

# 25

## Writing Unit Generator Plug-ins

Dan Stowell

Writing a unit generator (UGen) for SuperCollider 3 can be extremely useful, allowing the addition of new audio generation and processing capabilities to the synthesis server. The bulk of the work is C++ programming, but the API (Application Programming Interface) is essentially quite simple—so even if you have relatively little experience with C/C++, you can start to create UGens based on existing examples.

You’re probably already familiar with UGens from other chapters. Before creating new UGens of your own, let’s first consider what a UGen really is, from the plug-in programmer’s point of view.

### 25.1 What Is a UGen, Really?

A UGen is a component for the synthesis server, defined in a plug-in, which can receive a number of floating-point data inputs (audio- or control-rate signals or constant values) and produce a number of floating-point data outputs, as well as “side effects” such as writing to the post window, accessing a buffer, or sending a message over a network. The server can incorporate the UGen into a synthesis graph, passing data from 1 UGen to another.

When using SC language, we need to have available a representation of each UGen which provides information about its inputs and outputs (the number, type, etc.). These representations allow us to define synthesis graphs in SC language (Synth-Defs). Therefore, each UGen also comes with an SC class; these classes are always derived from a base class, appropriately called `UGen`.

So to create a new UGen you need to create both the plug-in for the server and the class file for the language client.

### 25.2 An Aside: Pseudo UGens

Before we create a “real” UGen, we’ll look at something simpler. A *pseudo UGen* is an SC class that “behaves like” a UGen from the user’s point of view but doesn’t

involve any new plug-in code. Instead, it just encapsulates some useful arrangement of existing units. Let's create an example, a simple reverb effect:

```
Reverb1 {
  *ar {| in |
    var out = in;
    out = AllpassN.ar(out, 0.05, 0.05.rand, 1);
    ^out;
  }
}
```

This isn't a very impressive reverb yet, but we'll improve it later.

As you can see, this is a class like any other, with a single class method. The `*ar` method name is not special—in fact, you could use any method name (including `*new`). We are free to use the full power of SC language, including constructs such as `0.05.rand`, to choose a random delay time for our effect. The only real requirement for a pseudo UGen is that the method returns something that can be embedded in a synth graph. In our simple example, what is returned is an `AllpassN` applied to the input.

Copy the above code into a new file and save it as, for instance, *Reverb1.sc* in your *SCClassLibrary* or *Extensions* folder; then recompile. You'll now be able to use `Reverb1.ar` within your `SynthDefs`, just as if it were a “real” UGen. Let's test this:

```
s.boot;
(
x = {
  var freq, son, out;
  // Chirps at arbitrary moments
  freq = EnvGen.ar(Env.perc(0, 0.1, 10000), Dust.ar(1));
  son = SinOsc.ar(freq, 0, 0.1);
  // We apply reverb to the left and right channels separately
  out = {Reverb1.ar(son, cutoff: 2500)}.dup;
}.play(s);
)
x.free;
```

You may wish to save this usage example as a rudimentary Help file, *Reverb1.html*.

To make the reverb sound more like a reverb, we modify it to perform 6 similar all-pass delays in a row, and we also add some LPF units in the chain to create a nice frequency roll-off. We also add parameters:

```
Reverb1 {
  *ar {| in, wet = 0.3 , cutoff = 3000|
    var out = in;
    6.do{out = LPF.ar(AllpassN.ar(out, 0.05, 0.05.rand, 1), cutoff)};
  }
```

```
        ^(out * wet) + (in * (1 - wet));  
    }  
}
```

This is on the way toward becoming a useful reverb unit without having created a real plug-in at all.

This approach has definite limitations. It is of course confined to processes that can be expressed as a combination of existing units—it can't create new types of processing or new types of server behavior. It may also be less efficient than an equivalent UGen, because it creates a small subgraph of units that pass data to each other and must maintain their own internal states separately.

Now let's consider what is involved in creating a “real” UGen.

### 25.3 Steps Involved in Creating a UGen

1. First, consider exactly what functionality you want to encapsulate into a single unit. An entire 808-drum machine, or just the cymbal sound? Smaller components are typically better, because they can be combined in many ways within a SynthDef. Efficiency should also be a consideration.
2. Second, write the Help file. Really—it's a good idea to do this before you start coding, even if you don't plan to release the UGen publicly.

As well as being a good place to keep the example code which you can use while developing and testing the UGen, it forces you to think clearly about the inputs and outputs and how the UGen will be used in practice, thus weeding out any conceptual errors.

A Help file is also a good reminder of what the UGen does—don't underestimate the difficulties of returning to your own code, months or years later, and trying to decipher your original intentions!

The Help file will be an HTML file with the same name as the UGen. There is a “Documentation Style Guide” in the SC Help system which includes tips and recommendations for writing Help documentation. But, of course, during development the Help file doesn't need to be particularly beautiful.

3. Third, write the class file. You don't need to do this before starting on the C++ code, but it's a relatively simple step. Existing class files (e.g., for SinOsc, LPF, Pitch, Dwhite) can be helpful as templates. More on this shortly.
4. Fourth, write the plug-in code. The programming interface is straightforward, and again existing plug-in code can be a helpful reference: all UGens are written as plug-ins—including the “core” UGens—so there are lots of code examples available.

We now consider writing the class file and writing the plug-in code.

## 25.4 Writing the Class File

A class file for a UGen is much like any other SC class, with the following conditions:

It must be a subclass of UGen. This is so that methods defined in the UGen class can be used when the language builds the SynthDef (synth graph definition).

The name of the class must match the name used in the plug-in code—the class name is used to tell the server which UGen to instantiate.

It must implement the appropriate class methods for the rates at which it can run (e.g., \*ar, \*kr, and/or \*ir). These method names are referenced for rate checking during the SynthDef building process.

The class methods must call the multiNew method (defined in the main UGen class), which processes the arguments and adds the UGen correctly to the SynthDef that is being built.

The class file does not have any direct connection with the C++ plug-in code—after all, it’s the server that uses the plug-in code, while the class file is for the language client.

Let’s look at a well-known example:

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

As you can see, SinOsc is a subclass of UGen and implements 2 class methods. Both of these methods call multiNew and return the result, which is 1 or more instances of the UGen we are interested in. The methods also call madd, which we’ll discuss shortly.

The first argument to multiNew is a symbol to indicate the rate at which the particular UGen instance will be operating: this could be “audio,” “control,” “scalar,” or “demand.” The remaining arguments are those that will actually be passed to the C++ plug-in—here freq and phase. If any of these arguments are arrays, multiNew performs multichannel expansion, creating a separate unit to handle each channel. Indeed, this is why the method is called multiNew.

Note that the mul and add arguments are not being passed in to multiNew. This means that the actual plug-in code for SinOsc will never be able to access them. In-

stead, this UGen makes use of the `madd` method, which is essentially a convenience for multiplication and addition of the unit’s output. As well as avoiding the programmer’s having to implement the multiplication and addition part of the process, the `madd` method performs some general optimizations (e.g., in the very common degenerate case of multiplying by 1 and adding 0; no processing is really required, so the UGen is simply returned unaltered). It is the convention to add `mul` and `add` arguments to UGens as the final 2 arguments, as is done here; these 2 arguments are often very useful and are supported by many UGens. (Due to their commonness, they are often undocumented in Help files.)

Let’s start to draft the class file for a UGen we can implement. We’ll create a basic “flanger” which takes some input and then adds an effect controlled by rate and depth parameters:

```
Flanger : UGen {
  *ar {
    arg in, rate = 0.5, depth = 1.0, mul = 1.0, add = 0.0;
    ^this.multiNew('audio', in, rate, depth).madd(mul, add)
  }
  *kr {
    arg in, rate = 0.5, depth = 1.0, mul = 1.0, add = 0.0;
    ^this.multiNew('control', in, rate, depth).madd(mul, add)
  }
}
```

Save this as *Flanger.sc* in your extensions directory. If you recompile, you’ll find that this is sufficient to allow you to use `Flanger.ar` or `Flanger.kr` in `SynthDefs`, which the SuperCollider language will happily compile—but of course those `SynthDefs` won’t run yet, because we haven’t created anything to tell the server how to produce the Flanger effect.

### 25.4.1 Checking the Rates of Your Inputs

Because SuperCollider supports different signal rates, it is useful to add a bit of “sanity checking” to your UGen class to ensure that the user doesn’t try to connect things in a way that doesn’t make sense: for example, plugging an audio-rate value into a scalar-rate input.

The UGen class provides a `checkInputs` method which you can override to perform any appropriate checks. When the `SynthDef` graph is built, each UGen’s `checkInputs` method will be called. The default method defined in UGen simply passes through to `checkValidInputs`, which checks that each of the inputs is really something that can be plugged into a synth graph (and not some purely client-side object such as, say, an `SCWindow` or a `Task`).

The `BufWr` UGen is an example which implements its own rate checking. Let's look at what the class does:

```
checkInputs {
  if (rate == 'audio' and: {inputs.at(1).rate != 'audio'}, {
    ^("phase input is not audio rate:" + inputs.at(1) + inputs.at(1).
    rate);
  });
  ^this.checkValidInputs
}
```

If `BufWr` is used to write audio-rate data to a buffer, then the input specifying the phase (i.e., the position at which data is written) must also be at audio rate—there's no natural way to map control-rate index data to a buffer which is taking audio-rate data. Therefore the class overrides the `checkInputs` method to test explicitly for this. The `rate` variable is the rate of the unit under consideration (a symbol, just like the first argument to `multiNew`). The `inputs` variable is an array of the unit's inputs, each of which will be a UGen and thus will also have a `rate` variable. So the method compares the present unit's rate against its first input's rate. It simply returns a string if there's a problem (returning anything other than `nil` is a sign of an error found while checking input). If there's not a problem, then it passes through to the default `checkValidInputs` method—if you implement your own method checking, don't forget to pass through to this check.

Many UGens produce output at the same rate as their first input—for example, filters such as LPF or HPF. If you look at their class definition (or their superclass, in the case of LPF and HPF—an abstract class called `Filter`), you'll see that they call a convenience method for this common case called `checkSameRateAsFirstInput`. Observe the result of these checks:

```
s.boot;
x = {LPF.ar(WhiteNoise.kr)}.play(s); // Error
x = {LPF.ar(WhiteNoise.ar)}.play(s); // OK
x.free;
x = {LPF.kr(WhiteNoise.ar)}.play(s); // Error
x = {LPF.kr(WhiteNoise.kr)}.play(s); // OK
x.free;
```

What happens if you don't add rate checking to your UGens? Often it makes little difference, but ignoring rate checking can sometimes lead to unusual errors that are hard to trace. For example, a UGen that expects control-rate input is relatively safe, because it expects less input data than an audio-rate UGen—so if given audio-rate data, it simply ignores most of it. But in the reverse case, a UGen that expects audio-rate data but is given only control-rate data may read garbage input from memory that it shouldn't be reading.

Returning to the `Flanger` example created earlier, you may wish to add rate checking to that class. In fact, since the `Flanger` is a kind of filter, you might think it sensible to use the `checkSameRateAsFirstInput` approach, either directly or by modifying the class so that it subclasses `Filter` rather than `UGen`.

## 25.5 Writing the C++ Code

### 25.5.1 Build Environments: Xcode, `scons` . . .

`UGen` plug-ins are built just like any other C++ project. To make things easier for yourself as a developer, you can use and adapt 1 of the project files which are distributed along with `SuperCollider`'s source code:

On *Mac*, the Xcode project file *Plugins.xcodeproj* is used to build the core set of `SuperCollider` plug-ins. It's relatively painless to add a new "target" to this project in order to build your own plug-ins—this is the approach used in the `SuperCollider` Help document "Writing Unit Generators," which has more details about the Xcode specifics.

On *Linux*, the `scons` project file *SConstruct* is used to build `SuperCollider` as a whole. You can edit this file using a text editor to add your plug-in's build instructions. Alternatively, the "sc3-plug-ins" SourceForge project provides an *SConstruct* file purely for building `UGens`—you may find it easier to start from that as a template.

On *Windows*, Visual Studio project files are provided to compile plug-ins, including a *UGEN\_TEMPLATE\_VCPROJ.vcprojtemplate* file which you can use as a basis.

You can, of course, use other build environments if you prefer.

### 25.5.2 When Your Code Will Be Called

The server (`scsynth`) will call your plug-in code at 4 distinct points:

When `scsynth` boots, it calls the plug-in's `load()` function, which primarily declares which `UGens` the plug-in can provide.

When a `UGen` is instantiated (i.e., when a synth starts playing), `scsynth` calls the `UGen`'s *constructor* function to perform the setting up of the `UGen`.

To produce sound, `scsynth` calls each `UGen`'s *calculation* function in turn, *once for every control period*. This is typically the function which does most of the interesting work in the `UGen`. Since it is called only once during a control period, this function must produce either a single control-rate value or a whole block's worth of audio-rate values during 1 call. (Note: Demand `UGens` don't quite fit this description and will be covered later.)

When a synth is ended, some UGens may need to perform some tidying up, such as freeing memory. If so, these UGens provide a *destructor* function which is called at this point.

### 25.5.3 The C++ Code for a Basic UGen

The code in figure 25.1 shows the key elements we need to include in our Flanger plug-in code.

Here is what this code does:

First, the `#include` command calls the main header file for SuperCollider’s plug-in interface, *SC\_PlugIn.h*. This is sufficient to include enough SuperCollider infrastructure for most types of UGen. (For phase vocoder UGens, more may be needed, as described later.)

The static `InterfaceTable` pointer is a reference to a table of SuperCollider functions such as the ones used to register a new UGen.

We define a data structure (a “struct”) which will hold any data we need to store during the operation of the UGen. This struct, which needs to be remembered or passed from 1 audio block to the next, must be stored here. Note that the struct inherits from the base struct `Unit`—this is necessary so that `scsynth` can correctly write information into the struct, such as the rate at which the unit is running.

We declare our UGen’s functions, using the extern “C” specifier so that the `scsynth` executable is able to reference the functions using C linkage. In a given plug-in we are allowed to define 1 or more UGens. Each of these will have 1 constructor (“Ctor”) function, 1 or more calculation (“next”) functions, and optionally 1 destructor (“Dtor”) function.

Our constructor function, `Flanger_Ctor()`, takes a pointer to a Flanger struct and must prepare the UGen for execution. It must do the following 3 things:

1. Initialize the Flanger struct’s member variables appropriately. In this case we initialize the `delaysize` member to a value representing a 20-millisecond maximum delay, making use of the `SAMPLERATE` macro which the SuperCollider API provides to specify the sample rate for the UGen. For some of the other struct members, we wish to calculate the values based on an input to the UGen. We can do this using the `IN0()` macro, which grabs a single control-rate value from the specified input. Here, we use `IN0(1)`—remembering that numbering starts at 0, this corresponds to the second input, defined in the Flanger class file as “rate.” These macros (and others) will be discussed later.
2. Tell `scsynth` what the calculation function will be for this instance of the UGen. The `SETCALC` macro stores a reference to the function in our unit’s struct. In our example there’s only 1 choice, so we simply call `SETCALC(Flanger_next)`. It’s possible

```

#include "SC_PlugIn.h"

static InterfaceTable *ft;

// the struct will hold data which we want to "pass" from one function to another
// e.g. from the constructor to the calc func,
// or from one call of the calc func to the next
struct Flanger : public Unit {
    float rate, delaysize, fwdhop, readpos;
    int writepos;
};

// function declarations, exposed to C
extern "C" {
    void load(InterfaceTable *inTable);
    void Flanger_Ctor(Flanger *unit);
    void Flanger_next(Flanger *unit, int inNumSamples);
}

void Flanger_Ctor( Flanger *unit ) {

    // Here we must initialise state variables in the Flanger struct.
    unit->delaysize = SAMPLERATE * 0.02f; // Fixed 20ms max delay
    // Typically with reference to control-rate/scalar-rate inputs.
    float rate = IN0(1);
    // Rather than using rate directly, we're going to calculate the size of
    // jumps we must make each time to scan through the delayline at "rate"
    float delta = (unit->delaysize * rate) / SAMPLERATE;
    unit->fwdhop = delta + 1.0f;
    unit->rate = rate;

    // IMPORTANT: This tells scsynth the name of the calculation function
    // for this UGen.
    SETCALC(Flanger_next);

    // Should also calc 1 sample's worth of output -
    //ensures each ugen's "pipes" are "primed"
    Flanger_next(unit, 1);
}

```

**Figure 25.1**

C++ code for a Flanger UGen. This code doesn't add any effect to the sound yet, but contains the key elements required for all UGens.

```
void Flanger_next( Flanger *unit, int inNumSamples ) {

    float *in = IN(0);
    float *out = OUT(0);

    float depth = IN0(2);

    float rate    = unit->rate;
    float fwdhop  = unit->fwdhop;
    float readpos = unit->readpos;
    int  writepos = unit->writepos;
    int  delaysize = unit->delaysize;

    float val, delayed;

    for ( int i=0; i<inNumSamples; ++i) {
        val = in[i];

        // Do something to the signal before outputting
        // (not yet done)

        out[i] = val;
    }

    unit->writepos = writepos;
    unit->readpos  = readpos;
}

void load(InterfaceTable *inTable) {

    ft = inTable;

    DefineSimpleUnit(Flanger);
}
```

Figure 25.1  
(continued)

to define multiple calculation functions and allow the constructor to decide which one to use. This is covered later.

3. Calculate one sample’s worth of output, typically by calling the unit’s calculation function and asking it to process 1 sample. The purpose of this is to “prime” the inputs and outputs of all the unit generators in the graph and to ensure that the constructors for UGens farther down the chain have their input values available so they can initialize correctly.

Our calculation function, `Flanger_next()`, should perform the main audio processing. In this example it doesn’t actually alter the sound — we’ll get to that shortly — but it illustrates some important features of calculation functions. It takes 2 arguments passed in by the server: a pointer to the struct and an integer specifying how many values are to be processed (this will be 1 for control-rate, more for audio-rate — typically 64).

The last thing in our C++ file is the `load()` function, called when the `scsynth` executable boots up.

We store the reference to the interface table which is passed in — note that although you don’t see any explicit references to `ft` elsewhere in the code, that’s because they are hidden behind macros which make use of it to call functions in the server.

We must also declare to the server each of the UGens which our plug-in defines. This is done using a macro `DefineSimpleUnit(Flanger)`, which tells the server to register a UGen with the name *Flanger* and with a constructor function named *Flanger\_Ctor*. It also tells the server that no destructor function is needed. If we did require a destructor, we would instead use `DefineDtorUnit(Flanger)`, which tells the server that we’ve also supplied a destructor function named *Flanger\_Dtor*. You must name your constructor/destructor functions in this way, since the naming convention is hard-coded into the macros.

So what is happening inside our calculation function? Although in our example the input doesn’t actually get altered before being output, the basic pattern for a typical calculation function is given. We do the following:

Create pointers to the input and output arrays which we will access: `float *in = IN(0); float *out = OUT(0);` The macros `IN()` and `OUT()` return appropriate pointers for the desired inputs/outputs — in this case the first input and the first output. If the input is audio-rate, then `in[0]` will refer to the first incoming sample, `in[1]` to the next incoming sample, and so on. If the input is control-rate, then there is only 1 incoming value, `in[0]`.

We use the macro `IN0()` again to grab a single control-rate value, here the “depth” input. Note that `IN0()` is actually a shortcut to the first value in the location referenced by `IN()`. `IN0(1)` is exactly the same as `IN(1)[0]`.

We copy some values from the UGen’s struct into local variables. This can improve the efficiency of the unit, since the C++ optimizer will typically cause the values to be loaded into registers.

Next we loop over the number of input frames, each time taking an input value, processing it, and producing an output value. We could take values from multiple inputs, and even produce multiple outputs, but in this example we’re using only 1 full-rate input and producing a single output. Two important notes:

If an input/output is control-rate and you mistakenly treat it as audio-rate, you will be reading/writing memory you should not be, and this can cause bizarre problems and crashes; essentially this is just the classic C/C++ “gotcha” of accidentally treating an array as being bigger than it really is. Note that in our example, we assume that the input and output are of the same size, although it’s possible that they aren’t—some UGens can take audio-rate input and produce control-rate output. This is why it is useful to make sure your SuperCollider class code includes the rate-checking code described earlier in this chapter. You can see why the `checkSameRateAsFirstInput` approach is useful in this case.

The server uses a “buffer coloring” algorithm to minimize use of buffers and to optimize cache performance. This means that any of the output buffers may be the same as 1 of the input buffers. This allows for in-place operation, which is very efficient. You must be careful, however, not to write any output sample before you have read the corresponding input sample. If you break this rule, then the input may be overwritten with output, leading to undesired behavior. If you can’t write the UGen efficiently without breaking this rule, then you can instruct the server not to alias the buffers by using the `DefineSimpleCantAliasUnit()` or `DefinedtorCantAliasUnit()` macros in the `load()` function, rather than the `DefineSimpleUnit()` or `DefinedtorUnit()` macros. (The Help file on writing UGens provides an example in which this ordering is important.)

Finally, having produced our output, we may have modified some of the variables we loaded from the struct; we need to store them back to the struct so the updated values are used next time. Here we store the rate value back to the struct—although we don’t modify it in this example, we will shortly change the code so that this may happen.

The code in figure 25.1 should compile correctly into a plug-in. With the class file in place and the plug-in compiled, you can now use the UGen in a synth graph:

```
s.boot
(
x = {
  var son, dly, out;
  son = Saw.ar([100, 150, 200]).mean;
  out = Flanger.ar(son);
```

```

        out.dup * 0.2;
    }.play(s);
}

```

Remember that Flanger doesn't currently add any effect to the sound. But we can at least check that it runs correctly (outputting its input unmodified and undistorted) before we start to make things interesting.

#### 25.5.4 Summary: The Three Main Rates of Data Output

Our example has taken input in 3 different ways:

Using `IN0()` in the constructor to take an input value and store it to the struct for later use. Since this reads a value only once, the input is being treated as a *scalar-rate* input.

Using `IN0()` in the calculation function to take a single input value. This treats the input as *control-rate*.

Using `IN()` in the calculation function to get a pointer to the whole array of inputs. This treats the input as *audio-rate*. Typically the size of such an input array is accessed using the `inNumSamples` argument, but note that if you create a control-rate UGen with audio-rate inputs, then `inNumSamples` will be wrong (it will be 1), so you should instead use the macro `FULLBUFLLENGTH` (see table 25.2).

If the data that one of your UGen's inputs is fed is actually audio-rate, there is no danger in treating it as control-rate or scalar-rate. The end result is to ignore the "extra" data provided to your UGen. Similarly, a control-rate input can safely be treated as scalar-rate. The result would be crude downsampling without low-pass filtering, which may be undesirable but will not crash the server.

#### 25.5.5 Allocating Memory and Using a Destructor

Next we can develop our Flanger example so that it applies an effect to the sound. In order to create a flanging effect, we need a short delay line (around 20 milliseconds). We vary the amount of delay and mix the delayed sound with the input to produce the effect.

To create a delay line, we need to allocate some memory and store a reference to that memory in the UGen's data structure. And, of course, we need to free this memory when the UGen is freed. This requires a UGen with a destructor. Figure 25.2 shows the full code, with the destructor added, as well as the code to allocate, free, and use the memory. Note the change in the `load()` function—we use `DefinedtorUnit()` rather than `DefineSimpleUnit()`. (We've also added code to the calculation function which reads and writes to the delay line, creating the flanging effect.)

```

#include "SC_PlugIn.h"

static InterfaceTable *ft;

// the struct will hold data which we want to "pass" from one function to another
// e.g. from the constructor to the calc func,
// or from one call of the calc func to the next
struct Flanger : public Unit {
    float rate, delaysize, fwdhop, readpos;
    int writepos;

    // a pointer to the memory we'll use for our internal delay
    float *delayline;
};

// function declarations, exposed to C
extern "C" {
    void load(InterfaceTable *inTable);
    void Flanger_Ctor(Flanger *unit);
    void Flanger_next(Flanger *unit, int inNumSamples);
    void Flanger_Dtor(Flanger *unit);
}

void Flanger_Ctor( Flanger *unit ) {

    // Here we must initialise state variables in the Flanger struct.
    unit->delaysize = SAMPLERATE * 0.02f; // Fixed 20ms max delay
    // Typically with reference to control-rate/scalar-rate inputs.
    float rate = IN0(1);
    // Rather than using rate directly, we're going to calculate the size of
    // jumps we must make each time to scan through the delayline at "rate"
    float delta = (unit->delaysize * rate) / SAMPLERATE;
    unit->fwdhop = delta + 1.0f;
    unit->rate = rate;
    unit->writepos = 0;
    unit->readpos = 0;

    // Allocate the delay line
    unit->delayline = (float*)RTAAlloc(unit->mWorld, unit->delaysize *
sizeof(float));
    // Initialise it to zeroes

```

Figure 25.2  
Completed C++ code for the Flanger UGen.

```

memset(unit->delayline, 0, unit->delaysize * sizeof(float));

// IMPORTANT: This tells scsynth the name of the calculation function
//for this UGen.
SETCALC(Flanger_next);

// Should also calc 1 sample's worth of output -
//ensures each ugen's "pipes" are "primed"
Flanger_next(unit, 1);
}

void Flanger_next( Flanger *unit, int inNumSamples ) {

    float *in = IN(0);
    float *out = OUT(0);

    float depth = IN0(2);

    float rate    = unit->rate;
    float fwdhop  = unit->fwdhop;
    float readpos = unit->readpos;
    float *delayline = unit->delayline;
    int writepos  = unit->writepos;
    int delaysize = unit->delaysize;

    float val, delayed, currate;

    currate = IN0(1);

    if(rate != currate){
        // rate input needs updating
        rate = currate;
        fwdhop = ((delaysize * rate * 2) / SAMPLERATE) + 1.0f;
    }

    for ( int i=0; i<inNumSamples; ++i) {
        val = in[i];

        // Write to the delay line
        delayline[writepos++] = val;
        if(writepos==delaysize)
            writepos = 0;
    }
}

```

Figure 25.2  
(continued)

```
    // Read from the delay line
    delayed = delayline[(int)readpos];
    readpos += fwdhop;
    // Update position, NB we may be moving forwards or backwards
    //(depending on input)
    while((int)readpos >= delaysize)
        readpos -= delaysize;
    while((int)readpos < 0)
        readpos += delaysize;

    // Mix dry and wet together, and output them
    out[i] = val + (delayed * depth);
}

unit->rate = rate;
unit->fwdhop = fwdhop;
unit->writepos = writepos;
unit->readpos = readpos;
}

void Flanger_Dtor( Flanger *unit ) {
    RTFree(unit->mWorld, unit->delayline);
}

void load(InterfaceTable *inTable) {

    ft = inTable;

    DefineDtorUnit(Flanger);
}
```

Figure 25.2  
(continued)

**Table 25.1**  
Memory Allocation and Freeing

Typical C Allocation/Freeing	In SuperCollider (using the real-time pool)
<code>void *ptr = malloc(numbytes)</code> <code>free(ptr)</code>	<code>void *ptr = RTAlloc(unit-&gt;mWorld, numbytes)</code> <code>RTFree(unit-&gt;mWorld, ptr)</code>

SuperCollider UGens allocate memory differently from most programs. Ordinary memory allocation and freeing can be a relatively expensive operation, so SuperCollider provides a *real-time pool* of memory from which UGens can borrow chunks in an efficient manner. The functions to use in a plug-in are in the right-hand column of table 25.1, and the analogous functions (the ones to avoid) are shown in the left-hand column.

`RTAlloc` and `RTFree` can be called anywhere in your constructor/calculation/destructor functions. Often you will `RTAlloc` the memory during the constructor and `RTFree` it during the destructor, as is done in figure 25.2.

Memory allocated in this way is taken from the (limited) real-time pool and is not accessible outside the UGen (e.g., to client-side processes). If you require large amounts of memory or wish to access the data from the client, you may prefer to use a buffer allocated and then passed in from outside—this is described later.

### 25.5.6 Providing More Than 1 Calculation Function

Your UGen’s choice of calculation function is specified within the constructor rather than being fixed. This gives an opportunity to provide different functions optimized for different situations (e.g., 1 for control-rate and 1 for audio-rate input) and to decide which to use. This code, used in the constructor, would choose between 2 calculation functions according to whether the first input was audio-rate or not:

```
if (INRATE(0) == calc_FullRate) {
    SETCALC(Flanger_next_a);
} else {
    SETCALC(Flanger_next_k);
}
```

You would then provide both a `Flanger_next_a()` and a `Flanger_next_k()` function.

Similarly, you could specify different calculation functions for audio-rate versus control-rate *output* (e.g., by testing whether `BULENGTH` is 1; see table 25.2), although this is often catered for automatically when your calculation function uses the `inNumSamples` argument to control the number of loops performed, and so on.

**Table 25.2**  
Useful Macros for UGen Writers

Macro	Description
IN(index)	A float* pointer to input number <i>index</i>
OUT(index)	A float* pointer to output number <i>index</i>
INO(index)	A single (control-rate) value from input number <i>index</i>
OUT0(index)	A single (control-rate) value at output number <i>index</i>
INRATE(index)	The rate of input <i>index</i> , an integer value corresponding to 1 of the following constants: calc_ScalarRate (scalar-rate) calc_BufRate (control-rate) calc_FullRate (audio-rate) calc_DemandRate (demand-rate)
SETCALC(func)	Set the calculation function to <i>func</i>
SAMPLERATE	The sample rate of the UGen as a double. Note: for control-rate UGens this is not the full audio rate but audio rate/blocksize)
SAMPLEDUR	Reciprocal of SAMPLERATE (seconds per sample)
BUFLENGTH	Equal to the block size if the unit is audio rate and to 1 if the unit is control rate
BUFRATE	The control rate as a double
BUFDUR	The reciprocal of BUFRATE
GETBUF	Treats the UGen's first input as a reference to a buffer; looks this buffer up in the server, and provides variables for accessing it, including float* bufData, which points to the data; uint32 bufFrames for how many frames the buffer contains; uint32 bufChannels for the number of channels in the buffer
ClearUnitOutputs(unit, inNumSamples)	A function which sets all the unit's outputs to 0
Print(fmt, ...)	Print text to the SuperCollider post window; arguments are just like those for the C function printf
DoneAction(doneAction, unit)	Perform a "doneAction," as used in EnvGen, DetectSilence, and others
RTAlloc(world, numBytes)	Allocate memory from the real-time pool— analogous to malloc(numBytes)
RTRealloc(world, pointer, numBytes)	Reallocate memory in the real-time pool— analogous to realloc(pointer, numBytes)

Table 25.2  
(continued)

Macro	Description
RTFree(world, pointer)	Free allocated memory back to the real-time pool— analogous to free(pointer)
SendTrigger(node, triggerID, value)	Send a trigger from the node to clients, with integer ID, <i>triggered</i> , and float value <i>value</i>
FULLRATE	The full audio sample rate of the server (irrespective of the rate of the UGen) as a double
FULLBULENGTH	The integer number of samples in an audio-rate input (irrespective of the rate of the UGen)

The unit's calculation function can also be changed during execution—the SETCALC() macro can safely be called from a calculation function, not just from the constructor. Whenever you call SETCALC(), this changes which function the server will call, from the next control period onward.

The Help file on writing UGens shows more examples of SETCALC() in use.

### 25.5.7 Trigger Inputs

Many UGens make use of trigger inputs. The convention here is that if the input is nonpositive (i.e., 0 or negative), then crosses to any positive value, a trigger has occurred. If you wish to provide trigger inputs, use this same convention.

The change from nonpositive to positive requires checking the trigger input's value against its previous value. This means that our struct will need a member to store the previous value for checking. Assuming that our struct contains a float member *prevtrig*, the following sketch outlines how we handle the incoming data in our calculation function:

```
float trig = IN0(3); // Or whichever input you wish
float prevtrig = unit->prevtrig;
if(prevtrig<=0 && trig >0){
    // ... do something ...
}
unit->prevtrig = trig; // Store current value—next time it'll be the
"previous" value
```

The sketch is for a control-rate trigger input, but a similar approach is used for audio-rate triggering, too. For audio-rate triggering, you need to compare each value in the input block against the value immediately previous. Note that for the very first value in the block, you need to compare against the last value from the *previous* block (which you must have stored).

For complete code examples, look at the source of the Trig1 UGen, found in *TriggerUGens.cpp* in the main SC distribution.

### 25.5.8 Accessing a Buffer

When a buffer is allocated and then passed in to a UGen, the UGen receives the index number of that buffer as a float value. In order to get a pointer to the correct chunk of memory (as well as the size of that chunk), the UGen must look it up in the server's list of buffers.

In practice this is most easily achieved by using a macro called GET\_BUF. You can call GET\_BUF near the top of your calculation function, and then the data are available via a float pointer \*bufData along with 2 integers defining the size of the buffer, bufChannels and bufFrames. Note that the macro assumes the buffer index is the *first* input to the UGen (this is the case for most buffer-using UGens).

For examples which use this approach, look at the code for the DiskIn or DiskOut UGens, defined in *DiskIO\_UGens.cpp* in the main SC distribution.

Your UGen does not need to free the memory associated with a buffer once it ends. The memory is managed externally by the buffer allocation/freeing server commands.

### 25.5.9 Randomness

The API provides a convenient interface for accessing good-quality pseudo-random numbers. The randomness API is specified in *SC\_RGen.h* and provides functions for random numbers from standard types of distribution: uniform, exponential, bilinear, and quasi-Gaussian (such as sum3rand, also available client-side). The server creates an instance of the random number generator for UGens to access. The following excerpt shows how to generate random numbers for use in your code:

```
RGen & rgen = *unit->mParent->mRGen;
float rfl = rgen.frand(); // A random float, uniformly distributed, 0.0 to
1.0
int rval2 = rgen.irand(56); // A random integer, uniformly distributed, 0
to 55 inclusive
float rgaus = rgen.sum3rand(3.5); // Quasi-Gaussian, limited to range ±3.5
```

### 25.5.10 When Your UGen Has No More to Do

Many UGens carry on indefinitely, but often a UGen reaches the end of its useful “life” (e.g., it finishes outputting an envelope or playing a buffer). There are 3 specific behaviors that might be appropriate if your UGen does reach a natural end:

1. Some UGens set a “done” flag to indicate that they’ve finished. Other UGens can monitor this and act in response to it (e.g., `Done`, `FreeSelfWhenDone`). See the Help files for examples of these UGens. If you wish your UGen to indicate that it has finished, set the flag as follows:

```
unit->mDone = true;
```

This doesn’t affect how the server treats the UGen—the calculation function will still be called in future.

2. UGens such as `EnvGen`, `LineIn`, `Duty`, and `Line` provide a “doneAction” feature which can perform actions such as freeing the node once the UGen has reached the end of its functionality. You can implement this yourself simply by calling the `DoneAction()` macro, which performs the desired action. You would typically allow the user to specify the doneAction as an input to the unit. For example, if the doneAction is the sixth input to your UGen, you would call

```
DoneAction(IN0(5), unit)
```

Since this can perform behaviors such as freeing the node, many UGens stop calculating/outputting after they reach the point of calling this macro. See, for example, the source code for `DetectSilence`, which sets its calculation function to a no-op `DetectSilence_done` function at the point where it calls `DoneAction`. Not all doneActions free the synth, though, so additional output is not always redundant.

3. If you wish to output zeroes from all outputs of your unit, you can simply call the `ClearUnitOutputs` function as follows:

```
ClearUnitOutputs(unit, inNumSamples);
```

Notice that this function has the same signature as a calculation function: as arguments it takes a pointer to the unit struct and an integer number of samples. You can take advantage of this similarity to provide an efficient way to stop producing output:

```
SETCALC(*ClearUnitOutputs);
```

Calling this would mean that your calculation function would not be called in future iterations. Instead, `ClearUnitOutputs` would be called. Therefore this provides an irreversible but efficient way for your UGen to produce silent output for the remainder of the synth’s execution.

### 25.5.11 Summary of Useful Macros

Table 25.2 summarized some of the most generally useful macros defined for use in your UGen code. Many of these are discussed in this chapter, but not all are covered explicitly. The macros are defined in `SC_Unit.h` and `SC_InterfaceTable.h`.

## 25.6 Specialized Types of UGen

### 25.6.1 Multiple-Output UGens

In the C++ code, writing UGens which produce multiple outputs is very straightforward. The `OUT()` macro gets a pointer to the desired-numbered output. Thus, for a 3-output UGen, assign each one (`OUT(0)`, `OUT(1)`, `OUT(2)`) to a variable, then write output to these 3 pointers.

In the SuperCollider class code, the default is to assume a single output, and we need to modify this behavior. Let's look at the `Pitch` UGen to see how it's done:

```
Pitch : MultiOutUGen {
    *kr {arg in = 0.0, initFreq = 440.0, minFreq = 60.0, maxFreq = 4000.0,
        execFreq = 100.0, maxBinsPerOctave = 16, median = 1,
        ampThreshold = 0.01, peakThreshold = 0.5, downSample = 1;
    ^this.multiNew('control', in, initFreq, minFreq, maxFreq, execFreq,
        maxBinsPerOctave, median, ampThreshold, peakThreshold, downSample)
    }
    init {arg ... theInputs;
        inputs = theInputs;
        ^this.initOutputs(2, rate);
    }
}
```

There are 2 differences from an ordinary UGen. First, `Pitch` is a subclass of `MultiOutUGen` rather than of `UGen`; `MultiOutUGen` takes care of some of the changes needed to work with a UGen with multiple outputs. Second, the `init` function is overridden to say exactly how many outputs this UGen will provide (in this case, 2).

For `Pitch`, the number of outputs is fixed, but in some cases it might depend on other factors. `PlayBuf` is a good example of this: its number of outputs depends on the number of channels in the buffer(s) it is expecting to play, specified using the `numChannels` argument. The `init` method for `PlayBuf` takes the `numChannels` input (i.e., the first value from the list of inputs passed to `init`) and specifies that as the number of outputs.

### 25.6.2 Passing Arrays into UGens

#### 25.6.2.1 The class file

As described earlier, the `multiNew` method automatically performs multichannel expansion if any of the inputs are arrays—yet in some cases we want a single unit to handle a whole array, rather than having 1 unit per array element. The `BufWr` and `RecordBuf` UGens are good examples of UGens that do exactly this: each UGen can

take an array of inputs and write them to a multichannel buffer. Here's how the class file handles this:

```
RecordBuf : UGen {
    *ar {arg inputArray, bufnum = 0, offset = 0.0, recLevel = 1.0,
        preLevel = 0.0, run = 1.0, loop = 1.0, trigger = 1.0;
        ^this.multiNewList(['audio', bufnum, offset, recLevel, preLevel,
            run, loop, trigger] ++ inputArray.asArray);
    }
}
```

Instead of calling the UGen method `multiNew`, we call `multiNewList`, which is the same except that all the arguments are a single array rather than a separated argument list. This means that the `inputArray` argument (which could be either a single unit or an array), when concatenated onto the end of the argument list using the `++` array concatenation operator, in essence appears as a set of *separate* input arguments rather than a single array argument.

Note that `RecordBuf` doesn't know in advance what size the input array is going to be. Because of the array flattening that we perform, this means that the `RecordBuf` C++ plug-in receives a *variable number of inputs* each time it is instantiated. Our plug-in code will be able to detect how many inputs it receives in a given instance.

Why do we put `inputArray` at the *end* of the argument list? Why not at the beginning, in parallel with how a user invokes the `RecordBuf` UGen? The reason is to make things simpler for the C++ code, which will access the plug-in inputs according to their numerical position in the list. The `recLevel` input, for example, is always the third input, whereas if we inserted `inputArray` into the list before it, its position would depend on the size of `inputArray`.

The `Poll` UGen uses a very similar procedure, converting a string of text into an array of ASCII characters and appending them to the end of its argument list. However, the `Poll` class code must perform some other manipulations, so it is perhaps less clear as a code example than `RecordBuf`. But if you are developing a UGen that needs to pass text data to the plug-in, `Poll` shows how to do it using this array approach.

### 25.6.2.2 The C++ code

Ordinarily we access input data using the `IN()` or `IN0()` macro, specifying the number of the input we want to access. Arrays are passed into the UGen as a separate numeric input for each array element, so we access these elements in exactly the same way. But we need to know how many items to expect, since the array can be of variable size.

The `Unit` struct can tell us how many inputs in total are being provided (the member `unit->mNumInputs`). Look again at the `RecordBuf` class code given above. There

are 7 “ordinary” inputs, plus the array appended to the end. Thus the number of channels in our input array is (`unit->mNumInputs - 7`). We use this information to iterate over the correct number of inputs and process each element.

### 25.6.3 Demand-Rate UGens

#### 25.6.3.1 The class file

Writing the class file for a demand-rate UGen is straightforward. Look at the code for units such as `Dseries`, `Dgeom`, or `Dwhite` as examples. They differ from other UGen class files in 2 ways:

1. The first argument to `multiNew` (or `multiNewList`) is 'demand'.
2. They implement a single class method, `*new`, rather than `*ar/*kr/*ir`. This is because although some UGens may be able to run at multiple rates (e.g., audio rate or control rate), a demand-rate UGen can run at only 1 rate: the rate at which data are demanded of it.

#### 25.6.3.2 The C++ code

The C++ code for a demand-rate UGen works as normal, with the constructor specifying the calculation function. However, the calculation function behaves slightly differently.

First, it is not called regularly (once per control period) but only when demanded, which during a particular control period could be more than once or not at all. This means that you can't make assumptions about regular timing, such as the assumptions made in an oscillator which increments its phase by a set amount each time it is called.

Second, rather than being invoked directly by the server, the calculation function calls are actually passed up the chain of demand-rate generators. Rather than using the `IN()` or `IN0()` macros to access an input value (whose generation will have been coordinated by the server), we instead use the `DEMANDINPUT()` macro, which requests a new value directly from the unit farther up the chain, “on demand.”

Note: because of the method used to demand the data, demand-rate UGens are currently restricted to being single-output.

### 25.6.4 Phase Vocoder UGens

Phase vocoder UGens operate on frequency-domain data stored in a buffer (produced by the `FFT` UGen). They don't operate at a special “rate” of their own: in reality they are control-rate UGens. They produce and consume a control-rate signal which acts as a type of trigger: when an FFT frame is ready for processing, its value

is the appropriate buffer index; otherwise, its value is  $-1$ . This signal is often referred to as the “chain” in SC documentation.

#### 25.6.4.1 The class file

As with demand-rate UGens, phase vocoder UGens (PV UGens) can have only a single rate of operation: the rate at which FFT frames are arriving. Therefore, PV UGens implement only a single `*new` class method, and they specify their rate as “control” in the call to `multiNew`. See the class files for `PV_MagMul` and `PV_BrickWall` as examples of this.

PV UGens process data stored in buffers, and the C++ API provides some useful macros to help with this. The macros assume that the *first* input to the UGen is the one carrying the FFT chain where data will be read and then written, so it is sensible to stick with this convention.

#### 25.6.4.2 The C++ code

PV UGens are structured just like any other UGen, except that to access the frequency-domain data held in the external buffer, there are certain macros and procedures to use. Any of the core UGens implemented in `PV_UGens.cpp` should serve as a good example to base your own UGens on. Your code should include the header file `FFT_UGens.h`, which defines some PV-specific structs and macros.

Two important macros are `PV_GET_BUF` and `PV_GET_BUF2`, one of which you use at the beginning of your calculation function to obtain the FFT data from the buffer. These macros implement the special PV UGen behavior: if the FFT chain has “fired,” then they access the buffer(s) and continue with the rest of the calculation function; but if the FFT chain has not “fired” in the current control block, then they output a value of  $-1$  and *return* (i.e., they do not allow the rest of the calculation function to proceed). This has the important consequence that although your calculation function code will look “as if” it is called once per control block, in fact your code will be executed only at the FFT frame rate.

`PV_GET_BUF` will take the FFT chain indicated by the *first* input to the UGen and create a pointer to these data called `*buf`.

`PV_GET_BUF2` is for use in UGens which process 2 FFT chains and write the result back out to the first chain: it takes the FFT chain indicated by the *first* and *second* inputs to the UGen and creates pointers to the data called `*buf1` and `*buf2`.

It should be clear that you use `PV_GET_BUF` or `PV_GET_BUF2`, but not both.

Having acquired a pointer to the data, you will of course wish to read/write that data. Before doing so, you must decide whether to process the complex-valued data as polar coordinates or Cartesian coordinates. The data in the buffer may be in

either format (depending on what has happened to it so far). To access the data as Cartesian values you use

```
SCComplexBuf *p = ToComplexApx(buf);
```

and to access the data as polar values you use

```
SCPolarBuf *p = ToPolarApx(buf);
```

These 2 data structures, and the 2 functions for obtaining them, are declared in *FFT\_UGens.h*. The name *p* for the pointer is of course arbitrary, but it's what we'll use here.

FFT data consist of a complex value for each frequency bin, with the number of bins related to the number of samples in the input. But in the SuperCollider context the input is real-valued data, which means that (a) the bins above the Nyquist frequency (which is half the sampling frequency) are a mirror image of the bins below, and can therefore be neglected; and (b) phase is irrelevant for the DC and Nyquist frequency bins, so these 2 bins can be represented by a single-magnitude value rather than a complex value.

The end result of this is that we obtain a data structure containing a single DC value, a single Nyquist value, and a series of complex values for all the bins in between. The number of bins in between is given by the value *numbins*, which is provided for us by *PV\_GET\_BUF* or *PV\_GET\_BUF2*. The data in a Cartesian-type struct (an *SCComplexBuf*) are of the form

```
p->dc
p->bin[0].real
p->bin[0].imag
p->bin[1].real
p->bin[1].imag
...
p->bin[numbins - 1].real
p->bin[numbins - 1].imag
p->nyq
```

The data in a polar-type struct (an *SCPolarBuf*) is of the form

```
p->dc
p->bin[0].mag
p->bin[0].phase
p->bin[1].mag
p->bin[1].phase
...
p->bin[numbins - 1].mag
p->bin[numbins - 1].phase
p->nyq
```

Note that the indexing is slightly strange: engineers commonly refer to the DC component as the “first” bin in the frequency-domain data. However in these structs, because the DC component is represented differently, `bin[0]` is actually the first non-DC bin—what would sometimes be referred to as the second bin. Similarly, keep in mind that `numbins` represents the number of bins *not including* the DC or Nyquist bins.

To perform a phase vocoder manipulation, simply read and write to the struct (which actually is directly in the external buffer). The buffer will then be passed down the chain to the next phase vocoder UGen. You don’t need to do anything extra to “output” the frequency-domain data.

When compiling your PV UGen, you will need to compile/link against *SCComplex.cpp* from the main SuperCollider source, which provides the implementation of these frequency-domain data manipulations.

## 25.7 Practicalities

### 25.7.1 Debugging

Standard C++ debugging procedures can be used when developing UGens. The simplest method is to add a line into your code which prints out values of variables—you can use the standard C++ `printf()` method, which in a UGen will print text to the post window.

For more power, you can launch the server process, then attach a debugger such as *gdb* (the GNU debugger) or Xcode’s debugger (which is actually *gdb* with a graphical interface) to perform tasks such as pausing the process and inspecting values of variables. On Mac, if you use the debugger to launch *SuperCollider.app*, remember that the local server runs in a process different from the application. You can either launch the application using the debugger and booting the internal server, or you can launch just the server (*scsynth*) using the debugger, which then runs as a local server. In the latter case you need to ensure your debugger launches *scsynth* with the correct arguments (e.g., `-u 57110`).

When debugging a UGen that causes server crashes, you may wish to look at your system’s crash log for *scsynth*. The most common cause of crashes is introduced when using `RTAlloc` and `RTFree`—if you try to `RTFree` something that has not yet been `RTAlloc`’ed, or otherwise is not a pointer to the real-time memory pool, this can cause bad-access exceptions to appear in the crash log. If the crash log seems to reveal that your UGen is somehow causing crashes inside core UGens which normally behave perfectly, then check that your code does not write data outside of the expected limits: make sure you `RTAlloc` the right amount of space for what you’re

doing (for example, with arrays, check exactly which indices your code attempts to access).

### 25.7.2 Optimization

Optimizing code is a vast topic and often depends on the specifics of the code in question. However, we can suggest some optimization tips for writing SuperCollider UGens. The efficiency/speed of execution is usually the number-one priority, especially since a user may wish to employ many instances of the UGen simultaneously. The difference between a UGen that takes 2.5% and another that takes 1.5% CPU may seem small, but the first limits you to 40 simultaneous instances, while the second will allow up to 66; a 65% increase. Imagine doing your next live performance on a 4-year-old processor — that’s essentially the effect of the less efficient code.

*Avoid calls to “expensive” procedures* whenever possible. For example, *floating-point division* is typically much more expensive than multiplication, so if your unit must divide values by some constant value which is stored in your struct, rewrite this so that the *reciprocal* of that value is stored in the struct and you can perform a multiplication rather than a division. If you want to find an integer power of 2, use bit shifting ( $1 \ll n$ ) rather than the expensive math function ( $\text{pow}(2, n)$ ). Other expensive floating-point operations are *square-root* finding and *trigonometric* operations (*sin*, *cos*, *tan*, etc.). *Precalculate* and store such values wherever possible, rather than calculating them afresh every time the calculation function is called.

As a typical example, often a filter UGen will take a user parameter (such as cutoff frequency) and use it to derive internal filter coefficients. If you store the previous value of the user parameter and use this to check whether it has changed at all — updating the coefficients only upon a change — you can improve efficiency, since often UGens are used with fixed or rarely changing parameters.

One of the most important SuperCollider-specific choices is, for reading a certain input or even performing a given calculation, whether to do this at *scalar/control/audio rate*. It can be helpful to allow any and all values to be updated at audio rate, but if you find that a certain update procedure is expensive and won’t usually be required to run at audio rate, it may be preferable to update only once during a calculation function.

Creating *multiple calculation functions*, each appropriate to a certain context (e.g., to a certain combination of input rates, as demonstrated earlier), and choosing the most appropriate, can allow a lot of optimization. For example, a purely control-rate calculation can avoid the looping required for audio-rate calculation and typically produces a much simpler calculation as a result. There is a maintenance overhead in providing these alternatives, but the efficiency gains can be large. In this

tension between efficiency and code comprehensibility/reusability, you should remember the importance of adding comments to your code to clarify the flow and the design decisions you have made.

In your calculation function, store values from your struct as well as input/output pointers/values as *local variables*, especially if referring to them multiple times. This avoids the overhead of indirection and can be optimized (by the compiler) to use registers better.

*Avoid DefineSimpleCantAliasUnit and DefineDtorCantAliasUnit.* As described earlier, DefineSimpleCantAliasUnit is available as an alternative to DefineSimpleUnit in cases where your UGen must write output before it has read from the inputs, but this can decrease cache performance.

*Avoid peaky CPU usage.* A calculation function that does nothing for the first 99 times it's called, then performs a mass of calculations on the 100th call, could cause *audio dropouts* if this spike is very large. To avoid this, “amortize” your unit's effort by spreading the calculation out, if possible, by precalculating some values which are going to be used in that big 100th call.

On Mac, Apple's vDSP library can improve speed by vectorizing certain calculations. If you make use of this, or other platform-specific libraries, remember the considerations of platform independence. For example, use preprocessor instructions to choose between the Mac-specific code and ordinary C++ code:

```
#if SC_DARWIN
// The Mac-specific version of the code (including, e.g., vDSP functions)
#else
// The generic version of the code
#endif
```

SC\_DARWIN is a preprocessor value set to 1 when compiling SuperCollider on Mac (this is set in the Xcode project settings). Branching like this introduces a maintenance overhead, because you need to make sure that you update both branches in parallel.

### 25.7.3 Distributing Your UGens

Sharing UGens with others contributes to the SuperCollider community and is a very cool thing to do. A SourceForge project, “sc3-plug-ins,” exists as a repository for downloadable UGen plug-ins produced by various people. You may wish to publish your work either there or separately.

Remember that SuperCollider is licensed under the well-known GPL (GNU Public License) open-source license, including the plug-in API. So if you wish to distribute your plug-ins to the world, they must also be GPL-licensed. (Note: you retain

copyright in any code you have written. You do not have to sign away your copyright in order to GPL-license a piece of code.) Practically, this has a couple of implications:

- You should include a copyright notice, a copy of the GPL license text, and the source code with your distributed plug-in.
- If your plug-in makes use of third-party libraries, those libraries must be available under a “GPL-compatible” copyright license. See the GNU GPL Web site for further discussion of what this means.

## 25.8 Conclusion

This chapter doesn’t offer an exhaustive list of all that’s possible, but it provides you with the core of what all UGen programmers need to know. If you want to delve deeper, you will find the online community to be a valuable resource for answers to questions not covered here; and the source code for existing UGens provides a wealth of useful code examples.

The open-source nature of SuperCollider makes for a vibrant online developer community. Whether you are tweaking 1 of SuperCollider’s core UGens or developing something very specialized, you’ll find the exchange of ideas with SuperCollider developers can be rewarding for your own projects as well as for others and can feed into the ongoing development of SuperCollider as a uniquely powerful and flexible synthesis system.