AN EXTENSIBLE REAL-TIME SIGNAL PROCESSING ENVIRONMENT FOR MAX

David D. Zicarelli
Cycling '74
1186 Folsom Street
San Francisco, CA 94103
zicarell@cycling74.com

ABSTRACT

MSP is a set of additions to the Macintosh version of MAX for synthesis, signal processing, sampling, and hard-disk recording/playback directly on a PowerPC processor. It is based on ideas from MAX/FTS developed by Miller Puckette at IRCAM, and incorporates software from Puckette's cross-platform Pd (Pure Data) program.

INTRODUCTION

MSP stands for Max Signal Processing, as well as being the initials of Miller S. Puckette, to whom the software owes much of its existence. It currently runs on a PowerPC Mac OS system as a set of extensions to MAX 3.5, published by Opcode Systems (Dobrian and Zicarelli 1991). These extensions take the form of MAX external objects and a shared library that contains support functions used by each of the objects. An external object is a dynamically linked piece of C or C++ code that serves to extend the capabilities of MAX. Currently there are about 65 supported objects in MSP. In addition, MSP supports audio drivers, dynamically linked code modules that can address specific PCI-based audio interface cards. The goal of MSP was to help bring to a wider audience what was until recently a relatively inaccessible technology. The two ways this was accomplished were to use a commonly available hardware and software platform, and through careful refinements to the user interface design, documentation, and supporting materials.

HISTORY

The inspiration for MSP comes from MAX/FTS, software that originally ran on the IRCAM Signal Processing Workstation (ISPW) (Puckette 1991). MAX/FTS extends the usual set of MAX MIDI and control objects with a set of DSP objects that represent common operations in a signal processing algorithm. The connections between the objects specify the audio signal paths. This allows one to draw Music V diagrams and then control them in real-time with the usual features of MAX such as remapped MIDI input or mouse gestures that move sliders. On the NeXT-based ISPW, two copies of the MAX patch were running simultaneously. One ran on the host machine, and the other, which actually did the real-time signal and control computation, ran on one or more ISPW processor cards. The host program dealt with user interface and operating system information, and communicated it to the processor cards.

MSP, by contrast, runs on the host PowerPC processor, so there is no special communication between the user interface and the DSP or control algorithms. While this has the advantage of simplifying tasks such as controlling a DSP algorithm with the mouse or handling playback and recording to and from the computer’s hard disk, it does mean that if the DSP processing consumes too much of the computer’s CPU, the screen will begin to draw more slowly and the mouse may become unresponsive. This is partly because it seemed important to give the signal processing priority over other tasks the machine was doing—few people would appreciate clicks in the output signal when the cursor was updated to reflect mouse movement.

SIGNAL COMPILER AND OPTIMIZATION

Each MSP object typically implements a specific unit generator required to build synthesis or signal processing algorithms. Unit generators typically take one or more input vectors of signal values and compute one or more output vectors. Depending on how an object is connected to other MSP objects, the exact real-time algorithm used can vary. For instance, the interpolating table-lookup oscillator, called cycle~, has an input for frequency (in hertz) and phase (from 0-1 to move through a table of 512 samples). If the frequency will be controlled with a signal, cycle~ uses a slightly different algorithm than if the frequency will be constant over a signal vector. In the latter case, the algorithm does not read a frequency input signal vector, saving a vector’s worth of memory accesses. Uncached memory accesses constitute a significant portion of the processor’s time in computing a DSP algorithm,
so if the input signal vector didn't happen to be in the cache, the savings of not accessing the memory could be non-trivial.

Another aspect of this optimization process is that the user can choose to use either signals or MAX-style control messages for most MSP control inputs. The use of control inputs is often a convenience during the development of an MSP application. The illustration below shows an MSP implementation of a simple FM algorithm, in which some of the inputs are signals, such as the frequency input of the carrier oscillator, and some are controls, such as the frequency input of the modulating oscillator and the modulation index.

Given a graph of MSP signal objects as shown above, a signal compiler is used to determine the order in which each external object's signal computation routine should be called and how routines should communicate with signal vectors. In this way, a graphical representation is transformed into a chain of function calls. The signal compiler is based on the one in Miller Puckette’s Pd program (Puckette 1996). It contains a few enhancements, such as the ability for any object within a compilation to determine information about its signal inputs and outputs. This is used by the cycle~ object discussed above to select an optimal processing algorithm. Another enhancement allows an object to determine whether there is a signal path, either directly or indirectly, between it and another MSP object. This is used to create portions of the DSP network that can be switched on and off, allowing the user to manage processor resources.

Pd was also the source for some of the basic MSP unit generators, most of which have seen minor enhancements. For instance, the line~ object based on the Pd version can be given up to 60 pairs of times and target values, allowing multi-stage control envelopes to be triggered with one step.

In MAX/FTS and Pd, the difference between a signal connection and a control connection is not visually obvious. However, in MSP, a signal connection is displayed with a yellow-and-black striped “pipe cleaner” patch cord, while the traditional MAX control connection remains a solid black line, allowing the user to see at a glance which connections are in the signal domain and which are in the control domain. The user experience is further enhanced by allowing changes to the signal network while it is playing. The compiler immediately recompiles the signal network and restarts the processing whenever a signal connection is made or broken, or when a new MSP object is instantiated or deleted.
SIGNAL VECTORS

With the current Macintosh system (7 or 8), one pays a significant performance penalty if interrupts occur too frequently. This means that it is generally inadvisable to send as few as 64 samples to a PCI audio card at a time, since the card will interrupt the host processor each time it needs new samples. A more typical size for an output vector is 512 samples, representing about 11 milliseconds of audio at a 44.1 kHz sampling rate. The problem is that a number of signal processing algorithms work and sound better with smaller computation vectors. MSP addresses this problem by making the processing vector size independent of the I/O vector size. Currently, the minimum processing vector size is 64 samples, but this number can theoretically be reduced to a single sample to allow for certain types of feedback algorithms. The smaller computation vector size may also produce better cache performance in certain cases since less memory is used to communicate between DSP unit generators. At some point, this savings would be offset by the increased frequency with which a unit generator would be saving and restoring its internal state.

SOME MSP OBJECTS

MAX/FTS had a limited number of signal objects because the Intel i860 used in the ISPW had a very small instruction cache. This avoided performance degradation when instructions to execute DSP algorithms were not in the cache. By contrast, more recent processor designs such as the PowerPC contain on the order of 32K of instruction cache, so the code for a significant number of MSP unit generators can remain in the cache. As a result, there are far more MSP signal objects than in MAX/FTS. MSP introduces a new category of signal objects that are also MAX user interface objects. Primarily these objects do signal display and monitoring; they include a level meter, oscilloscope, a signal fader, and a special version of the MAX number box that prints an incoming signal value or directly generates an outgoing signal value.

In MSP, there has been an attempt to streamline the way objects deal with areas of memory containing sound samples. Any object that will read from or write to a fixed area of memory works with a buffer~ object, which handles storage and display of a sample, as well loading and saving sample memory using AIFF or Sound Designer II file formats.

The cycle~ object can use a 512 sample portion of a buffer~ as an interpolated wavetable, while the more computationally expensive wave~ object can use any portion of a buffer~ as a wavetable. The lookup~ object can use all or part of a buffer~ to perform waveshaping. play~ and groove~ are interpolating sample-playback objects that read from a buffer~. groove~ uses an input signal that specifies a sample increment, rather than an absolute sample position. record~ and poke~ write sample values into a buffer~, while peek~ and index~ can read them out.

Some other nominally novel MSP objects include train~, a pulse waveform generator that can also be used as a sample-accurate metronome; curve~, an multi-segment exponential ramp generator similar; and send~/receive~, a pair of objects that allow for dynamic signal routing.

6. WRITING MSP SIGNAL PROCESSING ALGORITHMS

The implementation of an MSP external object requires the initialization of a typical MAX object (Zicarelli 1991). In addition, the programmer must write one or more processing routines, and a DSP call definition routine. The processing routine will be called repeatedly at interrupt level to do the real-time signal calculations. The DSP call definition routine specifies the particular processing routine to be used, plus the arguments the routine accepts. Typically these arguments will be the signal input and output vectors and a pointer to the MAX object instance that holds the internal state of unit generator’s DSP algorithm between one vector and the next.

Here is the structure definition for a typical MSP object, in this case, one that is used to make a low-pass filter. The structure contains a header common to all MSP signal objects that holds information that allows and object’s inlets to receive either signals or control messages.

1 A special non-real-time mode does not run at interrupt level. This can be used in conjunction with signal processing networks that cannot be realized by the user’s hardware in real-time. It can also be used by the developer of a new MSP unit generator to debug her processing routines.
typedef struct _lowpass {
    t_pxobject l_obj; // header containing Max-specific information
    float l_xm1; // filter memory
} t_lowpass;

Here is an implementation of the processing routine for the low-pass filter algorithm.

```c
int *lowpass_perform (int *w) {
    float *in,*out, val, xml, res;
    int n;
    t_lowpass *x;
    in = (float *)w[1]; // input vector
    out = (float *)w[2]; // output vector
    n = w[3]; // vector size
    x = (t_lowpass *)w[4];
    if (x->l_obj.z_disabled) // check for a disabled network
        goto out;
    xm1 = x->l_xm1; // restore previous sample
    while (n--) {
        val = *in++;
        *out++ = (val + xm1)/2.
        xml = val;
    }
    x->l_xm1 = xm1; // save previous sample
    out: return w + 5
}
```

The lines of the function labeled `setup` constitutes the overhead for computing each block of samples. In particular, the last input sample of each vector needs to be saved to that it can be the first previous sample of the next vector. The lines labeled `process` are those that actually perform the filtering.

CONCLUSION

Because MSP was designed to work on inexpensive hardware that is common in the computer music community, the hope is that it can be a platform for exchanging various types of music synthesis and processing development. The ability to add new MSP objects should allow the implementation of many types of DSP and synthesis applications in the future, and allow them to be used within the myriad of contexts that has been characteristic of MAX for almost a decade.

ACKNOWLEDGMENTS AND REFERENCES

The author wishes to thank the following people who have contributed to the development of MSP: Stefan Bilbao, Christopher Dobrian (who wrote the manual and tutorials), Toni Dove, Richard Dudas, Peter Elsea, Jean Laroche, Cort Lippe, Tom Mays, Robert Miller, Miller Puckette, Zack Settel, and Leslie Stuck.