task as the callee, e.g.:

```
arguments.callee.task.interval=250; // adjust timing of task to 250
arguments.callee.task.cancel(); // have the task cancel itself
```

We use this reflexive capability to change our Task object's timing interval from within the task function. When the timing interval decreases to a suitably low value (5 milliseconds in our case), we also use this feature to have our task function cancel the Task that called it in the first place.

Using a Math Object

- Look at the code for `mytask()` again, paying attention to the line that changes the decay value with every bounce:

```
decay = decay*Math.exp(++count*dcoeff); // increment decay variable
```

In addition to the objects that allow for interaction between JavaScript and Max (Maxobj, Task), JavaScript has a number of core objects that can be useful when writing programs for **js**. The Math object has a large library of built-in properties and methods that allow you to perform commonly needed mathematical functions. In our code, we use the `exp()` method to the Math object, which returns the value of *E* (the base of the natural logarithm: roughly 2.71828) to the power of its argument (in this case, our decay coefficient multiplied by the next ball count). This is crucial to the modeling of the exponentially increasing rate of the bounce event.

Note: The Math object in JavaScript is roughly analogous in features to the math library in C or the **expr** object in Max (which is itself based on the C math library). A number of other predefined core objects (e.g. Date, String) provide similar extensions to the language that more-or-less match their C equivalents (e.g. time, string).

Arguments to the js Object

Two of our script's variables (*dcoeff* and *note*) are determined by the arguments given to the **js** object.

```
js globaltask.js -0.015 68
```

These arguments are parsed in our global code block by checking the **js**arguments property of our **js** object:

```
if(jsarguments.length>1) // argument 0 is the name of the js file
{
   dcoeff = jsarguments[1];
}
if(jsarguments.length>2)
{
   note = jsarguments[2];
}
```

Note that argument 0 is the name of the JavaScript file (e.g. 'globaltask.js'), so, realistically, we will usually start looking at the arguments starting at 1. The above code checks to make sure that the arguments exist before we attempt to assign their values to variables.

The Global Object

- Close the **js** object's editor and return to the Tutorial patch for a moment. In the lower-left hand corner, look at the **number box** connected to the **message** box containing the text ; bounce starttime $1. Type 2000 into the **number box** and enter the value. Send a bounce by clicking on the **button** at the top. Notice that the timing between bounces in all the objects is wider than before. Try changing the number box to a small value. The timing interval should start out quicker.

We would expect our **message** box to have sent the message starttime 2000 to a **receive** object somewhere in our Max patch called bounce. In fact, it sets the starttime property of a Global object (assigned to respond to the name bounce) to 2000 within our **js** objects. We accomplish this be declaring a Global object in our global code:

```
// Global (Max namespace) variables
glob = new Global(“bounce”);
glob.starttime = 500;
```

In our code, we’ve created a variable (*glob*) and assigned it to a new Global object. The argument to the global object (“bounce”) is the name in the Max namespace that will be tied to the object. Any message sent to bounce within Max will attempt to set properties of the Global object using that name. Note that internally, we refer to the Global object by a variable name of our choosing (*glob*), not by the symbol with which Max and our **js** object communicate.

We’ve added a property (“starttime”) to our object simply by assigning it in our global block. Now, any message beginning with starttime sent to bounce in our Max patch will set that property to its arguments.

Furthermore, this object is truly global, in the sense that not only can Max set it from outside of a **js** object, multiple **js** objects share the specific instance of this object and its properties. You could use this feature to have multiple **js** objects share information, as well as have Max broadcast information to multiple **js** objects.

Summary

JavaScript allows you to schedule events dynamically using the Task object. You create a Task and bind it to a function that gets called by the Task. You can activate and cancel the task and set the Task’s timing interval and how often it repeats. Furthermore, by using the “callee” property of the function called by the Task, you can set these things from within the scheduled event itself. All methods in **js** objects (whether called internally or by Max messages) are executed at low priority in the scheduler.

JavaScript has a number of core objects that provide functionality for common programming routines that you may find necessary. The Math object, for example, gives you access to a variety of mathematical functions that you would find in the C math library or in the Max **expr** object.

Tutorial 50

Tasks, Objects and Global Arguments in JavaScript

Arguments to the **js** object are handled by the "jsarguments" property to the object. The object starts numbering its arguments at 0, but the first argument to **js** is the name of the source file it had loaded.

Global objects in JavaScript allow communication between **js** objects, and allow for object properties to be set directly from the Max environment.

Code Listing

```
// globaltask.js
//
// generate a stream of numbers timed to an exponentially
// decaying time curve. arguments set the curve and the
// value to output.
//
// rld, 5.04
//
// inlets and outlets
inlets = 1;
outlets = 2;

// define global variables and set defaults
var tsk = new Task(mytask, this); // our main task
var count = 0;
var decay = 1.0;
// defaults for arguments
var dcoeff = -0.0002; // decay coefficient
var note = 60; // note to trigger upon bounce
// process arguments (decay coefficient, note to trigger)
if(jsarguments.length>1) // argument 0 is the name of the js file
{
   dcoeff = jsarguments[1];
}
if(jsarguments.length>2)
{
   note = jsarguments[2];
}
// Global (Max namespace) variables
glob = new Global("bounce");
glob.starttime = 500;
```

```
   // bang -- start the task
   function bang()
   {
      tsk.cancel(); // cancel the bounce, if it's going already
      count = 0; // reset the number of bounces
      decay = 1.0; // reset the initial decay
      tsk.interval = glob.starttime; // set the initial task interval
      tsk.repeat(); // start the bouncing
   }

// stop -- allow the user to stop the bouncing
   function stop()
   {
      tsk.cancel(); // cancel our task
   }
   // mytask -- the scheduled task - output number and reschedule next
   task
   function mytask()
   {
      if(arguments.callee.task.interval>5) // keep bouncing
      {
         outlet(0, note); // send a note value
         decay = decay*Math.exp(++count*dcoeff); // increment the decay
   variable
         arguments.callee.task.interval=arguments.callee.task.inter-
   val*decay; // update the task interval
      }
      else // bounce interval is too small, so consider it 'floored'
      {
         arguments.callee.task.cancel(); // cancel the task
         outlet(1, bang); // send a bang out the right outlet to sig-
   nify that we're done bouncing
      }
   }
   mytask.local = 1; // prevent triggering the task directly from Max
```

See also

js	Max JavaScript object
expr	Evaluate a mathematical expression
send	Send messages without patch cords
var	Share a stored number with other objects

Tutorial 51: Designing User Interfaces in JavaScript

The **jsui** object allows you use JavaScript to design graphical user interface objects for use in the Max environment. The JavaScript implementation for the **jsui** object is similar to that used in the **js** object, with an added API that supports two- and three-dimensional vector graphics drawn with OpenGL commands. It also includes methods for handling mouse interaction in the **jsui** object window.

In addition to the advantages provided by JavaScript, **jsui** provides a number of built-in features that make UI development flexible:

- **jsui** objects draw their geometries relative to the size of the **jsui** object box; resizing a **jsui** object will correctly resize all of the drawn elements inside of it.

- **jsui** objects work with a vector graphics language (OpenGL) that supports a wide variety of simple shape and drawing primitives. In addition, a number of higher-level graphics functions are available. The **jsui** object can also perform anti-aliasing on the image to give you as smooth an object as possible, though this comes with a decrease in performance.

- The **jsui** object allows you to draw a scene that exceeds the boundaries of the object box. By adjusting the camera orientation in the OpenGL space, you can create and manage different 'views' of the same UI object.

In addition to using the **jsui** object for user-interface design, one could use the object simply as an OpenGL graphics engine built into the Max patcher for algorithmic drawing operations.

This Tutorial assumes that you've already looked at the JavaScript Tutorials in this manual. The **jsui** object bases most of its graphics language on OpenGL functions, the specifics of which are beyond the scope of this Tutorial. The OpenGL 'Redbook' is the standard reference for these functions. An online version is available at:

http://www.opengl.org/documentation/red_book_1.0/

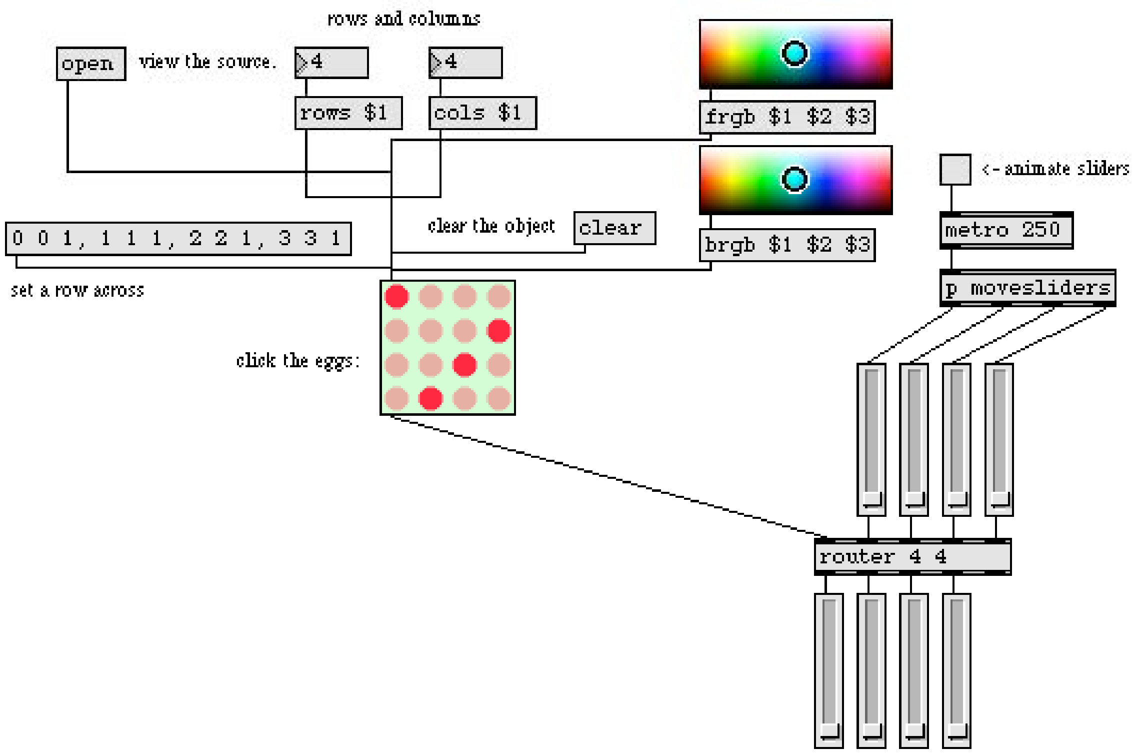
The OpenGL API supported by **jsui** is contained in an object called **jsui** sketch. This object understands most OpenGL commands and symbolic constants. Converting between OpenGL code (e.g. as given in C in the 'Redbook' code examples) and sketch methods and properties for **jsui** JavaScript code is quite straightforward if you observe the following guidelines:

- All OpenGL commands are lowercase in the **jsui** sketch object, e.g. glColor() becomes `sketch.glcolor()`.
- OpenGL symbolic constants, in addition to being lowercase, lose their 'GL_' prefix, so that GL_CLIP_PLANE1 becomes `clip_plane1()`, for example.

A number of higher-level drawing and shape commands are available which may speed up user interface development. The **jsui** sketch reference found in the **Javascript in Max** manual (and the help patch for the **jsui** object) contain lists of these commands.

jsui in Action

- Open the tutorial patch *51.Javascript UI.pat*. You will see a **jsui** object containing a grid of light red circles against a green background. Clicking on a circle inside the object will change the circle's color to a darker red. Clicking on the same circle again will change the color back to light red.



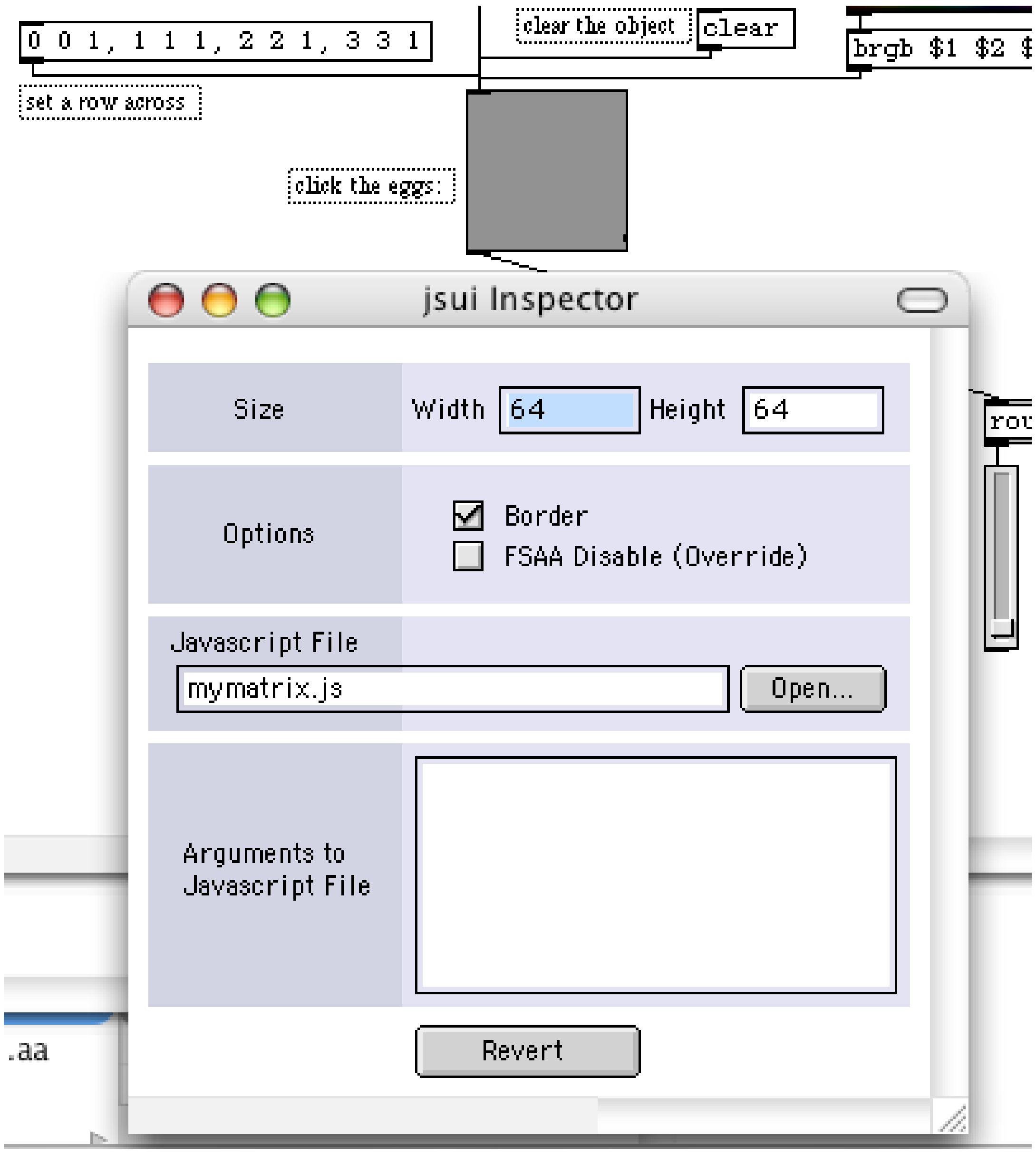
- Click on some circles so that they are highlighted (in dark red). At the right of the patch, manipulate some of the **uslider** objects above the **router** object. Note the correspondence between which circles are clicked and which **uslider** objects below the **router** echo the values from above. Click on the **message** box labeled clear attached to the **jsui** object. The circles should all go to light red and the **router** object will no longer pass messages from the **uslider** objects above. Click on the **message** box containing the list 0 0 1, 1 1 1, etc. The **jsui** object should update to show a diagonal row. The **router** object will now pass messages from the first **uslider** above to the first **uslider** below, and so on.

Our **jsui** object uses JavaScript code to emulate some of the functionality of the Max **matrixctrl** object. The columns represent the input to the **router** object; the rows specify the output. Our **jsui** communicates with the **router** object by sending lists (in the format *input output state*) that tell the **router** object to pass messages received at an inlet to all the appropriate outlets given the **jsui** object's current configuration.

Like the **js** object, the **jsui** object gets its program from a file written in JavaScript saved somewhere in the search patch. Because it is a graphical object, there is no object box to type in the name of the file. Instead, we set the JavaScript source file using the **jsui** object.'s Inspector

- Unlock the Tutorial patch and highlight the **jsui** object. Under the Object menu, select **Get Info…**. An Inspector should appear with the name of our **jsui** source file ('mymatrix.js') in the text field labeled 'JavaScript File.'

You can also set the size of the object in the Inspector, as well as turn on or off a border around the object. Disabling the object border, combined with setting the background color of your Max patch to match that of your **jsui** object, can help you design a seamless user interface.



Size	Width 64 Height 64
Options	[✓] Border [ ] FSAA Disable (Override)
Javascript File	mymatrix.js Open...
Arguments to Javascript File	

Revert

The Drawing Code

- The **jsui** object is graphical, so double-clicking the object will not open the text editor as it does with the **js** object. Instead, click the **message** box labeled open. The text editor containing our JavaScript file ('mymatrix.js') will appear. Our JavaScript file is saved on disk in the same folder as the Tutorial patch.

  As with a **js** script, our code for the **jsui** object starts with a global block that allows us to define inlets and outlets for the object and global variables for our code. It is also where we type commands that we want to occur when the object is initialized:

```
// inlets and outlets
inlets = 1;
outlets = 1;
// global variables
var ncols=4; // default columns
var nrows=4; // default rows
var vbrgb = [0.8,1.,0.8,0.5];
var vmrgb = [0.9,0.5,0.5,0.75];
var vfrgb = [1.,0.,0.2,1.];
// initialize state array
var state = new Array(8);
for(i=0;i<8;i++)
{
   state[i] = new Array(8);
   for(j=0;j<64;j++)
   {
      state[i][j] = 0;
   }
}
// set up jsui defaults to 2d
sketch.default2d();
// initialize graphics
draw();
refresh();
```

Our JavaScript code defines two global variables (*ncols* and *nrows*) which can be accessed by all functions, as well as a number of global Array objects that define colors for the drawing and a *state* array that we will use to hold information about which circles are 'on' and which are 'off' in our user interface.

Note: Multi-dimensional arrays in JavaScript are allocated by an Array object in which each element of the Array is itself an Array object (and so on, if more than two dimensions are needed). This may seem somewhat unusual if you've worked in other programming languages where multi-dimensional arrays can be declared directly (e.g. C). The `for()` loops in our global block accomplish this allocation and initialize all the elements of the state array to 0. Once a multi-dimensional array is created, it can be referenced using common bracket notation, e.g. `state[4][2]`.

Following our variable and array declarations, we find three commands that refer to the graphical behavior of the **jsui** object. The first, `sketch.default2d()`, tells our **jsui** object to initialize a number of default behaviors with the assumption that we will be giving it graphics commands for a two-dimensional scene. It sets a default view upon the OpenGL rendering context and performs a number of utility routines to make it easy for us to simply start placing graphical elements in the window. The `draw()` command (which could be named anything) refers to our main graphics function which we write to contain all the commands needed to draw the user interface of the **jsui** object. The `refresh()` command copies the OpenGL backbuffer (where the drawing is done initially to prevent flicker) to the actual screen display. Commenting out the `refresh()` command will prevent our **jsui** object from ever showing us anything.

- Below the global block, examine the `draw()` function. This is the function that provides **jsui** with all the commands it needs to draw our screen interface:

```
// draw -- main graphics function
function draw()
{
   with (sketch)
   {
     // set how the polygons are rendered
     glclearcolor(vbrgb[0],vbrgb[1],vbrgb[2],vbrgb[3]); // set the
clear color
     glclear(); // erase the background
     colstep=2./ncols; // how much to move over per column
     rowstep=2./nrows; // how much to move over per row
     for(i=0;i<ncols;i++) // iterate through the columns
     {
       for(j=0;j<nrows;j++) // iterate through the rows
       {
          moveto((i*colstep + colstep/2)-1.0, 1.0 - (j*rowstep +
rowstep/2), 0.); // move the drawing point
          if(state[i][j]) // set 'on' color
```

```
            {
               glcolor(vfrgb[0],vfrgb[1],vfrgb[2],vfrgb[3]);
            }
               else // set 'off' color (midway between vbrgb and vfrgb)
            {
               glcolor(vmrgb[0],vmrgb[1],vmrgb[2],vmrgb[3]);
            }
            circle(0.7/Math.max(nrows,ncols)); // draw the circle
         }
      }
    }
}
```

The graphics commands (everything beginning with 'gl', as well as the `circle()` command) are all methods and properties of the sketch object, which encapsulates most of the OpenGL API, much as the Math object encapsulates most common math functions.

> Note: The JavaScript `with()` statement allows us to use properties and methods belonging to an object (in this case the sketch object that provides us with our OpenGL functionality) without having to reference 'sketch' in every command. Without the `with()`, we would have to write `sketch.glcolor()` instead of `glcolor()`, `sketch.circle()` instead of `circle()`, etc. This useful trick could also be implemented in functions that rely heavily on other objects (e.g. Task or Patcher).

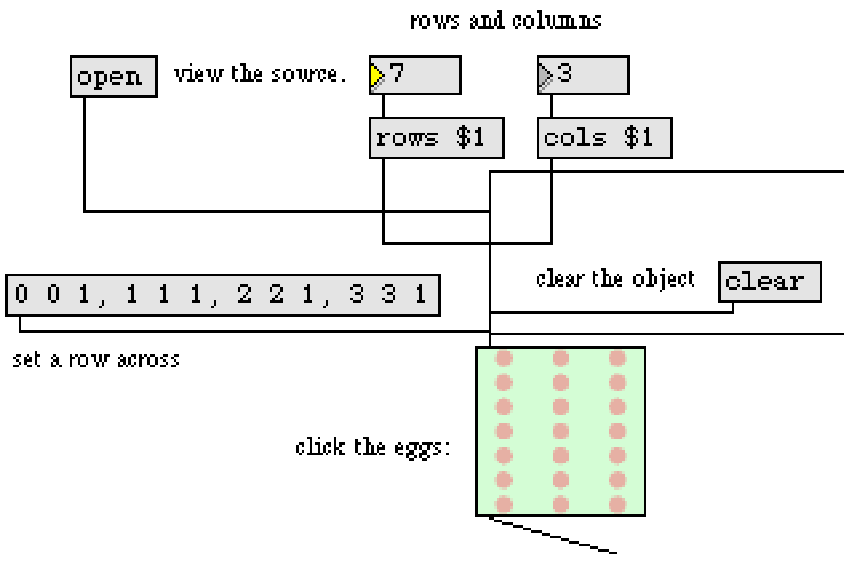
Our `draw()` function sets some default drawing behaviors and clears the window with the color defined by the *vbrgb* array. It then iterates through our *state* array based on how many columns (*ncols*) and rows (*nrows*) we've defined for our object. If a particular state element is 0 (off), it draws a circle in the color defined by *vmrgb*. If an element is 1 (on), the circle is drawn in the *vfrgb* color. The position of the circles is determined by the number of rows and columns, and is based on boundaries of our OpenGL world, which are set to be between –1.0 and 1.0 on both axes (i.e. the middle of our **jsui** window in OpenGL coordinates is 0, 0).

> Note: The size of the world is not limited to coordinates in the range of –1.0 to 1.0; our default viewport merely sets us at the center of the scene whose y range is –1.0 to 1.0 and whose x range is scaled based on the aspect ration of the object. Because our object box happens to be square, our ranges are the same on both axes. Changing the viewport (by manipulating the position of our virtual "camera", for example) will change what coordinates are visible in the jsui object's box.

Setting Parameters

- In the Tutorial patch, change the **number box** objects that set the rows and columns. Note that the object will dynamically add up to eight rows and columns of circles based on those values. Look at the `rows()` and `cols()` functions in the JavaScript code. Note that they call a `bang()` function after setting their variables.

```
// rows -- change number of rows in jsui
function rows(val)
{
   if(arguments.length)
   {
      nrows=arguments[0];
      bang(); // draw and refresh display
   }
}
// cols -- change number of columns is jsui
function cols(val)
{
   if(arguments.length)
   {
      ncols=arguments[0];
      bang(); // draw and refresh display
   }
}
```



The `bang()` function, which we call after nearly every change made to the object from Max (including mouse events), simply calls `draw()` and `refresh()` as we did in our global block, causing the **jsui** object to update its window to reflect any graphical changes.

```
// bang -- draw and refresh display
```

```
function bang()
{
   draw();
   refresh();
}
```

By only doing the drawing when necessary, we are able to reduce the amount of processor time the object uses.

- In the Tutorial patcher, change the **swatch** objects that set the frgb and brgb messages to our **jsui** object. Look at the corresponding functions (`frgb()` and `brgb()`) in the JavaScript code. Note that the array for the 'off' circle's color (*vmrgb*) is midway between the colors set by frgb and brgb messages:

```
// frgb -- change foreground (clicked) circle color
function frgb(r,g,b)
{
   vfrgb[0] = r/255.;
   vfrgb[1] = g/255.;
   vfrgb[2] = b/255.;
   vmrgb[0] = 0.5*(vfrgb[0]+vbrgb[0]);
   vmrgb[1] = 0.5*(vfrgb[1]+vbrgb[1]);
   vmrgb[2] = 0.5*(vfrgb[2]+vbrgb[2]);
   bang(); // draw and refresh display
}
// brgb -- change background color
function brgb(r,g,b)
{
   vbrgb[0] = r/255.;
   vbrgb[1] = g/255.;
   vbrgb[2] = b/255.;
   vmrgb[0] = 0.5*(vfrgb[0]+vbrgb[0]);
   vmrgb[1] = 0.5*(vfrgb[1]+vbrgb[1]);
   vmrgb[2] = 0.5*(vfrgb[2]+vbrgb[2]);
   bang(); // draw and refresh display
}
```

Note: Color in OpenGL is represented as four floating-point values in the range 0.0-1.0, corresponding to the red, green, blue, and alpha (transparency) amounts, respectively. This is in contrast to many video systems that commonly refer to color in the integer 0-255 (with no alpha value). Most of the work in our `frgb()` and `brgb()` functions is to convert from the latter (used by the **swatch** object) into the former (understood by the **jsui** object).

Mouse Interaction

- Unlock the Tutorial patcher and resize the **jsui** object (the circles should resize dynamically). Lock the patcher and notice that the mouse clicks still change the states of the correct circles. Lock the patch again and look at the `onclick()` function in the JavaScript code.

The `onclick()`, `ondblclick()`, and `ondrag()` functions, when defined, tell our **jsui** object what to do when a user clicks, double-clicks, or drags the mouse across the object. The function is called with arguments for where in the object's window the action occurred, as well as a number of flags (such as whether the mouse was depressed, the state of the shift key, etc.). In our `onclick()` function, we only use first two arguments, corresponding to the *x* and the *y* of the mouse click.

```
// onclick -- deal with mouse click event
function onclick(x,y)
{
   worldx = sketch.screentoworld(x,y)[0];
   worldy = sketch.screentoworld(x,y)[1];
   colwidth = 2./ncols; // width of a column, in world coordinates
   rowheight = 2./nrows; // width of a row, in world coordinates
   x_click = Math.floor((worldx+1.)/colwidth); // which column we
clicked
   y_click = Math.floor((1.-worldy)/rowheight); // which row we
clicked
   state[x_click][y_click] = !state[x_click][y_click]; // flip the
state of the clicked point
   outlet(0, x_click, y_click, state[x_click][y_click]); // output
the coordinates and state of the clicked point
   bang(); // draw and refresh display
}
```

Our OpenGL graphics world is defined in terms of floating-point coordinates (in our case, between –1.0 and 1.0). The **jsui** mouse functions return coordinates based on which pixel (counting away from the upper left-hand corner of the object box) the mouse event occurred. We need to convert between these two systems (of *world* coordinates and *screen* coordinates, respectively) in order to properly evaluate the mouse events for our grid of circles. The sketch methods `worldtoscreen()` and `screentoworld()` perform these conversions for us:

```
worldx = sketch.screentoworld(x,y)[0];
worldy = sketch.screentoworld(x,y)[1];
```

Once we know the *width* and *height* where we clicked, we can subdivide it based on how many circles we have on each axis:

```
colwidth = 2./ncols; // width of a column, in world coordinates
```

```
    rowheight = 2./nrows; // width of a row, in world coordinates
```

We can then plug in the coordinates of the mouse click to figure out which circle we clicked nearest to:

```
    x_click = Math.floor((worldx+1.)/colwidth); // which column we
clicked
    y_click = Math.floor((1.-worldy)/rowheight); // which row we
clicked
```

We then reverse the element of the state array corresponding to the circle we clicked:

```
    state[x_click][y_click] = !state[x_click][y_click];
```

After we've set the *state* array correctly, we send out a list corresponding to the change out our **jsui** object's outlet into Max. We then bang our own **jsui** object, updating the graphics to reflect the change:

```
    outlet(0, x_click, y_click, state[x_click][y_click]);
    bang();
```

Note that we have set our `onclick()` function to be *local*, so that it can't be triggered from an onclick message sent from our Max patch.

- Familiarize yourself with the JavaScript code and how it relates to the behavior of the **jsui** object in the patcher. Click on the toggle object to activate the **metro** object at the right of the patch. This will simulate some input from the **uslider** objects. Place `post()` statements in the JavaScript code to help navigate the values as they are passed from mouse click to list output.

Summary

The **jsui** object is a powerful tool to allow you to design and implement customizable user interfaces using JavaScript as a programming language. The key points in the program involve the main drawing function (which defines a sequence of commands to describe the **jsui** object's graphical display) and the mouse interaction functions `onclick()`, `ondblclick()`, and `ondrag()`. Important things to note are the differences in color representation (floating point vs. integer) and spatial coordinates (floating-point world coordinates vs. Cartesian pixel coordinates) between the OpenGL API used in the **jsui** sketch object and the Max environment, respectively.

Code Listing

```
// mymatrix.js
//
// simulates a simple grid of clickable widgets (a la matrixctrl)
//
// rld, 5.04
//
// inlets and outlets
inlets = 1;
outlets = 1;
// global variables
var ncols=4; // default columns
var nrows=4; // default rows
var vbrgb = [0.8,1.,0.8,0.5];
var vmrgb = [0.9,0.5,0.5,0.75];
var vfrgb = [1.,0.,0.2,1.];
// initialize state array
var state = new Array(8);
for(i=0;i<8;i++)
{
   state[i] = new Array(8);
   for(j=0;j<64;j++)
   {
      state[i][j] = 0;
   }
}
// set up jsui defaults to 2d
sketch.default2d();
// initialize graphics
draw();
refresh();
// draw -- main graphics function
function draw()
{
   with (sketch)
   {
      // set how the polygons are rendered
      glclearcolor(vbrgb[0],vbrgb[1],vbrgb[2],vbrgb[3]); // set the
clear color
      glclear(); // erase the background
      colstep=2./ncols; // how much to move over per column
      rowstep=2./nrows; // how much to move over per row
      for(i=0;i<ncols;i++) // iterate through the columns
      {
         for(j=0;j<nrows;j++) // iterate through the rows
         {
            moveto((i*colstep + colstep/2)-1.0, 1.0 - (j*rowstep +
rowstep/2), 0.); // move the drawing point
            if(state[i][j]) // set 'on' color
            {
               glcolor(vfrgb[0],vfrgb[1],vfrgb[2],vfrgb[3]);
            }
            else // set 'off' color (midway between vbrgb and vfrgb)
            {
               glcolor(vmrgb[0],vmrgb[1],vmrgb[2],vmrgb[3]);
            }
            circle(0.7/Math.max(nrows,ncols)); // draw the circle
         }
      }
   }
}
// bang -- draw and refresh display
function bang()
{
   draw();
   refresh();
}
// rows -- change number of rows in jsui
function rows(val)
{
   if(arguments.length)
   {
      nrows=arguments[0];
      bang(); // draw and refresh display
   }
}
// cols -- change number of columns is jsui
function cols(val)
{
   if(arguments.length)
   {
      ncols=arguments[0];
      bang(); // draw and refresh display
   }
}
```

```
// list -- update our state to respond to a change from Max
   function list(v)
   {
      if(arguments.length==3) // bail if incorrect number of arguments
      {
         state[arguments[0]][arguments[1]]=arguments[2]; // update our
   internal state based on the list
         outlet(0, arguments[0], arguments[1], arguments[2]); // echo
   the list out the outlet
      }
      bang(); // draw and refresh display
   }
   // clear -- wipe the state clean
   function clear()
   {
      for(i=0;i<ncols;i++)
      {
         for(j=0;j<nrows;j++)
         {
            state[i][j]=0; // wipe the state
         }
      }
      outlet(0, "clear"); // clear the router or matrix~ downstream
      bang(); // draw and refresh display
   }
   // frgb -- change foreground (clicked) sphere color
   function frgb(r,g,b)
   {
      vfrgb[0] = r/255.;
      vfrgb[1] = g/255.;
      vfrgb[2] = b/255.;
      vmrgb[0] = 0.5*(vfrgb[0]+vbrgb[0]);
      vmrgb[1] = 0.5*(vfrgb[1]+vbrgb[1]);
      vmrgb[2] = 0.5*(vfrgb[2]+vbrgb[2]);
      bang(); // draw and refresh display
   }
   // brgb -- change background color
   function brgb(r,g,b)
   {
      vbrgb[0] = r/255.;
      vbrgb[1] = g/255.;
      vbrgb[2] = b/255.;
      vmrgb[0] = 0.5*(vfrgb[0]+vbrgb[0]);
      vmrgb[1] = 0.5*(vfrgb[1]+vbrgb[1]);
      vmrgb[2] = 0.5*(vfrgb[2]+vbrgb[2]);
      bang(); // draw and refresh display
   }
   // onresize -- deal with a resized jsui box
   function onresize(w,h)
   {
      bang(); // draw and refresh display
   }
   onresize.local = 1; // make function private to prevent triggering
   from Max

// onclick -- deal with mouse click event
   function onclick(x,y)
   {
      worldx = sketch.screentoworld(x,y)[0];
   worldy = sketch.screentoworld(x,y)[1];
   colwidth = 2./ncols; // width of a column, in world coordinates
      rowheight = 2./nrows; // width of a row, in world coordinates
      x_click = Math.floor((worldx+1.)/colwidth); // which column we
   clicked
      y_click = Math.floor((1.-worldy)/rowheight); // which row we
   clicked
      state[x_click][y_click] = !state[x_click][y_click]; // flip the
   state of the clicked point
      outlet(0, x_click, y_click, state[x_click][y_click]); // output
   the coordinates and state of the clicked point
      bang(); // draw and refresh display
   }
   onclick.local = 1; // make function private to prevent triggering
   from Max
   // ondblclick -- pass buck to onclick()
   function ondblclick(x,y)
   {
      onclick(x,y);
   }
   ondblclick.local = 1; // make function private to prevent triggering
   from Max
```

See also

jsui	JavaScript UI object
js	Max JavaScript object
lcd	Draw graphics in a Patcher window

matrixctrl	Matrix switch control
router	Max message router

Tutorial 52: Patcher Storage

What is pattr?

Max has a wide variety of objects for storing and managing data. Objects such as **table**, **funbuff**, **coll**, **umenu**, and **ubumenu** all store information that you can set manually or via a mechanism within a patch itself. The **preset** object, which also stores data, is useful for capturing and recalling the 'states' of user-interface objects within a patcher, giving you the ability to rapidly restore settings to multiple parts of your patch in one step. See Tutorial 37: Data Structures for a review of these objects.

However, sometimes a project in Max will require a more sophisticated state management system. The pattr family of objects (**pattr**, **pattrhub**, **autopattr**, and **pattrstorage**) provides the functionality of the **preset** object and adds a more extensive set of state management features.

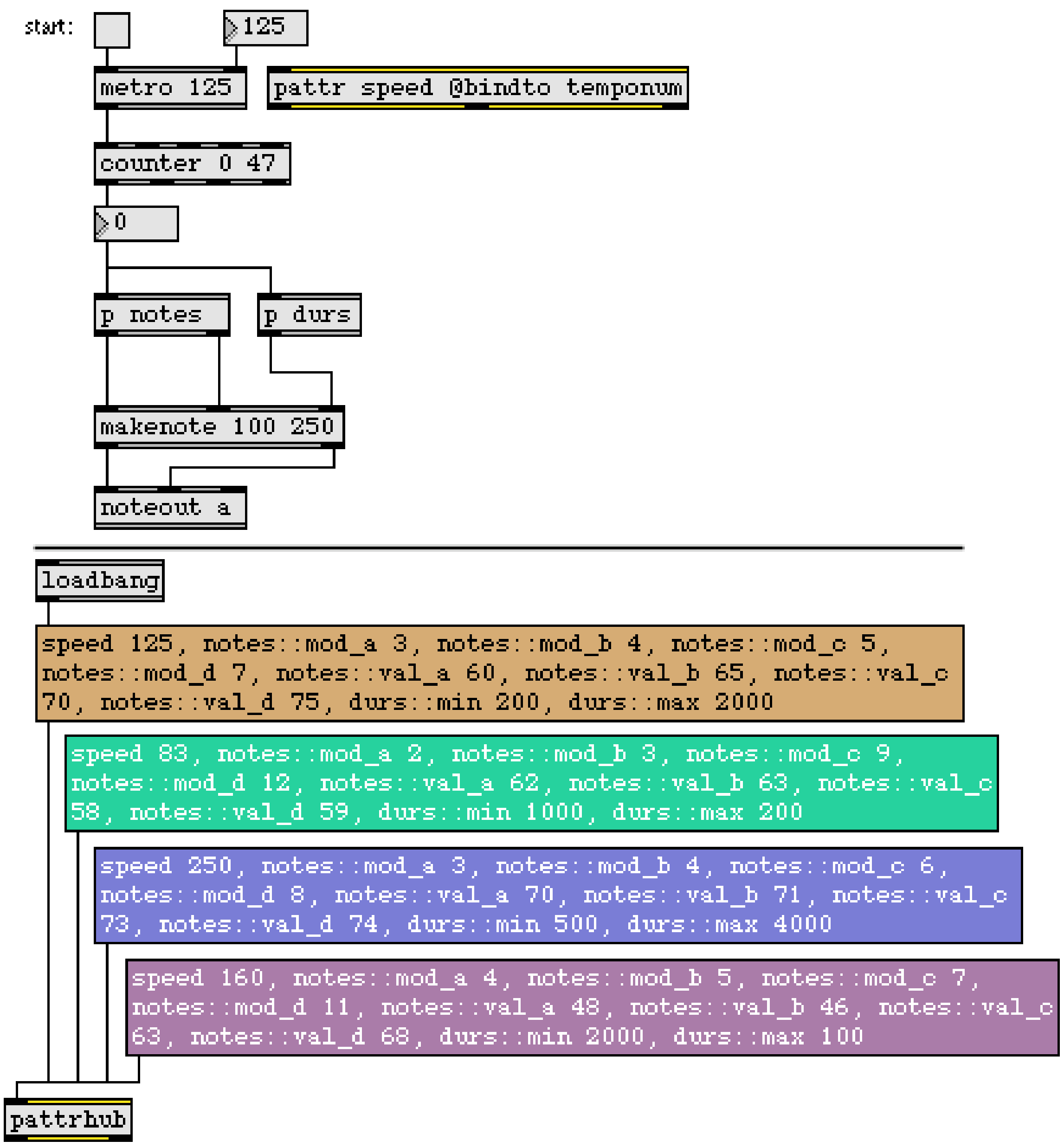
- We can maintain sets of data from objects throughout a Max patch hierarchy. For example, with **pattr** you can control the state of objects inside of **patcher** and **bpatcher** objects all from the top level of the patch.
- We can store any type of data used by any type of object (including data simply stored inside a **pattr** itself, without any other objects involved).
- We can use **pattr** objects to remotely set and query the state of objects controlled by the **pattr** system from anywhere within the patch.
- The **pattr** objects store groups of settings as XML files, allowing us to easily read and edit saved data outside of Max.
- The **pattr** objects can recall the state of objects in a specific order, avoiding difficulties with, for example, the recalling of a **toggle** object that starts a process before all the variables are in place for the process to function correctly.
- Not only can we store the states of many objects under a single address, we can also interpolate between these states, allowing for a seamless crossfade between multiple settings.
- The **pattr** objects feature a high-level interface for viewing and managing the current state of controlled objects and the states that have been saved.

Tutorial 52

In this tutorial, we'll look at some of the basic features of **pattr** and **pattrhub**, showing how the objects can be used to communicate remotely with objects anywhere in a Max patch.

Object Binding

- Open the tutorial patch *52.pattr Basics.pat* and turn on the **metro** object with the **toggle** object box labeled 'start'. You should hear some notes out of your default MIDI synthesizer. The **number box** attached to the **metro** object controls the speed of the generated pattern. Click on the colored message boxes to the right of the patch and note that they change the value of that **number box** even though there are no patch cords or **send**, **forward**, or **receive** objects in the patch.



```
speed 125, notes::mod_a 3, notes::mod_b 4, notes::mod_c 5,
notes::mod_d 7, notes::val_a 60, notes::val_b 65, notes::val_c
70, notes::val_d 75, durs::min 200, durs::max 2000
```

```
speed 83, notes::mod_a 2, notes::mod_b 3, notes::mod_c 9,
notes::mod_d 12, notes::val_a 62, notes::val_b 63, notes::val_c
58, notes::val_d 59, durs::min 1000, durs::max 200
```

```
speed 250, notes::mod_a 3, notes::mod_b 4, notes::mod_c 6,
notes::mod_d 8, notes::val_a 70, notes::val_b 71, notes::val_c
73, notes::val_d 74, durs::min 500, durs::max 4000
```

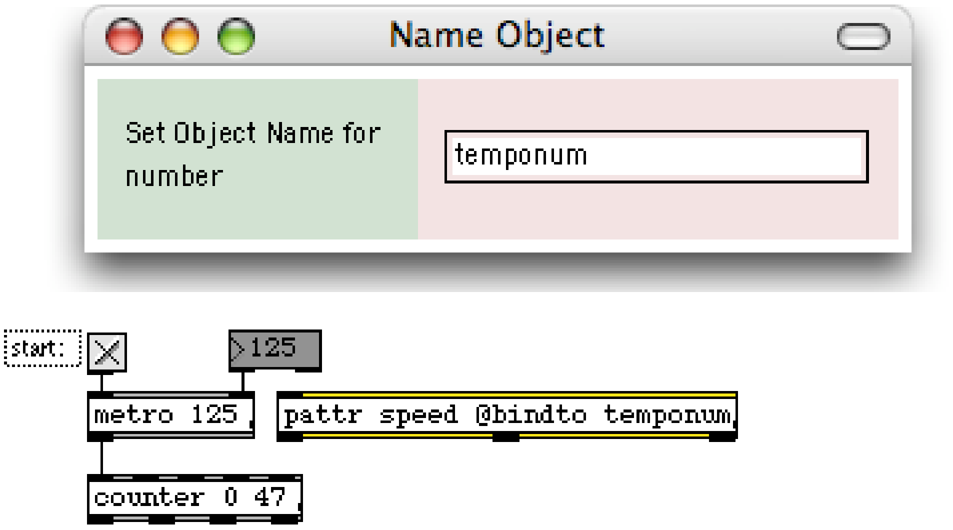
```
speed 160, notes::mod_a 4, notes::mod_b 5, notes::mod_c 7,
notes::mod_d 11, notes::val_a 48, notes::val_b 46, notes::val_c
63, notes::val_d 68, durs::min 2000, durs::max 100
```

Clicking in a colored message box changes the metro object's time value

The **pattr** object to the right of the **metro** object is controlling the value in the **number box** above it. In the language of patcher storage, we say that our **pattr** object is *bound* to the **number box**. When we send messages to that **pattr** object, it will at all times refer to the data stored in the **number box**, allowing us to ask the **pattr** object what the number is currently set to, as well as tell the **pattr** object to set the number to a specific value remotely.

Our **pattr** object has a name—speed—which is set by the object's argument and allows other pattr-family objects to communicate with it. The **pattrhub** object at the bottom of the patch can receive messages and forward them to any **pattr** objects in the patch. By sending the message speed 125 to the **pattrhub**, we are telling the object to find the **pattr** with the name 'speed.' If it finds the relevant **pattr** object, it will instruct the object to send the message 125 to whatever object it is bound to (in this case, our **number box**).

- Unlock the patch and click on the **number box** attached to the **metro** object. Select **Name…** from the Object menu.



temponum

```
metro 125
pattr speed @bindto temponum
counter 0 47
```

Our **pattr** object is bound to our number box because the bindto attribute of our **pattr** object is set to the name of our **number box** ('temponum'). For more on naming Max objects, and the application of this technique to patcher scripting, see Tutorial 46: Basic scripting. Note that the **pattrhub** object controls our **number box** via the name of the **pattr** object to which it is bound ('speed'), not the name of the **number box** itself ('temponum').

- Rename the **number box** to something else and close the Name Object inspector. Try changing settings for the patch by clicking on the **message** boxes again.

  Note that if we rename our referred object, our **pattr** object can no longer bind to it. In order to rebind it, we need to tell **pattr** to bind to our **number box** again.

  Open the Name Object inspector for the number box and name it back to 'temponum'. Close the inspector and recreate the **pattr** object binding to 'temponum' (either delete the object and undo the deletion, or create a new **pattr** object). Verify that it is receiving updates from the **pattrhub** object by clicking on the **message** boxes again.

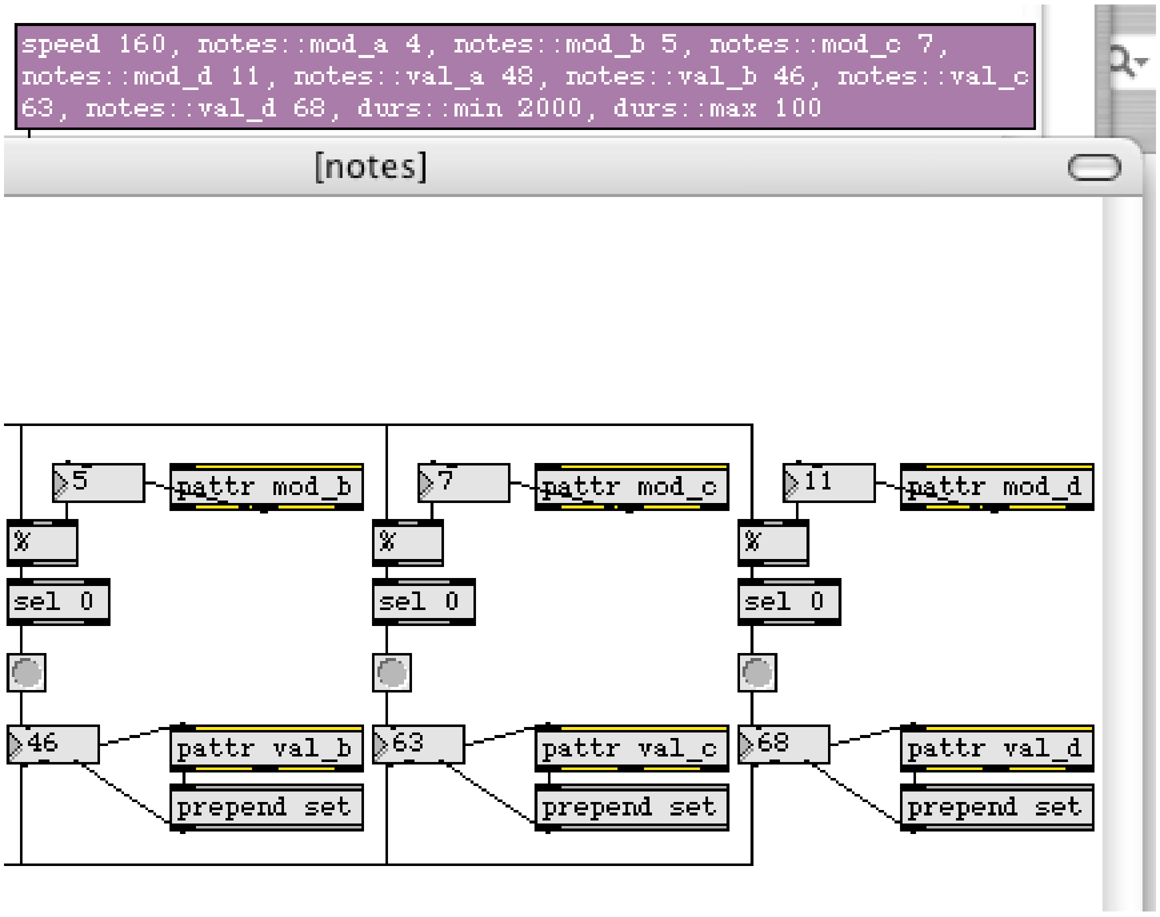
Instead of recreating the **pattr** object, we could have sent our newly 'unbound' **pattr** object a bindto message with the name of the **number box**, setting the 'bindto' attribute to a valid object.

More Ways Than One

Naming a Max object and explicitly binding a **pattr** to it with its bindto attribute is only one way to get a **pattr** to refer to an object in our patch. It has the advantage of requiring no connecting patchcords between the two objects, allowing them to communicate from spatially disparate parts of the patcher.

- Open the subpatch **notes** by double-clicking the **patcher** object containing it. This subpatch contains eight **pattr** object-bound **number box** objects that define the four pitches in our pattern generator, as well as the number of beats between each trigger of the pitch. This is the part of the patch that allows us to construct a polyrhythmic pattern off of a single **counter** object (driven by the **metro** object in the main patcher).

Click on the **message** boxes back in the main patcher with this window open and notice that the **number box** objects in the subpatch update correctly when you click on the different patterns.


```
speed 160, notes::mod_a 4, notes::mod_b 5, notes::mod_c 7, notes::mod_d 11, notes::val_a 48, notes::val_b 46, notes::val_c 63, notes::val_d 68, durs::min 2000, durs::max 100
```

Clicking on the message box updates the subpatch values, too!

In addition to binding objects to **pattr** through explicit naming and the bindto attribute, **pattr** objects can control other Max objects through patchcord connections.

The **notes** subpatch shows two possible ways to do this

1. Using the **pattr** object itself to store the data

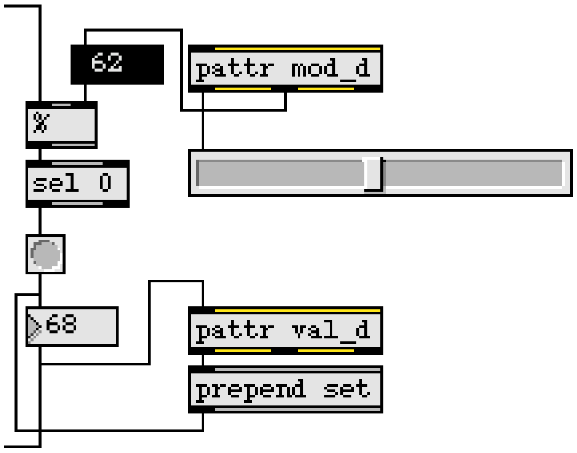
2. Automatically binding a **pattr** object to an object using the **pattr** object's middle outlet.

Tutorial 52

Internal Storage

If a **pattr** object is unbound, it will store any messages sent in its left inlet and recall them from its left outlet. Messages sent to an unbound **pattr** object from a **pattrhub** object will echo out of the left outlet as well.

- Unlock the **notes** subpatch and attach an **hslider** object to the left outlet of one of the **pattr** objects named 'val_a', 'val_b', 'val_c', or 'val_d'. Recall the **pattr** objects' states by clicking on the **message** boxes in the main patch. Scroll the **number box** attached to the **pattr** object. The **hslider** object should move as well.



The **pattr** objects controlling our pitches are not bound to any single object, and are maintaining the values of the **number box** objects attached to them not by referring to the objects directly, but by receiving their values as messages. As a result, the **number box objects** attached to these **pattr** objects can be deleted and replaced with other objects (or more than one object). Because the **pattr** object outputs its values when changed (either directly or remotely) we can easily attach multiple objects to the **pattr** object to set and display values stored in the object.

Note: The use of the **prepend** object with 'set' argument in **pattr** connections is to prevent the triggering of MIDI events when we recall the **pattr** object's state. Unlike most Max objects, **pattr** objects can be connected to other objects that are directly connected to them in turn without causing a stack overflow.

Automatic Binding

- In the same subpatch, try to connect an **hslider** object to the middle inlet of one of the **pattr** objects labeled 'mod_a', 'mod_b', 'mod_c', or 'mod_d'. The Max window should print an error message and the program should prevent you from making the connection.

The middle outlet of **pattr** allows for the automatic binding of the **pattr** to a single object in the same patcher. The **number box** objects controlling the **%** objects are communicating with their respective **pattr** objects just as if we had named them and bound the **pattr** objects to them manually.

- Look at the names for the **number box** objects by selecting **Name…** from the Object menu.

The 'bindto' (middle) outlet of **pattr** objects automatically gives the referring object a name if it doesn't have one already. The name is based on the class of object attached, and multiple objects of the same class in a patch will be given incrementing notation to uniquely identify them (e.g. 'number', 'number[1]', 'number[2]', etc.).

- Back in the main patch, open the **durs** subpatch of the main patch by double-clicking the **patcher** object containing it. Observe that the two **pattr** objects in there ('min' and 'max') have been bound to two **number box** objects through their middle outlets.

One Hub to Rule Them All

- Close the subpatchers and go back to the main Tutorial patch. Look at the **message** boxes again and notice the notation used in them to refer to **pattr** objects inside subpatchers.

Because most of our **pattr** objects are inside subpatchers, the **pattrhub** object refers to them using a colon-separated notation that takes the form

```
subpatcher::pattr_name
```

We can communicate to **pattr** objects in nested subpatchers by the same convention, e.g. a **pattr** named 'yikes' in a patcher called stuff in a **bpatcher** called things would be accessed through **pattrhub** with the name:

```
things::stuff::yikes
```

- Using the **ubumenu** object in the main Tutorial patch, select one of the **pattr** objects in the patch.

In addition to setting the state of **pattr** objects through **pattrhub**, we can also query the state of **pattr** objects, which will then tell **pattrhub** their internal state or the value of the object to which they're bound. We do this by sending **pattrhub** the message get followed by the name of the **pattr** object we want to query, with no space in the message, e.g. getspeed will poll the state of the **pattr** named 'speed'.

The **number box** to the right of the **ubumenu** reflects the current value of the selected **pattr**, output by the second outlet of **pattrhub** as a message beginning with the name of the queried **pattr**.

- Change the value of the selected **pattr** by changing the **number box** next to the **ubumenu**. Check to see that the value changed by looking at the **number box** controlled by that **pattr**. Select some other **pattr** objects and change their values, familiarizing yourself with how the system allows you to query and set the states of objects anywhere in the patch.

Summary

The **pattr** objects provide a powerful way to manage data from multiple objects in a Max patch. The **pattr** object can maintain its own internal information (sent and returned through its leftmost inlet and outlet), or it can bind to a Max user interface object either by naming that object and using the **pattr** object's bindto attribute, or by using the middle outlet of the **pattr** object to automatically name and bind to an object. Each **pattr** object can only bind to one object at a time. The **pattrhub** object allows you to control the state of all **pattr** objects within a patch, including those within subpatchers, which can be accessed through a hierarchical notation system of successive patcher names separated by double colons (::) followed by the name of the appropriate **pattr** object. By sending a get message (followed by the name of a **pattr** with no intervening space) you can query the state of any **pattr** object within a patch from a **pattrhub**.

In the next tutorial we'll look at two other objects in the **pattr** family, which allow you to automatically bind multiple Max objects to the **pattr** system and store and recall sets of data from **pattr** objects.

See Also

pattr	patcher-specific named data wrapper
pattrhub	make your **pattr** objects work for you!
patcher	Create a subpatch within a patch
preset	Store and recall the settings of other objects
thispatcher	Send messages to a patcher

Tutorial 53: More Patcher Storage

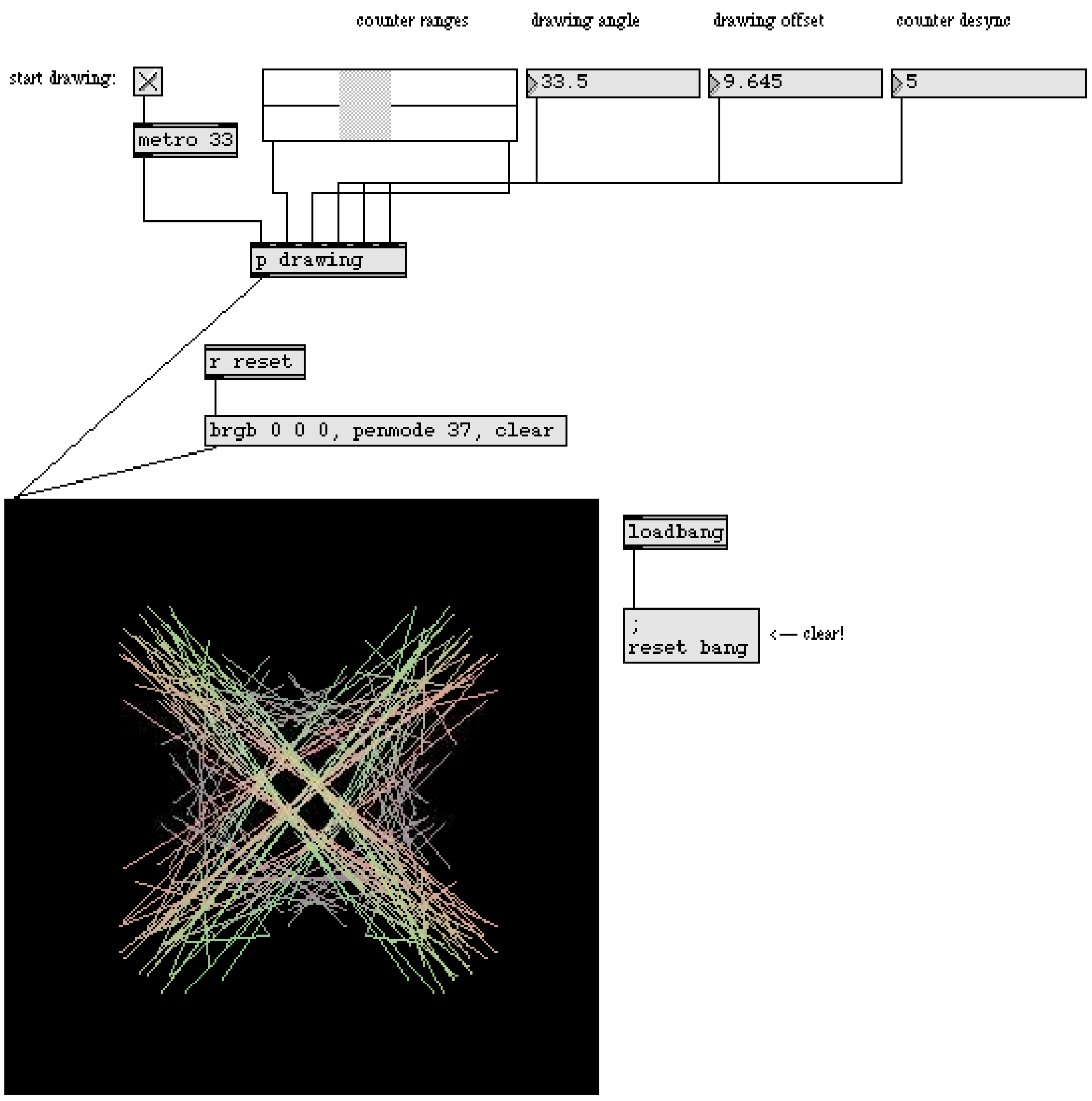
Introduction

In the last tutorial, we learned how to use the Max **pattr** object to help you manage data, either by binding to an object and referencing its data or by storing the data itself. The **pattrhub** object allows you to remotely query and set the data stored in **pattr** objects throughout a patcher hierarchy. In this tutorial, we'll look at two powerful extensions to the **pattr** family: **autopattr**, which simplifies the binding of large groups of objects to the **pattr** system, and **pattrstorage**, which provides an extensive system for storing and recalling 'states' of data stored in multiple **pattr** objects.

Automatic Binding

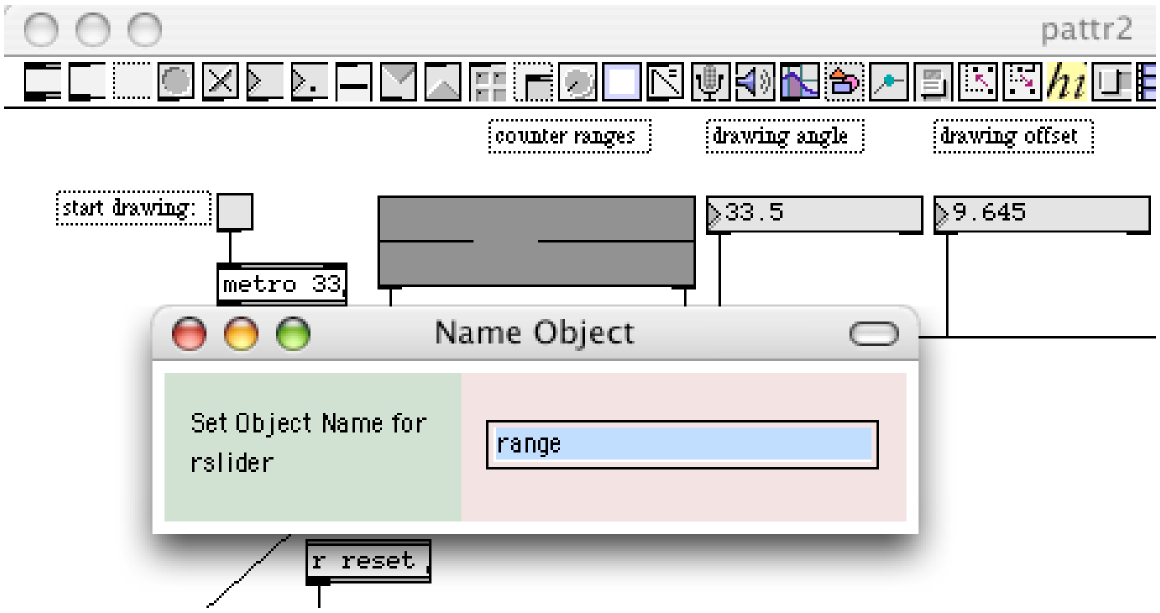
- Open the tutorial patch *53.autopattr.Bindings.pat* and start the patch by clicking the **toggle** box attached to the **metro** object in the upper-left corner of the patch. The patch creates an algorithmic drawing based on the parameters stored in the **rslider** object and **number box** objects attached to the patcher named 'drawing'. The **message** box that sends a bang to the **receive** object named 'reset' allows you to clear the **lcd** display and reset the **counter** objects in the drawing subpatch.

The object called **autopattr** in the lower-right corner has a number of **message** box objects connected to it that contain different example settings for the drawing controls.



The **autopattr** object has much of the same functionality as **pattrhub**, allowing you to send and query values from objects exposed to the **pattr** system within a patch.

- Unlock the patch and select one of the drawing interface controls other than the **toggle** object (i.e. the **rslider** or **number box** objects at the top). Under the Object menu, select **Name….**



Set Object Name for rslider

range

Note that our four 'control' objects in the patch have names ('range', 'angle', 'offset', and 'desynch'). If you create a group of user-interface objects and give them names, an **autopattr** object will automatically expose those objects into the **pattr** system, exactly as if they had **pattr** objects all bound to them individually. In this sense, **autopattr** also functions as a set of **pattr** objects, allowing you to include multiple objects for patcher storage with one object.

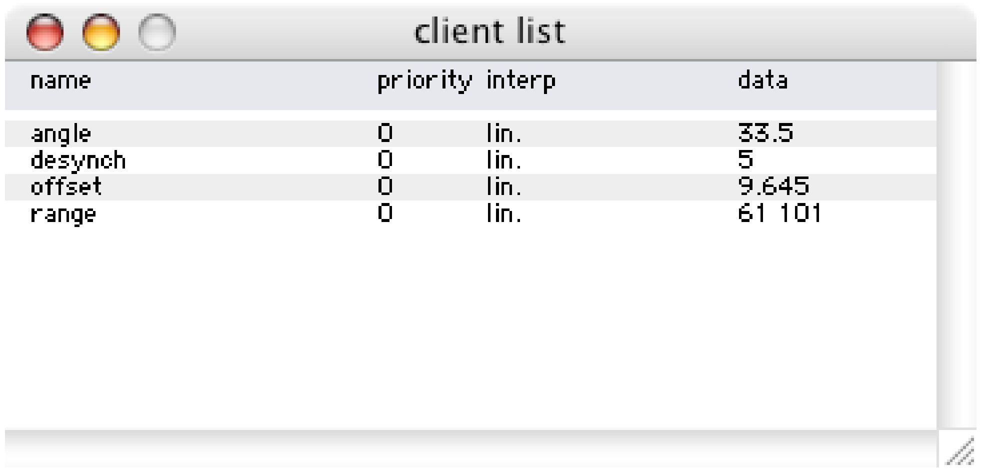
The default behavior of the **autopattr** object is to expose only named user interface objects in a patcher when the object is created. The autoname attribute to **autopattr**, when set to 1, will automatically name all user interface objects in the patcher and expose them to the **pattr** system. Like the Max **preset** object, the **autopattr** object will exclude any objects connected to its second inlet or, if objects are connected to its left inlet, allow you to only specify a few objects explicitly for naming and binding into the **pattr** system.

Because the **autopattr** object combines many of the features of the **pattr** and **pattrhub** objects, it allows you to rapidly include large numbers of Max objects with a single object, while allowing you to query and set their data through the same object.

Tutorial 53

Storing States

- Click on the **message** box labeled 'clientwindow' attached to the **pattrstorage** object. The window that appears gives you a tabulated list of all the **pattr** objects exposed in the patch, as well as some information about them. Change the settings in some of the control objects for the drawing machine and watch the window update to reflect the new settings.



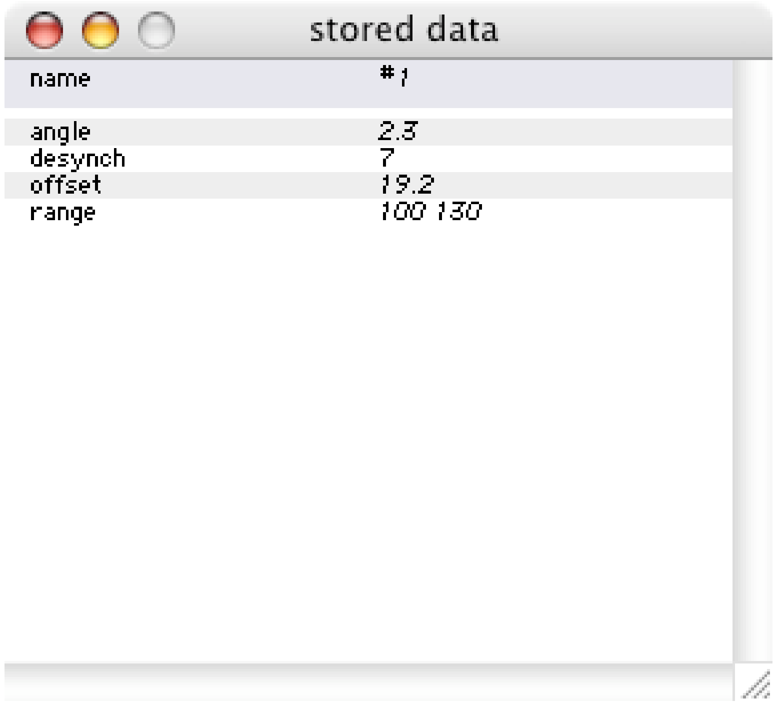
name	priority	interp	data
angle	0	lin.	33.5
desynch	0	lin.	5
offset	0	lin.	9.645
range	0	lin.	61 101

The **pattrstorage** object provides a number of features for high-level parameter management. The client window shows the names of all the objects exposed to **pattr** in the patch (including those controlled by **pattr** objects or **autopattr** objects). It shows their current values in the last column. The middle two columns show the recall priority and interpolation mode for the objects, which can be changed to fine-tune the order in which the objects are recalled to a saved state, and how objects respond to an in-between state (see below).

- Click on the **message** box labeled 'storagewindow' attached to the **pattrstorage**. A window with a list of the control objects should appear. Create a setting you like with the four objects that control the drawing machine. Once you have a setting you're happy with, click the **button** object attached to the **int** object above the **pattrstorage**. This will send the message store 1 to **pattrstorage**.

Sending a store message, followed by an integer value, tells **pattrstorage** to capture the current state of all exposed objects and save them as a preset (or state) in a numbered slot inside the object. The slot will then receive a column in the storage window, showing us the number of the slot as a column header, with all the appropriate values tabulated below it. State 0 is a special slot that allows you to save a state temporarily without having it save in a file (see below).

- Now that you've saved a preset, change one of the pattr-exposed objects. Watch how the storage window responds.



name	#1
angle	*2.3*
desynch	7
offset	*19.2*
range	*100 130*

When you make a change to an object that deviates from the most recently saved or recalled preset slot, the value for the appropriate object becomes italicized in the storage window.

- Create some more preset states by making changes to the control objects, then storing their settings in **pattrstorage** by incrementing the **number box** attached to the int object and clicking the **button**. Note that if you re-use a slot number (e.g. you save a state under slot 2, then save again) the old settings are overwritten.

Note that once you have more than one state saved in the **pattrstorage**, the most recent state is shown in boldface, while older states are set in a normal typeface.

- Click on the **number box** labeled 'recall a preset'. Set it to one of the values you've stored. The user interface objects should recall to their saved values, and the slot column representing that state in the storage window should go to boldface.

Sending an integer to **pattrstorage** will recall the slot corresponding to that integer. Attempting to recall a slot that has never been saved is harmless, and **pattrstorage** will ignore you.

Interpolation

- Click on the floating-point **number box** labeled ‘interpolate between presets’. Type in a value between two adjacent slot you’ve saved (e.g. if you’ve saved a state under 1 and a state under 2, type 1.5). The pattr-exposed objects will go to an interpolated value between those two states. Drag with the mouse in the **number box** so that you effect a smooth transition from one state to another. The controlled objects will smoothly interpolate as well.

Floating point values sent to **pattrstorage** will interpolate values between saved states. Non-adjacent states can be interpolated as well, using the message fade a b n, where a and b are the slot numbers and n is a value from 0.0 (100% a) to 1.0 (100% b). Note that interpolated values will italicize both of the participating slot in the storage window.

Saving and Loading Files

- After you’ve saved a couple of states into preset slots, click on the **message** box labeled ‘write’ attached to the **pattrstorage** object. A file save dialog will appear, asking you to save a file with the suffix *.xml*. Save the file somewhere and open it in any application that can read text files (Max, for example). Look over the file.

The **pattrstorage** object reads and writes collections of preset states as XML (extended markup language) files, a format which uses nested tags as identifiers for different types of data in a file.

An example XML file from our Tutorial patch could look something like this:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<pattrstorage name = "u518000001">
   <slot number = "1">
      <pattr name = "angle" value = “60.” />
      <pattr name = “desynch” value = “7” />
      <pattr name = “offset” value = “0.” />
      <pattr name = “range” value = “50 100" />
   </slot>
   <slot number = “2”>
      <pattr name = “angle” value = “33.5” />
      <pattr name = “desynch” value = “39” />
      <pattr name = “offset” value = “8.8” />
      <pattr name = “range” value = “50 147" />
   </slot>
</pattrstorage>
```

The first relevant tag that encompasses all our data defines the information as belonging the a **pattrstorage** object. The <pattrstorage> tag also tells us the name of the **pattrstorage** argument, which is randomly generated unless we set it explicitly with an argument to the object.

Between the <pattrstorage> tags, we have <slot> tags defining the saved states. Note that slot 0 is not saved in the file. Within each <slot>, we have <pattr> tags that list the exposed objects by name and list their preset value for that state.

- Make a change to the numerical data in one of the slots and save the XML file. Be careful to preserve the quotation marks around the values. Read it back into the patch by clicking the 'read' **message** box and selecting the file. The storage window will update to the new preset value, which you can then recall.

Because XML is a straightforward, human-readable format, it's quite simple to edit preset files saved using **pattrstorage**. We could even use a piece of software (e.g. Max) to automatically generate preset files for our patches once they've been set up to use **pattrstorage**!

Summary

The **autopattr** object functions as a hybrid of the **pattr** and **pattrhub** objects, allowing you to automatically expose multiple user interface objects to the **pattr** system in one easy step. The object allows you to automatically name objects for inclusion (through the autoname attribute), as well as explicitly include or exclude objects through the first two outlets, respectively. The **pattrstorage** object allows you to view the current settings of pattr-exposed objects, as well as save states of settings into numerical slots (through the save message) and recall them (by sending an integer to the object). Furthermore, you can interpolate between preset slots, as well as save and recall your saved states in an easy-to-read XML format.

See Also

autopattr	freeing **pattr** users from the burdens of thought
pattr	patcher-specific named data wrapper
pattrhub	make your **pattr**objects work for you!
pattrstorage	view/modify **pattr** data, store and recall presets
patcher	Create a subpatch within a patch
preset	Store and recall the settings of other objects
lcd	Draw graphics in a Patcher windo

Index

Index

Index

Index

Index

Index

Index