How to Write MSP Externals

Revision 3 of 1 August 1998

written by
David Zicarelli

MSP is published by

David Zicarelli
zicarell@scruznet.com
http://www.cycling74.com

Contents

CHAPTER 1

Preliminaries

This document describes how to write signal processing externals for Max using the API of the MSP signal processing environment. It assumes familiarity with the basic elements of writing Max externals, as described in the *Writing Externals Objects for Max* document. MSP externals are very similar to Max externals, but they have two additional methods specific to signal processing. One is the **perform method** that actually calculates the signal. Typically it takes one or more input buffers and uses them to produce one or more output buffers. MSP assembles calls to objects' perform routines into a **DSP call chain** connected by signal buffers. The second additional method you need to write, called when MSP builds the DSP call chain and sends your object the dsp message, tells MSP the address of your perform routine and the arguments it requires. We'll refer to it as the **dsp method.**

In addition to the perform and dsp methods, there are calls you need to make in the initialization, new instance, and free routines. There are two sets of calls, depending on whether you are writing a normal object or a user interface object. However, there are no differences between normal and user interface objects in writing the dsp and perform methods.

How MSP Works

The MSP functions described here reside in a shared library called *Max Audio Library*, a file that must be located in the same folder as the Max application. This shared library exports some of its functions and globals. You include it in your project, but since it's a shared library, it doesn't get included in your compiled code. Rather, the inclusion provides the linker with the definition of the symbols you reference in your code when you call one of MSP's functions. Then, when your object (or the first MSP object) is about to be loaded into memory, the operating system is notified that Max Audio Library is needed, and it is loaded. At that point, all the MSP calls and globals are available. (Max Audio Library is included within the special MAXplay that works with MSP externals.)

Making a CodeWarrior Project

These directions assume you're using Metrowerks CodeWarrior 11. Currently, we recommend CodeWarrior 11 because CodeWarrior Pro is not compatible with the Motorola compiler, which has been used for most of the standard MSP external objects because it can provide significant optimizations, and with DSP, speed is

paramount. However, in developing your project, you should start with the CodeWarrior compiler and leave all the optimizations off until you get your object working properly. At that point, you might either try the CodeWarrior optimizations or consider investing in the Motorola compiler. If you have CodeWarrior Pro only, you shouldn't have too much difficulty extrapolating the project settings from the list below, or you can convert the sample project.

In addition to your source file and any resource files you might have, you need to include the following files:

- Max Audio Library
- MaxLib
- InterfaceLib
- MathLib (or libmoto, if you want much faster but possibly 603-incompatible math routines)
- MWCRuntime.Lib
- SoundLib, if you use newer Sound Manager routines

There is no penalty for including MathLib and SoundLib if they're not needed, so it's probably a good idea to include them.

Refer to the *Writing External Objects for Max* documentation to get the proper Project Settings for CodeWarrior project. However, it's probably easier just to modify a copy of one of the example MSP projects included in the software development kit. Then all the right files are included and the settings will be correct except for the name of your object in the settings for PPC Project. The only addition you might need to make to a standard Max external project is to add the **MSP #includes** folder to the Access Paths. Or you can just copy this folder inside of you MAX #includes folder, and CodeWarrior will find it.

Project Resource File

In addition to the libraries listed above and the source files you write, a good citizen MSP project will contain a resource file that contains at least two items. The first is the STR# resource used by your object's assist method (see *Writing External Objects for Max* if you are not familiar with this), and a **mAxL** resource that contains a small amount of 68K code. When loaded, this code reports that the object does not work on a 68K processor. You can use the mAxL resource that does this very thing found in the file **nono.68K** included with the software development kit. Open nono.68K in ResEdit and copy the mAxL resource to your project's resource file. Then select the resource in your project's file and choose Get Resource Info… from the Resource menu. The resource's ID doesn't matter, but the name must be changed from "nono" to the name of your object. 68K Max finds your object using the name of the mAxL resource.

Adding the mAxL resource allows your PowerPC object to be found when it is inside a collective or a standalone application. If you look at a standalone application created with MSP external objects, you'll see a bunch of empty mAxL resources, one for each external that is included.

CHAPTER 2

Writing the Code

This chapter covers the basic information you need to write an MSP external.

Include Files

Since MSP objects are PowerPC only, you should not include SetUpA4.h and A4Stuff.h as you would for FAT externals. For normal objects, you should have the following at the beginning of your source file.

```
#include "ext.h"  // standard include file for Max externals
#include "z_dsp.h" // contains MSP info
```

The include file **z_dsp.h** references a number of other include files; they will be mentioned when relevant below.

Defining Your Object Structure

An MSP object has a `t_pxobject` as its first field rather than `t_object`. `t_pxobject` is a `t_object` with some additional fields, most notably a place for an array of **proxies**, used to allow inlets to MSP objects to accept either signals or floats as input. If you're not familiar with proxies, refer to Writing External Objects for Max and the **buddy** external object sample code. In general, MSP handles most of the details of using proxies for you. User interface objects use a similar header, called a `t_pxbox`, that combines the standard `t_box` user interface object header with the fields of a `t_pxobject`. Both structures are defined in the include file **z_proxy.h**.

Here's an example declaration of an MSP external object:

```
typedef struct _sigobj {
    t_pxobject x_obj; // header
    float x_val; // additional fields
} t_sigobj;
```

Writing the Initialization Routine

The initialization routine sets up the class information for a Max external. In the call to the setup function which initializes your class—generally the first thing you do in any Max external—you should pass `dsp_free` as your free routine unless you need to write your own free routine for memory you allocate in your new instance routine.

Here's an example for an object that doesn't allocate any memory and doesn't take any initial arguments.

```
setup(&sigobj_class, sigobj_new, (method)dsp_free, (short)sizeof(t_sigobj),
    0L, 0);
```

After the call to setup, your initialization routine needs to bind your object's dsp method (discussed below), using the A_CANT argument type specifier as follows:

```
addmess(sigobj_dsp, "dsp", A_CANT, 0);
```

You also need to call the following function to finish setting up your MSP external's class.

dsp_initclass

Use dsp_initclass to set up your object's class to work with MSP.

```
void dsp_initclass(void);
```

This routine must be called in your object's initialization routine. It adds a set of methods to your object's class that are called by MSP to build the DSP call chain. These methods function entirely transparently to your object so you don't have to worry about them. However, you should avoid binding anything to their names: signal, drawline, userconnect, and enable. This routine is for normal (non-user-interface objects).

dsp_initboxclass

Use dsp_initboxclass to set up your user interface object's class to work with MSP.

```
void dsp_initboxclass(void)
```

Call this routine in a user interface object's initialization (main) routine instead of dsp_initclass. In addition adding the four methods bound to the names listed above, dsp_initboxclass also uses the name bxdsp.

New Instance Routine

Typical Max new instance routines specify how many inlets and outlets an object will have. An MSP signal object is no execption, but it uses proxies if you want more than a single signal inlet. You specify how many signal inlets you want with the dsp_setup call (or dsp_setupbox for user-interface signal objects). There is a requirement that signal inlets must be to the left of all non-signal inlets. Similarly, all signal outlets—declared simply with a type of "signal"—must be to the left of all non-signal outlets.

Here is an example of the initialization routine for an object that has two signal inlets and two signal outlets.

```
void *sigobj_new(void)
{
    t_sigobj *x;

    x = newobject(sigobj_class);
    dsp_setup((t_pxobject *)x,2);  // set up object and inlets
    outlet_new((t_object *)x,"signal"); // and outlets
    outlet_new((t_object *)x,"signal");
    return x;
}
```

Note that unlike the initialization routine in a typical Max object, the example routine above doesn't store pointers to its outlets. An MSP object almost never directly references its signal outlets. They are used by the signal compiler (and of course, the user, in order to connect your object's signal outputs to other objects' signal inputs).

dsp_setup

Use `dsp_setup` to initialize an instance of your class and tell MSP how many signal inlets it has.

```
void dsp_setup(t_pxobject *x, short num_signal_inputs);
```

Call this routine after creating your object in the new instance routine with `newobject`. Cast your object to `t_pxobject` as the first argument, then specify the number of signal inputs your object will have. `dsp_setup` initializes fields of the `t_pxobject` header and allocates any proxies needed (if `num_signal_inputs` is greater than 1). Some signal objects have no inputs; you should pass 0 for `num_signal_inputs` in this case. After calling `dsp_setup`, you can create additional non-signal inlets using `intin`, `floatin`, or `inlet_new`.

dsp_setupbox

Use `dsp_setupbox` to initialize an instance of your user interface object class and tell MSP how many signal inlets it has.

```
void dsp_setupbox(t_pxbox *x, short num_signal_inputs);
```

This routine is a version of `dsp_setup` for user interface signal objects.

Special Bits in the t_pxobject Header

There are three bits you can set in the `t_pxobject` or `t_pxbox` header that affect how your object is treated when MSP builds the DSP call chain. The explanation of these settings will make more sense once you have read more about the `dsp` and `perform` methods, but they are explained here because you need to set them in your new instance routine. Both `t_pxobject` and `t_pxbox` contain a field called `z_misc`; by default it is 0 meaning that all of the following settings are disabled.

```
#define Z_NO_INPLACE 1
```

If you set this bit in `z_misc`, the compiler will guarantee that all the signal vectors passed to your object will be unique. It is common that one or more of the output vectors your object will use in its perform method will be the same as one or more of its input vectors. Some objects are unable to handle this restriction; typically, this occurs when an object has pairs of inputs and outputs and writes an entire output on the basis of a single input before moving on to another input-output pair.

```
#define Z_PUT_LAST 2
```

If you set this bit in `z_misc`, the compiler puts your object as far back as possible on the DSP call chain. This is useful in two situations. First, your object's `dsp` routine might require that another object's `dsp` routine is called first in order to work properly. Second, if your object wants another object's perform routine to run before its own perform routine. For example, to minimize delay times, a delay line reading object probably wants the delay line writing object to run first. However, setting this flag does not guarantee any particular ordering result.

```
#define Z_PUT_FIRST 4
```

If you set this bit in `z_misc`, the compiler puts your object as close to the beginning of the DSP call chain as possible. This setting is not currently used by any standard MSP objects.

The dsp Method

Your `dsp` method is called by MSP when it is building the DSP call chain. If you want to add something to the chain, your method should call `dsp_add`, which adds your perform method to the DSP call chain. Your method should be declared as follows:

```
void sigobj_dsp(t_sigobj *x, t_signal **sp, short *count);
```

You are passed an array of `t_signal` structures that define your perform method's signal inputs and outputs. A `t_signal` contains a buffer of floats and a size `s_n`, which specifies the number of samples computed during any particular call to your perform routine. (This size is sometimes referred to as a **vector size**.) Currently, the vector size will be the same for all the signals you receive, although future versions of MSP may allow special objects that accept vectors of different sizes. A signal also has a **sampling rate**; currently this will always be equal to the global sampling rate. See Accessing Global Information below on how to obtain the global sampling rate and other current MSP settings. The `t_signal` structure is defined in **z_dsp.h**.

In addition to the array of `t_signals`, your `dsp` method is passed an array that specifies the number of connections to each input and output. Some MSP objects use this information to put different perform methods on the DSP call chain. For instance, the *~ object does some optimizing by using a simpler routine that multiplies a signal

signal by a constant if there is no signal connected to one of its inputs. In this case it uses the object's internal value, set either as an argument or via a float sent to the right inlet.

You may wish to use the dsp method initialize other internal variables used by your perform routine. For example, many objects require dividing by the sampling rate. Rather than dividing during the perform routine, which is expensive, you can calculate the reciprocal of the sampling rate in the dsp routine, store it, and then multiply by the reciprocal in the perform routine. The sampling rate may be different every time your dsp method is called, so if you use this technique, you must recalculate any values that depend on it in the dsp method, rather than just once in your new instance routine.

dsp_add

Use dsp_add to add your object's perform routine to the DSP call chain.

```
void dsp_add(t_perfroutine p, long argc, ...);
```

This function adds your object's perform method to the DSP call chain and specifies the arguments it will be passed. argc, the number of arguments to your perform method, should be followed by argc additional arguments, all of which must be the size of a pointer or a long.

dsp_addv

Use dsp_addv to add your object's perform routine to the DSP call chain and specify its arguments in an array rather than as arguments to a function.

```
void dsp_addv(t_perfroutine p, long argc, void **vector)
```

This function is a variant of dsp_add that allows you to construct an array of the arguments you wish to pass to your perform routine.

Here's an example of dsp method that doesn't pay attention to the connection count information might do. It has two inputs and two outputs. The inputs appear first in the array of signals, followed by the outputs, so sp[0] is the left input, sp[1] is the right input, sp[2] is the left output, and sp[3] is the right output.

```
void sigobj_dsp(t_sigobj *x, t_signal **sp, short *count)
{
    dsp_add(sigobj_perform, 5, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
        sp[3]->s_vec, sp[0]->s_n);
}
```

The above call to dsp_add specifies the name of the perform method, followed by the number of arguments that will be passed to it, followed by each argument. The s_vec field of a signal is its array of floats. In this case, the two input arrays are passed, followed by the two output arrays, followed by the vector size. You can pick

any t_signal to use for the vector size. By convention, most MSP objects use the first input signal.

Next, here's a more complex dsp method that uses a different perform method if its right input and right output are disconnected. One object that does something similar is fft~, where a routine that calculates only the real part of an FFT is used if the imaginary input and output are disconnected. In this example, sigobj_perform2 takes only three arguments, the signal vectors for the left input and left output, plus the vector size.

```
void sigobj_dsp(t_sigobj *x, t_signal **sp, short *count)
{
    if (count[1] || count[3]) // right input or right output connected
        dsp_add(sigobj_perform, 5, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
            sp[3]->s_vec, sp[0]->s_n);
    else
        dsp_add(sigobj_perform2, 3, sp[0]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}
```

Even if, according to the information in the count array passed to your dsp method, a t_signal is not connected to your object, the t_signal still contains a valid vector as well as valid sample rate and vector size information.

If for some reason you want to put several functions on the DSP call chain, you can do so: just make as many calls to dsp_add as you want.

The Perform Method

Your perform method is called repeatedly to calculate signal values. MSP calls each perform method in its DSP call chain in order with the arguments that were specified by the object's call to dsp_add. However, the arguments are not passed on the stack; instead, a pointer to an array containing the arguments is passed to the object. The perform method must return a pointer into the array just after the last argument specified by its dsp_add call. If this is not done, MSP will crash. Your method should be declared as follows:

```
t_int *sigobj_perform(t_int *w);
```

In the current implementation, MSP calls perform methods at interrupt time. As with any interrupt routine, your perform method should be written as efficiently as possible. It cannot call routines that would move memory, nor should it call post (for debugging), since at a 44.1 kHz sampling rate and vector size of 256 samples, each perform method is called about every 5.8 milliseconds. You can however, set a qelem or, if you're careful, use defer_low (*not* defer, since the MSP interrupt is not the Max scheduler interrupt, and thus defer doesn't know that it is being executed at interrupt level) to delay a function until the main level. You need to be careful because, if you defer a call every 5.8 milliseconds, you will cause a huge backlog of main event level functions that need to be run, as well as allocate a large amount of memory at interrupt level.

Here is a sample perform method that takes two signals as input, adds them together to produce one output and subtracts them to produce the other. The method is written so that it would be compatible with the `sigobj_dsp` example shown above. The type `t_int` is the same size as a pointer (int or long on the PowerPC); it is used for some degree of source code compatibility with `Pd` perform methods.

Note that the first argument that you specified in your call to `dsp_add` is at offset 1 in the array passed to your perform method. Offset 0 contains the address of your perform routine.

```
t_int *sigobj_perform(t_int *w)
{
    float *in1,*in2,*out1,*out2,val,val2;
    long n;

    in1 = (float *)(w[1]);   // input 1
    in2 = (float *)(w[2]); // input 2, second arg
    out1 = (float *)(w[3]); // arg 3, first output
    out2 = (float *)(w[4]); // arg 4, second output
    n = w[5]; // vector size

    // calculation loop
    while (n--) {
        val = *in1++;
        val2 = *in2++;
        *out1++ = val + val2
        *out2++ = val - val2;
    }

    return w + 6;  // always return a pointer to one more than the
                   // highest argument index
}
```

In the calculation loop, the code is written so that even if the output and input signal vectors are the same, the result is still correct. However, if for some reason you can't do this, you can specify that the input and output signal vectors should be unique with the `Z_NOINPLACE` flag. How to do this is explained above in the section entitled Special Bits in the `t_pxobject` header. (Only two standard MSP objects—**fft~** / **ifft~** and **tapout~**—require this feature.)

The Free Routine

If your normal object doesn't allocate any memory or need anything to be turned off when an instance is freed, you can pass `dsp_free` as the free method to `setup` in your initialization (main) routine. (User interface objects, even if they don't allocate memory themselves, require a free routine because they need to call `box_free`.)

If you do write your own free routine, your normal object should call `dsp_free` in it, and your user interface object should call `dsp_freebox`.

dsp_free

Use `dsp_free` in your object's free routine.

```
void dsp_free(t_pxobject *x);
```

This function disposes of any memory used by proxies allocated by `dsp_setup`. It also notifies the signal compiler that the DSP call chain needs to be rebuilt if signal processing is active.

dsp_freebox

For user interface objects, use `dsp_freebox` instead of `dsp_free`.

```
void dsp_freebox(t_pxbox *x);
```

This function disposes of any memory used by proxies allocated by `dsp_setupbox`. It also notifies the signal compiler that the DSP call chain needs to be rebuilt if signal processing is active.

CHAPTER 3

Handling Parameters

Real-time signal processing isn't just about calculating signals. You also want your DSP routines to respond to changes in input parameters. This task is often referred to as **parameter updating**. In Max, DSP parameters are typically control messages. You can either use a **sig~** or **line~** object to convert control messages to signals, or you can update the internal state of a signal object directly. The latter approach has the disadvantage of possible discontinuities in the output, but for many applications or when the user is experimenting, it is easier, not to mention more efficient.

Many MSP objects need to pass a pointer to themselves to the perform method to access internal state information. For example, a filter object that accepts floats to specify coefficients would need to pass itself to the perform method so that these coefficients can be accessed.

A Filter Example

As an example, here are the dsp and perform methods of a simple lowpass filter object called **lop~** that uses a coefficient stored in the object. Let's first assume that the coefficient, which is specified via the right inlet of the object, can only be passed as a float, not a signal. This means you'll have to declare an additional inlet and a method to accept the parameter. Here is the object declaration:

```
typedef struct _lop {
    t_pxobject x_obj;      // header
    float x_coeff;         // coefficient
    float x_m1;            // filter memory
} t_lop;
```

Here is the initialization routine:

```
void main(void)
{
    setup(&lop_class, lop_new, (method)dsp_free, (short)sizeof(t_lop), 0L,
        A_DEFFLOAT, 0);
    addftx(lop_ft1,1); // bind right inlet method
    addmess(lop_dsp,"dsp", A_CANT, 0);
    dsp_initclass();
}
```

Here is the right inlet method..

```
void lop_ft1(t_lop *x, double f)
{
    x->x_coeff = f;
}
```

Here is the new instance routine. There is only a single signal input, in the left inlet, so 1 is passed as the signal input count to `dsp_setup`.

```
void *lop_new(double initial_coeff)
{
    t_lop *x = newobject(lop_class);
    dsp_setup((t_pxobject *)x, 1);
    floatin((t_object *)x,1);
    outlet_new((t_object *)x,"signal");
    x->x_coeff = initial_coeff; // initialize coefficient
    x->x_m1 = 0.; // initialize previous state
    return x;
}
```

Here is the `dsp` method. It instructs MSP to pass the object's pointer, the input vector, the output vector, and the vector size to the perform routine. No signal connection counting is required; indeed, you could declare the method without the `count` parameter if you wanted to.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    dsp_add(lop_perform1, 4, x, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
}
```

Finally, here is the perform method. We have called it `lop_perform1` because we'll be writing alternative perform methods as we continue with the example. Note how we get the filter coefficient out of the object's structure and place it in a local variable. This is far more efficient than reading it out of the object during the calculation loop since the vector could be up to 2048 samples. Since the perform routine is executing at interrupt level, we are guaranteed that the coefficient won't change in the middle of the routine. The same is true for the filter's memory that is also stored inside the object.

```
t_int *lop_perform1(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]); // object is first arg
    float *in = (float *)(w[2]); // input is next
    float *out = (float *)(w[3]); // followed by the output
    long n = w[4]; // and the vector size
    loat xm1 = x->x_m1; // local to keep track of previous state
    float coeff = x->x_coeff,val; // and coefficient

    // filter calculation
    while (n--) {
        val = *in++;
        *out++ = coeff * (val + xm1);
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 5;
}
```

Now let's rewrite the filter to accept either a float or a signal for the coefficient value. There are two strategies for doing this depending on how often you want to read the coefficient value from a signal vector. First, let's write it with a single perform routine that makes a decision about whether to get the coefficient from a signal or from the float value stored inside the object. In this implementation, the coefficient is only read from the first value of the signal vector, and the rest of the vector is ignored.

We will add a field to our object that tells the perform routine whether a the dsp routine found that a signal was connected to the right inlet or not.

```
typedef struct _lop {
    t_pxobject x_obj;
    float x_coeff;
    float x_m1;
    short x_connected;
} t_lop;
```

Since MSP will be using a proxy to get the signal and the float in the right inlet, we need to change our initialization routine sightly. We replace

```
addftx(lop_ft1,1);
```

with

```
addfloat(lop_float);
```

Other than being renamed, the float method remains the same as the one above.

Here is the revised new instance routine that specifies two signal inlets. We have removed the creation of the additional inlet and changed the number of signal inlets specified in the call to `dsp_setup` to 2. `dsp_setup`, using a proxy, creates the right inlet for us.

```
void *lop_new(double initial_coeff)
{
    t_lop *x = newobject(lop_class);
    dsp_setup((t_pxobject *)x,2); // changed from previous example
    outlet_new((t_object *)x,"signal");
    x->x_coeff = initial_coeff; // initialize coefficient
    x->x_m1 = 0.; // initialize previous state
    return x;
}
```

Here is the revised dsp method. Since there are now two signal inlets, the output vector is at `sp[2]` rather than `sp[1]`.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    x->x_connected = count[1];    // save whether right inlet has a signal
                                  // going into it
    dsp_add(lop_perform2, 5, x, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
        sp[0]->s_n);
}
```

Here is the revised perform method. Depending on the value of the `x_connected` field of the object, it uses either the first value from the signal vector passed on the stack or the stored float value.

```
t_int *lop_perform2(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]);
    float *in = (float *)(w[2]);
    float coeff = x->x_connected? *(float *)(w[3]) : x->x_coeff; // use either
                //  signal or stored coefficient
    float *out = (float *)(w[4]);
    long n = w[5];
    float xm1 = x->x_xm1,val;

    while (n--) {
        val = *in++;
        *out++ = coeff * (val + xm1);
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 6; // 6 because there were now five arguments
}
```

The second strategy uses two different perform methods. The dsp method decides which one to use based on the count of signals connected to the right input. Other than the elimination of the `x_connected` field of the `t_lop` structure, only the dsp and perform methods change from the previous implementation of **lop~**. Here is the revised dsp method, which makes reference to the original `lop_perform1` method defined above. Even though there is an additional input signal to the object now, we can still use `lop_perform1` by passing only the left input signal vector and the output signal vectors. `lop_perform1` has no idea that there was another input signal vector.

```
void lop_dsp(t_lop *x, t_signal **sp, short *count)
{
    if (count[1])
        dsp_add(lop_perform3,5,x, sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec,
            sp[0]->s_n);
    else
        dsp_add(lop_perform1,4,x, sp[0]->s_vec, sp[2]->s_vec, sp[0]->s_n);
            // skip unused sp[1] signal
}
```

Finally, here is `lop_perform3`, which uses all of the values from the input coefficient signal in calculating the low-pass filter output. It is ignorant that there is a stored internal coefficient.

```
t_int *lop_perform3(t_int *w)
{
    t_lop *x = (t_lop *)(w[1]);
    float *in = (float *)(w[2]);
    float *coeff = (float *)(w[3]);
    float *out = (float *)(w[4]);
    long n = w[5];
    float xm1 = x->x_xm1,val;

    while (n--) {
        val = *in++;
        *out++ = *coeff++ * (val + xm1);
                // use each value in the coefficient signal vector
        xm1 = val;
    }
    x->x_m1 = xm1; // re-save old state for the next time
    return w + 6;
}
```

CHAPTER 4

Access to MSP Global Information

The following routines provide access to the global state of the DSP environment. The results may not be valid in your object's main (initialization) routine, and they may change between your object's new instance routine and its dsp method.

sys_getblksize

Use `sys_getblksize` to find out the current DSP vector size.

```
long sys_getblksize(void);
```

sys_getsr

Use `sys_getsr` to find out the current sampling rate.

```
float sys_getsr(void);
```

sys_getch

Use `sys_getch` to find out the current maximum number of channels.

```
long sys_getch(void);
```

sys_getdspstate

Use `sys_getdspstate` to find out whether the DSP is active or not.

```
long sys_getdspstate(void);
```

This function returns 1 if the DSP is active, 0 if it is not.

The following function returns information about an object's context in a DSP network.

dsp_isconnected

Use `dsp_isconnected` to determine whether two signal objects are connected.

```
short dsp_isconnected(t_object *src, t_object *dst, short
*index);
```

This function is useful only if you call it in your `dsp` method. It can be used to determine whether there is a signal connection from an outlet of src to an inlet of dst. The function returns a non-zero result if there is a connection, and zero if there isn't. The result is a count of the number of objects in between `src` and `dst` plus one. For example, if `dst` were directly below `src`, `dsp_isconnected` would return 1. `dsp_isconnected` returns in `index` the inlet number of `dst` (starting at 0) where the connection occurs. If there is more than one connection, information about the leftmost connection is returned.

Index