The Patcher

Miller Puckette

IRCAM, 11/8/88

Introduction.

The Patcher is a graphical environment for making real-time computer music, currently with midi-controllable synthesizers. Its main purpose is to control the instantiation and configuration of objects in the MAX system [Koechlin, et. al., 1986 ICMC Proceedings.] The main entities in MAX are windows, which may be text editors, patchers, and function tables. (In the future, maybe also a sequence editor and a 2-d function editor.) At the moment the Patcher is the most interesting window type in MAX.

MAX views a performance as a collection of independent objects which communicate by passing messages. It provides a set of subroutines for timing, midi I/O, fast memory management, and message-passing between objects. In addition there is a simple interpreter which converts text to messages and passes them appropriately. The current version runs on a Macintosh personal computer.

The bulk of the MAX system is a set of classes which define the objects themselves and how they interact. The classes are highly insular; the rest of MAX has very little knowledge about the inside of any given class. As a result, the classes can be changed, added to or removed, or new ones created, with little or no effect on the performance of the rest of the system. The design of MAX aims to make a wide range of performances possible with little or no need for additional programming; nonetheless it is easy to add new functionality to the system.

MAX has hooks for patching a digital synthesizer, which may just be (as now) an interpreting program in the Macintosh or may (in the future) be a plug-in signal-processing card. As a part of the work of controlling a digital synthesizer, certain objects in MAX create signal-processing elements in the synthesizer. These objects can intercommunicate by means of signals, special messages through which the objects will arrange to pass a signal in the synthesizer from one element to the other. Since this capability is still experimental, it will not be described here.

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 Jarrel, and Philippe Manoury, assisted by Thierry Lancino, Cort Lippe, and Jan Vandenheede.

Description of the Patcher.

The Patcher presents a visualization of an object in MAX as a box in a window, showing some of the state of the object. Each box in the Patcher's window has some number of graphical *inlets* and *outlets*. A connection between two objects is represented by a segment from an outlet of one object to an inlet of another. You can create a new connection by selecting any outlet of any object and dragging to an inlet of another object; a segment appears between them. An outlet may be connected to many inlets and vice versa. Figure 1 shows an example of a patch, with three "slider" controls controlling three parameters of a stream of notes to be sent over MIDI. Outlets of a box are always on the bottom and inlets

on the top. An object is usually taken to be its own first inlet, through which you may direct any desired message to the associated object; other inlets are for specific functions, so that standard messages may be used to cause a variety of different actions in the same object.

A box in the patcher executes code only when something happens to it, usually as a result of a message coming down a line to one of its inlets, but sometimes because the user clicks on it 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. (An important exception to the "only" is that some boxes (signal processors) run continuously, communicating over "signal" lines -- but we aren't describing that here.)

Messages which go down the lines (or any other message in the system) are a symbol followed by any number of arguments. The arguments may be fixed or floating point numbers, more symbols, and/or references to other objects. A fixed-point slider, for example, sends messages like "fix 123". Here "fix" is a symbol saying that what follows will be a fixed-point number, and "123" is the goods. The messages 'bang', 'fix', and 'float' are considered *standard*. They mean nothing but what the following types will be; hence they could really be considered equivalent to the data types. The selector "bang" takes no arguments and is usually used to trigger things. The system is optimized for standard messages; outlets work faster on them than on other messages. Another message, "fixfix", which holds two fixed-point numbers, is being phased out in favor of a more general facility. At any rate, you never need to type the words "fix", "float", or "fixfix" since these are automatically filled in by the interpreter when needed.

You can see what messages are appearing at a given outlet by connecting the outlet to the inlet of a "print" object; see for example figure 2. The printouts occurred when the slider was slid up and down.

You can do arithmetic as in figure 3. You may set the initial value of the second inlet for "+" by adding an argument to "+". You can get at function table windows as in figure 4. If you send integers (i.e. 'fix' messages) to the table box, the table's value for that "x" location appears on the box's outlet. You may set table values with "fixfix" messages.

Since patches are often too complicated to fit inside a single window, a mechanism for abstraction is provided, as shown for example in figure 5. There, the "untitled" window is itself the object associated with the "patcher" box in the window titled "imbed.max"; the boxes in "untitled" are of classes "inlet" and "outlet" and correspond to inlets and outlets in the "patcher" box.

Most of the functionality of the system is derived from the classes that can be typed into a "new" box. To be able to write interesting graphical programs, you need classes for flow control, arithmetic, timing, I/O, and signal processing. A library of these objects is provided with the system. It is possible to add new ones; how to do this is not described here.

Messages and the Interpreter.

Whenever you load 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 locked or by sending a standard message to its inlet,) MAX calls its *interpreter*. The interpreter converts text into messages; the messages are separated by semicolons or commas.

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 symbol (the message's selector) and any arguments. If no destination is given (for instance using "eval" in a text-edit window) 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.) For example in figure 6, the messages "foo bar" and "fix 123" go to the outlet (and hence to the "print"), and the message "ralph bang" goes to the "send". ("Send" has the special property that it creates a receiver object with its argument as a name.)

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

Editing.

The patcher object, in its initial state, presents a window with a menu and a lock button. The menu presents the classes of graphical objects available from the patcher. You can create an *instance* of a class by selecting it on the menu. The instance appears as a new shape drawn in the window. You can then drag the shape to any place in the window. The lock button controls whether the patcher's configuration may, at the moment, be edited (the unlocked state), or whether further gestures from the mouse and keyboard are to be interpreted as real-time controls.

To make a connection between two boxes the patcher must be unlocked. You click on an outlet of some box and drag toward some inlet, as in figure 7. If the inlet has no method associated to the symbol of the outlet and also has no "anything" method, the connection is not made; if the outlet has no symbol the connection is always made. To delete a connection, select it and hit "cut" or "clear".

When the patcher is unlocked, 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 width 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.

If you click in the text portion of a box while it's selected you will select the text; the usual toolbox text editing rules apply. Changes you make in the text are only internalized when you deselect the box. If the result is an error the box will get no inlets of outlets.

When the Patcher is locked, if you double-click a "new" box it will send a double-click message to its associated object, which may activate an editing window for it. In particular, if the box is an included Patcher or Table it will activate its own window; if it is another "new" object it puts up a help window for it. The Patcher saves itself to a file by writing an interpretable text which evaluates to a copy of itself. (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 can in principle restore their states entirely. A Patcher is recreated from a file by the native "load" command in the underlying system. The file may be ASCII or binary.

Built-In objects.

The patcher's nine types of boxes are divided into three groups: controls, I/O, and text. Examples of all the boxes, in the locked and unlocked state, are shown in figure 8.

The controls include a momentary pushbutton, a toggle button, a sliding potentiometer, and a number box; all have one outlet and are their own inlets. The momentary button sends a "bang" message to its outlet whenever it is clicked; the toggle sends "fix 1" or "fix 0" alternately when clicked. The slider and number box send out numeric messages which can be increased or decreased by dragging up or down; the slider slides a bar up and down and the number box prints out its value.

The I/O boxes are "inlet" and "outlet". They only have a meaning when the patcher is imbedded in another via a "new" box (see below). If the patcher is included in another patcher as a box, that box will have inlets and outlets corresponding to the inlet and outlet boxes in the included patcher. If, in the included patcher, you send a message to an outlet box, the message appears on the corresponding outlet on the "object" box that created it.

There are three kinds of text boxes: "new", "message", and "comment". Each contains a text edit record; while the patcher is unlocked you may select and edit the contained text. The "new" 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. If the text is changed, the old object is destroyed and a new one is created.

Normally the object that *new* creates becomes the object box's first inlet and all other inlets and outlets owned by the object are also drawn on the box. Some objects (for instance a Patcher) prevent this and manage their own inlets and outlets.

The "message" box also contains a message, but it is sent to the box's outlet every time a standard message is sent to the box. The message may contain variables which are set to the arguments of the incoming message. If the patcher is locked, clicking on the message box is equivalent to sending a "bang" message.

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

Non-built-in objects.

The set of non-built-in objects is still changing, but the following gives a nearly complete listing. The most up-to-date listing is given by the on-line documentation, which is being provided by Cort Lippe. Each class has a help window which is actually a patch featuring the class (figure 9.) There is also a general help window (figure 10) which lists the available classes. You can double-click on an object in this window (or anywhere else) to get the help window for that class. A rough list is:

1. Numbers: int, float. They have two inlets and two outlets; the left outlet sends a 'bang' after the right one sends the value. The right inlet changes the value without sending it; you can send it later by sending 'bang' in the the left inlet.

2. Message decomposition: pack, unpack, switch, bang. Unpack takes 'fixfix' and splits the arguments into separate outlets; "pack" does the opposite. Switch interchanges the two inputs in a way which inverts their order. Bang has two outlets; they both send "bang", the left side before the right.

3. Arithmetic:

fixed- or floating-point: +, -, *, /,

fixed-point only: &, |, &&, ||, %, ==, !=, !, >, <, <=, >=,

not yet implemented: sqrt, sin, cos, tan, asin, acos, atan, atan2, pow, exp, log.

The first two rows, and "atan2" and "pow" from the last row, are binops. If they are given an argument it is the default value for the right inlet (so that you can use the left one to add, subtract, etc, a constant.)

4. Filters: change, sel, gate. "Change" takes a fixed-point input and sends it to its outlet only when it changes. "Sel" selects messages with a specific numeric value and sends them through, changing "fix" to "bang" and "fixfix" to "fix" (stripping the first argument, acting only when it's equal to "sel"'s argument. "Gate" lets stuff through from its right inlet when "1" (or anything else nonzero) is sent to its left; not when "0".

5. MIDI: midiin, midiout, ..., follow, ms, midiparse, midiformat. Midi "channels" 17 through 32 refer to the printer port. Midiseq (actually "ms" and "oms") and follow are a midi sequencer and score follower. Midiparse and midiformat convert raw MIDI bytes to messages and back; thus midiparse has one inlet and 5 outlets (the midi messages), and midiformat has 5 inlets and one outlet.

6. Clock: del, metro, timer, speedlim, line. "del" sends a bang a given interval after a "bang" has come in; if in the meantime another "bang" comes in it reschedules its output to arrive later. "metro" puts out a metronomic pulse; timer acts like a stopwatch; speedlim downsamples an input to change no more frequently than desired; "line" makes breakpoint envelopes.

7. Note-oriented: note, d2nz, notegroup. "note" takes a "fixfix" and supplies, a given time later, a corresponding note-off "fixfix". "d2nz" filters out note-offs (i.e. fixfixes where the second number is zero.) Notegroup waits for a cue and extracts a tempo from it.

8. Misc.: table, patcher, funbuff, send, receive. "Table" and "patcher" are window types. "Funbuff" (by Zack Settel) maintains a linked list of (x,y) pairs for breakpoint functions or control sequencing. The "send" object creates an outlet in common to all "receives" of the same name, so that input to any send of that name appears as output to all of them.

Conclusion.

The system described here lets musicians choose from a wide range of real-time performance possibilities by drawing message-flow diagrams. These patches can grow large gracefully, because of a strong facility for embedding and interconnection between windows; the results can still be fast and reliable enough for stage performance. A rich set of objects is provided to do computation and MIDI I/O; their definitions are abstract enough to make them useful in many different situations.

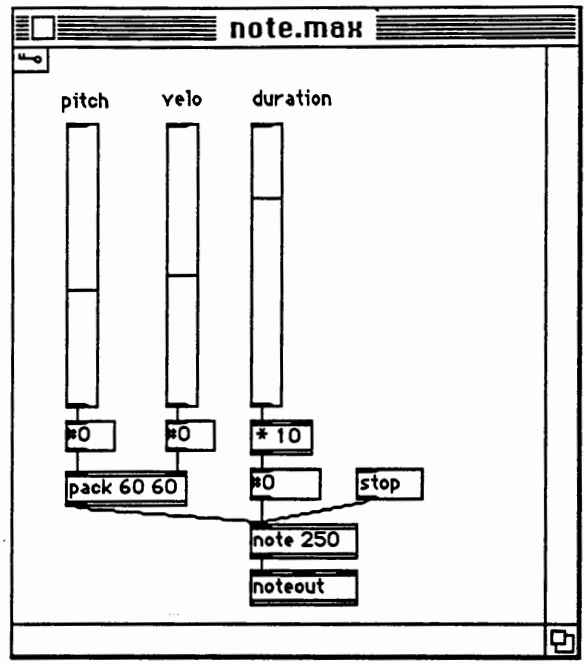

Figure 1. A sample patch.

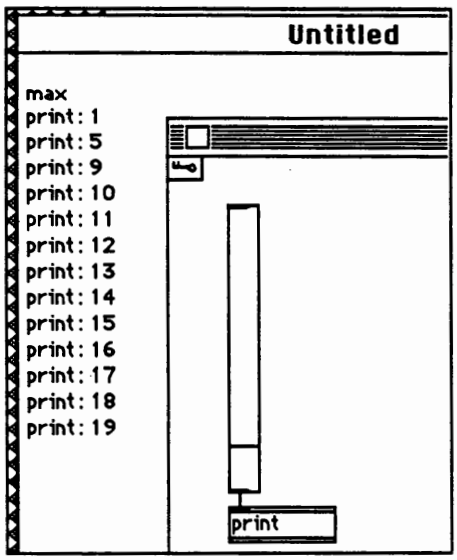

Figure 2. Printout.

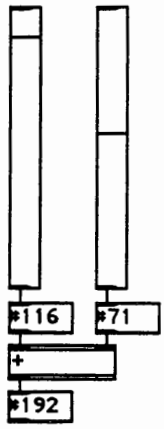

Figure 3. Arithmetic.

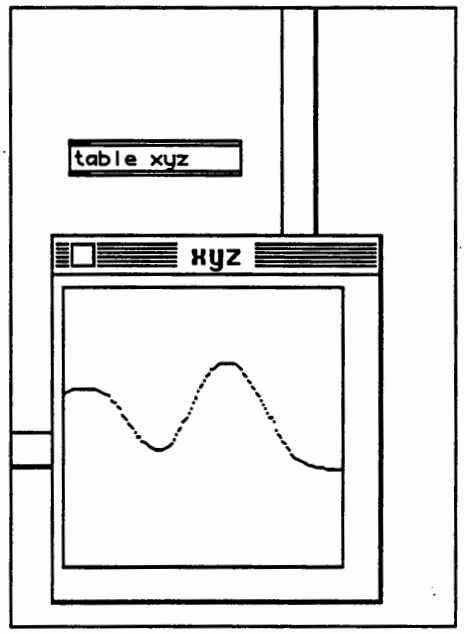

Figure 4. Function tables.

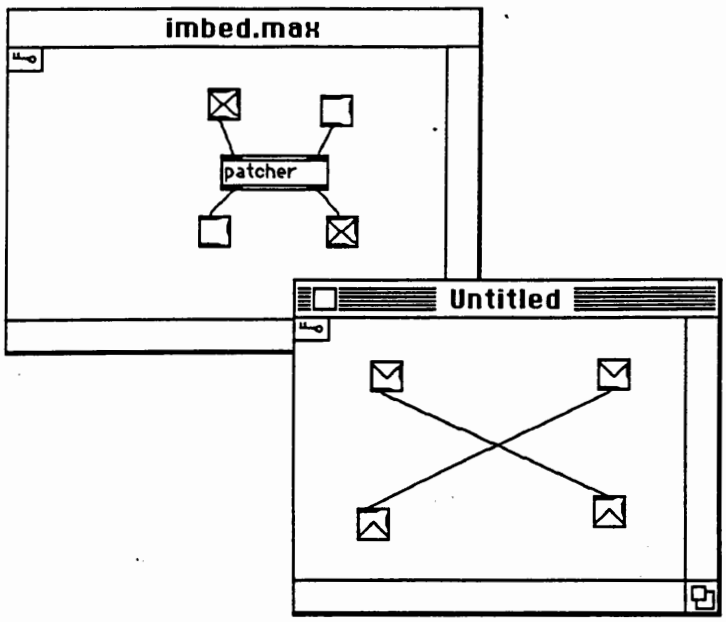

Figure 5. Imbedding.

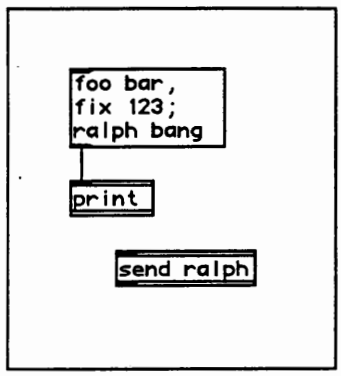

Figure 6. The interpreter.

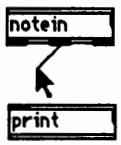

Figure 7. Editing.

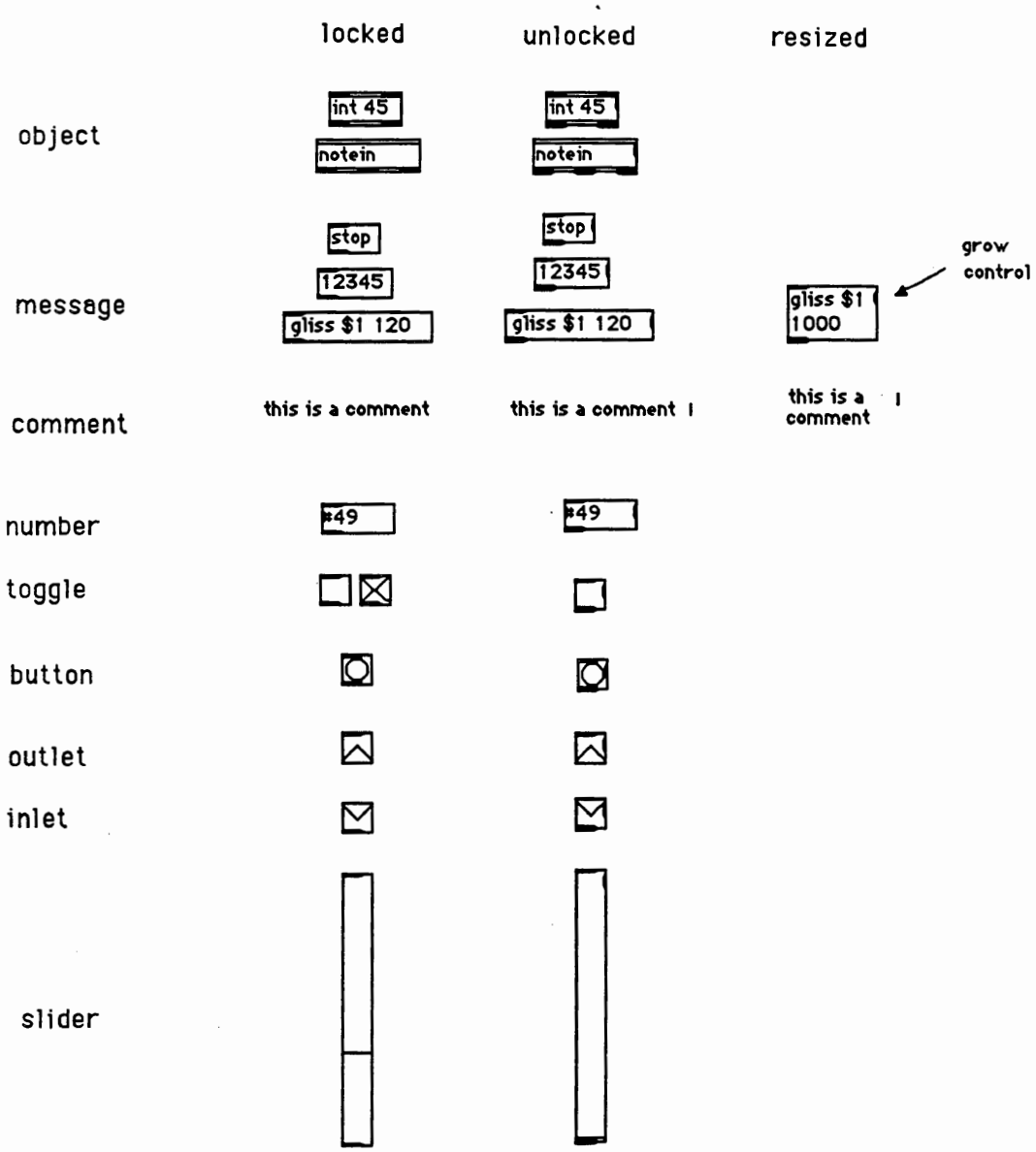

Figure 8. Built-in objects.

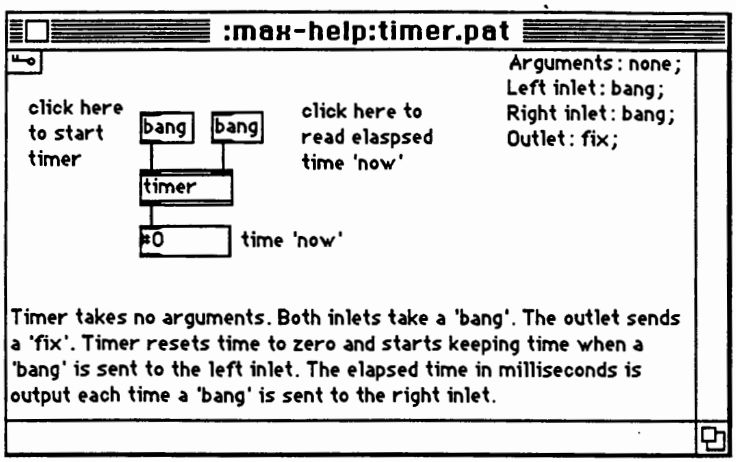

Figure 9. A help window.

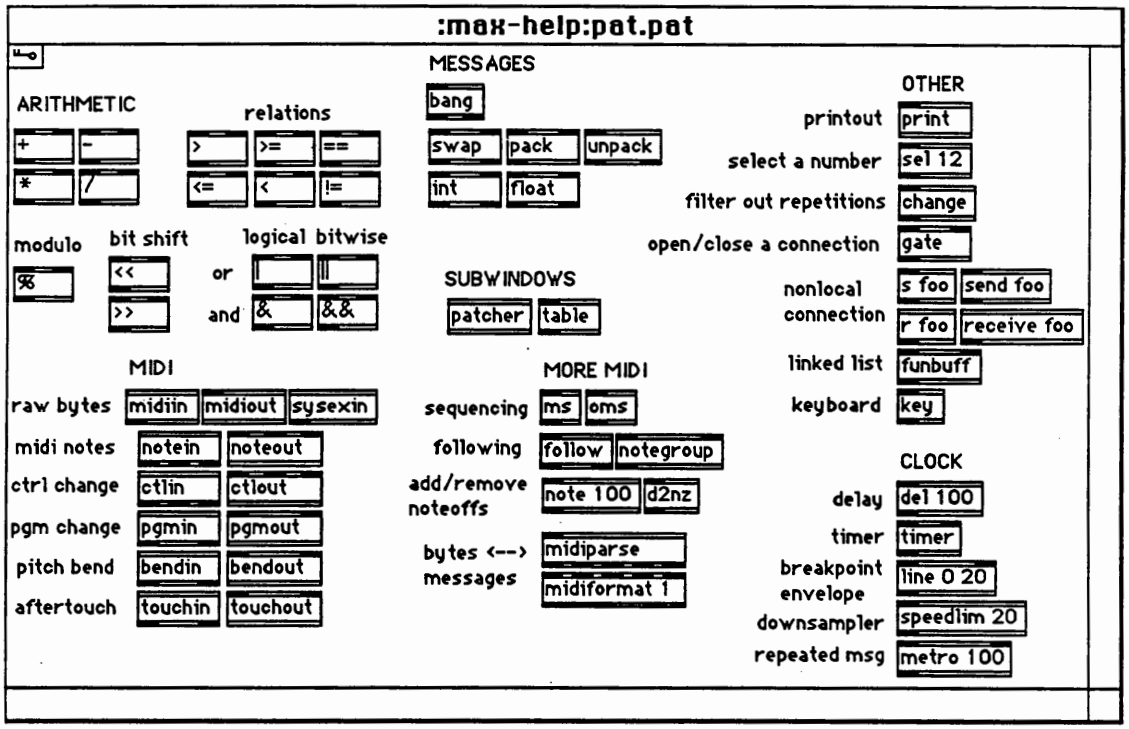

Figure 10. The main help window.