Writing Externals for Max 6

Max 6 introduces 64-bit audio processing. Pre-existing audio externals built for Max 5 can be run under Max 6 without alterations. Pre-existing Max 5 code can be compiled for Max 6 without alterations, and the resulting external will run on Max 6 and earlier versions. In order to make use of 64-bit processing, you will need to revise your code slightly. This primarily involves writing a new 64-bit perform routine, and a new dsp method bound to the message "dsp64" to call the 64-bit perform routine. Once these revisions are in place, Max 6 will call your "dsp64" perform routine, and earlier versions of Max will call your original perform routine.

Revising multy~

We will use *multy~* from chapter 3 to demonstrate porting from Max 5 to Max 6. At the time of this writing a Max 6 SDK has not been publicly released. Therefore we cannot provide an actual Max 6 project on the CD-ROM. Instead, the revised code file "multy~.c" is included in this folder. When the Max 6 SDK is available, you can simply move the revised *multy~* code into a Max 6 project.

Writing the 64-bit Perform Routine

We first refresh our memory of the Max 5 perform routine for *multy~*, shown in Figure 1.

```
 1 t_int *multy_perform(t_int *w)
 2 {
 3         t_multy *x = (t_multy *) (w[1]);
 4         t_float *in1 = (t_float *) (w[2]);
 5         t_float *in2 = (t_float *) (w[3]);
 6         t_float *out = (t_float *) (w[4]);
 7         t_int n = w[5];
 8         while(n--){
 9              *out++ = *in1++ * *in2++;
10         }
11         return w + 6;
12 }
```

Figure 1 The Max 5 perform routine for *multy~*.

All signal vectors, along with the object, are taken from the `*w` integer array. The order of these elements must correspond to the order in which they are sent in the `dsp_add()` function call inside your dsp method. In line 11, make certain to return the correct pointer, or the external will crash.

Now let's have a look at the 64-bit version of this perform routine, shown in Figure 2.

```
 1 void multy_perform64(t_multy *x, t_object *dsp64,
     double **ins, long numins, double **outs,
     long numouts, long vectorsize, long flags,
     void *userparam)
 2 {
 3         t_double *in1 = ins[0];
 4         t_double *in2 = ins[1];
 5         t_double *out = outs[0];
 6         int n = vectorsize;
 7         while(n--){
 8              *out++ = *in1++ * *in2++;
 9         }
10 }
```

Figure 2 The Max 6 64-bit perform routine for *multy~*.

In line 1 of Figure 2, several new arguments are passed. These arguments provide more structure than in the previous style (characteristic of Max 4, Max 5 and Pd), where any mix of signal vectors, objects and other data could be passed in any order on the integer array `*w`. The 64-bit perform routine distinguishes between inlets and outlets, provides the signal vector size, and provides the number of signal inlets and outlets. Since we are now doing 64-bit processing, the signal vectors are declared as type `t_double` (which is defined as `double` in the new header file "z_sampletype.h").

Despite these changes, the DSP algorithm itself (in lines 7-9 of Figure 2) remains exactly the same as for the older *multy~* perform routine. However there is one thing missing: We no longer need to return a pointer to the next address on the DSP chain. Max 6 deals with signal routing behind the scenes, in order to facilitate smooth transitions whenever the DSP configuration changes (thus no more glitches when adding new audio objects to a patch). Since a pointer is no longer returned, you can no longer crash Max/MSP by returning the wrong pointer. With nothing to return, `multy_perform64()` is declared as type `void` rather than `t_int*`.

Writing the dsp64 Method

The remaining steps are purely mechanical. We need to write a dsp64 method that calls our 64-bit perform routine. To see the differences, compare the old dsp method in Figure 3 with the new one in Figure 4.

In line 3 of both methods is a diagnostic post statement which will confirm that the appropriate dsp method is called when the DACs are turned on.

```
1 void multy_dsp(t_multy *x, t_signal **sp, short *count)
2 {
3         post("Executing the 32-bit perform routine");
4         dsp_add(multy_perform, 5, x, sp[0]->s_vec,
               sp[1]->s_vec,sp[2]->s_vec, sp[0]->s_n);
5 }
```

Figure 3 The Max 5 dsp method for multy~.

```
1 void multy_dsp64(t_multy *x, t_object *dsp64,
     short *count, double samplerate, long maxvectorsize,
     long flags)
2 {
3         post("Executing the 64-bit perform routine");
4         dsp_add64(dsp64, (t_object*)x,
               (t_perfroutine64)multy_perform64,
               0, NULL);
5 }
```

Figure 4 The Max 6 dsp64 method for *multy~*.

In line 1 of Figure 4, a few new arguments are passed to the dsp64 method. The sampling rate and vector size are passed as arguments, which is more convenient then extracting them from signal vector components, as we previously did in Max 5 code. The object `*dsp64` is actually the signal chain to which your object belongs. In Max 6, multiple signal chains are possible. Each top level Max 6 patcher has its own signal chain, as does each instance of *poly~* and *pfft~*. You need to pass the dsp64 signal chain object to your 64-bit perform routine, but can otherwise safely ignore it. The call to `dsp_add64()` passes the signal chain, your object (which must be cast to type `t_object*`), your 64-bit perform routine (cast to `t_perfroutine64`), and then any flags, and any user parameters. We use `0` and `NULL` respectively for those parameters.

In the dsp64 method, individual signal vectors and the argument count are no longer passed as parameters. This makes the coding easier, and eliminates a frequent source of fatal bugs. Of course the treatment of signal inlets and outlets in the 64-bit perform routine must still be consistent with how they were defined in the new instance routine. This is, however, a much easier task than keeping track of the arbitrary assignment of signal vectors in the old-style dsp method and perform routine.

All that remains is to bind the `dsp64()` routine to the message “dsp64” in `main()` and to provide function prototypes for both the dsp64 routine and the new perform routine. The new binding is shown in Figure 5.

```
class_addmethod(multy_class, (method)multy_dsp64, "dsp64",
      A_CANT, 0);
```

Figure 5 Binding the dsp64 method for *multy~*.

The file “multy~.c” in this folder contains complete code for the Max 6 version of multy~.

Backwards Compatibility with Max 5

Max 5 does not implement the function `dsp_add64`(), so as soon as you introduce this function in your dsp64 method, the resulting external will no longer load under Max 5. This is not a problem if you are only targeting Max 6 as a platform. But in order to compile an external that will load under both Max 5 and Max 6, a different approach is required. In the code shown in Figure 4, line 4, your object is added to the DSP chain with a call to `dsp_add64()`. However, since the DSP chain has already been passed to your dsp64 routine as the dsp chain object `*dsp64`, we can add *multy~* to the DSP chain by using the `object_method()` function to send the message "dsp_add64" directly to the dsp chain object. This is shown in line 4 of Figure 6.

```
1 void multy_dsp64(t_multy *x, t_object *dsp64, short *count,
     double samplerate, long maxvectorsize, long flags)
2 {
3    post("Executing the 64-bit perform routine");
4    object_method(dsp64, gensym("dsp_add64"), x,
           multy_perform64, 0, NULL);
5 }
```

Figure 6 Adding *multy~* to the DSP chain with an `object_method()` call.

Buffers and Max 6

Currently, buffers in Max 6 contain 32-bit floats, just like in Max 5. This means that *bed* from chapter 7 will run perfectly in Max 6 without modification. Since *bed* is a non-real-time external, there is no need to even recompile it for Max 6. But if you do, *bed* will still work with Max 5, since there is no call to `add_dsp64()`. In the event that a later version of Max introduces 64-bit buffers, then *bed* would need to be revised, mainly by changing `float` variables to `double`.

How 64-bit is Max 6?

Audio processing in Max 6 using the new style of perform routines is done with 64-bit precision. Older perform routines do their internal processing with 32-bit precision, with conversion down to 32 bits at their signal inputs, and conversion up to 64 bits at their signal outputs. As we have seen, buffer operations are done with 32-bit precision, and floating point Max messages remain limited to 32-bit precision. All of this suggests that Max 6 represents a transitional stage of the program, which appears to be moving toward implementing a consistent 64-bit signal path throughout the entire program.