Table of Contents

Abstract

Extempore is a live coding environment designed for procedural generation of real-time audio-visual experiences, where the programmer's code itself – written in Scheme and the Extempore-specific XTLang – is the user interface (Sorensen, 2018). In this project, the author experiments with using physical MIDI keyboard with Extempore, implementing behaviours such as MIDI arpeggiation, recording, sequencing and looping. An overview of the process of implementation is given followed by detailed analysis of correctness and performance, particularly as pertains to MIDI latency. Having become experienced in Extempore's use, design and philosophy through conducting the project, the author finishes with a discussion regarding and critique of Extempore as a development environment, concluding with some suggestions regarding further development – both of the MIDI functionality developed herein and for determining Extempore's long-term prognosis within the live-programming milieu.

1. Introduction

This research project experiments with implementing MIDI interaction within the Extempore live-coding environment, allowing physical input – such as traditional Western diatonic note input via an external MIDI keyboard – to control and interact with real-time, procedural music generation.

The first aim of the project is to build upon Extempore's existing, low-level MIDI implementations in order to provide a flexible, high-level interface supporting multiple concurrent MIDI devices and communication via virtual ports. Given this foundation, various potential applications of MIDI input are developed, with an eye to their smooth integration with the existing Extempore ecosystem.

Live arpeggiation – the playing of a broken chord, or set of notes, often in a set pattern – is implemented, utilising Extempore's built-in pattern language to allow live, run-time control over the algorithm. The ability to record notes and concomitant timing information, optionally playing back the resultant recording in a loop in the style of a guitar loop pedal (see Rudrich, 2017) is developed and demonstrated, as is more nuanced interaction with physical MIDI devices – such as using MIDI control change events and controller pad buttons to dynamically interact with executing programs and performances. Sequencing MIDI notes into time-quantised streams, able to be used in lieu of physical keyboard input, is implemented, as is an algorithm to convert recorded note events into the standard Extempore pattern language format in recognition of the primacy of *code itself* as the platform's user interface.

The project serves to provide a means to demonstrate a broad range of software development skills, across paradigms – from writing low-level, imperative code to interface with native system C libraries to producing high-level abstractions utilising higher-order functions and procedurally generated code in a dynamic, garbage collected and runtime modifiable language. Thought is given to the efficiency analysis of key algorithms in use, appropriate data structures are discussed and implemented and the end results of the project are subjected to testing of correctness and performance. Significant analysis is given to the issue of MIDI round-trip latency due to its effects on instrument playability (Dahl and Bresin, 2001), leading to insight into a poorly-performing aspect of the Extempore infrastructure and significant performance gains post-optimisation after refactoring the code to reduce its impact; ultimately, very low MIDI round trip times were achieved comparable with the lower limits achievable on the hardware in use.

Extempore is itself an experimental project with a unique approach to the problem of live coded audio-visual experiences – other than its use of Scheme, the key differentiating factor is that of XTLang, Sorensen's (2018) attempt to produce a low-level Scheme-like programming language capable of general purpose systems programming and interfacing with foreign C code in a run-time modifiable way. Sitting on this foundation, this project serves as a sounding board for the evaluating Extempore itself – both as a development environment and target and in rationale; does Extempore's design and implementation justify its raison d'etre of being a true "full-stack" live-programming environment? Particular focus is given to the necessity for XTLang, Extempore's major differentiating factor from previous systems (Sorensen, 2018).

The report concludes with a reflection on the learning and issues encountered by the author whilst undertaking this project.

2. Background

2.1 Live programming

Live programming – also referred to as “live coding”, “just-in-time programming”, “cyber-physical” or “on-the-fly” programming – refers to a programming paradigm featuring the central tenet that a program’s source code *is the user interface* of the system (Sorensen, 2013, p.23). *Liveness*, as described by Tanimoto (1990), describes the responsiveness of the system as its code – or graph, in the case of visual languages – changes; systems with a high degree of liveness feature programs which continue executing through source code modification, with committed changes immediately being reflected in altered execution.

Live programming as a concept is not new – programming Lisp through one of the standard interfaces provided by a Lisp interpreter, known as the Read Evaluate Print Loop (henceforth REPL), has been around since the implementation by John McCarthy of Lisp interpreters during the late 1950s and 1960s (McCarthy, 1978). A canonical example of its usefulness is provided by the story of the Remote Agent bug present on the Deep Space 1 mission of NASA’s New Millennium program, where a race condition, unforeseen in pre-flight testing, occurred whilst in space; it was fixed and debugged remotely using a Lisp REPL executing on the spacecraft (Garrett, 2002).

Sorensen (2018) differentiates between *half-stack* and *full-stack* live coding systems, reflecting the overall reach of the dynamically-programmable, run-time modifiable aspect of the system. For instance, the Impromptu audio-visual environment consists of a Scheme runtime – the interpreter for which serving as the infrastructure for any “live” programming – sitting atop a native C/C++ layer which is responsible for the real business of generating oscillator waveforms and interacting with audio hardware (Sorensen, 2018). This split is common throughout interpreted languages – for instance, consider Python, the reference implementation of which, CPython, outsources routines requiring high performance or involving system hardware access or I/O to native C library code – there is no way to implement this behaviour in Python itself (Python Software Foundation, 2020). R provides a further example of an interactive, live – but slow to execute – interpreted scripting language sitting atop a non-runtime modifiable but more performant C and FORTRAN foundation (Wickham, 2019).

Extempore is Sorensen's (2018) attempt to formulate a full-stack, live programming environment – that is, an environment able to span the divide from low-level hardware and system programming to high-level, dynamic and memory managed co-ordination (via Scheme), all in a run-time modifiable way. It is discussed in detail below.

A significant community in the present day concerned with the use of live programming is that built around live coded music. Several live coding platforms exist with audio-visual creativity in mind – such as Impromptu, Extempore's precursor (Sorensen, 2005), ChucK (Wang et al., 2015) and SuperCollider (McCartney, 2002), the server-side of which serves as a back-end for myriad other environments, such as Sonic Pi (Aaron S., 2016), based on Ruby and Overtone (Aaron and Blackwell, 2013), a library for Clojure. Procedurally generated music, coded on-the-fly, is presented at live coding concerts to an audience, generally involving the projection of the programmer-musician's screen (Brown and Sorensen, 2007); as an art form, the aesthetic impact is that of witnessing the creative construction of a musical piece given the restrictions of the medium – having to specify the algorithms in use on-the-fly, in functional code, whilst remaining within the time constraints needed to produce a pleasing performance for the audience (Cox et al., 2001).

2.2 MIDI

The Musical Instrument Digital Interface (MIDI) is a standard for communication between musical devices – primarily keyboards with digital synthesizers, but since adopted for use for a variety of uses, such as control of lighting and effects equipment – the first version of which was made public in 1983 (Rothstein, 1995). Roughly based on the existent idea of local area networks in computer networking, MIDI messages are traditionally passed through MIDI cables between non-hierarchically organised devices, many of which may be chained together.

MIDI messages are generally three bytes long, consisting of a status byte and two data bytes. For identification, the most significant bit of the status byte is a one and the most significant bits of the data bytes are zero; this leaves seven bits per byte available for carriage of useful information. After the status byte's initial one, the next three most-significant bits denote the message type (for instance, note on, note off or control change). The remaining four bits encode the channel, giving a maximum sixteen channels. The idea is that devices can be configured to listen to only events occurring on specific channels, allowing a simple, multi-cast network architecture – messages may be sent to all devices in the network and devices will disregard those not addressed to them in a way similar to how MAC addresses are used in non-switched Ethernet networks (Rothstein, 1995).

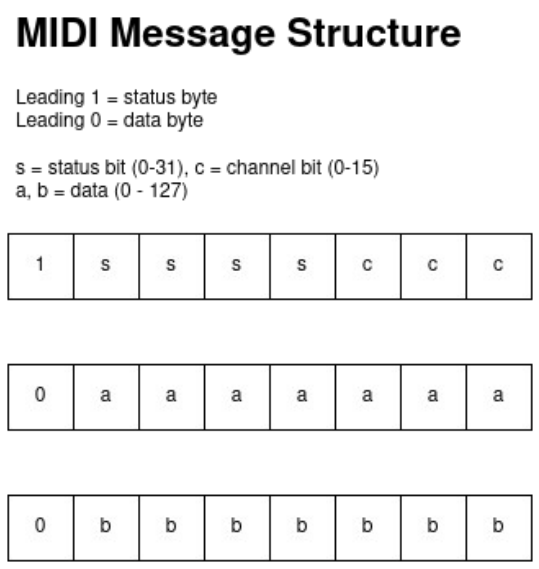


Figure 1: Structure of a typical MIDI message. Note that there are not always two data bytes; some status codes are followed by only a single data byte.

For use in simple musical messaging, the convention is to use the first data byte as the note value.

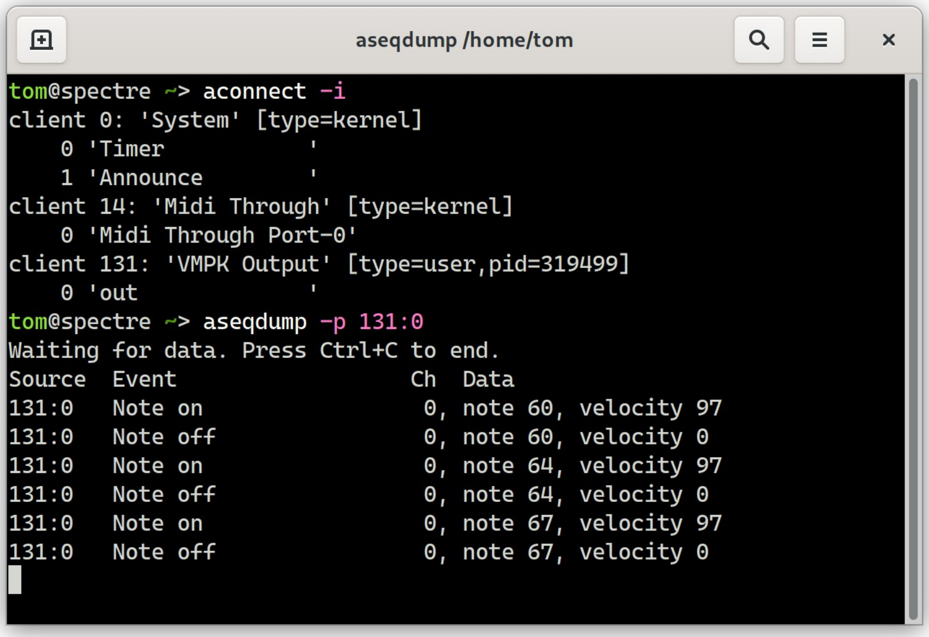
Notes therefore span from zero to 127, with middle C of a piano lying in the centre of the range at value 60.

The second data byte is used to encode velocity values, which may again range from zero to 127, with 127 representing maximum loudness (Rothstein, 1995).

A full breakdown of the various MIDI status messages can be seen as listed per the MIDI Manufacturer's Association (2020).

2.3 Linux MIDI infrastructure

Since Linux kernel 2.6, the default sound system in Linux has been that of the Advanced Linux Sound Architecture (hereafter ALSA), which provides raw audio input and output support for most audio interfaces from consumer to professional level; it also provides the ability to manage virtual (software) MIDI ports and their connections, routing both hardware- or software-generated events as desired through the virtual MIDI network (Phillips, 2005).



```
tom@spectre ~> aconnect -i
client 0: 'System' [type=kernel]
    0 'Timer           '
    1 'Announce        '
client 14: 'Midi Through' [type=kernel]
    0 'Midi Through Port-0'
client 131: 'VMPK Output' [type=user,pid=319499]
    0 'out             '
tom@spectre ~> aseqdump -p 131:0
Waiting for data. Press Ctrl+C to end.
Source  Event                  Ch  Data
131:0   Note on                 0, note 60, velocity 97
131:0   Note off                0, note 60, velocity 0
131:0   Note on                 0, note 64, velocity 97
131:0   Note off                0, note 64, velocity 0
131:0   Note on                 0, note 67, velocity 97
131:0   Note off                0, note 67, velocity 0
```

Figure 2: A demonstration of viewing raw MIDI events as they occur - in this case, the notes of a C Major chord - via ALSA.

JACK – a self-recursive acronym standing for JACK Audio Connection Kit – is a software layer providing low-latency audio and MIDI routing between applications; it is concerned solely with connection and depends on other software for actual audio input and output. A variety of back-ends are available – but generally ALSA is used (Phillips, 2005). JACK is an established part of the professional audio Linux landscape (Letz et al., 2004); there are, however, ports of the JACK server to MacOS and Windows, allowing it to serve as a cross-platform means of routing MIDI and audio streams between compatible applications.

A variety of utility software exists on Linux for interaction with and configuration of JACK, such as *Catia*, used for management of the audio and MIDI routing graph (KXStudio, 2020).

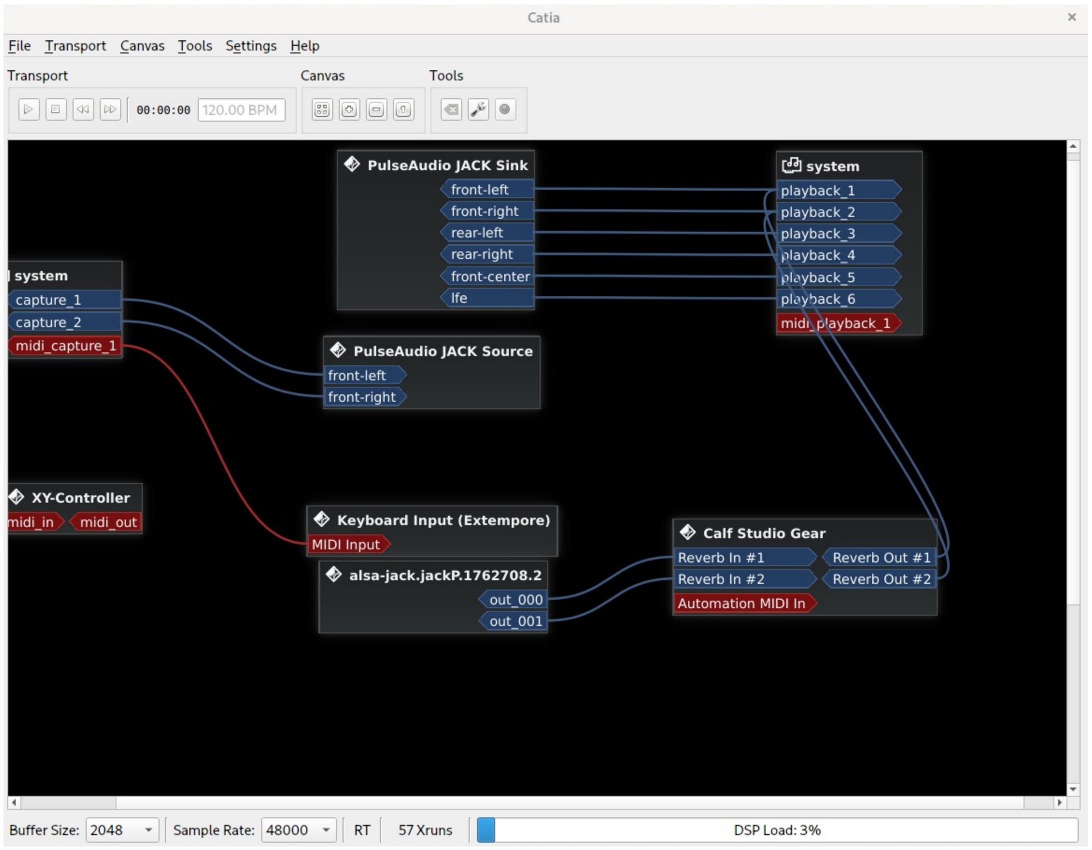


Figure 3: Catia, an open-source tool for managing the JACK audio and MIDI connection graph.

2.4 Scheme

Extempore's high-level control language is Scheme, a dialect of Lisp (Dybvig, 2009). Common to other languages in the Lisp family, it uses a simple, consistent syntax for expressions – known as *S-expressions*.

S-expressions take the form of either *atoms* – symbols, strings, procedures or other primitive data types – or composite structures denoted in parentheses in the form of `(A . B)`, where `A` and `B` are both S-expressions (McCarthy and Levin, 1985). The basic operation `cons` is used to form a new S-expression by conjoining two others, such that `cons(A,B)` produces the pair `(A . B)`. By repeating this process, large, composite linked lists can be produced, which are represented in code using the shorthand of parenthesised lists – for instance, the list `(+ 3 2 1)` is equivalent to `(cons + (cons 3` `(cons 2 (cons 1 ‘()))).` Lisp implementations generally use the *cons cell* as a fundamental data structure, used to store such pairs in memory; the linked lists formed by chaining these cells – by convention terminated with the empty list, in Scheme denoted by `()` – together are used to represent both code and data. This simple, uniform syntax allows Lisp programs to manipulate code as data in a way impossible in languages which have arbitrary syntax (McCarthy and Levin, 1985), a feature which shall be discussed further below.

A Scheme parser interprets the first element of an S-expression as an operator or procedure to be applied and the remaining elements as the arguments to *apply* to the procedure (Abelson et al, 1996). Before application, each of the elements are first evaluated, which may also involve procedure application if they represent nested S-expressions.

For instance, entering `(+ 3 2 1)` in a Scheme interpreter will return the value `6`. First, each element, `+`, `3`, `2` and `1` are evaluated; `+` refers to a built-in procedure for summation and `3`, `2` and `1` represent primitive integers, so each evaluates to itself. Then, the procedure referenced by the first element, `+`, has the arguments `3`, `2` and `1` applied to it, resulting in the value `6`.

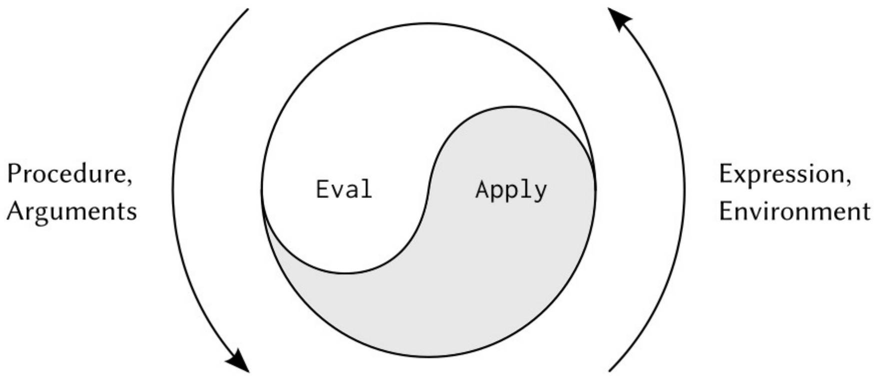


Figure 4: The yin and yang of the eval-apply cycle, taken from Abelson et al. (1996).

This process is referred to as the “eval-apply” cycle (Abelson et al., 1996, p.365). The piece of the puzzle yet to be elaborated here is that of the *environment*, which incidentally are first-class objects in Scheme (Dybvig, 2009). Environments contain variable bindings and may be nested; in this way Scheme achieves lexical scoping for variables. When a symbol is evaluated, its binding is looked up from the current reference frame outward, until the top-most binding frame is reached. This allows variable bindings to be shadowed – for the same symbolic names to refer to different values in different contexts – not unlike how variables in C-like languages may be bound in a block-specific way.

Scheme is a simple language with a minimal standard library and few built-in procedures yet, due to the rigorous and uniform syntax and first-class continuations, able to be adapted and expanded in a very fluid way to meet domain-specific needs (Dybvig, 2009). Key procedures include `cons`, `car` and `cdr`, which create a cons cell and retrieve the first element and second element respectively, `list`, used to create linked lists of cons cells, operations to mutate data – such as `set!`, `set-car!`, `set-cdr!` - and `let` and `define`, used to establish variable bindings. Procedures are defined using the lambda calculus inspired `lambda`, which creates an anonymous function. Branching is possible using the basic operator `if` and iteration is accomplished using tail recursion – where a procedure calls itself (or another procedure) as a *tail call*, defined semantically as a call having the same continuation as the continuation passed to the procedure containing the call. Implementations generally optimise tail calls into `jmp` instructions, thereby avoiding the stack growth often associated with recursion using procedure calls (Dybvig, 2009).

Continuations themselves – representing pending computations awaiting the result of a procedure application – are first-class objects in Scheme, accessed using the built-in procedure `call-with-current-continuation`, often abbreviated to the more concise `call/cc`. Access to continuations allows programmers to implement advanced control structures in pure Scheme, such as backtracking and non-local exits; in this way, it is similar to the `goto` mechanism in C, with the advantage that continuations take execution back to a previous, internally-consistent program state, which is not always the case with `goto` (Madore, 2002).

Scheme's use of S-expressions for representing both code and data makes it relatively trivial – compared to languages with arbitrary syntax – to write code which transforms other representations of code. The macro system in Scheme – accessed through the built-in `define-macro` or more powerful `syntax-rules` mechanisms can be used to transform domain-specific languages (hereafter DSL) into valid Scheme and are widely used by Scheme programmers to provide higher-level abstractions – and therefore shorter, more succinct code – than would otherwise be possible. Such transformations are used in Extempore's *pattern language*, for instance, which is an example of a DSL for the repetitive, rhythmic playing of a musical note sequence.

2.5 Extempore

Extempore is the result of Sorensen's (2018) PhD research into producing a full-stack live coding environment – a live coding environment equally suited to low-level systems programming as it is to high-level scripting and co-ordination.

Given the use case – executing code as it is typed and sent from an editor – the interpreter uses a server-client architecture whereby plain-text commands (in Scheme or XTLang, which will be described further below) are received, compiled (in the case of XTLang) or interpreted (in the case of Scheme) and thereby executed. This serves to de-couple the interpreter interface from the user's specific choice of editor; plugins are available for Visual Studio Code, Vim and Emacs.

Interaction with Extempore is done through two programming languages. Scheme is used for high-level control and co-ordination – and ultimately all executable code must be triggered by a top-level Scheme interpreter. Scheme, however, is a garbage-collected, dynamically typed language; not a natural fit for interacting with systems-level processes which are often heavily reliant on the C ABI, expecting manual memory management and the ability to specify explicit data structures. Sorensen's (2018) answer is to introduce a new language, XTLang, which is a Scheme/C hybrid. It is discussed further below – but it is essentially an explicitly-typed, manually memory-managed Scheme dialect, compiled on-the-fly into native code via Extempore's LLVM back-end.

It is XTLang – and the infrastructure used to compile it – which is Extempore's unique contribution toward the problem of developing a full-stack environment; using XTLang one can use system calls, call into native C libraries and develop performant, imperative algorithms in the style of C, when direct memory management and access is advantageous.

The boundary between XTLang and Scheme can pose problems in development, however, as experienced during this project. XTLang code, for instance, has no access to Scheme data or functions; there is no means to call a Scheme procedure from XTLang – the flow is strictly for Scheme to invoke pre-compiled (albeit just ahead of time) XTLang functions and not vice versa. The system does, however, succeed in enabling liveness of changes in systems-level XTLang code; functions can be re-compiled and the new versions will be executed on the next call – with the proviso that Extempore's *scheduler* is appropriately invoked by the function call.

2.5.1 Time in Extempore

Scheduling and time are important issues in the design of Extempore. Extempore's key use cases - live, dynamic audio-visual experiences and other interactive displays – impose demanding, near real-time time constraints. As reflected by Sorensen (2018), missing deadlines in live coded audio performances can result in not only embarrassment – a distorted signal or missed beat – but potentially aural damage, in the case of high decibel concerts.

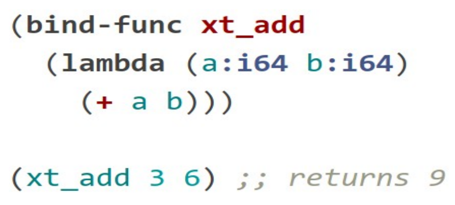
The Extempore *scheduler* is at the heart of time management. When programming with Extempore, events are scheduled to occur at a time in the very near future using a global callback mechanism which registers events with a scheduler – implemented as part of the native, C++ infrastructure of the interpreter. In order to allow executing code greater certainty over the time at which statements will be executed, Sorensen (2018) decided to opt for a co-operative multi-tasking model. The scheduler forms the basis for this. Each Scheme interpreter process is independent and does not share state – they communicate, if at all, by message passing; each has its own scheduler queue. Code executing has free reign of the Extempore thread in which it is running, blocking execution of any other registered callbacks until completion. When the current procedure ends, control is implicitly yielded to the scheduler, which executes the next task in the queue at the appropriate time (measured in audio samples – Extempore uses the audio device as a high-resolution clock).

Both XTLang and Scheme code can utilise the scheduler to register future callbacks. Much utility is made in Extempore – particularly for audio-visual applications – of so-called *temporal recursion*, a name for the process by which a procedure recursively schedules its own future callback (Sorensen, 2013); in such a way can regular, rhythmic processes be programmed. Passing symbolic names for functions to the Extempore scheduler (rather than function addresses, for instance) is what allows the run-time modification ("live coding") of Extempore programs; function re-definitions will therefore take force when the scheduler next evaluates a given function name symbol.

2.5.2 XTLang

Sorensen's (2018) purpose for the creation of Extempore was to achieve the goal of a "full-stack" live programming environment – an environment, as discussed above, whereby one may dynamically define and re-define procedures ranging the full gamut from low-level systems programming to high-level scripting and co-ordination. XTLang is

Sorensen's (2018) attempt to bridge the gap between interpreted languages like Scheme and compiled, manually memory-manged languages like C – it is essentially a C/Scheme hybrid, featuring syntax akin to S-expressions but with type annotations and unmanaged, direct memory access.



```
(bind-func xt_add
  (lambda (a:i64 b:i64)
    (+ a b)))

(xt_add 3 6) ;; returns 9
```

Figure 5: Adding two integers together in XTLang. Note the requirement for type annotations (a and b are 64-bit integers) but otherwise the similarity to Scheme. Source: Sorensen, 2016a.

A primary motivation for the development of XTLang was performance (Sorensen, 2018). XTLang code, unlike Scheme code in Extempore, is compiled ahead-of-time (*just* ahead-of-time in the case of code sent at run-time to the interpreter) using an LLVM-based – LLVM is an open-source project providing modular compiler technologies (Lattner, 2000) – back-end into native machine code. Extempore's Scheme interpreter, as admitted by Sorensen (2016b), is very slow – he cites an example where XTLang performs approximately 300 times faster for a numerical benchmark calculating the highest-common-factor of a set of integers – and therein lies much of the justification for introducing XTLang into Extempore. The division of labour is such that XTLang is to be used for systems programming – and high-performance computation – while Scheme is to be used as a control language for triggering events (Sorensen, 2016b).

XTLang's static typing, mirroring C, allows relatively easy – at least in theory; see further discussion below – interfacing with native C dynamic libraries. Libraries can be bound at run-time using the `bind-dylib` and `bind-lib` forms, which auto-generate Scheme wrappers for C functions, provided their typing information can be correctly specified in advance (Sorensen, 2016c). Extempore's standard library is in large part based around this use of XTLang's foreign function interface (hereafter FFI) – the shipped MIDI functionality, for instance, is built around XTLang bindings to the C libraries for *Portmidi* and *RtMidi*, two open-source MIDI libraries.

2.5.3 The pattern language

Since it is a key part of the high-level musical infrastructure of Extempore (Sorensen, 2020) and used as part of the demonstration of MIDI interaction herein, the pattern language in Extempore is briefly expounded here.

The pattern language is a domain-specific language, implemented as set of Scheme macros, used to abstract the process of rhythmically triggering a procedure given a set of timings in beat counts and a list of note values.

Patterns are defined and started with the `:>` macro. A breakdown of the form is shown below:

```
;; A demonstration pattern

(:> infinity-saxophone 20 0 (mplay midi-out @1 120 dur 3)
   `(c5 d5 eb5 | | | | _ c5 d5 | | | | _ d5 | | | | _ bb4 c5 | | | | _))
```

Figure 6: An example of a pattern playing a sequence of musical notes in Extempore.

The `:>` macro is invoked with the name of the pattern, here `infinity-saxophone`, the amount of beats over which to play it (here `20`) and the amount of beats to offset the pattern, here `0`. It then takes an expression; this expression is evaluated rhythmically on each metronome beat – Extempore has a global metronome object responsible for keeping track of time, accessible and adjustable via the `*metro*` closure – with each successive element of the final list(s) substituted for `@1`, `@2`, …, `@n`. In this example, only one list is supplied – a list of note values and special symbols – and hence only `@1` will be defined when the supplied expression `(mplay midi-out @1 120 dur 3)` is evaluated.

Two special symbols are permitted within lists of notes, `|` and `_`, corresponding to musical ties, representing an extension for another beat of the previous note, and pauses, representing a beat of silence, respectively. The symbol `dur` is bound prior to evaluating the user-supplied play expression to the calculated duration of the present note. Multiple notes per beat can be played by recursively nesting lists of notes – such as duplets, e.g. `(… (60 62) …)` or triplets, e.g. `(… (60 62 60) …)`.

The standard library macro `mplay` is worth mentioning briefly here since it is used extensively in the demonstration code of this project. It takes as arguments a reference to the output MIDI device, the note value to play, the volume (a 7-bit integer, as per the MIDI standard, ranging from 0 to 127), the duration in audio samples to play the note and, finally, the MIDI channel upon which to send the message.

3. Problem statement

Extempore lacks well-rounded MIDI support. Basic wrappers for two C libraries – *rtmidi* and *portmidi* – are included which provide a rudimentary, but limited interface; there is no support for routing via JACK, for opening virtual (software) ports, or for manipulating more than one input or output MIDI port at a time. All events are handled in a single, global event loop, calling predefined callback functions.

It would be of benefit to have an extensible, high-level MIDI interface abstracting over the underlying low-level back-ends, optionally with the ability to choose between back-ends but otherwise falling back on JACK as a cross-platform, low-latency routing solution.

Given this base layer of abstraction, it would be worthwhile to develop higher-level library functions to abstract commonly-used functionality such as live generation of arpeggiation, recording and looping abilities, as well as demonstrate using non-note controls (such as control change events and MIDI controller pad buttons) to interact with and influence live-coded algorithms and data in Extempore.

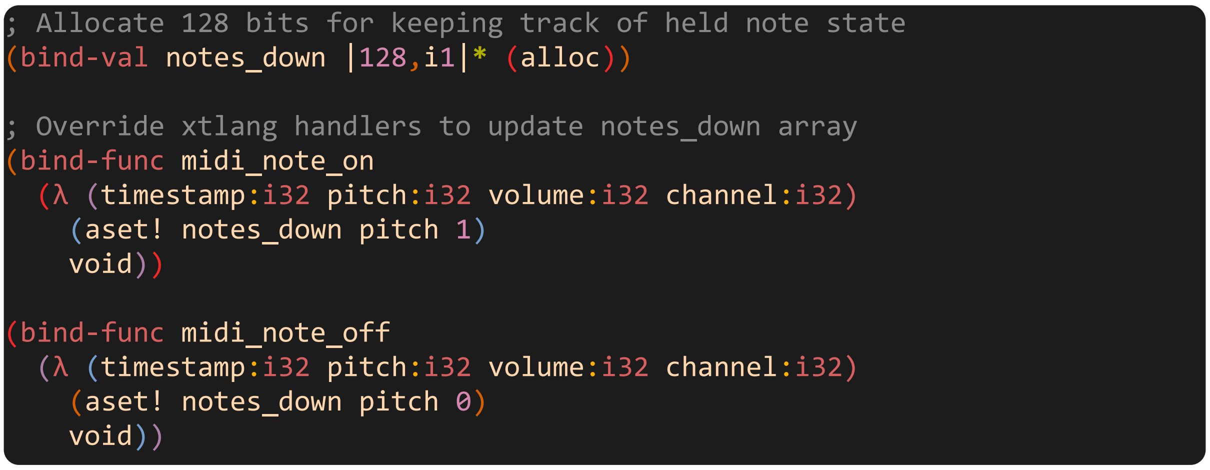
Once implemented, it would be desirable to evaluate Extempore as a platform for provision of such capability, including an evaluation of the success of the XTLang/Scheme split design in achieving Extempore's aim of being a "full-stack" live coding environment. The necessity of XTLang would be of interest, in particular, with regard to minimising MIDI latency and improving code performance; it is an open question as to whether adopting an alternative, more performant Scheme implementation may render the XTLang infrastructure – the defining feature of Extempore over its predecessor, Impromptu (Sorensen, 2016b) – unnecessary.

4. Approach

After developing a basic understanding of the underlying framework – Extempore itself, as well as its particular subset of Scheme and XTLang – the author's first concern was to find a way to gather notes entered from a hardware MIDI device and represent them as a list within Scheme for further manipulation. Given such a list, demonstrating live, run-time modifiable arpeggiation becomes trivial owing to Extempore's built-in pattern language – it can simply be substituted as the note list parameter of the `:>` macro.

This early work is represented by the code listing of *midi-held-notes.xtm*, which demonstrates using the built-in portmidi wrapper (written by Sorensen) to gather the list of notes currently held down on a specified MIDI input port.

This implementation draws attention to some of the deficits in the Extempore-provided MIDI library. To begin with, it is very heavy on global state. Interaction with the MIDI scheduler provided as part of the Extempore portmidi library requires overriding two global XTLang functions, `midi_note_on` and `midi_note_off`, which here are used to update an XTLang array. XTLang declarations are global across Scheme interpreter "processes" (which are actually implemented as threads) and are therefore not thread-safe. There is also no way to listen to more than one MIDI input simultaneously.

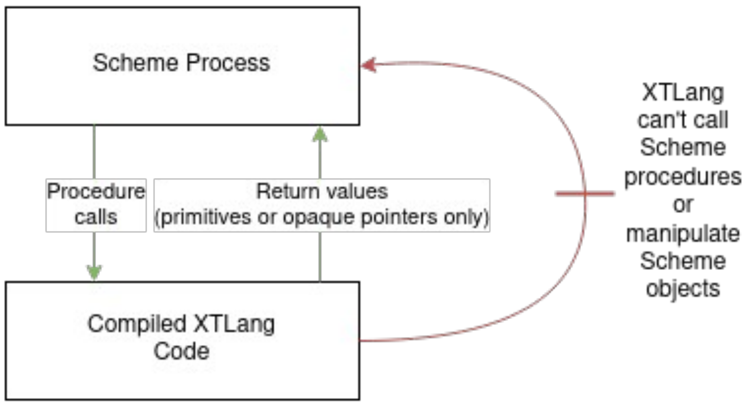

```
; Allocate 128 bits for keeping track of held note state
(bind-val notes_down |128,i1|* (alloc))

; Override xtlang handlers to update notes_down array
(bind-func midi_note_on
  (λ (timestamp:i32 pitch:i32 volume:i32 channel:i32)
    (aset! notes_down pitch 1)
    void))

(bind-func midi_note_off
  (λ (timestamp:i32 pitch:i32 volume:i32 channel:i32)
    (aset! notes_down pitch 0)
    void))
```
Figure 7: Updating a global, 128-bit array to store information on the currently-held MIDI notes using the built-in callbacks.

Note how XTLang is more verbose than Scheme – type annotations are required and each of these functions must explicitly return `void` in order to allow the compiler to compute their return type ahead of time. It is also required to manually `alloc`ate the memory for the `notes_down` array synonymously to allocating dynamic memory on the heap in C with `malloc()`.

This suffices for dynamically populating the `notes_down` array with boolean data. Passing this information back across the XTLang → Scheme divide is not intuitive, however. There is no way to access Scheme objects from XTLang – nor any way to generate Scheme objects in XTLang code, nor trigger evaluation of Scheme expressions in XTLang; XTLang must be triggered by the top-level Scheme interpreter in a unidirectional way (Sorensen, 2016b).


Scheme Process
Procedure calls
Return values (primitives or opaque pointers only)
Compiled XTLang Code
XTLang can't call Scheme procedures or manipulate Scheme objects
Figure 8: Scheme and XTLang inter-operability runs in one direction only, from the Scheme interpreter to XTLang code.

Converting the `notes_down` array into a Scheme list therefore involves repeatedly calling a Scheme function which triggers an XTLang closure which walks through the array, returning each non-zero element sequentially.

Other than being inelegant, this is not a thread-safe solution and involves iteration through the entire 128-element array using Extempore's slow Scheme interpreter for every call to `midi-held-notes` – which is likely to be multiple times per beat in the case of use for arpeggiation – as seen in the code listing below.

```
; Provide a means to read the notes_down array from Scheme
; midi_get_notes_down can be called in a loop to retrieve each held note
in turn
; returns -1 when list complete - N.B: not thread safe
(bind-func midi_get_next_note_down
  (let ((index:i8 -1))
    (λ ()
      (set! index (+ 1 index))
      (if (< index 0:i8)
        (set! index -1:i8)
        (if (= (aref notes_down index) 0:i1)
          (midi_get_next_note_down)
          index)))))

; Scheme wrapper function allowing Scheme code to get a list of held notes
; Not thread safe since midi_get_next_note_down is not thread safe
(define (midi-held-notes)
  (let ((note (midi_get_next_note_down)))
    (if (= note -1)
      '() ; finished
      (cons note (midi-held-notes))))) ; otherwise continue building list
```

Figure 9: Working around the limitation that only primitives can be returned by XTLang by iterating through the array using a Scheme wrapper.

Ideally, it would be possible to manipulate the incoming notes directly in Scheme – doing such polling per-event rather than multiple times per beat, which is likely to be significantly less frequent. For managing such events in a way appropriate for polling several devices simultaneously, without involving global state, streams – delayed lists, in the style of Abelson et al. (1996) – are suggested as a well-fitting high-level abstraction, allowing applications to monitor specific streams for events, asynchronously, as well as easily utilise standard higher-order functions (`map`, `filter`, `reduce`, etc.) to pre-process such streams for their unique needs. This would also allow for many different use cases to derive from one device event stream – rather than forcing the user-programmer to manually composite functions into a global XTLang callback.

5. Implementation

5.1 Low-level MIDI interfacing

This project leveraged partial implementation of RtMidi support, written by Ben Swift, the second major contributor to Extempore after Andrew Sorensen. Swift's incomplete RtMidi bindings were taken as a starting point for iteratively building upon the MIDI interface of Extempore with more flexible, high-level abstractions in mind.

RtMidi is a cross-platform library designed to abstract common functionality over its supported platforms – Windows via the Windows Multimedia Library, MacOSX using CoreMIDI or JACK and Linux using ALSA or JACK (Scavone, 2019). For this project, the author utilised RtMidi's JACK support – a performant, cross-platform solution for MIDI routing with support for virtual (software) ports – although an ALSA fallback is available if JACK support isn't detected in the system's shared RtMidi library.

5.2 Virtual ports

Opening virtual ports was implemented trivially by using XTLang's C FFI to call native entry points in *librtmidi*, as shown below:

```
; Helpers to open virtual ports
(bind-func make-rtmidi-in-port
  (λ (portname api)
    (let ((in (rtmidi_in_create api portname 128)))
      (rtmidi_open_virtual_port in "MIDI Input")
      in)))

(bind-func make-rtmidi-out-port
  (λ (portname api)
    (let ((out (rtmidi_out_create api portname)))
      (rtmidi_open_virtual_port out "MIDI Output")
      out)))
```

Figure 10: Using librtmidi's C API via XTLang's FFI. ***`rtmidi_in_create`*** *and* ***`rtmidi_out_create`*** *are trivially bound to the external shared C library using XTLang's FFI wrappers defined in libs/contrib/rtmidi.xtm.*

Virtual ports have the advantage that users do not need to know system MIDI ports in advance of starting an application – not the case with the built-in MIDI library – with routing able to be dynamically modified at run-time using utility software for JACK, as demonstrated above.

This behaviour is abstracted in a more idiomatic Scheme wrapper allowing input and output virtual ports, more suitable as the end-user interface (bearing in mind that the interface of the live-coding system is code itself):

```
(define (midi:create-port direction portname)
  ((if (eq? direction 'in)
     make-rtmidi-in-port
     make-rtmidi-out-port) portname (if rtmidi-jack-support?
                                         RTMIDI_API_UNIX_JACK
                                         RTMIDI_API_LINUX_ALSA)))
```

Figure 11: Idiomatic scheme wrapper for creation of virtual MIDI ports. Falls back to ALSA if JACK support is unavailable. Definition of `rtmidi-jack-support?` omitted for brevity – see tom/rtmidi-stream.xtm.

Since this procedure returns opaque pointers to RtMidi devices, sending messages over an output port is trivially implemented with aliasing:

```
(define midi-send rtmidi_send)
(define midi:send midi-send)
```

Figure 12: Two Scheme aliases for the dynamically bound, external `rtmidi_send`. The former is for compatibility with existing standard library functionality which uses the hyphenated naming convention.

Before discussing polling for events and processing input, here follows a discussion of the streams paradigm chosen to represent flows of incoming data.

5.3 Asynchronous streams

The streams paradigm expounded by Abelson et al. (1996) provides a way of producing lazily-evaluated lists of indefinite length, implemented in Scheme by replacing the default behaviour of `cons` with a procedure that delays its second argument – by wrapping it in an anonymous procedure or *thunk*, only to be evaluated when execution of other code depends upon its value.

This approach was taken as the inspiration for the handling of incoming MIDI messages. An approach was required which could handle registration of multiple callbacks of user-written code for each actively-polled MIDI input device yet provide a level of abstraction similar to that available when manipulating basic lists in Scheme – such as the ability to write idiomatic Scheme by utilising such higher-order functions as `map`, `filter` and `for-each` to pre-process data in a modular way – and ideally be able to be written in a simple, synchronous style yet without blocking the entire Scheme interpreter by actively waiting for events.

Building lists in the style of Abelson et al. (1996), however, would have posed a memory burden given the large amount of MIDI events generated by physical devices (turning a rotary control is sufficient to generate hundreds of control change events) and was not a natural fit for dealing with real-time input; when dealing with incoming MIDI events, one rarely cares in practice about historical events occurring in the stream (although building a list manually, as in the recording functionality described later, is trivial given the stream abstraction).

5.3.1 Closures

The streams abstraction was implemented instead as a closure holding state consisting of a list of callbacks, triggered on each new event, taking advantage of Extempore's built-in scheduler queue to schedule user-provided procedures for execution, as per the general non-preemptive, single-threaded design.

```
(define (stream:new)
  (let ((callbacks '()))
    (λ (command . args)
      (cond ((eq? command 'send)
             (for-each (λ (callback-fn)
                         (apply callback (now)
                                (if (symbol? callback-fn)
                                  (eval callback-fn)
                                  callback-fn) args))
                       callbacks))
            ((eq? command 'register-callback!)
             (set! callbacks (cons (car args) callbacks)))
            ((eq? command 'remove-callback!)
             (set! callbacks (filter (λ (callback-fn)
                                       (not (eq? callback-fn (car args))))
                                     callbacks)))
            ((eq? command 'purge-callbacks!)
             (set! callbacks '()))
            ((eq? command 'get-callbacks)
             callbacks)
            (else
              (println 'stream:new: 'unknown 'command: command))))))
```

Figure 13: The closure returned by `stream:new` *maintains a list of callback procedures as local state.*

Use of closures – `stream:new` is a function which returns an anonymous function, encapsulated with some additional state – in this manner is a Scheme idiom which enables data encapsulation analogously to the object-oriented paradigm (Abelson et al,, 1996). Callbacks may be registered by adding them to the beginning of the list (using the beginning, with `cons`, takes constant time; `append`ing to the end of the list would be of O(*n*) complexity since list traversal would be required) or removed, by filtering them out. When data is sent, the Extempore scheduler is invoked using the built-in procedure `callback` and fed each registered callback function with the incoming data providing the parameters for execution as soon as possible – `(now)`, which returns the current time in audio samples.

Further wrappers are provided to abstract the closure's internal commands (`register-callback!`, `remove-callback!`, etc.) with procedures for ease of use and debugging. Undefined procedure names will generate an error – but specifying the wrong internal command to the closure lambda will not and may easily go unnoticed.

```scheme
(define (stream:send stream . data)
  (apply stream 'send data))
(define (stream:register-callback! stream fn)
  (stream 'register-callback! fn))
(define (stream:remove-callback! stream fn)
  (stream 'remove-callback! fn))
(define (stream:purge-callbacks! stream)
  (stream 'purge-callbacks!))
```

Figure 14: Wrapper procedures to hide implementation details - a common pattern when passing around closures in Scheme.

5.3.2 Higher-order functions

Some higher-order functions are provided for manipulating streams of data. By using these functions, internal management of callbacks is hidden in such a way as to allow declarative programming with streams – as if writing synchronous code – without the cognitive burden (see Leger and Fukuda, 2016) of managing many layers of nested, potentially inter-dependent callbacks.

The definitions of `stream:for-each`, `stream:map`, `stream:filter` and `stream:merge` are shown in the code listing below:

```
(define (stream:for-each fn stream)
  (stream:register-callback! stream fn))

(define (stream:map fn stream)
  (let ((output-stream (stream:new)))
    (stream:register-callback! stream (λ data
                                         (stream:send output-stream
                                                      (apply fn data))))
    output-stream))

(define (stream:filter predicate stream)
  (let ((output-stream (stream:new)))
    (stream:for-each (λ data
                       (if (apply predicate data)
                         (apply stream:send output-stream data)))
                     stream)
    output-stream))

(define (stream:merge . streams)
  (let ((output-stream (stream:new)))
    (for-each (λ (stream)
                (stream:for-each (λ data
                                   (apply stream:send output-stream data)))
              streams)
    output-stream))
```

Figure 15: Implementation of some standard higher-order functions for asynchronous (callback-based) streams.

For-each is trivially implemented as registering the procedure to be called on each incoming datum as a stream callback. Mapping, filtering and merging streams are accomplished by creating a new stream and forwarding events – transformed as specified in the case of `stream:map` – as required to new, output streams, the references to which being returned by each procedure.

5.3.3 Asynchronous waiting

One final example is provided of how this approach to creating asynchronous streams meshes well with Extempore's scheduler queue and the utility of first-class continuations in Scheme for achieving custom control structures. Several languages have an `async/await` syntax, such as ES6 Javascript (see e.g. Kantor, 2020), whereby asynchronous code may be written in synchronous style using special syntactic forms. In Extempore, this behaviour can be accomplished in a minimal amount of pure Scheme, as seen in the implementation of `stream:await` – a function which skips execution of the remainder of its parent procedure, waits for the next stream event and *then* re-starts execution, seemingly returning the value of the event thus received – all without blocking execution of any other scheduled code.

```
(define (stream:await stream)
  (call/cc (λ (current-continuation)
    (letrec ((callback-fn (λ args
                            (stream:remove-callback! stream callback-fn)
                            (apply current-continuation args))))
      (stream:register-callback! stream callback-fn)
      (*sys:toplevel-continuation* 'awaiting)))))
```

Figure 16: Using first-class continuations and callbacks in Scheme to implement asynchronous waiting behaviour.

The procedure `stream:await` simply registers a stream callback which, when invoked, de-registers itself and then passes on the stream data received to the parent continuation – the procedure waiting for a value from `stream:await`. Here, `*sys:toplevel-continuation*` refers to a continuation representing the top-level state in the interpreter's REPL; calling it effectively jettisons the current execution context with a non-local exit and is used to prevent the rest of the parent procedure from executing until a stream event is received.

For context, below is an example showing how this allows programming iteration in a straightforward, synchronous style. This snippet collects the next ten events from `input-stream` into the list `events` – without blocking the REPL or any other scheduled procedures.

```
(define events '())
(dotimes ((i 10))
  (set! events (append events (stream:await input-stream))))
```

Figure 17: A demonstration of `stream:await`.

5.4 Processing MIDI Input into streams

Given the above, providing a stream-based interface to MIDI input requires forming a stream closure, polling the MIDI device for events and forwarding these events through the stream with `stream:send`.

5.4.3 Polling for MIDI events

RtMidi provides two mechanisms for obtaining events – active polling or providing a callback function for incoming events. Unfortunately, registering a callback function would be of no advantage; Extempore is designed such that XTLang cannot call Scheme code, and the callback function would of necessity have to be implemented in XTLang (due to its C ABI interoperability), hence all ensuing user code would be limited to being written in XTLang. Given, as noted repeatedly, that the user interface *is* code, it would be desirable to keep this at the highest achievable level of abstraction – which means providing our user-accessible functionality as a Scheme rather than XTLang library.

A Scheme polling loop is therefore required, actively polling for events and forwarding them as required to our stream closures – and thereafter downstream user code.

The design of this was slightly convoluted due to limitations of Extempore's threading and inter-process communication functionality. It was decided to implement active polling in a separate thread in order to allow high-frequency polling of multiple devices concurrently without choking the Scheme scheduler in the main interpreter thread. Extempore refuses to serialise opaque pointers for inter-"process" (Scheme interpreter process, that is – which are implemented as OS threads) communication (hereafter IPC), meaning that a workaround involving passing global data through XTLang constructs has to be used instead. XTLang has direct memory access to the entire process memory space, making it suitable for this purpose, provided mutual exclusion is used appropriately (Sorensen, 2016c).

```
;; XTLang values / procedures used for passing data across the IPC divide
;; (since Extempore won&apos;t serialise pointers)

(bind-val midi_current_device_mutex i8* (mutex_create))
(bind-val midi_current_device_ptr i8* 0)
(bind-func set_midi_current_device_ptr (λ (value)
                                         (set! midi_current_device_ptr value)))
(bind-func get_midi_current_device_ptr (λ () midi_current_device_ptr))
```

Figure 18: Using XTLang to pass data across the divide between Scheme processes - a mutex, a global opaque pointer and accessor functions.

Extempore also will not serialise closures across the IPC divide. This means that we cannot directly pass stream objects to the foreign (polling) thread for direct use on incoming events; an array is therefore maintained of stream closures, with their respective indices into the array serving as an IPC-serialisable means of identification. Scheme features a vector data type – contiguous elements in memory, akin to C arrays – but it is of static size; an array-list abstraction inspired by Java's dynamic ArrayList (see Oracle, 2020) was developed for use in this scenario to support indefinitely large amounts of devices and associated streams. The implementation of the array-list structure is omitted here for brevity – see the code listing of *tom/arraylist.xtm* for details.

The poll loop itself is depicted in the following listing. It uses the XTLang bindings to `rtmidi_get_msg` to get the length of the next pending message, which is thereafter stored in a global XTLang binding, accessible in Scheme through calling the wrapper function `rtmidi_get_message_global`. No mutual exclusion locking is required for this section despite the use of global XTLang state since this function is only called from within the poll loop thread, which processes each device sequentially.

```
(define (rtmidi-scheme-poll-loop device callback-fn)
  (if (null? deregistration-notices)
    (let ((len (rtmidi_get_msg device))
          (msg (rtmidi_get_message_global)))
      (if (= len 0)
        (callback (+ (now) 48) rtmidi-scheme-poll-loop device callback-fn)  ;; Check
for events every 1ms (48000 Hz / 1000)
        (let ((type (msg_type msg))
              (chan (msg_chan msg))
              (a (msg_a msg))
              (b (msg_b msg)))
          (callback (now) callback-fn type chan a b)
          (rtmidi-scheme-poll-loop device callback-fn))))
    (if (not (member device deregistration-notices))
      (callback (now) rtmidi-scheme-poll-loop device callback-fn))))
      ;; Let other devices be processed until deregistration is handled
```

Figure 19: Scheme polling loop for MIDI events - this runs continuously in the polling thread.

The only other non-intuitive part of this code listing is the need to handle de-registration notices, a means by which polling for a given device can be stopped (such as if we wish to pause polling – or if the device is deleted and its memory `free`d); this is done via keeping track of messages (passed in from the primary thread) in a `deregistration-notices` list and checking this before proceeding with each iteration.

The helper functions `msg_type`, `msg_chan`, `msg_a` and `msg_b` read the appropriate parts of a pointer to a MIDI message using bit-wise logic, given the message structure given above:

```
(bind-func msg_type:[i8,i8*]*
  (λ (msg:i8*) (>> (pref msg 0) 4)))
(bind-func msg_chan:[i8,i8*]*
  (λ (msg:i8*) (& (pref msg 0) 15)))
(bind-func msg_a:[i8,i8*]*
  (λ (msg:i8*) (pref msg 1)))
(bind-func msg_b:[i8,i8*]*
  (λ (msg:i8*) (pref msg 2)))
```

Figure 20: XTLang bitwise logic to extract information from MIDI messages.

We can now create a new Scheme process responsible for executing the polling loop. The following listing demonstrates the set-up of the polling process in Extempore, defining foreign functions via IPC and ensuring Scheme wrappers for the required XTLang functions are generated in the foreign process:

```
; Start midi event polling thread on port 1337
(ipc:new "midi-poller" 1337)

;; Define Scheme callback function in polling thread
(ipc:define "midi-poller" 'rtmidi-scheme-poll-loop rtmidi-scheme-poll-loop)

;; Define xtlang-Scheme wrappers in polling process
(for-each (λ (name) (ipc:bind-func "midi-poller" name))
          (list 'get_midi_current_device_ptr 'rtmidi_get_message_global
                'rtmidi_get_msg 'msg_type 'msg_chan 'msg_a 'msg_b))

;; Define message-handling functions for polling thread
;; NB. Device pointer passed through xtlang as cannot serialise cptrs
(ipc:define "midi-poller" 'register-device
            (λ (stream-id)
              (rtmidi-scheme-poll-loop (get_midi_current_device_ptr)
                                       (λ (type chan a b)
                                         ;; Phone home with event details
                                         (ipc:call-async "primary" 'midi:handle-
event stream-id type chan a b)))))
```

Figure 21: Setting up the polling loop functionality in a separate Scheme process and handling new device registrations in the polling thread – each new registration triggers starting a new polling loop, with the stream id state encapsulated in the callback closure.

As seen above, the polling thread uses IPC to inform the primary (REPL) thread of the stream ID and details of incoming events as they are found. The primary thread forwards these events on after looking up the stream matching the index provided:

```
;; Define handler for incoming events from polling thread
(define (midi:handle-event stream-id type chan a b)
  (stream:send (array-list:get midi:stream-refs stream-id) type chan a b))
```

Figure 22: Handling incoming event-received messages in the primary thread.

The final major piece of the polling implementation is `midi:input-port→stream`, the public interface, which returns a stream given an input RtMidi device pointer:

```
(define midi:stream-refs (array-list:new 16))

(define (midi:input-port->stream input-port)
  (let*
    ((output-stream (stream:new))
     (stream-ref (array-list:add! midi:stream-refs output-stream)))
    ;; Guard XTLang IPC with a mutex for thread safety
    ($ (mutex_lock midi_current_device_mutex))
    (set_midi_current_device_ptr input-port)
    (ipc:call "midi-poller" ‘register-device stream-ref)
    ($ (mutex_unlock midi_current_device_mutex))
    output-stream))
```

Figure 23: The user interface to this functionality - obtaining an actively-polled stream from an input port reference.

5.5 Implemented behaviours

5.5.1 Arpeggiation

Arpeggiation can be implemented easily in Extempore using the built-in pattern language, which takes a procedure – used as the algorithm for playing the pattern – and a list of notes to play, as discussed above.

Converting a stream of MIDI events into a list of currently-held notes is implemented with the holder abstraction, implemented in *tom/recording.xtm*. Holder closures, created with `holder:new`, register a simple stream callback which maintains local state concerning which notes in a stream are currently depressed. Note that in the below, the returned list is built *backwards* so that constant-time `cons` operations can be used without having to reverse the list (at otherwise O(*n*) time and memory cost) upon completion.

```
(define (holder:new . input-stream)
  (let* ((held-notes (make-vector 128))
         (stream #f)
         (callback-fn (λ (type channel note velocity)
                        (cond ((= type *midi-note-on*)
                               (vector-set! held-notes note velocity))
                              ((= type *midi-note-off*)
                               (vector-set! held-notes note ‘())))))
         (note-vector->list (λ (notes-vector)
                              (let loop ((i 127)
                                         (notes-list ‘()))
                                (if (< i 0)
                                  notes-list
                                  (loop (- i 1)
                                        (if (null? (vector-ref notes-vector i))
                                          notes-list
                                          (cons i notes-list))))))))
    ;; Rest of code (accessors, wrappers etc.) omitted for brevity
```

Figure 24: Holder closures provide a means of obtaining a list of currently-held notes given a MIDI event stream. The implementation uses Scheme vectors for constant-time access and reference – converting this data to a list is performed by `holder:get-notes`, which calls the internal procedure `note-vector→list`.

An example of this behaviour in use is demonstrated below. Using this framework, a simple arpeggiator can be created in just a few lines of Scheme:

```
;; Create a virtual MIDI input port and get access to a stream of events
(define keyboard-stream (midi:input-port->stream
                          (midi:create-port ‘in "Keyboard Input")))
(define midi-out (midi:create-port ‘out "Extempore MIDI Output"))

;; Create a “holder” to keep state regarding keys currently depressed
(define keyboard-holder (holder:new keyboard-stream))

;; Simplest possible arpeggiated pattern - plays notes in low-to-high order
(:> simple-arppegiator 0 0 (mplay midi-out @1 80 dur 0)
    (holder:get-notes keyboard-holder))
```
Figure 25: Minimal arpeggiation example using pattern language, streams and holders.

Since the pattern language can utilise any valid Extempore expression – and the list returned by `(holder:get-notes keyboard-holder)` is a standard, Scheme list of notes – the user has complete flexibility over the arpeggiation algorithm. Notes may be mapped to other notes, scaled, or otherwise modified procedurally in a run-time modifiable way. The following is an example of a slightly more complex expression in use – this plays an arpeggio over two octaves, running up and down the notes currently held on the MIDI keyboard stream:

```
(:> octave-up-down-arp 4 0 (mplay midi-out @1 80 (* 2 dur) 0)
    (let* ((held-notes (take (holder:get-notes keyboard-holder) 4))
           (held-notes^12 (map (λ (x) (+ x 12)) held-notes)))
      `(,@held-notes
        ,@held-notes^12
        ,@(reverse held-notes^12)
        ,@(reverse held-notes))))
```
Figure 26: Demonstrating a more elaborate pre-processing of the retrieved notes list; arpeggiating up and down over two octaves.

Here, the utility function `take` defined in *tom/utils.xtm* is used in order to obtain a set amount of elements from the list, repeating the list where necessary. This is to circumvent the pattern language's default scaling up or down of note durations as per the length of the note list, which otherwise causes the rhythm to undesirably slow or hasten depending on the amount of keys depressed.

5.5.2 Interacting with MIDI controls and non-note buttons

Two helper functions, `midi:on-cc` and `midi:on-keypress` are implemented to aid in using midi control change events – such as are generated by physical rotary dials – and midi pads, which generally function in the same way as keyboard keys but are frequently used as triggers rather than for musical input (Rothstein, 1995).

The definition of `midi:on-keypress` is shown below; `midi:on-cc` is very similar in implementation.

```
(define-macro (midi:on-keypress input-stream channel note . function-expr)
  (let ((fn_name (string->symbol (string-append "key_callback_" (number->string
channel) "_" (number->string note)))))
    `(begin
       (stream:remove-callback! ,input-stream (quote ,fn_name))
       (define ,fn_name (λ (type chan a b)
                           (if (and (= type *midi-note-on*)
                                    (= chan ,channel)
                                    (= a ,note))
                             (begin ,@function-expr))))
       (stream:register-callback! ,input-stream (quote ,fn_name)))))
```

Figure 27: A macro to enable quick binding of control code to button presses - suitable for using with MIDI controller pads.

This is an example of code generation using Extempore's provision of the Scheme syntax `define-macro`. A function is generated and defined in the top level environment, with boilerplate conditional checks – ensuring the procedure is only invoked on a specified key and channel – automatically included before the user-specified code. This allows for succinct bindings of Scheme expressions to key presses – triggering events, for example – as is seen demonstrated in the recording and looping functionality discussed below.

The following code listing is an example of combining midi control change callbacks with the minimal pattern-language arpeggiator demonstrated above. Manipulating the rotary dial on the controller has the effect of shortening or prolonging each note duration for a simple *trancegate*-like effect – each note duration is scaled by `@1 + 1` (the implicit new value of the controller plus one – ranging therefore from 1 to 128 – Extempore will not accept zero duration values) divided by 64, hence from 1/64 to two. The symbol `@1` is defined within the `midi:on-cc` macro as a succinct means of accessing the new value of the controller within the specified trigger expression:

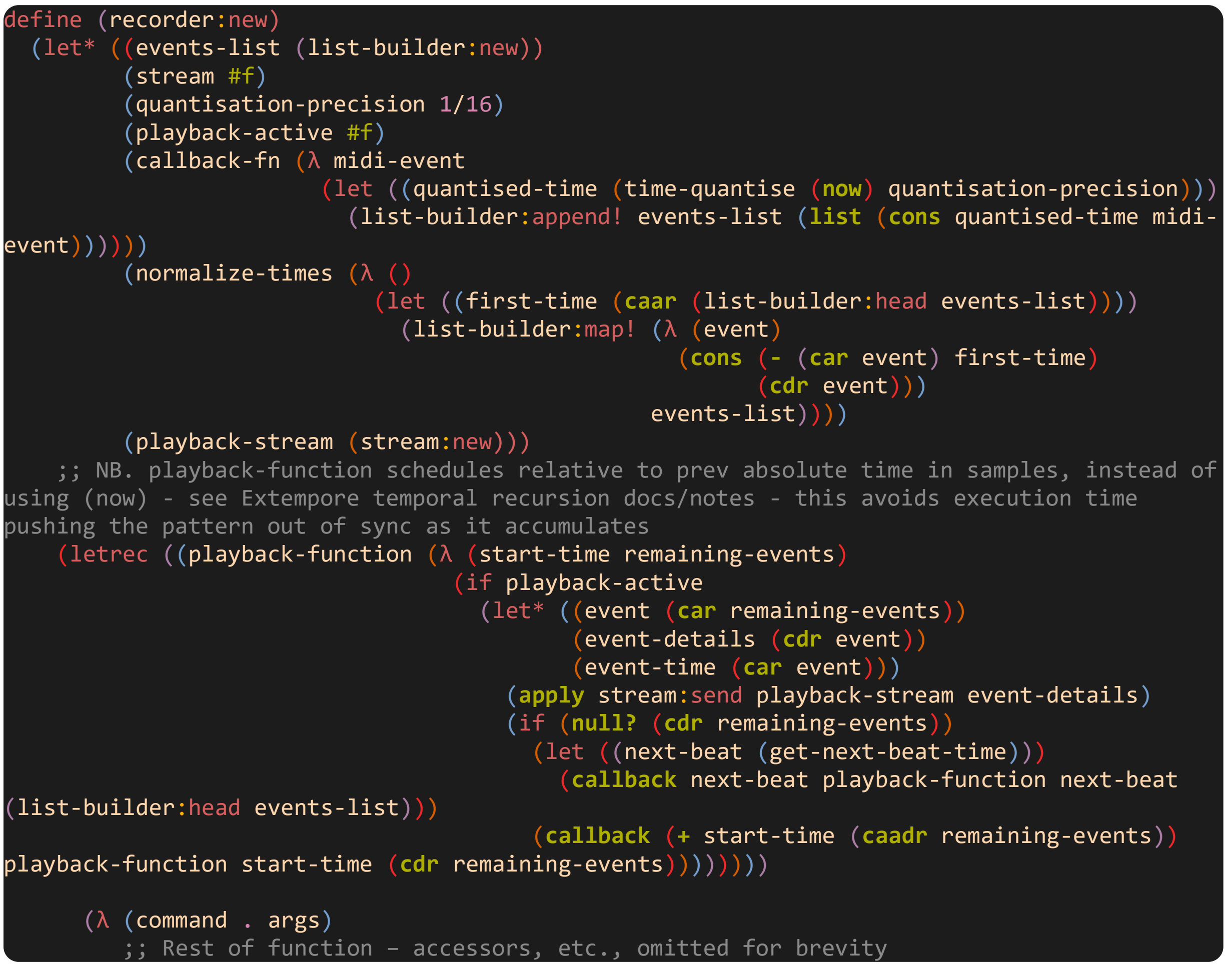
```
(define duration-scale-factor 1.0)
(midi:on-cc keyboard-stream 0 0 (set! duration-scale-factor (/ @1 64)))

(:> duration-arp 2 0 (mplay midi-out @1 80 (* dur duration-scale factor) 0)
    (holder:get-notes keyboard-holder))
```

Figure 28: Using midi control change events to dynamically influence arpeggiation - in this case, note duration.

5.5.3 Recording

Recording and sequencing MIDI input is ubiquitous across digital audio software – being one of the foundations of digital audio workstation software such as Cubase (Steinberg 2020) and Ableton Live (Ableton, 2020). Approaching this problem in Extempore began with implementing a means to record and re-play MIDI streams, encapsulated in the recorder closure, created via `recorder:new`, defined in *tom/recording.xtm.* Since recording is implemented here by building lists of stream events, a utility data structure, `list-builder`, was first implemented which allows constant-time appending to Scheme lists by keeping track of the final element in the linked list and mutating it upon each `list-builder:append!` operation. The implementation details are omitted here for brevity but may be seen in *tom/utils.xtm*. The code listing below shows the majority of the logic of the recording functionality:



```
define (recorder:new)
  (let* ((events-list (list-builder:new))
         (stream #f)
         (quantisation-precision 1/16)
         (playback-active #f)
         (callback-fn (λ midi-event
                        (let ((quantised-time (time-quantise (now) quantisation-precision)))
                          (list-builder:append! events-list (list (cons quantised-time midi-
event))))))
         (normalize-times (λ ()
                            (let ((first-time (caar (list-builder:head events-list))))
                              (list-builder:map! (λ (event)
                                                   (cons (- (car event) first-time)
                                                         (cdr event)))
                                                 events-list))))
         (playback-stream (stream:new)))
    ;; NB. playback-function schedules relative to prev absolute time in samples, instead of
using (now) - see Extempore temporal recursion docs/notes - this avoids execution time
pushing the pattern out of sync as it accumulates
    (letrec ((playback-function (λ (start-time remaining-events)
                                  (if playback-active
                                    (let* ((event (car remaining-events))
                                           (event-details (cdr event))
                                           (event-time (car event)))
                                      (apply stream:send playback-stream event-details)
                                      (if (null? (cdr remaining-events))
                                        (let ((next-beat (get-next-beat-time)))
                                          (callback next-beat playback-function next-beat
(list-builder:head events-list)))
                                        (callback (+ start-time (caadr remaining-events))
playback-function start-time (cdr remaining-events)))))))

      (λ (command . args)
         ;; Rest of function – accessors, etc., omitted for brevity
```

Figure 29: Recording functionality defined as part of the recorder:new closure.

In essence, the recorder closure maintains a local list of events and registers a callback function with a given input stream which saves each event and the time it occurred – although this time is *quantized* to the Extempore metronome's beat beforehand, discussed further below – to this local list. A playback stream is created; when playback is triggered using `recorder:play!`, the saved events are sent through this stream, repeated indefinitely and maintaining the appropriate timing intervals between events. The Scheme `letrec` form defines a binding able to recursively reference itself – the `playback-function` binding, for instance, needs to be able to reference itself to establish a temporal recursion.

Two commands accepted by the closure are detailed in the following code listing due to their importance in achieving the recorder's behaviour; they are triggered by `recorder:start-recording!` and `recorder:stop-recording!` respectively:

```
             ((eq? Command 'start-recording!)
               (set! events-list (list-builder:new))
               (stream:register-callback! stream callback-fn))
              ((eq? command 'stop-recording!)
               (stream:remove-callback! stream callback-fn)
               ;; Add "dummy events" to generate correct spacing between last + first
recorded events. First dummy event is synchronized with beat prior to start of pattern -
this ensures that first event's timing is preserved
               (let ((dummy-event '(0 0 0 0)))
                 (list-builder:append-head! events-list (cons (get-first-beat-before (caar
(list-builder:head events-list))
                                                               dummy-event))
                 (list-builder:append! events-list (list (cons (get-next-beat-time)
                                                               dummy-event))))
               ;; Now normalize the recorded times so they start at zero
               (normalize-times))
```

Figure 30: Code responsible for starting and stopping the recording process – keeping the pattern in synchrony with the metronome.

When a recording is played, the recording is always scheduled to occur on the Extempore metronome's beat – this decision was taken to ensure that pattern language constructs and recordings stay synchronised over time. In order to preserve event timings, a dummy event – which does nothing, consisting of all zero data fields – is inserted at the beginning of the recording, timed with the previous beat. Similarly, a null event is inserted a the end of the stream on the beat following the last event – this ensures that the next repetition of playback will occur on-beat, ensuring the timing does not drift away from that of the metronome.

5.5.3.1 Time quantisation

The functions utilised above which quantise event times relative to the metronome are defined in *tom/quantise.xtm*. Time quantisation was implemented in order to keep recordings in synchrony with the Extempore beat – as well as being a first step towards implementation of sequencing behaviour.

The key function is `time-quantise`, which takes a time (in audio samples) and a precision, generally specified as a fractional value. As seen above, the default for the recorder closure is 1/16 – this is modifiable using the `recorder:set-precision!` accessor function. The denominator of the precision determines how many times by which to subdivide each beat. A smaller denominator means that events will be increasingly beat-aligned but suffer a greater loss of timing precision. With a precision of 1, for instance, all events will occur on-beat; with a precision of 1/3, triplets are possible.

```
;; time-quantise: sample-time precision -> adjusted-sample-time
;; Given a time in samples and a precision,
;; return a new adjusted sample-time quantised to occur at a
;; multiple in beats of precision
;; e.g. assuming sample rate of 48kHz
;; (time-quantise 49233 1/4) -> 54000.00
;; The new value is certain to take place on a quarter beat per *metro*

(define (time-quantise sample-time precision)
  (let* ((beat-at-sample-time (*metro* 'beat-at-time sample-time))
         (sample-adjustment (*metro* (- (round-to-closest beat-at-sample-time
precision) beat-at-sample-time))))
    (+ sample-time sample-adjustment)))
```

Figure 31: Quantising sample times relative to the Extempore metronome.

It is worth noting that the Extempore `*metro*` closure, when invoked with a single argument – representing the beat number – returns the time in audio samples that this beat is scheduled to occur upon. The inverse operation, obtaining the beat at a given time in samples, is obtained by passing in the `‘beat-at-time` command. The ability to customise the quantisation precision is due to the utility function `round-to-closest`, which rounds the first argument to the nearest multiple of the second; its definition is omitted here for brevity.

5.5.4 Looping

Looping – playing the same musical sequence in an indefinite, repetitive loop – is often used for performance purposes utilising such devices as guitar loop pedals, where music can be built up in a series of sequentially-recorded loops overlaid upon each other for aesthetic effect (Rudrich, 2017). The looping functionality implemented here is relatively trivially – given the existent time-quantisation – implemented on top of the recording functionality discussed above.

The following code listing shows the full implementation of the looper closure:

```
 (define (looper:new input-stream output-stream output-channel)
  (let ((recorder (recorder:new))
        (holder (holder:new))
        (status ’empty)
        (passthrough (λ (type chan a b)
                       (stream:send output-stream type output-channel a b))))
    ;; Set up recorder & holder closures
    (recorder:set-input-stream! recorder input-stream)
    (stream:register-callback! (recorder:get-playback-stream recorder) passthrough)
    ;; Return dispatcher lambda
    (λ (command . args)
      (cond ((eq? command ’trigger)
             ; On trigger, the looper is started if stopped or stopped if started...
             (cond ((eq? status ’empty)
                    (set! status ’recording)
                    (holder:set-input-stream! holder input-stream)
                    (stream:register-callback! input-stream passthrough)
                    (recorder:start-recording! recorder))
                   ((eq? status ’recording)
                    ;; Recording finished
                    (if (null? (recorder:get-events recorder))
                      (set! status ’empty)
                      (begin
                        (set! status ’playing)
                        (stream:remove-callback! input-stream passthrough)
                        (holder:set-input-stream! holder (recorder:get-playback-stream
recorder))
                        (recorder:stop-recording! recorder)
                        (recorder:play! recorder))))
                   ((eq? status ’playing)
                    ;; Stop playback
                    (set! status ’stopped)
                    (recorder:stop! recorder)
                    ;; Clear notes from holder - useful when used with arpeggiators/pattern
lang. etc.
                    (holder:purge-notes! holder))
                   ((eq? status ’stopped)
                    (set! status ’playing)
                    (recorder:play! recorder))))
            ((eq? command ’get-recorder) recorder)
            ((eq? command ’get-stream) output-stream)
            ((eq? command ’get-held-notes) (holder:get-notes holder))
            ((eq? command ’status) status)
            (else (println ’looper:create/lambda: ’unknown ’command command))))))
```

Figure 32: The looper closure handles automatic recording and looped playback.

The looper construct is implemented as a simple finite state machine dependent upon the internal variable binding `status`; it handles setting up a recorder and has one main interface – the command `‘trigger` – which toggles its state as appropriate from recording to playback to halting playback.

It is worth noting that the looper closure contains within it a holder closure, allowing code to query which notes are currently depressed in the recorded loop at a given instant in time. This was implemented in order to facilitate use of loopers for control of arpeggiation – for instance, using a looped recording to seed the notes for a pattern language arpeggiator; the procedure `looper:get-held-notes` returns a list of currently depressed notes.

5.5.4.1 Using a MIDI controller as a looper interface

The design of the looper closure was such that it would be appropriate to use with a MIDI controller's pad buttons, as seen on the author's Akai APC Key 25, featuring forty MIDI pad buttons, as pictured below.

A looper object could be created and have its trigger command bound to one of the MIDI pad buttons, for instance, allowing simple recording and looping functionality to be available without looking away from the MIDI controller; the state of the looper closure could further be communicated to the user using the controller's programmable LED lights which lie beneath each of the translucent pad buttons.

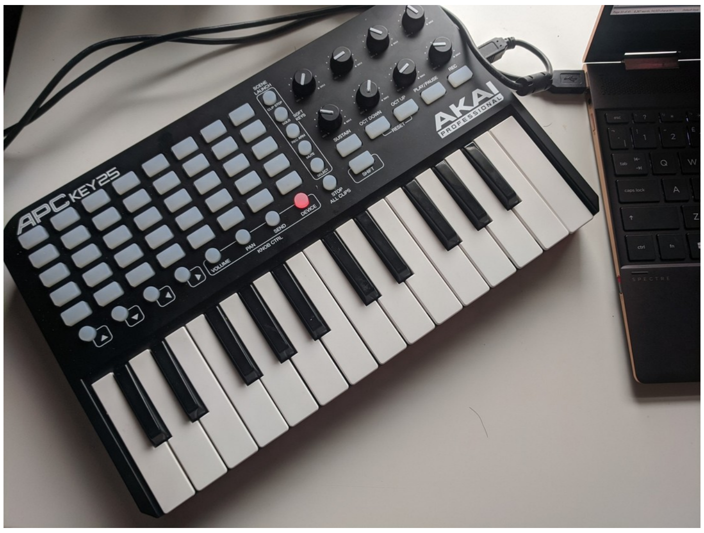


Figure 33: The author's Akai APC Key 25 MIDI controller, featuring minature piano keys, rotary dials and pad buttons.

This involved first defining some useful, keyboard-specific constants and procedures. Utilising the LEDs on the keyboard is accomplished by sending MIDI note on messages with specific note and velocity values – see the code listing of *tom/apckey25.xtm* for full details – which were obtained by observing which note values the keyboard itself sent using ALSA's `aseqdump` utility when each pad was pressed.

```
(define *apc-output* (midi:create-port ’out "Extempore APC25 Control"))

; Helper functions for use with lights
(define (apc25:set-light! button-id colour)
  (midi:send *apc-output* *midi-note-on* 0 button-id colour))
```

Figure 34: A function to set the programmable lights on the APC Key 25. Loading tom/apckey25.xtm executes the line creating the virtual output port which must be linked to the keyboard’s input port via JACK.

Registering key presses is trivial given the existing stream architecture – a simple wrapper is included to encapsulate that the keyboard’s non-note outputs always occur on channel zero:

```
(define-macro (apc25:on-keypress midi-stream note expr)
  `(midi:on-keypress ,midi-stream 0 ,note ,expr))
```

Given this, setting up loops attached to the keyboard’s pad buttons can be accomplished with minimal user code by invoking the macro apc25:setup-looper, which, given a note reference, input and output streams and output channel, handles the process of creating the looper closure, binding the looper:trigger command to the appropriate key press event and adjusting the pad’s LED lighting based on the new status of the looper closure:

```
;; Helper for quickly setting up loops attached to APC25 pads
(define-macro (apc25:setup-looper key input-stream output-stream output-channel)
  (let ((looper-name (string->symbol (string-append "looper_key_" (number->string key)))))
  `(begin
     (define ,looper-name (looper:new (stream:filter (λ (type chan a b)
                                                          (not (and (= chan 0)
                                                                    (not (= type *midi-
cc*)))))
                                                      ,input-stream)
                                       ,output-stream ,output-channel))
     (apc25:on-keypress ,input-stream ,key
                        (begin
                          (looper:trigger ,looper-name)
                          (let ((status (looper:status ,looper-name)))
                            (cond ((eq? status ’stopped)
                                   (apc25:set-light! ,key *apc-amber-solid*))
                                  ((eq? status ’recording)
                                   (apc25:set-light! ,key *apc-red-solid*))
                                  ((eq? status ’playing)
                                   (apc25:set-light! ,key *apc-green-solid*))))))
     ;; Start light as amber, flashing, signalling looper in empty state
     (apc25:set-light! ,key *apc-amber-blink*)
     ,looper-name)))
```

Figure 35: A macro for quickly setting up loopers attached to MIDI controller pads.

5.5.5 Sequencing to pattern language

Given the primacy of code in the live-coding paradigm embraced by Extempore, it would be desirable to provide a means of returning *code* itself as an end-product of a MIDI recording in a form amenable to direct, user-programmer modification at runtime. Sequencing a recording of MIDI stream events to an Extempore pattern language S-expression – the usual, *de facto* way of playing rhythmic note sequences in modern Extempore (Sorensen, 2020) – was the author's attempt to achieve this.

The process is accomplished in stages. Note that the recorder closure described above stores events in the form of `(time . data)` pairs, where time refers to the time in audio samples at which a given event occurs and data is a list of the form `(type chan a b)`, corresponding to each part of the MIDI message recorded. The first step in the sequencing process is to transform this list into a list containing data in the format of `(start-time note duration)`, with one such entry for each note played, where `duration` and `start-time` are measured in audio samples:

```
(define (event-list->note-duration-list event-list)
  (define note-type-pairs (map (λ (event)
                                 (let ((event-time (car event))
                                       (event-type (cadr event))
                                       (event-note (cadddr event)))
                                   (if (or (= event-type *midi-note-on*)
                                           (= event-type *midi-note-off*))
                                     (cons event-note (cons event-type event-time))
                                     '()))) event-list))
  (set! note-type-pairs (filter (λ (pair)
                                  (not (null? pair))) note-type-pairs)) ;; Remove control
change events etc.
  (let loop ((remaining-events note-type-pairs)
             (return-list '()))
    (if (null? remaining-events)
      (reverse return-list)
      (let* ((event (car remaining-events))
             (note (car event))
             (event-type (cadr event))
             (event-time (cddr event)))
          (let ((release-event (assoc note (cdr remaining-events))))
            (loop (cdr remaining-events)
                  (if (= event-type *midi-note-on*) ;; If it's a note-on event, append
details...
                    (cons (cons event-time (cons note (- (cddr (if release-event
                                                                 release-event
                                                                 (tail remaining-events)))
event-time)))
                          return-list)
                    return-list)))))))  ;; Otherwise just loop & ignore
```

Figure 36: Converting the recorder's representation of events into a sequence of notes and their durations (in sample-time).

The list of note and sample-time duration pairs thus obtained is then converted into a list containing data in the format of (start-time note duration) – but with the start-time and duration specified in terms of *beats*:

```scheme
;; Helper function to convert output of above function into sub-beat times and durations
;; (from absolute time values)

(define (time-note-duration->beat-note-pair event-list quantisation-factor)
  (define initial-beat (*metro* 'beat-at-time (caar event-list)))
  (map (λ (event)
         (list (real->integer (* quantisation-factor (- (*metro* 'beat-at-time (car event))
initial-beat)))
               (cadr event)
               (real->integer (* quantisation-factor (*metro* 'beat-at-time (cddr
event))))))
       event-list))
```

Figure 37: Converting absolute time values (in samples) into beat duration values.

Possessing such a list of beats and their durations is then sufficient to generate the pattern list. The idea encapsulated in the following code listing, which performs the whole conversion from the events-list format (obtained from the recorder closure) into a pattern language note sequence, is to sequentially step through the note-beat-duration pair list (generated using the functions above) and generate a list consisting of each note followed by sufficient ties (|) to produce the correct duration – or the maximum possible duration before the next note begins (the pattern language conversion implemented here is monophonic).

Gaps between notes are identified by the lack of overlap in adjacent beat-durations and start-times and an appropriate amount of _ symbols are inserted to preserve timing; the end of the sequence is further padded with silence to ensure that the total amount of beats is integer-divisible by the quantisation precision factor, required since the pattern language expression demands an integer amount of beats to be specified for the total-pattern playback time.

```
;; Put it all together - produce a pattern-list expression given a recorder-generated list
of MIDI events
(define (event-list->pattern-list event-list quantisation-factor)
  (define beat-note-pairs (time-note-duration->beat-note-pair

                            (event-list->note-duration-list event-list) quantisation-
factor))
  (define pattern-list '())
  ;; Need to look for "gaps" and insert silences appropriately (_s)
  (let loop ((remaining-events beat-note-pairs))
    (let* ((event (car remaining-events))
           (note (cadr event))
           (note-start-beat (car event))
           (note-duration (caddr event))
           (event-finish-beat (+ note-start-beat note-duration)))
      (if (null? (cdr remaining-events))
        (set! pattern-list (append pattern-list     ;; Reached end of list -> will return
pattern-list
                                   (cons note (nof (- note-duration 1) '|))))
        (let* ((next-event (cadr remaining-events))
               (next-event-start-beat (car next-event))
               (beats-til-next-note (- next-event-start-beat note-start-beat))
               (spacing-length (- next-event-start-beat event-finish-beat)))

          (set! pattern-list (append pattern-list
                                     (cons note (nof (- (min note-duration beats-til-next-
note) 1) '|))))
          (if (> spacing-length 0)
            ;; Need to insert silence(s)
            (set! pattern-list (append pattern-list (nof spacing-length '_))))
          (loop (cdr remaining-events))))))

  ;; Pad the produced pattern list so that it ends on a complete beat

  ;; i.e. total sub-beat count should be divisible by quantisation-factor without remainder

  (let ((padding-required (- (closest-multiple-above (length pattern-list) quantisation-
factor)
                             (length pattern-list))))
    (set! pattern-list (append pattern-list (nof padding-required '_)))))
```
Figure 38: Building the full pattern language note expression given a list of pairs in the format `(time . (type chan a b))`*, such as are obtained from a recorder closure.*

Tying up the loose ends and providing a cleaner programmer-user interface, we conclude this functionality with `recorder→pattern-expr`, which takes a recorder closure and generates a full pattern expression:

```
(define (recorder->pattern-expr rec)
  (define quantisation-factor (/ 1 (recorder:get-precision rec)))

  (define pattern-list (event-list->pattern-list (recorder:get-events rec) quantisation-
factor))
  `(:> _pattern_name_ ,(real->integer (/ (length pattern-list) quantisation-factor)) 0
       (mplay midi-out @1 80 dur 0)
       (quote ,pattern-list)))
```
Figure 39: The public interface for this functionality - given a recorder, it returns a fully-formed pattern language expression.

To demonstrate the output produced, this procedure was applied to a recorder closure containing note data, generating the following output – a valid Extempore pattern language expression, able to be evaluated as-is:

```
(:> _pattern_name_ 8 0 (mplay midi-out @1 80 dur 0) (quote (60 | | | | _ _ _ 60 | |
| | _ _ _ _ 57 | | | | _ _ 64 | | | | _ _ _ _ 62 | | | | | _ _ _ _ _ _ _ _ _ 59 | |
| | | _ _ _ _ _ _ _ _ _ _ 60 | | | | _ _ _ 60 | | | | _ _ _ 57 | | | _ _ _ _ 64 | |
| | _ _ _ 62 | | | | _ _ _ _ _ _ _ _ _ 60 | | | | | _ _ _ _ _ _ _ _ _ _ _ _)))
```

Figure 40: A demonstration pattern language construct returned by `recorder→pattern-expr`.

6. Analysis

The analysis herein first examines the correctness of the behaviour of the code produced during this project – through using unit testing, including some discussion of how such unit tests were implemented. This is followed by discussion of a key non-functional requirement given the live coding, live performance domain – latency, and therefore implicitly, execution performance.

Extempore as a development target and environment is evaluated – and alternative approaches to providing a Scheme-based, live coding environment with MIDI control are considered, with particular focus on evaluating whether Extempore's split-language design is strictly necessary or rather a consequence of Extempore's slow Scheme interpreter.

6.1 Testing

6.1.1 Correctness of data structures

6.1.1.1 A Scheme testing language

There are limited means to test code shipped with Extempore. Unit testing for XTLang code is accomplished through the `xtmtest` macro and a Cmake script (Sorensen, 2016d) – but this is not suitable for testing Scheme procedures. A testing DSL was therefore implemented using a Scheme macro for use in quickly defining unit tests for the utility functions and data structures upon which this project depends.

The testing language features two operators, → and `←,` being used for testing and assignment, respectively. Temporary variables can be bound using ← and expressions tested against expected values using →. An example test definition is shown below:

```
(tests:define-tests 'list-group
                    lst <- '(1 2 3 4 5 6)
                    (list-group lst 1) -> `((1) (2) (3) (4) (5) (6))
                    (list-group lst 2) -> `((1 2) (3 4) (5 6))
                    (list-group lst 4) -> `((1 2 3 4) (5 6)))
```

Drawing 1: Unit testing the `list-group` function from utils.xtm.

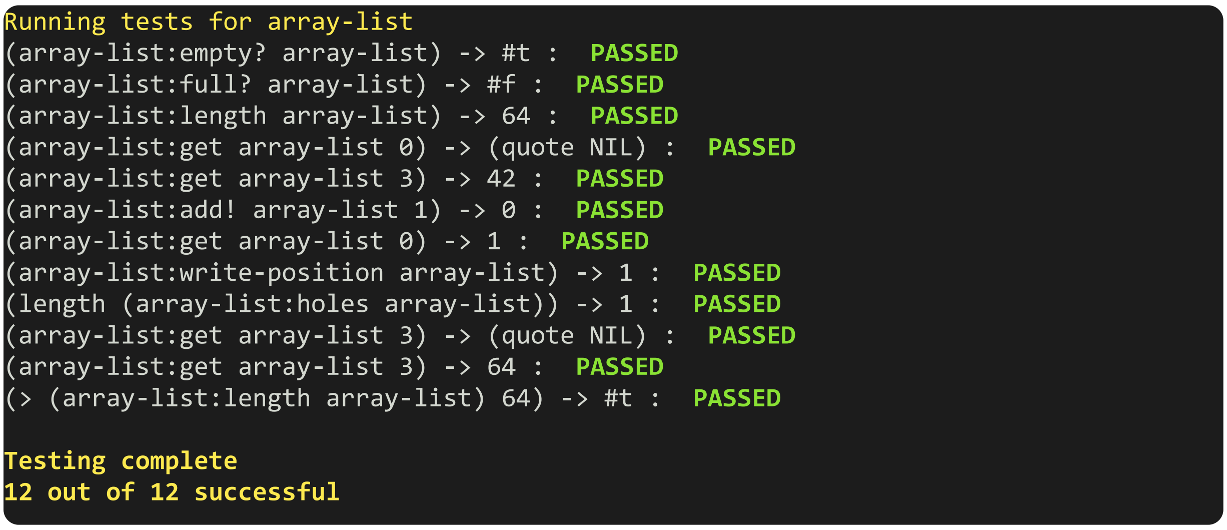
In this test for list-group, a procedure to group lists into groups of up to *n* elements, the temporary variable binding `lst` is first created and assigned the value of a list of integers from one to six. Various expression tests then follow, checking that the list is suitably partitioned into appropriately sized sub-lists per the parameter n. Each of these will be checked separately and any errors highlighted in the Extempore terminal output. Interleaving variable bindings with test expressions allows testing in an imperative-style code block, allowing unit testing of closures holding internal state, as in the case of the array-list functionality.

Variable assignment to a dummy variable _ is used in the below excerpt in order to allow execution of code statements which influence internal state, which in turn are followed by test expressions to validate the state change:

```
(test:define-tests
  ’array-list
  array-list <- (array-list:new 64)
  (array-list:empty? array-list) -> #t
  (array-list:full? array-list) -> #f
  (array-list:length array-list) -> 64
  (array-list:get array-list 0) -> ‘()
  _ <- (array-list:set! array-list 3 42)
  (array-list:get array-list 3) -> 42
  (array-list:add! array-list 1) -> 0
  (array-list:get array-list 0) -> 1
  (array-list:write-position array-list) -> 1
  _ <- (array-list:remove! array-list 3)
  (length (array-list:holes array-list)) -> 1
  (array-list:get array-list 3) -> ‘()
  _ <- (array-list:add! array-list 64)
  (array-list:get array-list 3) -> 64
  _ <- (array-list:set-write-position! array-list (array-list:length array-list))
  ; Check array list expands appropriately
  _ <- (array-list:add! array-list 1337)
  (> (array-list:length array-list) 64) -> #t)
```

Drawing 2: Using the testing DSL to cause and validate internal state change within an array-list closure.

Tests are therefore defined ahead of time with `test:define-tests`, which takes a descriptive name for the test sequence; to execute a set of tests, `test:run-tests` is invoked with the names of the test sets to use. After running the above code listing, running `(test:run-tests ‘array-list)` results in the following terminal output:



```
Running tests for array-list
(array-list:empty? array-list) -> #t :  PASSED
(array-list:full? array-list) -> #f :  PASSED
(array-list:length array-list) -> 64 :  PASSED
(array-list:get array-list 0) -> (quote NIL) :  PASSED
(array-list:get array-list 3) -> 42 :  PASSED
(array-list:add! array-list 1) -> 0 :  PASSED
(array-list:get array-list 0) -> 1 :  PASSED
(array-list:write-position array-list) -> 1 :  PASSED
(length (array-list:holes array-list)) -> 1 :  PASSED
(array-list:get array-list 3) -> (quote NIL) :  PASSED
(array-list:get array-list 3) -> 64 :  PASSED
(> (array-list:length array-list) 64) -> #t :  PASSED

Testing complete
12 out of 12 successful
```

Drawing 3: Terminal output after successfully testing the array-list functionality.

As can be seen, each test expression is printed and reported as passed or failed, followed by a count of successful versus total tests executed – useful when executing several test sets at once for identifying when issues have arisen.

The testing DSL was implemented using a macro which, given a list of expressions, first groups them into groups of three elements – as per the X ⟶ Y or X ⟵ Y syntax. The type of the second element, ⟶ or ⟵ is checked and assignment (using `define`) or checking of test expressions (via `eval`) is performed as appropriate, with additional code handling counting the successful versus unsuccessful test count. The key functionality is handled by the interplay of the macro `test:define-tests` and a utility function, `define-tests-helper`, which handles the logic of processing each expression group.

The procedure `test:get-fn-from-name` generates what will become the global binding name for a thunk – an anonymous procedure of no arguments – representing pending sets of tests by prepending a suitably long string before the test set name in order to avoid global namespace conflicts; Extempore lacks Scheme `gensym` support.

```
(define (test:get-fn-from-name name)
  (string->symbol (string-append "test:macrogen:" (symbol->string name))))

(define-macro (test:define-tests . args)
  (let ((title (car args))
        (grouped-exprs (list-group (cdr args) 3)))
    `(define ,(test:get-fn-from-name (cadr title))
       (λ ()
         (let ((total-tests 0)
               (tests-passed 0))
           (print-with-colors ’yellow ’default #t (println ’Running ’tests
’for ,title))
           ,(cons ’begin (map define-tests-helper grouped-exprs))
           (println)
           (cons total-tests tests-passed))))))
```
Figure 41: The macro test:define-tests parses the test-specific syntax.

```
define (define-tests-helper group)
  ;; Group consists of (expr direction expr)
  (let ((first (car group))
        (direction (cadr group))
        (last (caddr group)))
    (cond ((eq? direction ’<-)
           `(define ,first ,last))
          ((eq? direction ’->)
           `(let ((passed? (equal? (eval ,first) (eval ,last))))
              (set! total-tests (+ 1 total-tests))
              (if passed? (set! tests-passed (+ 1 tests-passed)))
              (print (quote ,first) ’-> (quote ,last) ’: " ")
              (print-with-colors (if passed? ’green ’red) ’default #t
                                 (println (if passed? ’PASSED ’FAILED)))))
          (else ’unknown-direction))))
```
Figure 42: The helper procedure define-tests-helper processes the parsed syntax as appropriate - either creating variable bindings or evaluating test expressions.

This simple testing DSL is easily applied to each of the other utility functions and structures used within the project; details are omitted here for brevity but can be viewed within each appropriate code listing – see *arraylist.xtm*, *quantise.xtm*, and *utils-test.xtm* and *async-streams.xtm* for details.

6.1.2 Correctness of behaviour: asynchronous streams

Testing the correctness of the asynchronous stream behaviour – other important factors in usability, such as latency and performance, are examined below – is possible with the above testing framework; it is detailed briefly here owing to its implementation in the given testing syntax being non-intuitive at first owing to the time dependency of stream events and callbacks.

The solution to achieving concise but expressive test cases for the asynchronous stream framework came from combining Extempore's scheduler with the aforementioned `stream:await` procedure, which allows asynchronous waiting to be written in a synchronous, blocking style. First, some helper procedures are defined to serve as shorthand for scheduling sending data through a stream at a future time:

```
(define (stream:send-at time stream . data)
  (callback time (λ () (apply stream:send stream data))))

;; Wrapper for the above for using millisecond delays
(define (stream:send-after-ms ms stream . data)
  (apply stream:send-at (+ (now) (* (/ *second* 1000) ms)) stream data))
```

Figure 43: Utility procedures for scheduling sending events via streams in the future.

Given this, a test case can be written for the procedure `stream:await` by creating a stream, scheduling data to be sent through it in the near future and finally by checking that awaited events match the data sent, in the correct time order:

```
(test:define-tests
  'stream:await
  stream <- (stream:new)
  _ <- (stream:send-after-ms 0 stream 42)
  _ <- (stream:send-after-ms 100 stream 100)
  (stream:await stream) -> 42
  (stream:await stream) -> 100)
```

Figure 44: Unit test for `stream:await`.

```
Running tests for stream:await
(stream:await stream) -> 42 :  PASSED
(stream:await stream) -> 100 :  PASSED

Testing complete
2 out of 2 successful
```

Figure 45: Terminal output after running the tests for `stream:await`.

Each of the stream procedures is thereafter tested for correctness using a similar approach. For one final example, showing a more elaborate expression as part of the test definition, the following is the test case for `stream:filter`, which passes through only events meeting a predicate from the input to an output stream. The built-in macro `dotimes` is used to send 0, 1, 2, 3, and 4 in sequence through the input stream; only 0, 2 and 4 are expected to be sent through the filtered stream.

```
(test:define-tests
  'stream:filter
  stream <- (stream:new)
  evens <- (stream:filter even? stream)
  _ <- (dotimes (i 5) (stream:send-after-ms (* i 50) stream i))
  (stream:await evens) -> 0
  (stream:await evens) -> 2
  (stream:await evens) -> 4)
```
Figure 46: Unit test for `stream:filter`.

```
Running tests for stream:filter
(stream:await evens) -> 0 :  PASSED
(stream:await evens) -> 2 :  PASSED
(stream:await evens) -> 4 :  PASSED

Testing complete
3 out of 3 successful
```
Figure 47: Terminal output after running the tests for `stream:filter`.

See *async-stream.xtm* for the unit tests for each of the other stream procedures.

6.1.3 Correctness of behaviour: holders and recorders

The *holder* abstraction – which provides a means of viewing which keys are depressed at a given instant in a given MIDI stream – depends only on stream behaviour rather than time-based MIDI input and is readily testable using the testing DSL. A means of faking MIDI events in streams is first required:

```
(define (send-midi-notes stream type . notes)
  (for-each (λ (note)
              (stream:send stream type 0 note 127))
            notes))

(define (depress-notes stream . notes)
  (apply send-midi-notes stream *midi-note-on* notes))
(define (release-notes stream . notes)
  (apply send-midi-notes stream *midi-note-off* notes))
```
Figure 48: Utility functions for faking MIDI events on a given input stream.

The following snippet demonstrates the approach used for testing the internal state of the holder object; the remainder of the unit test is omitted for brevity:

```
(test:define-tests
  'holder
  input-stream <- (stream:new)
  holder <- (holder:new input-stream)
  (holder:get-notes holder) -> `()
  _ <- (depress-notes input-stream 60 62 65)
  (holder:get-notes holder) -> `(60 62 65)
  _ <- (depress-notes input-stream 65 62 60)
  (holder:get-notes holder) -> `(60 62 65)   ; Check for duplication errors
  _ <- (depress-notes input-stream 42)
  (holder:get-notes holder) -> `(42 60 62 65)
  _ <- (release-notes input-stream 60 62 65)
  (holder:get-notes holder) -> `(42)
  ; Rest of test omitted for brevity
```
Figure 49: Demonstrating unit testing the holder abstraction.

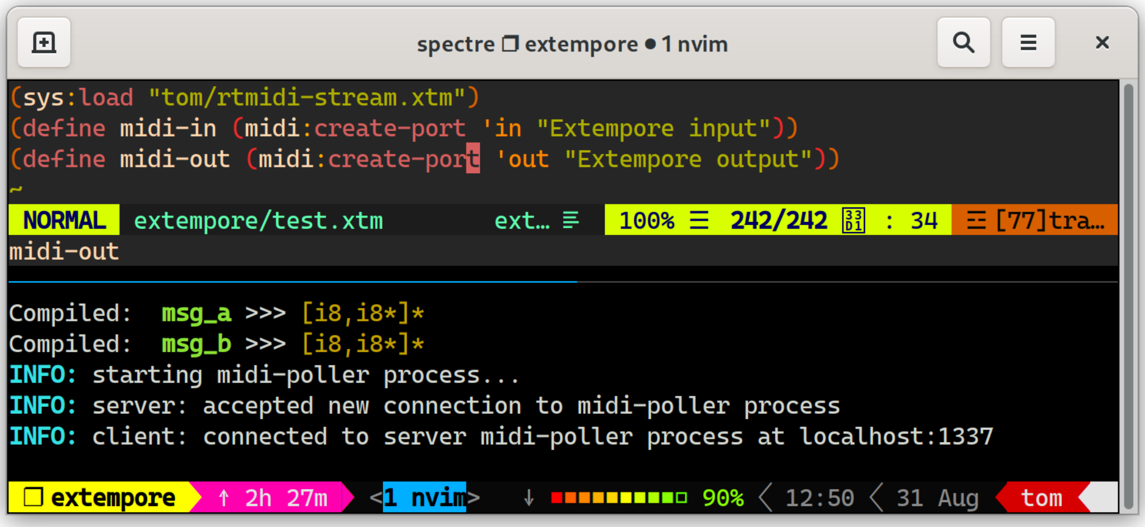
Similarly, recorders are dependent only upon stream callbacks and not on raw MIDI input or timing. The recorder unit test, present in *recording.xtm*, feeds a recorder closure dummy events and checks for the correct sequence of events to be sent through an output stream upon triggering playback.

6.1.4. Correctness of behaviour: MIDI input and output

The low-level MIDI functionality cannot be unit tested with the above DSL since the desired behaviour of its procedures – such as opening virtual ports, sending and receiving MIDI messages – consist of side effects, measurable only using other system tools. Since the infrastructure which allows this to happen – such as the JACK daemon – are out of our control, the only available approach is that of black box testing – where inputs are passed to the system and we check to see if the desired outputs, or side-effects, take place (Jorgensen, 2014).

6.1.4.1 Virtual ports

Opening virtual ports is accomplished using `(midi:open-port direction portname)` where `direction` may be `'in` for an input or `'out` for an output port; `portname` specifies a human-readable name which will be displayed by the connection graph manager. Black-box testing this functionality involves observing whether the appropriate input and output ports are created in the JACK graph post-execution.



```
(sys:load "tom/rtmidi-stream.xtm")
(define midi-in (midi:create-port 'in "Extempore input"))
(define midi-out (midi:create-port 'out "Extempore output"))
~
 NORMAL  extempore/test.xtm        ext… ≣  100% ☰  242/242 ㏑ : 34  ☲ [77]tra…
midi-out

Compiled:   msg_a >>> [i8,i8*]*
Compiled:   msg_b >>> [i8,i8*]*
INFO: starting midi-poller process...
INFO: server: accepted new connection to midi-poller process
INFO: client: connected to server midi-poller process at localhost:1337

❐ extempore  ↑ 2h 27m  <1 nvim>  ↓ ■■■■■■■■■□ 90% 〈 12:50 〈 31 Aug  tom
```

Figure 50: Loading the MIDI library and creating an input and output port.

Checking the JACK graph after sending these lines to the interpreter shows that the ports have successfully been registered with the JACK daemon:

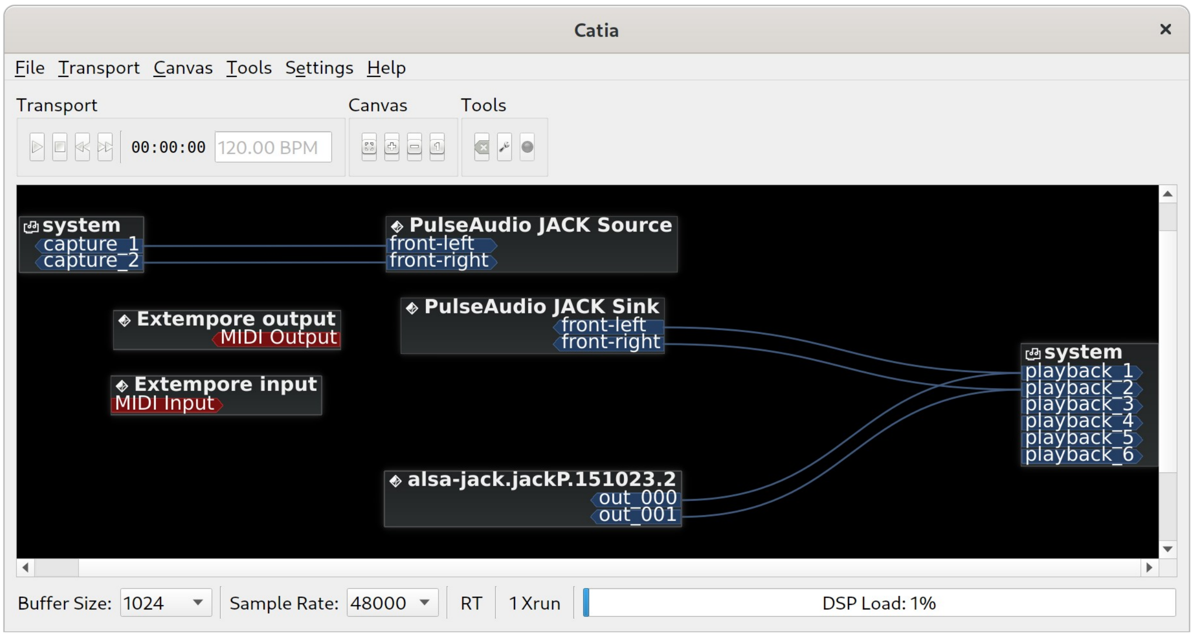
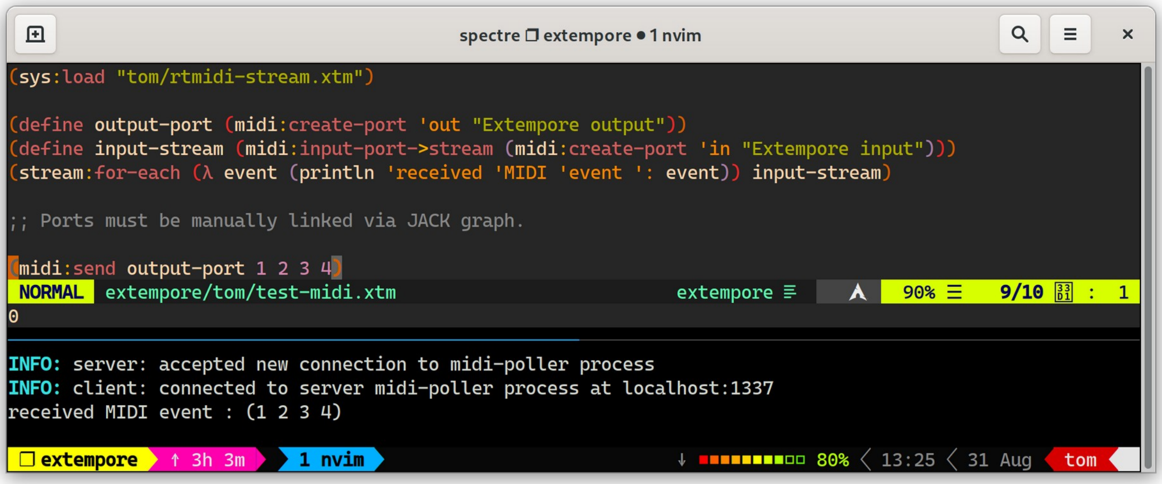


Figure 51: JACK connection graph via showing successful port creation.

6.1.4.2 Sending and receiving MIDI messages

Sending and receiving MIDI messages is dependent upon the underlying system software, such as JACK. Black box testing of this functionality (see screenshot below) involves setting up a minimal example – in this case, creating two ports and sending a message between them. This requires user intervention as *librtmidi* provides no means of linking connections together programatically; *Catia* or another tool needs to be used to link the input and output port prior to executing the `midi:send` expression.



```
(sys:load "tom/rtmidi-stream.xtm")

(define output-port (midi:create-port 'out "Extempore output"))
(define input-stream (midi:input-port->stream (midi:create-port 'in "Extempore input")))
(stream:for-each (λ event (println 'received 'MIDI 'event ': event)) input-stream)

;; Ports must be manually linked via JACK graph.

(midi:send output-port 1 2 3 4)
 NORMAL  extempore/tom/test-midi.xtm                        extempore ≡     ⋏   90% ☰   9/10 ㏑ :  1
0

INFO: server: accepted new connection to midi-poller process
INFO: client: connected to server midi-poller process at localhost:1337
received MIDI event : (1 2 3 4)

 ❐ extempore  ↑ 3h 3m   1 nvim                              ↓ ■■■■■■■■□□ 80% 〈 13:25 〈 31 Aug   tom
```

Figure 52: Black box testing MIDI transit across virtual ports. The console output demonstrates that the message was successfully sent and received.

6.2 Latency and performance

As reviewed by McPherson et al. (2016), latency – the delay between input and output, such as between a keyboard key being depressed and the corresponding audio signal being produced by a software synthesizer – is of critical importance for the playability aesthetic of a digital instrument. High latency results in the loss of transparency of the digital instrument's internal processing, drawing attention away from the act of playing itself and causing difficulty with rhythmic timing – the latter being especially an issue where there is significant variation in latency, a phenomenon termed *jitter* (Travis and Lesso, 2004).

Estimates for acceptable upper limits on latency vary within the literature from between 6ms to 30ms, depending on instrumentation (McPherson et al., 2016), with percussive instruments – presumably, where rhythmic timing is more prominent – being on the lower end of the spectrum and those involving gesticulation – such as playing the Theremin – being on the upper end. Dahl and Bresin (2001), in an experimental study manipulating drum signals by introducing artificial latency in the range of 1 – 127ms, noted that players would unconsciously start playing their beats *earlier* than the beat in order to compensate for the delay – but that this behaviour only persisted until a break-point of around 55ms, after which only one of out of their four, musically-trained subjects was able to stay synchronised with the metronome. The lowest levels of jitter perceptible in the literature for a 100ms rhythmic sequence are reported by Brandt and Dannenberg (1998) to lie somewhere between 1ms to 5ms; Friberg and Sundberg (1995) report an experimental lower bound, just noticeable difference threshold in timing variation of 6ms for all monotonic, rhythmic sequences tested.

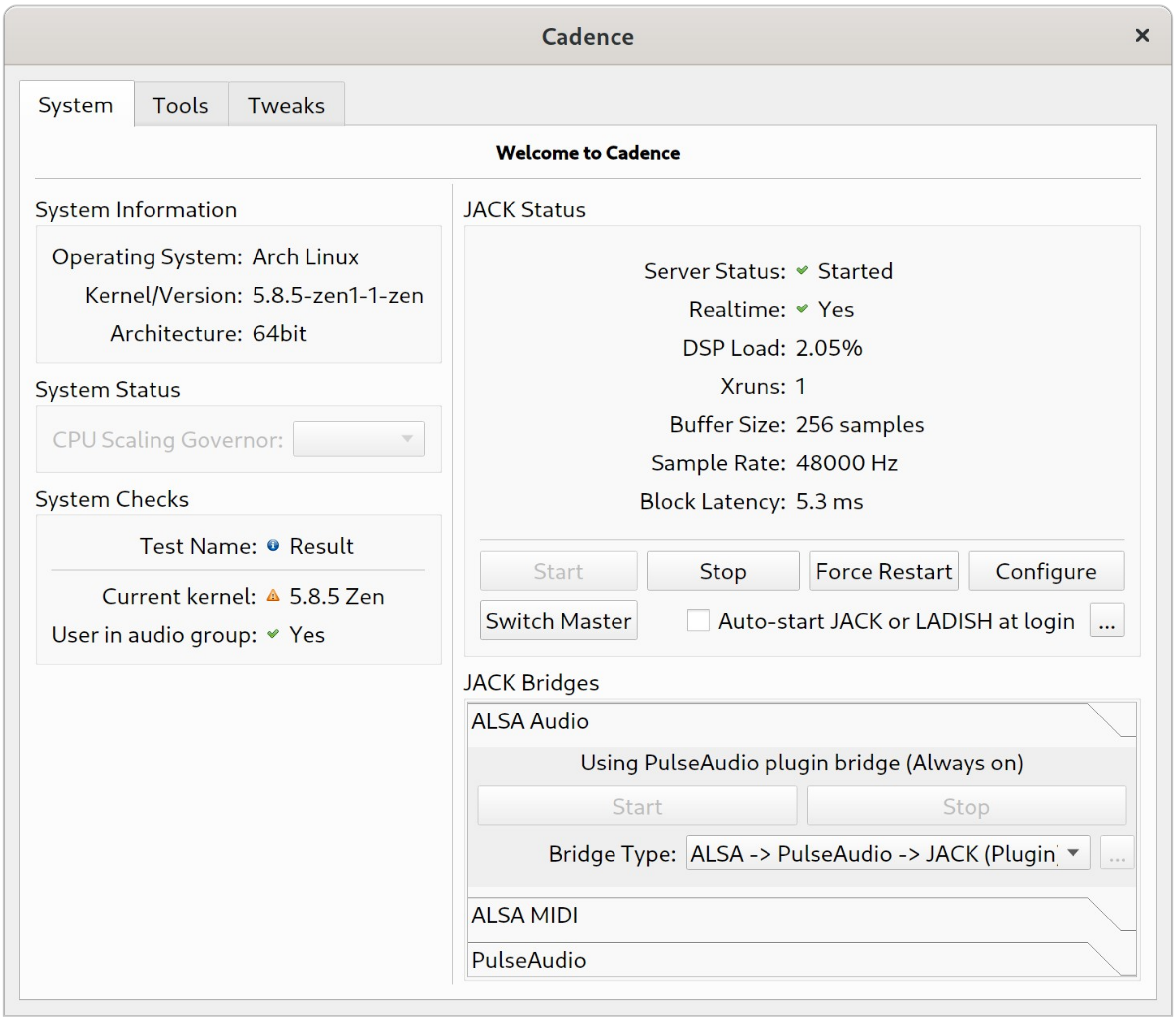
Wang et al. (2010) found that typical latency achievable using digital audio workstation (DAW) software across a selection of consumer audio equipment had high variation, from 1.68ms through to an upper limit of 399ms, dependent upon operating system and audio API, with ALSA on Linux and CoreAudio on MacOSX generally achieving audio latency in the range of 2 – 30ms on their hardware.

For this project, we are concerned with MIDI latency rather than that of generation of audio signals; the minimal case of MIDI passthrough is studied here. Further consideration is then given to identifying parts of the system contributing most to the latency, with a particular focus on whether the asynchronous streams framework causes issues given Extempore's relatively slow (Sorensen, 2016b) Scheme interpreter.

6.2.1 System configuration and baseline performance

The following tests and latency benchmarks were conducted using the author's consumer-grade laptop. The hardware in use consisted of an Intel Core i5-8528U CPU with a base clock of 1.60GHz, 8GB LDDR3 RAM and integrated, Realtek ALC295 audio interface. The software platform consisted of Arch Linux, running Linux kernel 5.8.5 with the Zen patchset, *threadirqs* kernel parameter and 1000 Hz kernel timer, running JACK version 1.9.14 and Extempore version 0.8.7 with a user in the *realtime* group and JACK configured to have realtime scheduling privileges.

The JACK daemon was configured to output to the ALC295 audio interface via ALSA, with 48 kHz sample rate, a block size of 256 and a periods-per-block count of 4 – the lowest-latency settings that the author was able to configure on this hardware without incurring significant amounts of buffer under-runs.



System Information

Operating System: Arch Linux
Kernel/Version: 5.8.5-zen1-1-zen
Architecture: 64bit

System Status

CPU Scaling Governor:

System Checks

Test Name: Result
Current kernel: 5.8.5 Zen
User in audio group: Yes

JACK Status

Server Status: Started
Realtime: Yes
DSP Load: 2.05%
Xruns: 1
Buffer Size: 256 samples
Sample Rate: 48000 Hz
Block Latency: 5.3 ms

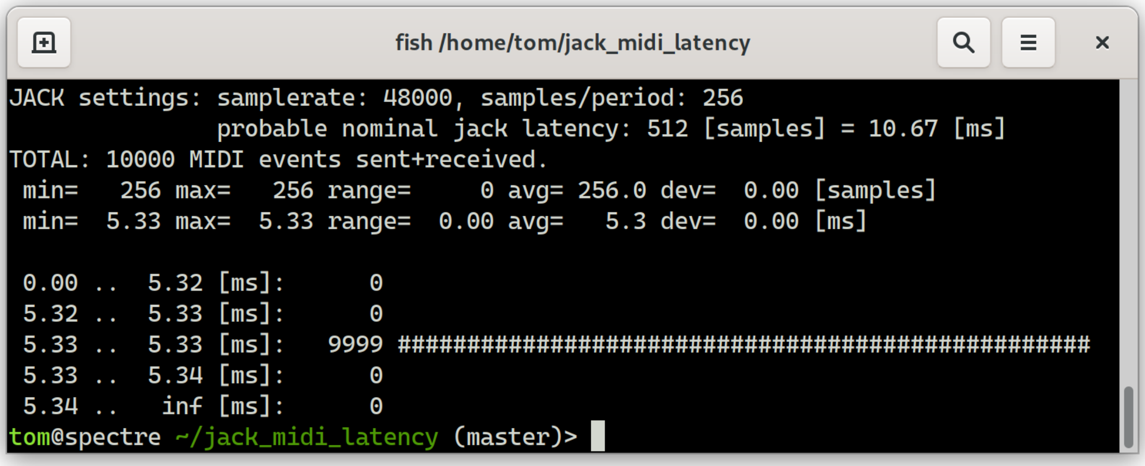
Using PulseAudio plugin bridge (Always on)

Bridge Type: ALSA -> PulseAudio -> JACK (Plugin)

Figure 53: JACK settings: 48 kHz sampling with 256 samples per buffer giving a block latency of 5.3ms.

As calculated by *Catia* above, the block latency given this configuration is 256 / 48000 s^{-1} = 5.3 ms. Note that this is *input* latency only – audio latency for input and output is likely to exceed twice this value in practice (PreSonus Audio Electronics, 2020).

Utilising the C utility jack_midi_test (see Gareus, 2013), it was experimentally verified that the minimum MIDI latency under JACK corresponds to the block latency. The utility measures MIDI round trip time between two virtual ports; connecting these ports together directly provides a minimum limit as to the time taken for MIDI transport using JACK.



```
JACK settings: samplerate: 48000, samples/period: 256
               probable nominal jack latency: 512 [samples] = 10.67 [ms]
TOTAL: 10000 MIDI events sent+received.
 min=   256 max=   256 range=     0 avg= 256.0 dev=  0.00 [samples]
 min=  5.33 max=  5.33 range=  0.00 avg=   5.3 dev=  0.00 [ms]

 0.00 ..  5.32 [ms]:      0
 5.32 ..  5.33 [ms]:      0
 5.33 ..  5.33 [ms]:   9999 ##################################################
 5.33 ..  5.34 [ms]:      0
 5.34 ..   inf [ms]:      0
tom@spectre ~/jack_midi_latency (master)>
```

Figure 54: Testing minimal MIDI round-trip time between two JACK virtual ports.

A minimum round-trip latency of 5.3ms is therefore established as a lower limit and point of reference for the ensuing discussion and benchmarking of the Extempore MIDI project.

6.2.2 Timing using Extempore

Extempore exposes the Scheme function `clock:clock`, which is a FFI wrapper to get the OS' underlying high-precision real time clock information – on Linux, it calls `clock_gettime` with the `CLOCK_REALTIME` parameter – which is returned with nanosecond precision.

```
;; Return execution time of (eval expr) in ms
(define (time expr)
  (let ((start-time (clock:clock)))
    (eval expr)
    (* 1000 (- (clock:clock) start-time))))
```

Figure 55: A procedure to measure the run-time of the expression `expr`*, expressed in milliseconds.*

Timing the evaluation of a Scheme primitive – zero – using the above function was repeated 10,000 times using the following snippet and the average execution time measured in order to give an estimate of clock precision and a lower-bound time taken for invocation of `eval` within the Scheme interpreter:

```
(let ((test-count 10000))
  (/ (foldl + 0 (map (λ () (time '0)) (range test-count)))
     test-count))
```

Figure 56: Determining the lower-bound on Scheme execution time.

This returned the value `0.000888`, corresponding to 888 nanoseconds and thereby demonstrating sub-microsecond precision of the system timer – more than adequate for the present purpose of dealing with time values of millisecond order of magnitude.

6.2.3 MIDI round-trip time in Extempore

The following code was used to benchmark minimum round-trip time using the above Extempore framework:

```
(define input (midi:create-port 'in "Extempore benchmark input"))
(define output (midi:create-port 'out "Extempore benchmark output"))
(define start-time 0)
(define input-stream (midi:input-port->stream input))
(define round-trip-times `())
(define benchmark-times 10000)

(stream:for-each (λ event
                   (let ((runtime (- (clock:clock) start-time)))
                     (set! round-trip-times (cons runtime round-trip-times))
                     (callback (now) run-benchmark)))
                 input-stream)

(define (run-benchmark)
  (if (>= (length round-trip-times) benchmark-times)
    (for-each (λ (time)
                (display time)
                (display ",")) round-trip-times)
    (begin
      (set! start-time (clock:clock))
      (midi:send output 1 2 3 4))))

(with-output-to-file "benchmark.csv" run-benchmark)
```

Figure 57: Round-trip-time for MIDI messages using this project's Extempore framework.

This establishes two virtual ports – one input, one output – which were connected to each other by the author using JACK. Ten thousand MIDI events are thereafter sent between the ports, measuring the time between sending and receiving each event; the details are saved in comma separated values format to an external file for analysis.

These results are plotted below. The range of observed latency values was from 11.35ms through to 23.23ms; the median time was 15.68ms.

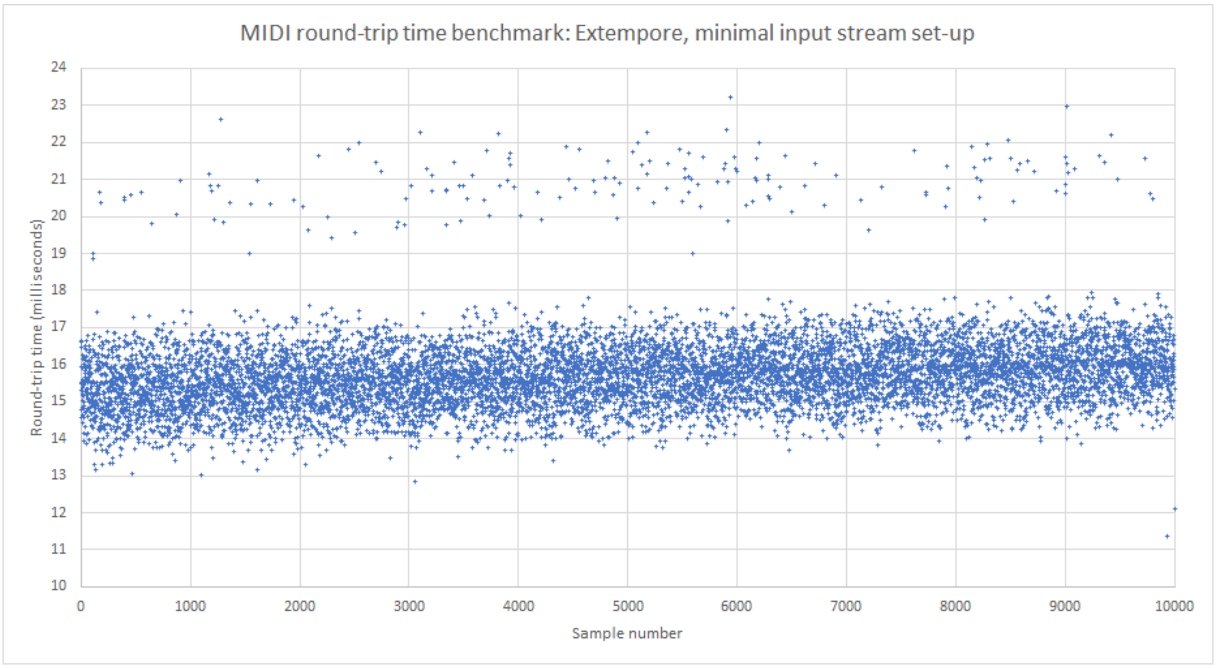


Figure 58: A graph of round-trip times in milliseconds for a minimal Extempore MIDI streams example featuring 10,000 samples.

Given the minimal `jack_midi_latency` tool – which uses the JACK C API directly – achieves round-trip latencies of 5.3 ms, this implies a median overhead of approximately 10 ms from using Extempore and this project's asynchronous stream interface for receiving events. This is a significant amount – the Extempore round-trip latencies pictured above already significantly exceed the literature's conventional figure of 10 ms as an upper threshold for latency (McPherson et al., 2016) – although not so high as to make a live MIDI instrument unplayable.

6.3.4 MIDI round-trip time using pure XTLang and librtmidi

In the interest of determining the cause of this latency, a benchmark was conducted of the latency achievable in Extempore using a pure XTLang interface into librtmidi. This was achieved using the following code, which uses only XTLang functions to send and receive MIDI messages by registering a librtmidi callback function, calculating the time difference using the same logic as in the previous example.

```
(bind-val start_time double 0.0)
(bind-val midi_in RtMidiOutPtr (make-rtmidi-in-port "Benchmark"
RTMIDI_API_UNIX_JACK))
(bind-val midi_out RtMidiInPtr (make-rtmidi-out-port "Benchmark"
RTMIDI_API_UNIX_JACK))

(bind-func benchmark_send
  (λ ()
    (set! start_time (clock_clock))
    (rtmidi_send midi_out 9 0 60 127)
    void))

(bind-func rtmidi_callback:RtMidiCCallback
  (λ (time:double message:i8* message_size:i64 user_data:i8*)
    (printf "%f\n" (* 1000.0 (- (clock_clock) start_time)))
    (set! start_time 0.0)
    void))

($ (rtmidi_in_set_callback midi_in (cast (get_native_fptr rtmidi_callback)
RtMidiCCallback) 0:i8*))

(let loop ((n 0))
  (if (<= n 10000)
    (if (= 0.0 ($ start_time))
      (begin
        ($ (benchmark_send))
        (loop (+ n 1)))
      (callback (+ (now) 100) loop n))))
```

Figure 59: Benchmarking librtmidi round trip time latency when used directly with XTLang in Extempore.

Since using `callback` within the `rtmidi_callback` function in order to trigger sending the next MIDI event in a way analogous to the previous Scheme example caused segmentation faults – and, alas, there are minimal effective ways of debugging XTLang code, a point revisited later – the logic was re-structured so as to use start-time as a way of keeping track when it was safe to send another event; it is cleared to zero when the last event has been received.

The results show that the direct librtmidi callback interface, accessed through XTLang in Extempore, achieves very good, low round-trip latency values. Below is a plot of 10,000 round trip time samples, obtained using the above code listing.

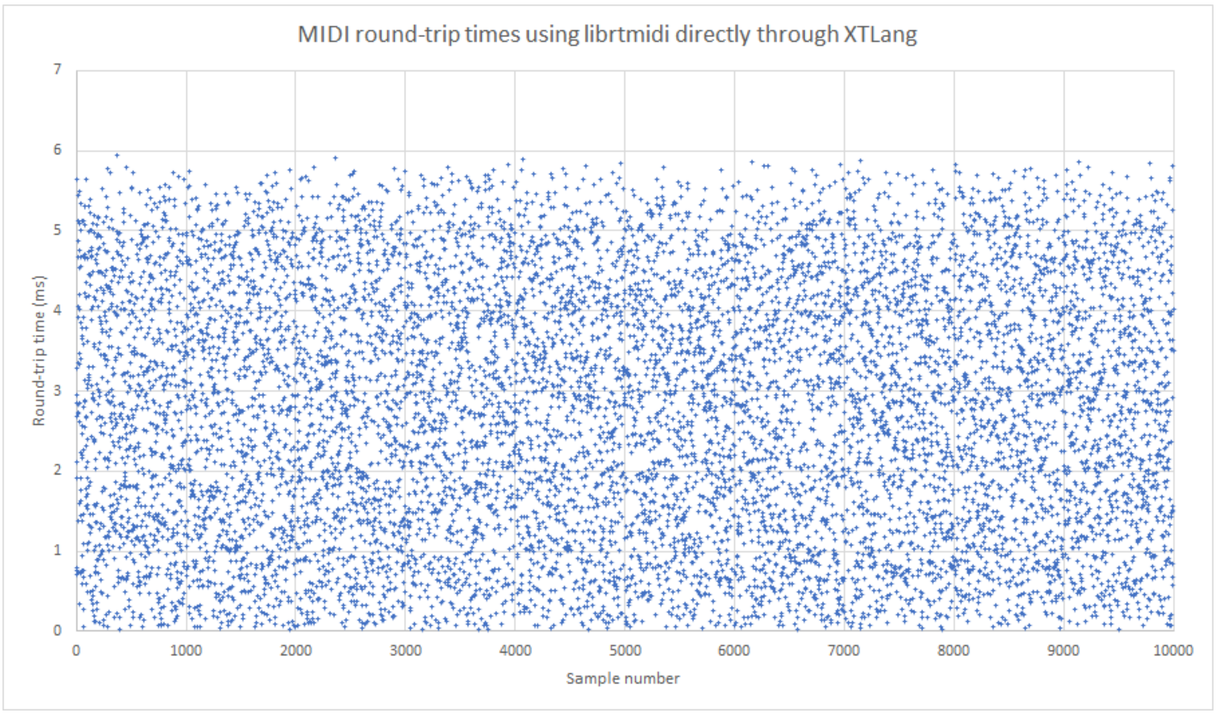


Figure 60: A scatter graph of round-trip times in milliseconds obtained using the above code listing.

As is immediately apparent, the latencies are much lower than in the asynchronous streams interface example – the range here is from a suspiciously low 0.02 ms through to a maximum of 5.95 ms. The median round trip time was a respectable 2.78 ms; the distribution throughout the range was relatively even, as can been seen from the graph. It is not clear to the author why this benchmark is able to achieve latencies lower than the `jack_midi_latency`; this would be worth further investigation as both appear to internally use the JACK API's `jack_set_process_callback` to trigger a callback function on receipt of data from JACK.

6.3.5 Scheduler performance in Extempore

This poses the question of from where the additional latency encountered with the asynchronous streams MIDI interface arises.

One possible culprit is the Extempore scheduler, which is invoked in the Scheme but not XTLang example. As noted above, the Extempore scheduler is written in native C++ and is part of the Extempore interpreter infrastructure, not amenable to modification from user code – a gap in the "full-stack" justification for Extempore, although it does appear that an XTLang replacement is under development.

A simple benchmark was executed to generate statistics regarding the minimum execution delay encountered from scheduling events. A series of events were scheduled as soon as possible – by passing `(now)` as the time to execute parameter – and the delay between scheduling events and their execution measured.

```
(define scheduler-latencies '())

(define (benchmark-scheduler start-time)
  (set! scheduler-latencies (cons (- (clock:clock) start-time) scheduler-
latencies))
  (if (< (length scheduler-latencies) 10000)
    (callback (now) 'benchmark-scheduler (clock:clock))
    'finished))

(benchmark-scheduler (clock:clock))
```

Drawing 4: Taking 10,000 samples of Extempore scheduler delay.

Plotting the latencies (after conversion to milliseconds) captured in the `scheduler-latencies` list results in the graph below:

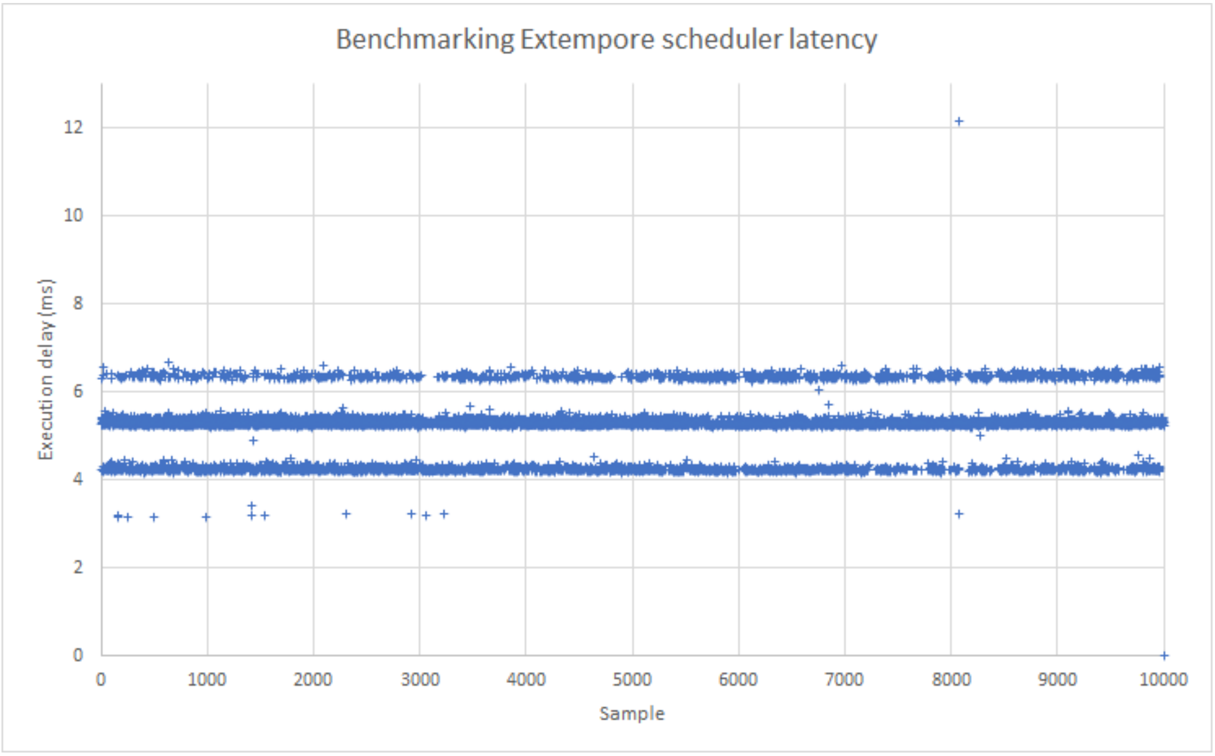


Figure 61: Benchmarking execution delay caused by Extempore's scheduler.

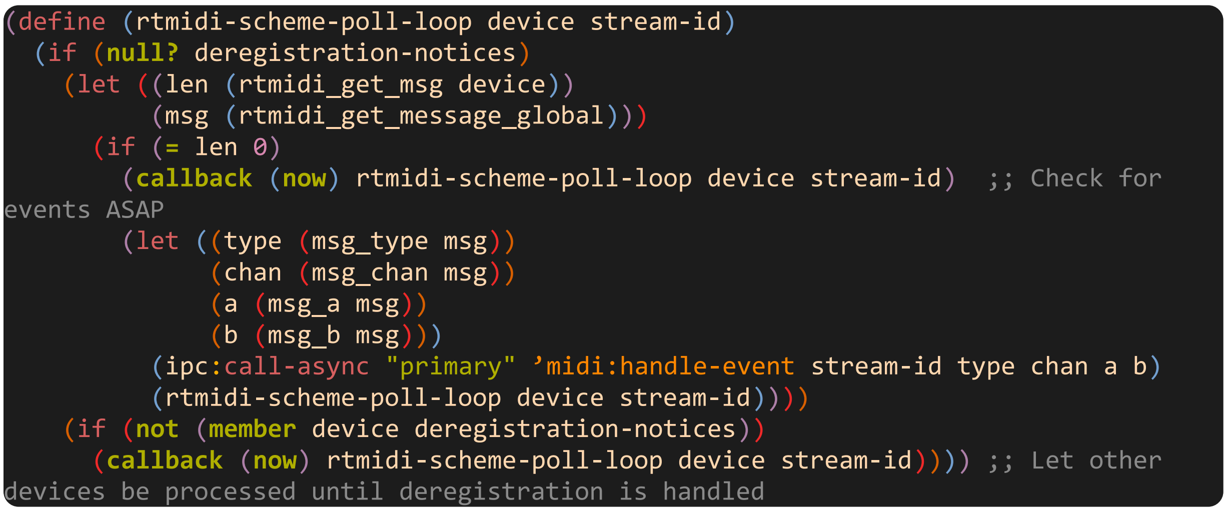
The minimum observed value – although this was only one data point – was 0.0026ms. The maximum observed value – again, a sole outlier – was 12.15ms. The median delay incurred while waiting for the scheduler to execute a piece of code was 5.28 ms – representing approximately double the median MIDI round trip time measured in the

XTLang librtmidi example without actually having performed any useful processing of events.

6.3.6. Optimising Scheme MIDI round-trip-time

It is unclear why the scheduler performs so poorly given Extempore's real-time common use cases – the production of live audio-visual experiences, where a 5 millisecond delay can be significant. Given project time constraints, such investigation is considered in the further work section below. Nevertheless, given this knowledge, this project's asynchronous streams and MIDI interface were re-worked to eliminate calls to `callback` – the Scheme function to add an item to the Extempore scheduler queue – so far as is possible and benchmarked for round-trip-time latency again, post-optimisation.

Modification of the rtmidi-stream.xtm code listing to remove unnecessary uses of `callback` – such as by refactoring the callback function in the polling loop so as to call `ipc:async-call` directly rather than through a closure – and re-running the benchmark on a further 10,000 sample set resulted in obtaining a median round trip time of 10.02 ms, an improvement of around 5 ms, corresponding to the approximate median scheduling delay.



```
(define (rtmidi-scheme-poll-loop device stream-id)
  (if (null? deregistration-notices)
    (let ((len (rtmidi_get_msg device))
          (msg (rtmidi_get_message_global)))
      (if (= len 0)
        (callback (now) rtmidi-scheme-poll-loop device stream-id)  ;; Check for
events ASAP
        (let ((type (msg_type msg))
              (chan (msg_chan msg))
              (a (msg_a msg))
              (b (msg_b msg)))
          (ipc:call-async "primary" 'midi:handle-event stream-id type chan a b)
          (rtmidi-scheme-poll-loop device stream-id))))
    (if (not (member device deregistration-notices))
      (callback (now) rtmidi-scheme-poll-loop device stream-id)))) ;; Let other
devices be processed until deregistration is handled
```

Figure 62: The optimised version of the polling loop - use of the scheduler has been eliminated for callbacks but is still required for triggering the next poll.

Searching for further invocations of `callback` in time-critical code within the project drew attention to the streams API utilising the scheduler as a means to queue callbacks for execution. Stripping this out was simple – stream callbacks could simply be executed during the callback processing loop rather than scheduled for future execution. The downside to this change, however, was that any code utilising continuations for non-local exits within callbacks would likely break the execution of the callback processing loop, preventing other callbacks in the list from being reached and applied. One key application of continuations in this way is the use of `stream:await` – utilised heavily in the unit testing code. Fortunately, the issue can be side-stepped by utilising the scheduler for asynchronous execution in the callback itself, rather than in the callback handler – as per the new definition of `stream:await` below:

```
(define (stream:await stream)
  (call/cc (λ (current-continuation)
    (letrec ((callback-fn (λ args
                            (callback (now)
                                      (λ ()
                                        (stream:remove-callback! stream callback-fn)
                                        (apply current-continuation args))))))
      (stream:register-callback! stream callback-fn)
      (*sys:toplevel-continuation* 'awaiting)))))
```

Figure 63: Using continuations within stream callbacks now requires wrapping the continuation application in an Extempore scheduler callback.

Once these changes had been made, the initial benchmark – sending and receiving 10,000 MIDI events and measuring the round-trip latency – was again performed. The results are plotted below:

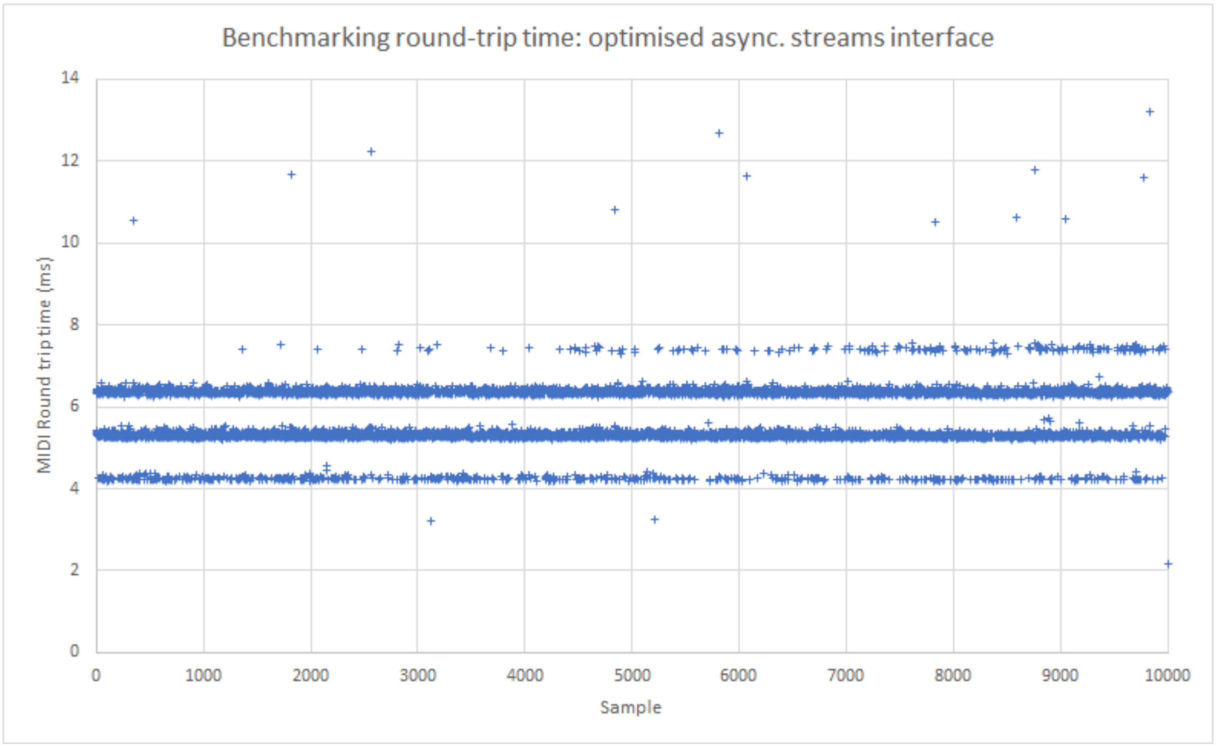


Figure 64: Benchmark results for optimised asynchronous streams interface.

The minimum round-trip time with the optimised Scheme, asynchronous streams interface was measured at 2.18 ms (compared to 11.35 ms before removing the scheduler invocations – a reduction of 80.8%) and a maximum round-trip time of 13.22 ms (compared to 23.23 ms previously) – although these values represent a very small percentage of outliers at the far edge of the distribution. The median round trip time was down to 5.34 ms – compared to 15.68 ms previously – a net reduction of 66% and thereby bringing the ball-park latency target of 10 ms discussed by McPherson et al. (2016) for a complete MIDI input, processing and audio output system into the realm of achievability.

The distribution of the round trip times seen on the graph is strikingly similar to the distribution obtained for the benchmarking of the Extempore scheduler itself – as is the median latency, 5.28 ms for the scheduler versus 5.34 ms for MIDI round trip time using the Scheme framework. The scheduler's execution delay time now appears to be the determining factor for the MIDI round-trip latency. This is thus the lowest achievable limit without modification of the Extempore code base to allow faster scheduled task execution; the scheduler is required in order to periodically execute the poll loop. Without this – for instance, if the polling loop recurs using tail recursion rather than (immediate) temporal recursion using the scheduler – the polling code can only listen to one device at a time and is ultimately killed by the Extempore interpreter for taking too long to execute.

6.3.7 Asynchronous stream overhead

Having achieved acceptable levels of latency using the MIDI framework, one final benchmark is conducted pertaining to the asynchronous streams framework – since in practice stream callbacks may be nested several layers deep as part of a processing pipeline for incoming MIDI events, as is seen with the holding and recording abstractions defined in *recording.xtm*.

A synthetic benchmark of nesting eight stream callbacks is tested for total execution time, obtaining 5,000 samples. This is primarily a test of the speed of the Extempore Scheme interpreter as it does not depend upon MIDI transport – but gives a ballpark figure for how additional Scheme processing may affect latency. A pair was passed through a sequence of eight, nested steam callbacks , each performing a fundamental mathematical operation (see benchmark-asyncstream.xtm for details).

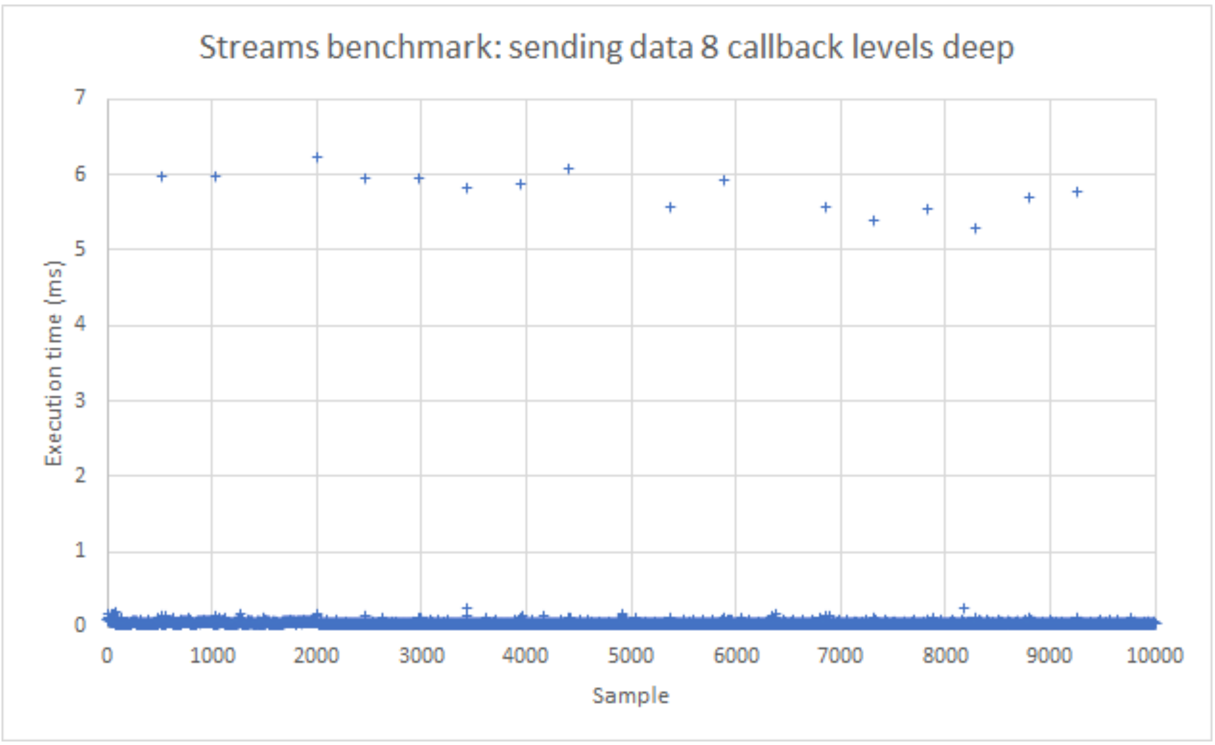


Figure 65: Measuring the overhead incurred passing data through 8 levels of stream callbacks in Scheme.

The median execution time is 45 microseconds – as can be seen on the graph, the distribution is tightly clustered around this value. There is, however, a row of delayed execution times – peaking at a significant 6.23 ms – all with a similar, approximately six millisecond delay.

These results are generally re-assuring from a MIDI latency perspective – stream processing of data is unlikely to have a significant effect on latency on average, although very occasionally significant latency of approximately six milliseconds is introduced. It is unclear why this is the case – more in-depth analysis of Extempore's performance is out of the scope of the present project – one explanation could be that these delays correspond to stop-the-world garbage collection events.

6.4 Evaluating Extempore

6.4.1 Scheme and XTLang

In this section, an evaluation of Extempore as a development environment – and target – is elaborated, particularly focusing on Extempore's raison d'etre and particular contribution to the live coding software environment niche: the combination of low-level XTLang and high-level Scheme to produce a full-stack live coding language.

As discussed by Sorensen (2016b), the XTLang and Scheme split is almost entirely driven by performance considerations; a numerical benchmark is cited whereby the XTLang example performs some 300 times faster than the equivalent example using Extempore's Scheme interpreter. This is ostensibly due to XTLang's ahead-of-time compilation into native code using the Extempore compiler, utilising the significant optimisation capabilities of the underlying LLVM framework. The Scheme interpreter, on the other hand, is admittedly relatively slow (Sorensen, 2016b), adapted from Sorensen's previous project of Impromptu and, before that, loosely adapted from TinyScheme; code is interpreted and no effort is made regarding ahead-of-time compilation of Scheme procedures.

It is an open question as to whether a more performant Scheme interpreter would be able to reduce the need for XTLang and the extent to which this displacement would be possible. The recently open-sourced Chez Scheme, for instance, features an optimising ahead-of-time compiler which compiles Scheme expressions to native code prior to execution (Cisco Systems, 2020) and performs very well on Scheme performance benchmarks; the Racket project, for instance, is in the process of abandoning its purpose-built, optimised Scheme interpreter and adopting Chez in lieu (Flatt, 2020). At least one project – Prokopchuk's (2017) *ad libitum* – appears to be successfully using Chez Scheme to write Extempore-style digital sound processing (DSP) callbacks, something which is an XTLang-specific niche in Extempore owing to performance considerations.

Re-implementation in a more performant Scheme would also not sacrifice the "live coding" principles of Extempore, provided that a similar non-premptive scheduler mechanism was implemented – as can be seen in *ad libitum* – in order to provide the dynamic lookup facility for scheduled symbols at run-time. Following this approach would limit the role for XTLang to two areas: direct system calls and other necessarily low-level system programming and its use as a foreign function interface (FFI).

6.4.2 XTLang for systems programming

While XTLang code is hot swappable – as per Sorensen's (2018) ambition – *provided that* the Extempore scheduler is invoked on each function-call which requires dynamic look-up, the author did not find it a convenient language during the course of this project for systems programming. Compared to a mature language like C, tooling for XTLang is essentially non-existent. One frequent frustration encountered when attempting to implement behaviour using pure XTLang concerned the lack of debugging facilities. Error messages are often cryptic, providing no line numbers nor stack trace – or may not happen at all, as mentioned in the previous section; attempting to set up a basic callback using the Extempore scheduler in XTLang resulted in a segmentation fault and no means to more deeply investigate the issue. While such issues can occur in C, well-established tools such as the GNU Project Debugger (GNU Project, 2020) provide the ability to introspect running C code, trace execution, probe variables and memory and even inject code mid-execution. Given the small size of the project, such tools will likely never exist for XTLang – one is limited to liberal use of `printf` for debugging. It is telling that Extempore's standard library itself largely uses XTLang for FFI rather than for behaviour.

6.4.3 XTLang as a foreign function interface

In practice, XTLang is largely used for its ability to serve as an FFI. System dynamic libraries may be loaded using XTLang and XTLang wrappers for library functions generated by using the syntactic forms `bind-dylib` and `bind-lib` respectively, as seen below.

```
;; set up the current dylib name and path (for AOT compilation)
(bind-dylib libportmidi
  (cond ((string=? (sys:platform) "OSX")
         "libportmidi.dylib")
        ((string=? (sys:platform) "Linux")
         "libportmidi.so")
        ((string=? (sys:platform) "Windows")
         "portmidi.dll"))
  "xtmportmidi")

(bind-lib libportmidi Pm_Initialize [PmError]*)

;; /**
;;     Pm_Terminate() is the library termination function - call this after
;;     using the library.
;; */
(bind-lib libportmidi Pm_Terminate [PmError]*)
```

Figure 66: XTLang as FFI: an extract from Sorensen's portmidi wrapper.

All that is required is passing in an appropriate XTLang type reference for the function signature – specifying its parameters and return type (Sorsensen, 2016c). For examples such as the above, this works well. It becomes difficult, however, when the C library requires passing *structs* as arguments which consist of many elements. XTLang has no support for defining struct types – using them, instead, requires specifying each and every struct element manually, in sequence. The following example – the author attempting to implement a basic wrapper around *libwebsockets*, a C library – demonstrates that this can easily lead to confusion and error without a means of naming or otherwise keeping track of constituent parts:

```
bind-type lws_context_creation_info
<i64,i8*,lws_protocols*,lws_extension*,i64,i8*,i8*,i8*,i8*,i8*,i8*,i64,i64,i64,i64,
i64,i64,i64,i64,i16,i16,i64,i64,i64,i64,i8*,i8*,i8*,lws_protocol_vhost_options,i64,
i8*,lws_http_mount*,i8*,i64,i64,i64,i64,i64,i64,i64,i8*,i8*,i8*,i64,i8*,i64,i64,i8*
,i64,i64,i8*,i64,i64,i8*,i64,i64,i64,i64,i16,i16,i32,i8*,i8*,i64,i64,i64,i64,i64,i6
4,i64,i64,i64,i8*,i8*,i8*,i8*,lws_protocols*,i64,i64,i64,i64,i64,i64,i64,i8*,i8*,i8
*,i64,i64,i8*,i64,i64,i8,i8,i64,i64>)
```

Figure 67: Working with large compound types in XTLang quickly becomes unworkable.

If we are in the position of only needing XTLang for FFI – such is the case for this, MIDI-specific project – it may be worth considering the options available in other Scheme environments. Chez Scheme, for instance, features a fully-powered C FFI with support for named records, equivalent to C structs, allowing similar dynamic binding to external libraries – but also has the additional advantage of allowing compatible C code to interface with Scheme objects (Cisco Systems, 2020), something not possible across the XTLang/Scheme divide despite their tighter coupling in Extempore.

6.4.4 Non-preemptible XTLang

One consideration against the total elimination of XTLang – or a similar low-level language – concerns garbage collection, major, stop-the-world events being one potential cause for the up to 6 ms spikes in latency witnessed in the Extempore Scheme callbacks benchmark above.

XTLang code, while executing, has free reign of the processor core; it will not be pre-empted by any other mechanism within Extempore, and this is explicitly stated as part of the design justification for Extempore in Sorensen's (2018) thesis, who viewed this ability as key for production of a workable real-time system.

It is an open question – although outside the scope of this project – as to whether a re-implementation of the Extempore stack, such as *ad libitum*, would be significantly affected by garbage collection pauses. If so, this may be most noticeable during continuous audio playback generated by a pure Scheme DSP callback function – monitoring for audio buffer under-runs compared to an equivalent Extempore/XTLang or C example could be an effective benchmark.

8. Future work

As noted above, further research into whether the Extempore architecture could be replaced with a re-implementation in the more performant Chez Scheme environment would be worthwhile – particularly in order to allow for more objective evaluation as to the utility of a language such as XTLang and the niche into which it falls. Prokopchuk's (2017) *ad libitum* could serve as a foundation for this.

Given more time, there are several elements of this project which could be developed further. Of particular interest to the author – given the primacy of code in the live coding paradigm represented by Extempore's design – would be a mechanism to alow interoperability between Extempore and a code editor. This could be achieved using an editor plugin, for instance, which communicates via message passing with the Extempore interpreter in order to allow Extempore code to modify the contents of the user's editor buffer. The MIDI abstractions developed herein provide much of their functionality "behind the scenes", as black boxes unavailable for real-time modification in the way that, for instance, a pattern language sequence in one's code editor could be. Having this functionality would allow this behaviour to become more visible and directly modifiable; having recorded and sequenced a note pattern, for instance, editor interoperability would allow the generated Scheme pattern-language sequence to be outputted into the user's editor buffer where it would be immediately amenable to modification and use.

Further work may be useful on the pattern language sequencer's algorithm in order to bring it in line with some changes made later on in the project to the recorder's timing algorithm – in particular, the sequencer currently ignores pauses prior to the first note, always starting a sequence on the beat. It is possible that MIDI timestamp information, available via *librtmidi* but presently ignored, could also be incorporated in order to provide more precise MIDI event timings – accommodating for the time the events have spent in the system queue – which may marginally improve the accuracy of the recorder's time quantisation for notes.

One minor improvement would also be to provide a new, low-level JACK wrapper in XTLang built around the libjack C API. At present, librtmidi spawns a new JACK process name for each new virtual port created, cluttering the JACK connection graph; using JACK directly would allow Extempore to have one unified entry in the JACK graph with all Extempore-owned virtual ports visible in one location.

9. Conclusion

This project has explored an approach for interfacing with MIDI equipment in Extempore for the purpose of achieving common digital music goals – arpeggiation, sequencing, recording and looping. A flexible Scheme-based set of abstractions was developed in order to handle multiple MIDI devices and provide the above functionality, with an overview of implementation, justification thereof and discussion of some of the issues raised by – and benefits of – programming with Extempore's unique approach to the domain of live, procedural audio generation. Unit testing was utilised as a means of showing correctness of behaviour so far as possible; detailed performance analysis of the underlying framework demonstrated areas in need of optimisation and resulted in the Scheme MIDI interface achieving acceptable MIDI latencies in the realm of five milliseconds.

With Extempore and XTLang, Sorensen (2018) has set out to produce a live coding environment with reach from low-level systems programming to high-level command and control; as demonstrated by his own myriad examples, a goal largely achieved. Questions remain, however, as to the utility for XTLang – and indeed its long-term sustainability – given its present immaturity, poor developer tooling and the potential that a more efficient Scheme interpreter could render much of its present use cases unnecessary. Further research into replicating the Extempore stack within the recently open-sourced Chez Scheme would be illuminating in this regard to provide an objective basis for evaluating the proper niche of XTLang – and hence Extempore – within the modern programming languages ecosystem.

References

Aaron S. (2016) Sonic Pi – performance in education, technology and art. *International journal of performance arts and digital media,* 12(2), pp. 171 – 178.

Aaron S. and Blackwell A. F. (2013) From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages. *FARM '13: Proceedings of the first ACM SIGPLAN workshop on functional art, music, modelling & design.* September 2013, pp 35 – 46.

Abelson H., Sussman G. J and Sussman J. (1996) *Structure and interpretation of computer programs.* 2nd ed. London, MIT Press.

Ableton (2020) *Ableton live: MIDI.* Available at: https://help.ableton.com/hc/en-us/sections/202237409-MIDI. Accessed 23rd August 2020.

Brandt E. And Dannenberg R. B (1998) Low-latency music software using off-the-shelf operating systems. *Proceedings of the International Computer Music Conference.* San Francisco: International Computer Music Association.

Brown A. and Sorensen A. (2009) Interacting with generative music through live coding. *Contemporary music review,* 28(1), pp. 17 – 29.

Cisco Systems (2020). *Chez Scheme Version 9 User's Guide.* Available at: https://cisco.github.io/ChezScheme/csug9.5/csug9_5.pdf. Accessed 10th September 2020.

Cox G., McLean A. and Ward A. (2001). The aesthetics of generative code. *Proc. of generative art*, 51.

Dahl S. and Bresin R. (2001) Is the player more influenced by the auditory than the tactile feedback from the instrument? *Proceedings of the COST G-6 conference on digital audio effects (DAFX-01), Limerick, Ireland.* December 6-8, 2001.

Dybvig R. K. (2009) *The scheme programming language.* 4th Ed. Cambridge, Massachusetts: MIT Press.

Flatt M. (2020) *Racket-on-chez status: February 2020.* Available at: https://blog.racket-lang.org/2020/02/racket-on-chez-status.html. Accessed 10th September 2020.

Friberg A. And Sundberg J. (1995) Time discrimination in a monotonic, isochronous sequence. *The journal of acoustical society of America,* 98(2524), pp. 12 – 14.

Garius R. (2013) *x42/jack_midi_latency: utility to measure JACK audio latency & jitter.* Available at: https://github.com/x42/jack_midi_latency. Accessed 1st September 2020.

Garret R. (2002) *Lisping at JPL*. Available online at: http://www.flownet.com/gat/jpl-lisp.html. Accessed 11th August 2020.

Hardwick P. (2016) *Unit tests are the best documentation.* Available at: https://capgemini.github.io/development/unit-tests-as-documentation/. Accessed 28th August 2020.

Jorgensen P. C. (2014) *Software testing: a craftman's approach.* 4th ed. London: CRC Press.

Kantor I. (2020) *Async/await.* Available at: https://javascript.info/async-await. Accessed 18th August 2020.

KXStudio (2020) *KXStudio: applications: Catia.* Available at: https://kx.studio/Applications:Catia. Accessed 15th August 2020.

Lattner C. A. (2002) *LLVM: an infrastructure for multi-stage optimization.* MSc thesis, University of Illinois, 2002.

Leger P. and Fukuda H. (2016) Using continuations and aspects to tame asynchronous programming on the web. *MODULARITY companion 2016: companion proceedings of the 15th international conference on modularity, March 2016.* pp. 79 – 82.

Letz S., Fober D, Orlarey Y and Davis P. (2005) Jack audio server: MacOSX port and multi-processor version. *Sound and Music Computing Converence*, 2004, Paris, France. pp. 177 – 183.

Madore D. (2002) *A page about call/cc.* Available at: http://www.madore.org/~david/computers/callcc.html. Accessed 15th August 2020.

McCarthy J. (1978) History of LISP. *ACM SIGPLAN Notices, August 1978.*

McCarthy J. and Levin M. I. (1985) *LISP 1.5 programmer's manual*. 2nd ed. Cambridge, Massachusetts: MIT Press.

McLean A and Rohrhuber J. (2003) Live coding in laptop performance. *Organised sound*, 2003.

MIDI Manufacturer's Association (2020) *Summary of MIDI messages.* Available at: https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message. Accessed 11th September 2020.

Nilson C. (2007) Live coding practice. *Proceedings of the 2007 conference on new interfaces for musical expression (NIME07)*. New york, NY, USA.

Oracle (2020) *ArrayList (Java Platform SE 8).* Available at: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. Accessed 22nd August 2020.

Phillips D. (2005) A user's guide to ALSA. *Linux journal,* 2005(136), p. 3.

PreSonus Audio Electronics (2020) *Digital audio latency explained.* Available at: https://www.presonus.com/learn/technical-articles/Digital-Audio-Latency-Explained. Accessed 1st September 2020.

Prokopchuk R. (2017) *Ad libitum: scheme live coding environment.* Available at: https://github.com/ul/ad-libitum. Accessed 11th September 2020.

Python Software Foundation (2020) *The python standard library*. Available at: https://docs.python.org/3/library/. Accessed 11th August 2020.

Rothstein J. (1995) MIDI: A comprehensive introduction. 2nd ed. Madison, Wisconsin: A-R Editions, Inc.

Rudrich D. (2017) *Timing-improved guitar loop pedal based on beat tracking.* MSc thesis, University of Music and Performing Arts Graz, January 2017. Available at: https://iem.kug.ac.at/fileadmin/media/iem/projects/2015/rudrich.pdf. Accessed 23rd August 2020.

Scavone G. (2019) *The RtMidi tutorial.* Available at: https://www.music.mcgill.ca/~gary/rtmidi/. Accessed 18th August.

Sorensen A. (2005) Impromptu: an interactive programming environment for composition and performance. *Proceedings of the Australasian Computer Music Conference, 2005*.

Sorensen A. (2013) *The many faces of a temporal recursion.* Available at: http://extempore.moso.com.au/temporal_recursion.html. Accessed 18th August 2020.

Sorensen A. (2016a) *Extempore docs: xtlang types.* Available at: https://digego.github.io/extempore/types.html. Accessed 14th August 2020.

Sorensen A. (2016b) *Extempore docs: the Extempore philosophy.* Available at: https://digego.github.io/extempore/philosophy.html. Accessed 14th August 2020.

Sorensen A. (2016c) *Extempore docs: C-xtlang interop.* Available at: https://digego.github.io/extempore/c-xtlang-interop.html. Accessed 15th August 2020.

Sorensen A. (2016d) *Extempore docs: unit testing in Extempore.* Available at: https://digego.github.io/extempore/testing.html. Accessed 31st August 2020.

Sorensen A. (2018) *Extempore: the design, implementation and application of a cyber-physical programming language.* PhD thesis, The Australian National University.

Sorensen A. (2020) *Extempore docs: pattern language.* Available at: https://extemporelang.github.io/docs/guides/pattern-language/ Accessed 16th August 2020.

Steinberg (2020) *Cubase Pro 10.5.20: operation manual.* Available at: https://steinberg.help/cubase_pro/v10.5/en/index.html. Accessed 23rd August 2020.

Tanimoto S. L. (1990) VIVA: A visual language for image processing. *Journal of visual languages and computing, 1,* pp. 127 – 139.

Travis C. And Lesso P. (2004) Specifying the jitter performance of audio components. *Audio Engineering Society 117th Convention, San Francisco, CA.* 2004. October 28-31.

Wang G., Cook P. R. and Salazar S. (2015) ChucK: a strongly timed computer music language. *Computer music journal*, 39(4), pp. 10 – 29.

Wang Y., Stables R. And Reiss J. (2015) Audio latency measurement for desktop operating systems with onboard soundcards. Audio Engineering Society, 128th Convention. London, May 22-25.

Wickham H. (2019). *Advanced R.* 2nd Ed. London: CRC Press.

Reflection

This is a reflection on the process of completing this project, structured using the model of Driscoll (1994).

What?

This project, as selected on PATS, was titled “Real-time Music Programming with Extempore”. I chose the project after watching Sorensen’s OS Convention speech, featured on the Extempore documentation website – Extempore’s code-first philosophy intrigued me; I had not previously been involved with anything within the “live coding” domain. Being based in Scheme was an advantage also – I’ve been intrigued by Lisp-family languages for some time, in particular for the code-generation and functional-style reliance on higher order functions and first-class procedures, but had yet had reason to engage in a more substantial project using one.

On discussion with my supervisor, Prof. David Marshall, it was decided to go down the route of MIDI interfacing. Preliminary suggestions were to develop arpeggiation functionality, perhaps followed by some kind of recording or looping functionality, as time allowed and I was able, giving that this was an unusual platform and would take some time to learn to use effectively. I proceeded iteratively, developing first a minimal arpeggiation example using Extempore’s built-in portmidi MIDI wrapper and the pattern language, learning Extempore’s standard library and syntax as I proceeded using the online documentation.

Once I felt like I had a reasonable understanding of Extempore, I decided that the built-in MIDI interface was inflexible and would not scale well – it would be neater to build another layer on top, using the partial rtmidi support available in the standard library, in order to provide a unified Scheme way of interacting with MIDI devices through virtual ports. Constructing this platform thus came second; once established, the other abstractions described above were implemented essentially in the order that they are described above.

Writing the report took a significant amount of time, perhaps underestimated initially whilst the focus was on creating a functional implementation. The analysis section in particular required some planning regarding the content; many of the unit tests required some thought to implement so as to avoid reducing the entire testing section to that of black box testing, as may be tempting given the situation where the inputs are key presses and the output that of audible noise.

So what?

Choosing to use Extempore meant intentionally choosing to devote some weeks to developing using what may be considered an archaic language, Scheme, within the relatively obscure niche of live audio-visual programming. I was not new to Scheme or Lisp, having previously been working through The Structure and Interpretation of Computer Programs (SICP), a classic computer science textbook using the language, prior to starting the MSc. Nevertheless, working with Scheme – which traditionally has a minimal to non-existent standard library – provided an excellent opportunity to develop skills in algorithmic thinking, forcing one to focus, as it does, on implementing many of the otherwise taken-for-granted routines of other, bulkier languages. This can be seen, for instance, with the need to implement array-list and array-builder-like abstractions in the report above prior to their use in the project. Scheme also provides great flexibility for design – between its excellent (due to the consistency of its syntax) macro system and such features as first-class continuations, one has essentially free reign over producing what may be considered, in the spirit of SICP, new, domain-specific languages, a point which came to mind repeatedly while completing the project.

Working with the other aspect of Extempore – XTLang – provided some interesting insight into C-interoperability with other languages; I would certainly be more confident in the future attempting to write C library wrappers and linking these with other, dynamic programming languages – as well as being more aware of the breadth of C libraries out there for use.

Completing the analysis part of the project allowed practising benchmarking an application and trying to locate the source of performance issues; identifying that the Extempore scheduler was the cause and therein significantly speeding up the streams and polling loop interface was a pleasing achievement.

Outside of the technical domain, the project naturally demanded planning, organisational and time management skills. While considered at the beginning, I ultimately did not use any time-planning tools such as GANNT charts. Such a chart, in retrospect, may have been helpful, at least at the level of granularity of specifying when I should start to devote my time to report writing versus attempting to implement more functionality. Since this was a research project, however, anything more specific may have been more of a hindrance – the project was developed iteratively out of necessity, with goals changing at the end of each iteration, rather than from a set of strict requirements. Such an approach is a miniature form of the agile process often utilised within modern software engineering teams and may serve as a personal foretaste of this.

Now what?

Moving forward, I will aim to take many of the skills developed further through completing this prolonged piece of personal work – such the aforementioned computational thinking and insight into iterative development – into my future software development career.

It is difficult to know at this stage what I would have done differently regarding this project given the benefit of hindsight. I would perhaps suggest utilising some kind of time management tool – perhaps GANNT charts – in order to provide more psychological assurance that I am on track with the workload remaining, although in practice this was, as noted, a relatively vague notion.

Perhaps a useful thing to do would have been to more thoroughly investigate the environment of the project's subject – other competing systems, other approaches – before I commenced planning or designing my ideas for the project. While I did ultimately investigate, to a surface degree, many of Extempore's competing live coding environments, this was relatively late on in the project, after I had started writing the report. For future projects, I would probably do this first – not only to provide guidance as to what is possible, but to gauge what has already been done and how my contribution can make the most of any available niche.

References

Driscoll J. (1994) Reflective practice for practice. *Senior Nurse, 13*, pp. 47 – 50.