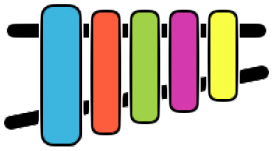


MidiTalk

McLaren Labs

Nov 15, 2024

by Tom Sheffler
Version 1.0
Copyright © McLaren Labs 2024

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

```
MidiTalk := MIDI + GNUstep + SmallTalk + StepTalk
```

MidiTalk is a scriptable application for manipulating MIDI, Audio and Patterns in real-time using a scripting language called StepTalk. MidiTalk comes with a large collection of predefined classes and objects so that it is easy to create customized applications with small scripts. Most scripts simply provide the "glue" that connect objects in interesting ways for a particular application.

What are some of the ways that you can use MidiTalk?

- a programmable virtual control surface
- a MIDI translator and processor
- a programmable Synth
- a programmable arpeggiator

MidiTalk is built on the foundation of the GNUstep programming environment and the StepTalk interpreted language. StepTalk interfaces directly with Objective-C objects and classes. All Objective-C classes (including Foundation and AppKit) can be scripted by StepTalk, which makes it very powerful.

In the MidiTalk application, the McLaren Labs Objective-C libraries known as the `AlsaSoundKit` and `McLarenSynthKit` are used to manage MIDI and Audio devices on Linux. StepTalk can be used with these with no modifications to the libraries themselves. The MidiTalk application simply provides a GUI with a small collection of widgets that are useful for MIDI and audio. The rest is done with scripts.

1.1 A Full Example

The example script in Listing 1.1 shows how to make a synth sound and connect the on-screen piano keyboard to play it. It consists of two methods: `configurePiano` and `main`.

The `main` method is called first when a script is loaded. Here it creates an instance of the `Synth80` class and assigns it to the variable "synth". Synth80 is a McLaren Labs synthesizer program that accompanies MidiTalk. The Synth80 initializer sets its output to

`ctx` - the default audio output context of MidiScript. The “FatBass” patch is then loaded into this instance of Synth80 for playing.

The `configurePiano` method is then called. This method is interesting: it registers two blocks that are called on the “noteOn” and “noteOff” events of the on-screen piano keyboard. These blocks receive two arguments: the `midiNote` played and the `vel` (velocity) of the keypress. MidiTalk maps input MIDI events to StepTalk, which then calls these blocks to handle the events.

That’s it! We’ve shown how to use a built-in object - the piano keyboard. And we have shown how to use a built-in class - the Synth80 synthesizer. Our program has stiched the two together with StepTalk blocks as the callback functions.

We’ve also shown some of the syntax of a StepTalk script. A script consists of a series of method definitions separated by the “!” (bang) symbol. A script starts with `[|` (bracket, vertical-bar) and ends with `]` (bracket). Each statement ends with a “.” (period). If you’re familiar with Objective-C programming you may recognize some familiar constructs too. The `alloc` and `init` allocator and initializer methods used with the `Synth80Synth` class are exactly the same methods that would be called in Objective-C, except that StepTalk syntax is used here.

> ***Aside***: In some ways the StepTalk conceptualization of this example program is more powerful and convenient than plain Objective-C. Using blocks for callbacks is convenient and natural here, and they are objects just like everything else in StepTalk. In Objective-C 2.0, blocks are similarly convenient, but they are not exactly objects, because they cannot be evaulated by sending them a message.

1.2 System Context

The MidiTalk program interacts with its environment through Audio and MIDI interfaces. There is one Audio-OUT, one Audio-IN, one MIDI-OUT and one MIDI-IN pre-allocated by the program[1]. When it starts, the MidiTalk program has no defined behaviors, beyond displaying the user interface controls and displaying the audio levels. Figure 1.1 below illustrates the System Context and major system interfaces of MidiTalk.

Behaviors are added to MidiTalk through StepTalk scripts. A number of examples and useful scripts are provided with the MidiTalk program in the Application Bundle (`MidiTalk.app/Scripts`). Users are **encouraged** to add their own scripts to their system as well! Scripts are searched for in standard locations. On Linux user-defined scripts are usually kept in `~/Library/StepTalk/Scripts/MidiTalk/`.

[1] It is possible to create more interfaces, and we will show how later.

Listing 1.1 Connecting the keyboard to Synth80

```
"Demonstration: connect the keyboard to a synth sound"
[|

configurePiano

  piano onNoteOn: [ :midiNote :vel |
    synth noteOn:midiNote vel:vel .
  ] .

  piano onNoteOff: [ :midiNote :vel |
    synth noteOff:midiNote vel:vel .
  ] .
!

"the main method is called when the script is loaded"
main
  synth := Synth80Synth alloc initWithCtx:ctx .
  synth loadPatch:'FatBass' .

  self configurePiano .

]
```

1.3 Tour of the User Interface

The user interface consists of a collection of controls and indicators, some of which have predefined behaviors and many that do not. Figure 1.2 shows the GUI widgets in MidiTalk. Each is described in the list below: starting at the top-left and proceeding clockwise around the window.

downbeat indicator
: This indicator blinks light-blue on the downbeat of each measure.

metronome controls
: These buttons start, stop and continue the metronome. The metronome provides the time-base for user-defined patterns. By default, the metronome makes no sounds. These must be defined by patterns in scripts.

tempo control
: The tempo control is a slider. The current value of the tempo is shown in the textbox to the right of it.

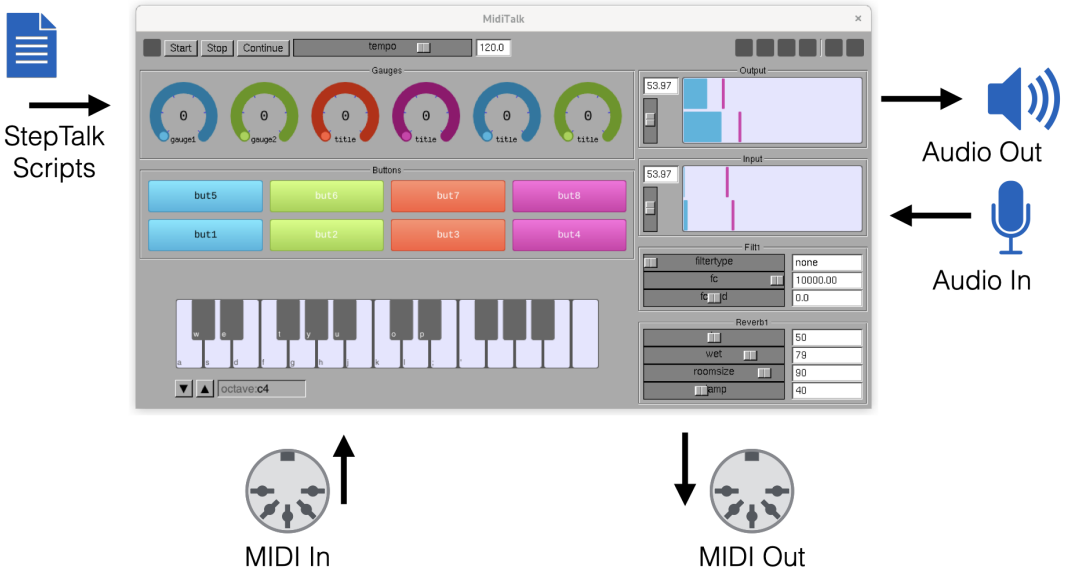


Figure 1.1: System Interfaces

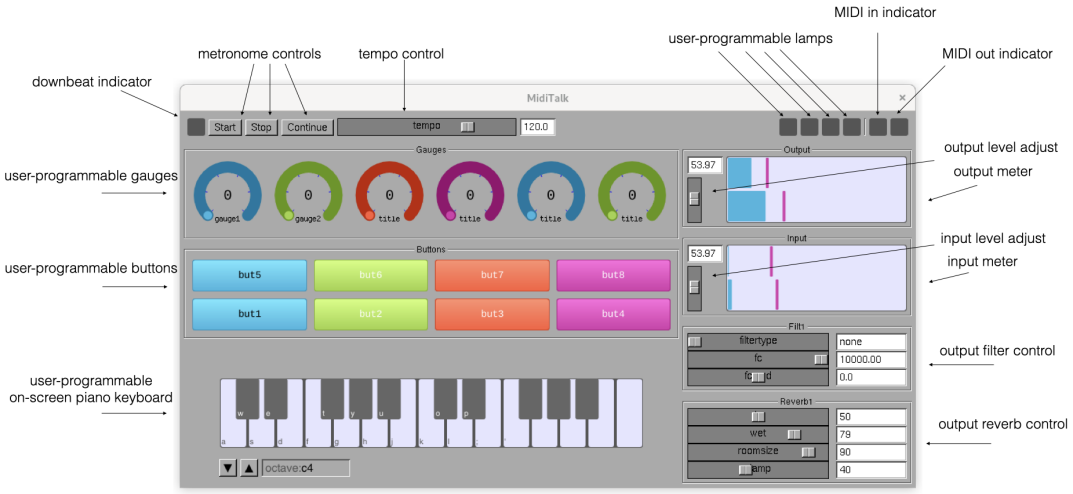


Figure 1.2: GUI Widgets of MidiTalk

user-programmable lamps
The four lamps can be turned on and off by scripts. Their "on" color may also be changed.

MIDI in indicator
The MIDI-In indicator flashes when a MIDI event is sent to the sequencer interface of the MidiTalk program.

MIDI out indicator
The MIDI-Out indicator flashes when the MidiTalk program emits a MIDI event.

output level adjust
The output level adjustment consists of the slider and text box in the "Output" area.

output meter
The output VM meter displays the absolute peak and RMS values of the audio output.

input level adjust
The input level adjustment controls the gain of the audio input.

input meter
The input VM meter displays the absolute peak and RMS values of the audio output.

output filter control
The output "effects" stage consists of a filter and a reverb unit. The filter type, cutoff frequency and modulation value are adjusted by the controls in the "Filt1" area.

output reverb control
The second stage of the output "effects" chain consists of a reverb unit. The controls in the "Reverb1" area adjust its parameters.

on-screen piano keyboard
The Piano keyboard is both an input device and an output device. In StepTalk, you may respond to `onNoteOn:` and `onNoteOff:` messages. To show the keys as depressed you may call the `noteOn` method, and to un-depress a key call `noteOff`.

user-programmable buttons
The buttons are input devices that emit MIDI-like `onNoteOn`, `onNoteOff` and `onKeyPressure` messages. While a button is depressed, dragging the mouse caues a continual series of `onKeyPressure` messages to be emitted with a velocity value related to how far the mouse is dragged.

Buttons may be assigned different colors or titles through StepTalk code.

user-programmable gauges
The gauges also emit MIDI-like control change messages that can be subscribed to with the `onChange` method. The title and color of the gauges can be changed in StepTalk.

1.4 Predefined Objects

MidiTalk provides a collection of predefined useful objects. These have been `alloc`'d and `init`'d and configured before any script is loaded. By having these ready to go, most scripts do not need to deal with the initialization of Audio and MIDI devices. The predefined objects are shown in Figure 1.3.

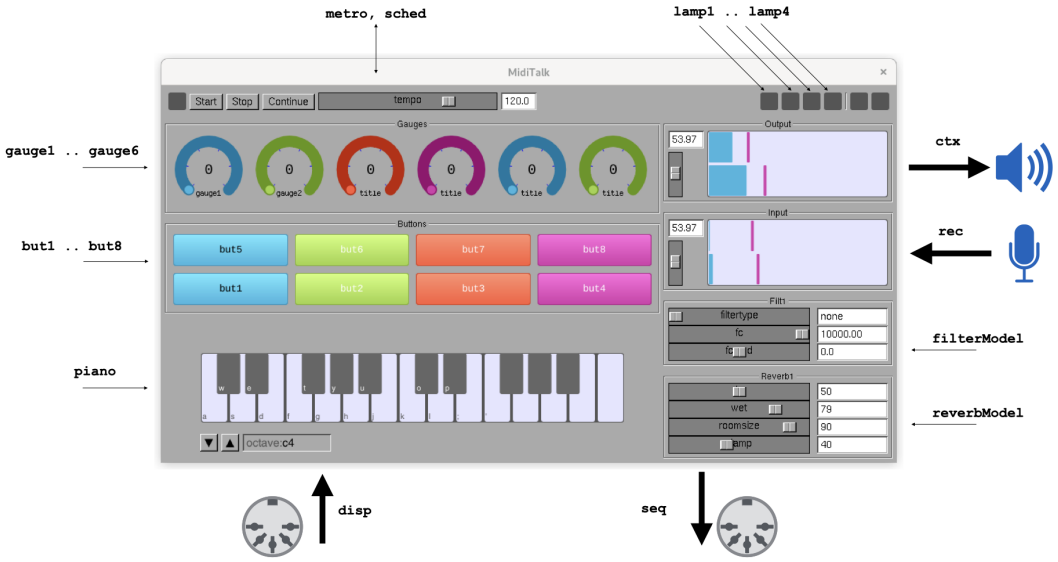


Figure 1.3: Predefined Objects of MidiTalk

Each of the pre-defined objects is described in the list that follows. The class of each is also given.

`ctx : MSKContext`
: This is the default audio output context.

  It opens the ALSA audio output device named "default".

`rec : MSKContext`
: This is the default audio input context.

  It opens the ALSA audio input device named "default".

`seq : ASKSeq`
: The default MIDI ouptut sequencer device. MidiTalk creates a new ALSA MIDI Sequencer client named "miditalk".

`disp : ASKSeqDispatcher`
: The default MIDI input event dispatcher. MidiTalk registers to listen to events sent to the ALSA MIDI Sequencer client named “miditalk”.

`but1 .. but8 : MLExpressiveButton`
: There are eight on-screen expressive buttons. They are called “expressive” because they emit continuous `keyPressure` messages when mouse-drag events are sent to them. Their titles are easily changed in StepTalk.

`gauge1 .. gauge6 : MLGauge`
: There are six on-screen gauges. Their titles are easily changed in StepTalk. By default, the range of each gauge is 0..127 with an increment of 1. These values may be changed in StepTalk, as may be the formatting of the value.

`piano : MLPiano`
: The on-screen piano keyboard. This is both an input and an output. Mouse presses turn into `onNoteOn` and `onNoteOff` messages. But the keys may be shown depressed or raised with the `noteOn` and `noteOff` messages.

`lamp1 .. lamp4 : MLLamp`
: Four on-screen programmable lamps. They may be turned on and off with `on` and `off` messages. Their illuminated color may also be changed in StepTalk.

`metro : MSKMetronome`
: The metronome is used to set the tempo and time signature of the scheduler.

`sched : MSKScheduler`
: The pattern scheduler is used to program repeating sequences and rhythms. There will be an entire chapter devoted to this topic later.

`filterModel : MSKFilterModel`
: The output effects pipeline begins with a filter. Its parameters may be adjusted by accessing this model.

`reverbModel : MSKReverbModel`
: The output effects pipeline ends with the reverb unit. Its parameters may be adjusted by accessing this model.

1.5 Polyphony

At this point in our introduction, it is appropriate to note the approach to polyphony in MidiTalk. MidiTalk has been designed from the ground up to be polyphonic. It gains this capability because it is built on the `McLarenSynthKit`.

The `McLarenSynthKit` defines a family of operators, called “voices”, that can be chained together into directed graphs that define audio-processing pipelines. Voices are what

define oscillators, envelopes, filters and effects. Any number of graphs may be running at the same time — up to the limit of the processor MidiTalk is running on.

Listing 1.2, below, is a StepTalk program for launching oscillators each time a key is pressed on the on-screen piano. The method `makeNote` accepts an argument, `midiNote`, that is the note to be played. The method creates two objects: an Envelope and an Oscillator and hands them to the output context for playing.

The envelope is configured as a "one-shot", which fires for a specified duration and then begins its release. The oscillator is assigned the note value, and its envelope input is set to the previously-defined envelope. Note that both of the Voice objects have the `compile` method called after their configuration is complete. This method causes the internals of the Voice to be finalized.

The `main` method defines the models for the envelope and the oscillator. All default values are used, so there is no configuration of these objects. These models are used for all sounding notes, so that if there are many active in the output context at the same time, they are all getting their model parameters from the same place. This is the mechanism by which configuration fans out over polyphonically sounding voices. Voices that share models receive the same parameters as they play.

> **ASIDE:** Why does the `main` method retain `self`? A StepTalk script is normally released to the memory manager for reclamation after it is read and executed. Here, we have an issue with that policy. The `onNoteOn:` block references `self`, which is the StepTalk script itself. Since the script is reclaimed, when the block is executed, the script is gone and the `makeNote` method is not found. The solution is to store a reference to the script in the environment.

1.6 A Pattern example with Lamps

One of the interesting things about MidiTalk is its support for Patterns. A Pattern is a little program that executes in its own context, almost like a thread, but it is driven by the metronome timebase. A Pattern can synchronize with the Metronome and wait for beats, or delay for MIDI ticks. A Pattern expresses a computation over time that may repeat.

The example below, in Listing 1.3, turns on the lamps in sequence. The Pattern definition illustrates a couple of interesting things.

Line 3 defines the new pattern object and gives it a name, "pattern1". The name is used mostly for debugging.

Lines 4–11 define the pattern. It consists of a series of synchronizations with the metronome, and blocks that are executed when the synchronization occurs. For example, line 5 specifies the pattern synchronizing with the metronome on the next downbeat, and

then executing the block that turns off lamp4 and turns on lamp1. Line 7 synchronizes with the next beat, and lines 9 and 11 are similar.

When a pattern reaches its last statement it exits, unless it has been configured to repeat.

Line 14 adds the pattern to the "LaunchSpec" of the Scheduler, which means that when the metronome is restarted this pattern will be relaunched.

If you load this script and press the `[Start]` button of the metronome, you will see the lamps illuminated in turn at the tempo selected.

1.7 Starting MidiTalk and connecting external MIDI devices

MidiTalk provides an ALSA sequencer client named "miditalk" but does not do anything to connect it up to your keyboards or synths. (In the future such a capability may be added..)

For now, you will need to connect up your external MIDI devices with the `aconnect` shell command. Assume you have an external CME X-Keys USB-MIDI keyboard connected to your system. You could connect it to MidiTalk with the following shown in Listing 1.4.

1.8 Another example using Gauge6 as a MIDI Volume control

One of the typical uses of MidiTalk is to map its widgets to send MIDI messages, and to update the state of widgets on receiving MIDI messages. In MIDI, CC7 (Control Change parameter number 7) is typically mapped to the Volume function of external gear. The script of Listing 1.5 shows how to map Gauge6 to CC7, and also how to update the label of the Gauge to something meaningful.

In the script, the `setLegend:` method of the Gauge sets a new display string for its label. The call to `setNeedsDisplay:` causes the Cocoa widget to re-draw itself to get this new label. The Gauge has a block that receives a new value when it changes, and we send a MIDI message to CC7 with this value.

To also make the Gauge reflect the value of any external control, we use the Dispatcher (`disp`) to watch for ControlChange messages. If there is a message on channel 2 to CC7, we use its value to update the Gauge.

1.9 Summary

This chapter has provided a brief introduction and tour of MidiTalk. We described the GUI controls, the pre-defined objects and the role of scripts. We described how scripts are found and run. Through a few examples we showed how StepTalk scripts can create Voice objects and Patterns. See if you can run the examples shown in this chapter.

Listing 1.2 Launching Oscillators with the Piano

```
"file: launching-oscillators.st"
"launch a one-shot note on piano keypress"
[|

makeNote:midiNote
  | env osc |
  env := MSKExpEnvelope alloc initWithCtx:ctx .
  env setOneshot: true .
  env setShottime: 0.1 .
  env setModel: envModel .
  env compile .

  osc := MSKGeneralOscillator alloc initWithCtx:ctx .
  osc setINote:midiNote .
  osc setModel: oscModel .
  osc setSEnvelope: env .
  osc compile .

  ctx addVoice:osc
!

main

  "retain the script in the environment"
  currentScript := self .

  envModel := MSKEnvelopeModel alloc init .
  oscModel := MSKOscillatorModel alloc init .

  self makeNote:60 .
  piano onNoteOn: [ :midiNote :vel |
    self makeNote:midiNote .
  ] .

]
```

Listing 1.3 Pattern and Lamps

```
"file: pattern-lamps.st"

pat1 := Pattern alloc initWithName:'pattern1' .
pat1
  sync: #downbeat ;
  play: [ lamp4 off. lamp1 on. ] ;
  sync: #beat ;
  play: [ lamp1 off. lamp2 on. ] ;
  sync: #beat ;
  play: [ lamp2 off. lamp3 on. ] ;
  sync: #beat;
  play: [ lamp3 off. lamp4 on. ] .

sched addLaunch:pat1 .
```

Listing 1.4 Launching the program

```
$ openapp MidiTalk &
$ sleep 1       # wait for MidiTalk to open its devices
$ aconnect Xkeys miditalk
```

Listing 1.5 Linking Gauge6 to CC7

```
"file: controlchange7-example.st"
"This example sends and receives CC7 on Gauge6"
[|

main
  currentScript := self .

  VOLUME := 7 . "the MIDI controller for VOLUME"

  "set the label of gauge6 and update the GUI"
  gauge6
    setLegend:'volume' ;
    setNeedsDisplay: true .

  gauge6 onChange: [ :value |
    evt := ASKSeqEvent eventWithControlChange:VOLUME val:value chan:2 .
    evt setSubs .
    evt setDirect .
    seq output:evt .
  ] .

  disp onControlChange: [ :param :val :chan |
    ((2 == chan) and: (VOLUME == param)) ifTrue: [
      gauge6 setValue:val .
    ] .
  ] .
]
```

2 StepTalk

StepTalk is a small dialect of SmallTalk built on the GNUstep Objective-C runtime. It exposes select classes and objects of GNUstep applications as scriptable elements that can be combined in interesting ways. The language itself is very small, and it gains most of its functionality by the ability to manipulate objects of almost any Objective-C class. This makes it a very powerful tool for extending applications with highly customizable features.

StepTalk is an interpreted language for interacting with Objective-C objects. The StepTalk interpreter can be used in two different ways. The first way it can be used is as a standalone interpreter. This capability is provided by the `stexec` program. In this configuration StepTalk is the "main" program. It can import GNUstep classes and operate on almost any Objective-C class in Foundation or AppKit. In fact, scripts executed by `stexec` can be full-featured little programs that create AppKit windows and widgets. A good place to look for information about this use case is the Examples directory of the libs-steptalk project.

The other configuration is called "Application Scripting". In this configuration, there is an application with its own "main" and it is enhanced by adding scripting capability to it. The program may have a GUI and predefined application logic already. StepTalk scripting is used to allow users to add new behaviors to the program.

This is how MidiTalk is designed to operate. The MidiTalk "main" program sets up the GUI and MIDI and Audio interfaces and then receives the rest of its behavior through the loading of StepTalk scripts.

2.1 The Scripts Window and the Transcript

When StepTalk is added to an application as the "Application Scripting" component it brings along two windows and a few menu items with it. A desktop with the "Scripts" window and the "Transcript" window are shown in Figure 2.1. The two windows are shown above the MidiTalk main window.

As currently written, MidiTalk automatically opens these two windows, but they may be opened by using the subitems of the `Scripting` menu item.

Figure 2.1: Scripts Window and the Transcript

2.1.1 The Scripts Window

The main function of this window is to shown the available scripts found in the system. StepTalk scripts are discovered if they reside in any of the following directories. On Linux, these are:

- `MidiTalk.app/Scripts` (the application bundle itself)
- `~/GNUstep/Library/StepTalk/Scripts/MidiTalk/Scripts`
- `/usr/GNUstep/Library/StepTalk/Scripts/MidiTalk/Scripts`
- `~/GNUstep/Library/StepTalk/Scripts/General`

The lower portion of the Scripts Window (the white area) displays an associated “information” string for the selected script, if there is any. For a script named `foo.st`, if there is an assocated file named `foo.st.stinfo` in the same directory, then it is taken as the “info” file for the script.

There are a number of commands implemented in the Scrips Window.

Update List
: This will cause the system to scan the scripts directories and reconstruct the script list.

Reveal
: This will show the selected script in the context of the file system using the `GWorkspace` file manager (if `GWorkspace` is installed).

Edit Script
: This will open the script in `TextEdit` (if it is installed).

Edit Info
: This will open an editor for modifying the information in the `.stinfo` file.

Help
: Will open associated Help if that has been configured.

There is also a `[Run]` button. This will execute the currently selected script. Double-clicking a script will also execute it.

2.1.2 The Scripting Transcript Window

In StepTalk the `Transcript` object is where output strings are printed by programs.

With StepTalk, the Scripting Transcript can also be used execute StepTalk statements interactively. To do so, enter some text in the Transcript window. Then select it. Now execute either of the menu items “Do Selection (#e)” or “Do and Show Selection (#d)”. The first interprets the selection string, the second interprets the selection string and shows the value it produces.

To try this out, enter the following in the Transcript area:

```
Transcript showLine:Environment objectDictionary .
```

Then highlight the entire line, and press “#d” (where the # key is normally Left-Alt on Linux.) This will produce a listing of every key and object that is in the StepTalk environment. Interactive use of the StepTalk environment in this way is useful for probing the current state of things.

2.2 Script Syntax

A StepTalk script consists of a series of method declarations in one file. Method declarations occur between an opening symbol, “[|” (left-bracket, bar) and a closing symbol ”]” (right-bracket). Method declarations are separated with a “!” (bang). A script may define “global” variables whose scope extends across the methods of the script file. See Listing 2.1 for an outline of this syntax.

There is also an abbreviated version of a script that simply executes a series of statements and does not define a method. In that case, the opening and closing brackets are omitted and the statements are executed in turn. Listing 2.2 illustrates this form.

Listing 2.1 StepTalk Script File Syntax

```
[|  :global1 :global2 :global3

method1:arg1 arg2:arg2
  statment .
  statement
!

method2:arg1 arg2
  | local1 local2 |
  statement .
  statement .
  ^retval
]
```

Listing 2.2 Simple Script Syntax

```
statement .
statement .
statement .
```

The evaluation of a script creates a new object of class `STScriptObject`. If there is a method named `main`, it will be evaluated immediately after the script is loaded. It will respond to messages sent to the methods defined. This is true whether the script is invoked from StepTalk or Objective-C! (See Listing 2.3 for an example of a script with a `main` method.)

Listing 2.3 StepTalk script with a 'main' method

```
[|
main
  Transcript showLine:'main is executed when this script is loaded'
]
```

NOTE: In the context of the script, the value `self` refers to the script object.

NOTE: StepTalk scripts do not define new classes. A script is an unnamed object with a unique set of methods.

2.3 Blocks

In StepTalk, blocks are created by the compiler, and they are objects just like everything else.

A StepTalk block can respond to methods with the selectors listed below, in Listing 2.4. (See file `STBlock.h` for details.)

Listing 2.4 StepTalk Block Selectors

```
#value
#value:
#value:value:
#value:value:value:
```

These methods invoke blocks with zero, one, two or three arguments.

The following example script in Listing 2.5 illustrates creating a block with one argument and invoking it by sending it a message.

Listing 2.5 StepTalk invoking a Block

```
myblock := [ :amount |
  Transcript show:'my block received: ' ; showLine:amount .
]

myblock value:45 .
```

2.4 Special Symbols

There are a number of singleton objects in the system that are special.

- Application: the singleton instance of `NSApplication`
- Transcript: the Transcript console output. Class `STTranscript` in the `ApplicationScripting` directory of `libs-steptalk`.
- Enviroment: the global environment

2.5 Cumulative Loading of Scripts

A single StepTalk environment is created and exists for the duration of the execution of MidiTalk. As StepTalk scripts are loaded, their effect on the environment is cumulative. Sometimes this property can be useful.

Consider two separate scripts: one that configures the buttons to play an assortment of "FatBass" sounds from the Synth80 Synthesizer, and a mapping of the piano keyboard to send MIDI events to an external device. If these two capabilities were defined in separate scripts, then loading either the first script, or the second script, or both, would all be valid configurations.

2.6 Future Language Features

This section discusses some of the current limitations of the StepTalk environment and a few of the ways in which it might evolve in the future.

> NOTE: The current incarnation of MidiTalk is attempting to explore the best features of StepTalk as it is without adding anything to the language or existing implementation.

2.6.1 Including Scripts in other Scripts

At the moment, there is not a `loadScript` method (or similar) in the standard Environment. We anticipate adding one in the future. It will probably differentiate between loading a script at an absolute or relative path, and loading one by name in the standard search paths.

Having such a facility will make it more convenient to define libraries and modules of scripts.

2.6.2 Classes

StepTalk does not have the ability to define classes at the present time. StepTalk scripts are all of the class `STScriptObject`, but these differ from normal Objective-C classes in that each instance has its own method definitions.

We have no plans at the present time to extend StepTalk to classes.

3 MidiTalk API Reference

This chapter describes the main class and category interfaces for use in StepTalk scripts with MidiTalk. We define the interfaces of the pre-defined objects, as well as convenience categories for dealing with MIDI events. This subset of the complete Objective-C environment is what we call the MidiTalk API Reference.

3.1 Conventions

In this section we will use Objective-C syntax, with an informal addition for StepTalk blocks. Objects that emit messages implement a method with a name like `onEvent:`. Their argument is a StepTalk Block. To simplify show the arguments sent to the block, we will show it like this.

```
1 - (void) onEvent: [ :arg1 :arg2 | ... ];
```

This means that the `onEvent:` method takes a StepTalk block as an argument and that when it is invoked it will receive two arguments. The names of the arguments in the documentation will describe their purpose.

In a few cases, we will also show the Objective-C type of an argument if it is important for the discussion.

```
1 - (void) onEvent: [ :(int)arg1 :(NSObject*)arg2 | ... ];
```

Properties in Objective-C can be read and written directly. For a property named `foo`, the read accessor `foo` or `getFoo` will return its value. To set its value, the setter `setFoo:` would be used. StepTalk is smart enough to coerce a numeric type in StepTalk to the appropriate C type of the Objective-C interface.

3.2 Piano

The on-screen piano can send and receive messages. When a key is pressed or released, messages are sent to blocks through the `onNoteOn:` and `onNoteOff:` messages.

Listing 3.1 shows the interface for the piano. The piano keyboard has a property called `octave` that sets the bottom "C" of the keyboard. It can be read or written in StepTalk. The `velocity` property sets the velocity value sent with `onNoteOn:` and `onNoteOff:` events.

The keys of the piano can be manipulated through the `noteOn:` and `noteOff:` messages. This will show the corresponding key as depressed or raised.

Listing 3.1 MLPiano interface definition

```
1  @interface MLPiano : NSObject
2
3  @property (readwrite) int octave;
4  @property (readwrite) unsigned velocity;
5
6  - (void) noteOn:(int)midiNote;
7  - (void) noteOff:(int)midiNote;
8
9  - (void) onNoteOn: [ :midiNote :vel | ... ];
10 - (void) onNoteOff: [ :midiNote :vel | ... ];
11
12 @end
```

Examples of interacting with the piano in StepTalk syntax are shown below, in Listing 3.2.

Listing 3.2 MLPiano examples in StepTalk

```
1 piano setOctave:3 .
2 piano noteOff:45 .
3 piano onNoteOn: [ :midiNote :vel |
4   Transcript show:'piano: '; showLine:midiNote .
5 ] .
```

3.3 Dispatcher

The dispatcher is a pre-defined object that handles the MIDI messages sent to MidiTalk from external programs and devices (like MIDI keyboards). The dispatcher has been defined to make it easier to handle just the messages you are interested in and ignore the rest. For each MIDI message type, there is an event handler defined that calls a block. The interface definition is shown in Listing 3.3.

Listing 3.3 ASKSeqDispatcher interface definition

```
1  @interface ASKSeqDispatcher : NSObject
2
3  - (void) onNoteOn: [ :midiNote :vel :channel | ... ] .
4  - (void) onNoteOff: [ :midiNote :vel :channel | ... ] .
5  - (void) onKeyPressure: [ :midiNote :vel :channel | ... ] .
6  - (void) onControlChange: [ :param :value : channel | ... ];
7  - (void) onPgmChange: [ :value :channel | ... ];
8  - (void) onChanPress: [ :value :channel | ... ];
9  - (void) onPitchBend: [ :value :channel | ... ];
10
11 - (void) onAnyEvent: [ :(ASKSeqEvent*)evt | ... ];
12
13 - (void) onUsr1: [ :d0 :d1 :d2 | ... ];
14 - (void) onUsr2: [ :d0 :d1 :d2 | ... ];
15 @end
```

The `onAnyEvent` is special: if a block is registered, then *every* MIDI event received by MidiTalk will be passed to it. The object passed will be an `ASKSeqEvent*`.

The `onUsr1:` and `onUsr2:` methods are for user-defined event types. Most users will not need these.

An example of interacting with the dispatcher in StepTalk syntax is shown below in Listing 3.4.

3.4 ASKSeq

The sequencer interface is the MIDI-out interface of MidiTalk. After preparing a MIDI Event (class `ASKSeqEvent*`) it can be sent to the output queue with `output:` or `outputDirect:`. The `outputDirect:` method sends it immediately while the `output:` method sends it through the queuing system. For most applications `output:` should be used.

Listing 3.4 ASKSeqDispatcher examples in StepTalk

```
disp onNoteOn: [ :midiNote :vel :chan |
  lamp1 on.
].

disp onNoteOff: [ :midiNote :vel :chan |
  lamp1 off.
].
```

Listing 3.5 ASKSeq interface definition

```
@interface ASKSeq : NSObject

- (void) output:(ASKSeqEvent*) ev;
- (void) outputDirect:(ASKSeqEvent*) ev;
- (int) getClient;
- (int) getPort;
- (int) getQueue;

@end
```

A sequencer interface has three important identifiers associated with it, assigned by the ALSA (Advanced Linux Sound Architecture) system layer. The first is the `client` id. This is an integer in the range of 1 to 128 (or 256, on some systems) that uniquely identifies the sequencer client. In our case, the client identifies the MidiTalk program itself. The `getClient` method returns this id.

The second is the `port` id. A sequencer may have a number of ports. In MidiTalk there is a pre-defined port for use and the `getPort` method returns this id.

A sequencer may also have a number of queues associated with it. The MidiTalk pre-allocates a queue for normal use. The `getQueue` method returns the id of this pre-allocated queue.

An example of using the sequencer in StepTalk syntax is shown below in Listing 3.6.

3.5 ASKSeqEvent(additions)

Class `ASKSeqEvent` is the MIDI sequencer event type. MidiTalk provides convenience constructors to make creating them straightforward. These are shown in Listing 3.7.

Listing 3.6 ASKSeq examples in StepTalk

```
"create a note-on event"
evt := ASKSeqEvent eventWithNoteOn:45 vel:127 chan:0.
evt setSubs.
evt setDirect.

"send the event immediately to all subscribed listeners"
seq output:evt .
```

The first group (`eventWithXXX`, etc.) create MIDI event objects with numeric parameters given.

The next groups, “set destination” and “set queue” require some understanding of the ALSA sequencer interface. An event may be sent to all subscribers, all clients or a particular destination. The three methods `setSubs`, `setBroadcast` and `setDest:port:` configure an event for one of these destinations.

The next group sets the queuing policy for the event. Events sent through the ALSA system do so on a particular timing queue. A queue can have its own time base and time increment. The first method, `setDirect` specifies avoiding a queue all together. The second, `setScheduleTick:` schedules the event for delivery in the future at a “tick” time, which is related to the queue tempo. The size of the tick is defined by the queue. The third, `setScheduleReal` delivers a message at an absolut time specified in seconds and nanoseconds.

In practice, with MidiTalk, its enough to use the default queue `queue := seq getQueue` for all the queue arguments of the timing methods.

Listing 3.8 shows how these methods could be used in StepTalk to create and configure an event in various ways. All created events need to have a destination and a queue policy set.

There are 120 ticks per quarter note.

Another group of additions to `ASKSeqEvent` provide methods for querying the type and parsing the parameters of sequencer events. These are shown in Listing 3.9. The first group, predicates, are straightforward enough. They return `true` if the event is of the given type.

The second group helps to query the type of the event and parse its values, all in one fell swoop.

Listing 3.10 shows some examples of using the MIDI event parsers. The dispatcher’s `onAnyEvent:` method is used to receive the stream of all MIDI messages sent to MidiTalk. The parsers are executed in turn on the event, and if the event matches the type of the

Listing 3.7 ASKSeqEvent additions for creating MIDI events

```
1
2  @interface ASKSeqEvent(additions)
3
4  + (ASKSeqEvent*) eventWithNoteOn:note vel:vel chan:chan;
5  + (ASKSeqEvent*) eventWithNoteOff:note vel:vel chan:chan;
6  + (ASKSeqEvent*) eventWithKeyPressure:note vel:vel chan:chan;
7  + (ASKSeqEvent*) eventWithControlChange:param val:val chan:chan;
8  + (ASKSeqEvent*) eventWithPgmChange:val chan:chan;
9  + (ASKSeqEvent*) eventWithChanPressure:val chan:chan;
10 + (ASKSeqEvent*) eventWithPitchBend:val chan:chan;
11
12 + (ASKSeqEvent*) eventWithUsr1:d0 d1:d1 d2:d2;
13 + (ASKSeqEvent*) eventWithUsr2:d0 d1:d1 d2:d2;
14
15 // set destination
16
17 - (void) setSubs;
18 - (void) setBroadcast;
19 - (void) setDest:client port:port;
20
21 // set queue
22
23 - (void) setDirect;
24 - (void) setScheduleTick:queue isRelative:rel ttick:ttick;
25 - (void) setScheduleReal:queue isRelateive:rel sec:sec nsec:nsec;
26 @end
```

parser its associated block is executed. Parsers return a BOOL value (`true` , `false`) but the return type is ignored here.

3.6 Lamp

There are four pre-defined lamps in the GUI interface: `lamp1`, `lamp2`, `lamp3` and `lamp4`. A lamp may either be in the ON state (it is colored) or the OFF state (it is uncolored). The color of a lamp may be adjusted by setting the `color` property of the lamp.

The code below in Listing 3.12 shows the ON color of `lamp1` being set to BLUE and the lamp being turned on.

Listing 3.8 ASKSeqEvent creation and configuration in StepTalk

```
note := 62 .
vel  := 127 .
evt1 := ASKSeqEvent eventWithNoteOn:note vel:vel chan:0 .
evt1 setSubs . "send to all subscribers"
evt1 setDirect . "send immediately"

q := seq getQueue .   "is probably the value 0"

evt2 := ASKSeqEvent eventWithNoteOff:note vel:vel chan:0 .
evt2 setSubs
"schedule this event for 120 ticks in the future"
evt2 setScheduleTick:q isRelative:true ttick:120 .

"send and schedule the two events"
seq output:evt1 .
seq output:evt2 .
```

3.7 Gauge

A Gauge is a rotary control whose purpose is similar to a slider. A MidiTalk Gauge has an assocated label called its “legend”. Each Gauge also displays its current value in its center. The interface definition for a Gauge is shown in Listing 3.13.

The color property of the gauge sets the main color, and the application logic of the gauge itself darkens the color for the background of the dial. In addition to setting the “legend” with a string, the format of the number inside can be adjusted. The format is a C-style “printf” format string and it is applied to the value of the gauge before displaying it.

An example that configures gauge6 and ties it to the piano keyboard is shown in Listing 3.14. Note the call to `setNeedsDisplay` after adjusting the color property. This tells the Cocoa graphics system to re-draw itself.

3.8 Button

The MidiTalk Buttons are an extension of the standard Cocoa `NSButton`. In Cocoa, a button typically has a single action associated with it. In MidiTalk the DOWN action (`onNoteOn:`) is as important as the UP action (`onNoteOff:`). Also, we have designed these buttons so that they emit `onKeyPressure:` events as the mouse is dragged when

the button is down. The value of the key pressure is related to how far the mouse is dragged from its initial location. The interface for the MidiTalk Buttons is shown in Listing 3.15.

Buttons behave like one-key of the piano in some ways. They have an assigned `midiNote` that they emit as part of their `onNoteOn:` and `onNoteOff:` events. They also have an assigned `velocity` that is used in the same way.

The example in Listing 3.16 shows a complete script that configures `but1` to create a new note when depressed and to release it when the button is released. The button is also configured to modulate the “cents” property of the oscillator with the `onKeyPressure:` event. The key pressure value will be somewhere between 0 and 127. We multiple this value by 2 before applying it to the model to make the effect more pronounced. Try it out! Press the button, drag mouse, and here the real-time variation in the oscillator pitch.

3.9 Synth80

Synth80 is a companion program to MidiTalk. It is a synthesizer built on the same McLaren Synth Kit foundation that MidiTalk employs. See Figure 3.1 for an illustration of Synth80.

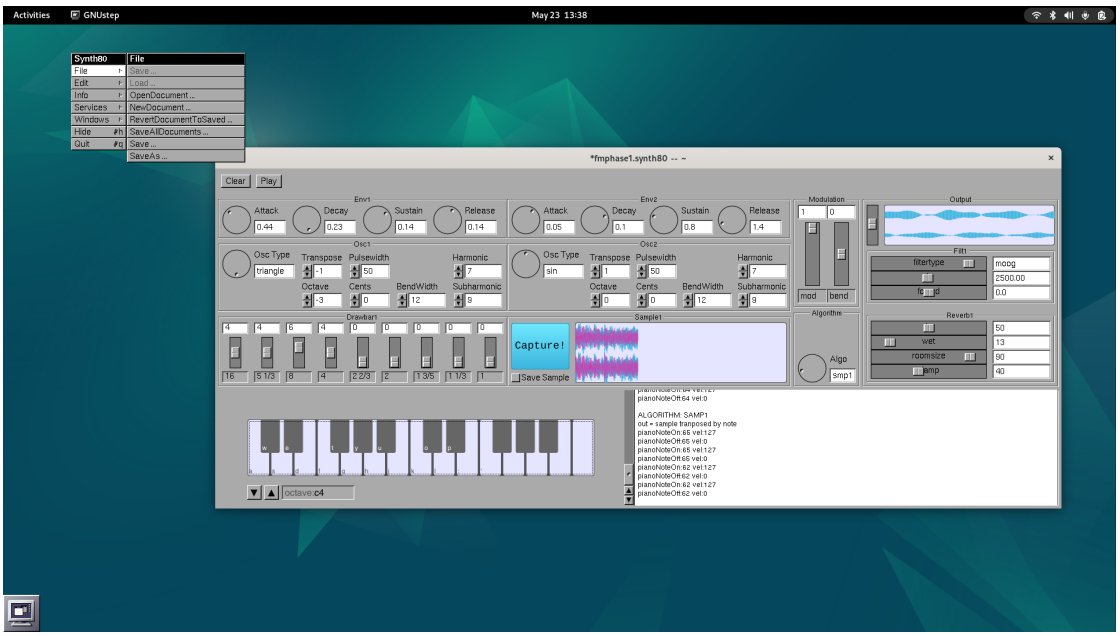


Figure 3.1: Synth80 Screenshot

MidiTalk includes the synthesizer engine of Synth80 directly in its code and allows Synth80 patches to be used directly. This makes it convenient to design sounds using

oscillators, envelopes and samples in Synth80 to create a library of sounds. Then, the resulting Patches can be used in MidiTalk as sound sources. This section describes the interface to the Synth80 module.

Listing 3.17 shows the interface for the class. When initialized, a Synth80 object must be attached to a context.

The `loadPatch:` method takes the name of a patch — omitting the path components and suffix of a patch. MidiTalk will look for patches in some pre-defined locations. These are similar to the search paths used for scripts. A Patch should end with the suffix ".synth80". The return value from the `loadPatch:` method is the full pathname of the patch loaded, or `nil` if none was found with the name given.

- `MidiTalk.app/Patches` (the application bundle itself)
- `~/GNUstep/Library/Patches/MidiTalk/Patches`
- `~/GNUstep/Library/Patches/`
- `/usr/GNUstep/Library/Patches/MidiTalk/Patches`

The `noteOn:` and `noteOff:` methods are self-explanatory. They start new notes and release playing ones. If a `noteOn:` is directed to a note that is already playing, the first one is terminated and the new note is sounded.

3.10 Metronome

The metronome provides a time-base and transport for MidiTalk. It is tightly integrated with the ALSA MIDI system for precise timing. There are only a few methods that make sense to access from MidiTalk. These are shown in Listing 3.19.

`num` the numerator of the time signature. Set this to "5" for a 5/4 time signature, for example.

`den` the denominator of the time signature. Set this to "8" for an eighth note to get one beat.

`measure`
the current measure count

`start`
start the metronome from measure zero

`stop`
stop the metronome

`kontinue`
continue the metronome from where it stopped

`setTempo`
adjust the tempo of the metronome. Note that this requires the use of an `NSErrorPtr`, which is introduce later in the book.

`setTimesig`
: set the time signature.

The callbacks of the metronome are not available for use by user. They are all used by the Scheduler, which is described in the next section.

3.11 Scheduler

The Scheduler drives the running of Patterns, and it is driven by the Metronome to get timing events, as well as the START, STOP and CONTINUE commands.

The Scheduler manages two types of Patterns

- Normal Patterns - they can be launched immediately or at measure zero
- Liveloop Patterns - these are Patterns, identified by name, that can be manipulated or replaced while running.

Listing 3.20 shows the interface to the Scheduler, `sched`. The following two sections describe its properties and methods.

3.11.1 Properties

`log` If set to `true` log the scheduler will log the execution of Patterns and Liveloops to the terminal (NSLog). This can be very helpful to understand the execution of Patterns.

`measure`
: This property gives the current measure number.

`beat`
: This property gives the current beat within a measure.

3.11.2 Launching Methods

`addLaunch:`
: this method adds a Pattern to the list of Patterns that are launched when the metronome begins at measure zero. This occurs when the `start` method of the metronome or the `[Start]` button is pressed.

  Use this when you want a specified Pattern to always play when you restart the Metronome.

`launch:`
: this method launches a Pattern immediately and lets it run to completion. (It is like spawning a Thread in a programming language.)

  Use this when you want to launch a rhythmic figure immediately as the result of some other event.

3.11.3 Liveloops Methods

The Liveloop capability makes it possible to substitute a Pattern with a different Pattern in such a way that there are no dropped beats. When a Pattern is added to the Scheduler as a Liveloop it is handled specially. The Pattern will play until its end, and then the Scheduler will restart it. This repetition will occur indefinitely. (I.e. it will stay "live" forever.)

If a Liveloop is replaced, by calling `setLiveloop:` with a new Pattern, then the current Pattern is allowed to play until the end. However, instead of repeating, the replacement Pattern is started. This is done in such a way that there are no dropped beats.

The `disableLiveloop:` and `enableLiveloop:` methods "mute" or "un-mute" a Liveloop. The "disable" method causes the named Liveloop to not be re-started at the end of its Pattern. The "enable" method will restart it from the beginning.

Liveloops are all identified by a NAME parameter. This is different from the name of a Pattern, since a Liveloop with one name might play different patterns at different times.

`setLiveooop`: If there is no Liveloop of the given name, then start the Pattern as the named Liveloop. If there is already one with that name, allow the current Pattern to play until its end and then swap the new Pattern in.

`disableLiveloop`: Allow the Pattern with the given Liveloop name to play until its end and do not restart it.

`enableLiveloop`: Restart the Pattern with the given Liveloop name.

3.11.4 Debugging Log

When the `log` property is set to `true` the Scheduler will log what it is doing to the terminal window (output of NSLog). An example is shown in Listing 3.21. This is output captured from running the demo in file "pattern-lamps.st" from the Introduction.

The front of the line including the date, the time and the process name and ID is all given by the GNUstep runtime system.

We first see the "metro start" message, which indicates that the `[Start]` button was pressed. Then we see the scheduler launching a pattern named "Pattern1". The value

in the angle brackets “<1>” is the “repeat spec” of the Pattern. It shows how many times the pattern repeats.

At each synchronization time point (due to a `sync:` or `ticks:` or `seconds:`) a line is printed. The first number is the current “tick” time of the metronome if it is a `sync:` or `ticks:` synchronization, or the realtime if it is a `seconds:` delay.

The next column is the current time in measures and beats. The value “7.1” is measure seven, beat 1 (beats start at 0).

The next column is the name given to the Pattern when it was created. This is the string given to the `Pattern initWithName:` method.

The final column shows the type of synchronization that has occured.

We then notice the “thread exited” message that says “Pattern1” has completed.

Finally, we see the “metro stop” message which is printed when the user Stops the Metronome.

3.12 Pattern

A Pattern is a little program that expresses the passage of time relative to the Metronome. The Pattern language is tiny: there are only a few statement types. These express the passage of time or annotate parts of the Pattern for repeating. Actions on objects in the MidiTalk system happen in blocks embedded in the Pattern. When the pattern reaches a block, it executes it immediately.

Patterns may call other patterns in a way that is reminiscent of the way songs have verses, chorsuses and bridges. Using sub-Patterns to construct other Patterns helps structure code for reuse.

SmallTalk “cascade” syntax helps to make the definition of a Pattern readable. Each method call adds a “statement” to the pattern or an “annotation” about the pattern. We’ll show how these concepts go together in this section.

Listing 3.22 shows the interface to the Pattern class. The initializer creates an empty Pattern with the given name. The name is used mainly for the debugging log.

3.12.1 Time-consuming Methods

The following methods are defined to time-consuming. The Scheduler executes the statements of a Pattern one by one. When the interpreter reaches a time-consuming method, it suspends the Pattern until the waiting condition is met.

`sync:`
: The `sync:` method takes an identifier as an argument that is called the "wait-channel". The wait-channel is a named event emitted by the metronome. Currently, the only named events are "downbeat" and "beat".

`ticks:`
: The `ticks:` method causes the Pattern to suspend until the given number of Metronome ticks elapses. There are 120 ticks per quarter note, and the absolution duration of a tick is a function of the tempo of the Metronome.

`seconds:`
: The `seconds:` method causes the Pattern to suspend for the given amount of Real-Time. The duration is independent of the tempo of the Metronome.

3.12.2 Executing Code

`play:`
: There is only one construct for executing code in the Pattern little-language. It is the execution of a block by the `play:` method. The block accepts no parameters. The statements of the block should not cause the program or thread to block. Some of the things that are safe to do are listed below.

- manipulate the values of models
- send MIDI events
- launch new Voices to be played
- write to the Transcript or the log
- change the values of lamps or controls

3.12.3 Sub-Pattern and Annotations

The following methods add structure to a Pattern.

`pat:`
: The `pat:` method adds a sub-Pattern to the target at the current position. When the Scheduler finds a sub-pattern in the executing of a Pattern, it immediately begins executing its statements in turn until a time-consuming method is found.

`intro:`
: The `intro:` method marks the current position as the end of the "introduction" of the pattern.

`repeat:`
: The `repeat:` method annotates the Pattern to repeat for the number of times given. A value of -1 means to repeat forever. The default value is 1 — a Pattern executes once and exits.

Note that the `repeat:` annotation is not a statement. It is not important where it is placed in the Pattern: it simply marks the entire pattern as repeating “4” times. However, we think it is nice to place it at the end.

We’ll explain the `intro:` and `repeat:` features with the following example that creates a pattern named “introAndRepeat” shown in Listing 3.23.

If this pattern was executed you would see the following on the Transcript Window as shown in Listing 3.24. The lines would be printed at the tempo of the Metronome because each statement is preceded by a `sync: #beat` method call.

Each `play:` block is preceded by a synchronization with the beat, so the pattern will execute at the tempo of the Metronome. The first two beats will execute and the lines “one” and “two” will be printed in the Transcript.

The `intro:` method marks this position as the end of the introduction in the Pattern. It will not be repeated as part of the repetition count.

Next, the two lines with the beat and the Transcript messages “three” and “four” are repeated four times. Finally, the Pattern exits.

Listing 3.9 ASKSeqEvent additions for parsing MIDI events

```
1
2  @interface ASKSeqEvent(additions)
3
4  // predicates
5
6  - (BOOL) isNoteOn;
7  - (BOOL) isNoteOff;
8  - (BOOL) isKeyPressure;
9  - (BOOL) isControlChange;
10 - (BOOL) isPgmChange;
11 - (BOOL) isChanPressure;
12 - (BOOL) isPitchBend;
13 - (BOOL) isUsr1;
14 - (BOOL) isUsr2;
15
16 // parsers
17
18 - (BOOL) parseNoteOn:  [ :note :vel :chan | ... ] ;
19 - (BOOL) parseNoteOff: [ :note :vel :chan | ... ] ;
20
21 - (BOOL) parseKeyPressure : [ :note :vel :chan | ... ];
22 - (BOOL) parseControlChange : [ :param :val :chan | ... ] ;
23 - (BOOL) parsePgmChange: [ :val :chan | ... ];
24 - (BOOL) parseChanPressure: [ :val :chan | ... ];
25 - (BOOL) parsePitchBend: [ :val :chan | ... ];
26 - (BOOL) parseUsr1: [ :d0 :d1 :d2 | ... ];
27 - (BOOL) parseUsr2: [ :d0 :d1 :d2 | ... ];
28
29 @end
```

Listing 3.10 ASKSeqEvent parser examples in StepTalk

```
disp onAnyEvent: [ evt |

  evt parseNoteOn: [ :note :vel :chan |
     "do something with the noteOn event"
  ] .

  evt parseNoteOff: [ :note :vel :chan |
     "do something with the noteOff event"
  ] .

  evt parseControLChange: [ :param :val :chan |
    "set gauge1 to the value of the control"
    gauge1 setValue:val .
  ] .
] .
```

Listing 3.11 MLLamp interface definition

```
@interface MLLamp : NSView

@property (readwrite) NSColor *color;

- (void) on;
- (void) off;

@end
```

Listing 3.12 MLLamp example in StepTalk

```
blue := NSColor blueColor .
lamp1 setColor:blue .
lamp1 on .
```

Listing 3.13 MLGauge interface definition

```
1
2  @interface MLGauge : NSView
3
4  @property (readwrite) NSColor *color;
5  @property (readwrite) NSString* legend;
6  @property (readwrite) NSString* format;
7
8  - (void) setValue:value;
9  - (double) value;
10
11 - (void) onChange: [ :value | ... ] ;
12
13 @end
```

Listing 3.14 MLGauge example in StepTalk

```
1  "file: gauge-example.st"
2  [|
3
4  main
5    gauge6 setLegend:'PianoInput' .
6    gauge6 setColor: NSColor redColor .
7    gauge6 setNeedsDisplay: true . "refresh the control"
8
9    piano onNoteOn: [ :midiNote :vel |
10     gauge6 setValue:midiNote .
11   ].
12 ]
13
```

Listing 3.15 MLExpressiveButton interface definition

```
1
2  @interface MLExpressiveButton : NSButton
3
4  @property (readwrite) int midiNote;
5  @property (readwrite) unsigned velocity;
6
7  - (void) noteOn:note vel:vel; // highlight if note matches midinote
8  - (void) noteOff:note vel:vel;
9
10 - (void) onNoteOn: [ :note :vel | ... ];
11 - (void) onNoteOff: [ :note :vel | ... ];
12 - (void) onKeyPressure: [ :note :vel | ... ];
13
14 @end
```

Listing 3.16 Button Example with modulation in StepTalk

```
"file: button-example.st"
"Make an expressive button play and manipulate a note"
[|

configureBut1
  "configure button1 to play and modulate a note"

  but1 setMidiNote:64 .

  but1 onNoteOn: [ :midiNote :vel |

    env1 := MSKExpEnvelope alloc initWithCtx:ctx .
    env1 setOneshot: NO .
    env1 setModel: envModel1 .
    env1 compile .

    osc1 := MSKGeneralOscillator alloc initWithCtx:ctx .
    osc1 setINote: midiNote .
    osc1 setModel: oscModel1 .
    osc1 setSEnvelope: env1 .
    osc1 compile .

    ctx addVoice: osc1 .
  ] .

  but1 onNoteOff: [ :midiNote :vel |
    env1 noteOff .
  ] .

  but1 onKeyPressure: [ :midiNote :vel |
    "modulate the cents property of the oscillator with the keypressure value"
    oscModel1 setCents: 2 * vel .
  ] .
!
main

  "Save the script in the environment"
  currentScript := self.

  envModel1 := MSKEnvelopeModel alloc init .
  oscModel1 := MSKOscillatorModel alloc init .

  "Set up Button One"
  self configureBut1 .
]
```

Listing 3.17 Synth80Synth interface definition

```
@interface Synth80Synth : NSObject

- (id) initWithCtx:(MSKContext*)ctx;

- (NSString*) loadPatch:(NSString*);
- (BOOL) noteOn:(unsigned)note vel:(unsigned)vel;
- (BOOL) noteOff:(unsigned)note vel:(unsigned)vel;
@end
```

Listing 3.18 Synth80 Example in StepTalk

```

```

Listing 3.19 MSKMetronome interface definition

```
@interface MSKMetronome : NSObject

@property (readonly) int num;
@property (readonly) int den;
@property (readonly) int measure;

- (void) start;
- (void) stop;
- (void) kontinue;

- (void) setTempo:(int)tempo error:(NSError**)error;
- (void) setTimesig:(int)num den:(int)den;
@end
```

Listing 3.20 MSKScheduler interface definition

```
@interface MSKScheduler : NSObject

@property (readwrite) BOOL log;

@property (readonly) int measure;
@property (readonly) int beat;

// launching patterns
- (void) addLaunch:(MSKPattern*)pat;
- (void) launch:(MSKPattern*)pat;

// liveloops
- (void) setLiveLoop(NSString*)name pat:(MSKPattern*)pat;
- (BOOL) disableLiveloop:(NSString*)name;
- (BOOL) enableLiveloop:(NSString*)name;

@end
```

Listing 3.21 Pattern Debugging Log

```
2024-12-06 07:53:34.462 MidiTalk[154602:154602] metro start
2024-12-06 07:53:34.462 MidiTalk[154602:154602] launch:pat:Pattern1<1>
2024-12-06 07:53:34.463 MidiTalk[154602:154602]       0   11.0 Pattern1 #downbeat
2024-12-06 07:53:34.963 MidiTalk[154602:156360]     120   11.1 Pattern1 #beat
2024-12-06 07:53:35.463 MidiTalk[154602:155761]     240   11.2 Pattern1 #beat
2024-12-06 07:53:35.963 MidiTalk[154602:155761]     360   11.3 Pattern1 #beat
2024-12-06 07:53:35.963 MidiTalk[154602:155761] thread exited
2024-12-06 07:53:47.615 MidiTalk[154602:154602] metro stop
```

Listing 3.22 Pattern interface definition

```
@interface Pattern : MSKPattern

- (id) initWithName:(NSString*)name;

// time-consuming operators
- (void) sync:(NSString*)waitchan;
- (void) ticks:(long)ticks;
- (void) seconds:(double)seconds;

// executing arbitary non-pattern code
- (void) play: [ ... ] ;

// pattern annotations
- (void) pat:(MSKPattern*)pat;
- (void) intro;
- (void) repeat:(int)repeatSpec.

@end
```

Listing 3.23 Pattern with an Introduction and Repeat

```
pat := Pattern alloc initWithName:'introAndRepeat'.
pat
  sync: #beat ;  play: [ Transcript showLine:'one'. ] ;
  sync: #beat ;  play: [ Transcript showLine:'two'. ] ;
  intro ;
  sync: #beat ;  play: [ Transcript showLine:'three'. ] ;
  sync: #beat ;  play: [ Transcript showLine:'four. ] ;
  repeat: 4 ;
```

Listing 3.24 Transcript output with Introduction and Repeat

```
one
two
three
four
three
four
three
four
three
four
```

4 The McLaren Synth Kit

The McLaren Synth Kit (abbreviated “MSK”) is the object-oriented audio processing layer in MidiTalk. With the McLaren Synth Kit audio algorithms are expressed as connected graphs of operators, called “Voices.” ” Voices can be assembled into complex graphs in either StepTalk or Objective-C. This chapter will give a short tour of how Voices work and how to use them to build audio pipelines.

4.1 What is a Voice?

In the McLaren Synth Kit, a “Voice” is the name given to the objects that produce audio samples. A Voice may be connected to other Voices to produce complex algorithms. The MSK library provides voices that are oscillators, envelopes and effects. A Voice implements the low-level operations on samples. Users of Voices are hidden from the details of samples and view audio operations as a connection of Voices.

4.2 Voices process samples

In the audio pipeline, it is useful to understand two rates associated with processing samples.

1) the audio sample rate
2) the period rate

The “audio sample rate” is that ultimate number of samples produced per second that is sent to the soundcard. For most applications, this will be 44000, 48000, or possibly 22500 on a lower-end CPU. The other rate is the “period rate”. This is that rate at which the soundcard asks for a chunk of new samples. It is related to the “period size.”

Figure 4.1 illustrates the audio buffer and the relationship between periods and samples. For each audio output, an audio buffer is allocated. It is divided up into some number of periods, each of which contains samples. The lower portion of the figure illustrates how sample transfer to the audio device is managed. For audio output, software writing samples uses the buffer corresponding to one period, and the device reading the samples

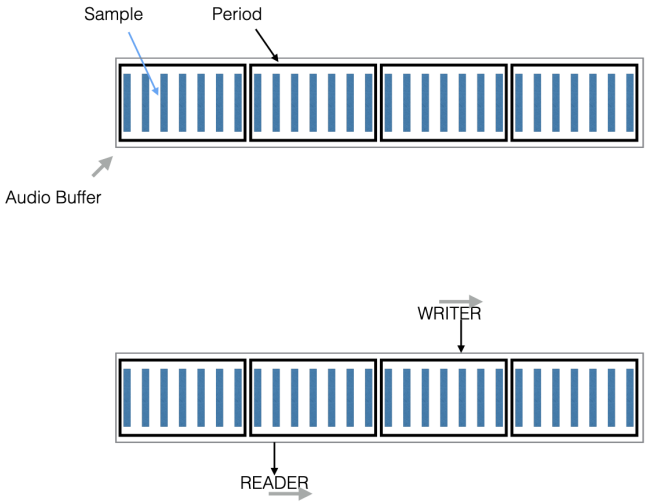


Figure 4.1: Audio Output Buffer and Periods

reads from a different period. This way the writer and reader do not interfere with one another.

Audio sample generation is achieved by the soundcard receiving small chunks of samples at regular intervals to fill the output buffer. The size of the chunk of samples is called the "period size".Typical period sizes are 1024, 512 or 256 samples. The smaller the period size, the greater the CPU load. Selecting a period size of 128 is possible but sometimes overloads a machine.

If the period size is 1024 and the sample rate is 44000, then the period rate is

```
44000 / 1024 = 42.96 periods per second
```

Why are these two rates important? Because of the overheads of Objective-C and StepTalk method calls, we want to avoid calling a complex method at the sample rate (44000 per second), but we are comfortable calling methods at the period rate (42.96 per second).

McLaren Synth Kit voices are designed to operate on a period-size chunk at a time. Each voice has an internal buffer that is large enough to hold one period's data (for instance 1024 samples). Voices perform different functions on sample buffers. By chaining together voices we can produce differrent audio effects.

So where does the period rate come in? Each time the soundcard asks for a new buffer of samples (1024 for instance) a Voice's `auRender` method is called. In other words, the

render method is called at the period rate. Voices are designed so that they access the parameters that control their behavior only once each period.

In the McLaren Synth Kit, parameters are held externally from the voices in an associated Model. By separating parameters from Voices and placing them in Models, one Model can be applied to many simultaneously sounding Voices.

4.3 A Sketch of a Voice

Let's look at some Objective-C like pseudo-code that describes a Voice that generates audio and has a single Model parameter. We'll call this a Source since it emits audio and has no audio input.

Listing 4.1 Objective-C Source Voice

```
@interface SourceModel : NSObject
@property (double) param1;
@end

@interface SourceVoice : MSKContextVoice
@property (MyModel*) model;
@end

@implementation SourceVoice
- (void) auRender {
  int p1 = model.param1;
  for (int i = 0; i < _persize; i++) {
    _buf[i] = fn(i, p1);
  }
}
@end
```

The `SourceModel` class defines a single property. It will be read by the audio render method of the `SourceVoice`.

In a Voice, the `auRender` method is called once each period. The instance variable `_persize` defines the period size. The render method obtains the parameter values it needs from the associated model *once* each time it is called, and then it generates `persize` output samples. Each sample produced is placed in the buffer named `_buf`. In this example, the `fn` function actually produces the sound samples, the Voice manages the allocation of the buffer and the calling of the `auRender` method.

Now let's consider the design of a second Voice-like class that also accepts a sample-rate input. We'll call this a filter, and it will have its own model class, just as the previous example.

This Filter Voice will also have a sample-rate input: a link to an instance of a `SourceVoice` class.

Listing 4.2 Objective-C Filter Voice

```
@interface MyFilterModel : NSObject
@property (float) param2;
@end

@interface MyFilterVoice : MSKContextVoice
@property (MyVoice*) input;
@property (MyFilterModel*) model;
@end

@implementation MyFilterVoice

- (BOOL) auRender {
  int p2 = model.param2;

  [input auRender];

  for (int i = 0; i < _persize; i++) {
    float in = input.buf[i];
    _buf[i] = fn2(in, p2);
  }
}
@end
```

In the `auRender` method of the Filter Voice, each time it is called it retrieves the needed parameter from the model. Then, before it accesses the sample values of the input Voice, it asks this Voice to render itself by calling the `auRender` method of the input. It then produces its output samples using both the values of the input voice and its own parameters.

This small example shows how the backward chaining of voices works. The `auRender` method of a voice causes its model parameters to be accessed and its input voices to be evaluated for the current period.

Using these classes to construct an audio chain in Objective-C would look like the following.

```
- (NSObject*) createGraph {
  MyModel *myModel = [[MyModel alloc] init];
  MyModel.param1 = 25;

  MyVoice *voice1 = [[MyVoice alloc] init];
  voice.model = myModel;

  MyFilterModel *myFilterModel = [[MyFilterModel alloc] init];
  myFilterModel.param2 = 42.0;

  MyFilterVoice *voice2 = [[MyFilterVoice alloc] init];
  voice2.model = myFilterModel;
  voice2.input = voice1;

  return voice2;
}
```

or in StepTalk it would look like the code below.

```
createGraph
  | myModel voice1 myFilterModel voice2 |
  myModel := MyModel alloc init .
  myModel setParam1:25 .

  voice1 := MyVoice alloc init .
  voice1 setModel:myModel .

  myFilterModel := MyFilterModel alloc init .
  myFilterModel setParam2:42.0 .

  voice2 := MyFilterVoice alloc init .
  voice2 setModel:myFilterModel .
  voice2 setInput:voice1 .

  ^voice2
```

Once an audio chain of voices has been created and parameterized, it can be added to an output context to cause it to be evaluated. The output audio context arranges to call the `auRender` method on each period, when the soundcard needs more samples.

The output context arranges for the newly added Voice to be appended ot the list of audio generators and does it in such a way that there are no audible “glitches” when this happens.

```
// in objecttve C
[ctx addVoice:voice2];

"in StepTalk"
ctx addVoice:voice2 .
```

4.4 Model Adjustment

The properties of a Model can be adjusted by GUI controls, MIDI messages or other means. They can be written and read, and they can even be connected to Cocoa controls with bindings.

The collection of pre-defined Voices in the McLaren Synth Kit have been carefully designed to respond to changes in the properties of models without glitches.

4.5 Voice Lifetime

Voices can be added to the Context for rendering and when they are no longer sounding they are removed from the Context and released to be reclaimed by the memory management system. An `MSKVoice` implements a simple protocol for keeping track of when a Voice is no longer active. This section will describe how it works.

Each Voice has a flag called `active` that is set to `YES` when the Voice is initialized. After the Context evaluates a voice by calling its `auRender` method it examines the `active` flag to see if it is ready to be released. Each type of Voice computes the value of its `active` flag in a manner that makes sense for its function.

Listing 4.3 is a sketch of the code for a very simple-minded Envelope generator. It has an associated model with a single paramter: `attackSamples` which specifies how many samples it takes the envelope to transition from 0.0 to 1.0. After counting up to 1.0, it begins its descent to 0.0. The `auRender` method computes the output buffer values for this envelope. It also includes a final calculation that sets the `active` flag to `NO` if the last sample in the buffer is 0. (This occurs when the envelope has completed its downward transition.)

An Envelope is not normally directly passed to the Context for playing. An Envelope is normally an input to a Voice like an Oscillator. An Oscillator remains active as long as its input Envelope is active in the McLaren Synth Kit.

A simple sinusoidal oscillator that maintains its `active` flag is sketched out in Listing 4.4. The oscillator's associated model has a single property called `frequency` that specifies its frequency. The oscillator voice has a model input, and also a envelope input. The

envelope modulates the amplitude of the oscillator and also determines the value of the `active` flag.

The oscillator maintains a state variable called `phi` that is the current angular phase of the oscillator. Each time the `auRender` method is called it computes the phase increment, `dphi`, as a function of the desired oscillator frequency and the contexts sample rate, `_rate`. It then fills the buffer with successive values of the `sin()` function multiplied by the envelope. After the buffer is filled, the `active` flag is set to that of the envelope. If the envelope is done producing samples, then so is the oscillator.

Listing 4.5 illustrates how the envelope and the oscillator are chained together and handed to the context for rendering. An envelope model is allocated and its parameter is set to 4400 - approximately a tenth of a second for a sample rate of 44000. The oscillator model is allocated and the frequency is set to that of A-440.

Next the envelope is allocated, and its model attached. The oscillator is allocated and its model is attached. The envelope is then attached to the oscillator. This connection creates the call-graph of the two voices. Lastly, the oscillator is added to the context. When the context evalutes the oscillator, the oscillator will evaluate the envelope. After the envelope is marked not active, so will be the oscillator. The context will release its reference to the oscillator and the entire graph can be reclaimed if there are no more references to it.

4.6 Voice and Model Lifetime

Each Voice in the McLaren Synth Kit has a property that marks it as `active`. When a voice is no longer active, it is no longer rendered and references to it are released. This makes it a candidate for deallocation when its reference count reaches zero.

Consider the audio chain we created earlier and added to the output context.

```
createGraph
  | voice1 voice2 |
  voice1 := MySourceVoice alloc init .
  ...
  voice2 := MyFilterVoice alloc init .
  voice2 setInput: voice1 .
  ...
  ctx addVoice: voice2 .
```

After this method exits, the context retains a reference to `voice2`, which retains a reference to `voice1`. When the active flag of `voice1` becomes `NO`, the context will release `voice2`, which will then release `voice1`. The entire chain of two voices will be reclaimed by the memory allocator.

4.7 A Real Example

Our previous examples illustrated some of the aspects of Voices including Models, the role of the `auRender` method and how Voice objects are reclaimed when they are done sounding. In this section, we will show code using the real McLaren Synth Kit oscillators and envelopes.

The method shown below creates an Envelope model and an Oscillator model and stores references to them in the global variables `envModel` and `oscModel`. We chose the exponential envelope model. Each of the four properties “attack”, “decay”, “sustain” and “release” are set in the one long statement that spans 5 lines.

This example shows the “cascade” operator (“;” - semicolon) which applies a series of method calls to one object.

The oscillator model is configured to set the oscillator type to the value corresponding to MSK_OSCILLATOR_TYPE_SQUARE.

The `makeModels` method will be called once in our script to establish the two models used for all notes.

```
makeModels
  envModel := MSKExpEnvelopeModel alloc init .
  envModel
    setAttack:0.01 ;
    setDecay:0.1 ;
    setSustain:0.9 ;
    setRelease:0.3 .

  oscModel := MSKOscillatorModel alloc init .
  oscModel
    setOsctype: MSK_OSCILLATOR_TYPE_SQUARE .
```

The method below creates an envelope Voice and an oscillator Voice that will be rendered in the audio output context, `ctx`. The initializer for each of these objects includes `ctx` as an argument.

Envelopes are somewhat special voices in the McLaren Synth Kit, because they are pure sources. Envelopes can be created as a “one shot” with a specified duration. Such an envelope will sound for the duration and then begin its release. If the `oneshot` property is set to `NO`, then at some later time the envelope can be told to begin its release by calling the `noteOff` method.

To be able to call the `noteOff` method on an envelope that is currently playing, we simply need to retain a reference to it. In the method below, `env` and `osc` are both global variables. Every time we call `makeNote` it will store a new `env` in the global environment.

```
makeNote:midiNote
  "env will be stored in a global variable"
  env := MSKExpEnvelope alloc initWithCtx:ctx .
  env setOneshot:NO .
  env setModel:oscModel .
  env compile .

  osc := MSKGeneralOscillator alloc initWithCtx:ctx .
  osc setINote:midiNote .
  osc setModel:oscModel .
  osc setSEnvelope:env .
  osc compile .

  ctx addVoice:osc .
```

The main method stitches it all together. First it calls the `makeModels` method to create the two models that will be used for all notes. Then it registers two callbacks on Button1 that create the audio graph to start a note, and to call the `noteOff` method on the envelope of a sounding note.

Take note that even thought the `onNoteOff` block tells the envelope to begin its release and the reference to it is released (by assigning `nil` to `env`) the envelope and the oscillator will play until the completion of the release period. By pressing Buttton1 rapidly in succession, it is possible to have many notes playing all at various phases of the release period.

```
main
  "create the models that are used for all notes"
  self makeModels .

  "register the callbacks to start and stop a note"
  but1 onNoteOn: [ :midiNote :vel |
    self makeNote:midiNote .
  ]

  but1 onNoteOff: [ :midiNote :vel |
    env noteOff .
    env := nil .
    osc := nil .
  ] .
```

4.8 The Graph Structure

It can help to visualize the structures being generated and connected to the Output Context. Figure 4.2 shows the object structure and references created in the previous examples with one envelope and one oscillator.

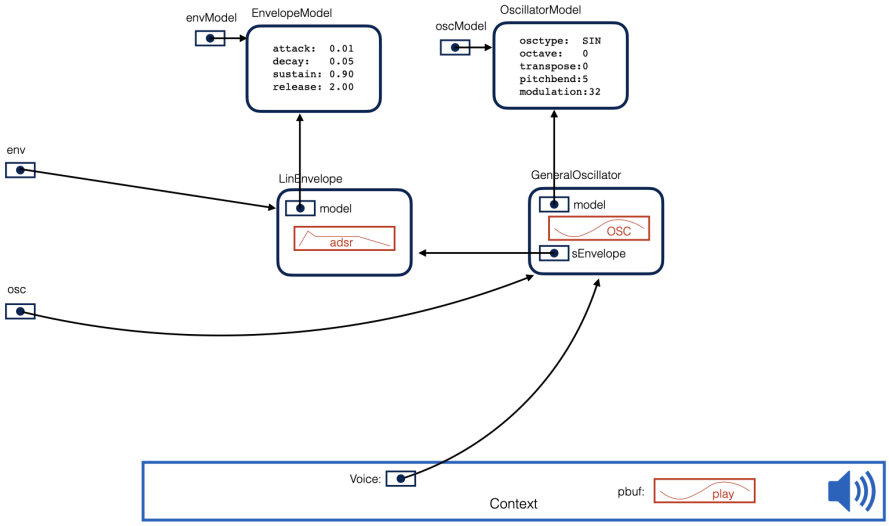


Figure 4.2: Oscillator and Envelope Graph Structure

The Output Context is shown at the bottom. It has one internal buffer named “pbuf” that is sized for one period. This is the buffer that is transferred to the output device, shown here as a speaker. When it needs to be filled, the Context traverses the graph beginning with its “Voice” value. It first encounters the oscillator, but before that can be rendered, its predecessor the envelope is rendered. In the course of rendering the envelope, its model is read. Then the oscillator can be rendered and the result will be added into the “pbuf” buffer for output.

After adding the graph to the context, we usually drop references to the objects in the graph. We usually do this by using temporary variables that are automatically cleared, or by explicitly setting some references to `nil`. This leaves the graph with only one reference: that of the Context.

Figure 4.3 illustrates the state of the playback graph just immediately after we add it to the context for playing.

Finally, when the envelope is finished playing it is marked inactive and then so is the oscillator. The Context will notice after rendering this last period that the oscillator is inactive and will release its reference to the graph. This situation is illustrated in

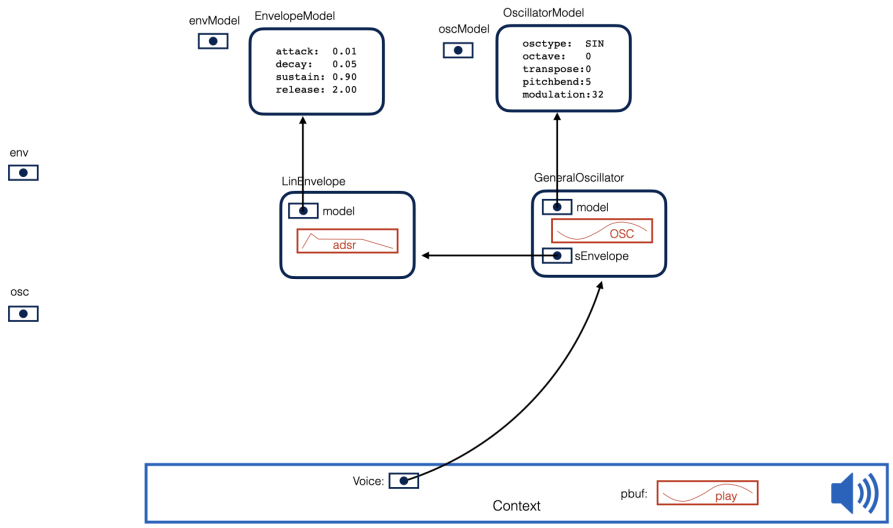


Figure 4.3: Graph Structure With References Released

Figure 4.4. The Context has released its reference to the graph and there are no references to it. Object reference counting will determine that there are no references to the objects and they will be reclaimed and handed back to the memory manager.

4.9 The ‘compile’ method

In the previous section we introduced the `compile` method but did not explain it. Now we will.

After an `MSKVoice` has been initialized and configured with its initial values and connections, the `compile` method **must** be called. Calling this method instructs the voice that the calling code is done configuring it and attaching inputs. Some voices use this point in the lifetime of a Voice to select an optimized implementation of the underlying algorithm.

For instance, an oscillator with an envelope needs to access the samples of its input envelope to create its output samples. If the oscillator is not connected to an envelope, an optimized algorithm that avoids accessing input samples can be used.

Examples of selecting an optimized algorithm are in the source code of the MSKGeneralOscillator, where C++ Templates are used to generate optimized loops.

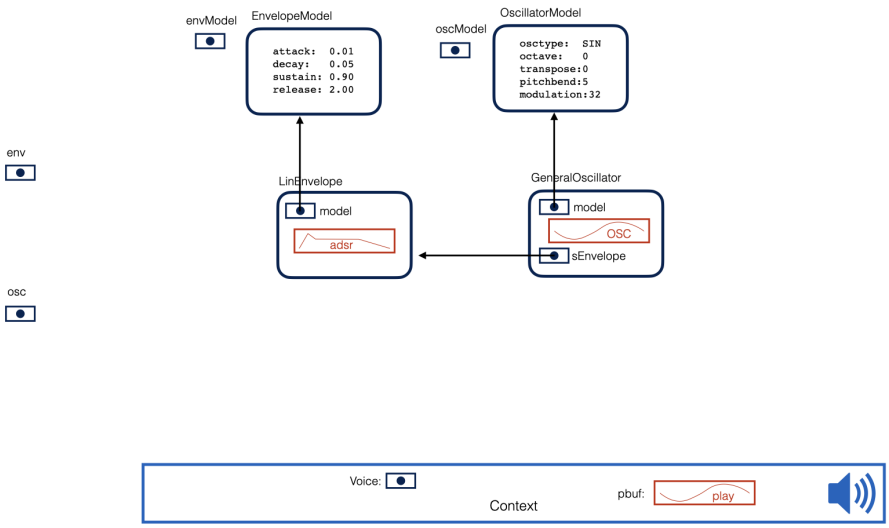


Figure 4.4: Graph Structure after it becomes Inactive

4.10 Keeping track of sounding Envelopes

In the previous example, there was a single button playing a note. Whenever the `onNoteOff` block was called, we could know that the global variable `env` held the last envelope that was created. This made it simple for our example, but does not extend well to polyphony.

In this section we will show a common idiom for keeping track of the envelopes of notes that are still playing. It is a slight extension to the simple case above. This time, we'll use the on-screen piano keyboard as an input device. We will also allocate an `NSMutableDictionary` to keep track of notes that are still sounding.

The `main` method is shown below. It creates the models, as before, and then it allocates an `NSMutableDictionary` for keeping track of envelopes that are still playing.

The piano callbacks are defined next. When an `onNoteOn` event sounds, the envelope for the corresponding `midiNote` is retrieved from the envelope dictionary, `envDict`. The `noteAbort` message is sent to this envelope which causes it to end (abruptly) in the current period. Next, we start the new note sounding and hang onto the envelope so that we can put a reference to it in `envDict`.

The `onNoteOff` block retrieves the corresponding envelope for the note playing and calls the `noteOff` method for it to begin its release.

```
main
  "make the models as before"
  self makeModels .

  "a dictionary to keep track of active envelopes"
  envDict := NSMutableDictionary alloc init .

  "configure the piano callbacks"
  piano onNoteOn: [ :midiNote :vel |
    env := envDict @ midiNote .
    env noteAbort .
    env := self makeNote:midiNote .
    envDict setObject:env forKey:midiNote .
  ] .

  piano onNoteOff: [ :midiNote :vel |
    env := envDict @ midiNote .
    env noteOff .
  ] .
```

For this example, we will re-write the `makeNote` method slightly to return the newly created envelope. We will also make `env` and `osc` local variables so that they do not pollute the global environment.

```
makeNote:midiNote
  | env osc |
  "env will be stored in a global variable"
  env := MSKExpEnvelope alloc initWithCtx:ctx .
  env setOneshot:NO .
  env setModel:oscModel .
  env compile .

  osc := MSKGeneralOscillator alloc initWithCtx:ctx .
  osc setINote:midiNote .
  osc setModel:oscModel .
  osc setSEnvelope:env .
  osc compile .

  ctx addVoice:osc .
  ^env
```

4.11 Playing notes from an external MIDI device

It is not difficult to modify the previous example to play notes arriving from an exteral MIDI source or keyboard. The `main` method below illustrates the difference. Instead of registering callback blocks for the piano, we would register callbacks from the MIDI dispatcher, `disp`. The listing below illustrates the change. Note that the `onNoteOn` and `onNoteOff` callback blocks from the dispatcher include a third parameter, `chan`. This is the MIDI channel (in the range 0..15) of the event.

```
main
  "make the models as before"
  self makeModels .

  "a dictionary to keep track of active envelopes"
  envDict := NSMutableDictionary alloc init .

  "configure the piano callbacks"
  disp onNoteOn: [ :midiNote :vel :chan |
    env := envDict @ midiNote .
    env noteAbort .
    env := self makeNote:midiNote .
    envDict setObject:env forKey:midiNote .
  ] .

  disp onNoteOff: [ :midiNote :vel :chan |
    env := envDict @ midiNote .
    env noteOff .
  ] .
!
```

The script can still use the `makeModels` and `makeNote` methods from the previous example.

To get this to play, you would need to use an external program like `aconnect` on the command line to connect the external device to MidiTalk. The listing below shows how to connect a CME XKey MIDI controller to MidiTalkk on the command line.

```
# this is a comment line
$ aconnect Xey25 miditalk
```

The other way to make connections is by client and port number. You can also see the current devices by using the `aconnect` command. For instance, on my system I see the following.

```
$ aconnect -i -o -l
client 0: 'System' [type=kernel]
    0 'Timer           '
    Connecting To: 144:0
    1 'Announce        '
    Connecting To: 144:0
client 14: 'Midi Through' [type=kernel]
    0 'Midi Through Port-0'
client 20: 'Xkey25' [type=kernel,card=1]
    0 'Xkey25 MIDI 1   '
client 128: 'miditalk' [type=user,pid=120370]
    0 '__port__        '
    1 'control         '
```

Instead of using the client names, I could connect “Xkeys” to MidiTalk with the following

```
$ aconnect 20:0 128:0
```

> NOTE: There are ways to make these connections directly in the McLaren Synth Kit using StepTalk and we’ll show how later in this document.

4.12 Recording with a Capture Context

The recording path works similarly to the playback path in that voices are evaluated in a graph from tail to head. However, instead of producing output samples, the objects ingest samples from the recording buffer of the Context. The `render` object of each method causes it to read its inputs and models, as before, but the result buffer of the tail object is ignored.

In this section we will discuss a paradigm for recording to a Sample data structure. The McLaren Synth Kit provides an object `MSKSample` that can hold a sample buffer of a predetermined length. There is also defined an `MSKSampleRecorder` that reads from its sample input and writes to an associated `MSKSample`. An `MSKSampleRecorder` responds to two methods.

- `recOn` - start recording
- `recOff` - stop recording and become inactive

The code in Listing 4.6 illustrates a method for allocating an `MSKSample` and recording to it with an `MSKSampleRecorder`. The method `recordToSample` allocates a Sample and a Recorder. Note that the Recorder is initialized with the `rec` context and not the playback

context here. The Recorder's input is set to the recording buffer of the `rec` context. The `setSample` method attaches the Sample buffer where the recording will go.

Button1 is configured to be the recording control. Press it, and a new recording is started, release it, and the recording is stopped.

Figure Figure 4.5 illustrates the graph created and handed over to the recording Context. Internally, the recording Context has a buffer called `rbuf` into which samples from the input are placed. This buffer is sized to be one period of the input device, as are all of the buffers in the objects initialized with the `rec` context.

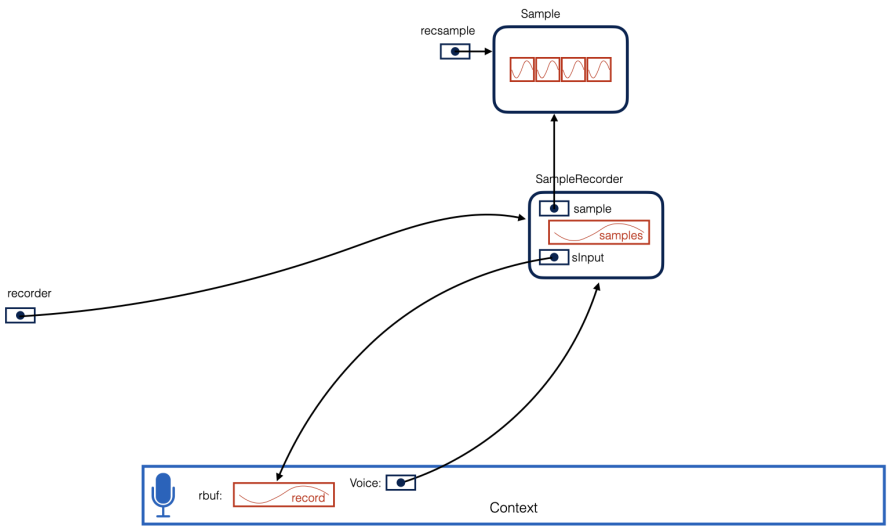


Figure 4.5: Recording from the Capture Context to a Sample

The `rec` context is always running and new samples are placed in `rbuf` every period.

The `MSKSample` may be sized arbitrarily. In our example, we make it long enough to hold 4 seconds worth of sample at 44.1 KHz. The SampleRecorder reads from its input and writes to the Sample, with each successive period following the previous. After `recOff` is called, the Recorder is marked as inactive and the `rec` Context releases its reference.

The significance of the context to which a Voice is attached is two-fold. When initialized, a Voice belonging to a given context will be sized to match the period of the attached device. Also, the objects of a context define a "synchronization domain" with respect to the device. Recording context voices are evaluated *after* the `rbuf` is filled. Playback context voices are evaluated *before* their results are added into the `pbuf` of their context.

4.13 A Recording Example

The script named “recording-example.st” in the program distribution illustrates a use of Sample Recording and Playback.

Button1 is configure to be the recorder control: press it to record, release it to stop recording. To make things interesting, the captured sample is scheduled to be played at every beat of a pattern, as shown in Listing 4.7.

When this script is loaded, it will play the sample named `playsample` on every beat. Press Button1 to record a new sample, and release it to stop recording. At this point, the sample will be played anew at each beat. You can experiment with this to make rhythmic patterns using your microphone as input. You can even record a new sample and the pattern will begin playing that.

Listing 4.3 A Simpified Envelope Generator

```
@interface EnvelopeModel : NSObject
@property (int) attackSamples
@end

@interface Envelope : MSKVoice {
@property (EnvelopeModel*) model;
@end

@implementation Envelope {
  int counter;
  float env; // initialized to zero
}

- (void) auRender {
  double slope;

  if (counter < model.attackSamples)
    incr = 1.0 / model.attackSamples;
  else
    incr = -1.0 / model.attackSamples;

  for (int i = 0; i < _persize; i++) {
    _buf[i] = val;
    val += incr;
    if (_buf[i] < 0) _buf[i] = 0;
  }
  if (_buf[_persize-1] == 0)
    _active = NO;
}
@end
```

Listing 4.4 Simplified Oscillator with an Envelope input

```
@interface OscillatorModel : NSObject
@property (double) frequency;
@end

@interface Oscillator : MSKVoice {
@property (OscillatorModel*) model;
@property (Envelope*) envelope;
@end

@implementation Oscillator {
 double _phi;
}

- (void) auRender {
  double dphi = (2 * M_PI) * (model.frequency) / _rate;

  [envelope auRender]; // evalute the envelope

  for (int i = 0; i < _persize; i++) {
    _buf[i] = sin(_phi) * envelope.buf[i];
    _phi += dphi;
  }

  _active = envelope.active;
}
@end
```

Listing 4.5 Simple Oscillator and Envelope Chain

```
1
2  EnvelopeModel *envModel = [[EnvelopeModel alloc] init];
3  envModel.attackSamples = 4400;
4
5  OscillatorModel *oscModel = [[OscillatorModel alloc] init];
6  oscModel.frequency = 440;
7
8  Envelope *env = [[Envelope alloc] init];
9  env.model = envModel;
10
11 Oscillator *osc = [[Oscillator alloc] init];
12 osc.model = oscModel;
13 osc.envelope = env;
14
15 [ctx addVoice:osc];
```

Listing 4.6 Recording from the Capture Context to a Sample

```
"file: recorder-example.st"

configureButton1
  but1 onNoteOn: [
    self recordToSample .
    recorder recOn .
  ] .

  but1 onNoteOff: [
    recorder recOff .
  ]
!

recordToSample
  recsample := MSKSample alloc initWithFrames:(44100*4) channels:2 .
  recorder := MSKSampleRecorder alloc initWithCtx:rec .

  recorder setSample: recsample .
  recorder setSInput: rec rbuf .
  recorder compile .

  rec addVoice: recorder .
!

main
  self configureButton1 .
```

Listing 4.7 Recording Playback with a Pattern

```
"file: recorder-example.st"
main
  currentScript := self .

  playsample := MSKSample alloc initWithFrame:(44100*4) channels:2 .

  self configureButton.

  pat1 := Pattern alloc initWithName:'pattern1' .
  pat1
    sync: #beat ;
    play: [ self playSavedSample . ] ;
    repeat: 100 .

  sched addLaunch:pat1 .
  metro start .
```

5 McLaren Synth Kit API Reference

The previous chapter introduced the concepts of the McLaren Synth Kit and its theory of operation. This chapter describes the interfaces to the classes the comprise the kit.

A note about property name prefixes. In the API interface to voices, you might notice that some properties start with “i” and some start with “s”. These are helpful reminders of the way in which these are used.

i an “i” property is a scalar, and it is used ONCE to set the “initial” value of some parameter in the calculation of a voice.

s an “s” property is something that produces “samples”. This means that such a property can point to a subclass of `MSKVoice` in the same context.

5.1 Envelopes

An Envelope generates a series of samples with values in the range [0 .. 1] that describes the shape of the amplitude of a sounding note. An envelope begins at amplitude 0 and eventually returns there. There are two types of envelopes in the basic kit: a Linear Envelope (class `MSKLinEnvelope`) and an Exponential Envelope (class `MSKExpEnvelope`). They are both subclasses of `MSKContextEnvelope` - the envelope base class. The two types of envelopes share the same model and respond to the same methods.

Figure 5.1 illustrates the two Envelope types along with their models. The lifetime of an envelope is described by four parameters. A fifth parameter, sensitivity, is described a little later.

When an Envelope is created, it immediately begins it ascent in the attack phase, and then continues with its release phase. It will continue sounding at the sustain level until it receives the `noteOff` method. When it does, it will begin its release phase. When the Envelope completes the release phase, it is marked inactive.

attack
: the time (in seconds) to reach the peak value

decay
: the time (in seconds) to reach the sustain value

sustain
: the amplitude of the sustain period

release
: the time (in seconds) to reach zero amplitude

sens
: a sensistivity value (in the range [0 .. 1]) that is used to map a velocity value to an initial gain value for an envelope

A Linear Envelope computes a linear transition function in its segments, and an Exponential Envelope computes an exponential function in its segments. An Exponential is more appropriate for modulating the loudness of a voice, while a Linear Envelope is more suited to varying other values such as modulation.

The `sens` (sensitivity) value and `iGainForVel:` methods work together and are optional. A value closer to 1.0 maps velocity more linearly to gain. A value of 0.5 maps the velocity range to a gain range of [0.5 .. 1.0], etc.

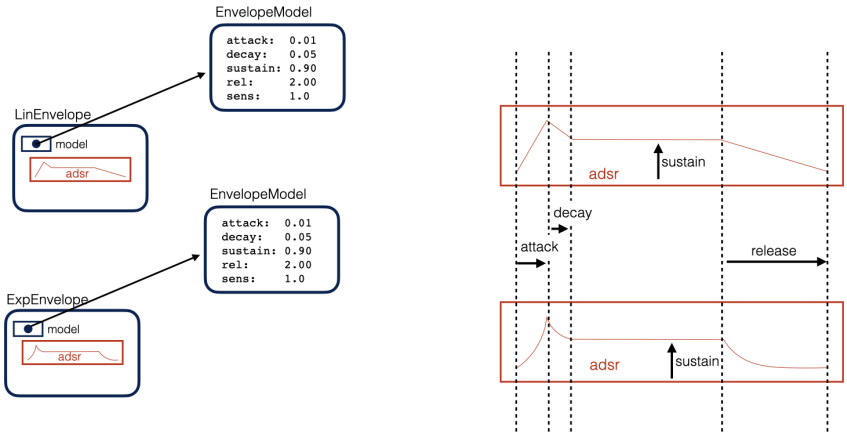


Figure 5.1: Envelopes Explanation

The interface to the Envelope Model and base class Envelope are shown in Listing 5.1. The Model is straightforward, with its parameters defining the shape of the envelope.

Common to all envelope instances is the `iGain` property, which sets the “initial gain” value. This is a multiplier applied to the shape of the envelope.

The rest of the Envelope interface is a little more complicated, because an Envelope can be used in a couple of different ways. These are the **oneshot** and **normal** modes. The **oneshot** mode is selected by setting the `oneshot` property to `true`.

In the **oneshot** mode, the duration of the attack/release/sustain phase is given by the `shottime` parameter. After the shottime has elapsed, the envelope begins its release phase and eventually becomes inactive. Use this capability to make a sound that fires for a predetermined amount of time.

In the **normal** mode, the Envelope continues it sustain phase until a `noteOff` message is sent to it. This mode is usually used in a `noteOff:` callback to release a sounding note.

In both modes, the receipt of a `noteAbort` message will cause the Envelope to transition to 0 within the current audio period and become inactive immediately. This feature is often used to silence an envelope immediately to begin the same pitch again with a new voice graph.

Listing 5.1 Envelope Model and Voice interface definition

```
// MODEL
@interface MSKEnvelopeModel
@property (readwrite) double attack;
@property (readwrite) double decay;
@property (readwrite) double sustain;
@property (readwrite) double rel;
@property (readwrite) double sens;

- (double) iGainForVel:(unsigned) vel;
@end

// ENVELOPE
@interface MSKContextEnvelope : MSKContextVoice
@property (readwrite) BOOL oneshot;
@property (readwrite) double shottime;

@property (readwrite) double iGain;

@property (readwrite) MSKEnvelopeModel *model;

- (id) initWithCtx:(MSKContext*)ctx;
- (BOOL) noteOff;
- (BOOL) noteAbort;
- (BOOL) noteReset:(int)idx;

@end
```

The `noteReset` method instructs the Envelope to restart its entire lifetime from the

beginning. (The `idx` parameter must match that of the Envelope.) This facility is not very well developed yet, but it can be used for some interesting effects. For instance, in a Pattern, each beat could can the pitch of a sounding oscillator and simultaneously call `noteReset` on its envelope, thus restarting the oscillator sound. It's a nice effect for some bass patterns.

5.2 Oscillators

Oscillators are the basic tone-generating voices in the McLaren Synth Kit. This section describes the built-in oscillator types and their configuration properties.

5.2.1 Oscillator Model and Modulation Model

Two models are common to all of the oscillator variants. They are the Oscillator Model and Modulation Model. Their interface is shown in Listing 5.2.

The `MSKOscillatorModel` defines some properties that are common to all oscillators, and a few properties that are specific to a kinds or modes. Property `osctype` defines the shape of the wave. Table 5.1 shows the predefined oscillator wave shapes. The properties `octave`, `transpose` and `cents` all affect the frequency of the oscillator, adjusting it up and down by octaves, semitones or cents (there are 100 cents per semitone).

A list of the properies and their descriptions is presented below.

osctype
: the oscillator type: SIN, SAW, SQUARE, TRIANGLE, REVSAW.

octave
: adjust the frequency up and down by an octave

transpose
: adjust the frequency up and down by a semitone

cents
: adjust the frequency up and down by a cent (there are 100 cents per semitone)

bendwidth
: the width of pitchbend in semitones. A value of 12 sets it to one octave.

pw the pulse width (duty-cycle) for a square wave. Values of 5 to 95 are permitted.

harmonic
: in a pure FM oscillator, the numerator multiplying the phase

subharmonic
: in a pure FM oscillator, the denominator dividing the phase

The `MSKModulationModel` has just two properties: a `modulation` amount and the `pitchbend` value. Oftentimes, these two are linked to the Modulation and Pitchbend wheels on a keyboard. The pitchbend value can range from -1 to 1 and is used with the `bendwidth` Oscillator Model parameter to determine the current bend value. The `modulation` value is used in different ways by the oscillators.

Listing 5.2 Oscillator Model and Modulation Model interface definition

```
// OSCILLATOR MODEL
@interface MSKOscillatorModel : NSObject
@property (readwrite) enum osctype;

@property (readwrite) int octave;
@property (readwrite) int transpose;
@property (readwrite) int cents;
@property (readwrite) int bendwidth;
@property (readwrite) int pw; // SQUARE WAVE

@property (readwrite) double harmonic;     // FM
@property (readwrite) double subharmonic;  // FM
@end

// MODULATION MODEL
@interface MSKModulationModel : NSObject
@property (readwrite) double modulation;
@property (readwrite) double pitchbend;
@end
```

Table 5.1: Oscillator Types

osctype	description
MSK_OSCILLATOR_TYPE_SIN	math sin() function
MSK_OSCILLATOR_TYPE_SAW	ramp from -1 to 1, then repeat
MSK_OSCILLATOR_TYPE_SQUARE	+1/-1 with duty cycle set by pw
MSK_OSCILLATOR_TYPE_TRIANGLE	ramp from -1 to 1, then 1 to -1
MSK_OSCILLATOR_TYPE_REVSAW	ramp from 1 to -1, then repeat

5.2.2 General Oscillator

Figure 5.2 illustrates the General Oscillator. An instance of the oscillator is parameterized by two models: an oscillator model and a modulation model. It has a single “s” input,

which is customarily an envelope. The function defining the output of the oscillator is shown in the figure. Each output sample, $out[i]$, is the product of an envelope sample and the $sin()$ of the current phase at the time t_i corresponding to the sample number i. The $sin()$ function is shown, but it generalizes to the other oscillator types listed.

The frequency, f, of the oscillator gives the angular velocity, ω.

$$\omega = 2\pi f$$

It is determined by a combination of the initial note (`iNote`), and the octave, transpose and cents parameters of the model.

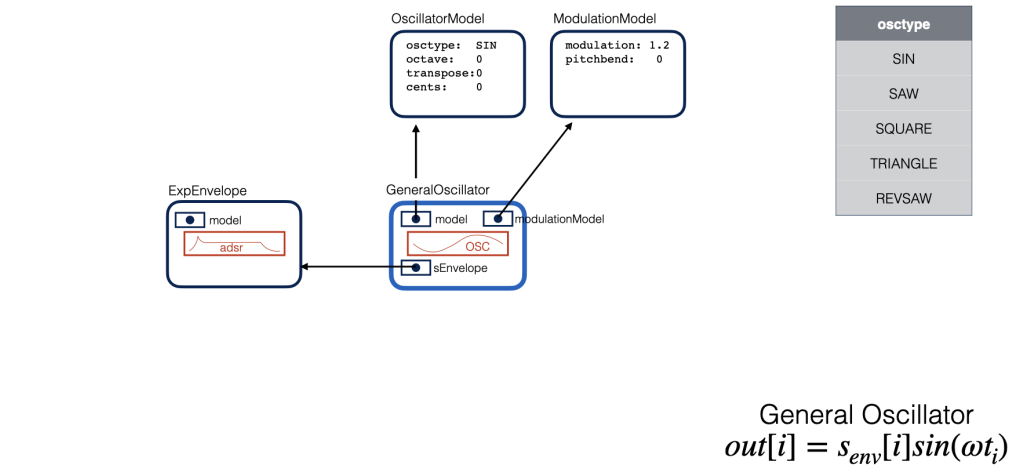


Figure 5.2: General Oscillator Explanation

The interface to the General Oscillator is described in Listing 5.3. The properties are as follows.

iNote
: The initial value used to compute the note frequency.

model
: The Oscillator Model.

modulationModel
: The Modulation Model.

sEnvelope
A voice providing samples for the envelope input.

If this input is omitted (set to nil) the envelope is assumed to be the constant “1” and the oscillator remains active forever.

Listing 5.3 General Oscillator interface definition

```
1 @interface MSKGeneralOscillator : MSKContextVoice
2
3 @property (readwrite) unsigned iNote;
4 @property (readwrite) MSKOscillatorModel *model;
5 @property (readwrite) MSKModulationModel *modulationModel;
6 @property (readwrite) MSKContextEnvelope *sEnvelope;
7
8 - (id) initWithCtx:(MSKContext*)ctx;
9 @end
```

5.2.3 Ring Oscillator Topology Example

A Ring-oscillator is constructed with two oscillators where the output is the product of the two. The way to construct a ring-oscillator is to chain the output of one oscillator into the envelope of another. Figure 5.3 illustrates how to do this.

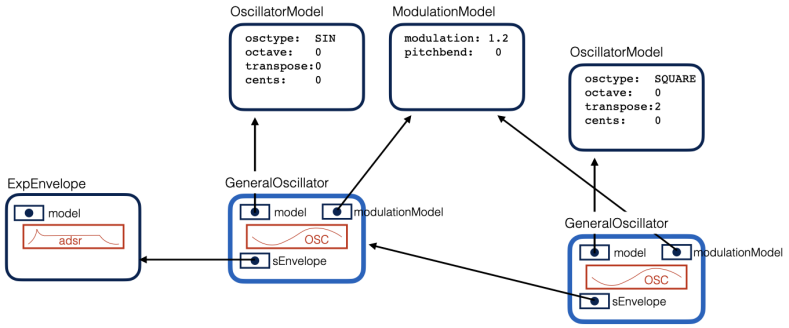
In the figure, there is one envelope shaping the note, and two oscillators. The two oscillators shown here have individual Oscillator Models, but share a Modulation Model to retrieve pitchbend information. Interesting effects are heard when the properties of the oscillators differ only slightly. In the example, we show a SIN and a SQUARE oscillator type, with the pitch of the second slightly offset by setting the `transpose` parameter.

In general, setting the envelope input of an oscillator to another oscillator is the way to form the product. The way to form the sum of two oscillators is to add both into the Context.

5.2.4 Phase Distortion Oscillator

The Phase Distortion Oscillator uses a second sample input, called `sPhaseenvelope` to offset (or distort) the pure phase calculation of a simple oscillator. Figure 5.4 illustrates the properties and models associated with the Phase Distortion Oscillator.

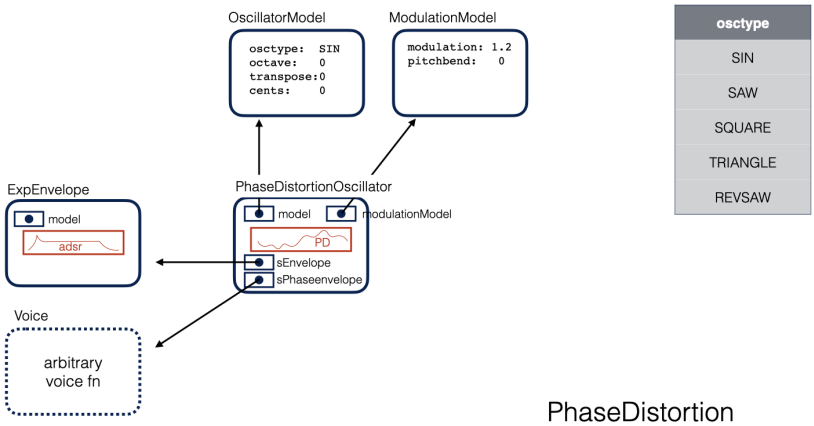
The function defining the output of the oscillator is shown in the figure. Each output sample applies the waveform function (shown as $sin()$ here) to the phase, ωt_i, offset with



Ring Oscillator Topology

$$out[i] = s_{env}[i] * sin(\omega_1 t_i) * sin(\omega_2 t_i)$$

Figure 5.3: Ring Oscillator Topology



PhaseDistortion

$$out[i] = s_{env}[i] \times sin(\omega t_i + mod \times s_{phase}[i])$$

Figure 5.4: Phase Distortion Oscillator Explanation

the phase envelope input. The magnitude of the offset is controlled by the `modulation` property of an associated Modulation Model. The $sin()$ function generalizes to the other oscillator types listed.

Listing 5.4 Phase Distortion Oscillator interface definition

```
1  @interface MSKPhaseDistortionOscillator : MSKContextVoice
2
3  @property (readwrite) unsigned iNote;
4  @property (readwrite) MSKOscillatorModel *model;
5  @property (readwrite) MSKModulationModel *modulationModel;
6  @property (readwrite) MSKContextEnvelope *sEnvelope;
7  @property (readwrite) MSKContextEnvelope *sPhaseenvelope;
8
9  - (id) initWithCtx:(MSKContext*)ctx;
10 @end
```

The interface to the Phase Distortion Oscillator is described in Listing 5.4. The properties are as follows.

iNote
: The initial value used to compute the note frequency.

model
: The Oscillator Model.

modulationModel
: The Modulation Model.

sEnvelope
: A voice providing samples for the envelope input.

  If this input is omitted (set to nil) the envelope is assumed to be the constant “1” and the oscillator remains active forever.

sPhaseenvelope
: A voice providing samples that distort the phase of the oscillator.

  If this input is omitted (set to nil) the envelope is assumed to be the constant “1” and the oscillator remains active forever.

5.2.5 FM Phase Envelope Oscillator

The FM Phase Distortion Oscillator modulates the phase of a primary oscillator with a secondary oscillator whose frequency is a pure harmonic of the first oscillator. The partial function

$$sin(\frac{harm}{sub} \times \omega t_i)$$

is computed using the `harmonic` and `subharmonic` properties of the attached Oscillator Model.

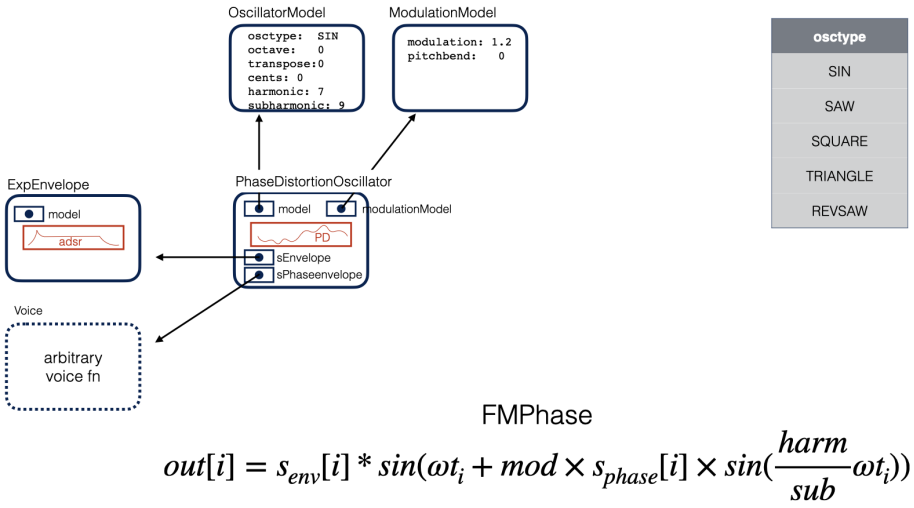
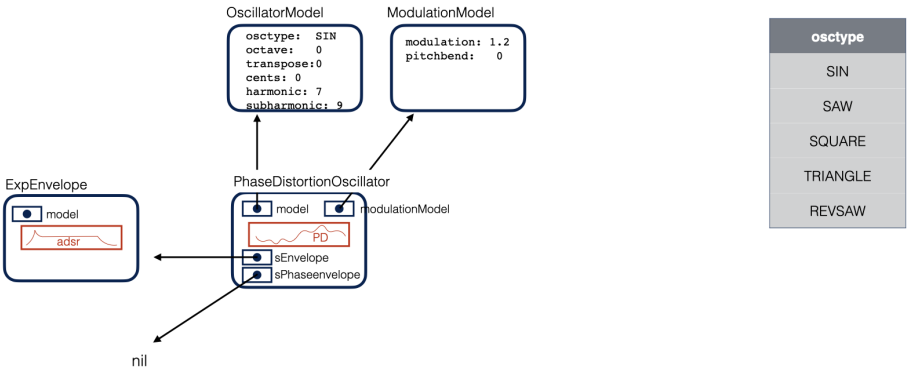


Figure 5.5: FM Phase Envelope Oscillator Explanation

If the `sPhaseenvelope` input is omitted, then the result is as shown in Figure 5.6. Here we can see that without the `sPhaseenvelope` input, the output formula simplifies to phase offset by a harmonic of the root phase.

Listing 5.5 FM Phase Distortion Oscillator interface definition

```
1  @interface MSKFMPhaseEnvelopeOscillator : MSKContextVoice
2
3  @property (readwrite) unsigned iNote;
4  @property (readwrite) MSKOscillatorModel *model;
5  @property (readwrite) MSKModulationModel *modulationModel;
6  @property (readwrite) MSKContextEnvelope *sEnvelope;
7  @property (readwrite) MSKContextEnvelope *sPhaseenvelope;
8
9  - (id) initWithCtx:(MSKContext*)ctx;
10 @end
```



Pure FM Topology

$$out[i] = s_{env}[i] * sin(\omega t_i + mod \times sin(\frac{harm}{sub}\omega t_i))$$

Figure 5.6: Pure FM Oscillator Example

The interface to the FM Phase Envelope Oscillator is described in Listing 5.5. The properties are as follows.

iNote
The initial value used to compute the note frequency.

model
The Oscillator Model.

In addition to the usual properties, this oscillator also uses the `harmonic` and `subharmonic` properties of the model.

modulationModel
The Modulation Model.

sEnvelope
A voice providing samples for the envelope input.

If this input is omitted (set to nil) the envelope is assumed to be the constant “1” and the oscillator remains active forever.

sPhaseenvelope
A voice providing samples that multiply the phase offset of the FM oscillator.

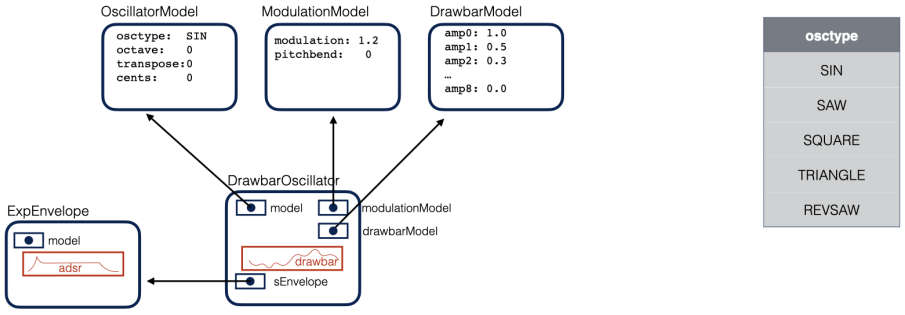
5.2.6 FM Phase Envelope Example

An example demonstrating some of the sounds of this oscillator can be found in the file “fmphase-example.st” accompanying the program. This example sets the gauges to adjust the envelope and harmonic parameters. Buttons one and two are used to disable or enable the `sPhaseenvelope` input so you can hear how the inclusion of an additional input oscillator adds complexity to the output.

One interesting thing about the example is the way a linear envelope is used to vary the modulation of the FM phase value. When trying it out, set the “attack” time to something long like 1 second and listen to the timbre change across the note’s duration.

5.2.7 Drawbar Oscillator

The final oscillator in the collection, is the Drawbar Oscillator. This oscillator produces a series of harmonics, all multiples of the base frequency. The amplitude of each is controlled by an associated property. Figure 5.7 illustrates a Drawbar Oscillator along with its Drawbar Model controlling the amplitudes. The output function is shown in the figure.



Drawbar

$$out[i] = s_{env}[i] \times (a_0 sin(\omega t_i) + a_1 sin(3\omega t_i) + a_2 sin(2\omega t_i) + a_3 sin(4\omega t_i) \ldots)$$

Figure 5.7: Drawbar Oscillator Explanation

The interface to the Drawbar Model and Drawbar Oscillator is shown in Listing 5.6. The properties `amp0` through `amp8` set the amplitudes of up to 8 harmonices. The value of the propert `overtones` determines how many harmonics are actually used. For example,

if it is set to 3, then only the first three harmonics will be evaluated. Setting this lower than 9 can help optimize evaluation.

Listing 5.6 Drawbar Model and Oscillator interface definition

```
@interface MSKDrawbarModel : NSObject
@property (readwrite) double amp0;
@property (readwrite) double amp1;
@property (readwrite) double amp2;
@property (readwrite) double amp3;
@property (readwrite) double amp4;
@property (readwrite) double amp5;
@property (readwrite) double amp6;
@property (readwrite) double amp7;
@property (readwrite) double amp8;

@property (readwrite) int overtones;
@end

@interface MSKDrawbarModel : MSKContextVoice
@property (readwrite) unsigned iNote;
@property (readwrite) MSKOscillatorModel *model;
@property (readwrite) MSKModulationModel *modulationModel;
@property (readwrite) MSKDrawbarModel    *drawbarModel;
@property (readwrite) MSKContextEnvelope *sEnvelope;

- (id) initWithCtx:(MSKContext*)ctx;
@end
```

The interface to the Drawbar Oscillator is described in Listing 5.6. The properties are as follows.

iNote
: The initial value used to compute the note frequency.

model
: The Oscillator Model.

modulationModel
: The Modulation Model.

drawbarModel
: The Drawbar Model which provides the amplitudes of each of the harmonics.

sEnvelope
: A voice providing samples for the envelope input.

If this input is omitted (set to nil) the envelope is assumed to be the constant “1” and the oscillator remains active forever.

5.3 Further References

For more information, the developer guide has detailed information about the components of the McLaren Synth Kit.

- https://mclarenlabs.github.io/libs-mclaren-alpha/

6 Devices and Connections

MidiTalk defines one sequencer interface (`seq`) and one audio output context (`ctx`), but it is possible to define other devices in your scripts. For instance, you may have two external sound cards and wish to send different sounds to each of them. Or you may want to define a new sequencer interface, or manage MIDI routing in StepTalk. This section will show you how.

6.1 The NSErrorPtr - a neccessary evil

A common idiom in Objective-C is to provide an extra parameter to a method that can hold an error. In C, the `&` (address-of) operator makes it easy to use an argument as an "out" value.

```
NSError *err =  nil;
[object method:foo error:&err];
```

If an error occurs in the method, then `err` can be written with a new value and it can be checked and easily logged or handled.

```
if (err != nil) {
  NSLog(@"There was an error:%@", err);
}
```

This construct doesn't translate well to StepTalk. To overcome this limitation, MidiTalk has provided a special object, called `NSErrorPtr` that can hold a pointer to an `NSError`. It can be used like this to rewrite the example above in StepTalk.

```
errp := NSErrorPtr alloc init .
object method:foo error:errp;

errp hasValue
  ifTrue:[
    errDescription := errp err description .
    Transcript show:'there was an error: ' ; showLine:errDescription .
```

```
  ] .
.
```

An `NSErrorPtr` object has two methods of interest to StepTalk code.

1. `hasValue`
2. `err`

The `hasValue` method is a predicate that returns YES if the `NSErrorPtr` contains a value. The `err` method returns the actual `NSError` object reference.

6.2 Making TO and FROM MIDI connections

It is nice to be able to construct scripts that set-up the desired connections TO and FROM your attached MIDI devices. Recall that a MIDI device is identified in the ALSA Sequencer interface by a client-id and port-id. The client-id can change depending on which devices are currently in use, so if y our product names are unique, it is convenient to be able to connect them by name.

To find out the name of your connected devices, use the `aconnect` terminal command.

```
$ aconnect -i -o -l
```

On our system, we have a CME XKeys device and its name is “Xkey25”. The script in Listing 6.1 illustrates a StepTalk method for connecting to receive events FROM the “Xkey25” device. This method has been written with enough error checking to identify what it is doing.

The first thing the method does is to look up the MIDI device by name with the `parseAddress:` method of the `seq`. If the named device is found, an address is returned. An address consists of a “client” and a “port”. If the returned address is `nil`, then the device was not found.

The `connectFrom:` method creates a connection FROM the address to our sequencer interface, `seq`. If it failes, it could be because the device is already connected. Our script simply notes this fact so that if it is run every time the script is loaded, there is no harm done.

> **NOTE:** To change the direction of the connection from FROM to TO, replace the call to the `connectFrom:error:` method with `connectTo:error:`.

6.3 Creating a new Sequencer device

There is nothing special about the default `seq` created for you by MidiTalk. You might have an application where you want to have two separate devices. It is easy in MidiTalk, but you will be entering "unsafe" territory. So follow the guidelines here and you'll be fine.

The first step to creating an `ASKSeq` is to create an `ASKSeqOptions` and configure it as you desire. An `ASKSeqOptions` is a simple Objective-C object with character strings and integer values. Interestly, the StepTalk interpreter knows how to cast an 'NSString*' into a 'char*', but it does so by reference to the underlying string rather than copying. (See file `NSInvocation+additions.m` if interested.) So be careful.

Listing 6.2 shows the `createSeqWithName:` method that builds a new `ASKSeq` and puts it in the Environment with the identifier `seq2`. The first thing it does is to create a sequencer "options" object to configure. Line 7, where the 'sequencer_name' field is set is especially interesting. In Objective C, the field is defined like this.

```
char *sequencer_name;
```

StepTalk recognizes the method name "setSequencer_name:" as a setter for the field "sequencer_name". StepTalk also coerces an `NSString*` into a `char*`, so that the setter

```
(void) setSequencer_name:(NSString*);
```

can set the value of the `char *sequencer_name` field.

A note about safety: at this point, the `char *sequencer_name` field is pointing to the bytes of the `name` parameter. It is important that `name` is retained until the `ASKSeq` is initialized, because at this time the ALSA system copies the bytes of the sequencer name into its own storage. Our `createSeqWithName` method does just that!

It is enough to create an `ASKSeq` for sending MIDI events, but if we want to receive them and handle them, the we want a dispatcher too. The method also creates `disp2` for dispatching received MIDI events.

If our script simply called `createSeqWithName:` every time it was loaded, you would observe a new sequencer at every load. To avoid this, our `main` method in the example looks in the Environment for an object named `seq2`. If one is found already, then the script does not create a new one.

6.4 Creating a new Output Context

Hardware is inexpensive these days, and a very inexpensive USB DAC can be had for just a few dollars. Good ones are slightly more. In our lab, we have a mixer, a Yamaha MG10XU that has its own USB audio input. What if we want to create a specific audio context just for this mixer? That way we could play sounds on the default audio output and the MG10 simultaneously!

Listing 6.3 shows the code required to create an audio output context. This code is a straightforward translation of the `createContext` method in `MidiTalk_AppDelegate.m` into StepTalk syntax.

Creating a context consists of a just a few steps. First, create a "context request": this identifies the sampling parameters desired for the sound device. Parameters include the sampling rate, period size and number of periods.

Next, allocate and initialize an audio context with the desired device name. Then, configure the audio parameters of the audio context with the context request. If not successful, the request may be incompatible with the abilities of the audio device.

Finally, start the context running with the `startWithError:` method. This method starts the background thread that for the context.

6.5 Playing two different Contexts

MidiTalk includes a script called "create-second-context-example.st" that illustrates playing notes on two different contexts: the default audio output, and a second USB device (our attached "MGXU"). See Listing 6.4. After creating the second context, this example creates a pattern that plays random notes on the two contexts in succession. It does this using a pattern. Note that here the script ensures the metronome is running before launching the patterns. Just to make it interesting, it launches two copies of the patterns so that pairs of notes from a pentatonic scale play on each context.

6.6 Finding the names of your Audio Device

To determine how the audio devices in your system are named, we need to run a command line tool. The `aplay` command comes in handy here. On our system, using `aplay -L` to list the devices shows the following as shown in Listing 6.5.

We can see that there are two hardware cards: 0 and 1. Card 1 is our MG-XU device. The device name is the string immediately following the card number: Here it is "MGXU". The

device name given to the alsa system is “hw:MGXU” to denote the hardware interface. We could also give “hw:1,0” to specify card 1, device 0.

Listing 6.1 Connecting to receive events FROM a MIDI device

```
"file: connect-example1.st"
"An example of (safely) connecting a MIDI device"
[|

connectFromDevice:name
  "lookup name on the SEQ and connect FROM it to receive events"

  errp := NSErrorPtr alloc init .
  addr := seq parseAddress:name error:errp .

  addr isNil ifTrue: [
    Transcript show:'Cannot find device: '; showLine:name .
    ^nil
  ] .

  Transcript show:'Found device at: '; showLine:addr .

  errp := NSErrorPtr alloc init .
  success := seq connectFrom:addr getClient port:addr getPort error:errp .

  success ifFalse: [
    Transcript show:'Could not connect from device: '; showLine:name .
    Transcript showLine:'Possibly already connected. ' .
  ]
  ifTrue: [
    Transcript show:'Connected device: '; showLine:name .
  ]
!

main
  currentScript = self .
  self connectFromDevice:'Xkey25' .

]
```

Listing 6.2 Creating a new named SEQ device and Dispatcher

```
"file: create-seq-example.st"
"Create a second sequencer interfae"

createSeqWithName:name
  | opts errp |
  opts := ASKSeqOptions alloc init .
  opts setSequencer_name:name .

  errp := NSErrorPtr alloc init .
  seq2 := ASKSeq alloc initWithOptions:opts error:errp .

  seq2 isNil ifTrue: [
    Transcript show:'Could not create seq: ' ; showLine:errp err .
  ]
  if False: [
    disp2 := ASKSeqDispatcher alloc initWithSeq:seq2 .
  ]
!

main
  "Do not create seq2 if there is already one"
  seq2 isNil ifTrue: [
    self createSeqWithName:'foo' .
  ]
```

Listing 6.3 Creating a second Audio Context

```
createContext:name
  | rqst errp ok |

  rqst := MSKContextRequest alloc init .
  rqst setRate: 44000 .
  rqst setPersize: 1024 .
  rqst setPeriods: 2 .

  errp := NSErrorPtr alloc init .
  ctx2 := MSKContext alloc initWithName:name andStream:0 error:errp .

  errp hasValue ifTrue: [
    Transcript show:'could not create context: '; showLine: errp err .
    ^nil
  ] .

  errp := NSErrorPtr alloc init .
  ok := ctx2 configureForRequest:rqst error:errp .

  ok ifFalse: [
    Transcript showLine:'could not configure request' .
    Transcript showLine: errp err .
    ^nil
  ] .

  errp := NSErrorPtr alloc init .
  ok := ctx2 startWithError:errp .

  ok ifFalse: [
    Transcript showLine:'could not start thread' .
    ^nil
  ]
!

main

  "retain reference to this script"
  currentScript := self .

  ctx2 isNil ifTrue: [
    self createContext:'hw:MGXU' .
  ] .
```

Listing 6.4 Playing Two Different Contexts

```
main

  "retain reference to this script"
  currentScript := self .

  ctx2 isNil ifTrue: [
    self createContext:'hw:MGXU' .
  ] .

  "Construct a pattern that plays a random note on each of the contexts"
  pat1 := Pattern alloc initWithName:'pattern1' .
  pat1
    sync: #beat ;
    play: [ self createNote:self randNote context: ctx . ];
    sync: #beat ;
    play: [ self createNote:self randNote context: ctx2 . ];
    repeat: 8 .

  metro start .
  sched launch:pat1 .
  sched launch:pat1 .
```

Listing 6.5 The output of the `aplay` command

```
$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: PCH [HDA Intel PCH], device 0: ALC256 Analog [ALC256 Analog]
  Subdevices: 0/1
  Subdevice #0: subdevice #0
card 0: PCH [HDA Intel PCH], device 3: HDMI 0 [DELL U2421E]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 0: PCH [HDA Intel PCH], device 7: HDMI 1 [HDMI 1]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 0: PCH [HDA Intel PCH], device 8: HDMI 2 [HDMI 2]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 0: PCH [HDA Intel PCH], device 9: HDMI 3 [HDMI 3]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: MGXU [MG-XU], device 0: USB Audio [USB Audio]
  Subdevices: 0/1
  Subdevice #0: subdevice #0
```

7 Advanced Topics

7.1 Main Thread

All events processed by StepTalk are handled on the Main Thread. GUI (Cocoa) events are always processed on the main thread. MIDI events are dispatched back to the main thread by MidiTalk with the `disp` object.

MIDI event sending and Audio processing is handled by background threads. These are managed by the McLaren Synth Kit internals. In MidiTalk, audio graphs are constructed as combinations of Voices and Models and these are handed over to the audio thread by the `addVoice` method. Scripts can modify the values of Models safely from the main thread.

The Metronome also runs on its own thread, but `play:[]` blocks are scheduled on the Main Thread. As a result, computations in a `play:[]` block cannot interrupt the Metronome. However, care should be taken to not place OS-blocking code in such blocks.

7.2 Changing the default Audio-In and Audio-Out

MidiTalk opens the audio devices named “default” for the input Context and the output Context. On most Linux systems this is a multiplexed audio device connected to the desktop. Sometimes, this may not be what you want.

Currently, the way to change the devices is to edit the Objective-C code and recompile. The methods `makeContext` and `makeRec` in `MidiTalk_AppDelegate.m` are where this happens.

7.3 Debugging

A good practice is to set the NSExceptionHandlerMask to “3”. This can help identify typos in your scripts.

```
$ defaults write NSGlobalDomain NSExceptionHandlerMask 3
```

This will log the exception, and put run a GUI Panel to display the error. (The meanings of these values are defined in NSApplication.m).

```
#define NSLogUncaughtExceptionMask 1
#define NSHandleUncaughtExceptionMask 2
#define NSLogUncaughtSystemExceptionMask 4
#define NSHandleUncaughtSystemExceptionMask 8
#define NSLogRuntimeErrorMask 16
#define NSLogUncaughtRuntimeErrorMask 32
```

It can also be helpful to turn on various debugging logs.

```
$ openapp ./MidiScriptDemo.app --GNU-Debug=dflt --GNU-Debug=STEngine \
    --GNU-Debug=STBytecodeInterpreter --GNU-Debug=STExecutionContext \
    --GNU-Debug=STSending
```

This will send additional information to the `NSLog` output.

7.4 Inspecting Methods and Variables

StepTalk provides extra capabilities through the concept of modules. Modules are dynamically loadable bundles that provide collections of classes and methods.

One useful module is called “ObjectiveC” and it provides introspection of objects and classes. Its features can be useful to find out what methods and instance variables. To try it out, either create a script with the following, or just use the Transcript and “Do & Show” each line as a selection.

```
Environment loadModule:'ObjectiveC' .

this methodNames .
NSObject instanceVariableNames .
(Environment objectDictionary @ 'ctx') methodNames .
```

This capability can come in handy for examining the environment.

8 Conclusion

This document describes the MidiTalk interpreted environment for building MIDI and Audio widgets using StepTalk as an integration language on top of Objective-C. It brings together a number of existing technologies in a new, and novel, way. Objective-C and the Cocoa architecture bring a firm foundation for the user interface components. The ALSA (Advanced Linux Sound Architecture) brings not only the interfaces to MIDI and Sound devices, but the solid timing of the Metronome. StepTalk provides an environment that effortlessly exposes the runtime of Objective-C as an interpreted language.

The McLaren Synth Kit and Alsa Sound Kit are Objective-C libraries that wrap ALSA components in an object-oriented Objective-C layer. This makes them instantly accessible as components in StepTalk. The Metronome and the Scheduler are new, and I believe the Pattern facility is somewhat unique in its ability to write figures in a tempo-independent manner.

The end result is a toolbox of components that can be combined in new, and interesting ways. Try it out and experiment! After writing a few scripts, customized to your project or workflow, I think you'll be delighted at what you find.

Index