Reference Manual for version 2.0
(update #2: 89.6)

Miller Puckette

IRCAM, 26 octobre 1988

Preface.

This is the reference manual for the Max graphical programming environment for computer music. Max is now available in version 2.0 alpha40 for internal use at IRCAM. People are asked not to give out copies of Max to people outside IRCAM; the commercialized version should be ready in Fall 1989.)

This reference manual attempts to describe, concisely, all there is to know about Max from the user's standpoint; the "Manuel d'utilisation" by Philippe Manoury goes into much more detail and is intended to get musicians started using Max. There is also an on-line help facility by Cort Lippe, a paper copy of which is at the end of this manual.

About future and past compatibility: This manual describes only the parts of Max that are "standard." You can count on these standard features always existing and working in the same way (except for bugs). Many other "obsolete" features exist only for backward compatibility; they are summarized in section 11. You should not use them and if you have an old patch which does, it might not work with future releases of Max.

On two occasions, Max was changed to use a different file format (alpha24 and alpha37). New versions should be able to read the old-fashioned formats, but old versions may not read files you have made with newer versions. If this happens, your patch will open as a text window (but in the commercial version this will be detected as an error.)

Acknowledgements. Parts of Max were contributed by Lee Boynton, Cort Lippe, and Zack Settel; brave composers who used it early (and thus helped its development) include Frederic Durieux, Michael Jarrell, and Philippe Manoury, assisted by Thierry Lancino and Jan Vandenheede. In addition to all of them, I would like to thank David Wessel, without whom this project would never have succeeded. And of course Max Mathews, for whom Max is named. Not only did he greatly influence me in the months we worked together, but his RTSKED program (1982) marked the first use of the real-time scheduling approach that Max adopts.

Contents.

1. Introduction.

MAX is a graphical music programming environment for people who have hit the limits of the usual sequencer and voicing programs for MIDI equipment. These "black box" programs make restrictive assumptions about how you are using your MIDI studio: you walk in, voice the instruments, play several keyboard tracks, and finally hit the "play" button on the Macintosh (after hitting the "record" button on your tape recorder.) The first few tapes will sound pretty good, but soon they will all start to sound alike. This "canned" sound is due to the closed nature of the software.

To take complete control of all the possibilities of a synthesizer you need to be able to specify independently where all the elements are coming from: the basic pitch and tempo material, timbral changes, pitch articulation, whatever. They should be controllable physically, sequentially, or algorithmically; if algorithmically, the inputs to the algorithm should themselves be controllable in any way. If you have more than one synthesizer, and/or unusual input devices, the need becomes more acute to be able to define, for yourself, exactly who will control whom, and how.

Anything which could give you such wide-open control of your MIDI equipment is not going to be for beginners. MAX was written at a computer music production studio (IRCAM) and tested on musicians who were already experienced computer users. Even if you are already familiar with a Macintosh and with the MIDI equipment you are using, it may still take you a month or two to take full advantage of what MAX can offer you.

Getting started.

To use MAX you will need a Macintosh computer with at least one megabyte of memory, a MIDI interface for it, and some kind of MIDI equipment. (You can actually run MAX without the MIDI equipment but it won't be very interesting.) The system on the Macintosh should be at least version 4.0.

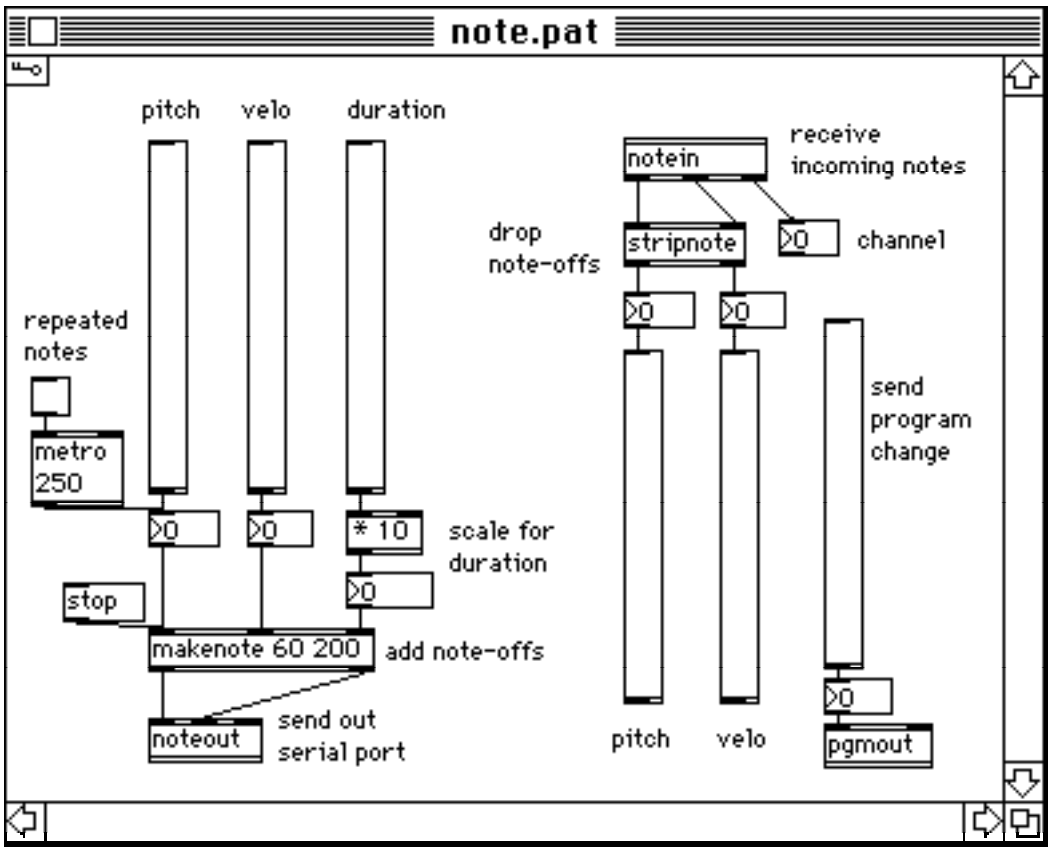
The MAX distribution disk contains the MAX program, a file named "CLICK HERE" and two folders: max-help and patches. MAX can run without the other files, but it will be helpful to have them. The max-help folder contains files which MAX will search for if you ask for help; it will find them only if max-help is in the same place as MAX itself (i.e. the same folder.) The patches folder contains some patches to get you started. The "CLICK HERE" file contains the latest news about the current version of MAX.

WARNING: MAX will try to open the serial ports. If one or more other program already has one or both serial ports open, MAX will not open them; but once MAX *does* open them it will keep the open until you quit from MAX, regardless of what you might specify as its MIDI setup.

ANOTHER WARNING: MAX might hang up if you turn off your MIDI interface while MAX is running.

Load your first patch.

The folder "patches" contains a number of sample patches which you can load and play with without going further. So that the next few paragraphs make sense, you might want to take a moment to load "note.pat" from there. If you are in the Finder, open the "patches" folder and click on "note.pat"; if you are already in MAX, you can open it by choosing "open" in the File menu. You should get a window like:



You should hear a stream of notes when you slide the leftmost ("pitch") slider; the next two ("velo" and "duration") are for setting the velocity and duration of the notes. You get a toggle switch ( ) labelled "repeated notes" which turns a stream of notes on and off, equal to the last one you have played. Further to the right there is a network which shows you incoming notes; and finally there is one more slider for sending program changes.

In addition to the sliders and the toggle, there are class boxes like * 10, a message box (stop), number boxes ( ) and comments ("pitch", "velo", "duration"). The borders of the boxes tell you what type they are.

You can put the patch in "edit" mode by clicking on the key symbol ( ) which will open up into the patcher palette ( ). Clicking the key again returns the patch to "run" mode and closes the palette. While the patch is in "edit", you can select boxes (by clicking on them), drag them, cut and paste them, and create new ones by clicking on the palette.

There are actually four types of windows in MAX: the "MAX" window for logging printouts from MAX, and editing windows for patchers, function tables, and text. These editing windows can be saved and recalled to/from Macintosh files using the usual "Open", "Save", and "Save as" commands in the File menu. It is a good idea to keep an eye on the MAX window, especially if something is going wrong; it is used for logging any warning messages for which it might not be appropriate to stop everything for an alert window.

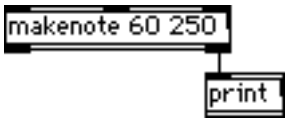
Getting help.

This is a good moment to mention the on-line help facility. To get help in general, choose "help" in the MAX menu. This will give you a text window explaining quickly how to edit patches and

tables and how to use the other menu items. You also get a large patcher window with one each of all the built-in objects you can invoke.

To get help on a specific box in a patcher, hold the option key down and click once on the box. You will get a window which uses that kind of box in a simple, clear way, with comments and a short paragraph to describe it. When you are done with the window, you can click in its close box to get rid of it.

You can see what messages are appearing at a given outlet by connecting it to the inlet of a "print" object; for instance in the patch shown above, you could make a new connection:



to get a printout of the velocities of all the notes.

2. The patcher.

The patcher is the most important window type in . The contents of a patcher, called a patch, is a collection of boxes connected by line segments. The boxes represent objects which wait passively for something to happen to them, at which time they may respond by activating other boxes. They may have inlets and outlets, which appear as dark rectangles on top of and on bottom of the boxes. The line segments connect outlets to inlets, indicating that any message the source object puts on its outlet is sent to all the inlets connected to it.

The boxes can work as:

- controls, like sliders and buttons, which you click, drag, or type at to change things in a patch;

- indicators, which can inform you of a numeric value or that an event has occured;

- objects, which you describe in text, which do some kind of computation;

- messages, also in text, which are to be sent to other boxes.

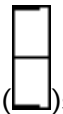
The messages can be symbols like "bang", numbers, or contain any combination of the two. The rest of this section will describe all this in more detail.

The graphical objects.

The patcher's ten types of boxes are divided into three groups: controls, i/o, and text. The controls are a momentary button ( ), a toggle button ( ), a slider ( ), and fixed and floating point number boxes (or); all have one inlet and outlet.

The momentary button appears in a patch as . It sends a "bang" message to its outlet whenever you click on it or send a message to it. You can use it as an indicator; it flashes whenever it is activated.

The toggle sends "1" or "0" alternately when you click on it. You can send it fixed-pont numbers to turn it on or off (on for nonzero; off for zero); it also sends the number to its outlet. The "bang" message toggles it, also causing output. A toggle appears as if it is on and otherwise.

The slider ()sends out fixed-point numbers when you drag it up and down; the bottom always gives 0 and the top gives 127 by default. You can send it fixed-point numbers to set its value, or "bang" just to cause it to send its value out without changing it. If you want to change its value WITHOUT sending any output, you can use the "set" message with an integer argument. To change the range of a slider, select it and choose "range..." from the MAX menu.

The number boxes ( ; ) display fixed or floating point numbers and repeat them to their outlets. As controls they are similar to sliders but are not limited to a fixed range, as sliders are. They take "set" and "bang" as sliders do. When you click on a number box, its appearance changes to "  " (for instance) for a few seconds. During this interval of time you can type at it to change its value. [When I fix this, the new value will be sent as a message only when you hit "return" or it times out.]

The I/O boxes are "inlet" () and "outlet" (). They only have a meaning when the patcher is imbedded in another; see section 6 on embedding.

There are three kinds of text boxes: "class"(), "message"(), and "comment"(). Each contains a text edit record; while the patcher is in "edit" you may select and edit the text. The "class" box allows you to type out a message which is sent to the "new" object in the underlying system; it should evaluate to a newly created object, for instance,  . If the text is changed, the old object is destroyed and a new one is created, as soon as you deselect the box. The number of inlets and outlets in a class box depends on the object it contains.

The "message" box also contains a message, but it is sent to the box's outlet every time "bang", a number, or a list is sent to the box. The message may contain variables which are set to the arguments of the incoming message. If the patcher is in "run", clicking on the message box is equivalent to sending it a "bang" message. Message boxes are described more carefully at the end of this section.

The "comment" box allows you to write text on the patch for labels and comments.

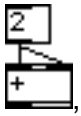
What patches do.

The boxes in a patch do their computation only when something happens to them, usually as a result of a message coming down a line to one of its inlets, but sometimes because you click on the box or a timer goes off. When any of these happen, control is passed to that object which might: send messages down lines connected to its outlets, draw something on the screen, and/or set a timer.

Messages which go down the lines (or any other message in the system) consist of an ordered list of atoms; an atom can be a fixed-point number, a floating-point number, or a symbol. (Internally, MAX will prepend a special symbol to messages which start with a number, but you won't see this unless you look under the hood.) A slider, for example, sends integers like "123". The most frequent messages sent and received by boxes are numbers (fixed or floating point), lists of numbers, and the symbol "bang" (widely used as a trigger.) MAX is heavily optimized for sending these particular messages around.

By convention, when an object has more than one inlet, its leftmost inlet is the "active" one; sending a message to that inlet causes something to happen, and sending messages to the other inlets simply changes the state of the object. For example, the box  multiplies two numbers

which are taken from the two inlets (call them x and y). The values of x and y are initially 0 and 10. When the number 5, for example, is sent to the left inlet, x is set to five and the box sends the product x*y, which is 50. If you send the number 3 to the right inlet, y is set to 3 but nothing is output. Now send 2 to the left inlet and you will get 2*3 or 6.

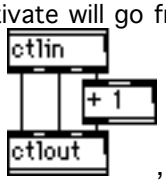
It is frequently important to get messages to their destinations in the right order, with the information for the leftmost inlet arriving last. For example, the innocent-looking patch, 

, is not guaranteed to put four on its outlet every time you activate 

. It might be that the 2 is sent to the left inlet first, while some other value is still in the right inlet; it will put out an incorrect value. Even worse, it will immediately hide the evidence, setting its inlet to 2 because of the second "2" it receives. In general, it is not defined in what order an outlet sends a message down all the lines connecting it to inlets, and this order might change unexpectedly in a patch that "used to work." There are several ways you can ensure the order of two messages; the sample patches at the end of this manual show the most frequently used ones. (There are two exceptions to the "right-to-left" inlet rule: 

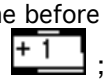
and 

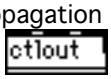
; see their "help" windows.)

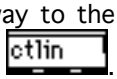
By convention, boxes with more than one outlet to activate will go from right to left. Thus it is often possible to make unambiguous connections like 

, which repeats all incoming MIDI control changes to the MIDI output with the channel offset by one. There is no order problem for 

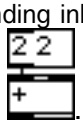
because it gets its leftmost value last, after the other values are set correctly.

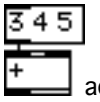
This example brings up an important point: message-passing in Max is synchronous, in the sense that everything that depends on an outlet will be done before the outlet itself is considered done. There was no propagation delay to get through the 

; the result gets all the way to the rightmost inlet of 

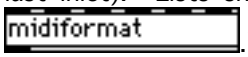
before control is passed on to the middle and left outlet of 

. The only way to put something off until later is to use a delay.

If you send a list (any message starting with a number but with extra arguments) to the leftmost inlet of an object, the items of the list will be sent to the corresponding inlets, right to left. Thus to get out of trouble in the 2+2 example above you could use 

. Extra arguments are dropped and missing ones taken from the rightmost inward, so 

adds 3 and 4, and 

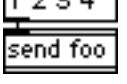
sets control 45 (the middle inlet is control number) to 23 without changing the MIDI channel (the last inlet). Lists should only be sent to the leftmost inlet of a box. (A minor exception is 

.)

Some objects (message boxes; 

, 

, and 

) handle lists specially to be able to take more items than they have inlets; thus 

sends "1 2 3 4" to the channel "foo" and not just "1" as it would if it handled lists normally.

By convention, boxes which have more than one inlet can take optional arguments to set the "value" of the inlets. Examples are: + 2 (and all the other arithmetic boxes), delay 50, select 3, split 60 72, makenote 60 250, and several others. The intent is to save work when the value in question will often be left constant. Inlet-setting values are always optional (makenote 60 250 could be used as makenote 60 or simply makenote; a missing argument means zero). Even if you initialize the inlet's value you can override it with a message. There are several exceptions to this convention, usually because it made no sense to leave the inlet's value constant; for example, in noteout 5 the "5" refers to the channel (the rightmost inlet), because you would almost never want to specify a constant velocity.

More on message boxes.

A message box really can contain more than one message; the only limitation is that there can be no more than 256 atoms in a text box. The messages are separated by commas (,) and/or semicolons (;) and can go to the message box's outlet or elsewhere. If two message are separated by a comma, the second takes the same destination as the first; if by a semicolon, you must start the new message with the name of an object to send it to. For instance, the message box: walk the dog, water the horse sends the two messages, "walk the dog" and "water the horse" to the box's outlet; whereas walk the dog;
fred water the horse, milk the cat sends "walk the dog" to the outlet and "water the horse" and "milk the cat" to whatever object is named "fred".

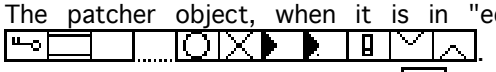
Only certain types of objects can have names: send (actually send/receive/value sets), and table. These objects name themselves, so that they can clear the name when they are destroyed. The send/receive/value and table classes actually create one object shared among all of them with the same name; thus s fred ("s" is short for "send") and r fred ("receive") share an object actually named fred. If you have a send/receive/value named "fred", you may not have a table fred or vice versa. Anyway, if you have a r fred and if you activate the box, walk the dog;
fred water the horse, milk the cat, the messages "water the horse" and "milk the cat" will be sent to the outlet of r fred.

You do not have to send anything to the outlet of a message box; for instance, ;
MIT 5;
CalTech 3 sends the message "5" to MIT and "3" to CalTech. You can also use the box only to send several messages to the outlet; for instance 1, 2, 3, 4 sends the messages "1", "2", "3", and "4" to its outlet, one after the other.

To activate a message box, you may click on it or send "bang", a number, or a list to its inlet. If you activate it with a number or a list, you may use the information you receive in the message via the special arguments $1 through $9. If you send the message box a number, $1 becomes that number and $2-$9 are 0 (fixed-point). If you send it a list, $1 through $9 are replaced by the first nine arguments of the list; if the list has less than nine arguments the remaining $s are replaced by 0. Thus, if you send bonk $1 the message "23.7" it will send "bonk 23.7" to its outlet; if you send 144, $1, $2 the message "60 100" it will send the three messages "144",

"60", and "100" (note that this is a very simple way to format raw MIDI messages out of a stream of notes.)

3. Editing patches.

The patcher object, when it is in "edit", presents a window with a menu at the top:  . The first item,  , is the edit button. You may turn editing on and off by clicking on  or with the "Edit" item from the "MAX" menu. When the patcher is in "run" (i.e. when it's not in "edit") the rest of the menu disappears.

You can create a box by choosing it on the menu, and drag it whereever you want it. The  button controls whether the patcher's configuration may, at the moment, be edited (the "edit" state), or whether further gestures from the mouse and keyboard are to be taken as real-time controls.

You can connect two boxes when the patcher is in "edit", by clicking on an outlet of one and dragging to an inlet of the other (actually you can let go anywhere in the box; the patcher will connect to the nearest inlet.) The outlet sometimes declares to the patcher what messages it will send; if this is incompatible with the inlet the connection will not be made. (Sometimes this can't be determined ahead of time and an error will be printed when the incompatible message is actually sent.) To delete a connection, select the line and choose "cut" or "clear" from the Edit menu.

When the patcher is in "edit", you can select boxes and lines by clicking on them; shift-clicking adds or deletes an object from the current selection. Cut, copy, paste and clear work from the "edit" menu in the usual way. If you copy and paste a collection of boxes, the lines connecting two copied boxes are also copied. You can move boxes by dragging them; dragging the mark on the right side changes the size of the box. If other boxes are selected they move or grow in parallel. Lines move only as a result of boxes moving or growing.

You can also move one or more selected boxes by using the arrow keys.

You can select the text inside a text box, if the text box is the only selected box, by clicking on the text. The usual Macintosh text editing rules apply. Changes you make in the text become valid when you deselect the box. If you choose "cut" when a portion, but not all, of the text in a box is selected, you will cut only that text and not the whole box. Even if the whole box is cut the text is cut too; thus when you "paste" the patcher can either paste the whole box or else text into an existing box. Only the text is pasted if, again, you have selected part (or even none) of the text inside some box. [this should be changed].

Whenever you copy from a patch, the result is also available as a picture which you can paste into another application. You can paste text from another application into a text box of a patch, but (sorry) you can't paste in pictures.

When the patcher is in "run", you can double-click a 

or 

to open its window. You may click on any box with the option key down to get help on the box; alternatively, you can select it and then choose "help" from the Max menu.

To summarize about clicking: click to select and drag (in "edit") or to use controls (otherwise); double-click to open subwindows (only in "run"); and option click for help.

Menus.

puts up six menus, "File", "Edit", "New", "Max", "Size", "Windows". The first two are as usual in a Macintosh application; "New" is for creating new windows, "Max" is for editing and setup

commands specific to MAX, "Size" sets the size of a piece of text, and "Windows" is for controlling the window layout.

The File menu items "open", "close", "save", "save as" and "quit" are just as in any Macintosh application. Open will decide automatically whether to open a file as a text editor, table, or patcher; you may open a file specifically as text with the "open as text" item, whether it is text or binary. The "make text files" item specifies whether the window is to be saved as text or binary. Files are binary by default; "text" is much slower. "New" makes a new patcher window.

In the Edit menu, the "undo", "cut", "copy", "paste", and "clear" items are as you would expect them (except that "undo" is useful only for patchers and not for tables or text editors.) The "duplicate" item is only for patchers and makes a copy of all selected boxes which is slightly offset from the originals.

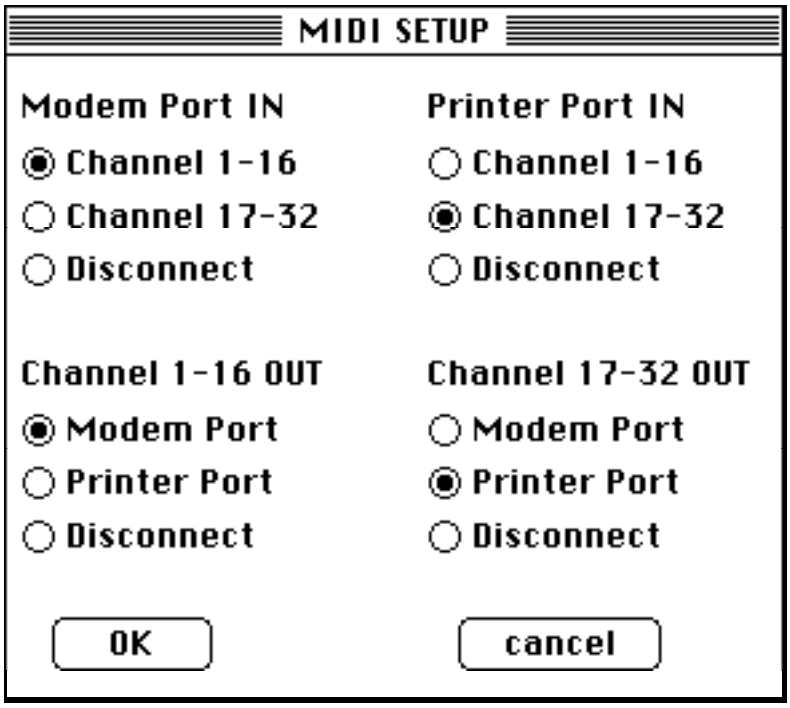
The New menu is for creating new patchers, tables, or text editors.

The Max menu contains "edit", "fix width", "line up", "range...", "midi...", and "help". The "edit" item puts the topmost patcher in "edit" or "run" mode. "Fix width" takes one or more text boxes and sets their width to match the text inside. "Line up" lines the selected boxes up either horizontally or vertically (it guesses which one you meant.) The "range" item lets you change the range of a slider or several parameters about a table; see section 7. The "midi..." item gives you a midi setup dialog; see below under "midi setup". The "help" item puts up a text window summarising the editing commands and a "help" patcher with one of each of the classes. For help on a specific class you can option-click it in the help patcher (or wherever else you find it.) In the patcher, if you choose "help" with a box selected you will get help on the box.

The "Windows" menu has "lock", "compact", and "send back" items, and an item to bring forward each open window on the screen. The "lock" item locks the windows where they are; you can then go between windows without having to activate them first. All menu equivalent keys are ignored when the windows are locked. To unlock, click anywhere in the menu bar. The "compact" item closes all windows which are owned by some other window: i.e. subpatches and tables which are owned by a patch. "Send back" sends the frontmost window to the back.

Midi setup.

The Midi setup dialog (which you get by choosing "midi..." from the "MAX" menu, appears as:



Using this dialog window, you can map incoming MIDI to channels 1-16 or 17-32, according to which serial port it comes in on. ("Channels" 17-32 are used simply to let you distinguish between the two ports). You can also select which channel ranges go to which serial port on output. Note that MAX leaves both serial ports open, even if you are not using both of them.

4. Messages.

MAX is a message-passing system, and to fully understand how it works you will have to know what the messages contain and how they are acted upon. A message is a collection of atoms which can be numbers or symbols. When you type a message to MAX, the first thing it does is to convert the text into a list of atoms. This chore is handled by the *reader*. The reader is called on the text in a box whenever you deselect it; if you open a patch from a file it is called on the whole file.

The reader.

Messages are separated by semicolons or commas. Within a single message, the atoms are separated by white space (carriage returns, tabs, and/or spaces in any combination.) After the reader separates the text into a list of strings, each string is converted into an atom. If the string contains only digits (like "123"), or is a hyphen followed by digits ("-123"), the atom will be an integer. If in addition to the digits there is exactly one decimal point (like "12.", ".004", or "-13.88") the atom is a floating point number. If neither integer nor floating point number, it's a symbol. Beware: don't put any white space anywhere inside a number (as in "- 1"); it will be read as two atoms.

The backslash character "\" removes the special significance of a following special character (any of ",", ";", "$", "#", or another "\".) Backslash followed by any other character is read as that character. If you really want the symbol "12" (you probably don't) you can type "\12". Sorry: you can't use backslash to make a symbol with white space in it.

Symbols in MAX have a name and a possible binding to an object. The only objects in MAX which can normally be named (i.e. have a symbol bound to them) are channels (shared by "send",

"receive", and "value" objects) and tables (see section 7.) In addition to naming objects, symbols are used as message selectors and occasionally just as strings.

The interpreter.

Whenever you open a patcher from a file, evaluate text from the text editor, change the contents of an "object" box or evaluate a "message" box (by clicking on it when the patcher is in "edit" or by sending a bang, number, or list to its inlet), Max calls its *interpreter*. The interpreter takes the reader's output, which is a collection of messages, and sends them to the appropriate objects. A message consists of a destination, a selector, and any number of arguments up to 256. For example the message "cis boom bah" has "cis" as a destination, "boom" as selector and "bah" as the only argument.

The destination is a symbol which should be the name of an object (i.e. bound to that object.) The destination is omitted in two cases: in a message box it is understood to be the box's outlet, and if two messages are separated by commas, the second takes the same destination as the first.

The selector is a symbol; it too may be dropped if the first argument is a number. (In this case the selector is given as "int" or "float" if there is only that one argument and "list" otherwise.)

If the text to be interpreted is in a message box, the interpreter is called with the outlet as its destination. If a destination is given, the message should contain a selector (if needed) and any arguments. If no destination is given (for instance when loading a patch from a file) the message should start with a destination and continue as before. If a message ends in a comma and there is another message afterward, it is given the same destination as the earlier one (and hence you shouldn't retype it); if it ends in a semicolon the destination is cleared before the next message is interpreted (and hence should be specified in the next message.)

The interpreter also expands arguments named $1, $2, ... into whatever arguments the interpreter was given; hence if a message box gets "1 2" it sets $1 to 1 and $2 to 2.

The message sender.

The message is finally sent to its destination by the "messenger". This is done following the instructions in the object's entry for the message. The entry depends only on the selector and the destination's class. (Part of that class is a list of message entries, each with a selector, an associated function, and a type list.) The sender looks up this entry (if none it is an error), checks the types of the message against the types specified by the type list (it's an error if they don't match), and calls the function with the given arguments. The function is sent a pointer to the object as an "understood" first argument.

In checking the types of a message's arguments, the sender will automatically convert float to int and back as needed. An object's type list for a message can also declare optional arguments, which default to 0 or to the empty symbol "".

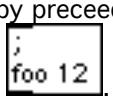
Inlets and outlets.

An outlet is an object which maintains a list of all the inlets it is connected to. Whenever an object wishes to, it can send one of its outlets a message, which the outlet sends on to all the inlets. An object may have any number of inlets, and can also act as an inlet itself (most objects do.) If the object is one of its own inlets, it is the leftmost one. Other inlets are objects of the "inlet" class. These true inlets are really message translators; they take an incoming message of a specific type (almost always integers or floating-point numbers, but occasionally "bang" or "list"), translate their selector to a prearranged symbol, and then pass the message on to the

owner. Thus an object can arrange to have several fixed-point inlets, for example, which can cause different actions.

By convention, an object uses its inlets to set up parameters, and itself (i.e. its leftmost inlet) to cause actions. This is the most sensible choice, since it is only the leftmost inlet which can take more than one different message selector. (But note that the inlets can do int/float conversion as needed.)

Messages and text boxes.

You can change the contents of a message box with the "set" and "append" messages. The arguments can be any mixture of symbols and numbers. You can even put commas, semicolons, or dollar signs in by preceeding them with a "\"; for instance the message "set \; foo 12" will set a message box to 

.

A cross-hatch has a special meaning in a text box; it is useful for editing your own classes. See section 6.

5. Built-in classes.

Max comes with about sixty built-in classes which you can access from a "class" box. They divide roughly into these groups: messengers, arithmetic, clock, subwindows, midi, nonlocal, and other. It is possible to add new ones; how to do this is very roughly described in the document, "Inside Max". This section will just summarize what exists; see the appendix for complete descriptions.

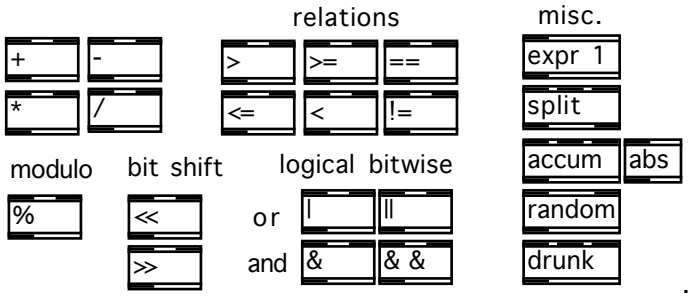
Messengers.

The messengers are: 

. Int and float send integers and floating-point numbers to their outlets; they can store values to send out later. Pack and unpack let you put together and take apart lists. Trigger and bangbang puts out up to ten messages, right to left, useful for solving order problems (trigger is more general; bangbang only puts out "bang" messages). Swap switchs the order of two messages, in case you want to activate another box but you do not get the value for its leftmost inlet last.

Arithmetic.

The arithmetic operations are:



.

There are the usual +-*/ (addition, subtraction, multiplication, division); the relations (which put out 1 or 0 depending on whether the result is true) "%" to get the remainder from a division; << and >> to shift a number left or right; |, ||, &, && for logical and bitwise "or" and "and"; "split" to do a "keyboard split", "accum" to accumulate sums or products; "abs" for absolute value; "drunk" and "random" for two kinds of random number sequences. "Expr" lets you evaluate c-like expressions, with access to library functions.

The "binops" (binary operations), which are everything except split, accum, abs, and drunk, can take an optional argument to set the initial values for their right inlets, as in + 15. To make your life slightly easier, they also take "bang" on the left to mean, "do the operations with whatever values you have", so that you can change the right inlet and bang the left one to get the output.

The +, -, * and / may be forced into floating point by giving a floating point argument as in + 0. or + .3. In this case the outlet will also be floating point. (You can always send floating point values to a fixed-point operation but it will be converted to an integer before the operation ever sees it.) The six relations may also be made to do floating point comparisons as in >= 0.; in this case, however, the outlet is always fixed-point. The other binops are all fixed point only.

Clock.

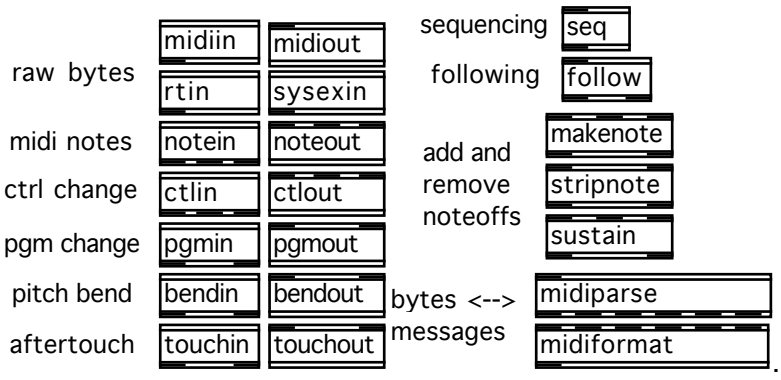
The clock classes are: timer delay pipe metro speedlim line. The timer reports the time in milliseconds between two incoming messages; delay and pipe are simple and complicated delay lines, metro is a delay loop that makes a metric pulse, speedlim slows down the speed of a changing number, and line generates smooth ramps from one value to another.

Subwindows.

The subwindows are: patcher table. They create an embedded window (either another patcher or a table.) You can put your own inlets and outlets on patcher; see section 6.

MIDI.

The MIDI classes are:



At left are the various sorts of MIDI input and output, which goes over the modem or printer port. (The printer port appears as MIDI "channels" 17 through 32.) Notes have a pitch, velocity, and channel which correspond to the three inlets of noteout or the three outlets of notein, and so on for the other message types. At right, seq is a MIDI sequencer (raw MIDI bytes in and out), follow is a score follower (pretty complicated; see section 9); makenote and stripnote are for

adding or removing note-off messages from a note stream; midiparse and midiformat convert raw MIDI bytes to and from their real numeric information.

Nonlocal.

The nonlocal classes are: send foo receive foo value foo (you can replace "foo" with any symbol.) These three allow you to move information from one place to another without the need for a connecting line. Any message you send to s foo, for example, will appear in the outlet of all the r foo boxes in the system. (You can use "s", "r", and "v" as abbreviations for "send", "receive", and "value".) You can use v foo as a global storage cell for one number, which you can set from anywhere and read from anywhere else.

Other.

The rest is:

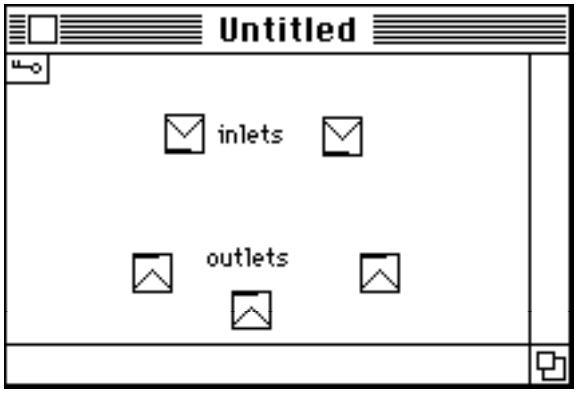
select a number	select
fancy "select"	route 1
filter out repetitions	change
open/close a connection	gate
polyphonic voice alloc	poly
keyboard	key
printout	print
set fun	bag
linked list	funbuff

select filters a stream of numbers for a specific value or values, putting out "bang" for each match. route 1 is similar but it can also select for symbols, and can pass the rest of an incoming message after stripping the selected item off. change filters out the repetitions in a stream of numbers. gate allows you to make and break a connection dynamically. poly does polyphonic voice allocation. key gives you access to the Macintosh keyboard. print prints out any message it receives onto the MAX window. bag maintains a set of integers which you can insert/delete and dump. funbuff remembers a set of (x, y) values which can grow and shrink dynamically; it is useful for sequencing a control value.

6. Embedding patches.

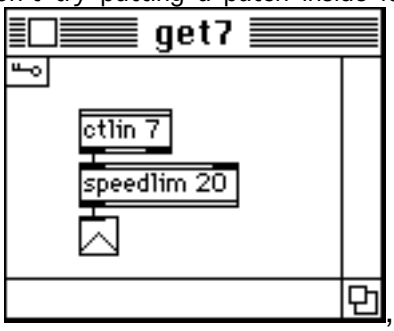
If you have a patch that is too big for one window, you can put parts of it in sub-patchers. In the main patch, these appear as class boxes: patcher. When you first type the word "patcher" in the class box, a new patcher window appears on the screen. If you close the sub-patcher window, the sub-patcher will become invisible but it will still exist; you can make it visible again by double-

clicking on patcher. To communicate with the sub-patcher you can give it inlets ( ) and outlets ( ) which will appear on the box; for instance if the sub-patcher is

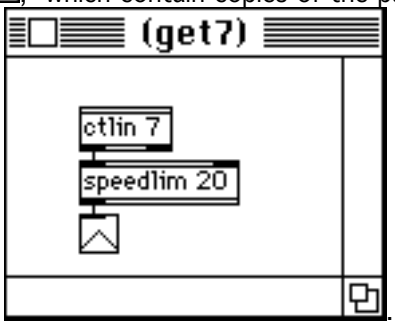


the box will appear as patcher. If you send any message to an inlet on patcher the message will appear on the corresponding in the sub-patcher; if you send a message to any there it will come back out of patcher. Sub-patchers may in turn have their own sub-patchers, and so on to any depth.

You can also use a patch as a new class; if you type the name of an existing patch in a class box, will read in the patch and embed it in the box. (Don't try putting a patch inside itself, though.) For instance, if you have made and saved a patch,

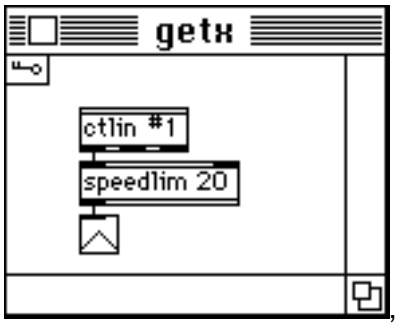


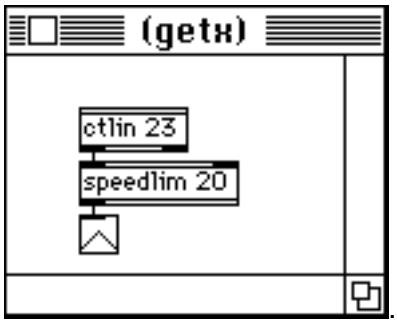
, you can then make class boxes, get7, which contain copies of the patch "get7". If you double-click on get7, you will get a window:



.

You can use arguments to vary the contents of the window; for instance, if you make a file with the patch,



, then getx 23 will give you 

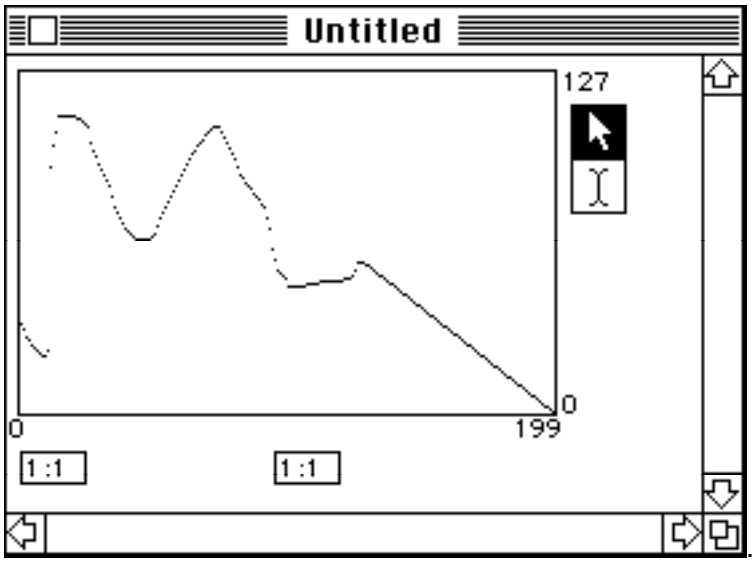
. In this example, the new class "getx", which you have created, takes an integer argument and reads in MIDI control values, downsampling them so as not to exceed 50 per second.

The two types of embedded patchers save differently to files; if you have an unnamed sub-patch (as in patcher) the window will be untitled and the patch will be saved as part of the containing patch. If you have a class like getx 23, you will not be able to edit the sub-patch; you have to open it as its own file-window. The meaning of the parentheses in the titles of the sub-patches is that they do not correspond to files.

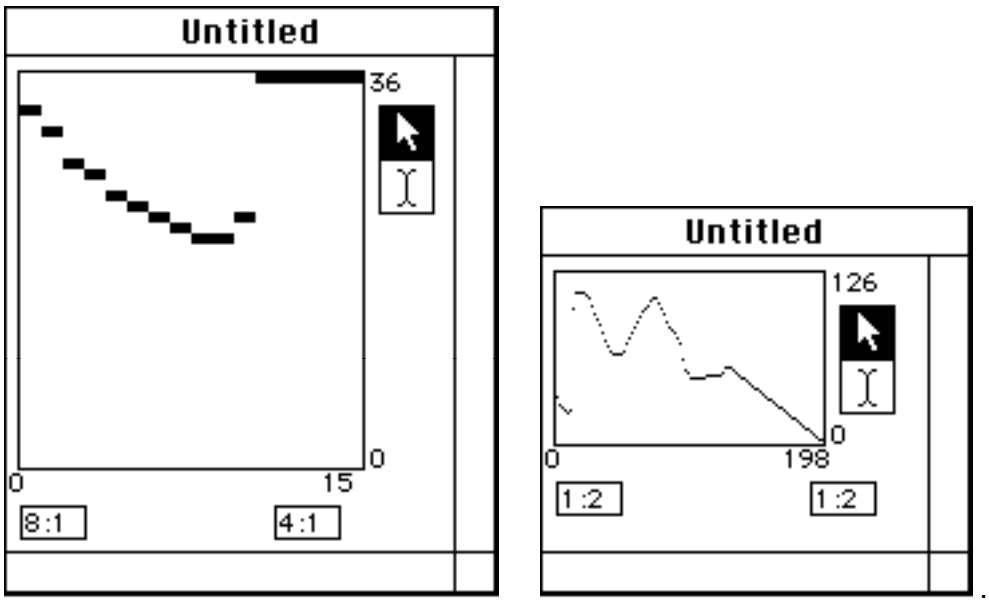
The files for class patches should be either in the folder where the owning window is, or else in the "max-classes" folder, itself in the same folder as Max. If you are editing a window which you have not saved, it cannot know what folder you will save it into; you should save it first if you want it to find classes out of its own folder.

7. Function tables.

The box, table, creates an embedded function table window; you can double-click on table to see the window, or else make a new one from the "New" menu, for instance:

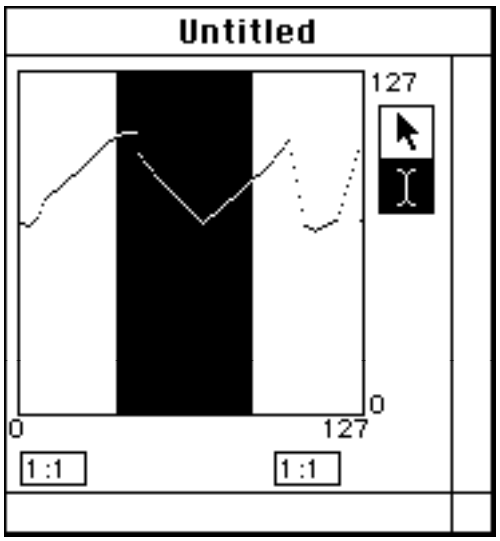


You can edit the contents of the table by drawing with the mouse; if you shift-click it will make a straight line segment from the last point you had drawn. You can drag up and down in the 1 :1 boxes below the table to zoom in and out, independently in x and y; for instance the same table could appear as:



or

The  palette selects whether you are drawing in the table (as above) or selecting/cutting/pasting in it, as in



You can shift-click to extend a selection. When you paste into a table, it pastes before the selected region without deleting it; the selection remains as it was so that you can now hit "cut" or "clear" if you wanted to delete it. Copy and paste works between tables and text. If you copy from a table and paste into a text-editor or box, you will get a printout of the table's contents. You can copy any collection of white-space-separated numbers from text back into a table. Sorry, tables do not copy to draw or paint programs.

If a table is imbeded in a patch and you send an integer to the left inlet of table it will put the table's value at that location (another integer) on its outlet. You may set table values (x, y) by sending x and y as a list or by putting y in the right inlet before putting x in the left. A table can have a name as in table baz; all the tables with the same name are identified, so that you can write into a table in one part of a patch and read from it in another. The table's window then is actually named "baz" as an object in Max; you can send messages to "baz" nonlocally from a message box.

You can set many values of a table at once with the "set" message, giving the first x value followed by all the y values; for instance, the message "set 3 23 34 45" sets the table values for 3, 4, and 5 to 23, 34, and 45. The table will not grow or shrink as a result of "set". The message "size" with an integer argument sets the number of elements in a table.

You can do inverse table lookups: if you send the table "invtab 5" it will report the first x value where the table's y value is at least 5. (This works well only if the table is nondecreasing.) You may also do quantiles (this is experimental!); you send the table "quantile 1000" and it returns the x value of the 1000th quantile on a scale from 0 to 32767. If you just send a "bang" the table will return a random quantile; i.e. an x value chosen so that the table's y values give each x's proportional probablility. Funny things might happen if the table contains negative values.

When a table is created with a name, as in table baz, if there is not yet a table named "baz" it will look for a file named "baz" in the folder the patcher belongs to. (If the patcher does not yet have a file to save to, the folder is taken to be where Max was found.) The contents of a table are thrown away without a complaint when you dispose of it; if you really want to keep the contents of a table you must remember to save it explicitly. Also, the table will not be saved just because you save a patch which uses it.

The "range" item in the "Max" menu, when selected with a table as the top window, puts up a dialog window that lets you change the size of the table, its range, and whether it is "signed" and "scratch". If the table is "signed" it is displayed with y=0 in the middle of the vertical axis; otherwise y is zero at the bottom. If it is "scratch" it will disappear without warning you if you close it after changing it. Range, singned, and scratch are remembered by the owning patch, not the table; if you want your choices to me remembered, save the patch, not the table.

8. Text editors.

The "new" item in the File menu creates a text-editor window; also if you use "open as text" on a Max patcher, table, or any text file you will get an editor window on it. If you try to "open" a text file that does not start with the word "max" or "table" (or, for compatibility with old versions, "new" or "_N") it will assume that this is not a Max file and open it as text. The text editor window has a limit of 32767 characters.

9. Sequencer/follower.

Complicated. You can find a good description of these objects in Philippe Manoury's manual.

10. Files.

The patcher saves itself to a file by writing an interpretable text which evaluates to a copy of the patcher. (This mechanism is also used for cut and paste.) The patcher asks all non-built-in objects to save themselves if they have methods to do so; hence all objects could in principle restore their states entirely. In practice, this is done only for objects which contain a large amount of data: tables and sub-patchers.

Any file (patcher or table) can be written in a machine-dependent binary form which can be opened more quickly than text. The "use text files" item in the File menu, if checked, indicates that the frontmost window saves as a text file; if unchecked, it will save to a binary one. The ascii version is designed for machine and OS independence.

11. Obsolete features.

If you are using any of the boxes:

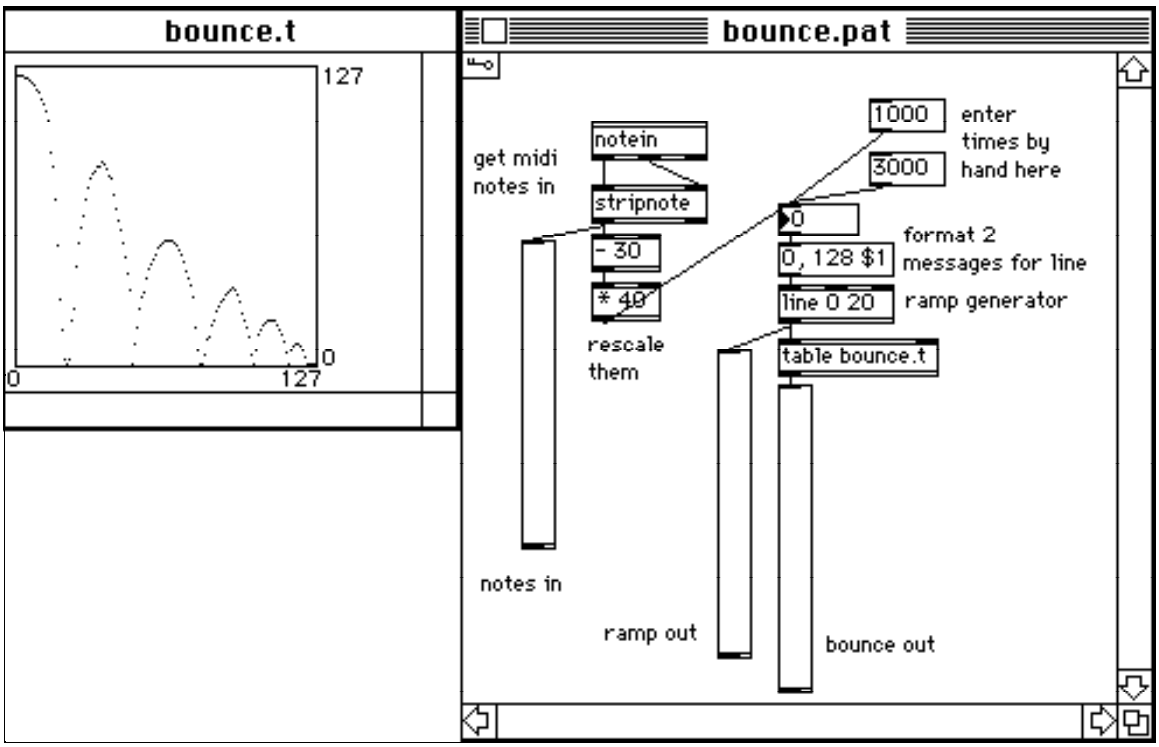
you should replace them with their equivalents from section 5. Also, if you use the messages "fix" or "fixfix" you should erase the "fix" or "fixfix" and use only the number or list. Future releases of may not support these objects.

The whole sequencing setup is likely to change, but this is a serious project which might take a year or so. If it succeeds, you can expect "seq", "follow", "midiparse" and "midiformat" to be declared obsolete; nonetheless, since there will doubtless be many applications using them, they will be kept around indefinitely.

12. Example patches with commentary.

The patchers described here are all found in the folder "patches" which comes with the distribution. Each patch described here comes with an excercise or two; you can try them to some practical experience with .

Bounce.pat.



This patch makes a bouncing pot. The table "bounce.t" is owned by the patch. You can close it and reopen it by double-clicking on table bounce.t. If you click on 1000 or 3000 you will see the two pots at right move; the middle one ramping upward and the rightmost "bouncing". The box 0, 128 $1 formats two messages, "0" and "128 1000", if it is sent "1000", for example. The first makes line 0 20 jump to zero and the second makes it ramp to 128 in 1000 milliseconds. Finally, table bounce.t looks up the output of line 0 20 in the table "bounce.t", making the bouncing action.

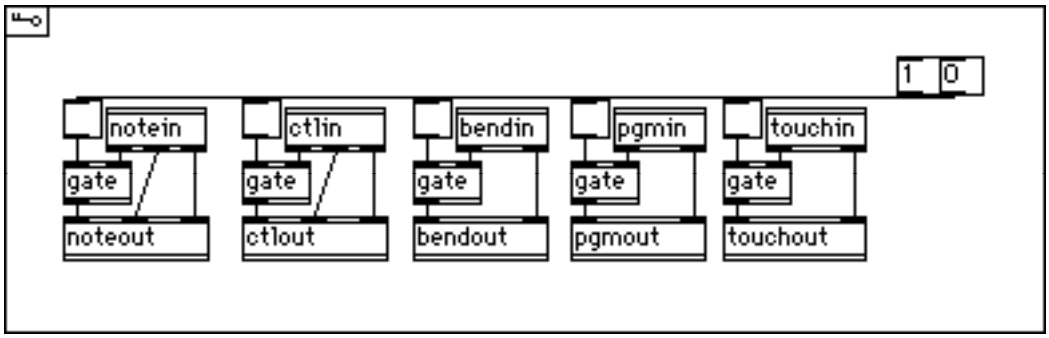
The left part of the patch takes incoming midi notes, and uses their pitch to do the same thing. The notein gets the incoming pitches and velocities of the beginnings and endings of notes. stripnote keeps only the note-ons (dropping notes with velocity zero). This patch uses only the first outlet, which gives the pitches. The - 30 and * 40 adjust the scale of the pitch to one reasonable for the right half of the patch.

Excercises.

1. (easy) get help on line 0 20 (or whatever) by option-clicking on it, to verify that Max can find the max-help folder.

2. (easy) Change the patch to use the "bounce" output to control pitch-bend on your synthesizer. (Use the bendout object to generate pitch-bend messages.) Edit the table to get it to sound like a short vibrato.

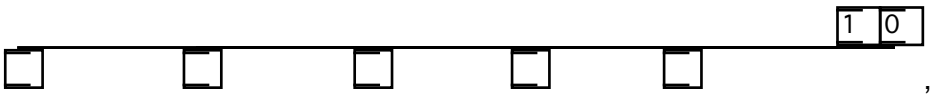
Midi-thru.pat.



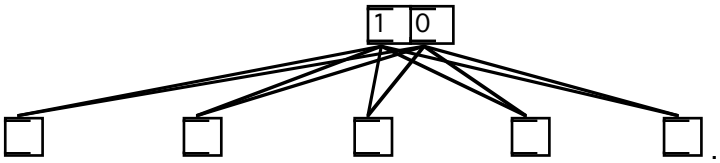
(Starting here I'll dispense with the window borders.) This patch just copies MIDI input to output. You can control the five message types independently. Notes, for example, work as follows: notein reports all incoming note-on messages (over either the Modem or Printer port of the MacIntosh.) The channel (1 to 32; normally 1-16 are the Modem port and 17-32 are the Printer port) appears on the rightmost outlet of notein, then the velocity on the middle, then the pitch on the left. The first two are put directly into the corresponding outlets of noteout but the pitch is sent through gate first, which will send it on only when its first inlet has received a nonzero value. (Notice that gate's inputs are not in the order you would expect; the left inlet opens and closes it and the right inlet is either passed through or dropped.)

The gate is only needed on the leftmost line for each connection; the inlets other than the left ones can receive any number of messages; only the most recent will count when a message is received in the left.

The message boxes at the top, 0 and 1, turn all the five toggles on (☒) and off (☐). You can also turn them on and off individually. (You can't see it but the horizontal line connecting them,



is really ten lines; if you drag the message boxes upward you'll get something like:



In the patch they are all lined up to avoid visual clutter.)

Exercises.

1. (easy) Add a feature to transpose MIDI notes; for instance, wire a + 3 into the pitch path. Make your own pitch-mapping functions by putting in a table instead.

Appendix. The help windows.

The help windows are being changed somewhat; I'll stick them in the next draft of this doc.