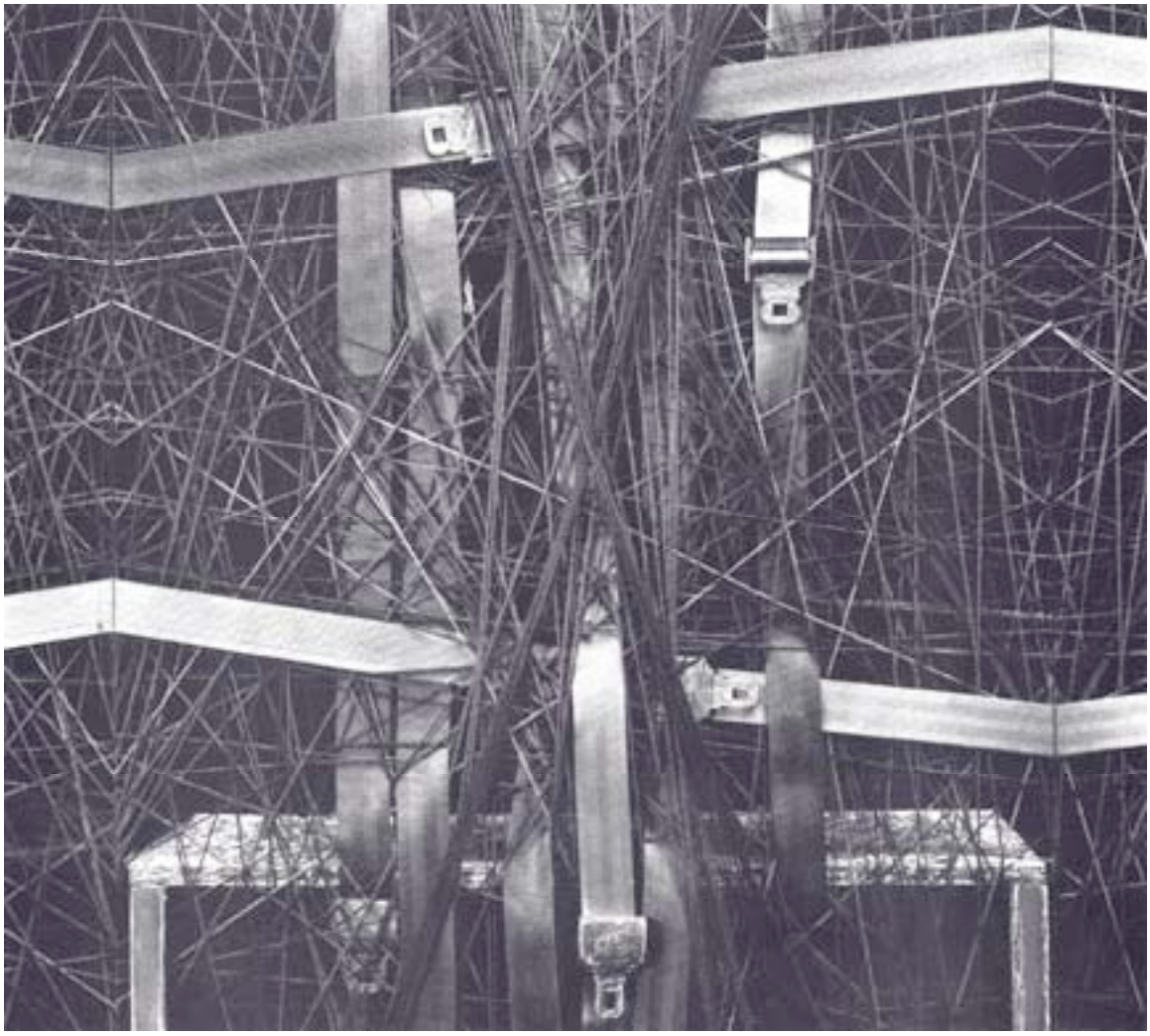
MAX



Max Topics

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: David Zicarelli, Gregory Taylor, Adam Schabtach

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Tutorials and Topics in Max

This manual contains discussions on issues of programming and using Max, including discussions on tips and shortcuts for everyday use, data structures, loops, encapsulation, debugging, graphics, and making standalone applications. They cover material that is beyond the scope of the Max Tutorial, but of general interest to Max users.

Manual Conventions

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

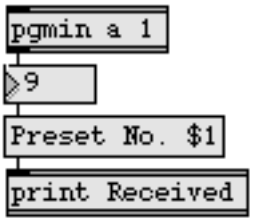
Messages (the arguments that are passed to and from objects) are displayed in plain type, like this.

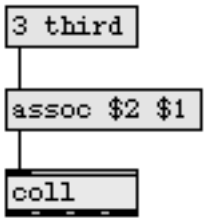
In the **See Also** sections, anything in regular type is a reference to a section of this manual, the Max Tutorial, or the Max Reference manual.

Arguments: $ and #, Changeable Arguments to Objects

$ in a message box

The dollar sign ($) is a special character which can be used in a **message** box to indicate a changeable argument. When the **message** box contains a $ and a number in the range 1-9 (such as $2) as one of its arguments, that argument will be replaced by the corresponding argument in the incoming message before the **message** box sends out its own message.

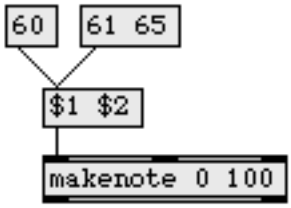


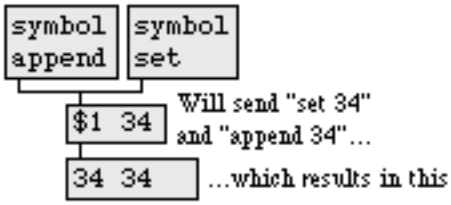


In the left example above, the $1 argument in the **message** box is replaced by the number received in the inlet (in this case 9) before the message is sent out. The message printed in the Max window will read Received: Preset No. 9.

The right example shows that both symbols and numbers can replace changeable arguments. It also shows that changeable arguments can be arranged in any order in the **message** box, making it a powerful tool for rearranging messages. In the example, the message assoc third 3 is sent to the **coll** object.

When a **message** box is triggered without receiving values for all of its changeable arguments (for instance, when it is triggered by a bang), it uses the most recently received values. The initial value of all changeable arguments is 0.





In the left example above, a message of 60 will initially send 60 0 to the **makenote** object. After the 61 65 message has been received, however, the number 65 will be stored in the $2 argument, so a message of 60 will send 60 65 to **makenote**.

A **message** box will not be triggered by a word received in its inlet (except for bang), unless the word is preceded by the word symbol. In such a case, the $1 argument will be replaced

by the word, and not by symbol. In the right example, the $1 argument is replaced by either set or append, and the message set 34 or append 34 is sent to the next **message** box.

To include a special character such as a dollar sign in a message without it having a special meaning, precede the character with a backslash (\).

$ in an object box

A changeable $ argument can also be used in some object boxes, such as the **expr** and **if** objects. In these objects, the $ must be followed immediately by the letter i, f, or s, indicating whether the argument is to be replaced by an int, a float, or a symbol.

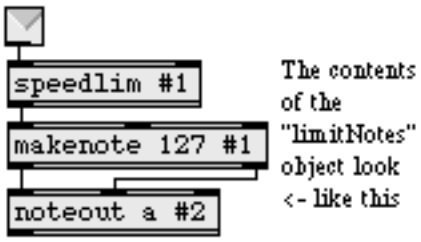
If the message received in the inlet does not match the type of the changeable argument (for example, if an int is received to replace a $f argument), the object will try to convert the input to the proper type. The object cannot convert symbols to numbers, however, so an error message will be printed if a symbol is received to replace a $i or $f argument. Other objects in which a $ argument is appropriate include **sxformat** and **vexpr**.

in object and message boxes

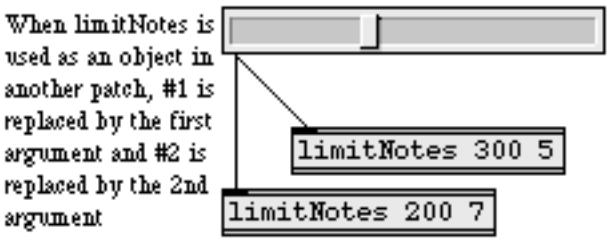
When you are editing a patcher which will be used as a subpatch within another Patcher, message boxes and most object boxes in the subpatch can be given a changeable argument by typing in a pound sign and a number (for example, #1) as an argument. Then, when the subpatch is used inside another Patcher, an argument typed into the object box in the Patcher replaces the # argument inside the subpatch.

In this way, **patcher** objects and your own objects can require typed in arguments to supply them with information, just as many Max objects do. A symbol such as #1 is a changeable argument, and is replaced by whatever number or symbol you type in as the corresponding argument when you use the patch as an object inside another patch. A changeable argument cannot be used to supply the name of an object itself, but can be used as an argument anywhere inside your object.

In the following example, arguments typed into the **limitNotes** object boxes supply values to the objects inside **limitNotes**. When the **hslider** is moved, one **limitNotes** object plays a note every 300 milliseconds on MIDI channel 5, and the other plays a note every 200ms on MIDI channel 7.

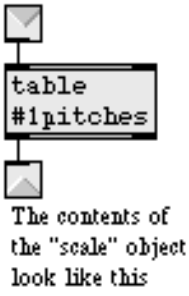


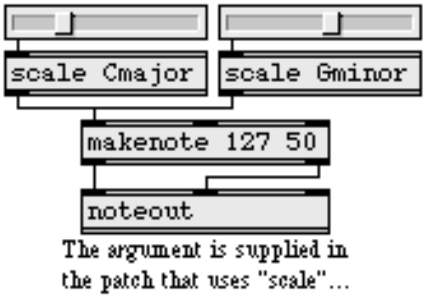
These are Max objects

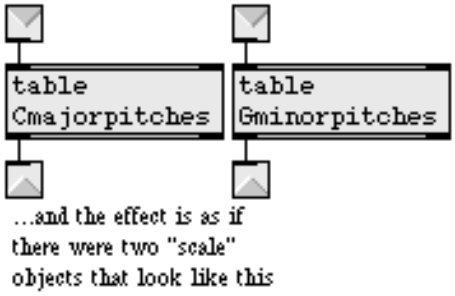


limitNotes is a patch saved as a document

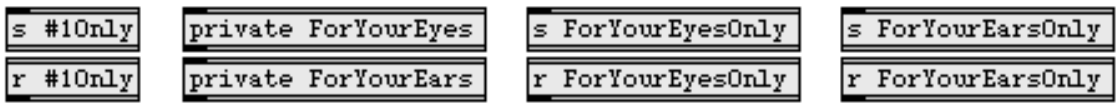
A pound sign and a number can even be part of a symbol argument, providing variations on a name, provided that the changeable argument is the first part of the symbol. In the example below, the #1 part of the changeable argument inside **scale** is replaced by the argument in the patch that uses **scale**. The **scale** objects will each use a different pre-saved **table** file, producing different results.







The same technique can be used to give unique names to **send** and **receive** objects in a subpatch, making the exchange of messages between them private (local to that one instance of the subpatch).

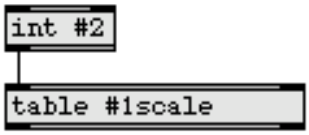


If these objects exist in a patch named private, *and the patch is used for two subpatches like this,* *the objects appear with this name in one patch,* *and with a unique name in the other.*

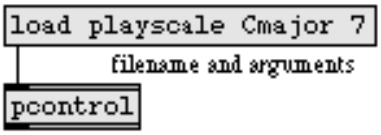
Arguments

$ and #, Changeable Arguments to Objects

When opening a patcher file automatically with the **load** message to a **pcontrol** object, changeable **#** arguments inside the patch being loaded can be replaced by values that are provided as additional arguments in the **load** message, as in the example below.



If these objects exist in a patch



and this message is sent to a pcontrol object,

the patch will open with objects looking like this.

#0 has a special meaning. It can be put at the beginning of a symbol argument, transforming that argument into an identifier unique to each patcher (and its subpatchers) when the patcher is loaded. This allows you to open several copies of a patcher containing objects such as **send** and **receive** without having the copies interfere with each other.

See Also

expr	Evaluate a mathematical expression
message	Send any message
pcontrol	Open and close subwindows within a patcher
Punctuation	Special characters in objects and messages

Collectives: Grouping Files into a Single Project

What is a Collective?

When you open a Max patcher, you may need to open a number of other Max files—even though it seems as if you are opening only one file:

- The patcher might require certain external objects.
- The patcher may contain subpatches (other Max documents used as objects within a patcher).
- The patcher may load other files used by Max objects. This category would include MIDI files, **coll** files, **env** script files, **funbuff** files, **mtr** files, **preset** files, **seq** files, **table** files, **timeline** files, action patches, PICS files, PICT files, QuickTime movies, and so on.

A program you write in Max may actually be divided up among a potentially large number of different files, and the absence of any one of those files may prevent your program from functioning properly. To avoid this problem, Max allows you to gather most of the files necessary for a program that you write into a single group, called a *collective*. Once you have done this, you can be assured that all the necessary subpatches and data are available to your patch. You can also give your collective to someone else to use, without worrying whether you've included all the necessary files. If the person you give your collective to doesn't own Max, you can give (but not sell!) them the MaxMSP Runtime application along with your collective. This will allow them to run (but not edit) your program.

In addition, you can combine a collective with a copy of MaxMSP Runtime to create a *standalone application*, which requires neither Max nor MaxMSP Runtime in order to run.

You can also make an audio plug-in (VST, RTAS, and AU) with Max/MSP. For details and examples of building plug-ins with Max/MSP see the Pluggo Developer Materials that are installed with Pluggo, which can be downloaded from www.cycling74.com.

Making Your Own Program

A program written in Max most commonly consists of one main patch—a Max document— which contains other subpatches as objects inside it. Alternatively, you might choose to design the program so that the user keeps two or more different patches open at once for doing different tasks. In either case, at least one patcher window has to

be opened by the user, and this is referred to as a *top-level patch*. A collective can have more than one top-level patch, and each one will be opened when the collective is opened. Other patches used as objects within a top-level patch are called *subpatches*.

To make your own program into a collective, you'll need to determine which patch (or patches) will be the top level patch for the program. When you build a collective using that patch, Max includes in the collective any external objects or subpatches that the top-level patch requires to operate (unless you add the "excludeexternals true" command to your collective script).

You may also need to include some other data files explicitly (data files used by objects such as **coll**, **seq**, etc.) to complete the collective. You will then have a complete working program that originally consisted of many diverse files, saved in a single file.

Once you have saved a collective, you can open it as you would any other Max document by choosing the Open... command from the File menu or by double-clicking on the collective in the Finder on Macintosh or the Windows Browser on Windows.

You cannot load it into another patch as a subpatch by typing its name into an object box (nor can you load it into a **bpatcher**). If you make changes to any patch that is being used as a subpatch in a collective, those changes will not be updated in the collective—the subpatch in the collective remains just as it was at the moment you saved the collective.)

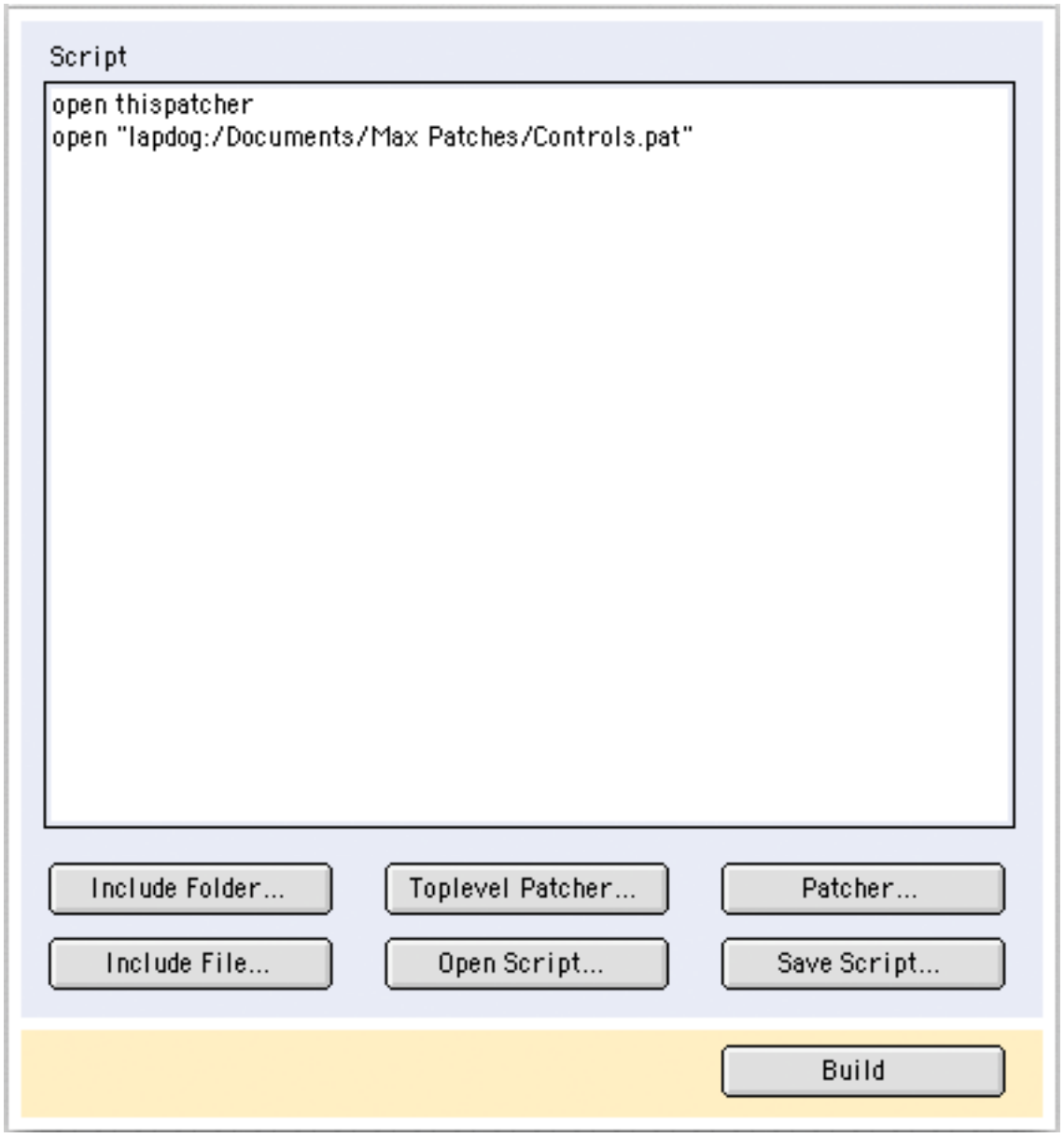
Steps for Building a Collective

1. With your top-level patch in the foreground, choose Build Collective / Application / Plug-in … from the File menu.

   You will be presented with a script window, in which you create a list of things Max must do to create the collective. Max has already made the first entry in its script—open thispatcher— instructing itself to load in your top-level patch. Any external objects required by your patch, any subpatches used as objects in your top-level patch (or used in a **bpatcher**), and any nested subpatches (sub-subpatches used in subpatches of the top-level patch) will all be included automatically in the collective. In addition, certain objects, such as **js**, **fpic**, and **hi**, copy files they use into the collective for you. For the **js** object, the script text file is copied, but not files the script uses.

   If you want your program to have more than one top-level patch, you can add other patches to the script by clicking on the *Toplevel Patcher...* button and choosing another patcher from the file dialog box. Max writes a new line into the script, indicating that it will also open that newly selected file.

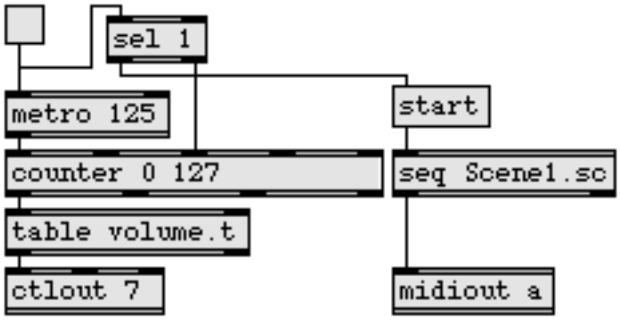
In the following example, a patch named *Arpeggiator* is being saved as the top- level patch in a collective, and a second top-level patch named *Controls* has just been added.



When the collective is opened by a user, top-level patchers will be opened in the order in which they are listed in this window. If you want to change the order in which they will be opened, you can edit the script.

2. Besides the externals and subpatches which are included automatically, there may be other files used by your top-level patch(es). Add any other necessary files to your collective by clicking on the *Include File...* button (or the *Patcher...* button if the file is a Max patch) and choosing the appropriate file from the ensuing dialog box.

   There are three reasons why you may need to include files explicitly in this way. First of all, it's frequently the case that some object in a patch loads in data from a separate file. Consider the following example.



   When this patch is loaded it looks for the **table** file *volume.t*, and the **seq** file *Scene1.sc*. These files will not be included automatically because they are neither patchers nor external objects, so you must list them in the script yourself. Some objects handle inclusion of their files for you, as mentioned above.

   > Currently, QuickTime movies used by the **movie** and **imovie** objects (as well as Jitter objects) cannot be included in collectives.

   Second, it is possible that the program may load some additional patch(es) dynamically (with a load message to the **pcontrol** object, for example). Because such a patch does not appear as an object box in the top-level patcher, it is not included automatically, and you must include it yourself. In the following example, the file *panic* is not a subpatch of the top-level patch, but it could be needed, nevertheless, and should be explicitly included in the collective. We added *panic* using the *Patcher…* button, so that any external objects used in the patch would be included in the collective. Likewise, the **timeline** file *MultiTrack.ti* should be included. The action patches used by the **timeline** object will be included automatically.

   If you have an entire folder of data files you want to include, you can include all the files by clicking the *Include Folder...* button and selecting the folder from the ensuing

dialog box. Note that this will only include files in the folder itself; folders inside the folder you select will not be included.

3. Once you have created a collective, you cannot easily make changes to it. So, before you actually click on the *Build* button to construct your collective, you may want to save your script as a separate Text file, by clicking on the *Save Script...* button. That way, if you later make changes to some of the patches or files in your collective, and therefore need to rebuild or modify the collective, you can simply open the original script by clicking on the *Open Script...* button, and you'll have a head start toward rebuilding your collective.

4. Once you have added all the top-level patches you want (they appear with an open instruction in the script) and have included all necessary files and/or folders (they appear as include and folder instructions in the script), your collective is complete. Click on the *Build* button and give your collective a unique name.

Adding Non-Max Files to a Collective

When you create a collective, you can include files of any type (which may have been created with applications other than Max) by clicking the *Include File...* button in the Collective Script dialog. This allows you to add, for example, graphics files to your collective for use with the **fpic** object.

Note that you can leave externals out of your collective by adding the command "excludeexternals true" at the top of your collective script. This must come before the "open thispatcher" command. You may want to do this if you want to manage installing the external objects separately from the collective. You can use this in conjunction with the include command to only include a few specific external objects in your collective. Note that when building a VST plug-in that externals are always excluded unless explicitly added via an include command.

Testing a Collective

A collective can function as a complete program: one or more (top-level) Max patches combined with all the other files they need to function correctly. Before you give your collective to someone else to use, however, you should test it to be sure that it's really complete, and that you haven't forgotten to include any essential files. The best way to do that is to open the collective by choosing Open... from the File menu or by double-clicking on the collective in the Finder on Macintosh or the Windows Browser on Windows.

Collective Formats

On Macintosh, a pop-up menu when saving the collective allows you to choose the Max collective format or the Old Format Collective. The only reason to use the Old Format Collective is if you are building a plug-in to use with Pluggo, or wish to produce something that can be used by older versions of Max. But note that for compatibility with older versions, your collective cannot contain any external objects that are specific to Max/MSP 4.6. These can be stripped out manually using a Macintosh resource editor such as Resourcerer—simply delete all `mAxL` resources.

The "new" Max collective format is cross-platform; however, included external objects are not cross-platform, so if your collective uses any non-standard external you will have to supply these as separate files for each platform. If you are going be giving Macintosh-created collectives to Windows XP users, you are strongly encouraged to use the *.mxf* file extension. Otherwise, it won't be possible for Windows XP users to open the files.

Building a Standalone Application

When you click on the **Build** button in the "Collective Editor" dialog, you are presented with a standard Save As dialog allowing you to name your collective and save it to disk. In the lower part of this dialog you are presented with a Format menu. By default this menu is set to save a Max Collective file, but if you want to save your patch as a standalone application, all you need to do is select **Application** from this menu. It's as easy as that—Max will automatically combine your collective with the MaxMSP Runtime application and save the result as a single file (or folder on Windows) which appears and functions as a standalone application, requiring neither Max nor MaxMSP Runtime. If you have multiple versions of MaxMSP Runtime in your Max folder, Max will use the application that has the word "Runtime" in its name, and the most recent creation date. Building an audio plug-in is accomplished in a very similar fashion – just choose Plug-in from the drop down menu.

Windows Standalone Format

A standalone application on Windows is actually a folder containing MaxMSP Runtime executable plus a *.mxf* collective file containing your patches and files. In addition, the standalone folder contains a support folder with files necessary to run your application.

The following list shows the arrangement of files and folders.

YourApplication [folder]
    YourApplication.exe [modified MaxRT.exe – launch this to launch your app]
    YourApplication.mxf [new format collective containing your patches]
    msvcr70.dll
        [Microsoft C Runtime Library used by MaxRT.exe and externals]
        support [folder]
            ad [folder containing MSP audio driver objects]
            mididrivers [folder containing Max midi driver objects]
            MaxAPI.dll [Max API for external objects]
            MaxAudio.dll [MSP library]
            MaxQuicktime.dll [Max QT interface]
            YourApplication.rsr [Mac style Resources for your application]
            asintppc.dll [Support DLL needed for Max]
            asiport.rsr [Support resources needed for Max]
            asifont.map [Support file needed for Max]

Macintosh Standalone Format

On Macintosh, Max builds universal binary standalone applications. The standalone is an application package, which is also a folder that looks like a file in the Finder. Double-clicking on the icon launches the application. You can peek into the package by control-clicking on it and choosing **Show Package Contents** from the contextual menu. As with the Windows standalone, inside the Contents/MacOS folder you will find a *.mxf* collective, a runtime application, and a support folder.

YourApplication.app [note: the .app is not shown in the Finder]
    Contents [folder]
        Frameworks [folder] – needed for external object support
        Info.plist [copied from your .plist if included in a collective]
        YourApplication.mxf [collective containing your patches]
        MacOS [folder]
            YourApplication [actually Max/MSP Runtime]
        support [folder] – audio and MIDI support filess
        Resources [folder]
            [custom icon file goes here]

Developers may wish to note is the behavior of the message sendapppath to the Max object. This message now reports the file path of the application bundle rather than the executable file inside the MacOS folder. Since this is typically what developers wanted in the first place, we hope this change does not pose a problem.

When you use the standard save file dialog box to name your plug-in, collective or standalone, the filename extension automatically changes when you choose a different output type. Mach-O VST plug-ins must end in .vst. Applications must end in .app. And it is always a good idea to name collectives to end in .mxf, particularly if you want to open them on Windows.

> Do not rename the .mxf file inside your standalone, otherwise it will not load when the runtime executable is launched.

Customizing Your Standalone

You may want to customize some of the features of your standalone application, such as the Overdrive setting, the application icon, or whether or not users will be allowed to close the top level patcher(s). There are two mechanisms for doing this: first, there are additional collective script messages that apply only to standalone applications, and second, the standalone object can be added to your main top-level patcher to configure certain options.

appsplash — The word appsplash, followed by a full pathname of a PICT (Macintosh) or BMP (Windows), specifies the image to be used as a splash screen in place of the Max about box.

appicon — The word appicon, followed by a full pathname of a platform-specific icon file (Windows icon resource or Mac OS X icns resource), specifies the icon to use for the application.

If you are not using Javascript or Jitter in your application, you can remove MaxJSRef.framework and/or JitterAPI.framework from the Frameworks folder inside the application package to reduce your app's download size. Do not remove the other framework items found in the Frameworks folder.

If you do not want the user to see the dialog asking about installation of support for external objects from older versions of Max, remove the MaxMSPCFMSupport.pkg file from your standalone's Resources folder.

For your standalone's icons and other Finder-related customization, standalone applications use the Info.plist file found in the Contents folder of the standalone, plus

.icns files in the Resources folder. Previous versions used BNDL, FREF, and icn# resources. To learn more about the Info.plist file and creating and using .icns files, this Apple documentation may be helpful:

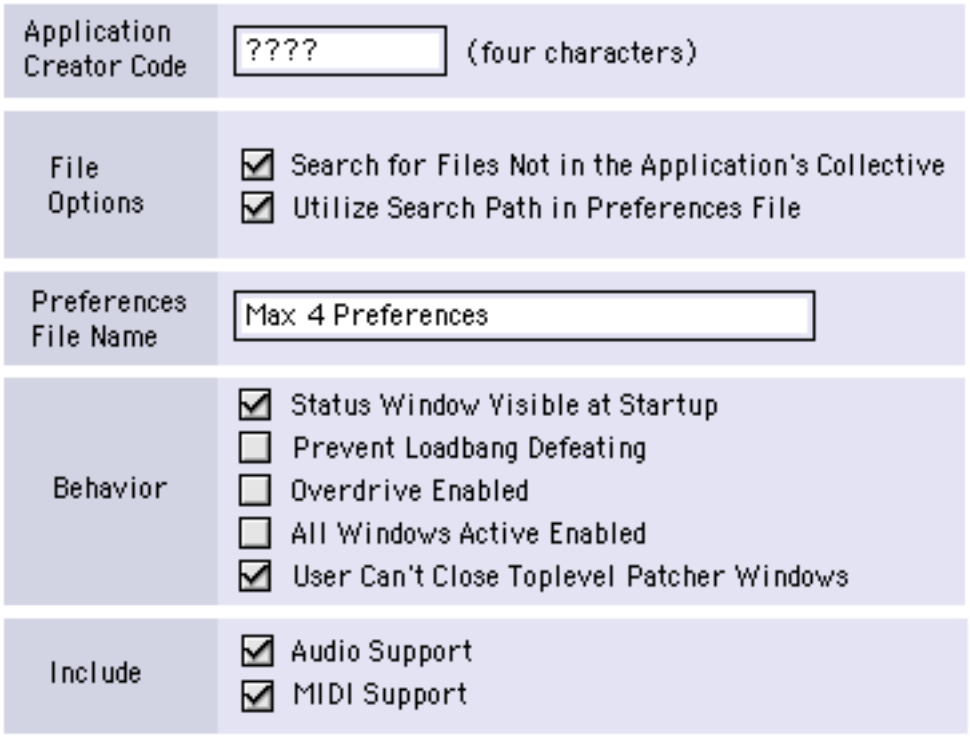
http://developer.apple.com/documentation/Carbon/Conceptual/DesktopIcons/ch13.html

http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/ConfigApplications.html

The standalone Object Inspector

To control other settings, add the **standalone** object to your main top-level patch, and editing its parameters with its Inspector. Since the **standalone** object and its settings are stored with your patch, you do not need to specify the settings each time you save a new version of your standalone application.

Here is an overview of the various settings available in the **standalone** object's inspector:



The *Application Creator Code* is a Macintosh only setting. It specifies a four character ID that the Finder uses to distinguish your application from others (including Max and MaxMSP Runtime). The default creator, ????, is assigned for generic files and applications. You can change this to any combination of four characters you like, but if you choose one already in use by another application, your application will run when you double-click on a document for the application whose creator you used. For instance, if you used max2 for a creator, double-clicking on a Max document would launch your application instead of Max.

If you want to guarantee your character combination is unique, you will want to register it with Apple at the following URL:

http://developer.apple.com/dev/cftype/

The File Options section lets you customize some aspects of how your standalone application deals with files and the file system.

If some of the supporting files used by Max/MSP objects in your patch will not be included in the collective itself, check the *Search for Files Not in the Application's*

Collective option. (It is checked by default.) Unchecking this option can be useful for ensuring that you have included all necessary files in the collective that you are making into a standalone application. If you create your application with this option turned off, your application will not look outside the collective for any files it cannot find, such as missing sequences or **coll** files that your application attempts to load. So, you can make your application with *Search for Files Not in the Application's Collective* unchecked, and then run it to see if it works properly. If your application is unable to find a file that it needs, you will get an error message to that effect, and you will know that you have to rebuild your standalone application.

In some cases, however, you may want your application to look for a file outside of the collective. For example, you may want it to look for a MIDI file that can be supplied by the user of your application. In that case, you will naturally want the *Search for Files Not in the Application's Collective* option to be on. Please also note that this feature restricts itself to looking in folders nested only three levels deep.

When your application searches for files outside the collective, you can control where it looks with the *Utilize Search Path in Preferences File* option. If this option is on (which it is by default), your application will use the search path settings stored in the Max 4 Preferences file instead of using the default search path.

You can instruct your application to use its own Preferences file instead of the default Max 4 Preferences by supplying a preferences file name in this field. If the *Utilize Search Path in Preferences File* is checked and you type in a name other than the default Max 4 Preferences, your application will make its own unique preferences file (in ~/Library/Preferences, where ~ represents your home directory) the first time it is run. From then on, your application will use that preferences file to recall the settings for options such as Overdrive and All Windows Active.

The Options section of the inspector lets you change the various user-related options for your standalone application.

If the *Status Window Visible at Startup* option is unchecked, the Status window (the same as the Max window in the regular Max application) will not be visible when the application is opened. Unchecking this option can help give your application a particular (perhaps less obviously Max- like) look. By default this option is enabled.

Normally, one can stop all **loadbang** objects from sending out their `bang` messages by holding down the Shift and Command keys on Macintosh or Shift and Control keys on Windows while the patch (or collective, or standalone) is loading. You can disable that **loadbang**-defeating capability in a standalone application by checking the *Prevent loadbang Defeating with Cmd-Shift* option. (This option is turned off by default.)

The *Overdrive Enabled* and *All Windows Active Enabled* options allow you to preset these menu options to configure your application's initial behavior. They are both off by default.

If you check the *User Can't Close Toplevel Patcher Windows* option, top-level patchers will have no close box in their title bar, and the Close command in the File menu will be disabled whenever a top-level patcher is the foreground window in your application. Since closing the top-level patcher in most cases renders the application useless, this option is checked by default.

The final two options allow you to include all the files necessary to see the MIDI Setup and DSP Status windows, as well as all of the audio and MIDI driver objects. This option does not include any non-Max files that might be necessary for MIDI and audio—for example, checking Audio Support does not include the MSP ReWire driver. This driver does not belong in the standalone folder; if you want to use it, you'll have to instruct users of your application to copy it to the correct location, or build you own installer that does this.

Custom Icons for Mac OS X

In order to use a custom icon for your standalone application, follow these steps:

1. "Open" your standalone by control-clicking on it in the Finder and choosing Show Package Contents from the pop-up menu that appears.

2. Place icon files created with Apple's Icon Composer tool or another application that creates these files in the standalone's Resources folder.

3. Edit the Info.plist file in the standalone's Contents folder using Apple's Property List Editor or a text editor (where you will edit the XML directly). Change the entry for the CFBundleIconFile from Max.icns to the name of the icon file you added.

In some cases, you may need to log out and log back in again before the standalone's icon will show up.

Custom Icons and Splash Screens for Windows

In order to create a custom icon for your standalone application, follow these steps:

1. Create your ICON using the Windows *.ico* format.

2. Using the Collective Editor add a line to your Collective Script with the following syntax: appicon filename

   Hint: you can use the "Include File" button to choose the .ICO file and then change the "include" command to "appicon".

In order to create a custom splash screen for your standalone application, follow these steps:

1. Create your splash screen using the Windows ".bmp" format.

2. Using the Collective Editor add a line to your Collective Script with the following syntax:

   ```
   appsplash filename
   ```

   Hint: you can use the "Include File" button to choose the .ICO file and then change the "include" command to "appsplash".

The Search Path in Standalone Applications

It may be important for developers of standalones to know the order in which Max searches for files; it is slightly different than the search order in the editing version of Max/MSP. This information is relatively advanced and will probably not be of much interest to users who are not developing standalone applications.

On Mac OS X the *Utilize Search Path* option, when checked, does the following for the search path of a standalone:

1. Adds all of the folders inside of the folder containing the standalone application (i.e., the Contents folder and all of its subfolders)..

2. Adds the Cycling '74 folder. A Cycling '74 folder at the same location as the application is used if it exists. If it does not, the location */Library/Application Support/Cycling '74* is used if it exists.

On Windows XP the *Utilize Search Path* option, when checked, does the following for the search path of a standalone:

1. Adds all of the folders inside of the application folder (i.e. the folder containing YourApplication.exe).

2. Adds the Cycling '74 folder. A Cycling '74 folder at the same location as the application is used if it exists. If it does not, the location *c:\Program Files\Common Files\Cycling '74* is used if it exists.

When *Utilize Search Path* is not checked, the only folder(s) added to the search path are the support folder inside the standalone application's folder, and any of its subfolders.

The order in which folders will be searched is as follows:

1. The collective file (and any other open collective files)

2. The support folder

3. The folder containing the application and its subfolders (optional, if *Utilize Search Path* is checked)

4. The Cycling '74 folder (optional, if *Utilize Search Path* is checked)

The *Search for Files Not in the Application's Collective* option is different from the *Utilize Search Path* option. It prevents Max from looking for any files that are not in open collectives, including the support folder, which means that you cannot uncheck *Search for Missing Files* and have either audio or MIDI support. This means that *Search for Missing Files* is now of limited usefulness for standalones.

The former role of *Search for Missing Files* was for testing the collective to make sure you were including all of the files you need. With the advent of the support folder and its ability to contain audio and MIDI files, this testing role now falls mainly to the *Utilize Search Path* option. *Utilize Search Path* specifically allows you to check whether any files in the Cycling '74 folder are needed by your standalone application: if, after turning off *Utilize Search Path*, you see errors indicating "no such object" or "can't find files" in the Max window, you know you aren't properly including all of the supporting files you need.

A tip that may help sort out path problems: Put ; max paths in a **message** box in your patch so you can click on it when the standalone is running. The paths message prints out the file paths currently in use.

See Also

Encapsulation	How much should a patch do?
standalone	Configure Parameters for a Standalone Application

Storing Data

Max has objects specifically designed for storing and recalling information, ranging from simple objects that store a single number to more powerful objects for storing any combination of messages.

The simple **int** and **float** objects store a number and then output it in response to a bang. They are comparable to a variable in traditional programming languages. The **value** object allows a single value to be changed or recalled from different Patcher windows (functioning like a global variable). The **accum** object stores a single number which can be added to or multiplied.

A data structure stores a group of information together in a consistent format, so that a particular item can be retrieved using the address (location) of the item.

Arrays

The **table** object is an array of numbers, where each number stored in the table has a unique index number—its address. When an address is received in the left inlet, the value stored at that address is sent out the left outlet. Storing numbers in an easily accessible way is the main utility of such an array, but the **table** object has many powerful features for modifying and using the numbers it stores.

The values in a **table** are displayed graphically in the table editing window, showing the addresses on the x axis of a graph, and the values on the y axis. You can change the values displayed in the table window by drawing in the graph with drawing tools, or by cutting and pasting regions of the graph.

Other messages sent to a **table** can store new values, change its size, report the sum of all its values, step forward or backward through different addresses, report the address of a specific value, and provide statistical information about its values.

The **funbuff** object stores addresses and values, but unlike a **table**, the addresses can be any number, and gaps can exist between addresses. If **funbuff** receives in its inlet an address that does not actually exist (a number that falls in a gap between existing addresses), it finds the next smallest address, and outputs the value at that address.

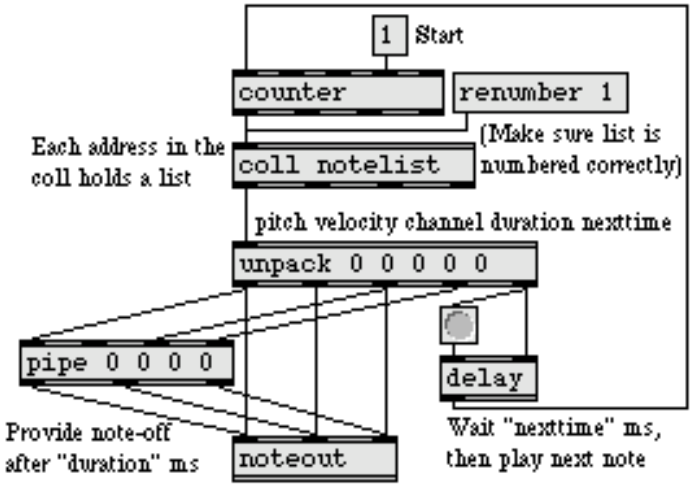
The **bag** object stores a collection of numbers without any addresses. Numbers can be added to and deleted from the **bag**, and a bang in its inlet sends out all of the currently stored numbers.

Complex Data Structures

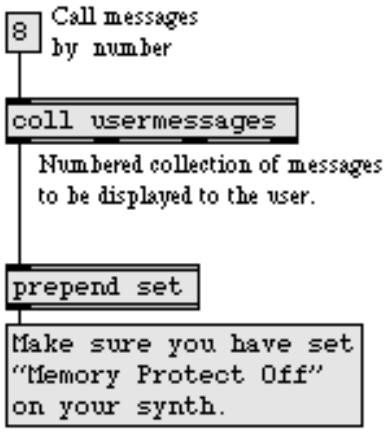
The **preset** object is also a kind of array, but each address in its array contains the settings of other user interface objects in a Patcher window. When an address number is received in the **preset** object's inlet (or when you click on one of the **preset** object's buttons), the settings of those objects are changed to the values stored in the **preset**. In this way, every user interface object in the same window as the **preset** object has its settings stored and recalled. Alternatively, you can connect the outlet of a **preset** to some of the window's user interface objects, making them the only ones affected by that **preset**.

The **coll** object (short for collection) stores numbers, symbols, and lists. A single address in **coll** can be either a number or a symbol. You can also modify stored data in a **coll** with messages such as sub, which changes a single item in a stored location, or merge, which appends additional data to a location. You can also access an individual item in a list stored in a coll with the nth message.

A **coll** object is useful for recording and playing back a "score" that has lists of times, pitches, and durations. Or you could use a **coll** to store a collection of text messages to be shown to the user when certain numbers or symbols are received.



A method of using coll to play a list of notes

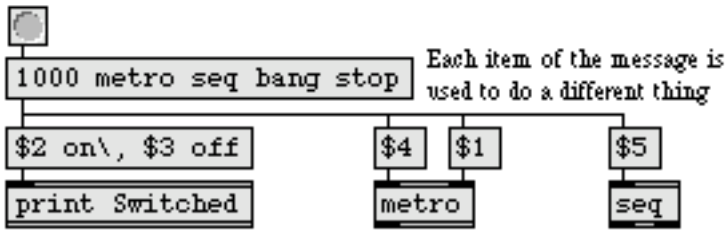


Storing text messages in coll

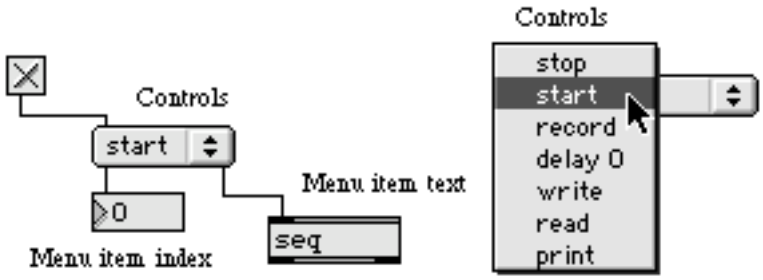
You can edit the contents of a **coll** in a standard Max text editor window by double-clicking on its object box when the Patcher window is locked. The standard Max text

editor window will open. The text format used is discussed in the description of the **coll** object.

The **message** box can be considered a kind of data structure, since it can hold up to 256 different items as arguments. The contents of a **message** can be modified using set and append messages, and **message** boxes can include changeable arguments which are replaced by the arguments of messages it receives in its inlet. Individual items in a **message** box can be accessed by sending its contents to another **message** box with changeable arguments, as shown in the example below.



The user interface object **menu** is essentially an array of symbols. When the number of a menu item is received in the inlet, the item text is displayed and can also sent out the right outlet if desired. Item text is changed with a setitem message. When you choose a menu item with the mouse, you are specifying a symbol, causing the symbol's address to be sent out the left outlet.



Items can be accessed by index number or with the mouse

See Also

coll	Store and edit a collection of different messages
funbuff	Store x,y pairs of numbers together
menu	Pop-up menu, to display and send commands
message	Send any message

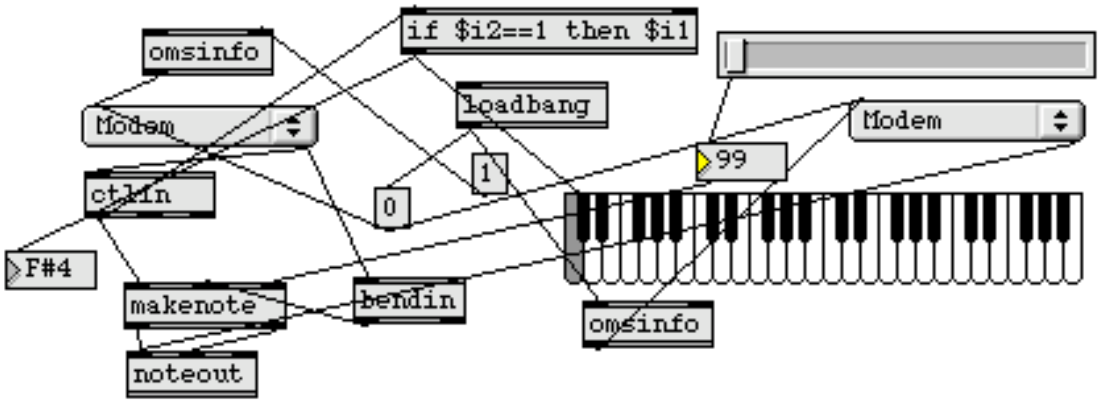
table	Store and graphically edit an array of numbers
Tables	Graphic editing window for creating **table** files

Catching Your Own Bugs

You might occasionally make mistakes when writing a program in Max, and you will then have to figure out why your patch is not working as you intended. In some instances a bug might come from an error in the conceptual design of your program; that is, you might simply be mistaken about what you want the computer to do. Other bugs might be errors of syntax specific to Max such as a misunderstanding of how an object works, a mistake in predicting what messages an object will receive and send, or a mistaken analysis of the order in which messages are being sent. Max does its best to prevent you from making such syntactical errors and provides various means of analyzing and debugging your programs. In this chapter we offer some advice (and a few tools) for preventing or eradicating bugs in your Max patches.

Planning Your Program

One of the best aspects of Max is the fact that you can improvise a program, patching objects together and trying things out, without a clear idea of what you want the results to be. While this is a perfectly valid method of working and can result in some interesting new ideas, it also often leads to the infamous Max spaghetti patch.

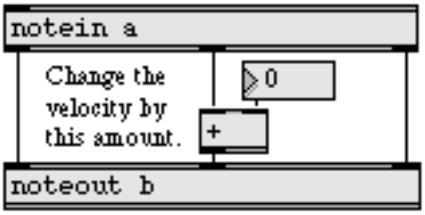


Patch cord spaghetti is often indicative of a lack of planning

This patch works just about as well as a neatly organized patch, but it's certainly more difficult to analyze what's going on or find bugs in such a patch. If you want to ensure that your patch works correctly, it's best to plan it out conceptually before you begin to implement it in Max.

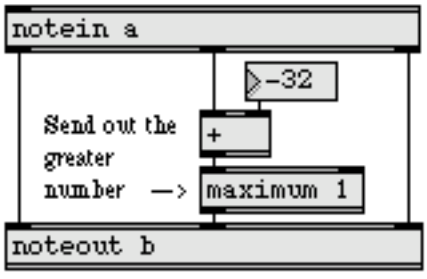
Debugging

Even with careful planning, you may think you know exactly what you want your program to do, begin to write a patch in Max, and then discover that the problem was more complex than you at first thought. For example, you might discover that your plan is appropriate for some cases but not for others. The following example is a (problematic) patch for modifying the velocity of incoming MIDI notes, and sending them back out on a different port. Superficially, it may seem like a reasonable patch, but it will malfunction in many instances. Analyze its problems and see if you can think of good solutions.

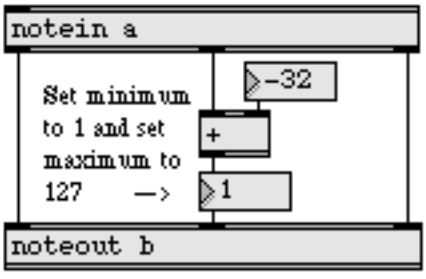


This patch contains bugs

When the change is 0, of course, there is no problem. However, there are three ways this patch can malfunction. The first problem is not serious, because the **noteout** object automatically limits velocity values in its middle inlet to keep them in the valid range from 0 to 127. The second problem is easily solved by limiting the values coming out of the **+** object to be always greater than 0, with a **maximum** 1 object, for example. In fact, you can limit both the minimum and maximum values by passing through a **number box**, and setting its minimum and maximum values (by selecting it and choosing the **Get Info...** command from the Object menu). This has the added advantage of showing you what velocities you're actually sending out.

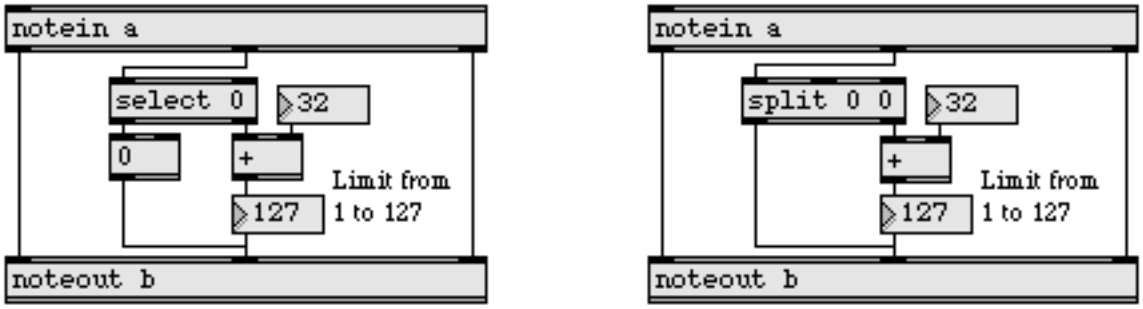


This fixes bug No. 2



This fixes bugs Nos. 1 and 2

The third problem arises because we neglected to consider a velocity of 0 as a special case, which needs to be treated completely differently from all other velocity values. We actually want to leave velocity values of 0 unchanged. The following example shows a couple of possible ways to do that, by sending only the non-zero velocities to the **+** object.



Two possible correct versions of the program

The bugs we saw here did not have anything to do with misunderstanding how Max works; they had to do with mistakenly formulating the task at hand. Max can't really protect you from making that sort of error. It just dutifully performs what you ask it to do. The best way to protect against such bugs is just to plan your program carefully, try to account for as many eventualities as possible, then constantly test the correctness of your plan as you implement it in Max.

Test As You Go

It is infinitely easier to debug a small patch than it is to debug a big, complicated one. It is also much easier to debug a large, complicated patch when you know for sure that certain parts of it work correctly.

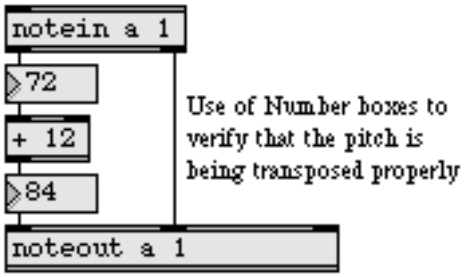
At every single stage in the development of a patch, test everything as you go along. Try sending extreme and unusual messages to your patch, as well as normal, expected ones, to make sure that your patch doesn't malfunction in situations you haven't considered. Once you are sure that a portion of your program works properly, you may want to encapsulate that portion by saving it in a separate file, and using it as a subpatch in a larger patch.

Viewing Messages

There are several good ways to see exactly what messages are passing through the patch cords of your program, so that you can be sure it's doing what you want. The best way to

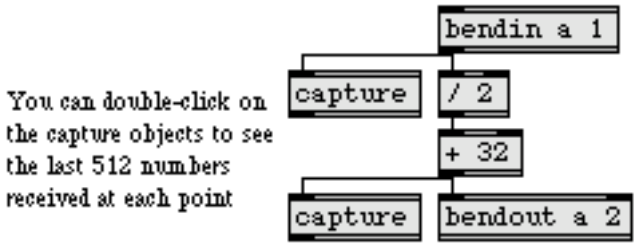
view messages is to include extra objects in your patch temporarily, which "intercept" the messages as they are sent.

For viewing numbers, the **number box** can be used as a kind of "wiretap" in any patch cord. Numbers will pass through the **number box** unchanged, but they will also be displayed as they go through.



The **number box** has several drawbacks as a debugging tool. It can only show a single int or float value, not a list of different values. Numbers may pass through it too fast for your eye to follow them. If the same number passes through several times in a row, you won't see any change in the display. And finally, there is no way to see previous numbers once a new number is displayed.

The **capture** object solves all of these problems by handling both numbers and lists of numbers, and by storing an arbitrary number of values at a time. Hook up a **capture** object off to the side at the point where you want to look at some numbers, then double-click on its object box to open a text editing window which displays the numbers that have recently been received. The default number of values **capture** holds is 512, but this size can be adjusted with a typed-in argument.



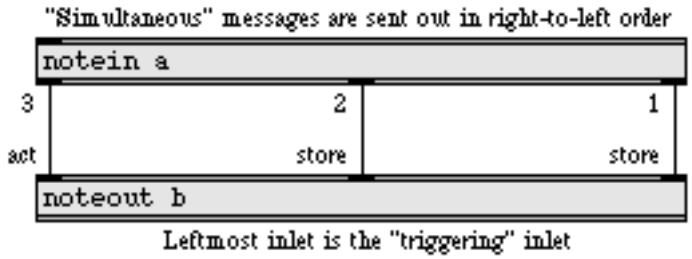
Another potential advantage of **capture** is that you can copy numbers from its editing window and paste them into another file or into a **table**. Even though **capture** continues to receive numbers, they do not automatically appear in the editing window, so you have to re-open its editing window each time you want to view any newly received numbers.

To see any kind of message—symbols, numbers, lists, whatever—you can use the **Text** object. It works similarly to **capture**, although its memory capacity is somewhat more limited. To see any message directly in the Max window, use **print**. The **print** object does not try to understand the messages it receives, it just posts them verbatim in the Max window. The Max window scrolls as each new message is printed, and you can scroll up to see previous messages. The disadvantage of **print** is that the time needed to print the messages and scroll the Max window is often greater than the time between messages, so **print** may get behind, affecting the timing of your patch.

If all you need to do is verify that some message, any message, has been sent, use a **button**, which will flash each time it receives any kind of message.

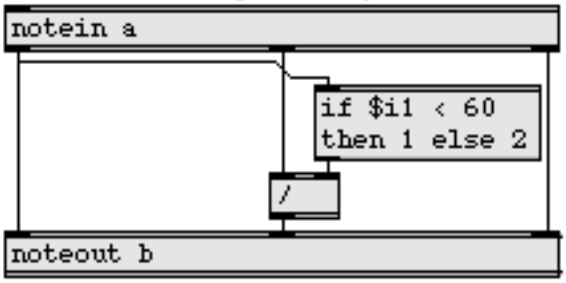
Message Order

Sending messages in incorrect order is a frequent cause of bugs. It's important to remember the basic rules of message order, which will help you write your patches correctly.



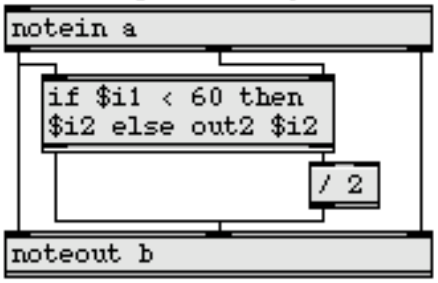
Ignoring this right-to-left ordering can lead to bugs like the one in the following example. Here the intent is to reduce the velocity of all notes from Middle C on up. In the patch on the left, however, because the velocity value is sent out of **notein** before the pitch value, and the **/** object is triggered by the message received in its left inlet, the pitch value gets to the **if** object too late.

Rather than halve the velocity of notes with pitch 60 or greater...



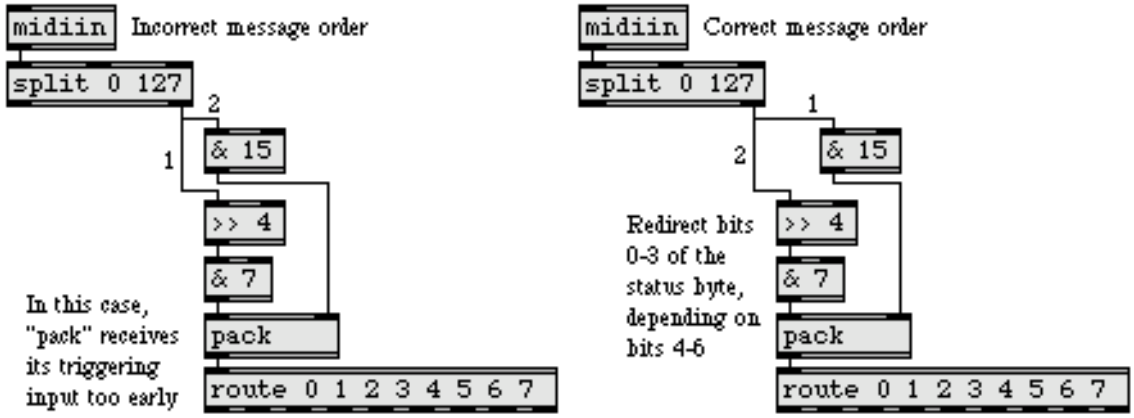
...this uses the pitch of the previous note to determine whether to halve the velocity

This halves the velocity only if the pitch is 60 or greater...



...by storing the velocity (in the right inlet) until the pitch value arrives in the left inlet

In the patch on the right, the velocity value is stored in the right inlet of the **if** object until the pitch value arrives, so the patch works properly.



The positioning of objects on the screen affects the way the patch functions

In the patch on the left, the **>> 4** object and the **& 15** object are perfectly aligned vertically, and therefore the **>> 4** object receives its input first. As a result, the **pack** object gets triggered before the number arrives in its right inlet. In the example on the right, the patch has been debugged simply by moving the **& 15** object a few pixels to the right.

If you're not aware of the right-to-left (and bottom-to-top) order in which Max messages are sent, you may be troubled by the fact that moving an object one pixel can potentially change the way a patch works. However, if you remember these ordering principles, you can tell at a glance the exact order in which messages will be sent.

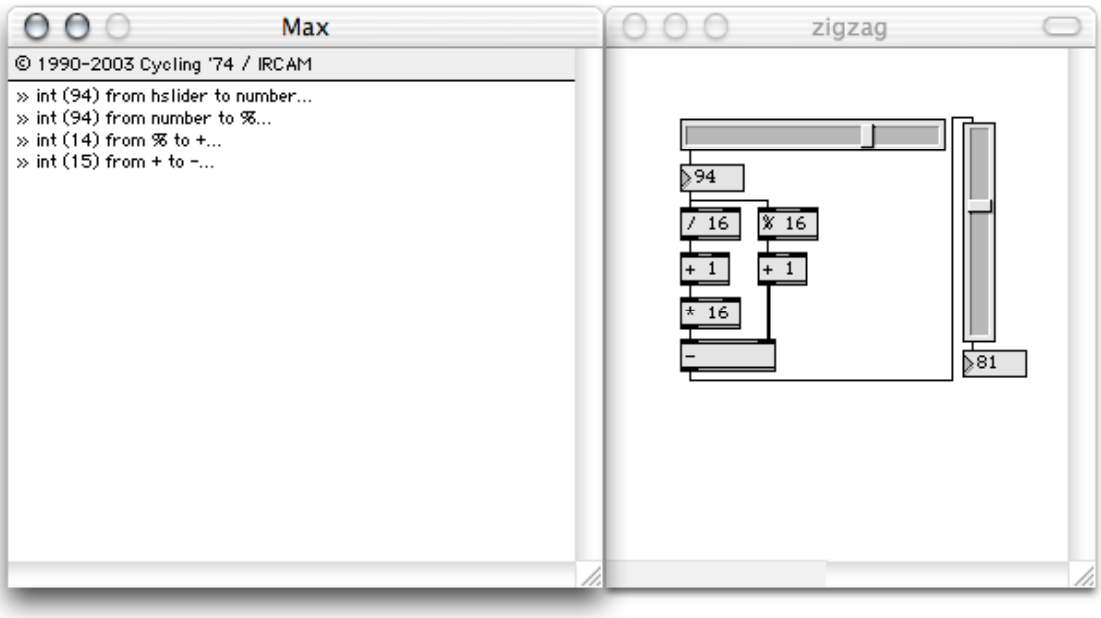
Tracing Messages

When you're working with a complex patch, it may be difficult to analyze the order of messages at a glance (going into and out of subpatches, memorizing what has happened so far, etc.). Fortunately, Max has a message tracing feature which displays the order of messages to you by blinking the patch cord through which a message is about to be sent.

By choosing **Enable** command from the Trace menu, you enable Max's message tracing feature. (Note: You cannot enable message tracing if Overdrive is enabled in the Options menu, and you cannot enable Overdrive if message tracing is on.) You then set the patch into action (cause a message to be sent) by sending it a MIDI event, entering a keystroke

on the computer's keyboard, or clicking on a user interface object with the mouse. Max will blink the patch cord through which the message is about to be sent, and will report information in the Max window about the sending and receiving objects and the message that is being sent.

From that point on, each time you choose **Step** from the Trace menu, Max moves on to the next message to be sent, flashes the patch cord through which it will travel, and reports about it in the Max window. In this way you can step through the workings of your patch at your own pace.



In Trace mode, the patch cord flashes and the message is printed in the Max window

Alternatively, you can choose Auto Step from the Trace menu, and Max will step through the different messages at a constant moderately fast rate, reporting as it goes. If you choose Continue from the Trace menu, Max will go on tracing, but at full speed.

Before tracing messages, you can select one or more particular patch cords (when the Patcher window is unlocked) and choose **Set Breakpoint** from the Trace menu. That will cause message tracing to pause each time a message gets sent through one of those patch cords. In that way, you can move through the messages at full speed with the **Continue** command, and Max will pause when it reaches the patch cord you have designated as a breakpoint, allowing you to examine the state of the patch at that moment.

Error **Messages**

If you make a programming error, Max will often print an error message telling you about the mistake in the Max window. Many errors are reported while you are editing in the Patcher window (such as trying to put an object into your patch that doesn't exist), but other errors do not become evident until you actually run your program (such as sending a certain message to an inlet that doesn't understand that message). A list of error messages, likely causes of each message, and possible solutions can be found in this Tutorials and Topics Manual under Errors.

Comment

There is probably no known case of a programmer complaining because a program contains too many comments. Explanatory notes in a **comment** object can help others understand your patch, and can help you remember what you have done, when you go back and look at it later. It is surprising how fast you can forget why you wrote a program the way you did. You may even want to use a Text window to make notes to yourself or to jot down ideas for future reference.

Using colors for patch cords and objects can also be a form of commenting your patch. You could, for example, use a distinctive color to mark all the objects and connections where MIDI information flows through a patch, distinguishing it from objects and connections that handle the user interface. An easy way to set the color of a patch cord or object is to control-click on it to get a contextual menu, then choose a color from the Color submenu.

See Also

Efficiency	Issues of programming style
Encapsulation	How much should a patch do?
Errors	Explanation of error messages

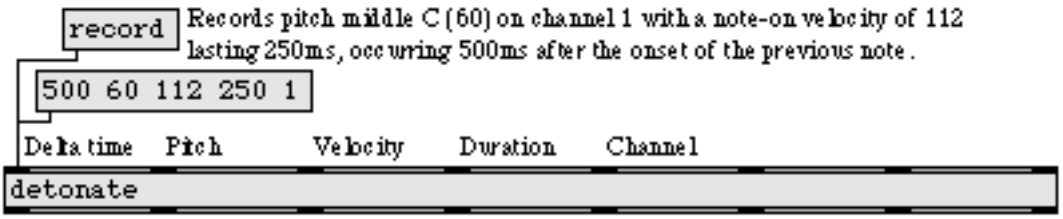
Uses of detonate

The **detonate** object is a flexible sequencing, graphic editing, and score-following object. It can record a list of notes tagged with time, duration, and other information. You can save the note list as a single-track (format 0) or multi-track (format 1) MIDI file, and you can read in any MIDI file that has been saved to disk by **detonate**, **seq**, or some other sequencer such as Cubase. Double- clicking on a **detonate** object displays its contents in a graphic editor window, allowing you to use the mouse to add or modify notes inside it. It is also able to act as a "score-reader," much like the **follow** object; it looks at incoming pitch numbers and reports whenever an incoming pitch matches the current pitch in the stored score.

Unlike other sequencing objects such as **seq**, **follow**, **mtr**, and **timeline**, however, **detonate** does not really run on an internal clock of its own. Timing and duration information must be recorded into it from elsewhere in the patch, and the patch must also use that information to determine the rhythm and speed at which notes will be played back from **detonate**. Although this means you'll be required to do some additional Max programming to make it do exactly what you want, it also means that you can program recording and playback options not available with the other sequencing objects, such as non-realtime recording, continuously variable playback tempo, and triggering individual events of the sequence in any desired rhythm.

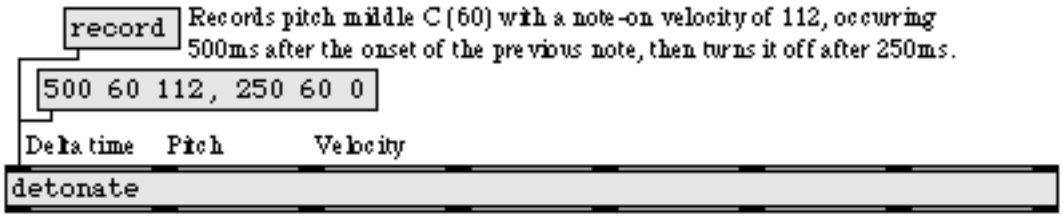
Recording Into detonate

You can use **detonate** as a sequencer of MIDI notes, to store pitch, velocity, and MIDI channel information. This basic MIDI information must be combined with timing information telling when the note should occur, and how long it should last. The "when" is established by recording a delta time in the left inlet for every note event. The delta time is the number of milliseconds between the beginning of that note and the beginning of the previous note. The "how long" is determined by the number most recently received in the 4th (duration) inlet.



Recording delta time and note duration as part of the note event

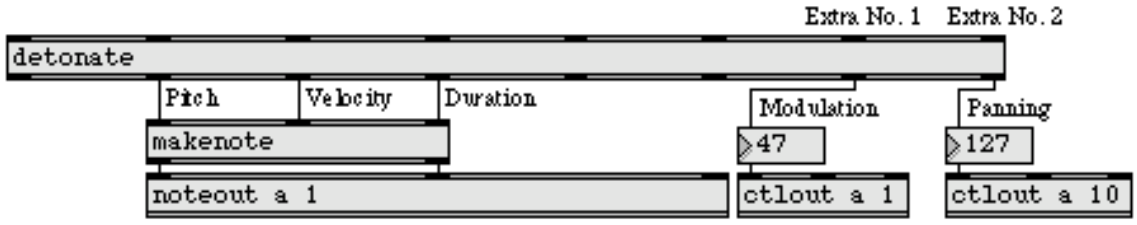
The duration can also be established by a later note-off message (a note with velocity of 0) on the same pitch. When a note-off event is received after a corresponding note-on, the delta time between the two events is used (actually, the sum of any delta times between the two, if there were other intervening events) to set the duration of the note-on message, and the note-off message does not actually get recorded as a separate event.



Letting detonate determine duration based on the delta time between note-on and note-off

A track number may be supplied in the 6th inlet, which is useful for separating recorded note events into different streams to be saved as a multi-track MIDI file. Notes recorded on different tracks show up as different colors in the graphic editor window, and the track number can be used as a criterion for selectively muting notes in **detonate** or selectively modifying them on playback.

The 7th and 8th inlets are for any additional information you may want to record as part of a note event. For example, each note could be assigned its own vibrato depth and pan position when recording, and those data would be sent out when the notes are played back.



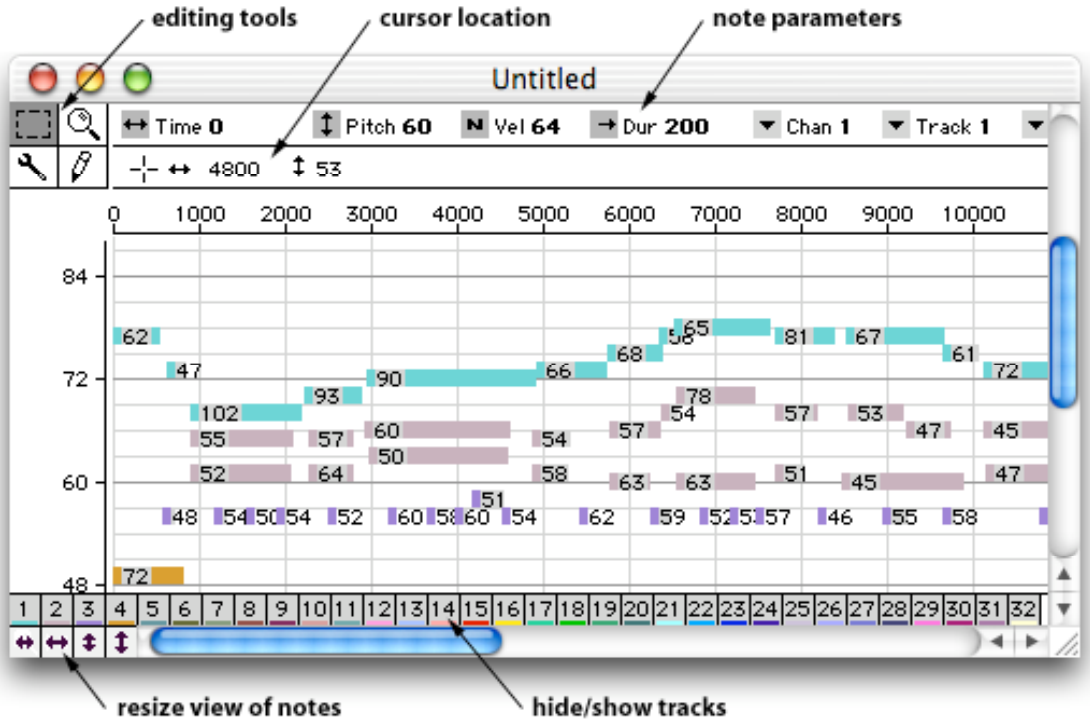
Additional data may be associated with each note event

The detonate Editor Window

Double-clicking on the **detonate** object in a locked Patcher opens a graphic editor window for viewing and modifying its contents. The recorded notes are shown in the editor window in a piano-roll-like view. Time is shown on the x axis, pitch is on the y axis, the

duration of notes is proportional to their length, and the velocity of each note appears as a number on it.

Each track of a multi-track file is shown in a different color.

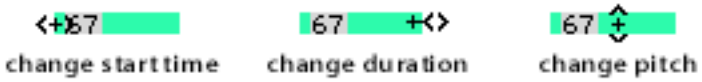


To select a specific note for editing, choose the selection tool from the palette in the upper left corner of the window, then click on—or drag around—the note you want to edit. You can select multiple notes by dragging around them or by Shift-clicking on them one at a time.

You can change the starting time, pitch, or duration of the selected notes simply by dragging on one of them with the selection tool (or the tweak tool for finer resolution dragging). The cursor will change, depending on where you click on the note:

- If you click on the left side of the note you can drag horizontally to change the starting time of the selected note(s).
- If you click on the right side of the note you can drag horizontally to change the duration(s).

- If you click in the middle of a note you can drag vertically to transpose the pitch(es).

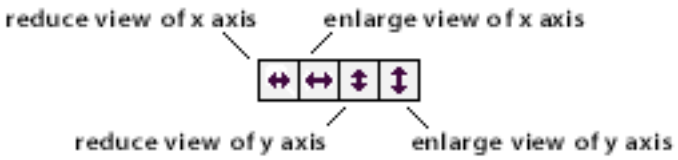


You can also change the value of any of the parameters of the selected notes by dragging on the **number box** objects at the top of the window.

If you want to add new notes to the score, you can simply draw them in with the pencil tool. Where you draw determines the start time, pitch, and duration of the note; all other parameters are determined by the values shown at the top of the window at the time you draw the note.

Changing the View in the Editor Window

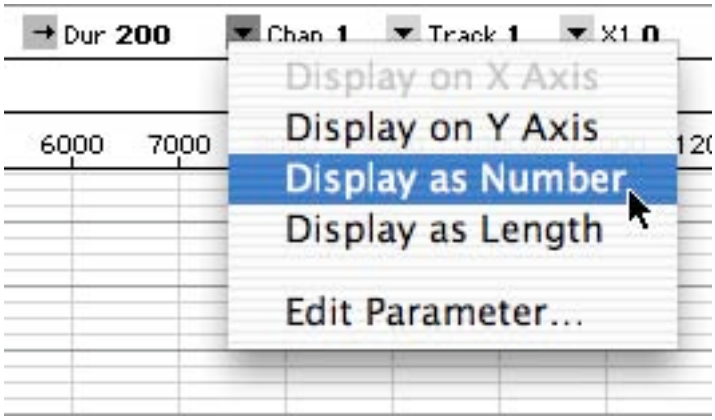
You can zoom in and out on the view of the score by clicking on the resizing arrows at the bottom left corner of the window.



To zoom in on a particular spot in the score, choose the zoom tool and click on the spot you want to enlarge. Option-click on Macintosh or Alt-click on Windows on the spot to zoom back out.

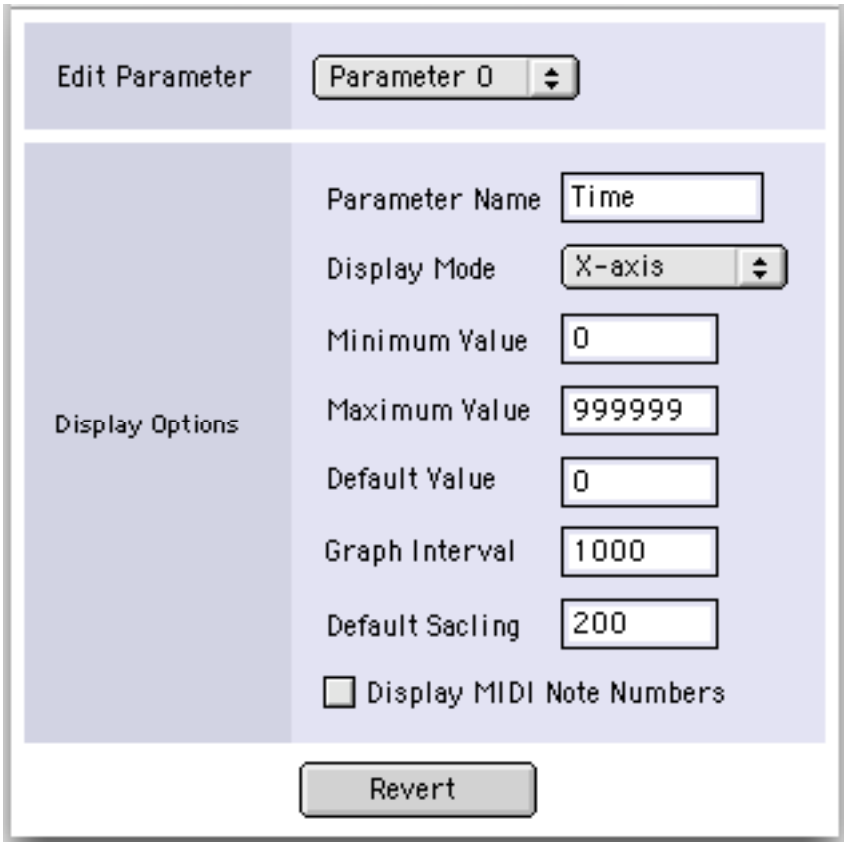
Although the depiction of the note parameters is normally as described in this chapter, you can change the depiction by reassigning the way each parameter is shown. When you click on the icon to the left of a parameter name, the icon becomes a pop-up menu, letting you choose how you would like that parameter to be depicted.

So, for example, rather than showing velocity as a number on the note, you could choose to show MIDI channel instead.



Icon becomes a pop-up menu for changing the display of a given parameter

As a matter of fact, by choosing Edit Parameter... from the pop-up menu, you can change many other aspects of how the parameter is displayed.



You can change the name of the parameter, its minimum and maximum possible values, and the default value that will be used for that parameter in notes where it is left unspecified. Graph Interval affects the view only if the parameter is displayed on the y axis; it controls how often numbers will be shown along the y axis (every 12 semitones in the above example). Default Scaling is a factor that determines the default zoom of the axis on which the parameter is being displayed. 1 is maximum zoom, and larger numbers are successively smaller scales. The values on the y axis can be displayed as MIDI notes instead of decimal numbers only for parameter 1 (pitch); this option is disabled for all other parameters. The start time (the leftmost parameter) is an exceptional case because it can only be displayed on the x axis; so, for that parameter Graph Interval and Default Scaling refer only to the x axis.

The fact that the name and characteristics of all the parameters can be so easily changed suggests that **detonate** can actually be used as a collector of arbitrary lists of numbers. It is designed for holding lists that represent note events, but the numbers can in fact mean anything (as is true of almost all numbers in Max), so you can use it to store and recall virtually any collection of lists of integers that you might want to represent and edit graphically.

Editing Shortcuts

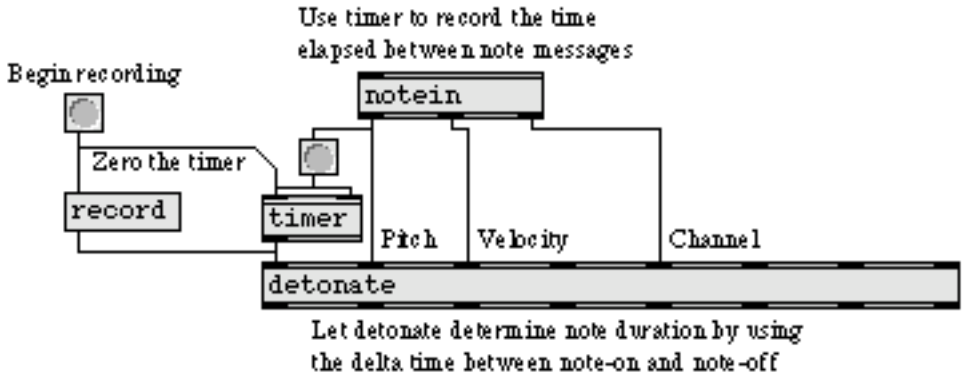
Certain keys on the computer's keyboard are shortcuts for switching editing tools. If you are currently using the pencil tool, holding down the Option key on Macintosh or the Alt key on Windows switches you temporarily to the selection tool, and vice-versa. Holding down the Command key on Macintosh or Control key on Windows temporarily switches to the tweak tool, and the Control key on Macintosh or a Right-click on Windows temporarily invokes the zoom tool.

Shift-clicking on a note adds it to, or removes it from, the current selection.

Techniques for Using detonate

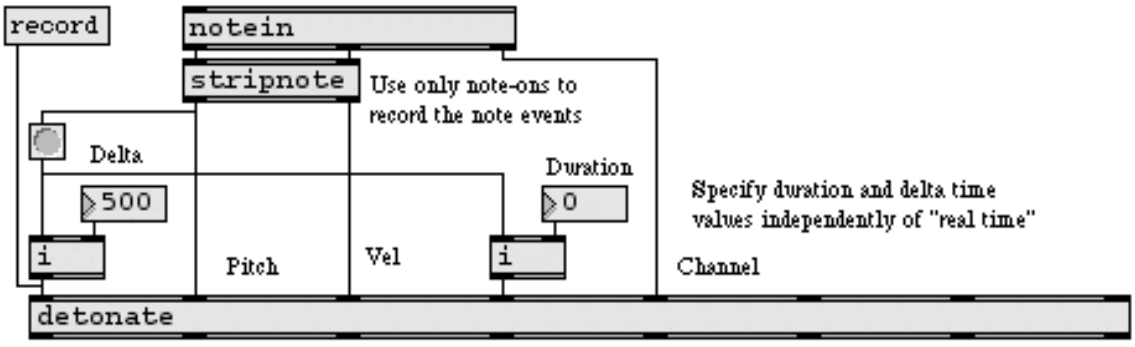
To use **detonate** as a sequencer for timed playback of note events, you will need to a) produce values for recording the duration and delta time parameters of each event and b) use some sort of timing object to control the speed with which **detonate** sends out the note data, presumably using the delta time value to determine the time between notes.

The following example shows the simplest method for recording delta times and durations directly from incoming MIDI note messages, in real time.



At the same time as we send the record message to **detonate**, we start the **timer**. Each note message that comes in causes **timer** to report the elapsed time—which gets recorded along with the pitch, velocity, and channel—and then restarts the **timer**. The duration value for each note event is calculated by **detonate** itself. It measures the time elapsed between a note-on and its corresponding note-off, uses the time difference as the duration value, then throws away the note-off message.

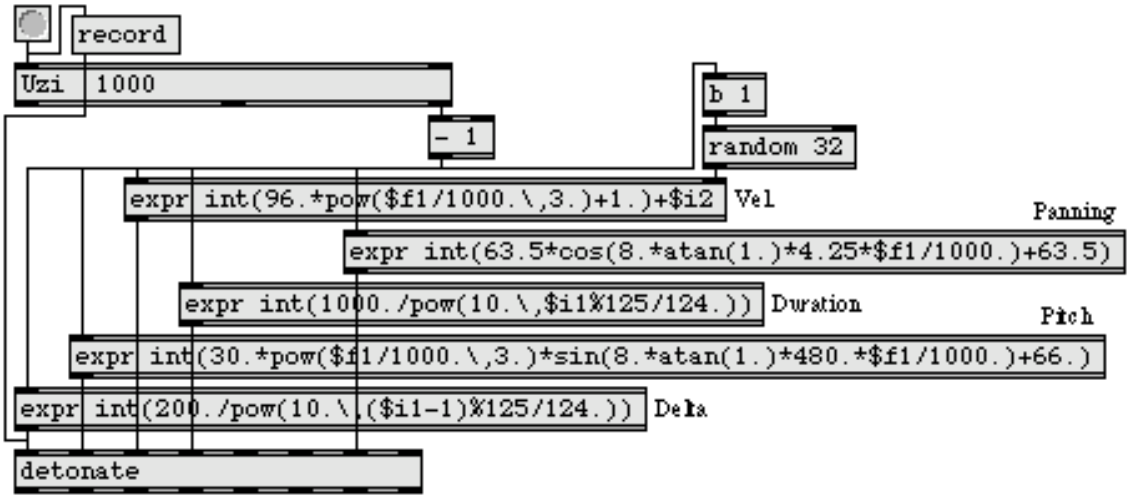
It is noteworthy that **detonate** doesn't have any sense of "real time." It dutifully records the received delta time, but it doesn't really care how much time actually passes between received messages. It simply stores note events in the order received. For that reason, it's very easy to record notes into **detonate** in non-real time, as with the "Step Record" feature in many MIDI sequencing applications.



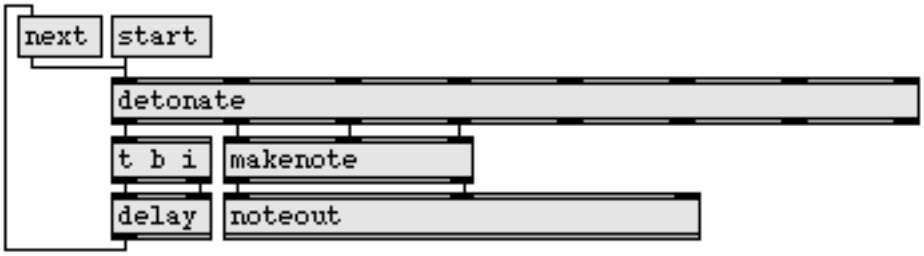
Using detonate as a non-real time "step" recorder

And, of course, just as the rhythm and note durations can be manufactured "artificially," all the other note parameters can likewise be generated algorithmically within Max, rather

than being played in via MIDI. The following example composes and records a 1000-note melody instantly at the click of a button, using mathematical expressions to calculate different curves for pitch, velocity, panning, and rhythm. (You can examine and hear the results in the example patch for Tutorial 44.)



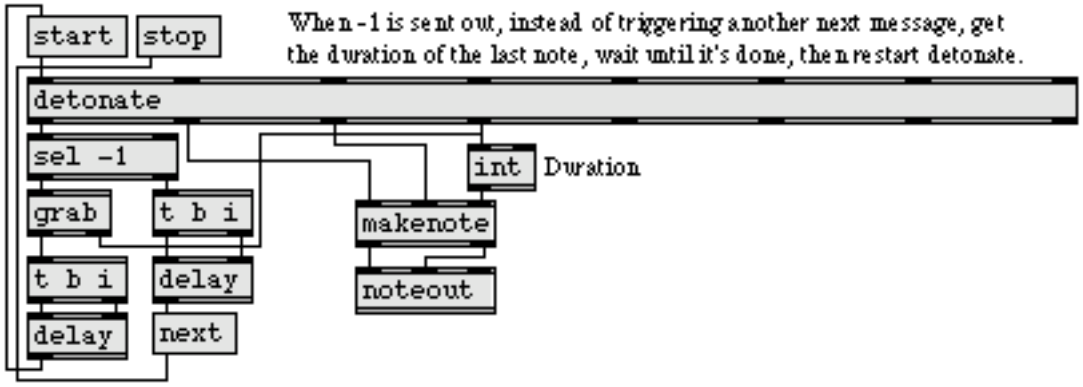
When **detonate** receives a start message, it does nothing except send out the delta time of its first note event. After that, each next message received causes **detonate** to send out the rest of the data for the current note, and the delta time for the next note. So, the delta time can simply be used as a delay time before sending the next next message, as shown in the following example.



The delta time of the next note is used as the delay time before triggering the next note

When the very last note in the score gets triggered by a next message, there is no following note, so **detonate** cannot possibly send out the next delta time. In place of a delta time, it sends out -1, which is a signal that the last note has been played. Your patch can look for that signal, and use it to trigger some process.

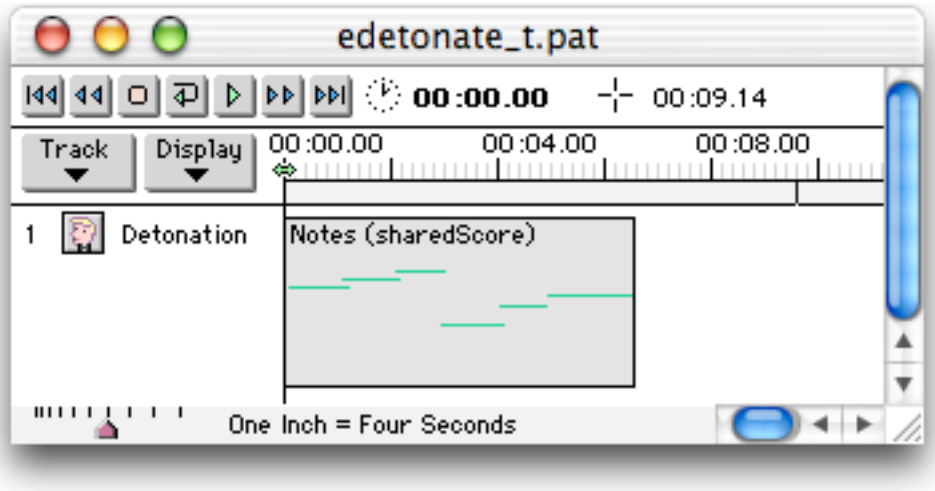
In the following example, the end-of-score signal is used to restart **detonate** when the last note has ended, in order to play the score in a loop.



A delta time of -1 signals that the last note has been played

Using detonate in a Timeline

A timeline event editor called **edetonate** can send list messages from a timeline. For any timeline event that sends a list message to **ticmd**, an **edetonate** may be placed in the timeline to represent that event. You can then double-click on it to open its own editor window, draw in the note events, and when the timeline is played the notes will be sent out over the period of time represented by the length of the **edetonate** in the timeline.

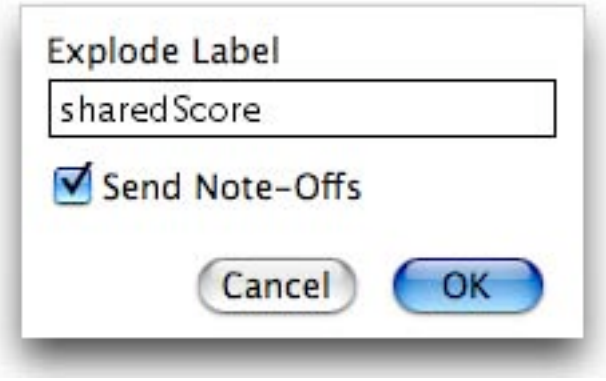


This means that the time units shown in the editor window of **edetonate** are actually relative time units, because the real time in which they occur depends on the length of the

event in the timeline. In the preceding example, for instance, each of the notes was drawn into **edetonate** as a 1-second note, but because the event stretches out over precisely one second in the timeline, the list messages will actually be sent to **ticmd** Notes every 1/6 of a second when the timeline is played.

However, if you want the notes in **edetonate** to be played at exactly the same rate as they were drawn in the graphic editor window, select the **edetonate** and choose **Fix Width** from the Object menu. The length of the **edetonate** will be changed so that its notes play at the same rate as they were drawn in **edetonate**'s editor window.

By selecting an **edetonate** editor and choosing **Get Info...** from the Object menu, you can assign it a name. It will then share its contents with any other **edetonate** editors that have the same name, or with a single **detonate** object that has the name as a typed-in argument in a patcher.



You can also choose to have **edetonate** suppress note-off messages, by deselecting the Send Note- Offs option. When Send Note-Offs is checked, **edetonate** uses the duration information of the note events to decide when to send a corresponding note-off message to **ticmd**.

See Also

detonate	Graphic score of note events
follow	Compare a live performance to a recorded performance
Sequencing	Recording and playing back MIDI performances
Timeline	Creating a graphic score of Max messages
Tutorial 44	Sequencing with **detonate**

Editing: Templates, Clippings, Prototypes and Shortcuts

An Overview of Editing Features

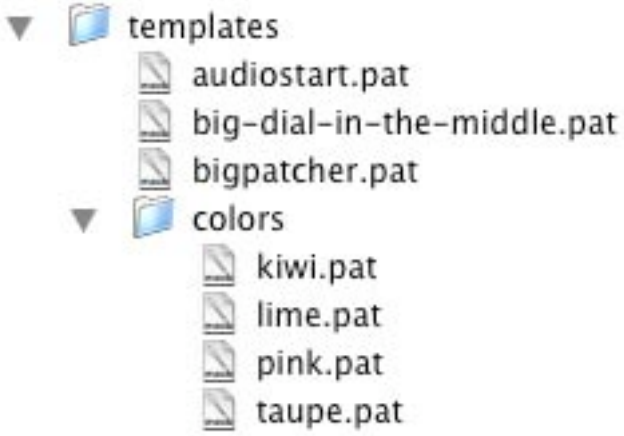
Starting with Version 4.5.5, Max includes a number of features in the patcher editing environment that permit customization to make your everyday work with the software more productive. This chart provides a brief description of these features, their purpose, use and scope, and a listing of the location of the files.

Feature	Useful For…	How to Use	Applies To…	Location of files
Templates	Creating common starting points	Choose from File->New menu	Entire patcher windows	:patches:templates
Clippings	Adding commonly used groups of objects to patches	In an unlocked patcher, control- or Right-click on blank space to get a menu, then choose an item in the Paste From… submenu	Inserts a patcher's contents in another patcher	:patches:clippings
Prototypes	Creating pre-configured user interface elements	Select an object, then choose Prototypes from the Object menu	Individual user interface objects	:patches:object-prototypes
Shortcuts	Reducing excessive typing of object names and arguments	Type shortcut then press the Esc key	Text in object boxes	Init folder in Cycling '74 folder

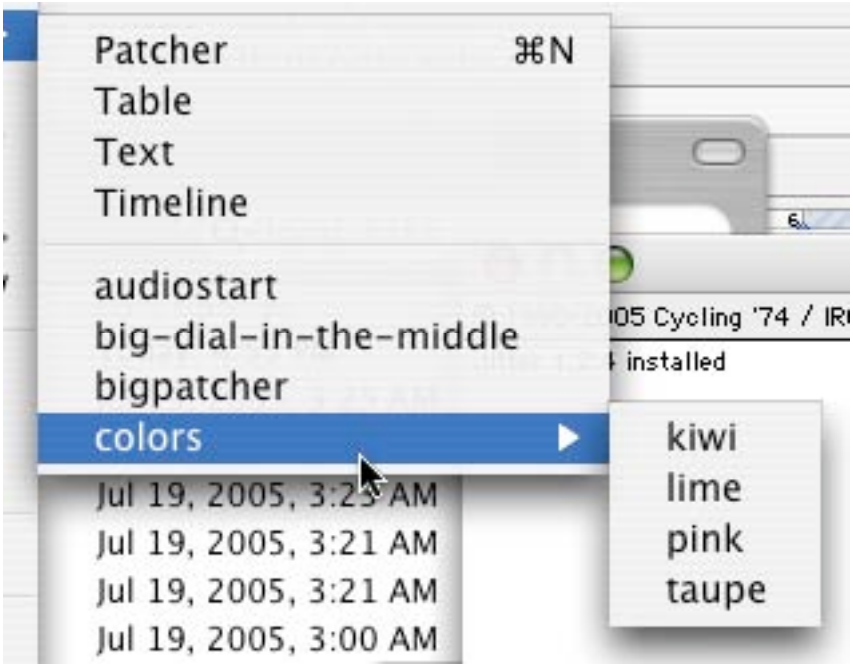
Templates

Templates are probably the easiest to understand of the new editing features. The patches folder inside the Max application folder contains a folder called templates. In this folder you can place what some applications call "stationery." In Max's case, these are patcher files you wish to use as a starting point for further work.

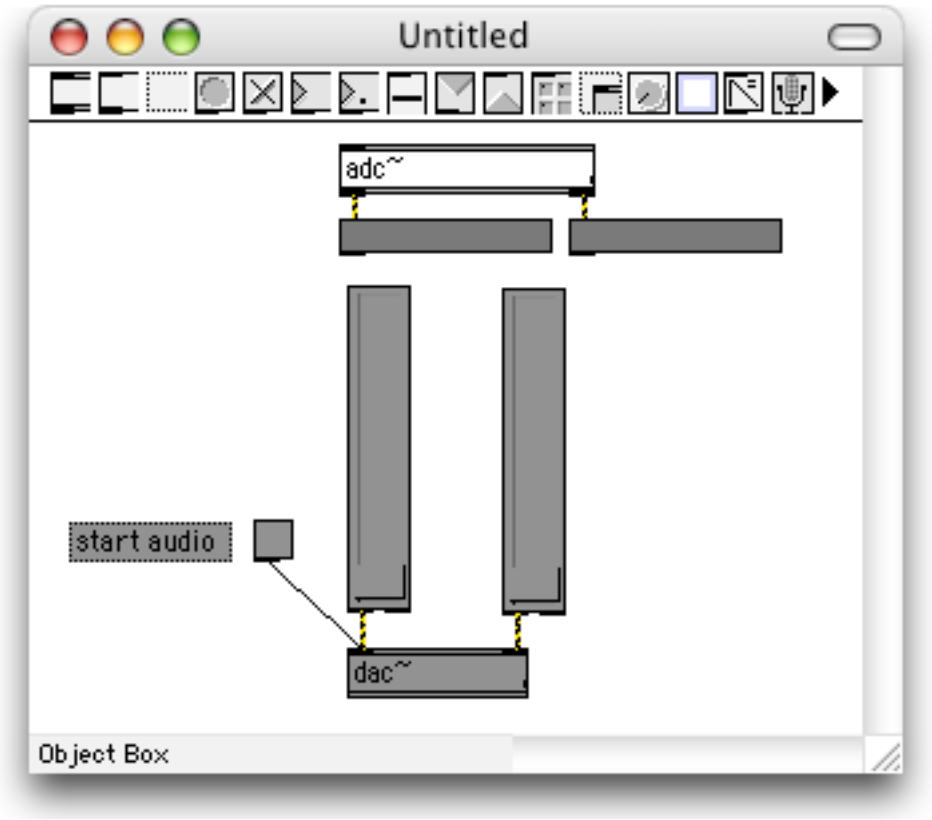
Here are the files you might see in a typical templates folder:



The New submenu of the File menu in Max reflects this organization as shown below. Templates appear in the New menu below Patcher, Table, Text and Timeline.



When you choose one of these items in the New menu, the patcher file is opened, but the window is untitled, unlocked, and not marked as modified. Here is the audiostart patcher opened as a Template (we've made it smaller to fit on the page):



The idea of the audiostart template file is that pretty much every audio patch you make will have a **dac~** object, some **gain~** sliders, a **toggle** to start audio etc. Why put them in your patch manually each time?

Other templates might save particular sizes and shapes of patcher windows or include a particular color background (that's what the user did above with the colors folder). We're sure you'll think of other Templates that will be useful starting points. For example, if you write plug-ins, you could create a template with the **plugin~, plugout~,** and **plugconfig** objects as well as the testing mechanisms you commonly use.

If you want to modify a template file, choose Open from the File menu and navigate to the templates folder inside the patches folder inside the application folder. Only when opened via the New menu do Template files behave differently.

Clippings

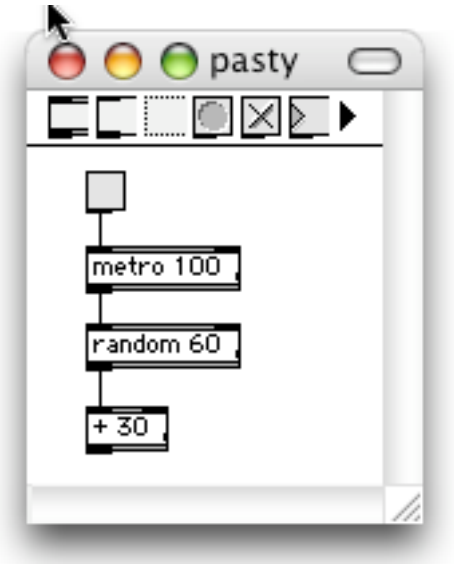
Many people end up patching the same little things every time they use Max. It doesn't help that much to use a subpatcher or abstraction for these "bits of code" because you often want them in the same patcher that you're working on. Abstractions have the disadvantage that they can't be modified easily. And both subpatchers and abstractions put the commonly used group of objects in a different window where it's hard to get to them.

If you find yourself repeating the same patch over and over again, you might use the clippings folder, located in the patches folder inside the Max application folder. The contents of the clippings folder is added to submenus of the **Paste From…** item in the patcher contextual menu. Perhaps you've never even used the patcher contextual menu, but we think you should consider checking it out, because **Paste From…** could save you a lot of time. **Paste From…** pastes the *contents* of a patcher file right into the patcher you're working on, with the top-left corner of the patcher window located at the current cursor position (i.e., where you clicked to obtain the contextual menu).

To obtain the patcher contextual menu, control- (Mac) or right- (Windows) click in a *blank space* in an unlocked patcher (i.e., not on an object or patch cord). **Paste From…** is the last item and its submenu will list all of the patchers in the clippings folder.

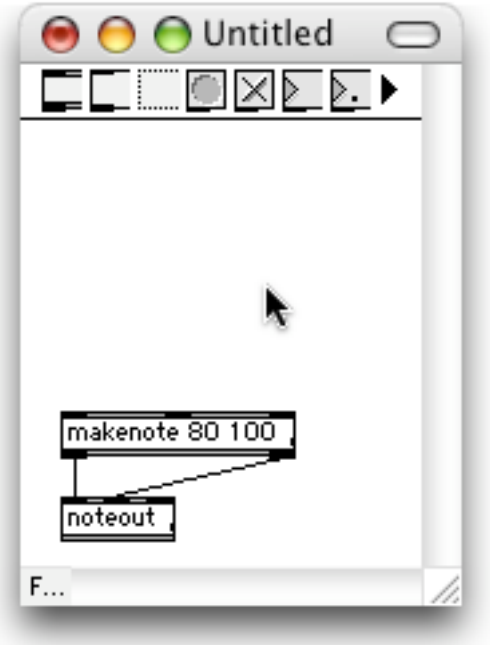
Choose one of the items in the submenu. Its contents will be pasted at the location you clicked to get the submenu.

For example, consider the following patcher window:

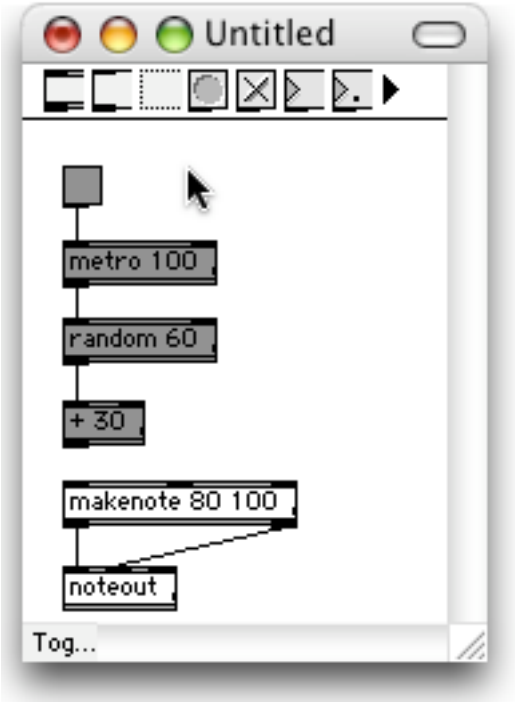


We saved the above window as a file called **pasty** in the clippings folder inside the patches folder in the Max/MSP application folder.

Now we can use **Paste From…** to put this into another patcher, which could really use some random notes.



Control- or right-click to obtain the Patcher Contextual Menu. Then choose pasty from the **Paste From…** submenu. The objects in pasty appear where you clicked.



The **Paste From…** menu contents from the clippings folder contains some very basic ideas to get you started. You can use the **Other…** item to open any patcher and paste its contents into the patcher you're working on.

Prototypes

Prototypes transform individual user interface objects with commonly used combinations of settings.

Some Max user interface objects have a large amount of tweaky configurations you can set in an Inspector window. In particular, you can make beautiful sliders and dials with objects such as **pictslider** and **pictctrl**, but once you've made them, they're probably sitting in a patcher somewhere. You have to remember where the patcher is, copy the object out of the file, and then paste it into the patcher you're editing. Or, often as not, recreate the object from scratch.

Prototypes turn retrieving pre-configured user interface objects into a one-step process. Prototypes contain all of the settings for an object that you would otherwise set one at a time in the object's Inspector window.

Prototypes can be applied to all user interface objects except object boxes.

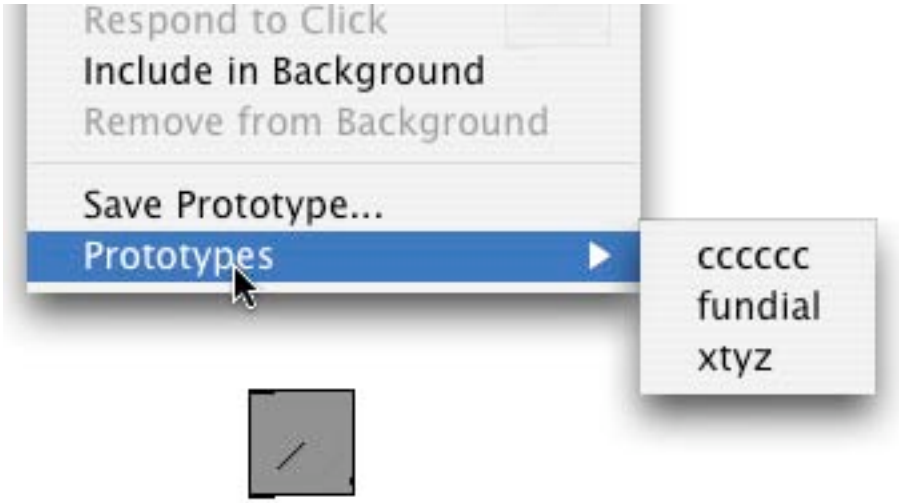
When you move the mouse over a user interface object's icon in the patcher window palette or scroll through the menu of icons to the right of the palette, you'll see that some objects have a number of prototypes listed in parentheses after the object description in the assistance area of the patcher window. For example, the text below will tell you that the **pictctrl** object has one prototype:

Picture Control (1 Prototype)

When you create a new **pictctrl** object, you'll get a default object. It has a generic size, no associated picture, and no behavioral settings. It's pretty useless. But it can instantly be made useful by selecting one of the object's prototypes. Here's how you do it:

When creating a new object, position the cursor where you want the object to go, then click. But instead of releasing the button as you normally would do, hold it down for a second or so. You'll see a menu listing all of the available prototypes. Choose one and the prototype will be applied instantly to the object.

- At any time after creating an object, select the object and choose an item from the Prototypes submenu of the Object menu. The prototype you choose will be applied to the selected object.

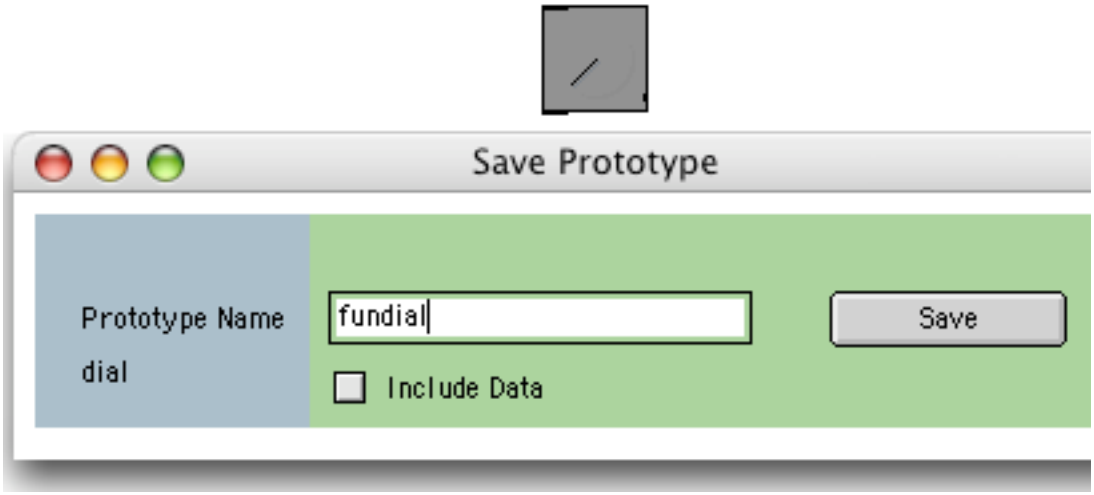


- Use the object contextual menu to obtain the Prototypes submenu when control- or right-clicking on an object in a patcher. The prototype you choose is applied to the object on which you clicked.

Applying a prototype is undo-able, but you can only apply a prototype to one object at a time.

Saving Prototypes

Once you have a collection of object settings that you like, you can save it as a prototype to use later. Select the object you want to save and choose **Save Prototype…** from the Object menu.



Name the prototype in the window that appears, then click the Save button. Your prototype is saved in a subfolder of the object-prototypes folder.

If you use an object with a prototype in a patcher you save, you don't need to worry about keeping the prototype around for the next time you open the patcher. The prototype feature is really an editing tool, it merely *replaces* the object you have with a new object that is created according to the instructions in the prototype file. In other words, unlike an "abstraction" a prototype is not a reference to an object. If you save over an existing prototype, all of the objects that were created with that prototype will be unaffected.

Prototypes and Object Data

A prototype can contain preset data from an object—check *Include Data* before saving the prototype. The data in the existing object is always replaced, either by the preset data in the prototype, or by the default data. In some cases, the "data" of an object is not necessarily its preset data. For example, the text of menu items for the **umenu** and **ubumenu** objects is saved with an object in a patcher, not in a preset (the current menu item selected is saved in a preset).

An object's connections, patcher scripting name (if any), and imageburger are preserved when a prototype is applied.

Prototypes for the bpatcher Object

One of the most powerful uses of the Prototype feature is its ability to create a collection of commonly used patcher elements using **bpatcher** objects. The prototype will save the current settings of the bpatcher (for example, the visible area of its client patcher). This could be useful if you are trying to create a catalog of visual "components" that you want to patch together.

Patcher Selection of Text Objects

Storing data from objects in a patcher that you copy as text exposed an ambiguity about selection in a patcher. When you click on an object box, message box, or a comment, have you selected the box or the text inside the box? Here's how Max works:

- When a box appears selected for text editing, and you copy or cut it, you copy the text inside of the box to the clipboard, not the data for recreating the box.
- When the box is selected "as a whole" the box itself is copied to the clipboard (i.e., the data for recreating the box) so you could paste a copy of the object somewhere else.

Max explicitly copies both the text inside the box and the box as a whole—when you have clicked inside an text object to select the text for editing, the text from an object you copied will be pasted. Otherwise the object as a whole will be pasted.

Clicking on a text object selects the text inside of it for editing. If you click and drag the box somewhere before releasing the button, it will *not* be enabled for text editing (at least visually).

Dragging *around* a box to select it does *not* select the text inside for editing (at least visually).

For more information on setting options for selecting text in a patcher window, See the description of the **Text Selection...** portion of the Option menu found in the Menus section of the Max Fundamentals manual.

When the **Typing Automatically Edits Selected Box** option is on (which it is by default), typing in an unlocked patch will route key presses to the selected text object and enable it for editing *even if it is selected as-a-whole*. When it is off, the text *must* be visually selected (as shown below) or there must be an insertion point in the text before you can edit it by typing.

When the **Select Text on Click** option is on (which it is by default), clicking on a box without moving it immediately selects the text for text editing. When it is off, clicking on a box always selects it as-a-whole. Clicking on it again, selects all of the text, and clicking on it yet again moves the insertion point to the place you clicked. This behavior is consistent with the Mac OS X Finder.

A nice addition to all of this is the change to the role of the Enter key (on both Mac and Windows). The Enter key now enters and exits text editing mode for a text object. Let's say you have an object selected as a whole…

Press Enter and the box's contents are now ready to be edited as text.

Press Enter again and your changes (if any) are updated. The object is selected as a whole again. Pressing Enter moves between the two selection modes.

> Note that Windows XP machines only have an Enter key while Mac OS X machines have separate Return and Enter keys. On Windows XP the Enter key functions as the Mac OS X Return key and Shift+Enter functions as the Mac OS X Enter key. So, on Windows XP pressing Enter adds a new line while editing the text of a text object, and pressing Shift+Enter toggles between text selection and box selection modes.
>
> An Editing Options patcher has been added to the Options menu, for changing the settings of *Select Text on Click* and *Typing Automatically Edits Selected Box,*

Program Size and Speed

When you are writing very big, complicated patches, or are linking many subpatches together inside one main patch, matters of program size and computational efficiency come into play.

When you open a patcher file, each object in the patch is loaded into the internal memory of the computer. A very large patch containing many objects and subpatches can take up a considerable amount of memory and can take a long time to load. Therefore, you may wish to consider how to build patches that avoid superfluous messages and objects.

There are usually several ways to accomplish the same programming task in Max, and usually one way will be more efficient than another in terms of program size and speed.

There are three efficiency issues to consider:

1. The loaded size of a Max program is a function of the number of objects (and subpatches), and the complexity of each one.

2. The load time of a complex program is also a function of the same two factors.

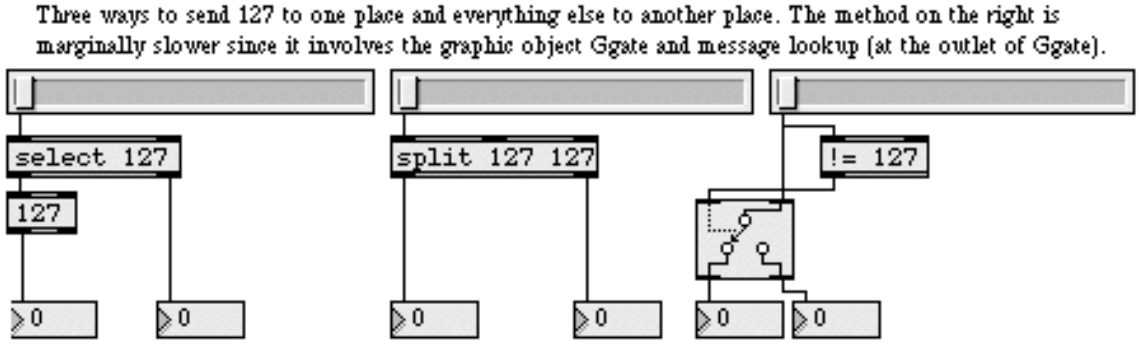
3. The "real-time" computational efficiency of a program is affected by the fact that some objects are more efficient than others in operation and communication.

Principles of Efficiency

Since there are so many different kinds of messages that can be sent in Max, an object often has to "look up" the meaning of the message it sends or receives. Computational speed is achieved primarily by avoiding this message lookup. Look at the description of the inlets and outlets of two connected objects in the Objects section to see if they share the same message type. In this case, Max will not have to do any "interpretation" of a message.

Efficiency

gate, **switch**, **Ggate**, and **Gswitch** have no message lookup when a value is sent in a right inlet. However, these objects always do a message lookup on output. Therefore, it's better, for integers, to use something like **select** or **==** if you're looking for a specific number.



Message lookup is a factor in computation speed, and redrawing graphic objects takes time

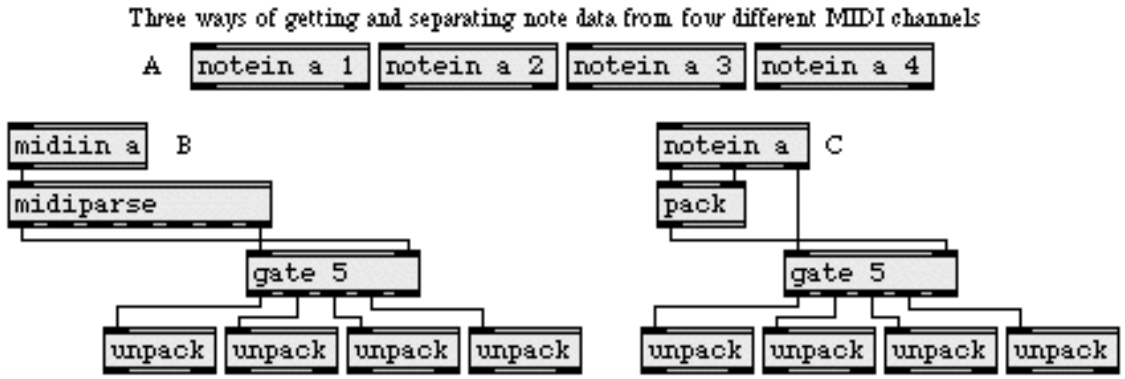
If you're not running in Overdrive mode, graphic objects slow you down because it takes time to redraw the screen. If you are in Overdrive, they don't slow you down, unless there's a message lookup involved. There is no message lookup with **number box** objects, for example, because they handle only numbers.

A message lookup is always performed on the output of **message** boxes. Therefore, it's better to type a number into an object box—which creates an **int** object—if you want to produce a constant value in an efficiency-conscious program. Of course you have to send `bang` to such an object (whereas a **message** box can be triggered by a variety of messages in its inlet), but if this can be arranged, it's a bit more efficient than using a **message** box. In the vast majority of cases, the difference in speed is negligible, but if enough instances like this are added up, they can have a noticeable effect.

If you send the same message repeatedly through the same outlet of a **message** box or other object whose outlets can send a variety of messages, a message lookup is generally performed only the first time the message is sent (due to a feature called *outlet caching*).

Efficiency

If you want to filter MIDI messages according to channel, it's better to use a channel argument in the MIDI receiving objects than it is to try and use the channel number output to route information later.



Method A is the most compact and efficient, both in memory and speed

Memory Usage

If you have written a rather large program (and especially if you have a computer with limited RAM) you will want to try and keep down the amount of memory your patch uses when it is loaded. Doing so will also make your patch load faster.

Try to avoid doing similar tasks with many copies of a single subpatch, since copies of all the objects contained within the subpatch are created for each instance of the subpatch you use. It is better to design your subpatch to work with a variety of incoming values than to use the #1-#9 argument feature to differentiate 50 copies of a subpatch.

There is a memory overhead of at least 100 bytes for every visible box on the screen, though boxes in closed windows take up less space.

See Also

Encapsulation How much should a patch do?

Complex Patches

Once you start writing relatively complicated programs, try to build them out of different parts, rather than one enormous, tangled patch in a single Patcher window. The way to do this is to divide your program up into different Patcher files. The different files can be subpatches of one main patch, so that they are all loaded when the main patch is opened.

Subpatches can communicate with each other via inlets and outlets, or via **send** and **receive** objects, and they can share data by using **coll**, **table**, or **value** objects which have the same name as an argument. There is no reason that a large and complicated program cannot be composed of many smaller parts, and the advantages of this approach are considerable.

Modularity

There are several important reasons why it is a good idea to use a modular approach to programming. One reason is that it makes it easier to verify that your program actually works, especially in extreme or unusual cases. This becomes harder and harder to do as a program grows in size and complexity. By building small modules and ensuring that each one works as its supposed to in and of itself, you reduce the number of possible problem spots when the modules are combined in a larger context.

A second reason is that many tasks in a program are used again in different contexts. Once you have built a small module that performs a certain task, you can use that module wherever the need for that task arises, rather than rewriting it each time.

Another reason is that many tasks in a program are similar to other tasks. By writing a small, general-purpose module (usually one that takes arguments so that its exact function can be modified by the argument), you can use that one module with different inputs or arguments, to do many similar things.

Finally, by encapsulating different portions of the program, you make it easier for yourself (or others) to see how the program works long after you develop it.

Encapsulation

The different modules of a program are best designed to encapsulate a single task. Name the module for what it does, and reuse the module should you ever wish to perform the task again in another program.

Encapsulation

By keeping certain values in one place, you only have to change them once if you decide they need to be modified. If the same values are distributed throughout your program, you have to find every instance of that value, and change each one individually.

One way to keep values in a single place, yet still make them available to many different objects is to store the values in a single file that can be accessed by any patch. For example, many different patches can read in values from the same **table** file by using **table** objects with the same filename as an argument. Changing the contents of that one file then changes the values used by all the patches that share that file.

Messages between Patches

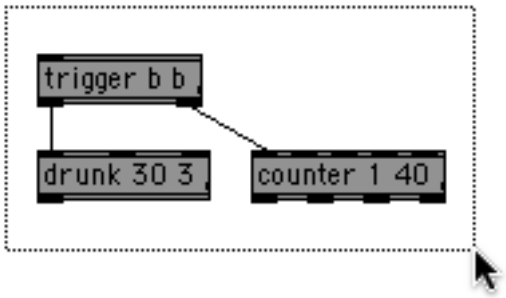
When designing small modules (patches) which will be combined in a larger program, it is important to consider not only what the patch does internally, but also the context in which it will be used. The context will determine what kind of messages you want each patch to produce and accept. For example, you might wish to use a `bang` to trigger a process, numbers to toggle something on and off or to provide values for calculation, or symbols (such as `start` and `stop`) to control a more complex task such as a sequencer.

The simpler the messages that a patch receives and sends, and the simpler the function of each patch, the greater the number of contexts in which that patch is likely to be effective.

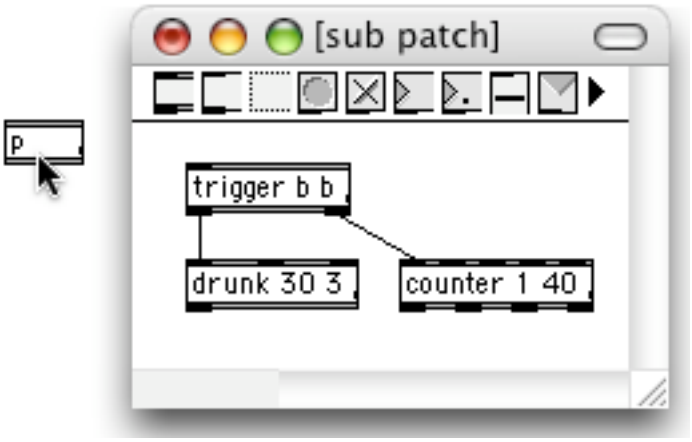
Encapsulation and De-Encapsulation

You can use encapsulation to clean up a patch you are making by putting a group of objects in a subpatcher.

- Simply choose the objects you wish to place in the subpatcher.



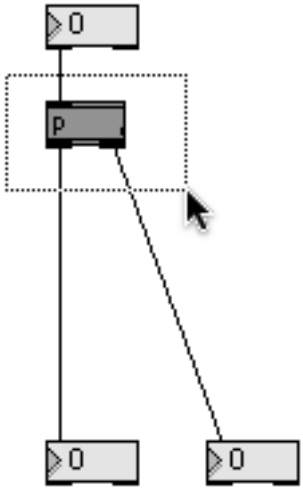
- Then, choose **Encapsulate** from the Edit menu.



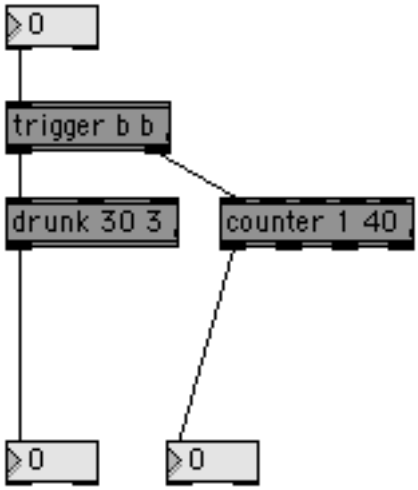
The objects are swept into a newly created subpatcher, inlet and outlet objects are added as appropriate. Don't like what happened? You can undo it.

The inverse operation is also possible. Sometimes objects stuck in a subpatcher are bothersome when trying to manage two windows to keep track of everything. You can now bring objects in a subpatcher "home" to their parent patcher with the *De-encapsulate* feature.

- Select a subpatcher



- Choose **De-encapsulate** from the Edit menu.



The subpatcher disappears and its contents are placed in the parent patcher, preserving any existing connections. De-encapsulation can be undone too.

Documenting Subpatches

Here are three tips for documenting your own patches that will be used as subpatches:

1. Give your subpatches informative names, so you'll remember what each one does.

2. Put Assistance text in each **inlet** and **outlet** object, to remind you of the inlet or outlet's purpose when using the patch.

3. If your subpatch is complicated, include **comment** boxes inside it to explain its operation.

See Also

Debugging	Techniques for debugging patches
Efficiency	Issues of programming style

Error Reports in the Max Window

Max prints an error report in the Max window when you make a programming mistake. Below is a list of error messages you may encounter, along with likely causes of each message.

"$" variable out of range
Occurs when you refer to an argument number out of the range $1-$9 in a message sent to a **message** box.

<filename>: error opening file (and variations)
An error occurred opening a file that was properly located. Most likely the file or media has a problem.

<objectname>: <filename>: can't open
Occurs when a patch is loading or when an object is created that reads its data from a separate file. The file that was to be read in automatically was not found in Max's search path or was not a type of file that the object is capable of opening. The erroneous filename has usually been specified as an argument to an object such as **coll**, **seq** or **table**. Make sure that the file is in Max's search path.

<objectname>: bad argument
Occurs when creating a new object with typed-in arguments. There is something wrong with what you typed after the name of an object. Usually the object is expecting a symbol, and you typed in a number, or vice versa. Check the object's argument specification list in the Objects section.

<objectname>: bad arguments for message <message>
Occurs when an object receives a message that it understands, but one of the arguments in the message is not what the receiving object expected. Usually the object was expecting a symbol argument and got a number, or vice versa. Check the object's input list in the Objects section.

<objectname>: doesn't understand <message selector>
Occurs when an object receives a message that it doesn't expect. It is possible to make patch cord connections that will result in improper messages being sent to an inlet. For example, Max will let you connect the outlet of a **message** box to almost any inlet, because there's no way of knowing what message will come out of the outlet. In such a case, the error does not become evident until you test the program and the message is actually sent.

<objectname>: message too long <message>
A message was sent that contained more than 256 elements.

<objectname>: missing arguments for message <message>
Occurs when an object receives a message that it understands, but one or more of the expected arguments in the message is missing. Check the object's input description in the Objects section.

<objectname>: No such object
Occurs when creating a new object or loading a document. When you are editing a patcher, and you type the name of a nonexistent object into an object box (or the name of an object or subpatch that is not in Max's search path), Max produces this error message.

When you open a document that contains an object that Max cannot find (either because it is not located in Max's search path or because it just doesn't exist), Max displays the object as if it existed, except that the object name is surrounded by a dotted outline in the object box, and an error message is printed in the Max window. This preserves the connections to the object box in case you can retype the object to create it properly.

A similar box is created when a user interface object referenced inside a file cannot be located.

<objectname>: fragment file not found
This error occurs when a collective references an external object that has been improperly stored in the collective. It should not happen with the current version of Max.

<objectname>: <filename>: file not found
This error occurs when a file name is either passed to an object as the argument to a read message or stored within an object saved within a patcher. The file cannot be located, either within Max's search path or with its full pathname.

<symbol>: bad arg types
Occurs when a patch is running and a symbol is received in the inlet of a bitwise operator such as **&**, **|**, **<<**, or **>>**. Make sure that only number messages are sent to bitwise operators.

<symbol>: no such object
Occurs when a message is sent to a **send** object, or to a **message** box that contains a

semicolon followed by the name of a receiver, and there is no **receive** object with the name specified in the **send** object or **message** box.

<number>: not a symbol
Occurs when the element that follows a semicolon in a **message** box (specifying a receiver for the message) is not a symbol.

<filename>: bad magic number
<filename>: corrupt binary format file
The file you tried to open is corrupted or is not a properly formatted Max document. Restore the file from a backup copy if available.

<filename>: error creating file
There was an error writing a file; the disk may be write-protected or full

<filename>: out of memory writing file
There is insufficient memory to write the file you're trying to save. If possible, close other files and windows that don't relate to the file you're saving.

ad: Floating point exceptions were caught <number of exceptions>
(Windows only) This message is sent to the max window when audio is stopped if floating point exceptions were caught while processing audio. The tells you how many exceptions were caught to give you an indication of the severity of the problem. This can be triggered by underflow of floating point operations causing denormal numbers to be generated. You may want to try modifying your patcher to cause the exceptions to stop as it may impact the performance.

admme: unable to open wave input device.
admme: unable to open wave output device.
admme: unable to start output.
admme: unable to start input.
ad_mme: stopping due to error.
ad_mme: No MME input or output devices found.
(Windows only) Please check that you have the latest driver update for your audio device. Please exit all other audio applications, reboot if necessary, and try again. Also, please check your settings in the DSP Status window to insure appropriate choices are selected for Input Device, Output Device, Sampling Rate, IO Vector Size, and Signal Vector Size. If the problem persists, contact Cycling '74 support.

ad_directsound: can't create directsound object.
ad_directsound: can't create directsoundcapture object.
ad_directsound: Failed to set cooperative level to priority.

ad_directsound: Failed to create primary buffer.
ad_directsound: Failed to set format of primary buffer.
ad_directsound: failed to create output DirectSoundBuffer.
ad_ds: No directsound input or output devices found.
ad_directsound: unable to Play output buffer.
ad_directsound: unable to Start input buffer.
ad_directsound: stopping due to error.
(Windows only) Please check that you have the latest driver update for your audio device. Please exit all other audio applications, reboot if necessary, and try again. Also, please check your settings in the DSP Status window to insure appropriate choices are selected for Input Device, Output Device, Sampling Rate, IO Vector Size, and Signal Vector Size. If the problem persists, contact Cycling '74 support.

ASIOCreateBuffers error
(Windows only) A problem was encountered initializing the ASIO device. Please check that you have the latest driver update from your audio device manufacturer. Please also try different settings for the device buffer sizes and latency in the control panel for your audio device provided by your device manufacturer. Check that another audio application is not using the audio device. Also check that the audio device is not the default audio device for Windows System Sounds.

bad message
Same as <objectname>: doesn't understand <message selector>.

bad receiver
Same as <objectname>: doesn't understand <message selector>.

bag | float | int | pack | table: missing or incorrect arguments to send
Occurs when the patch is running and a **bag**, **float**, **int**, **pack**, or **table** object receives a send message without an argument, or with an argument that is not a symbol or is not the name of an existing **receive** object.

can't connect <objectname> to <objectname>
Advisory message produced when you try to connect an outlet to an inlet that doesn't understand the message sent by the outlet. You will also notice that the inlet was not highlighted when you dragged the mouse over it.

can't fragload <objectname>: missing <libraryname>, err <number>
An external object that depends upon a particular shared library was not loaded because the shared library is not available. You'll see this error if you try to use an object for MSP with the non- MSP version of Max (the missing library will be called MaxAudioLib in this case) or if you try to use some external objects created for an earlier version of Max/MSP

(e.g., attempting to load an OS 9 external in an OS X version of Max/MSP).Otherwise, to solve this problem, you may need to relocate the shared library or update your system.

check failed: t_newptr in overdrive

This message occurs when an object attempts to allocate too much memory at interrupt level. Unless it represents a bug in the object, it may mean that you'll have to modify your patch to use a defer object where memory is being allocated. One example would be attempting to store large lists of data in a coll object. See the defer object page in the Max Reference manual for more details.

check failed: <message>

Occurs when there is a bug in the Max application or in an external object. Please report the con- tents and context of any such message to Cycling '74.

could not load QuickTime function:

(Windows only) A necessary QuickTime function was not found. Make sure you have installed QuickTime for Windows and chosen a complete install of all optional components.

Error loading external file <filename>

Occurs when Max is installing an external object in the startup folder. The external object file is damaged. Try restoring a copy from the original disk.

funbuff: bad file type

funbuff: file not found

Occurs when a patch is loaded or when a **funbuff** object is created that reads in from a separate file. There was an error in reading a file into a **funbuff**, either because the file was not in the proper format (it must start with the word funbuff, followed by a space-separated list of numbers) or because a Max or text file with that name could not be found. Ensure that the file is in located in Max's search path, and that it is in the proper format.

grab: can only connect to leftmost inlet

Occurs when you try to connect the right outlet of a **grab** object to the wrong inlet of another object. The right outlet of **grab** should be connected only to the leftmost inlet of other objects.

graphic: <name> already exists

Occurs when you create a **graphic** object with a name that has already been taken by another object, such as a **table** or **send/receive** pair.

inlet: wrong message or type
Occurs when a patch is running and an object receives a message that it doesn't expect in some inlet other than the left inlet.

midi_mme: unable to open midi input device
midi_mme: unable to open output device
(Windows only) Max was unable to open the midi input or output device. Please exit from all other midi applications and try again.

MSP/ASIO: Unexpected error loading driver
MSP/ASIO: error loading ASIO driver for
MSP ASIO: Error loading driver
(Windows only) A problem was encountered loading the ASIO driver. Please check that you have the latest driver update from your audio device manufacturer. Check that another audio application is not using the audio device. Also check that the audio device is not the default audio device for Windows System Sounds.

MSP ASIO: : initialization error
MSP/ASIO: : can't deal with bufsize
MSP/ASIO: : data format <format> not supported
MSP/ASIO: : driver start error
(Windows only) A problem was encountered initializing the ASIO device. Please check that you have the latest driver update from your audio device manufacturer. Please also try different settings for the device buffer sizes and latency in the control panel for your audio device provided by your device manufacturer. Check that another audio application is not using the audio device. Also check that the audio device is not the default audio device for Windows System Sounds.

no inspector for <objectname>
The inspector patch for an object that expects to have an inspector cannot be found when you choose **Get Info…** from the Object menu with the object selected. Inspector files are normally in a folder called inspectors within the patches folder, and their names are of the form <objectname>-insp.pat. But they can be located anywhere in the search path as long as the name is properly constructed.

no resource <filename>
This error occurs when you are testing a standalone application and the Search for Missing Files option has been turned off. The named object or file has not been included in the collective from which the standalone was created, and since the runtime Max is not going to look for the file, it declares it missing after it was not found inside the standalone as a resource.

not enough memory to open <filename>
<filename>: can't load, out of memory
The file is too large to be opened. Note that to open a patcher file you need more memory than would be required to actually use the file.

object box has comma or semicolon:
Indicates that you typed a comma or a semicolon character into an object box. If this error occurs when reading in a patch, it's like that the file is damaged.

offscreen buffer couldn't be allocated
Insufficient memory available when working with objects in a graphic window

patcher: unknown script keyword <keyword>
A keyword argument to the script message to sent to the thispatcher object is not recognized.

patcher connect: inlet <number> out of range
Occurs when editing the name or arguments of an object that has already been created in a patcher, and patch cords that used to be connected to the object can no longer be connected. Changing the contents of the object box may change an object's number of inlets or outlets, or Max may be unable to create the object at all if you type in the wrong thing.

QT images not supported in 8 bit color mode
(Windows only) You are trying to load a QuickTime image but your display resolution is set to 8- bit color depth. Change your Display Settings to increase your Color Depth, preferably to 24 or 32 bits.

read failed
Occurs when a file is read into an object. Max encountered an error reading a file and could not load in the data. Check to make sure that the file is in the proper format for the object reading it in.

rescopy: failed to add XXXX N, error N
Occurs when installing an external object. This message (especially if you see a lot of them) may indicate a problem with the Max Temp file used to store resources for external objects. If you only see one or two of these errors, it may be a resource missing in the object or a conflict between two or more objects attempting to use the same ID number. If XXXX is STR#, this problem only affects the strings shown when getting assistance on an object and should not be considered a major problem.

rescopy: failed to get <resource type> <ID number>
Occurs when installing an external object. The external object file is corrupted. Restore a new copy of the external object from your original disk.

script: <keyword>: variable <variablename> empty
Occurs when a script message to the **thispatcher** object references a variable that is no longer assigned to an object.

script: <keyword>: no variable <variablename>
Occurs when a script message to the **thispatcher** object references a variable that has not yet been defined or given a value.

script: instance <number> of <objectname> not found
Occurs when using the nth script message to the **thispatcher** object and the specified index is greater than the number of objects of the specified class in the patcher.

script: name <variablename> already in use
Occurs when a script message to the **thispatcher** object attempts to assign an object to a variable name that is already been used. This error will not occur if you choose **Name…** window from the Object menu to assign a name to an object.

send: <symbol>: already exists
receive: <symbol>: already exists
Occurs when you type in a name as an argument to a **send** or **send** receive which is already being used for a **table** or other object.

sxformat: illegal type in message
Occurs when the patch is running and some message other than an int is received in the inlet of **sxformat**.

text: <filename>: file is protected
You've tried to open a Max binary patcher file protected against editing as text

textbox: bad args
Occurs when opening a Max document. The document has been damaged.

Unable to load MaxQuickTime.dll. Error code.
(Windows only) A required component of Max was missing. Try reinstalling to see if it fixes the problem. If not please contact Cycling '74 support and provide the message and that was reported.

warning: extra arguments for message
Occurs when an object is given more typed-in arguments than it expects, or when too many arguments are present in an incoming message. Usually this is just a warning of something that's not quite right but is basically harmless.

warning: <objectname>: no port <symbol>, using <default port>
Occurs when a port argument is typed into the object box of a MIDI object, and the port name is not currently valid. The valid port names are listed in the MIDI Setup dialog box. The default port is the first name in the device list in the MIDI Setup dialog.

Error Dialogs

When an error occurs that requires your immediate attention, the error is reported in a dialog box. The following errors can appear in dialogs.

Choose Resume from the Edit menu to restart the Max scheduler…
It is possible to get Max working so hard it doesn't have time to respond to your commands (say, if you have a number of **metro** objects sending out bang messages as fast as they can, or if you have created a loop that overloads Max, causing a Stack Overflow error). Holding down the Command key on Windows or the Control key on Windows and typing a period will stop Max's scheduler, giving you time to turn off some of the overloading processes. When Max's timer is stopped, the above message is shown in a dialog box. Choose **Resume** from the Edit menu to restart Max's timer.

No help available for <objectname>.
A help file in the max-help folder can't be located for the named object. Restore the help file from your original Max disks.

Stack Overflow
Occurs when an object's output is being fed back into its inlet in some type of loop. After stopping the process that is causing the stack overflow, choose **Resume** from the Edit menu to restart Max's scheduler.

See Also

Debugging	Techniques for debugging patches

When Max Looks for a File...

Max may look for a file at several different times. Here are some examples:

- When you open a patcher that contains an external object that has not been used yet, Max will search for the file that corresponds to that object.
- When you open a patcher that contains a subpatcher that is a file (i.e., it doesn't begin with the word “patcher” or “p”), Max will search for the patcher.
- When you send the message read with a file name argument to an object such as **table** or **coll**, Max will search for the named file.

Here is how Max searches for files.

The first place Max looks for a file is the *default location*. If you have selected a patcher file from an Open File dialog, the default location will be the folder containing that file. If you have not loaded a file yet, the default location will be the folder containing the Max application. The default location changes dynamically, as you open files in Max. The default location is only a single folder—if you open a patcher file from folder A, subfolders of A are not searched.

The next place Max looks for a file is what we call the *search path*. The search path is partially configured using the File Preferences window. The search path includes files inside the folder containing the Max application you are currently using, as well as the entire contents of the Cycling ’74 folder, located at /Library/Application Support/Cycling ’74 on Macintosh and C:\Program Files\Common Files\Cycling ’74 on Windows. The folders inside the Max application folder are searched before those in the Cycling ’74 folder.

The standard Max installation contains patcher and data files inside folders within the Max application folder, and external objects inside the Cycling ’74 folder. See below for more details about sub-folders of the Cycling ’74 folder.

More Details About Searching:

- When a folder is listed for searching in the File Preferences window, all subfolders of that folder are added to the search path as well.

- Max searches for files in a depth-first order—if there is an entry called patches in the search path followed by one called examples, Max will search all the subfolders of patches before it looks at examples.

Speeding up file searches

It is possible that your max search path can contain many files. When max searches for a file generally it looks for a file from a set of file types such as audio files, images, or max patcher files. For example, a search may be done for an image file named "my_bg_image". If an exact match is not found for the base file name then an extended search is done looking for the base name with matching file extensions such as '.pct', '.jpg', etc.

To speed up such searches a cache is used to speed up file searches by building a representation of the search path in memory. If you prefer to use this memory for other purposes you can disable the file cache by removing the file "pathcacheenabler.txt" from the init folder in the Cycling '74 folder.

What's in the Cycling '74 folder

You can add folders to the Cycling '74 folder and they will automatically be included in the search path the next time you launch Max. But some folders have specific names that cannot be changed without affecting the operation of the software.

The *max-startup* folder contains external objects that are loaded at startup. You can add anything to this folder, including patchers, that you want loaded at startup. Note that the contents of any folders inside the max-startup folder will not be loaded at startup. Generally, the max-startup folder contains user interface external objects that need to be shown in the patcher window's palette.

The *init* folder also executes all of its items at startup, but it is generally not used for external objects. Instead, it contains text files that configure how Max works. You can add additional items to the init folder, but you shouldn't modify the existing files unless you know what you are doing. The init folder is handled before the max-startup folder, and it is also used with the runtime version. The max-startup folder is not loaded by the runtime version.

The *ad* folder, included with MSP, contains audio driver objects. See the MSP documentation for more information.

The *mididrivers* folder contains one or more MIDI driver objects.

The *externals* folder contains all of the external objects not in any of the other folder. It can be renamed if desired. Installers for collections of external objects, such as Cycling '74's Jitter, may install additional folders inside the Cycling '74 folder, or place folders inside the externals folder.

File Path Syntax

A *file path* is a way to specify the location of a file. You're probably familiar with these specifiers in URLs used in web browsers. Here's an example:

http://www.cycling74.com/products/dlmaxmsp.html

This specifies that the file *dlmaxmsp.html* is inside the *products* folder which is inside the root level of the Cycling '74 web site.

In a similar way, you may want to tell Max about a file or folder location on your hard drive. Max has several options for specifying file locations. First, you can choose to use the cross-platform slash (/), the Macintosh-native colon (:), or the Windows-native backslash (\) to separate folder names. However, the backslash is used in Max as an escape character and may lead to unexpected behavior, so we encourage you to use the slash.

Here are some acceptable examples of file locations:

C:/MaxFolder/extras/mystuff/mypatcher.pat (cross-platform, using slashes)

Disk:MaxFolder:extras:mystuff:mypatcher.pat (Macintosh-specific using colons)

C:\MaxFolder\extras\mystuff\mypatcher.pat (Windows-specific, using backslashes)

Versions prior to 4.3 on the Macintosh used colons for separating path elements.

Max objects that accept paths as input will recognize slashes, colons, and backslashes, but they will generally output file paths using the cross-platform pathstyle. The **conformpath** object can be used to convert among different path location conventions.

In addition to the choice of separator characters, you can choose to specify a file or folder's location with:

- an *absolute* path, starting with a hard disk name as shown above

- a path *relative to the Max application*, starting with a ./(cross platform), : (Macintosh), or .\ (Windows), for example: ./patches is the patches folder inside the Max application folder

- a path *relative to the Cycling '74 folder*, starting with c74:, for example, c74:externals/buddy.mxe

- a path *starting with the boot volume*, starting with ^ (Macintosh, for use with the colon syntax), / (cross-platform), or \ (Windows). For example /Documents/mystuff/mypatcher.exe

- a file anywhere in the search path or default folder, which contains no path separator characters at all

File Types and Filename Extensions

On Windows, Max uses *filename extensions*—a period, followed by a series of letters—as the basis of the way the application knows what file format is associated with a given application. If a file does not have an extension, Max can look at its contents and to try to determine what kind of file format it might be. We refer to this as "sniffing" the file.

On Macintosh, Max classifies files first by checking their Mac OS-specific file type information. In the absence of such information, Max looks at the file's extension. If neither of these are definitive, it will "sniff" the file to try to determine its format.

Cross-platform Filename Extensions

The following filename extensions are recognized on both Macintosh and Windows:

Extension *Description*

.pat	Max patch file (in either Max binary or text format file)
.help	Max help file (in either Max binary or text format file)
.txt	Generic text file (Max patcher in text form or other Max text file)
.mxb	Max binary patcher format
.mxt	Max text format patcher file
.mxf	Cross-platform Max collective format

Windows-only Filename Extensions

The following filename extension is recognized on Windows only:

Extension	*Description*
.mxe	Windows-only external object

Macintosh-only Filename Extensions

The following filename extensions are recognized on Macintosh:

Extension	*Description*
.mxd	Macintosh-only external object
.mxc	Old format (pre-version 4.3) Macintosh-only collective

Mapping Filename Extensions to File Types

A *file type* (or *file format*) is a description of how information in a file is arranged. For example, different audio file formats, such as WAV, are specified so that different applications can read and write sound data. The file format tells the application where to expect the sound data, the sampling rate, and other information. Just about every application you use will store information in one or more file types or formats.

File extensions are associated with file types by using the fileformat message to max. The standard set of associations, which you can modify if you want, is found in a file called max-fileformats.txt in the init folder inside the Cycling '74 folder. A second file for audio file types, audio-fileformats.txt, is also present if MSP is installed.

These files consists of a series of messages that take the following form

```
max fileformat <extension> <filetype> [<description>];
```

For example, the entry max fileformat .txt TEXT; tells Max that files ending with .txt are text files. These four-character filetype codes were originally used as Mac OS type information, but they are used internally by Max on all platforms to specify file formats.

The <description> in square brackets above is optional and is used to provide a description on Windows XP that is displayed in the file open and save dialog boxes. For example, rather than displaying maxb to the user it could display Max Patcher.

Here is a list of the standard associations between filename extensions and file types.

Extension	*File Type*	*Description*
.pat	maxb, TEXT	Max patch file (in either Max binary or text format file)
.help	maxb, TEXT	Max help file (in either Max binary or text format file)
.txt	TEXT	Text file (Max patcher in text form or other Max text file)
.mxb	maxb	Cross-platform Max binary patcher format
.mxt	TEXT	Cross-platform Max text format patcher file
.mxf	mx@c	Cross-platform Max collective format
.mxd	iLaF	Macintosh-only external object
.mxc	maxc	Old format (pre-version 4.3) Macintosh-only collective

Note: Case is important: MAXB is an entirely different format than maxb. An extension can be associated with more than one file type. In that case, Max will have to look at the file's contents to see if it can determine the type. For example, files ending in .pat could be either text or Max binary format. Historically, on Macintosh, OS-specific file type information determined the nature of the file, but when this is absent, the extension alone is ambiguous.

However, most file extensions are not ambiguous, and map one-to-one to file formats.

Mapping file types and file extensions are important in several situations when working with Max:

- Files with designated extensions that do not have type information in them will show up in the Open File dialog where they might not otherwise appear. For example, if you have an object that opens sound files of type AIFF. Without a file extension mapping, files without AIFF type information stored in them would not appear in the Open File dialog. If you map the extension .aiff to this type, a file with a name like sound.aiff would be one that you could select.

- Files can be found without requiring the extension as part of the name. For example, assuming that .pat file extensions have been mapped to Max binary file formats (maxb). You can type foo into an object box to load a patcher file, and if there is a file

in the search path called foo.pat, it will be loaded. Note, however, that if there were a file called foo anywhere in the search path with the proper Macintosh type information or, on all platforms, was actually the right type of file (after Max examined its contents), it would be found before foo.pat. This is because Max goes through its entire search path looking for an exact match before it tries to match based on filename extensions.

To associate a file name extension with a file type, within a patcher you can simply type the message to Max into a message box, preceded with a semicolon (e.g., ; max fileformat .txt TEXT.). This may be useful if you don't want a file name extension and file type to be automatically associated every time you launch Max.

External Object Name Mappings

There are other files in the init folder used to specify mappings between object names and file names. On some operating systems, it is not possible to use certain characters in filenames. However Max, has traditionally had object names with some of these characters in them. In order to avoid problems with these objects, a mapping between an object name and a filename can be established using the objectfile message to the max object. For example:

```
max objectfile !- rminus;
```

specifies that when you type !- into an object box, Max will look for an external object file called rminus. In addition, when you ask for help on the **!-** object, Max will look for a help file called rminus.help, not one called !-.help.

Max-specific mappings are found in a file called max-objectmappings.txt. MSP-specific mappings are in a file called audio-objectmappings.txt.

As with the fileformat message to max, the objectfile message can be sent within a message box. But placing it in a file in the init folder ensures that mappings are available each time you use Max.

Developers of third-party external objects can add their own files to the init folder with object name mapping messages in them.

Graphics: Overview of Graphics Windows and Objects

Introduction

Max has several objects for color graphics and animation. These objects use the same principles as objects that are used for music processing, so a single patcher can combine user interface, music, and graphics functions. This allows you to experiment with various ways of combining and synchronizing sound and image.

There are three ways you can present graphics in Max: in a Patcher window, in a QuickTime movie window, or in a special *graphics window*. Most graphics objects draw within special graphics windows, associated with a **graphic** object. Each **graphic** object is given a name, and each object that draws something must have the name of a graphics window as an argument.

When Max is in Overdrive mode, objects that draw graphics are de-prioritized, so that a process that both plays music and displays graphics can run at any speed and the music will not be affected by the speed of the display. For example, if an animation would ordinarily fall behind the music, the animation automatically skips frames to keep up. (User interface objects such as **slider** objects do this too, by the way.)

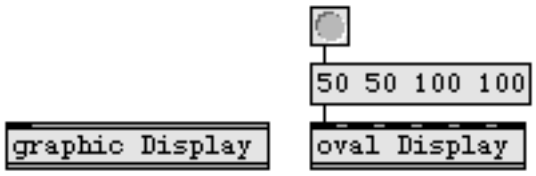
All the objects that draw graphics are external objects, and additional graphics objects can be written by C programmers. Each object that draws in a graphics window is a *sprite* associated with a particular window. Sprites allow objects to pass in front of or behind each other according to a *priority number*. Higher-numbered sprites are drawn in front of lower numbered sprites. The priority number of these graphics objects can be changed with the priority message.

Graphics In a Graphics Window

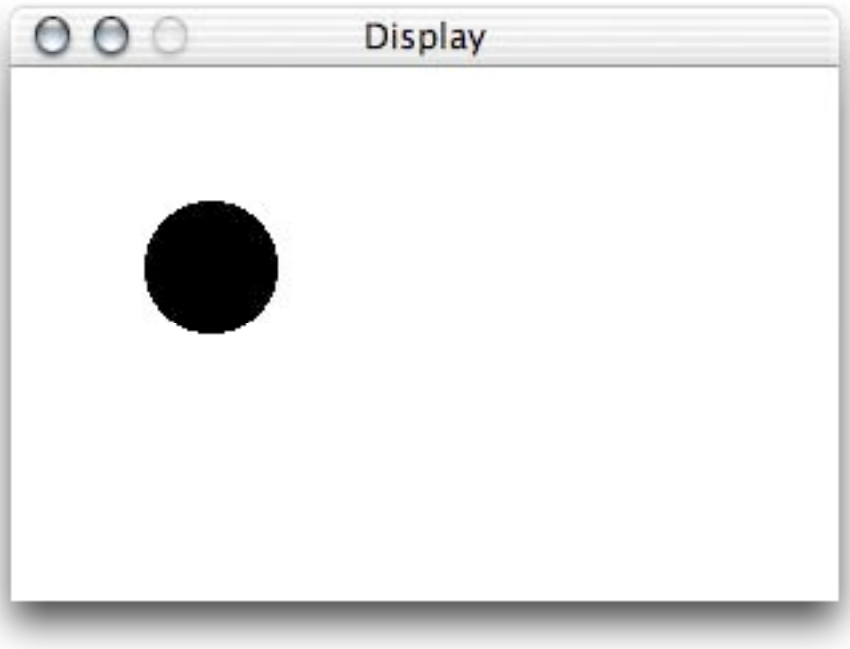
You need a **graphic** object in your patch to open a graphics window. Once you have a patch containing a **graphic** object, you need one or more drawing objects. There are three basic objects included with Max for drawing in a graphics window: members of the **oval** family (**oval**, **rect**, **ring**, and **frame**) which draw shapes and **pict**, which displays PICT files. The first argument of any drawing object is the name of the **graphic** object whose window will be used for drawing. The **graphic** object need not exist at the time the drawing object is created, but the drawing object will do nothing until there is a valid (and visible) graphics window with the same name specified as drawing object's argument.

Here is a simple patch that draws a black oval within the rectangular pixel area 50,50,100,100 in the graphics window titled *Display* when the user clicks on the **button**.

The **oval** has six inlets, for left, top, right, and bottom screen location, drawing mode, and color (index into the graphics window's palette). A list of four numbers sent to an **oval** object causes it to draw within the pixel coordinates specified in the list: left, top, right, and bottom.



A graphics window titled *Display* appears when the **graphic** object is created. Below you can see the result of clicking on the **button** in this patch.

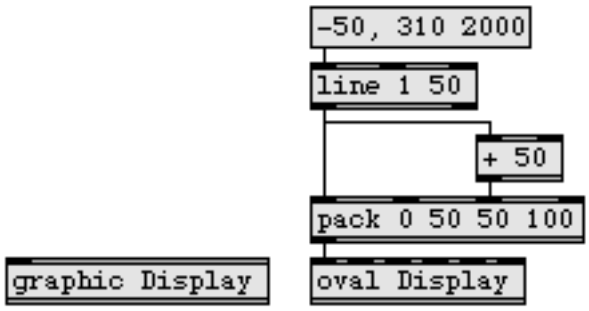


If the oval is redrawn with different coordinates, the old oval is erased automatically, because the oval acts as a sprite which has changed location (and possibly size).

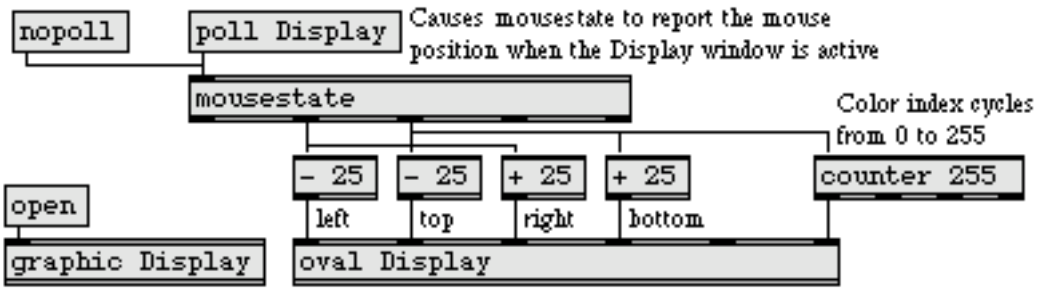
Ways to Move Objects

Two useful objects which let you move a sprite-based object such as an **oval** across the screen are **line** and **mousestate**. **line** can move the sprite smoothly in a trajectory, while **mousestate** can be used to make a sprite follow the mouse.

Here is a program that uses **line** to move an **oval** from one side of the window to the other. Notice that only the left and right coordinates are being changed by the output of the **line** object.



The next example uses **mousestate** to follow the mouse. When **mousestate** receives the poll message with the name of a **graphic** object as an argument, it will begin polling the mouse when the associated graphics window becomes the active window (or, if **All Windows Active** is enabled in the Options menu, it polls all the time). The local coordinates of the mouse in the graphics window are sent out the second (horizontal) and third (vertical) outlets of **mousestate**.This patch draws an oval, centered at the mouse location, which changes color each time the mouse is moved.

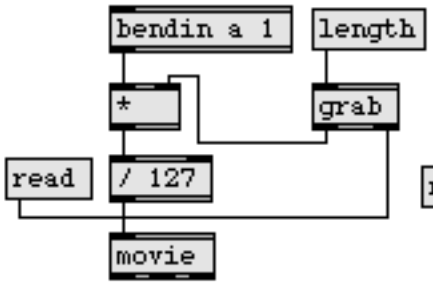


Any picture that is saved as a PICT file can be displayed in a graphics window with the **pict** object. The PICT is displayed in the graphics window at full size, and the location of its upper left corner is determined by the numbers received in the second and third inlets of the pict object. (So in most cases you'll want the area of your PICT to be only large enough to contain the image you want to display, with no extraneous white space around it.) Its placement in the window and its sprite priority can be controlled similarly to the geometric shapes described above.

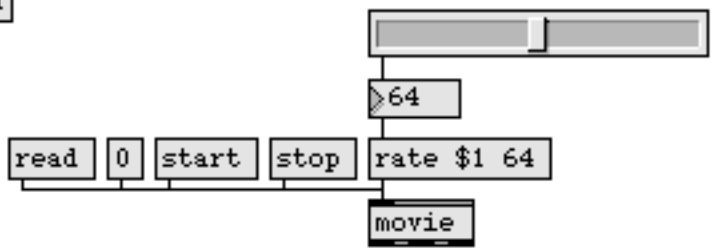
QuickTime Movies

If you have QuickTime installed on your computer, you can play QuickTime movies in Max using the objects **movie** and **imovie**. The two objects function very similarly. The **movie** object displays the movie in a window of its own, and **imovie** is a user interface object which displays the movie in a box inside a Patcher window.

The standard QuickTime controls are available in the form of a separate user interface object called **playbar**, the outlet of which is to be connected to the inlet of a **movie** or **imovie** object. You can also control **movie** and **imovie** directly with messages such as start and stop, you can change its speed with the rate message, you can jump to any frame in the movie immediately simply by specifying its location in the movie, and you can shuttle back and forth with the prev and next messages.



Using the pitchbend wheel to shuttle back and forth through the movie



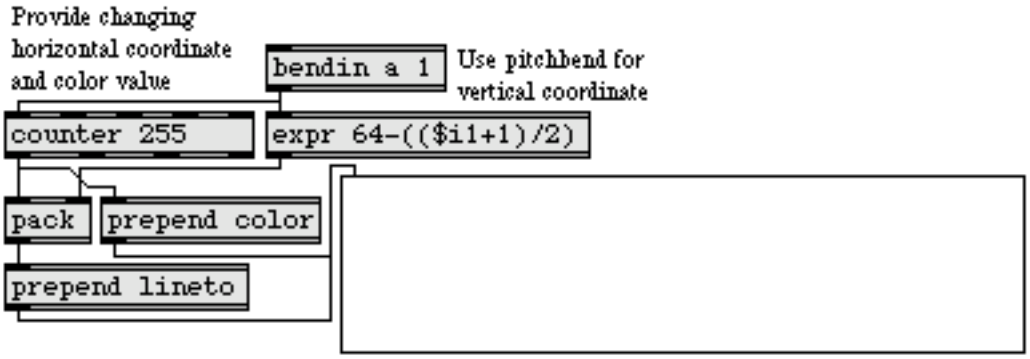
Using **hslider** to control the speed of the movie

Graphics in a Patcher Window

There are a number of ways to design the graphic appearance of a Patcher window. You can copy a picture in PICT format from another application and place it in your Patcher by choosing the **Paste Picture** command from the Edit menu in Max. You can also use the **fpic** object to load a separate PICT file into your patch. Using pictures—in combination with the transparent button object **ubutton**—and the various graphical user interface objects provided, you can give your user interface any appearance you wish. You can even change the appearance of a patcher automatically while it's running by sending an offset message to a **bpatcher** object, thus displaying a different portion of its embedded patch.

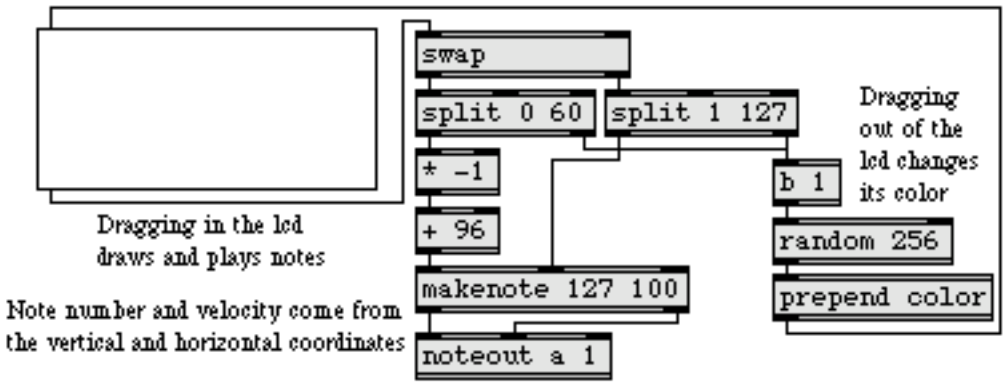
The **imovie** and **lcd** objects allow you to build animation and drawing capabilities right into your Patcher window. The **lcd** object understands messages similar to basic QuickDraw

commands, such as moveto, lineto, paintoval, paintrect, frameoval, framerect, etc., so you can write a patch that paints automatically directly into its own Patcher window. The messages move and moveto are used to place the cursor, the messages color and penmode govern the way pixels will be painted, and the other commands draw lines, shapes, or letters in the **lcd** object.



Drawing with the pitchbend wheel in the lcd

The **lcd** object also responds to mouse movements in the manner of a color painting program. When the user clicks or drags within it, **lcd** draws using the selected color (based on the most recently received color message), and also reports the coordinates of the mouse—with respect to the upper left corner of the **lcd**—out its outlet. Thus, the drawing motions can be used to generate music, as well, as demonstrated in the following example.



Drawing in lcd to play notes with the mouse

The **imovie** object lets you embed a QuickTime movie directly into your Patcher. It displays the movie in the same way as the **movie** object (see QuickTime Movies above), and reports the mouse location whenever the mouse is clicked within it.

See Also

graphic	Open a graphics window
imovie	Play a QuickTime movie in a Patcher window
lcd	Draw QuickDraw graphics in a Patcher window
mousestate	Report the status and location of the mouse
movie	Play a QuickTime movie in a window
oval	Draw solid oval in graphics window
pict	Draw picture in graphics window
Tutorial 42	Graphics
Tutorial 43	Graphics in a patcher

Getting the Picture

The **pictctrl**, **pictslider**, and **matrixctrl** objects are user-interface object for creating buttons, sliders, switches, knobs, and other controls. These objects can open PICT files and, if QuickTime is installed, other picture file formats that are listed in the QuickTime appendix found in the Max Reference Manual. Since the these objects use images from picture files for their appearance, you can create these files using any graphics program (such as Photoshop™ or Canvas™) with whatever appearance you desire

Each picture-based control expects the picture file to be in a particular layout. The layouts vary somewhat depending on the control, but they have some common characteristics:

- Each picture file contains a rectangular array of one or more images. Each image represents one *state* of the control. The state of a control includes its current value, whether the user is clicking it with the mouse, and so on. At any given time, the user sees only one image from the array contained in the picture file.

- All images in the array are the same size. This size may correspond to the size of the object as it appears in the Max patcher, or the object may alter the image's size.

- Some parts of the array are optional. For example, the controls can optionally display a different image when the user clicks them. You do not need to create blank images in the array for optional images that your control doesn't use. Just leave the row or column out of the array altogether.

- The manner in which the control chooses what portions of the picture file to display is determined by the object's attributes that you set with its Inspector and by the overall dimensions of the picture in the file.

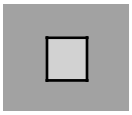
Picture File Construction

The easiest way to understand how picture files must be constructed, and how the corresponding object attributes must be set, is to look at some examples.

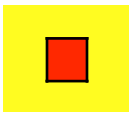
We'll look at several examples using **pictctrl**. The **pictctrl** objects use only one picture file, so it's the simplest to work with.

A simple button control has only two states: either the user is clicking on it, or not. Thus, a **pictctrl** being used as a button needs a picture file with two distinct images—one for the clicked state, and one for the idle state.

Our **pictctrl**-based button will look like this when it is idle:

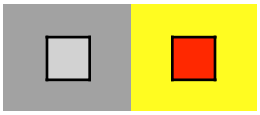


and look like this when it is clicked:



Yes, it's just a boring grey rectangle with a square inside it, which turns yellow and red when you click it.

The picture file for this **pictctrl** would look like this:



The image for the idle state is on the left, and the image for the clicked state is on the right. The appropriate image is shown based on the state of the control, and the other image is hidden. We've included this file, called *boring button.pct*, within the *picts* folder inside the *patches* folder. By default this folder is in the Max search path.

To use this picture in a Max patcher, you would add a new **pictctrl** object to your patcher and then choose **Get Info...** from the Object menu to open the object's Inspector. Click the *Open...* button near the bottom of the **pictctrl** Inspector to choose this picture file. That's all you have to do, since the default mode of **pictctrl** is button mode.

Making Toggles

Next we'll look at a picture file for a **pictctrl** that uses the toggle mode. The **pictctrl** object's toggle mode emulates "push-on push-off" buttons found on some hardware: you push and release them once to turn something on, and push and release them again to turn the same thing off again. They "toggle" between two states, off and on. In a more general sense, they toggle between two values, zero and one. The standard checkbox you're used to using in dialog boxes works this way too.

Since the control can have two values, and the mouse button can either be idle or clicked, the **pictctrl** object's toggle mode has four states. We might draw a chart to represent these four states:

		Mouse Button: Idle	Mouse Button: Clicked
Control Value	0	Idle 0	Clicked 0
	1	Idle 1	Clicked 1

Each of the four quadrants in the chart represents one state of the control—a combination of its current value and the position of the mouse button.

This chart is arranged the same as the layout required for picture files for the toggle mode of the **pictctrl** object. The picture is divided into four equal-sized quadrants, each of which contains the image displayed for the corresponding state of the control.

Here's an example picture which implements a toggle-mode **pictctrl** that resembles the pushbuttons with embedded lights found on some hardware synthesizers:

The images in the left column will be used to draw the control when it is idle, and the images on the right will be used when the user is clicking the control with the mouse. The top row of images will be used when the control's value is zero, and the lower row will be used when the control's value is one. So, for example, the upper-right image will be displayed when the control's value is zero and the user is clicking it.

This picture is in the file *LED button.pct* in the *picts* folder. To use it in a control, add a new **pictctrl** to your Max patcher and set its mode to *Toggle* by clicking the radio button near the top of its Inspector. Notice that the control doesn't display the correct portion of the picture until after you've set its mode. This is because **pictctrl** uses different regions of the picture file based on which mode it's using, and how you have the various properties set, using the checkboxes in the Inspector.

Inactive States

Controls created with **pictctrl** can have a separate set of images for their inactive state. You can use these images to indicate that the control won't respond to mouse clicks, similar to how the Macintosh "greys out" inactive controls.

In pictures for **pictctrl**, the inactive images appear below the regular images. In the following picture (found in the file *LED button w/ inactive.pct*, we've added inactive images to our light-up button:

The images are blurred to indicate that the control is inactive. Notice that there are two inactive images, one for the control when its value is zero and one for when the value is one. To use this picture with the **pictctrl** object's toggle mode, you would check *Has Inactive Images* in the Inspector.

Image Masks

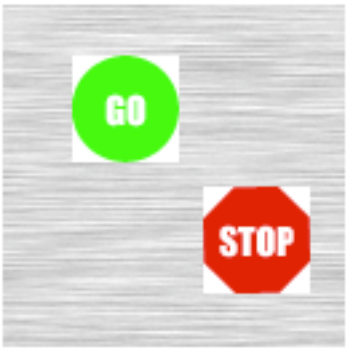
All Max objects have a rectangular "bounding box" which defines their location and size. You can create controls that have a non-rectangular appearance by using a feature called *masks*. Masks are special images within picture files that define which portions of the images are visible, and which portions are transparent or invisible. Black pixels in the image define visible areas, and white pixels define transparent areas.

The following illustration shows how rectangular images and masks combine to create a non-rectangular image:

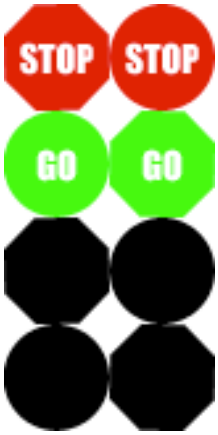
To demonstrate a Max control that uses masks, we'll create a toggle button that looks like an octagonal STOP sign when its value is zero, and a circular GO sign when its value is one. To add a little visual interest, we'll make its shape—but not its color—change when it is clicked with the mouse. Its picture file looks like this:

This picture is in the file *stop-go.pct*. To try it out, create a new **pictctrl**, open its Inspector, choose the *stop-go.pct* file as its picture file, and set its mode to *Toggle*. After locking the patcher, click the control a few times. Notice that it switches from the red octagon to the red circle first—switching from the value=0, not-clicked image to the value=0, clicked image. When you release the button, it displays the green circle, the value=1, not-clicked image.

Everything looks fine as long as the control is placed only upon a blank, white window. But suppose we want to put the control on top of a colored panel object, or a picture of a faux brushed- aluminum surface. We see the white areas of our images:

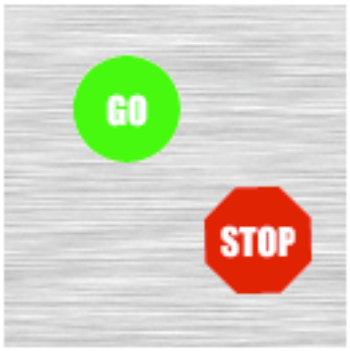


The solution to this aesthetic problem is to use masks to define which parts of our image should be drawn, and which parts should be transparent, allowing whatever is underneath the control to be visible. For our stop/go control, we need masks that have the same outline as the colored areas. This will make the white areas transparent. The picture file with masks added, called *stop-go mask.pct*, looks like this:



The masks are placed below the images, in the same relative positions. (In many cases you can create masks for your images simply by duplicating the images and using a "paint bucket" tool to fill the duplicates with black.) Try this picture by choosing *stop-go mask.pct* as your control's picture file.

Check the *Has Image Mask* checkbox in the Inspector. Now the white areas of the control won't be drawn:



Picture files for the **pictslider** and **matrixctrl** objects are constructed in much the same manner. Refer to **pictslider** and **matrixctrl** manual pages in the Max Reference Manual for the layouts needed for arranging their images. Remember that not all of the images in the layout charts are necessary, so that you can start by working with simple picture files and later add images to create more elaborate controls.

Note: The picture's dimensions must be exactly the size of the array of images, and no more, since the picture-based controls use the overall dimensions of the graphics file to calculate the size of the images. Some graphics applications are known to add a one-pixel wide strip of blank pixels to the lower and right edges of all graphics files it creates, which causes the control images to appear to move slightly when they change state, since the control has been given inaccurate information about the size of the images. If you see this problem, try using another graphics application to create the artwork.

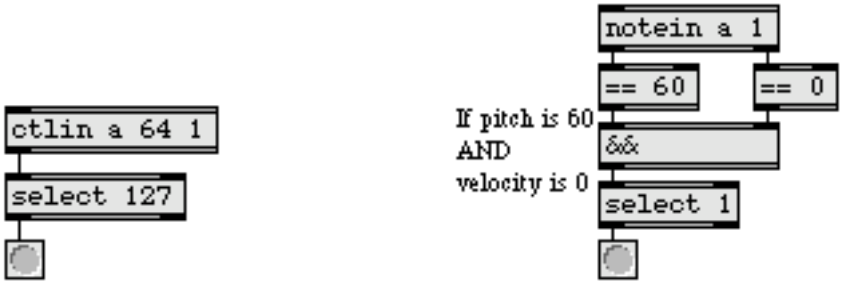
See Also

matrixctrl	Matrix switch control
pictctrl	Picture-based control
pictslider	Picture-based slider control

Repeated Actions

Many aspects of music-making involve repeated actions. For example, we might count 25 measures of rests, hit a gong 4 times in succession, repeat the whole section a total of 3 times, etc. In programming, repeated actions are called loops, because conceptually we think of the computer completing an action, then looping back to the place in its stored program where it started and performing the action again. A loop generally involves some sort of a check before or after each repetition, to see whether the action should be performed again (without the check, the process would continue endlessly).

In Max, a bang message can be used to signal that an event has occurred. In the example below, a bang is sent each time the sustain pedal (controller number 64) is pressed down, or each time the note Middle C (key number 60) is released.

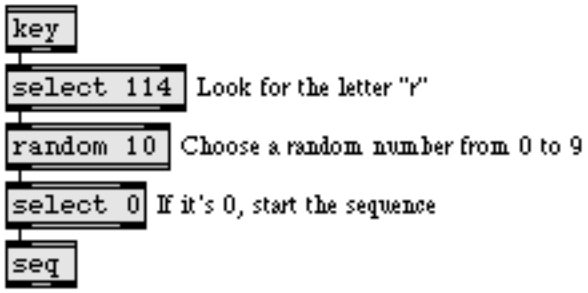


Since bang is the generic indicator that something has happened, there is an object designed to count bang messages, called **counter**. It counts the bang messages it receives in its inlet, and sends the count out its outlet. You can set minimum and maximum output values for **counter**, and set it to count up, down, or up and down. In the following example, **counter** counts from 1 up to 5, then starts again at 1. The right outlet reports how many times the maximum (5) has been reached.

It can be useful just to send out a succession of numbers from a **counter**. For example, the numbers could be used as addresses to get values from a **table**. Other times, in order to make the loop useful, there needs to be a unique event when a certain condition is met. Actually, **counter** has its own built-in conditions and reactions, such as "when the maximum is reached, send the number of times it has been reached out the right outlet,

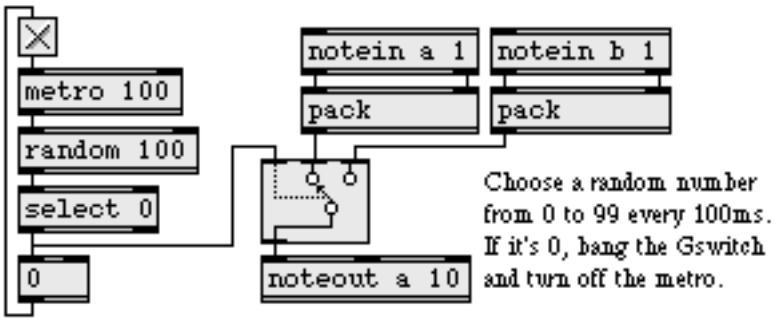
send the number 1 out the right-middle outlet, and go back to the minimum," but sometimes we may want another condition to cause a certain result.

In the example below, a bang is sent to **random** every time the letter r is typed on the computer's keyboard. When **random** produces the number 0, a sequence is played.



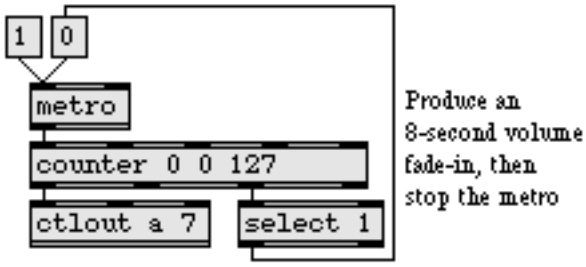
In this case, we aren't counting the number of times something happens (we might have to type the letter r any number of times before a 0 is chosen at random), we're just repeating until a certain condition is met.

When the condition we are testing for is met, something should happen as a result—a **gate** opened, a process started, a note played, etc.—and, if the repetitions are being supplied by a timed process (such as a **metro** sending a bang every 100 milliseconds), the repetitions should be stopped.



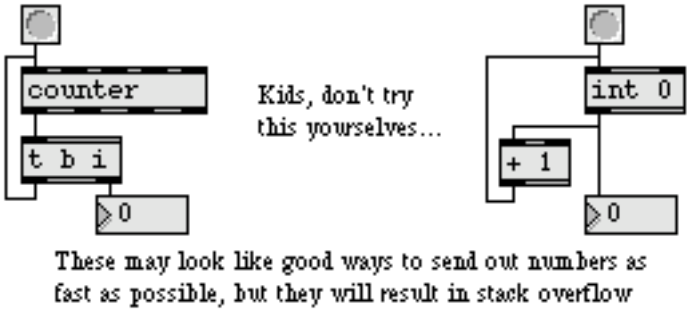
Timed Repetition

Since time is such an important factor in music, you'll want to have repeated actions occur at a specific speed. The **metro**, **clocker**, and **tempo** objects are used for producing output at regular intervals—bang in the case of **metro**, numbers in the case of **clocker** and **tempo**. (Of course, **metro** can also produce a succession of numbers when its output is sent to a **counter**.)



Stack Overflow

Automatic timed repetitions must be separated by at least a millisecond or two, otherwise Max will generate a Stack Overflow, which stops Max's internal scheduler until you shut off the repetitions. Below are a couple of examples of what not to do, because you will cause a stack overflow.



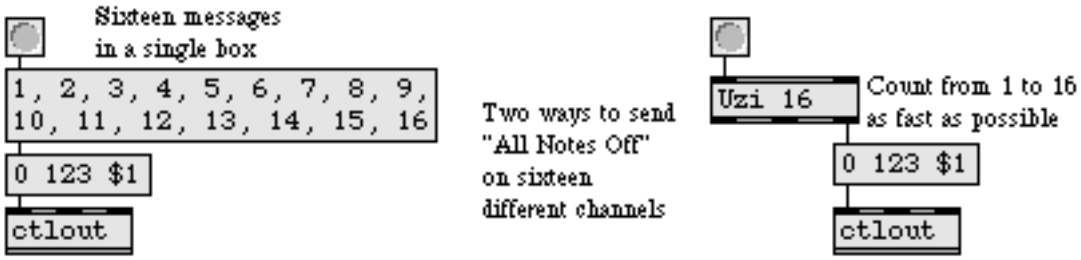
These programs contain bugs!

The patches in this example attempt to increment a count as fast as possible. Each solution has two flaws, however. The first problem is that there is no stopping condition; the numbers will increase indefinitely. The second problem is that each patch feeds an object's output back into its triggering inlet with no time delay. Max keeps trying to do

more and more things, without being given any more time in which to do them, and quickly complains that its "to do" stack is overflowing.

Instantaneous Loops

When you want to use a loop that completes all its repetitions as fast as possible—that is, you want to send out a series of events "at the same time"—you must use an object that is designed to send out multiple messages. The **message** box can contain multiple messages separated by commas, and sends them all out in immediate succession. In the same spirit, the **uzi** object is designed to send out a succession of bang messages as fast as possible. This series of bang messages can be sent to a **counter** object to convert them to a series of numbers. **uzi** itself counts the bang messages as it goes and sends the count out its right outlet, so it can be used to send a sequence of numbers as fast as possible. The following example shows two ways to send sixteen different MIDI messages as fast as possible.



Multiple messages sent in immediate succession

See Also

clocker	Report the elapsed time, at regular intervals
counter	Count the bang messages received, output the count
metro	Output bang, at regular intervals
tempo	Output numbers at a metronomic tempo
uzi	Send a specific number of bang messages

Executable Formats and Processor Architectures

An *executable format* describes how machine code is stored in files on your computer's hard drive. There are two executable formats on the Mac OS, Mach-O and CFM. CFM, which was originally introduced for the PowerPC prior to the introduction of Mac OS X, is no longer supported by Apple, and is not implemented for native Intel applications. Thus, we changed MaxMSP 4.6 to use the Mach-O executable format for both the Max application and all external objects.

The executable format is different from your computer's *processor architecture*. Apple uses the term Universal Binary to refer to Mach-O format files that contain machine code for both the PowerPC and Intel Pentium processors.

MaxMSP 4.6—running on a PowerPC processor *only*—can load objects written for the older CFM format, if you install the Max CFM Adaptor support for it. You are asked when MaxMSP is launched for the first time if you want to install support for older Max objects.

MaxMSP 4.5 can run on an Intel processor and use its older CFM objects (although some objects, such as **mxj**, do not work). However, the limitations of Apple's Rosetta technology are such that if the host application is native Intel, it cannot load plug-ins written only for the PowerPC, even if they use the Mach-O executable format

The chart below summarizes executable format and processor architecture compatibility issues:

Processor	*MaxMSP Version*	*Compatible Object Executable Formats*
Intel	4.6	Universal Binary Mach-O
PowerPC	4.6	Universal Binary Mach-O CFM objects written for PowerPC (with CFM Adaptor libraries installed)
Intel	4.5	Most CFM objects written for PowerPC
PowerPC	4.5	CFM objects written for PowerPC Mach-O objects written for PowerPC (see note below) Mach-O Universal Binary objects with OS 10.3.9 or later

Macintosh Externals

Libraries for Older External Objects

When Max/MSP 4.6 starts up for the first time on the PowerPC machine, it will ask you if you want to install support libraries for older external objects. These libraries allow you to use third-party external objects written for Max/MSP 4.5 or earlier version inside Max/MSP 4.6. These libraries also can enhance audio processing performance in Max/MSP 4.6.

However, these libraries may conflict with the use of older Max/MSP plug-ins (such as those included with Pluggo, Mode, and Hipno as well as certain third-party plug-ins) in other host applications. If you experience problems using plug-ins, you may want to uninstall these support libraries:

- Delete */Library/CFMSupport/MaxCFMAdaptor.shlb*
- Delete */Library/CFMSupport/MaxAudioCFMAdaptor.shlb*
- Look in */Library/Application Support/C74 Plug-In Support* for MaxPlugLibCarbon.shlb, MaxAudioPlugLibCarbon.shlb, and VstPlugLib.shlb. If these files are present, move them to */Library/CFMSupport*

The ; max **message**

Using a message box, you can control the Max application. All such messages begin with; max (as if there were a **receive** object named **max**). Here is a list of messages the Max application understands.

Messages Understood by Max

#SM	see the section **MIDI Configuration Messages** at the end of this chapter.
boxcolor	Sets one of the 15 default object colors in the Color submenu. boxcolor is followed by five arguments. The first is the index (between 1 and 15), followed by three values in the range 0-255 that specify the color for this index, and a final value that specifies whether this change overwrites the user preferences (1) or not (0).
buildcollective	The word buildcollective, followed by a reference name symbol and an output filename, builds a collective using the patcher associated with the symbol. The collective is named with the output filename.
buildplugin	The word buildcollective, followed by a reference name symbol and an output filename, builds a VST plug-in using the patcher associated with the symbol. The plug-in is named with the output filename.
checkpreempt	The word checkpreempt, followed by a symbol, sends the current Overdrive mode to the **receive** object named by the symbol.
clean	Causes Max not to display a Save Changes dialog when you close a window or quit, even if there are windows that have been modified. This is useful in conjunction with the quit message below.
closefile	The word closefile, followed by a symbol, closes the patcher file previously opened with the openfile message to Max associated with the symbol.
debug	The word debug, followed by a zero or one, toggles the sending of Max's internal debugging output to the Max window. Debug information may be of limited use for anyone who isn't debugging Max itself.
enablepathcache	The word enablepathcache, followed by a zero or one, turns on (or off) Max's search path cache. This should only be done if you noticed unusual behavior when opening files.

enablerefresh (Macintosh only) The word enablerefresh, followed by a zero or one, toggles an alternative to the standard way in which the screen contents are updated, resulting in better "visual performance. This feature is enabled by default. The rate at which refresh is done can be set by using the setrefreshrate message.

externs List all of the external objects currently loaded in the Max window.

fileformat The word fileformat, followed by two symbols that specify a file extension and a four-character file type, tells Max to associate a filename extension with a particular filetype. The message max fileformat .tx TEXT associates the extension .tx with TEXT (text) files. This allows a user to send a message read george and locate a file with the name "george.tx" It also ensures that files with the extension .tx will appear in a standard open file dialog where text files can be chosen.

fixwidthratio The word fixwidthratio, followed by a floating-point number, sets the ratio of the box to the width of the text when the user chooses Fix Width from the Object menu. The default value is 1.0. A value of 1.1 would make boxes wider than they needed to be, and a value of 0.9 would make boxes narrower than they need to be.

getboxcolor The word getboxcolor, followed by an number between 1 and 15 and a symbol, sends the RGB values for the default object colors at the specified index as a list to the **receive** object named by the symbol.

getdefaultpatcherheight

The word getdefaultpatcherheight followed by a symbol used as the name of a **receive** object, causes Max to report the current default patcher height in pixels to the named **receive** object (See also the setdefaultpatcherheight message to Max.)

getdefaultpatcherwidth

The word getdefaultpatcherwidth followed by a symbol used as the name of a **receive** object, causes Max to report the current default patcher width in pixels to the named **receive** object (See also the setdefaultpatcherwidth message to Max.)

getenablepathcache

The word getenablepathcache, followed by a symbol used as the name of a **receive** object, will report whether the path cache is enabled to the named **receive** object. (See also the enablepathcache message to Max.)

getenablerefresh (Macintosh only.) The word getenablerefresh, followed by a symbol used as the name of a **receive** object, will report whether enhanced refresh is enabled to the named **receive** object. (See also the enablerefresh message to Max.)

geteventinterval The word geteventinterval, followed by a symbol used as the name of a **receive** object, will report the event interval to the named **receive** object. (See also the seteventinterval message to Max.)

getfixwidthratio The word getfixwidthratio followed by a symbol used as the name of a **receive** object, reports the current fix with ratio value to the named **receive** object. (See also the fixwidthratio message to Max.)

getpollthrottle The word getpollthrottle followed by a symbol used as the name of a **receive** object, reports the current poll throttle value to the named **receive** object. (See also the setpollthrottle message to Max.)

getqueuethrottle The word getqueuethrottle followed by a symbol used as the name of a **receive** object, causes Max to report the current queue throttle value to the named **receive** object. (See also the setqueuethrottle message to Max.)

getrefreshrate (Macintosh only) The word getrefreshrate followed by a symbol used as the name of a **receive** object, causes Max to report the current refresh rate in Hertz to the named **receive** object. (See also the setrefreshrate message to Max.)

getruntime The word getruntime followed by a symbol used as the name of a **receive** object,sends a 1 to the named **receive** object if the current version of Max is a runtime version, and a 0 if not.

getsleep The word getsleep followed by a symbol used as the name of a **receive** object, causes Max to report the sleep time to the named **receive** object. (See also the setsleep message to Max.)

getslop The word getslop followed by a symbol used as the name of a **receive** object, reports the scheduler slop value to the named **receive** object. (See also the setslop message to Max.)

getsystem — The word getsystem, followed by a symbol used as the name of a **receive** object, will report the name of the system (macintosh or windows) to the named **receive** object.

hidecursor — Hides the cursor if it is visible.

hideglobal — Hides the floating inspector window.

hidemenubar — Hides the menu bar. Although the pull-down menus are not available when the menu bar is hidden, menu shortcut (accelerator) keys continue to work.

htmlref — The word htmlref, followed by an object name as a symbol, looks for a file called <object-name>.html in the search path. If found, a web browser is opened to view the page.

interval — The word interval, followed by a number from 1 to 20, sets the timing interval of Max's internal scheduler in milliseconds. The default value is 1. This message only affects the scheduler when Overdrive is on and scheduler in audio interrupt (available with MSP) is off. (When using scheduler in audio interrupt mode the signal vector size determines the scheduler interval.) Larger scheduler intervals can improve CPU efficiency on slower computer models at the expense of timing accuracy.

launchbrowser — The word launchbrowser, followed by a URL as a symbol, opens a web browser to view the URL. For example:
; max launchbrowser http://www.cycling74.com

maxinwmenu — When using the runtime version of Max, maxinwmenu followed by the number 1 will place an item called Status in the Windows menu, allowing users to see the Max window (labeled Status in the runtime version). When maxinwmenu is followed by 0 the menu item is not present. The default is for the Status item to be present in the Windows menu.

midi — The word midi, followed by a variable-length message, allows messages to be sent to configure the system MIDI object. The following is a list of the available options:

autosetup

Duplicates the action of clicking on the Auto Setup button in the MIDI Setup window

portabbrev <*innum* / *outnum*> <*portname*> <*abbrev*>

innum specifies an input port, *outnum* specifies an output port, *portname* is the name of the port as a single symbol (i.e. It is necessary to use double quotes). An *abbrev* value is 0 for no abbrev (- in menu), 1 for 'a' and 26 for 'z'

portenable <*portname*> <*0/1*>

Enables (1) or disables (0) the port specified by *portname*. All ports are enabled by default.

portoffset <*innum / outnum*> <*portname*> <*offset*>

Similar to portabbrev, but *offset* is the channel offset added to identify input or output ports when a MIDI object can send to or receive from multiple ports by channel number. Must be a multiple of 16 (e.g. max midi portoffset innum PortA 16 sets the channel offset for PortA device to 16).

midilist — Prints the names of all current MIDI devices in the Max window. (See also MIDI Messages to Max, above.)

notypeinfo — (Macintosh) The word notypeinfo followed by 0 or 1, sets whether Max saves files with traditional Mac OS four-character type information. By default, Max does save this information in files.

objectfile — The word objectfile followed by two symbols that specify an object name and a file name, creates a mapping between the external object and its filename. For example, the ***~** object is in a file called times~ so at startup Max executes the command max objectfile *~ times~.

openfile — The word openfile followed by two symbols that specify an reference name and a file name or path name, attempts to open the patcher with the specified name. If successful, the patcher is associated with the reference symbol, which can be passed as argument to the buildcollective, buildplugin, and closefile messages to Max. The openfile message is intended for batch plug-in or collective building.

paths — List the current search paths in the Max window. There is a button in the File Preferences window that does this.

preempt — The word preempt, followed by a 1 (on) or 0 (off), toggles Overdrive mode.

pupdate — The word pupdate, followed bytwo integer values that specify horizontal and vertical position, moves the mouse-cursor to that global location.

quit — Quits the Max application; equivalent to choosing Quit from the File menu. If there are unsaved changes to open files, and you haven't sent Max the clean message, Max will ask whether to save changes.

refresh — Causes all Max windows to be updated.

runtime — The word runtime, followed by a zero or one and a message, executes the message if the current version of Max is a runtime version (1) or non-runtime (0).

sendinterval — The word sendinterval, followed by a symbol, sends the current scheduler interval to the **receive** object named by the symbol.

sendapppath — The word sendapppath, followed by a symbol, sends a symbol with the path of the Max application to the **receive** object named by the symbol.

setdefaultpatcherheight

The word setdefaultpatcherheight, followed by an integer value greater than 100, sets the default patcher height in pixels.

setdefaultpatcherwidth

The word setdefaultpatcherwidth, followed by an integer value greater than 100, sets the default patcher width in pixels.

seteventinterval — The word seteventinterval, followed by an integer value, sets the time between invocations of the event-level timer (The default value is 2 milliseconds). The event- level timer handles low-priority tasks like drawing user-interface updates and playing movies.

setrefreshrate — (Macintosh only) The word setrefreshrate, followed by a number, sets the rate, in frames per second, at which the visual display is updated. On Macintosh systems, the rate at which the screen is refreshed is unrelated to the rate at which you change its contents. Better "visual performance" can be acheived—at the cost of a slight performance decrease in Jitter, and little or no performance decrease for audio processing—by specifying a higher frame rate. When enabled using the enablerefresh 1 message, the default rate is 28.57 FPS. Refresh enable is off by default.

setsleep — The word setsleep, followed by a number, sets the time between calls to get the next system event, in 60ths of a second. The default value is 2.

setpollthrottle — The word setpollthrottle, followed by an integer, sets the maximum number of events the scheduler executes each time it is called (The default value is 20). Setting this value lower may decrease accuracy of timing at the expense of efficiency.

setqueuethrottle — The word setqueuethrottle, followed by an integer value, sets the maximum number of events handled at low-priority each time the low-priority queue handler is called (The default value is 2). Changing this value may affect the responsiveness of the user interface.

setslop — The word setslop, followed by a floating-point value, sets the scheduler slop value—the amount of time a scheduled event can be earlier than the current time before the time of the event is adjusted to match the current time. The default value is 25 milliseconds.

shortcut — The word shortcut, followed by a symbol and some additional text, will replace the symbol by the text when the symbol is typed into an object or message box and the escape key is pressed. Typically, shortcuts are added to files in the init folder in the Cycling '74 folder. Default shortcuts are in the files max-shortcuts.txt and audio-shortcuts.txt.

showcursor — Shows the cursor if it is hidden.

showglobal — Shows the floating inspector window.

showmenubar — Shows the menu bar after it has been hidden with hidemenubar.

size — Prints the number of symbols in the symbol table in the Max window.

system — The word system, followed by the name of an Operating System (windows or macintosh) and a message, will execute the message if Max is running on the named OS.

MIDI Configuration Messages

The following messages are used to configure MIDI on your system and to access the built-in DLS (Downloadable Soundfont) synthesizer for MIDI playback: By default, a single *augraph* (on Mac OS X) or *midi_dm* (on Windows) port is created. However, you can create additional MIDI synthesizer ports and assign new DLS sound bank files to each one. Addressing the DLS synthesizers currently requires the use of the **message** box technique where you send messages to named objects by typing a semicolon followed by the message text into a message box, then click the message box.

Here is a list of messages that you can use to access the DLS synthesizer:

Creating a DLS Port:

;#SM createoutport <portname> <drivername>

where *drivername* is midi_dm on Windows and augraph on the Macintosh. *portname* is the name you assign to the port. For example:

; #SM createoutport myOtherSynth midi_dm

; #SM createoutport myOtherSynth augraph

Deleting a DLS Port:

;#SM deleteoutport <portname> <drivername>

where *drivername* is midi_dm on Windows and augraph on the Macintosh. *portname* is the name of your choice. For example:

; #SM deleteoutport myOtherSynth midi_dm

; #SM createoutport myOtherSynth augraph

Loading a DLS Bank (type 1 or 2)

;#SM driver loadbank <filename> <portname>

where *filename* is the name of an existing DLS bank file, and *portname* is the name of the port that will use this bank. If portname is omitted, all DLS ports will use the bank. On Mac OS X, the folder /Library/Audio/Sounds/Banks is added to the search path when looking for a DLS bank file.

Loading the DLS Default GM Bank

;#SM driver loadbank 0 <portname>

Turning Reverb On and Off

;#SM driver reverb 1/0 <portname>

By default reverb is off in both augraph and midi_dm.

Setting MIDI Output Latency (midi_dm only)

;#SM driver latency <time> <portname>

where *time* is a value in milliseconds and *portname* is the port that is set to this value. For example, the following message would set the latency to 10 milliseconds:

;#SM driver latency 10 portname

Getting Information About Ports

;#SM inportinfo <portname> <receive name>

;#SM outportinfo <portname> <receive name>

The inportinfo and outportinfo messages send information about MIDI ports to named receive objects. The information is contained in an infolist message with the following arguments:

the port's name (symbol)
the port's driver name (symbol)
the port's unique ID (int)
the port's abbreviation (int)
the port's channel offset (int)
whether the port is enabled or disabled (1 if enabled, 0 if disabled)
whether the port was created dynamically (1 if yes, 0 if no)

Adding Virtual MIDI Ports (Macintosh only)

By default, Max creates two "virtual" MIDI ports for both input and output. If you wish to add additional virtual ports to your MIDI system, you can use the same **message** box technique described above to send the createoutport and createinport messages.

;#SM createoutport <portname> <drivername>

;#SM createinport <portname> <drivername>

where *portname* is the name you assign to the port, and *drivername* is the driver to be used (currently, only CoreMIDI is supported). For example, the following two messages:

;#SM createoutport myvirtualport CoreMIDI

;#SM createinport myvirtualport CoreMIDI

Would create a virtual MIDI input and output port, each named “myvirtualport”. These virtual ports are not saved as part of the Max/MSP setup, so they will have to be recreated each time you restart Max.

Examples

Control the behavior of Max from within a patch

See Also

pcontrol	Open and close subwindows within a patcher
thispatcher	Send messages to a patcher

Punctuation: Special Characters in Objects and Messages

Punctuation in Object Boxes

Many non-alphabetic characters have a special meaning in Max when included in objects and messages.

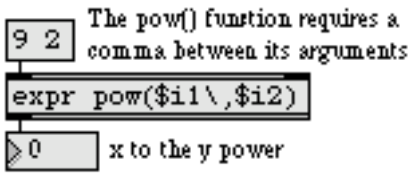
Many characters are object names in their own right, signifying arithmetic, relational, and bitwise operators for numerical calculations. These object names are **+**, **-**, *****, **/**, **%** (arithmetic operators), **<**, **<=**, **==**, **!=**, **>=**, **>**, **&&**, **||** (relational operators), and **&**, **|**, **<<**, **>>** (bitwise operators). See the descriptions for these objects at the end of the Objects section for more information.

The dollar sign ($) and the pound sign (#) are used in object boxes to indicate changeable arguments. A changeable argument is replaced by a value supplied either in the inlet (in the case of $) or as typed-in arguments to a patch that contains the object (in the case of #). The Arguments chapter has detailed information about $ and # in object boxes.

The semicolon (;) indicates the end of a message, and is not allowed in object boxes. Semicolons are also a way of forcing a carriage return in a **comment** object (except in two-byte compatible mode).

The semicolon indicates the end of a line in text files containing the contents of **coll**, **mtr**, and **seq** objects and in text files which contain a script for the **lib** object.

A comma (,) is generally another character to avoid using in object boxes, but may be used in an **expr** or **if** object, to separate items within a function in a mathematical expression, as in the example below. Note that a comma in an object box should always be preceded by a backslash (\), so that Max does not try to interpret it as a special character.



Use a backslash when you want to use a special character, but don't want Max to interpret it as such. In the example above, the comma is needed to separate arguments to the `pow` function.

Punctuation

Punctuation in a Message Box

The dollar sign ($) can be used in a **message** box to indicate a changeable argument. When the **message** box contains a $ and a number (such as $2) as one of its arguments, that argument will be replaced by the corresponding argument in the incoming message before the **message** box sends out its own message.

The pound sign, followed by a number (such as #2), in a **message** box has the same meaning as in an argument of an object box. When the patch containing a # argument is used as a subpatch inside another Patcher, the # argument is replaced by the corresponding argument typed into the subpatch object box in the main Patcher. See the Arguments chapter for examples.

A comma (,) in a **message** box is used to send a series of separate messages. The comma indicates the end of one message and the beginning of the next message.

In the above example, the **message** box on the left sends out a single message, 60 64 1 as a list. The **message** box on the right sends out three separate messages—first 144, then 60, then 64.

A semicolon (;) in a **message** box is used to send messages to remote **receive** objects. When a semicolon is present in a **message** box, the first item after the semicolon is a symbol indicating 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, rather than out the **message** box's outlet.

As in an object box, the backslash (\) in a message negates the special characteristics of the character it immediately precedes.

The number-letter combination 0x (zero-x) allows numbers to be typed into object and **message** boxes in hexadecimal form (useful for people who think of MIDI bytes in hex). For example, the message 0x9F 0x3C 0x40 is equivalent to the message 159 60 64.

See Also

Arguments	$ and #, changeable arguments to objects

Quantile: Using a Table for Probability Distribution

The quantile **message**

One of the messages understood by the **table** object is the word quantile, followed by a number. If you have read the description of this message, under **table**, you may have wondered what utility this complicated calculation might have. This section provides some examples. Here is the description of what quantile does.

quantile — In left inlet: The word quantile, followed by a number, multiplies the number by the sum of all the values in the **table**. This result is then divided by 2^{15} (32,768). Then, **table** sends out the address at which the sum of all values up to that address is greater than or equal to the result.

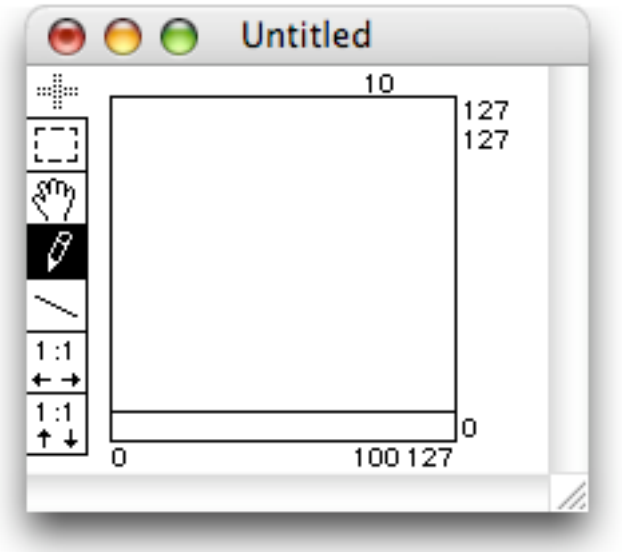
As the argument of the quantile message progresses from 0 to 32,768, each address in the **table** occupies a portion (quantile) of the 0 to 32,768 range, proportional to the "weight" given by the value stored at that address. Repeated quantile messages using random numbers cause each address to be sent out with a frequency roughly proportional to the value at that address.

The fquantile **message**

The fquantile message does the same thing as the quantile message, but it accepts a float argument between 0 and 1. Rather than require you to calculate the proportion of 32,768 that represents a fraction of the table length, fquantile allows you to specify it as a decimal number. For example, fquantile 0.5 is the same as quantile 16384, and fquantile 1.0 is equivalent to quantile 32768.

Examples

Suppose we have a **table** of 128 numbers, all set to 10.



Here are the results of some quantile messages on this **table**. Note that the total sum is 128 * 10 or 1280.

quantile 0 — Always causes an output of 0.

quantile 16384 — Returns the index up to which the sum of the values is half of the total sum. In this case, this would be 63, since 64 * 10 = 640 which is half of 1280.

bang — A bang is equivalent to a quantile message with a random number between 0 and 32768 as its argument, or an fquantile message with an argument randomly chosen between 0 and 1. Repeated bangs to a **table** will return **table** indices which contain higher values more often than indices which contain lower values. In the quantile example above, all indices are equally likely to be returned by bang, because all the values in the **table** are the same. However, if one of the values were 1000, the index at which the value was 1000 would occur far more frequently than any other **table** index. Exactly how frequently? This is determined by first taking the sum of all values in the **table**, which, for a **table** with 127 indices set to 10 and one at 1000 would be 2270. For the one index set to 1000, we divide 1000 by the sum 2270 and get a probability of 44 percent. For any of the other

127 indices set to 10, the probability is .44 percent that any one will be chosen. So, the index set to 1000 will occur about 100 times more frequently than an index set to 10.

See Also

histo	Make a histogram of the numbers received
table	Store and edit an array of numbers
Tutorial 33	Probability tables

seq

Max has four objects for recording and playing back MIDI performances: **seq**, **follow**, **mtr**, and **detonate**. The "performance" can come from outside Max—from a MIDI controller, or another MIDI application using the IAC Bus—or can be generated algorithmically within Max.

The basic sequencer in Max is **seq**, which records raw MIDI data received in its inlet from **midiin** or **midiformat**, and can play the data back at any speed. The recording and playback process is controlled with messages such as record, start, and stop.

Sequences recorded by **seq** can be written into a separate file to be used again later. Under OSX, "Max text file" and "Max binary file" are the two options for Save As... Under Windows, the options are "maxb Files (*.mxb, *.pat, *.help)" and "TEXT Files (*.txt, *.pat, *.help, *.mxt)" When **seq** receives a write message, it calls up the standard Save As dialog box. If the file is saved as text (by choosing *Max Format Text File* from the Format pop-up menu in the Save As dialog box), it can be edited by hand by choosing **Open As Text…** from the File menu. MIDI files can also be loaded into **seq** with a read message.

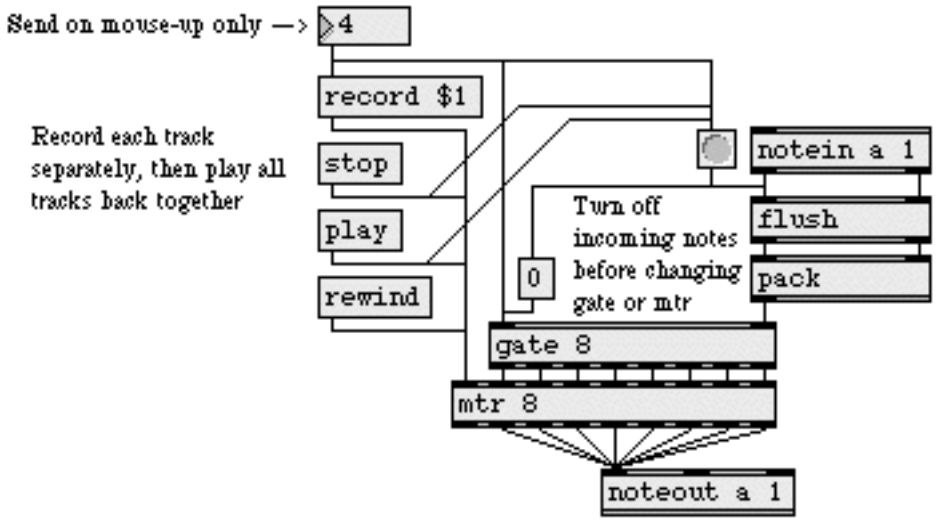
follow

The **follow** object functions exactly like **seq**, but has the added ability to compare a live performance to the performance it has recorded earlier. **follow** can record not only raw MIDI data, but also individual numbers such as note-on pitch values. You can step through the set of recorded notes (or numbers) using the next message. Most interestingly, **follow** contains a score following algorithm, activated by the follow message. **follow** will compare incoming numbers to those stored in its recorded sequence. If an incoming number matches the next number in the recorded sequence (or a nearby number, just in case the live performer makes a mistake), **follow** reports the index of the matched note. The index can then be used to read other numbers from a **table** or **coll** (providing an accompaniment to the live performer), or can be used to trigger any other process.

mtr

The **mtr** object is a multi-track sequencer that can record up to 32 individual tracks of numbers, lists of numbers, or symbols. With such versatility, it is easy to record not only MIDI events, but a wide variety of other messages. Tracks can be recorded, played, or

stepped through using the next message, either individually or collectively, and some tracks can be muted while other tracks continue to play. The contents of **mtr** can be written to and read in from separate files, either as individual tracks or as an entire set of tracks:



Sample patch using mtr

For editing complete MIDI messages as text, **seq** is perhaps more appropriate since it arranges raw MIDI data into a standard MIDI file format. However, raw MIDI data can be filtered with midiparse before being sent to **mtr**. Also, a sequence recorded in **seq** can easily be cut out and pasted into an **mtr** file, using Max's text editor.

detonate

The **detonate** object is a flexible sequencing, graphic editing, and score-following object. It can record a list of notes tagged with time, duration, and other information. You can save the note list as a single-track (format 0) or multi-track (format 1) MIDI file, and you can read in any MIDI file that has been saved to disk by **detonate**, **seq**, or some other sequencer. Double-clicking on a **detonate** object displays its contents in a graphic editor window, allowing you to use the mouse to add or modify notes inside it. It is also able to act as a "score-reader," much like the **follow** object; it looks at incoming pitch numbers and reports whenever an incoming pitch matches the current pitch in the stored score.

But unlike other sequencing objects such as **seq**, **follow**, **mtr**, and **timeline**, however, **detonate** does not really run on an internal clock of its own. Timing and duration information must be recorded into it from elsewhere in the patch, and the patch must also use that

information to determine the rhythm and speed at which notes will be played back from **detonate**—allowing for recording and playback options not available with the other sequencing objects, such as non-realtime recording, continuously variable playback tempo, and triggering individual events of the sequence in any desired rhythm.

timeline

The **timeline** object is designed for graphically editing a multi-track sequence of Max messages to be sent to specific objects at specific times. The **timeline** object does not record MIDI data in real time; it is for placing predetermined events in non-real time. However, once you have entered messages in the timeline—which the timeline could send to a patch containing MIDI output objects—the **timeline** object allows you great flexibility of playback of those stored messages. See the Timeline chapter for details.

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
timeline	Time-based score of Max messages
Timeline	Creating a graphic score of Max messages
Tutorial 35	Sequencing

Locked Patcher Window

- If **Help from Locked Patchers** in the Options menu is checked, Option-clicking on Macintosh or Alt-clicking on Windows on any object's box opens a help file for that object.
- Command-clicking on Macintosh or Control-clicking on Windows in any white space unlocks the Patcher window (if it's editable).
- Option-clicking on Macintosh or Alt-clicking on Windows on a window's close box closes all windows except the Max window.
- Option-clicking on Macintosh or Alt-clicking on Windows on the title bar of a subpatch window pops up a menu that allows you to bring any parent windows of the subpatcher to the front. If the subpatcher is an "edit-only" Patcher window (produced by double-clicking on a Patcher object which was read from a Max document), the top item in the list opens the Max document for editing.
- Typing Command-period on Macintosh or Control-period on Windows stops the Max scheduler, allowing you to recover from a runaway process which might be taking up too much CPU time to stop by normal methods. After you have remedied the situation which caused the process to get out of hand, choose **Resume** from the Edit menu to restart the scheduler.
- Holding down both the Command and Shift keys on Macintosh or the Control and Shift keys on Windows while a patch is loading prevents **loadbang** objects in that patch from sending any output.

Unlocked Patcher Window

Contextual Menus

- Control-clicking on Macintosh or Right-clicking on Windows in a Patcher window brings up a menu of useful editing commands. For more information about the Patcher window contextual menus, see the Menus topic.
- Option-Control-clicking on Macintosh or Alt-Right-clicking on Windows on any object displays a menu of all the messages you can send to an object. For more information on this menu, see the Menus topic.

Shortcuts

Selecting and Moving Objects

- Option-clicking on Macintosh or Alt-clicking on Windows on any object's box opens a help file for that object.
- Shift-clicking on an object box reverses the selected state of the object without changing the selected state of other objects.
- Shift-dragging an object box helps constrain the dragging of the object in the horizontal or vertical dimension.
- Option-clicking on Macintosh or Alt-clicking on Windows on one or more objects and dragging will duplicate the object(s).
- Hold down the shift key when clicking to place an object. Except for the object box, the cursor will remain the same, so you can make multiple copies of the same object.
- The arrow keys move the selected boxes by 1 pixel in any direction.
- Option-dragging a rectangle around a set of objects and patch cords or Alt-dragging on Windows selects both the patch cords and the objects.
- To drag a text object which has been selected for editing, move the cursor to the top or bottom edge of the box, then drag, as shown below. This is a handy way to move an object after duplicating it.

Click at the top or bottom edge of a selected text box to drag it

- Command-clicking on Macintosh or Control-clicking on Windows on any user interface object, such as a **slider** or **number box**, operates the object as if the Patcher window were locked.
- Command double-clicking on Macintosh or Control double-clicking on Windows edits objects such as **patcher**, **table**, and **coll** that open when you double-click them in a locked Patcher window.
- Command-clicking on Macintosh or Control-clicking on Windows in any white space locks or unlocks the Patcher window.

- With no objects selected, holding down the Option key on Macintosh or the Alt key on Windows and choosing **Send to Back** from the Object menu sends all **comment** objects to the background so they will not appear in front of other objects when the Patcher is locked. This may be helpful for updating files from previous versions of Max in which **comment** objects appear in front of other objects.

- After typing into an object or message box, Option-clicking on Macintosh or Alt-clicking on Windows outside the box prevents Auto Fix Width from changing the box's size. Option- clicking on Macintosh or Alt-clicking on Windows outside a comment box invokes Auto Fix Width on the comment, where it is normally disabled.

Patch Cord Shortcuts

- If Segmented Patch Cords is not checked in the Options menu, shift-clicking on an outlet of an object uses **Segmented Patch Cords** mode, in which subsequent clicks define the "corners" of the patch cord.

- Shift-clicking on an inlet of an object when making a connection lets you make multiple connections from a single outlet; another patch cord will be created immediately for the next connection.

- If **Segmented Patch Cords** is checked in the Options menu, shift-clicking uses the normal mode of dragging from outlet to inlet to make a straight patch cord.

- Hold down the control key if you are making a segmented patch cord and want to make a corner over an object—the normal auto-connection feature will be disabled.

- Command-clicking on Macintosh or Control-clicking on Windows while making a segmented patch cord gets rid of the patch cord and cancels the operation.

- Option-clicking on Macintosh or Alt-clicking on Windows while making a segmented patch cord gets rid of the last segment.

Creating Objects

- Option-clicking on Macintosh or Alt-clicking on Windows after selecting the Object Box tool in the palette places the object box without showing the New Object List window.

- Pressing the Delete (Backspace) key after selecting a tool from the palette cancels the operation and returns to the normal cursor.

- Clicking on the white area at the far left of the palette also cancels the selected palette tool.

New Object List

- Option-clicking on Macintosh or Alt-clicking on Windows on an empty object box opens the New Object List window.
- Delete (Backspace) hides the New Object List window.
- The Space bar enters the text of the selected item in the New Object List into the object box and adds a space afterwards for typing in any arguments.
- Return or Enter enters the text of the selected item in the New Object List into the object box.
- The Up and Down Arrow Keys scroll the selected item up and down.
- Tab switches which column of items is affected by typing.
- Holding the Option key on Macintosh or the Alt key on Windows down while double-clicking or typing Return, Space, or Enter opens a help file on the selected item.

You can add your own shortcuts to the New Object List. See the Text Macros section below.

send, receive, and value

- Double-clicking on a **send**, **receive**, or **value** object provides a contextual menu with a list of instances of these objects.

Table Editing Window

- Command-clicking on Macintosh or Control-clicking on Windows with the *Pencil* tool magnifies the area around where you clicked. If you are at 8:1 x 8:1 zoom, Command-clicking on Macintosh or Control-clicking on Windows returns to 1:1 x 1:1 magnification, or the minimum allowed zoom above 1:1.

Shortcuts

Any Window

- Command-clicking on Macintosh or Control-clicking on Windows in any window while **All Windows Active** is enabled brings that window to the front (if the window is not already in front).

  Option-clicking on Macintosh or Alt-clicking on Windows on the close box of a window closes all windows except the Max window.

Inspectors

- The Cut, Copy, and Paste keyboard shortcuts using the Command keys on Macintosh or the Control keys on Windows work in the text fields of any object's Inspector window.

Text Macros

Once upon a time, Max power-user Jasch was tired of typing "prepend set" all the time, so he created an object called _ (underscore) that did exactly what the **prepend** object did with "set" as an argument. Inspired by his efforts to reduce repetetive stress injury (RSI) among users, Max includes the ability to set up text macros for things you type into object boxes—including Jasch's underscore shortcut.

To try it out, create a new object box, then type the underscore character followed by the Escape key (ESC). The underscore will be replaced by "prepend set" and the insertion point will be after the word set (in the unlikely event you want to type something else).

Another example: let's say you do a lot of work with multichannel audio I/O. Now you can type `d6 <ESC>` instead of `dac~ 1 2 3 4 5 6`.

To add your own shortcuts, you place a message to the max object in a text file in the init folder as follows:

```
max shortcut <shortcut-text> <replacement-text>;
```

The shortcut text must be a single symbol, which means if you want to include a space in the shortcut, you will need to put all of the text in double quotes. The replacement text does *not* need quotes around spaces. For example, to replace underscore with prepend set, you would add the following message to a text file in the init folder:

Shortcuts

```
max shortcut _ prepend set;
```

Examine the files *max-shortcuts.txt* and *audio-shortcuts.txt* in the init folder for examples and inspiration.

Timeline: Creating a Graphic Score of Max Messages

Introduction

A timeline is a graphic editor for creating a score (like a musical score) of Max messages. When you tell the timeline to play that score, it sends its specified messages to the specified patches at the specified times.

There are three basic steps in creating a timeline.

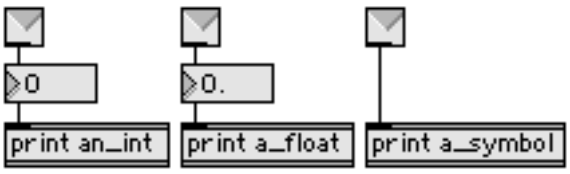
1. Create (or modify) at least one patch to communicate with the timeline. This patch must contain at least one **ticmd** object. Just as you would use an **inlet** object in a patch to receive messages from a parent patch, you use **ticmd** to receive messages from the timeline. Such a patch, which communicates with a timeline via the **ticmd** object, is called an action.

2. Create a timeline, and create at least one track within that timeline. A track corresponds to, and communicates with, a specific action (patch) you have created.

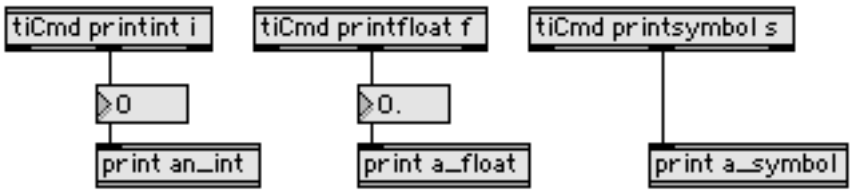
3. Place events in the timeline's track(s), specifying messages to be sent to the **ticmd** objects in the track's action.

Creating an Action

Any patch that receives messages from an inlet can easily be converted to receive messages from a timeline track. For example, the patch shown below receives a symbol, an int, and a float in its inlets, and prints them in the Max window.



But in order for this patch to receive messages from a timeline, the inlets must be replaced with **ticmd** objects, as shown below.



The **ticmd** object requires two or more arguments. The first argument is a command name by which the timeline can refer to the **ticmd** object. The remaining arguments indicate the type of message **ticmd** is expecting, and determine the number of outlets it will have. Each argument after the command name creates an outlet, and specifies the type of information to be sent out of that outlet: `i` for int, `f` for float, `l` for list, `s` for symbol, `b` for bang, and `a` for any message. (You will notice that there is an additional outlet on each end of the **ticmd** objects; these outlets will be explained later.)

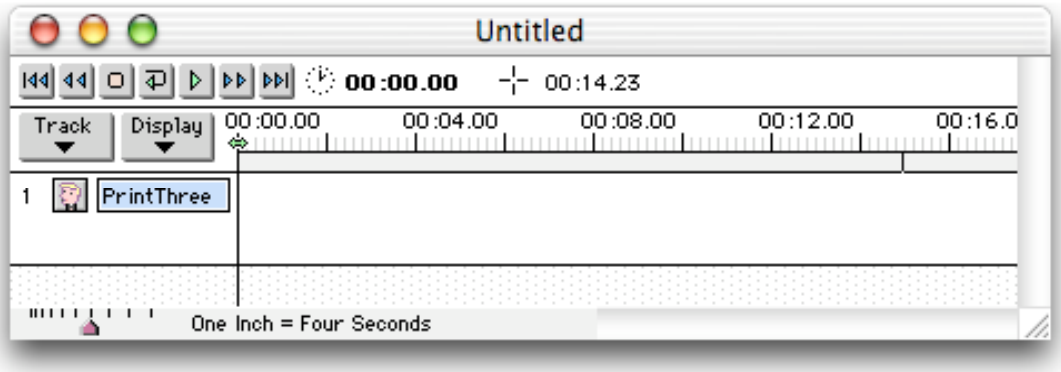
Any patch that contains at least one **ticmd** object is ready to be used as an action. You may save it anywhere, but if you save it in the *Timeline Action* Folder (as specified by the File Preferences... command in the Edit menu) it will automatically appear on timeline's pop-up Track menu. When you first install Max, the *Timeline Action* Folder is a folder named *tiAction* inside the Max application folder.

Creating a Timeline

To create a new timeline, choose Timeline from the New menu. It is also possible to create a new timeline by typing **timeline** into a new object box. Either way, a graphic timeline editor window will be opened for you.

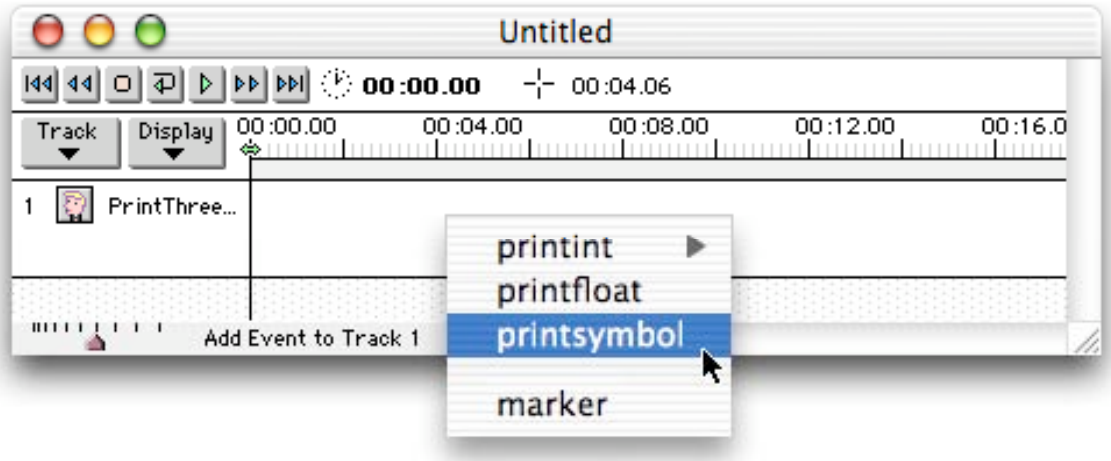
When you first open a timeline editor window, it contains no tracks. To create a new track in the window (and thus load a specific action), click the Track button and select an action file by name from the pop-up menu. The pop-up menu will show all the patches contained in Timeline Action Folder. If you don't see the name of the action patch you want, choose Other… from the pop-up Track menu and you will be able to load the action with a standard open file dialog.

Once you have created a track, you can view and edit the action by double-clicking on the little Max icon in the leftmost portion of the track.



Creating Timeline Events

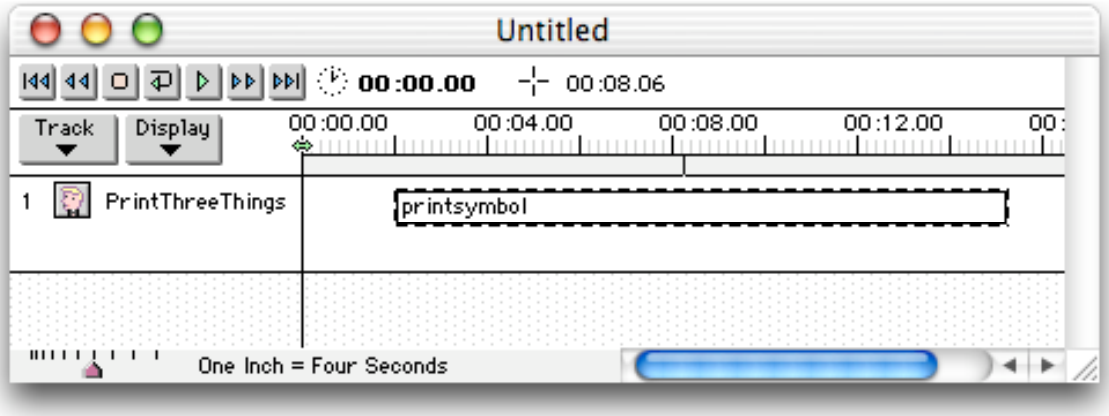
An event is an object you place in a timeline track; the event sends one or more messages to a particular **ticmd** object in that track's action. You place an event onto the timeline by Option-clicking on Macintosh or Alt-clicking on Windows in the right side of the track. This reveals a pop-up menu of names corresponding to the names of **ticmd** objects within the action. You can also place an event in a timeline track simply by clicking in the event portion of the track and holding the mouse down until the pop-up menu of possible events appears, then choosing the event.



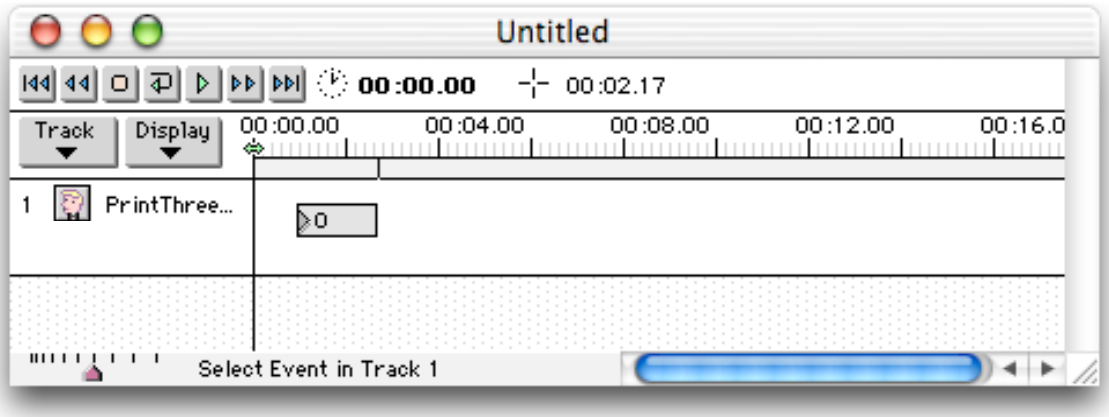
When you choose a command name from this menu, you are actually specifying which **ticmd** object you want to send a message to. Based on the b, i, f, l, s, or a arguments in that

ticmd object in the action, the timeline knows what kind of message is appropriate for that event, and places an object (known as an editor) for that message in the track.

When you place an event that sends a bang, a symbol, or a list, Max will give you an editor known as the **messenger**. The **messenger** looks just like the **message** object, except that it has a label showing the command name of the **ticmd** object to which its message will be sent.

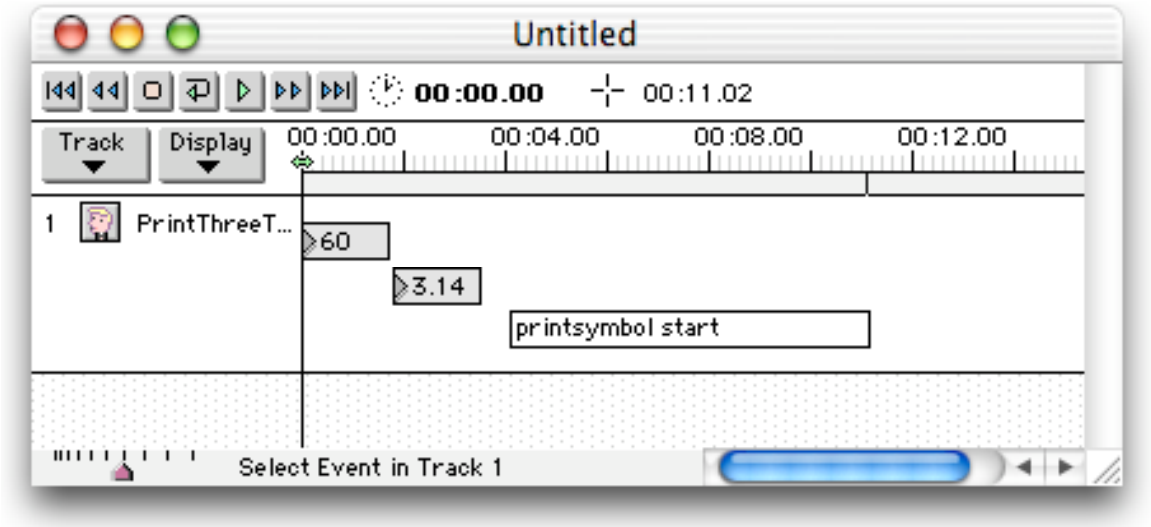


To place a single number as an event, you will use the **int** and **float** editors, which look just like the **number box** object.



Once you have placed an event in the track, you can edit the event's contents (change the message it will send to the **ticmd** object) or drag it to a new location in the track (change the time at which its message will be sent). You can also cut or copy events from one track and paste them into another track, provided they are appropriate events to be placed in that other track.

If the action contains different **ticmd** objects (as is the case with our example **PrintThreeThings** action), then a track can contain different kinds of event editors. In the following example, when the timeline is played it will send an int 60 to be printed at time 0, a float 3.14 to be printed at time 1000 milliseconds (1 second), and the symbol start to be printed at time 2000 (2 seconds).



You can choose the format in which you want time to be displayed by clicking on the Display button and choosing Time Units from the pop-up menu. You can choose to display time in milliseconds rather than the minutes, seconds, and frames (for film or video) shown in this example.

Once you have completed the three steps of creating a timeline—creating an action, creating a timeline, and creating timeline events—you can play the timeline using the tape recorder-like controls in the upper left corner of the timeline window.

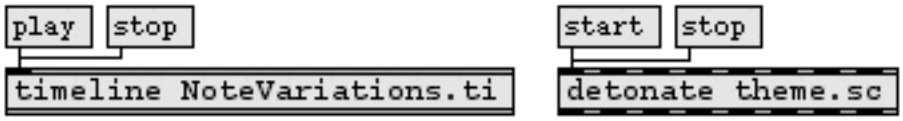
The edetonate Editor

An event editor called **edetonate**, which works just like the graphic editor window of a **detonate** object, can be used in a timeline for sending list messages to **ticmd**. Once you have placed it in a timeline track, you can double-click on it to open its graphic note event editing window. For details of this graphic editor window, see the Detonate Topic.

Because of the **edetonate** object's orientation as a sequencer of note events, it is especially well suited to sending list messages that will be used as note events in the action patch. When the timeline is played, **edetonate** sends out the note-on events that you have drawn

into it, and also sends out corresponding note-off messages after the amount of time specified by each note's duration value.

You can suppress the note-off messages by selecting the **edetonate** editor, choosing **Get Info...** from the Object menu, and unchecking the Send Note-Offs option. In the same dialog box, you can type a name for the editor in the Explode Label box. All **edetonate** editors that share the same name also share the same data. They can also share their data with a single **detonate** object that has the name typed in as an argument.



A single detonate object with a typed-in argument shares data with any edetonate editors with the same name in the timeline

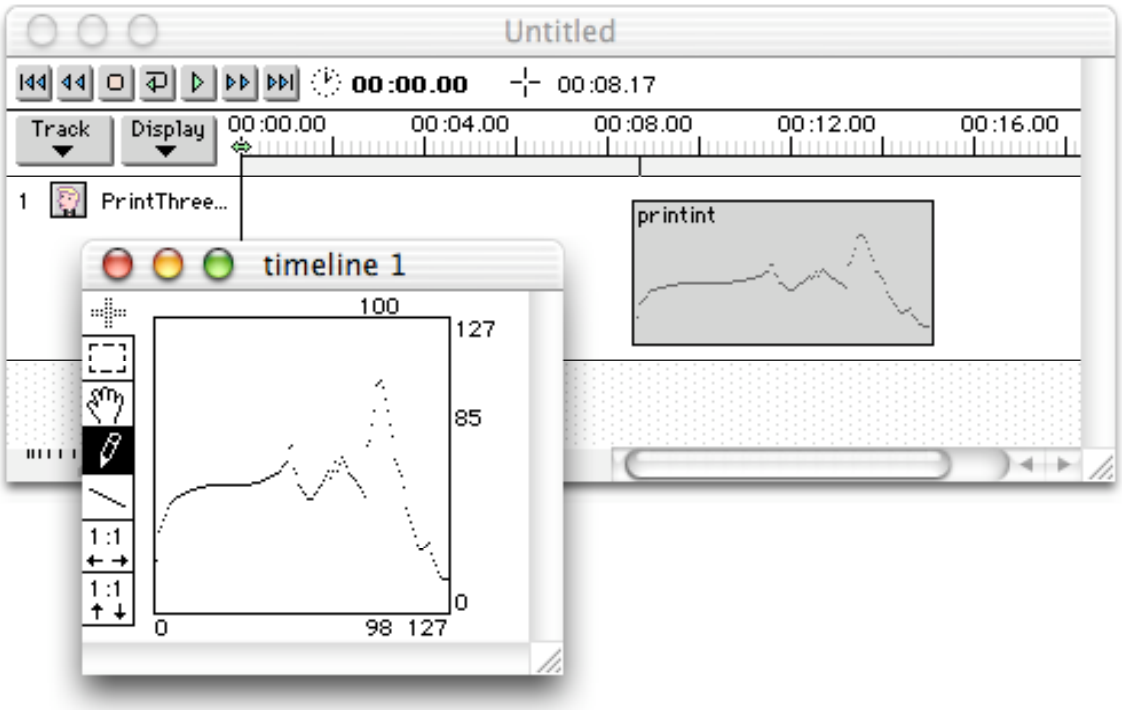
Note that the horizontal length of an **edetonate** on the timeline determines its real duration. The time and duration values in the **edetonate** editor window are actually relative times, which will be scaled when the timeline is played, to fit in the time occupied by **edetonate** in the timeline. Selecting an **edetonate** editor and choosing the **Fix Width** command from the Object menu makes the length of the **edetonate** equal to the length of the sequence it contains. If you make any subsequent changes to the contents of the **edetonate** (or its associated **detonate** object in a patcher), you will have to adjust it once again with the Fix Width command in order for it to play without its time being scaled.

The etable Editor

There are actually three different possible event editors for sending messages to **ticmd** objects that expect a single integer: **int** (like the **number box**) **etable**, and **efunc**.

When you create an **etable** editor, it appears as a shaded box in the event area of the track. The command name of the corresponding **ticmd** object is displayed in its upper left corner.

Double clicking on this box will display the familiar table editor.



Any changes you make in the table editor will appear in the **etable**. When you play the timeline, the **etable** will send its stored values to the corresponding **ticmd** object in order from left to right. The values from the **etable** being played by the timeline are sent out the **ticmd** object's middle outlet.

By clicking and dragging the lower right corner of the **etable**, you can resize it. Resizing the **etable** horizontally will change its duration on the timeline, causing its values to be sent out at a different rate. (Resizing the **etable** vertically has no effect on the data sent to **ticmd**.)

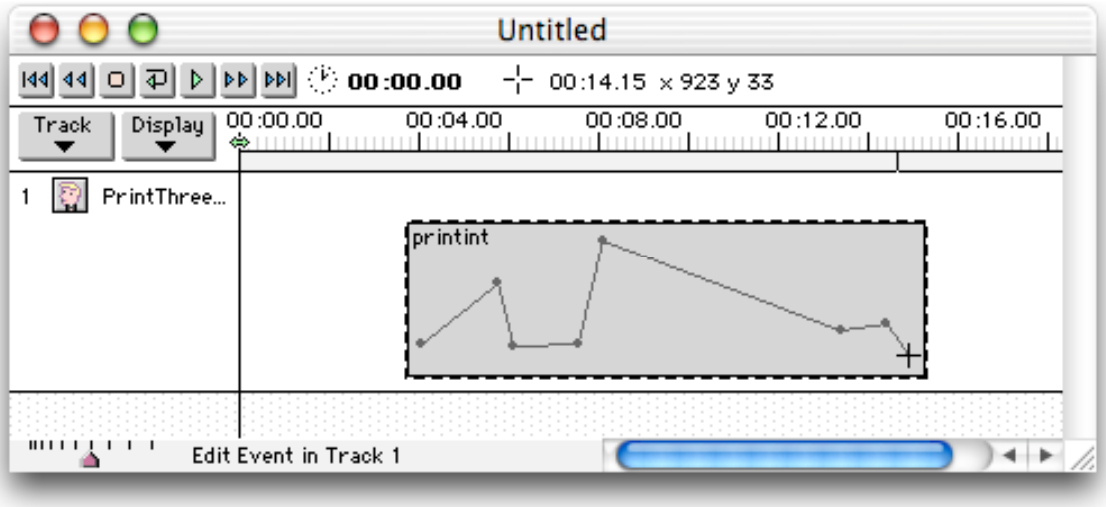
When the timeline is being played, and reaches the left edge of the **etable**, a `bang` is sent out the corresponding **ticmd** object's left outlet.

The contents of an **etable** will ordinarily be saved with the timeline that contains it. You can also link an **etable** to an existing **table** object. By clicking on an **etable** and choosing **Get Info...** from the Object menu, you can enter a label for the **etable**. Once labeled, it will share the data of a loaded **table** object bearing the same name. This **table** object may be in an open patch, or in an action within the timeline. Once an **etable** his been labeled, you

can still edit it graphically by double-clicking on it (which will also alter contents of the **table** to which it is linked).

The efunc Editor

When you create an **efunc** editor, a shaded box similar to the **etable** editor appears. By clicking in the **efunc** editor box, you specify a point to be stored as an x,y pair of numbers. When you click in **efunc**, the actual values of x and y for the point where you click are shown at the top of the timeline window. Each time you click at a different point, you create a new x,y pair of numbers, and **efunc** connects all the points with lines segments from left to right.



You can move any existing point simply by dragging it. The coordinates of the point are displayed as you drag it.

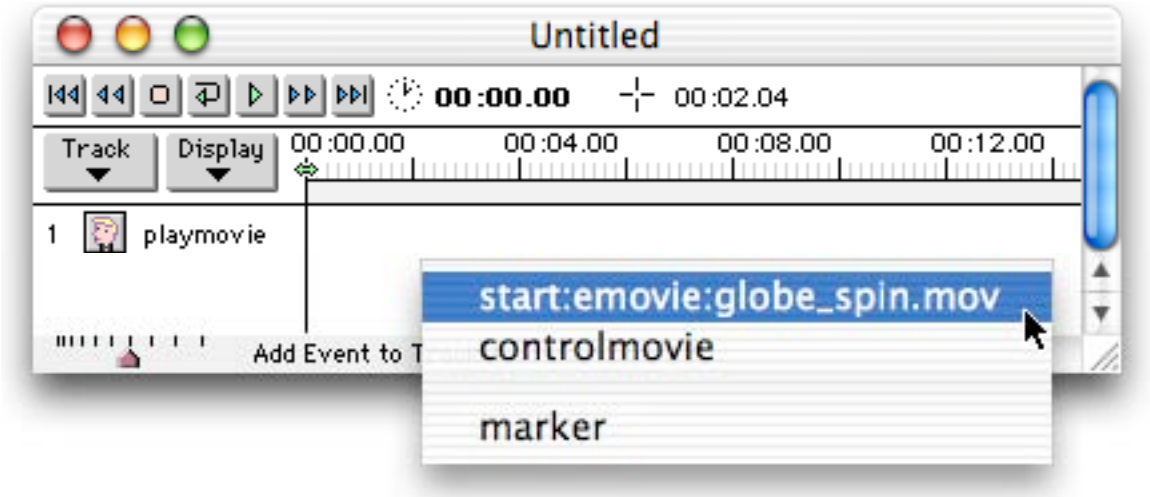
When timeline plays back the data in an **efunc** editor, it sends the y (vertical) value of each x,y pair out of the middle outlet of the appropriate **ticmd** object, at a time corresponding to the value of x. By default, **efunc** does not interpolate between points; that is, it does not supply intermediate points along the connecting line segments. In order to make timeline interpolate the values between points (fill in the "ramps" between points), select the **efunc**, choose **Get Info...** from the Object menu, and enter a nonzero value for Interpolation Time Grain. This number will determine the resolution of the interpolation. A value of 1 will provide the highest resolution interpolation, causing **efunc** to report its current value to the **ticmd** object every millisecond. A value of 100 will cause **efunc** to report every tenth of a second, and so on.

Choosing **Get Info...** from the Object menu also allows you to set the range of the x,y graph by specifying maximum x and y values for **efunc** coordinates. You can also enter a label which will link the **efunc** editor to a **funbuff** object of the same name. Once an **efunc** editor is linked to a **funbuff** object, you can still edit the **funbuff** through the **efunc** editor, and changes will be reflected in all **funbuff** objects sharing its label.

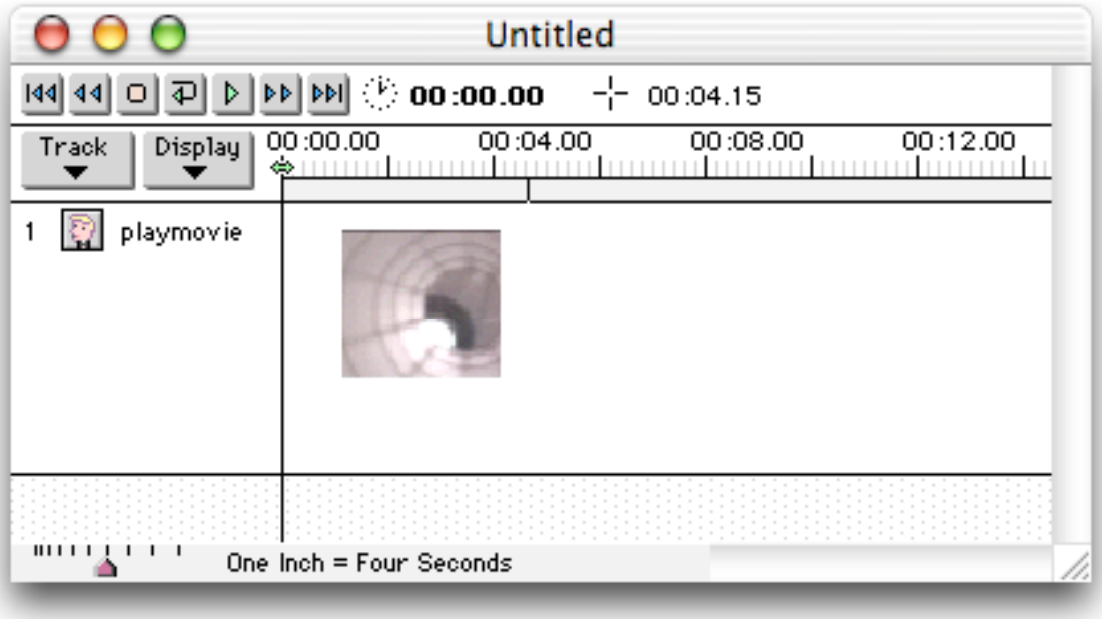
Horizontal resizing of the **efunc** editor has the same effect as resizing the **etable** editor—changing the total duration in which the numbers are sent out when the timeline is played—but does not change the time grain of the interpolated output.

The emovie editor

As explained earlier, when you create a new timeline track—and thus assign it a particular action—the command name of each **ticmd** object in that action becomes available as an event which can be placed in the event portion of that timeline track. Additionally, whenever one of the actions used in your timeline contains a **movie** object, into which a QuickTime movie has been read (either with a typed-in argument specifying a movie file, or via a read message), the movie window will be opened and a new type of event editor will become available in that action track. The new event editor is called **emovie**. It allows you to place a start event in the track, which will be sent directly to the **movie** object (without having to go through **ticmd**).

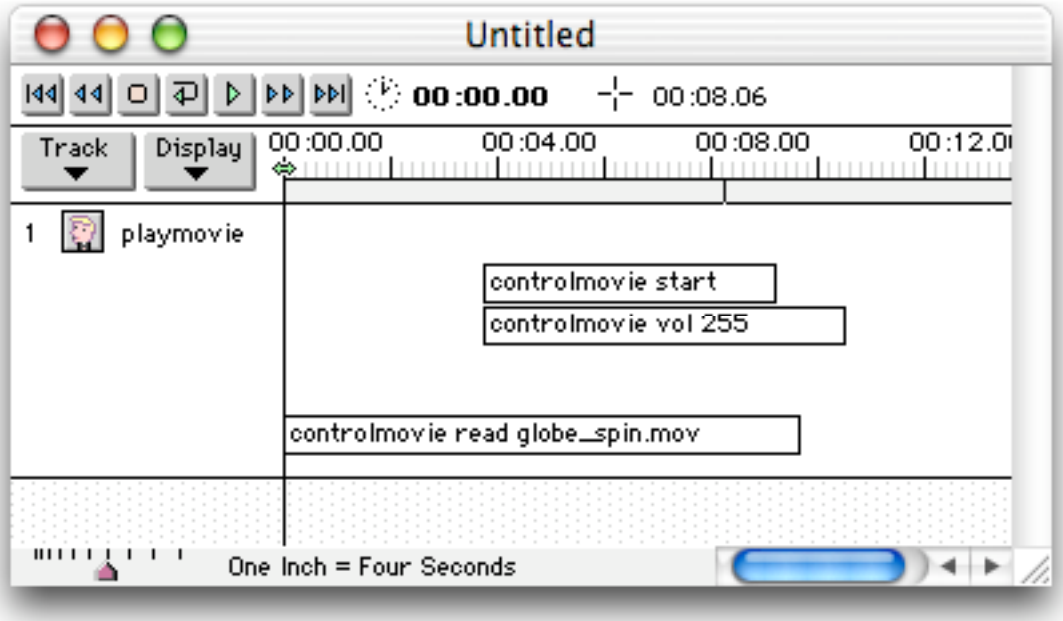


When you place an **emovie** event in the track, a "thumbnail" miniature frame of the movie is shown in the track to remind you what movie will be started at that time.



Of course, it is also possible to send messages to a **movie** object in an action just the same way you would send any other messages: via a **ticmd** object.

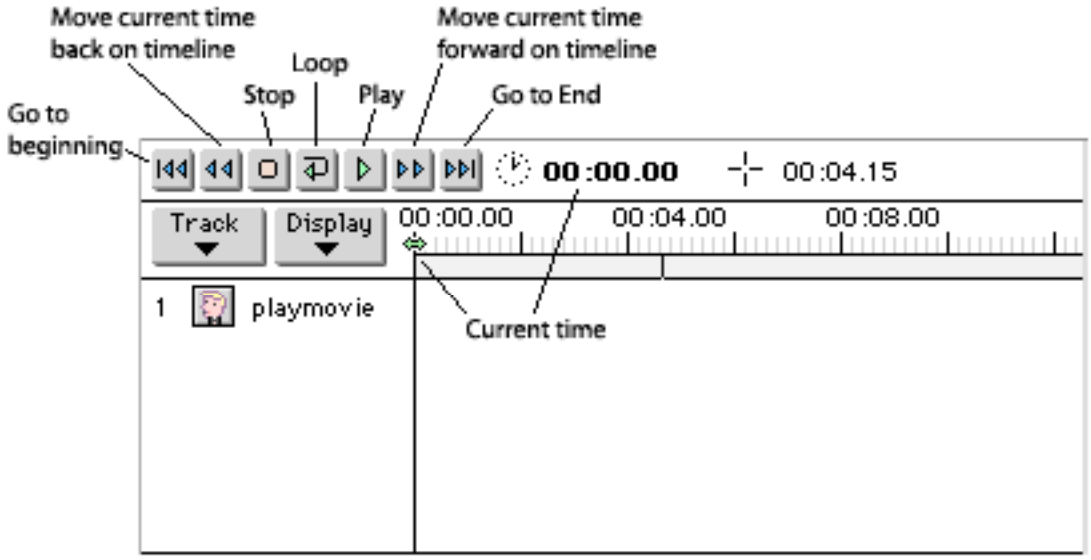
For example you could send a message to load a movie (the word **read** followed by the name of a movie file), set the volume, and start the movie, all from within a timeline, via **ticmd**.



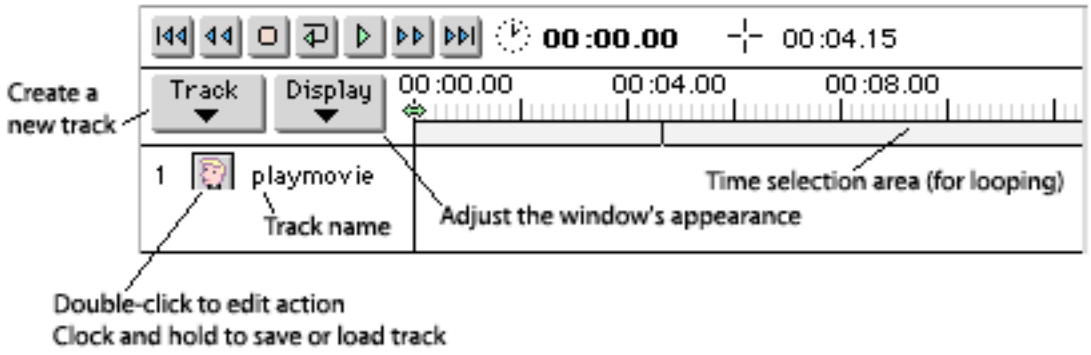
Features of the timeline Window

In the upper left corner of the timeline window there are tape recorder-like controls for playing the timeline. Next to the controls there is a clock icon and a digital readout of the "current time" as recognized by the timeline (the current point of the timeline's progress). The current time is also indicated by the little arrow indicator on the timeline itself.

Next to the current time, the current cursor position is displayed. This is useful as a reference for placing events accurately in a track with the mouse.



You create a new track by choosing an action from the pop-up menu labeled Track. In the left part of the track you are shown the track number and the track name. The track name is initially set to be the same as the name of the track's action, but you can change the track name to something else (just by clicking on the name and editing it) without affecting the action assigned to that track.



To select an entire track, click on the track number. To select multiple tracks, select one track, then shift-click on the track number of the other tracks. Once you have selected one or more tracks, you can edit them with the commands in the Edit menu: cut, copy, and paste them, clear out all their events, etc. To relocate a track, select it, choose Cut from the Edit menu, then select the track after which you want to place the cut track, and

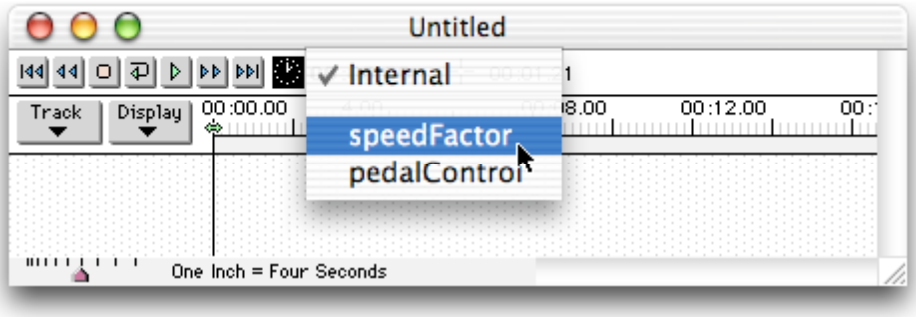
choose Paste from the Edit menu. Whenever you create a new track, it will become track number 1 if no track is currently selected; if any tracks are currently selected, though, the new track will be placed after the highest-numbered selected track. You can also adjust the visual height of a track—to allow you more vertical space for placing events—just by dragging up or down on the bottom edge of the track.

By double-clicking on the little Max icon next to the track name, you can view and even edit the action patch for that track. If you have more than one track using the same action, any changes you make (and save) in that action patch will immediately affect all of those tracks.

You can save the entire contents of an individual track in a separate file—its track name, action name, and all its events—then reload that track into a timeline at a later time. If you click once on the little Max icon in a track and hold the mouse button down, you will be presented with a pop- up menu which gives you two choices—Open Track File... and Save Track As...—for saving and reloading an individual track.

The Display pop-up menu lets you alter the look of your timeline window to suit your needs. You can display the Time Units in one of several different formats: Milliseconds, Midi Clock, or SMPTE format of Minutes:Seconds:Frames (24fps, 25fps, or 30fps). You can collapse tracks down to a single line of vertical space, thus allowing you to see many tracks at once, or you can expand them back out to their full height to see all of their contents. You can choose to Show Mute Buttons in the left part of the tracks; these buttons are useful for suppressing the events on individual tracks. And, with the Autoscroll While Playing option, you can choose whether the timeline display should follow the progress of time or remain stationary when the timeline is being played. All settings you specify in the Display menu are saved as part of the timeline file.

The timeline's clock can be synced to any **setclock** object in any currently loaded patch. Double- clicking on the little clock icon at the top of the timeline window displays a pop-up menu containing the names of all currently loaded **setclock** objects.



Choosing one of those names from the pop-up menu syncs the timeline to that **setclock** object. Choosing Internal from the pop-up menu returns the timeline to following Max's internal millisecond clock.

Holding down the Command key on Macintosh or Control key on Windows and clicking on an event sends that event's message to the **ticmd** object(s) in the action patch, allowing you test the effects of the message as you edit the timeline. You can play through a segment of the timeline in a repeated loop (also useful for testing a timeline as you edit it) by selecting a segment of time in the time selection area just under the ruler at the top of the timeline window, then clicking on the Loop button.

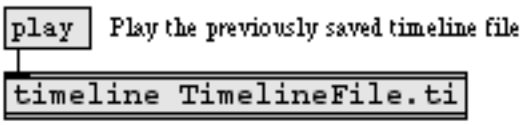
There is an additional event editor called a **marker**, which functions similarly to a **comment** object in a patch. The **marker** allows you to type in comments and notes about events in the timeline, or (more importantly) to mark a specific point on the timeline. When a **timeline** object in a patch receives the message search, followed by the first word of one of the **marker**s in the timeline, the current time pointer of the timeline moves to the location of that **marker**. (See Tutorial 41 for an example of searching for a **marker**.) You can even create a Marker Track in a timeline window: a track that does nothing but contain **marker** events.

When a timeline object receives the message markers, followed by the number of one of its outlets, it sends the first word of each **marker** contained in its tracks out the specified outlet, to be stored in a **menu** object. This **menu** can then be used to move the timeline's current time pointer to the location of a particular **marker** (by prepending the word search to the text output of the **menu**).

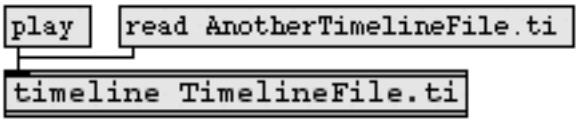
Using timeline in a patch

So far we have only discussed the use of the timeline editor window. Once you have created a score consisting of action tracks and messages to be sent to those actions, you will no doubt want to save your score for later use. Choose Save from the Edit menu, and save your timeline. Max recognizes timeline files as being different from patches, and when you reopen the file it will be displayed once again in the timeline editor window, and you can play or further edit your score. Once you have saved your timeline as a file, you can also load it automatically into a patch.

When you create a **timeline** object in a patch, without typing in an argument, a new timeline editor window is automatically opened for you. However, if you type in a timeline filename (that is located in Max's file search path) as an argument to **timeline**, that timeline file will be automatically loaded in, and you can then play that timeline score by sending a play message to the **timeline** object.



With the read message, you can load a different timeline file into the same **timeline** object (replacing any timeline score that was there previously) and play it.



Note that for this to work effectively, the timeline file(s) must be in Max's file search path (as specified by the File Preferences... command in the Edit menu, or in the same folder as the patch that is trying to load them) and the action patches used by those timelines must also be locatable (in the Timeline Action Folder specified in the File Preferences dialog, or in the same folder as the patch that contains the **timeline**). The **timeline** object understands a great many other messages for controlling it or altering its parameters. See the **timeline** page in the Objects section for details.

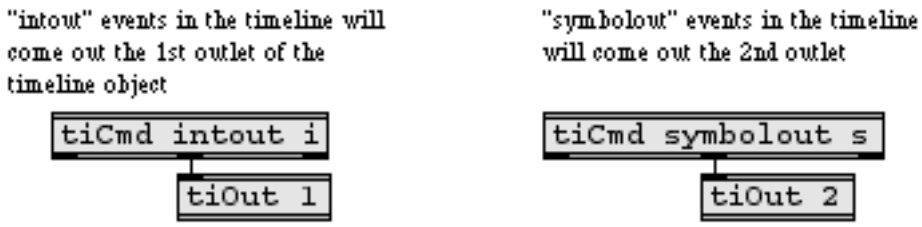
Playing a timeline from within a patch can seem a little mysterious since, once the messages are sent from the timeline, all the action takes place in the action patches, which in most cases are out of sight. However, you can create interaction between a timeline and the patch that contains it. Messages (which are sent to **ticmd** objects in actions) from the timeline event tracks can be redirected out outlets of the **timeline** object. In fact, actions

can themselves send messages out outlets of the **timeline** object. This type of interaction is achieved by using the **tiout** object in an action patch, and by creating outlets in the **timeline** object itself.

A second argument typed into a **timeline** object specifies the number of outlets the object will have (the first argument is a timeline filename to be read in automatically).

The second argument determines the number of outlets

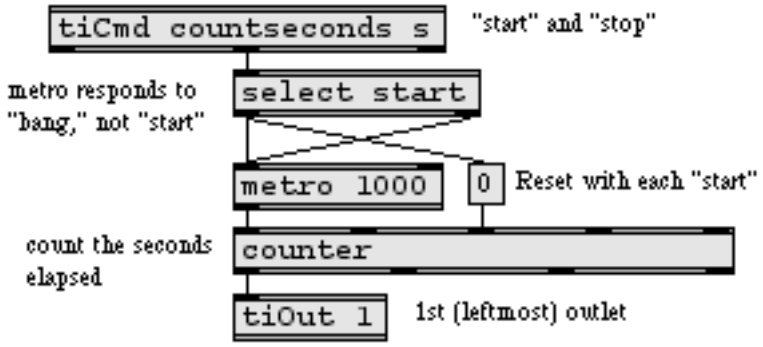
For messages to come out of those outlets, at least one of the actions used in the timeline must contain a **tiout** object. Any message that goes into a **tiout** object in the action will come out of the appropriate outlet of the **timeline** object using that action. Here is an action that is specially designed to send integers out the left outlet of the **timeline** object that uses it, and symbols out the second outlet.



The argument to tiOut tells which outlet the message will be sent out

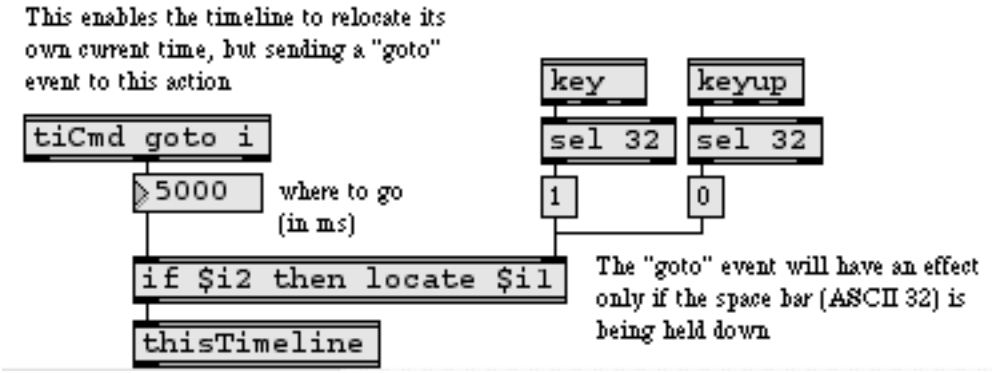
The actual messages to be sent out the outlets of **timeline** need not originate in the timeline event editor; they may be generated within the action patch itself. Below is an example of an action which understands a “countseconds” event.

When the timeline messenger event countseconds start occurs, the action begins to send integers out the left outlet of the **timeline** object, until the countseconds stop event occurs.



So, in this case, the timeline sends symbols (start and stop) to **ticmd**, and the action itself sends ints (the count of the number of elapsed seconds since a start message was received) to **tiout**, which sends them out the outlet of the **timeline** object.

As you have seen, a timeline can be controlled either with the buttons in the timeline window or by messages received in the inlet of a **timeline** object in a patch. There is a third way that a timeline can be controlled: it can control itself. An action patch can contain an object called **thistimeline**, which sends messages back to the timeline that is using that action. A message received in the inlet of **thistimeline** in an action is sent to the timeline itself, allowing an action to control the timeline that is using it.



In this example action, a "goto" event in the track will cause the timeline to relocate to whatever time location it gives itself. (Here it is telling itself to go to a point five seconds into the timeline.) In the example, a conditional clause has been built into the action so that the "goto" event will only be enacted by the action if the space bar is currently being held down. The interaction between a timeline, the patch that contains it, and the actions

it employs can be as complex as you care to make it. You will need to plan your program very carefully to be sure that you understand which object is actually acting at any given moment: the patch containing a **timeline** object, the timeline itself, or the action(s) being used by the timeline.

See Also

mtr	Multi-track sequencer
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
Tutorial 41	Timeline of Max messages

Index

Index

Index

Index