SynthScript Specification R1.1.3

Gregory Pat Scandalis

Other Reference Documents

Documentation for the R1.0 Unit Generator Set

Documentation for the Public API

Glossary

.dla

The extension for a file that contains down loadable algorithms. A .dla file may contain either a single algorithm (Algorithm and Subalgorithms), a collection of Algorithms, a preset or a collection of presets, a palette of algorithms and presets, a collection of palettes, or SynthScript runtime command language.

Algorithm

An Audio Signal Processing Algorithm. For legacy reasons the older keywords Patch and Subpatch will be supported through R1.5.

Algorithm is the hierarchical root of an algorithm. Allocated algorithm instances are constructed by starting from the root algorithm, flattening the algorithm to its leaf UnitGenerator and EventFilter instances, and allocating all of those instances. Algorithms are mapped to MIDI program locations, meaning that they are "stored" in MIDI space at program locations which consist of a Bank MSB, a Bank LSB, and a Program location. Algorithms are allocated on MIDI channels.

Allocation Order

The instances of UnitGenerators and EventFilters defined in a patch are allocated in the order that they are instanced in the SynthScript. This is important because the allocation order determines the evaluation order. Evaluation order in algorithms is important because feedback loops create implied delay at some point in the loop, depending on the order that the UnitGenerators in the loop are evaluated.

Allocated Algorithm Instance

A running (computing) instance of an algorithm.

Control Pin

This is a pin on a hierarchical SubAlgorithm that can be used to enable/disable the SubAlgorithm from calculation.

Down Loadable Algorithm (DLA)

An algorithm that has been loaded into SynthCore as a template and allocated as an allocated algorithm instance.

Event

An object that is passed down a EventFilter chain. This object can contain any number of parameters, some of which are common MIDI parameters.

Event Driven

Computation that responds to events in time.

EventFilter

An event driven computation unit that is used to process control data.

Instance

In SynthScript, algorithms consist of instances of UnitGenerators, EventFilters, AudioNets and EventNets.

Palette

A collection of Algorithms, SubAlgorithms and presets that share a MIDI program space. SynthCore provides a default Palette,

Part

A part corresponds roughly to a MIDI channel. SynthStreams have 16 parts which can be mapped to MIDI channels.

Physical Model

An algorithm that simulates a physical sound generation process.

Pin

A connection point on a UnitGenerator or EventFilter.

State Variable

Variables that are part of a UnitGenerator or EventFilter that reflect its current state.

StateProc

A procedure that can set the value of a state variable for a particular instance of a UnitGenerator or a EventFilter.

SubAlgorithm

A hierarchical element of a patch.

SynthBuilder

Staccato’s internal (for now) tool for authoring SynthScript algorithms.

SynthCore

Staccato’s realtime software algorithmic and wavetable synthesis engine.

SynthScript

Staccato's interchange format for describing algorithms.

SynthStream

Simply, a SynthStream is an input to the system mixer, where Synthesis algorithms can be allocated (on parts which are roughly MIDI channels). On the WinX platform, the benefit of allocating algorithms on different SynthStreams, is that they can be allocated as DirectSound 3-D buffers that can then be controlled from the DS3-D APIs.

Template

An algorithm that has been read into SynthCore is stored as a template. This template can be used to allocate running (computing) instances of the algorithm. Templates are stored either in the default palette, or in a named palette.

Tick Size

The number of samples that are calculated for each calculation loop. This corresponds to the size in samples in an AudioNet that connects two UnitGenerators.

UnitGenerator

An Audio Signal Processing Element. This is a term that was coined by Max Mathews in his MusicX series of programs.

CHAPTER 0: Introduction

SynthScript is a text based interchange format that Staccato Systems Inc has designed to represent downloadable audio signal processing algorithms or DLAs. DLAs that are defined in SynthScript can be "allocated" and controlled by Staccato's Software Synthesis Engine, SynthCore. Examples of possible DLA algorithms are Physical Models, such as a feedback distortion guitar, or a car engine. Other possible algorithms are familiar signal processing units such a reverb or a flanger.

Staccato has developed a tool known as SynthBuilder that is used to author SynthScript algorithms. SynthBuilder communicates to Synthcore using SynthScript. On the WinX platforms, this interactive connection is accomplished with a COM interface in SynthCore. SynthBuilder can also save an authored algorithm as a Down Loadable Algorithm file, known as a DLA file (.dla). This DLA file can be loaded into SynthCore independent of the SynthBuilder authoring environment.

SynthScript's design is in part influenced by the design of the DLS standard, and because of this is MIDI-centric.

Note that other tools may one day exist that also generate SynthScript.

0.1 About SynthScript

SynthScript is an interchange format. An algorithm described in SynthScript primarily consists of instantiations of signal processing and control objects, connectivity between these objects, and data declarations. The following are some of the features of SynthScript.

- SynthScript is free format text. This allows users to edit raw SynthScript if necessary. Also other authoring tools can easily be developed that emit SynthScript.
- It provides a compact representation for connectivity between instances of signal processing and control units.
- It provides a compact representation for assigning values to the state variables of signal processing and control

units.

- It provides a compact representation for binding control to the state variables of signal processing and control units. Algorithms can be controlled with common MIDI controls as well as user defined controls that can be accessed from the API.
- It provides data types for representing data that is a part of an algorithm.
- It provides a representation for mulitple algorithms as well as a representation of hierarchical processing units known as "SubAlgorithms".
- The allocation system supports "flattening" of Algorithms and SubAlgorithms, which is analogous to link resolution in programming languages.
- It provides a representation of algorithms that share a common program change space, known as "Palettes".
- It is compatible with MIDI. Algorithms have associated program change numbers, and can be allocated with a simple MIDI program change. Algorithms can be allocated on MIDI channels and controlled with common MIDI controllers.
- There is a SynthCore API that can be used to load a SynthScript file, allocate the algorithms defined in a SynthScript file, and manipulate the exported controllers of an allocated algorithm.
- A tag mechanism is provided to support user defined extensions to SynthScript. This mechanism can be use to represent things such as graphical coordinates or any other user defined value.
- A simple XOR based encryption mechanism is provided for encryption/decryption of SynthScript files.

0.2 A Brief History of SynthScript

The original NeXT based version of SynthBuilder was a graphical editing program that made direct API calls to the NeXT Music Kit, which in turn rendered audio algorithms on the NeXT computer's Motorola 56k DSP. With this version of SynthBuilder it was possible to author algorithms, but there was not an easy way to export the algorithms in a runtime format that was independent of SynthBuilder. Further the Synthesis engine was not portable because it was coupled to the 56k.

SynthScript was designed as a way to decouple the synthesis engine from the authoring tool. Further we envisioned SynthCore as a portable synthesis system that would use host based computing resources. Because algorithms are authored in SynthScript, they can be run on a version of SynthCore that has been ported to different platforms.

The first version of SynthScript was developed in the spring 1996 at Stanford University as a part of the Sondius program. An early version of SynthCore and SynthScript were demonstrated at Stanford's CCRMA in May of 1996.

Nick Porcaro assisted with the initial design of SynthScript.

The connectivity and data format of SynthScript is similar to CAD formats, such as Verilog or VHDL, that are used to exchange logic design data.

The R0.7 version of SynthScript, dating to the 1996 pre Staccato time frame only supported connectivity, and a single algorithm.

The R0.8 version of SynthScript that dates to the Oct 1997 time frame had limited support for data, and still only supported a single algorithm.

The R0.9 version of SynthScript that dates to the April 1998 Concerto Prototype is uses the list and array paradigms for data, and is upwardly compatible with the R1.0 version of SynthScript.

The R1.0 version of SynthScript adds an include file mechanism, a new data type "Event", support for reading sound files (.wav), the Palette mechanism, support for probing, and support for Presets.

Future versions of SynthScript will probably support Scenes (a predefined collection of allocations), and an event processing language.

0.3 Other documents to refer to

Note that there are several other documents that should be referred to when reading this spec.

This spec does not discuss in depth, the signal processing and control units which are known respectively as UnitGenerators (a term coined by Max Mathews in the late 50s as a part of his Music line of programs), and EventFilters. The UnitGenerators and EventFilters that are part of R1.0 are defined in the document "UgDocR1.0.doc".

This spec does not discuss in depth the SynthCore public API that can be used to load, allocate and control SynthScript algorithms. That API is discussed in the document "SDKpublicAPI.doc".

0.4 A Note about this document

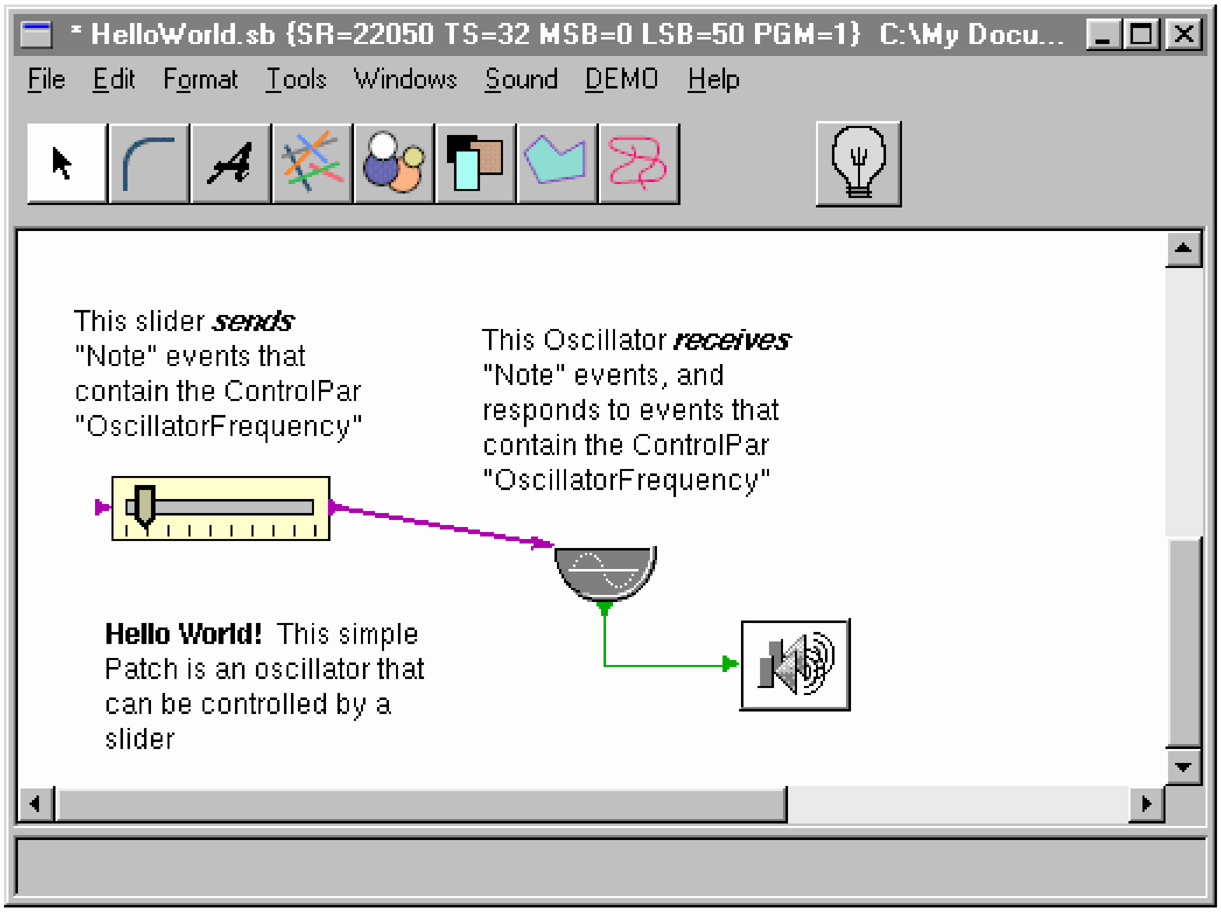
This document approaches the specification of SynthScript at 3 levels. The first level consists of a tutorial in the form of a number of examples. The second level consists of qualitative explanations of each portion of the syntax and semantics of SynthScript. The last level is the formal syntax of SynthScript.

CHAPTER 1: A Tutorial Introduction

1.1 The HelloWorld Oscillator Algorithm

Let us begin with a quick introduction to SynthScript. The following example will show some of the essential elements of a SynthScript algorithm. Later, details will be explained about the programming model that SynthScript presents, as well as the run time system that supports this. Note that through out this document, I'll use SynthBuilder algorithms as examples. However, this document is not intended to be a SynthBuilder tutorial.

This is an example of a very simple Synthesis algorithm, known as an "Algorithm". This very simple example doesn't do very much. It's a simple sine wave oscillator, and the frequency of the oscillator can be tuned by moving a slider on the screen. What is interesting about this example, is that it demonstrates some of the things that are common to most algorithms, and how those things are represented in SynthScript.


* HelloWorld.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM=1} C:\My Docu...
File Edit Format Tools Windows Sound DEMO Help

This slider ***sends*** "Note" events that contain the ControlPar "OscillatorFrequency"

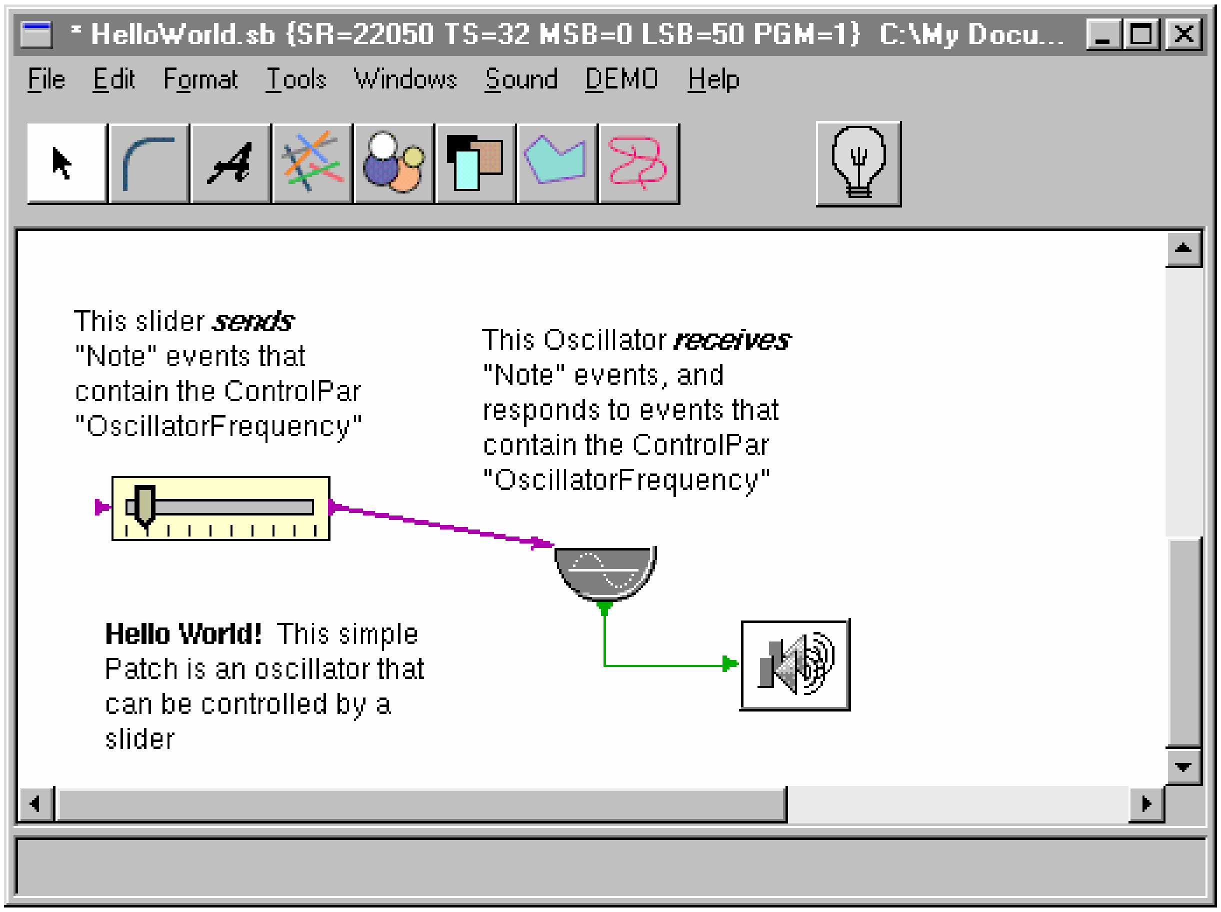
This Oscillator ***receives*** "Note" events, and responds to events that contain the ControlPar "OscillatorFrequency"

Hello World! This simple Patch is an oscillator that can be controlled by a slider

The HelloWorld Oscillator Algorithm

Here are some things to notice about this algorithm:

- Part of the algorithm, the oscillator and the speaker, deals with the audio processing chain and is computed synchronously at the algorithm's sampling rate. Notice that the oscillator and speaker are connected or "wired" together with what is known as an "AudioNet".
- Part of the algorithm is used to "control" the audio processing chain, the slider. The control portion of the algorithm is computed asynchronously and is "event driven". Notice that the slider and the oscillator are "wired" together with what is known as an "EventNet".
- The slider object sends "events" when it is moved. Events are very fundamental to how SynthCore works. An event in SynthCore is simply an object that is passed down a chain of EventNets. This event object can contain a list of any number of control parameters known as ControlPars. The ControlPars that are part of an event object have types (such as Real, Int, RealArray, IntArray, List as well as other possible data types) and values. I'll discuss SynthCore/SynthScript data types in greater detail later. In this example, the slider is sending events that contain the ControlPar "freq", which is of type Real, and contains values such as 440.0.
- A slider object can be specify the ControlPar that is inserted into events that it ***sends***. For example in this algorithm a slider is sending the "OscillatorFrequency" ControlPar (oscillator frequency parameter).
- An oscillator object also can be used to specify the ControlPar that it responds to when it ***receives*** events. We say that the ControlPar "OscillatorFrequency" has been *bound* to the StateProc "freq" on the oscillator. StateProcs will be explained more later, but can be thought of as exposed control pins on an object that can have their states *set* from some other parameter.
- Also notice that the algorithm itself has a few useful properties as well. There is some information displayed in the title bar of the algorithm that indicates the sample rate that the algorithm is running at, the size of a computation tick (this will be explained later in detail), and the program location that the algorithm has been assigned to. The program location makes it possible to allocate copies of this algorithm on MIDI channels via a simple MIDI program change.


* HelloWorld.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM=1} C:\My Docu...
File Edit Format Tools Windows Sound DEMO Help

This slider *sends* "Note" events that contain the ControlPar "OscillatorFrequency"

This Oscillator *receives* "Note" events, and responds to events that contain the ControlPar "OscillatorFrequency"

Hello World! This simple Patch is an oscillator that can be controlled by a slider

1.2 SynthScript Instance Statements

Ok, so how is this represented in SynthScript? We need to start with a short discussion of one of the fundamental constructs in SynthScript, an instance statement. An instance statement is SynthScript's compact notation that describes when an algorithm is allocated, how to allocate an instance of one of its objects, the object's connections to other objects, the initial values passed to the object's StateProcs, and the binding of ControlPars to the object's StateProcs. Here is the instance statement for the oscillator in our example.

```
                    oscgUG MyOscillator

                    (out=oscgUGOut, control=sliderNFOut,

                              [amp=0.5,

freq=440.0:OscillatorFrequency]

                    );
```

The first token in the instance statement declares the class of the object that is to be allocated. In this case, we are allocating an instance of an oscgUG object. The second token is a unique identifier for this instance of the object.

Next, in parenthesis, you will find declarations for connectivity, and declarations for state. Immediately after the open parenthesis, you will find a list of assignment statements that describe networks that the object is connected to. These networks can be either AudioNets, or EventNets. In this case, MyOscillator has a pin called "out" that is connected to an AudioNet called "oscgUGOut", and another pin called "control" that is connected to a EventNet called "sliderNFOut".

Next you will find in square brackets a "vector" of StateProcs, assigned to initial values, and sometimes bound to ControlPars. First there is a StateProc called "amp" which is initially set to a value of 0.5. From UgDocR1.0.doc we can find that this StateProc sets the amplitude (volume) of this oscillator, and that it ranges from 0.0 to 1.0. Next there is a StateProc called freq, which is initially set to 440.0, and is bound to a ControlPar called OscillatorFrequency.

Finally you will find a ';' character which is the termination of the instance statement.

SynthScript algorithms consist of a series of instance declarations, some of which are object instances, and some of which are data.

1.3 The HelloWorld Oscillator Algorithm in SynthScript.

On the next page, you will find the complete SynthScript description for the simple HelloWorld algorithm, along with extensive comments describing the notation in the algorithm. Here are a few things to notice about the SynthScript description.

- Its free format text
- Comments can be entered with standard c, c++ notation /* */ and //.
- Note that when SynthScript is read into SynthCore it is validated syntactically with the SynthScript Parser. If errors are detected they are indicated with error messages such as this:

```
        ...

        (0000028) oscgUG MyOscillator (0000029) (out=oscgUGOut, ; control=sliderNFOut, ^

        parse error at line [0000029] between <,>, and <;>.

        (0000030) [amp=0.5, (0000031) freq=440.0:OscillatorFrequency, (0000032) phase=0.0,
        (0000033) trace=0](0000034) );

        ...

        SynthCore: The SynthScript patch HelloWorld was read,

        0 warnings, 1 errors.
```

```
/* This is an algorithm called HelloWorld.

This is a SynthScript Version 1 file.

The algorithm is running at 22k sampling rate,

The algorithm is assigned to program location 0/50/0.

This means that a simple MIDI program change to bank 50, program 1

will direct SynthCore to allocate and instance of this algorithm.

/*

Algorithm HelloWorld([Version=1,

                    SamplingRate=22050,

                    TickSize=32,

                    BankMSB=0,

                    BankLSB=50,

                    Program=1])

{ // begin HelloWorld!

// OscillatorFrequency is a ControlPar that will be used in

// this algorithm

ControlPar OscillatorFrequency;

// EventNet that connects the slider to the oscillator

EventNet sliderNFOut ();
// AudioNet that connects the oscillator to the speaker

AudioNet oscgUGOut ();

// This is an instance of an oscillator

oscgUG MyOscillator

          (out=oscgUGOut,

          control=sliderNFOut,

                    [amp=0.5,

                    freq=440.0:OscillatorFrequency]

);

// This is an instance of the speaker

out2sumUG MySpeaker

          (in=oscgUGOut,

                    [bearing=0.0,

                    scale=1.0]

          );

// This is an instance of the slider. It is exported

// so that it can be accessed and controlled with the

// SynthCore API. Notice that the instance name

// is not just a plain token, it’s a quoted string.

// This will allow for a nice name on an exported UI

EXPORTED numberVarNF "My Freq Event Slider"

          (out=sliderNFOut,

          [par= OscillatorFrequency,

          initialize=1,

          max=5000.0,

          min=0.0,

          realValue=1913.58,

          trace=0]);

} // end of HelloWorld!
```

1.4 What happens after the algorithm is read into SynthCore?

When an algorithm is read into SynthCore its not actually computing. The process of reading an algorithm into SynthCore only creates a template assigned to a MIDI program location that can be used to allocate **multiple** copies of the algorithm. The algorithm must be allocated in order for it to begin computing. An algorithm can be allocated one of 3 possible ways:

- A MIDI program change
- A procedure call that is provided in the SynthCore API

- A SynthScript command statement.

The SynthScript command that will allocate this algorithm on MIDI channel number one looks like this:

```
SynthstreamAssignment(MidiInChannel=1,

BankMSB=0,

BankLSB=50,

Program=1);
```

Note that allocation commands can be included in a SynthScript file so that the SynthScript file is self allocating.

When an algorithm is allocated, the template is used as guide to create instances of objects in SynthCore that will do the actual computation of the algorithm. The StateProcs of these instances are initialized with the values defined in the SynthScript file, Also, ControlPars are bound to these StateProcs based on what is defined in the SynthScript file.

An important detail is that instances of objects are allocated in the order that they are defined. Also StateProcs are evaluated in the order that they are defined. Finally any data that is a part of the algorithm is stored in the template, and each instance of the algorithm references this single copy of the data.

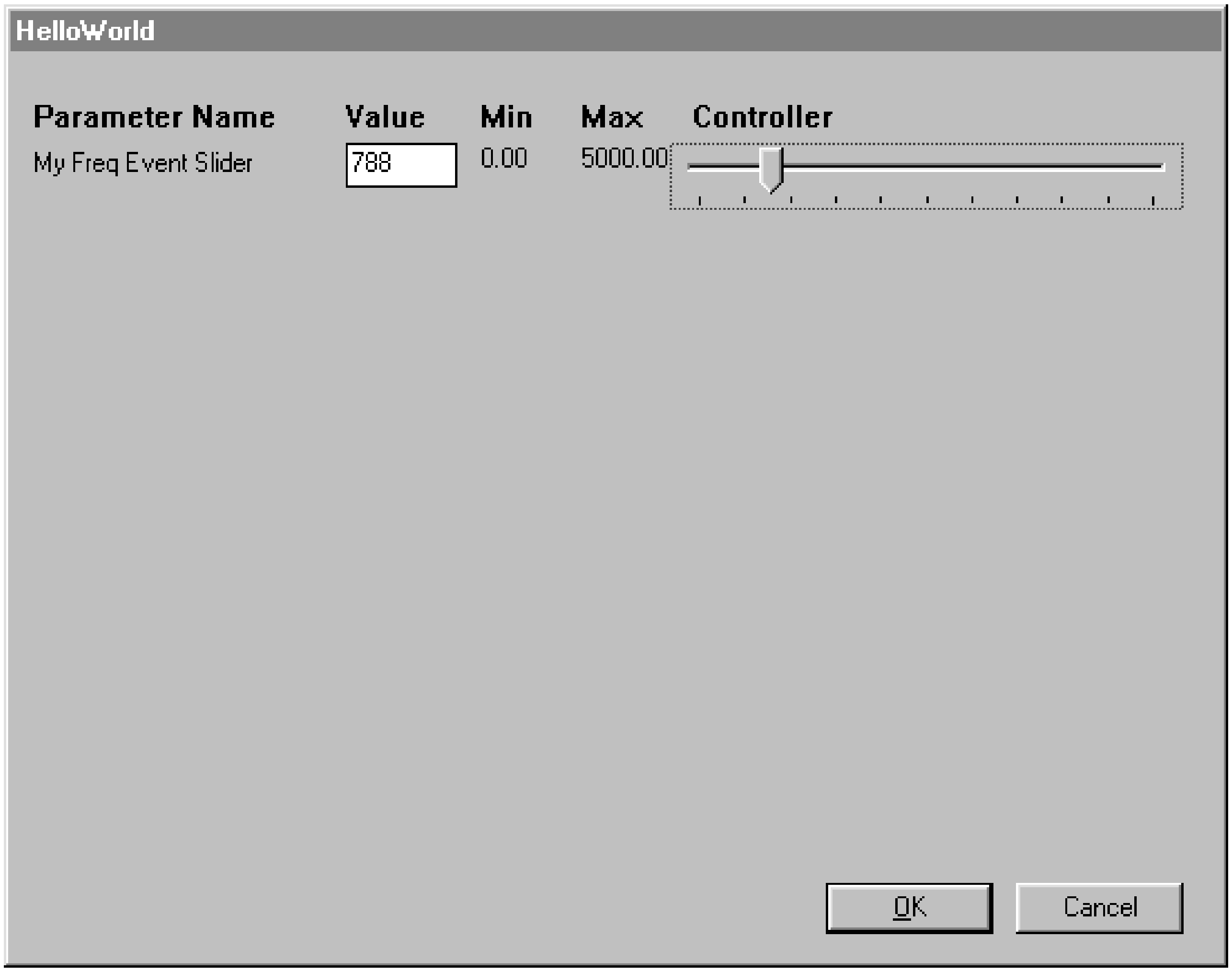
Because SynthScript supports hierarchical descriptions of algorithms, a significant part of the allocation process is flattening the hierarchy of an algorithm into allocated running objects. This flattening process is similar to linking in conventional programming languages. The example in the next chapter will demonstrate a hierarchical algorithm.

1.5 How does the public API address an algorithm?

This document will not attempt to cover deeply how the public API is used. However, its important use this example to illustrate a few points about the public API. The public API is minimal. The goals of the Public API are to initialize and terminate SynthCore, to load algorithms, and to allocate and deallocate algorithms, and to set/get exported controllers of algorithms.

Once the algorithm is loaded, and allocated, it is possible, via the API to get/set the values of exported controllers in the algorithm (recall the EXPORTED keyword associated with slider objects). The main point to take from this, is that all of the functionality of an algorithm is defined in the algorithm itself, and this functionality can be controlled from the algorithm’s exported control interface.

Below is a screen shot from one of the Staccato SynthCore Control Panels. This control interface was constructed by calling the public API to load the HelloWorld algorithm, and then set/get values associated with the exported controller "My Freq Event Slider".

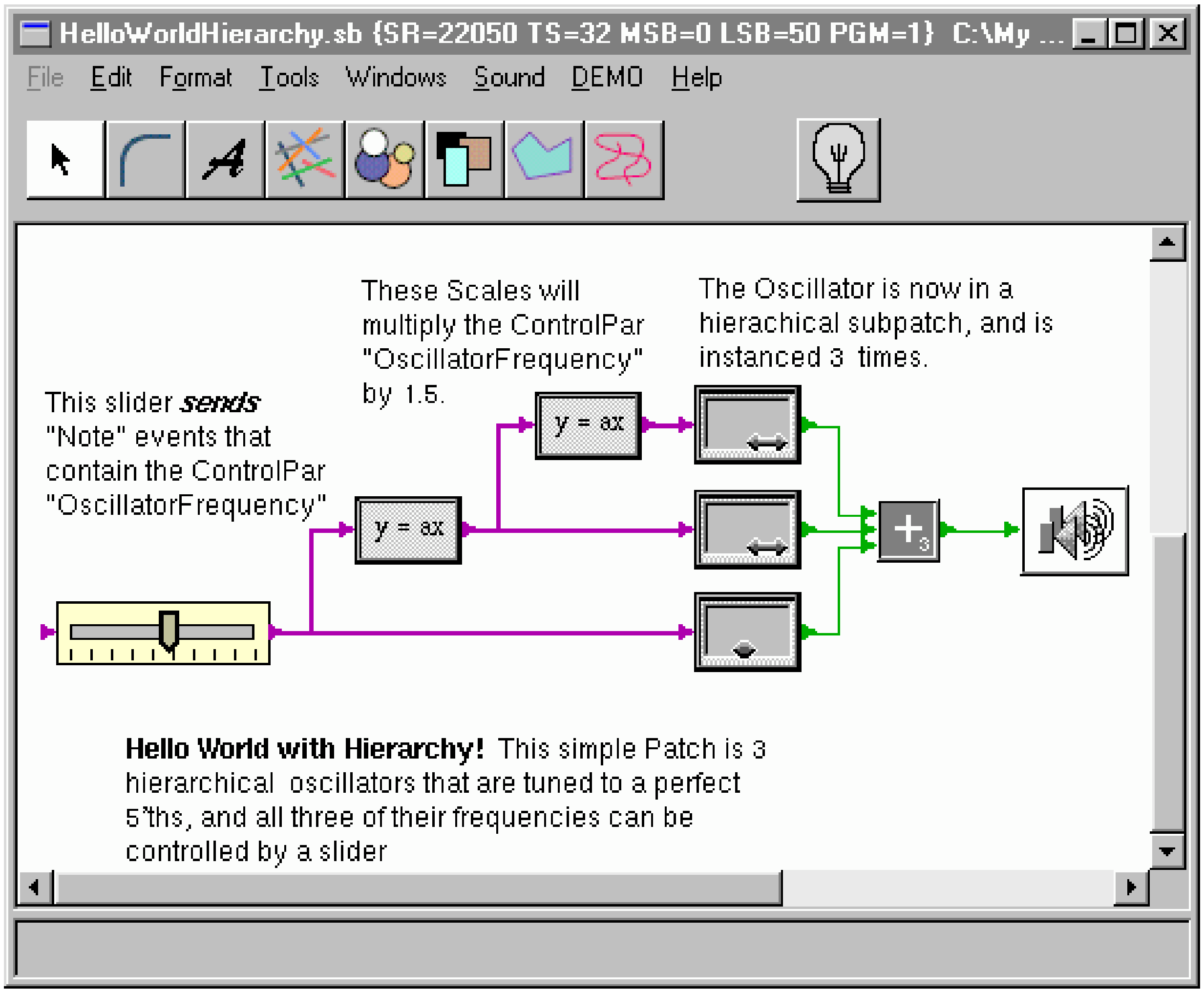

HelloWorld

Parameter Name	Value	Min	Max	Controller
My Freq Event Slider	788	0.00	5000.00	

CHAPTER 2: A Second Example with Hierarchy

2.1 The HelloWorld Example with Hierarchy.

Lets do a second example that demonstrates how hierarchy works. Hierarchy allows "Algorithms" to be constructed from primitives as well as hierarchical elements known as "SubAlgorithms". Here is the original HelloWorld algorithm modified so that there are 3 instances of a SubAlgorithm that contains the oscillator. Note that in SynthBuilder each hierarchical level of the algorithm is a separate document.


HelloWorldHierarchy.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM=1} C:\My ...
File Edit Format Tools Windows Sound DEMO Help
y = ax
y = ax

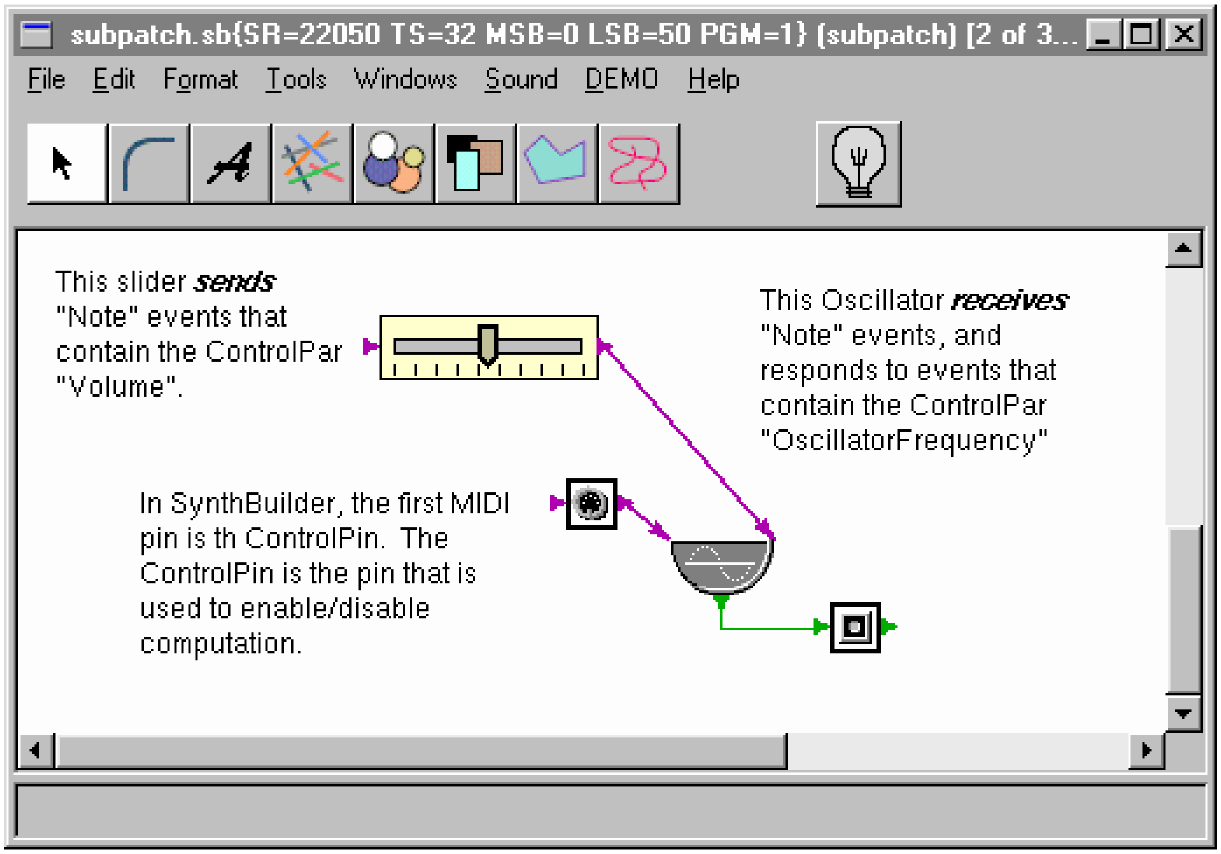
These Scales will multiply the ControlPar "OscillatorFrequency" by 1.5.

The Oscillator is now in a hierachical subpatch, and is instanced 3 times.

This slider ***sends*** "Note" events that contain the ControlPar "OscillatorFrequency"

Hello World with Hierarchy! This simple Patch is 3 hierarchical oscillators that are tuned to a perfect 5'ths, and all three of their frequencies can be controlled by a slider

The top level patch for HelloWorldHierarchy.


subpatch.sb{SR=22050 TS=32 MSB=0 LSB=50 PGM=1} (subpatch) [2 of 3...
File Edit Format Tools Windows Sound DEMO Help

This slider ***sends*** "Note" events that contain the ControlPar "Volume".

This Oscillator ***receives*** "Note" events, and responds to events that contain the ControlPar "OscillatorFrequency"

In SynthBuilder, the first MIDI pin is th ControlPin. The ControlPin is the pin that is used to enable/disable computation.

The oscillator SubAlgorithm for HelloWorldHierarchy.

The hierarchical SubAlgorithm uses special elements called "pins" to establish connectivity between the lower and upper levels of hierarchy. One of the Event pins that appears in a SubAlgorithm must be designated as the ControlPin. This is the pin that will be able to respond to enable/disable ControlPars in events that are sent to the pin.

Hierarchy provides a powerful mechanism to encapsulate sections of an algorithm. Some of the benefits of using hierarchical elements are:

- Hierarchical elements may be instanced multiple times, allowing SynthScript files to be more compact.
- Infinite levels of hierarchy are supported.
- Hierarchical elements may be shared by different algorithms. The definition of SubAlgorithms within a palette may be shared by all patches in the palette. This means that a single SubAlgorithm called "reverb" can be instanced in different patches, but defined in the palette only once. Further, all data that is defined in a hierarchical SubAlgorithm is shared by all instances of the SubAlgorithm.
- Each hierarchical element can have its own sampling rate and tick size.
- SubAlgorithms may be enabled or disabled for computation. This allows portions of an algorithm to be shut down for computational savings. Further the voice allocator has specific support to automatically enable/disable SubAlgorithms. More about the voice allocator later.

2.2 The HelloWorld Example with Hierarchy in SynthScript

```
// This is the SubAlgorithm for the oscillator

// Notice that the SubAlgorithm has an audio Pin and

// a MIDI pin. The MIDI pin " control " has been

// designated as the ControlPin. This is the

// pin that can respond to enable/disable

// controlParameters.

// Also notice that the SubAlgorithm can have its

// own sampling rate and tick size.

SubAlgorithm OscillatorSubAlgorithm (AudioNet oscAudioOut,

ControlPin SubAlgorithmControl,

[SamplingRate=22050,

TickSize=32])

{ // begin OscillatorSubAlgorithm

// ControlPars can be declared at any hierarchical level, and

// can even be duplicated between hierarchal levels.

// This is so that SubAlgorithms are self contained.

ControlPar OscillatorFrequency, Volume;

// Network from the slider

EventNet MyFreqEventSliderOut ();

// Network from the merger

EventNet midiMergerOut ();
```

```
// This merges the hierarchical in, and the
// outputof the slider.
mergerNF aMerger (out= midiMergerOut,
in1=MyFreqEventSliderOut,
in2= SubAlgorithmControl,
[inputCount=2]);
// This is a slider that is nested in the hierarchy
// of the patch. Once the patch is flattened there
// are actually 3 instances of this slider. Their names
// are Oscillator_1/MyVolumeSlider, Oscillator_2/MyVolumeSlider,
// Oscillator_3/MyVolumeSlider
EXPORTED numberVarNF
MyVolumeSlider (out=MyFreqEventSliderOut,
[par=Volume,
                              initialize=1,
                              max=1.0,
                              min=0.0,
                              realValue=0.5]);
oscgUG MyOscillator
(out= oscAudioOut, control= midiMergerOut,
[amp=0.5,
freq=440.0:OscillatorFrequency]
);
} // end of OscillatorSubAlgorithm
// Here is the top level patch.
Algorithm HelloWorldHierarchy([Version=1,
SamplingRate=22050,
TickSize=32,
BankMSB=0,
BankLSB=50,
Program=1])
{
// This is the ControlPar that is for frequency
ControlPar OscillatorFrequency;
// Here are the network declarations. Event that
// hierarchcal networks are declared at the
// highest hierarchical level where they appear.
EventNet MyFreqEventSliderOut2 ();
EventNet scaleParamNF2Out ();
EventNet scaleParamNFOut2 ();
AudioNet add3UGOut ();
AudioNet SubAlgorithmAudioPin ();
AudioNet SubAlgorithmAudioPin2 ();
AudioNet SubAlgorithmAudioPin3 ();
// Here are 3 instances of the Oscillator SubAlgorithm.
// Notice that the pins on the SubAlgorithm instance
// are the names of the pins defined in the SubAlgorithm.
OscillatorSubAlgorithm
Oscillator_1 (oscAudioOut=SubAlgorithmAudioPin2,
SubAlgorithmControl=scaleParamNFOut2,
[enabled=1]);
OscillatorSubAlgorithm
Oscillator_2 (oscAudioOut=SubAlgorithmAudioPin3,
SubAlgorithmControl=MyFreqEventSliderOut2,
[enabled=1]);
OscillatorSubAlgorithm
Oscillator_3 (oscAudioOut=SubAlgorithmAudioPin,
SubAlgorithmControl=scaleParamNF2Out,
[enabled=1]);
// This is the mixer
add3UG
add3UG (out=add3UGOut,
in1=SubAlgorithmAudioPin3,
in2=SubAlgorithmAudioPin2,
in3=SubAlgorithmAudioPin);
out2sumUG
out2sumUG (in=add3UGOut,
[scale=1.0]);
// Continued on the next page . . .
EXPORTED numberVarNF
```

```
MyFreqEventSlider (out=MyFreqEventSliderOut2,
[par=OscillatorFrequency,
initialize=1,
max=500.0,
min=0.0,
realValue=271.60493]);
scaleParamNF
scaleParamNF (out=scaleParamNFOut2,
in=MyFreqEventSliderOut2,
[parIn=OscillatorFrequency,
parOut=OscillatorFrequency,
scale=1.5,
op="mul"]);
scaleParamNF
scaleParamNF2 (out=scaleParamNF2Out,
in=scaleParamNFOut2,
[parIn=OscillatorFrequency,
parOut=OscillatorFrequency,
scale=1.5,
op="mul",
trace=0]);
} // End of patch HelloWorldHierarchy
```

2.3 What happens when a hierarchical algorithm is flattened in SynthCore?

As mentioned previously, when an algorithm is read into SynthCore its not actually computing. The process of reading an algorithm into SynthCore only creates a template assigned to a MIDI program location that can be used to allocate **multiple** copies of the algorithm.

When an algorithm is allocated, it is flattened. This means that beginning from its root, the algorithm is traversed depth first recursively until all instances and SubAlgorithms have been visited. Along the way, UnitGenerators and EventFilters are allocated and connectivity is resolved. Connectivity resolution is accomplished with the assistance of a connection stack in much the same way that programming languages handle argument passing to subroutines.

As mentioned in the previous chapter, instances of objects are allocated in the order that they are defined. This is true for hierarchical instances as well.

Several special points are worth mentioning

- Two mechanisms are provided to switch on and off the computation of hierarchical SubAlgorithms. All hierarchical SubAlgorithms have a StateProc called "enabled" that can be bound to a ControlPar, and can be used to switch on and off the computation of the SubAlgorithm and all of its child SubAlgorithms. Also if one of the MIDI pins leading into a SubAlgorithm is designated as the "ControlPin", and if an event is passed to this pin with a phrase status of phraseEnd, then the SubAlgorithm and all of its child SubAlgorithms will be disabled from computing.

- Instances of objects receive hierarchical names when they are flattened. Hierarchical names are formed as a '/' separated "path" list of instances, much like Unix file system path names. Notice in the previous example, each instance of the slider that is in the oscillator SubAlgorithm has a unique name, *Oscillator_1/MyVolumeSlider, Oscillator_2/MyVolumeSlider and Oscillator_3/MyVolumeSlider* Hierarchical names are important because the API can be used to access a controller by name. The name that is passed to the API may be a hierarchical name. Also StateProcs can be assigned to a symbolic name. This symbolic name may be a hierarchical name. Hierarchical names will be discussed in greater detail in the section about symbolic references
- SubAlgorithms do not have instances of networks that are connected to pins that lead in and out of the SubAlgorithm. This is because the network is allocated at the highest level in the hierarchy of the patch. However the pins are declared as a part of the interface of the SubAlgorithm (see the example).

CHAPTER 3: A Third Example with Data

3.1 The HelloWorld Example with Data.

Lets do a third example that demonstrates the use of data. There are a number of data types in SynthScript. Here is a short summary of the data Types in SynthScript.

Int – A 32 bit integer value (long)

Real – A 64 bit floating point value (double)

Objects – Several objects exist. Internally an object is

IntArray - An array of 32 bit integers (long)

RealArray – An array of 64 bit floating point values (double)

RomUG – An array of A 32 bit floating point values (float)

String - An array of characters. (char)

Event – An array of ControlPars and values. This is an event object.

List – A hierarchical list of objects, specific lists are defined by convention

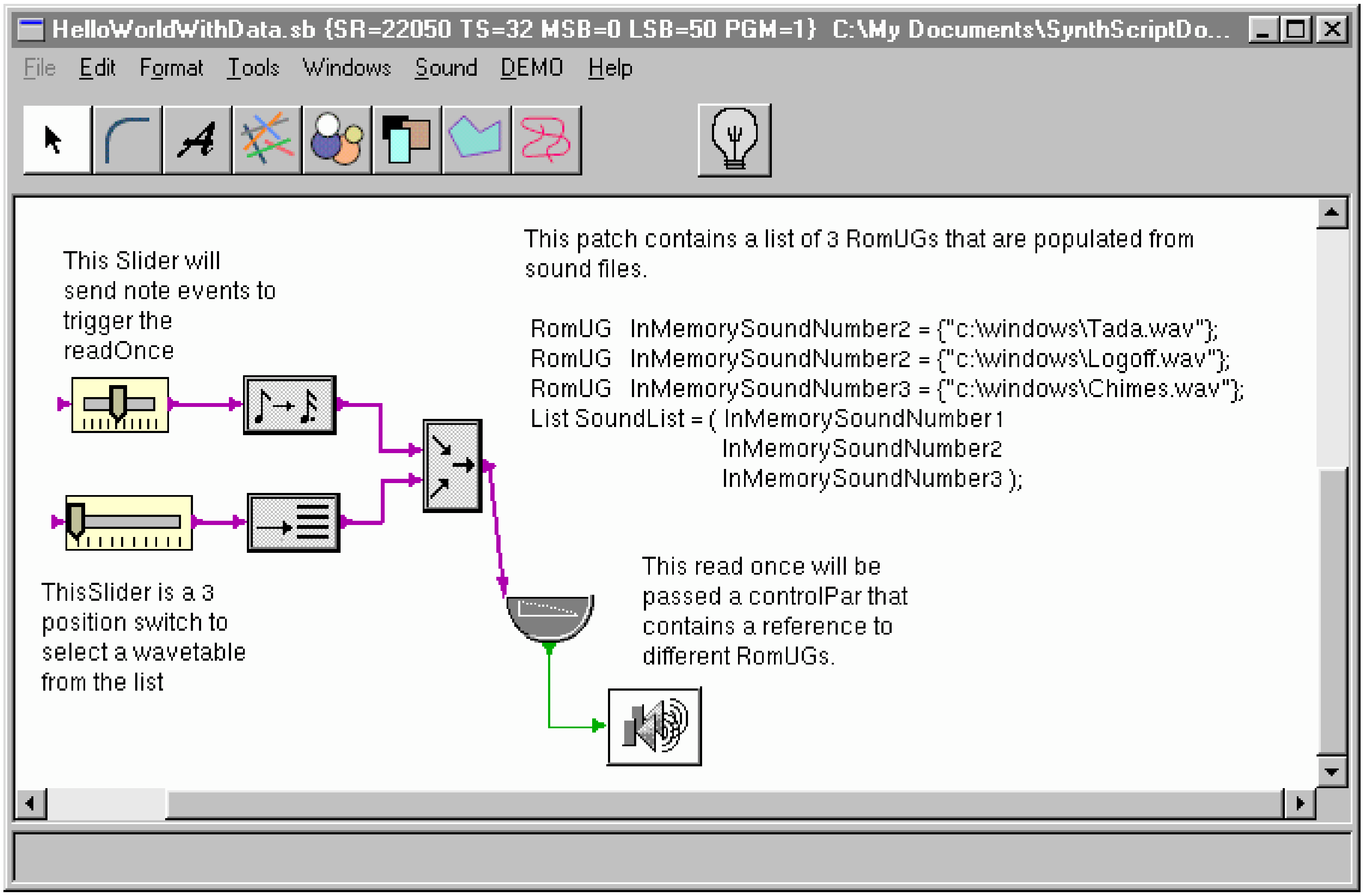
> **EventList** – A list of time stamped note events.
>
> **LookupTable** – A list used for one dimensional lookups.
>
> **Envelope** – A list used to define an envelope for en envelope handler.
>
> **Partials** – A list of harmonic partials.

Instance – Instance is actually an object, and can be passed to StateProcs.

Typically data is instanced in the SynthScript so that it can be passed to StateProcs, either implicitly (proc = value), by symbolic reference, or by reference in an event. The following demonstrates both an implicit assignment of data, and a symbolic assignment.

```
...

RealArray anExampleRealArray = {1.0, 2.0, 3.0};

SomeUG MyUG

(in = net1, out = net2,
// This StateProc is assigned to an implict RealArray

// and is bound to a ControlPar called aRealArrayPar

// that can be used to pass a reference to a RealArray

[aStateProc = {1.0, 2.0, 3.0}:aRealArrayPar,

// This StateProc is assigned to symbolic reference to

//a RealArray, and is bound to a ControlPar called

// anotherRealArrayPar that can be used to pass a reference

// to a RealArray

anotherStateProc = anExampleRealArray:anotherRealArrayPar]);

...
```

A very powerful datatype in SynthScript is List. List can be used to store an arbitrary hierarchical list of objects. The members of a list can be accessed and passed by reference in a ControlPar with the use of the indexNF EventFilter. List can be used to construct arbitrarily complex "data structures".


HelloWorldWithData.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM=1} C:\My Documents\SynthScriptDo...
File Edit Format Tools Windows Sound DEMO Help

This Slider will send note events to trigger the readOnce

ThisSlider is a 3 position switch to select a wavetable from the list

This patch contains a list of 3 RomUGs that are populated from sound files.

```
RomUG  InMemorySoundNumber2 = {"c:\windows\Tada.wav"};
RomUG  InMemorySoundNumber2 = {"c:\windows\Logoff.wav"};
RomUG  InMemorySoundNumber3 = {"c:\windows\Chimes.wav"};
List SoundList = ( InMemorySoundNumber1
                   InMemorySoundNumber2
                   InMemorySoundNumber3 );
```

This read once will be passed a controlPar that contains a reference to different RomUGs.

Here is the HelloWorld algorithm modified so that there are 3 RomUGs that are populated from sound files. These 3 RomUGs are collected together into a list, that can be indexed with the indexNF EventFilter. This algorithm uses one slider with a value 0..2 to select which RomUG to pass to the readOnceUG. It uses another slider to trigger the readOnceUG.

A Read Once that is passed references from a List of RomUGs.

3.2 The HelloWorld Example with Data in SynthScript

```
// gps 4/21/99

// This is an example patch to show

// how data is represented in SynthScript.

// This patch will show how WaveTable Data can

// be stored in a data type called "RomUG"

// which is an in memory array of sound file data.

//

// A list of RomUGs is defined, and then individual

// RomUGs can be selected from the list using the

// listIndexNF.

Algorithm HelloWorldWithData([Version=1,

SamplingRate=22050,

TickSize=32,

BankMSB=0,

BankLSB=1,

Program=1])

{

ControlPar waveTableIndex, // Used to select a particular RomUG

keyNum; // This is a MIDI keyNum, its used to

// trigger the readOnce

// These 3 RomUGs are populated from Mono sound files

// It is important to note that the data that is

// stored in these RomUGs is allocated in the template ONLY

// All instances of this algorithm refer back to the data

// in the template.

RomUG InMemorySoundNumber1 = {"c:\My Documents\Tada.wav"};

RomUG InMemorySoundNumber2 = {"c:\My Documents\Logoff.wav"};

RomUG InMemorySoundNumber3 = {"c:\My Documents\Chimes.wav"};

// Here is a list of RomUGs

List SoundFileList = ( InMemorySoundNumber1

InMemorySoundNumber2

InMemorySoundNumber3 );

// Network Allocations
```

```
EventNet SelectAWavetable2Out ();

EventNet convertEventTypeNFOut ();

EventNet indexNFOut ();

EventNet mergerNFOut ();

EventNet sliderNFOut ();

AudioNet readonceUGOut ();

// This is the streaming read once. Notice

// that it is initially assigned to one of the

// RomUGs, but RomUGs can be passed to

// it in an event parameter.

readonceUG StreamFromMemory

(out=readonceUGOut,

control=mergerNFOut,

[wavetable=InMemorySoundNumber1:waveTableIndex]);

out2sumUG Speaker

(in=readonceUGOut,

[bearing=0.0, scale=1.0]);

// This is used to index the list of RomUGs.

indexNF IndexASoundFromList

(out=indexNFOut,

in=sliderNFOut,

[index=0,

indexPar=waveTableIndex,

outPar=waveTableIndex,

list=SoundFileList,

triggerMode="index"]);

// Notice that the name of the slider is

// actually a string. This allows the UI

// to display this string as the name of the

// controller.

EXPORTED numberVarNF "Select a wavetable 0-2"

(out=sliderNFOut,

[par=waveTableIndex,

initialize=1,

max=2.0,
min=0.0,

intValue=0]);

EXPORTED numberVarNF "Click on this to trigger the sound!"

(out=SelectAWavetable2Out,

[par=keyNum,

initialize=1,

max=65.0,

min=55.0,

intValue=60]);

convertEventTypeNF ConvertToEventOn

(out=convertEventTypeNFOut,

in=SelectAWavetable2Out,

[toType="noteOn", fromType="noteUpdate"]);

mergerNF Merge

(out=mergerNFOut,

in1=convertEventTypeNFOut,

in2=indexNFOut,

[inputCount=2]);

} // End
```

3.3 What happens to data, in SynthCore?

Data is stored in the template, and all instances of an algorithm reference the data. For example, if a template for an algorithm contained 4 meg of wavetable data (RomUG), it would be unreasonable for each instance of the algorithm to have a copy of the wavetable data. Thus each running instance of the algorithm references this data, rather than having copies of the data.

A question comes up about how different hierarchical parts of an algorithm can reference data that is defined in another level of the hierarchy of an algorithm. There are 2 mechanisms that are used to resolve symbolic references to data:

- Data can be referred to symbolically by its fully qualified hierarchical name.
- If a symbol for a data item is not found in the current level of hierarchy, it is resolved by searching for the symbol up through the hierarchy of the patch. This is really just another way of saying that symbolic references are resolved using the same scope rules that C uses for automatic variables in blocks.

For instance if an algorithm consists of the following instance/class hierarchy, and data definitions:

The first thing to note is that the fully qualified hierarchical names for each of the pieces of data are:

```
ThisListIsInTheTopLevel

ThisIsAList

Osc_1/ThisListIsInTheOsc

Osc_1/ThisIsAList

Osc_2/ThisListIsInTheOsc
```

```
        Osc_2/ThisIsAList

        Osc_3/ThisListIsInTheOsc

        Osc_3/ThisIsAList
```

Any StateProc at any level of the hierarchy could be assigned to a the fully qualified hierarchical name. Thus UnitGenerators instanced in the mixer SubAlgorithm can access data that is defined in an instance of the osc using the fully qualified name.

The second thing to note, is that there is a list called "ThisIsAList" in the instances of the osc, as well as in the top level patch. If a StateProc in an instance of the osc is assigned to "ThisIsAList", then it is assigned to the locally scoped data. If a StateProc in the mixer SubAlgorithm is assigned to "ThisIsAList", a locally scoped data cannot be found. The assignment to the data will be resolved by walking up the hierarchy until a data called "ThisIsAList" can be found. In this case, it will be resolved to the data in the top level patch.

Chapter 4 SynthScript Files, High level view

4.1 SynthScript Syntactic Hierarchy

The three previous examples have demonstrated that SynthScript is free format text, and that SynthScript can be used to represent a hierarchical patch as a collection of patch and SubAlgorithm definitions, that contain instances of UnitGenerators, EventFilters and data. Event that SynthScript can be contained in a file, or streamed interactively to SynthCore.

- SynthScript files can actually contain a bit more than a single patch, and its associated SubAlgorithms.
- Multiple patches can be defined in a SynthScript file or streaming session.
- An alternate form of a patch called a Preset can be defined.
- Different patches can share SubAlgorithms, such as a processing unit (reverb, harmonizer)
- Collections of patches that share a common MIDI program space can be collected into a Palette, and multiple palettes can be defined.
- Commands are provided to allocate algorithms
- Update commands are provided to interact with the internals of an algorithm as a part of the voicing process.

The following describes the hierarchy of SynthScript

SynthScript(File or Interactive Streaming)

Palette
- Algorithm
- SubAlgorithm
- Preset
- Data (Not in 1.1)

Algorithm (in the default palette)

SubAlgorithm (in the default palette)

Preset (in the default palette)

Data (Not in 1.1)

Command

Update

4.2 SynthScript Parsing

The parser for SynthScript was implemented with Lex/Yacc. The full grammar is formally defined in Chapter 9. The goal of the parser is to read and validate SynthScript for correct syntax and semantic formation and install parsed data into template tables. Thus the parser would reject partially formed SynthScript instance statements or other types of syntactic errors. An example of a semantic error would be if a non-existent UnitGenerator is instanced.

As was noted with the first example, when SynthScript is read into SynthCore it is validated with the SynthScript Parser. If errors are detected they are indicated with error messages such as this:

```
...

(0000028) oscgUG MyOscillator (0000029) (out=oscgUGOut, ; control=sliderNFOut, ^

parse error at line [0000029] between <,>, and <;>.

(0000030) [amp=0.5, (0000031) freq=440.0:OscillatorFrequency, (0000032) phase=0.0,
(0000033) trace=0](0000034) );

...

SynthCore: The SynthScript patch HelloWorld was read,
```

```
0 warnings, 1 errors.
```

The parser consists of 3 functional sections, Input, Lexical Analysis and Parsing.

The SynthScript Project is really organized into 3 functional modules, parser, template management, and allocated algorithm management (DLA manager)

The processing goes something like this. The parser reads SynthScript, and creates template tables. Allocation commands use the template tables to create running allocated instances of algorithms. The runtime support will be described in greater detail in the next chapter.

4.3 Include files

A simple lexical level include file mechanism is provided in SynthScript. Here is an example of how include files work. First there is a file called osg.dla. Here is the SynthScript that is defined in osg.dla:

```
// include the definition of the speaker SubAlgorithm

#include "speaker.dla"

Algorithm osg(

[BankMSB=55,

BankLSB=55,

Program=56,

SamplingRate=44100])

{

AudioNet oscgUGOut();

oscgUG oscgUG

(out=oscgUGOut,

[amp=0.5,

freq=440]);

speaker speaker(IN=oscgUGOut);
```

```
          }
```

Here is a second file that is found in the same directory as osg.dla, called speaker.dla:

```
          SubAlgorithm speaker(AudioNet IN) {

          out2sumUG out2sumUG (in=IN, [bearing=0, scale=1.0, trace=0]);

          } /* speaker */
```

Event that the include file is found relative to the location of the parent .dla file. This presents a problem for streaming SynthScript, because its not really coming from a file. Streaming SynthScript can use fully qualified hierarchical file names. An example would be:

```
          #include "c:\temp\bla.notelist"
```

Event that this is system dependent. Currently the only system that we support is winX. However its possible in the future that we just do the right thing and consider all the major file ref mechanisms (winX, Mac, Unix, URL) to be interchangeable.

Event that file paths relative to the parent .dla file are supported as well.

```
          #include "..\..\bla.notelist"
```

Event that unlike C, #include may appear anywhere in a line, and can even appear multiple times.

4.4 Encryption

A simple XOR based encryption mechanism is provided. Encryption/Decryption is handled at the input level of the lexical analyzer. This means that by the time that characters have been read into the lexical analyzer, they have already been decrypted. The goal of encryption in R1.1 was prevent the casual user from viewing DLA files. The goal was NOT to proved a bullet proof method of keeping hard core hackers from viewing SynthScript files.

SynthBuilder supports exporting SynthScript files in encrypted format. Also a command line program called xorencrypt is provided to encrypt/decrypt a .dla file. This command line program may be invoked with the following:

```
          xorencrypt password < file1 > file2
```

Event that encryption is a commuting operation. Thus if a file is encrypted once with the password "Foo" it will be in encrypted format. If its encrypted a second time with the password "Foo", it will be in clear text.

4.5 Pretty Printing

A simple pretty printing program is provided to pretty print SynthScript files. This program can be found in …\srctree\tools\win32dlapp. . This command line program may be invoked with the following:

```
          dlapp password < file1 > file2
```

Chapter 5 SynthScript Runtime Support

As has been mentioned in the three tutorial examples, when an algorithm is read into SynthCore its not actually computing. The process of reading an algorithm into SynthCore only creates a template assigned to a MIDI program location that can be used to allocate **multiple** copies of the algorithm. The following diagram shows schematically the architecture of the SynthScript runtime system.

Chapter 6 SynthScript Constant Data Types

This chapter will cover details about the SynthScript constant data types. A formal grammar for these data types is presented in Chapter 9. Note that SynthScript is largely a set of constant data declarations. Because SynthScript is parsed into template tables, and is not fully resolved until an algorithm is allocated, there are no define before use restrictions. However for clarity its always a good thing to define something before its used.

6.1 ControlPar

A ControlPar is used to carry values within events. For instance an event might be passed from a MidiInNF EventFilter that contains the ControlPar keyNum, which is used to pass keyNum to a UnitGenerator. keyNum is an example of a system defined ControlPar.

Users may also create a new ControlPar as needed. For instance an event might be passed to an offsetNF EventFilter. This EventFilter could insert a new ControlPar called "NewKeyNum" with a value of keyNum+7. The ControlPar NewKeyNum needs to be defined in SynthScript so that it can be used by UnitGenerators and EventFilters.

A ControlPar(s) can be defined anywhere in the body of a patch or SubAlgorithm with statements like this:

```
ControlPar waveTableIndex, aSelection;

ControlPar anotherControlPar;
```

Event that each SubAlgorithm may want to define the ControlPar(s) that it uses, rather than depending on the ControlPar(s) being defined in the top level patch. Its ok to have duplicate ControlPar(s) in a patch, or in the hierarchy of a patch. Thus the following is ok:

```
SubAlgorithm speaker(AudioNet IN)

{

ControlPar volume; // volume is defined in both the

// SubAlgorithm and the top level patch

out2sumUG out2sumUG (in=IN, [scale=1.0:volume]);

} /* speaker */

Algorithm osg([BankMSB=55, BankLSB=55, Program=56,

SamplingRate=44100])

{

ControlPar volume; // volume is defined in both the

// SubAlgorithm and the top level patch

AudioNet oscgUGOut();

oscgUG oscgUG (out=oscgUGOut, [amp=0.5:volume,freq=440]);

speaker speaker(IN=oscgUGOut);

} /* osg */
```

6.2 Int

Int is a fundamental data type. In SynthScript Ints are 32 bit longs. Ints may be defined either explicitly (symbolically), implicitly or passed as a value in a ControlPar.

The following snippet of SynthScript demonstrates all three of these.

```
          ControlPar LengthControlPar;

          Int Length = 128;

          ramUG r1 ([size=2048, constant=0.0]);

                                                    // The StateProc delayLength

                                                    // is initially assigned to

                                                    // a symbolic value

                                                    // "Length" that is

          // defined else where.

          delayUG delayUG ([delayMemory=r1,

          delayLength= Length:LengthControlPar]);
```

6.3 Real

Real is a fundamental data type. In SynthScript Reals are 64 bit doubles. Reals may be defined either explicitly (symbolically), implicitly or passed as a value in a ControlPar.

The following snippet of SynthScript demonstrates all three of these.

```
          ControlPar freqControlPar;

          Real amplitudeValue = 0.5;

          // The StateProc amp

                                                    // is initially assigned to

                                                    // a symbolic value

                                                    // "amplitudeValue" that is

                                                    // defined else where.

          oscgUG oscgUG (out=oscgUGOut, [amp= amplitudeValue,

          // The StateProc "freq" is

          // assigned to the implicit

          // Real value 440.0. Notice

          // that its actually set to an

          // Int value, but the DLAmanger

          // will do the type conversion

          // Also "freq" is bound to a

                                                    // ControlPar "freqControlpar"

                                                    // The stateProc "freq" will

                                                    // respond to Events containing

                                                    // this ControlPar

                                                              freq=440:freqControlPar]);
```

6.4 Object

SynthScript supports a number of objects to be described below. Objects may be assigned to StateProcs as Symbolic references. Event that an instance of a UnitGenerator or a EventFilter is an object as well.

6.4.1 IntArray

IntArray is an object that represents an array of Ints. IntArrays may be defined either explicitly (symbolically), implicitly or passed as a value in a ControlPar. None of the current unitGenerators has an IntArray StateProc. Normally IntArrays appear implicitly as a part of a list. Event that the first element of an implicit array, either IntArray or RealArray, is used to disambiguate its type.

The following example is for a contrived UnitGenerator. However it illustrates how an IntArray can be assigned to a StateProc.

```
          ControlPar aControlPar;

          IntArray Arr = {1, 2, 3,4, 5, 6};

          aContrivedUG MyUG ([StateProc1={1,2,3,4},

          StateProc2= Arr: aControlPar]);
```

The elements of an IntArray can be either integers, reals (reals will be cast to int), or a file reference. Thus the following is possible.

```
          IntArray Arr = {1, 2, 3.0,

                                        "c:\aSoundFile.wav",4, 5,

                                        "c:\anotherSoundFile.wav", 6.0};
```

Also because the include mechanism can be used anywhere, its also possible to populate an IntArray from an external data file.

```
          IntArray Arr = {1, 2, 3.0,

                                        "c:\aSoundFile.wav",4, 5,

                              #include "myDataFile1",6.0,

                              #include "myDataFile2", #include "myDataFile2"

                              };
```

6.4.2 RealArray

RealArray is an object that represents an array of Reals, (double). RealArrays may be defined either explicitly (symbolically), implicitly or passed as a value in a ControlPar. Event that the first element of an implicit array, either IntArray or RealArray, is used to disambiguate its type.

The following example iillustrates how an RealArray can be assigned to a StateProc. This twopole filter supports using a RealArray to set its coefficents.

```
          ControlPar CoeffPar;

          RealArray CoeffArray = {0.1, 0.2, 0.3};

          twopoleUG myFilter1 ([rectCoeffs = { 0.9, 0.3, 0.4}:CoeffPar]);

twopoleUG myFilter2 ([rectCoeffs = CoeffArray:CoeffPar]);
```

The elements of an RealArray can be either integers(Ints will be cast to real, reals or a file reference. Thus the following is possible.

```
          RealArray Arr = {1.0, 2, 3.0,
                                        "c:\aSoundFile.wav",4, 5,

                                        "c:\anotherSoundFile.wav", 6.0};
```

Also because the include mechanism can be used anywhere, its also possible to populate an IntArray from an external data file.

```
RealArray Arr = {1.0, 2, 3.0,

                              "c:\aSoundFile.wav",4, 5,

                    #include "myDataFile1",6.0,

                    #include "myDataFile2", #include "myDataFile2"

                    };
```

6.4.3 RomUG (memory Streaming)

RomUG is an object that represents an array of DSPReals, (float). RomUG may be defined either explicitly (symbolically), or passed as a value in a ControlPar. RomUG cannot be defined implicitly, because it cannot be disambiguated from RealArray. A common use for RomUG, is to use it to store sound data that will be streamed from Memory.

The following example iillustrates how an RealArray can be assigned to a StateProc.

```
ControlPar RomPar;

RomUG rom1 = {0.0, 0.0};

RomUG rom2 = {0.0, "c:\awaveFile.wav", 0.0};

readonceUG readonceUG ([wavetable=rom1:RomPar]);
```

The elements of an RomUG can be either integers(Ints will be cast to real, reals or a file reference. Thus the following is possible.

```
RomUG Data = {1.0, 2, 3.0,

                              "c:\aSoundFile.wav",4, 5,

                              "c:\anotherSoundFile.wav", 6.0};
```

Also because the include mechanism can be used anywhere, its also possible to populate an IntArray from an external data file.

```
RomUG Data = {1.0, 2, 3.0,

                              "c:\aSoundFile.wav",4, 5,

                    #include "myDataFile1",6.0,

                    #include "myDataFile2", #include "myDataFile2"

                    };
```

6.4.4 Event

Event is an object that includes a type and an unordered collection of ControlPars and their values. The notation is similar to that of arrays. The first element is an event type. All successive elements are ControlPars assigned to values, and the order is irrelevent.

Events may be stored in Lists, passed to UG or NF state procs, etc. In fact, SynthScript cares nothing about the contents of an event. It is up to the UG or NF that processes the event to interpret the controlPars contained therein.

One of the main consumers of events is the eventlistPlayerNF. This EventFilter takes a list of events and sends them to its output at a time gleaned from the *beat* controlPar in each event. *Beat* is a 0-based time, which is so-named because it is modified by the eventlistPlayerNF's tempo. The times are relative to when the notelistPlayerNF is activated. For example, the same event list could be played by two eventlistPlayerNFs, each activated at a different time. This would produce a musical canon or "round".

Events may be defined either explicitly (symbolically), implicitly or passed as a value in a ControlPar.

The following are examples of Events.

```
Event N1 = {NoteOff, beat=0.4, freq=440.0}; // Event with real ControlPar

Event N2 = {NoteOn, beat =0.4, keyNum=60}; // Event with integer ControlPar

Event N3 = {NoteDur, beat =0.4, dur=2.1,

someIntArray={100, 200, 300}}; // Event with implicit IntArray

// ControlPar

Event N4 = {NoteUpdate, beat =0.4,

someRealArray={100.0, 200.0, 300.0}}; // Event with implicit

// RealArray ControlPar
```

Note that the first element of this array type, is an EventType and , is used to disambiguate it from similar-looking array types. The following are valid EventTypes:

- NoteOn - This is an event that signals the start of a musical note.

Generated by SynthCore in response to incoming MIDI

NoteOn messages.

- NoteOff - This is an event that signals the beginning of the release portion

  of a musical note. Generated by SynthCore in response to

  incoming MIDI NoteOff messages.
- NoteDur -This is an event that represents a combined NoteOn/NoteOff pair.

  The time from the NoteOn to the NoteOff is indicated by the *dur*

  controlPar (which is in seconds, not beats.) Event lists generated

  from incoming MIDI or from Standard MIDI files do not have

  NoteDurs.
- NoteUpdate - This is an update to a sounding musical note. Frequently this is a

  MIDI controller event.
- Mute - This is a general purpose Event event not specifically associated

  with a sounding musical note. It is used to represent things like

  SysEx, or Copyright.

Here is a typical event list that might come from recording incoming MIDI:

```
List el = ( {NoteOn, beat=0.4, keyNum=60, normalizedVelocity=0.7},
{NoteUpdate, beat =0.5, pitchBend=7123},

{NoteOff, beat =0.8, keyNum=60})
```

Each of the successive elements in the Event, are ControlPars, and they are assigned to a value that can be any valid SynthScript constant data type. All of the ControlPars that are predefined are listed in sections 9.3 and 9.4.

6.4.5 List

List is a powerful data type in SynthScript. We borrow the Lisp notation for list. In SynthScript, lists can contain any object, or a symbolic referance to an object. List is used to collect together the basic data types into something that can passed around as a single object.

Here is a simple example of a list:

```
          IntArray A = {1,2,3,4,5};

          IntArray B = {4,5,6,7,8};

          List Z = (A B);
```

The following shows some more complex examples. Note that its possible to create implicit and explict objects directly in the body of the list. Also the identifiers in a list are not scoped by the depth in the list. Thus in the case below, IntArray F can be referenced by anything in the patch or SubAlgorithm that it is in.

```
          IntArray A = {1,2,3,4,5};

          IntArray B = {4,5,6,7,8};

          RealArray C = {1.0, 2.0, 3.0, 4.0};

          RealArray D = {5.0, 6.0, 7.0, 8.0};

          List Z = (A B (C D) A );

          List E = ( A B (C D) D (A B C)

          C A Z

          {1.0, 2.0, 3.0}

          {1,2,3,4,5}

          {IntArray F = {1,2,3,4,5}}

          {RealArray G = {1.0,2.0,3.0,4.0,5.0}}

          F G

          {RomUG H = {1.0, 2.0, 3.0}}

          H H

          );
```

An important point about lists, is that they are fully hierarchical, and the special EventFilter indexNF can be used to index into a list. The indexNF can also be cascaded to index deeply into hierarchical lists. Possiblities are lists of wavetables that can be selected, lists of lists of notes (lists of scores), that can be selected.

There are a number of special cases of List, that specific StateProcs respond to. They are defined by convention. These specific lists are:

- Event List - a score of events.

- Lookup Table - a single argument lookup
- Envelope - an Envelope for an Envelope handler
- Partials - a List of partials for partial based synthesis.

There will probably be more conventions in the future.

6.4.5.1 EventList

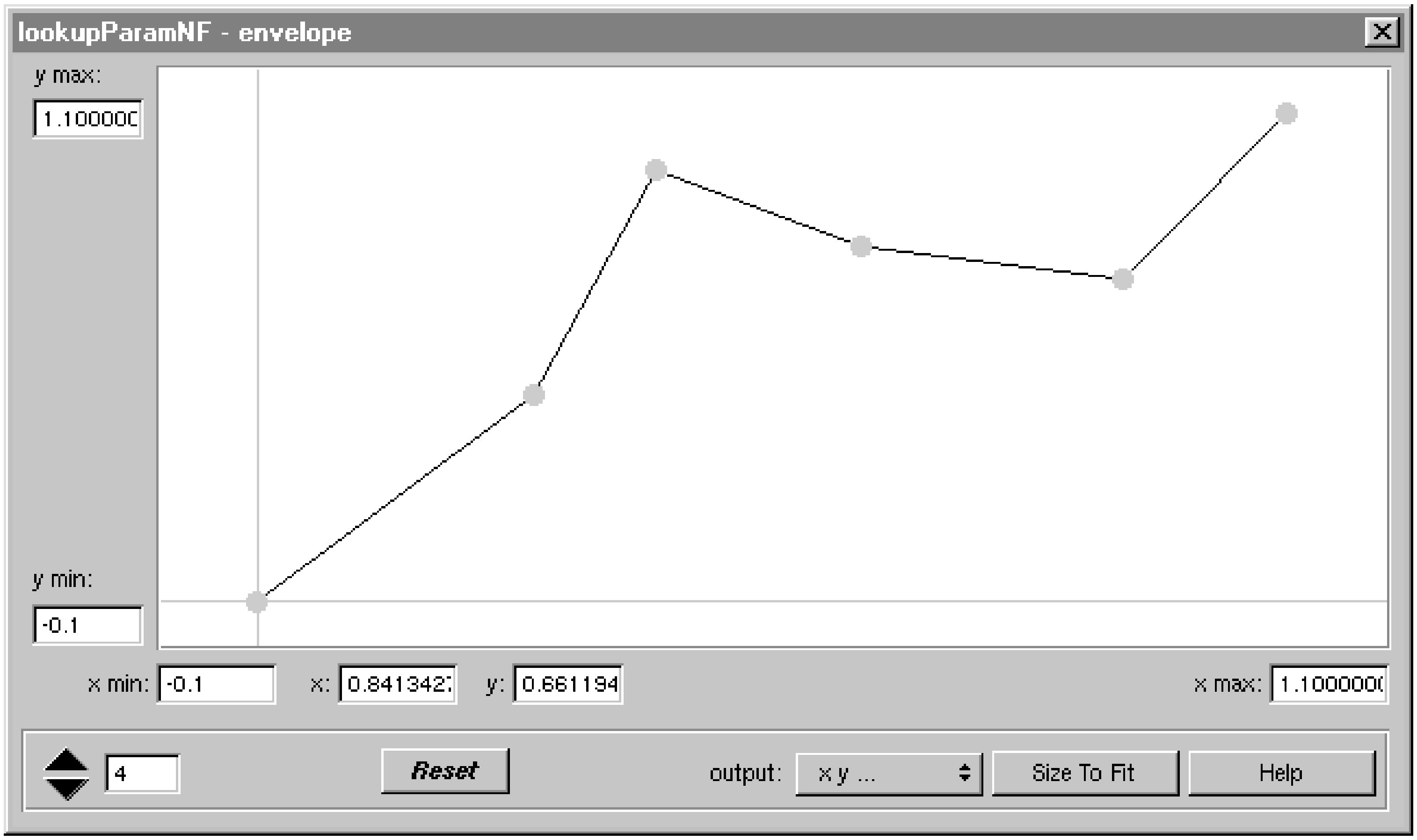
Event list is a used to collect together note events into a list that can be scheduled for execution.

The special EventFilter notelistPlayerNF, is used to schedule these events. The special EventFilter notelistRecorderNF, is used to record these events. The following is an example of this of two noteLists that are collected into another list, a list of Scores. The Index EventFilter can be used to select these scores for performance.

```
ControlPar EventList, ScoreList;

          List MiniScore1 = ( {EventOn, beat=0.4, freq=440.0}

          {EventOn, beat=0.5, freq=441.0}

          {EventOn, beat=0.6, freq=442.0}

          {EventOn, beat=0.7, freq=443.0}

          {EventOn, beat=0.8, freq=444.0}

          {EventOn, beat=0.9, freq=445.0}

          );

          List MiniScore2 = ( {EventOn, beat=0.4, freq=220.0}

          {EventOn, beat=0.5, freq=221.0}

          {EventOn, beat=0.6, freq=222.0}

          {EventOn, beat=0.7, freq=223.0}

          {EventOn, beat=0.8, freq=224.0}

          {EventOn, beat=0.9, freq=225.0}

          );

          // Use Index EventFilter to select between these

          // 2 mini-scores

List TwoScores = ( MiniScore1 MiniScore2 );

          EventNet indexNFOut ();

EventNet sliderNFOut ();

EXPORTED numberVarNF sliderNF

(out=sliderNFOut,

[par=EventList,initialize=1, max=1.0, min=0.0,

          intValue=0]);

indexNF indexNF

          (out=indexNFOut, in=sliderNFOut,

[index=0, indexPar=ScoreList,
                                                   outPar=EventList, list= TwoScores,
                                                   triggerMode="index"]);

notelistPlayerNF notelistPlayerNF

          (in=indexNFOut,

[notelist=MiniScore1,

parEventlist=EventList, tempo=60.0]);
```

6.4.5.2 LookupTable

Lookup Tables are useful for precalculating lookups so that they don’t have to be calculated on the fly. The lookupParamNF uses lookup tables which are represented as a list of two RealArrays. Here is an example of a lookup table.



y max: 1.100000
y min: -0.1
x min: -0.1 x: 0.841342 y: 0.661194 x max: 1.100000
4

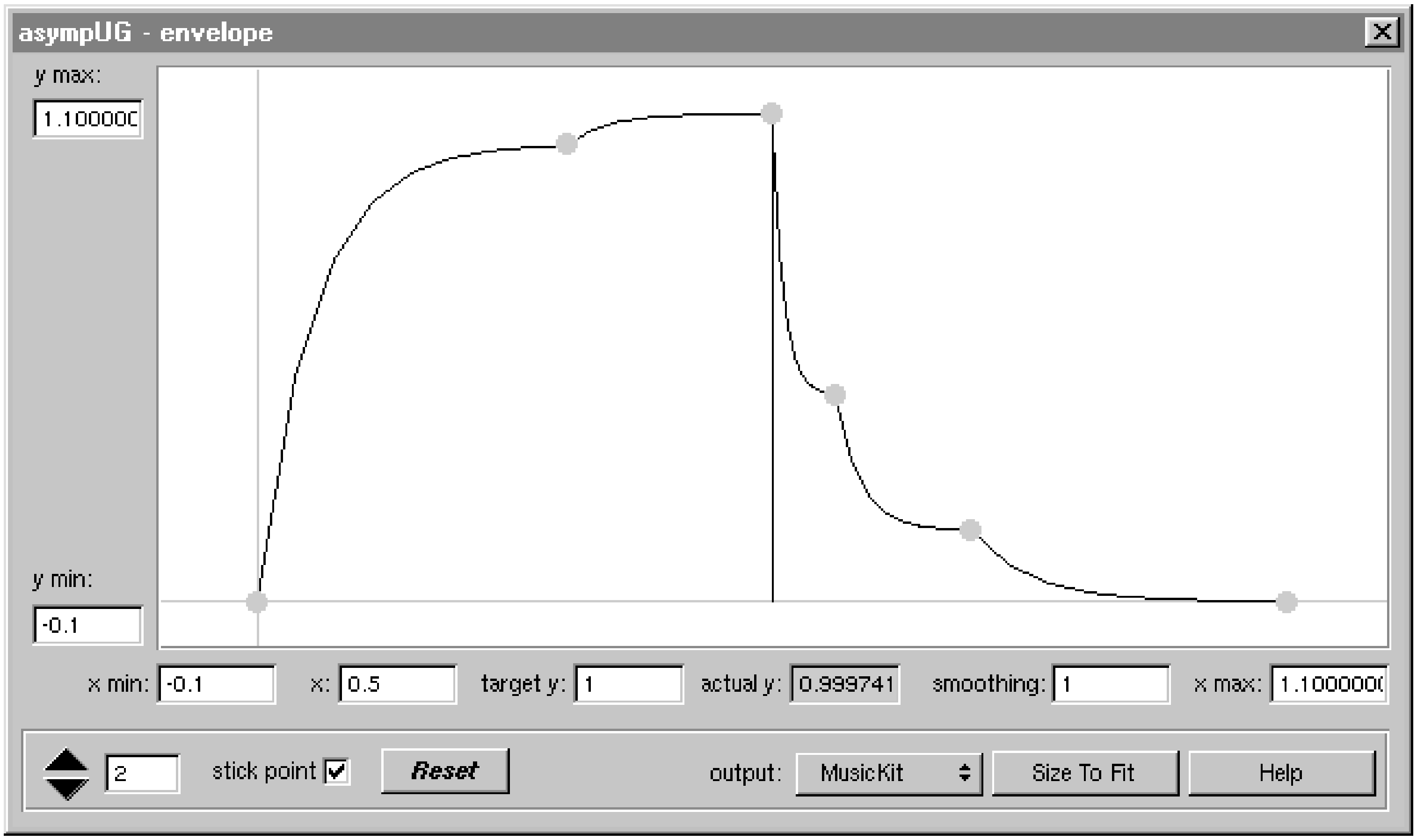
Here is how this lookupParamNF is instanced using an implict list of two RealArrays.

```
          lookupParamNF lookupParamNF ([lookup=

          ( // List of 2 RealArrays

          // X coordinates

          {0.0,0.15441,0.222261, 0.3558304,0.3600,

          0.44063,0.51272,0.673851,0.69293284,0.7904593,

          0.90494,0.9240282,1.0}

          // Y coordinates

                    {0.0,0.64776,0.383582,0.5716418,0.1149253,0.52238,

                    0.334328,0.688059,0.46417,0.8985,
                    0.3074626,0.670149,1.0}

          )

]);
```

The list could have also been declared explicitly and either assigned or passed in a ControlPar.

6.4.5.3 Envelope

Envelopes are useful for controlling UnitGenerators. They can be used to control things such as frequency or amplitie
The asympUG uses lookup tables which are represented as a list of 4 RealArrays. Here is an example of a lookup table.



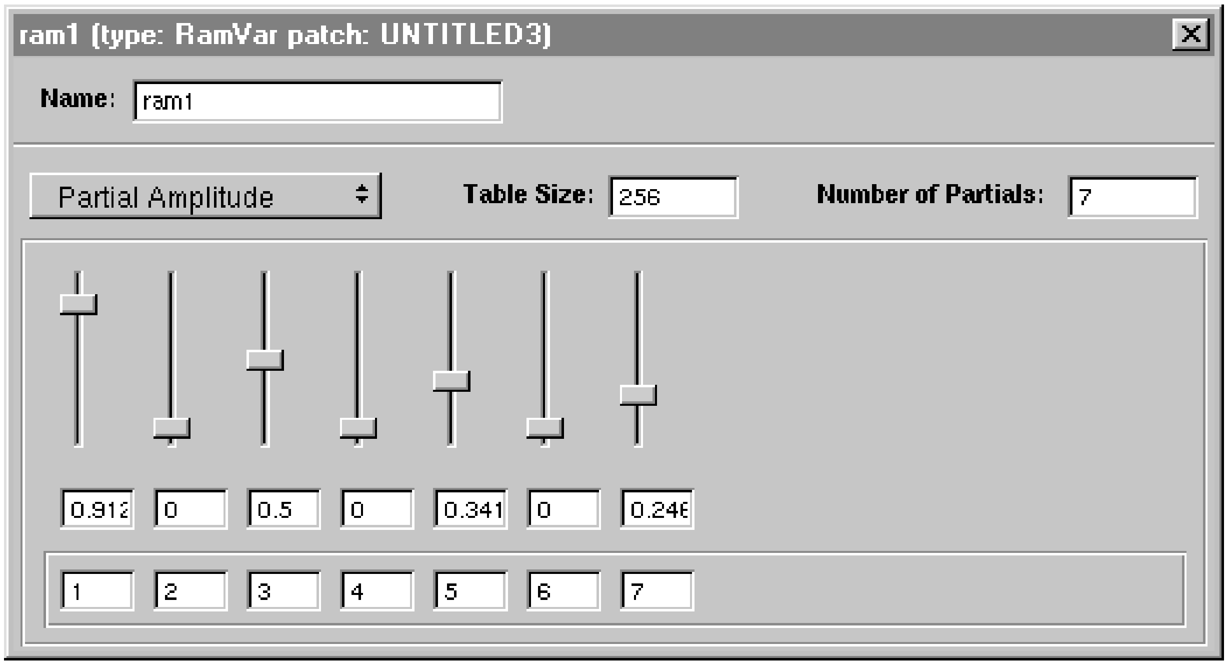
y max: 1.100000
y min: -0.1
x min: -0.1
x: 0.5
target y: 1
actual y: 0.999741
smoothing: 1
x max: 1.100000
2
stick point ✓

```
asympUG asympUG

([yScale=1.0, yOffset=0.0, timeScale=1.0,

releaseTimeScale=1.0, articulationMode=1,

envelope=(

// X Coords

{0.0,0.30070,0.5,0.5614,0.6929,1.0}

          // X Coords

{0.0,0.9388,1.0,0.42388,0.146268,0.0}

// Smooting factor

{1.0,1.0,1.0,1.0,1.0,1.0}

// Which point is the stick point.
{2} ),
stopVoiceAtEndOfEnvelope=0, trace=0]);
```

The list could have also been declared explicitly and either assigned or passed in a ControlPar.

6.4.5.4 Partials

Partials useful for describing waveform shapes The ramUG (memory) can be populated from a partials list. This list is represented as a list of 3 RealArrays. Here is an example of a ramUG populated from a partials list.



Name: ram1

Table Size: 256

Number of Partials: 7

0.912	0	0.5	0	0.341	0	0.246
1	2	3	4	5	6	7

```
          ramUG ram1 ([size=256, fromPartials=(
          {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0}
          {0.91269, 0.0, 0.5, 0.0, 0.341269, 0.0, 0.246031}
          {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0})
]);
          oscgafiUG oscgafiUG ([incScaler=1.0, wavetable=ram1]);
```

The list could have also been declared explicitly and either assigned or passed to the ramUG in a ControlPar.

6.5 Instance

As has been illustrated in the examples so far, instance is the data type that is used to create instances of objects running in SynthCore. In its simplest form, instance consists of something like this:

```
          // ClassName InstanceName
          oscgUG MyOscillator
          // Connectivity
          (out= oscAudioOut, control= midiMergerOut,
          // State values and controlPar bindings.
          [amp=0.5,
          freq=440.0:OscillatorFrequency]
          );
```

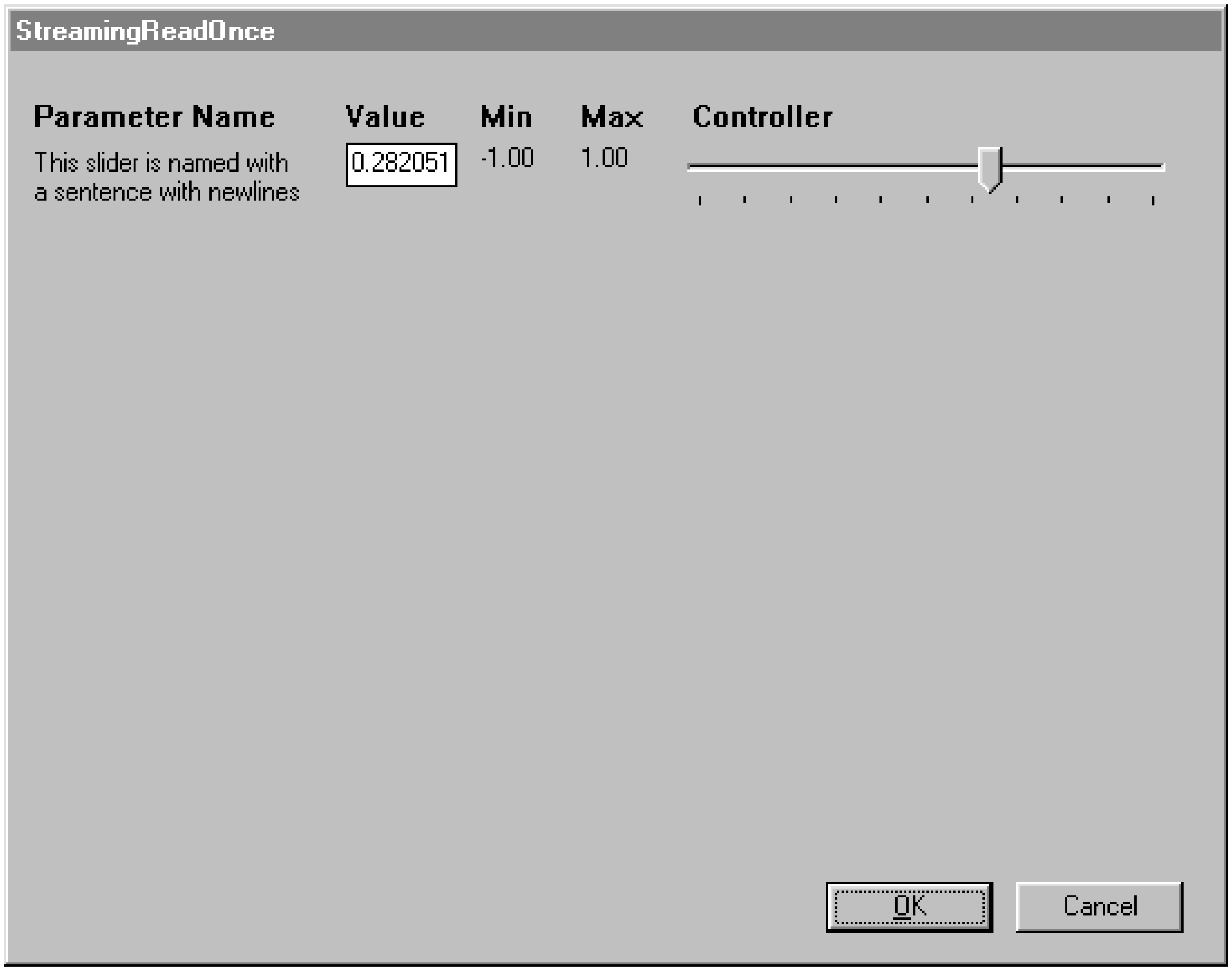
There are some special cases that need to be considered.

6.5.1 InstanceNames

In general instanceNames are just simple identifiers. However, its also possible for an instance name to be a String. This allows for descriptive names for controllers to be exported to the public API. For instance an exported slider might look like this:

```
          EXPORTED numberVarNF "This slider is named with
          a sentence with newlines"
          (out=sliderNFOut, [par=foo,
          initialize=1,
                                        max=1.0,
                                                  min=-1.0,
                                                  realValue=0.282051,
                                                  trace=0]
          );
```

The public API returns the name of the slider as a string with a newline. It can be displayed like this:



Parameter Name	Value	Min	Max	Controller
This slider is named with a sentence with newlines	0.282051	-1.00	1.00	

6.5.2 Connectivity

Connectivity is specified by assignment of a pin to a network. An example would be:

```
          …
          (
          pin1 = networkName,
          pin2 = anotherNetworkName, …
          );
```

6.5.3 StateProcs

StateProcs are initally assigned to a SynthScript constant data type, and can be bound to a ControlPar. When ever an event is passed to the particular UnitGenerator, if it contains that EventPar, then the stateProc that is bound to it will be called. An example of a StateProc assignment, and binding would be:

```
          …
          [ proc1= 1.0:aControlPar1,
          proc2 = aSymbolicRefernce:aControlPar2 …
          ]
          …
```

6.5.3.1 Enable

Enable is a StateProc that is owned by all instances of SubAlgorithms. This StateProc can be used to enable and disable compuation of the SubAlgorithm.

6.5.3.2 Trace

Trace is a StateProc that is owned by all instances. This StateProc can be used to enable and disable verbosity levels for tracing. It has bit fields, to enable different types of tracing. Common values are 0, no Tracing, 63 verbose tracing.

6.5.3.3 Monitor

Monitor is a StateProc that is owned by all instances. This StateProc can be used to enable and disable monitoring of computation for that UG. Monitoring is currently written to the log file.

6.5.3.4 ClearOutputs

ClearOutputs is a StateProc that is owned by all instances. This StateProc can be used to clear all of the output patchpoints (audioNets) of the UG.

6.5.4 Tags - representing extended data

Tags are similar to state. They are bracketed by ‘<’ ‘>’ and contain symbolic names assigned to SynthScript constant data types. They are not used for Synthesis, they are user defined data that can be queried from the public API, and can be used to represent things like graphical coordinates, or other user defined data.

Here is an example:

```
Algorithm example ([BankMSB=55,
          BankLSB=55,
          Program=56,
          SamplingRate=44100]
```

```
        // here are some tags
                            <xbounds = 100.00,
                            ybounds = 100.00,
                            color = "Red"> )
        { // begin example
        // . . . other stuff
        oscgUG MyOscillator
        (out= oscAudioOut <pinSequence = 1>, // Note the pin tag
        control= midiMergerOut <pinSequence = 2>,
        // State values and controlPar bindings.
        [amp=0.5,
        freq=440.0:OscillatorFrequency],
        // This is a "vector" of instance tags.
        <xcoord = 100.0,
        ycoord = 100.0,
        icon="aFileReference">
        );
        //. . . other stuff
        } // end of example
```

Tags can appear in 3 possible contexts.

- In the definition of Algorithm, SubAlgorithm, or Preset.
- In the body of an instance
- Associated with a Pin.

Tags are provided so that extra data can be represented that does not apply to synthesis. A possible application for tags are to read SynthScript back into SynthBuilder. The tags can be used to store the coordinates, sizes, etc of all of the objects needed to create a SynthBuilder document from the SynthScript. Another possible application for tags would be to use SynthBuilder to define the GUI for a patch, and pass to a GUI application in the SynthScript.

6.5.5 SampleRates

Each instance of a UnitGenerator can have its own SamplingRate. Sampling Rate is an optional real number that cann appear after the instanceName. Here is an example:

```
oscgUG oscgUG 44100.0 ([amp=0.5, freq=440.0]);
```

Valid values are 88200.0, 44100.0, 22050.0 11025.0, 5512.5, 2756.25, 1378.125,

689. 0625

6.6 Symbolic Reference

StateProcs can be assigned to symbolic references. These will be resolved the the algorithm is allocated. They may be hierarchical references.

6.6.1 Hierarchical references

Names by be hierarchical references. This is described with an example in Section 3.3

Any StateProc at any level of the hierarchy could be assigned to a the fully qualified hierarchical name. Thus UnitGenerators instanced in the mixer SubAlgorithm can access data that is defined in an instance of the osc using the fully qualified name.

If a symboic reference is not fully qualified then the symbolic reference will be resolved by walking up the hierarchy until a reference can be found.

6.7 Algorithm & SubAlgorithm

Algorithm is the root of an algorithm. Algorithm can have a number of parameters that describe it, program location, SamplingRate, TickSize, Version Number. An Algorithm declaration looks something like this:

```
          Algorithm osg(

          [BankMSB=55,

          BankLSB=55,

          Program=56,

SamplingRate=44100])

          {

          // body of the algorithm

          }
```

Possible parameters that can appear here are:

- BankMSB – An Integer (assumed to be 0 if omitted)
- BankLSB – An Integer (assumed to be 0 if omitted)
- Program – An Integer
- SamplingRate – A valid SamplingRate
- TickSize - A valid TickSize
- Version - SynthScript Version. Currently only the integer 1 is supported.

The only required parameter is Program.

SubAlgorithm is a hierarchical element. SubAlgorithm can have a number of parameters associated with it. In particular, its connectivity interface must be defined.

```
          SubAlgorithm OscillatorSubAlgorithm (AudioNet oscAudioOut,

          ControlPin SubAlgorithmControl,

          [SamplingRate=22050,

          TickSize=32])

          {

          // body of the sub patch

          }
```

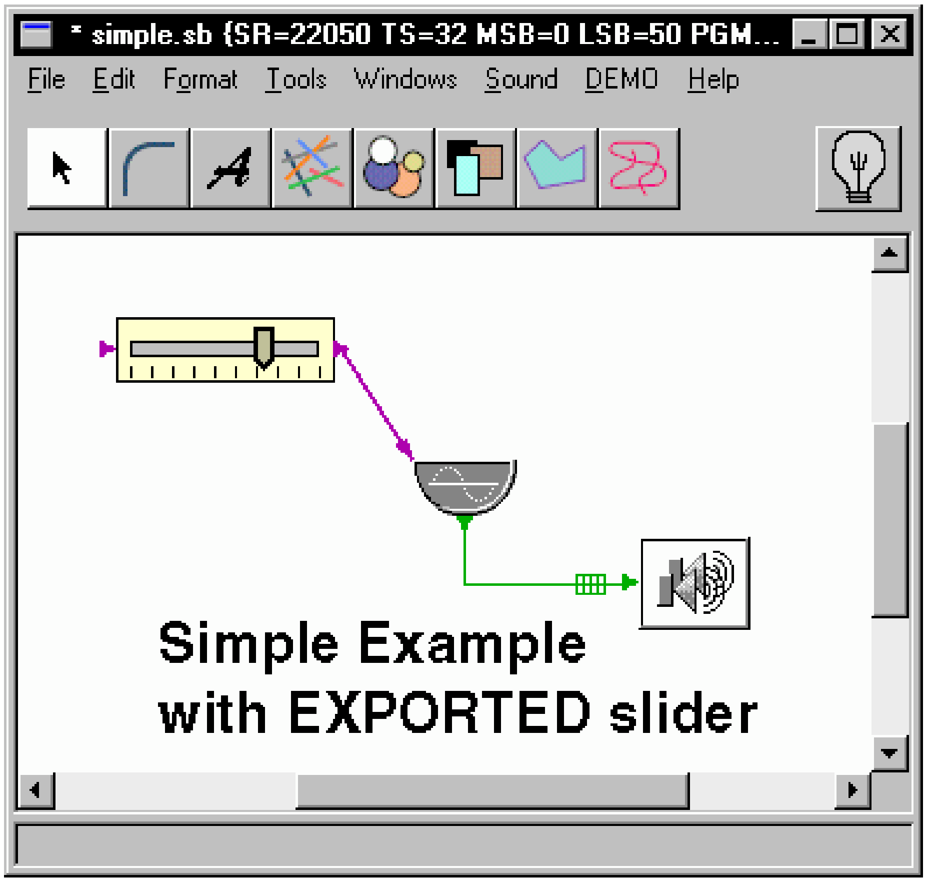
The connectivity interface is defined as pins. Event that one of the MidiPins can be designated as the ControlPin. This pin will be the pin that can respond to phrase status messages that can enable and disable the patch. An imporant point to note, is that networks are not declared for pins. Bank and program locations are not valid for SubAlgorithms. However SubAlgorithms can have SamplingRates and TickSizes and Versions.

6.8 Preset

Slider control objects (numVarNF), can be tagged with the keyword directive "EXPORTED". All of the numVarNFs that are tagged as EXPORTED can be introspected by SynthCore's public API. We refer to these EXPORTED numVarNFs as the "Exported Control Surface" for the patch.

The following is a simple example that shows how this works in both SynthBuilder, and in SynthScript.

```
Algorithm simple([Version=1,

SamplingRate=22050.000000,

TickSize=32,

BankMSB=0,
```


* simple.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM...
File Edit Format Tools Windows Sound DEMO Help
Simple Example
with EXPORTED slider

```
BankLSB=50,

Program=1])

{

ControlPar freq;

EventNet sliderNFOut (

);

AudioNet oscgUGOut (

);

out2sumUG out2sumUG (

in=oscgUGOut,
[bearing=0.0,

scale=1.0,

trace=0]

);

oscgUG oscgUG (

out=oscgUGOut,

control=sliderNFOut,

[amp=0.5,

freq=440.0:freq,

phase=0.0,

trace=0]

);

EXPORTED numberVarNF FreqControl (

out=sliderNFOut,

[par=freq,

initialize=1,

max=1000.0,

min=1.0,

realValue=716.333333333333,

trace=0]

);

} /* simple() */
```

Presets are supported with a variation on Algorithm(). "Preset()" is at the same level in the SynthScript grammar as Algorithm() and SubAlgorithm().

- A Preset() references by name, the Algorithm() that it can be applied to.
- Presets like Algorithms are located in MidiProgram space by a unique ProgramNumber (BankMSB/BankLSB/Program).
- A preset may be allocated by either the public API or by a MIDI program change in the same way that a Algorithm can be allocated. If the Algorithm() that the preset applies to is not yet allocated, it will be allocated.
- The body of preset will consists of a list of values that can be applied to the Exported Control Surface of the referenced patch.

The following is an example of a Preset():

```
        Preset simplePreset1([Version=1,

        Algorithm=Simple,

        BankMSB=0,

        BankLSB=50,

        Program=2])
```

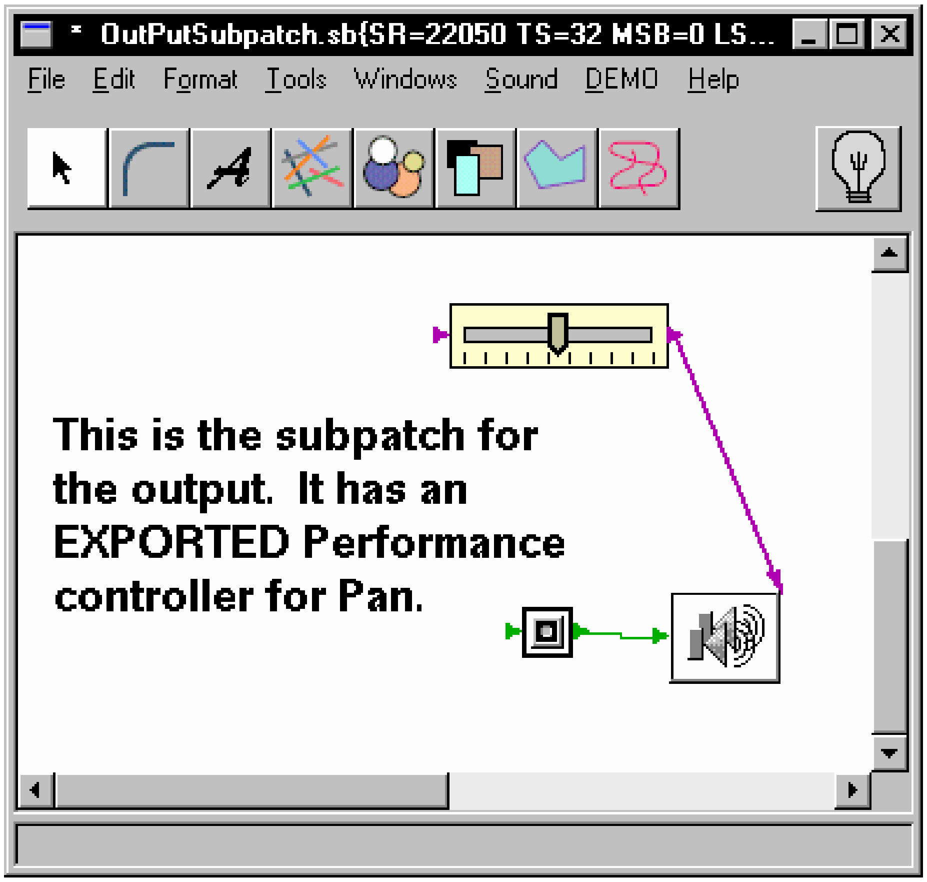
```
          {
          // This preset sets the realValue
          // of numVar instance FreqControl to 120.0
          FreqControl([realValue=123.0]);
          }
```

In order to support a differentiation between "Performance" controllers and "Tuning" controllers, a tag can be added to indicate if an exported controller is for "Performance" or for "Tuning". The following is an example of each of these types of

EXPORTED controllers.

```
          EXPORTED numberVarNF PanControl <ControllerType = "Performance"> (
          out=sliderNFOut,
          [par=pan,
          initialize=1,
          max=45.0,
          min=-45.0,
          realValue=0.282051,
          trace=0]
);
          EXPORTED numberVarNF PanControl <ControllerType = "Tuning"> (
          out=sliderNFOut,
          [par=pan,
          initialize=1,
          max=45.0,
          min=-45.0,
          realValue=0.282051,
          trace=0]
)
```

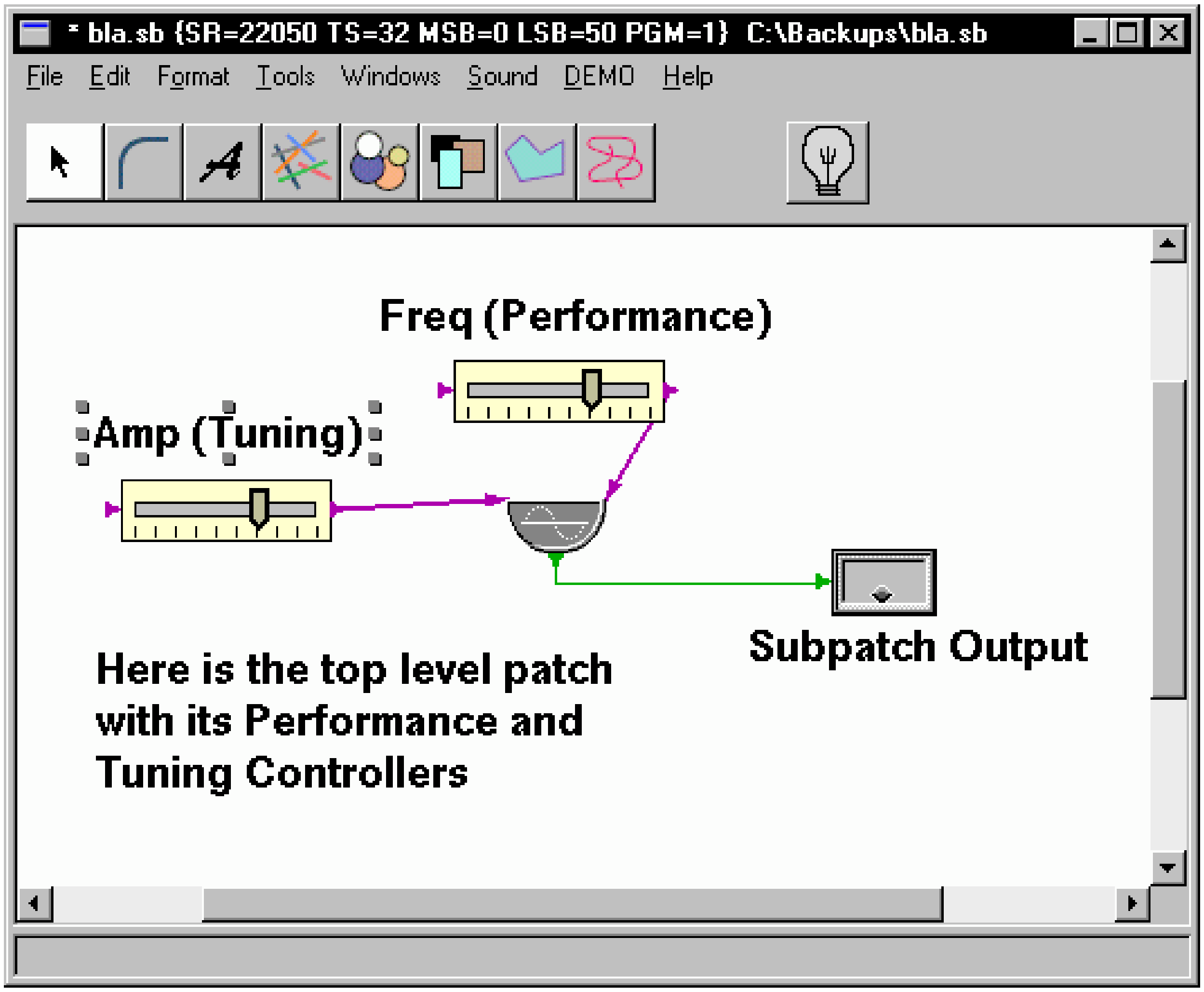
The following example shows a hierarchical patch, and its presets.



This is the subpatch for the output. It has an EXPORTED Performance controller for Pan.

```
// SubAlgorithm outSubAlgorithm.
          SubAlgorithm OutSubAlgorithm (AudioNet audioPin2,
          [Version=1,
          SamplingRate=22050.000000,
          TickSize=32])
          {
          ControlPar pan;
          EventNet sliderNFOut (
          );
          out2sumUG anOutput (
                    in=audioPin2,
          control=sliderNFOut,
          [bearing=0.0:pan,
          scale=1.0,
          trace=0]
          );
          // exported Performance controller
          EXPORTED numberVarNF PanControl
          <ControllerType = "Performance">
```

```
        (out=sliderNFOut,
        [par=pan,
        initialize=1,
        max=45.0,
        min=-45.0,
        realValue=0.282051,
        trace=0]
        );
        } /* OutSubAlgorithm() */
```


* bla.sb {SR=22050 TS=32 MSB=0 LSB=50 PGM=1} C:\Backups\bla.sb
File Edit Format Tools Windows Sound DEMO Help
Freq (Performance)
Amp (Tuning)
Subpatch Output

Here is the top level patch with its Performance and Tuning Controllers

```
// Top Level Algorithm
Algorithm SimpleOsc([Version=1,
SamplingRate=22050.000000,
TickSize=32,
BankMSB=0,
BankLSB=50,
Program=1])
```

```
{

ControlPar freq;

EventNet sliderNFOut (

);

EventNet sliderNFOut1 (

);

AudioNet oscgUGOut (

);

oscgUG anOsc (out=oscgUGOut,

control=sliderNFOut,

[amp=0.5,

freq=440.0:freq,

phase=0.0,

trace=0]

);

// This is the instance of the hierarchical SubAlgorithm

OutSubAlgorithm OutSubAlgorithm1 (

audioPin2=oscgUGOut,

[enabled=1,

trace=0]

);

EXPORTED numberVarNF FreqControl <ControllerType = "Performance"> (

out=sliderNFOut,

[par=freq,

initialize=1,

max=1000.0,

min=1.0,

realValue=691.666666666667,

trace=0]

);

EXPORTED numberVarNF AmpControl <ControllerType = "Tuning"> (

out=sliderNFOut1,

[par=amp,

initialize=1,

max=1.0,
min=0.0,

realValue=1.0,

trace=0]

);

} // SimpleOsc()

          // Preset 1

          Preset SimpleOscPreset1([Version=1,

          Algorithm=SimpleOsc,

          BankMSB=0,

          BankLSB=50,

          Program=2])

          {

          FreqControl([realValue=123.0]);

          AmpControl([realValue=0.9]);

          // Event the hierarchical notation.

          OutSubAlgorithm1/PanControl([realValue=-37.9.0]);

          }


          // Preset 2

          Preset SimpleOscPreset1([Version=1,

          Algorithm=SimpleOsc,

          BankMSB=0,

          BankLSB=50,

          Program=3])

          {

          FreqControl([realValue=456.0]);

          AmpControl([realValue=0.99]);

          // Event the hierarchical notation.

          OutSubAlgorithm1/PanControl([realValue=34.0]);

          }


          // Preset 3

          Preset SimpleOscPreset1([Version=1,

          Algorithm=SimpleOsc,

          BankMSB=0,
          BankLSB=50,
          Program=4])
          {
          FreqControl([realValue=440.0]);
          AmpControl([realValue=0.8]);
          // Event the hierarchical notation.
          OutSubAlgorithm1/PanControl([realValue=-17.9.0]);
          }
```

6.9 Palette

Palette is a level of hierarchy around a collection of patches and SubAlgorithms and presets. All patches and SubAlgorithms and presets within a palette share the same MIDI program space. Also SubAlgorithms within a palette can be shared by all the patches in the palette. Any patches SubAlgorithms and presets that are defined outside the scope of a palette are entered into the default palette. The following example is a palette with patches on different program locations and a shared SubAlgorithm.

```
Palette ExamplePalette ( ) {
// This is a shared SubAlgorithm
          SubAlgorithm OscillatorSubAlgorithm (AudioNet oscAudioOut,
          ControlPin SubAlgorithmControl,
          [SamplingRate=22050,
          TickSize=32])
          { // begin OscillatorSubAlgorithm
          ControlPar OscillatorFrequency, Volume;
          EventNet MyFreqEventSliderOut ();
          EventNet midiMergerOut ();
          mergerNF aMerger (out= midiMergerOut,
          in1=MyFreqEventSliderOut,
          in2= SubAlgorithmControl,
          [inputCount=2]);
          EXPORTED numberVarNF
          MyVolumeSlider (out=MyFreqEventSliderOut,
          [par=Volume,
          initialize=1,
          max=1.0,
          min=0.0,
          realValue=0.5]);
          oscgUG MyOscillator
          (out= oscAudioOut, control= midiMergerOut,
          [amp=0.5,
          freq=440.0:OscillatorFrequency]
          );
          } // end of OscillatorSubAlgorithm
          // Here is the top level patch number 1
Algorithm HelloWorldHierarchyNumber1([Version=1,
SamplingRate=22050,
TickSize=32,
BankMSB=0,
BankLSB=50,
Program=1])
{
ControlPar OscillatorFrequency;
EventNet MyFreqEventSliderOut2 ();
EventNet scaleParamNFOut2 ();
AudioNet add2UGOut ();
AudioNet SubAlgorithmAudioPin2 ();
AudioNet SubAlgorithmAudioPin3 ();
OscillatorSubAlgorithm
Oscillator_1 (oscAudioOut=SubAlgorithmAudioPin2,
SubAlgorithmControl=scaleParamNFOut2, [enabled=1]);
OscillatorSubAlgorithm
Oscillator_2 (oscAudioOut=SubAlgorithmAudioPin3,
SubAlgorithmControl=MyFreqEventSliderOut2,
enabled=1]);
// This is the mixer
add2UG
add2UG (out=add2UGOut,
in1=SubAlgorithmAudioPin3,
in2=SubAlgorithmAudioPin2);
out2sumUG
out2sumUG (in=add2UGOut,
[scale=1.0]);
EXPORTED numberVarNF
```

```
MyFreqEventSlider (out=MyFreqEventSliderOut2,

[par=OscillatorFrequency,

initialize=1,

max=500.0,

min=0.0,

realValue=271.60493]);

scaleParamNF

scaleParamNF (out=scaleParamNFOut2,

in=MyFreqEventSliderOut2,

[parIn=OscillatorFrequency,

parOut=OscillatorFrequency,

scale=1.5,

op="mul"]);

} // HelloWorldHierarchyNumber1

// Continued on the next page

          Algorithm HelloWorldHierarchyNumber2([Version=1,

          SamplingRate=22050,

          TickSize=32,

          BankMSB=0,

          BankLSB=50,

          Program=2])

          {

          ControlPar OscillatorFrequency;

          EventNet MyFreqEventSliderOut2 ();

          AudioNet SubAlgorithmAudioPin2 ();

          OscillatorSubAlgorithm

          Oscillator_1 (oscAudioOut=SubAlgorithmAudioPin2,

          SubAlgorithmControl= MyFreqEventSliderOut2,

          [enabled=1]);

          out2sumUG

          out2sumUG (in= SubAlgorithmAudioPin2,

          [scale=1.0]);

          EXPORTED numberVarNF

          MyFreqEventSlider (out=MyFreqEventSliderOut2,

          [par=OscillatorFrequency,
          initialize=1,
          max=500.0,
          min=0.0,
          realValue=271.60493]);
          } // HelloWorldHierarchyNumber1
} /* end of palette examplePalette */
// Here is where the program assignments occur
SynthstreamAssignment(BankMSB=1,
                              BankLSB=1,
                                        Program=1,
                                        MidiInChannel=1);
SynthstreamAssignment(BankMSB=1,
                              BankLSB=1,
                              Program=2,
                              MidiInChannel=2);
```

Chapter 7 SynthScript Update commands

SynthScript update commands are used to update the value of an internal point in an algorithm. Event that these points are different than what the public API can access. SynthScript update commands can be use to directly set values in the implementation of an algorithm. Typically update commands are used to update the values from sliders on inspectors. Event however, that though update commands can be use to update values in the implementation of algorithm, they cannot be used to change the topology of the algorithm.

Update commands require the MIDI channel that the algorithm is running on in order to be able to access the algorithm. An example of an update command for the freq slider on an oscillator that is allocated on MIDI channel 1 is the following:

```
          1:SubAlgorithm1/myOscillator([freq=445.0]);
```

Notice that the name of the oscillator is a fully hierarchical path. When a slider is moved , these commands can be sent down, and evaluated immediately. For instance a slider move might produce the following:

```
          ...
          1:SubAlgorithm1/myOscillator([freq=445.0]);
          1:SubAlgorithm1/myOscillator([freq=449.0]);
          1:SubAlgorithm1/myOscillator([freq=500.0]);
          1:SubAlgorithm1/myOscillator([freq=530.0]);
          1:SubAlgorithm1/myOscillator([freq=560.0]);
```

```
            1:SubAlgorithm1/myOscillator([freq=600.0]);

            ...
```

In R1.0 only Int, Real values and ControlPar bindings can be sent in update messages. In R1.1, update messages can send any SynthScript constant data type.

Chapter 8 SynthScript Run Time Support Commands

A number of SynthScript commands are provided to support things like runtime allocation of patches, system reset, probing and muting of audio and midi cables.

8.1 Algorithm allocation

Allocation of patches is supported with the command SynthStreamAssignment. The command looks like this. This will allocate the program found at 0/50/1 on MidiChannel number 1 on the default SynthStream. R1.1 may add support for allocating patches on different SynthStreams.

```
            SynthStreamAssignment(BankMSB=0,

            BankLSB=50,

            Program=1,

            MidiInChannel=1);
```

Also for legacy support with older versions of SynthScript, this command may be also invoked as:

```
            MidiChannelAssignment(BankMSB=0,

            BankLSB=50,

            Program=1,

MidiInChannel=1);
```

Also a patch can be deallocated from a MIDI channel with the MidiChannelClear command. For instance, if you wish to deallocate the algorithm allocated on MIDI channel 1, it can be deallcoated with:

```
            MidiChannelClear(1);
```

8.2 MidiChannel Mute/Unmute.

MIDI channels can be muted and unmuted with the MidiChannelMute and MidiChannelUnmute commands. For example muting and unmuting of MIDI channel 1:

```
            MidiChannelMute(1) ;

            MidiChannelUnmute(1) ;
```

8.3 Instance Probe/Unprobe, Mute/Unmute

These commands are a variation on update commands. They require the MIDI channel that the algorithm is running on in order to be able to access the algorithm. They are used to probe/unprobe AudioNets, and to mute/Unmute EventNets.

Probe will probe the output audio pin of a UnitGenerator. It needs to be passed the midiChannel, a fully hierarchical instance name, and the name of the pin to be probed. Here is an example of a probe on the "out" pin of an oscg in a running patch running on MidiChannel 1;

```
            1:Probe(SubAlgorithm1/myOsc, out) ;
```

Here is how to remove that probe.

```
        1:UnProbe() ;
```

Probe will mute the output MIDI pin of a EventFilter. As with Probe, It needs to be passed the midiChannel, a fully hierarchical instance name, and the name of the pin to be muted. Here is an example of a mute on the "out" pin of an scaleNF in a running patch running on MidiChannel 1;

```
        1:Mute(SubAlgorithm1/myScale, out) ;
```

Here is how to remove that mute.

```
        1:UnMute() ;
```

8.4 Simple Commands

The following commands do not take arguments, and are terminated with a semicolon.

```
QUIT Quit streaming SynthScript session

LISTING Enable SynthScript listing generation

NO_LISTING Disable SynthScript listing generation

TRACE_GLOBAL_NOTE_UPDATES Enable tracing of note updates

NO_TRACE_GLOBAL_NOTE_UPDATES Disable tracing of note updates

TRACE_GLOBAL_NOTE_ONS" Enable tracing of note ons

NO_TRACE_GLOBAL_NOTE_ONS Disable tracing of note ons

TRACE_GLOBAL_NOTE_OFFS Enable tracing of note offs

NO_TRACE_GLOBAL_NOTE_OFFS Disable tracing of note offs

TRACE_GLOBAL_NOTE_DURS Enable tracing of note durs

NO_TRACE_GLOBAL_NOTE_DURS Disable tracing of note durs

TRACE_OBJECTS_VERBOSELY Enable tracing of objects verbosely

NO_TRACE_OBJECTS_VERBOSELY Disable tracing of objects verbosely
```

The following commands are only available in debug builds.

```
LIST_TOKENS Enable listing lexical tokens

NO_LIST_TOKENS Disable listing lexical tokens

DUMP_TABLES Dump all DLA tables to the log file

SYSTEM_RESET Reset the whole system, frees memory

MEM_USAGE Dump statistics on mem usage for all tables
```

Chapter 9 Formal Grammar

9.1 Lexical Conventions

Lexically SynthScript consists of tokens, special characters and blank space. All of these lexical entities are formally defined by regular expressions. The lexical analyzer partitions the input stream into token types (which are integers), and values for certain tokens. The parser uses this integer based stream of token types to parse and recognize legitimate syntactic formulations, and to execute actions on reduction of a known formulation.

The lexical conventions for SynthScript are primarily defined as regular expressions. Event that this document will not define the syntax of regular, expressions. These are defined in other books such as the Aho and Ulmans "Principles of Compiler Design". The lexical analyzer in SynthScript is implemented with Lex. The following are the lexical conventions for SynthScript.

9.1.1 Comments

C and C++ style comments are supported. These comments are recognized by the following regular expressions:

```
c_comment "/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

```
cpp_comment "//".*$
```

9.1.2 White Space

White space is defined as either a newline, a tab, a backspace or a carriage return. White space is recognized by the following regular expression:

```
whiteSpace [ \n\t\b\r]
```

9.1.3 Integer Constants

Integer constants can be specified in either conventional integer notation (example: 123), or in hex notation (example: 0xff0b4104). Integer constants are recognized by the following regular expressions:

```
int -?[0-9]+
```

```
hex 0x[a-fA-F0-9]+
```

9.1.4 Real Constants

Real constants can be specified in floating point or scientific notation. Event that corner cases such as 1E-1 and 1e1 are all considered to be real constants.). Real constants are recognized by the following regular expression:

```
real -?(([0-9]+)|([0-9]*(\.[0-9]+)?)([eE][-+]?[0-9]+)?)
```

9.1.5 String Constants

String Constants are defined as quoted strings. String constants are recognized by the following regular expression:

```
string \"[^"]*["]
```

Event that string constants CAN contain a newline character.

9.1.6 Identifiers

Identifiers are recognized by the following regular expression:

```
identifier [a-zA-Z_/][a-zA-Z_0-9/]*
```

9.1.7 Special Characters

The following special characters are recognized as defined.

```
blockBegin "{"

blockEnd "}"

instanceInfoBegin "("

instanceInfoEnd ")"

assignmentToConstant "="

seperator ","

controlParBind ":"

statementTerminator ";"
```

```
        tagBegin "<"

        tagEnd ">"
```

9.1.8 Lexical Include

A lexical include mechanism is provided. When the special token #include is recognized, the next token must be a quoted string. At the time that #include is recognized, the current lexical state is pushed on a stack, and the quoted string is opened a s a file reference. Lexical analysis/parsing continues with this new file, and new lexical state until end of file (EOF) is encountered. When EOF is encountered, the new file is closed, and the lexical state is restored from the stack. Event that unlike C, #include may appear anywhere in a line, and can even appear multiple times.

9.1.9 Keywords

The following identifiers are SynthScript keywords.

```
        "SubAlgorithm"

        "Algorithm"

        "Real"

        "Int"

        "String"

        "IntArray"

        "RealArray"

        "RomUG"

        "List"

        "ControlPar"

        "AudioNet"

        "EventNet"

        "ControlPin"

        "BankMSB"

        "BankLSB"

        "SamplingRate"

        "TickSize"

        "Program"

        "DLEBus"

        "MidiInChannel"

        "MidiOutChannel"

        "EXPORTED"

        "Version"

        "Event"

        "EventOff"

        "EventOn"
```

```
          "EventDur"

          "EventUpdate"

          "Mute"

          "UnMute"

"Probe"

"UnProbe"

"DLS"

"Palette"

"SynthstreamAssignment"

"MidiChannelAssignment"

"MidiChannelClear"

"MidiChannelMute"

"MidiChannelUnmute"

"LISTING"

"NO_LISTING"

"LIST_TOKENS"

"NO_LIST_TOKENS"

"DUMP_TABLES"

"SYSTEM_RESET"

"QUIT"

"MEM_USAGE"

"TRACE_GLOBAL_NOTE_UPDATES"

"NO_TRACE_GLOBAL_NOTE_UPDATES"

"TRACE_GLOBAL_NOTE_ONS"

"NO_TRACE_GLOBAL_NOTE_ONS"

"TRACE_GLOBAL_NOTE_OFFS"

"NO_TRACE_GLOBAL_NOTE_OFFS"

"TRACE_GLOBAL_NOTE_MUTES"

"NO_TRACE_GLOBAL_NOTE_MUTES"

"TRACE_GLOBAL_NOTE_DURS"

"NO_TRACE_GLOBAL_NOTE_DURS"

"TRACE_OBJECTS_VERBOSELY"

"NO_TRACE_OBJECTS_VERBOSELY"
```

9.1.10 Support for error handling

The lexical analyzer provides to mechanisms for error handling.

- If any lexical item does not conform to the SynthScript lexical conventions, an error message is generated, and parsing continues. Event that the parser itself has a mechanism for error handling. This will be descibed later.
- The lexical analyzer keeps a track of a 2 token stack and the index into the current line of text that is being analyzed. If the parser generates an error, this information can be used to generate an error message, that bounds the place where the error occurs. This kind of error message looks something like this:

```
...

(0000028) oscgUG MyOscillator (0000029) (out=oscgUGOut, ; control=sliderNFOut, ^

parse error at line [0000029] between <,>, and <;>.

(0000030) [amp=0.5, (0000031) freq=440.0:OscillatorFrequency, (0000032) phase=0.0,
(0000033) trace=0](0000034) );

...

SynthCore: The SynthScript patch HelloWorld was read,
```

```
0 warnings, 1 errors.
```

9.1.11 Encryption

A simple XOR based encryption mechanism is provided. Encryption/Decryption is handled at the input level of the lexical analyzer. This means that by the time that characters have been read into the lexical analyzer, they have already been decrypted. The goal of encryption in R1.1 was prevent the causual user from viewing DLA files. The goal was NOT to proved a bullet proof method of keeping hard core hackers from viewing SynthScript files.

SynthScript files may be in 3 possible forms:

- Clear text.
- User password encrypted.
- Staccato password encrypted.

There is a magic byte that is not an ASCII character that can optionally appear as the first byte of a SynthScript file. If the byte does not appear as the first byte of a SynthScript file then the file is clear text. If the byte appears with the value STACCATO_ENCRYPTED (0x02), then the file is encrypted with the Staccato password ("Hafnarfjordur") . If the byte appears with the value USER_ENCRYPTED (0x01), then the file is encrypted with a user defined password.

The actual algorithm for encryption is defined in …srctree\tools\xorencrypt. Its based on calculating an XOR byte for each character in the file using the password and the characters file position as a part of the calculation function, and then XOR-ing this byte with the current character.

9.1.12 Reading SynthScript from a file or from an interactive stream

SynthScript can be read from a file, or from an interactive stream. The public API provides procedure calls to either open and read a SynthScript file, or to pass SynthScript to the interactive stream. The reading of SynthScript, either from a file, or from an interactive stream is handled at the input level of the lexical analyzer.

9.2 Parsing SynthScript

Once SynthScript is reduced to a stream of token types (integers) and values for certain tokens, then parsing can begin. . The parser uses this integer based stream of token types to parse and recognize legitimate syntactic formulations, and to execute actions on reduction of a known formulation.

The syntax for SynthScript is specified formally in BNF (Backus-Naur Form). Event that this document will not define BNF. This are defined in other books such as Aho and Ulmans "Principles of Compiler Design". The syntax SynthScript consists of a recursive set of formulations, that are constructed from either terminal token types, or from formulations. These formulations are reduced to a set of parse tables.

Parsing consists using a shift/reduce stack machine to compare the input token type stream to the parse tables. Token types are shifted into the stack machine. If a formulation is recognized in a steam of token types, then the token type stream is reduced to the formulation. This continues until either the input is exhausted, the root formulation is reduced, or an error is generated. Event however, that even errors are defined as a part of the syntax, so that they can be handled, and so that parsing can continue. The parser for SynthScript was written with Yacc.

9.2.1 Error handling in Parsing SynthScript

The key to successful error handling in Yacc, is to include error handling in the definition of the syntax. Event that Yacc places the special token "error" on the shift/reduce stack machine, when an error occurs. The goal is, when an error is detected, to skip forward to a terminating symbol, so that parsing can continue. The trick, is to include the token "error" in the definition of all recursive formulations. Recursive formulations, are basically lists. The following table explains how to transform recursive formulations for effective error handling.

The following are possible recursive formulations.

```
CLOSURE (OPTIONAL SEQUENCE) ==>

(x*)

x_list :

| x_list x

POSITIVE CLOSURE (SEQUENCE) ==>

(x+)

x_list : x

| x_list x

SEQUENCE WITH SEPERATOR ==>

x_list : x

| x_list SEP x
```

These can be transformed for error handling in the following way:

```
x_list : x_list :

| x_list x | x_list x

{ yyerrok;}

| x_list error

x_list : x ==> x_list : x

| x_list x | x_list x

{ yyerrok;}

| error

| x_list error

x_list : x ==> x_list : x
```

```
| x_list SEP x | x_list SEP x

{ yyerrok;}

| error

| x_list error

| x_list error x

{ yyerrok;}

| x_list SEP error
```

9.2.2 BNF syntax for SynthScript.

```
SynthScript

: moduleList

;

moduleList

: /* nothing */

| moduleList module

;

module

: emptyStatement

| patch

| update

| command

| palette

;

emptyStatement

: ';'

;

palette

:

paletteHeader '{' patchList '}'

;

paletteHeader

: PALETTE_KW paletteName '(' ')'

;

paletteName

: IDENTIFIER

;
```

```
patchList

: /* nothing */

| patchList patch

;

patch

: patchHeader patchBody

;

patchHeader

: patchType patchName '(' optionalInterfaceDeclarationList ')'

;

patchType

: PATCH_KW

| SUBPATCH_KW

;

patchName

: patchNameIdOrString

;

patchNameIdOrString

: IDENTIFIER

| STRING

;

optionalInterfaceDeclarationList

: /* nothing */

| interfaceDeclarationList

;

interfaceDeclarationList

: interfaceDeclaration

| interfaceDeclarationList ',' interfaceDeclaration

;

interfaceDeclaration

: hierarchicalConnection

| patchStateVector

;

hierarchicalConnection

: AUDIONET_KW IDENTIFIER

| MIDINET_KW IDENTIFIER

| CONTROL_PIN_KW IDENTIFIER

;

patchStateVector

: '[' patchStateDeclarationList ']'
```

```
;

patchStateDeclarationList

: patchStateDeclaration

| patchStateDeclarationList ',' patchStateDeclaration

;

patchStateDeclaration

: BANKMSB_KW '=' SSP_INT

| BANKLSB_KW '=' SSP_INT

| PROGRAM_KW '=' SSP_INT

| SAMPLING_RATE_KW '=' samplingRateValue

| TICK_SIZE_KW '=' SSP_INT

| VERSION_KW '=' SSP_INT

| IDENTIFIER '=' SSP_INT

| IDENTIFIER '=' SSP_REAL

;

samplingRateValue

: SSP_REAL

| SSP_INT

;

patchBody

: '{' statementList '}'

;

statementList

: /* nothing */

| statementList statement

;


statement

: emptyStatement

| controlParamDeclaration

| instance

| variableAssignment

;

controlParamDeclaration

: CONTROLPARAMETER_KW controlParameterList ';'

;

controlParameterList

: controlParameter

| controlParameterList ',' controlParameter

;
```

```
controlParameter

: IDENTIFIER

;

instance

: instanceBody ';'

;

instanceBody

             : instanceClass instanceName optionalSamplingRate

             '(' optionalAssignmentList ')'

| EXPORTED_KW instanceClass instanceName optionalSamplingRate

             '(' optionalAssignmentList ')'

;

instanceClass

: IDENTIFIER

| STRING

| AUDIONET_KW

| MIDINET_KW

;

/* the reduction of "instanceName" starts adding things to the tables */

instanceName

: instanceNameIdOrString

;

instanceNameIdOrString

: IDENTIFIER

| STRING

;

optionalSamplingRate

: /* nothing */

| SSP_REAL

| SSP_INT

;

optionalAssignmentList

: /* nothing */

| assignmentList

;

assignmentList

: assignment

;

assignment

: patchingAssignment
```

```
| stateAssignmentVector

;

patchingAssignment

: pinName '=' networkName

;

pinName

: IDENTIFIER

;

networkName

: IDENTIFIER

;

stateAssignmentVector

: '[' optionalStateAssignmentList ']'

;

optionalStateAssignmentList

: /* nothing */

| stateAssignmentList

;

stateAssignmentList

: stateAssignment

| stateAssignmentList ',' stateAssignment

;

stateAssignment

: stateAssignmentVariableName '=' SSP_REAL

| stateAssignmentVariableName '=' SSP_REAL ':' IDENTIFIER

| stateAssignmentVariableName '=' SSP_INT

| stateAssignmentVariableName '=' SSP_INT ':' IDENTIFIER

| stateAssignmentVariableName '=' STRING

| stateAssignmentVariableName '=' STRING ':' IDENTIFIER

| stateAssignmentVariableName '=' array

| stateAssignmentVariableName '=' array ':' IDENTIFIER

| stateAssignmentVariableName '=' note

| stateAssignmentVariableName '=' note ':' IDENTIFIER

| stateAssignmentVariableName '=' list

| stateAssignmentVariableName '=' list ':' IDENTIFIER

| stateAssignmentVariableName ':' IDENTIFIER

| stateAssignmentVariableName '=' IDENTIFIER

| stateAssignmentVariableName '=' IDENTIFIER ':' IDENTIFIER

;

stateAssignmentVariableName
```

```
: IDENTIFIER

;

/* variableAssignment */

variableAssignment

: variableTypeDeclaration ';'

;

variableTypeDeclaration

: realVariable

| intVariable

| stringVariable

| noteVariable

| listVariable

| realArrayVariable

| intArrayVariable

| DSPrealArrayVariable

;

realVariable

: SSP_REAL_KW variable '=' SSP_REAL

;

intVariable

: SSP_INT_KW variable '=' SSP_INT

;

stringVariable

: STRING_KW variable '=' STRING

;

noteVariable

: NOTE_KW variable '=' note

;

note

: '{' noteType '}'

| '{' noteType ',' noteParList '}'

;

noteType

: NOTEON_KW

| NOTEOFF_KW

| NOTEUPDATE_KW }

| NOTEDUR_KW

| MUTE_KW

;

noteParList
```

```
: noteParAssignment

| noteParList ',' noteParAssignment

;

noteParAssignment

: notePar '=' SSP_REAL

| notePar '=' SSP_INT

| notePar '=' STRING

| notePar '=' array

| notePar '=' list

;

notePar

: IDENTIFIER

;

realArrayVariable

: realArrayKeyword variable '=' array

;

realArrayKeyword

: REAL_ARRAY_KW

;

intArrayVariable

: intArrayKeyword variable '=' array

;

intArrayKeyword

: INT_ARRAY_KW

;

DSPrealArrayVariable

: DSPrealArrayKeyword variable '=' array

;

DSPrealArrayKeyword

: ROMUG_KW

;

listVariable

: LIST_KW variable '=' list

;

variable

: IDENTIFIER

;

// the array data type

array

: '{' arrayList '}'
```

```
;

arrayList

: arrayListItem

| arrayList ',' arrayListItem

| error

| arrayList error

| arrayList error arrayListItem

{ yyerrok;}

| arrayList ',' error

;

arrayListItem

: SSP_INT

| SSP_REAL

| STRING

| STRING '@' dlsDescriptor

;

dlsDescriptor

: DLS_KW '(' SSP_INT ':' SSP_INT ':' SSP_INT ':' SSP_INT ':' SSP_INT ')'

| DLS_KW '(' SSP_INT ':' SSP_INT ':' SSP_INT ')'

;

// the list data type

list

: startList listBody endList

;

listBody

: /* nothing ... the nil list () is supported as well */

| listBody listItem

;

listItem

: array

| note

| '{' instanceBody '}'

| '{' variableTypeDeclaration '}'

| IDENTIFIER

| list

;

startList

: '('

;

endList
: ')'

;

/*

code to handle updates, Updates are sent by

inspector sliders to the server

*/

update

: stateProcUpdate

| probe

| unprobe

| mute

| unmute

;

stateProcUpdate

: midiChannel ':' updateInstanceName '(' updateVector ')' ';'

;

midiChannel

: SSP_INT

;

updateInstanceName

: IDENTIFIER

;

updateVector

: '[' updateExpression ']'

;

updateExpression

: updateStateProcName '=' SSP_REAL

| updateStateProcName '=' SSP_REAL ':' IDENTIFIER

| updateStateProcName '=' SSP_INT

| updateStateProcName '=' SSP_INT ':' IDENTIFIER

| updateStateProcName '=' STRING

| updateStateProcName '=' STRING ':' IDENTIFIER

| updateStateProcName '=' array

| updateStateProcName '=' array ':' IDENTIFIER

| updateStateProcName '=' list

| updateStateProcName '=' list ':' IDENTIFIER

| updateStateProcName '=' IDENTIFIER

| updateStateProcName ':' IDENTIFIER

;

updateStateProcName
```

```
: IDENTIFIER

;

/*

Probe and Mute commands

*/

probe

: midiChannel ':' PROBE_KW '(' updateInstanceName

',' probeInstancePin ')' ';'

;

probeInstancePin

: IDENTIFIER

;

unprobe

: midiChannel ':' UNPROBE_KW '(' ')' ';'

;

mute

: midiChannel ':' MUTE_KW '(' updateInstanceName ','

muteInstancePin ')' ';'

;

muteInstancePin

: IDENTIFIER

;


unmute

: midiChannel ':' UNMUTE_KW '(' updateInstanceName ','

unmuteInstancePin ')' ';'

;

unmuteInstancePin

: IDENTIFIER

;

/* commands */

command

: synthstreamAssignment

| channelClear

| channelMute

| channelUnmute

| simpleCmd

;

/* channel synthstreamAssignment */

synthstreamAssignment
```

```
: synthstreamAssignmentKW '('

             optionalSynthstreamAssignmentParameterList ')' ';'

;

synthstreamAssignmentKW

: MIDI_CHANNEL_ASSIGNMENT_KW

| SOUND_STREAM_ASSIGNMENT_KW

;

optionalSynthstreamAssignmentParameterList

: /* nothing */

| soundSteamAssignmentParameterList

;

soundSteamAssignmentParameterList

: soundSteamAssignmentParameter

| soundSteamAssignmentParameterList ',' soundSteamAssignmentParameter

;

soundSteamAssignmentParameter

: MIDI_IN_CHANNEL_KW '=' SSP_INT

| MIDI_OUT_CHANNEL_KW '=' SSP_INT

| BANKMSB_KW '=' SSP_INT

| BANKLSB_KW '=' SSP_INT

| PROGRAM_KW '=' SSP_INT

| DLE_KW '=' SSP_INT

| IDENTIFIER '=' SSP_INT

| IDENTIFIER '=' SSP_REAL

;

channelClear

: MIDI_CHANNEL_CLEAR_KW '(' SSP_INT ')' ';'

;

channelMute

: MIDI_CHANNEL_MUTE_KW '(' SSP_INT ')' ';'

;

channelUnmute

: MIDI_CHANNEL_UNMUTE_KW '(' SSP_INT ')' ';'

;

/* simple commands */

simpleCmd

: QUIT_KW ';'

| DUMP_TABLES_KW ';'

| LISTING_KW ';'

| NOLISTING_KW ';'
| LISTTOKENS_KW ';'

| NOLISTTOKENS_KW ';'

| MEM_USAGE_KW ';'

| SYSTEM_RESET_KW ';'

| TRACE_GLOBAL_NOTE_UPDATES_KW ';'

| NOTRACE_GLOBAL_NOTE_UPDATES_KW ';'

| TRACE_GLOBAL_NOTE_ONS_KW ';'

| NOTRACE_GLOBAL_NOTE_ONS_KW ';'

| TRACE_GLOBAL_NOTE_MUTES_KW ';'

| NOTRACE_GLOBAL_NOTE_MUTES_KW ';'

| TRACE_GLOBAL_NOTE_DURS_KW ';'

| NOTRACE_GLOBAL_NOTE_DURS_KW ';'

| TRACE_GLOBAL_NOTE_OFFS_KW ';'

| NOTRACE_GLOBAL_NOTE_OFFS_KW ';'

| TRACE_OBJECTS_VERBOSELY_KW ';'

| NOTRACE_OBJECTS_VERBOSELY_KW ';'

;
```

9.3 Pre-defined ControlPars in SynthScript

Some ControlPars are, by convention, predefined in SynthCore to have particular uses.

9.3.1 ControlPars used in the representation of MIDI

MIDI messages are converted into events when they enter SynthCore as listed below.

9.3.1.1 MIDI Channel Voice Messages

Note that there is no representation for MIDI channel in the event object—the MIDI channel is used by the DLA Manager to forward the event to the proper DLA.

- **Note On messages** are converted to event objects with eventType *noteOn* and controlPars *keyNum* [0-127] and *normalizedVelocity* [0.007-1.0]. MIDI Note Ons with a velocity of 0.0 are considered the same as MIDI Note Off.
- **Note Off messages** are converted to event objects with eventType *noteOff* and controlPars *keyNum* [0-127] and *normalizedRelVelocity* [0.0-1.0], which specifies the release velocity. MIDI Note Ons with a velocity of 0.0 are converted to *noteOff* events, but are not given a *normalizedRelVelocity* parameter.
- **Polyphonic Key Pressure messages** are converted to event objects with eventType *noteUpdate* and controlPars *keyNum* [0-127] and *normalizedKeyPressure* [0.0-1.0].
- **After-touch (channel pressure) message** are converted to event objects with eventType *noteUpdate* and controlPar *normalizedAfterTouch* [0.0-1.0].
- **Program change** messages are converted to event objects with eventType *noteUpdate* and controlPar *programChange* [0-127]. In addition, program change events cause the DLA Manager to (possibly) select a new DLA for the given MIDI channel.
- **Pitch bend messages** are converted to event objects with eventType *noteUpdate* and controlPar *pitchBend* [0-0x3fff]. The default value is 0x2000, which represents no pitch bend.
- **Controller messages** are converted to event objects with eventType *noteUpdate* and a controlPar that specifies which controller was changed. The controlPars for MIDI controllers 0-120 are the following (names taken from the MIDI specification). All are normalized in the range [0.0-1.0], except as noted. LSBs are not currently

automatically combined with MSBs except where noted. ControlPars in parenthesis are used internally only.

ControlPar	MIDI Number	Special Notes
bank Select	0	*[0-127]
modWheel	1	
breathController	2	
control3	3	
footController	4	
portamentoTime	5	
dataEntryMSB	6	See below
channelVolume	7	*
balance	8	
control9	9	
pan	10	*
expressionController	11	*
effectControl1	12	*
effectControl2	13	*
control14,	14	
control15	15	
generalPurposeController1-		
generalPurposeController4	16-19	
control20-control31	20-31	
bankSelectLSB	32	[0-127]
modWheelLSB	33	
breathControllerLSB	34	
<etc. as above: LSB>	35-63	< * as with MSB >
damperPedal	64	
portamentoOnOff	65	
sostenuto	66	
softPedal	67	
legatoFootswitch	68	
hold2	69	
soundController1-		
soundController10	70-79	*
generalPurpose5-		
generalPurpose8	80-83	
portamentoControl		
control85-90	85-90	
effects1Depth-		
effects5Depth	91-95	*
dataIncrement	96	See below

```
dataDecrement 97 See below

(nonRegisteredParameterNumberLSB) 98 See below

nonRegisteredParameterNumber 99 [0-0x3fff] See below

(registeredParameterNumberLSB) 100 See below

(registeredParameterNumberMSB) 101 See below

control102-control119 102-119
```

Special notes

1. *dataIncrement*, *dataDecrement*, dataEntryMSB and dataEntryLSB are sent as controlPars only when they apply to NRPNs. When the apply to RPNs, they are intercepted by the SynthCore MIDI parser and interpreted to set the appropriate value.

2. The *nonRegisteredParameterNumber* controlPar contains a 14-bit number with both the MSB and LSB of the NRPN.

3. RPNs are interpreted by the MIDI parser and sent as special parameters:

a. the RPN "pitchBendSensitivity" is converted to the controlPar

pitchBendSensitivity, with units in semitones (from -127.99 to 127.99).

b. the RPNs "fineTuning" and "coarseTuning" are combined and included in

a single parameter *tuning* with units in semitones (from -64.99 to 64.99).

c. the RPN "tuningProgramSelect" is sent as an integer

d. the RPN "tuningBankSelect" is sent as an integer

4. '*' under "special notes" above indicates that the controller does not get reset when the MIDI resetAllControllers message is received.

5. When a DLA is allocated, it is sent an event that describes the sticky state of the MIDI channel on which the DLA is listening. This event is of type noteUpdate and contains the following information:

   a. Current value of pitchBend. (initial value = center position = 0x2000)
   b. Current value of normalizedAfterTouch. (initial value = 1.0)
   c. All MIDI controllers-based controlPars that have either been received via MIDI or that have default values that are not 0.0.

The list of MIDI controllers-based controlPars with non-zero defaults is:

channelVolume (initial value = 100/127.0)

expressionController (initial value = 1.0)

pan (initial value = 0.5)

effects1Depth (initial value = 40/127.0)

Any MIDI controllers-based controlPar not received by the DLA should be assumed to have the value 0.0.

9.3.1.2 MIDI Channel Mode

Messages are converted to event objects with eventType *mute* and a controlPar *chanMode*, which can accept the following constant values:

```
allSoundOff

resetControllers
```

`localControlModeOn`

`localControlModeOff`

`allNotesOff`

`omniModeOff`

`omniModeOn`

`monoMode`

`polyMode`

In addition to being included in the event object, certain of these are interpreted by SynthCore in special ways. *allSoundOff* and *allNotesOff* both stop any sounding voices. *resetControllers* behaves as suggested by the MIDI Developer Guidelines. It resets all MIDI controllers to 0 except those indicated above with '*' in section 9.3.1.1. This is sent to the DLA listening on that MIDI channel in the form of an event with the appropriate parameters. In addition, the event contains a pitchBend value of 0x2000 (center value) and a normalizedAfterTouch value of 0.0. Then a series of 128 events are sent to reset the normalizedKeyPressure to 0.0 for each of the 128 keyNum values. Each of these events includes both a normalizedKeyPressure value and a keyNum value.

All of the other *chanMode* values have no pre-defined effect. Note that the *monoMode* value includes a second controlPar, *monoChans* [0-15], as specified in the MIDI specification.

9.3.1.3 MIDI System Common

Messages are converted to event objects with eventType *mute* and a controlPar that depends on the type of message. MIDI Time Code Quarter Frame messages include a controlPar *timeCodeQ* [0-127]. Song Position Pointer messages include a controlPar *songPosition* [0-0x3fff], which contains both the MSB and the LSB of the song position. Song Select messages include the controlPar *songSelect* [0-127]. Tune Request messages are indicated by the presence of the controlPar *tuneRequest*; its value is irrelevant.

9.3.1.4-MIDI System Real Time

Messages are converted to event objects with eventType *mute* and a controlPar *sysRealTime*, which can accept the following constant values:

`sysClock`

`sysUndefined0xf9`

`sysStart`

`sysContinue`

`sysStop`

`sysUndefined0xfd`

`sysActiveSensing,`

`sysReset`

9.3.1.5-MIDI System Exclusive

Messages are converted to event objects with eventType *mute* (unless otherwise specified below) and a controlPar *sysExclusive*, whose value is an IntArray containing the system exclusive data, packed little-endian.

Several special System Exclusive messages are interpreted by SynthCore. These include:

• MIDI master tuning – This is encoded in the *masterTuning* controlPar as a real value in semitones. The voiceAllocNF uses this to produce the proper tuning offset for each note. This parameter is included in an event of type *noteUpdate*.

9.3.1.6 MIDI Channel Mode

Messages are converted to event objects with eventType

9.3.2 ControlPars provided by the Voice Allocator

Cetrain pre-defined controlPars are managed by SynthCore to make it easy to develop sophisticated synthesis algorithms.

9.3.2.1 Automatic Management of Frequency

The voiceAllocNF provides two controlPars, *freq* and *pitch*, as a convenience to the DLA developer. It does this by combining all the MIDI controlPars that deal with the sounding pitch of the note (*keyNum*, *pitchBend*, *tuning*, *pitchBendSensitivity*, etc.), and converting them to a single convenient value, which is presented in two formats: *freq* in Hertz, and *pitch* in scale degrees. Both values are reals.

An example will help to clarify this. If *pitchBendSensitivity* is 1, *pitchBend* is 0x1000, and *keyNum* is 69, then *pitch* will be set to 68.5 and *freq* will be set to a quarter tone below 440.0.

Note that *freq* and *pitch* are provided in any event that deals with pitch. For example, all incoming events with controlPar *pitchBend* are converted into events with *freq* and *pitch* controlPars by voiceAllocNF.

9.3.2.2 Information on Voice Status

The voiceAllocNF does sophisticated management of synthesis resources, forwarding notes to appropriate voices (depending on its mode), and preempting voices when synthesis resources are scarce. Often, the voices (subpatches) themselves need some information about what role they are currently playing. To support this, the voiceAllocNF inserts a controlPar *voiceStatus* into the events it sends. The following values of *voiceStatus* are supported:

- *VoiceStart*—A new voice, not previously sounding, has been allocated in response to a *noteOn* event.
- *VoiceRestart*—A voice that was previously sounding, and had already received a *noteOff* event, has received a *noteOn* event, and is thus being "restarted" (aka "retrigger".)
- *VoiceRestartLegato*—A voice that was previously sounding, and has not already received a *noteOff* event, has received another *noteOn* event.
- *VoiceRelease*—A voice that was previously sounding has just received a *noteOff* event.
- *VoicePreempt*—A voice that was previously sounding is about to be preempted because synthesis resources are low.
- *VoiceEnd*—A voice that was previously sounding is finished making sound and is about to return to the available pool of voices.

Events with *voiceStatus* set to *voicePreempt* also contain an additional controlPar, *preemptionTime*, which is the duration, in seconds, when the voice will be sent the *voiceEnd* value. This enables a voice to know how much time it has to "clean up" in a smooth manner. For example, envUG optionally smooths its envelope to its final value.

9.3.3 ControlPar to Support NoteDur events

Events may have a number of different event types. These are discussed elsewhere in this document [or are they?]. One eventType, *NoteDur*, requires a special controlPar, *dur*, which specifies the duration of the event. That is, a *NoteDur* is a combination of a *NoteOn* and a *noteOff* in a single event. The specified duration is the time (in seconds) between the implied *noteOn* and the implied *noteOff*. It is important to understand that not all objects interpret *dur*. In order for a *NoteDur* to be properly interpreted it should be sent to one of the following objects:

voiceAllocNF

midiOutNF

9.3.4 ControlPar to Support Standard MIDI File events

Standard MIDI files are represented as event lists. Most of the controlPars needed to represent them are included in section 9.3.1. However, a few special controlPars apply only to events derived from Standard MIDI Files:

9.3.4.1 File channel, track and port meta-events

All of these meta-events deal with the location of other events in a sequence. While events derived from performed MIDI have no channel parameter, since the DLA manager forwards them to the right destination automatically, events derived from Standard MIDI Files do need to specify the MIDI channel as it appeared in the file [0-15]. These appear as a controlPar *fChan* in any event that has a channel. In addition, Standard MIDI Files have track structure. This is represented in the events with a controlPar *track* [0-infinity]. Finally, the MIDI port that an event is associated with is represented by the *fPort* [0-15] controlPar.

9.3.4.2 Meta events

Standard MIDI Files define a variety of "meta-events" which are represented as SynthCore events depending on their type.

The following controlPars have numeric values:

```
sequenceNumber [0-15]

tempo (in BPM, not "microseconds per MIDI quarter-note")
```

The following controlPars have string values:

```
textEvent

copyrightNotice

sequenceOrTrackName

instrumentName

lyric

marker

cuePoint
```

The following controlPars have object values:

```
smpteOffset -- IntArray

timeSignature -- IntArray

keySignature –- IntArray

sequencerSpecific -- Packed IntArray (big-endian)

escapeEvent -- Packed IntArray (big-endian)
```

9.4 Midi file representation in EventList

Midi files can be represented as noteLists in SynthScript. In order to represent time, a new ControlPar called "beat" is introduced. Also note that a new ControlPar called fChan has been introduced to represent MidiChannel.

```
List BachFugue = (

{Mute, tempo=0.151667, beat=0, fChan=1, fPort=1, track=1}

{Mute, sequenceOrTrackName="part1", beat=0, fChan=1, fPort=1, track=3}

{Mute, sequenceOrTrackName="part2", beat=0, fChan=1, fPort=1, track=4}

{Mute, sequenceOrTrackName="part3", beat=0, fChan=1, fPort=1, track=5}

{EventOn, keyNum=67, normalizedVelocity=0.503937, beat=3.0332, fChan=4, fPort=1, track=4}

{EventOff, keyNum=67, beat=3.41211, fChan=4, fPort=1, track=4}

{EventOn, keyNum=69, normalizedVelocity=0.503937, beat=3.41211, fChan=4, fPort=1, track=4}

{EventOff, keyNum=69, beat=3.79297, fChan=4, fPort=1, track=4}

{EventOn, keyNum=71, normalizedVelocity=0.503937, beat=3.79297, fChan=4, fPort=1, track=4}

{EventOff, keyNum=71, beat=4.17188, fChan=4, fPort=1, track=4}
{EventOn, keyNum=72, normalizedVelocity=0.503937, beat=4.17188, fChan=4, fPort=1, track=4}

{EventOff, keyNum=72, beat=4.54883, fChan=4, fPort=1, track=4}

{EventOn, keyNum=67, normalizedVelocity=0.503937, beat=4.54883, fChan=4, fPort=1, track=4}

. . .

{EventOff, keyNum=67, beat=202.853, fChan=3, fPort=1, track=3}

{EventOff, keyNum=43, beat=202.853, fChan=4, fPort=1, track=4}

{EventOn, keyNum=36, normalizedVelocity=0.503937, beat=202.853, fChan=4, fPort=1, track=4}

{EventOff, keyNum=64, beat=208.919, fChan=2, fPort=1, track=2}

{EventOff, keyNum=72, beat=208.919, fChan=2, fPort=1, track=2}

{EventOff, keyNum=36, beat=208.919, fChan=4, fPort=1, track=4}

);
```