Developing Supercollider Unit Generators Under GNU/ Linux

Matthew John Yee-King

June 4, 2007

1 Introduction

This is a series of tutorials about building custom Unit Generators for the SuperCollider music software. I am a fairly experienced Java programmer who has been using SuperCollider for a couple of years. I reached the point where I wanted to start coding my own UGens for timbral analysis and thought I should document the process by which I learnt how to do this in the form of a tutorial. This is my first time using C++ so this learning process is also documented. Hopefully this will be of use to others who are new to C++ too.

In this first part, we will set up the development environment, compile the example MySaw UGen from the SuperCollider documentation then write and compile a simple square wave oscillator.

2 Set up the developement environment

You will first need to set up your development environment by installing various libraries and software build tools. If you already have a self compiled SC3 running under Linux, you have already installed everything you need. If not, instructions can be found on the Linux pages of [2]. The following steps tell you how to set up a simple directory structure for building your own UGens along with a scons ([1]) build script:

1. Create your development folder, e.g. myk_plugs.
2. Create a folder called source and another called classes
3. In the top level folder, create a file called SConstruct which contains the text in 4.1.

3 Compiling the example from the SC3 documentation

In the SuperCollider documentation, there is some code for a simple example UGen called MySaw. Let's find out how to compile this under Linux.

1. In the source folder, create a file called MySaw.cpp containing the code in 4.2 (taken from the SuperCollider docs):

   Note that you should edit the line sc3_source so it points to the location of the SuperCollider source code tree. It can be a relative path as in the example or an absolute path. You are now ready to create the UGen.

2. cd back to the top level folder and type 'scons'. Your UGen will now be compiled. You should see a file called MySaw.so in the top level folder. This is the dynamically linkable library that tells the SuperCollider server program about your UGen.

3. So the server can see this library, you must add it to the search path. In my case, I created a symbolic link from /usr/share/SuperCollider/Extensions/myk_plugs to my top level build directory which contains the MySaw.so file.

   ```
   ls -l /usr/share/SuperCollider/Extensions/
   2007-06-03 16:59 myk_plugs -> /home/matthew/src/SC_64/myk_plugs
   2007-05-25 15:27 scel
   ```

   This is not an elegant solution, but we will improve this later.

4. The next step is to tell the language side about your new UGen so that sclang is able to create send synthdefs to the server which use this UGen. In the classes folder of your build directory, create a file called MySaw.sc containing the code in 4.3

5. Now start the server and test your UGen, e.g.:

   ```
   {MySaw.ar(MouseX.kr(20, 500))}.play
   ```

4 Code

4.1 SConstruct

```
# scons build script.
# original by blackrain at realizedsound dot net - 11 2006

# edit this to point to your SuperCollider3 source directory

sc3_source = '../supercollider'
```

```
##############################################
# simple ugens
headers = sc3_source + '/headers'

plugs = [
        'MySaw'
]

for file in plugs :
        Environment(
                CPPPATH = [headers + '/common', headers + '/plugin_interface', headers +
                CPPDEFINES = ['SC_LINUX', '_REENTRANT', 'NDEBUG', ('SC_MEMORY_ALIGNMENT'
                CCFLAGS = ['-Wno-unknown-pragmas'],
                CXXFLAGS =  ['-Wno-deprecated', '-O3'],
                SHLIBPREFIX = '',
                SHLIBSUFFIX = '.so'
        ).SharedLibrary(file, 'source/' + file + '.cpp');
```

4.2 MySaw.cpp

```
#include "SC_PlugIn.h"

// InterfaceTable contains pointers to functions in the host (server).
static InterfaceTable *ft;

// declare struct to hold unit generator state
struct MySaw : public Unit
{
  double mPhase; // phase of the oscillator, from -1 to 1.
  float mFreqMul; // a constant for multiplying frequency
};

// declare unit generator functions
extern "C"
{
  void load(InterfaceTable *inTable);
  void MySaw_next_a(MySaw *unit, int inNumSamples);
  void MySaw_next_k(MySaw *unit, int inNumSamples);
  void MySaw_Ctor(MySaw* unit);
};

//////////////////////////////////////////////////////////////////
```

```
// Ctor is called to initialize the unit generator.
// It only executes once.

// A Ctor usually does 3 things.
// 1. set the calculation function.
// 2. initialize the unit generator state variables.
// 3. calculate one sample of output.
void MySaw_Ctor(MySaw* unit)
{

  // 1. set the calculation function.
  if (INRATE(0) == calc_FullRate) {
    // if the frequency argument is audio rate
    SETCALC(MySaw_next_a);
  } else {
    // if the frequency argument is control rate (or a scalar).
    SETCALC(MySaw_next_k);
  }

  // 2. initialize the unit generator state variables.
  // initialize a constant for multiplying the frequency
  // scales from 0-22050 -> 0-1
  unit->mFreqMul = 2.0 * SAMPLEDUR;
  // get initial phase of oscillator
  unit->mPhase = IN0(1);
  // 3. calculate one sample of output.
  MySaw_next_k(unit, 1);
}

//////////////////////////////////////////////////////////////////

// The calculation function executes once per control period
// which is typically 64 samples.

// calculation function for an audio rate frequency argument
void MySaw_next_a(MySaw *unit, int inNumSamples)
{
  // get the pointer to the output buffer
  float *out = OUT(0);
  // get the pointer to the input buffer
  float *freq = IN(0);
  // get phase and freqmul constant from struct and store it in a
  // local variable.
  // The optimizer will cause them to be loaded it into a register.
  float freqmul = unit->mFreqMul;
  double phase = unit->mPhase;
  // perform a loop for the number of samples in the control period.
  // If this unit is audio rate then inNumSamples will be 64 or whatever
  // the block size is. If this unit is control rate then inNumSamples will
  // be 1.
  for (int i=0; i < inNumSamples; ++i)
    {
      // out must be written last for in place operation
      float z = phase;
      phase += freq[i] * freqmul;
      // these if statements wrap the phase a +1 or -1.
      if (phase >= 1.f) phase -= 2.f;
      else if (phase <= -1.f) phase += 2.f;
      // write the output
      out[i] = z;
    }

  // store the phase back to the struct
  unit->mPhase = phase;
}

//////////////////////////////////////////////////////////////////

// calculation function for a control rate frequency argument
void MySaw_next_k(MySaw *unit, int inNumSamples)
{
  // get the pointer to the output buffer
  float *out = OUT(0);

  // freq is control rate, so calculate it once.
  float freq = IN0(0) * unit->mFreqMul;
  // get phase from struct and store it in a local variable.
  // The optimizer will cause it to be loaded it into a register.
  double phase = unit->mPhase;
  // since the frequency is not changing then we can simplify the loops
  // by separating the cases of positive or negative frequencies.
  // This will make them run faster because there is less code inside the loop.
  if (freq >= 0.f) {
    // positive frequencies
    for (int i=0; i < inNumSamples; ++i)
      {
out[i] = phase;
phase += freq;
if (phase >= 1.f) phase -= 2.f;
      }
  } else {
    // negative frequencies
    for (int i=0; i < inNumSamples; ++i)
      {
out[i] = phase;
phase += freq;
if (phase <= -1.f) phase += 2.f;
      }
  }

  // store the phase back to the struct
  unit->mPhase = phase;
}

////////////////////////////////////////////////////////////////////

// the load function is called by the host when the plug-in is loaded
void load(InterfaceTable *inTable)
{
  ft = inTable;

  DefineSimpleUnit(MySaw);
}

////////////////////////////////////////////////////////////////////
```

4.3 MySaw.sc

```
MySaw : UGen {
  *ar { arg freq = 440.0, iphase = 0.0, mul = 1.0, add = 0.0;
^this.multiNew('audio', freq, iphase).madd(mul, add)
  }
  *kr { arg freq = 440.0, iphase = 0.0, mul = 1.0, add = 0.0;
^this.multiNew('control', freq, iphase).madd(mul, add)
  }
}
```

References

[1] Various. SCons: A software construction tool. Website, 2007.

[2] Various. SuperCollider swiki. website, 2007.