Combining Event and Signal Processing in the MAX Graphical Programming Environment

Miller Puckette
l'Institut de Recherche et Coordination Musique/Acoustique
31 Rue St. Merri, 75004 Paris, France
miller.puckette@ircam.fr

©1991 MIT. Reprinted from Computer Music Journal 15(3), pp. 68-77.

MAX (Puckette 1988; Opcode 1990) is a graphical programming environment for developing real-time musical applications. First written for the Apple MacIntosh computer, it has been ported to the NeXT computer as a part of the IRCAM Music Workstation ("IMW") project (Lindemann 1991a). From its earliest conception, MAX was intended as a unified environment for describing both control and signal flow. Historically it has developed as a MIDI (i.e., control) program primarily because the 4X (Favreau 1986), IRCAM's earlier signal processing engine, could only communicate with the MacIntosh over a MIDI (serial) line.

The IMW offers an opportunity to join MAX more intimately with a number-crunching engine capable of doing high-quality audio synthesis and processing in real time. The DSP cards now available for the MacIntosh line of computers, while many times less powerful than the IMW, offer a similar possibility. This paper describes how MAX has been extended on the NeXT to do signal as well as control computations. Since MAX as a control environment has already been described elsewhere, here we will offer only an outline of its control aspect as background for the description of its signal processing extensions.

The main purpose for making electronic music production run in real time is so that a musician can exercise some sort of live control over the music. The problem of defining that control is a much harder one than that of defining the signal processing network which ultimately will generate the samples. The sample generation problem has historically been considered "hard" simply because of its stringent computational requirements. Today, a real-time programmable audio synthesis and processing engine can be bought at a price that researchers, and even some musicians, can pay. It is therefore not surprising that many systems are now being proposed for graphical signal network editing; recent ones are described in (Bate 1990), (Minnick 1990), and (Helmuth 1990). But the control problem, that of making the signal network respond in an instrument-like way to live human control, is not made appreciably easier by the availability of faster and faster hardware. Today, the challenge for a signal processing network editor is to open itself up to a wide range of control possibilities.

To take complete control of all the possibilities of some kind of signal processing "patch," or network, it may be necessary to specify independently where all the control is coming from: the basic pitch and tempo material, timbral changes, pitch articulation, whatever. These should be controllable physically, sequentially, or algorithmically; if algorithmically, the inputs to the algorithm should themselves be controllable in any way. The more a given situation relies on unusual synthesis methods or input devices, the more acutely we need to be able to specify exactly what will control what, and how.

MAX as a Control Environment

The following description will adhere to the NeXT version of MAX as it stands at the time of this writing. David Zicarelli has made many extensions to the MacIntosh version of the program which are not yet available on the NeXT. He has also refined and extended the graphical presentation of the MacIntosh program in many ways which are also not reflected in the NeXT version shown here.

The fundamental concept in MAX is the patch. A patch is a collection of boxes connected by lines. The boxes represent objects which wait for messages to be passed to them, at which time they may respond by passing messages to other boxes. Boxes 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; any message the source object passes to its outlet is passed on to all the inlets connected to it.

The messages that are passed down the lines consist of an ordered list of atoms, each of which may be a number (fixed or floating point) or a symbol. Any message that can be passed has a printable equivalent. Messages frequently consist of a single number, or of the symbol "bang," which is used conventionally to denote an event which has no parameters.

In addition to sending and receiving messages (usually via the inlet/outlet mechanism), objects in MAX may access the clock or MIDI I/O. The clock is accessed via a simple callback mechanism. An object may allocate any number of desired "virtual clock" objects. Each virtual clock may be set to call the client object back at a given time; the callback time of a virtual clock may be changed at will or the callback may be cancelled. There is only one priority. An object wishing to receive incoming MIDI messages inserts itself on the appropriate MIDI callback list. MIDI output is spooled by calling a library function.

MAX may therefore be implemented by a fairly simple scheduler, at least compared to the schedulers described in (Boynton 1986) or (Anderson 1986). On the NeXT, the scheduler is provided by the FTS system (Puckette 1991), which

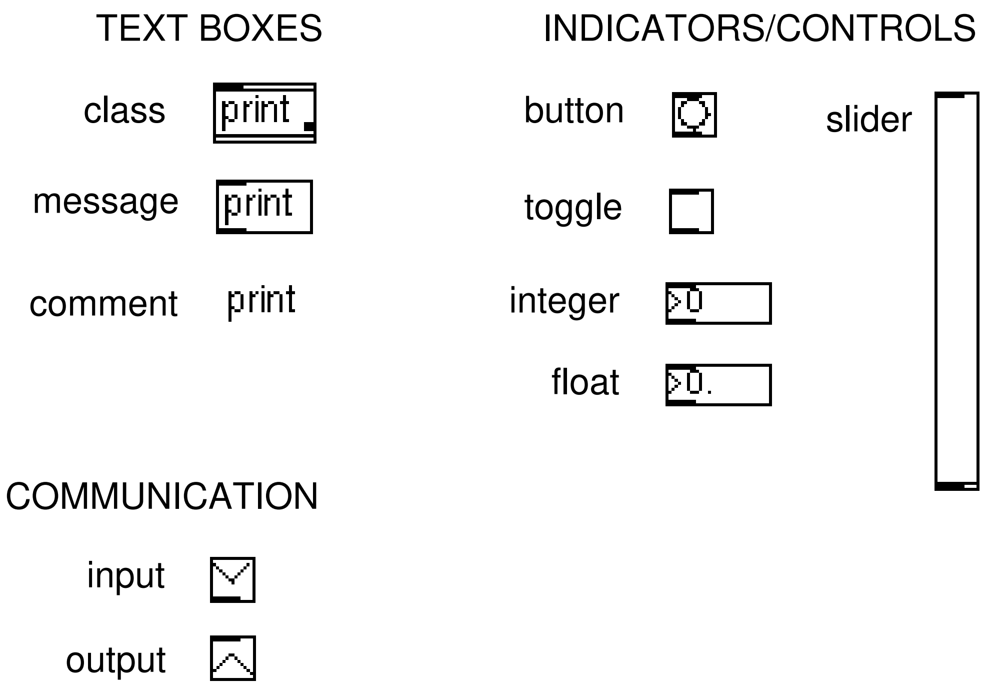

Figure 1: The box types in MAX.

also provides a DSP duty cycle clock which is used for the signal processing extension, and an interprocessor messaging facility.

A patch is either in "run" or "edit" mode. In run mode, mouse and key actions are sent directly by the patcher to the appropriate object; in edit mode the same actions are used to add, remove, or change boxes and lines.

The patcher's ten types of boxes are divided into three groups: indicators/controls, text, and input/output, as shown in Fig. 1. The controls include a momentary button, a toggle switch, a slider, and fixed and floating point number boxes. All have one inlet and one outlet. The momentary button outputs the message "bang" (i.e., passes "bang" to its outlet) whenever it is either clicked on (in run mode) or it receives a MAX message. The other controls maintain numerical values and output them whenever either their values are changed by typing or mousing, or they receive a "bang" or any message whose first argument is a number (in which case the value is updated accordingly before being output.)

There are three kinds of text boxes: "class," "message," and "comment." In a class box the text is taken to be the class and creation arguments for an

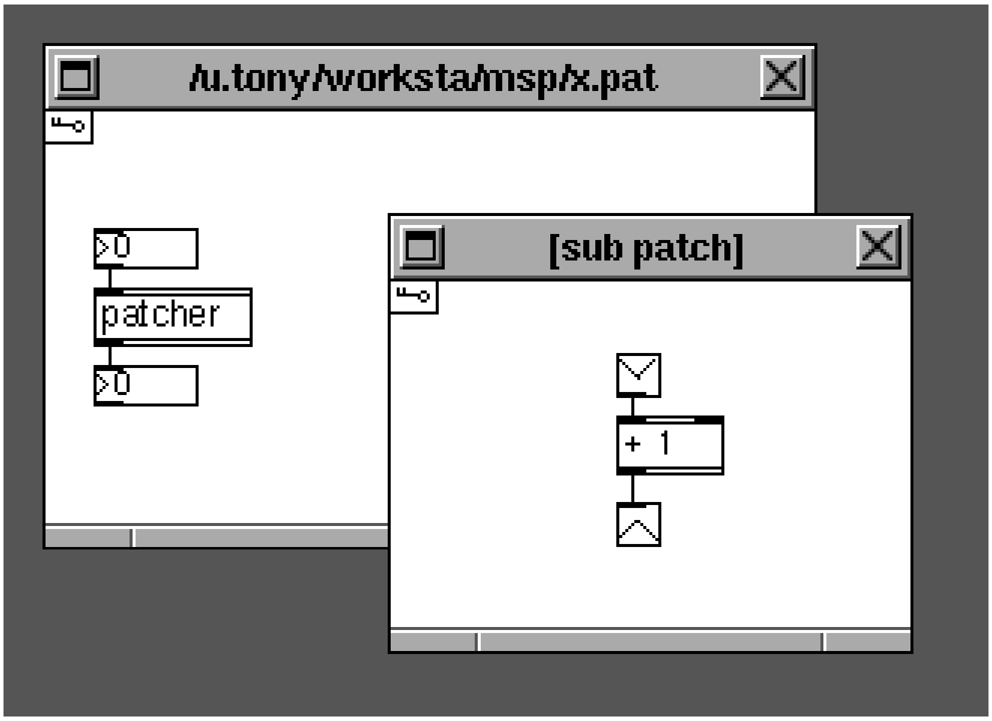

Figure 2: The sub-patch at right defines the "patcher" box at left.

object which will inhabit the box. The object thus created may in turn create some number of inlets and/or outlets; these are shown graphically on the box. The only mouse or key action which class boxes may respond to in run mode is a double click, which is often defined to open a subwindow relevant to a given object.

The "message" box contains one or more message(s) which are sent to their destinations every time either "bang" or a message starting with a number is sent to the box. The message may contain variables which are set to the arguments of the incoming message. Clicking on the message box (in run mode) is equivalent to sending it a "bang" message. Multiple messages may be separated by commas or semicolons. After a semicolon, the next atom of text in the message box is taken to specify the name of a new target; thus, messages in a single box may be sent to many different destinations. The message target is initially the message box's outlet.

The "comment" box is used to write text on the patch for labels and comments.

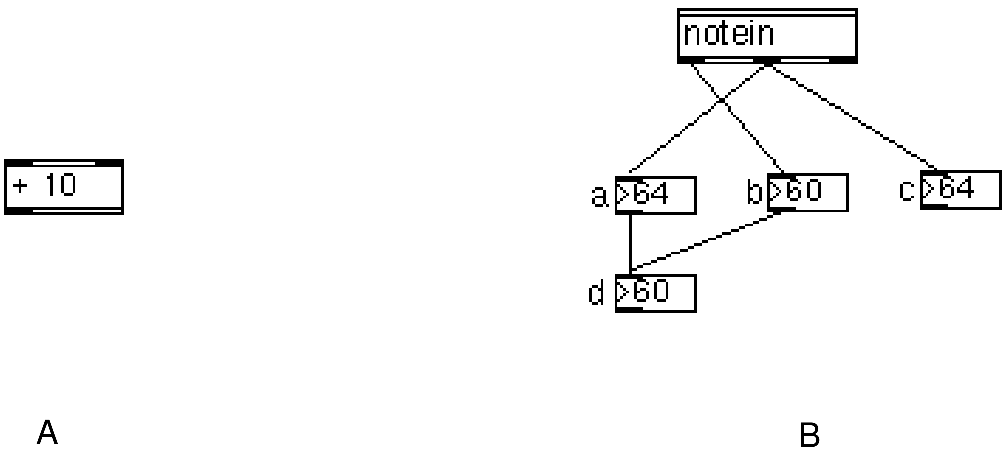

Figure 3: (A.) a class box in MAX; (B.) a small network.

The input/output boxes are used in a patch which is to be represented as a box appearing in another patch. (The interior patch is called a sub-patch.) An example of a sub-patch, shown in Fig. 2, adds one to values appearing on its inlet and passes the result to its outlet. Patches may be nested as deeply as desired. Alternatively to typing "patcher" in a class box, one may type the name of a file containing a patch. The embedded patch is then called an abstraction; the abstractions mechanism is useful when many copies of the same patch are needed, as in Figs. 5 and 6. Changes to the file then propagate to all the instances.

Execution Order

By convention, if an object has more than one inlet, its leftmost inlet is the "active" one; passing a message to that inlet causes something to happen, and passing messages to the other inlets simply changes the state of the object. For example, the box shown in Fig. 3 adds 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 2, for example, is sent to the left inlet, x is set to 2 and the box outputs the sum x+y, or 12. If the number 3 is now sent to the right inlet, y is set to 3 but nothing is output. Sending 4 to the left inlet would now output 7.

More complicated objects may take a wider variety of different messages than can be conveniently differentiated by the inlet mechanism. For example, a sequencer might take "start," "stop," "pause," "continue," and so on, as messages. In such a case, message boxes would be used to specify the desired messages. Inlets other than the rightmost inlet (which really makes a direct connection to the object itself) serve as a shorthand for an appropriate message box, which could be, for example, "in1 $1" where the variable $1 assumes the value of an incoming message's first argument in the style of a shell in UNIX.

It is frequently important to get messages to their destinations in a particular order, with an action-causing message arriving at a given object after all the modifying information (such as values sent to inlets other than the leftmost) has been communicated. For example, the multiplication described above would give a different output if the "3" were sent before the "2".

The order in which events will occur may be predicted by looking at a patch. Any box which passes messages to more than one outlet for a single event (i.e., a single incoming message, timeout, or MIDI input) passes messages to the outlets in right-to-left order. This choice was taken since a box's leftmost inlet should be passed a message last if messages to other inlets are to have an effect on its action. When an outlet which is connected to more than one inlet is passed a message, the inlets receive the message in right-to-left order. Messages are function calls which do not return until all resulting messages have also been passed. Thus, in Fig. 3, the number boxes would be sent values in the order (c, a, d, b, d); note that "d" receives two messages.

MAX is not a dataflow language; the boxes which make up a patch usually contain some local state. Dataflow's independence of the order in which the inputs to an operation become available cannot be achieved here. On the other hand, MAX's object-oriented approach is much more appropriate for systems which must respond to external requests for action. In this scenario, which is typical of live human/machine interaction, the order in which transactions occur is often significant. A violin should be tuned before playing it, not after, for best results.

The Tilde Classes

Signal processing in MAX is carried out by a collection of "tilde classes" which communicate via inlets and outlets through the message, "signal." As a convention, the names of tilde classes all end in tilde, as in sig , osc1 , etc. At the time of this writing, twenty-four tilde classes have been implemented and eight more are planned. An object belonging to a tilde class is called a tilde object.

The tilde objects carry out signal processing tasks on vectors of a fixed size N, typically between 16 and 64. The DSP duty cycle time is set to the sampling rate divided by N. For each DSP duty cycle period, every running tilde object is called to carry out its duty cycle action. This action may reference signal inputs and/or outputs (which appear as vectors of size N), and/or the tilde object's instance space.

The tilde objects must intercommunicate at setup time in order to determine a calling order, and the addresses of input and output signals, to be used in the DSP duty cycle. This is managed via the "signal" message, which the tilde objects pass among themselves via inlets and outlets. "Signal" is an ordinary MAX message, and no extension of the inlet/outlet mechanism was made to introduce it.

A network change is reflected in a change in the DSP duty cycle call list whenever a dac object is sent the "start" message. The "stop" message clears the call list. If a tilde object is destroyed by editing the patch, the object acts to clear the call list as part of its cleanup, since the call list might refer to the object's instance space.

Tilde classes communicate with control objects through their instance data. For example, a two-pole filter, f2p , is defined which maintains three filter coefficients which are used during the DSP duty cycle. These coefficients may be changed via messages, since the same instance data an f2p uses during DSP processing is accessible to it as it handles messages.

DSP objects may not initiate messages during the DSP duty cycle (the other DSP objects might not be in a coherent state), but they may set timeouts. For example, a threshold class could be defined, which would set a timeout with zero delay when its input signal met a certain condition; the timeout would then occur during message processing after the DSP duty cycle finished. There is therefore a small but nonzero round-trip delay between messaging and signal computation.

Examples

A simple patch, shown in Fig. 4, outputs a cosine wave whose frequency is controlled by incoming MIDI note-on messages. The signal-processing part of the patch is defined by the sig , osc1 , line , * , and dac objects. The sig is a message-to-signal convertor, taking floating-point numbers as input and creating a signal output whose samples are all equal to the most recent value received. The osc1 takes a frequency and phase offset as signal inputs and outputs a cosine wave (calculated by interpolating table lookup) of unit amplitude. Since its phase offset input is disconnected it is taken as zero. The cosine is then multiplied (via *) by the output of the line breakpoint envelope generator, which may be passed breakpoints as messages of the form (target-value time-in-milliseconds). The dac then sends the result to both channels of audio output.

The message box at upper left is used as a button; clicking on it passes the messages "start" to the dac , "440." to the sig , and "0.1 50" to the line . (The objects such as "r freq" act as remote message receivers; "s freq" is a remote message sender.) The chain at lower left takes in MIDI note-on messages, discards those of velocity zero, displays the resulting pitch (ignoring velocity), converts to a frequency and sends it to the "r freq." The two message boxes at top center ramp the amplitude up and down when activated.

In this example, three asynchronous event sources are merged: mouse clicks, incoming MIDI, and the DSP duty cycle. The boundaries between event sources occur at the inlets of the tilde objects, which change their state in ways that is

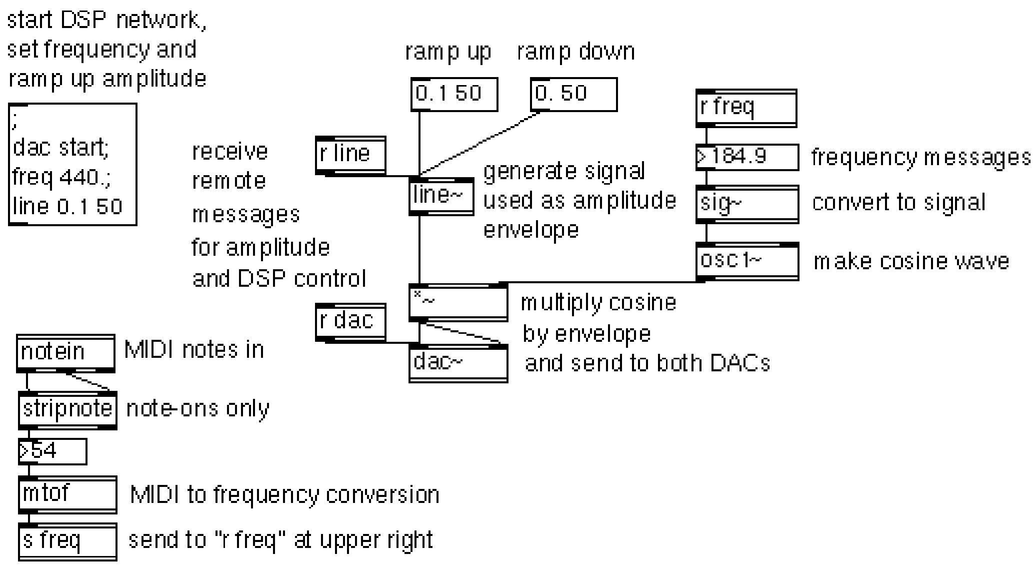

Figure 4: A patch to generate a single amplitude-controlled sine wave.

later reflected in their DSP behavior.

Fig. 5 shows a MIDI-controllable bank of eight oscillators whose outputs are summed. Here, the majority of the DSP duty cycle is hidden in the osc.pat abstraction, which is shown in Fig. 6. Osc.pat has a signal inlet, a control inlet, and a signal outlet. The oscillator's output is added to the signal inlet and the sum put on the signal outlet. The control inlet of osc.pat takes (velocity, pitch) pairs. If the velocity is zero the line is turned off with the "0. 200" message; otherwise a new frequency is sent to the sig and the line is turned on.

The sig and line also have signals connected to their inlets. These signals are ignored but serve to constrain the DSP duty cycle building algorithm not to schedule them too far in advance (which would waste memory since their results would have to be stored longer.)

Finally, Fig. 7 shows a patch which grew out of a sonic experiment, and which is shown to better indicate the range of available signal processing and control elements. The patch makes a spectrally rich signal by waveshaping (via the leftmost osc1 and the clip), and flanges and reverberates the result. Additional tilde objects include delwrite and vd (writing to a delay line named "del1" and reading from it with a variable delay time), and print which prints out a vector of N samples for debugging, whenever it receives the "bang" message.

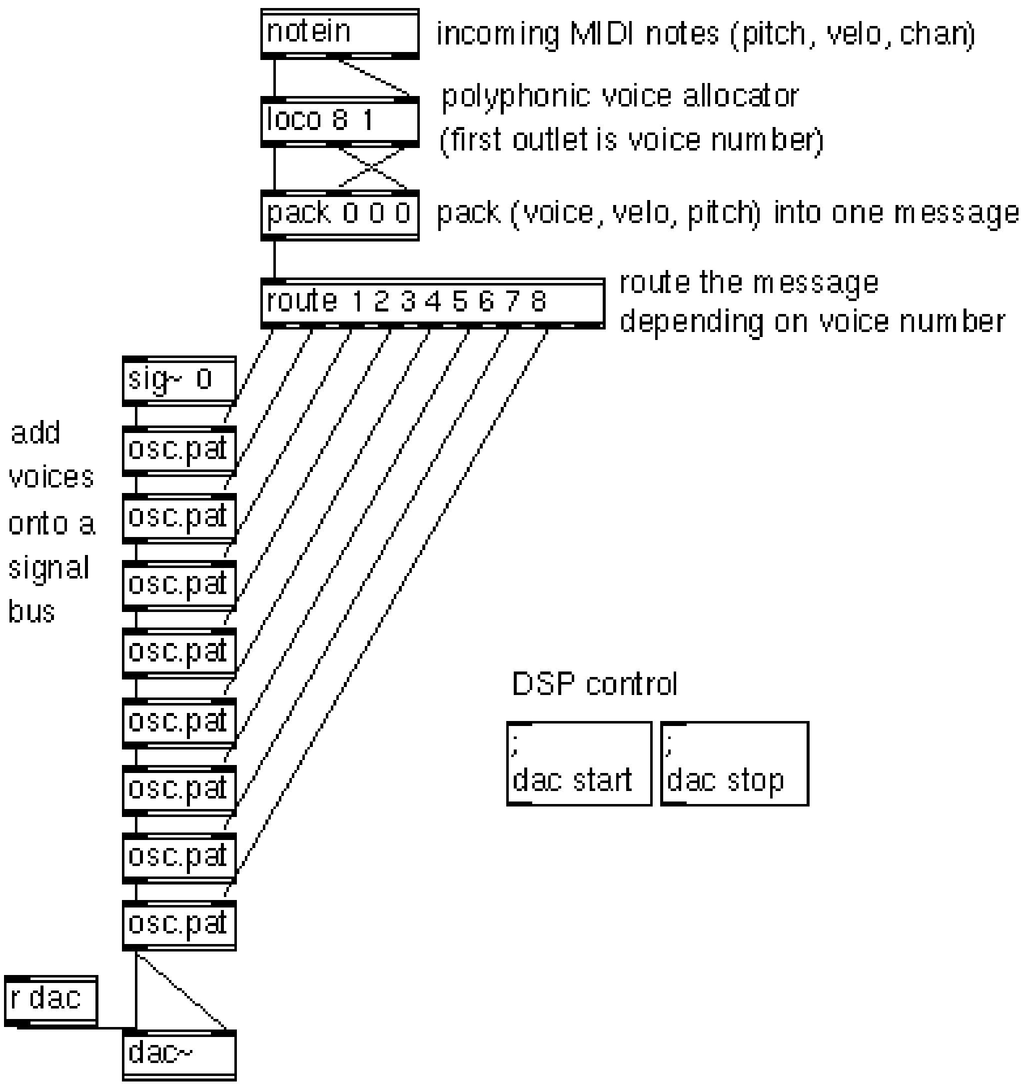

Figure 5: Polyphony, using the "loco" object and an abstraction, "osc.pat".

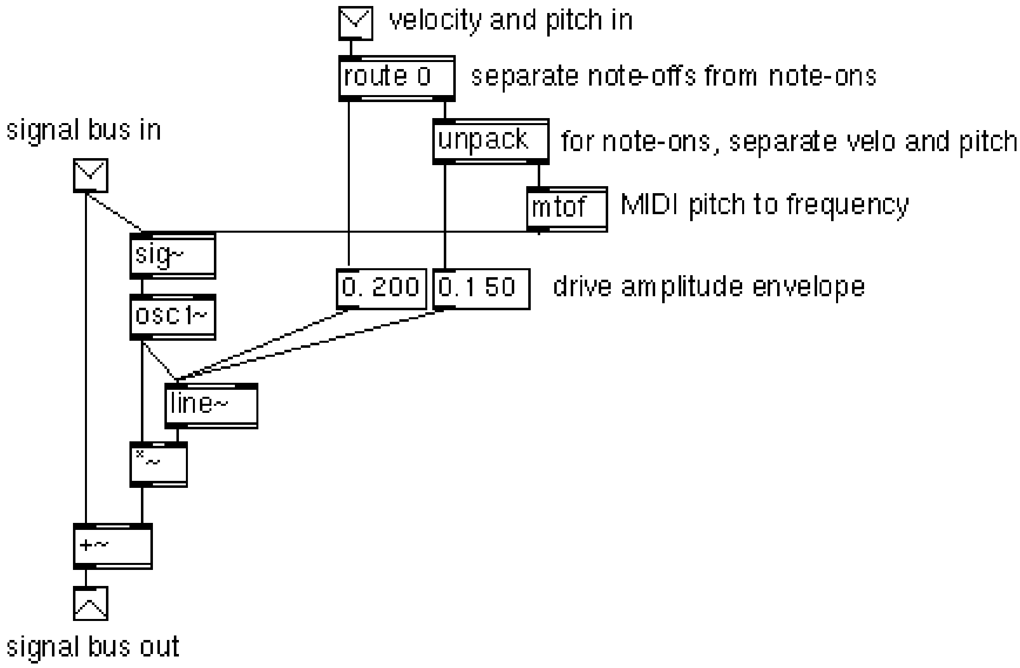

Figure 6: The definition of “osc.pat” used in Figure 5.

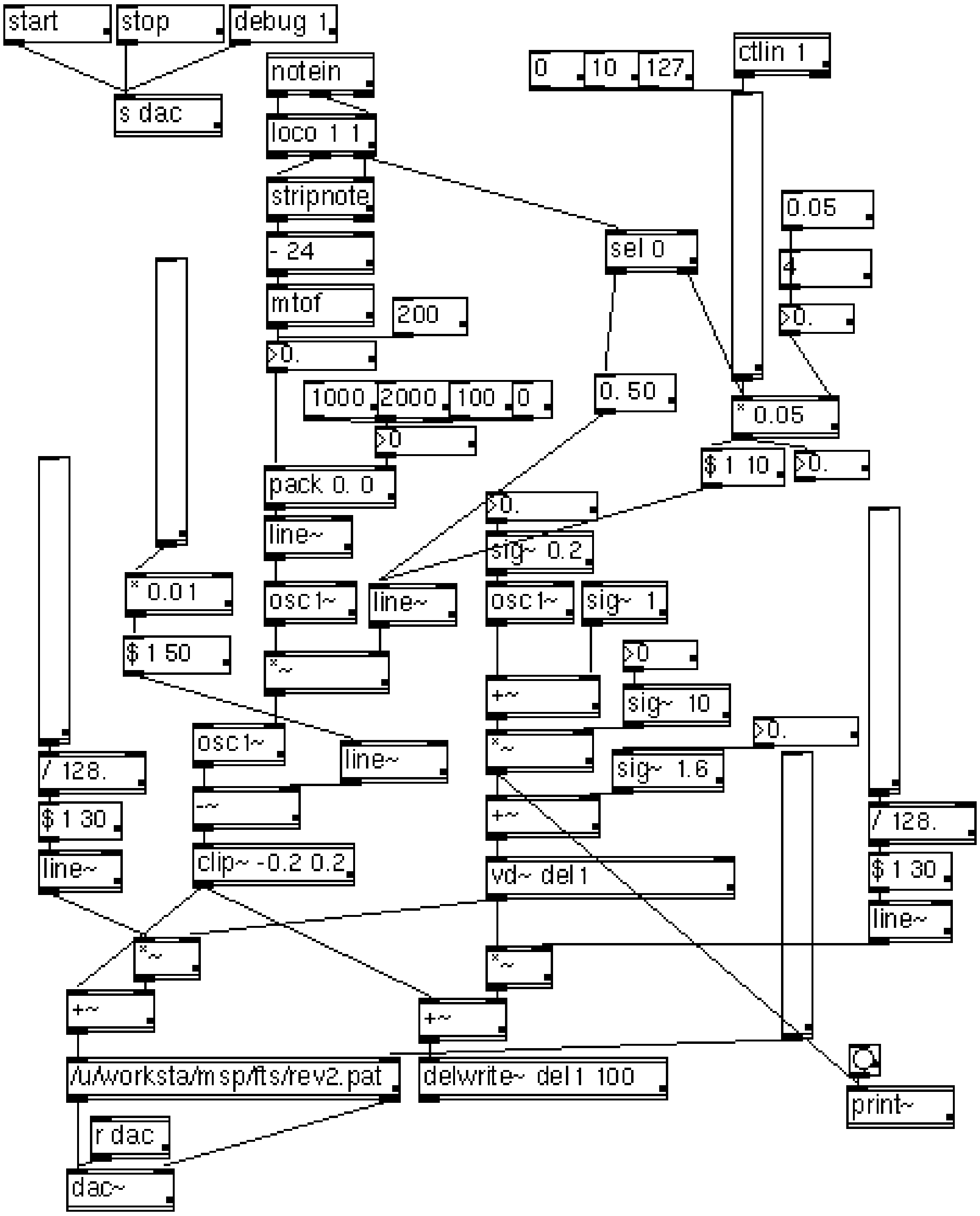

Figure 7: A real patch.

How the DSP Duty Cycle Call List is Built

The "signal" message acts as a token used by the tilde objects to simulate a data-driven dataflow network; that is to say, each tilde object can "run" when all its signal inputs are defined, and running a tilde object defines all its signal outputs. As the simulation is carried out, each time a tilde object is "run" it appends itself to the call list, specifying any arguments needed by its duty cycle action.

The "signal" message takes two integer arguments: a selector (one of the constants COUNTINPUTS, COUNTOUTPUTS, or DOIT), and the address of a signal object. When a dac object receives the "start" message, it first traverses the list of all tilde objects, causing each one to pass a "signal COUNTINPUTS" message to all its signal outlets. For each signal inlet a count (its input count) is thus taken, equal to the number of signal outlets connected to it.

The list of all tilde objects is then traversed a second time, and any tilde object which has no signal inlets, or whose input counts are all zero, is put on the call list.

Each time a tilde object is put on the call list, new signals are allocated for all its signal outlets, and the outlets are passed first a "signal COUNTOUTPUTS", then a "signal DOIT", message with the address of the newly allocated signal. The first message is used simply to count the number of inlets connected to each outlet (which is remembered by the signal). The DOIT message informs the inlet that the signal in question may now be used.

Thus, when a tilde object receives the "signal DOIT" message in some inlet, it can determine whether all its inputs are available. After decrementing the inlet's count, if all inlet counts are now zero we can put the receiving tilde object on the call list, and repeat the actions of the above paragraph recursively. At the time we put the new object on the call list, we know the addresses of all the input and output signals, as well as any instance data that might be needed by the duty cycle routine.

The signal that is allocated for a signal outlet when a tilde object is put on the call list may be freed for reuse as soon as the last tilde object having it as an input is put on the call list. As each tilde object is put on the list, it decrements the count that had been obtained by the signal using the "signal COUNTOUTPUTS" message, and when that count reaches zero, the signal is freed. The freed signal may be immediately reused by the tilde object which was the last to use it as an input; thus, a chain of tilde objects, each with one signal input and one signal output, typically reuses the same signal in place. This is an important optimization in processors such as the Intel i860 or the Motorola MC56001, each of which has a limited internal memory (where one would try to place the signal vectors.)

At the end of the call list building process, if any tilde objects are not on the call list, a signal loop has been detected. If a signal loop is actually desired, a delay read/write pair must be used; the minimum delay that may be obtained is one duty cycle period.

In the example of Fig. 4, the DSP call chain is built as follows. The sig and line objects are found to have no signal inputs and either of the two may start the call chain; suppose it is the sig . The sig is put on the call list with an output signal address (call it signal 1). Putting the sig on the call list satisfies all the signal inlets of osc , so it too is put on, with signal 1 as both its input and its output. The right-hand inlet of * is then found to be satisfied, but not the left-hand one, so the search continues for tilde objects with no signal inputs, and the line is found. Since signal 1 is still needed to hold the output of osc1 , another (signal 2) is allocated to hold the output of the line . Now both inlets of the * are satisfied and it and the dac are put on the call list, finishing it.

In building a signal network, the user is not obliged to worry about the order in which the signal inputs of a module become available; for instance, in the above example, the inputs of * may be calculated in either order without changing the behavior of the network. This is made possible by the fact that the tilde objects all carry out their DSP actions synchronously; there is only one DAC clock. In contrast, the control portions of a patch may be activated in different ways from different event sources, so one needs to pay explicit attention to order-of-execution problems.

Implementation

In the NeXT implementation of MAX, all real-time processing is offloaded onto an Intel i860 coprocessor running the FTS real-time monitor program (Puckette 1991), under the CPOS operating system (Viara 1991). Each box defined in MAX gives rise to two objects, one on the NeXT which is used for editing, controlling, and viewing, and one on the i860 for doing the computation. The two must sometimes communicate for control or graphical feedback; this communication is described in (Puckette 1991).

At the time of this writing, the implementation is capable of running 41 voices of the type shown in Fig. 6 on a single 40 MHz. processor, at a sampling rate of 44.1 KHz.; thus, a six-processor system would be expected to run 246 such voices. We expect, perhaps not too optimistically, that further optimizations will extend this number to at least 300 (50 per processor.)

Possible Extensions

In order to take advantage of the multiprocessing hardware of the IMW, MAX on the NeXT has been extended to allow user control over which processor a given object is to reside on. (The FTS model requires an instance to reside on a specific processor, and all message handling for that instance is done on that processor.)

To simplify the user interface, the choice of processor is made for an entire window at a time. Thus, the only message paths that may go across processors are the input/output boxes of a sub-patch, or send/receive pairs. These two mechanisms have been extended to detect processor boundaries and automatically set up remote-send/remote-receive pairs in FTS as needed. Passing messages or signals across an FTS processor boundary incurs a delay equal to the FTS-defined latency of the sending processor. A prototype version of MAX with this extension is running, but certain issues raised by processor boundaries have not yet been resolved.

A planned improvement to the DSP call chain building algorithm would cause it to automatically generate "sig " objects for signal inlets in cases where a tilde object has initializing arguments; thus, "+ 1" would add one to a signal and the "1" could be updated by messages.

As an efficiency measure, extensions will be required to allow multirate signal processing (so that a vibrato could be calculated at a lower sampling rate than audio, for example), and to turn subsets of the DSP call list on and off so that different synthesis networks can be used at different times without the need to run them all at once. How these two shortcuts will be specified graphically has yet to be decided.

No mention has been made here of real-time soundfile access. Tilde classes to read and write soundfiles, while easy enough to specify, have so far only been implemented in non-real-time simulation. It seems realistic to hope for at least four channels per disk of uncompressed digital audio, if the disk is otherwise idle.

The question naturally comes up of an implementation of the tilde classes for the MacIntosh computer, either using one of the several Motorola MC56001-based cards now on the market, or else using some future i860-based one. This seems entirely feasible. The major novelty would be that the control code could no longer run on the external processor (at least, not without making major changes in the MacIntosh version of MAX.) Thus, the tilde classes themselves would have to manage instance data on both processors and use explicit, time-tagged messaging between them. This will result in a higher round-trip time between messaging and DSP calculation, as well as complicating the tilde objects themselves.

Acknowledgements

David Zicarelli has made a huge contribution to the MacIntosh version of MAX, some of which has made its way to the NeXT version described here as well. Other contributions of code to MAX were made by Lee Boynton, Cort Lippe, Zack Settel, and David Yadegari. Brave composers who used MAX early in its development (and thus helped push it forward) include Frederic Durieux, Michael Jarrell, and Philippe Manoury, assisted by Thierry Lancino, Cort Lippe, Jan Vandenheede, and Nicolas Verin. I would also like to thank David Wessel, without whose encouragement and guidance I would never have tried to write MAX. And of course Max Mathews, for whom MAX is named. Not only

did he greatly influence me in the few months we worked together, but also his RTSKED program (Mathews 1981) introduced the real-time scheduling approach that MAX adopts.

References

Anderson, D. and R. Kuivila. 1986. "A Model of Real-Time Computation for Computer Music." Proceedings of the 1986 International Computer Music Conference. San Francisco: Computer Music Association, pp. 35-41.

Bate, J. 1990. "UNISON – a Real-Time Interactive System for Digital Sound Synthesis." Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association, pp. 172-174.

Boynton, L. et al. 1986. "MIDI-LISP: A LISP-Based Music Programming Environment for the MacIntosh." Proceedings of the 1986 International Computer Music Conference. San Francisco: Computer Music Association, pp. 183-186.

Favreau, E. et al. 1986. "Software Developments for the 4X Real-Time System." Proceedings of the 1986 International Computer Music Conference. San Francisco: Computer Music Association, pp. 369-373.

Helmuth, M., 1990. "PATCHMIX: a C++ X Graphical Interface to Cmix." Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association, pp. 273-275.

Lindemann, E. et al. 1991. "The Architecture of the IRCAM Music Workstation." Computer Music Journal 15(3): pp. 41-49.

Lindemann, E. 1991. "ANIMAL – a Rapid Prototyping Environment for Computer Music Systems." Computer Music Journal 15(3): pp. 78-100.

Mathews, M. and J. Pasquale. 1981. "RTSKED, a Scheduled Performance Language for the Crumar General Development System." Proceedings of the 1981 International Computer Music Conference. San Francisco: Computer Music Association, p. 286.

Minnick, M. 1990. "A Graphical Editor for Building Unit Generator Patches." Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association, pp. 253-255.

Opcode, Inc. 1990. "MAX" (documentation for MacIntosh software). Palo Alto: Opcode, Inc.

Puckette, M. 1991. "FTS: A Real-time Monitor for Multiprocessor Music Synthesis." Computer Music Journal 15(3): 58-67.

Puckette, M. 1988. "The Patcher." Proceedings of the 1986 International Computer Music Conference. San Francisco: Computer Music Association, pp. 420-429.

Viara, E. 1991. "CPOS: A Real-Time Operating System for the IRCAM Music Workstation." Computer Music Journal 15(3): 50-57.