TIDALBIND: TOWARDS INTERACTIVE REAL-TIME CONTROL AND DECLARATIVE MAPPING IN LIVE CODING

Gordan KREKOVIĆ (0000-0002-0028-7721) [1] and **Antonio POŠĆIĆ** (0000-0001-8513-6377) [2]

[1] **Visage Technologies**, Zagreb, Croatia
[2] **Academy of Music, University of Zagreb**, Zagreb, Croatia

ABSTRACT

TidalBind bridges the gap between symbolic logic and gestural immediacy in live coding by enabling real-time MIDI control in Tidal Cycles. Its novelty is the use of declarative in-code mapping of MIDI interactions based on semantic markups. Being an integral part of the code, mapping configurations become dynamic material, enabling interactivity, while reducing performer's cognitive load. TidalBind provides triggering functions that initiate certain actions, such as evaluating specific blocks or navigating the code sequence, based on conditional MIDI events. Additionally, it includes a mapping function that dynamically substitutes semantic markups with raw or linearly scaled MIDI values immediately before code execution. The variety of mappable MIDI events implies a great control flexibility. TidalBind improves the ergodynamics of live coding and maintains the "show us your screens" principle by displaying incoming MIDI data directly within the text buffer. While current limitations include the initialization of continuous parameters, TidalBind offers a more embodied and transparent live coding workflow.

1. INTRODUCTION

As performance art, computer music is defined by a tension between the easily realized flexibility of symbolic logic and the elusive immediacy of its embodiment and gestural control. Emerging in the early 21st century, live coding is a practice and, in some instances, a style of computer music that tries to expand the static facets of so-called laptop musicians into dynamic live art by mediating the technological-human relationship. In this process, live coding adopts tenets characteristic of improvisation in free jazz and non-idiomatic music [1].

Live coding resolves the distinction between logic and gestural control by privileging the symbolic: the text editor becomes the stage, the code becomes the score, and the digital instrument is made to be epistemic, existing only in the logic of the running program [2]. In this paradigm and in the context of live events, the performer is largely reduced to the act of typing, creating a visual and performative disconnect between the produced sonic affects and the presence of the artist on stage. Recent developments in Hybrid Live Coding Interfaces (HLCI) have been focused on reintegrating physical controllers – most commonly, MIDI (Musical Instrument Digital Interface) devices – into the live coding workflow [3]. This hybridization of the form is a step towards the rediscovery of the ergodynamics of traditional instruments (the nuances of velocity, continuous pressure, and tactile feedback), while retaining the generative capabilities of algorithmic patterns [4].

Nonetheless, the integration of these two approaches poses architectural and cognitive issues. In most existing systems, like those related to the live coding environment Tidal Cycles, the definition of mappings or relationships between physical controls and code parameters is handled in separate, often static, backend layers, leading to a phenomenon known as calcification, which entails the interface becoming rigid and disconnected from the improvisational flow of the code [5].

Copyright: © 2026. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2. SCORE-EMBEDDED INTEGRAL MAPPING

2.1 The Paradigm

This paper introduces the initial version of TidalBind, an add-on for Tidal Cycles based on a paradigm of score-embedded integral mapping, where MIDI interactions are defined declaratively within the Tidal Cycles code using semantic markup. By situating mapping logic inside the active text buffer alongside the musical patterns it controls, we transform the mapping from a static configuration into a dynamic, improvisational material. This approach aligns with Knuth's vision of Literate Programming [6] and Magnusson's notion of code as symbolic inscription [7], ensuring that the code serves as a complete, transparent documentation of the performance state. Technically, this allows for the real-time redefinition of the instrument: a performer can rewrite a markup tag to reassign a controller mid-performance, effectively composing the instrument alongside the music.

Furthermore, by visualising the reception of MIDI data within the code (e.g., highlighting triggered blocks and replacing markup tags with control values), the system restores the causal link between the performer's physical interventions and the sonic output. This visibility creates a critical perception of "liveness", and a feedback loop that benefits both the performer, by reducing the cognitive cost of context-switching between editor and backend, and the audience, by adhering to the "show us your screens" principle from TOPLAP's manifesto [8] in a more causally

transparent way. Ultimately, markup-based mapping represents a shift towards a fully embodied live coding practice, where the friction between the symbolic and the gestural is resolved not by hiding the wires, but by making them a legible, malleable part of the performance narrative.

2.2 Related Work

The prevailing architecture in Tidal Cycles enforces a rigid separation of concerns, sequestering MIDI mapping logic within an imperative backend layer (SuperCollider) distinct from the visible pattern language. Prominent practitioners like Kindohm and Lil Data mitigate this through extensive pre-configuration, utilising hard-coded helper functions or fixed DAW templates.

Mike Hodnick (Kindohm) largely uses Tidal to output MIDI signals to hardware. He sequences parameters on the Elektron machines using Tidal's CC patterns relying on hard-coded and explicit mapping [9]. Jack Armitage (Lil Data) integrates Tidal with DAWs like Ableton Live, typically relying on pre-configured Ableton sets to avoid the risk of live setup [10]. Alex McLean (Yaxu) introduced "control busses" in Tidal Cycles 1.7 to enable real-time control of effects, also using external MIDI controllers [11], but without a possibility to visualize incoming control values.

Several other live coding environments have explored "in-code" mapping and annotation. Strudel allows variables like `let cc = midin(...)` to be used directly in patterns, allowing the use of standard variable assignment for mapping [12]. While functional, it requires the user to write boilerplate code.

GALiCA integrates MIDI control directly into the live coding system through specialized sequencer conceptualizations [13], while Sardine leverages Python decorators to manage temporal recursion and external communication, moving configuration closer to the musical syntax [14]. Gibber establishes a critical precedent for "code annotations", using the text buffer as a canvas for dynamic signal visualization [15]. However, existing annotation frameworks predominantly visualize output signals rather than defining input mappings. The markup-based approach proposed in this paper diverges by situating the mapping definition itself within the code as a declarative, "literate" layer, transforming the mapping from a hidden backend utility into a visible, mutable component of the score.

3. LIVE CODING LOOP

3.1 Interactive Integration

In the paradigm's complete form, the aim is to allow for the usual modalities of live coding to be combined with MIDI-controlled inputs, integrating TidalBind with common programming editors like Tidal Cycles' recommended Pulsar Edit [16] via apposite plugins. Specifically, the performer/live coder will be able to employ their usual toolsets and flows, writing and manipulating the code as normal with keyboard and mouse, while also annotating blocks using TidalBind's markup and essentially expanding their code-writing practice with the gestural qualities of MIDI controllers.

When executed, the annotated blocks behave like regular Tidal Cycles code segments, which are then augmented and processed using the mapped values obtained from MIDI controls. Writing `d1 $ note "{- ! map, noteOn -}" # s "superpiano"`, for example, would play whichever note was pressed on the MIDI keyboard after this code is executed. It is important to remark that once TidalBind evaluates the markup, the mappings remain valid and available for triggering with MIDI controls even as other blocks are executed. This remains true as long as the mappings are not overwritten via other markup. In the above example, pressing MIDI note keys will trigger the d1 $ note function, each time passing the actual value of the pressed note.

Thanks to its immediacy and natural integration with existing coding patterns, this approach lends its usefulness to live performances and ad hoc written code, in particular.

3.2 Proof Of Concept

While most aspects of the approach described above have been rendered in the proof of concept provided for the purposes of this paper, the current implementation of TidalBind does not include integration with programming editors. Instead, the Tidal Cycles blocks are read in "headless" mode from a Tidal source file that contains units of code, which are then triggered (using MIDI controls, e.g. a footswitch) and executed sequentially.

The latter behavior is based on the sequential pointer which moves from the current to the next code block in the source file. In practice, the performer uses a switch pedal to move the global pointer to the next block of code from a Tidal file, which TidalBind then reads and executes. The block may contain Tidal code, markup, or both. Because of the mappings remaining valid once evaluated (see previous section), previously executed codes of blocks may be executed again without influencing the position of the global pointer, when triggered using MIDI or when the MIDI controls mapped to them change. For example, the performer may use MIDI pads to trigger some of the previously executed Tidal blocks, but then continue the performance from the global pointer position using a footswitch.

The design decision to use sequential processing of code blocks is based on the observation that performers usually modify existing blocks of code [1] or execute those they have just added, reflecting a sequential workflow. While this approach requires the code to be prepared in advance of the performance, the other aspects of its use coincide closely with the core tenets of live coding.

4. IMPLEMENTATION

4.1 Markup Structure

To achieve declarative and explicit mapping, semantic markups are embedded directly within Tidal Cycles code. TidalBind employs a specialized inline comment syntax: `{-! markup -}` to ensure backward compatibility. This encapsulation allows the same code enriched with

markup to be evaluated in a standard Tidal environment without introducing syntax errors. While MIDI interactions are not expected to work without TidalBind, musicians can still reuse and exchange the code.

Each markup adheres to a comma-delimited structure, {-! function, source, ...-}, where the mandatory initial element specifies the operation. TidalBind provides two types of functions: 1) triggering functions which initiate specific actions upon receiving conditional MIDI events, and 2) a mapping function, which dynamically substitutes the markup within the code block with raw or scaled MIDI values prior to its execution.

For most functions, the second element identifies the MIDI message. Supported identifiers include control change values (cc0 to cc127, including aliases like sustain for cc64), note events returning pitch numbers (noteOn and noteOff), note velocities (note0 to note127), the velocity of the last pressed note (velocity), and pitch band (pitchBand).

For example, {-! map, noteOn -} dynamically resolves to the pitch number upon each note-press event, while {-! triggerNextBlock, CC64, =, 127 -} serves as a navigational trigger, automatically jumping to and evaluating the subsequent Tidal code block when CC64 (typically switch pedal) transmits a value of 127.

4.2 TidalBind Functions

Triggering functions execute specified operations upon satisfying defined conditions, adhering to the syntax: {-! function, source, operator, value -}. Supported conditional operators include standard comparators (=, <, >, <=, >=) and the any wildcard, which validates the event's presence regardless of its value, rendering the value element redundant. Upon initial evaluation, a triggering function instantiates a persistent listener that remains active until the mapping is explicitly redefined or the session terminates.

The triggerBlock function is designed to be placed at the top of a Tidal code block. It enables the isolated evaluation of that specific block when the condition is met, bypassing the global sequence pointer. The following example demonstrates how a MIDI pad assigned to note 36 (C2) can be mapped to trigger a kick drum pattern:

```
 {-! triggerBlock, note36, >, 0 -}
d1 $ sound "bd"
```

Unlike triggerBlock, other triggering functions operate independently of Tidal code blocks and are defined within standalone blocks. All triggering functions adhere to a consistent markup syntax and are comprehensively listed in Table 1.

The mapping function is designed for inline application, dynamically substituting the markup within the code block with raw or scaled MIDI values immediately prior to execution. For example, the following code enables control of a low-pass filter's cutoff frequency via MIDI CC 74:

```
 d1 $ sound "bd sd" # lpf 0{-! map,
cc74, 0, 127, 100, 9000 -}
```

The map function supports two distinct syntaxes: {-! function, source -} for substituting the raw MIDI value, and {-! function, source, cMin, cMax, oMin, oMax -} for linear scaling. In the latter configuration, the controller's input range (cMin to cMax) is mapped to the designated output range (oMin to oMax).

Function	Description
triggerBlock	Evaluates the block without changing the sequence pointer.
triggerNextBlock	Moves the sequence pointer to the next block and evaluates it.
hush	Evaluates hush.
resetCycles	Evaluates resetCycles.

Table 1. Triggering functions in TidalBind.

The versatility of the map function allows for complex interactions across a variety of supported MIDI messages. For instance, the following code demonstrates how to generate a note per cycle based on the currently pressed key, while dynamically mapping the keystroke's velocity to the gain parameter:

```
 d1 $ note "{-! map, noteOn -}" # s
"superpiano" # gain {-! map, velocity,
0, 127, 0, 1 -}
```

4.3 Technical Implementation

TidalBind is a Python script which connects to Tidal using a system pipe with the interactive Haskell environment (GHCi), which communicates to SuperDirt. This architectural approach builds upon our previous work [17, 18]. The TidalBind script listens to MIDI controllers, parses the Tidal source code, resolves the markups, and executes the resulting code through the GHCi session.

5. USE CASE

The motivation for developing TidalBind emerged while one of the authors was preparing a performance that combined live coding, poetry, and piano. In some parts of the performance, Tidal code serves as a fixed-structured accompaniment with lower-level indeterminacies, while in others, the spotlight shifts from the piano and poetry to interactions with the code.

Initially, the performer used a text editor to modify and execute pre-composed Tidal code, but this approach proved inefficient, as even sequential execution required multiple keystrokes on a computer keyboard to properly position the pointer before executing next blocks of code. Playing the piano and interacting with the code became mutually exclusive, limiting expressivity and dynamism of the performance. In pieces with dominant piano parts, the author wrote most of the code in form of function compositions assigned to variables whose execution was then chained and automatized using seqP() function. Interacting with this code required frequent scrolling, triggering variable assignment, and the subsequent execution of



Figure 1. Live performance with a laptop and a piano which created an obstacle between the performer and the audience. Photo by Katarina Pešić.

`seqP()` blocks. This workflow lacked performative diversity and immediacy; significant portions of the code had to be executed before the sonic results became apparent, while the code projected on-screen appeared static and overly complex.

Furthermore, frequently switching between the piano and the laptop increased the performer's cognitive load, disrupting their state of flow and diminishing the connection with the audience.

We designed and developed TidalBind to address these observed impediments. The concept of sequential reading and execution of predefined code blocks proved to be a significant benefit in itself; it allowed the performer to rely primarily on a footswitch, minimizing hand-switching and shifts in attention between the laptop and the piano. This approach also enhanced the visual aspect of the performance, as the code appeared on the screen gradually rather than being displayed in its entirety within a static text editor. Adding markups to existing code was an intuitive process that preserved the overall code organization, though TidalBind facilitated the breaking of long `seqP()` sequences into multiple shorter segments. This made the performance more dynamic and provided greater flexibility for improvisational timing.

The hardware setup consisted of a laptop, a piano, and a MIDI controller equipped with keys, pads, and a footswitch. While the performer mainly used the footswitch to progress through the code, they also utilized MIDI pads to re-trigger specific blocks and knobs to manipulate continuous parameters.

In one particular piece without piano parts, the performer navigated the code solely via the MIDI controller, without touching the computer keyboard. This significantly contributed to faster transitions, deeper immersion, and a tighter connection with the audience, according to the performer's assessment. The code below contains two blocks from the mentioned piece intended for triggering using two neighbouring pads on the MIDI controller:

```
{-! triggerBlock, note36, >, 0 -}
d3 $ mask "{t ~ t ~ t ~ t ~ t?}%8" $
  s "[viosg:0, viosc:0]*8"
  # speed "{[1, 1.8] [0.99, 1.5] \
          \ [0.8, 1.4] [1, 1.2]}%4"
  # sustain 1 # release 0.4
  # shape 0.3 # gain 0.835
  # cutoff (range 1100 2500 rand) # room 0.4
  # size 0.7 # orbit 1

{-! triggerBlock, note37, >, 0 -}
d3 $ stack [
   (
    s "viosc:0*8"
    # speed "{0.5 [0.74 | 0.5] 1 1.25 \
             \ [1.51 | 1.4] 1.51 1.8}%8"
    # gain 0.85
   ),
   (
    s "viosc:0*8"
    # speed 1.{-! map, cc102, 0, 127, 0, 9 -}
    # gain 0.79
   )
 ]
 # sustain 1 # release 0.33 # shape 0.4
 # cutoff (range 1100 2600 rand)
 # room 0.4 # size 0.7 # orbit 1
```

A noted drawback of the current declarative implementation is the lack of specified default values within the mapping function. This results in a markup resolving into an empty string if the code block is executed before the first control value arrives. While the performer found temporary workarounds (e.g., adding a zero right before the markup if the default value is expected to be 0, or quickly changing the control parameter after executing the block of code with a problematic markup), a more robust and elegant solution for handling initialization is necessary.

In conclusion, even in this initial version, TidalBind has served as a practical tool for facilitating more immersive live coding performances. Its most effective feature appears to be the ability to navigate code using footswitches and MIDI pads. However, we are aware that such a use can be limited to code written upfront, so integration to text editor should remain a priority for future work.

6. CONCLUSIONS AND FUTURE WORK

Declarative mapping between MIDI and Tidal Cycles code proved as a promising approach for increasing ergodynamics and immediacy of gestural control in live coding. This method maintains the mapping as an integral part of the code, keeping it dynamic, visible, changeable, and available for improvisation. Although semantic markups do not allow implementation of complex logic beyond functions provided in TidalBind, the variety of MIDI messages and the ability to combine their values with triggering and mapping ensure greater flexibility than is directly available in Tidal (e.g., through control busses). In practice, TidalBind proved to be particularly effective for triggering, though somewhat limited for continuous parameter mapping.

Future versions of TidalBind should therefore focus on enhancing continuous control (handling predefined values

and using control busses to avoid re-execution for each parameter change) and achieving editor integration.

7. REFERENCES

[1] G. Kreković and A. Pošćić, "Modalities of improvisation in live coding," in *7th Conference on Computation, Communication, Aesthetics & X (xCoAx)*, 2019, pp. 199–211.

[2] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, pp. 321–330, 2003.

[3] S. W. Lee and G. Essl, "Live coding the mobile music instrument," in *International Conference on New Interfaces for Musical Expression (NIME)*, 2013, pp. 493–498.

[4] T. Magnusson, "Ergodynamics and a semiotics of instrumental composition," *Tempo*, vol. 73, no. 287, pp. 41–51, 2019.

[5] A. Sarwate, "Calcification and hybrid live-coding," in *Proceedings of Audio Mostly 2016*, 2016, pp. 55–60.

[6] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[7] T. Magnusson, "Of epistemic tools: Musical instruments as cognitive extensions," *Organised Sound*, vol. 14, no. 2, pp. 168–176, 2009.

[8] TOPLAP Wiki, "Manifesto draft," https://toplap.org/wiki/ManifestoDraft, 2025, accessed: Dec. 28, 2025.

[9] M. Hodnick, "Making music: Hardware versus computer," https://mikehodnick.ghost.io/making-music-hardware-versus-computer/, 2025, accessed: Dec. 28, 2025.

[10] "Tidal club: Live stream #8 - tidalcycles x daws," https://club.tidalcycles.org/t/live-stream-8-tidalcycles-x-daws-with-jack-armitage/1495, 2025, accessed: Dec. 28, 2025.

[11] Tidal Cycles, "Control busses," https://tidalcycles.org/docs/reference/control_busses/, 2025, accessed: Dec. 28, 2025.

[12] Strudel docs, "Midi, osc, and mqtt," https://strudel.cc/learn/input-output/, 2025, accessed: Dec. 29, 2025.

[13] L. Savoldy, "Galica: A gestural approach to live coding algorithms," Doctoral dissertation, Massachusetts Institute of Technology, 2024.

[14] R. M. Forment and J. Armitage, "Sardine: a modular python live coding environment," in *International Conference on Live Coding (ICLC)*, Utrecht, 2023.

[15] C. Roberts and J. Kuchera-Morin, "Gibber: Live coding audio in the browser," in *International Computer Music Conference (ICMC)*, Ljubljana, 2012.

[16] Tidal Cycles Contributors, "Pulsar," https://tidalcycles.org/docs/getting-started/editor/Pulsar/, 2025, accessed: 2025-12-27.

[17] A. Pošćić and G. Kreković, "On the human role in generative art: A case study of ai-driven live coding," *Journal of Science and Technology of the Arts*, vol. 12, no. 3, pp. 45–62, 2020.

[18] G. Kreković and A. Pošćić, "Data-driven generative live coding for music creation," in *2nd Conference on AI Music Creativity (AIMC 2021)*, online, 2021.