Scheme For Max

Release 0.1

Iain CT Duncan

Nov 26, 2021

CONTENTS:

CHAPTER

ONE

OVERVIEW

Scheme For Max (S4M) is an open source project that enables scripting and live coding Max 8 and Max For Live with S74 Scheme. S74 Scheme is a thin layer over s7 Scheme, which is a minimal, Common-Lisp inspired Scheme implementation, created by Bill Schottstaedt at CCRMA, and used in the Common Music algorithmic composition toolkit and the Snd audio editor. S74 adds some beginner friendly and higher level Scheme language features to core S7, drawing from sources such as Racket, Clojure, and Chicken Scheme. Scheme for Max is authored by Iain Duncan, and hosted on GitHub.

If you'd like to see a demo of Scheme For Max in action, there are videos on the Music With Lisp channel on YouTube.

If you're new to Scheme, there is a tutorial that assumes no Lisp experience here: https://iainctduncan.github.io/learn-scheme-for-max/introduction.html

And a more advanced tutorial on building sequencers here: https://iainctduncan.github.io/s4m-stk/

These docs are in Scheme-For-Max-Docs repository on GitHub. If you find anything unclear, incorrect, or just have suggestions, please post a ticket or let me know on the Google Group.

S4M is available as a Max 8 package for OSX and both 32 and 64 bit Windows, as well as source code for Windows and OSX, for Max 7 or 8. It also is tested using Ableton Live 10 or 11 and Max For Live.

Features of v0.3 include:

- Hot reloading of scheme code
- A built in REPL terminal editor for interactive coding
- Max messages on inlet 0 automatically execute as Scheme code
- Dynamically registered listener functions for Max messages on inlets 1+
- Sending messages to remote objects by scripting name
- Ability to run in either the high or low priority thread
- Table access and i/o
- Buffer access and i/o
- Dictionary access and i/o, including nested lookup
- Integration with the Max transport controls
- High-accuracy event and function scheduling
- Support for Max time notation for scheduling
- Quantization with master transport settings
- Support for Max4Live and the Live API
- Garbage collector interface functions for high performance

The scheduling and thread support make Scheme For Max particularly useful for timing critical tasks, where the JavaScript object can not be reliably used as it runs only in the low-priority thread.

S4M provides the following Max patchers:

- s4m - The embedded interpreter
- s4m.repl - A patcher for making a terminal REPL in Max
- s4m.help - An extensive help file demoing all features with sample source code

Development is tracked on GitHub at github.com/iainctduncan/scheme-for-max

CHAPTER

TWO

MOTIVATION - WHY LISP IN MAX?

The purpose of Scheme For Max is to bring some of the power, convenience, and flexibility of Scheme / Lisp to the Max platform.

Project Goals:

- Hot-reloading of code, so that one can use the Lisp workflow of incrementally changing code while the program runs, preserving state where desired
- Ability to send code to a running max patch from a REPL in a terminal, to allow live coding directly in the Max environment or from a connected network REPL (ie Emacs)
- Ability to create domain specific languages (DSLs) appropriate for algorithmic composition, optionally building on the work that has been done in this area with projects like Common Music
- Deep integration with Max, through the C SDK, with fine grained control of execution timing and threading
- Ability to move between a Lisp context and the Max context more easily than is possible with the JS object

Embedding the S7 Scheme interpreter in a Max external has worked out well for achieving these.

We can load source files into the interpreter without needing to wipe out the running session. This allows us to keep state variables in one file and code under development in another, reloading function definitions as we go, while the patch is running.

We can live-code from a REPL editor by sending blocks of code to the s4m object. **s4m.repl** is a simple multi-line terminal editor object that we can use to interact with the interpreter in real time in Max. Future plans include supporting network REPLs through Emacs and the like.

Scheme also allows us to create simple DSL's so that we can send blocks of code to the interpreter that we dynamically generate with Max patcher objects. Scheme's minimal syntax and use of whitespace as token separators make this easy to implement, and match Max's semantics quite well. The s4m object takes any list of max symbols sent to inlet 0, and treats them as a list of tokens in a Lisp s-expression, evaluating them as if they were enclosed in an outer set of parantheses. This enables us to assemble valid Scheme expressions from standard Max objects and symbols. With S7's support for Common Lisp style macros, we can extend this into a domain specific language as powerful as we want.

The **s4m** external is open-sourced, and written using the C SDK and the S7 Scheme Foreign Function Interface. S7's FFI enables defining Scheme functions from C, and assembling Scheme objects in C from Max atoms. The user-developer can thus add any functionality in the Max SDK to the Scheme interpreter, and can also call Scheme functions from C code in response to messages to the **s4m** object. We can thus move operations from Scheme to C and vice versa, allowing us to opimize for performance or flexibility wherever we'd like.

Of course, as Scheme For Max is open-source, you're welcome to port it to another Lisp implementation too or even use the code as a sprinboard to embed a different lanuage!

CHAPTER

THREE

SETUP AND INSTALLATION

Scheme for Max is currently available as a package for Max 8 on OSX or Windows, and as source code that should compile for Max 7.

The Scheme For Max package can be downloaded from the Releases tab on GitHub.

To install the package, download the archive, and uncompress it in your Max packages directory. On OSX this is normally in **~/Documents/Max 8/Packages**. Once you have uncompressed it, in Max you can open the package manager (File -> Show Package Manager), and select "Installed Packages" in the upper right hand corner. You should now see an entry for Scheme For Max. Clicking **Launch** ought to open the Help tab.

In addition the Max external, the package contains:

- **help/**
  - The help file, **s4m.maxhelp**
  - The Scheme source files used in the help (**s4m_help_basics.scm** etc)
- **extras/**
  - The demo patcher(s), **s4m.demo.maxpat**, used in the demo videos
- **patchers/**
  - The repl patcher, **s4m.repl**
  - The main s4m Scheme file, **s4m.scm**
  - Optional extra Scheme files from s4m or base S7 that you may want

Important: When you install a Max Package, the patchers, extras, and help directories are automatically added to your Max file search path. If you move the Scheme files from the patchers directory, you will need to add their new location to the search path. In particular, the **scm4max.scm** file is essential, it bootstraps the Scheme side of Scheme For Max. If you decide to alter it, you probably want to make a backup (though you can always get it again from GitHub too).

Providing the above are all where they should be, you should be able run the help file and try out the features of Scheme for Max.

CHAPTER

FOUR

USING SCHEME FOR MAX

This page details all functionality offically released in the v0.2. The **s4m** help file includes example patches for most of what is here.

4.1 The s4m object

The **s4m** object is the core patcher object for Scheme For Max. It works fairly similarly to the **js** object, and can have a source file argument to load. The interpreter is thread safe, so we can have multiple objects in a patch, with each running its own isolated Scheme environments.

On creation of an **s4m** object, the interpreter first loads **s4m.scm**, which boostraps the S4M environment with any housekeeping that has been implemented in Scheme. This includes loading several other files, all of which must be present in the Max file path. If you have installed S4M from a package, this should all "just work". If you're getting errors about missing .scm files, you have likely installed the package incorrectly, have moved somethings, or have an error with your Max filepaths settings. Max *must* be able to find both the compiled s4m external and the scm files for bootstrapping.

One of the files loaded by **s4m.scm** is **s74.scm**. This holds pure Scheme functions that have nothing to do with Max, but augment S7 with convenience functions you may want to use. If you want to run pure S7, you can comment out this inclusion. It's mostly functions that are common in more batteries-included Schemes, such as Racket or Clojure.

You can change the **s4m.scm** file if you want to change bootstrap behaviour across all your **s4m** objects as well, such as to always load a personal library of Scheme definitions.

4.2 Arguments & Attributes

{filename.scm} An (optional) source filename as the first argument in the s4m object box will be loaded automatically (after the bootstrap files) searching the Max file search path. Changing this file in the box will always wipe the Scheme environment and reloads the file, as this recreates the **s4m** object. Reset messages will also reload this file.

@ins {num} Sets the number of inlets to num. This can only be set in the object box. Changing will reset the interpreter as it recreates the object.

@outs {num) Sets the number of outlets to num. This can only be set in the object box. Changing will reset the interpreter as it recreates the object.

@thread { h | l | a } Sets the thread in which Scheme code is executed. Setting to **h** will ensure all messages are promoted to the high-priority scheduler thread, while **l** defers all inputs to the low priority main thread. Setting to **a** (any) allows execution in whichever thread the incoming message is in, with no promotion or deferal. (**a** is meant for testing/debugging and is unstable.) This attribute can only be set in the object box. Changing will reset the interpreter as it recreates the object.

@log-repl {1 | 0} Sets whether the interpreter prints the output from the REPL to the Max console. When log-repl is enabled, any Scheme expression evaluated will have its return value printed. This can be changed anytime in both the inspector and by sending a **log-repl 1** message.

@log-null {1 | 0} Sets whether the interpreter prints the output from the REPL to the Max console when the output is the null list in Scheme, returned by many Scheme functions that are called for their side effect. It is often useful to turn this off when running scheduled Scheme functions on a timer, for example. This can be changed anytime in both the inspector and by sending a **log-null 1** message.

@heap {integer in KB} Sets the initial heap-size for s7. The default is 64KB, and lowest possible is 8KB (set with **@heap 8**). The heap-size changes how fast the gc runs - it takes longer to run over a large heap. The interpreter will resize the heap as needed, but this does introduce some latency every time it's done, so you may find performance increases if you set this to a value close to what you will need. To watch the gc-stats, run **(set! (*s7* 'gc-stats) #t)**, and you'll see information on the heap-size and the gc runs in the console.

When a source file has been listed in the object box, double clicking the object will display the full path to the file in the console. We can reboot the interpreter at any time with the **reset** message, which will also reload this file.

4.3 Reserved Words and Naming Conventions

In Max, messages to an object that consist of the name of an attribute and a value are (typically) executed automatically to set that attribute, if the object permits it. This means that we can't have Scheme functions with names that collide with attributes of **s4m** if we want these functions to be callable as automatically evaluated s-expressions on inlet 0 - the message will never get to the Scheme interpreter. A further complication is that in Max, the messages **int**, **float**, **symbol**, and **list** are implicit - if you send an object the message **int 4** or **list a b c**, they are treated by the receiving object as being the exact same as the message **4** or **a b c**.

In addition, S4M has some internal private functions used where the public API function calls a private helper implemented in C. These begin with **s4m-**.

The easiest solution is to avoid using the following as names in Scheme unless you know exactly why you are doing it:

Max reserved words: **int**, **float**, **symbol**, **get**, **set**

S4M attributes: **thread**, **ins**, **outs**, **log-repl**, **log-null**, **heap-size**

S4M commands: **reset**, **scan**, **read**, **eval-string**

S4M internal functions: anything starting with **s4m-**

The astute reader will have noticed that **list** is missing above. This is because **list** is already a Scheme function. This is also why our handlers for bang, int, float, and list are called **f-bang**, **f-int**, **f-float**, and **f-list** - to avoid a weird inconsistency as we can't call the Scheme list hander **list**.

S4M does not yet use the **get** and **set** message for anything special, but it's common in Max to use these to set state of an object, so consider them reserved for future use. Fortunately for us, in Scheme the set command is **set!**, so we can happily have **set foo bar** messages implemented in the object (but not in Scheme) in future without collision.

With regard to naming conventions, Scheme variables and functions can accept special characters and normally separate words with hyphens. Reader functions are usually **(foobar-ref {key})** and setter functions are usually **(foobar-set! {key} {value})**. We are following Scheme and Max naming conventions where possible, with Scheme as the priority when in doubt. Predicates end in ? and functions with side effects end in !, so we have for example: **dict-ref**, **dict-set!**, and **dict?**. Some people also use the convention of ear-muffs for globals. The special globals ***s7*** and ***s4m*** follow this.

S4M also includes short form aliases for most function names to simplify live coding and building s-expressions in Max messages where space may be at a premium. These are 4 to 6 character abbreviations and do not follow Scheme conventions (e.g. **tabr**, **tabs**, etc.) Aliases do not create an extra stack frame; they are registered at the C level so there is no speed penatly in using them.

There are various functions defined in the bootstrap files. If you want to know whether your function name is going to shadow anything already defined, just evaluate it in the REPL and see if you get anything. You should get an 'unbound variable' error in the console if there is no naming collision.

4.4 Messages to inlet 0

Messages to inlet 0 are either handled directly by the s4m object as a Max message, or evaluated as Scheme code if no direct handling exists. If the incoming message starts with a (character, S4M assumes this is code and evaluates it. If it doesn't, there are two possibilities.

If the message is not handled by the **s4m** object as a Max message, the interpreter will essentially pretend the message is enclosed in parentheses, evaluating the message as a Scheme s-expression to be called, with any symbols evaluating to their value in the Scheme environment. For example, the message **foobar 1 2 3** will be evaluated in Scheme as **(foobar 1 2 3)**. This also means that the message **out 0 a** will only execute properly if the variable **a** has been defined. To have **a** treated as a Max symbol, not a Scheme variable, we need to quote it in the standard Lisp fashion. For example, **out 0 'a** will send the symbol **"a"** out output 0, having been evaluated as **(out 0 'a)** in Scheme.

The messages **reset**, **read**, **scan**, and **eval-string** are handled directly by the **s4m** object, without passing to the interpreter (see Message reference below).

To evaluate a string as a code (for example, if received over the network from a udpreceive object), we prepend the message by using a **prepend eval-string** object. This will result in **s4m** receiving something like **eval-string "(define a 99)"**, which will execute as **(eval-string "(define a 99)")** in the Scheme interpreter.

Practically speaking, there is no difference in execution between sending inlet 0 of an **s4m** object either **define a 99**, **(define a 99)**, or **eval-string "(define a 99)"**.

4.4.1 Messages reference for inlet 0:

reset Resets the Scheme intepreter, wiping all active definitions, and reloading any sourcefile specified in the s4m object box itself. Also scans the current patcher for scripting names (see entry for scan).

read {filename} Loads the file {filename} from the Max search path. Executes the Scheme **load** function internally, after finding the full filepath by searching the Max filepaths. Loading does not erase any already active definitions unless they are redefined. Rereading a file will redefined any definitions.

scan Scans the current patcher and all descendents for objects with scripting names, adding them to an internal registry so that they can receive messages with the **send** scheme function.

{bang | int | float | list} Evaluates the function **f-bang**, **f-int**, **f-float**, or **f-list**, with the respective max atom(s) as argument(s). I.E. the int **4** sent to inlet 0 will executes in Scheme as **(f-int 4)**. If no f-type function defined, prints an error message to console. Return value of evaluation is printed to the Max console.

eval-string {string} Evaluates {string} in scheme. Return value of evaluation is printed to the Max console.

{symbol ...} Evalues the symbol or list of symbols as a Scheme s-expression. If sent the list **my-fun 1 2 3**, scheme will evaluate **(my-fun 1 2 3)**. Any symbols will be evaluated as Scheme variables unless quoted. For example, on **my-fun 1 a**, there will be an error if **a** has not been defined. Return value of evaluation is printed to the Max console. If the first symbol is not callable as a Scheme procedure, will produce an error (as if a symbol is evaluated surrounded with parentheses).

4.5 Messages to inlet 1+

Messages to inlet 1+ are treated as plain Max values or lists of Max atoms; they are not evaluated as Scheme code. Symbols will become quoted symbols in scheme, with no variable evaluation. As these are not evaluated as code, the messages may begin with any type. Keywords are useful here as they indicate visually that the message is not a function call, becauses keywords can not be used as function names in Scheme. Thus **(:foobar 1 2)** is always an error in Scheme. This means **:foobar 1 2** will always be an invalid message to inlet 0, but may be valid in inlet 1+ depending on what we have defined. It won't *do* anything unless we have registered a listener function on the inlet receiving the message, with the keyword **:foobar**. See Registering Listeners below. (If all this is confusing, skip it for now - you could use s4m productively never ever using inlets 1+)

Messages reference for inlet 0:

{bang | int | float | list} Evaluates Scheme function **(s4m-dispatch {inlet} {:bang | :int | :float | :list} {arg})**, which will call registered listener functions, with inlet as arg 1, and data as arg 2. For example, the message **4** on inlet 1 will call **(s4m-dispatch 1 :int 4)**, which will in turn call a listener function if a matching listener is registered. If no listener function is registered for the inlet used and the associated keyword (:int, :bang, etc), this will produce an error. See Registering Listeners.

{symbol ...} Evaluates scheme function **(s4m-dispatch {inlet} {symbol} arg)**, dispatching to listener functions registered with the symbol. Any arguments after the first symbol will be bound up in a list passed as **arg**, which may be empty. For example, the message **:foobar 1 2 3** on inlet 2 will call **(s4m-dispatch 2 :foobar arg)**, where **arg** is the Scheme value **(list 1 2 3)**. This will in turn call a listener function if registered. If no listener function is registered for the inlet used and the associated symbol, this will produce an error. See Registering Listeners for more details.

4.5.1 Registering Listeners for Max messages

Inlet 0

For inlet 0 to respond to bang, int, float, or list, we define functions named as below:

```
;; respond to bang messages by logging to console and sending bang out
(define (f-bang)
   (post "f-bang got the bang!")
   (out 0 'bang))

;; respond to int messages by logging to console and sending int + 1
(define (f-int num)
   ;; log and output num + 1
   (post "f-int got the int: ", num)
   (out 0 (+ 1 num)))

;; respond to float messages by sending out arg * 0.5
(define (f-float num)
   (post "f-float got the float: ", num)
   (out 0 (* 0.5 num)))

;; respond to lists by sending out in reverse the list elements as sequential messages
(define (f-list list-arg)
   (map out-0 (reverse list-arg)))
```

Note that the f-list function will not respond to lists starting with a *symbol*. Max doesn't consider those to be *list* messages, they are the message of the first *symbol*.

Any message starting with a symbol that is not alread reserved by S4M will be called as a Scheme function.

```
;; responds to max message "foobar 99" by outputing 99
;; if sent max message "foobar my-var", will output the value of variable my-var
;; if sent max message "foobar 1 2 3", will be "too many arguments" error
(define (foobar value)
   (post "foobar exectuting, value:" value)
   (out 0 value))

;; responds to max messages "foobaz ..." with any number of args
;; . args bundles variable list of optional args into a list
(define (foobaz . args)
   ;; log and output num + 1
   (post "foobaz executing, num args: " (length args))
   ;; output the first arg if there is one, or null list if not
   (cond
      ((> (length args) 0) (out 0 (args 0)))
      (else (post "no arg"))))
```

Note that in the above example we need to explictly check length args is > 0, because in Scheme anything except **#false** is **#true** - there is no automatic cast from **0** to **#f**.

Inlet 1+

For inlet 1+, we need to explictly register listener functions. The listener functions registered with **listen** should always take one argument, expecting it to be a list that may be of length zero. This allows the **s4m-dispatch** to be generic.

(listen {inlet} {symbol} {function}) Register the function to listen on inlet {inlet} for messages starting with {symbol}. Listeners are called by s4m's **s4m-dispatch** function, which will dispatch calls with the keyword symbols **:bang**, **:int**, **:float**, and **:list** for non symbolic messages.

```
;; define a listener for bangs, note that it takes an arg of a list
;; even though this will in practise be empty on bang messages
(define (my-bang-func args)
    (post "got the bang!"))
;; register it to listen for bangs on inlet 1
(listen 1 :bang my-bang-func)

;; define a listener for int messages, using an anonymous function
(listen 1 :int (lambda (args)
   (out 1 (args 0))))

;; the same function can listen for multiple messages
(define (num-listener args)
   (out 0 (+ 1 (args 0))))
(listen 1 :int num-listener)
(listen 1 :float num-listener)

;; a listener using a let to hide the signature weirdness
(define (my-listener args)
   (let ((num (args 0)))
      (post "num is:" num)
      (out 0 (+ 1 num))))
```

Listeners are stored internally in the **s4m-listeners** registry, a nested hashtable of {inlet} {symbol}. To remove a listener, you can put an empty function in:

```
;; remove the listener by registering false
(listen 1 :int #f)
```

Note that if you redefine a named listener function, it will not change what happens on the listened-to message until you re-register it, by virtue of how Scheme functions work. (We are registering the actual function, not the symbol of the function!)

4.6 s4m.repl patcher

The s4m.repl object is intended to be put in a **bpatcher** and then hooked up. The left outlet sends the output as a single text symbol. To evaluate as Scheme, we send it to a **prepend eval-string** object and send to inlet 0. This makes it the equivalent of:

```
(eval-string "(define a 99)")
```

The right outlet sends out a bang on each output to let you know it went out.

The s4m.repl patcher wraps the **textedit** box, which has some quirks/bugs. It wants to send out a bang when one bangs or hits enter in an empty box. In order to prevent Scheme error messages on this instance, **s4m.repl** filters these out.

If you select **Control Keys** on it, the **s4m.repl** object is listening to *all* key strokes, no matter where your focus is. So if you use this feature, be sure to turn it off when you're done. This can be especially confusing if you have multiple REPLs in different Max windows!

Future plans include making a proper terminal GUI object with history.

CHAPTER

FIVE

S4M SCHEME API

Below are details of the Scheme API. Core S7 functions are not listed here, only functions added by S4M. Scheme functions are either defined in **s4m.c** using the S7 Foreign Function Interface, or in Scheme code boostrapped from the initial load of **s4m.scm**, including the **s74.scm** file.

5.1 File Loading

(load-from-max {filename string}) Load a file into Scheme. Acts excactly like Scheme standard **load** but finds the full path on the Max file search path first.

5.2 Console Output

(post {args...}) Post to the Max console. All arguments will be converted to string representations automatically. Post returns the null list.

```
(define a 99)
(post "my var a is" a)
s4m> my var a is 99
```

Note: Console output by default does not print null responses, so that the console does not print a message on every function call for side effects, such as **out**. This can be changed by setting **log-null** to 1, with either a message, the attribute, or the inspector.

(s4m-filter-result res) Hook function to allow users to customize which results will be logged to the console. Redefine or alter this to return the keyword value **:no-log** to indicate result should not be logged.

```
;; if we replace what would be returned (res) by :no-log, s4m will not print to console
(define (s4m-filter-result res)
   (cond
     ;; turn off logging of the returned integers
     ((int? res) :no-log)
     ;; for everything else, do the normal thing
     (else res)))
```

5.3 Outputs

(out {outlet} {value}) Output {value} through outlet {outlet}. Value must be a single object. Returns the null list, thus by default does not log to console.

```
;; output number 99
(out 0 99)
;; output a max list of ints
(out 0 (list 1 2 3))
(out 0 '(1 2 3))
;; output a bang
(out 0 'bang)
;; output the value of my-var
(out 0 my-var)
;; output the max symbol "set"
(out 0 'set)
;; output the max message "set 99"
(out 0 (list 'set 99))
```

(out-0 {value}) Convenience helper for **out X**, as sometimes a single argument function is helpful. Defined for outs 0-7, to add more, edit **s4.scm** Returns the null list, thus by default does not log to console.

(out* {list or vector}) Unpacks the argument and spreads across all available outlets. Input lists with more items than there are outlets have the remaining items sent out as list out the final outlet.

```
;; assuming there are 3 outlets
(out* '(a b c d))
;; out 0: a, out 1: b, out 2: c d
```

5.4 Sending Messages

We can send arbitrary Max messages to other Max objects that have been given a scripting name. Before doing so, we must send the **scan** message to the **s4m** object, which will scan the current patcher and all descendents, registering scripting names interally in the **s4m** object (in the C code).

(send {target} {msg} ... {atoms}) Send the message {msg}, which may be followed by any number of values to be handled as Max atoms.

```
;; update the contents of a number box that has scripting name num-target
;; we quote num-target below as we want the symbol num-target, not the
;; value of a variable named num-target.
(send 'num-target 99)

;; send a message box a message to update to the contents "foobar 1 2 3"
(send 'msg-target 'set 'foobar 1 2 3)

;; this means we can send a message as list using scheme's apply function
(define msg-list (list 'set 1 2 3))
(apply send (cons 'msg-target msg-list))
```

There are also two convenience helper, **send-list** and **send***.

(send-list {target} {msg-list}) Send the message {msg-list}, a single arg of a list.

```
(define msg-list (list 'set 1 2 3))
(send-list 'msg-target msg-list)
```

(send* {args}) Flattens all lists passed as arguments and calls send

```
(send-list 'msg-target 'set '(a b c) '(1 2 3))
; calls (send 'msg-target 'a 'b 'c 1 2 3)
```

This can be used to integrate with all kinds of Max objects, including updating colls, dicts, tables, etc. We can copy whatever message the object receives and send them.

5.5 Table I/O

We can write and read integer data directly to and from named Max tables, as well as query for the existence and size of Max tables. Max tables are useful for storing integer data that we want to share between an **s4m** object and the rest of Max (or another **s4m** object. Tables are queried in each function, so if you want to read or write a lot of data, the vector conversions functions will be faster than a loop of single point table operations.

Short form aliases exist for all functions and are identical except for the function name.

(table? {table-name}) Returns #true if the symbol {table-name} corresponds to a named Max table.

(table-length {table-name}) Returns integer length of table {table-name}. Error if not a valid table. Alias: **tabl**

(table-ref {table-name} {index}) Returns value at {index} in table {table-name}. Alias: **tabr**

(table-set! {table-name} {index} {value}) Sets value at {index} in table {table-name} to {value}. Alias: **tabs**

(table->vector {table-name} {opt. index} {opt. count}) Returns a new Scheme vector from contents of a table, starting at index 0 or {index} in the table, copying entire table or {count} points if optional {count} given. Alias **t->v**

(table-set-from-vector! {table-name} {table-index} {vector} {opt. vector-index} {opt. count}) Sets contents in a table from a Scheme vector, starting at {table-index}. Copies the entire vector, or from {vector-index} for a total of {count} points if given. If table is not large enough, copies whatever fits. Returns a new vector of the contents copied. Alias: **tabsfv**

(vector-set-from-table! {vector-name} {vector-index} {table-name} {opt. table-index} {opt. count}) Sets content in an existing vector from a Max table, starting at {vector-index}. Copies the entire table, or from {table-index} for a total of {count} points if given. If vector is not large enough, copies whatever fits. Returns a new vector of the contents copied. Alias: **vecsft**

5.6 Buffer I/O

We can write and read float data directly to and from named Max buffers, as well as query for the existence and size of Max buffers. Max buffers are useful for storing float data that we want to share between an **s4m** object and the rest of Max (or another **s4m** object, they do not need to be used for audio samples but can hold any collection of floats. Buffers are queried and locked in each function, so if you want to read or write a lot of data, the vector conversions functions will be faster than a loop of single point buffer operations.

Short form aliases exist for all functions and are identical except for the function name.

(buffer? {buffer-name}) Returns #true if the symbol {buffer-name} corresponds to a named Max buffer.

(buffer-samples {buffer-name}) Returns integer length of buffer {buffer-name}. Error if not a valid buffer. Note that this is called buffer-size and not buffer-length to match Max naming, where length gives the length in seconds and size in samples. Alias: **bufsmp**

(buffer-ref {buffer-name} {opt. channel} {index}) Returns value at {index} in buffer {buffer-name}. If called with two arguments, channel defaults to zero. Alias: **bufr**

(buffer-set! {buffer-name} {opt. channel} {index} {value}) Sets value at {index} in buffer {buffer-name} to {value}. If called with two arguments, channel defaults to zero. Alias: **bufs**

(buffer->vector {buffer-name} {opt. channel} {opt. index} {opt. count}) Returns a new Scheme vector from contents of a buffer, reading from {channel} or channel zero, starting at index 0 or {index} in the buffer, copying entire buffer or {count} points if optional {count} given. Alias **b->v**

(buffer-set-from-vector! {buffer-name} {opt. buffer-channel} {opt. buffer-index} {vector} {opt. vector-index} {opt. count}) Sets contents in a buffer from a Scheme vector, using {buffer-channel} or channel zero and starting at {buffer-index} or index zero. Copies the entire vector, or from {vector-index} for a total of {count} points if given. If buffer is not large enough, copies whatever fits. Returns a new vector of the contents copied. Alias: **bufsv**

Note: there is not yet a buffer version of table-set-from-vector!

(vector-set-from-buffer! {vector-name} {vector-index} {buffer-name} {opt. buffer-index} {opt. count}) Sets content in an existing vector from a Max buffer, starting at {vector-index}. Copies the entire buffer, or from {buffer-index} for a total of {count} points if given. If vector is not large enough, copies whatever fits. Returns a new vector of the contents copied. Alias: **bufsft**

5.7 Dictionary I/O & hash-tables

We can read and write key-value stores in Max dictionaries and S7 Scheme hash-tables, which are unordered key-value hashmaps. When converting nested values in Max dictionaries to Scheme, arrays become Scheme vectors, and nested dictionaries become Scheme hash-tables. Only symbol, keyword, or string keys are supported by the conversion functions.

In Lisp dialects that support keywords, the idiomatic practice is to generally use keywords for hash-table keys. A keyword is a symbol that starts with a colon, and always evaluates to itself, which means that they do not need to be quoted. This works well in Max too, Max will simply treat them as string keys that happen to start with a colon. This ensures you never get mixed up about whether a symbol in a Max message is supposed to be evaluated or not - if it's a keyword it won't matter if this gets eval'd by the interpreter. Keywords are recommended as dict and hash-table keys wherever possible.

Note that in S7, querying a hash-table for a non-existent key returns **#false**. In Max, there is no boolean false value, and booleans are usually expressed as **0**. This means **0** is often a valid value to store, and thus getting **#false** back can be misleading. Instead of returning a potentially valid value on a key error, the Max dict functions raise Scheme errors, which can be caught to return a default value of your choosing.

5.7.1 S7 hash-table and keyword examples

```
;; a keyword always evaluates to itself
(eval :foobar)
s4m> :foobar

(define key :foobar)
(eval key)
s4m> :foobar
```

(continues on next page)

(continued from previous page)

```
;; which means quoting is unnecessary and does nothing
(eval ':foobar)
s4m> foobar

;; make a hashtable with two keyword slots
(define my-hash (hash-table :a 1 :b 2))
s4m> (hash-table :a 1 :b 2)

;; get value at :a
(hash-table-ref my-hash :a)
s4m> 1

;; applicative syntax of the same
(my-hash :a)
s4m> 1

;; a non-existing key from an S7 hash-table returns false (not an error!)
(my-hash :z)
s4m> #f

;; set a value in a hash-table, sets value and returns value
(hash-table-set! my-hash :foobar 99)
s4m> 99

;; applicative syntax of the same
(set! (my-hash :foobar) 99)
s4m> 99

;; make a nested hash-table
(set! (my-hash :pets) (hash-table))
s4m> (hash-table)

;; now we can set and get recursively using applicative syntax
(set! (my-hash :pets :dog) 'Poppy-Poodle)
s4m> Poppy-Poodle

(eval my-hash)
s4m> (hash-table :a 1 :b 2 :foobar 99 :pets (hash-table :dog Poppy-Poodle))

(hash-table-ref my-hash :pets :dog)
s4m> Poppy-Poodle
(my-hash :pets :dog)
s4m> Poppy-poodle
```

5.7.2 Max Dictionary API

Examples below can be used with the tests-dict.maxpat test patcher in the patchers directory.

(dict-ref {dict-name} {symbol|list key}) Return value in dict {dict-name} at {key}, where dict-name is a symbol, and key can be either a list or symbol. If key is a list, recurses through the list. Raises an **'key-error'** error if the key is invalid. Alias **dictr**

```
;; get a value from max dict named "test-dict", at key "a"
(dict-ref 'test-dict 'a)
s4m> 1

;; get a value that is a nested dict, becomes a hash-table
(dict-ref 'test-dict 'b)
s4m> (hash-table 'ba 2 'bb 3)

;; get value at key "ba" in nested dict at key "b"
(dict-ref 'test-dict (list 'b 'ba) )
;; same thing with alternative quoting syntax
(dict-ref 'test-dict '(b ba))
s4m> 2

;; get the value at index 2 in the nested vector at key "c"
(dict-ref 'test-dict '(c 2) )
s4m> 33

;; out of range index 3 raises error
(dict-ref 'test-dict '(c 3) )
s4m> ERROR
```

(dict-set! {dict-name} {symbol|list key} {value}) Sets value in dict {dict-name} at {key}, where dict-name is a symbol, and key can be either a list or symbol. If key is a list, recurses through the list. Returns value set. Raises error if key invalid. Alias **dicts**

```
;; set a value in max dict named "test-dict", at key "z"
(dict-set! 'test-dict 'z 44)
s4m> 44

;; set a value that is a hash-table, becomes a nested dict
(dict-set! 'test-dict 'y (hash-table :a 1 :b 2))
s4m> (hash-table :a 1 :b 2)

;; set value at key "bc" in nested dict at key "b"
(dict-set! 'test-dict (list 'b 'bc) 111)
s4m> 111

;; attempt set at invalid key list (there is no 'foo entry to recurse through)
(dict-set! 'test-dict (list 'foo 'b) 99)
s4m> ERROR
```

(dict-replace! {dict-name} {symbol|list key} {value}) Sets value in dict {dict-name} at {key}, where dict-name is a symbol, and key is a list of symbols or integers. If the list recursion hits an unused key, creates a nested dict for it, similar to the Max "replace" message to dicts. (Nested arrays do *not* get automatically created, also like the Max replace message). Returns value set. Raises error message if key or dict invalid. Alias **dicts***

```
;; set a value that is a hash-table, creating an intermediate hash-table automatically
(dict-replace! 'test-dict (list 'foo 'bar) 99)
s4m> 99
```

(dict->hash-table {dict-name}) Return a hash-table of a Max dictionary. Nested dicts become nested hash-tables, arrays become vectors. Raises error on bad dict-name. Alias **d->h**

(hash-table->dict {hash-table} {dict-name}) Write entire contents of a hash-table to a named Max dict. If {dict-name} does not exist, creates a dictionary. If {dict-name} already exists, replaces entire contents. Alias **h->d**.

5.8 Max Time and Transport API

Transport functions exist to interact with the Max master transport. Named transports in addition to the master transport are not yet supported, and behaviour in Max For Live is unknown (but will be tackled in future!).

(time) Returns current time in float ms. This is the global Max time, *not* the transport time. This only resets to 0 on restarting Max.

(transport-state) Returns **#t** if transport is running, **#f** otherwise. Alias **t-state**

(transport-state-set! {boolean|int state}) Starts master transport on **#t** or **1**, stops on **#f** or **0**. Returns state set. Alias **t-state!**.

(transport-bpm-set! {int bpm}) Starts master transport tempo to bpm. Note, there is no get version in the Max C SDK, strangely. Returns bpm set. Alias **t-bpm!**

(transport-time-sig) Returns current time signature as list of (numerator denominator). Alias **t-time-sig**

(transport-time-sig-set! {int numerator} {int denominator}) Sets master transport time signature. Alias **t-time-sig!**

(transport-ticks) Returns current master transport location in ticks (float). Alias **t-ticks**

(transport-seek {opt. bars} {opt. int beats} {float|int ticks}) Sets master transport location immediately. If called with three arguments, sets with Max bbu (bars beats units) format, otherwise sets location in ticks. Returns new transport location in ticks (float). Alias **t-seek**

(ticks->ms {number ticks}) Converts ticks to float ms according to current settings of the master transport.

(ticks->bbu {number ticks}) Converts ticks to list of (bars, beats, ticks) according to current settings of the master transport.

(ms->ticks {number ms}) Converts ms to float ticks according to current settings of the master transport.

(ms->bbu {number ms}) Converts ms to list of (bars beats units) according to current settings of the master transport.

(bbu->ms {int bars} {int beats} {number units}) Converts bars-beats-units to float ms according to current settings of the master transport.

(bbu->ticks {int bars} {int beats} {number units}) Converts bars-beats-units to float ticks according to current settings of the master transport.

5.9 Scheduling, Delays, & Timers

S4M uses Max's clock facilities to interact with the Max scheduler to create accurate delays and timers. As of 0.3, these can be run from both high priority and low priority thread objects. In the low priority thread, they are scheduled with the high priority scheduler, but defered when they start.

Delays can also interact with the global transport, including quantizing to transport-aware time values.

Internally, delays register callback functions in the global **s4m-callback-registry** hash-table. Cancelling an event consists of removing the callback function. Functions that are scheduled with the **delay** functions or used as clock listeners must take no arguments, with the exception of the tick listeners, which will get the current tick as a single argument.

5.9.1 Helpful Functions

Some helpful functions for working with timers and delay functions.

(isr?) Returns true if exectuting in the the high-priority scheduler thread.

(time) Returns current Max time in ms.

5.9.2 Delay Functions

Functions to schedule execution of a Scheme function in the Max scheduler.

(delay {number time} {function}) Schedule function to execute in {time} ms. Returns a symbol of the callback key, which is the key underwhich the generated callback function is registered, and which can be used to cancel an event.

```
;; schedule a function that posts to the console in 1 sec
(delay 1000 (lambda () (post :tada)))

;; or
(define (delayed-func) (post :tada))
(delay 1000 delayed-func)
```

(delay-eval {number time} {var-args}) Convenience function to allow calling delay without having to make a function. If called with 1 arg in var-args and it is a list, it will be evaluated at the time scheduled. If called with many args, they will be treated as a list and evaluated at the time scheduled.

```
;; schedule a function that outputs 99 from outlet 0 in 1 second
(delay-eval 1000 (list out 0 99) )

;; or
(delay-eval 1000 out 0 99)
```

(delay-t {number|symbol time} {function}) Transport aware version of delay. Time can be number of ticks, or Max time notation such as **4n**. If using time notation as a symbol, you will need to quote it or pass it in a variable. Otherwise identical to delay.

```
;; delay for 480 ticks (1 quarter note)
(delay-t 480 my-function)

;; or
(delay-t '4n my-function)
```

(continues on next page)

(continued from previous page)

```
;; or
(define quarter '4n)
(delay-t quarter my-function)
```

(delay-t-eval {number|symbol time} {var args}) Transport aware version of delay-eval.

(delay-tq {number|symbol time} {number|symbol quantize-time} {function}) Transport aware and quantized version of delay. Time and quantize-time can be number of ticks, or Max time notation such as **4n**. Execution is scheduled for the next boundary of {quantize-time}. Note that this is independent of whether the transport is playing - the correct time is calculated and scheduled. Stopping the transport does not cancel the event.

(delay-tq-eval {number|symbol time} {number|symbol quantize-time} {var args}) Eval version of delay-tq.

(cancel-delay {handle}) Given a delay call that returns {handle}, cancel execution of the function. Note that this function is used for all the delay variants, as they all return guaranteed unique callback handles.

```
;; schedule a function and then cancel it
(define cb-handle (delay 1000 (lambda () (post :tada))))
(cancel-delay cb-handle)
```

5.9.3 Timer Functions

Several functions are available to register periodic timers, with or without following the global transport. These are implemented at present only for the high-priority thread. (Low priority timers to be implemented in future). These all return the keyword **:clock-registered** to let you know they succeeded.

(clock-ms {number ms} {function callback}) Register function {callback} to be executed every {ms} milliseconds. The callback registered should accept no arguments, but can get the the current time by calling **(time)** in its body.

```
;; register a time callback to post current time in ms every second
(define (time-callback) (post "current time" (time)))
(clock-ms 1000.0 time-callback)
```

(cancel-clock-ms) Cancels the clock-ms timer.

(clock-ms-t {number ms} {function callback}) Transport aware version of **clock-ms**. The clock used only advances if transport is playing.

(cancel-clock-ms-t) Cancels the clock-ms-t timer.

(clock-ticks {number ticks} {function callback}) Transport aware version that receives the current transport tick location as its argument.

```
;; register a time callback to post current ticks every quarter note
(define (tick-callback current-tick) (post "current tick:" (current-tick)))
(clock-ticks 480 tick-callback)
```

(cancel-clock-ticks) Cancels the clock-ticks timer.

5.9.4 Garbage Collector Functions

Advanced users may want to tweak how the garbage collector runs to allow running with lower latency and larger programs. For example, one might set the heap-size quite low and deliberately run the gc very frequently off a timer, quantized such that it runs at an unimportant time musically such as between 16th notes every 4th 16th note or something. Or we might decide not to run it at all until a song is over, and then run when the music is paused.

(gc-disable) Turn off the gc. Note that your heap-size will grow, and once you turn it back on, it will take a lot longer to run the gc if the heap is now big.

(gc-enable) Enables the gc. This does not tell the gc to run though.

(gc-try) Runs the gc if it's enabled, does nothing if not.

(gc-run) Forces the gc to run, whether or not it's enabled. Does not change the enable status.

(set! (*s7* 'gc-stats #t)) Turns on and off gc stats logging to the console, which will show you how fast its running and over how much memory.

(*s7* 'heap-size) Returns the current heap size. You can use this to see how big the heap got while you had the gc disabled.

(*s4m* :heap-size) Returns the s4m *starting* heap-size. You can use this to set how much s4m allocates at the beginning to prevent unnecessary heap resizes if you're planning on locking out the gc.

Note that once the heap has grown, the only way to shrink it again is to recreate your s4m object by editing the s4m box. Reset does not shrink the heap.

If you want to run with no gc, the best thing to do is to turn on stats, play for as long as you plan to run, and note the heap-size. Then start your s4m object off with this heap-size. This will avoid the work of resizing the heap during playback. However, this does mean any gc run will be slow, so you will need to keep it disabled until you can afford an underrun.

Alternately, you can set the heap-size very low, and run the gc proactively very frequently. This will not run as fast as locking it out entirely (assuming you've made sure the heap won't resize), but avoids the problem of eventually needing to run a really slow gc round.

CHAPTER

SIX

ALGORITHMIC PROCESSES AND LIVE-CODING

We can use the **delay** and **clock** functions to create sequencers and repeating algorithmic processes, and interact with these in real-time by changing variables or redefining functions. This section provides some helpful examples and tips. This document assumes familiarity with lambda functions and let blocks, though one could certainly use the examples here to learn them.

The Scheme for Max Sequencer Toolkit tutorial uses these functions extensively, and is here: https://iainctduncan.github.io/s4m-stk/

This kind of algorithmic composition is also explored in detail in the (highly recommended!) book **Notes from the Metalevel**, by Dr Heinrich Taube. The book uses S7 Scheme with the Common Music runtime, but examples are easily adapted to Scheme For Max.

6.1 Dynamically Redefining Scheduled Functions

We can use the interpreter to redefine scheduled functions on the fly, but this requires some care to get the behaviour you are expecting. When we register a scheduled function or a clock listener function, we are registering the *function*, not the *symbol for the function*. This means that if a function is registered in the scheduler (whether as a listener or using delay), redefining the function attached to the symbol for the function's *name* has no effect on the function called when the timer callback fires. If we want to ensure that redefining the value of a given symbol changes what an already registered listener does, we need to make a wrapper function that evaluates the global symbol on each pass. This is best illustrated with examples:

Example of function registration that is unaffected by changes to the symbol used to define the function:

```
;; define a function that posts :foobar to the console
(define (my-listener) (post :foobar))

;; register it to get evaluated every second
(clock-ms 1000 my-listener)

;; now we see :foobar posted every second

;; redefine my-listener
(define (my-listener) (post :foobaz) )

;; no change to what is getting posted, because the old
;; my-listener function is still registered!

;; register the new function, replacing the function registered for clock-ms
(clock-ms 1000 my-listener)
```

(continues on next page)

(continued from previous page)

```
;; now we get :foobaz posted
```

Example of function registration that ensures changing the listener will have an immediate effect:

```
;; define a function that posts :foobar to the console
(define (my-listener) (post :foobar))

;; register a lambda function that evaluates whatever is currently
;; attached to the symbol 'my-listener
(clock-ms 1000 (lambda () (eval (list 'my-listener))))
;; same, different syntax
(clock-ms 1000 (lambda () (eval '(my-listener))))

;; every second we are getting :foobar posted now

;; redefine my-listener
(define (my-listener) (post :foobaz) )

;; now foobaz is getting posted
```

The same scoping rules apply for variables that your function uses. If we want changes to a top level variable to immediate be reflected in our registered functions, we have those functions use values looked up from global symbols:

```
;; define and register function that outputs the note-number currently at 'note-num
;; this will print an error message if note-num does not exist
(define note-num 60)
(define (my-listener) (out 0 note-num))

(clock-ms 1000 my-listener)
;; function is outputing 60

(set! note-num 72)
;; function is now outputing 72
```

6.2 Capturing variables with lexical closures

To capture the value of a global (or outer) variable, but keep the listener function using it regardless of any subsequent changes to the global variable, we need to use a lexical closure. There are several ways of doing this, one is to use a **let** block.

```
;; assume we have a global var, note-num, that we manipulate somehow
(define note-num 60)

(define my-listener
  (let ((captured-note-num note-num))
    (lambda () (out 0 captured-note-num))))

(clock-ms 1000 my-listener)
```

(continues on next page)

(continued from previous page)

```
;; 60 now coming out every second

(set! note-num 72)

;; no change, captured-note-num is not subject to changes to symbol note-num
```

This is a very common pattern in Lisp languages, often refered to as "let over lambda". A variable is declared in the let definitions and its value is read from the global when the let block runs. The let block then returns a function that uses the symbol local to the let block, thus the function returned takes the let environment with it.

Another way of doing this is to use function arguments, and capture those with wrapper lambdas. In this case, we would define a function we want to have run at the scheduled time, with some local arguments, and register a wrapper function that takes no arguments as the actualy delayed callback:

```
;; assume we have a global var, note-num, that we manipulate somehow
(define note-num 60)

(define (my-delay-fun note-num-param)
  (out 0 note-num-param))

(define my-delay-fun-callback
  (let ((captured-note-num note-num))
    (lambda ()
      (my-delay-fun captured-note-num))))

(clock-ms 1000 my-delay-fun-callback)

;; 60 now coming out every second
(set! note-num 72)

;; no change, the local note-num-param captured the value of
;; note-num at the time of the lambda
```

Below is an example of reading from both mutable top level variables and captured variables:

```
;; top level vars for note-num and velocity
(define note-num 60)
(define velocity 90)

(define my-listener
  (let ((captured-note-num note-num))
    ;; captured-note-num is in the let, but velocity is global
    (lambda () (out 0 (list captured-note-num velocity)))))

(clock-ms 1000 my-listener)

;; 60 90 now coming out every second

(set! note-num 72)
(set! velocity 120)

;; 60 120 coming out - velocity changes, but note-num does not
```

6.3 Temporal Recursion / Recursive Scheduling

It is safe to have a scheduled function re-schedule another function, sometimes called "temporal recursion" in the algorithmic composition literature. The function definition uses standard Lisp recursion:

```
;; my self-scheduled callback
(define (my-callback)
  (post "executing!")
  (delay-t 480 my-callback))

;; start it up immediately, it will then repeat every 480 ticks
(delay 0 my-callback)
```

These require some management if you want to be able to cancel their execution (without just resetting the intrepreter!) One method is to write to a global (or other scope) variable that

> stores the last registered callback handler and can thus be used to stop playback.

```
;; a variable to hold the handle, outside of the scope of the callback
(define cb-handle #f)

(define (my-callback)
  (post "executing!")
  (set! cb-handle (delay-t 480 my-callback)))

;; start it up, and capture the callback
(set! cb-handle (delay 0 my-callback))

;; at some point, we can stop playback by cancelling the callback
(cancel-delay cb-handle)
```

If we use the transport aware delay functions with quantize (**delay-tq**) this can be used to spawn repeating processes synchronized with musical time and kept in sync with the transport tempo. You may want to make your own version of transport control functions that manage callback cancellation and then stop the transport using the aforementioned transport functions.

6.4 Recurive Scheduling and Lexical Closures

By combining lexical clojures with self-scheduling delay calls, we can make functions that change the arguments for their next iteration. Below is an example of a function that counts down 4 repetitions.

```
(define global-num-reps 3)

(define (my-fun rep-num)
  (post "rep:" rep-num)
  ;; do stuff for this iteration

  ;; make arguments for next iteration
  (let ((next-rep-num (- rep-num 1)))

    ;; in let body we schedule next iteration with the captured arg
    (delay 1000 (lambda ()
```

(continues on next page)

(continued from previous page)

```
      (my-fun next-rep-num)))))

;; now we need to kick it off with a lambda capturing the first arg
(delay 1000 (lambda ()(my-fun global-num-reps)))
```

The above will just count forever. Below is the same improved so that the process stops after the count reaches the last rep:

```
(define global-num-reps 3)

(define (my-fun rep-num)
  (post "rep:" rep-num)
  ;; do stuff for this iteration

  ;; make arguments for next iteration
  (let ((next-rep-num (- rep-num 1)))

    ;; schedule next iteration only if next-rep-num is greater than zero
    ;; otherwise, post :done
    (if (and playing (> next-rep-num 0))
      (delay 1000 (lambda () (my-fun next-rep-num)))
      (post :DONE))))

;; start it up
(delay 1000 (lambda ()(my-fun global-num-reps)))
```

6.5 Controlling Recursive Processes

We might need to be able to stop a process early, so we can also check a global to see if we should still be playing before scheduling the next iteration:

```
;; toggle for stopping playback
(define playing #t)
(define global-num-reps 100)

(define (my-fun rep-num)
  (post "rep:" rep-num)
  ;; do stuff for this iteration

  ;; make arguments for next iteration
  (let ((next-rep-num (- rep-num 1)))

    ;; schedule next iteration only if next-rep-num is greater than zero
    ;; otherwise, post :done
    (if (and playing (> next-rep-num 0))
      (delay 1000 (lambda () (my-fun next-rep-num)))
      (post :DONE))))

;; start it up
(delay 1000 (lambda ()(my-fun global-num-reps)))
```

(continues on next page)

(continued from previous page)

```
;; stop playback early
(set! playing #f)
```

Finally, what if we want to be able to stop only certain processes? If we capture the callback handle for each delay call in a top level variable, we can cancel recursion for a process at anytime by cancelling that handle:

```
;; toggle for stopping playback
(define playing #t)
(define global-num-reps 100)
(define process-handle #f)

(define (my-fun rep-num)
  (post "rep:" rep-num)
  ;; do stuff for this iteration

  ;; make arguments for next iteration
  (let ((next-rep-num (- rep-num 1)))

    ;; schedule next iteration only if next-rep-num is greater than zero
    ;; otherwise, post :done
    (if (and playing (> next-rep-num 0))
      ;; schedule and save the handle to the global process-handle var
      (set! process-handle (delay 1000 (lambda () (my-fun next-rep-num))))
      (post :DONE))))

;; start the process, saving the callback handle
(set! process-handle (delay 1000 (lambda ()(my-fun global-num-reps))))

;; stop only this process
(cancel-delay process-handle)
```

CHAPTER

SEVEN

MAX THREADS, THE SCHEDULER AND THE S7 GARBAGE COLLECTOR

One of S4M's distinguishing features is its support for execution in either the Max high-priority scheduler or low priority UI thread, not possible with either the JS or Node For Max objects. This allows S4M to be used for precise timing and scheduling operations for sequencing, live coding, altering real-time musical input, and for algorithmic composition. This page discusses how threading works within Max and S4M and why it matters, and what you should know about Max Settings to get the best performance.

7.1 Max's scheduling model

To understand scheduling and threading in Scheme For Max, we need to understand how Max's scheduler and threading model works. In Max, there are normally between one and three threads running, depending on your configuration settings and what you have in your patch. These are the main thread, the scheduler thread, and the audio thread. (There could also be additional threads if a Max object explictly creates and manages them, but this is not relevant for our purposes, and can be safely ignored for now.)

7.1.1 Main thread

The main thread is also called the UI thread, and runs with the lowest priority. When we initiate activity by using Max user interface objects (clicking bangs and messages, moving sliders, etc), we trigger execution in this thread. This thread is also responsible for screen redrawing and file i/o, and executes at a lower priority so that the screen can refresh without interupting audio playback or delaying musical events. Generally speaking, operations that involve loading files, refreshing UI elements, or creating visuals (with Jitter for example) should be done in this thread. In Max For Live, the Live API also runs in the low priority thread, similar to control surface scripts in Live.

7.1.2 Scheduler thread

The scheduler thread is also called the high-priority thread. This is where timing critical Max messages should execute, and by default MIDI i/o and **metro** object messages execute in this thread, as well as any events triggered from the **transport** object. We also execute in this thread if we trigger Max event messages from audio signal processing, such as with the **~edge** object. This thread context only exists as an actual separate thread if you have enabled "Overdrive" in your settings. Overdrive moves the scheduler thread into an Interupt Service Routine for high timing accuracy and more frequent scheduling. The only reason not to enable Overdrive is if a patch is entirely aimed at visuals. We can configure this thread by setting the 'Scheduler Interval' setting in Scheduler preferences to control how frequently the thread runs (the default is 1ms) and the Poll Throttle setting to set the maximum number of event messages processed on one scheduler thread pass. (Though as is explained shortly, these don't actually matter if you also have selected Audio in Interupt.)

7.1.3 Audio thread

The audio thread is used to calculate MSP audio rendering. It only runs if audio processing is turned on, and is the thread that executes all real-time audio processing. On each pass, it calculates the audio samples necessary to fill up one signal vector of samples. Max is a buffered audio system, meaning that each audio pass calculates a series of audio samples that are then sent out in a chunk. We set this number of samples with the Signal Vector Size setting in the Audio Status window. This setting also affects audio latency as we will always have a latency of at least the size of the Signal Vector multiplied by the time for one sample at the current sample rate (before adding any sound card latency). Note that in Max For Live, the signal vector size is locked at 64 samples.

7.1.4 Audio in Interrupt

If, in the Audio Status window, we have selected Audio Interrupt as well as Scheduler Overdrive, or we are running in Max For Live, we will actually have only two threads running, because the audio thread and the scheduler thread are merged into one thread that alternates between rendering a vector of audio samples and handling scheduler events. This also means that events are processed once per signal vector size, with event onset times quantized to this boundary.

This may or may not be better for your use case, depending on how big your audio vector size is and how much audio CPU intensive processing your patch needs to do. For example, if you are running heavy audio calculation and thus need a large signal vector size, you might see scheduler thread activity get delayed by the time for one signal vector of audio samples.

7.2 Delay and Defer operations:

Max has several objects for delaying Max events and moving them into different threads. The **delay** and **pipe** objects delay bangs and messages respectively by some number of milliseconds, putting the delayed message on the *scheduler thread* queue to execute in the scheduler thread at a later time. To move an event from the main (UI) thread into the scheduler thread, we can send it through a **delay** or **pipe** object with the delay time set to zero, which will put the message on to the scheduler thread's queue for execution on the *next* pass of the scheduler thread.

The **defer** and **deferlow** objects do the opposite: any incoming messages will be will be put on the queue for the low priority thread to execute on its next pass. The **deferlow** object puts the event on the end of the low priority queue while **defer** puts on the front. If we send a message executing in the low priority thread into a **deferlow**, it will be moved to the back of the queue, while a low priority event into a **defer** simple passes the message through without change.

7.3 The S7 Garbage Collector

S7 Scheme is a garbage-collected language, meaning that interpreter occasionally does a check for memory that has been allocated but no longer has any references to it, and can thus be freed. This saves as us from having to manually manage memory, as one does in languages like C, but has latency implications because we never know when it will run. Garbage collectors (GCs) are also used in dynamic languages such as JavaScript, Lisp, Ruby, and Python. (S7's GC is a mark-and-sweep garbage collector in case your into that sort of thing.) It will run occasionally, and typically takes 1-2 ms to run on my machine, depending on how long it has been since it last ran, how many s7 objects are in memory, and how much memory s7 is using.

We can tell the gc to run on demand by calling **(gc-run)** from Scheme. We can enable and disable the garbage collector as well, with **(gc-enable)** and **(gc-disable)**, and ask to run it only if it's enabled with **(gc-try)**. Most users won't need to bother with these, but if you are trying to achieve low-latency with a large Scheme program, manually running the gc at certain intervals or turning it off for the duration of a piece and then running it between pieces can be helpful.

In practical terms, this means Scheme is not likely to work well for *audio* generation, where 2 ms is a long time, but for manipulation of musical events or musical input, it's not noticeable. We notice a lag of 5 ms as smearing, and 20

ms as a discrete event, so a 2 ms lag of the scheduler is not discernable, and if we are running with an audio latency of over this, it probably won't have any effect at all. For example, if we have an i/o vector and signal vector set such that the system has an overall latency of 5-10ms, the lag from the garbage collector is likely to be lower than the lag from our built in latency and won't cause any issues.

7.4 S4M Thread Selection

The **s4m** object is locked to *either* the high priority or low priority thread with the **@thread** attribute. This can be set to **h** or **l**, and is **h** by default if you don't set it explicitly. In the s4m object, any incoming message will be promoted or defered to ensure it's handled in the right thread before being evaluated by the Scheme intepreter. This prevents any threading errors in Scheme by ensuring that the interpreter won't be interupted in the middle of a critical operation (such as the garbage collector running) to start some other conflicting operation in the other thread. The **@thread** attribute can *only* be set in the object box (not the inspector) as it cannot change during the life of an **s4m** object. Leaving this empty is the same as **@thread h**. Note that this works fine with or without Audio Interrupt set to on; if you do have Audio Interrupt set, **s4m** will execute all its events once per audio buffer calculation. (Or at least as many as your scheduler settings allow).

If you want to use Scheme code in both low and high threads, the correct approach is to have two **s4m** objects, and allow them to communicate with each other either by sending messages to each other, or by using one of the thread-protected Max data store objects to which Scheme can write, namely buffers, tables, and dictionaries. This essentially creates an Actor model, and if you wanted to, you could even implement a three-priority actor model by having one **s4m** object only receive messages that had gone through a **deferlow**. This would delay events to the third (lowest) **s4m** object such that they will only get serviced if Max has time to get through the whole low priority queue on the main thread pass.

Note that this is also how the Ableton Live API objects work in Max For Live - it is always in the low thread and any incoming messages are defered. So if you want to interact programmatically with the Live API, you should do it from a low-thread s4m object. If you want to interact with the Live API from a seqencer you're running in the high thread, you should have your high thread s4m object send the low thread object a message telling it to trigger an API action.

7.5 Comparison to JS and Node For Max

The **js** (JavaScript) object in Max *only* executes in the low priority thread. This is done to ensure there is no chance of a thread context switch corrupting data and creating unknown conditions, as detailed above. The fact that the **js** object always defers messages to the low priority thread has ramifications for what we can reliably do with the **js** object. Realistically, we can never guarantee that an event handled in JavaScript code will be serviced in a timely manner. While there are delay operations in JS, they only create a *minimum* delay. The event won't be serviced until the delay is done, and the low priority thread gets to run again. It's not at all uncommon for the low priority thread to hiccup: visual redrawing operations or any file system interaction can do it. The upshot of this is that while the **js** object has quite a comprehensive set of features, it's not really reliable for timing critical operations. Note that this only matters for events that will be rendered to *audio* - user interface and Jitter operations execute in the low priority thread anyway, making **js** suitable for UI or Jitter manipulation.

There is often some confusion about how Node For Max fits into all this. Node For Max uses a completely different model, where the Node script executes in a separate *process* that is spawned and supervised by the Max process. Under the hood, Node scripts do not get executed by Max system code, (the way the **js** and **s4m** objects do), and thus don't have programatic access to all of Max's internals through the Max C API functions. The Node code is executed by a Node.JS runtime, outside of Max, with communication to and from Max over a network connection. Messages and data are serialized and deserialized automatically so that we don't need to deal with any of the networking complexity. The advantage of this is that the Node script can work with the external file system or communicate with the outside world over networks and we never have to worry about how long that takes. When Node finishes its work, messages come back to Max. However, this means that Node is, like *js*, not reliable for timing critical code, as we have no guarantees of when Node code will complete and send messages back to Max. This does, however, make Node For Max a good

compliment to S4M. By adding a Node object to our patchers, we can use it for operations that might take a while and which we don't want blocking either a low or high priority **s4m** object. We can have an **s4m** object send messages to Node to request a long running operation (such as saving or loading a file), or to take advantage of the rich libraries for interacting with the world outside of Max, and have the results sent back as messages to **s4m** or make them accessible in Max dictionaries that **s4m** can read.

CHAPTER

EIGHT

BUILDING FROM SOURCE

Below are some (in progress) instructions for building from source. Help in improving this area would be much appreciated, please get in touch if you want to build from source and are not managing.

8.1 Short version

I'm using the Max 8 SDK, though I think the Max 7 should work too. We are using the "canonical" C API, not the newer C++ API, as it is much better documented at this point.

The repo includes **s7.c**, **s7.h**, and an empty file **mus-config.h**. You need **mus-config.h** to build **s7.o**, and then you need to link in **s7.o**. There is only one source file for the external, **s4m.c**. Building it will build the Max external **s4m**. The other objects are just standard Max patchers.

There is an XCode and a VisualStudio project file in the repo. The VS one works properly, the XCode one is not set up right as I haven't figured out how to get S7 buidling properly in XCode, so I'm just being stupid and compiling S7 to object code with GCC and linking it in. (Pull requests or patches to fix the XCode file would be lovely!)

My linker flags:

- -framework MaxAudioAPI
- -framework JitterAPI
- ${C74_SYM_LINKER_FLAGS}
- s7.o
- -L.
- -ldl
- -lm

8.2 Long version

To build from your source, you'll need to have installed the Max 8 SDK. I believe S4M should compile fine with the Max 7 SDK as well, but this has not been verified. If you have not previously compiled a Max external with the SDK, I recommend you start by compiling some of the samples that come with the SDK, as the XCode and VisualStudio setup is finicky. It should theoretically be possible to build without the full XCode IDE, but I haven't managed to make this work. If you know how, please let me know.

The SDK comes with sample projects with project files for XCode and VisualStudio. I'll be honest, I know just enough about XCode to be dangerous, so the instructions below are not "the right way". I altered a sample XCode project file from the samples in the SDK, and my alterations are detailed here.

Note than on XCode at least, these XCode project files from C74 assume that the SDK files are in a specific place relative to your code. So you should clone the repository such that you have the following path layout, where the SDK path matches whatever version you download, and **Scheme4max** is the top level of the Git repository.

~/Documents/Max 8/Packages/max-sdk-8.0.3/source/Scheme4max

So for me, the following works:

$ cd ~/Documents/Max8/Packages/max-sdk-8.0.3/source

$ git clone https://github.com/iainctduncan/Scheme-for-max.git

In the cloned repository, you should see a source directory, **s4m**, with the following:

- **s7.c** - the main S7 C file (from S7)
- **s7.h** - the S7 header file (from S7)
- **mus-config.h** - an empty file that is necessary to compile S7
- **scm/** - a directory with various Scheme files.
- **build_s7.sh** - a bash helper for building S7
- **s4m.xcodeproj** - the XCode project file, tweaked from a C74 example

Someone who knows XCode could tell me how to set it up to build S7 properly as a dependency, but as I don't, I'm just building the S7 object file and then linking it. build_s7.sh does this too.

$ gcc -c s7.c

If it worked, you should now have **s7.o**. Now we can build from XCode. Let's open the xcode project file and verify the set up.

We need to edit the linker flags. Select **s4m** in the left hand toolbar, and look for the **Linking** section in the main window, with an entry called **Other Linker Flags**. Double click this to add Linker Flags. My linker flags are the following:

- -framework MaxAudioAPI
- -framework JitterAPI
- ${C74_SYM_LINKER_FLAGS}
- s7.o
- -L.
- -ldl
- -lm

The first three are for the C74 SDK, and the rest are linking in S7. With these in place, we should be able to build. Building will generate *a lot* of warnings about S7 pointers casts. I need to either figure out how to tell XCode it's ok or fix these, as I'm following examples from the S7 FFI, but they are harmless right now.

A successful build should build a new version of the external, by default in **Packages/max-sdk-8.x.x/externals**. If you want to change where this goes you can do so, it's somewhere in the XCode config labyrinth. Note that on Windows, you need to build the 32 and 64 bit versions separately by compiling with different targets.

If you plan on hacking on Scheme for Max, you'll want to get Max launching automatically on compile. To make the development workflow better, under **Product -> Scheme** you should see **max-external**. Choose **Edit Scheme** and the **Run** submenu. Select your Max.app as the executable to run. This will relaunch Max whenever you compile, so you don't have to close and open Max to load your new external whenever you make changes.

If you have previously installed the binary package, you may need to move it out of your Max package path to make sure you get the right external loading. The Python script **make-release.py** automates bundling up a build, putting the

tarball in the right place, and moving my SDK package out of view so that I can test a new build. **uninstall.py** just puts things back. You can alter these for your system if they seem useful, but be warned, it makes file system changes! This is *not* necessary for building, it's just a convenience tool to make testing releases more practical.

If you want to experiment further with S7, you can download the S7 source directly from CCRMA. It comes with a lot of examples of using S7 with C. If you want to add features to Scheme For Max, you'll probably want to do this. But all we need for simple compilation is **s7.c**, **s7.h**, and an empty **mus-config.h**.

I will happily improve these instructions with feedback, and add a section on VisualStudio if someone can help there. If you get a build going off the instructions, please let me know on the Google Group so I know someone has managed. If you need help, post there too.

There is a regression test suite in the patchers folder. Some of the stuff tested requires delays to avoid interacting tests. You might not get the same results as I do. If they ever pass, we are good. If they sometimes fail (or fail when you try to run them all) there is a good chance it's just execution time differences on your system - you can try running the subsuites separately, or even openin those and running the tests. I do make sure they all pass on both OSX and Windows prior to releases. Hopefully they can eventually be made robust enough to work everywhere. They allow complete regression testing from the click of one button though, so that's something!

CHAPTER

NINE

WHY S7 SCHEME?

9.1 About S7 and S74 Scheme

The purpose of this page is to provide an intro to S7 along with the rationale for choosing it, aimed at people with some Lisp familiarity. Those interested in Lisp related languages in general may find this interesting - regular users can skip it.

S7 Scheme is a minimal scheme designed specifically for embedding in music contexts, made by Bill Schottstaedt at CCRMA. It's the scheme engine in the Common Music and Grace algorithmic composition platforms, the Snd library, and the Radium editor. S7 is mostly R4RS Scheme, along with extras features chosen for the specific use case of scripting music and audio in a host application. In brief, it was chosen for Scheme For Max because:

- It has a feature set that maps well to our problem space
- It's thread-safe, allowing us to have multiple independent interpreters in Max patches
- It's fast, small, and easy to hack on and to embed
- It has a permissive (BSD) license, so it can be used in stand-alones and M4L devices

S7 was based originally on TinyScheme, and is similar to Guile. In addition to most of the standard Scheme linguistic features, it has a number of features also seen in Common Lisp and shared with Lisp relatives like Clojure. Like Clojure, it's a bit of a hybrid: the syntax is clearly Scheme, with a single name-space for functions and variables, but it does not have hygeinic macros using syntax-case or syntax-rules, preferring Common Lisp style macros, and it has various other features that aren't in all all Scheme implementations.

Features of note for our use case include:

- Scheme simplicity (one namespace, tail call optimization, etc.)
- Keywords
- Applicative syntax
- First class environments
- Common Lisp style macros
- Simple FFI
- Used in music (Common Music, etc)

9.2 S74 Scheme

The author of S7 intends to keep S7 as minimal as is practical. While great for ensuring it's small and easy to embed, this means the language is missing many convenience functions that users coming from Clojure or Racket might be used to. S74 is simply a layer of additional functions on S7 that is loaded automatically by Scheme For Max. The only reason it has its own name is so that documentation can indicate easily which functions are added to S7 by S74. S74 additions are implemented (so far) as Scheme code and can be disable if the user desires by altering the bootstrap files.

9.3 Advantages of S7

The remainder of this page goes into more detail on the decision to use S7 and its drivers, and may or may not be of interest to the reader.

9.3.1 Our Use Case

In order to evaluate whether S7's feature set is a good one for Scheme For Max, let's examine our use case. The goal here is to make building and extending music systems in Max easier, more flexible, and smaller (in code). I anticipate most users are going to be working in teams of one to a few people, and making many similar projects rather than working on one large and continually growing codebase. Given this, issues around onboarding new developers are not likely to be relevant. The code should be easy to learn, but only if this can be done without sacrificing power and expressiveness. We don't need to be accessible to the casual or beginning programmer - that niche is already well filled by the JS object. We want a system that enables a sufficiently motivated developer to express themselves as succinctly and flexibly as possible. They should be able to build a re-usable toolkit flexible enough to use in a wide variety of artistic contexts. And the resulting code should be small, with very minimal syntax, allowing us to embed and manipulate code in Max messages.

A further thought is that in music hacking, we have two very different creation times: creating the system vs. creating the artistic product. When we are creating the system, we want the most advanced tools we can get our hands on, as the semantic and grammar problems of music are, compared to business applications for example, rather fierce. (I spent 15 years doing business application development, for what that's worth…) On the other hand, when we are creating the artistic product, we want the code to "just get out of the way". Any barrier to expression slows down the artistic process. This is, in a nutshell, the reason advanced languages like Lisp and Haskell are great for music, the developer can used advanced tools to make a very smart engine, that is then "played" with a high level and expressive domain specific language.

Ok, with that pontificating out of the way, let's look at the features of S7 and how they fit in with our needs.

9.3.2 Scheme Simplicity

Being a Scheme, S7 has a single name space for functions and variables, tail call optimization, and named let, among other things. In my personal opinion, these lead to recursive code that is smaller and easier to read than Common Lisp, and make it simpler to write dynamically evaluated code in Max messages. In many ways, S7 Scheme is a Scheme implementation with a bunch of Common Lisp's best features baked in. Whether this is a *good* thing I think depends on your taste and use case. I happen to believe that everything I've seen in S7 makes sense for our use case. As discussed above, we're after small, clear code, that can be changed quickly and expressed with the simplest list of tokens, even if one must have a fair degree of language familiarity to read it easily. Scheme's recursion idioms fall in this category. We can point the user at "The Little Schemer", and when they come back, they'll be happily writing tiny code.

9.3.3 Keywords

Keywords (in Lisps that support them) are symbols that always evaluate to themselves. In S7, these are symbols starting with a colon. Keywords are really useful in situations where one is doing a lot of dynamic evaluation, as we are doing when we pass over the Max-Scheme boundary. We need never worry about whether we ought to be quoting a keyword, or what it will become after evaluation, it just stays a keyword. In most places (aside from object scripting names), Max is happy to accept keywords as symbols, and the resulting Max code is readable because it's clear at a glance what a message with a keyword will be when in Scheme.

9.3.4 Applicative Syntax

Applicative syntax allows us to use data structures in the function slots of s-expressions, with arguments to get their contents. In S7 we can do this automatically with lists, vectors, and hash-tables. This makes for some terse code, a good thing when we want to build things in Max messages.

```
(define (my-listener args)
   ;; output the first arg
   (out 0 (args 0))
(define my-hash (hash-table :a 1 :b 2))
;; applicative syntax with keywords:
(out-1 (my-hash :a))
```

9.3.5 First class environments

In S7, environments are first class objects, meaning we can easily capture lexical bindings and pass them around. Again, we're doing a lot of dynamic evaluation as we cross the boundary, so it's helpful to be able to have this degree of control. It will also be useful as we add more objects to Scheme For Max, with the ability to control their executing environment, potentially sharing them or passing them between environments. Environments are also used in S7 to prevent hygeine issues in macros.

9.3.6 CL Macros:

S7 supports Common Lisp style macros, with **defmacro** and **gensym**, and some other more advanced CL style macro options (see the S7 docs). It does not do syntax-case or syntax-rules Scheme/Racket style macros. While it might be nice to have syntax-case and/or syntax-rules as well, if only for uptake from the broader scheme community, I think the Common Lisp style makes more sense for our use case. They are arguably easier to learn, and can arguably be more powerful, at the cost of some chance of hygeine issues. For easy creation of DSLs, on small code bases, in small teams, I think defmacro is a strong approach. They are also used in Clojure, Common Lisp, and Elixir, from whom I hope to attract users. Also with S7's first-class environments, macro-hygeine is not a big concern - details on this are in the S7 reference page for the macro enthusiast.

9.3.7 Simple FFI and embedding

Embedding S7 is a snap: there's one C file to build and link, and one header file to include. And the S7 FFI is really easy to use. It's simple to define Scheme functions from C, and to build Scheme expressions to hand to the functions in C. This has proved tremendously helpful for the rather cumbersome tasks of taking Max's concept of a list of atoms and converting it to a valid Scheme sexp and vice versa. Further, one of the project goals is to enable people to hack on Scheme For Max itself, and after looking at the embedding story for Chicken, Chez, Gambit, and Racket, I realized this would make a big difference to the accessibility of the code.

9.3.8 Use in music

Another big advantage of S7 is its use in music circles already. Common Music, by Heinrich Taube, is a very mature algorithmic composition toolkit with a rich library of music related functions that will be helpful to users. This library depends a lot on S7's lisp-y-ness, leaning heavily on various Common Lisp macros. Prof. Taube also has an excellent book, "Notes From The Metalevel", with a wonderfully clear intro to Scheme and Lisp specifically with musical examples that will be applicable to Max. One goal of the project is to ensure that the learner can use this book as a reference, and that people using Common Music outside of Max can easily port code to a Max environment. Common Music's application, Grace, is built with the JUCE toolkit, and examples on this are available. I felt this would be very valuable in encouraging adoption in music hackers.

9.4 Disadvantages of S7

Of course, S7 has some disadvantages too.

It's not well known, being used in a small specialist community. Adoption in the larger Scheme world has not been a priority of the project, as its purpose is specifically to enable advanced academic computer music. Community involvement in the development process has not been a priority, which one might consider a good or bad thing depending on your take.

The documentation is sparse, and is aimed at someone conversant with Scheme. The docs are definitely not suitable for someone just learning Scheme. On the other hand, they do include extensive examples of embedding in C and using the FFI, which was extremely helpful and which few others implementations had.

S7 is a bit of a hybrid, and thus does not try to exactly meet the various standards. It's mostly R5RS, some R7RS, but meeting R7RS small is not a goal of the project. This may mean that people looking for a general purpose scheme are going to prefer implementations working towards R7RS, such as Chicken or Guile.

On the whole, after looking at the options, I came to the conclusion that the advantages significantly outweighed the disadvantages for the stated goals of Scheme For Max. Documentation and community facing material can be written more easily than one can improve a Scheme implementation, and if development process becomes an issue, there is always the option of writing a thin layer on top of S7 to act as an abstraction between upstream S7 and the Scheme used in Scheme For Max. This approach has been taken successfully in many open source projects before, and may well make sense at some time in the future if we wind up adding features to S7 at the C level that are not of interest to the main S7 project. I don't expect this being necessary for some time as so much of Scheme can be extended from Scheme itsel, that I should be able to get a long way on library modules added to S7.

9.5 Other options that were considered

When beginning the project, I looked for quite a while at many Scheme and Lisp options. If you care about that sort of thing, you might find this interesting. These included:

- Clojure
- Guile
- Chicken
- Gambit
- Chez
- Racket
- Embeddable Common Lisp
- Clojure/ClojureScript

Below are my subjective thoughts in case anyone wants to try porting Scheme For Max to another Scheme. (I'd love to see the results if you do!) None of the others have extensive computer music usage (with the exception of Common Lisp) so I won't even list that as a disadvantage - take it as given.

9.5.1 Clojure

Clojure is it's own lisp-like langauge, that runs on the Java Virtual Machine. It has a lot of adoption, with the resuting rich library of books and documentation resources. However, it only runs on the Java Virtual Machine or on Node.js as ClojureScript. My first attempt at this project was with with ClojureScript on Node For Max. Overall, I really like the language, but the ties to the JVM make it unsuitable for embedding or for low latency native use. ClojureScipt on Node for Max works, but runs in a separate process that Max communicates with under the hood with serialization over a network connection. As a result, the level of integration possible is minimal compared to something embedded in a C external.

There are some native or compile-to-c languages influenced by Clojure that look promising, such as Ferret, Fennel, Janet, and Carp. However, none of them are as mature as S7 at this point, and would give us the disadvantages of S7 (sparse docs, small community size) without the computer music specific maturity.

9.5.2 Guile

Guile is a very nice minimal, embeddable scheme. It's easy to embed, is working towards R7RS, and has excellent documentation. It's under very active development, though mostly by one person, and it's pretty small. The main disadvantages of Guile are that it uses a less permissive license. If S7 weren't around, version 1 would probably be Guile. Like S7 Guile is just an interperter, making the FFI situation pretty simple.

9.5.3 Chicken

Chicken is a modern (R7RS) scheme that compiles to C. It's got great docs, a great module system with lots of libraries, and a really great community. It's popular as a scheme to build products in for business and scientific applications. Unfortunately, it's not multi-thread safe out of the box, and the FFI docs were minimal. The embedding situation is also not as simple as Guile or S7, as it's both a compiler and interpreter.

9.5.4 Gambit

Gambit is a large, mature, industrial strength Scheme that has been around for 30 years. It compiles down to highly performant C, and supports many kinds of macros. Gambit was another high contender. The big issue for Gambit was lack of documentation on embedding and a much more difficult embedding story. I will likely try to get Gambit working out of curiosity, there are some interesting developments in Gambit such as compiling to JavaScript, and it is very flexible with how it interacts with C.

9.5.5 Chez

Chez is similar to Gambit, in that it's a big, industrial grade Scheme, with high performance. Racket is being redone on top of Chez. Also like Gambit, the embedding and FFI story scared me off pretty quickly. I would use Gambit or Chez if I were starting a commercial product on Scheme and expected to be on it for a long time, with a large team, building tons of code. They are like the Common Lisps of Scheme, with Chez being R6RS, meaning it's got a lot in the language, by Scheme standards.

9.5.6 Racket

Racket was originally called PLT Scheme, and came out of the academic side of scheme. It has a very advanced macro system, and fairly accessible syntax. Unfortunately, it changes a lot (currently being redone on Chez) and embedding is not a high priority. Pretty much the only priorities for Racket are language research and teaching. Embedding can be done, but it looks very cumbersome. Racket also does not support Common Lisp macros with defmacro, which I wanted. It does have fantastic documentation though!

9.5.7 Gerbil

Gerbil is a modern "opinionated" (their words) Scheme built on top of Gambit. It does not support defmacro macros. It is aimed at system programming primarily. By the time I looked at Gerbil closely, I was pretty sure I wanted defmacro. However, tt does look worth looking at more closely for other Scheme uses.

9.6 Final Thoughts

Overall, after several months of evaluation, S7 was elected. Four months in, I've been happy with the decision. The ease of the FFI has been a huge win, and I'm really looking forward to supporting Common Music on Max and making it possible to run the code in "Notes From The Metalevel" on Max. The docs meant there have been times where it took a while to figure stuff out, but the code comes with a lot of examples, and Bill has been helpful with questions on the CCRMA CM Dist mailing list. The language design continues to grow on me.

Over the next months, I plan to add a resources page and a crash course page to these docs to help people learn S7 for Scheme For Max, and hopefully can introduce more people to the language.

If this document piques interest in others who wish to try something similar or port the project to another Scheme/Lisp, I would certainly be interested in communicating. Come join the Google Group!

Enjoy! Iain Duncan Victoria BC, Apr 2020.