Scheduling Musical Events as Procedures with Scheme for Max

IAIN C.T. DUNCAN, University of Victoria, Canada

We are confident that scheduling musical events as Scheme procedures using Max/MSP and Scheme for Max is the bomb.

CCS Concepts: • **Applied computing** → **Sound and music computing**; **Media arts**.

Additional Key Words and Phrases: music computing, event scheduling, live-coding

ACM Reference Format:
Iain C.T. Duncan. 2021. Scheduling Musical Events as Procedures with Scheme for Max. 1, 1 (June 2021), 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Max (also known as Max/MSP) is a programming environment for creating interactive music and multi-media programs through a visual programming language accessible to non-programmers. Programs, or "patches" in the Max nomeclature, are created by placing visual boxes representing Max objects on a canvas and connecting them visually with "patch-cords", a paradigm similar to that of modular synthesizers and familiar to many musicians. First created in the mid 1980's by Miller Puckette while at IRCAM (Institut de recherche et coordination acoustique/musique), Max is now developed and sold by the San Francisco software company Cycling 74, and is widely used in both academic and commercial music contexts as well as in multi-media installations. A rich library of Max objects exists, both provided with Max and available as open-source extensions, enabling users to rapidly create interactive systems with gestural input from physical sources such as on screen widgets, physical MIDI controllers, and custom hardware communicating over serial networks.

One powerful feature of Max is its ability to be programmed while the engine is playing music. Patches can be altered in without necessarily interrupting patch activity (depending on the design of the program), and this can even be performed live, an activty known as "live-coding", which has given birth to an entire musical sub-culture of live programming performances with tools such as Max, SuperCollider, Csound, and others. This approach to music making overlaps with the related discipline of algorithmic music, in which programmatic algorithms are used not just for affecting musical parameters such as volume, pitch, and timbre, but also for the generation of musical content such as melodies and rhythms.

In both the fields of live-coding and algorithmic music, the ability for the performer/composer to schedule events in the future with high temporal accuracy is of major benefit. While this is possible in the Max visual patching language, it is cumbersome and thus the implementation is not optimal for live-coding or algorithmic music where the speed with which the user can

Author's address: Iain C.T. Duncan, iainctduncan@gmail.com, University of Victoria, , Victoria, BC, Canada,

XXXX-XXXX/2021/6-ART $15.00
https://doi.org/10.1145/1122445.1122456

accomplish this is an important consideration (i.e., fast enough not to bore an audience or to be frustrating while composing).

In addition to the Max visual patching language, Max can be extended by creating new Max objects ("externals" in the Max nomenclature) in C or C++ using the Max SDK and API, and can also be programmed in text-based languages through regular Max objects that themselves provide embeddded language interpreters. Max provides one such interpreter in the **js** object, which embeds a JavaScript interpreter, and others contributed by third parties exist for languages such as for Lua, Python, and Ruby.

Scheme for Max (a.k.a. S4M) is an open-source external developed by the author that embeds the s7 Scheme interpreter, a Scheme dialect created by Bill Schottstaedt at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. Originally developed from TinyScheme, s7 is a Scheme dialect designed for use in computer music platforms and is used in various music programs such as the Snd editor and the Common Music algorithmic composition platform. Similar in functionality to the built in **js** object, the **s4m** object enables the user to program Max in Scheme, and provides Scheme functions to interact with the Max engine and environment through a foreign function interface implemented in C and using the Max C API.

The author believes that S4M extends Max in a way that is highly valuable to the algorithmic musician and live-coding performer and is also of significant use to the broader Max community. The ability to update a running Scheme program during playback is a major benefit and is a capability that has been, while technically possible, impractical with pre-existing solutions such as JavaScript. Additionally, the syntax of Scheme bears a convenient similarity to Max message syntax, and thus one can even use Max visual widgets and messages to generate small Scheme programs in the patcher. However, the most interesting benefit is the ease with which the user can create and schedule functions for evaluation in the future, with input into these functions coming from Max patching widgets, and flexible control over the current and future evaluation context of the functions and variables used.

This paper provides an overview of the problem of musical scheduling in computer music, and of the Max platform and its solutions to this problem, along with a discussion of the limitations of these solutions. It then introduces Scheme for Max and examines how the use of Scheme overcomes these limitations elegantly, and provides details of the implementations of scheduling in S4M, both at the Scheme and C programming levels. Finally, it concludes with discussion of future possibilities and directions of study with Scheme for Max.

2 BACKGROUND - PROGRAMMING MUSIC IN MAX/MSP

2.1 The Max Environment

Max is a visual programming environment for interactive multi-media in music academia, as well as in commercial music circles through "Max for Live", a version of Max embedded in the Ableton Live digital audio workstation. Max was originally created so that composers of interactive computer music could change their pieces without the assistance of a programmer. As previously mentioned, Max patches are created by placing visual boxes on a canvas and connecting them graphically with visual patch cords, where a box may be any of the Max object types installed on the user's system. A box placed on the patcher results in the instantiation of an object in memory from a prototypical class for the object, with text fields typed in the visual patching box used as constructor arguments. Thus a Max patch consists of a collection of instantiated objects that send messages to each other in a directed graph, producing a data-flow execution model whereby a message from a source object triggers execution in one or more receiving objects, who may in turn send on messages to other similarly connected objects.

Patch activity, in the form of messages moving through the graph, can be initiated by various forms of real-time input, such as keyboard and mouse events, MIDI input, and networking events, as well as by scheduled events through Max objects such as the metronome, which sends out messages at regular time intervals. Messages are stored internally as lists of Max atoms, which may be symbols, integers, or floating point numbers. There also exists a Max object for visually displaying and altering messages called the message-box, which allows messages to by typed directly in to object in the visual patcher, and sent by clicking the box. Execution follows a depth first and right-to-left order, enabling the programmer to deterministically control the execution flow with the visual layout of the patch cords. (i.e., A source object sending messages out to multiple receiving sub-graphs results in the right hand message path completing execution before moving left, rather than spawning two concurrent threads of execution.) When patching, messages can be inspected by sending them to a print object (which prints to the Max console), to a message-box object (which will update its visual display of the message) or though a built in debugger using a feature Max calls "probing".

In addition to the event-based message execution model, Max also supports a stream-based digital audio execution model, originally provided separately as a product called MSP, but now included as part of the single Max product. MSP activity normally runs in a separate thread from the event and message Max operations, and uses a a separate class of objects that pass constant streams of digital audio to each other through differently coloured patch cords, though MSP objects may additionally receive messages for controlling paramaters. As S4M executes only the event/message level and does not implement DSP operations, MSP is not discussed further here.

2.2 Max Message and Object Implementation

Internally, Max messages are data entities consisting of a symbol that acts as a Smalltalk-style message selector and an optional array of Max atoms. Each atom entity contains a member for the atom type and another for its value, where the type may be any of integer, float, or symbol. Note that while the message selector symbol is always present at the C level, in the visual patcher the selector may be implicit and hidden from the user. (i.e., the message originating from a message box that appears to contain only an integer will actually consist of the selector "int" followed by an atom storing the numerical value.) The symbol "bang" in Max is a special symbol that can be used for a one-element message that essentially means "run", and the act of triggering execution by sending a bang message from the **bang** object is called "banging" in the nomenclature. Taken together, this means that in Max there are five kinds of messages: bang, symbol, int, float, and list. (The bang message is technically a symbol message of "bang", but this is essentially treated as a type of its own in the nomenclature.)

Activity in an object (represented visually by a single visual box in the patch) is triggered by sending the object a Max message, most commonly from an object connected to it through a patch cord running to an "inlet" of the object. In order for the receiving object to do anything, it must be sent a message with a message selector for which it has a bound method, a situation referred to in the nomenclature as "responding to the message". Most objects have a principal activity that typically ends with outputing the result of their calculation, and this is often triggered by sending the object a single bang message or by sending a value to their first, or “hot”, inlet. In addition, they may respond to other messages to change internal state data or configuration, for example the “set” message is commonly implemented to update internal state without outputting any result. The end result of object methods run on receipt of a message falls broadly into three (non-exclusive) categories: the object may update some internal state, it may send a message or messages out of its “outlets”, and it may cause a side effect in the broader Max environment, such as printing to the console or updating a global data element such as an audio buffer.

As an example, in the patcher screen-shot below we see a **message-box** object that will update its state (and visual display) with the symbols “hello” and “world” when sent the message “set hello world”. Near it is a circular **bang** object, which when clicked will send the **message-box** the “bang” message, causing it to output the message “list hello world”. This message will be received by the **zl.len** object, which counts the number of elements in any lists it receives, immediately outputting the count. The three columns in the screen-shot show the patch as it is prior to any clicks, as it is after clicking message-box 1, and as it is after clicking both message-box 1 and the bang.

Objects are not limited to interacting with other objects through messages passing in to inlets and out of outlets. A C API exists that enables objects to query and control various engine components (e.g. the transport mechanism), and it is also possible for objects to send messages to other objects directly or through the scheduler queues, without the sending and receiving objects necessarily being connected visually in the patcher. Fundamentally however, the same mechanism is used - when an object calls a method on another object at the C level, this is still done by creating a data structure of a message selector and optional atom arguments, and then sending this to the receiving object through a generic message sending function.

As each visual box in a patcher has state that is retained between messages, we can see that when programming the visual patcher (a.k.a. “patching”), we are in effect programming with object instantiations rather than classes. In fact, in the Max engine, the act of creating a new visual-box and placing it on the patcher canvas does indeed instantiate an object, creating a fresh copy of the prototype’s data structure and adding it to a graph of other objects. This is in contrast to computer music languages such as Csound, where the user programs instruments with functions and the engine creates an instantiated data structure on each note-event sent to the synthesizer - in effect the instruments act as object builders and note-events become objects that exist for the duration of the note played.

2.3 Max Externals

While Max is a commercial, closed-source product, it includes a software development kit for extending Max by writing a Max “external”. An external is a compiled plug-in that defines the prototype class (data structure and methods) used to create new objects in the patcher. Externals are developed in C or C++ in an object oriented manner, the C API using data structures and pointers to simulate class-based programming and the more recent C++ API using C++ classes. A typical external will implement a class that provides some object state for instantiated objects along with constructor and destructor functions, and methods for receiving and sending Max messages through the object’s inlets and outlets.

As externals are compiled plug-ins, they do not need to be distributed as source-code, but can be made available as binaries for Windows and OSX. Extending Max through externals has been possible since very early versions of Max, and thus thousands of 3rd party externals now exist, both open and closed source, of which Scheme For Max is one.

2.4 Lisp in Computer Music

Scheme For Max is far from the first Lisp-based computer music tool, or even the first real-time music tool in Lisp. There is a rich history of Lisp in music, ranging from Common Lisp Music, created by Bill S. in the 1980's, to more recent systems such as Common Music 3 Nyquist and Extempore (formerly named Impromptu). Nor is Scheme for Max unique in providing a Lisp based extension facility to a commercial music platformk - the author unknowingly programmed in Lisp in the early 1990's while using the Cakewalk MIDI sequencer, which came with an embedded Lisp interpreter in the form of the Cakewalk Application Language (CAL).

However, Scheme for Max is unique in bringing a fully-fledged Scheme interpreter to the Max environment as a first-class Max object. This is significant in that Max is arguably the most widely used truly programmable music platform, and certainly the most widely deployed through its use as "Max For Live" in the very successful "Ableton Live" digital audio workstation.

In contrast with S4M, systems such as Common Music, Nyquist, and Extempore all run as stand-alone applications, with events originating from them often destined for other systems for translation into audio. While these systems are, like Max, also capable of receiving gestural input, they must, in essence, "be their own boss" - they provide their own scheduler and timing clocks and act as the principal engine from which temporal events originate. If they are to be used alongside another temporal engine, the two must be synchronized and events sent back and forth over network connections of some kind.

There have have also been previous externals created to allow one to use Lisp in Max. However, these have been proxies to external Lisp processes rather than interpreters embedded in a Max external. This difference means that in contrast to these offerings, the S4M interpreter has full access to the Max C API and can call API functions through foreign function interface calls, with the results of said calls being exactly the same as if the functions were called in C. The ramification of this is that S4M is unique among Max Lisp projects in being able to operate within the native Max scheduling system, enabling highly accurate timing and retaining deterministic control flow within a patcher. (i.e., There is no hidden networking layer with communication to an outside process, thus making calls actually asynchronous.) This provides the user with opportunities to use Scheme for tasks beyond what is possible with pre-existing options. For example, we are in effect able are able to say "do this Max-internal thing based on other internal Max state at such a time, as determined by the Max scheduler". An examples of this might be to control the Max transport mechanism based on activity in a table that is used as a shared data store by various patcher objects.

3 IMPLEMENTATION OF THE SCHEME FOR MAX EXTERNAL

In this section, a high-level overview of the implementation of the S4M external is provided. The S4M external includes in its data structure a reference to the s7 Scheme interpreter, which is available as two C files (.c and .h) incuded in the S4M source. The s7 API enables one to define new Scheme functions in C (for Scheme-to-C calls) and to call any Scheme code from C (for C-to-Scheme calls). The S4M external's data structure includes a reference to the s7 interpreter, and during the initialization process, a Scheme variable is created that holds a void pointer to the S4M object itself. These enable any code that is called from C or Scheme to find both the Max object reference and the running s7 interpreter.

Scheme-to-C calls are accomplished by defining a C function that takes a pointer to the s7 instance as an argument, followed by a second s7 pointer that is a reference to the arguments to the function as a Scheme list. In C code, Scheme wrapper functions are used to get arguments from this list, do work, and then a new s7 pointer is created that is used to returning the value to s7. This function is bound in a call in the S4M setup function.

C-to-Scheme calls

Below is an example of C-to-Scheme call that originates from a message sent to the Max object.

To evaluate Scheme code that is sent to the Max object dynamically, two mechanisms are use that is sent to the Max objectd

4 THE IMPORTANCE OF SCHEDULING EVENTS IN COMPUTER MUSIC AND MAX'S IMPLEMENTATION OF SUCH.

Music is fundamentally a temporal art form - there may be music with static pitch or static amplitude, but absent rhythm, there is no music. Thus a core problem in computer music platforms is that of providing a flexible means of triggering events in time. Further, in platforms intended to support both live and algorithmic musical interaction, a viable solution must enable the performer to interact with scheduled events satisfactorally, both programmatically and gesturally.

One of the major advantages of Max compared to other computer music platforms is the ease with which the user can create interactive environments in which the performer's actions change musical parameters in real-time in complex ways, enabling musically rich performances. It is, for example, trivial to add GUI elements to change musical parameters, and only slightly less trivial to add handlers for MIDI input, so that users can connect to their programs physical devices such as piano-style keyboards, and mixing board knobs, faders, and buttons.

For performers and composers exploring the intersections of live performance and algorithmic composition, one is ideally able to use gestural inputs not just to affect what is happening now (as with a traditional musical instrument), but also what will happen in the future. For example, a performer may have two physical dials, and may wish to schedule an event in the future that will use parameters derived from these dials, but may wish one parameter to be used as the dial is now (at the time of event dispatch) and the other to be used as the performer will have the dial at the scheduled time. Further, the absolute time of the event may not even be known at the time of triggering if the event time was specified in musical terms (i.e. the down-beat of the next 8 bar section) and the tempo is variable.

This problem is one not well solved in the Max environment prior to Scheme for Max. Max does provide facilities to delay Max messages by some amount of time. In the context of visual patcher programming, users can schedule a Max message (number, symbol, or list of both) for the future by sending it to a 'pipe' object. The amount of time by which it is delayed is specified as an argument of (or message to) the pipe object, and can be expressed in milliseconds or in a tempo relative format with the actual time determined by the Max master transport tempo. The pipe object's delay time time can be set dynamically by sending a numerical message to the right inlet of pipe, and messages sent to the left inlet will then be passed out the pipe outlet(s) after the specified time.

While moving events in to the future with the pipe object functions adequately, it has several limitations. The most immediately noticeable is that it also splits list input messages into individual elements, sending them out individual outlets. If the user wishes to delay a complex event with many parameters, something easily expressed in list format, it thus requires the user to specify how many atoms will be in an incoming list message in advance and to reasssemble the message manually. If the length of the list is unknown, the solution is more complex and requires an exterior storage mechanism be used to allow the outgoing pipe message to refetch the original list from another object after delay. Using pipe for delaying complex operations is thus cumbersome at best.

A larger problem is that of how to capture gestural values for use in the delayed events. Max's visual patching language is fundamentally modelled similarly to a modular synthesizer - there is one instance for each visual object, and messages can only pass through them one at a time. Creating visual programs where execution of delayed events will trigger cascades of messages through objects that are also being used in the interim becomes notoriously complex and there exists no straight-forward facility to express "make a new container for this variable as it is now so that we can fetch it later". A naive solution would use a visual pather object to store the variable's state at trigger time for use later, however this only works if there is only one scheduled event - adding more scheduled events requires adding additional storage objects for each event. This programming-with-instances paradigm is very convenient when we want to ensure there is only one of a given function, such as in the design of a classical monosynth, but difficult in a poly-phonic time context such as the creation and eventual playback of some unknown number of delayed events.

5 COMPARISON WITH THE MAX JAVASCRIPT OBJECT

We can see that the visual patching language of Max is not well suited to implementing this use case. Max does provide an alternative programming solution through its 'js' object, which embeds a JavaScript (ECMA5) interpreter in the Max environment. The js object is functionally very similar to S4M, and in fact, the desire to work with Max this way, combined with disatisfaction with the limitations of the js object was the impetus for development of S4M. The user is able to load JavaScript code in the js object from a file given as an argument to the object, and this code can interact with the Max environment and with object inlets and outlets, similar to how one can in C code used to develop a Max external. This does allow one to create functions and variables, and technicallycould be used to solve our use case described above.

However, the js object is not an ideal solution for several reasons. The most serious is that it is limited to running only in the Max low-priority GUI thread. As this thread is used for file i/o and graphic redrawing, the result is that latency is high and timing accuracy is unpredictable and often poor. The human ear is very sensitive to time, and perceives gaps of 20ms as discreet events. Delays of tens of ms are more than enough to sound like errors in playback of highly rhythmic music. This means that it takes little other activity in the GUI thread to delay our scheduled events enough to be audibly incorrect. For some purposes, this is acceptable, but for the creation of accurate sequencers or algorithmic playback devices, a js based solution is not usable.

Secondly, the js object provides no convenient facilities for loading new code during playback. The object reads in a single source file at instantiation time. One could technically add dynamic code evaluation to the JS object by wrapping code to be evaluated in strings and passing this message to a JavaScript function that uses the JavaScript "eval" function. However, compared to dynamic code evaluation facilities in Scheme, this is again cumbersome. This is of interest to algorithmic performers, for example in the "live-coding" oevre, who might want to create a new function and schedule while a piece plays.

Finally, the js object gives us JavaScript's implementation of anonymous functions and closures, which while functional, are verbose in terms of syntax, and require punctuation that is not easily used in Max messages. When compared to the syntax of Scheme, JavaScript is thus not appropriate for generating code in Max messages.

In contrast, Scheme syntax lends itself well to expression in Max messages. Both Max and Scheme use whitespace as token separators, and the punctuation marks that have significance in Max messages ($,) are easily avoided in Scheme. (Note that this restriction applies only to Scheme code built in Max messages, collisions do not happen at all for code loaded from files or strings.)

The advantages of being able to build Scheme code in Max messages themselves is that it becomes very simple to add functional input to a Scheme system, even while it is playing. Max provides a facility whereby messages can act as templates, with the $ sign used to indicate placeholders. This allows one to attach an input element, for example a dial, to a message-box and have the box output the results of template interpolation whenver ever the dial is moved. The example below shows a Max and S4M patch in which a Max dial widget updates triggers a message of "set! foobar dial-setting", which is then passed on to an S4M object for evaluation. One of the convenience facilities provided by S4M is that of treating unhandled messages as Scheme expressions as if they were wrapped in parantheses, enabling easy creation of Scheme one-liners in Max messages. Thus the patch below is all that is required to capture input from a dial widget and update a variable in the Scheme environment.

While the above is convenient to the visual programmer, S4M is not limited to one-line code. Should we wish to handle code in Max messages with complete Scheme syntax, we must only convert the message to a quoted symbol and pass the resulting string to the S4M object as the body of an 'eval-string' message.

6 SCHEME FOR MAX AND SCHEDULING EVENTS

Given the above, we can see that the problem of elegantly implementing a delayed event system that can differentiate between current and future values for variables representing input gestures is one perfectly suited to expression in S4M. Scheme is notable for the ease and conciseness with which one can create an anonymous function and store a reference to this function (taking along its environment) in some data store to be retrieved an arbitrary point in time. Further, it is straightforward to differentiate in the function between variables that should be used with their value as they are at function definition time versus as they are at eventual evaluation time. The underlying host system is required only to implement the ability to execute a callback at some time in the future, where the callback is able to retrieve some indicator of the correct function stored. Scheme for Max brings this facility to the Max environment, enabling users to dynamically define functions during playback of a piece, and schedule these in musical terms for execution in the future.

6.1 Clock Implementation in Max

To discuss the implementation of scheduled functions in S4m, we must first show the implementation of objects in C and the facilities for delaying functions.

At a high level, a Max external for message handling must implement the following:

- a data structure to hold state used by the object
- a main function used to create the class in C
- a new function that acts as a constructor for instances
- any methods that will be bound to messages as event handlers

A sample of a minimal external that holds an integer and updates the integer on receipt of an 'int' message is below.

```
// simplified example from the Max SDK documentation

// data structure for our class, contains instance data
typedef struct _simp {
    t_object s_obj;     // member for the actual instance
    long s_value;       // state variable for the integer
} t_simp;

// global pointer to our class definition that is setup in ext_main()
static t_class *s_simp_class;

// ext_main, the setup function that builds the s_simp class
void ext_main(void *r){
    t_class *c;
    c = class_new("simp", (method)simp_new, (method)NULL, sizeof(t_simp), 0L, 0);
    // bind handlers for the messages we want to be able to receive
    class_addmethod(c, (method)simp_int, "int", A_LONG, 0);
    class_addmethod(c, (method)simp_bang, "bang", 0);
    class_register(CLASS_BOX, c);
    s_simp_class = c;
}

// constructor for our object
// the value returned gets stored in the s_obj field of our t_simp struct
void *simp_new(){
    t_simp *x = (t_simp *)object_alloc(s_simp_class);
    x->s_value = 0;
    return x;
}

// a method handler for int messages that updates the internal state
void simp_int(t_simp *x, long n){
    x->s_value = n;
}

// a method handler for bang messages, post the internal in to console
void simp_bang(t_simp *x){
    post("value is %ld",x->s_value);
}
```

We can see that the pattern for adding functionality to our class is to add state variables to the t_simp structure, and add methods that are functions expecting a pointer to our object as the first argument, normally named 'x'. In the example above, a 'bang' message causes our object to run, with the side effect of posting the stored value. A more realistic example would likely output the stored value, but this adds more code and is not necessary for our demonstration of scheduling.

Let us imagine instead that the 'bang' message should delay the activity (posting) by 1000ms. We can use the Max clock facility for high-accuracy delay. (An older, and more flexible, facility exists as well but as it is less accurate temporally, it is not used in S4M)

We create a clock object, which takes as arguments a pointer and a function reference, with the pointer normally used to hold a reference to our instantiated object. The standard method for doing this in Max (taken from the SDK documentation) is to use a clock object that gets added to our data structure, and to add the act of starting the clock to the bang handler.

```
// Max standard clock use

// data structure for our class, contains instance data
typedef struct _simp {
    t_object  m_obj;          // member for the actual instance
    long      m_value;        // state variable for the integer
    void      *m_clock;       // will hold clock pointer
} t_simp;

/ update the constructor to make a clock
void *simp_new(){
    t_simp *x = (t_simp *)object_alloc(s_simp_class);
    x->s_value = 0;
    // create a clock bound to the simp_callback method
    x->m_clock = clock_new(x, (method)simp_callback);
    return x;
}

// update the bang handler to start the clock
void simp_bang(t_simp *x){
  clock_fdelay(clock, 1000);
}

// our callback that will run from the clock
void simp_callback(t_simp x){
    post("value is %ld",x->s_value);
}
```

From the above we can see that, while accurate, the clock functionality is limited - the callback must be a a single-arity function expecting a pointer, normally to the object.

6.2 Implementation in Scheme for Max

Scheme for Max builds on the clock facility provided by the Max API to allow scheduling Scheme procedures. These functions are limited to zero-arity signatures, but as creation of lambda functions in Scheme is trivial, this is of no practical significance to user.

To the user of S4M, scheduling the execution of a procedure is simple:

```
; schedule a function for 1000ms in the future
; it will send the int 99 out outlet 0
(delay 1000 (lambda()(out 0 99)))
```

In addition to scheduling the function, the delay function also returns a unique symbolic handle that can be used to cancel a delayed function before it runs.

```
; delay and store the handle
(define handle
  (delay 1000 (lambda()(out 0 99))))

; cancel it
(cancel-delay handle)
```

The Scheme implementation of this is straightforward:

- the delay procecure creates a unique symbolic handle, and stores the procedure in a hash-table, keyed by the handle
- it then calls an internal Scheme procedure, 's4m-schedule-delay', which is implemented in C using the s7 foreign function interface (FFI) and takes as arguments the delay time and the symbolic handle
- s4m-schedule-delay creates a clock, with a callback that will receive the handle
- when the clock callback runs, it uses the FFI to call the scheme procedure s4m-execute-callback, which also takes the symbolic handle as an argument.
- s4m-execute-callback, defined in Scheme, receives the handle, retrieves the procedure from the hash-table (deleting it in the process), and runs the procedure.

Cancelling a delay function consists of merely replacing the callback registered in the hash-table with the value #false, in effect letting the clock fire harmlessly.

The s7 Scheme code for this is show below. It uses an s7's 'gensym' function to create the symbolic handle that is guaranteed to be unique to this instance of the interpreter.

```
; internal registry of callbacks
(define s4m-callback-registry (hash-table))

; procedure to register a callback by a handle and return handle
(define (s4m-register-callback cb-function)
  (let ((key (gensym)))
    (set! (s4m-callback-registry key) cb-function)
    key))

; fetch a callback from the registry
(define (s4m-get-callback key)
  (let ((cb-function (s4m-callback-registry key)))
    cb-function))

; internal function to get a callback from the registry and run it
; this gets called from C code when the Max clock runs
(define (s4m-execute-callback key)
  ; get the func, note that this might return false if was cancelled
  (let ((cb-fun (s4m-get-callback key)))
    ; de-register the handle
    (set! (s4m-callback-registry key) #f)
    ; if callback retrieval got false, return null, else execute
    (if (eq? #f cb-fun)
      '()
      ; call our cb function, catching any errors here and posting
      (catch #t
        (lambda () (cb-fun)) (lambda err-args (post "ERROR:" err-args))))))

; public function to delay a function by time ms (int or float)
; returns the gensym callback key, which can be used to cancel it
(define (delay time fun)
  ; register the callback and return the handle
  (let ((cb-handle (s4m-register-callback fun)))
    ; call the C FFI and return the handle
    (s4m-schedule-delay time cb-handle)
    cb-handle))
```

The implementation in the C code for the S4M external is more involvedto work around the signature limitations of the clock facilities in Max.

When the s4m-schedule-delay function is run, receiving the symbolic handle and delay time as arguments, the following occurs:

- a data structure (t_s4m_clock_callback) is dynamically allocated and used to store a reference to the s4m object and to the handle
- a Max clock is created, passing in a void pointer to this structure and a generic clock callback, and the clock's timer is started
- the clock_callback struct is also stored in a C hashtable (Max's hashtab implementation), keyed by the handle, so that cancellation or object deletion can find all clocks
- when the generic clock callback function runs, it uses the pointer argument get the the delay handle and the instatiated s4m object, through which it can get the s7 interpreter pointer
- the interpreter and handle are then used to call the Scheme function s4m-execute-callback, running our Scheme procedure
- the callback then removes the clock reference from the C hashtable, deletes the clock, and frees the memory allocated for the callback info struct.

```
// the struct for the s4m object, with most elements removed
typedef struct _s4m {
    t_object obj;
    // pointer to the s7 interpreter (initialization of which is not shown)
    s7_scheme *s7;
    // a Max hash table for storing clocks
    t_hashtab *clocks;
} t_s4m;

// the clock callback struct
typedef struct _s4m_clock_callback {
   t_s4m obj;
   t_symbol *handle;
} t_s4m_clock_callback;

// schedule delay, registered in object setup to be run from Scheme as s4m-schedule-
static s7_pointer s7_schedule_delay(s7_scheme *s7, s7_pointer args){
    t_s4m *x = get_max_obj(s7);

    // get the arguments we need (time and handle) from the s7 args list
    // that represents the arguments passed to s4m-schedule-delay in Scheme
    // first arg is float of time in ms
    double delay_time = s7_real( s7_car(args) );

    // second arg is the symbolic handle
    char *cb_handle_str;
    s7_pointer *s7_cb_handle = s7_cadr(args);
    cb_handle_str = s7_symbol_name(s7_cb_handle);

    // allocate memory for our clock_callback struct and populate
    // NB: this gets cleaned up by the receiver in the clock callback above
    t_s4m_clock_callback *clock_cb_info = (t_s4m_clock_callback *)sysmem_newptr(size
    clock_cb_info->obj = *x;
    clock_cb_info->handle = gensym(cb_handle_str);

    // make a clock, setting our callback info struct as the owner, as void pointer
    // when the callback method fires, it will retrieve this pointer as an arg
    // and use it to get the handle for calling into scheme
    void *clock = clock_new( (void *)clock_cb_info, (method)s4m_clock_callback);

    // store the clock ref in the s4m clocks hashtab (used to get at them for cancel
    hashtab_store(x->clocks, gensym(cb_handle_str), clock);

    // schedule it, this is what actually kicks off the timer
    clock_fdelay(clock, delay_time);

    // return the handle to the Scheme caller
    return s7_make_symbol(s7, cb_handle_str);
}

// the generic clock callback, this fires after being scheduled with clock_fdelay
// gets access to the handle and s4m obj through the clock_callback struct
void s4m_clock_callback(void *arg){
    t_s4m_clock_callback *ccb = (t_s4m_clock_callback *) arg;
    t_s4m *x = &(ccb->obj);
    t_symbol handle = *ccb->handle;

    // call into scheme with the handle, where scheme will call the registered delay
    // we must build a Scheme list through the FFI to use as the arguments
    s7_pointer *s7_args = s7_nil(x->s7);
    s7_args = s7_cons(x->s7, s7_make_symbol(x->s7, handle.s_name), s7_args);

    // call the Scheme s4m-execute-callback function
    // s4m_s7_call is a simple wrapper around s7's s7_call with error handling and l
    s4m_s7_call(x, s7_name_to_value(x->s7, "s4m-execute-callback"), s7_args);

    // remove the clock(s) from the clock registry and free the cb struct
    hashtab_delete(x->clocks, &handle);
    // free the memory for the clock callback struct
    sysmem_freeptr(arg);
}

// the s4m object's free method, called by Max internals on object deletion
// it must cancel and free any outstanding allocated clocks
void s4m_free(t_s4m *x){
    //  ... various other cleanups trimmed ...

    // clocks must all be stopped and deleted.
    // iterate through the hashtab to stop them
    hashtab_funall(x->clocks, (method) s4m_cancel_clock_entry, x);
    // this will free all the clocks
    object_free(x->clocks);
}

// iterator used above to cancel a clock
void s4m_cancel_clock_entry(t_hashtab_entry *e, void *arg){
    if (e->key && e->value) {
        clock_unset(e->value);
    }
}
```

7 CONCLUSION

Well that was cool, I wonder how I should conclude such a thing???

My test citation [**?**] is here.

My test citation [**?**] is here.