User Manual
The Synthesis ToolKit in C++

by Perry R. Cook and Gary P. Scavone

©1995–2002
## Contents

1. **STK Hierarchical Index**  
   1.1 STK Class Hierarchy ........................................ 1

2. **STK Compound Index**  
   2.1 STK Compound List .................................. 5

3. **STK Page Index**  
   3.1 STK Related Pages ...................................... 9

4. **STK Class Documentation**  
   4.1 ADSR Class Reference ............................... 11
   4.2 BandedWG Class Reference ....................... 14
   4.3 BeeThree Class Reference ....................... 16
   4.4 BiQuad Class Reference ......................... 18
   4.5 BlowBotl Class Reference ....................... 21
   4.6 BlowHole Class Reference ....................... 23
   4.7 Bowed Class Reference .......................... 25
   4.8 BowTabl Class Reference ....................... 27
   4.9 Brass Class Reference ........................... 29
   4.10 Chorus Class Reference ......................... 31
   4.11 Clarinet Class Reference ...................... 33
   4.12 Delay Class Reference ......................... 35
   4.13 DelayA Class Reference ....................... 38
   4.14 DelayL Class Reference ....................... 40
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.46 Resonate Class Reference</td>
<td>106</td>
</tr>
<tr>
<td>4.47 Reverb Class Reference</td>
<td>108</td>
</tr>
<tr>
<td>4.48 Rhodey Class Reference</td>
<td>110</td>
</tr>
<tr>
<td>4.49 RtAudio Class Reference</td>
<td>112</td>
</tr>
<tr>
<td>4.50 RtAudio::RTAUDIO_DEVICE Struct Reference</td>
<td>119</td>
</tr>
<tr>
<td>4.51 RtDuplex Class Reference</td>
<td>122</td>
</tr>
<tr>
<td>4.52 RtError Class Reference</td>
<td>125</td>
</tr>
<tr>
<td>4.53 RtMidi Class Reference</td>
<td>127</td>
</tr>
<tr>
<td>4.54 RtWvIn Class Reference</td>
<td>129</td>
</tr>
<tr>
<td>4.55 RtWvOut Class Reference</td>
<td>132</td>
</tr>
<tr>
<td>4.56 Sampler Class Reference</td>
<td>135</td>
</tr>
<tr>
<td>4.57 Saxofony Class Reference</td>
<td>137</td>
</tr>
<tr>
<td>4.58 Shakers Class Reference</td>
<td>139</td>
</tr>
<tr>
<td>4.59 Simple Class Reference</td>
<td>142</td>
</tr>
<tr>
<td>4.60 SingWave Class Reference</td>
<td>144</td>
</tr>
<tr>
<td>4.61 Sitar Class Reference</td>
<td>147</td>
</tr>
<tr>
<td>4.62 SKINI Class Reference</td>
<td>149</td>
</tr>
<tr>
<td>4.63 Socket Class Reference</td>
<td>152</td>
</tr>
<tr>
<td>4.64 Sphere Class Reference</td>
<td>156</td>
</tr>
<tr>
<td>4.65 StifKarp Class Reference</td>
<td>158</td>
</tr>
<tr>
<td>4.66 Stk Class Reference</td>
<td>160</td>
</tr>
<tr>
<td>4.67 StkError Class Reference</td>
<td>164</td>
</tr>
<tr>
<td>4.68 SubNoise Class Reference</td>
<td>165</td>
</tr>
<tr>
<td>4.69 Table Class Reference</td>
<td>167</td>
</tr>
<tr>
<td>4.70 TcpWvIn Class Reference</td>
<td>169</td>
</tr>
<tr>
<td>4.71 TcpWvOut Class Reference</td>
<td>172</td>
</tr>
<tr>
<td>4.72 Thread Class Reference</td>
<td>175</td>
</tr>
<tr>
<td>4.73 TubeBell Class Reference</td>
<td>177</td>
</tr>
<tr>
<td>4.74 TwoPole Class Reference</td>
<td>179</td>
</tr>
<tr>
<td>4.75 TwoZero Class Reference</td>
<td>182</td>
</tr>
<tr>
<td>4.76 Vector3D Class Reference</td>
<td>185</td>
</tr>
</tbody>
</table>

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
Chapter 1

STK Hierarchical Index

1.1 STK Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

- Phonemes ................................................. 93
- RtAudio ................................................. 112
- RtAudio::RTAUDIO_DEVICE ......................... 119
- RtError .................................................. 125
- Sphere ................................................... 156
- Stk ......................................................... 160
- BowTabl ................................................... 27
- Chorus .................................................... 31
- Echo ....................................................... 44
- Envelope ............................................... 46
  - ADSR .................................................. 11
  - Filter ................................................ 48
    - BiQuad .............................................. 18
    - FormSwept ......................................... 53
  - Delay ................................................. 35
    - DelayA ............................................. 38
    - DelayL ............................................. 40
  - OnePole ............................................. 87
  - OneZero .............................................. 89
  - PoleZero .......................................... 100
  - TwoPole ............................................ 179
  - TwoZero ............................................ 182
- Instrmnt ................................................. 63
- BandedWG .............................................. 14
- BlowBotl ............................................... 21
<table>
<thead>
<tr>
<th>STK Hierarchical Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlowHole</td>
</tr>
<tr>
<td>Bowed</td>
</tr>
<tr>
<td>Brass</td>
</tr>
<tr>
<td>Clarinet</td>
</tr>
<tr>
<td>Drummer</td>
</tr>
<tr>
<td>Flute</td>
</tr>
<tr>
<td>FM</td>
</tr>
<tr>
<td>BeeThree</td>
</tr>
<tr>
<td>FMVoices</td>
</tr>
<tr>
<td>HevyMetl</td>
</tr>
<tr>
<td>PercFlut</td>
</tr>
<tr>
<td>Rhodey</td>
</tr>
<tr>
<td>TubeBell</td>
</tr>
<tr>
<td>Wurley</td>
</tr>
<tr>
<td>Mesh2D</td>
</tr>
<tr>
<td>Modal</td>
</tr>
<tr>
<td>ModalBar</td>
</tr>
<tr>
<td>Plucked</td>
</tr>
<tr>
<td>PluckTwo</td>
</tr>
<tr>
<td>Mandolin</td>
</tr>
<tr>
<td>Resonate</td>
</tr>
<tr>
<td>Sampler</td>
</tr>
<tr>
<td>Moog</td>
</tr>
<tr>
<td>Saxofony</td>
</tr>
<tr>
<td>Shakers</td>
</tr>
<tr>
<td>Simple</td>
</tr>
<tr>
<td>Sitar</td>
</tr>
<tr>
<td>StifKarp</td>
</tr>
<tr>
<td>VoicForm</td>
</tr>
<tr>
<td>Whistle</td>
</tr>
<tr>
<td>JetTabl</td>
</tr>
<tr>
<td>Messager</td>
</tr>
<tr>
<td>Modulate</td>
</tr>
<tr>
<td>Noise</td>
</tr>
<tr>
<td>SubNoise</td>
</tr>
<tr>
<td>PitShift</td>
</tr>
<tr>
<td>ReedTabl</td>
</tr>
<tr>
<td>Reverb</td>
</tr>
<tr>
<td>JCRRev</td>
</tr>
<tr>
<td>NRev</td>
</tr>
<tr>
<td>PRCRev</td>
</tr>
<tr>
<td>RtDuplex</td>
</tr>
<tr>
<td>RtMidi</td>
</tr>
<tr>
<td>SingWave</td>
</tr>
<tr>
<td>SKINI</td>
</tr>
</tbody>
</table>
1.1 STK Class Hierarchy

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>152</td>
</tr>
<tr>
<td>Table</td>
<td>167</td>
</tr>
<tr>
<td>Thread</td>
<td>175</td>
</tr>
<tr>
<td>Voicer</td>
<td>187</td>
</tr>
<tr>
<td>WvIn</td>
<td>199</td>
</tr>
<tr>
<td>RtWvIn</td>
<td>129</td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>169</td>
</tr>
<tr>
<td>WaveLoop</td>
<td>193</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>RtWvOut</td>
<td>132</td>
</tr>
<tr>
<td>TcpWvOut</td>
<td>172</td>
</tr>
<tr>
<td>StkError</td>
<td>164</td>
</tr>
<tr>
<td>Vector3D</td>
<td>185</td>
</tr>
</tbody>
</table>
Chapter 2

STK Compound Index

2.1 STK Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

- **ADSR** (STK ADSR envelope class) ........................................... 11
- **BandedWG** (Banded waveguide modeling class) ........................ 14
- **BeeThree** (STK Hammond-oid organ FM synthesis instrument) ... 16
- **BiQuad** (STK biquad (two-pole, two-zero) filter class) .............. 18
- **BlowBotl** (STK blown bottle instrument class) ............................ 21
- **BlowHole** (STK clarinet physical model with one register hole and one tonehole) ................................................................. 23
- **Bowed** (STK bowed string instrument class) ............................... 25
- **BowTabl** (STK bowed string table class) .................................... 27
- **Brass** (STK simple brass instrument class) ................................. 29
- **Chorus** (STK chorus effect class) ............................................. 31
- **Clarinet** (STK clarinet physical model class) .............................. 33
- **Delay** (STK non-interpolating delay line class) ............................ 35
- **DelayA** (STK allpass interpolating delay line class) ................. 38
- **DelayL** (STK linear interpolating delay line class) ..................... 40
- **Drummer** (STK drum sample player class) ................................. 42
- **Echo** (STK echo effect class) .................................................. 44
- **Envelope** (STK envelope base class) ......................................... 46
- **Filter** (STK filter class) ....................................................... 48
- **Flute** (STK flute physical model class) ..................................... 51
- **FM** (STK abstract FM synthesis base class) .............................. 53
- **FMVoices** (STK singing FM synthesis instrument) ................. 56
- **FormSweep** (STK sweepable formant filter class) ..................... 58
- **HevyMetl** (STK heavy metal FM synthesis instrument) ............ 61
- **Instrmnt** (STK instrument abstract base class) .......................... 63
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCRev</td>
<td>John Chowning’s reverberator class</td>
</tr>
<tr>
<td>JetTabl</td>
<td>STK jet table class</td>
</tr>
<tr>
<td>Mandolin</td>
<td>STK mandolin instrument model class</td>
</tr>
<tr>
<td>Mesh2D</td>
<td>Two-dimensional rectilinear waveguide mesh class</td>
</tr>
<tr>
<td>Messenger</td>
<td>STK input control message parser</td>
</tr>
<tr>
<td>Modal</td>
<td>STK resonance model instrument</td>
</tr>
<tr>
<td>ModalBar</td>
<td>STK resonant bar instrument class</td>
</tr>
<tr>
<td>Modulate</td>
<td>STK periodic/random modulator</td>
</tr>
<tr>
<td>Moog</td>
<td>STK moog-like swept filter sampling synthesis class</td>
</tr>
<tr>
<td>Noise</td>
<td>STK noise generator</td>
</tr>
<tr>
<td>NRev</td>
<td>CCRMA’s NRev reverberator class</td>
</tr>
<tr>
<td>OnePole</td>
<td>STK one-pole filter class</td>
</tr>
<tr>
<td>OneZero</td>
<td>STK one-zero filter class</td>
</tr>
<tr>
<td>PercFlut</td>
<td>STK percussive flute FM synthesis instrument</td>
</tr>
<tr>
<td>Phonemes</td>
<td>STK phonemes table</td>
</tr>
<tr>
<td>PitShift</td>
<td>STK simple pitch shifter effect class</td>
</tr>
<tr>
<td>Plucked</td>
<td>STK plucked string model class</td>
</tr>
<tr>
<td>PluckTwo</td>
<td>STK enhanced plucked string model class</td>
</tr>
<tr>
<td>PoleZero</td>
<td>STK one-pole, one-zero filter class</td>
</tr>
<tr>
<td>PRCRev</td>
<td>Perry’s simple reverberator class</td>
</tr>
<tr>
<td>ReedTabl</td>
<td>STK reed table class</td>
</tr>
<tr>
<td>Resonate</td>
<td>STK noise driven formant filter</td>
</tr>
<tr>
<td>Reverb</td>
<td>STK abstract reverberator parent class</td>
</tr>
<tr>
<td>Rhodey</td>
<td>STK Fender Rhodes electric piano FM synthesis instrument</td>
</tr>
<tr>
<td>RtAudio</td>
<td>Realtime audio i/o C++ class</td>
</tr>
<tr>
<td>RtAudio::RTAUDIO DEVICE</td>
<td>The public device information</td>
</tr>
<tr>
<td>RtDuplex</td>
<td>STK realtime audio input/output class</td>
</tr>
<tr>
<td>RtError</td>
<td>Exception handling class for [RtAudio]</td>
</tr>
<tr>
<td>RtMidi</td>
<td>STK realtime MIDI class</td>
</tr>
<tr>
<td>RtWvIn</td>
<td>STK realtime audio input class</td>
</tr>
<tr>
<td>RtWvOut</td>
<td>STK realtime audio output class</td>
</tr>
<tr>
<td>Sampler</td>
<td>STK sampling synthesis abstract base class</td>
</tr>
<tr>
<td>Saxofony</td>
<td>STK faux conical bore reed instrument class</td>
</tr>
<tr>
<td>Shakers</td>
<td>PhISEM and PhOLIES class</td>
</tr>
<tr>
<td>Simple</td>
<td>STK wavetable/noise instrument</td>
</tr>
<tr>
<td>SingWave</td>
<td>STK &quot;singing&quot; looped soundfile class</td>
</tr>
<tr>
<td>Sitar</td>
<td>STK sitar string model class</td>
</tr>
<tr>
<td>SKINI</td>
<td>STK SKINI parsing class</td>
</tr>
<tr>
<td>Socket</td>
<td>STK TCP socket client/server class</td>
</tr>
<tr>
<td>Sphere</td>
<td>STK sphere class</td>
</tr>
<tr>
<td>StiffKarp</td>
<td>STK plucked stiff string instrument</td>
</tr>
<tr>
<td>Stk</td>
<td>STK base class</td>
</tr>
<tr>
<td>StkError</td>
<td>STK error handling class</td>
</tr>
<tr>
<td>SubNoise</td>
<td>STK sub-sampled noise generator</td>
</tr>
<tr>
<td>Table</td>
<td>STK table lookup class</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>STK internet streaming input class</td>
</tr>
<tr>
<td>TcpWvOut</td>
<td>STK internet streaming output class</td>
</tr>
<tr>
<td>Thread</td>
<td>STK thread class</td>
</tr>
<tr>
<td>TubeBell</td>
<td>STK tubular bell (orchestral chime) FM synthesis instrument</td>
</tr>
<tr>
<td>TwoPole</td>
<td>STK two-pole filter class</td>
</tr>
<tr>
<td>TwoZero</td>
<td>STK two-zero filter class</td>
</tr>
<tr>
<td>Vector3D</td>
<td>STK 3D vector class</td>
</tr>
<tr>
<td>Voicer</td>
<td>STK voice manager class</td>
</tr>
<tr>
<td>VoicForm</td>
<td>Four formant synthesis instrument</td>
</tr>
<tr>
<td>WaveLoop</td>
<td>STK waveform oscillator class</td>
</tr>
<tr>
<td>Whistle</td>
<td>STK police/referee whistle instrument class</td>
</tr>
<tr>
<td>Wurley</td>
<td>STK Wurlitzer electric piano FM synthesis instrument</td>
</tr>
<tr>
<td>WvIn</td>
<td>STK audio data input base class</td>
</tr>
<tr>
<td>WvOut</td>
<td>STK audio data output base class</td>
</tr>
</tbody>
</table>
Chapter 3

STK Page Index

3.1 STK Related Pages

Here is a list of all related documentation pages:

- Class Documentation .................................................. 209
- Compiling ................................................................. 210
- Control Input ............................................................. 214
- Download and Release Notes ......................................... 217
- Hello Sine! ................................................................. 222
- General Information ..................................................... 225
- Instruments ............................................................... 228
- Miscellaneous Links ..................................................... 230
- The Mail List ............................................................. 231
- Multi-Channel I/O ......................................................... 232
- Voice Management ....................................................... 233
- Realtime Audio .......................................................... 237
- Synthesis toolKit Instrument Network Interface (SKINI) .... 239
- System Requirements ................................................... 247
- Tutorial ................................................................. 249
- Usage Documentation ................................................... 250
Chapter 4

STK Class Documentation

4.1 ADSR Class Reference

STK ADSR envelope class.

#include <ADSR.h>

Inheritance diagram for ADSR::

```
#include <ADSR.h>

Public Types

- enum { ATTACK, DECAY, SUSTAIN, RELEASE, DONE }
  
Envelope states.

Public Methods

- ADSR (void)
  
Default constructor.
```
• `~ADSR` (void)
  Class destructor.

• `void keyOn` (void)
  Set target = 1, state = ADSR::ATTACK.

• `void keyOff` (void)
  Set target = 0, state = ADSR::RELEASE.

• `void setAttackRate` (MY_FLOAT aRate)
  Set the attack rate.

• `void setDecayRate` (MY_FLOAT aRate)
  Set the decay rate.

• `void setSustainLevel` (MY_FLOAT aLevel)
  Set the sustain level.

• `void setReleaseRate` (MY_FLOAT aRate)
  Set the release rate.

• `void setAttackTime` (MY_FLOAT aTime)
  Set the attack rate based on a time duration.

• `void setDecayTime` (MY_FLOAT aTime)
  Set the decay rate based on a time duration.

• `void setReleaseTime` (MY_FLOAT aTime)
  Set the release rate based on a time duration.

• `void setAllTimes` (MY_FLOAT aTime, MY_FLOAT dTime, MY_FLOAT sLevel, MY_FLOAT rTime)
  Set sustain level and attack, decay, and release state rates based on time durations.

• `void setTarget` (MY_FLOAT aTarget)
  Set the target value.

• `int getState` (void) const
  Return the current envelope state (ATTACK, DECAY, SUSTAIN, RELEASE, DONE).

• `void setValue` (MY_FLOAT aValue)
Set to state = ADSR::SUSTAIN with current and target values of aValue.

- MY_FLOAT tick (void)
  Return one envelope output value.

- MY_FLOAT* tick (MY_FLOAT* vector, unsigned int vectorSize)
  Return vectorSize envelope outputs in vector.

4.1.1 Detailed Description

STK ADSR envelope class.

This subclass implements a traditional ADSR (Attack, Decay, Sustain, Release) envelope. It responds to simple keyOn and keyOff messages, keeping track of its state. The state = ADSR::DONE after the envelope value reaches 0.0 in the ADSR::RELEASE state.


The documentation for this class was generated from the following file:

- ADSR.h
4.2 BandedWG Class Reference

Banded waveguide modeling class.

#include <BandedWG.h>

Inheritance diagram for BandedWG:

```
  Stk
  |   
  v   
Instrmnt
  |   
  v   
BandedWG
```

Public Methods

- **BandedWG()**
  
  Class constructor.

- **~BandedWG()**
  
  Class destructor.

- **void clear()**
  
  Reset and clear all internal state.

- **void setStrikePosition(MY_FLOAT position)**
  
  Set strike position (0.0 - 1.0).

- **void setPreset(int preset)**
  
  Select a preset.

- **void setFrequency(MY_FLOAT frequency)**
  
  Set instrument parameters for a particular frequency.

- **void startBowing(MY_FLOAT amplitude, MY_FLOAT rate)**
  
  Apply bow velocity/pressure to instrument with given amplitude and rate of increase.

- **void stopBowing(MY_FLOAT rate)**
  
  Decrease bow velocity/breath pressure with given rate of decrease.
4.2 BandedWG Class Reference

- **void pluck**(MY_FLOAT amp)
  
  *Pluck the instrument with given amplitude.*

- **void noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  
  *Start a note with the given frequency and amplitude.*

- **void noteOff**(MY_FLOAT amplitude)
  
  *Stop a note with the given amplitude (speed of decay).*

- **MY_FLOAT tick()**
  
  *Compute one output sample.*

- **void controlChange**(int number, MY_FLOAT value)
  
  *Perform the control change specified by number and value (0.0 - 128.0).*

4.2.1 Detailed Description

Banded waveguide modeling class.

This class uses banded waveguide techniques to model a variety of sounds, including bowed bars, glasses, and bowls. For more information, see Essl, G. and Cook, P. "Banded Waveguides: Towards Physical Modelling of Bar Percussion Instruments", Proceedings of the 1999 International Computer Music Conference.

Control Change Numbers:

- Bow Pressure = 2
- Bow Motion = 4
- Strike Position = 8 (not implemented)
- Vibrato Frequency = 11
- Gain = 1
- Bow Velocity = 128
- Set Striking = 64
- Instrument Presets = 16
  
  - Uniform Bar = 0
  - Tuned Bar = 1
  - Glass Harmonica = 2
  - Tibetan Bowl = 3


The documentation for this class was generated from the following file:

- **BandedWG.h**
4.3 BeeThree Class Reference

STK Hammond-oid organ FM synthesis instrument.

```
#include <BeeThree.h>
```

Inheritance diagram for BeeThree::

```
Inheritance diagram for BeeThree::

```Stk
```
```
Instrmnt
```
```
FM
```
```
BeeThree
```

Public Methods

- **BeeThree()**
  - *Class constructor.*

- **~BeeThree()**
  - *Class destructor.*

- **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
  - *Start a note with the given frequency and amplitude.*

- **MY_FLOAT tick()**
  - *Compute one output sample.*

4.3.1 Detailed Description

STK Hammond-oid organ FM synthesis instrument.

This class implements a simple 4 operator topology, also referred to as algorithm 8 of the TX81Z.

Algorithm 8 is:

```
1  --
2  "\|
  +-> Out
```
Control Change Numbers:

- Operator 4 (feedback) Gain = 2
- Operator 3 Gain = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- BeeThree.h
4.4 BiQuad Class Reference

STK biquad (two-pole, two-zero) filter class.

#include <BiQuad.h>

Inheritance diagram for BiQuad::

```
Stk
  ↓
Filter
  ↓
BiQuad
  ↓
FormSwep
```

Public Methods

- **BiQuad()**
  Default constructor creates a second-order pass-through filter.

- **virtual ~BiQuad()**
  Class destructor.

- **void clear()**
  Clears all internal states of the filter.

- **void setB0(MY_FLOAT b0)**
  Set the b[0] coefficient value.

- **void setB1(MY_FLOAT b1)**
  Set the b[1] coefficient value.

- **void setB2(MY_FLOAT b2)**
  Set the b[2] coefficient value.

- **void setA1(MY_FLOAT a1)**
  Set the a[1] coefficient value.

- **void setA2(MY_FLOAT a2)**
  Set the a[2] coefficient value.
4.4 BiQuad Class Reference

Set the \( a[2] \) coefficient value.

- void \texttt{setResonance} (MY\_FLOAT frequency, MY\_FLOAT radius, bool normalize=FALSE)  
  Sets the filter coefficients for a resonance at frequency (in Hz).

- void \texttt{setNotch} (MY\_FLOAT frequency, MY\_FLOAT radius)  
  Set the filter coefficients for a notch at frequency (in Hz).

- void \texttt{setEqualGainZeroes} ()  
  Sets the filter zeroes for equal resonance gain.

- void \texttt{setGain} (MY\_FLOAT theGain)  
  Set the filter gain.

- MY\_FLOAT \texttt{getGain} (void) const  
  Return the current filter gain.

- MY\_FLOAT \texttt{lastOut} (void) const  
  Return the last computed output value.

- MY\_FLOAT * \texttt{tick} (MY\_FLOAT *vector, unsigned int vectorSize)  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.4.1 Detailed Description

STK biquad (two-pole, two-zero) filter class.

This protected \texttt{Filter} subclass implements a two-pole, two-zero digital filter. A method is provided for creating a resonance in the frequency response while maintaining a constant filter gain.


4.4.2 Member Function Documentation

4.4.2.1 void \texttt{BiQuad::setResonance} (MY\_FLOAT frequency, MY\_FLOAT radius, bool normalize = FALSE)  

Sets the filter coefficients for a resonance at frequency (in Hz).
This method determines the filter coefficients corresponding to two complex-conjugate poles with the given frequency (in Hz) and radius from the z-plane origin. If normalize is true, the filter zeros are placed at $z = 1, z = -1$, and the coefficients are then normalized to produce a constant unity peak gain (independent of the filter gain parameter). The resulting filter frequency response has a resonance at the given frequency. The closer the poles are to the unit-circle (radius close to one), the narrower the resulting resonance width.

4.4.2.2  void BiQuad::setNotch (MY_FLOAT frequency,  
          MY_FLOAT radius)

Set the filter coefficients for a notch at frequency (in Hz).

This method determines the filter coefficients corresponding to two complex-conjugate zeros with the given frequency (in Hz) and radius from the z-plane origin. No filter normalization is attempted.

4.4.2.3  void BiQuad::setEqualGainZeroes ()

Sets the filter zeroes for equal resonance gain.

When using the filter as a resonator, zeroes places at $z = 1, z = -1$ will result in a constant gain at resonance of $1 / (1 - R)$, where $R$ is the pole radius setting.

4.4.2.4  void BiQuad::setGain (MY_FLOAT theGain)  [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented from Filter

The documentation for this class was generated from the following file:

- BiQuad.h
4.5 BlowBotl Class Reference

STK blown bottle instrument class.
#include <BlowBotl.h>

Inheritance diagram for BlowBotl:

```
   Stk
      ↓
   Instrmnt
      ↓
BlowBotl
```

Public Methods

- **BlowBotl ()**
  
  *Class constructor.*

- **~BlowBotl ()**
  
  *Class destructor.*

- **void clear ()**
  
  *Reset and clear all internal state.*

- **void setFrequency (MY_FLOAT frequency)**
  
  *Set instrument parameters for a particular frequency.*

- **void startBlowing (MY_FLOAT amplitude, MY_FLOAT rate)**
  
  *Apply breath velocity to instrument with given amplitude and rate of increase.*

- **void stopBlowing (MY_FLOAT rate)**
  
  *Decrease breath velocity with given rate of decrease.*

- **void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)**
  
  *Start a note with the given frequency and amplitude.*

- **void noteOff (MY_FLOAT amplitude)**
  
  *Stop a note with the given amplitude (speed of decay).*

- **MY_FLOAT tick ()**
Compute one output sample.

- void controlChange(int number, MY_FLOAT value)
  
  *Perform the control change specified by number and value (0.0 - 128.0).*

### 4.5.1 Detailed Description

STK blown bottle instrument class.

This class implements a helmholtz resonator (biquad filter) with a polynomial jet excitation (a la Cook).

Control Change Numbers:

- **Noise** Gain = 4
- **Vibrato Frequency** = 11
- **Vibrato Gain** = 1
- **Volume** = 128


The documentation for this class was generated from the following file:

- **BlowBotl.h**
4.6 BlowHole Class Reference

STK clarinet physical model with one register hole and one tonehole.

```cpp
#include <BlowHole.h>
```

Inheritance diagram for BlowHole::

```
Inheritance diagram for BlowHole:

```

Public Methods

- **BlowHole**(MY_FLOAT lowestFrequency)
  
  *Class constructor.*

- **~BlowHole**()
  
  *Class destructor.*

- void **clear**()
  
  *Reset and clear all internal state.*

- void **setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **setTonehole**(MY_FLOAT newValue)
  
  *Set the tonehole state (0.0 = closed, 1.0 = fully open).*

- void **setVent**(MY_FLOAT newValue)
  
  *Set the register hole state (0.0 = closed, 1.0 = fully open).*

- void **startBlowing**(MY_FLOAT amplitude, MY_FLOAT rate)
  
  *Apply breath pressure to instrument with given amplitude and rate of increase.*

- void **stopBlowing**(MY_FLOAT rate)
  
  *Decrease breath pressure with given rate of decrease.*
• void **noteOn** (MY_FLOAT frequency, MY_FLOAT amplitude)
  
  *Start a note with the given frequency and amplitude.*

• void **noteOff** (MY_FLOAT amplitude)

  *Stop a note with the given amplitude (speed of decay).*

• MY_FLOAT **tick**()

  *Compute one output sample.*

• void **controlChange** (int number, MY_FLOAT value)

  *Perform the control change specified by number and value (0.0 - 128.0).*

### 4.6.1 Detailed Description

STK clarinet physical model with one register hole and one tonehole.

This class is based on the clarinet model, with the addition of a two-port register hole and a three-port dynamic tonehole implementation, as discussed by Scavone and Cook (1998).

In this implementation, the distances between the reed/register hole and tonehole/bell are fixed. As a result, both the tonehole and register hole will have variable influence on the playing frequency, which is dependent on the length of the air column. In addition, the highest playing frequency is limited by these fixed lengths.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

• Reed Stiffness = 2

• Noise Gain = 4

• Tonehole State = 11

• Register State = 1

• Breath Pressure = 128


The documentation for this class was generated from the following file:

• **BlowHole.h**
4.7 Bowed Class Reference

STK bowed string instrument class.

#include <Bowed.h>

Inheritance diagram for Bowed::

```
  Stk
  Instrmnt
  Bowed
```

Public Methods

- **Bowed**(MY_FLOAT lowestFrequency)
  
  Class constructor, taking the lowest desired playing frequency.

- **~Bowed**()
  
  Class destructor.

- void **clear**()
  
  Reset and clear all internal state.

- void **setFrequency**(MY_FLOAT frequency)
  
  Set instrument parameters for a particular frequency.

- void **setVibrato**(MY_FLOAT gain)
  
  Set vibrato gain.

- void **startBowing**(MY_FLOAT amplitude, MY_FLOAT rate)
  
  Apply breath pressure to instrument with given amplitude and rate of increase.

- void **stopBowing**(MY_FLOAT rate)
  
  Decrease breath pressure with given rate of decrease.

- void **noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  
  Start a note with the given frequency and amplitude.
• void noteOff (MY_FLOAT amplitude)
  Stop a note with the given amplitude (speed of decay).

• MY_FLOAT tick()
  Compute one output sample.

• void controlChange (int number, MY_FLOAT value)
  Perform the control change specified by number and value (0.0 - 128.0).

4.7.1 Detailed Description

STK bowed string instrument class.

This class implements a bowed string model, a la Smith (1986), after McIntyre, Schumacher, Woodhouse (1983).

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

• Bow Pressure = 2
• Bow Position = 4
• Vibrato Frequency = 11
• Vibrato Gain = 1
• Volume = 128


The documentation for this class was generated from the following file:

• Bowed.h
4.8 BowTabl Class Reference

STK bowed string table class.

#include <BowTabl.h>

Inheritance diagram for BowTabl::

```
  Stk
   
BowTabl
```

Public Methods

- **BowTabl ()**
  
  Default constructor.

- **~BowTabl ()**
  
  Class destructor.

- void setOffset (MY_FLOAT aValue)
  
  Set the table offset value.

- void setSlope (MY_FLOAT aValue)
  
  Set the table slope value.

- MY_FLOAT lastOut (void) const
  
  Return the last output value.

- MY_FLOAT *tick (const MY_FLOAT input)
  
  Return the function value for input.

- MY_FLOAT *tick (MY_FLOAT *vector, unsigned int vectorSize)
  
  Take vectorSize inputs and return the corresponding function values in vector.

4.8.1 Detailed Description

STK bowed string table class.
This class implements a simple bowed string non-linear function, as described by Smith (1986).

4.8.2 Member Function Documentation

4.8.2.1 void BowTabl::setOffset (MY_FLOAT aValue)

Set the table offset value.
The table offset is a bias which controls the symmetry of the friction. If you want the friction to vary with direction, use a non-zero value for the offset. The default value is zero.

4.8.2.2 void BowTabl::setSlope (MY_FLOAT aValue)

Set the table slope value.
The table slope controls the width of the friction pulse, which is related to bow force.

4.8.2.3 MY_FLOAT BowTabl::tick (const MY_FLOAT input)

Return the function value for input.
The function input represents differential string-to-bow velocity.
The documentation for this class was generated from the following file:

- BowTabl.h
4.9 Brass Class Reference

STK simple brass instrument class.

```cpp
#include <Brass.h>
```

Inheritance diagram for Brass:

```
    Stk
     |
  Instrmnt
     |
     Brass
```

Public Methods

- **Brass**(MY_FLOAT lowestFrequency)
  
  *Class constructor, taking the lowest desired playing frequency.*

- **~Brass**( )
  
  *Class destructor.*

- void **clear**( )
  
  *Reset and clear all internal state.*

- void **setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **setLip**(MY_FLOAT frequency)
  
  *Set the lips frequency.*

- void **startBlowing**(MY_FLOAT amplitude, MY_FLOAT rate)
  
  *Apply breath pressure to instrument with given amplitude and rate of increase.*

- void **stopBlowing**(MY_FLOAT rate)
  
  *Decrease breath pressure with given rate of decrease.*

- void **noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  
  *Start a note with the given frequency and amplitude.*
• **void noteOff (MY_FLOAT amplitude)**
  
  Stop a note with the given amplitude (speed of decay).

• **MY_FLOAT tick ()**
  
  Compute one output sample.

• **void controlChange (int number, MY_FLOAT value)**
  
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.9.1 Detailed Description

STK simple brass instrument class.

This class implements a simple brass instrument waveguide model, a la Cook (TBone, HosePlayer).

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Lip Tension = 2
- Slide Length = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Volume = 128


The documentation for this class was generated from the following file:

• **Brass.h**
4.10 Chorus Class Reference

STK chorus effect class.

```c
#include <Chorus.h>
```

Inheritance diagram for Chorus::

```
Stk
    Chorus
```

Public Methods

- `Chorus(MY_FLOAT baseDelay)`
  
  *Class constructor, taking the longest desired delay length.*

- `~Chorus()`
  
  *Class destructor.*

- `void clear()`
  
  *Reset and clear all internal state.*

- `void setModDepth(MY_FLOAT depth)`
  
  *Set modulation depth.*

- `void setModFrequency(MY_FLOAT frequency)`
  
  *Set modulation frequency.*

- `void setEffectMix(MY_FLOAT mix)`
  
  *Set the mixture of input and processed levels in the output (0.0 = input only, 1.0 = processed only).*

- `MY_FLOAT lastOut() const`
  
  *Return the last output value.*

- `MY_FLOAT lastOutLeft() const`
  
  *Return the last left output value.*

- `MY_FLOAT lastOutRight() const`
  
  *Return the last right output value.*
• MY_FLOAT tick (MY_FLOAT input)
  Compute one output sample.

• MY_FLOAT* tick (MY_FLOAT* vector, unsigned int vectorSize)
  Take vectorSize inputs, compute the same number of outputs and return them in vector.

4.10.1 Detailed Description

STK chorus effect class.

This class implements a chorus effect.


The documentation for this class was generated from the following file:

• Chorus.h
4.11 Clarinet Class Reference

STK clarinet physical model class.

```cpp
#include <Clarinet.h>
```

Inheritance diagram for `Clarinet`:

```
Stk
    └── Instrmnt
        └── Clarinet
```

Public Methods

- `Clarinet(MY_FLOAT lowestFrequency)`
  
  Class constructor, taking the lowest desired playing frequency.

- `~Clarinet()`
  
  Class destructor.

- `void clear()`
  
  Reset and clear all internal state.

- `void setFrequency(MY_FLOAT frequency)`
  
  Set instrument parameters for a particular frequency.

- `void startBlowing(MY_FLOAT amplitude, MY_FLOAT rate)`
  
  Apply breath pressure to instrument with given amplitude and rate of increase.

- `void stopBlowing(MY_FLOAT rate)`
  
  Decrease breath pressure with given rate of decrease.

- `void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)`
  
  Start a note with the given frequency and amplitude.

- `void noteOff(MY_FLOAT amplitude)`
  
  Stop a note with the given amplitude (speed of decay).
• MY_FLOAT \texttt{tick} ()
  
  \textit{Compute one output sample.}

• \texttt{void controlChange (int number, MYFLOAT value)}

  \textit{Perform the control change specified by number and value (0.0 - 128.0).}

4.11.1 Detailed Description

STK clarinet physical model class.

This class implements a simple clarinet physical model, as discussed by Smith (1986), McIntyre, Schumacher, Woodhouse (1983), and others.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

• Reed Stiffness = 2
• Noise Gain = 4
• Vibrato Frequency = 11
• Vibrato Gain = 1
• Breath Pressure = 128


The documentation for this class was generated from the following file:

• \texttt{Clarinet.h}
4.12 Delay Class Reference

STK non-interpolating delay line class.

```
#include <Delay.h>
```

Inheritance diagram for Delay::

```
  Stk
   |
   Filter
   |
   Delay
   |
 DelayA DelayL
```

Public Methods

- **Delay()**
  
  Default constructor creates a delay-line with maximum length of 4095 samples and zero delay.

- **Delay(long theDelay, long maxDelay)**
  
  Overloaded constructor which specifies the current and maximum delay-line lengths.

- **virtual ~Delay()**
  
  Class destructor.

- **void clear()**
  
  Clears the internal state of the delay line.

- **void setDelay(long theDelay)**
  
  Set the delay-line length.

- **long getDelay(void) const**
  
  Return the current delay-line length.

- **MY_FLOAT energy(void) const**
  
  Calculate and return the signal energy in the delay-line.
• MY_FLOAT contentsAt (long tapDelay) const
  Return the value at tapDelay samples from the delay-line input.

• MY_FLOAT lastOut (void) const
  Return the last computed output value.

• virtual MY_FLOAT nextOut (void) const
  Return the value which will be output by the next call to tick).

• virtual MY_FLOAT * tick (MY_FLOAT * vector, unsigned int vectorSize)
  Input vectorSize samples to the delay-line and return an equal number of
  outputs in vector.

4.12.1 Detailed Description

STK non-interpolating delay line class.

This protected Filter subclass implements a non-interpolating digital delay-line.
A fixed maximum length of 4095 and a delay of zero is set using the default
constructor. Alternatively, the delay and maximum length can be set during
instantiation with an overloaded constructor.

A non-interpolating delay line is typically used in fixed delay-length applications,
such as for reverberation.


4.12.2 Member Function Documentation

4.12.2.1 void Delay::setDelay (long theDelay)

Set the delay-line length.

The valid range for theDelay is from 0 to the maximum delay-line length.

4.12.2.2 MY_FLOAT Delay::contentsAt (long tapDelay) const

Return the value at tapDelay samples from the delay-line input.

The valid range for tapDelay is 1 to the delay-line length.
4.12 Delay Class Reference

4.12.2.3 MY_FLOAT Delay::nextOut (void) const [virtual]

Return the value which will be output by the next call to [tick].
This method is valid only for delay settings greater than zero!
The documentation for this class was generated from the following file:

- Delay.h
4.13 DelayA Class Reference

STK allpass interpolating delay line class.

```
#include <DelayA.h>
```

Inheritance diagram for DelayA:

```
  Stk
   ↓
  Filter
   ↓
  Delay
   ↓
DelayA
```

Public Methods

- **DelayA()**
  
  Default constructor creates a delay-line with maximum length of 4095 samples and zero delay.

- **DelayA(MY_FLOAT theDelay, long maxDelay)**
  
  Overloaded constructor which specifies the current and maximum delay-line lengths.

- **~DelayA()**
  
  Class destructor.

- **void clear()**
  
  Clears the internal state of the delay line.

- **void setDelay(MY_FLOAT theDelay)**
  
  Set the delay-line length.

- **MY_FLOAT getDelay(void)**
  
  Return the current delay-line length.

- **MY_FLOAT nextOut(void)**
  
  Return the value which will be output by the next call to tick().
4.13 DelayA Class Reference

- MY_FLOAT tick (MY_FLOAT sample)
  
  *Input one sample to the delay-line and return one output.*

4.13.1 Detailed Description

STK allpass interpolating delay line class.

This Delay subclass implements a fractional-length digital delay-line using a first-order allpass filter. A fixed maximum length of 4095 and a delay of 0.5 is set using the default constructor. Alternatively, the delay and maximum length can be set during instantiation with an overloaded constructor.

An allpass filter has unity magnitude gain but variable phase delay properties, making it useful in achieving fractional delays without affecting a signal's frequency magnitude response. In order to achieve a maximally flat phase delay response, the minimum delay possible in this implementation is limited to a value of 0.5.


4.13.2 Member Function Documentation

4.13.2.1 void DelayA::setDelay (MY_FLOAT theDelay)

Set the delay-line length.

The valid range for theDelay is from 0.5 to the maximum delay-line length.

4.13.2.2 MY_FLOAT DelayA::nextOut (void)

Return the value which will be output by the next call to tick().

This method is valid only for delay settings greater than zero!

The documentation for this class was generated from the following file:

- DelayA.h
4.14 DelayL Class Reference

STK linear interpolating delay line class.

#include <DelayL.h>

Inheritance diagram for DelayL:

```
  Stk
   ↓
  Filter
   ↓
  Delay
   ↓
DelayL
```

Public Methods

- **DelayL()**
  Default constructor creates a delay-line with maximum length of 4095 samples and zero delay.

- **DelayL(MY_FLOAT theDelay, long maxDelay)**
  Overloaded constructor which specifies the current and maximum delay-line lengths.

- **~DelayL()**
  Class destructor.

- **void setDelay(MY_FLOAT theDelay)**
  Set the delay-line length.

- **MY_FLOAT getDelay(void) const**
  Return the current delay-line length.

- **MY_FLOAT nextOut(void)**
  Return the value which will be output by the next call to tick().

- **MY_FLOAT tick(MY_FLOAT sample)**
  Input one sample to the delay-line and return one output.
4.14 DelayL Class Reference

4.14.1 Detailed Description

STK linear interpolating delay line class.

This DelayL subclass implements a fractional-length digital delay-line using first-order linear interpolation. A fixed maximum length of 4095 and a delay of zero is set using the default constructor. Alternatively, the delay and maximum length can be set during instantiation with an overloaded constructor.

Linear interpolation is an efficient technique for achieving fractional delay lengths, though it does introduce high-frequency signal attenuation to varying degrees depending on the fractional delay setting. The use of higher order Lagrange interpolators can typically improve (minimize) this attenuation characteristic.


4.14.2 Member Function Documentation

4.14.2.1 void DelayL::setDelay (MY_FLOAT theDelay)

Set the delay-line length.

The valid range for theDelay is from 0 to the maximum delay-line length.

4.14.2.2 MY_FLOAT DelayL::nextOut (void)

Return the value which will be output by the next call to tick.

This method is valid only for delay settings greater than zero!

The documentation for this class was generated from the following file:

- DelayL.h
4.15 Drummer Class Reference

STK drum sample player class.
#include <Drummer.h>

Inheritance diagram for Drummer:

```
Stk
    ↓
Instrmnt
    ↓
Drummer
```

Public Methods

- `Drummer()`  
  Class constructor.

- `~Drummer()`  
  Class destructor.

- void `noteOn` (MY_FLOAT instrument, MY_FLOAT amplitude)
  Start a note with the given drum type and amplitude.

- void `noteOff` (MY_FLOAT amplitude)
  Stop a note with the given amplitude (speed of decay).

- MY_FLOAT `tick()`  
  Compute one output sample.

4.15.1 Detailed Description

STK drum sample player class.

This class implements a drum sampling synthesizer using WvIn objects and one-pole filters. The drum rawwave files are sampled at 22050 Hz, but will be appropriately interpolated for other sample rates. You can specify the maximum polyphony (maximum number of simultaneous voices) via a define in the Drummer.h.

4.15.2 Member Function Documentation

4.15.2.1 void Drummer::noteOn (MY_FLOAT instrument, 
MY_FLOAT amplitude) [virtual]

Start a note with the given drum type and amplitude.
Use general MIDI drum instrument numbers, converted to frequency values as 
if MIDI note numbers, to select a particular instrument.
Reimplemented from Instrmnt.
The documentation for this class was generated from the following file:

- Drummer.h
4.16 Echo Class Reference

STK echo effect class.

#include <Echo.h>

Inheritance diagram for Echo:

```
    Stk
     |
     v
    Echo
```

Public Methods

- **Echo (MY_FLOAT longestDelay)**
  
  *Class constructor, taking the longest desired delay length.*

- **~Echo()**
  
  *Class destructor.*

- **void clear()**
  
  *Reset and clear all internal state.*

- **void setDelay (MY_FLOAT delay)**
  
  *Set the delay line length in samples.*

- **void setEffectMix (MY_FLOAT mix)**
  
  *Set the mixture of input and processed levels in the output (0.0 = input only, 1.0 = processed only).*

- **MY_FLOAT lastOut() const**
  
  *Return the last output value.*

- **MY_FLOAT* tick (MY_FLOAT* vector, unsigned int vectorSize)**
  
  *Input vectorSize samples to the filter and return an equal number of outputs in vector.*
4.16 Echo Class Reference

4.16.1 Detailed Description

STK echo effect class.

This class implements an echo effect.


The documentation for this class was generated from the following file:

- [Echo.h]
4.17 Envelope Class Reference

STK envelope base class.

```cpp
#include <Envelope.h>
```

Inheritance diagram for Envelope::

```
Stk

Envelope

ADSR
```

Public Methods

- `Envelope (void)`
  
  Default constructor.

- virtual `~Envelope (void)`
  
  Class destructor.

- virtual void `keyOn (void)`
  
  Set target = 1.

- virtual void `keyOff (void)`
  
  Set target = 0.

- void `setRate (MY_FLOAT aRate)`
  
  Set the rate.

- void `setTime (MY_FLOAT aTime)`
  
  Set the rate based on a time duration.

- virtual void `setTarget (MY_FLOAT aTarget)`
  
  Set the target value.

- virtual void `setValue (MY_FLOAT aValue)`
  
  Set current and target values to aValue.

- virtual int `getState (void) const`
4.17 Envelope Class Reference

Return the current envelope state (0 = at target, 1 otherwise).

- virtual MY_FLOAT tick (void)
  Return one envelope output value.

- virtual MY_FLOAT* tick (MY_FLOAT* vector, unsigned int vectorSize)
  Return vectorSize envelope outputs in vector.

- MY_FLOAT lastOut (void) const
  Return the last computed output value.

4.17.1 Detailed Description

STK envelope base class.

This class implements a simple envelope generator which is capable of ramping to a target value by a specified rate. It also responds to simple keyOn and keyOff messages, ramping to 1.0 on keyOn and to 0.0 on keyOff.


The documentation for this class was generated from the following file:

- Envelope.h
4.18 Filter Class Reference

STK filter class.

#include <Filter.h>

Inheritance diagram for Filter:

```
Stk
  |
  ↓
Filter
  |
BiQuad
  |
Delay
  |   OnePole
  |   OneZero
  |   PoleZero
  |   TwoPole
  |   TwoZero
  |
FormSwep
  |
DelayA
  |
DelayL
```

Public Methods

- **Filter**(void)
  
  Default constructor creates a zero-order pass-through "filter".

- **Filter**(int nb, MY_FLOAT *bCoefficients, int na, MY_FLOAT *aCoefficients)
  
  Overloaded constructor which takes filter coefficients.

- virtual **~Filter**(void)
  
  Class destructor.

- **clear**(void)
  
  Clears all internal states of the filter.

- void **setCoefficients**(int nb, MY_FLOAT *bCoefficients, int na, MY_FLOAT *aCoefficients)
  
  Set filter coefficients.

- void **setNumerator**(int nb, MY_FLOAT *bCoefficients)
  
  Set numerator coefficients.

- void **setDenominator**(int na, MY_FLOAT *aCoefficients)
  
  Set denominator coefficients.

- virtual void **setGain**(MY_FLOAT theGain)
4.18 Filter Class Reference

Set the filter gain.

- virtual MY_FLOAT \texttt{getGain} (void) const
  
  Return the current filter gain.

- virtual MY_FLOAT \texttt{lastOut} (void) const
  
  Return the last computed output value.

- virtual MY_FLOAT \texttt{tick} (MY_FLOAT sample)
  
  Input one sample to the filter and return one output.

- virtual MY_FLOAT* \texttt{tick} (MY_FLOAT* vector, unsigned int vectorSize)
  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.18.1 Detailed Description

STK filter class.

This class implements a generic structure which can be used to create a wide range of filters. It can function independently or be subclassed to provide more specific controls based on a particular filter type.

In particular, this class implements the standard difference equation:

\[ a[0] \cdot y[n] = b[0] \cdot x[n] + \ldots + b[nb] \cdot x[n-nb] - a[1] \cdot y[n-1] - \ldots - a[na] \cdot y[n-na] \]

If \( a[0] \) is not equal to 1, the filter coefficients are normalized by \( a[0] \).

The \textit{gain} parameter is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0. This structure results in one extra multiply per computed sample, but allows easy control of the overall filter gain.


4.18.2 Constructor & Destructor Documentation

4.18.2.1 Filter::Filter (int \textit{nb}, MY_FLOAT* \textit{bCoefficients}, int \textit{na}, MY_FLOAT* \textit{aCoefficients})

Overloaded constructor which takes filter coefficients.

An \texttt{StkError} can be thrown if either \textit{nb} or \textit{na} is less than one, or if the \( a[0] \) coefficient is equal to zero.
4.18.3 Member Function Documentation

4.18.3.1 void Filter::setCoefficients (int nb, MY_FLOAT * bCoefficients, int na, MY_FLOAT * aCoefficients)

Set filter coefficients.

An StkError can be thrown if either nb or na is less than one, or if the a[0] coefficient is equal to zero. If a[0] is not equal to 1, the filter coefficients are normalized by a[0].

4.18.3.2 void Filter::setNumerator (int nb, MY_FLOAT * bCoefficients)

Set numerator coefficients.

An StkError can be thrown if nb is less than one. Any previously set denominator coefficients are left unaffected. Note that the default constructor sets the single denominator coefficient a[0] to 1.0.

4.18.3.3 void Filter::setDenominator (int na, MY_FLOAT * aCoefficients)

Set denominator coefficients.

An StkError can be thrown if na is less than one or if the a[0] coefficient is equal to zero. Previously set numerator coefficients are unaffected unless a[0] is not equal to 1, in which case all coefficients are normalized by a[0]. Note that the default constructor sets the single numerator coefficient b[0] to 1.0.

4.18.3.4 void Filter::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented in BiQuad, OnePole, OneZero, PoleZero, TwoPole, and TwoZero.

The documentation for this class was generated from the following file:

- Filter.h
4.19 Flute Class Reference

STK flute physical model class.

```cpp
#include <Flute.h>
```

Inheritance diagram for Flute:

```
  Stk
   |
   ↓
Instrmnt
   |
   ↓
Flute
```

Public Methods

- **Flute**(MY_FLOAT lowestFrequency)
  
  Class constructor, taking the lowest desired playing frequency.

- **~Flute**()
  
  Class destructor.

- void **clear**()
  
  Reset and clear all internal state.

- void **setFrequency**(MY_FLOAT frequency)
  
  Set instrument parameters for a particular frequency.

- void **setJetReflection**(MY_FLOAT coefficient)
  
  Set the reflection coefficient for the jet delay (-1.0 - 1.0).

- void **setEndReflection**(MY_FLOAT coefficient)
  
  Set the reflection coefficient for the air column delay (-1.0 - 1.0).

- void **setJetDelay**(MY_FLOAT aRatio)
  
  Set the length of the jet delay in terms of a ratio of jet delay to air column delay lengths.

- void **startBlowing**(MY_FLOAT amplitude, MY_FLOAT rate)
  
  Apply breath velocity to instrument with given amplitude and rate of increase.
• void `stopBlowing` (MY_FLOAT rate)
  *Decrease breath velocity with given rate of decrease.*

• void `noteOn` (MY_FLOAT frequency, MY_FLOAT amplitude)
  *Start a note with the given frequency and amplitude.*

• void `noteOff` (MY_FLOAT amplitude)
  *Stop a note with the given amplitude (speed of decay).*

• MY_FLOAT `tick` ()
  *Compute one output sample.*

• void `controlChange` (int number, MY_FLOAT value)
  *Perform the control change specified by number and value (0.0 - 128.0).*

### 4.19.1 Detailed Description

STK flute physical model class.

This class implements a simple flute physical model, as discussed by Karjalainen, Smith, Waryznyk, etc. The jet model uses a polynomial, a la Cook.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Jet Delay = 2
- Noise Gain = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Breath Pressure = 128


The documentation for this class was generated from the following file:

- `Flute.h`
4.20 FM Class Reference

STK abstract FM synthesis base class.

#include <FM.h>

Inheritance diagram for FM:

```
Stk
  |
  | FM
  |
Instrmnt
  |
  | BeeThree FMVoices HevyMetl PercFlut Rhodey TubeBell Wurley
```

Public Methods

- **FM** (int operators=4)
  
  Class constructor, taking the number of wave/envelope operators to control.

- virtual **~FM** ()
  
  Class destructor.

- void **clear** ()
  
  Reset and clear all wave and envelope states.

- void **loadWaves** (const char **filenames)
  
  Load the rawwave filenames in waves.

- virtual void **setFrequency** (MY_FLOAT frequency)
  
  Set instrument parameters for a particular frequency.

- void **setRatio** (int waveIndex, MY_FLOAT ratio)
  
  Set the frequency ratio for the specified wave.

- void **setGain** (int waveIndex, MY_FLOAT gain)
  
  Set the gain for the specified wave.

- void **setModulationSpeed** (MY_FLOAT mSpeed)
Set the modulation speed in Hz.

- \texttt{void setModulationDepth (MY\_FLOAT mDepth)}
  
  Set the modulation depth.

- \texttt{void setControl1 (MY\_FLOAT cVal)}
  
  Set the value of control1.

- \texttt{void setControl2 (MY\_FLOAT cVal)}
  
  Set the value of control1.

- \texttt{void keyOn ()}
  
  Start envelopes toward "on" targets.

- \texttt{void keyOff ()}
  
  Start envelopes toward "off" targets.

- \texttt{void noteOff (MY\_FLOAT amplitude)}
  
  Stop a note with the given amplitude (speed of decay).

- \texttt{virtual MY\_FLOAT tick ()=0}
  
  Pure virtual function ... must be defined in subclasses.

- \texttt{virtual void controlChange (int number, MY\_FLOAT value)}
  
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.20.1 Detailed Description

STK abstract FM synthesis base class.

This class controls an arbitrary number of waves and envelopes, determined via a constructor argument.

Control Change Numbers:

- Control One = 2
- Control Two = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128
The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- `FM.h`
4.21  FMVoices Class Reference

STK singing FM synthesis instrument.

```c
#include <FMVoices.h>
```

Inheritance diagram for FMVoices:

```
  Stk
   ↓
  Instrmnt
   ↓
   FM
   ↓
  FMVoices
```

Public Methods

- **FMVoices()**
  
  *Class constructor.*

- **~FMVoices()**
  
  *Class destructor.*

- **virtual void setFrequency (MY_FLOAT frequency)**
  
  *Set instrument parameters for a particular frequency.*

- **void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)**
  
  *Start a note with the given frequency and amplitude.*

- **MY_FLOAT tick()**
  
  *Compute one output sample.*

- **virtual void controlChange (int number, MY_FLOAT value)**
  
  *Perform the control change specified by number and value (0.0 - 128.0).*

4.21.1  Detailed Description

STK singing FM synthesis instrument.
This class implements 3 carriers and a common modulator, also referred to as algorithm 6 of the TX81Z.

Algorithm 6 is :

\[
\begin{align*}
\text{\rightarrow} & 1 \text{ \rightarrow} \\
4 \rightarrow & 2 \rightarrow \text{Out} \\
\text{\rightarrow} & 3 \leftarrow
\end{align*}
\]

Control Change Numbers:

- Vowel = 2
- Spectral Tilt = 4
- LFO Speed = 11
- LFO Depth = 1
- \textbf{ADSR} 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- \textbf{FMVoices.h}
4.22 FormSweep Class Reference

STK sweepable formant filter class.

#include <FormSweep.h>

Inheritance diagram for FormSweep:

```
    Stk
     |   
     v
  Filter
     |   
     v
BiQuad
     |   
     v
FormSweep
```

Public Methods

- `FormSweep()`
  Default constructor creates a second-order pass-through filter.

- `~FormSweep()`
  Class destructor.

- `void setResonance(MY_FLOAT aFrequency, MY_FLOAT aRadius)`
  Sets the filter coefficients for a resonance at frequency (in Hz).

- `void setStates(MY_FLOAT aFrequency, MY_FLOAT aRadius, MY_FLOAT aGain=1.0)`
  Set both the current and target resonance parameters.

- `void setTargets(MY_FLOAT aFrequency, MY_FLOAT aRadius, MY_FLOAT aGain=1.0)`
  Set target resonance parameters.

- `void setSweepRate(MY_FLOAT aRate)`
  Set the sweep rate (between 0.0 - 1.0).

- `void setSweepTime(MY_FLOAT aTime)`
  Set the sweep rate in terms of a time value in seconds.
4.22 FormSwep Class Reference

- MY_FLOAT \texttt{tick} (MY_FLOAT sample)
  
  Input one sample to the filter and return one output.

- MY_FLOAT* \texttt{tick} (MY_FLOAT* vector, unsigned int vectorSize)
  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.22.1 Detailed Description

STK sweepable formant filter class.

This public BiQuad filter subclass implements a formant (resonance) which can be "swept" over time from one frequency setting to another. It provides methods for controlling the sweep rate and target frequency.


4.22.2 Member Function Documentation

4.22.2.1 void FormSwep::setResonance (MY_FLOAT aFrequency, MY_FLOAT aRadius)

Sets the filter coefficients for a resonance at frequency (in Hz).

This method determines the filter coefficients corresponding to two complex-conjugate poles with the given frequency (in Hz) and radius from the z-plane origin. The filter zeros are placed at $z = 1$, $z = -1$, and the coefficients are then normalized to produce a constant unity gain (independent of the filter gain parameter). The resulting filter frequency response has a resonance at the given frequency. The closer the poles are to the unit-circle ($radius$ close to one), the narrower the resulting resonance width.

4.22.2.2 void FormSwep::setSweepRate (MY_FLOAT aRate)

Set the sweep rate (between 0.0 - 1.0).

The formant parameters are varied in increments of the sweep rate between their current and target values. A sweep rate of 1.0 will produce an immediate change in resonance parameters from their current values to the target values. A sweep rate of 0.0 will produce no change in resonance parameters.

4.22.2.3 void FormSwep::setSweepTime (MY_FLOAT aTime)

Set the sweep rate in terms of a time value in seconds.
This method adjusts the sweep rate based on a given time for the formant parameters to reach their target values.

The documentation for this class was generated from the following file:

- FormSwep.h
STK heavy metal FM synthesis instrument.

```
#include <HevyMetl.h>
```

Inheritance diagram for HevyMetl::

```
Stk --> Instrmnt --> FM --> HevyMetl
```

### Public Methods

- **HevyMetl()**
  - *Class constructor.*

- **~HevyMetl()**
  - *Class destructor.*

- **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
  - *Start a note with the given frequency and amplitude.*

- **MY_FLOAT tick()**
  - *Compute one output sample.*

### 4.23.1 Detailed Description

STK heavy metal FM synthesis instrument.

This class implements 3 cascade operators with feedback modulation, also referred to as algorithm 3 of the TX81Z.

```
Algorithm 3 is: 4--\ 3-->2-- + --1-->Out
```

Control Change Numbers:

- Total Modulator Index = 2
- Modulator Crossfade = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- HevyMetL.h

---

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
4.24 Instrmnt Class Reference

STK instrument abstract base class.

```cpp
#include <Instrmnt.h>
```

Inheritance diagram for Instrmnt:

```
Class
  Instrmnt
    BandedWG
    Block
    Bowed
    Brass
    Clarinet
    Drummer
    FM
    Modal
    Model
    Pitch
    Pluck
    Resonant
    Sampler
    Simple
    Sitar
    StifKarp
    VoicForm
    Whistle
```

Public Methods

- `Instrmnt()`
  
  Default constructor.

- virtual `~Instrmnt()`
  
  Class destructor.

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
• virtual void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)=0  
  Start a note with the given frequency and amplitude.

• virtual void noteOff (MY_FLOAT amplitude)=0  
  Stop a note with the given amplitude (speed of decay).

• virtual void setFrequency (MY_FLOAT frequency)  
  Set instrument parameters for a particular frequency.

• MY_FLOAT lastOut () const  
  Return the last output value.

• virtual MY_FLOAT tick ()=0  
  Compute one output sample.

• virtual MY_FLOAT * tick (MY_FLOAT *vector, unsigned int vectorSize)  
  Computer vectorSize outputs and return them in vector.

• virtual void controlChange (int number, MY_FLOAT value)  
  Perform the control change specified by number and value (0.0 - 128.0).

4.24.1 Detailed Description

STK instrument abstract base class.
This class provides a common interface for all STK instruments.
The documentation for this class was generated from the following file:

• Instrmnt.h
4.25 JCRev Class Reference

John Chowning’s reverberator class.
#include <JCRev.h>

Inheritance diagram for JCRev:

```
Stk
   ^
   |  \-- Reverb
       \-- JCRev
```

Public Methods

- **JCRev**(MY_FLOAT T60)
  
  *Class constructor taking a T60 decay time argument.*

- **~JCRev**()
  
  *Class destructor.*

- **void clear**()
  
  *Reset and clear all internal state.*

- **MY_FLOAT tick**(MY_FLOAT input)
  
  *Compute one output sample.*

4.25.1 Detailed Description

John Chowning’s reverberator class.

This class is derived from the CLM JCRev function, which is based on the use of networks of simple allpass and comb delay filters. This class implements three series allpass units, followed by four parallel comb filters, and two decorrelation delay lines in parallel at the output.


The documentation for this class was generated from the following file:

- **JCRev.h**
4.26 JetTabl Class Reference

STK jet table class.

```cpp
#include <JetTabl.h>
```

Inheritance diagram for JetTabl::

```
Stk

JetTabl
```

Public Methods

- `JetTabl()`  
  Default constructor.

- `~JetTabl()`  
  Class destructor.

- `MY_FLOAT lastOut() const`  
  Return the last output value.

- `MY_FLOAT tick(MY_FLOAT input)`  
  Return the function value for input.

- `MY_FLOAT* tick(MY_FLOAT* vector, unsigned int vectorSize)`  
  Take vectorSize inputs and return the corresponding function values in vector.

4.26.1 Detailed Description

STK jet table class.

This class implements a flue jet non-linear function, computed by a polynomial calculation. Contrary to the name, this is not a "table".

Consult Fletcher and Rossing, Karjalainen, Cook, and others for more information.


The documentation for this class was generated from the following file:
- JetTabl.h
4.27 Mandolin Class Reference

STK mandolin instrument model class.

```cpp
#include <Mandolin.h>
```

Inheritance diagram for Mandolin::

```
           Stk
            ↓
           Instrmnt
            ↓
          PluckTwo
            ↓
           Mandolin
```

Public Methods

- **Mandolin**(MY\_FLOAT lowestFrequency)
  
  Class constructor, taking the lowest desired playing frequency.

- virtual **~Mandolin**()
  
  Class destructor.

- void **pluck**(MY\_FLOAT amplitude)
  
  Pluck the strings with the given amplitude (0.0 - 1.0) using the current frequency.

- void **pluck**(MY\_FLOAT amplitude, MY\_FLOAT position)
  
  Pluck the strings with the given amplitude (0.0 - 1.0) and position (0.0 - 1.0).

- virtual void **noteOn**(MY\_FLOAT frequency, MY\_FLOAT amplitude)
  
  Start a note with the given frequency and amplitude (0.0 - 1.0).

- void **setBodySize**(MY\_FLOAT size)
  
  Set the body size (a value of 1.0 produces the "default" size).

- virtual MY\_FLOAT **tick**()
  
  Compute one output sample.
virtual void controlChange (int number, MY_FLOAT value)

Perform the control change specified by number and value (0.0 - 128.0).

4.27.1 Detailed Description

STK mandolin instrument model class.

This class inherits from PluckTwo and uses "commuted synthesis" techniques to model a mandolin instrument.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others. Commuted Synthesis, in particular, is covered by patents, granted, pending, and/or applied-for. All are assigned to the Board of Trustees, Stanford University. For information, contact the Office of Technology Licensing, Stanford University.

Control Change Numbers:

- Body Size = 2
- Pluck Position = 4
- String Sustain = 11
- String Detuning = 1
- Microphone Position = 128


The documentation for this class was generated from the following file:

- Mandolin.h
4.28 Mesh2D Class Reference

Two-dimensional rectilinear waveguide mesh class.

```c
#include <Mesh2D.h>
```

Inheritance diagram for Mesh2D::

```
Stk

Instrmnt

Mesh2D
```

Public Methods

- `Mesh2D (short nX, short nY)`
  
  *Class constructor, taking the x and y dimensions in samples.*

- `~Mesh2D ()`

  *Class destructor.*

- `void clear ()`

  *Reset and clear all internal state.*

- `void setNX (short lenX)`

  *Set the x dimension size in samples.*

- `void setNY (short lenY)`

  *Set the y dimension size in samples.*

- `void setInputPosition (MY_FLOAT xFactor, MY_FLOAT yFactor)`

  *Set the x, y input position on a 0.0 - 1.0 scale.*

- `void setDecay (MY_FLOAT decayFactor)`

  *Set the loss filters gains (0.0 - 1.0).*

- `void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)`

  *Impulse the mesh with the given amplitude (frequency ignored).*

- `void noteOff (MY_FLOAT amplitude)`

---

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
Stop a note with the given amplitude (speed of decay) ... currently ignored.

- **MY_FLOAT energy()**
  
  Calculate and return the signal energy stored in the mesh.

- **MY_FLOAT tick()**
  
  Compute one output sample, without adding energy to the mesh.

- **MY_FLOAT tick(MY_FLOAT input)**
  
  Input a sample to the mesh and compute one output sample.

- **void controlChange(int number, MY_FLOAT value)**
  
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.28.1 Detailed Description

Two-dimensional rectilinear waveguide mesh class.

This class implements a rectilinear, two-dimensional digital waveguide mesh structure. For details, see Van Duyne and Smith, "Physical Modeling with the 2-D Digital Waveguide Mesh", Proceedings of the 1993 International Computer Music Conference.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- X Dimension = 2
- Y Dimension = 4
- Mesh Decay = 11
- X-Y Input Position = 1


The documentation for this class was generated from the following file:

- **Mesh2D.h**

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
4.29 Messager Class Reference

STK input control message parser.

#include <Messager.h>

Inheritance diagram for Messager::

```
Stk
    Messager
```

Public Methods

- **Messager** (int inputMask=0, int port=2001)
  
  Constructor performs initialization based on an input mask and an optional socket port.

- **∼Messager** ()
  
  Class destructor.

- **long nextMessage**(void)
  
  Check for a new input message and return the message type.

- **void setRtDelta**(long nSamples)
  
  Set the delta time (in samples) returned between valid realtime messages. This setting has no affect for scorefile messages.

- **long getDelta**(void) const
  
  Return the current message “delta time” in samples.

- **long getType**(void) const
  
  Return the current message type.

- **MY_FLOAT getByteTwo**( ) const
  
  Return the byte two value for the current message.

- **MY_FLOAT getByteThree**( ) const
  
  Return the byte three value for the current message.

- **long getChannel**( ) const
4.29 Messager Class Reference

Return the channel number for the current message.

4.29.1 Detailed Description

STK input control message parser.

This class reads and parses control messages from a variety of sources, such as a MIDI port, scorefile, socket connection, or pipe. MIDI messages are retrieved using the RtMidi class. All other input sources (scorefile, socket, or pipe) are assumed to provide SKINI formatted messages.

For each call to `nextMessage()`, the active input sources are queried to see if a new control message is available.

This class is primarily for use in STK main() event loops.

One of the original goals in creating this class was to simplify the message acquisition process by removing all threads. If the windoze select() function behaved just like the unix one, that would have been possible. Since it does not (it can't be used to poll STDIN), I am using a thread to acquire messages from STDIN, which sends these messages via a socket connection to the message socket server. Perhaps in the future, it will be possible to simplify things.


4.29.2 Constructor & Destructor Documentation

4.29.2.1 Messager::Messager (int inputMask = 0, int port = 2001)

Constructor performs initialization based on an input mask and an optional socket port.

The default constructor is set to read input from a SKINI scorefile. The flags STK_MIDI, STK_PIPE, and STK_SOCKET can be OR'ed together in any combination for multiple "realtime" input source parsing. An optional socket port number can be specified for use when the STK_SOCKET flag is set. For realtime input types, an StkError can be thrown during instantiation.

4.29.3 Member Function Documentation

4.29.3.1 long Messager::nextMessage (void)

Check for a new input message and return the message type.

Return type values greater than zero represent valid messages. If an input scorefile has been completely read or all realtime input sources have closed, a
negative value is returned. If the return type is zero, no valid messages are present.

The documentation for this class was generated from the following file:

- Messager.h
4.30 Modal Class Reference

STK resonance model instrument.

```cpp
#include <Modal.h>
```

Inheritance diagram for Modal::

```
  Stk
    |    
  Instrmnt
    |    
  Modal
    |    
  ModalBar
```

### Public Methods

- **Modal** (int modes=4)
  
  *Class constructor, taking the desired number of modes to create.*

- virtual **~Modal** ()
  
  *Class destructor.*

- void **clear** ()
  
  *Reset and clear all internal state.*

- virtual void **setFrequency** (MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **setRatioAndRadius** (int modeIndex, MY_FLOAT ratio, MY_FLOAT radius)
  
  *Set the ratio and radius for a specified mode filter.*

- void **setMasterGain** (MY_FLOAT aGain)
  
  *Set the master gain.*

- void **setDirectGain** (MY_FLOAT aGain)
  
  *Set the direct gain.*

- void **setModeGain** (int modeIndex, MY_FLOAT gain)

---

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
Set the gain for a specified mode filter.

- virtual void \texttt{strike}(MY\_FLOAT amplitude)
  
  \textit{Initiate a strike with the given amplitude (0.0 - 1.0).}

- void \texttt{damp}(MY\_FLOAT amplitude)

  \textit{Damp modes with a given decay factor (0.0 - 1.0).}

- void \texttt{noteOn}(MY\_FLOAT frequency, MY\_FLOAT amplitude)

  \textit{Start a note with the given frequency and amplitude.}

- void \texttt{noteOff}(MY\_FLOAT amplitude)

  \textit{Stop a note with the given amplitude (speed of decay).}

- virtual MY\_FLOAT \texttt{tick}()

  \textit{Compute one output sample.}

- virtual void \texttt{controlChange}(int number, MY\_FLOAT value)=0

  \textit{Perform the control change specified by number and value (0.0 - 128.0).}

\subsection*{4.30.1 Detailed Description}

STK resonance model instrument.

This class contains an excitation wavetable, an envelope, an oscillator, and \(N\) resonances (non-sweeping BiQuad filters), where \(N\) is set during instantiation.


The documentation for this class was generated from the following file:

- \texttt{Modal.h}
4.31 ModalBar Class Reference

STK resonant bar instrument class.

```cpp
#include <ModalBar.h>
```

Inheritance diagram for ModalBar:

```
Stk
    
Instrmnt
    
Modal
    
ModalBar
```

Public Methods

- `ModalBar()`
  - *Class constructor.*

- `~ModalBar()`
  - *Class destructor.*

- `void setStickHardness(MY_FLOAT hardness)`
  - *Set stick hardness (0.0 - 1.0).*

- `void setStrikePosition(MY_FLOAT position)`
  - *Set stick position (0.0 - 1.0).*

- `void setPreset(int preset)`
  - *Select a bar preset (currently modulo 9).*

- `void setModulationDepth(MY_FLOAT mDepth)`
  - *Set the modulation (vibrato) depth.*

- `void controlChange(int number, MY_FLOAT value)`
  - *Perform the control change specified by number and value (0.0 - 128.0).*

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
4.31.1 Detailed Description

STK resonant bar instrument class.

This class implements a number of different struck bar instruments. It inherits from the Modal class.

Control Change Numbers:

- Stick Hardness = 2
- Stick Position = 4
- Vibrato Gain = 11
- Vibrato Frequency = 7
- Volume = 128
- Modal Presets = 16
  - Marimba = 0
  - Vibraphone = 1
  - Agogo = 2
  - Wood1 = 3
  - Reso = 4
  - Wood2 = 5
  - Beats = 6
  - Two Fixed = 7
  - Clump = 8


The documentation for this class was generated from the following file:

- ModalBar.h
4.32 Modulate Class Reference

STK periodic/random modulator.

```cpp
#include <Modulate.h>
```

Inheritance diagram for Modulate::

```
   Stk
    |
    v
Modulate
```

Public Methods

- **Modulate ()**
  
  *Class constructor.*

- **~Modulate ()**
  
  *Class destructor.*

- void **reset ()**
  
  *Reset internal state.*

- void **setVibratoRate (MY_FLOAT aRate)**
  
  *Set the periodic (vibrato) rate or frequency in Hz.*

- void **setVibratoGain (MY_FLOAT aGain)**
  
  *Set the periodic (vibrato) gain.*

- void **setRandomGain (MY_FLOAT aGain)**
  
  *Set the random modulation gain.*

- MY_FLOAT **tick ()**
  
  *Compute one output sample.*

- virtual MY_FLOAT* **tick (MY_FLOAT *vector, unsigned int vectorSize)**
  
  *Return vectorSize outputs in vector.*

- MY_FLOAT **lastOut () const**
  
  *Return the last computed output value.*
4.32.1 Detailed Description

STK periodic/random modulator.

This class combines random and periodic modulations to give a nice, natural human modulation function.


The documentation for this class was generated from the following file:

- [Modulate.h]
4.33 Moog Class Reference

STK moog-like swept filter sampling synthesis class.

```cpp
#include <Moog.h>
```

Inheritance diagram for Moog:

```
    Stk
     ↓
    Instrmnt
     ↓
    Sampler
     ↓
    Moog
```

Public Methods

- `Moog ()`
  
  Class constructor.

- `~Moog ()`
  
  Class destructor.

- `virtual void setFrequency (MY_FLOAT frequency)`
  
  Set instrument parameters for a particular frequency.

- `virtual void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)`
  
  Start a note with the given frequency and amplitude.

- `void setModulationSpeed (MY_FLOAT mSpeed)`
  
  Set the modulation (vibrato) speed in Hz.

- `void setModulationDepth (MY_FLOAT mDepth)`
  
  Set the modulation (vibrato) depth.

- `virtual MY_FLOAT tick ()`
  
  Compute one output sample.

- `virtual void controlChange (int number, MY_FLOAT value)`
  
  Perform the control change specified by number and value (0.0 - 128.0).
4.33.1 Detailed Description

STK moog-like swept filter sampling synthesis class.

This instrument uses one attack wave, one looped wave, and an ADSR envelope (inherited from the Sampler class) and adds two sweepable formant (FormSweep) filters.

Control Change Numbers:

- **Filter** Q = 2
- **Filter** Sweep Rate = 4
- Vibrato Frequency = 11
- Vibrato Gain = 1
- Gain = 128


The documentation for this class was generated from the following file:

- **Moog.h**
4.34 Noise Class Reference

STK noise generator.

#include <Noise.h>

Inheritance diagram for Noise:

```
SubNoise
  |
  v
Noise
  |
  v
Stk
```

### Public Methods

- **Noise()**
  
  Default constructor.

- **virtual ~Noise()**
  
  Class destructor.

- **virtual MY_FLOAT tick()**
  
  Return a random number between -1.0 and 1.0 using rand().

- **virtual MY_FLOAT* tick(MY_FLOAT* vector, unsigned int vectorSize)**
  
  Return vectorSize random numbers between -1.0 and 1.0 in vector.

- **MY_FLOAT lastOut() const**
  
  Return the last computed value.

### 4.34.1 Detailed Description

STK noise generator.

Generic random number generation using the C rand() function. The quality of the rand() function varies from one OS to another.


The documentation for this class was generated from the following file:
• Noise.h
4.35 NRev Class Reference

CCRMA’s NRev reverberator class.

#include <NRev.h>

Inheritance diagram for NRev::

```
Stk
  ↓
Reverb
  ↓
NRev
```

Public Methods

- **NRev** (MY_FLOAT T60)
  
  *Class constructor taking a T60 decay time argument.*

- **~NRev** ()
  
  *Class destructor.*

- **clear** ()
  
  *Reset and clear all internal state.*

- **tick** (MY_FLOAT input)
  
  *Compute one output sample.*

4.35.1 Detailed Description

CCRMA’s NRev reverberator class.

This class is derived from the CLM NRev function, which is based on the use of networks of simple allpass and comb delay filters. This particular arrangement consists of 6 comb filters in parallel, followed by 3 allpass filters, a lowpass filter, and another allpass in series, followed by two allpass filters in parallel with corresponding right and left outputs.


The documentation for this class was generated from the following file:

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995-2002
• NRev.h
4.36 OnePole Class Reference

STK one-pole filter class.

#include <OnePole.h>

Inheritance diagram for OnePole:

```
Stk
  
Filter
  
OnePole
```

Public Methods

- **OnePole()**
  
  Default constructor creates a first-order low-pass filter.

- **OnePole(MY_FLOAT thePole)**

  Overloaded constructor which sets the pole position during instantiation.

- **~OnePole()**

  Class destructor.

- **void clear()**

  Clears the internal state of the filter.

- **void setB0(MY_FLOAT b0)**

  Set the b[0] coefficient value.

- **void setA1(MY_FLOAT a1)**

  Set the a[1] coefficient value.

- **void setPole(MY_FLOAT thePole)**

  Set the pole position in the z-plane.

- **void setGain(MY_FLOAT theGain)**

  Set the filter gain.

- **MY_FLOAT getGain()**

  Get the filter gain.
Return the current filter gain.

- **MY_FLOAT**\_{\text{lastOut}}(\text{void}) \text{ const}
  
  Return the last computed output value.

- **MY_FLOAT**\_{\text{tick}}(\text{MY_FLOAT sample})
  
  Input one sample to the filter and return one output.

- **MY_FLOAT**\_\*(\text{tick} (\text{MY_FLOAT \_\*vector, unsigned int vectorSize})
  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.36.1 Detailed Description

STK one-pole filter class.

This protected Filter subclass implements a one-pole digital filter. A method is provided for setting the pole position along the real axis of the z-plane while maintaining a constant peak filter gain.


4.36.2 Member Function Documentation

4.36.2.1 void OnePole::setPole (MY_FLOAT thePole)

Set the pole position in the z-plane.

This method sets the pole position along the real-axis of the z-plane and normalizes the coefficients for a maximum gain of one. A positive pole value produces a low-pass filter, while a negative pole value produces a high-pass filter. This method does not affect the filter gain value.

4.36.2.2 void OnePole::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented from Filter

The documentation for this class was generated from the following file:

- OnePole.h
4.37 OneZero Class Reference

STK one-zero filter class.

```cpp
#include <OneZero.h>
```

Inheritance diagram for OneZero::

```
Stk
  ↓
Filter
  ↓
OneZero
```

Public Methods

- **OneZero()**
  
  Default constructor creates a first-order low-pass filter.

- **OneZero(MY_FLOAT theZero)**
  
  Overloaded constructor which sets the zero position during instantiation.

- **~OneZero()**
  
  Class destructor.

- **void clear(void)**
  
  Clears the internal state of the filter.

- **void setB0(MY_FLOAT b0)**
  
  Set the b[0] coefficient value.

- **void setB1(MY_FLOAT b1)**
  
  Set the b[1] coefficient value.

- **void setZero(MY_FLOAT theZero)**
  
  Set the zero position in the z-plane.

- **void setGain(MY_FLOAT theGain)**
  
  Set the filter gain.

- **MY_FLOAT getGain(void) const**
Return the current filter gain.

- **MY_FLOAT** lastOut (void) const
  
  Return the last computed output value.

- **MY_FLOAT** tick (MY_FLOAT sample)
  
  Input one sample to the filter and return one output.

- **MY_FLOAT** *tick (MY_FLOAT *vector, unsigned int vectorSize)
  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

### 4.37.1 Detailed Description

STK one-zero filter class.

This protected Filter subclass implements a one-zero digital filter. A method is provided for setting the zero position along the real axis of the z-plane while maintaining a constant filter gain.


### 4.37.2 Member Function Documentation

#### 4.37.2.1 void OneZero::setZero (MY_FLOAT theZero)

Set the zero position in the z-plane.

This method sets the zero position along the real-axis of the z-plane and normalizes the coefficients for a maximum gain of one. A positive zero value produces a high-pass filter, while a negative zero value produces a low-pass filter. This method does not affect the filter gain value.

#### 4.37.2.2 void OneZero::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented from Filter.

The documentation for this class was generated from the following file:

- **OneZero.h**
4.38 PercFlut Class Reference

STK percussive flute FM synthesis instrument.

#include <PercFlut.h>

Inheritance diagram for PercFlut::

```
    Stk
     |   
     v   v
Instrmnt
     |   
     v   v
      FM
           
               PercFlut
```

Public Methods

- `PercFlut()`
  
  *Class constructor.*

- `~PercFlut()`
  
  *Class destructor.*

- `void setFrequency(MY_FLOAT frequency)`
  
  *Set instrument parameters for a particular frequency.*

- `void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)`
  
  *Start a note with the given frequency and amplitude.*

- `MY_FLOAT tick()`
  
  *Compute one output sample.*

4.38.1 Detailed Description

STK percussive flute FM synthesis instrument.

This class implements algorithm 4 of the TX81Z.

Algorithm 4 is:

4--3--\  
2-- + --1--Out
Control Change Numbers:

- Total Modulator Index = 2
- Modulator Crossfade = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- PercFlut.h
4.39 Phonemes Class Reference

STK phonemes table.
#include <Phonemes.h>

Static Public Methods

- **const char* name(unsigned int index)**
  
  Returns the phoneme name for the given index (0-31).

- **MY_FLOAT voiceGain(unsigned int index)**
  
  Returns the voiced component gain for the given phoneme index (0-31).

- **MY_FLOAT noiseGain(unsigned int index)**
  
  Returns the unvoiced component gain for the given phoneme index (0-31).

- **MY_FLOAT formantFrequency(unsigned int index, unsigned int partial)**

  Returns the formant frequency for the given phoneme index (0-31) and partial (0-3).

- **MY_FLOAT formantRadius(unsigned int index, unsigned int partial)**

  Returns the formant radius for the given phoneme index (0-31) and partial (0-3).

- **MY_FLOAT formantGain(unsigned int index, unsigned int partial)**

  Returns the formant gain for the given phoneme index (0-31) and partial (0-3).

4.39.1 Detailed Description

STK phonemes table.

This class does nothing other than declare a set of 32 static phoneme formant parameters and provide access to those values.


The documentation for this class was generated from the following file:

- **Phonemes.h**
4.40 PitShift Class Reference

STK simple pitch shifter effect class.

#include <PitShift.h>

Inheritance diagram for PitShift::

```
+-----------------+  +-----------------+
|                 |  |                 |
|     Stk         |  |     PitShift    |
|                 |  |                 |
```

Public Methods

- **PitShift()**
  
  *Class constructor.*

- **~PitShift()**
  
  *Class destructor.*

- **void clear()**
  
  *Reset and clear all internal state.*

- **void setShift(MY_FLOAT shift)**
  
  *Set the pitch shift factor (1.0 produces no shift).*

- **void setEffectMix(MY_FLOAT mix)**
  
  *Set the mixture of input and processed levels in the output (0.0 = input only, 1.0 = processed only).*

- **MY_FLOAT lastOut() const**
  
  *Return the last output value.*

- **MY_FLOAT* tick(MY_FLOAT input)**
  
  *Compute one output sample.*

- **MY_FLOAT* tick(MY_FLOAT* vector, unsigned int vectorSize)**
  
  *Input vectorSize samples to the filter and return an equal number of outputs in vector.*
4.40.1 Detailed Description

STK simple pitch shifter effect class.
This class implements a simple pitch shifter using delay lines.
The documentation for this class was generated from the following file:

- PitShift.h
4.41 Plucked Class Reference

STK plucked string model class.

#include <Plucked.h>

Inheritance diagram for Plucked::

```
+-------------------+            +-------------------+
|                   |            |                   |
|       Stk         |            |       Instrmnt    |
|                   |            |                   |
|                   +-------------------+ Plucked
```

Public Methods

- **Plucked**(MY_FLOAT lowestFrequency)
  
  *Class constructor, taking the lowest desired playing frequency.*

- **~Plucked**()
  
  *Class destructor.*

- void **clear**()
  
  *Reset and clear all internal state.*

- virtual void **setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **pluck**(MY_FLOAT amplitude)
  
  *Pluck the string with the given amplitude using the current frequency.*

- virtual void **noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  
  *Start a note with the given frequency and amplitude.*

- virtual void **noteOff**(MY_FLOAT amplitude)
  
  *Stop a note with the given amplitude (speed of decay).*

- virtual MY_FLOAT **tick**()
  
  *Compute one output sample.*
4.41.1 Detailed Description

STK plucked string model class.

This class implements a simple plucked string physical model based on the Karplus-Strong algorithm.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others. There exist at least two patents, assigned to Stanford, bearing the names of Karplus and/or Strong, by Perry R. Cook and Gary P. Scavone, 1995 - 2002.

The documentation for this class was generated from the following file:

- Plucked.h
4.42 PluckTwo Class Reference

STK enhanced plucked string model class.

```cpp
#include <PluckTwo.h>
```

Inheritance diagram for PluckTwo:

```
Stk
   |
   v
Instrmnt
   |
   v
PluckTwo
   |
   v
Mandolin
```

Public Methods

- **PluckTwo**(MY_FLOAT lowestFrequency)
  
  *Class constructor, taking the lowest desired playing frequency.*

- virtual **~PluckTwo**()
  
  *Class destructor.*

- void **clear**()
  
  *Reset and clear all internal state.*

- virtual void **setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **setDetune**(MY_FLOAT detune)
  
  *Detune the two strings by the given factor. A value of 1.0 produces unison strings.*

- void **setFreqAndDetune**(MY_FLOAT frequency, MY_FLOAT detune)
  
  *Efficient combined setting of frequency and detuning.*

- void **setPluckPosition**(MY_FLOAT position)
  
  *Set the pluck or "excitation" position along the string (0.0 - 1.0).*

- void **setBaseLoopGain**(MY_FLOAT aGain)
4.42 PluckTwo Class Reference

Set the base loop gain.

- virtual void noteOff (MY_FLOAT amplitude)
  Stop a note with the given amplitude (speed of decay).

- virtual MY_FLOAT tick()=0
  Virtual (abstract) tick function is implemented by subclasses.

4.42.1 Detailed Description

STK enhanced plucked string model class.

This class implements an enhanced two-string, plucked physical model, a la Jaffe-Smith, Smith, and others.

PluckTwo is an abstract class, with no excitation specified. Therefore, it can’t be directly instantiated.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.


4.42.2 Member Function Documentation

4.42.2.1 void PluckTwo::setBaseLoopGain (MY_FLOAT aGain)

Set the base loop gain.

The actual loop gain is set according to the frequency. Because of high-frequency loop filter roll-off, higher frequency settings have greater loop gains.

The documentation for this class was generated from the following file:

- PluckTwo.h
4.43 PoleZero Class Reference

STK one-pole, one-zero filter class.

```
#include <PoleZero.h>
```

Inheritance diagram for PoleZero:

![Inheritance Diagram](image)

Public Methods

- **PoleZero()**
  
  Default constructor creates a first-order pass-through filter.

- **~PoleZero()**
  
  Class destructor.

- **void clear()**
  
  Clears the internal states of the filter.

- **void setB0(MY_FLOAT b0)**
  
  Set the b[0] coefficient value.

- **void setB1(MY_FLOAT b1)**
  
  Set the b[1] coefficient value.

- **void setA1(MY_FLOAT a1)**
  
  Set the a[1] coefficient value.

- **void setAllpass(MY_FLOAT coefficient)**
  
  Set the filter for allpass behavior using coefficient.

- **void setBlockZero(MY_FLOAT thePole=0.99)**
  
  Create a DC blocking filter with the given pole position in the z-plane.

- **void setGain(MY_FLOAT theGain)**
Set the filter gain.

- **MY_FLOAT getGain**(void) const
  
  Return the current filter gain.

- **MY_FLOAT lastOut**(void) const
  
  Return the last computed output value.

- **MY_FLOAT tick**(MY_FLOAT sample)
  
  Input one sample to the filter and return one output.

- **MY_FLOAT tick**(MY_FLOAT *vector, unsigned int vectorSize)
  
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

### 4.43.1 Detailed Description

STK one-pole, one-zero filter class.

This protected Filter subclass implements a one-pole, one-zero digital filter. A method is provided for creating an allpass filter with a given coefficient. Another method is provided to create a DC blocking filter.


### 4.43.2 Member Function Documentation

#### 4.43.2.1 void PoleZero::setAllpass (MY_FLOAT coefficient)

Set the filter for allpass behavior using coefficient.

This method uses coefficient to create an allpass filter, which has unity gain at all frequencies. Note that the coefficient magnitude must be less than one to maintain stability.

#### 4.43.2.2 void PoleZero::setBlockZero (MY_FLOAT thePole = 0.99)

Create a DC blocking filter with the given pole position in the z-plane.

This method sets the given pole position, together with a zero at z=1, to create a DC blocking filter. thePole should be close to one to minimize low-frequency attenuation.
4.43.2.3  void PoleZero::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values.
The default gain value is 1.0.

Reimplemented from Filter

The documentation for this class was generated from the following file:

- PoleZero.h
4.44 PRCRev Class Reference

Perry’s simple reverberator class.

#include <PRCRev.h>

Inheritance diagram for PRCRev::

```
PRCRev
    |        |
    | Reverb |
    |        |
    | Stk    |
```

Public Methods

- **PRCRev**(MY_FLOAT T60)
  
  *Class constructor taking a T60 decay time argument.*

- **∼PRCRev**()
  
  *Class destructor.*

- void **clear**()
  
  *Reset and clear all internal state.*

- **MY_FLOAT tick**(MY_FLOAT input)
  
  *Compute one output sample.*

4.44.1 Detailed Description

Perry’s simple reverberator class.

This class is based on some of the famous Stanford/CCRMA reverbs (NRev, KipRev), which were based on the Chowning/Moorer/Schroeder reverberators using networks of simple allpass and comb delay filters. This class implements two series allpass units and two parallel comb filters.


The documentation for this class was generated from the following file:

- **PRCRev.h**

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995-2002
4.45 ReedTabl Class Reference

STK reed table class.

```cpp
#include <ReedTabl.h>
```

Inheritance diagram for ReedTabl:

```
  Stk
 /\   
+-- ReedTabl
```

**Public Methods**

- **ReedTabl ()**
  
  *Default constructor.*

- **~ReedTabl ()**
  
  *Class destructor.*

- **void setOffset (MY_FLOAT aValue)**
  
  *Set the table offset value.*

- **void setSlope (MY_FLOAT aValue)**
  
  *Set the table slope value.*

- **MY_FLOAT lastOut () const**
  
  *Return the last output value.*

- **MY_FLOAT *tick (MY_FLOAT *vector, unsigned int vectorSize)**
  
  *Take vectorSize inputs and return the corresponding function values in vector.*

4.45.1 Detailed Description

STK reed table class.
This class implements a simple one breakpoint, non-linear reed function, as described by Smith (1986). This function is based on a memoryless non-linear spring model of the reed (the reed mass is ignored) which saturates when the reed collides with the mouthpiece facing.

See McIntyre, Schumacher, & Woodhouse (1983), Smith (1986), Hirschman, Cook, Scavone, and others for more information.


### 4.45.2 Member Function Documentation

#### 4.45.2.1 void ReedTabl::setOffset (MY_FLOAT aValue)

Set the table offset value.

The table offset roughly corresponds to the size of the initial reed tip opening (a greater offset represents a smaller opening).

#### 4.45.2.2 void ReedTabl::setSlope (MY_FLOAT aValue)

Set the table slope value.

The table slope roughly corresponds to the reed stiffness (a greater slope represents a harder reed).

#### 4.45.2.3 MY_FLOAT ReedTabl::tick (MY_FLOAT input)

Return the function value for input.

The function input represents the differential pressure across the reeds.

The documentation for this class was generated from the following file:

- ReedTabl.h
4.46 Resonate Class Reference

STK noise driven formant filter.

```cpp
#include <Resonate.h>
```

Inheritance diagram for Resonate::

```
Stk
  
Instrmnt
  
Resonate
```

Public Methods

- **Resonate()**
  
  *Class constructor.*

- **~Resonate()**
  
  *Class destructor.*

- **void clear()**
  
  *Reset and clear all internal state.*

- **void setResonance(MY_FLOAT frequency, MY_FLOAT radius)**
  
  *Set the filter for a resonance at the given frequency (Hz) and radius.*

- **void setNotch(MY_FLOAT frequency, MY_FLOAT radius)**
  
  *Set the filter for a notch at the given frequency (Hz) and radius.*

- **void setEqualGainZeros()**
  
  *Set the filter zero coefficients for constant resonance gain.*

- **void keyOn()**
  
  *Initiate the envelope with a key-on event.*

- **void keyOff()**
  
  *Signal a key-off event to the envelope.*

- **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
Start a note with the given frequency and amplitude.

- void noteOff (MY_FLOAT amplitude)
  Stop a note with the given amplitude (speed of decay).

- MY_FLOAT tick ()
  Compute one output sample.

- virtual void controlChange (int number, MY_FLOAT value)
  Perform the control change specified by number and value (0.0 - 128.0).

4.46.1 Detailed Description

STK noise driven formant filter.

This instrument contains a noise source, which excites a biquad resonance filter, with volume controlled by an ADSR.

Control Change Numbers:

- Resonance Frequency (0-Nyquist) = 2
- Pole Radii = 4
- Notch Frequency (0-Nyquist) = 11
- Zero Radii = 1
- Envelope Gain = 128


The documentation for this class was generated from the following file:

- Resonate.h
4.47 Reverb Class Reference

STK abstract reverberator parent class.

```
#include <Reverb.h>
```

Inheritance diagram for Reverb::

```
Stk
   
Reverb
   
   JCRev NRev PRCRev
```

Public Methods

- **Reverb()**
  
  *Class constructor.*

- virtual **~Reverb()**

  *Class destructor.*

- virtual void **clear()**=0

  *Reset and clear all internal state.*

- void **setEffectMix(MY_FLOAT mix)**

  *Set the mixture of input and "reverberated" levels in the output (0.0 = input only, 1.0 = reverb only).*

- **MY_FLOAT lastOut()** const

  *Return the last output value.*

- **MY_FLOAT lastOutLeft()** const

  *Return the last left output value.*

- **MY_FLOAT lastOutRight()** const

  *Return the last right output value.*

- virtual **MY_FLOAT tick(MY_FLOAT input)**=0

  *Abstract tick function ... must be implemented in subclasses.*
• virtual MY_FLOAT* tick (MY_FLOAT *vector, unsigned int vectorSize)
   
   Take vectorSize inputs, compute the same number of outputs and return them in vector.

4.47.1 Detailed Description

STK abstract reverberator parent class.
This class provides common functionality for STK reverberator subclasses.
The documentation for this class was generated from the following file:

• Reverb.h
4.48 Rhodey Class Reference

STK Fender Rhodes electric piano FM synthesis instrument.

```cpp
#include <Rhodey.h>
```

Inheritance diagram for Rhodey:

```
  Stk
   |
   v
  Instrmnt
   |
   v
  FM
   |
   v
Rhodey
```

Public Methods

- **Rhodey()**
  
  `Class constructor.`

- **~Rhodey()**
  
  `Class destructor.`

- **void setFrequency(MY_FLOAT frequency)**
  
  `Set instrument parameters for a particular frequency.`

- **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
  
  `Start a note with the given frequency and amplitude.`

- **MY_FLOAT tick()**
  
  `Compute one output sample.`

4.48.1 Detailed Description

STK Fender Rhodes electric piano FM synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.
Algorithm 5 is:

4-->3--\
    + --> Out
      2-->1--/

Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- Rhodey.h
4.49  RtAudio Class Reference

Realtime audio i/o C++ class.

#include <RtAudio.h>

Public Methods

- **RtAudio()**
  The default constructor.

- **RtAudio(int *streamId, int outputDevice, int outputChannels, int inputDevice, int inputChannels, RTAUDIO_FORMAT format, int sampleRate, int *bufferSize, int numberOfBuffers)**
  A constructor which can be used to open a stream during instantiation.

- **~RtAudio()**
  The destructor.

- **int openStream(int outputDevice, int outputChannels, int inputDevice, int inputChannels, RTAUDIO_FORMAT format, int sampleRate, int *bufferSize, int numberOfBuffers)**
  A public method for opening a stream with the specified parameters.

- **void setStreamCallback(int streamId, RTAUDIO_CALLBACK callback, void *userData)**
  A public method which sets a user-defined callback function for a given stream.

- **void cancelStreamCallback(int streamId)**
  A public method which cancels a callback process and function for a given stream.

- **int getDeviceCount(void)**
  A public method which returns the number of audio devices found.

- **void getDeviceInfo(int device, RTAUDIO_DEVICE *info)**
  Fill a user-supplied RTAUDIO_DEVICE structure for a specified device number.

- **char* const getStreamBuffer(int streamId)**
  A public method which returns a pointer to the buffer for an open stream.
4.49 RtAudio Class Reference

- void tickStream(int streamId)
  Public method used to trigger processing of input/output data for a stream.

- void closeStream(int streamId)
  Public method which closes a stream and frees any associated buffers.

- void startStream(int streamId)
  Public method which starts a stream.

- void stopStream(int streamId)
  Stop a stream, allowing any samples remaining in the queue to be played out
  and/or read in.

- void abortStream(int streamId)
  Stop a stream, discarding any samples remaining in the input/output queue.

- int streamWillBlock(int streamId)
  Queries a stream to determine whether a call to the tickStream() method will
  block.

Static Public Attributes

- const RTAUDIO_FORMAT RTAUDIO_SINT8
- const RTAUDIO_FORMAT RTAUDIO_SINT16
- const RTAUDIO_FORMAT RTAUDIO_SINT24
- const RTAUDIO_FORMAT RTAUDIO_SINT32
- const RTAUDIO_FORMAT RTAUDIO_FLOAT32
- const RTAUDIO_FORMAT RTAUDIO_FLOAT64

4.49.1 Detailed Description

Realtime audio i/o C++ class.

RtAudio provides a common API (Application Programming Interface) for realtime
audio input/output across Linux (native ALSA and OSS), SGI, Macintosh
OS X (CoreAudio), and Windows (DirectSound and ASIO) operating systems.

RtAudio WWW site: http://www-ccrma.stanford.edu/~gary/rtaudio/

RtAudio: a realtime audio i/o C++ class Copyright (c) 2001-2002 Gary P.
Scavone

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Any person wishing to distribute modifications to the Software is requested to send the modifications to the original developer so that they can be incorporated into the canonical version.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.49.2 Constructor & Destructor Documentation

4.49.2.1 RtAudio::RtAudio ()

The default constructor.

Probes the system to make sure at least one audio input/output device is available and determines the api-specific identifier for each device found. An RtError error can be thrown if no devices are found or if a memory allocation error occurs.

4.49.2.2 RtAudio::RtAudio (int * streamId, int outputDevice, int outputChannels, int inputDevice, int inputChannels, RTAUDIO_FORMAT format, int sampleRate, int * bufferSize, int numberOfBuffers)

A constructor which can be used to open a stream during instantiation.

The specified output and/or input device identifiers correspond to those enumerated via the getDeviceInfo() method. If device = 0, the default or first available devices meeting the given parameters is selected. If an output or input channel value is zero, the corresponding device value is ignored. When a stream is successfully opened, its identifier is returned via the "streamId" pointer. An RtError can be thrown if no devices are found for the given parameters, if a memory allocation error occurs, or if a driver error occurs.
4.49.2.3 RtAudio::~RtAudio ()

The destructor.

Stops and closes any open streams and devices and deallocates buffer and structure memory.

4.49.3 Member Function Documentation

4.49.3.1 int RtAudio::openStream (int outputDevice, int outputChannels, int inputDevice, int inputChannels, RTAUDIO_FORMAT format, int sampleRate, int *bufferSize, int numberOfBuffers)

A public method for opening a stream with the specified parameters.

If successful, the opened stream ID is returned. Otherwise, an RtError is thrown.

Parameters:

outputDevice: If equal to 0, the default or first device found meeting the given parameters is opened. Otherwise, the device number should correspond to one of those enumerated via the getDeviceInfo() method.

outputChannels: The desired number of output channels. If equal to zero, the outputDevice identifier is ignored.

inputDevice: If equal to 0, the default or first device found meeting the given parameters is opened. Otherwise, the device number should correspond to one of those enumerated via the getDeviceInfo() method.

inputChannels: The desired number of input channels. If equal to zero, the inputDevice identifier is ignored.

format: An RTAUDIO_FORMAT specifying the desired sample data format.

sampleRate: The desired sample rate (sample frames per second).

bufferSize: A pointer value indicating the desired internal buffer size in sample frames. The actual value used by the device is returned via the same pointer. A value of zero can be specified, in which case the lowest allowable value is determined.

numberOfBuffers: A value which can be used to help control device latency. More buffers typically result in more robust performance, though at a cost of greater latency. A value of zero can be specified, in which case the lowest allowable value is used.
4.49.3.2 void RtAudio::setStreamCallback (int streamId, RTAUDIO_CALLBACK callback, void *userData)

A public method which sets a user-defined callback function for a given stream. This method assigns a callback function to a specific, previously opened stream for non-blocking stream functionality. A separate process is initiated, though the user function is called only when the stream is "running" (between calls to the startStream() and stopStream() methods, respectively). The callback process remains active for the duration of the stream and is automatically shutdown when the stream is closed (via the closeStream() method or by object destruction). The callback process can also be shutdown and the user function de-referenced through an explicit call to the cancelStreamCallback() method. Note that a single stream can use only blocking or callback functionality at the same time, though it is possible to alternate modes on the same stream through the use of the setStreamCallback() and cancelStreamCallback() methods (the blocking tickStream() method can be used before a callback is set and/or after a callback is cancelled). An RtError will be thrown for an invalid device argument.

4.49.3.3 void RtAudio::cancelStreamCallback (int streamId)

A public method which cancels a callback process and function for a given stream. This method shuts down a callback process and de-references the user function for a specific stream. Callback functionality can subsequently be restarted on the stream via the setStreamCallback() method. An RtError will be thrown for an invalid device argument.

4.49.3.4 void RtAudio::getDeviceInfo (int device, RTAUDIO_DEVICE *info)

Fill a user-supplied RTAUDIO_DEVICE structure for a specified device number. Any device integer between 1 and getDeviceCount() is valid. If a device is busy or otherwise unavailable, the structure member "probed" will have a value of "false" and all other members are undefined. If the specified device is the current default input or output device, the "isDefault" member will have a value of "true". An RtError will be thrown for an invalid device argument.

4.49.3.5 char *const RtAudio::getStreamBuffer (int streamId)

A public method which returns a pointer to the buffer for an open stream. The user should fill and/or read the buffer data in interleaved format and then
call the \texttt{tickStream()} method. An \texttt{RtError} will be thrown for an invalid stream identifier.

4.49.3.6 \textbf{void RtAudio::tickStream (int \textit{streamId})}

Public method used to trigger processing of input/output data for a stream. 
This method blocks until all buffer data is read/written. An \texttt{RtError} will be thrown for an invalid stream identifier or if a driver error occurs.

4.49.3.7 \textbf{void RtAudio::closeStream (int \textit{streamId})}

Public method which closes a stream and frees any associated buffers.
If an invalid stream identifier is specified, this method issues a warning and returns (an \texttt{RtError} is not thrown).

4.49.3.8 \textbf{void RtAudio::startStream (int \textit{streamId})}

Public method which starts a stream.
An \texttt{RtError} will be thrown for an invalid stream identifier or if a driver error occurs.

4.49.3.9 \textbf{void RtAudio::stopStream (int \textit{streamId})}

Stop a stream, allowing any samples remaining in the queue to be played out and/or read in.
An \texttt{RtError} will be thrown for an invalid stream identifier or if a driver error occurs.

4.49.3.10 \textbf{void RtAudio::abortStream (int \textit{streamId})}

Stop a stream, discarding any samples remaining in the input/output queue.
An \texttt{RtError} will be thrown for an invalid stream identifier or if a driver error occurs.

4.49.3.11 \textbf{int RtAudio::streamWillBlock (int \textit{streamId})}

Queries a stream to determine whether a call to the \texttt{tickStream()} method will block.
A return value of 0 indicates that the stream will NOT block. A positive return value indicates the number of sample frames that cannot yet be processed without blocking.

4.49.4 Member Data Documentation

4.49.4.1 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT8 [static]
8-bit signed integer.

4.49.4.2 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT16 [static]
16-bit signed integer.

4.49.4.3 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT24 [static]
Upper 3 bytes of 32-bit signed integer.

4.49.4.4 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT32 [static]
32-bit signed integer.

4.49.4.5 const RTAUDIO_FORMAT RtAudio::RTAUDIO_FLOAT32 [static]
Normalized between plus/minus 1.0.

4.49.4.6 const RTAUDIO_FORMAT RtAudio::RTAUDIO_FLOAT64 [static]
Normalized between plus/minus 1.0.

The documentation for this class was generated from the following file:

- `RtAudio.h`
4.50  RtAudio::RTAUDIO_DEVICE Struct Reference

The public device information structure for passing queried values.

#include <RtAudio.h>

Public Attributes

- char name [128]
- bool probed
- int maxOutputChannels
- int maxInputChannels
- int maxDuplexChannels
- int minOutputChannels
- int minInputChannels
- int minDuplexChannels
- bool hasDuplexSupport
- bool isDefault
- int nSampleRates
- int sampleRates [MAX_SAMPLE_RATES]
- RTAUDIO_FORMAT nativeFormats

4.50.1 Detailed Description

The public device information structure for passing queried values.

4.50.2 Member Data Documentation

4.50.2.1 char RtAudio::RTAUDIODEVICE::name

Character string device identifier.

4.50.2.2 bool RtAudio::RTAUDIODEVICE::probed

true if the device capabilities were successfully probed.

4.50.2.3 int RtAudio::RTAUDIODEVICE::maxOutputChannels

Maximum output channels supported by device.
4.50.2.4 int RtAudio::RTAUDIO_DEVICE::maxInputChannels

Maximum input channels supported by device.

4.50.2.5 int RtAudio::RTAUDIO_DEVICE::maxDuplexChannels

Maximum simultaneous input/output channels supported by device.

4.50.2.6 int RtAudio::RTAUDIO_DEVICE::minOutputChannels

Minimum output channels supported by device.

4.50.2.7 int RtAudio::RTAUDIO_DEVICE::minInputChannels

Minimum input channels supported by device.

4.50.2.8 int RtAudio::RTAUDIO_DEVICE::minDuplexChannels

Minimum simultaneous input/output channels supported by device.

4.50.2.9 bool RtAudio::RTAUDIO_DEVICE::hasDuplexSupport

true if device supports duplex mode.

4.50.2.10 bool RtAudio::RTAUDIO_DEVICE::isDefault

true if this is the default output or input device.

4.50.2.11 int RtAudio::RTAUDIO_DEVICE::nSampleRates

Number of discrete rates or -1 if range supported.

4.50.2.12 int RtAudio::RTAUDIO_DEVICE::sampleRates

Supported rates or (min, max) if range.

4.50.2.13 RTAUDIO_FORMAT RtAudio::RTAUDIO_DEVICE::nativeFormats

Bit mask of supported data formats.
The documentation for this struct was generated from the following file:

- `RtAudio.h`
4.51 RtDuplex Class Reference

STK realtime audio input/output class.

#include <RtDuplex.h>

Inheritance diagram for RtDuplex:

```
Inheritance diagram for RtDuplex:

  Stk
    ▼
     
    RtDuplex
```

Public Methods

- **RtDuplex** (int nChannels=1, MY_FLOAT sampleRate=Stk::sampleRate(), int device=0, int bufferFrames=RT_BUFFER_SIZE, int nBuffers=2)
  
  Default constructor.

- **RtDuplex**()
  
  Class destructor.

- void **start** (void)
  
  Start the audio input/output stream.

- void **stop** (void)
  
  Stop the audio input/output stream.

- MY_FLOAT **lastOut** (void) const
  
  Return the average across the last output sample frame.

- MY_FLOAT **tick** (const MY_FLOAT sample)
  
  Output a single sample to all channels in a sample frame and return the average across one new input sample frame of data.

- MY_FLOAT **tick** (MY_FLOAT *vector, unsigned int vectorSize)
  
  Output each sample in `vector` to all channels per frame and return averaged input sample frames of new data in `vector`.

- const MY_FLOAT **lastFrame** (void) const
  
  Return a pointer to the last output sample frame.
4.51 RtDuplex Class Reference

- MY_FLOAT* tickFrame (MY_FLOAT *frameVector, unsigned int frames=1)

  Output sample frames from frameVector and return new input frames in frameVector.

4.51.1 Detailed Description

STK realtime audio input/output class.

This class provides a simplified interface to `RtAudio` for realtime audio input/output. It is also possible to achieve duplex operation using separate `RtWvIn` and `RtWvOut` classes, but this class ensures better input/output synchronization.

RtDuplex supports multi-channel data in interleaved format. It is important to distinguish the `tick()` methods, which output single samples to all channels in a sample frame and return samples produced by averaging across sample frames, from the `tickFrame()` methods, which take/return pointers to multi-channel sample frames.


4.51.2 Constructor & Destructor Documentation

4.51.2.1 RtDuplex::RtDuplex (int nChannels = 1, MY_FLOAT sampleRate = Stk::sampleRate(), int device = 0, int bufferFrames = RT_BUFFER_SIZE, int nBuffers = 2)

Default constructor.

The `device` argument is passed to `RtAudio` during instantiation. The default value (zero) will select the default device on your system or the first device found meeting the specified parameters. On systems with multiple soundcards/devices, values greater than zero can be specified in accordance with the order that the devices are enumerated by the underlying audio API. The default buffer size of `RT_BUFFER_SIZE` is defined in Stk.h. An `StkError` will be thrown if an error occurs during instantiation.

4.51.3 Member Function Documentation

4.51.3.1 void RtDuplex::start (void)

Start the audio input/output stream.

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
The stream is started automatically, if necessary, when a `tick()` or `tickFrame` method is called.

### 4.51.3.2 void RtDuplex::stop (void)

Stop the audio input/output stream.

It may be necessary to use this method to avoid audio overflow/underflow problems if you wish to temporarily stop the audio stream.

### 4.51.3.3 MY_FLOAT RtDuplex::tick (const MY_FLOAT sample)

Output a single sample to all channels in a sample frame and return the average across one new input sample frame of data.

An `StkError` will be thrown if an error occurs during input/output.

### 4.51.3.4 MY_FLOAT * RtDuplex::tick (MY_FLOAT * vector, unsigned int vectorSize)

Output each sample in `vector` to all channels per frame and return averaged input sample frames of new data in `vector`.

An `StkError` will be thrown if an error occurs during input/output.

### 4.51.3.5 MY_FLOAT * RtDuplex::tickFrame (MY_FLOAT * frameVector, unsigned int frames = 1)

Output sample frames from `frameVector` and return new input frames in `frameVector`.

An `StkError` will be thrown if an error occurs during input/output.

The documentation for this class was generated from the following file:

- `RtDuplex.h`
4.52 RtError Class Reference

Exception handling class for RtAudio.
#include <RtAudio.h>

Public Types

- enum TYPE { WARNING, DEBUG_WARNING, UNSPECIFIED, NO_DEVICES_FOUND, INVALID_DEVICE, INVALID_STREAM, MEMORY_ERROR, INVALID_PARAMETER, DRIVER_ERROR, SYSTEM_ERROR, THREAD_ERROR }

Defined RtError types.

Public Methods

- RtError (const char *p, TYPE tipe=RtError::UNSPECIFIED)
The constructor.

- virtual ~RtError (void)
The destructor.

- virtual void printMessage (void)
Prints "thrown" error message to stdout.

- virtual const TYPE& getType (void)
Returns the "thrown" error message TYPE.

- virtual const char* getMessage (void)
Returns the "thrown" error message string.

4.52.1 Detailed Description

Exception handling class for RtAudio.

The RtError class is quite simple but it does allow errors to be "caught" by RtError::TYPE. Almost all RtAudio methods can "throw" an RtError, most typically if an invalid stream identifier is supplied to a method or a driver error occurs. There are a number of cases within RtAudio where warning messages may be displayed but an exception is not thrown. There is a private RtAudio
method, error(), which can be modified to globally control how these messages are handled and reported.

The documentation for this class was generated from the following file:

- **RtAudio.h**
4.53  RtMidi Class Reference

STK realtime MIDI class.

#include <RtMidi.h>

Inheritance diagram for RtMidi:

```
  Stk
  |   
  v   
  RtMidi
```

Public Methods

- **RtMidi** (int device=0)
  Default constructor with optional device argument.

- **RtMidi**()
  Class destructor.

- void **printMessage** (void) const
  Print out the current message values.

- int **nextMessage** (void)
  Check for and parse a new MIDI message in the queue, returning its type.

- int **getType** () const
  Return the current message type.

- int **getChannel** () const
  Return the current message channel value.

- MY_FLOAT **getByteTwo** () const
  Return the current message byte two value.

- MY_FLOAT **getByteThree** () const
  Return the current message byte three value.

- MY_FLOAT **getDeltaTime** () const
  Return the current message delta time value in seconds.
4.53.1  Detailed Description

STK realtime MIDI class.

At the moment, this object only handles MIDI input, though MIDI output code can go here when someone decides they need it (and writes it).

This object opens a MIDI input device and parses MIDI messages into a MIDI buffer. Time stamp info is converted to a delta-time value. MIDI data is stored as MY_FLOAT to conform with SKINI. System exclusive messages are currently ignored.

An optional argument to the constructor can be used to specify a device or card. When no argument is given, a default device is opened. If a device argument fails, a list of available devices is printed to allow selection by the user.

This code is based in part on work of Perry Cook (SGI), Paul Leonard (Linux), the RoseGarden team (Linux), and Bill Putnam (Windows).


4.53.2  Member Function Documentation

4.53.2.1  int RtMidi::nextMessage (void)

Check for and parse a new MIDI message in the queue, returning its type.
If a new message is found, the return value is greater than zero.

The documentation for this class was generated from the following file:

- RtMidi.h
4.54  RtWvIn Class Reference

STK realtime audio input class.

#include <RtWvIn.h>

Inheritance diagram for RtWvIn:

```
Stk
  WvIn
    ...
  RtWvIn
```

Public Methods

- **RtWvIn**(int nChannels=1, MY_FLOAT sampleRate=Stk::sampleRate(), int device=0, int bufferFrames=RT_BUFFER_SIZE, int nBuffers=2)
  Default constructor.

- **~RtWvIn**()
  Class destructor.

- **start**(void)
  Start the audio input stream.

- **stop**(void)
  Stop the audio input stream.

- **lastOut**(void) const
  Return the average across the last output sample frame.

- **tick**(void)
  Read out the average across one sample frame of data.

- **tick**(MY_FLOAT *vector, unsigned int vectorSize)
  Read out vectorSize averaged sample frames of data in vector.

- **lastFrame**(void) const
  Return a pointer to the last output sample frame.
4.54.1 Detailed Description

STK realtime audio input class.

This class provides a simplified interface to RtAudio for realtime audio input. It is a protected subclass of WvIn.

RtWvIn supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which return samples produced by averaging across sample frames, from the tickFrame() methods, which return pointers to multi-channel sample frames. For single-channel data, these methods return equivalent values.


4.54.2 Constructor & Destructor Documentation

4.54.2.1 RtWvIn::RtWvIn (int nChannels = 1, MY_FLOAT sampleRate = Stk::sampleRate(), int device = 0, int bufferFrames = RT_BUFFER_SIZE, int nBuffers = 2)

Default constructor.

The device argument is passed to RtAudio during instantiation. The default value (zero) will select the default device on your system or the first device found meeting the specified parameters. On systems with multiple soundcards/devices, values greater than zero can be specified in accordance with the order that the devices are enumerated by the underlying audio API. The default buffer size of RT_BUFFER_SIZE is defined in Stk.h. An StkError will be thrown if an error occurs during instantiation.

4.54.3 Member Function Documentation

4.54.3.1 void RtWvIn::start (void)

Start the audio input stream.
The stream is started automatically, if necessary, when a `tick()` or `tickFrame` method is called.

**4.54.3.2 void RtWvIn::stop (void)**

Stop the audio input stream.

It may be necessary to use this method to avoid audio underflow problems if you wish to temporarily stop audio input.

**4.54.3.3 MYFLOAT RtWvIn::tick (void) [virtual]**

Read out the average across one sample frame of data.

An `StkError` will be thrown if an error occurs during input.

Reimplemented from `WvIn`.

**4.54.3.4 MYFLOAT * RtWvIn::tick (MYFLOAT * vector, unsigned int vectorSize) [virtual]**

Read out `vectorSize` averaged sample frames of data in `vector`.

An `StkError` will be thrown if an error occurs during input.

Reimplemented from `WvIn`.

**4.54.3.5 const MYFLOAT * RtWvIn::tickFrame (void) [virtual]**

Return a pointer to the next sample frame of data.

An `StkError` will be thrown if an error occurs during input.

Reimplemented from `WvIn`.

**4.54.3.6 MYFLOAT * RtWvIn::tickFrame (MYFLOAT * frameVector, unsigned int frames) [virtual]**

Read out sample `frames` of data to `frameVector`.

An `StkError` will be thrown if an error occurs during input.

Reimplemented from `WvIn`.

The documentation for this class was generated from the following file:

- `RtWvIn.h`

---

*The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002*
4.55  RtWvOut Class Reference

STK realtime audio output class.

#include <RtWvOut.h>

Inheritance diagram for RtWvOut:

```
Stk
   ↓
WvOut
   ↓
RtWvOut
```

Public Methods

- **RtWvOut** (unsigned int nChannels=1, MY_FLOAT sampleRate=Stk::sampleRate(), int device=0, int bufferFrames=RT_BUFFER_SIZE, int nBuffers=4)
  
  Default constructor.

- **~RtWvOut** ()
  
  Class destructor.

- void **start** (void)
  
  Start the audio output stream.

- void **stop** (void)
  
  Stop the audio output stream.

- unsigned long **getFrames** (void) const
  
  Return the number of sample frames output.

- MY_FLOAT **getTime** (void) const
  
  Return the number of seconds of data output.

- void **tick** (const MY_FLOAT sample)
  
  Output a single sample to all channels in a sample frame.

- void **tick** (const MY_FLOAT *vector, unsigned int vectorSize)
  
  Output each sample in vector to all channels in vectorSize sample frames.
4.55 RtWvOut Class Reference

• void tickFrame (const MY_FLOAT *frameVector, unsigned int frames=1)
  
  Output the frameVector of sample frames of the given length.

4.55.1 Detailed Description

STK realtime audio output class.

This class provides a simplified interface to RtAudio for realtime audio output. It is a protected subclass of WvOut.

RtWvOut supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which output single samples to all channels in a sample frame, from the tickFrame() method, which takes a pointer to multi-channel sample frame data.


4.55.2 Constructor & Destructor Documentation

4.55.2.1 RtWvOut::RtWvOut (unsigned int nChannels = 1,
  MY_FLOAT sampleRate = Stk::sampleRate(), int device = 0, int bufferFrames = RT_BUFFER_SIZE, int nBuffers = 4)

Default constructor.

The device argument is passed to RtAudio during instantiation. The default value (zero) will select the default device on your system or the first device found meeting the specified parameters. On systems with multiple soundcards/devices, values greater than zero can be specified in accordance with the order that the devices are enumerated by the underlying audio API. The default buffer size of RT_BUFFER_SIZE is defined in Stk.h. An StkError will be thrown if an error occurs during instantiation.

4.55.3 Member Function Documentation

4.55.3.1 void RtWvOut::start (void)

Start the audio output stream.

The stream is started automatically, if necessary, when a tick() or tickFrame method is called.
4.55.3.2  void RtWvOut::stop (void)

Stop the audio output stream.
It may be necessary to use this method to avoid undesirable audio buffer cycling
if you wish to temporarily stop audio output.

4.55.3.3  void RtWvOut::tick (const MY_FLOAT sample)
[virtual]

Output a single sample to all channels in a sample frame.
An StkError will be thrown if an error occurs during output.
Reimplemented from WvOut

4.55.3.4  void RtWvOut::tick (const MY_FLOAT * vector, unsigned
int vectorSize)  [virtual]

Output each sample in vector to all channels in vectorSize sample frames.
An StkError will be thrown if an error occurs during output.
Reimplemented from WvOut

4.55.3.5  void RtWvOut::tickFrame (const MY_FLOAT * frameVector, unsigned int frames = 1)  [virtual]

Output the frameVector of sample frames of the given length.
An StkError will be thrown if an error occurs during output.
Reimplemented from WvOut
The documentation for this class was generated from the following file:

- RtWvOut.h
# 4.56 Sampler Class Reference

STK sampling synthesis abstract base class.

```cpp
#include <Sampler.h>
```

Inheritance diagram for Sampler::

```
Sampler
  |                 
  v                 
Instrmnt
  |                 
  v                 
Stk
  |                 
  v                 
Moog
```

**Public Methods**

- `Sampler()`  
  *Default constructor.*

- virtual `~Sampler()`  
  *Class destructor.*

- void `clear()`  
  *Reset and clear all internal state.

- virtual void `setFrequency(MY_FLOAT frequency)=0`  
  *Set instrument parameters for a particular frequency.*

- void `keyOn()`  
  *Initiate the envelopes with a key-on event and reset the attack waves.*

- void `keyOff()`  
  *Signal a key-off event to the envelopes.*

- virtual void `noteOff(MY_FLOAT amplitude)`  
  *Stop a note with the given amplitude (speed of decay).*

- virtual `MY_FLOAT tick()`
Compute one output sample.

- virtual void controlChange(int number, MYFLOAT value)=0
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.56.1 Detailed Description

STK sampling synthesis abstract base class.

This instrument contains up to 5 attack waves, 5 looped waves, and an ADSR envelope.


The documentation for this class was generated from the following file:

- Sampler.h
4.57 Saxofony Class Reference

STK faux conical bore reed instrument class.

#include <Saxofony.h>

Inheritance diagram for Saxofony::

```
  Stk
  Instrmnt
  Saxofony
```

Public Methods

- **Saxofony**(MY_FLOAT lowestFrequency)
  Class constructor, taking the lowest desired playing frequency.

- **~Saxofony**()
  Class destructor.

- void **clear**()
  Reset and clear all internal state.

- void **setFrequency**(MY_FLOAT frequency)
  Set instrument parameters for a particular frequency.

- void **setBlowPosition**(MY_FLOAT aPosition)
  Set the "blowing" position between the air column terminations (0.0 - 1.0).

- void **startBlowing**(MY_FLOAT amplitude, MY_FLOAT rate)
  Apply breath pressure to instrument with given amplitude and rate of increase.

- void **stopBlowing**(MY_FLOAT rate)
  Decrease breath pressure with given rate of decrease.

- void **noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  Start a note with the given frequency and amplitude.
• void noteOff (MY_FLOAT amplitude)
  Stop a note with the given amplitude (speed of decay).

• MY_FLOAT tick()
  Compute one output sample.

• void controlChange (int number, MY_FLOAT value)
  Perform the control change specified by number and value (0.0 - 128.0).

4.57.1 Detailed Description

STK faux conical bore reed instrument class.

This class implements a "hybrid" digital waveguide instrument that can generate a variety of wind-like sounds. It has also been referred to as the "blown string" model. The waveguide section is essentially that of a string, with one rigid and one lossy termination. The non-linear function is a reed table. The string can be "blown" at any point between the terminations, though just as with strings, it is impossible to excite the system at either end. If the excitation is placed at the string mid-point, the sound is that of a clarinet. At points closer to the "bridge", the sound is closer to that of a saxophone. See Scavone (2002) for more details.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

• Reed Stiffness = 2
• Reed Aperture = 26
• Noise Gain = 4
• Blow Position = 11
• Vibrato Frequency = 29
• Vibrato Gain = 1
• Breath Pressure = 128


The documentation for this class was generated from the following file:

• Saxofony.h
4.58  Shakers Class Reference

PhISEM and PhOLIES class.

```c
#include <Shakers.h>
```

Inheritance diagram for Shakers:

```
Shakers
   |        |
   v        v
   Instrmnt
      |        |
      v        v
      Stk
```

Public Methods

- **Shakers()**
  
  Class constructor.

- **~Shakers()**
  
  Class destructor.

- **virtual void noteOn(MY_FLOAT instrument, MY_FLOAT amplitude)**
  
  Start a note with the given instrument and amplitude.

- **virtual void noteOff(MY_FLOAT amplitude)**
  
  Stop a note with the given amplitude (speed of decay).

- **MY_FLOAT tick()**
  
  Compute one output sample.

- **virtual void controlChange(int number, MY_FLOAT value)**
  
  Perform the control change specified by number and value (0.0 - 128.0).

4.58.1  Detailed Description

PhISEM and PhOLIES class.
PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects. This class is a meta-model that can simulate a Maraca, Sekere, Cabasa, Bamboo Wind Chimes, Water Drops, Tambourine, Sleighbells, and a Guiro.

PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds) is a similar approach for the synthesis of environmental sounds. This class implements simulations of breaking sticks, crunchy snow (or not), a wrench, sandpaper, and more.

Control Change Numbers:

- Shake Energy = 2
- System Decay = 4
- Number Of Objects = 11
- Resonance Frequency = 1
- Shake Energy = 128
- Instrument Selection = 1071

  - Maraca = 0
  - Cabasa = 1
  - Sekere = 2
  - Guiro = 3
  - Water Drops = 4
  - Bamboo Chimes = 5
  - Tambourine = 6
  - Sleigh Bells = 7
  - Sticks = 8
  - Crunch = 9
  - Wrench = 10
  - Sand Paper = 11
  - Coke Can = 12
  - Next Mug = 13
  - Penny + Mug = 14
  - Nickle + Mug = 15
  - Dime + Mug = 16
  - Quarter + Mug = 17
  - Franc + Mug = 18
  - Peso + Mug = 19
  - Big Rocks = 20
  - Little Rocks = 21
  - Tuned Bamboo Chimes = 22

4.58.2 Member Function Documentation

4.58.2.1 void Shakers::noteOn (MY_FLOAT instrument,
MY_FLOAT amplitude) [virtual]

Start a note with the given instrument and amplitude.

Use the instrument numbers above, converted to frequency values as if MIDI
note numbers, to select a particular instrument.

Reimplemented from Instrmnt.

The documentation for this class was generated from the following file:

• Shakers.h
4.59 Simple Class Reference

STK wavetable/noise instrument.

```c
#include <Simple.h>
```

Inheritance diagram for Simple::

```
    Stk
     ↓
    Instrmnt
     ↓
    Simple
```

Public Methods

- **Simple()**
  
  *Class constructor.*

- virtual **~Simple()**
  
  *Class destructor.*

- **void clear()**
  
  *Clear internal states.*

- virtual **void setFrequency(MY_FLOAT frequency)**
  
  *Set instrument parameters for a particular frequency.*

- **void keyOn()**
  
  *Start envelope toward "on" target.*

- **void keyOff()**
  
  *Start envelope toward "off" target.*

- virtual **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
  
  *Start a note with the given frequency and amplitude.*

- virtual **void noteOff(MY_FLOAT amplitude)**
  
  *Stop a note with the given amplitude (speed of decay).*

- virtual **MY_FLOAT tick()**
Compute one output sample.

- virtual void controlChange(int number, MY_FLOAT value)
  
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.59.1 Detailed Description

STK wavetable/noise instrument.

This class combines a looped wave, a noise source, a biquad resonance filter, a one-pole filter, and an ADSR envelope to create some interesting sounds.

Control Change Numbers:

- Filter Pole Position = 2
- Noise/Pitched Cross-Fade = 4
- Envelope Rate = 11
- Gain = 128


The documentation for this class was generated from the following file:

- Simple.h
4.60 SingWave Class Reference

STK “singing” looped soundfile class.

#include <SingWave.h>

Inheritance diagram for SingWave:

```
  Stk
   |
   V
  SingWave
```

Public Methods

- **SingWave** (const char *fileName, bool raw=FALSE)
  Class constructor taking filename argument.

- **~SingWave** ()
  Class destructor.

- **void** reset ()
  Reset file to beginning.

- **void** normalize ()
  Normalize the file to a maximum of +-1.0.

- **void** normalize (MY_FLOAT peak)
  Normalize the file to a maximum of +- peak.

- **void** setFrequency (MY_FLOAT frequency)
  Set instrument parameters for a particular frequency.

- **void** setVibratoRate (MY_FLOAT aRate)
  Set the vibrato frequency in Hz.

- **void** setVibratoGain (MY_FLOAT gain)
  Set the vibrato gain.

- **void** setRandomGain (MY_FLOAT gain)
  Set the random-ness amount.
• void `setSweepRate` (MY_FLOAT aRate)
  
  Set the sweep rate.

• void `setGainRate` (MY_FLOAT aRate)
  
  Set the gain rate.

• void `setGainTarget` (MY_FLOAT target)
  
  Set the gain target value.

• void `noteOn` ()
  
  Start a note.

• void `noteOff` ()
  
  Stop a note.

• MY_FLOAT `lastOut` ()
  
  Return the last output value.

• MY_FLOAT `tick` ()
  
  Compute one output sample.

### 4.60.1 Detailed Description

STK "singing" looped soundfile class.

This class contains all that is needed to make a pitched musical sound, like a simple voice or violin. In general, it will not be used alone because of munchkinification effects from pitch shifting. It will be used as an excitation source for other instruments.


### 4.60.2 Constructor & Destructor Documentation

#### 4.60.2.1 SingWave::SingWave (const char ∗ fileName, bool raw = FALSE)

Class constructor taking filename argument.

An `StkError` will be thrown if the file is not found, its format is unknown, or a read error occurs.

The documentation for this class was generated from the following file:
• SingWave.h
4.61 Sitar Class Reference

STK sitar string model class.

```cpp
#include <Sitar.h>
```

Inheritance diagram for Sitar::

![Inheritance Diagram]

Public Methods

- **Sitar**(MY_FLOAT lowestFrequency)
  
  *Class constructor, taking the lowest desired playing frequency.*

- **~Sitar**()
  
  *Class destructor.*

- void **clear**()
  
  *Reset and clear all internal state.*

- void **setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- void **pluck**(MY_FLOAT amplitude)
  
  *Pluck the string with the given amplitude using the current frequency.*

- void **noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
  
  *Start a note with the given frequency and amplitude.*

- void **noteOff**(MY_FLOAT amplitude)
  
  *Stop a note with the given amplitude (speed of decay).*

- MY_FLOAT **tick**()
  
  *Compute one output sample.*
4.61.1 Detailed Description

STK sitar string model class.

This class implements a sitar plucked string physical model based on the Karplus-Strong algorithm.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others. There exist at least two patents, assigned to Stanford, bearing the names of Karplus and/or Strong.


The documentation for this class was generated from the following file:

- Sitar.h
4.62 SKINI Class Reference

STK SKINI parsing class.

#include <SKINI.h>

Inheritance diagram for SKINI::

```
SKINI
   ^
   |  Stk
```

Public Methods

- **SKINI()**
  Default constructor used for parsing messages received externally.

- **SKINI(char *fileName)**
  Overloaded constructor taking a SKINI formatted scorefile.

- **~SKINI()**
  Class destructor.

- **long parseThis(char *aString)**
  Attempt to parse the given string, returning the message type.

- **long nextMessage()**
  Parse the next message (if a file is loaded) and return the message type.

- **long getType() const**
  Return the current message type.

- **long getChannel() const**
  Return the current message channel value.

- **MY_FLOAT getDelta() const**
  Return the current message delta time value (in seconds).

- **MY_FLOAT getByteTwo() const**
  Return the current message byte two value.
• MY_FLOAT `getByteThree()` const
  Return the current message byte three value.

• long `getByteTwoInt()` const
  Return the current message byte two value (integer).

• long `getByteThreeInt()` const
  Return the current message byte three value (integer).

• const char* `getRemainderString()`
  Return remainder string after parsing.

• const char* `getMessageTypeString()`
  Return the message type as a string.

• const char* `whatsThisType(long type)`
  Return the SKINI type string for the given type value.

• const char* `whatsThisController(long number)`
  Return the SKINI controller string for the given controller number.

### 4.62.1 Detailed Description

STK SKINI parsing class.

This class parses SKINI formatted text messages. It can be used to parse individual messages or it can be passed an entire file. The file specification is Perry’s and his alone, but it’s all text so it shouldn’t be too hard to figure out.

SKINI (Synthesis toolKit Instrument Network Interface) is like MIDI, but allows for floating-point control changes, note numbers, etc. The following example causes a sharp middle C to be played with a velocity of 111.132:

```
noteOn 60.01 111.13
```

See also:
Synthesis toolKit Instrument Network Interface (SKINI)

4.62.2 Member Function Documentation

4.62.2.1 long SKINI::parseThis (char * aString)

Attempt to parse the given string, returning the message type. 
A type value equal to zero indicates an invalid message.

4.62.2.2 long SKINI::nextMessage (void)

Parse the next message (if a file is loaded) and return the message type. 
A negative value is returned when the file end is reached. 
The documentation for this class was generated from the following file:

- SKINI.h
4.63 Socket Class Reference

STK TCP socket client/server class.

#include <Socket.h>

Inheritance diagram for Socket:

```
                Stk
               /   |
              /    |
             Stk  Socket
```

Public Methods

- **Socket** (int port=2006)
  
  *Default constructor which creates a local socket server on port 2006 (or the specified port number).*

- **Socket** (int port, const char *hostname)
  
  *Class constructor which creates a socket client connection to the specified host and port.*

- **~Socket** ()
  
  *The class destructor closes the socket instance, breaking any existing connections.*

- **int connect** (int port, const char *hostname="localhost")
  
  *Connect a socket client to the specified host and port and returns the resulting socket descriptor.*

- **void close** (void)
  
  *Close this socket.*

- **int socket** (void) const
  
  *Return the server/client socket descriptor.*

- **int port** (void) const
  
  *Return the server/client port number.*

- **int accept** (void)
If this is a socket server, extract the first pending connection request from the queue and create a new connection, returning the descriptor for the accepted socket.

- **int writeBuffer (const void *buffer, long bufferSize, int flags=0)**
  
  Write a buffer over the socket connection. Returns the number of bytes written or -1 if an error occurs.

- **int readBuffer (void *buffer, long bufferSize, int flags=0)**
  
  Read a buffer from the socket connection, up to length bufferSize. Returns the number of bytes read or -1 if an error occurs.

### Static Public Methods

- **void setBlocking (int socket, bool enable)**
  
  If enable = false, the socket is set to non-blocking mode. When first created, sockets are by default in blocking mode.

- **void close (int socket)**
  
  Close the socket with the given descriptor.

- **bool isValid (int socket)**
  
  Returns TRUE is the socket descriptor is valid.

- **int writeBuffer (int socket, const void *buffer, long bufferSize, int flags)**
  
  Write a buffer via the specified socket. Returns the number of bytes written or -1 if an error occurs.

- **int readBuffer (int socket, void *buffer, long bufferSize, int flags)**
  
  Read a buffer via the specified socket. Returns the number of bytes read or -1 if an error occurs.

### 4.63.1 Detailed Description

STK TCP socket client/server class.

This class provides a uniform cross-platform TCP socket client or socket server interface. Methods are provided for reading or writing data buffers to/from connections. This class also provides a number of static functions for use with external socket descriptors.

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
The user is responsible for checking the values returned by the read/write methods. Values less than or equal to zero indicate a closed or lost connection or the occurrence of an error.


4.63.2 Constructor & Destructor Documentation

4.63.2.1 Socket::Socket (int port = 2006)

Default constructor which creates a local socket server on port 2006 (or the specified port number).

An StkError will be thrown if a socket error occurs during instantiation.

4.63.2.2 Socket::Socket (int port, const char *hostname)

Class constructor which creates a socket client connection to the specified host and port.

An StkError will be thrown if a socket error occurs during instantiation.

4.63.3 Member Function Documentation

4.63.3.1 int Socket::connect (int port, const char *hostname = "localhost")

Connect a socket client to the specified host and port and returns the resulting socket descriptor.

This method is valid for socket clients only. If it is called for a socket server, -1 is returned. If the socket client is already connected, that connection is terminated and a new connection is attempted. Server connections are made using the accept() method. An StkError will be thrown if a socket error occurs during instantiation.

See also: accept

4.63.3.2 int Socket::accept (void)

If this is a socket server, extract the first pending connection request from the queue and create a new connection, returning the descriptor for the accepted socket.
If no connection requests are pending and the socket has not been set non-blocking, this function will block until a connection is present. If an error occurs or this is a socket client, -1 is returned.

The documentation for this class was generated from the following file:

- [Socket.h](Socket.h)
4.64   Sphere Class Reference

STK sphere class.

#include <Sphere.h>

Public Methods

- **Sphere** (double initRadius)
  Constructor taking an initial radius value.

- **~Sphere** ()
  Class destructor.

- void **setPosition** (double anX, double aY, double aZ)
  Set the 3D center position of the sphere.

- void **setVelocity** (double anX, double aY, double aZ)
  Set the 3D velocity of the sphere.

- void **setRadius** (double aRadius)
  Set the radius of the sphere.

- void **setMass** (double aMass)
  Set the mass of the sphere.

- Vector3D **getPosition** ()
  Get the current position of the sphere as a 3D vector.

- Vector3D **getRelativePosition** (Vector3D *aPosition)
  Get the relative position of the given point to the sphere as a 3D vector.

- double **getVelocity** (Vector3D *aVelocity)
  Set the velocity of the sphere as a 3D vector.

- double **isInside** (Vector3D *aPosition)
  Returns the distance from the sphere boundary to the given position (< 0 if inside).

- double **getRadius** ()
  Get the current sphere radius.

- double **getMass** ()
Get the current sphere mass.

- void **addVelocity** (double anX, double aY, double aZ)
  
  Increase the current sphere velocity by the given 3D components.

- void **tick** (double timeIncrement)
  
  Move the sphere for the given time increment.

### 4.64.1 Detailed Description

STK sphere class.

This class implements a spherical ball with radius, mass, position, and velocity parameters.


The documentation for this class was generated from the following file:

- **Sphere.h**
4.65 StifKarp Class Reference

STK plucked stiff string instrument.

```
#include <StifKarp.h>
```

Inheritance diagram for StifKarp:

```
Stk
   ↓
Instrmnt
   ↓
StifKarp
```

**Public Methods**

- **StifKarp**(MY_FLOAT lowestFrequency)
  
  *Class constructor, taking the lowest desired playing frequency.*

- **~StifKarp**()
  
  *Class destructor.*

- **void clear**()
  
  *Reset and clear all internal state.*

- **void setFrequency**(MY_FLOAT frequency)
  
  *Set instrument parameters for a particular frequency.*

- **void setStretch**(MY_FLOAT stretch)
  
  *Set the stretch “factor” of the string (0.0 - 1.0).*

- **void setPickupPosition**(MY_FLOAT position)
  
  *Set the pluck or “excitation” position along the string (0.0 - 1.0).*

- **void setBaseLoopGain**(MY_FLOAT aGain)
  
  *Set the base loop gain.*

- **void pluck**(MY_FLOAT amplitude)
  
  *Pluck the string with the given amplitude using the current frequency.*

- **void noteOn**(MY_FLOAT frequency, MY_FLOAT amplitude)
Start a note with the given frequency and amplitude.

- void `noteOff` (MY_FLOAT amplitude)
  *Stop a note with the given amplitude (speed of decay).*

- MY_FLOAT `tick`()
  *Compute one output sample.*

- void `controlChange` (int number, MY_FLOAT value)
  *Perform the control change specified by number and value (0.0 - 128.0).*

### 4.65.1 Detailed Description

STK plucked stiff string instrument.

This class implements a simple plucked string algorithm (Karplus Strong) with enhancements (Jaffe-Smith, Smith, and others), including string stiffness and pluck position controls. The stiffness is modeled with allpass filters.

This is a digital waveguide model, making its use possibly subject to patents held by Stanford University, Yamaha, and others.

Control Change Numbers:

- Pickup Position = 4
- String Sustain = 11
- String Stretch = 1


### 4.65.2 Member Function Documentation

#### 4.65.2.1 void StifKarp::setBaseLoopGain (MY_FLOAT aGain)

Set the base loop gain.

The actual loop gain is set according to the frequency. Because of high-frequency loop filter roll-off, higher frequency settings have greater loop gains.

The documentation for this class was generated from the following file:

- `StifKarp.h`
4.66 Stk Class Reference

STK base class.

#include <Stk.h>

Inheritance diagram for Stk::

Static Public Methods

- MY_FLOAT \texttt{sampleRate} (void)
  
  \textit{Static method which returns the current STK sample rate.}

- void \texttt{setSampleRate} (MY_FLOAT newRate)
  
  \textit{Static method which sets the STK sample rate.}
• void swap16 (unsigned char *ptr)
  Static method which byte-swaps a 16-bit data type.

• void swap32 (unsigned char *ptr)
  Static method which byte-swaps a 32-bit data type.

• void swap64 (unsigned char *ptr)
  Static method which byte-swaps a 64-bit data type.

• void sleep (unsigned long milliseconds)
  Static cross-platform method to sleep for a number of milliseconds.

Static Public Attributes

• const STK_FORMAT STK_SINT8
• const STK_FORMAT STK_SINT16
• const STK_FORMAT STK_SINT32
• const STK_FORMAT STK_FLOAT32
• const STK_FORMAT STK_FLOAT64

Protected Methods

• Stk (void)
  Default constructor.

• virtual ~Stk (void)
  Class destructor.

Static Protected Methods

• void handleError (const char *message, StkError::TYPE type)
  Function for error reporting and handling.
4.66.1 Detailed Description

STK base class.

Nearly all STK classes inherit from this class. The global sample rate can be queried and modified via Stk. In addition, this class provides error handling and byte-swapping functions.


4.66.2 Member Function Documentation

4.66.2.1 void Stk::setSampleRate (MY_FLOAT newRate) [static]

Static method which sets the STK sample rate.

The sample rate set using this method is queried by all STK classes which depend on its value. It is initialized to the default SRATE set in Stk.h. Many STK classes use the sample rate during instantiation. Therefore, if you wish to use a rate which is different from the default rate, it is imperative that it be set BEFORE STK objects are instantiated.

4.66.3 Member Data Documentation

4.66.3.1 const STK_FORMAT Stk::STK_SINT8 [static]

-128 to +127

4.66.3.2 const STK_FORMAT Stk::STK_SINT16 [static]

-32768 to +32767

4.66.3.3 const STK_FORMAT Stk::STK_SINT32 [static]

-2147483648 to +2147483647.

4.66.3.4 const STK_FORMAT Stk::STK_FLOAT32 [static]

Normalized between plus/minus 1.0.

4.66.3.5 const STK_FORMAT Stk::STK_FLOAT64 [static]

Normalized between plus/minus 1.0.
The documentation for this class was generated from the following file:

- Stk.h
4.67  StkError Class Reference

STK error handling class.
#include <Stk.h>

Public Methods

- StkError(const char *p, TYPE tipe=StkError::UNSPECIFIED)
  The constructor.

- virtual ~StkError(void)
  The destructor.

- virtual void printMessage(void)
  Prints "thrown" error message to stdout.

- virtual const TYPE& getType(void)
  Returns the "thrown" error message TYPE.

- virtual const char* getMessage(void) const
  Returns the "thrown" error message string.

4.67.1  Detailed Description

STK error handling class.
This is a fairly abstract exception handling class. There could be sub-classes to take care of more specific error conditions ... or not.
The documentation for this class was generated from the following file:

- Stk.h
4.68 SubNoise Class Reference

STK sub-sampled noise generator.

#include <SubNoise.h>

Inheritance diagram for SubNoise::

```
Stk
  ↓
Noise
  ↓
SubNoise
```

Public Methods

- **SubNoise** (int subRate=16)
  
  Default constructor sets sub-sample rate to 16.

- **~SubNoise** ()
  
  Class destructor.

- int **subRate** (void) const
  
  Return the current sub-sampling rate.

- void **setRate** (int subRate)
  
  Set the sub-sampling rate.

- MY_FLOAT **tick** ()
  
  Return a sub-sampled random number between -1.0 and 1.0.

4.68.1 Detailed Description

STK sub-sampled noise generator.

Generates a new random number every ”rate” ticks using the C rand() function.

The quality of the rand() function varies from one OS to another.


The documentation for this class was generated from the following file:

---

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
• SubNoise.h
4.69 Table Class Reference

STK table lookup class.
#include <Table.h>
Inheritance diagram for Table:

```
Stk
  
  Table
```

Public Methods

- **Table (char *fileName)**
  
  *Constructor loads the data from fileName.*

- **~Table ()**
  
  *Class destructor.*

- **long getLength () const**
  
  *Return the number of elements in the table.*

- **MY_FLOAT lastOut () const**
  
  *Return the last output value.*

- **MY_FLOAT tick (MY_FLOAT index)**
  
  *Return the table value at position index.*

- **MY_FLOAT *tick (MY_FLOAT *vector, unsigned int vectorSize)**
  
  *Take vectorSize index positions and return the corresponding table values in vector.*

4.69.1 Detailed Description

STK table lookup class.

This class loads a table of floating-point doubles, which are assumed to be in big-endian format. Linear interpolation is performed for fractional lookup indexes.
An \texttt{StkError} will be thrown if the table file is not found.


4.69.2 Member Function Documentation

4.69.2.1 MY\_FLOAT Table::tick (MY\_FLOAT \textit{index})

Return the table value at position \textit{index}.

Linear interpolation is performed if \textit{index} is fractional.

The documentation for this class was generated from the following file:

- \texttt{Table.h}
4.70 TcpWvIn Class Reference

STK internet streaming input class.
#include <TcpWvIn.h>

Inheritance diagram for TcpWvIn:

```
Stk
    ↓
WvIn
    ↓
TcpWvIn
```

Public Methods

- **TcpWvIn**(int port=2006)
  Default constructor starts a socket server. If not specified, the server is associated with port 2006.

- **~TcpWvIn**()
  Class destructor.

- void **listen**(unsigned int nChannels=1, Stk::STK_FORMAT format=STK_SINT16)
  Listen for a (new) connection with specified data channels and format.

- bool **isConnected**(void)
  Returns TRUE is an input connection exists or input data remains in the queue.

- MY_FLOAT **lastOut**(void)\ const
  Return the average across the last output sample frame.

- MY_FLOAT **tick**(void)
  Read out the average across one sample frame of data.

- MY_FLOAT\* **tick**(MY_FLOAT\* vector, unsigned int vectorSize)
  Read out vectorSize averaged sample frames of data in vector.

- const MY_FLOAT\* **lastFrame**(void)\ const

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
Return a pointer to the last output sample frame.

- const MY_FLOAT*\texttt{tickFrame}(\texttt{void})

  Return a pointer to the next sample frame of data.

- MY_FLOAT*\texttt{tickFrame}(MY_FLOAT*\texttt{frameVector}, unsigned int frames)

  Read out sample frames of data to \texttt{frameVector}.

4.70.1 Detailed Description

STK internet streaming input class.

This protected Wvin subclass can read streamed data over a network via a TCP socket connection. The data is assumed in big-endian, or network, byte order.

TcpWvIn supports multi-channel data in interleaved format. It is important to distinguish the \texttt{tick()} methods, which return samples produced by averaging across sample frames, from the \texttt{tickFrame()} methods, which return pointers to multi-channel sample frames. For single-channel data, these methods return equivalent values.

This class starts a socket server, which waits for a single remote connection. The default data type for the incoming stream is signed 16-bit integers, though any of the defined STK_FORMATs are permissible.


4.70.2 Constructor & Destructor Documentation

4.70.2.1 TcpWvIn::TcpWvIn (int\texttt{port} = 2006)

Default constructor starts a socket server. If not specified, the server is associated with port 2006.

An \texttt{StkError} will be thrown if an error occurs while initializing the input thread or starting the socket server.

4.70.3 Member Function Documentation

4.70.3.1 void TcpWvIn::\texttt{listen} (unsigned int\texttt{nChannels} = 1,

Stk::STK_FORMAT\texttt{format} = \texttt{STK_SINT16})

Listen for a (new) connection with specified data channels and format.

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
An `StkError` will be thrown a socket error or an invalid function argument.

### 4.70.3.2 `bool TcpWvIn::isConnected (void)`

Returns TRUE is an input connection exists or input data remains in the queue. This method will not return FALSE after an input connection has been closed until all buffered input data has been read out.

The documentation for this class was generated from the following file:

- `TcpWvIn.h`
4.71 TcpWvOut Class Reference

STK internet streaming output class.

```cpp
#include <TcpWvOut.h>
```

Inheritance diagram for TcpWvOut:

![Inheritance Diagram](image)

Public Methods

- **TcpWvOut()**
  
  Default constructor ... the socket is not instantiated.

- **TcpWvOut(int port, const char *hostname="localhost", unsigned int nChannels=1, Stk::STK_FORMAT format=STK_SINT16)**
  
  Overloaded constructor which opens a network connection during instantiation.

- **~TcpWvOut()**
  
  Class destructor.

- **void connect(int port, const char *hostname="localhost", unsigned int nChannels=1, Stk::STK_FORMAT format=STK_SINT16)**
  
  Connect to the specified host and port and prepare to stream nChannels of data in the given data format.

- **void disconnect(void)**
  
  If a connection is open, write out remaining samples in the queue and then disconnect.

- **unsigned long getFrames() const**
  
  Return the number of sample frames output.

- **MY_FLOAT getTime() const**
  
  Return the number of seconds of data output.
4.71 TcpWvOut Class Reference

- **void tick**(MY_FLOAT sample)
  
  Output a single sample to all channels in a sample frame.

- **void tick**(const MY_FLOAT *vector, unsigned int vectorSize)
  
  Output each sample in vector to all channels in vectorSize sample frames.

- **void tickFrame**(const MY_FLOAT *frameVector, unsigned int frames=1)
  
  Output the frameVector of sample frames of the given length.

### 4.71.1 Detailed Description

STK internet streaming output class.

This protected WvOut subclass can stream data over a network via a TCP socket connection. The data is converted to big-endian byte order, if necessary, before being transmitted.

TcpWvOut supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which output single samples to all channels in a sample frame, from the tickFrame() method, which takes a pointer to multi-channel sample frame data.

This class connects to a socket server, the port and IP address of which must be specified as constructor arguments. The default data type is signed 16-bit integers but any of the defined STK_FORMATs are permissible.


### 4.71.2 Constructor & Destructor Documentation

**4.71.2.1 TcpWvOut::TcpWvOut**(int port, const char *hostname = "localhost", unsigned int nChannels = 1, Stk::STK_FORMAT format = STK_SINT16)

Overloaded constructor which opens a network connection during instantiation. An StkError is thrown if a socket error occurs or an invalid argument is specified.
4.71.3 Member Function Documentation

4.71.3.1 void TcpWvOut::connect (int port, const char * hostname = "localhost", unsigned int nChannels = 1, Stk::STK_FORMAT format = STK_SINT16)

Connect to the specified host and port and prepare to stream nChannels of data in the given data format.

An StkError is thrown if a socket error occurs or an invalid argument is specified.

4.71.3.2 void TcpWvOut::tick (MY_FLOAT sample) [virtual]

Output a single sample to all channels in a sample frame.

An StkError is thrown if a socket write error occurs.

Reimplemented from WvOut

4.71.3.3 void TcpWvOut::tick (const MY_FLOAT * vector, unsigned int vectorSize) [virtual]

Output each sample in vector to all channels in vectorSize sample frames.

An StkError is thrown if a socket write error occurs.

Reimplemented from WvOut

4.71.3.4 void TcpWvOut::tickFrame (const MY_FLOAT * frameVector, unsigned int frames = 1) [virtual]

Output the frameVector of sample frames of the given length.

An StkError is thrown if a socket write error occurs.

Reimplemented from WvOut

The documentation for this class was generated from the following file:

- TcpWvOut.h
4.72 Thread Class Reference

STK thread class.

```cpp
#include <Thread.h>
```

Inheritance diagram for Thread::

```
   Stk
     
    Thread
```

Public Methods

- **Thread()**
  
  Default constructor.

- **~Thread()**
  
  The class destructor waits indefinitely for the thread to end before returning.

- **bool start(THREAD_FUNCTION routine, void *ptr=NULL)**
  
  Begin execution of the thread routine. Upon success, TRUE is returned.

- **bool wait(long milliseconds=-1)**
  
  Wait the specified number of milliseconds for the thread to terminate. Return TRUE on success.

Static Public Methods

- **void test(void)**
  
  Test for a thread cancellation request.

4.72.1 Detailed Description

STK thread class.

This class provides a uniform interface for cross-platform threads. On unix systems, the pthread library is used. Under Windows, the C runtime threadex functions are used.
4.72.2 Member Function Documentation

4.72.2.1 bool Thread::start (THREAD_FUNCTION routine, void *
ptr = NULL)

Begin execution of the thread routine. Upon success, TRUE is returned.
The thread routine can be passed an argument via ptr. If the thread cannot be
created, the return value is FALSE.

4.72.2.2 bool Thread::wait (long milliseconds = -1)

Wait the specified number of milliseconds for the thread to terminate. Return
TRUE on success.

If the specified time value is negative, the function will block indefinitely. Other-
wise, the function will block up to a maximum of the specified time. A return
value of FALSE indicates the thread did not terminate within the specified time
limit.

The documentation for this class was generated from the following file:

• Thread.h
4.73 TubeBell Class Reference

STK tubular bell (orchestral chime) FM synthesis instrument.

#include <TubeBell.h>

Inheritance diagram for TubeBell::

```
Stk
  Instrmnt
    FM
      TubeBell
```

Public Methods

- **TubeBell()**
  
  Class constructor.

- **~TubeBell()**
  
  Class destructor.

- **void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)**
  
  Start a note with the given frequency and amplitude.

- **MY_FLOAT tick()**
  
  Compute one output sample.

4.73.1 Detailed Description

STK tubular bell (orchestral chime) FM synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.

```
  Algorithm 5 is : 4->3--\n                     + --> Out
                  2->1--/
```
Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- ADSR 2 & 4 Target = 128

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- TubeBell.h
4.74 TwoPole Class Reference

STK two-pole filter class.

```cpp
#include <TwoPole.h>
```

Inheritance diagram for TwoPole::

```
    Stk
   /   
  /    
Filter
   :   :
   :   :
TwoPole
```

Public Methods

- **TwoPole()**
  Default constructor creates a second-order pass-through filter.

- **~TwoPole()**
  Class destructor.

- **void clear()**
  Clears the internal states of the filter.

- **void setB0(MY_FLOAT b0)**
  Set the b[0] coefficient value.

- **void setA1(MY_FLOAT a1)**
  Set the a[1] coefficient value.

- **void setA2(MY_FLOAT a2)**
  Set the a[2] coefficient value.

- **void setResonance(MY_FLOAT frequency, MY_FLOAT radius, bool normalize=FALSE)**
  Sets the filter coefficients for a resonance at frequency (in Hz).

- **void setGain(MY_FLOAT theGain)**
  Set the filter gain.
• MY_FLOAT getGain (void) const
  Return the current filter gain.

• MY_FLOAT lastOut (void) const
  Return the last computed output value.

• MY_FLOAT tick (MY_FLOAT sample)
  Input one sample to the filter and return one output.

• MY_FLOAT *tick (MY_FLOAT *vector, unsigned int vectorSize)
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.74.1 Detailed Description

STK two-pole filter class.
This protected Filter subclass implements a two-pole digital filter. A method is provided for creating a resonance in the frequency response while maintaining a nearly constant filter gain.


4.74.2 Member Function Documentation

4.74.2.1 void TwoPole::setResonance (MY_FLOAT frequency, MY_FLOAT radius, bool normalize = FALSE)

Sets the filter coefficients for a resonance at frequency (in Hz).

This method determines the filter coefficients corresponding to two complex-conjugate poles with the given frequency (in Hz) and radius from the z-plane origin. If normalize is true, the coefficients are then normalized to produce unity gain at frequency (the actual maximum filter gain tends to be slightly greater than unity when radius is not close to one). The resulting filter frequency response has a resonance at the given frequency. The closer the poles are to the unit-circle (radius close to one), the narrower the resulting resonance width. An unstable filter will result for radius >= 1.0. For a better resonance filter, use a BiQuad filter.

See also:
  BiQuad filter class
void TwoPole::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented from Filter

The documentation for this class was generated from the following file:

- TwoPole.h
4.75 TwoZero Class Reference

STK two-zero filter class.

```cpp
#include <TwoZero.h>
```

Inheritance diagram for TwoZero::

```
  Stk
   |
   Filter
   |
   TwoZero
```

Public Methods

- `TwoZero ()`
  *Default constructor creates a second-order pass-through filter.*

- `~TwoZero ()`
  *Class destructor.*

- `void clear (void)`
  *Clears the internal states of the filter.*

- `void setB0 (MY_FLOAT b0)`
  *Set the b[0] coefficient value.*

- `void setB1 (MY_FLOAT b1)`
  *Set the b[1] coefficient value.*

- `void setB2 (MY_FLOAT b2)`
  *Set the b[2] coefficient value.*

- `void setNotch (MY_FLOAT frequency, MY_FLOAT radius)`
  *Sets the filter coefficients for a "notch" at frequency (in Hz).*

- `void setGain (MY_FLOAT theGain)`
  *Set the filter gain.*

- `MY_FLOAT getGain (void) const`
4.75 TwoZero Class Reference

Return the current filter gain.

- MY_FLOAT lastOut (void) const
  Return the last computed output value.

- MY_FLOAT tick (MY_FLOAT sample)
  Input one sample to the filter and return one output.

- MY_FLOAT * tick (MY_FLOAT *vector, unsigned int vectorSize)
  Input vectorSize samples to the filter and return an equal number of outputs in vector.

4.75.1 Detailed Description

STK two-zero filter class.

This protected Filter subclass implements a two-zero digital filter. A method is provided for creating a "notch" in the frequency response while maintaining a constant filter gain.


4.75.2 Member Function Documentation

4.75.2.1 void TwoZero::setNotch (MY_FLOAT frequency, MY_FLOAT radius)

Sets the filter coefficients for a "notch" at frequency (in Hz).

This method determines the filter coefficients corresponding to two complex-conjugate zeros with the given frequency (in Hz) and radius from the z-plane origin. The coefficients are then normalized to produce a maximum filter gain of one (independent of the filter gain parameter). The resulting filter frequency response has a "notch" or anti-resonance at the given frequency. The closer the zeros are to the unit-circle (radius close to or equal to one), the narrower the resulting notch width.

4.75.2.2 void TwoZero::setGain (MY_FLOAT theGain) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values. The default gain value is 1.0.

Reimplemented from Filter.
The documentation for this class was generated from the following file:

- [TwoZero.h](#)
4.76 Vector3D Class Reference

STK 3D vector class.

#include <Vector3D.h>

Public Methods

• Vector3D (double initX=0.0, double initY=0.0, double initZ=0.0)
  Default constructor taking optional initial X, Y, and Z values.

• ~Vector3D ()
  Class destructor.

• double getX ()
  Get the current X value.

• double getY ()
  Get the current Y value.

• double getZ ()
  Get the current Z value.

• double getLength ()
  Calculate the vector length.

• void setXYZ (double anX, double aY, double aZ)
  Set the X, Y, and Z values simultaneously.

• void setX (double aval)
  Set the X value.

• void setY (double aval)
  Set the Y value.

• void setZ (double aval)
  Set the Z value.
4.76.1 Detailed Description

STK 3D vector class.

This class implements a three-dimensional vector.


The documentation for this class was generated from the following file:

- Vector3D.h
4.77 Voicer Class Reference

STK voice manager class.

```
#include <Voicer.h>
```

Inheritance diagram for Voicer:

```
  Stk
   ↓
  Voicer
```

Public Methods

- **Voicer**(int maxInstruments, MY_FLOAT decayTime=0.2)
  
  *Class constructor taking the maximum number of instruments to control and an optional note decay time (in seconds).*

- **∼Voicer**()
  
  *Class destructor.*

- void **addInstrument**(Instrmnt ∗instrument, int channel=0)
  
  *Add an instrument with an optional channel number to the voice manager.*

- void **removeInstrument**(Instrmnt ∗instrument)
  
  *Remove the given instrument pointer from the voice manager’s control.*

- long **noteOn**(MY_FLOAT noteNumber, MY_FLOAT amplitude, int channel=0)
  
  *Initiate a noteOn event with the given note number and amplitude and return a unique note tag.*

- void **noteOff**(MY_FLOAT noteNumber, MY_FLOAT amplitude, int channel=0)
  
  *Send a noteOff to all voices having the given noteNumber and optional channel (default channel = 0).*

- void **noteOff**(long tag, MY_FLOAT amplitude)
  
  *Send a noteOff to the voice with the given note tag.*

- void **setFrequency**(MY_FLOAT noteNumber, int channel=0)
Send a frequency update message to all voices assigned to the optional channel argument (default channel = 0).

- void setFrequency(long tag, MY_FLOAT noteNumber)
  Send a frequency update message to the voice with the given note tag.

- void pitchBend(MY_FLOAT value, int channel=0)
  Send a pitchBend message to all voices assigned to the optional channel argument (default channel = 0).

- void pitchBend(long tag, MY_FLOAT value)
  Send a pitchBend message to the voice with the given note tag.

- void controlChange(int number, MY_FLOAT value, int channel=0)
  Send a controlChange to all instruments assigned to the optional channel argument (default channel = 0).

- void controlChange(long tag, int number, MY_FLOAT value)
  Send a controlChange to the voice with the given note tag.

- void silence(void)
  Send a noteOff message to all existing voices.

- MY_FLOAT tick()
  Mix the output for all sounding voices.

- MY_FLOAT *tick(MY_FLOAT *vector, unsigned int vectorSize)
  Computer vectorSize output mixes and return them in vector.

- MY_FLOAT lastOut() const
  Return the last output value.

### 4.77.1 Detailed Description

STK voice manager class.

This class can be used to manage a group of STK instrument classes. Individual voices can be controlled via unique note tags. Instrument groups can be controlled by channel number.

A previously constructed STK instrument class is linked with a voice manager using the `addInstrument()` function. An optional channel number argument can be specified to the `addInstrument()` function as well (default channel = 0). The
voice manager does not delete any instrument instances ... it is the responsibility of the user to allocate and deallocate all instruments.

The \texttt{tick} function returns the mix of all sounding voices. Each \texttt{noteOn} returns a unique tag (credits to the NeXT MusicKit), so you can send control changes to specific voices within an ensemble. Alternately, control changes can be sent to all voices on a given channel.


### 4.77.2 Member Function Documentation

#### 4.77.2.1 \texttt{void Voicer::addInstrument (Instrmnt \* instrument, int channel = 0)}

Add an instrument with an optional channel number to the voice manager.

A set of instruments can be grouped by channel number and controlled via the functions which take a channel number argument.

#### 4.77.2.2 \texttt{void Voicer::removeInstrument (Instrmnt \* instrument)}

Remove the given instrument pointer from the voice manager’s control.

It is important that any instruments which are to be deleted by the user while the voice manager is running be first removed from the manager’s control via this function!!

#### 4.77.2.3 \texttt{long Voicer::noteOn (MY_FLOAT noteNumber, MY_FLOAT amplitude, int channel = 0)}

Initiate a \texttt{noteOn} event with the given note number and amplitude and return a unique note tag.

Send the \texttt{noteOn} message to the first available unused voice. If all voices are sounding, the oldest voice is interrupted and sent the \texttt{noteOn} message. If the optional channel argument is non-zero, only voices on that channel are used. If no voices are found for a specified non-zero channel value, the function returns -1. The amplitude value should be in the range 0.0 - 128.0.

#### 4.77.2.4 \texttt{void Voicer::noteOff (MY_FLOAT noteNumber, MY_FLOAT amplitude, int channel = 0)}

Send a \texttt{noteOff} to all voices having the given noteNumber and optional channel (default channel = 0).
The amplitude value should be in the range 0.0 - 128.0.

4.77.2.5 void Voicer::noteOff (long tag, MY_FLOAT amplitude)
Send a noteOff to the voice with the given note tag.
The amplitude value should be in the range 0.0 - 128.0.

4.77.2.6 void Voicer::setFrequency (MY_FLOAT noteNumber, int channel = 0)
Send a frequency update message to all voices assigned to the optional channel argument (default channel = 0).
The noteNumber argument corresponds to a MIDI note number, though it is a floating-point value and can range beyond the normal 0-127 range.

4.77.2.7 void Voicer::setFrequency (long tag, MY_FLOAT noteNumber)
Send a frequency update message to the voice with the given note tag.
The noteNumber argument corresponds to a MIDI note number, though it is a floating-point value and can range beyond the normal 0-127 range.

The documentation for this class was generated from the following file:

- Voicer.h

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
4.78 VoicForm Class Reference

Four formant synthesis instrument.

#include <VoicForm.h>

Inheritance diagram for VoicForm::

```
VoicForm

Instrmnt

Stk
```

Public Methods

- `VoicForm()`
  
  Class constructor, taking the lowest desired playing frequency.

- `~VoicForm()`
  
  Class destructor.

- `void clear()`
  
  Reset and clear all internal state.

- `void setFrequency(MY_FLOAT frequency)`
  
  Set instrument parameters for a particular frequency.

- `bool setPhoneme(const char *phoneme)`
  
  Set instrument parameters for the given phoneme. Returns FALSE if phoneme not found.

- `void setVoiced(MY_FLOAT vGain)`
  
  Set the voiced component gain.

- `void setUnVoiced(MY_FLOAT nGain)`
  
  Set the unvoiced component gain.

- `void setFilterSweepRate(int whichOne, MY_FLOAT rate)`
  
  Set the sweep rate for a particular formant filter (0-3).
• void setPitchSweepRate (MY_FLOAT rate)
  
  Set voiced component pitch sweep rate.
• void speak ()
  
  Start the voice.
• void quiet ()
  
  Stop the voice.
• void noteOn (MY_FLOAT frequency, MY_FLOAT amplitude)
  
  Start a note with the given frequency and amplitude.
• void noteOff (MY_FLOAT amplitude)
  
  Stop a note with the given amplitude (speed of decay).
• MY_FLOAT tick ()
  
  Compute one output sample.
• void controlChange (int number, MY_FLOAT value)
  
  Perform the control change specified by number and value (0.0 - 128.0).

4.78.1 Detailed Description

Four formant synthesis instrument.

This instrument contains an excitation singing wavetable (looping wave with random and periodic vibrato, smoothing on frequency, etc.), excitation noise, and four sweepable complex resonances.

Measured formant data is included, and enough data is there to support either parallel or cascade synthesis. In the floating point case cascade synthesis is the most natural so that’s what you’ll find here.

Control Change Numbers:

• Voiced/Unvoiced Mix = 2
• Vowel/Phoneme Selection = 4
• Vibrato Frequency = 11
• Vibrato Gain = 1
• Loudness (Spectral Tilt) = 128


The documentation for this class was generated from the following file:

• VoicForm.h
4.79 WaveLoop Class Reference

STK waveform oscillator class.

#include <WaveLoop.h>

Inheritance diagram for WaveLoop::

```
    Stk
     |
     WvIn
     |
    WaveLoop
```

Public Methods

- **WaveLoop**(const char *fileName, bool raw=FALSE)
  
  *Class constructor.*

- virtual **~WaveLoop**()
  
  *Class destructor.*

- void **setFrequency**(MY_FLOAT aFrequency)
  
  *Set the data interpolation rate based on a looping frequency.*

- void **addTime**(MY_FLOAT aTime)
  
  *Increment the read pointer by aTime samples, modulo file size.*

- void **addPhase**(MY_FLOAT anAngle)
  
  *Increment current read pointer by anAngle, relative to a looping frequency.*

- void **addPhaseOffset**(MY_FLOAT anAngle)
  
  *Add a phase offset to the current read pointer.*

- const MY_FLOAT* **tickFrame**(void)
  
  *Return a pointer to the next sample frame of data.*
4.79.1 Detailed Description

STK waveform oscillator class.

This class inherits from WvIn and provides audio file looping functionality.

WaveLoop supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which return samples produced by averaging across sample frames, from the tickFrame() methods, which return pointers to multi-channel sample frames. For single-channel data, these methods return equivalent values.


4.79.2 Member Function Documentation

4.79.2.1 void WaveLoop::setFrequency (MY_FLOAT aFrequency)

Set the data interpolation rate based on a looping frequency.

This function determines the interpolation rate based on the file size and the current Stk::sampleRate. The aFrequency value corresponds to file cycles per second. The frequency can be negative, in which case the loop is read in reverse order.

4.79.2.2 void WaveLoop::addPhase (MY_FLOAT anAngle)

Increment current read pointer by anAngle, relative to a looping frequency.

This function increments the read pointer based on the file size and the current Stk::sampleRate. The anAngle value is a multiple of file size.

4.79.2.3 void WaveLoop::addPhaseOffset (MY_FLOAT anAngle)

Add a phase offset to the current read pointer.

This function determines a time offset based on the file size and the current Stk::sampleRate. The anAngle value is a multiple of file size.

The documentation for this class was generated from the following file:

- WaveLoop.h
4.80 Whistle Class Reference

STK police/referee whistle instrument class.

```cpp
#include <Whistle.h>
```

Inheritance diagram for Whistle:

```
Stk
   
Instrmnt
   
Whistle
```

Public Methods

- `Whistle()`  
  Class constructor.

- `~Whistle()`  
  Class destructor.

- `void clear()`  
  Reset and clear all internal state.

- `void setFrequency(MY_FLOAT frequency)`  
  Set instrument parameters for a particular frequency.

- `void startBlowing(MY_FLOAT amplitude, MY_FLOAT rate)`  
  Apply breath velocity to instrument with given amplitude and rate of increase.

- `void stopBlowing(MY_FLOAT rate)`  
  Decrease breath velocity with given rate of decrease.

- `void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)`  
  Start a note with the given frequency and amplitude.

- `void noteOff(MY_FLOAT amplitude)`  
  Stop a note with the given amplitude (speed of decay).

- `MY_FLOAT tick()`
Compute one output sample.

- void controlChange(int number, MY_FLOAT value)
  
  Perform the control change specified by number and value (0.0 - 128.0).

### 4.80.1 Detailed Description

STK police/referee whistle instrument class.

This class implements a hybrid physical/spectral model of a police whistle (a la Cook).

Control Change Numbers:

- Noise Gain = 4
- Fipple Modulation Frequency = 11
- Fipple Modulation Gain = 1
- Blowing Frequency Modulation = 2
- Volume = 128


The documentation for this class was generated from the following file:

- Whistle.h
4.81 Wurley Class Reference

STK Wurlitzer electric piano FM synthesis instrument.

```cpp
#include <Wurley.h>
```

Inheritance diagram for Wurley:

```
    Stk
   /   
Instrmnt  
   /     
    FM
   / 
Wurley
```

Public Methods

- `Wurley()`
  
  *Class constructor.*

- `~Wurley()`
  
  *Class destructor.*

- `void setFrequency(MY_FLOAT frequency)`
  
  *Set instrument parameters for a particular frequency.*

- `void noteOn(MY_FLOAT frequency, MY_FLOAT amplitude)`
  
  *Start a note with the given frequency and amplitude.*

- `MY_FLOAT tick()`
  
  *Compute one output sample.*

4.81.1 Detailed Description

STK Wurlitzer electric piano FM synthesis instrument.

This class implements two simple FM Pairs summed together, also referred to as algorithm 5 of the TX81Z.
Algorithm 5 is:
\[
\begin{align*}
4 & \rightarrow 3 \rightarrow 2 \\
\text{+} & \rightarrow \text{Out} \\
2 & \rightarrow 1 \rightarrow /
\end{align*}
\]

Control Change Numbers:

- Modulator Index One = 2
- Crossfade of Outputs = 4
- LFO Speed = 11
- LFO Depth = 1
- **ADSR 2 & 4 Target = 128**

The basic Chowning/Stanford FM patent expired in 1995, but there exist follow-on patents, mostly assigned to Yamaha. If you are of the type who should worry about this (making money) worry away.


The documentation for this class was generated from the following file:

- **Wurley.h**
4.82 WvIn Class Reference

STK audio data input base class.

```cpp
#include <WvIn.h>
```

Inheritance diagram for WvIn:

```
Stk
  
WvIn
  
RtWvIn  TcpWvIn  WaveLoop
```

Public Methods

- `WvIn()`
  Default constructor.

- `WvIn(const char *fileName, bool raw=FALSE)`
  Overloaded constructor for file input.

- `virtual ~WvIn()`
  Class destructor.

- `void openFile(const char *fileName, bool raw=FALSE)`
  Open the specified file and load its data.

- `void closeFile(void)`
  If a file is open, close it.

- `void reset(void)`
  Clear outputs and reset time (file pointer) to zero.

- `void normalize(void)`
  Normalize data to a maximum of ±1.0.

- `void normalize(MY_FLOAT peak)`
  Normalize data to a maximum of ±peak.

- `unsigned long getSize(void) const`
Return the file size in sample frames.

• unsigned int getChannels (void) const
  Return the number of audio channels in the file.

• MY_FLOAT getFileRate (void) const
  Return the input file sample rate in Hz (not the data read rate).

• bool isFinished (void) const
  Query whether reading is complete.

• void setRate (MY_FLOAT aRate)
  Set the data read rate in samples. The rate can be negative.

• virtual void addTime (MY_FLOAT aTime)
  Increment the read pointer by aTime samples.

• void setInterpolate (bool doInterpolate)
  Turn linear interpolation on/off.

• virtual MY_FLOAT lastOut (void) const
  Return the average across the last output sample frame.

• virtual MY_FLOAT tick (void)
  Read out the average across one sample frame of data.

• virtual MY_FLOAT* tick (MY_FLOAT* vector, unsigned int vectorSize)
  Read out vectorSize averaged sample frames of data in vector.

• virtual const MY_FLOAT* lastFrame (void) const
  Return a pointer to the last output sample frame.

• virtual const MY_FLOAT* tickFrame (void)
  Return a pointer to the next sample frame of data.

• virtual MY_FLOAT* tickFrame (MY_FLOAT* frameVector, unsigned int frames)
  Read out sample frames of data to frameVector.
4.82 WvIn Class Reference

4.82.1 Detailed Description

STK audio data input base class.

This class provides input support for various audio file formats. It also serves as a base class for "realtime" streaming subclasses.

WvIn loads the contents of an audio file for subsequent output. Linear interpolation is used for fractional "read rates".

WvIn supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which return samples produced by averaging across sample frames, from the tickFrame() methods, which return pointers to multi-channel sample frames. For single-channel data, these methods return equivalent values.

Small files are completely read into local memory during instantiation. Large files are read incrementally from disk. The file size threshold and the increment size values are defined in WvIn.h.

WvIn currently supports WAV, AIFF, SND (AU), MAT-file (Matlab), and STK RAW file formats. Signed integer (8-, 16-, and 32-bit) and floating-point (32- and 64-bit) data types are supported. Uncompressed data types are not supported. If using MAT-files, data should be saved in an array with each data channel filling a matrix row.


4.82.2 Constructor & Destructor Documentation

4.82.2.1 WvIn::WvIn (const char * fileName, bool raw = FALSE)

Overloaded constructor for file input.

An StkError will be thrown if the file is not found, its format is unknown, or a read error occurs.

4.82.3 Member Function Documentation

4.82.3.1 void WvIn::openFile (const char * fileName, bool raw = FALSE)

Open the specified file and load its data.

An StkError will be thrown if the file is not found, its format is unknown, or a read error occurs.
4.82.3.2  void WvIn::normalize (void)

Normalize data to a maximum of +/-1.0.

For large, incrementally loaded files with integer data types, normalization is computed relative to the data type maximum. No normalization is performed for incrementally loaded files with floating-point data types.

4.82.3.3  void WvIn::normalize (MY_FLOAT peak)

Normalize data to a maximum of +/-peak.

For large, incrementally loaded files with integer data types, normalization is computed relative to the data type maximum (peak/maximum). For incrementally loaded files with floating-point data types, direct scaling by peak is performed.

4.82.3.4  MY_FLOAT WvIn::getFileRate (void) const

Return the input file sample rate in Hz (not the data read rate).

WAV, SND, and AIF formatted files specify a sample rate in their headers. STK RAW files have a sample rate of 22050 Hz by definition. MAT-files are assumed to have a rate of 44100 Hz.

4.82.3.5  void WvIn::setRate (MY_FLOAT aRate)

Set the data read rate in samples. The rate can be negative.

If the rate value is negative, the data is read in reverse order.

4.82.3.6  void WvIn::setInterpolate (bool doInterpolate)

Turn linear interpolation on/off.

Interpolation is automatically off when the read rate is an integer value. If interpolation is turned off for a fractional rate, the time index is truncated to an integer value.

4.82.3.7  MY_FLOAT WvIn::tick (void)  [virtual]

Read out the average across one sample frame of data.

An StkError will be thrown if a file is read incrementally and a read error occurs.

Reimplemented in RtWvIn and TcpWvIn.
4.82.3.8  

MY_FLOAT * WvIn::tick (MY_FLOAT * vector, unsigned int vectorSize) [virtual]

Read out vectorSize averaged sample frames of data in vector.

An StkError will be thrown if a file is read incrementally and a read error occurs.

Reimplemented in RtWvIn and TcpWvIn.

4.82.3.9  

const MY_FLOAT * WvIn::tickFrame (void) [virtual]

Return a pointer to the next sample frame of data.

An StkError will be thrown if a file is read incrementally and a read error occurs.

Reimplemented in RtWvIn, TcpWvIn, and WaveLoop.

4.82.3.10  

MY_FLOAT * WvIn::tickFrame (MY_FLOAT * frameVector, unsigned int frames) [virtual]

Read out sample frames of data to frameVector.

An StkError will be thrown if a file is read incrementally and a read error occurs.

Reimplemented in RtWvIn and TcpWvIn.

The documentation for this class was generated from the following file:

- WvIn.h
4.83 WvOut Class Reference

STK audio data output base class.

#include <WvOut.h>

Inheritance diagram for WvOut:

```
Stk
    WvOut
          RtWvOut  TcpWvOut
```

Public Methods

- **WvOut()**
  
  Default constructor.

- **WvOut(const char *fileName, unsigned int nChannels=1, FILE_TYPE type=WVOUT_WAV, Stk::STK_FORMAT format=STK_SINT16)**
  
  Overloaded constructor used to specify a file name, type, and data format with this object.

- **virtual ~WvOut()**
  
  Class destructor.

- **void openFile (const char *fileName, unsigned int nChannels=1, FILE_TYPE type=WVOUT_WAV, Stk::STK_FORMAT format=STK_SINT16)**
  
  Create a file of the specified type and name and output samples to it in the given data format.

- **void closeFile (void)**
  
  If a file is open, write out samples in the queue and then close it.

- **unsigned long getFrames (void) const**
  
  Return the number of sample frames output.

- **MY_FLOAT getTime (void) const**
  
  Return the number of seconds of data output.
• virtual void tick (const MY_FLOAT sample)
  
  Output a single sample to all channels in a sample frame.

• virtual void tick (const MY_FLOAT *vector, unsigned int vectorSize)
  
  Output each sample in vector to all channels in vectorSize sample frames.

• virtual void tickFrame (const MY_FLOAT *frameVector, unsigned int frames=1)
  
  Output the frameVector of sample frames of the given length.

Static Public Attributes

• const FILE_TYPE WVOUT_RAW
• const FILE_TYPE WVOUT_WAV
• const FILE_TYPE WVOUT_SND
• const FILE_TYPE WVOUT_AIF
• const FILE_TYPE WVOUT_MAT

4.83.1 Detailed Description

STK audio data output base class.

This class provides output support for various audio file formats. It also serves as a base class for "realtime" streaming subclasses.

WvOut writes samples to an audio file. It supports multi-channel data in interleaved format. It is important to distinguish the tick() methods, which output single samples to all channels in a sample frame, from the tickFrame() method, which takes a pointer to multi-channel sample frame data.

WvOut currently supports WAV, AIFF, AIFC, SND (AU), MAT-file (Matlab), and STK RAW file formats. Signed integer (8-, 16-, and 32-bit) and floating-point (32- and 64-bit) data types are supported. STK RAW files use 16-bit integers by definition. MAT-files will always be written as 64-bit floats. If a data type specification does not match the specified file type, the data type will automatically be modified. Uncompressed data types are not supported.

Currently, WvOut is non-interpolating and the output rate is always Stk::sampleRate() by Perry R. Cook and Gary P. Scavone, 1995 - 2002.
4.83.2 Constructor & Destructor Documentation

4.83.2.1 WvOut::WvOut (const char *fileName, unsigned int nChannels = 1, FILE_TYPE type = WVOUT_WAV, Stk::STK_FORMAT format = STK_SINT16)

Overloaded constructor used to specify a file name, type, and data format with this object.

An StkError is thrown for invalid argument values or if an error occurs when initializing the output file.

4.83.3 Member Function Documentation

4.83.3.1 void WvOut::openFile (const char *fileName, unsigned int nChannels = 1, WvOut::FILE_TYPE type = WVOUT_WAV, Stk::STK_FORMAT format = STK_SINT16)

Create a file of the specified type and name and output samples to it in the given data format.

An StkError is thrown for invalid argument values or if an error occurs when initializing the output file.

4.83.3.2 void WvOut::tick (const MY_FLOAT sample) [virtual]

Output a single sample to all channels in a sample frame.

An StkError is thrown if a file read error occurs.

Reimplemented in RtWvOut and TcpWvOut.

4.83.3.3 void WvOut::tick (const MY_FLOAT *vector, unsigned int vectorSize) [virtual]

Output each sample in vector to all channels in vectorSize sample frames.

An StkError is thrown if a file read error occurs.

Reimplemented in RtWvOut and TcpWvOut.

4.83.3.4 void WvOut::tickFrame (const MY_FLOAT *frameVector, unsigned int frames = 1) [virtual]

Output the frameVector of sample frames of the given length.
An **StkError** is thrown if a file read error occurs.

Reimplemented in **RtWvOut** and **TcpWvOut**.

### 4.83.4 Member Data Documentation

**4.83.4.1** `const FILE_TYPE WvOut::WVOUT_RAW` [static]

STK RAW file type.

**4.83.4.2** `const FILE_TYPE WvOut::WVOUT_WAV` [static]

WAV file type.

**4.83.4.3** `const FILE_TYPE WvOut::WVOUT_SND` [static]

SND (AU) file type.

**4.83.4.4** `const FILE_TYPE WvOut::WVOUT_AIF` [static]

AIFF file type.

**4.83.4.5** `const FILE_TYPE WvOut::WVOUT_MAT` [static]

Matlab MAT-file type.

The documentation for this class was generated from the following file:

- `{WvOut.h}`
Chapter 5

STK Page Documentation

5.1 Class Documentation

- Class Hierarchy
- Class/Enum List
- File List
- Compound Members
5.2 Compiling

The Synthesis ToolKit can be used in a variety of ways, depending on your particular needs. Some people choose the classes they need for a particular project and copy those to their working directory. Others create Makefiles which compile project-specific class objects from common src and include directories. And still others like to compile and link to a common library of object files. STK was not designed with one particular style of use in mind.

5.3 "Realtime" vs. "Non-Realtime"

Most of the Synthesis ToolKit classes are platform independent. That means that they should compile on any reasonably current C++ compiler. The functionality needed for realtime audio and MIDI input/output, as well as realtime control message acquisition, is inherently platform and operating-system (OS) dependent. STK classes which require specific platform/OS support include RtAudio, RtWvOut, RtWvIn, RtDuplex, RtMidi, TcpWvIn, TcpWvOut, Socket, and Thread. These classes currently can only be compiled on Linux, Irix, Macintosh OS X, and Windows systems using the LINUX_OSS, LINUX_ALSA, IRIX_AL, MACOSX_CORE, WINDOWS_DS, or WINDOWS_ASIO preprocessor definitions.

Without the "realtime" classes, it is still possible to read SKINI scorefiles for control input and to read and write to/from a variety of audio file formats (WAV, SND, AIFF, MAT-file, and RAW). If compiling for a "little-endian" host processor, the LITTLE_ENDIAN preprocessor definition should be provided.

5.4 Unix Systems:

STK compiles with realtime support on the following flavors of the Unix operating system: Linux, Irix, and Macintosh OS X. Aside from differences in compilers, audio/MIDI APIs, and host endianness, the steps necessary to compile STK programs and classes on these platforms are the same. The following table summarizes these differences.
The available C++ compilers on any of these systems can vary.

One approach in using STK is to simply copy the class files needed for a particular program into a project directory. Taking the `sineosc.cpp` example from the previous tutorial chapter, it would be necessary to set up a directory that includes the files `sineosc.cpp`, the rawwave file `sinewave.raw` in a subdirectory called `rawwaves`, and the header and source files for the classes `Stk`, `WvIn`, `WaveLoop`, and `WvOut`. The program could then be compiled on a Linux system using the GNU g++ compiler as follows:

```
g++ -Wall -D__LITTLE_ENDIAN__ -o sineosc Stk.cpp WvIn.cpp WaveLoop.cpp WvOut.cpp sineosc.cpp
```

Note that the `sineosc.cpp` example does not make use of realtime audio or MIDI input/output classes. For programs using any of the STK realtime classes mentioned above, it is necessary to specify an audio/MIDI API preprocessor definition and link with the appropriate libraries or frameworks.

When working with a number of different projects that make use of ToolKit classes, the above approach can become cumbersome (especially when trying to synchronize with new STK releases). The example STK projects (e.g., demo, effects, ...) contain `Makefiles` (built by the configure script) which compile project-specific class objects from the distribution `src` and `include` directories. This approach makes it relatively easy when upgrading to a new STK release (by making path substitutions in the `Makefile` or by moving the projects to a similar relative path within the new STK source tree). A `Makefile` of this sort is provided in the `projects/examples` directory for compiling all the tutorial programs, as well as other example programs. To compile the `sineosc.cpp` program, for example, one need only type `make sineosc` from within the `projects/examples` directory.
5.4.1 Library Use:

The STK distribution provides a Makefile that can be used on Unix systems to build a static library. After unpacking the distribution (tar -xzf stk-4.x.tar.gz), run the configure script by typing ./configure from the top level distribution directory (see the INSTALL file in the same directory for more information). Then from within the src directory, type make. After a successful build, you may wish to move the library (libstk.a) and the contents of the include directory to standard library and include search paths on your system. For example, the linux RPM distribution of STK puts the library in /usr/lib/ and the STK header files in /usr/include/stk/.

Assuming the library is located in a standard search path and the header files are located in /usr/include/stk/, the sineosc.cpp example from the previous tutorial chapter can be compiled on a Linux system using the GNU g++ compiler as follows:

```
g++ -Wall -D__LITTLE_ENDIAN__ -I/usr/include/stk -o sineosc sineosc.cpp -lstk
```

With the header files in a standard search path, it is possible to modify the #include statements in the sineosc.cpp program as follows:

```
#include "stk/WaveLoop.h"
#include "stk/WvOut.h"
```

and then compile without an explicit include path argument to the compiler:

```
g++ -Wall -D__LITTLE_ENDIAN__ -o sineosc sineosc.cpp -lstk
```

5.5 Windows:

STK has been tested on Windows platforms using the Visual C++ compiler only. It is assumed here that you’re familiar with Visual C++ and its particular idiosyncrasies.

The approach when using Visual C++ is to build a project which includes the necessary ToolKit files from the distribution src and include directories. For the example program from the previous tutorial chapter, create a VC++ console application project, add the Stk, WvIn, WaveLoop, and WvOut class files, as well as sineosc.cpp, and make sure the sinewave.raw file is in the subdirectory rawwaves.

For programs using any of the STK realtime classes mentioned above, it is necessary to link with the DirectSound (dsound.lib), wimm.lib, and Wsock32.lib libraries, select the multithreaded library, and provide the _LITTLE_ENDIAN_ and _WINDOWS_DS_ preprocessor definitions.
For Steinberg ASIO support, use the __WINDOWS_ASIO__ preprocessor definition, include all the files in the src/asio/ directory (i.e., asio.h,cpp, asiodrivers.h,cpp, ...), and link with the wmm.lib, and Wsock32.lib libraries.
5.6 Control Input

Each Synthesis ToolKit instrument exposes its relevant control parameters via public functions such as `setFrequency()` and `controlChange()`. Programmers are free to implement the control scheme of their choice in exposing those parameters to the user.

A text-based control protocol called `SKINI` is provided with the Synthesis ToolKit. `SKINI` extends the MIDI protocol in incremental ways, providing a text-based messaging scheme in human-readable format and making use of floating-point numbers wherever possible. Each `SKINI` message consists of a message type (e.g., `NoteOn`, `PitchBend`), a time specification (absolute or delta), a channel number (scanned as a long integer), and a maximum of two subsequent message-specific field values. Knowing this, it should be relatively clear what the following `SKINI` “scorefile” specifies:

```
NoteOn 0.000082 2 55.0 82.3
NoteOff 1.000000 2 55.0 64.0
NoteOn 0.000082 2 69.0 82.8
StringDetune 0.100000 2 10.0
StringDetune 0.100000 2 30.0
StringDetune 0.100000 2 50.0
StringDetune 0.100000 2 40.0
StringDetune 0.100000 2 22.0
StringDetune 0.100000 2 12.0
NoteOff 1.000000 2 69.0 64.0
```

MIDI messages (with the exception of Sysex) are easily represented within the `SKINI` protocol.

The class `Messager` can be used to acquire and parse MIDI messages from a MIDI device and `SKINI` messages from STDIN and socket connections. Many of the example programs included with the ToolKit distribution use a `Messager` instance to accept control input from the accompanying tcl/tk graphical user interfaces, from external MIDI devices, or from `SKINI` scorefiles.

In the following example, we’ll modify the `bethree.cpp` program from the previous tutorial chapter and incorporate a `Messager` class to allow control via a `SKINI` scorefile.

```cpp
// controlbee.cpp

#include "BeeThree.h"
#include "RtWvOut.h"
#include "Messager.h"
#include "SKINI.msg"
#include <math.h>

int main()
{
```

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
// Set the global sample rate before creating class instances.
Stk::setSampleRate( 44100.0 );

Instrmnt *instrument = 0;
RtWvOut *output = 0;
Messager *messager = 0;
bool done = FALSE;

try {
   // Define and load the BeeThree instrument
   instrument = new BeeThree();

   // Define and open the default realtime output device for one-channel playback
   output = new RtWvOut(1);
}
catch (StkError &) {
   goto cleanup;
}

try {
   // Create a Messager instance to read from a redirected SKINI scorefile.
   messager = new Messager();
}
catch (StkError &) {
   goto cleanup;
}

// Play the instrument until the end of the scorefile.
int i, nTicks, type;
MY_FLOAT byte2, byte3, frequency;
while (!done) {

   // Look for new messages and return a delta time (in samples).
   type = messager->nextMessage();
   if (type < 0)
      done = TRUE;

   nTicks = messager->getDelta();
   try {
      for ( i=0; i<nTicks; i++ )
         output->tick( instrument->tick() );
   } catch (StkError &) {
      goto cleanup;
   }

   if ( type > 0 ) {
      // Process the new control message.
      byte2 = messager->getByteTwo();
      byte3 = messager->getByteThree();

      switch(type) {
      case __SK_NoteOn_:
         frequency = (MY_FLOAT) 220.0 * pow( 2.0, (byte2 - 57.0) / 12.0 );
instrument->noteOn( frequency, byte3 * ONE_OVER_128 );
break;

case __SK_NoteOff__:
    instrument->noteOff( byte3 * ONE_OVER_128 );
    break;

case __SK_ControlChange__:
    instrument->controlChange( (int) byte2, byte3 );
    break;

case __SK_AfterTouch__:
    instrument->controlChange( 128, byte2 );
    break;
}
}
}

cleanup:
    delete instrument;
    delete output;
    delete messager;

    return 0;
}

Assuming the program is compiled as controlbee and the SKINI scorefile bookert.ski is in the scores directory, the scorefile could be redirected to the program as:

controlbee < scores/bookert.ski

Only a few basic SKINI message type case statements are included in this example. It is easy to extend the program to support a much more elaborate set of instrument control parameters.

This example could also be easily extended to accept ”realtime” control input messages via STDIN, socket, or MIDI connections. The Messenger class constructor takes an optional argument consisting of a bitmask of the following options: STK_PIPE, STK_SOCKET, and/or STK_MIDI.

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
5.7 Download and Release Notes

Version 4.1.1, 24 October 2002

- Source distribution (1.2 MB tar/gzipped)
- Source with precompiled Windows binaries (2.0 MB tar/gzipped)
- Linux RPM (1.2 MB) Note: Library and Makefiles built for ALSA, though the rpm can be rebuilt to use OSS
- STK Manual (PDF) (1.62 MB) Note: HTML version already in /doc/html/ directory of distribution

5.8 Release Notes:

5.8.1 Version 4.1.1

- Bug fix in RtAudio for Macintosh OS X and Windows ASIO duplex operation.
- Windows ASIO fix in Stk.h.
- Documentation updates.
- Expanded tutorial.
- Fixed RtDuplex omission in src Makefile.

5.8.2 Version 4.1

- Macintosh OS X support added.
- New Whistle class.
- Added Voicer, SingWave, and VoicForm classes.
- Improvements/fixes to the banded waveguide instruments.
- Demo program now uses Voicer allowing polyphony.
- Demo tcl/tk scripts changed to use SKINI PitchChange instead of PitchBend.
- Demo program response to PitchBend modified to octave up/down.
- Several RtAudio fixes and improvements (OS X and Windows ASIO support added).
- Added nextOut() method to Delay classes.
- Documentation fixes for Reverb classes.
- RAWWAVE_PATH changed to include the "rawwave" directory.
- "configure" support added for unix systems.
- Multivoice flag (-n NUMBER) added as command line option to demo program.
- Sample rate flag added as command line option to example programs.
- Socket port number added as command line option to example programs.
5.8.3 Version 4.0

- New documentation and tutorial.
- Several new instruments, including Saxofony, BlowBotl, and StifKarp.
- New Stk base class, replacing Object class.
- New Filter class structure and methods.
- Extensive modifications to WvIn and WvOut class structures and methods.
- Looping functionality moved to WaveLoop (subclass of WvIn).
- Automatic file type detection in WvIn... hosed WavWvIn, AifWvIn, RawWavIn, SndWavIn, and MatWvIn subclasses.
- New file type specifier argument in WvOut... hosed WavWvOut, AifWvOut, RawWavOut, SndWavOut, and MatWvOut subclasses.
- Some simplifications of Messager class (was Controller).
- New independent RtAudio class.
- Extensive revisions in code and a significant number of API changes.

5.8.4 Version 3.2

- New input control handling class (Controller)
- Added AIFF file input/output support.
- New C++ error handling capabilities.
- New input/output internet streaming support (StrmWvIn/StrmWvOut).
- Added native ALSA support for linux.
- Added optional "device" argument to all "Rt" classes (audio and MIDI) and printout of devices when argument is invalid.
- WvIn classes rewritten to support very big files (incremental load from disk).
- Changed WvIn/WvOut classes to work with sample frame buffers.
- Fixed looping and negative rate calculations in WvIn classes.
- Fixed interpolation bug in RtWvIn.
- Windoze RtAudio code rewritten (thank Dave!).
- Simplified byte-swapping functions (in-place swapping).
- "Stereo-ized" RagaMatic.
- Miscellaneous renamings.
- Probably a bunch more fixes that I've long since forgotten about.

5.8.5 Version 3.1

- New RagaMatic project... very cool!!!
- Less clipping in the Shakers class.
- Added "microphone position" to Mandolin in STKdemo.
5.8 Release Notes:

- Fixed MIDI system message exclusion under Irix.
- Added a few bitmaps for the Shaker instruments.
- Made destructors virtual for Reverb.h, WvIn.h and Simple.h.
- Fixed bug setting delay length in DLineA when value too big.
- Fixed bug in WinMM realtime code (RTSoundIO).
- Added tick() method to BowTabl, JetTabl, and ReedTabl (same as lookup).
- Switched to pthread API on SGI platforms.
- Added some defines to Object.h for random number generation, FPU overflow checking, etc....
- A few minor changes, some bug fixes ... can’t remember all of them.

5.8.6 Version 3.0

- New define flags for OS and realtime dependencies (this will probably cause problems for old personal STK code, but it was necessary to make future ports easier).
- Expanded and cleaned the Shakers class.
- New BowedBar algorithm/class.
- Fixed Linux MIDI input bug.
- Fixed MIDI status masking problem in Windows.
- OS type defines now in Makefile.
- New RAWWAVE_PATH define in Object.h.
- Syntmono project pulled out to separate directory and cleaned up.
- Socketsing capabilities under Unix, as well as Windoze.
- Multiple simultaneous socket client connections to STK servers now possible.
- MD2SKINI now can merge MIDI and piped messages under Irix and Linux (for TCL->MD2SKINI->syntmono control).
- Defined INT16 and INT32 types and fixed various WvIn and WvOut classes.
- Updated MatWvIn and MatWvOut for new MAT-file documentation from Matlab.
- New demo Tcl/Tk GUI (TclDemo.tcl).
- Minor fixes to FM behavior.
- Added record/duplex capabilities to RTSoundIO (Linux, SGI, and Windoze).
- Fixed bugs in WavWvOut and MatWvOut header specifications.
- Added RawWvOut class.
- New WvIn class with RawWvIn, SndWvIn, WavWvIn, MatWvIn, and RTWvIn subclasses.
- Removed RawWave, RawShot, RawInterp, and RawLoop classes (supplanted by RawWvIn).
• Multi-channel data support in WvIn and WvOut classes using MY\_MULTI data type (pointer to MY\_FLOAT) and the methods mtick() and mlastOutput().
• Now writing to primary buffer under Windoze when allowed by hardware.
• Cleaned up Object.h a bit.
• Pulled various utility and thread functions out of syntmono.cpp (to aid readability of the code).

5.8.7 Version 2.02

• Created RawWave abstract class, with subclasses of RawLoop (looping rawwave oscillator), RawShot (non-looping, non-interpolating rawwave player ... used to be RawWvIn), and RawInterp (looping or non-looping, interpolating rawwave player ... used to be RawWave).
• Modified DrumSynt to correctly handle sample rates different than 22050 Hz.
• Modified syntmono parsing vs. tick routine so that some ticking occurs between each message. When multiple messages are waiting to be processed, the time between message updates is inversely proportional to the number of messages in the buffer.
• Fixed DirectSound playback bug in WinXX distribution. Sound was being played at 8-bit, 22 kHz in all cases. Playback is now 16-bit and dependent on SRATE.
• Fixed bug in MD2SKINI which prevented some NoteOff statements from being output.
• This distribution includes an example STK project, mus151, which demonstrates a means for keeping a user’s personal projects separate from the main distribution. This is highly recommended, in order to simplify upgrades to future STK releases.

5.8.8 Version 2

• Unification of the capabilities of STK across the various platforms. All of the previous SGI functionality has been ported to Linux and Windows, including realtime sound output and MIDI input.
• MIDI input (with optional time-stamping) supported on SGI, Linux (OSS device drivers only), and Windows operating systems. Time stamping under IRIX and Windows is quantized to millisecond and under Linux to hundredths of a second.
• Various Sound Output Options - .wav, .snd, and .mat (Matlab MAT-file) soundfile outputs are supported on all operating systems. I hacked out the MAT-file structure, so you don’t have to include any platform-specific libraries. Realtime sound output is provided as well, except under NeXTStep.
• **Multiple Reverberator Implementations** - Reverb subclasses of JCRv and NRev (popular reverberator implementations from CCRMA) have been written. Perry’s original reverb implementation still exists as PRCRev. All reverberators now take a T60 initializer argument.

• **MD2SKINI** - A program which parses a MIDI input stream and spits out SKINI code. The output of MD2SKINI is typically piped into an STK instrument executable (e.g., MD2SKINI | syntmono Clarinet -r -i). In addition, you can supply a filename argument to MD2SKINI and have it simultaneously record a SKINI score file for future reuse.

• **Modifications to Object.h for OS_TYPE compilation dependencies.** Makefile automatically determines OS_TYPE when invoked (if you have the GNU makefile utilities installed on your system).

• **A single distribution for all platforms.** The Unix and Windows versions have been merged into a single set of classes. Makefiles and Visual C++ workspace/project files are provided for compiling.
5.9 Hello Sine!

We'll begin our introduction to the Synthesis ToolKit with a simple sine-wave oscillator program. STK does not provide a specific oscillator for sine waves. Instead, it provides a generic waveform oscillator class, `WaveLoop`, which can load a variety of common file types. In this example, we load a sine “table” from an STK RAW file (defined as monophonic, 16-bit, big-endian data). We use the class `WvOut` to write the result to a 16-bit, WAV formatted audio file.

```
// sineosc.cpp

#include "WaveLoop.h"
#include "WvOut.h"

int main()
{
    // Set the global sample rate before creating class instances.
    Stk::setSampleRate( 44100.0 );

    // Define and load the sine wave file
    WaveLoop *input = new WaveLoop( "rawwaves/sinewave.raw", TRUE );
    input->setFrequency( 440.0 );

    // Define and open a 16-bit, one-channel WAV formatted output file
    output = new WvOut( "hellosine.wav", 1, WvOut::WVOUT_WAV, Stk::STK_SINT16 );

    // Run the oscillator for 40000 samples, writing to the output file
    int i;
    for ( i=0; i<40000; i++ ) {
        output->tick( input->tick() );
    }

    // Clean up
    delete input;
    delete output;

    return 0;
}
```

`WaveLoop` is a subclass of `WvIn`, which supports WAV, SND (AU), AIFF, MAT-file (Matlab), and RAW file formats with 8-, 16-, and 32-bit integer and 32- and 64-bit floating-point data types. `WvIn` provides interpolating, read once (“oneshot”) functionality, as well as methods for setting the read rate and read position.

The `WvIn` and `WvOut` classes are complementary, both supporting WAV, SND (AU), AIFF, MAT-file (Matlab), and RAW file formats with 8-, 16-, and 32-bit integer and 32- and 64-bit floating-point data types. However, `WvOut` does not perform data interpolation.

Nearly all STK classes implement `tick()` functions which take and/or return sample values. Within the `tick()` function, the fundamental sample calcula-
tions are performed for a given class. Most STK classes consume/generate a single sample per operation and their `tick()` method takes/returns each sample "by value". In addition, every class implementing a `tick()` function also provides an overloaded `tick()` function taking pointer and size arguments which can be used for vectorized computations.

The `WvIn` and `WvOut` classes support multi-channel sample frames. To distinguish single-sample frame operations from multi-channel frame operations, these classes also implement `tickFrame()` functions. When a `tick()` method is called for multi-channel data, frame averages are returned or the input sample is distributed across all channels of a sample frame.

Nearly all STK classes inherit from the `Stk` base class. `Stk` provides a static sample rate which is queried by subclasses as needed. Because many classes use the current sample rate value during instantiation, it is important that the desired value be set at the beginning of a program. The default STK sample rate is 22050 Hz.

Another primary concept that is somewhat obscured in this example concerns the data format in which sample values are passed and received. Audio and control signals throughout STK use a floating-point data type, the exact precision of which can be controlled via the `MY_FLOAT` #define statement in Stk.h. Thus, the ToolKit can use any normalization scheme desired. The base instruments and algorithms are implemented with a general audio sample dynamic maximum of +/-1.0, and the `WvIn` and `WvOut` classes and subclasses scale appropriately for DAC or soundfile input and output.

### 5.10 Error Handling

The ToolKit has some basic C++ error handling functionality built in. Classes which access files and/or hardware are most prone to runtime errors. To properly "catch" such errors, the above example should be rewritten as shown below.

```cpp
// sineosc.cpp

#include "WaveLoop.h"
#include "WvOut.h"

int main()
{
    // Set the global sample rate before creating class instances.
    Stk::setSampleRate( 44100.0 );

    WaveLoop *input = 0;
    WvOut *output = 0;

    try {
        // Define and load the sine wave file
```
input = new WaveLoop( "rawwaves/sinewave.raw", TRUE );

// Define and open a 16-bit, one-channel WAV formatted output file
output = new WvOut( "hellosine.wav", 1, WvOut::WVOUT_WAV, Stk::STK_SINT16 );
}
catch ( StkError & ) {
    goto cleanup;
}

input->setFrequency( 440.0 );

// Run the oscillator for 40000 samples, writing to the output file
for ( int i=0; i<40000; i++ ) {
    try {
        output->tick( input->tick() );
    }
    catch ( StkError & ) {
        goto cleanup;
    }
}

cleanup:
    delete input;
    delete output;
    return 0;
}

In this particular case, we simply exit the program if an error occurs (an error message is automatically printed to stderr). A more refined program might attempt to recover from or fix a particular problem and, if successful, continue processing. See the Class Documentation to determine which constructors and functions can throw an error.
5.11 General Information

References

- **ICMC99 Paper**
  A somewhat recent paper by Perry and Gary about the Synthesis ToolKit in C++.

- **SIGGRAPH96 Paper**
  A not-so-recent paper by Perry about the Synthesis ToolKit in C++.

- **Perry’s STK Web Page**
  This is a link to Perry Cook’s STK Web page. He has information about the Synthesis toolKit Instrument Network Interface (SKINI), the protocol used to control STK instruments, as well as a lot of other cool stuff.

What is the *Synthesis ToolKit*?

The Synthesis ToolKit in C++ (STK) is a set of open source audio signal processing and algorithmic synthesis classes written in C++. STK was designed to facilitate rapid development of music synthesis and audio processing software, with an emphasis on cross-platform functionality, realtime control, ease of use, and educational example code. The Synthesis ToolKit is extremely portable (it’s mostly platform-independent C and C++ code), and it’s completely user-extensible (all source included, no unusual libraries, and no hidden drivers). We like to think that this increases the chances that our programs will still work in another 5-10 years. In fact, the ToolKit has been working continuously for nearly 8 years now. STK currently runs with "realtime" support (audio and MIDI) on SGI (Irix), Linux, Macintosh OS X, and Windows computer platforms. Generic, non-realtime support has been tested under NeXTStep, Sun, and other platforms and should work with any standard C++ compiler.

The Synthesis ToolKit is free for non-commercial use. The only parts of the Synthesis ToolKit that are platform-dependent concern real-time audio and MIDI input and output, and that is taken care of with a few special classes. The interface for MIDI input and the simple Tcl/Tk graphical user interfaces (GUIs) provided is the same, so it’s easy to experiment in real time using either the GUIs or MIDI. The Synthesis ToolKit can generate simultaneous SND (AU), WAV, AIFF, and MAT-file output soundfile formats (as well as realtime sound output), so you can view your results using one of a large variety of sound/signal analysis tools already available (e.g. Snd, Cool Edit, Matlab).

What the *Synthesis ToolKit* is not.

The Synthesis ToolKit is not one particular program. Rather, it is a set of C++ classes that you can use to create your own programs. A few example
applications are provided to demonstrate some of the ways to use the classes. If you have specific needs, you will probably have to either modify the example programs or write a new program altogether. Further, the example programs don’t have a fancy GUI wrapper. If you feel the need to have a "drag and drop" graphical patching GUI, you probably don’t want to use the ToolKit. Spending hundreds of hours making platform-dependent graphics code would go against one of the fundamental design goals of the ToolKit - platform independence.

For those instances where a simple GUI with sliders and buttons is helpful, we use **Tcl/Tk** (which is freely distributed for all the supported ToolKit platforms). A number of Tcl/Tk GUI scripts are distributed with the ToolKit release. For control, the Synthesis Toolkit uses raw MIDI (on supported platforms), and **SKINI** (Synthesis ToolKit Instrument Network Interface, a MIDI-like text message synthesis control format).

### A brief history of the Synthesis ToolKit in C++

Perry Cook began developing a pre-cursor to the Synthesis ToolKit (also called STK) under NeXTStep at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University in the early-1990s. With his move to Princeton University in 1996, he ported everything to C++ on SGI hardware, added real-time capabilities, and greatly expanded the synthesis techniques available. With the help of Bill Putnam, Perry also made a port of STK to Windows95. Gary Scavone began using STK extensively in the summer of 1997 and completed a full port of STK to Linux early in 1998. He finished the fully compatible Windows port (using Direct Sound API) in June 1998. Numerous improvements and extensions have been made since then.

The Toolkit has been distributed continuously since 1996 via the Princeton Sound Kitchen [Perry Cook’s home page](http://www.princeton.edu/~cook/), Gary Scavone’s home page at Stanford’s Center for Computer Research in Music and Acoustics (CCRMA), and the [Synthesis ToolKit home page](http://www.princeton.edu/~cook/). The ToolKit has been included in various collections of software. Much of it has also been ported to MAX/MSP on Macintosh computers by Dan Trueman and Luke Dubois of Columbia University, and is distributed as [PeRColate](http://www.princeton.edu/~cook/). Help on real-time sound and MIDI has been provided by Tim Stilson, Bill Putnam, and Gabriel Maldonado.

### Legal and Ethical Notes

This software was designed and created to be made publicly available for free, primarily for academic purposes, so if you use it, pass it on with this documentation, and for free. If you make a million dollars with it, give us some. If you make compositions with it, put us in the program notes.

Some of the concepts are covered by various patents, some known to us and likely...
others which are unknown. Many of the ones known to us are administered by
the Stanford Office of Technology and Licensing. The good news is that large
hunks of the techniques used here are public domain. To avoid subtle legal
issues, we will not state what’s freely useable here, but we will try to note
within the various classes where certain things are likely to be protected by
patents.

Disclaimer

STK is free and we do not guarantee anything. We’ve been hacking on this
code for a while now and most of it seems to work pretty well. But, there surely
are some bugs floating around. Sometimes things work fine on one computer
platform but not so fine on another. FPU overflows and underflows cause very
weird behavior which also depends on the particular CPU and OS. Let us know
about bugs you find and we’ll do our best to correct them.
5.12 Instruments

The ToolKit comes with a wide variety of synthesis algorithms, all of which inherit from the Instrmnt class. In this example, we’ll fire up an instance of the BeeThree FM synthesis class and show how its frequency can be modified over time.

// bethree.cpp

#include "BeeThree.h"
#include "RtWvOut.h"

int main()
{
    // Set the global sample rate before creating class instances.
    Stk::setSampleRate( 44100.0 );

    Instrmnt *instrument = 0;
    RtWvOut *output = 0;
    MY_FLOAT frequency, amplitude, scaler;
    long counter, i;

    try {
        // Define and load the BeeThree instrument
        instrument = new BeeThree();

        // Define and open the default realtime output device for one-channel playback
        output = new RtWvOut(1);
    }
    catch (StkError &)
    { goto cleanup;
    }

    scaler = 1.0;
    frequency = 220.0;
    amplitude = 0.5;
    instrument->noteOn( frequency, amplitude );

    // Play the instrument for 80000 samples, changing the frequency every 2000 samples
    counter = 0;
    while ( counter < 80000 )
    {
        for ( i=0; i<2000; i++ )
        {
            try {
                output->tick( instrument->tick() );
            }
            catch (StkError &)
            { goto cleanup;
            }
        }
        counter += 2000;
        scaler += 0.025;
        instrument->setFrequency( frequency * scaler );
    }

    cleanup:
    delete instrument;
    delete output;
    return 0;
}
// Turn the instrument off with maximum decay envelope.
instrument->noteOff( 1.0 );

cleanup:
    delete instrument;
    delete output;

    return 0;
}

We have used an Instrument pointer when referencing the BeeThree instance above, so it would be simple to replace the BeeThree class with any other STK instrument class. It should be noted, however, that a few classes do not respond to the setFrequency() function (e.g., Shakers, Drummer).

The noteOn() function initiates an instrument attack. Instruments which are continuously excited (e.g., Clarinet, BeeThree) will continue to sound until stopped with a noteOff(). Impulsively excited instrument sounds (e.g., Plucked Wurley) typically decay within a few seconds time, requiring subsequent note-On() messages for re-attack.

Instrument parameters can be precisely controlled as demonstrated above. A more flexible approach to instrument control, allowing arbitrary scorefile or realtime updates, is described in the next tutorial chapter.
5.13 Miscellaneous Links

- The RtAudio WWW site
- Kern Scores: A Library of Electronic Musical Scores (with automatic conversion to SKINII format)
- PeRColate: A Port of STK for Max/MSP
- A Partial Port of STK to Squeak
5.14 The Mail List

An STK mailing list has been set up to facilitate communication among STK users. Subscribing to this list is your best way of keeping on top of new releases, bug fixes, and various user developments.

To join send a message to <stk-request@ccrma.stanford.edu> with the contents: subscribe

To be removed from the list send a message to <stk-request@ccrma.stanford.edu> with the contents: unsubscribe
5.15 Multi-Channel I/O

The ToolKit WvIn and WvOut classes (and their subclasses) support multi-channel audio data input and output. A set of interleaved audio samples representing a single time "slice" is referred to as a sample frame. At a sample rate of 44.1 kHz, a four-channel audio stream will have 44100 sample frames per second and a total of 176400 individual samples per second.

Most STK classes process single-sample data streams via their tick() function. In order to distinguish single-sample and sample frame calculations, the WvIn and WvOut classes implement both tick() and tickFrame() functions. The tickFrame() functions take or return a pointer to an array of audio data representing one or more sample frames. For single-channel streams, the tick() and tickFrame() functions produce equivalent results. When tick() is called for a multi-channel stream, however, the function either returns a sample frame average (WvIn) or writes a single sample argument to all channels (WvOut).

Multi-channel support for realtime audio input and output is dependent on the audio device(s) available on your system.

The following example demonstrates the use of the WvOut class for creating a four channel, 16-bit AIFF formatted audio file. We will use four sinewaves of different frequencies for the first two seconds and then a single sinewave for the last two seconds.

```cpp
#include "WaveLoop.h"
#include "WvOut.h"

int main()
{
   // Set the global sample rate before creating class instances.
   Stk::setSampleRate( 44100.0 );

   int i, j;
   WvOut *output = 0;
   WaveLoop *inputs[4];
   for ( i=0; i<4; i++ ) inputs[i] = 0;

   // Define and load the sine waves
   try {
      for ( i=0; i<4; i++ ) {
         inputs[i] = new WaveLoop( "rawwaves/sinewave.raw", TRUE );
         inputs[i]->setFrequency( 220.0 * (i+1) );
      }
   }
   catch (StkError &){
      goto cleanup;
   }

   // Define and open a 16-bit, four-channel AIFF formatted output file
```
try {
    output = new WvOut( "foursine.aif", 4, WvOut::WVOUT_AIF, Stk::STK_SINT16 );
}
catch (StkError &) {
    goto cleanup;
}

// Write two seconds of four sines to the output file
MY_FLOAT frame[4];
for ( j=0; j<88200; j++ ) {
    for ( i=0; i<4; i++ )
        frame[i] = inputs[i]->tick();

    output->tickFrame( frame );
}

// Now write the first sine to all four channels for two seconds
for ( j=0; j<88200; j++ ) {
    output->tick( inputs[0]->tick() );
}

cleanup:
for ( i=0; i<4; i++ ) delete inputs[i];
delete output;
return 0;

Next tutorial  Main tutorial page
5.16 Voice Management

The previous tutorial chapters were concerned only with monophonic ToolKit instrument playback and control. At this point, it should be relatively clear that one can instantiate multiple instruments and perhaps sum together their sounds or even direct their sounds to separate output channels. It is less clear how one might go about controlling a group of instruments. The Voicer class is designed to serve just this purpose.

The STK Voicer class is a relatively simple voice manager. The user can dynamically add and delete instruments from its “control”, with the option of controlling specific instruments via unique note tags and/or grouping sets of instruments via a “channel” number. All sounding instrument outputs are summed and returned via the tick() function. The Voicer class responds to noteOn, noteOff, setFrequency, pitchBend, and controlChange messages, automatically assigning incoming messages to the voices in its control. When all voices are sounding and a new noteOn is encountered, the Voicer interrupts the oldest sounding voice. The user is responsible for creating and deleting all instrument instances.

In the following example, we modify the controlbee.cpp program to make use of three BeeThree instruments, all controlled using a Voicer.

// threebees.cpp

#include "BeeThree.h"
#include "RtWvOut.h"
#include "Messager.h"
#include "Voicer.h"
#include "SKINI.msg"

int main()
{
    // Set the global sample rate before creating class instances.
    Stk::setSampleRate( 44100.0 );

    int i;
    RtWvOut *output = 0;
    Messager *messager = 0;
    Voicer *voicer = 0;
    bool done = FALSE;
    Instrmnt *instrument[3];
    for ( i=0; i<3; i++ ) instrument[i] = 0;

    try {
        // Define and load the BeeThree instruments
        for ( i=0; i<3; i++ )
            instrument[i] = new BeeThree();

        // Define and open the default realtime output device for one-channel playback
        output = new RtWvOut(1);

        // Other code...
    }

    // Other code...
}
} 
  catch (StkError &) { 
    goto cleanup;
  }
}

try {
  // Create a Messager instance to read from a redirected SKINI scorefile.
  messager = new Messager();
} 
  catch (StkError &) { 
    goto cleanup;
  }

// Instantiate the voicer for a maximum of three voices.
voicer = new Voicer( 3 );
for ( i=0; i<3; i++ )
  voicer->addInstrument( instrument[i] );

// Play the instrument until the end of the scorefile.
int nTicks, type;
MY_FLOAT byte2, byte3;
while (!done) {

  // Look for new messages and return a delta time (in samples).
  type = messager->nextMessage();
  if (type < 0)
    done = TRUE;

  nTicks = messager->getDelta();
  try {
    for ( i=0; i<nTicks; i++ )
      output->tick( voicer->tick() );
  } 
  catch (StkError &) { 
    goto cleanup;
  }

  if ( type > 0 ) {
    // Process the new control message.
    byte2 = messager->getByteTwo();
    byte3 = messager->getByteThree();

    switch(type) {
    case __SK_NoteOn_: 
      voicer->noteOn( byte2, byte3 );
      break;

    case __SK_NoteOff_: 
      voicer->noteOff( byte2, byte3 );
      break;

    case __SK_ControlChange_: 
      voicer->controlChange( (int) byte2, byte3 );
      break;
    }
case __SK_AfterTouch_
    voicer->controlChange( 128, byte2 );
    break;
}
}

cleanup:
    for ( i=0; i<3; i++ ) delete instrument[i];
    delete output;
    delete messager;
    delete voicer;

    return 0;
}

Assuming the program is compiled as threebees, the three-voice SKINI scorefile
bachfugue.ski could be redirected to the program as:

threebees < bachfugue.ski

For more fun, surf to Kern Scores for a huge assortment of other scorefiles
which can be downloaded in the SKINI format.

Another easy extension would be to use the STK MIDI constructor argument to
the Messager class and then play the instruments via a MIDI keyboard.

Main tutorial page
5.17 Realtime Audio

In this section, we modify the `sineosc.cpp` program in order to send the output to the default audio playback device on your system.

```cpp
// rtsine.cpp

#include "WaveLoop.h"
#include "RtWvOut.h"

int main()
{
    // Set the global sample rate before creating class instances.
    Stk::setSampleRate( 44100.0 );

    WaveLoop *input = 0;
    RtWvOut *output = 0;

    try {
        // Define and load the sine wave file
        input = new WaveLoop( "rawwaves/sinewave.raw", TRUE );

        // Define and open the default realtime output device for one-channel playback
        output = new RtWvOut(1);
    }
    catch (StkError &)
    {
        goto cleanup;
    }

    input->setFrequency(440.0);

    // Play the oscillator for 40000 samples
    int i;
    for ( i=0; i<40000; i++ )
    {
        try {
            output->tick(input->tick());
        }
        catch (StkError &)
        {
            goto cleanup;
        }
    }

    cleanup:
    delete input;
    delete output;

    return 0;
}
```

The class `RtWvOut` is a protected subclass of `WvOut`. A number of optional constructor arguments can be used to fine tune its performance for a given system.
Though not used here, an \texttt{RtWvIn} class exists as well which can be used to read realtime audio data from an input device. See the \texttt{record.cpp} example program in the \texttt{examples} project for more information.

It is possible to use an instance of \texttt{RtWvOut} and an instance of \texttt{RtWvIn} to simultaneously read and write realtime audio to and from a hardware device or devices. However, it is recommended to instead use a single instance of \texttt{RtDuplex} to achieve this behavior, in that it guarantees better synchronization between the input and output data. See the \texttt{effects} project or the \texttt{io.cpp} example program in the \texttt{examples} project for more information.

When using any realtime STK class (\texttt{RtAudio}, \texttt{RtWvOut}, \texttt{RtWvIn}, \texttt{RtDuplex}, \texttt{RtMidi}, \texttt{TcpWvIn}, \texttt{TcpWvOut}, \texttt{Socket}, and \texttt{Thread}), it is necessary to specify an audio/MIDI API preprocessor definition and link with the appropriate libraries or frameworks. For example, the above program could be compiled on a Linux system using the GNU g++ compiler and the ALSA audio/MIDI API as follows (assuming all necessary files exist in the project directory):

```

g++ -Wall -D__LINUX_ALSA__ -D__LITTLE_ENDIAN__  -o rtsine Stk.cpp WvIn.cpp WaveLoop.cpp WvOut.cpp \ RtWvOut.cpp RtAudio.cpp rtsine.cpp -lpthread -lasound -lstk
```

On a Macintosh OS X system, the syntax would be:

```
CC -D__MACOSX_CORE__ -o rtsine Stk.cpp WvIn.cpp WaveLoop.cpp WvOut.cpp RtWvOut.cpp RtAudio.cpp \ rtsine.cpp -lpthread -lstdc++ -lstk -framework CoreAudio -framework CoreMIDI -framework CoreFoundation
```

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, \textcopyright{} 1995–2002
5.18 Synthesis toolKit Instrument Network Interface (SKINI)

This describes the latest (version 1.1) implementation of SKINI for the Synthesis ToolKit in C++ (STK) by Perry R. Cook.

Too good to be true?
Have control and read it too?
A SKINI haiku.

Profound thanks to Dan Trueman, Brad Garton, and Gary Scavone for input on this revision. Thanks also to MIDI, the NeXT MusicKit, ZIPI and all the creators and modifiers of these for good bases upon/from which to build and depart.

5.19 MIDI Compatibility

SKINI was designed to be MIDI compatible wherever possible, and extend MIDI in incremental, then maybe profound ways.

Differences from MIDI, and motivations, include:

- Text-based messages are used, with meaningful names wherever possible. This allows any language or system capable of formatted printing to generate SKINI. Similarly, any system capable of reading in a string and turning delimited fields into strings, floats, and ints can consume SKINI for control. More importantly, humans can actually read, and even write if they want, SKINI files and streams. Use an editor and search/replace or macros to change a channel or control number. Load a SKINI score into a spread sheet to apply transformations to time, control parameters, MIDI velocities, etc. Put a monkey on a special typewriter and get your next great work. Life's too short to debug bit/nybble packed variable length mumble messages. Disk space gets cheaper, available bandwidth increases, music takes up so little space and bandwidth compared to video and graphics. Live a little.

- Floating point numbers are used wherever possible. Note Numbers, Velocities, Controller Values, and Delta and Absolute Times are all represented and scanned as ASCII double-precision floats. MIDI byte values are preserved, so that incoming MIDI bytes from an interface can be put directly into SKINI messages. 60.0 or 60 is middle C, 127.0 or 127 is maximum velocity etc. But, unlike MIDI, 60.5 can cause a 50 cent sharp middle C to be played. As with MIDI byte values like velocity, use of the integer and SKINI-added fractional parts is up to the implementor of the algorithm.
being controlled by SKINI messages. But the extra precision is there to be used or ignored.

5.20 Why SKINI?

SKINI was designed to be extensible and hackable for a number of applications: imbedded synthesis in a game or VR simulation, scoring and mixing tasks, real-time and non-real time applications which could benefit from controllable sound synthesis, JAVA controlled synthesis, or eventually maybe JAVA synthesis, etc. SKINI is not intended to be "the mother of scorefiles," but since the entire system is based on text representations of names, floats, and ints, converters from one scorefile language to SKINI or back, should be easily created.

I am basically a bottom-up designer with an awareness of top-down design ideas, so SKINI above all reflects the needs of my particular research and creative projects as they have arisen and developed. SKINI 1.1 represents a profound advance beyond versions 0.8 and 0.9 (the first versions), future SKINI's might reflect some changes. Compatibility with prior scorefiles will be attempted, but there aren’t that many scorefiles out there yet.

5.21 SKINI Messages

A basic SKINI message is a line of text. There are only three required fields, the message type (an ASCII name), the time (either delta or absolute), and the channel number. Don’t freak out and think that this is MIDI channel 0-15 (which is supported), because the channel number is scanned as a long int. Channels could be socket numbers, machine IDs, serial numbers, or even unique tags for each event in a synthesis. Other fields might be used, as specified in the SKINI.tbl file. This is described in more detail later.

Fields in a SKINI line are delimited by spaces, commas, or tabs. The SKINI parser only operates on a line at a time, so a newline means the message is over. Multiple messages are NOT allowed directly on a single line (by use of the ; for example in C). This could be supported, but it isn’t in version 1.1.

Message types include standard MIDI types like NoteOn, NoteOff, Control-Change, etc. MIDI extension message types (messages which look better than MIDI but actually get turned into MIDI-like messages) include LipTension, StringDamping, etc. Non-MIDI message types include SetPath (sets a path for file use later), and OpenReadFile (for streaming, mixing, and applying effects to soundfiles along with synthesis, for example). Other non-MIDI message types include Trilling, HammerOn, etc. (these translate to gestures, behaviors, and contexts for use by intellectual players and instruments using SKINI). Where possible I will still use these as MIDI extension messages, so foot switches, etc.
can be used to control them in real time.

All fields other than type, time, and channel are optional, and the types and usage of the additional fields is defined in the file SKINI.tbl.

The other important file used by SKINI is SKINI.msg, which is a set of defines to make C code more readable, and to allow reasonably quick re-mapping of control numbers, etc.. All of these defined symbols are assigned integer values. For Java, the defines could be replaced by declaration and assignment statements, preserving the look and behavior of the rest of the code.

5.22 C Files Used To Implement SKINI

SKINI.cpp is an object which can either open a SKINI file, and successively read and parse lines of text as SKINI strings, or accept strings from another object and parse them. The latter functionality would be used by a socket, pipe, or other connection receiving SKINI messages a line at a time, usually in real time, but not restricted to real time.

SKINI.msg should be included by anything wanting to use the SKINI.cpp object. This is not mandatory, but use of the _SKblah_ symbols which are defined in the .msg file will help to ensure clarity and consistency when messages are added and changed.

SKINI.tbl is used only by the SKINI parser object (SKINI.cpp). In the file SKINI.tbl, an array of structures is declared and assigned values which instruct the parser as to what the message types are, and what the fields mean for those message types. This table is compiled and linked into applications using SKINI, but could be dynamically loaded and changed in a future version of SKINI.

5.23 SKINI Messages and the SKINI Parser:

The parser isn’t all that smart, but neither am I. Here are the basic rules governing a valid SKINI message:

- If the first (non-delimiter ... see below) character in a SKINI string is ’/’ that line is treated as a comment and echoed to stdout.

- If there are no characters on a line, that line is treated as blank and echoed to stdout. Tabs and spaces are treated as non-characters.

- Spaces, commas, and tabs delimit the fields in a SKINI message line. (We might allow for multiple messages per line later using the semicolon, but probably not. A series of lines with deltaTimes of 0.0 denotes simultaneous
events. For read-ability, multiple messages per line doesn’t help much, so it’s unlikely to be supported later).

- The first field must be a SKINI message name (like NoteOn). These might become case-insensitive in future versions, so don’t plan on exciting clever overloading of names (like noTeOn being different from NoTeON). There can be a number of leading spaces or tabs, but don’t exceed 32 or so.

- The second field must be a time specification in seconds. A time field can be either delta-time (most common and the only one supported in version 0.8), or absolute time. Absolute time messages have an ‘=’ appended to the beginning of the floating point number with no space. So 0.10000 means delta time of 100 ms, while =0.10000 means absolute time of 100 ms. Absolute time messages make most sense in score files, but could also be used for (loose) synchronization in a real-time context. Real-time messages should be time-ordered AND time-correct. That is, if you’ve sent 100 total delta-time messages of 1.0 seconds, and then send an absolute time message of =90.0 seconds, or if you send two absolute time messages of =100.0 and =90.0 in that order, things will get really fouled up. The SKINI parser doesn’t know about time, however. The WvOut device is the master time keeper in the Synthesis Toolkit, so it should be queried to see if absolute time messages are making sense. There’s an example of how to do that later in this document. Absolute times are returned by the parser as negative numbers (since negative deltaTimes are not allowed).

- The third field must be an integer channel number. Don’t go crazy and think that this is just MIDI channel 0-15 (which is supported). The channel number is scanned as a long int. Channels 0-15 are in general to be treated as MIDI channels. After that it’s wide open. Channels could be socket numbers, machine IDs, serial numbers, or even unique tags for each event in a synthesis. A -1 channel can be used as don’t care, omni, or other functions depending on your needs and taste.

- All remaining fields are specified in the SKINI.tbl file. In general, there are maximum two more fields, which are either SK_INT (long), SK_DBL (double float), or SK_STR (string). The latter is the mechanism by which more arguments can be specified on the line, but the object using SKINI must take that string apart (retrieved by using getRemainderString()) and scan it. Any excess fields are stashed in remainderString.

### 5.24 A Short SKINI File:

```c
/* Howdy!!! Welcome to SKINI, by P. Cook 1999
```
5.25 The SKINI.tbl File and Message Parsing:

The SKINI.tbl file contains an array of structures which are accessed by the parser object SKINI.cpp. The struct is:

```c
struct SKINISpec {
    char messageString[32];
    long type;
    long data2;
    long data3;
};
```

so an assignment of one of these structs looks like:

```c
MessageStr$ ,type, data2, data3,
```

`type` is the message type sent back from the SKINI line parser.

`data<n>` is either:

<table>
<thead>
<tr>
<th>Message</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoteOn</td>
<td>0.000082 2 55 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>1.000000 2 55 0</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.000082 2 69 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.100000 2 10</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.100000 2 30</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.100000 2 50</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 69 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.100000 2 40</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.100000 2 22</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.100000 2 12</td>
</tr>
<tr>
<td>//</td>
<td></td>
</tr>
<tr>
<td>StringDamping</td>
<td>0.000100 2 0.0</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.000082 2 55 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.200000 2 62 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.100000 2 71 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.200000 2 79 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>1.000000 2 55 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 62 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 71 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 79 82</td>
</tr>
<tr>
<td>StringDamping</td>
<td>-4.000000 2 0.0</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.000082 2 55 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.200000 2 62 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.100000 2 71 82</td>
</tr>
<tr>
<td>NoteOn</td>
<td>0.200000 2 79 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>1.000000 2 55 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 62 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 71 82</td>
</tr>
<tr>
<td>NoteOff</td>
<td>0.000000 2 79 82</td>
</tr>
</tbody>
</table>
• NOPE : field not used, specifically, there aren’t going to be any more fields on this line. So if there is is NOPE in data2, data3 won’t even be checked.

• SK_INT : byte (actually scanned as 32 bit signed long int). If it’s a MIDI data field which is required to be an integer, like a controller number, it’s 0-127. Otherwise, get creative with SK_INTs.

• SK_DBL : double precision floating point. SKINI uses these in the MIDI context for note numbers with micro tuning, velocities, controller values, etc.

• SK_STR : only valid in final field. This allows (nearly) arbitrary message types to be supported by simply scanning the string to EndOfLine and then passing it to a more intelligent handler. For example, MIDI SYSEX (system exclusive) messages of up to 256 bytes can be read as space-delimited integers into the 1K SK_STR buffer. Longer bulk dumps, soundfiles, etc. should be handled as a new message type pointing to a FileName, Socket, or something else stored in the SK_STR field, or as a new type of multi-line message.

Here’s a couple of lines from the SKINI.tbl file

```{"NoteOff" , __SK_NoteOff_, SK_DBL, SK_DBL},
{"NoteOn" , __SK_NoteOn_, SK_DBL, SK_DBL},
{"ControlChange" , __SK_ControlChange_, SK_INT, SK_DBL},
{"Volume" , __SK_ControlChange_, __SK_Volume_ , SK_DBL},
{"StringDamping" , __SK_ControlChange_, __SK_StringDamping_, SK_DBL},
{"StringDetune" , __SK_ControlChange_, __SK_StringDetune_, SK_DBL} ,```

The first three are basic MIDI messages. The first two would cause the parser, after recognizing a match of the string ”NoteOff” or ”NoteOn”, to set the message type to 128 or 144 (__SK_NoteOff_ and __SK_NoteOn_ are defined in the file SKINI.msg to be the MIDI byte value, without channel, of the actual MIDI messages for NoteOn and NoteOff). The parser would then set the time or delta time (this is always done and is therefore not described in the SKINI Message Struct). The next two fields would be scanned as double-precision floats and assigned to the byteTwo and byteThree variables of the SKINI parser. The remainder of the line is stashed in the remainderString variable.

The ControlChange spec is basically the same as NoteOn and NoteOff, but the second data byte is set to an integer (for checking later as to what MIDI control is being changed).

The Volume spec is a MIDI Extension message, which behaves like a ControlChange message with the controller number set explicitly to the value for MIDI
5.26 Using SKINI:

Volume (7). Thus the following two lines would accomplish the same changing of MIDI volume on channel 2:

```
ControlChange 0.000000 2 7 64.1
Volume 0.000000 2 64.1
```

I like the 2nd line better, thus my motivation for SKINI in the first place.

The StringDamping and StringDetune messages behave the same as the Volume message, but use Control Numbers which aren’t specifically nailed-down in MIDI. Note that these Control Numbers are carried around as long ints, so we’re not limited to 0-127. If, however, you want to use a MIDI controller to play an instrument, using controller numbers in the 0-127 range might make sense.

5.26 Using SKINI:

Here’s a simple example of code which uses the SKINI object to read a SKINI file and control a single instrument.

```cpp
instrument = new Mandolin(50.0);
score = new SKINI(argv[1]);
while(score->getType() > 0) {
    tempDouble = score->getDelta();
    if (tempDouble < 0) {
        tempDouble = -tempDouble;
        tempDouble = tempDouble - output.getTime();
        if (tempDouble < 0) {
            printf("Bad News Here!!! Backward Absolute Time Required.\n");
            tempDouble = 0.0;
        }
    }
    tempLong = (long) (tempDouble * Stk::sampleRate());
    for (i=0;i<tempLong;i++) {
        output.tick(instrument->tick());
    }
    tempDouble3 = score->getByteThree();
    if (score->getType()==__SK_NoteOn_ ) {
        tempDouble3 *= NORM_MIDI;
        if (score->getByteThree() == 0) {
            tempDouble3 = 0.5;
            instrument->noteOff(tempDouble3);
        } else {
            tempLong = (int) score->getByteTwo();
            tempDouble2 = Midi2Pitch[tempLong];
            instrument->noteOn(tempDouble2,tempDouble3);
        }
    } else if (score->getType()==__SK_NoteOff_ ) {
```

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
tempDouble3 *= NORM_MIDI;
instrument->noteOff(tempDouble3);
}
else if (score->getType() == __SK_ControlChange_)
    {
    tempLong = score->getByteTwoInt();
instrument->controlChange(tempLong,temp3.0);
    }
score->nextMessage();
}

When the score (SKINI object) object is created from the filename in argv[1],
the first valid command line is read from the file and parsed.
The score->getType() retrieves the messageType. If this is -1, there are no
more valid messages in the file and the synthesis loop terminates. Otherwise,
the message type is returned.

getDelta() retrieves the deltaTime until the current message should occur. If
this is greater than 0, synthesis occurs until the deltaTime has elapsed. If delta-
Time is less than zero, the time is interpreted as absolute time and the output
device is queried as to what time it is now. That is used to form a deltaTime,
and if it’s positive we synthesize. If it’s negative, we print an error and pretend
this never happened and we hang around hoping to eventually catch up.

The rest of the code sorts out message types NoteOn, NoteOff (including Note-
On with velocity 0), and ControlChange. The code implicitly takes into account
the integer type of the control number, but all other data is treated as double
float.

The last line reads and parses the next message in the file.
5.27 System Requirements

General:

- A MIDI interface to use MIDI input controls. (NOTE: This may be built into the soundcard on your computer.)
- **Tcl/Tk** version 8.0 or higher to use the simple Tcl/Tk GUIs provided with the STK distribution (available free over the WWW for all supported realtime platforms).

Linux (specific):

- A soundcard to use realtime audio input/output capabilities. In order to use the **effects** project, the soundcard and drivers must support full duplex mode.
- **OSS** or **ALSA** device drivers for realtime sound output and MIDI input.

Macintosh OS X (specific):

- A C++ compiler does not ship by default with OS X. It is necessary to download the Developer Kit from the Apple WWW site in order to compile STK.
- The internal Macintosh audio hardware typically supports a sample rate of 44100 Hz only. Therefore, it is necessary to either specify this rate as a command-line option to the STK example programs or to change the default sample rate inside the Stk.h file before compilation. In addition, the RT_BUFFER_SIZE, specified in Stk.h, could be increased (to a higher power of two) for more robust performance.
- The tcl/tk interpreter does not ship by default with OS X, but must be downloaded from the internet. Binary distributions exist but it is instead recommended that you download recent tcl and tk source distributions ([http://dev.scriptics.com/software/tcltk/downloadnow84.tml](http://dev.scriptics.com/software/tcltk/downloadnow84.tml)) and compile them as follows:
  - make -C tcl/macosx deploy
  - make -C tk/macosx deploy
  - sudo make -C tcl/macosx install-deploy
  - sudo make -C tk/macosx install-deploy

  (Note: the tcl and tk directories specified in the above lines will more likely be appended with version numbers) The default installation will place a link to the wish interpreter at /usr/bin/wish. The latest 8.4.1 release of tcl/tk has been tested on a 10.2 system and found to work correctly. In particular, redirection of a tcl/tk script to the interpreter (e.g., wish < test.tcl) works normally (which is not the case with binary distributions tested thus far).
Initial tests have shown somewhat poor response between changes made in the tcl/tk script and the resulting audio updates. Also, it is not recommended to connect by socket from a tcl/tk script to an STK program because the tcl/tk interpreter does not appear to properly close the socket connection, leaving the STK program in a "hung" state.

Windows95/98/2000/XP (specific):

- A soundcard to use realtime audio input/output capabilities. In order to use the effects project, the soundcard and drivers must support full duplex mode.
- DirectX 5.0 (or higher) runtime libraries to use the precompiled binaries.
- Visual C++ 6.0 for compiling (though a precompiled distribution is available).
- For compiling the source (if not already in your system):
  - [dsound.h](/) header file (DirectX 6.1) - put somewhere in your header search path
  - [dsound.lib](/) library file (DirectX 6.1) - put somewhere in your library search path

WindowsNT (specific):

- DirectX support for NT is inadequate, so it is not possible to use STK under WindowsNT with realtime DirectX support. It may be possible to use STK under WindowsNT with realtime ASIO support, though this has not been tested.
The Synthesis ToolKit is a set of C++ classes. In order to go beyond the simple example programs we provide, it is necessary to know some basics about programming in C or C++. STK’s “target audience” includes people who:

- want to create audio DSP and/or synthesis programs
- want to save some time by using our unit generators and input/output routines
- want to learn about synthesis and processing algorithms
- wish to teach real-time synthesis and processing, and wish to use some of our classes and examples

Most ToolKit programmers will likely end up writing a class or two for their own particular needs, but this task is typically simplified by making use of pre-existing STK classes (filters, oscillators, etc.).

The following tutorial chapters describe many of the fundamental ToolKit concepts and classes. All tutorial programs are included in the `projects/examples` directory.

1. **Hello Sine!**
2. **Compiling**
3. **Realtime Audio**
4. **Instruments**
5. **Control Input**
6. **Multi-Channel I/O**
7. **Voice Management**
5.29 Usage Documentation

- **Directory Structure:**
- **Compiling:**
- **Control Data:**
- **Demo:** STK Instruments
- **Demo:** Non-Realtime Use
- **Demo:** Realtime Use
- **Realtime Control Input using Tcl/Tk Graphical User Interfaces:**
- **Realtime MIDI Control Input:**
- **Polyphony:**

5.30 Directory Structure:

The top level distribution contains the following directories:

- The src directory contains the source .cpp files for all the STK unit generator and algorithm classes.
- The include directory contains the header files for all the STK unit generator and algorithm classes.
- The rawwaves directory contains various raw, monophonic, 16-bit, big-endian soundfiles used with the STK classes.
- The doc directory contains documentation about STK.
- The projects directory contains various demo and example STK programs.

This release of STK comes with four separate "project" directories:

1. The demo project is used to demonstrate nearly all of the STK instruments. The demo program has been written to allow a variety of control input and sound data output options. Simple graphical user interfaces (GUIs) are also provided.

2. The effects project demonstrates realtime duplex mode (simultaneous audio input and output) operation, when available, as well as various delay-line based effects algorithms.

3. The ragamatic project is just cool. Fire it up and be enlightened.

4. The examples project contains several simple programs which demonstrate audio input/output, as well as the use of the audio internet streaming classes.
5.31 Compiling:

- **Windows95/98/2000/XP:** Realtime support is available using either DirectSound or ASIO audio drivers. For DirectSound support, use the \_\_WINDOWS\_\_DS\_ predefined preprocessor definition and link with the dsound.lib, winmm.lib, and Wsock32.lib libraries. For ASIO support, use the \_\_WINDOWS\_\_ASIO\_ predefined preprocessor definition, include all the files in the src/asio/ directory (i.e. asio.h, cpp, asiodrivers.h, cpp, ...), and link with the winmm.lib, and Wsock32.lib libraries. In addition, the \_\_LITTLE\_\_ENDIAN\_ predefined preprocessor definition is necessary for all Windows systems. A distribution of the release is available with precompiled binaries (using DirectSound) for all the projects. In order for these binaries to function properly, your system must have the DirectX 5.0 (or higher) runtime libraries installed (available from Microsoft). Further, the effects project requires that your soundcard and drivers provide full duplex mode capabilities. Visual C++ 6.0 project files are provided in each project directory as well should you wish to compile your own binaries. It is important to link with the non-debug libraries when compiling "release" program versions and debug libraries when compiling "debug" program versions.

- **WindowsNT:** DirectX support for NT is inadequate, so it is not possible to use STK under WindowsNT with realtime DirectX support. It may be possible to use STK under WindowsNT with realtime ASIO support, though this has not been tested.

- **Unix Systems:** A GNU configure shell script is included in the distribution for unix-based systems. From the top-level distribution directory, type './configure' and the script will create Makefiles in each project directory specific to the characteristics of the host computer. Then from within any given project directory (example demo), type 'make' to compile the project. In addition, an STK library can be compiled from within the src directory.

Several options can be supplied to the configure script to customize the build behavior:

- --disable-realtime to only compile generic non-realtime classes
- --enable-debug to enable various debug output
- --enable-midiator to enable native MS-124W MIDI support (linux only)
- --with-alsa to choose native ALSA API support (linux only)

In addition, it is possible to specify the location of the STK rawwaves and the STK include path as follows:

./configure RAWWAVE_PATH="/home/gary/rawwaves/"
./configure INCLUDE_PATH="/home/gary/include/"

For novice STK users, the default configuration should be adequate.
For those who wish to create their own system-specific **Makefiles:**

- **Linux:** Realtime support is enabled with either the `__LINUX_OSS__` or `__LINUX_ALSA__` preprocessor definitions, which are used to select the underlying audio/MIDI system API. Realtime programs must also link with the `pthread` library. When using the ALSA API, it is also necessary to link with the `asound` library. In addition, the `__LITTLE_ENDIAN__` preprocessor definition is necessary if compiling on a little-endian system. Special support exists under Linux for the MIDIator serial MIDI device, enabled using the `__MIDIATOR__` preprocessor definition (together with either the `__LINUX_ALSA__` or `__LINUX_OSS__` definitions). See the README-Linux file for further system configuration information.

- **Macintosh OS X:** Realtime support is enabled with the `__MACOSX_CORE__` preprocessor definitions, which incorporates the CoreAudio audio/MIDI API. Realtime programs must also link with the `pthread` library and the `CoreAudio`, `CoreMIDI`, and `CoreFoundation` frameworks. See the README-MacOSX file for further system configuration information.

- **SGI:** Realtime support is enabled with the `__IRIX_AL__` preprocessor definition and linkage with the `audio`, `md`, and `pthread` libraries. STK 4.0 (and higher) is confirmed to compile using CC version 7.30. There may be problems with old compiler versions.

- **Generic (non-realtime):** Most STK classes are operating system independent and can be compiled using any current C++ compiler. STK assumes big-endian host byte order by default, so if your system is little-endian (i.e., Intel processor), you must provide the `__LITTLE_ENDIAN__` preprocessor definition to your compiler. The demo project will compile without realtime support, allowing the use of SKINI scorefiles for input control and output to a variety of soundfile formats. The following classes **cannot** be used without realtime support: `RtAudio`, `RtWvIn`, `RtWvOut`, `RtDuplex`, `RtMidi`, `Socket`, `Thread`, `TcpWvIn`, `TcpWvOut`. Because of this, it is not possible to compile the effects, ragamatic, and most of the examples projects for non-realtime use.

### 5.32 Control Data:

All STK programs in this distribution take input control data in the form of SKINI or MIDI messages only. The `Message` class unifies the various means of acquiring control data under a single, easy to use set of functions. The way that SKINI messages can be sent to the programs is dependent upon the operating system in use, as well as whether the program is running in realtime or not. In general, it is possible to:

1. Redirect or pipe SKINI scorefiles to an executable.
2. Pipe realtime SKINI input messages to an executable (not possible under Windows 95/98).

3. Socket realtime SKINI input messages to an executable.

4. Acquire realtime MIDI messages from a MIDI port on your computer.

Tcl/Tk graphical user interfaces (GUI) are provided with this distribution which can generate realtime SKINI messages. Note that the Messenger class allows multiple simultaneous socket client connections, together with MIDI and/or piped input. The Md2Skini program (in the demo directory) is mostly obsolete but can be used to create SKINI scorefiles from realtime MIDI input.

5.33 Demo: STK Instruments

The demo project demonstrates the behavior of all the distributed STK instruments. The instruments available with this release include:

- Clarinet: Pretty good physical model of the clarinet
- BlowHole: A clarinet physical model with one tonehole and one register vent
- Saxofony: A pseudo-conical bore reed instrument which sometimes sounds like a saxophone
- Flute: Pretty good physical model of the flute
- Brass: Not so bad physical model of a brass instrument
- BlowBot: A basic helmholtz resonator and air jet model
- Bowed: Not hideous physical model of a bowed string instrument
- Plucked: Yer basic plucked string physical model
- StilKarp: A simple plucked, stiff string physical model
- Sitar: A simple sitar/plucked string physical model
- Mandolin: Two-string mandolin physical model
- Rhodey: Rhodes-like electric piano FM synthesis model
- Wurley: Wurlitzer-like electric piano FM synthesis model
- TubeBell: FM synthesis model
- HevyMetl: Distorted synthesizer FM synthesis model
- PercFlit: Percussive flute-like FM synthesis model
- BeeThree: Cheezy organ FM synthesis model
- Moog: Swept filter sampler
- FMVoices: Three-formant FM voice synthesis
- VocForm: Four-formant resonance filter voice synthesis
- Resonate: Noise through a BiQuad filter
- Drummer: Sampling synthesis
- **BandedWG**: Banded waveguide meta-object for bowed bars, tibetan bowls, etc.
- **Shakers**: Various stochastic event models of shaker instruments
- **ModalBar**: Various four-resonance presets (marimba, vibraphone, etc...)
- **Mesh2D**: Two-dimensional, rectilinear digital waveguide mesh
- **Whistle**: Hybrid physical/spectral model of a police whistle

### 5.34 Demo: Non-Realtime Use

See the information above with respect to compiling STK for non-realtime use.

In non-realtime mode, it is assumed that input control messages are provided from a [SKIN1] scorefile and that audio output is written to a soundfile (.snd, .wav, .aif, .mat, .raw). A number of [SKIN1] scorefiles are provided in the *scores* directory of the *demo* project. Assuming a successful compilation of the *demo* program, typing:

```
cat scores/bookert.ski | demo BeeThree -ow myfile.wav
```

or (on WindowsXX and/or Unix)

```
demo BeeThree -ow myfile.wav < scores\bookert.ski
```

from the *demo* directory will play the scorefile *bookert.ski* using the STK *BeeThree* instrument and write the resulting audio data to a WAV formatted soundfile called "myfile.wav". Typing *demo* without any arguments will provide a full program usage description.

### 5.35 Demo: Realtime Use

STK realtime audio and MIDI input/output and realtime [SKIN1] control input via socketing support is provided for Linux, SGI, Mac OS X, and Windows95/98/2000/XP operating systems. STK realtime [SKIN1] control input via piping is possible under Linux, SGI, Mac OS X, and Windows2000/XP only.

Control input and audio output options are typically specified as command-line arguments to STK programs. For example, the *demo* program is invoked as:

```
demo instrument flags
```

where instruments include those described above and flags can be any or all of:

- `-or` for realtime audio output,
5.36 Realtime Control Input using Tcl/Tk Graphical User Interfaces:

- `-ow <file name>` for WAV soundfile output,
- `-os <file name>` for SND (AU) soundfile output,
- `-om <file name>` for MAT-file output,
- `-ip` for realtime SKINI control input via piping,
- `-is <port> >` for realtime SKINI control input via socketing (with an optional port number),
- `-im <file name>` for MIDI control input
- `-s RATE` to specify a sample rate
- `-n NUMBER` to specify multivoice polyphony

The `<-ip>` and `<-is>` flags must be used when piping or socketing realtime SKINI control data to an STK program. The `<-im>` flag must be used to read MIDI control input from your MIDI port. Note that you can use all three input types simultaneously.

Assuming a successful compilation of the demo program, typing:

```
cat scores/bookert.ski | demo BeeThree -or
```

or (on WindowsXX and/or Unix)

```
demo BeeThree -or < scores\bookert.ski
```

from the demo directory will play the scorefile bookert.ski using the STK BeeThree instrument and stream the resulting audio data in realtime to the audio output channel of your computer. Typing demo without any arguments will provide a full program usage description.

## 5.36 Realtime Control Input using Tcl/Tk Graphical User Interfaces:

There are a number of Tcl/Tk GUIs supplied with the STK projects. These scripts require Tcl/Tk version 8.0 or later, which can be downloaded for free over the WWW. On Unix and Windows2000/XP platforms, you can run the various executable scripts (e.g. StkDemo.bat) provided with each project to start everything up (you may need to symbolically link the wishXX executable to the name wish). The Physical.bat script just implements the following command-line sequence:

```
wish < tcl/Physical.tcl | demo Clarinet -or -ip
```

On WindowsXX and Unix platforms, the following operations are necessary to establish a socket connection between the Tcl/Tk GUI and the STK program:
1. Open a DOS shell and start the STK program with the -is flag (ex. demo \texttt{Clarinet} -or -is).

2. Open the Tcl/Tk GUI (e.g. tcl/Physical.tcl) by double-clicking on it, or type \texttt{wish < tcl/Physical.tcl} in another DOS shell.

3. Establish the socket connection by selecting \texttt{Socket} under the Communications menu item in the Tcl/Tk GUI.

Note that it is possible to specify a hostname when establishing the socket connection from the socket client. Thus, the STK socket server program and the Tcl/Tk GUI need not necessarily reside on the same computer.

\section*{5.37 Realtime MIDI Control Input:}

On all supported realtime platforms, you can direct realtime MIDI input to the STK \texttt{Clarinet} by typing:

\texttt{demo Clarinet -or -im}

\section*{5.38 Polyphony:}

The \texttt{demo} program supports an arbitrary number of voices via the \texttt{-n NUMBER} command-line flag and argument. For example, you can play eight \texttt{BeeThree} instruments with realtime output and control them from a MIDI device by typing:

\texttt{demo BeeThree -n 8 -or -im}
Index

~ADSR
  ADSR, 12
~BandedWG
  BandedWG, 14
~BeeThree
  BeeThree, 16
~BiQuad
  BiQuad, 18
~BlowBotl
  BlowBotl, 21
~BlowHole
  BlowHole, 23
~BowTabl
  BowTabl, 27
~Bowed
  Bowed, 25
~Brass
  Brass, 29
~Chorus
  Chorus, 31
~Clarinet
  Clarinet, 33
~Delay
  Delay, 35
~DelayA
  DelayA, 38
~DelayL
  DelayL, 40
~Drummer
  Drummer, 42
~Echo
  Echo, 44
~Envelope
  Envelope, 46
~FM
  FM, 53
~FMVoices
  FMVoices, 56
~Filter
  Filter, 18
~Flute
  Flute, 51
~FormSwep
  FormSwep, 58
~HevyMetl
  HevyMetl, 61
~Instrmnt
  Instrmnt, 63
~JCRv
  JCRv, 65
~JetTabl
  JetTabl, 66
~Mandolin
  Mandolin, 68
~Mesh2D
  Mesh2D, 70
~Messager
  Messager, 72
~Modal
  Modal, 75
~ModalBar
  ModalBar, 77
~Modulate
  Modulate, 79
~Moog
  Moog, 81
~NRev
  NRev, 85
~Noise
  Noise, 83
~OnePole
  OnePole, 87
Index

OneZero 89
PRCRev 103
PercFlut 91
PitShift 94
PluckTwo 98
Plucked 96
PoleZero 100
ReedTabl 104
Resonate 106
Reverb 108
Rhodey 110
RtAudio 115
RtDuplex 122
RtError 125
RtMidi 127
RtWvIn 129
RtWvOut 132
SKINI 149
Samplar 135
Saxofony 137
Shakers 139
Simple 142
SingWave 144

Sitar 147
Socket 152
Sphere 156
StifKarp 158
Stk 161
StkError 164
SubNoise 165
Table 167
TcpWvIn 169
TcpWvOut 172
Thread 175
TubeBell 177
TwoPole 179
TwoZero 182
Vector3D 185
VoicForm 191
Voicer 187
WaveLoop 193
Whistle 195
Wurley 197
WvIn 199
WvOut 204
abortStream

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
INDEX 259

RtAudio, 117
accept Socket, 154
addInstrument Voicer, 189
addPhase WaveLoop, 194
addPhaseOffset WaveLoop, 194
addTime WaveLoop, 193
WvIn, 200
addVelocity Sphere, 157
ADSR, 11
～ADSR, 12
ADSR, 11
gate, 12
keyOff, 12
keyOn, 12
setAllTimes, 12
setAttackRate, 12
setAttackTime, 12
setDecayRate, 12
setDecayTime, 12
setReleaseRate, 12
setReleaseTime, 12
setSustainLevel, 12
setTarget, 12
setValue, 12
tick, 13

BandedWG, 14
～BandedWG, 14
BandedWG, 14
clear, 14
controlChange, 15
noteOff, 15
noteOn, 15
pluck, 15
setFrequency, 14
setPreset, 14
setStrikePosition, 14
startBowing, 14
stopBowing, 14
tick, 15

BeeThree, 16
～BeeThree, 16
BeeThree, 16
noteOn, 16
tick, 16
BeeThree, 16

BiQuad, 18
～Biquad, 18
BiQuad, 18
clear, 18
gain, 19
lastOut, 19
setA1, 18
setA2, 18
setB0, 18
setB1, 18
setB2, 18
tick, 19

BlowBotl, 21
～BlowBotl, 21
BlowBotl, 21
clear, 21
controlChange, 22
noteOff, 21
noteOn, 21
setFrequency, 21
startBowing, 21
stopBowing, 21
tick, 21
BlowBotl, 21

BlowHole, 23
～BlowHole, 23
BlowHole, 23
clear, 23
controlChange, 24
noteOff, 24
noteOn, 24
setFrequency, 23
setTonehole, 23
setVent, 23

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
startBlowing, 23
stopBlowing, 23
tick, 24
BlowHole, 23
Bowed, 25
～Bowed, 25
clear, 25
controlChange, 26
noteOff, 26
noteOn, 25
setFrequency, 26
setVibrato, 26
startBowing, 26
stopBowing, 26
tick, 26
BowTabl
～BowTabl, 27
BowTabl, 27
lastOut, 27
tick, 27
BowTabl, 27
setOffset, 28
setSlope, 28
tick, 28
Brass, 29
～Brass, 29
Brass, 29
clear, 29
controlChange, 30
noteOff, 30
noteOn, 29
setFrequency, 29
setLip, 29
startBowing, 29
stopBowing, 29
tick, 30
cancelStreamCallback
RtAudio, 116
Chorus, 31
～Chorus, 31
Chorus, 31
clear, 31
lastOut, 31
lastOutLeft, 31
lastOutRight, 31
setEffectMix, 31
setModDepth, 31
setModFrequency, 31
tick, 32
Clarinet, 33
～Clarinet, 33
Clarinet, 33
clear, 33
controlChange, 34
noteOff, 33
noteOn, 33
setFrequency, 33
startBowing, 33
stopBowing, 33
tick, 34
clear
BandedWG, 14
BiQuad, 18
BlowBotl, 21
BlowHole, 23
Bowed, 25
Brass, 29
Chorus, 31
Clarinet, 33
Delay, 35
DelayA, 38
Echo, 44
Filter, 48
Flute, 51
FM, 53
JCR Rev, 65
Mesh2D, 70
Modal, 75
NRev, 85
OnePole, 87
OneZero, 89
Pit Shift, 94
Plucked, 96
Pluck Two, 98
PoleZero, 100
PRC Rev, 103
Resonate, 106
Reverb, 108
Sampler, 135
Saxofony, 137
energy
  Delay, 35
  Mesh2D, 71
Envelope, 46
~Envelope, 46
Envelopes, 46
getEnvelope, 46
keyOff, 46
keyOn, 46
lastOut, 47
setRate, 46
setTarget, 46
setTime, 46
setValue, 46
tick, 47
Filter, 48
~Filter, 48
  clear, 48
Filter, 48
getGain, 49
lastOut, 49
setCoefficients, 50
setDenominator, 50
setGain, 50
setNumerator, 50
tick, 49
Flute, 51
~Flute, 51
  clear, 51
controlChange, 52
Flute, 51
noteOff, 52
noteOn, 52
setEndReflection, 51
setFrequency, 51
setJetDelay, 51
setJetReflection, 51
startBlowing, 51
stopBlowing, 52
tick, 52
FM, 53
~FM, 53
  clear, 53
controlChange, 54
FM, 53
keyOff, 54
keyOn, 54
loadWaves, 53
noteOff, 54
setControl1, 54
setControl2, 54
setFrequency, 53
setGain, 53
setModulationDepth, 54
setModulationSpeed, 53
tick, 54
FMVoices, 59
~FMVoices, 59
  controlChange, 56
FMVoices, 56
noteOff, 56
setFrequency, 56
tick, 56
formantFrequency
  Phonemes, 93
formantGain
  Phonemes, 93
formantRadius
  Phonemes, 93
FormSweep
~FormSweep, 58
FormSweep, 58
setStates, 58
setTargets, 58
tick, 59
FormSweep, 58
  setResonance, 59
  setSweepRate, 59
  setSweepTime, 59
getByteThree
  Messager, 72
  RtMidi, 127
SKINI, 149
getByteThreeInt
  SKINI, 150
getByteTwo
  Messager, 72
  RtMidi, 127
SKINI, 149
INDEX

getByteTwoInt
    SKINI, 150
getChannel
    Messenger, 72
    RtMidi, 127
    SKINI, 149
getChannels
    WvIn, 200
getDelay
    Delay, 35
    DelayA, 38
    DelayL, 40
getDelta
    Messenger, 72
    SKINI, 149
getDeltaTime
    RtMidi, 127
getDeviceCount
    RtAudio, 112
getDeviceInfo
    RtAudio, 116
getFileSize
    WvIn, 199
getState
    ADSR, 12
    Envelope, 46
getStreamBuffer
    RtAudio, 116
tick
    SKINI, 150
getVelocity
    Sphere, 156
getX
    Vector3D, 185
getY
    Vector3D, 185
getZ
    Vector3D, 185
handleError
    Stk, 161
hasDuplexSupport
    RtAudio::RTAUDIO_DEVICE, 120
HevyMetl
    ~HevyMetl, 61
    HevyMetl, 61
noteOn
    HevyMetl, 61
tick
    HevyMetl, 61
Instrmnt
    63

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>~Instrmnt</td>
<td>63</td>
</tr>
<tr>
<td>controlChange</td>
<td>64</td>
</tr>
<tr>
<td>Instrmnt</td>
<td>63</td>
</tr>
<tr>
<td>lastOut</td>
<td>64</td>
</tr>
<tr>
<td>noteOff</td>
<td>64</td>
</tr>
<tr>
<td>noteOn</td>
<td>64</td>
</tr>
<tr>
<td>setFrequency</td>
<td>64</td>
</tr>
<tr>
<td>tick</td>
<td>64</td>
</tr>
<tr>
<td>isConnected</td>
<td></td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>171</td>
</tr>
<tr>
<td>isDefault</td>
<td></td>
</tr>
<tr>
<td>RtAudio:RTAUDIO_DEVICE</td>
<td>120</td>
</tr>
<tr>
<td>isFinished</td>
<td></td>
</tr>
<tr>
<td>WvIn</td>
<td>200</td>
</tr>
<tr>
<td>isInside</td>
<td></td>
</tr>
<tr>
<td>Sphere</td>
<td>156</td>
</tr>
<tr>
<td>isValid</td>
<td></td>
</tr>
<tr>
<td>Socket</td>
<td>153</td>
</tr>
<tr>
<td>JCRex</td>
<td>65</td>
</tr>
<tr>
<td>~JCRex</td>
<td>65</td>
</tr>
<tr>
<td>clear</td>
<td>65</td>
</tr>
<tr>
<td>JCRex</td>
<td>65</td>
</tr>
<tr>
<td>tick</td>
<td>65</td>
</tr>
<tr>
<td>JetTabl</td>
<td></td>
</tr>
<tr>
<td>~JetTabl</td>
<td>66</td>
</tr>
<tr>
<td>JetTabl</td>
<td>66</td>
</tr>
<tr>
<td>lastOut</td>
<td>66</td>
</tr>
<tr>
<td>tick</td>
<td>66</td>
</tr>
<tr>
<td>JetTabl</td>
<td>66</td>
</tr>
<tr>
<td>keyOff</td>
<td></td>
</tr>
<tr>
<td>ADSR</td>
<td>12</td>
</tr>
<tr>
<td>Envelope</td>
<td>46</td>
</tr>
<tr>
<td>FM</td>
<td>54</td>
</tr>
<tr>
<td>Resonate</td>
<td>106</td>
</tr>
<tr>
<td>Sampler</td>
<td>135</td>
</tr>
<tr>
<td>Simple</td>
<td>142</td>
</tr>
<tr>
<td>keyOn</td>
<td></td>
</tr>
<tr>
<td>ADSR</td>
<td>12</td>
</tr>
<tr>
<td>Envelope</td>
<td>46</td>
</tr>
<tr>
<td>FM</td>
<td>54</td>
</tr>
<tr>
<td>Resonate</td>
<td>106</td>
</tr>
<tr>
<td>Sampler</td>
<td>135</td>
</tr>
<tr>
<td>Simple</td>
<td>142</td>
</tr>
<tr>
<td>lastFrame</td>
<td></td>
</tr>
<tr>
<td>RtDuplex</td>
<td>122</td>
</tr>
<tr>
<td>RtWvIn</td>
<td>129</td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>169</td>
</tr>
<tr>
<td>WvIn</td>
<td>200</td>
</tr>
<tr>
<td>lastOut</td>
<td></td>
</tr>
<tr>
<td>BiQuad</td>
<td>19</td>
</tr>
<tr>
<td>BowTabl</td>
<td>27</td>
</tr>
<tr>
<td>Chorus</td>
<td>31</td>
</tr>
<tr>
<td>Delay</td>
<td>36</td>
</tr>
<tr>
<td>Echo</td>
<td>44</td>
</tr>
<tr>
<td>Envelope</td>
<td>47</td>
</tr>
<tr>
<td>Filter</td>
<td>49</td>
</tr>
<tr>
<td>Instrmnt</td>
<td>64</td>
</tr>
<tr>
<td>JetTabl</td>
<td>66</td>
</tr>
<tr>
<td>Modulate</td>
<td>79</td>
</tr>
<tr>
<td>Noise</td>
<td>83</td>
</tr>
<tr>
<td>OnePole</td>
<td>88</td>
</tr>
<tr>
<td>OneZero</td>
<td>90</td>
</tr>
<tr>
<td>PitShift</td>
<td>94</td>
</tr>
<tr>
<td>PoleZero</td>
<td>101</td>
</tr>
<tr>
<td>ReedTabl</td>
<td>104</td>
</tr>
<tr>
<td>Reverb</td>
<td>108</td>
</tr>
<tr>
<td>RtDuplex</td>
<td>122</td>
</tr>
<tr>
<td>RtWvIn</td>
<td>129</td>
</tr>
<tr>
<td>SingWave</td>
<td>145</td>
</tr>
<tr>
<td>Table</td>
<td>167</td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>169</td>
</tr>
<tr>
<td>TwoPole</td>
<td>180</td>
</tr>
<tr>
<td>TwoZero</td>
<td>183</td>
</tr>
<tr>
<td>Voicer</td>
<td>188</td>
</tr>
<tr>
<td>WvIn</td>
<td>200</td>
</tr>
<tr>
<td>lastOutLeft</td>
<td></td>
</tr>
<tr>
<td>Chorus</td>
<td>31</td>
</tr>
<tr>
<td>Reverb</td>
<td>108</td>
</tr>
<tr>
<td>lastOutRight</td>
<td></td>
</tr>
<tr>
<td>Chorus</td>
<td>31</td>
</tr>
<tr>
<td>Reverb</td>
<td>108</td>
</tr>
<tr>
<td>listen</td>
<td></td>
</tr>
<tr>
<td>TcpWvIn</td>
<td>170</td>
</tr>
<tr>
<td>loadWaves</td>
<td></td>
</tr>
<tr>
<td>FM</td>
<td>53</td>
</tr>
<tr>
<td>Mandolin</td>
<td>68</td>
</tr>
<tr>
<td>~Mandolin</td>
<td>68</td>
</tr>
<tr>
<td>controlChange</td>
<td>69</td>
</tr>
</tbody>
</table>
INDEX

Mandolin, 68
noteOn, 68
pluck, 68
setBodySize, 68
tick, 68
maxDuplexChannels
RtAudio::RTAUDIO_DEVICE, 120
maxInputChannels
RtAudio::RTAUDIO_DEVICE, 119
maxOutputChannels
RtAudio::RTAUDIO_DEVICE, 119
Mesh2D, 70
~Mesh2D, 70
clear, 70
controlChange, 71
ergy, 71
Mesh2D, 70
noteOff, 70
noteOn, 70
setDecay, 70
setInputPosition, 70
setNX, 70
setNY, 70
tick, 71
Message, 72
~Message, 72
getByteThree, 72
getByteTwo, 72
getChannel, 72
getDelta, 72
getType, 72
Message, 73
nextMessage, 73
setRtDelta, 72
minDuplexChannels
RtAudio::RTAUDIO_DEVICE, 120
minInputChannels
RtAudio::RTAUDIO_DEVICE, 120
minOutputChannels
RtAudio::RTAUDIO_DEVICE, 120
Modal, 75
~Modal, 75
clear, 75
controlChange, 76
damp, 76
Modal, 75
noteOff, 76
noteOn, 76
setDirectGain, 75
setFrequency, 75
setMasterGain, 75
setModeGain, 75
setRatioAndRadius, 75
strike, 76
tick, 76
ModalBar
~ModalBar, 77
controlChange, 77
ModalBar, 77
setModulationDepth, 77
setPreset, 77
setStickHardness, 77
setStrikePosition, 77
ModalBar, 77
Modulate, 79
~Modulate, 79
lastOut, 79
Modulate, 79
reset, 79
setRandomGain, 79
setVibratoGain, 79
setVibratoRate, 79
tick, 79
Moog, 81
~Moog, 81
controlChange, 81
Moog, 81
noteOn, 81
setFrequency, 81
setModulationDepth, 81
setModulationSpeed, 81
tick, 81
name
Phonemes, 93

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
RtAudio::RTAUDIO
DEVICE, 119

nativeFormats
RtAudio::RTAUDIO
DEVICE, 120

nextMessage
Message, 73
RtMidi, 128
SKINL, 151

nextOut
Delay, 36
DelayA, 39
DelayL, 41
Noise, 83
～Noise, 83
lastOut, 83
Noise, 83
tick, 83

noiseGain
Phonemes, 93

normalize
SingWave, 144
WvIn, 201, 202

noteOff
BandedWG, 15
BlowBotl, 21
BlowHole, 24
Bowed, 26
Brass, 30
Clarinet, 33
Drummer, 42
Flute, 52
FM, 54
Instrmnt, 64
Mesh2D, 70
Modal, 76
Moog, 81
PercFlut, 91
Plucked, 96
Resonate, 106
Rhodey, 110
Saxofony, 137
Shakers, 141
Simple, 142
SingWave, 145
Sitar, 147
StifKarp, 158
TubeBell, 177
Voicer, 189
VoicForm, 192
Whistle, 195
Wurley, 197

NRev, 85
～NRev, 85
clear, 85
NRev, 85
tick, 85

nSampleRates
RtAudio::RTAUDIO
DEVICE, 120

OnePole
～OnePole, 87
clear, 87
getGain, 87
lastOut, 88
OnePole, 87
setA1, 87
setB0, 87
tick, 88
OnePole, 87
setGain, 88
setPole, 88
OneZero
~OneZero, 89
clear, 89
getGain, 89
lastOut, 90
OneZero, 89
setB0, 89
setB1, 89
tick, 90
OneZero, 89
setGain, 90
setZero, 90
openFile
WvIn, 201
WvOut, 206
openStream
RtAudio, 115
parseThis
SKINI, 151
PercFlut
~PercFlut, 91
noteOn, 91
PercFlut, 91
setFrequency, 91
tick, 91
PercFlut, 91
Phonemes, 93
formantFrequency, 93
formantGain, 93
formantRadius, 93
name, 93
noiseGain, 93
voiceGain, 93
pitchBend
Voicer, 188
PitShift
~PitShift, 94
clear, 94
lastOut, 94
PitShift, 94
setEffectMix, 94
setShift, 94
tick, 94
PitShift, 94
pluck
BandedWG, 15
Mandolin, 68
Plucked, 96
Sitar, 147
StifKarp, 158
Plucked, 96
~Plucked, 96
clear, 96
noteOff, 96
noteOn, 96
pluck, 96
Plucked, 96
setFrequency, 96
tick, 96
PluckTwo
~PluckTwo, 98
clear, 98
noteOff, 99
PluckTwo, 98
setDetune, 98
setFreqAndDetune, 98
setFrequency, 98
setPluckPosition, 98
tick, 99
PluckTwo, 98
setBaseLoopGain, 99
PoleZero
~PoleZero, 100
clear, 100
getGain, 101
lastOut, 101
PoleZero, 100
setA1, 100
setB0, 100
setB1, 100
tick, 101
PoleZero, 100
setAllpass, 101
setBlockZero, 101
setGain, 101

port
Socket, 152
PRCRev, 103
∼PRCRev, 103
clear, 103
PRCRev, 103
tick, 103

printMessage
RtError, 125
RtMidi, 127
StkError, 164

probed
RtAudio::RTAUDIO DEVICE, 119

quiet
VoicForm, 192

readBuffer
Socket, 153
ReedTabl
∼ReedTabl, 104
lastOut, 104
ReedTabl, 104
tick, 104
ReedTabl, 104
setOffset, 105
setSlope, 105
tick, 105

removeInstrument
Voicer, 189

reset
Modulate, 79
SingWave, 144
WvIn, 199
Resonate, 106
∼Resonate, 106
clear, 106
controlChange, 107
keyOff, 106
keyOn, 106
noteOff, 107
noteOn, 106
Resonate, 106
setEqualGainZeroes, 106
setNotch, 106
setResonance, 106
tick, 107
Reverb, 108
∼Reverb, 108
clear, 108
lastOut, 108
lastOutLeft, 108
lastOutRight, 108
Reverb, 108
setEffectMix, 108
tick, 108

Rhodey, 110
∼Rhodey, 110
noteOn, 110
Rhodey, 110
setFrequency, 110
tick, 110

RtAudio
getDeviceCount, 112
RtAudio, 114
RtAudio, 112
∼RtAudio, 115
abortStream, 117
cancelStreamCallback, 116
closeStream, 117
getDeviceInfo, 116
getStreamBuffer, 116
openStream, 115
RtAudio, 114
RTAUDIO FLOAT32, 118
RTAUDIO FLOAT64, 118
RTAUDIO SINT16, 118
RTAUDIO SINT24, 118
RTAUDIO SINT32, 118
RTAUDIO SINT8, 118
setStreamCallback, 115
startStream, 117
stopStream, 117
streamWillBlock, 117
tickStream, 117
RtAudio::RTAUDIO DEVICE, 119
hasDuplexSupport, 120
INDEX

isDefault, 120
maxDuplexChannels, 120
maxInputChannels, 119
maxOutputChannels, 119
minDuplexChannels, 120
minInputChannels, 120
minOutputChannels, 120
name, 119
nativeFormats, 120
nSampleRates, 120
probed, 119
sampleRates, 120

RTAUDIO_FLOAT32
  RtAudio, 118
RTAUDIO_FLOAT64
  RtAudio, 118
RTAUDIO_SINT16
  RtAudio, 118
RTAUDIO_SINT24
  RtAudio, 118
RTAUDIO_SINT32
  RtAudio, 118
RTAUDIO_SINT8
  RtAudio, 118
RtDuplex
  ~RtDuplex, 122
  lastFrame, 122
  lastOut, 122
  RtDuplex, 123
RtDuplex, 122
  ~RtDuplex, 122
  start, 123
  stop, 124
  tick, 124
  tickFrame, 124
RtError
  ~RtError, 125
  getMessage, 125
  getType, 125
  printMessage, 125
  RtError, 125
RtError, 125
RtMidi
  ~RtMidi, 127
  nextMessage, 128
RtWvIn
  ~RtWvIn, 129
  lastFrame, 129
  lastOut, 129
  RtWvIn, 130
RtWvIn, 129
  ~RtWvIn, 129
  start, 130
  stop, 131
  tick, 131
  tickFrame, 131
RtWvOut
  ~RtWvOut, 132
  getFrames, 132
  getTime, 132
  RtWvOut, 133
RtWvOut, 132
  ~RtWvOut, 132
  start, 133
  stop, 133
  tick, 134
  tickFrame, 134
Sampler, 135
  ~Sampler, 135
  clear, 135
  controlChange, 136
  keyOff, 135
  keyOn, 135
  noteOff, 135
  Sampler, 135
  setFrequency, 135
  tick, 135
  sampleRate
  Stk, 160
  sampleRates
  RtAudio::RTAUDIO_-DEVICE, 120
Saxofony, 137

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
~Saxofony, 137
  clear, 137
  controlChange, 138
  noteOff, 138
  noteOn, 137
  Saxofony, 137
  setBlowPosition, 137
  setFrequency, 137
  startBlowing, 137
  stopBlowing, 137
  tick, 138
setA1
  BiQuad, 18
  OnePole, 87
  PoleZero, 100
  TwoPole, 179
setA2
  BiQuad, 18
  TwoPole, 179
setAllpass
  PoleZero, 101
setAllTimes
  ADSR, 12
setAttackRate
  ADSR, 12
setAttackTime
  ADSR, 12
setB0
  BiQuad, 18
  OnePole, 87
  OneZero, 89
  PoleZero, 100
  TwoPole, 179
  TwoZero, 182
setB1
  BiQuad, 18
  OneZero, 89
  PoleZero, 100
  TwoZero, 182
setB2
  BiQuad, 18
  TwoZero, 182
setBaseLoopGain
  PluckTwo, 99
  StifKarp, 159
setBlocking
  Socket, 153
  setBlockZero
  PoleZero, 101
  set BlowPosition
  Saxofony, 137
  setBodySize
  Mandolin, 68
  setCoefficients
  Filter, 50
  setControl1
  FM, 54
  setControl2
  FM, 54
  setDecay
  Mesh2D, 70
  setDecayRate
  ADSR, 12
  setDecayTime
  ADSR, 12
  setDelay
  Delay, 36
  DelayA, 39
  DelayL, 41
  Echo, 44
  set Denominator
  Filter, 50
  setDetune
  PluckTwo, 98
  setDirectGain
  Modal, 75
  setEffectMix
  Chorus, 31
  Echo, 44
  PitShift, 94
  Reverb, 108
  setEndReflection
  Flute, 51
  set EqualGainZeroes
  BiQuad, 20
  Resonate, 106
  setFilterSweepRate
  VoicForm, 191
  setFreqAndDetune
  PluckTwo, 98
  setFrequency
  BandedWG, 14

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
INDEX

BlowBotl, 21
BlowHole, 23
Bowed, 25
Brass, 29
Clarinet, 33
Flute, 51
FM, 53
FMVoices, 56
Instrmnt, 64
Modal, 75
Moog, 81
PercFlut, 91
Plucked, 96
PluckTwo, 98
Rhodey, 110
Sampler, 135
Saxofony, 137
Simple, 142
SingWave, 144
Sitar, 147
StifKarp, 158
Voicer, 190
VoicForm, 191
WaveLoop, 194
Whistle, 195
Wurley, 197
setGain
  BiQuad, 20
  Filter, 50
  FM, 53
  OnePole, 88
  OneZero, 90
  PoleZero, 101
  TwoPole, 180
  TwoZero, 183
setGainRate
  SingWave, 145
setGainTarget
  SingWave, 145
setInputPosition
  Mesh2D, 70
setInterpolate
  Wvln, 202
setJetDelay
  Flute, 51
setJetReflection
  Flute, 51
  setLip
    Brass, 29
  setMass
    Sphere, 156
  setMasterGain
    Modal, 75
  setModDepth
    Chorus, 31
  setModeGain
    Modal, 75
  setModFrequency
    Chorus, 31
  setModulationDepth
    FM, 54
    ModalBar, 177
  setModulationSpeed
    FM, 53
    Moog, 81
  setNotch
    BiQuad, 20
    Resonate, 106
    TwoZero, 183
  setNumerator
    BiQuad, 20
    Filter, 50
setNX
  Mesh2D, 70
setNY
  Mesh2D, 70
setOffset
  BowTabl, 28
  ReedTabl, 105
setPhoneme
  VoicForm, 191
setPitchPosition
  StifKarp, 158
setPitchSweepRate
  VoicForm, 192
setPluckPosition
  PluckTwo, 98
setPole
  OnePole, 88
setPosition
  Sphere, 156
setPreset

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
BandedWG, 14
ModalBar, 77
setRadius
Sphere, 156
setRandomGain
Modulate, 79
SingWave, 144
setRate
Envelope, 40
SubNoise, 165
WvIn, 202
setRatio
FM, 53
setRatioAndRadius
Modal, 75
setReleaseRate
ADSR, 12
setReleaseTime
ADSR, 12
setResonance
BiQuad, 19
FormSwep, 59
Resonate, 106
TwoPole, 180
setRtDelta
Messager, 72
setSampleRate
Stk, 162
setShift
PitShift, 94
setSlope
BowTabl, 28
ReedTabl, 165
setStates
FormSwep, 58
setStickHardness
ModalBar, 77
setStreamCallback
RtAudio, 115
setStretch
StifKarp, 158
setStrikePosition
BandedWG, 14
ModalBar, 77
setSustainLevel
ADSR, 12
setSweepRate
FormSwep, 59
SingWave, 145
setSweepTime
FormSwep, 59
setTarget
ADSR, 12
Envelope, 46
setTargets
FormSwep, 58
setTime
Envelope, 46
setTonehole
BlowHole, 23
setUnVoiced
VoicForm, 191
setValue
ADSR, 12
Envelope, 46
setVelocity
Sphere, 156
setVent
BlowHole, 23
setVibrato
Bowed, 25
setVibratoGain
Modulate, 79
SingWave, 144
setVibratoRate
Modulate, 79
SingWave, 144
setVoiced
VoicForm, 191
setX
Vector3D, 185
setXYZ
Vector3D, 185
setY
Vector3D, 185
setZ
Vector3D, 185
setZero
OneZero, 90
Shakers, 139
~Shakers, 139
controlChange, 139
The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
INDEX

SubNoise, 165
subRate
  SubNoise, 165
swap16
  Stk, 161
swap32
  Stk, 161
swap64
  Stk, 161

Table, 167
  ~Table, 167
  getLength, 167
  lastOut, 167
  Table, 167
  tick, 167, 168

TcpWvIn
  ~TcpWvIn, 169
  lastFrame, 169
  lastOut, 169
  TcpWvIn, 170
  tick, 169
  tickFrame, 170

TcpWvIn, 169
  isConnected, 171
  listen, 170
  TcpWvIn, 170

TcpWvOut
  ~TcpWvOut, 172
  disconnect, 172
  getFrames, 172
  getTime, 172
  TcpWvOut, 172, 173

TcpWvOut, 172
  connect, 174
  TcpWvOut, 173
  tick, 174
  tickFrame, 174

test
  Thread, 175

Thread, 175
  ~Thread, 175
  start, 176
  test, 175
  Thread, 175
  wait, 176

tick
  ADSR, 13
  BandedWG, 15
  BeeThree, 16
  BiQuad, 19
  BlowBotl, 21
  BlowHole, 24
  Bowed, 26
  BowTabl, 27
  Brass, 30
  Chorus, 32
  Clarinet, 34
  Delay, 36
  DelayA, 39
  DelayL, 40
  Drummer, 42
  Echo, 44
  Envelope, 47
  Filter, 49
  Flute, 52
  FM, 54
  FMVoices, 56
  FormSwep, 59
  HevyMetl, 61
  Instrmnt, 64
  JCRev, 65
  JetTabl, 66
  Mandolin, 68
  Mesh2D, 71
  Modal, 76
  Modulate, 79
  Moog, 81
  Noise, 83
  NRev, 85
  OnePole, 88
  OneZero, 90
  PercFlut, 91
  PitShift, 94
  Plucked, 96
  PluckTwo, 99
  PoleZero, 101
  PRCRev, 103
  ReedTabl, 104
  Resonate, 107
  Reverb, 108
  Rhodey, 110

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
RtDuplex, 124
RtWvIn, 131
RtWvOut, 134
Sampler, 135
Saxofony, 138
Shakers, 139
Simple, 142
SingWave, 145
Sitar, 147
Sphere, 157
StifKarp, 159
SubNoise, 165
Table, 167, 168
TcpWvIn, 169
TcpWvOut, 174
TubeBell, 177
TwoPole, 180
TwoZero, 183
Voicer, 188
VoicForm, 192
Whistle, 197
Wurley, 197
WvIn, 202
WvOut, 206
tickFrame
  RtDuplex, 124
  RtWvIn, 131
  RtWvOut, 134
  TcpWvIn, 170
  TcpWvOut, 174
  WaveLoop, 193
  WvIn, 203
  WvOut, 206
tickStream
  RtAudio, 117
TubeBell
  ~TubeBell, 177
  noteOn, 177
  tick, 177
  TubeBell, 177
TubeBell, 177
TwoPole
  ~TwoPole, 179
  clear, 179
  getGain, 180
  lastOut, 180
  setA1, 179
  setA2, 179
  setB0, 179
tick, 180
TwoPole, 179
  setGain, 180
  setResonance, 180
TwoZero
  ~TwoZero, 182
  clear, 182
  getGain, 182
  lastOut, 183
  setB0, 182
  setB1, 182
  setB2, 182
tick, 183
TwoZero, 182
  setGain, 183
  setNotch, 183
Vector3D, 185
  ~Vector3D, 185
  getLength, 185
  getX, 185
  getY, 185
  getZ, 185
  setX, 185
  setXYZ, 185
  setY, 185
  setZ, 185
Vector3D, 185
voiceGain
  Phonemes, 93
Voicer, 187
  ~Voicer, 187
  addInstrument, 189
  controlChange, 188
  lastOut, 188
  noteOff, 189
  noteOn, 189
  pitchBend, 188
  removeInstrument, 189
  setFrequency, 190
  silence, 188

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick</td>
<td>188</td>
</tr>
<tr>
<td>Voicer</td>
<td>187</td>
</tr>
<tr>
<td>VoicForm</td>
<td>191</td>
</tr>
<tr>
<td>~VoicForm</td>
<td>191</td>
</tr>
<tr>
<td>clear</td>
<td>191</td>
</tr>
<tr>
<td>controlChange</td>
<td>192</td>
</tr>
<tr>
<td>noteOff</td>
<td>192</td>
</tr>
<tr>
<td>noteOn</td>
<td>192</td>
</tr>
<tr>
<td>quiet</td>
<td>192</td>
</tr>
<tr>
<td>setFilterSweepRate</td>
<td>191</td>
</tr>
<tr>
<td>setFrequency</td>
<td>191</td>
</tr>
<tr>
<td>setPhoneme</td>
<td>191</td>
</tr>
<tr>
<td>setPitchSweepRate</td>
<td>192</td>
</tr>
<tr>
<td>setUnVoiced</td>
<td>191</td>
</tr>
<tr>
<td>setVoiced</td>
<td>191</td>
</tr>
<tr>
<td>speak</td>
<td>192</td>
</tr>
<tr>
<td>tick</td>
<td>192</td>
</tr>
<tr>
<td>VoicForm</td>
<td>191</td>
</tr>
<tr>
<td>VoicForm</td>
<td>191</td>
</tr>
<tr>
<td>wait</td>
<td>176</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>WaveLoop</td>
<td>193</td>
</tr>
<tr>
<td>~WaveLoop</td>
<td>193</td>
</tr>
<tr>
<td>addTime</td>
<td>193</td>
</tr>
<tr>
<td>tickFrame</td>
<td>193</td>
</tr>
<tr>
<td>WaveLoop</td>
<td>193</td>
</tr>
<tr>
<td>addPhase</td>
<td>194</td>
</tr>
<tr>
<td>addPhaseOffset</td>
<td>194</td>
</tr>
<tr>
<td>setMask</td>
<td>194</td>
</tr>
<tr>
<td>whatsThisController</td>
<td></td>
</tr>
<tr>
<td>SKINI,</td>
<td>150</td>
</tr>
<tr>
<td>whatsThisType</td>
<td></td>
</tr>
<tr>
<td>SKINI,</td>
<td>150</td>
</tr>
<tr>
<td>Whistle</td>
<td>195</td>
</tr>
<tr>
<td>~Whistle</td>
<td>195</td>
</tr>
<tr>
<td>clear</td>
<td>195</td>
</tr>
<tr>
<td>controlChange</td>
<td>196</td>
</tr>
<tr>
<td>noteOff</td>
<td>195</td>
</tr>
<tr>
<td>noteOn</td>
<td>195</td>
</tr>
<tr>
<td>setMask</td>
<td>195</td>
</tr>
<tr>
<td>startBlowing</td>
<td>195</td>
</tr>
<tr>
<td>stopBlowing</td>
<td>195</td>
</tr>
<tr>
<td>tick</td>
<td>195</td>
</tr>
<tr>
<td>Whistle</td>
<td>195</td>
</tr>
<tr>
<td>writeBuffer</td>
<td></td>
</tr>
<tr>
<td>Socket</td>
<td>153</td>
</tr>
<tr>
<td>Wurley</td>
<td>197</td>
</tr>
<tr>
<td>~Wurley</td>
<td>197</td>
</tr>
<tr>
<td>noteOn</td>
<td>197</td>
</tr>
<tr>
<td>setFrequency</td>
<td>197</td>
</tr>
<tr>
<td>tick</td>
<td>197</td>
</tr>
<tr>
<td>WvIn</td>
<td>199</td>
</tr>
<tr>
<td>~WvIn</td>
<td>199</td>
</tr>
<tr>
<td>addTime</td>
<td>199</td>
</tr>
<tr>
<td>closeFile</td>
<td>199</td>
</tr>
<tr>
<td>getChannels</td>
<td>200</td>
</tr>
<tr>
<td>getDuration</td>
<td>200</td>
</tr>
<tr>
<td>isFinished</td>
<td>200</td>
</tr>
<tr>
<td>lastFrame</td>
<td>200</td>
</tr>
<tr>
<td>lastOut</td>
<td>200</td>
</tr>
<tr>
<td>reset</td>
<td>199</td>
</tr>
<tr>
<td>WvIn</td>
<td>199</td>
</tr>
<tr>
<td>getFileRate</td>
<td>202</td>
</tr>
<tr>
<td>normalize</td>
<td>201</td>
</tr>
<tr>
<td>openFile</td>
<td>201</td>
</tr>
<tr>
<td>setInterpolate</td>
<td>202</td>
</tr>
<tr>
<td>setRate</td>
<td>202</td>
</tr>
<tr>
<td>tick</td>
<td>202</td>
</tr>
<tr>
<td>tickFrame</td>
<td>203</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>~WvOut</td>
<td>204</td>
</tr>
<tr>
<td>closeFile</td>
<td>204</td>
</tr>
<tr>
<td>getChannels</td>
<td>204</td>
</tr>
<tr>
<td>getFrame</td>
<td>204</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>openFile</td>
<td>204</td>
</tr>
<tr>
<td>tick</td>
<td>204</td>
</tr>
<tr>
<td>tickFrame</td>
<td>204</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>WvOut</td>
<td>204</td>
</tr>
<tr>
<td>xmlObject</td>
<td></td>
</tr>
<tr>
<td>XMLObject</td>
<td></td>
</tr>
<tr>
<td>wavObject</td>
<td></td>
</tr>
<tr>
<td>WAVout</td>
<td></td>
</tr>
<tr>
<td>WAVOUT</td>
<td></td>
</tr>
<tr>
<td>WAVOUT MAT</td>
<td></td>
</tr>
<tr>
<td>WAVOUT AIF</td>
<td></td>
</tr>
<tr>
<td>WAVOUT RAW</td>
<td></td>
</tr>
<tr>
<td>WAVOUT SND</td>
<td></td>
</tr>
<tr>
<td>WAVOUT WAV</td>
<td></td>
</tr>
<tr>
<td>WAVOUT AIF</td>
<td></td>
</tr>
<tr>
<td>wavObject</td>
<td></td>
</tr>
</tbody>
</table>

The Synthesis ToolKit in C++ by Perry R. Cook and Gary P. Scavone, © 1995–2002
WVOUT_MAT
WvOut, 207
WVOUT_RAW
WvOut, 207
WVOUT_SND
WvOut, 207
WVOUT_WAV
WvOut, 207