# JSyn – A Real-time Synthesis API for Java

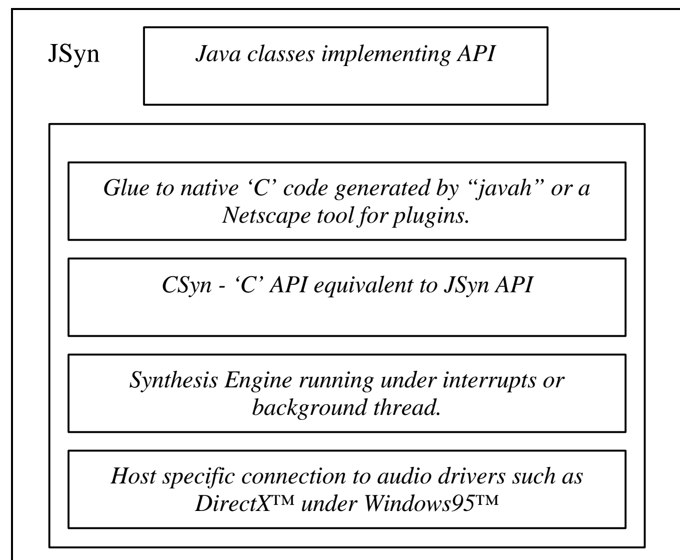
**Phil Burk**
75 Pleasant Lane, San Rafael, CA, 94901 USA
http://www.softsynth.com, philburk@softsynth.com

**Abstract:** JSyn provides real-time unit generator based synthesis for stand-alone Java applications, or Applets in a web page. Units can be created and connected "on the fly" to allow dynamic modification of synthesis topologies. The synthesis is performed by 'C' code hidden in a Netscape plug-in or DLL beneath Java native methods. JSyn uses a time-hashed event buffering system to provide accurate timing for output events. JSyn is available at "http://www.softsynth.com/jsyn".

## 1 Introduction

### *1.1 Rationale*

The majority of software synthesis packages are based on a proprietary language, or in some cases a graphical environment in which to compose music. They are excellent tools for composers but cannot be easily used by application developers such as game programmers who are using a mainstream programming language. So I decided to implement a real-time audio synthesis API (Application Programmers Interface) which can be used by programmers writing in Java. By using an existing language, I can leverage off of the large community of developers who are contributing to the Java language. This leaves me free to concentrate my efforts solely on the audio synthesis features of JSyn.




I chose Java because it is an excellent object oriented language that is designed to be host independent. Because of its portability, Java has become a popular language for the World Wide Web. Unfortunately, the original implementation of Java only supports simple playback of 8 bit ".au" audio samples at 22 KHz. But using JSyn, Java developers now have access to a flexible unit generator based synthesis engine that can provide high fidelity audio for the World Wide Web, or for stand-alone Java applications.

## 2 Application Programmer Interface (API)

### 2.1 Design Goals

I had several goals in mind when designing the API for JSyn. I wanted JSyn to be portable across multiple hosts which is a fundamental issue with any Java API. I also wanted to hide low level aspects of the implementation such as whether the low level synthesis engine was running on the CPU, or on a DSP, perhaps on a PC sound card. I also wanted the API to be minimal, yet orthogonal, complete and easy to use

### 2.2 Creating Unit Generators

Unit generators are created by instantiating Java classes defined under JSyn. For example to create a sawtooth wave oscillator, one would use:

```
SawtoothOscillator myOsc = new SawtoothOscillator();
```

### 2.3 Starting and Stopping units

Unit generators can be "started" which places them in a list of units to be executed by the synthesis engine. To start a unit, simply write:

```
myOsc.start();
```

### 2.4 Connecting ports

Unit generators have input and output ports which can be connected. This allows one to create complex circuits from multiple unit generators. To connect the output of the oscillator to the input of a filter, one would write:

```
myOsc.output.connect( myFilter.input );
```

Output ports can be connected to multiple input ports. Inputs can have only one output connected to them.

Internally, all signals that are passed between units are either `RAW_SIGNED` which ranges from -1 .0 to 1.0 or `RAW_UNSIGNED` which ranges from 0.0 to 2.0. This is to allow implementation of the synthesis engine on a fixed point DSP accelerator if needed.

### 2.5 Setting Unit Parameters and Signal Types

The inputs on a unit generator can also be set directly by the application. For example:

```
myOsc.frequency.set( fundamental * (3.0/2.0) );  /* perfect fifth */
```

The set() method accepts values in units that are appropriate to the unit generator port. For example, oscillator frequency is specified in Hertz. Other *signal types* that are converted include sample rate, filter cutoff, exponential lag decay rates, delay times, etc. The signal type of a port defines the range of allowable values, and how those values are converted to the native internal values. For example, the formula for converting between a frequency in Hertz and the internal phase increment value is:

```
nativePhaseIncrement =freqInHertz * (2.0 / audioCalculationRate );
```

Arithmetic units can be used to scale audio or control signals for precise modulation. In such a patch, the center frequency of an oscillator might be controlled by setting the input port of an `AddUnit`. Rather than force the programmer to specify frequency in that port's default signal type, `RAW_SIGNED`, the programmer can force the signal type of the port to be `OSC_FREQ`. Then the offset frequency can be specified in natural units.

```
myAdder.inputB.setSignalType(Synth.SIGNAL_TYPE_OSC_FREQ );
myAdder.inputB.set( 300.0 ); /* center frequency of modulated oscillator in Hz */
```

### 2.6 Busses

JSyn provides busses that can be used to mix an arbitrary number of signals together. Unit generators called BusWriters and BusReaders convert normal signals to bus signals. BusReaders have a `SynthBusInput` which can have multiple `SynthBusOutput` ports connected to it. All of the `SynthBusOutputs` connected to a `SynthBusInput` are summed together.

### 2.7 Data Types – Samples versus Tables, Envelopes

JSyn supports three classes of containers for data. The application can only access the data inside these classes through read() and write() methods. The data cannot reside in arrays in the user program while being used by the engine. This allows the implementer of JSyn to place the data in a special memory area such as the RAM on a sound card, or in local static RAM on a DSP. This also gives the implementer an opportunity to flush the CPU data cache in case the audio data needs to be accessed via DMA hardware.

A **`SynthSample`** object contains integer data, typically 16-bit mono or stereo. It is used for storing digital audio samples such as might be read from an AIFF or WAV file. The data can only be read or written sequentially by unit generators such as the `SampleReader_16V1` or the `SampleWriter_16F1`. Readers and writers can be combined to implement long delay effects like multi-tap echoes which are useful for modeling early reflections in a room. Since samples are large, they are most likely to reside in main memory or in DRAM on a sound card.

A **`SynthTable`** object may contain fixed point data, or floating point data if the host supports it. `SynthTables` can be accessed randomly by unit generators such as the `WaveShaper`, `TableOscillator` or short delay units. Tables are usually small and could possible be stored in static RAM on a DSP.

A **`SynthEnvelope`** contains duration-value pairs. They are accessed sequentially and may be stored in main memory or in static RAM on a DSP. Envelopes may be used for contour generation, or breakpoint oscillators.

### 2.8 Queuing

Applications can queue blocks of data within a `SynthSample` to a `SynthSampleQueue` that is a port on a unit generator. The blocks of data will be played in order until the queue is empty. As an option, a data block can be queued with the LOOP flag set which will cause the block to be played repeatedly if it is the last block in the

queue. If another block is queued while another block is looping, the loop will finish and then the new block will be played. Here is an example of queuing the attack portion of a sample followed by a sustain loop.

```
        mySampler.samplePort.queue( mySamp, 0, attackSize );
        mySampler.samplePort.queueLoop( mySamp, loopStart, loopSize );
```

Blocks of `SynthEnvelope` data can be queued for playback on the envelope playing unit generators in the same fashion.

### 2.9 Hierarchical Circuits (Patches)

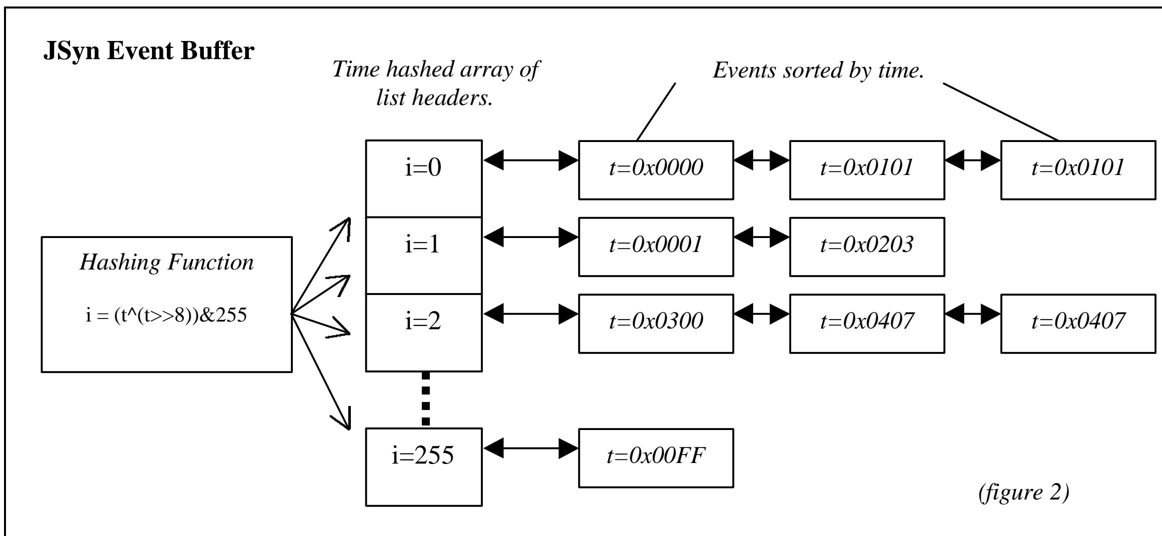
By sub-classing the `SynthCircuit` class, an application programmer can combine a number of unit generators connected together in a patch. These circuits can have programmer defined ports and can be used in place of unit generators. When a circuit is started, all of the units within it are started simultaneously allowing precise timing relationships between units to be preserved.

### 2.10 Event Buffering

The Java Virtual Machine does not provide the precise real-time scheduling generally required for music. JSyn, therefore, provides an event buffer that allows one to schedule critical operations at a specific time. You can schedule the starting and stopping of unit generators, the setting of unit parameters, and the queuing of sample and envelope data.

For timing, JSyn uses a tick counter which is incremented every time a block of 64 samples is calculated. Here is an example that queries the current time and then schedule operations in the future.

```
        int time = Synth.getTickCount();
    /* Start 300 ticks in the future. */
        myOsc.start( time + 300 );
    /* Set frequency to 220 Hz 400 ticks after starting. */
        myOsc.frequency.setAt( time + 700, 220.0 );
```



*(figure 2)*

## 3 Implementation

### 3.1 Time Hashed Event Buffer, Event types

The event buffer in JSyn stores time stamped events in linked lists. (See figure 2.) When events are placed into the buffer by the foreground task, the background task must be held off to prevent corruption of the linked lists. If the background task is held off too long then JSyn may not be able to keep up with real-time. So insertion, and also removal from the buffer, must be very efficient. I experimented with several techniques and settled on a time hashed array of linked lists.

When an event is to be posted, the timestamp value is converted to an array index using a hashing function. The index is used to access an array of linked list headers. The event is then inserted into the indexed list in sorted order. I initially used the hashing function {i=t&255} but found that when I scheduled my events at increments of

a power of 2 ticks, such as every 64 ticks, then the events piled up in just a few of the lists. Those lists became quite long which slowed down the insertion sort. By XORing the first and second bytes of the time value, {i = (t^(t>>8))&255}, I found that the events spread more evenly among the lists.

### 3.2 Secure API for Java, Tokens and Deferred Deletion

One of the key features of the Java language is that unfamiliar programs found on the web can be run safely on peoples' home computers. Java does this by using several techniques including eliminating the use of pointers and by checking all array references for over indexing. Native 'C' libraries used by Java must be designed so that unscrupulous programmers cannot wreak havoc on someone else's machine. For this reason the underlying 'C' API does not use any pointers except for pointers to safe Java objects. When CSyn, the native 'C' synthesis engine, allocates a data structure, the pointer is kept internally and an integer token is passed back to the Java program. When tokens are passed back to CSyn from Java they are checked for validity before use.

When a program dynamically allocates lots of resources, a problem can occur if a resource is deleted while there are still outstanding references to it. For example, if you used the event buffer to schedule the starting and stopping of a unit in the far future, then immediately deleted that unit, you might have references to a non-existent unit in the event buffer. When it came time to start that unit, the 'C' engine could crash. To avoid that problem, all references by one resource to another resource are counted, and deletion is deferred internally until all of the references are eliminated. Thus, in the above case, deletion will not take place internally until after the stop event has been executed. One can take advantage of this feature by scheduling sound events in the future and then immediately deleting all the objects involved. The actual deletion will not occur until after all the events have safely finished.

### 3.3 Performance Optimization

In order to achieve reasonable performance on a general purpose CPU, I employed several optimization techniques. All sample computation occurs in blocks of 8 samples. As suggested by Adrian Freed, arrays are indexed using explicit indices (a[I]) instead of pointers (*a++). Sine waves are generated using a Taylor expansion out to the term (x**9/9!) which is more accurate, and faster than a table lookup on a Pentium.

Here are benchmarks from a 233MHz Pentium MMX showing percentage CPU cycle utilization.

| Envelope_Segmented | 0.80% |
|---|---|
| Osc_Sine | 1.65% |
| Osc_Table | 1.45% |
| Sample_Read16V1 | 1.59% |

| Noise_White | 0.63% |
|---|---|
| Filter_StateVariable | 1.79% |
| Osc_Impulse | 0.43% |
| Osc_Triangle | 0.88% |

## 4 Applications Written Using JSyn

JSyn has been used to implement several interactive audio programs running on the web that can be explored by visiting http://www.softsynth.com/jsyn using Netscape Communicator. One application is a traditional drum-box that allows multiple players to perform together in real-time by communicating through a server written by the author. More abstract interactive compositions, and an on-line synthesis tutorial are in the works.

Nick Didkovsky and the author are currently using JSyn to implement JMSL, a Java Music Specification Language that incorporates many of the design concepts of HMSL (Burk et al). Robert Marsanyi has used JSyn as a foundation for Wire, a graphical patch editor.

## 5 Summary

JSyn provides a flexible unit-generator based synthesis API for Java programmers. The high fidelity of JSyn's audio output, and the precision with which musical events can be scheduled, make it a suitable API for the development of on-line games, interactive compositions, or virtual environments. The wealth of employment opportunities for Java programmers also suggests that JSyn would be an appropriate tool for teaching audio synthesis at the college level.

## 6 Reference Links

Phil Burk, Larry Polansky, David Rosenboom, "HMSL", http://www.softsynth.com/hmsl

Adrian Freed, "Clear, Efficient Audio Signal Processing in ANSI C", http://cnmat.CNMAT.Berkeley.EDU/~adrian/Csigproc.html