MC

MC is the system for multi-voice or multi-channel audio signals introduced with Max 8.

MC is not an entirely new API for MSP objects. Instead it is built on top of the existing MSP API. Is it implemented as some additions to the MSP signal compiler — the code that turns a graph of MSP objects into an ordered sequence of operations on signals — to deal with patch cords that hold more than one audio signal. Multi-channel signal patch cords co-exist with regular old- school patch cords as well as Jitter and event patch cords.

An important principle of Max is that outlet types define patch cord types. This is how Max knows, when the user clicks on an outlet to patch something, what kind of patch cord to make. Jitter objects have “matrix” outlets. Regular Max events travel through outlets that either have no type defined or a type such as “int” or “bang.”

Following this principle, if you want your MSP object to have outlets that produce multi-channel signals, you will have to change the type of the outlets from signal to multichannelsignal. An outlet of type multichannelsignal will have between 1 and 1024 signals in it. A patch cord coming from this type of outlet will be a special color and width, and can be connected to any MSP object, even those that don’t know about MC. (In that case, only the first channel will be used.)

The MC Wrapper

What are you hoping to do to your object within the MC universe? If it’s a filter, do you want N filters, one operating on every audio channel? If that’s all you want, you can avoid doing any coding simply by whitelisting your object to use the MC wrapper. Let’s say your object is called myfilter~ and you want the N-way version to be called mc.myfilter~. Add the following message to Max in any file that is evaluated at startup (that typically means you’ll put it in the init folder):

```
max objectfile mc.myfilter~ mc.wrapper~ myfilter~;
```

This establishes a mapping when the user types mc.myfilter~ into an object box. The MC wrapper looks at the name the user types, removes the “mc.” from the beginning, and looks for a Max object with the string that remains. So, this means you can’t do this:

```
max objectfile myNWAYfilter~ mc.wrapper~ myfilter~;
```

The name “myfilter~” at the end of this message specifies the name of the help file to open. If you want to make a special help file for the N-way version of your object, you could do this:

```
max objectfile mc.myfilter~ mc.wrapper~ mc.myfilter~;
```

Finally, you may provide an optional fourth argument to the objectfile message that specifies the tab to open in the help patcher. For example, when you open the help patcher for the mcs.limi~ object in Max the limi~ help patcher is opened to the tab named "mcs". This done using the pattern from this code:

```
max objectfile mc.myfilter~ mc.wrapper~ mc.myfilter~ <optional-helpfile-tabname>;
```

Multi-channel Inputs and Outputs

Maybe your concept of MC compatibility is not related to having N copies of your object in the wrapper. For example, your object might be concerned with audio signal input or output, either to the outside world or to a file. Perhaps you simply want to accept all inputs as a single multi- channel signal (which is nice if you don't want to decide in advance how many channels you will accept). Perhaps you want to produce a multi-channel signal instead of separate single-channel signals. Maybe your object mixed some stuff together that you realize would be nice not to mix together, so you'd like to provide each unmixed audio output together in a multi-channel patch cord.

For these cases, you can use the extensions to the MSP API described here. An MSP object that is MC-compatible will work in any version of Max with 64-bit floating-point. By convention, MSP objects that operate in both single-channel and multi-channel versions look at the object name symbol passed to the new instance routine and are multi-channel if the name begins with "mc." or "mcs." You are free to rebel against this standard and establish your own convention.

There are no special functions exported from Max or the MSP library specific to MC, so the only thing that will break in Max 7 and earlier versions is that your outlets won't work because only Max 8 knows about the multichannelsignal outlet type.

Creating Multi-channel Inlets and Outlets

For multi-channel inlets, you don't need to do anything special unless you want fewer inlets that you would otherwise. Just call **dsp_setup()** as you normally would and the user will be able to connect both single-channel and multi-channel patch cords. For multi-channel outlets, instead of

```
outlet_new(x, "signal");
```

use

```
outlet_new(x, "multichannelsignal");
```

Channel Counting in the Perform Method

Most of what you have to do is related to being a bit more careful about what you might previously have been able to assume about your perform method.

The prototype for your MSP perform method looks like this:

```
void myobject_perform64(t_myobject *x, t_object *dsp64, double **ins, long numins,
      double **outs, long numouts, long sampleframes, long flags, void *userparam);
```

Let's say your object will have one multi-channel signal input and one multi-channel signal output. As you can probably guess, the numins parameter to the perform method will be the count of channels in the input and the numouts parameter will be the count of channels in the output. Now let's consider some other cases because once we have more than one inlet and/or outlet, things get trickier.

Consider an object with two multi-channel inputs. You don't know in advance how many channels will be in each signal connected to your object. It could be 1 (if the user connects an old-style patch cord). It could be

100. It could be there is no connection at all to your object. What do you do?

If you're in this situation, you'll need to ask MSP how many channels are in each input in your dsp64 method, which is called before the DSP is turned on. The prototype for your dsp64 method looks like this:

```
void myobject_dsp64(t_myobject *x, t_object *dsp64, short *count, double samplerate,
       long maxvectorsize, long flags);
```

The dsp64 object passed to this method can be used to interrogate the number of channels in each of your object's inlets via the getnuminputchannels method. If your object has two inlets, here is how you can find out how many input channels each inlet has:

```
void myobject_dsp64(t_myobject *x, t_object *dsp64, short *count, double samplerate,
       long maxvectorsize, long flags)
{
    long leftinletchannelcount, rightinletchannelcount;
    leftinletchannelcount = (long)object_method(dsp64,gensym("getnuminputchannels"),
       x, 0);
    rightinletchannelcount = (long)object_method(dsp64,
       gensym("getnuminputchannels"), x, 1);
}
```

Note that an unconnected inlet has one channel, which, as has always been in the case in MSP, will be a signal containing all zeroes.

You might want to store these channel count values in your object so you can make use of them in your perform method. Then you'll know how to interpret the ins array of audio buffers you receive.

Specifying Output Channel Counts

The channel count for a multi-channel signal outlet is determined when MSP is building the DSP chain with the signal compiler. This means it can change each time the user turns the audio on if the graph has changed.

In MC, it's important to remember that outlets, not inlets, determine signal channel counts. You report the number of channels your object's outlets will have by supporting the multichanneloutputs method.

```
class_addmethod(c, (method)myobject_multichanneloutputs, "multichanneloutputs",
       A_CANT, 0);

...

long myobject_multichanneloutputs(t_myobject *x, long outletindex)
{
    return 4;
}
```

The above creates an outlet with four channels. Every time.

How to Auto-Adapt

If your object has defined multi-channel outlets it may receive the inputchanged message when the MSP signal compiler runs. This notifies your object how many channels are going to be sent to one of your object's inlets. You don't have to implement an inputchanged method, but you can use the information it provides to auto-adapt your object's number of output channels in one or more of your multi-channel signal outlets. The MC Wrapper performs auto- adapting when the user does not specify a fixed number of channels. For

example, if "mc.cycle~ 440 \@chans 64" is connected to the input of mc.*~, the wrapper will create 64 instances of a *~ object, one to multiply the output of each of the 64 cycle~ objects in the mc.cycle~.

Auto-adapting is a “conversational” protocol that involves both the inputchanged and multichanneloutputs methods.

Here’s how it works:

First, implement an the inputchanged method in addition to a multichannel outputs method:

```
class_addmethod(c, (method)myobject_inputchanged, "inputchanged", A_CANT, 0);

...

long myobject_inputchanged(t_myobject *x, long index, long count)
{
    // do something here and return true or false
}
```

Index is the inlet index of your object (with the leftmost being zero) and count is the number of audio channels in that particular inlet.

Your object’s inputchanged method should return true if its idea of how many outputs one of your outlets may be changing based on the information just received. It should return false it is not going to change. Returning false is polite and optimizes the speed of compiling the signal chain.

You should also store the count you receive somewhere in your object if you are going to use it to modify the count of output channels. After you return true from the inputchanged method your object’s multichanneloutputs method will be called for every multichannelsignal outlet your object has created. You can then return the new channel count based on the information received in the inputchanged method.

Here’s an illustration of the auto-adapting protocol with a simple example of an object with one inlet and one multichannelsignal outlet. First, the object will receive the inputchanged method:

```
long myobject_inputchanged(t_myobject *x, long index, long count)
{
    if (count != x->m_inputcount) {
        x->m_inputcount = count;
        return true;
    }
    else
        return false;
}

long myobject_multichanneloutputs(t_myobject *x, long index)
{
    return x->m_inputcount;
}
```

Note that the inputchanged and multichanneloutputs methods will be sent to your object before the dsp64 method.

It’s a good practice to use the getnuminputchannels technique inside your object’s dsp64 method as demonstrated above even if you support the inputchanged method. The signal compiler is not guaranteed to send the inputchanged message to your object in all cases — for example, it may not send inputchanged if there is nothing connected to one of your inlets, so for determining the output count you should assume one channel until inputchanged tells you something else. (Currently the inputchanged message is sent to objects

with unconnected inlets, but this adds some overhead, so we're investigating whether it's always necessary.)

How Many Output Channels Do I Have?

If you want to determine the count of signal output channels your object has for any of its multi- channel (or single-channel) outlets in your dsp64 method, you can send the message getnumoutputchannels to the dsp64 object:

```
long leftoutletchannelcount = (long)object_method(dsp64,
      gensym("getnumoutputchannels"), x, 0);
```

The getnumoutputchannels method of the dsp64 object works the same as the getnuminputchannels method.

Here's something potentially unintuitive about the MC signal compiler: it always gives you the number of output channels you want, whether or not that's a good idea. Here's what this means. Suppose you have an object with two multichannelsignal outlets. In your dsp64 method, you notice that one of these outlets is not connected to anything (its entry in the count[] array is zero). You might assume this means the output channel count is zero, or maybe one (as with the number of channels given to an unconnected inlet). However, in reality it will be the number of channels you returned for this outlet in your multichanneloutputs method. The principle that applies here, established since the first version of MSP, is that you should never have to modify the behavior of your perform method based on whether its outputs are connected. You could choose not to call dsp_add64 in a case where none of your object's outlets were connected in which case your perform method won't be called. But if the perform method is going to be called, it will receive the number of outputs you request.

To summarize, if your object has two outlets and it has returned a value of 12 for each outlet in its multichanneloutputs method, the perform method will receive a total of 24 output channels (in other words, the numouts parameter will be 24).

Handling MC Signals in Traditional MSP Objects

When an MC signal is connected to to a traditional MSP object, then only the first channel of a multi-channel patch cord is handed to the object.

If you want to modify this behavior and receive all of the channels the you must supply the Z_MC_INLETS flag. Having supplied this flag, a user can connect single-channel patch cords, multi-channel patch cords, or both – and you'll have to make the best out of the situation.

Additional Help File Support

In addition to the helpfile tab argument in objectmappings mentioned above, you may wish to provide additional help patchers for your object.

When a user asks to open a help patcher for an object and there is only one help patcher available then that one help patcher is opened immediately. If a user asks to open a help patcher for an object with multiple help patchers then a menu is provided and the user selects the appropriate help patcher. Support for multiple help patchers was added in Max 8 to support MC. Specifically, the features of the MC wrapper have thier own help patcher which can then be shared across multiple objects.

As an example, the mc.limi~ object uses the wrapper. To make the shared wrapper help patcher available as an option for mc.limi~, the following line is placed in an init file:

```
max classnamehelpcategory mc.limi~ mcwrap;
```

The category mcwrap is defined with a line also in the init file which looks like this:

```
max categoryhelp mcwrap mcwrapper-group "MC Wrapper Features Help";
```

The first argument to "categoryhelp" is the name of the category to create. The second argument is the name of a help patcher in the searchpath (minus its suffix). The third argument is the string that will appear in the menu.

To see the existing categories and mappings, look at the file named "mc-helpconfig.txt " inside the application bundle's init folder.

Examples

The first example is a signal visualizer called gridmeter~. It is included to demonstrate just two changes for objects receiving multi-channel inputs:

- First, the object checks the value returned by the getnuminputchannels method in the dsp64 method. This permits it to know how many channels to paint in the grid.
- Second, it does not assume a specific count of channels in its perform method, which was typically the case with meter objects in MSP. Instead, it has a loop for each input channel up to the value of the numins parameter.

The second example is called mc.rotate~ and demonstrates how to implement the auto-adapting protocol. mc.rotate~ simply rotates all the channels in any multi-channel signal it receives by one, but it will produce the same number of output channels as the number of inputs it receives.

Finally, there is also the example of mc.pack~ which takes only the first channel of any connected multi-channel signal. But if you type mc.combine~ instead it uses all the channels. mc.pack~ and mc.combine~ are the same object, but behave differently based on the convention that something called “pack” doesn’t produce more outputs than it has inlets. This demonstrates the use of the Z_MC_INLETS flag.