Notes on Implementing SF2/DLS Sound Synthesis

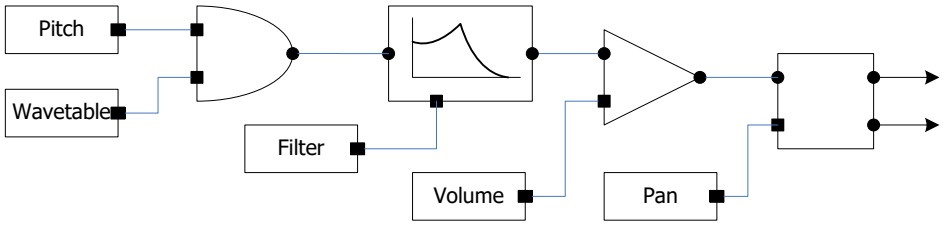
Daniel R. Mitchell

Sound Bank Instruments

SF2 and DLS share a common synthesis model. We will call this a *sound bank* instrument. A sound bank instrument uses a collection of wavetables and articulation data to define the parameters to the synthesizer.

Instrument Overview

The top-level structure of a sound bank instrument is shown below.



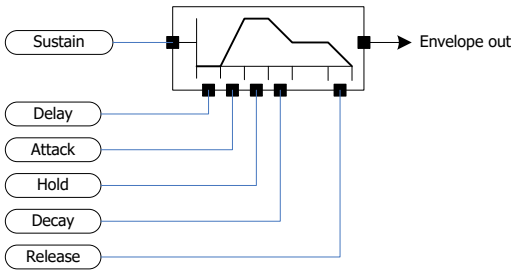
A wavetable oscillator is used to generate the audio signal, which is then passed through a low-pass filter, amplifier and pan control. This is a very generic synthesis structure and can potentially support multiple synthesis algorithms, including additive, subtractive, and sample playback. Typically, only recorded sound playback is used, implemented using a multi-period oscillator with separate transient and steady-state sections. However, a wavetable can be calculated using any of the methods shown in the chapter on complex waveforms, and then added to a sound bank.

Each synthesis parameter is a combination of initialization values, internal unit generators, and MIDI channel voice and controller values. Four unit generators are available as inputs to the synthesis parameters.

1. Volume EG
2. Modulation EG
3. Vibrato LFO
4. Modulation LFO

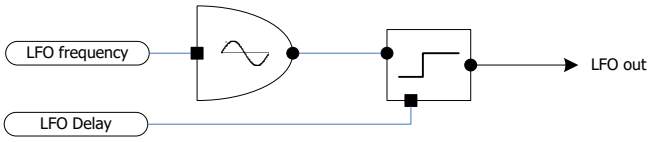
Note that DLS1 does not include the modulation LFO. Instead, the Vibrato LFO can be used as either a frequency control and/or an amplitude control, with the appropriate scaling unit for either frequency or amplitude.

Envelope generators have the form shown below.



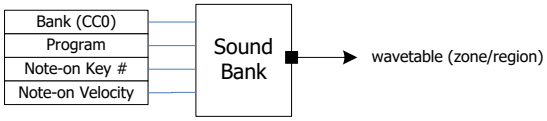
The EG is a constant-rate type with the peak output normalized to [0,1]. Sustain is specified as a percentage of peak output. Rates are specified in time cents (explained below). DLS1 does not include the delay or hold segments, using a typical ADSR instead, and the two values are set to 0.

LFO generators are sine wave oscillators, normalized to [-1,+1], with the output passed through a gate.



Wavetable Selection

Wavetable selection is made based on a combination of MIDI values.



A sample may span multiple key and/or velocity values, indicated by low-key, high-key, low-velocity, and high-velocity settings in the sound bank. The range of notes is called a *zone* or *region*. When a sound bank instrument contains multiple zones that match a key and velocity combination, the synthesis structure shown above must be replicated for each zone and the outputs summed to produce the final amplitude value.

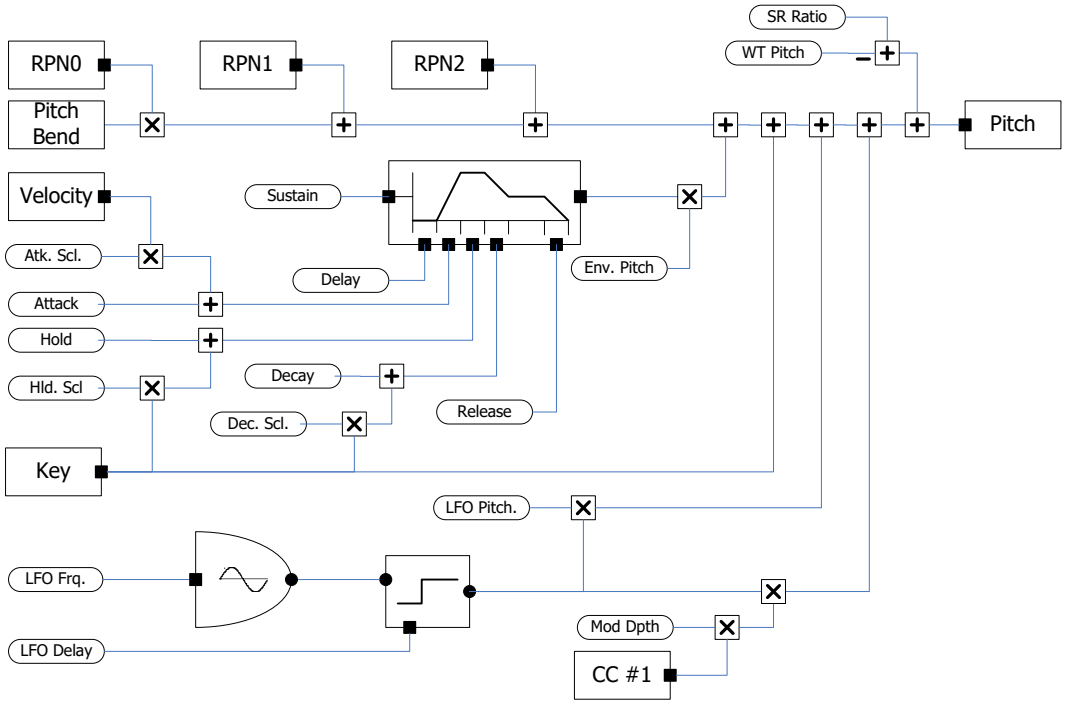
The wavetable may be looped or non-looped. For a non-looped table, the table is scanned once from beginning to end. For a looped table, the portion of the table prior to the loop start point is played through once, then the oscillator cycles over the samples between loop start and loop end.

The same sample data may be used for different zones, and may have different start and end loop points for each zone.

Stereo, or other multi-channel recordings, are produced by playing multiple zones with each zone panned appropriately. This allows for simulated stereo as well as a stereo recording. Note that this makes all wavetables single channel inherently, and is different from the interleaved samples found in WAV files. This is to be expected since single channel wavetables are required if the wavetable phase increment is to work properly.

Pitch Control

The Pitch parameter is set from the combination of values shown in the following diagram.

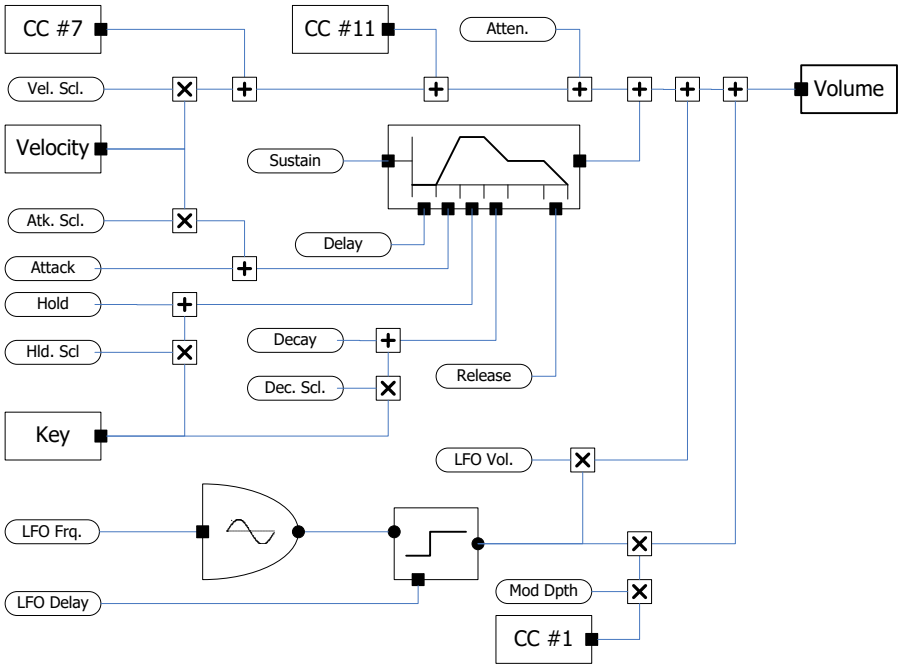


Parameters in labeled rectangles indicate MIDI source values. Parameters in ovals are taken from the sound bank file. The note-on key value is used to scale the envelope hold and decay rates. The note-on velocity is used to scale the envelope attack rate. The final pitch value is the sum of six source values:

1. Note on key
2. Pitch bend modified by pitch bend sensitivity (RPN0)
3. Coarse tuning (RPN2)
4. Fine tuning (RPN1)
5. Modulation envelope generator
6. Vibrato LFO
7. Vibrato LFO modified by modulation wheel (CC #1)
8. The calculations for sample rate ratio and wavetable pitch are explained in detail below.

Volume Control

The Volume parameter is set from the combination of values shown in the following diagram.

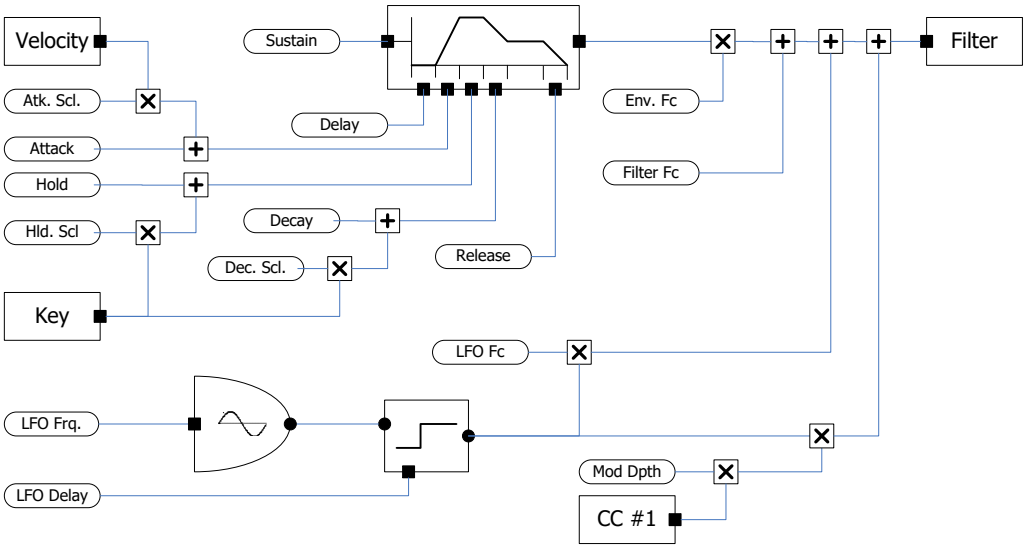


The final volume value is the sum of seven source values:

1. Initial Attenuation
2. Note on velocity
3. Channel volume (CC #7)
4. Expression controller (C #11)
5. Volume envelope generator
6. Volume LFO
7. Volume LFO modified by modulation wheel (CC #1)

Filter Control

The Filter parameter is set by the values shown below.



The final frequency is the sum of the source values:

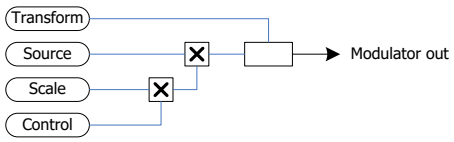
1. Initial Frequency
2. Modulation Envelope
3. Modulation LFO
4. Modulation LFO scaled by modulation wheel (CC#1)

Panning Control

Panning is set by the sum of the MIDI pan control and the instrument specific panning.

Modulators

The input values to the parameters shown in the diagrams above can be generalized. When a value is the result of scaling a normalized source, the value is defined as a modulator. The general structure of a modulator is shown below.



The transform is used to convert raw values to the appropriate scale. A modulator *n* defined as a function M(*n*) multiplies the input source (s_n), scaling amount value (a_n), and control source (c_n), and applies the transform function T(x).

$$M(n) = T(s_n a_n c_n)$$

The Pitch, Filter, Volume and Pan parameters can be defined as the sum of one or more modulators. This generalizes the definition of the synthesis instrument since the modulator may be an initialization value, the output of another generator, or a MIDI channel value.

Modulators also include a destination identifier, indicating which unit generator should receive the output value. All modulators for a given destination must use the same units for the scale value and are summed to produce the parameter value for the unit generator.

Some input source values are inherent in the MIDI protocol and are required to be implemented. The internal LFO and EG units also have some implied connections. These are shown in the following table.

Source	Destination	Use
NONE	Any	The source is considered to have a value of 1. This is used to supersede the default initialization value on a unit generator.
KEY	PITCH, EG1, EG2	MIDI note-on key value. This is applied to oscillator frequency, envelope decay rates, and used to select the wavetable(s).
VELOCITY	VOLUME, EG1, EG2	The MIDI note-on velocity value. This value is applied to volume and

		envelope attack rate scaling, and used to select the wavetable(s).
PITCHWHEEL	PITCH	MIDI pitch wheel value (CV message 12). Typically applied to the pitch. The scaling value is taken from RPN0.
LFO1	PITCH or VOLUME	The oscillator can be used for either vibrato or tremolo. For vibrato, the scaling value is in pitch cents. For tremolo, the scaling value is in centibels. The default is to use LFO1 for vibrato.
LFO2	VOLUME or FILTER	A second LFO. Typically, this is applied to volume or filter frequency. DLS1 does not include this generator.
EG1	VOLUME	Volume envelope generator.
EG2	PITCH or FILTER	Modulation envelope generator.
MODWHEEL	LFO1, LFO2	MIDI modulation wheel (CC #1). Applied as a control to the level of a LFO.
VOLUME	VOLUME	MIDI channel volume (CC #7). Transformed to a normalized concave curve, scaled to cB and added to the volume level.
PAN	PAN	MIDI pan (CC #10). Scaled and added to the pan value.
EXPRESSION	VOLUME	MIDI expression (CC #11). Transformed to a normalized, concave curve, scaled to cB and added to the volume level.
REVERB	REVERB	MIDI reverb send (CC #91).
CHORUS	CHORUS	MIDI chorus send (CC #93).
RPN0	PITCHWHEEL	Pitch wheel sensitivity. Applied as a control to the pitch bend modulator.
RPN1	PITCH	MIDI fine tune (cents). Added to the MIDI note-on key value to adjust the oscillator frequency.
RPN2	PITCH	MIDI course tune (semi-tones). Added to the MIDI note-on key

		value to adjust the oscillator frequency.

Because of the generic nature of modulators, it is theoretically possible to apply any input source to any destination, provided the scaling value is set to the appropriate units for the destination. The scale is in pitch cents when applied to pitch or filter, and centibels when applied to volume, time cents when applied to an EG rate, etc. However, some connections are not meaningful. For example, the envelope rates are applied at note start time and changing them with a performance controller while the note is playing may have no effect.

By defining all parameters as modulators, the program to implement the synthesizer can be defined as a loop that executes each modulator and the evaluates the oscillator, filter and amplifier. However, modulators are conceptual, and it is not necessary to implement all synthesizer functions with modulators. For example, a source or control of NONE sets the respective input value to a constant of 1. When both source and control are set to NONE, the scale is an initialization value for the destination. The output of these modulators cannot change during sound generation and thus only need to be evaluated when the note starts. The best implementation is to store these modulators directly into the unit generators. For the example programs we will consider any parameter, whether initialization or dynamic, to be represented by a modulator.

Frequency Values

The frequency of a recorded sound could be specified directly in the sound bank, but, for most wavetables based on recorded sound, the frequency is specified as combination of pitch class (or keyboard key number) and frequency deviation in cents (1/100 of a semitone). A frequency is calculated from a MIDI key number and pitch cents by:

$$f_o = 440 \cdot 2^{\frac{key-69}{12}} \cdot 2^{\frac{cents}{1200}} = 440 \cdot 2^{\frac{100(key-69)+cents}{1200}}$$

The value of 69 represents the MIDI key for A4, which has a frequency of 440Hz. Pitch cents have a one-octave range of [0,1200] and a total ten-octave range of [0,12000]. Note that we can use any frequency as the reference frequency so long as we adjust the pitch number offset appropriately. For example, if we use the frequency for pitch number zero (8.175Hz) in place of 440Hz, we eliminate the subtraction of the pitch number offset. Applying the rules for exponents we can combine the two exponential terms together, as shown in the second form of the equation.

Pitch cents added to a base frequency are called *relative pitch cents*, while the entire frequency range is called *absolute pitch cents*. Relative pitch cents can be positive or negative numbers. The conversion from semitones and cents to frequency can be done directly using the *pow* library function, or with two lookup tables for pitch and cents, with pitch covering a ten-octave range in semitones, and cents covering a two octave range, [-1200,+1200]

In order to improve performance, we use a table lookup to convert key to frequency and cents to a multiplier, avoiding the time consuming exponential calculations at runtime.

Given a MIDI key number and a detune or modulator value in cents, the code to calculate frequency directly is:

```
frq = 440.0 * pow(2.0, ((100.0*(key-69))+cents) / 1200.0);
```

For relative cents, the tables are calculated as:

```
for (n = 0; n < 2400; n++)
```

```
     relPC[n] = pow(2.0, (n-1200.0)/1200.0);
for (n = 0; n < 128; n++)
     frqST[n] = 440.0 * pow(2.0, (n-69)/12.0);
```

We use 128 instead of 120 for the semi-tone range since MIDI key numbers have that range. In actuality, MIDI key numbers rarely reach to the upper and lower limits, but we include the entire range to avoid testing the range during lookup.

Frequency can be calculated using a table lookup by:

```
frq = frqST[key] * relPC[cents+1200];
```

Frequency can be calculated from absolute pitch cents using two tables by:

```
frq = frqST[pcabs/100] * relPC[(pcabs%1200)+1200];
```

Alternatively, we can use a single table covering the entire 10 octave range. That means we need a table with 12800 entries.

```
for (n = 0; n < 12800; n++)
     absPC[n] = 440.0 * pow(2.0,(n-6900.0)/1200.0);
```

The lowest frequency in the table is 8.175Hz, and the highest is 13,289Hz. Frequency is calculated from key and cents, or absolute cents by:

```
frq = absPC[key*100+cents];
frq = absPC[pcabs];
```

Amplitude Values

Amplitude values are specified in 10^{ths} of a decibel (centibels or cB). Thus a range of [0,960] represents up to 96dB of attenuation. For 16-bit amplitude values, 96dB represents the maximum range of possible values. For 24-bit values, the range would extend to 144dB, or [0,1440] centibels.

Values in cB of attenuation are converted to normalized amplitude by:

$$a = 10^{\frac{-cB}{200}}$$

The negative sign in the exponent is needed because the cB value represents attenuation. In other words, a value of 960cB should map to 0 (or close to it), and 0cB should map to a normalized amplitude of 1. The code to convert cB to amplitude is:

```
amp = pow(10, cb/-200);
```

The exponential calculation can be made with a lookup table calculated as:

```
for (n = 0; n < MAX_CB; n++)
     ampCB[n] = pow(10, (double)n / -200.0);
```

The constant MAX_CB is typically 960, but can be set to 1440 to allow the full range for single precision float types. When performing a lookup, any value greater than or equal to MAX_CB represents maximum attenuation and should return 0. Any value less than 0 should return 1.

Envelope Rates

Envelope rate values are specified in *time cents* and are converted to a rate in seconds by:

$$r = 2^{\frac{tc}{1200}}$$

The calculation for *tc* is the inverse, i.e.:

$$tc = 1200 \cdot \log_2 r$$

This may seem like a strange way to represent time, but it does have a benefit. First, sub-second time values can be specified in integers rather than floating point. The *tc* value is simply a negative number in that case. We could use integers by a linear scaling of the time value to a fixed point value, but there is an advantage to using a logarithmic scale. As the envelope rate increases beyond 1s, small increments of time become less perceptible. In other words, an increase in the attack rate from 1ms to 100ms produces a perceptible change, but an increase from 5s to 5.1s does not. Consequently, most of a linear range is wasted. Because it is a logarithmic scale, the time in seconds between adjacent *tc* values increases as the time cent value increases, and each increase of *tc* is more likely to be a useful change in rate. Using time cents, a time of 32 seconds is a *tc* value of 6000 while a value of 1ms is a *tc* value of about -12000, resulting in a total range of 18,000. Extending the range to [-12000,8000] allows time values up to 100 seconds, but can still be specified with a short integer. Simple linear scaling with a 1ms resolution would be limited to about 32s.

We can calculate times directly from time cents, or use a table of absolute time cents and perform a lookup.

```
for (n = 0; n < 20000; n++)
     absTC[n] = pow(2.0,(n-12000.0)/1200.0);

secTm = absTC[tc+12000];
```

Envelope rates are modified by the pitch and velocity values. Since higher pitched sounds tend to have a shorter decay time, the time can be scaled automatically to simulate this effect. Scaling can be applied "full scale" by simply multiplying the key number by the scale factor and adding to the initial decay rate.

```
decay = decay + (key * scale);
```

Scaling can also be made relative to Middle C:

```
decay = decay + ((key - 60) * scale);
```

DLS files use the first form while SF2 files use the second form.

Attack times tend to be shorter for loud sounds and slightly longer for soft sounds. The normalized note on velocity value is used to scale the attack time.

```
attack = attack + ((velocity/127) * scale);
```

Oscillator

The standard wavetable oscillator uses a wavetable containing exactly one period of the waveform. However, when the wavetable contains a recorded sound rather than a calculated waveform it almost always has more than one period. In addition, the wavetable may have been recorded at a sample rate different from the playback sample rate.

We can change the pitch of the recorded sound if we use a phase increment other than 1, just as with single period wavetables. In order to calculate the phase increment for a desired frequency, we must know the frequency of the recorded waveform and the sample rate at which it was recorded. With those two values we can easily calculate one period of the waveform.

Once we know the frequency we can calculate the period of the waveform in samples. The period length in samples (P) is the time of one period (t_w) divided by the time of one sample (t_s). The time of one sample is the inverse of the wavetable sample rate (f_{sw}) and the time of one period is the inverse of the wavetable frequency (f_w).

$$t_w = \frac{1}{f_w}$$

$$t_s = \frac{1}{f_{sw}}$$

$$P = \frac{t_w}{t_s} = \frac{1 / f_w}{1 / f_{sw}} = \frac{f_{sw}}{f_w}$$

Thus, the period of the waveform is simply the sample rate of the wavetable divided by the frequency of the wavetable. We can also see this by using the phase increment for playback at the original frequency. In order to play the wavetable at its original frequency, the phase increment must be equal to 1. Rearranging the terms gives us the same result as before.

$$1 = \frac{P \cdot f_w}{f_{sw}}$$

$$P = \frac{f_{sw}}{f_w}$$

Once we know the period of the wavetable we can calculate the phase increment as before using the period (P) for table length (L). Substituting and then refactoring gives us the final equation for pitch shifting a sampled recording.

$$\phi = \frac{\left(f_{sw} / f_w \right) \cdot f_o}{f_s}$$

$$\phi = f_{sw} \cdot \frac{1}{f_w} \cdot f_o \cdot \frac{1}{f_s}$$

$$\phi = \frac{f_{sw}}{f_s} \cdot \frac{f_o}{f_w}$$

Intuitively, this is correct as it shows the phase increment is the ratio of the desired pitch to the recorded pitch. When the recorded sample rate is the same as the playback sample rate, the first factor is 1. Likewise, when the recorded frequency is the same as the playback frequency, the second factor is also 1. Thus the phase increment is 1 and the result is to playback the wavetable as recorded. An increase in pitch results in a phase increment greater than 1, while a decrease in pitch results in a phase increment less than 1.

To implement the oscillator, we need to calculate a phase increment and then iteratively add the increment to the phase for each sample. The phase increment for a multi-period oscillator is:

$$\varphi = \frac{f_{sw}}{f_s} \cdot \frac{f_o}{f_w}$$

Here, f_{sw} is the sample rate of the wavetable, f_s is the sample rate of the synthesizer, f_w the wavetable frequency, and f_o the desired oscillator frequency. Intuitively, when the sample rates are the same and the frequencies are the same, the phase increment is one, and we playback the sample at the recorded frequency.

The wavetable frequency may be given directly in Hz, but is typically defined by a combination of a *root key* (k_r) and a *detune* amount (d) in cents. The playback frequency for the oscillator uses the MIDI key number, adjusted by the MIDI RPN1 (fine tune) and RPN2 (coarse tune) values. This is accomplished by adding the coarse tune value (t_c) to the key number (k), converting to cents, and adding the fine tune (t_f).

$$f_o = 440 \cdot 2^{\frac{100(k+t_c-69)+t_f}{1200}}$$

$$f_w = 440 \cdot 2^{\frac{100(k_r-69)+d}{1200}}$$

Given each frequency in cents, we can use our pitch cents to frequency conversion tables to lookup the two frequencies. The oscillator is then implemented as follows.

```
oscFrq = frqST[key + coarse] * relPC[fine+1200];
wavfrq = frqST[root] * relPC[detune+1200];
// OR, using absolute pitch cents table:
oscFrq = absPC[(100 * (key + coarse)) + fine];
wavfrq = absPC[(100 * root) + detune];

phsIncr = (wavSampleRate / sampleRate) * (oscFrq / wavFrq);
phase = 0;

for (s = 0; s < totalSamples; s++) {
     if (phase >= tableEnd)
          sample[s] = 0;
     else {
          sample[s] = wavetable[phase] * volume;
          phase += phsIncr;
          if (looping && phase >= loopEnd)
               phase -= (loopEnd - loopStart)
```

```
        }
}
```

This code shows a direct index into the wavetable, truncating the fractional part of the phase value. Typically, we will use interpolation of the table in order to minimize quantization errors. The looping flag indicates whether the table should be played through once, or sustained by looping between the loop start and end points. The part of the table prior to the loop start point is always played through once.

Pitch Modulators

The frequency of the oscillator for a given sample is the combination of the note frequency and any modulators applied to pitch. Modulating a frequency involves multiplying the oscillator frequency by a frequency ratio. In other words, the output of a modulator must be converted from relative pitch cents to a frequency multiplier.

$$M(n) = T(s_n a_n c_n)$$

$$F(n) = 2^{\frac{M(n)}{1200}}$$

$$\varphi = \frac{f_{sw}}{f_s} \cdot \frac{f_o}{f_w} \cdot \prod_{n=0}^{m} F(n)$$

A straight forward implementation multiplies the base frequency by each modulator. But note that each factor in the calculation is a power of two, and thus we can calculate the sum of the exponents (modulators in cents) and add to the note pitch in cents without the intermediate conversion to frequency. We can also convert the values for f_{sw} *and* f_s into a power of two by calculating the cents value from the frequencies. The cents value is the logarithm of the frequency.

$$c_s = 1200\log_2(f_s / 440) + 6900$$

$$c_{sw} = 1200\log_2(f_{sw} / 440) + 6900$$

$$c_w = 100k_r + d$$

$$c_o = 100(k + t_c) + t_f$$

$$c_m = \sum_{n=0}^{m} M(n)$$

$$\frac{f_{sw}}{f_s} = \frac{440 \cdot 2^{c_{sw}/1200}}{440 \cdot 2^{c_s/1200}} = 2^{\frac{c_{sw} - c_s}{1200}}$$

$$\frac{f_o}{f_w} = \frac{440 \cdot 2^{c_o/1200}}{440 \cdot 2^{c_w/1200}} = 2^{\frac{co - cw}{1200}}$$

$$\varphi = 2^{\frac{c_{sw} - c_s}{1200}} \cdot 2^{\frac{c_o - c_w}{1200}} \cdot 2^{\frac{c_m}{1200}} = 2^{\frac{(c_{sw} - c_s) + (c_o - c_w) + c_m}{1200}}$$

The sum of c_{sw}, c_s, c_o, and c_w can be calculated once at the start of the note and then added to the modulator sum (c_m) for each sample. Using this form, the incremental phase calculation is done by accumulating the modulators in pitch cents and performing one exponential operation per sample value. The final value is a wavetable index increment, not a frequency, and represents the ratio of the recording frequency to the playback frequency. We will want to use a table lookup for the final conversion of pitch cents to phase increment, instead of exponentiation. The size of the table depends on the range of pitch shift we need to accommodate. The DLS specification requires a minimum of +2/-4 octaves requiring a table with a range [-4800,+2400]. The worst-case is the full 10 octave range, requiring values to cover [-12000,+12000] cents.

```
// -4, +2 octaves
for (n = 0; n < 6200; n++)
     phsPC[n] = pow(2, (n-4800)/1200);

// -10, +10 octaves
for (n = 0; n < 24000; n++)
     phsPC[n] = pow(2, (n-12000)/1200);
```

Because the table spans a negative to positive range, we need to add an offset to the index value. The offset value (4800 or 12000) can be added to the initial cents variable instead of adding to the phase increment on each sample. Since the formula for time cents is the same as for the conversion from pitch cents to phase increment we could potentially use the same table for both.

Using cents, the program is faster, but the difference in execution time is dependent on the number of modulators. If only one or two modulators are applied to pitch, the difference may be negligible.

Note: The C++ runtime library does not include a $\log_2$ function. However, we can define one using the formula for logarithm base conversion ($\log_n(x)=\log_e(x)/\log_e(n)$). The constant *ln2* is $\log_e(2)$.

```
const double ln2 = 0.693147180559945;
double log2(double n) { return log(n) / ln2; }
```

Scale Tuning

The SF2 specification includes a feature that allows varying the tuning ratio between adjacent pitches. Normally, each semi-tone represents an increase or decrease of 100 cents, but, when scale tuning is in effect, the amount of change is scaled. With a scale value of 100, pitches are the standard 12-tone scale. With a scale of 0, all pitches would sound at the same frequency.

In simplest terms, scale tuning changes the multiplier for conversion from semi-tones to pitch. But this must be made relative to some known pitch. Otherwise, scaling the key number by a multiplier less than 100 would move all pitches towards the bottom of the frequency range. What we need to do is calculate the difference in cents between the root pitch and the performed pitch. We can expand the calculation for oscillator cents as:

$$c_t = c_o - c_w$$
$$c_t = 100 \cdot (k + t_c) + t_f - (100k_r + d)$$
$$c_t = 100 \cdot k + 100 \cdot t_c + 100t_f / 100 - 100 \cdot k_r - d$$
$$c_t = 100 \cdot (k + t_c + (t_f / 100) - k_r) - d$$

We change the scale of the pitch by changing the coefficient of the first term to the scaling value. The value for c_t can now replace (c_o-c_w) in the phase calculation. Note that we don't want to scale the detune value (d) because that value adjusts the base frequency of the wavetable to be equivalent to the root key.

The final oscillator code is shown below. For now, we will represent the modulator values with function calls, and show the volume level as a single variable. (The transform is not shown.)

```
Modulator(n)                                                    {
     if (ModDst(n) == PITCH)
          phsIncr *= relPC[ModValue(n)];
}

if (sampleRate != wavSampleRate) {
     srCents  = (1200 * log2(sampleRate/440));
     wsrCents = (1200 * log2(wavSampleRate/440));
     phsCents = wsrCents - srCents;
} else {
     phsCents = 0;
}
oscCents = scaleTune * (key + coarse + (fine*0.01) - root);
phsCents += oscCents - detune;
phsCents += 4800; // 6 octave table
//phsCents += 12000; // 10 octave table

for (s = 0; s < totalSamples; s++) {
     if (phase >= tableEnd) {
          sample[s] = 0;
          continue;
     }
     phsIncrPC = phsCents;
     for (n = 0; n < m; n++)
          Modulator(n);
     sample[s] = wavetable[phase] * volume;
     phase += phsPC[phsIncrPC];
     if (looping && phase >= loopEnd)
          phase -= (loopEnd - loopStart)
  }
```

Amplitude Modulators

The combined amplitude value of modulators is the product of the normalized amplitude values. M(*n*) is a modulator with amplitude as the destination.

$$A(n) = 10^{\frac{M(n)}{-200}}$$

$$a = \prod_{n=0}^{m} A(n)$$

Since values in cB are already logarithms, we can sum the values, just as we did with frequency calculations, and then convert the sum into normalized amplitude.

$$cB = \sum_{n=0}^{m} M(n)$$

$$a = 10^{\frac{cB}{-200}}$$

The modulators applied to volume include initial attenuation, MIDI note-on velocity, MIDI volume (CC#7), MIDI expression (CC#11), LFO and volume envelope generator. The normalized values from these sources are scaled to cB, summed and converted to normalized amplitude, and then multiplied by the filter output to produce the final amplitude value.

MIDI values have a range of [0,127] and need to be scaled and transformed into *cB* units. For volume, the linear input range is transformed to a convex curve and then converted.

$$cB = 200\log_{10}\left(\frac{127^2}{v^2}\right)$$

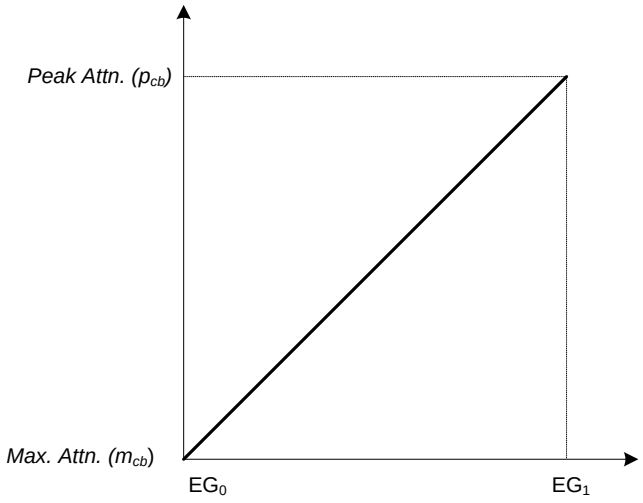
As we have been doing, we can implement the calculation directly, or use a lookup table.

```
midiCB[n] = 960;
for (n = 1; n < 128; n++)
     midiCB[n] = 200 * log10((127*127)/(n*n));
```

The LFO generator varies from [-1,1]. For tremolo, we want the value to be in the positive range [0,1] and scaled by *cB*.

```
lfoOut = (1 - LFO()) * 0.5 * lfoScale;
```

Since the output of an envelope generator can be applied to volume, pitch or filter, the envelope is usually calculated as a normalized value [0,1] and then scaled by a value in the appropriate unit. However, for the volume envelope generator, the scaled value represents attenuation, and we must transform the value differently. We must map the maximum envelope value to the peak attenuation and the minimum envelope value to maximum attenuation.



Here, p_{cb} is the attenuation at the envelope peak, m_{cb} is the attenuation at the envelope minimum, and EG_n is the output of the envelope generator at level *n*. Applying the equation for the line gives us the amplitude in *cB* for any envelope generator output level.

$$cB_n = (p_{cb} - m_{cb}) \cdot EG_n + m_{cb}$$

Both SF2 and DLS specifications have a fixed scale of [0,960] for volume envelope. We can reduce the calculation for *cB* by substituting a 0 value for peak and 960 for the minimum.

$$cB_n = (0 - 960) \cdot EG_n + 960$$
$$cB_n = 960 - 960 \cdot EG_n$$
$$cB_n = 960 \cdot (1 - EG_n)$$

Attack rates usually have a convex (exponential) curve. For the attack segment we can simply square the output of the envelope generator before converting to *cB*, but to achieve a linear attack ramp will need a more complicated solution. (See below on normalized transfer functions.)

For DLS files, an envelope sustain level is specified as a percentage of peak output and can be set directly on the envelope generator. For SF2 files, the envelope sustain level is specified as an increase in attenuation (decrease in volume) applied to the peak level. We calculate the envelope sustain level by setting the first equation to the sustain level and solving for the EG level.

$$cB_{sus} = p_{cb} + s_{cb}$$
$$p_{cb} + s_{cb} = (p_{cb} - m_{cb}) \cdot EG_{sus} + m_{cb}$$
$$(p_{cb} + s_{cb}) - m_{cb} = (p_{cb} - m_{cb}) \cdot EG_{sus}$$
$$EG_{sus} = \frac{(p_{cb} + s_{cb}) - m_{cb}}{p_{cb} - m_{cb}}$$

Setting the peak amplitude to 0 and the minimum to 960, we can simplify to:

$$EG_{sus} = \frac{s_{cb} - 960}{-960} = \frac{960 - s_{cb}}{960}$$

Combined with the code above, we now have the following for the modulator.

```
Modulator(n) {
```

```
      if (ModDst(n) == PITCH)
           phsIncr += ModValue(n);
      else if (ModDst(n) == VOLUME) {
           if (ModSrc(n) == EG1)
                ampVal += 960*(1-EG());
           else if (ModSrc(n) == MIDI)
                ampVal += midiCB[ModValue(n)];
           else if (ModSrc(n) == LFO)
                ampVal += ModValue(n) * (1-LFO());
      }
}
```

Normalized MIDI Inputs

MIDI values applied as modulators are normalized before multiplying with the modulator scale value. There are four basic normalization transfer functions, linear, concave, convex and switch. Each of these can be inverted.

The linear transfer function is simply a scaling of the MIDI value to the range [0,1]. The switch transfer function is true when the MIDI value is greater than the midpoint. The concave curve is the decibel value of the squared MIDI value scaled to the range [0dB,96dB], then normalized to a range of [0,1].

$$X(n) = \frac{-20 \cdot \log_{10}\left((127-n)^2/127^2\right)}{96}$$

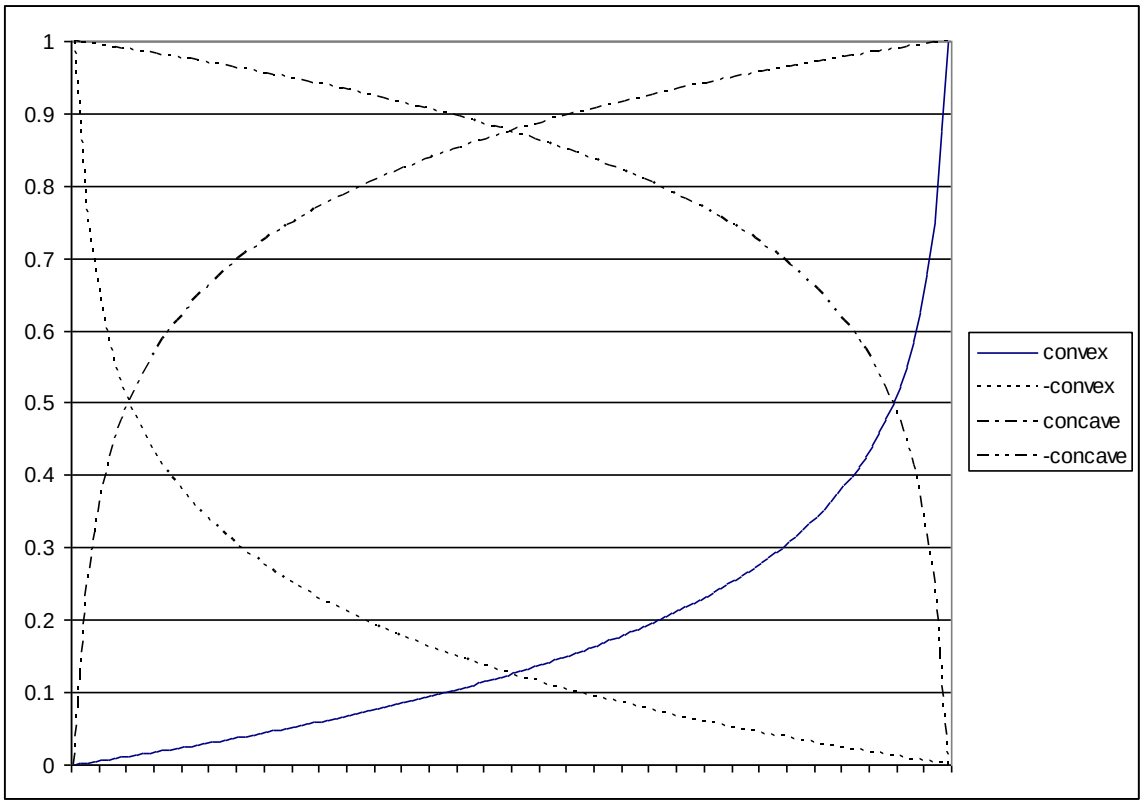
$$X^{-1}(n) = \frac{-20 \cdot \log_{10}\left(n^2/127^2\right)}{96}$$

Note that the function X(n) is undefined for n = 127 and $X^{-1}(n)$ is undefined for n = 0. We use 1 and 0, respectively in those cases.

The convex curve is the inverse of the concave curve.

$$V^{-1}(n) = 1 + \left[\frac{20 \cdot \log_{10}\left((127-n)^2/127^2\right)}{96}\right]$$

$$V(n) = 1 + \left[\frac{20 \cdot \log_{10}\left(n^2/127^2\right)}{96}\right]$$

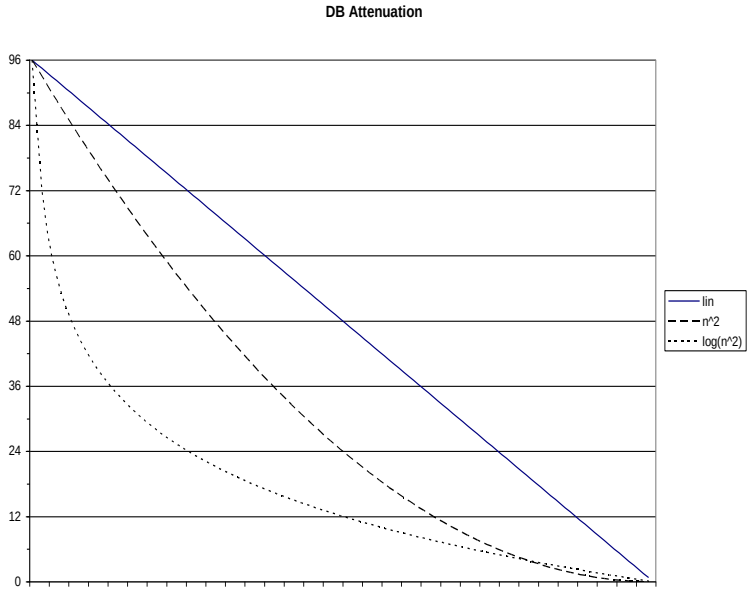


All the curves can be calculated from each other by either taking the inverse or reversing the end points. For the case when x=log(0), we initialize the appropriate end-point of each curve separately.

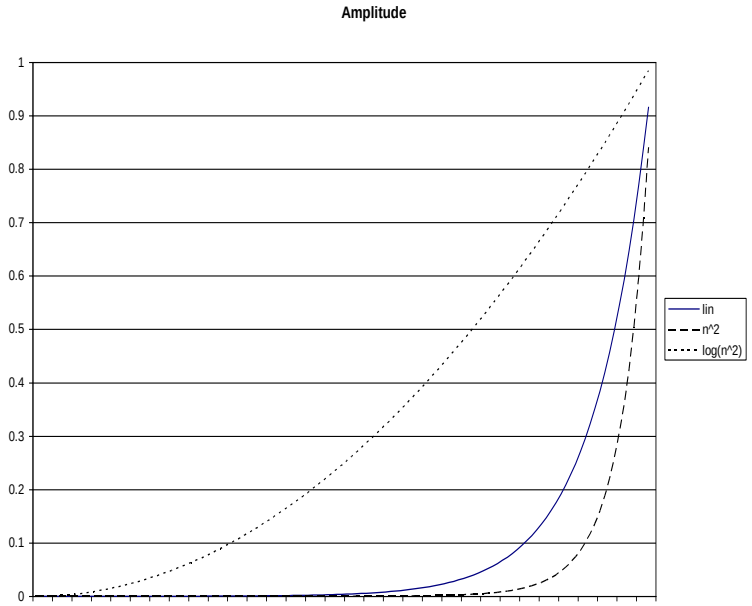
```
posLinear[0] = 0;
negLinear[0] = 1;
posConcave[127] = 1;
negConcave[0] = 1;
posConvex[0] = 0;
negConvex[127] = 0;
posSwitch[0] = 0;
negSwitch[0] = 1;

for (n = 1; n < 128; n++) {
     R = n/127;
     R2 = R * R;
     V = -20.0f/96.0f * log10(R2);
//   V = 20/96 * log10(1/R2);
     posLinear[n] = R;
     negLinear[n] = 1 - R;
     posConcave[127-n] = v;
     negConcave[n] = v;
     posConvex[n] = 1 - v;
     negConvex[127-n] = 1 - v;
```

```
        posSwitch[n] = R >= 0.5;
        negSwitch[n] = 1 - posSwitch[n];
}
```

The benefit of using these complicated normalization curves is to provide a linear output while still working with centibels of attenuation. The following graph shows how the negative linear and convex curve scales the raw MIDI input to decibels of attenuation. A n^2 curve is shown for comparison.



When converted to normalized amplitude, the negative convex curve results in a nearly flat amplitude scale. The linear transfer function produces a dB output curve.



Unrolling

The code examples have shown a loop that applies modulators. This is the generalized version and is normally used so that modulators matching defaults will supersede the implied connection. However, if there are no modulators other than implied modulators, or we always want to apply the defaults, we can apply the implied modulators directly in sequence without having to loop. Also, MIDI values do not change quickly, relative to the sample rate. If the block size for sample rendering is equal to 1ms or less, we only need to read the MIDI controller values once per block.

```
atten = initAtten
      + midiCB[velocity];
      + midiCB[chnlVol]
      + midiCB[chnlExpr];
pitchWheel = midiIn[chnlPw];
modWheel = midiIn[chnlMW];

for (s = 0; s < totalSamples; s++) {
    eg1Val = EG1();
    eg2Val = EG2();
    lfo1Val = LFO1();
    lfo2Val = LFO2();
    phsIncPC = phsCents
             + pitchWheel
             + lfo1Val * (lfoFrqScl + (modWheel*mwPC))
             + eg2Val * modFrqScale;

    ampVal = atten
           + lfo2Val * (lfoAmpScl + (modWheel * mwCB))
           + (1 - eg1Val) * 960;

    fltVal = initFilter
           + lfo2Val * (lfoFltScl+ (modWheel*mwFC))
           + EG2() * modFltScl;

    panVal = initPan + midiPan;

    out = FILTER(wavetable[phase]) * ampCB[ampVal];
    left[s] = PAN(out,-panVal);
    right[s] = PAN(out,panVal);

    phase += phsPC[phsIncPC];
    if (looping && phase >= loopEnd)
        phase -= (loopEnd - loopStart)
  }
```

This technique, known as *unrolling a loop*, replaces the loop with a series of inline statements. It eliminates the modulator loop overhead since we don't have to test the source and destination values. Thus, what we give up in flexibility we gain in performance.

Minimal Implementations

The synthesis model defined for sound banks is generic, but the SF2 and DLS specifications are clearly targeted to a hardware implementation. Both tacitly assume that any number of modulators can be evaluated without affecting the speed of sample generation. While most likely true for digital circuits, this assumption can create performance problems for a software synthesizer where all generators must be run sequentially with their execution interwoven with other sequencer operations. The more modulators we execute, the longer it will take to generate a sample. Ideally, a software implementation should skip execution of modulators that do not affect the sound output. For example, if the EG2 value is not used, or the scale value is specified as 0, there is no need to execute the EG2 code or add the value to the pitch and/or volume parameters. A simple optimization is to discard any modulators that have a scaling value of 0 as the sound bank is loaded into memory. Likewise, if the MIDI input file does not contain any pitch bend or modulation wheel messages, we can skip any modulators that use those values as either source or control values.

Unlike a hardware implementation, a software synthesizer can easily support multiple instruments. We obviously want to have one instrument that implements all features available in the sound bank and is fully compliant with the specification. But we can also have a minimal instrument that is optimal for limited cases. A common scenario is one where the sound bank uses recorded sounds and does not modify those sounds with the modulation envelope, second LFO, or filter, and the MIDI input contains only key, velocity, volume, and pitch bend. We can detect this situation when loading the sound bank and MIDI file and automatically select a minimal instrument, or simply select this instrument whenever we need fastest execution. In addition, volume, panning, reverb, and chorus apply to all notes on the channel. These values can be applied at a final mix-down point, rather than in the instrument.

The modulator evaluation code above is replaced with two simple statements:

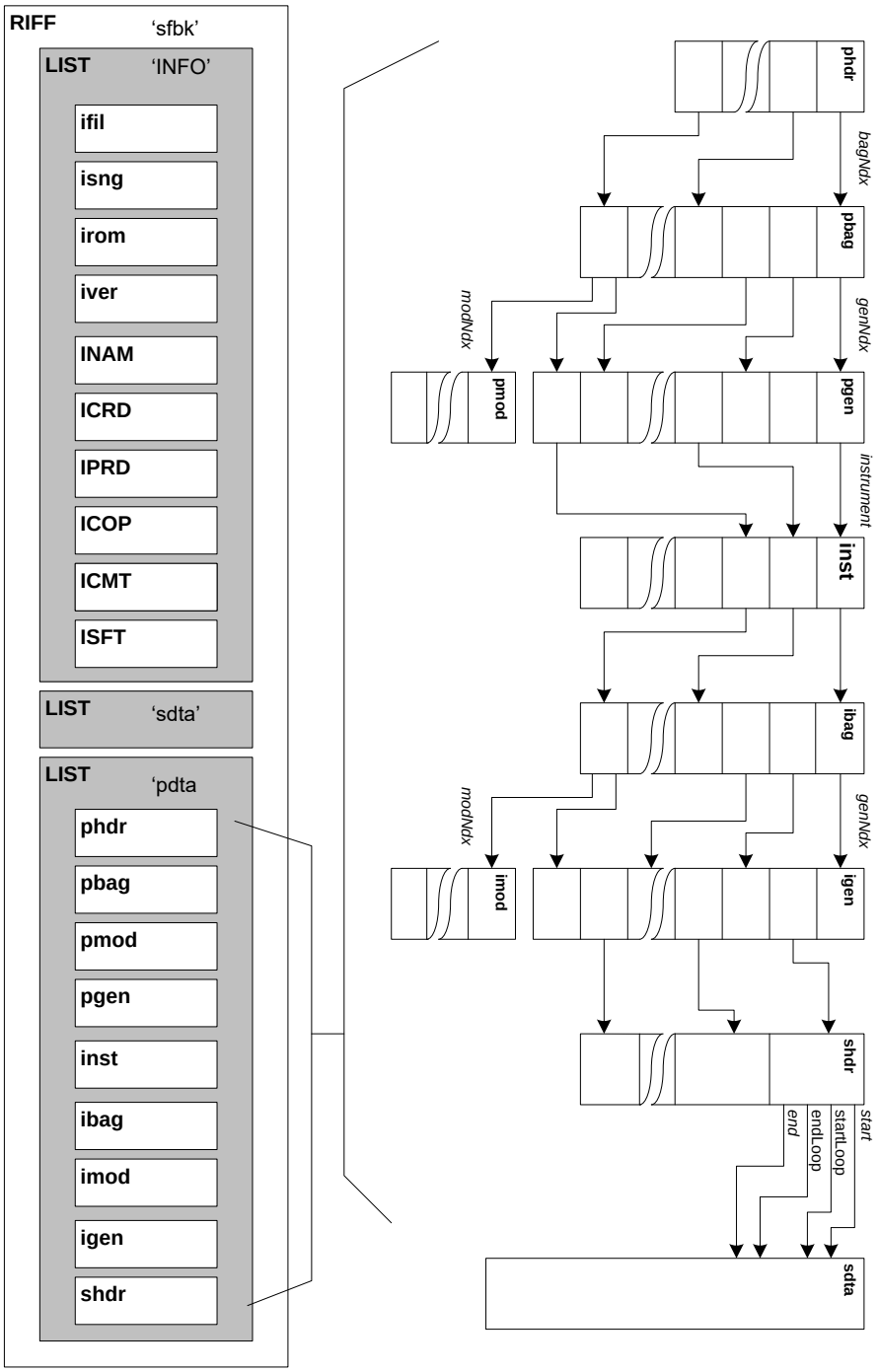
```
phsIncPC = phsCents + pitchWheel + lfo1Val * lfo1Scl;
ampVal = atten + ((1 – EG()) * 960);
```

Although this seems quite limited, it will correctly render a large number of sound bank and MIDI file combinations. Other modulators can be evaluated if the scaling value is non-zero. We can do this by using a set of flags indicating whether or not a modulator should be evaluated.

```
if (flags & EG1_PITCH)
     phsIncPC += eg1Val * eg1SclPitch;
if (flags & LFO2_VOLUME)
     ampVal += eg2Val * eg2SclVol;
<< etc. >>
```

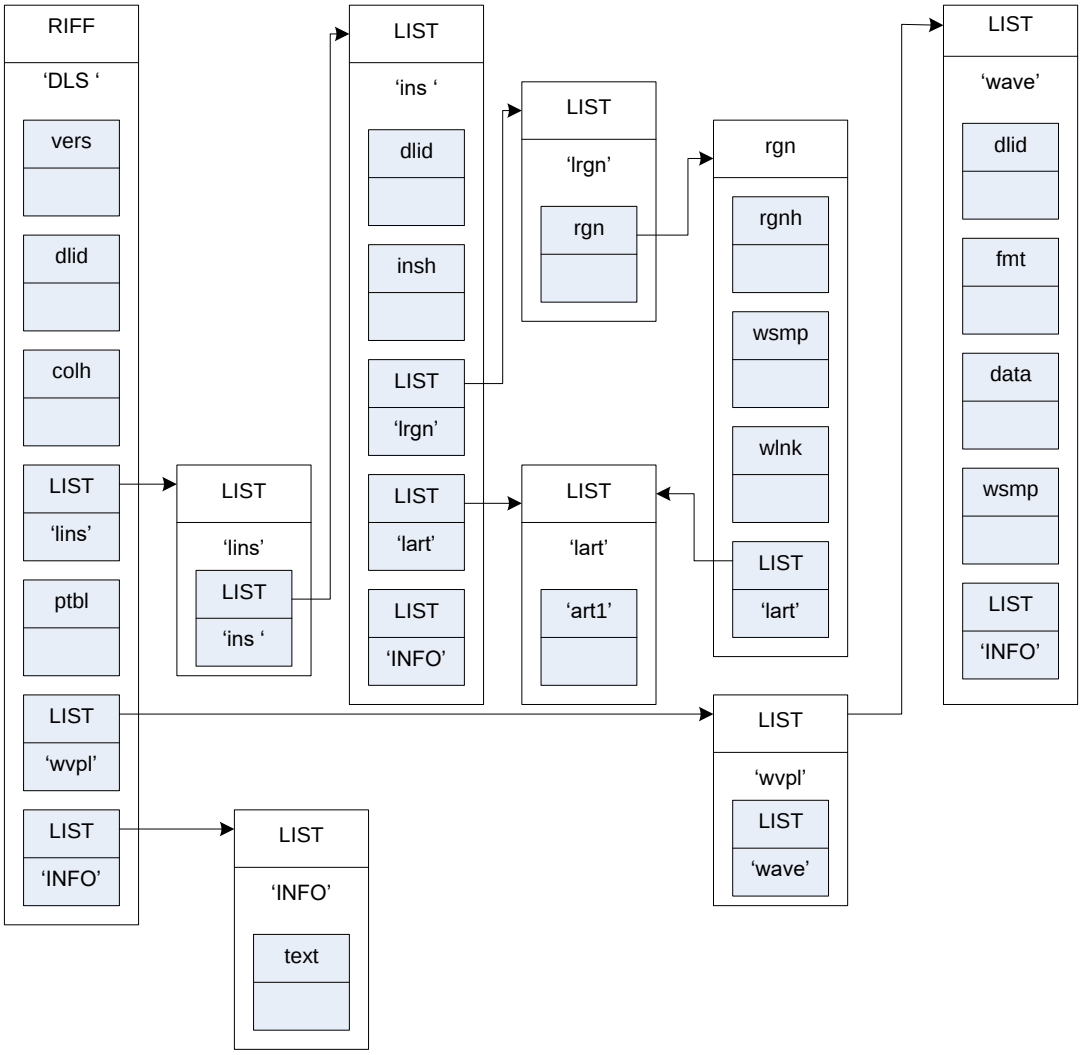
SoundFont File Structure

A SF2 file is a RIFF file with each chunk representing one of several data records. Overall, the SF2 file is similar to a ISAM file structure.



DLS File Structure

A DLS file is a RIFF file with a hierarchical structure.



References

Downloadable Sounds Level 1, MIDI Manufacturers Association, September 2004

Downloadable Sounds Level 2, MIDI Manufacturers Association

General MIDI 2, MIDI Manufacturers Association

SoundFont Technical Specification, Version 2.04, Creative/E-mu Systems, 1996.