AbletonOSC: A unified control API for Ableton Live

Daniel Jones
daniel@jones.org.uk

ABSTRACT

This paper describes AbletonOSC, an Open Sound Control API whose objective is to expose the complete Ableton Live Object Model via OSC. Embedded within Live by harnessing its internal Python scripting interface, AbletonOSC allows external processes to exert real-time control over any element of a Live set, ranging from generating new melodic sequences to modulating deeply-nested synthesis parameters. We describe the motivations and historical precedents behind AbletonOSC, provide an overview of its OSC namespace and the classes of functionality that are exposed by the API, and discuss new types of musical interaction that AbletonOSC enables.

Author Keywords

Open Sound Control, Ableton Live, API, music, control

CCS Concepts

•Applied computing → **Sound and music computing;** **•Networks** → **Presentation protocols;** **•Software and its engineering** → *Software libraries and repositories;*

1. INTRODUCTION

Ableton Live is one of the most popular digital audio workstations (DAWs) in use today, particularly amongst producers, composers and performers of electronic music [14]. Unlike traditional DAWs that are characterised by a horizontal, linear timeline, Live's default state is a vertical clip-based mode of operation, in which each track contains multiple rhythmic or melodic clips. This lends itself well to freeform musical interactions, in which clips are triggered and layered independently.

Users typically interface with Live via GUI-based controls, external hardware devices, or MIDI messages. The advent of the Max For Live extension, based upon Max/MSP, has enabled new forms of interaction with Live that go beyond typical MIDI-based note and control change events. However, programmatical control of arbitrary real-time parameters of Live is still relatively limited, and can only be accomplished by creating piecemeal Max For Live devices to control specific elements of the Live set, or by manually binding MIDI control change events to each parameter of interest.

The objective of this work is to expose a comprehensive API for Live that enables external devices and processes to control arbitrary properties of the Live Object Model (LOM) [4] via Open Sound Control (OSC). First introduced in 2001 [13] and formalised in 2002 [15], OSC has become a de facto standard in musical control interfaces due to its flexibility, minimal complexity of implementation, and low latency and high information rate courtesy of its lightweight UDP-based protocol.

AbletonOSC specifies and implements a hierarchical Open Sound Control namespace that encompasses all of the key classes of entity within Ableton Live, allowing clients to query and modify arbitrary parameters of a set without any additional case-specific setup required ahead of time.

2. DESIGN OBJECTIVES

AbletonOSC was designed with a number of key principles and objectives in mind.

- **Unified**: Although there are existing OSC interfaces for Ableton Live, such as the M4L Connection Kit[1] and Mapper4Live[2], these typically expose limited subsets of the Live API or require a manual assignment per parameter. A primary objective of this work is to ultimately reflect every object property within the Live Object Model, with no prior setup required.
- **Embedded**: Most existing OSC interfaces typically place the responsibility on the DAW user to insert additional Max For Live devices on each track, creating an additional burden and CPU overhead. *AbletonOSC* seeks to act as an invisible interface that is installed once and subsequently persists across future instantiations of Live.
- **Consistent with Live Object Model structure and naming**: Where possible, the namespace design should act as a thin layer between OSC and the existing API of the Live Object Model (LOM), mirroring the existing naming schemes and object relationships. This obviates the need to invent new terminology, and means that users familiar with the Max For Live implementation of the LOM have a minimal additional cognitive load when interacting via *AbletonOSC*.
- **Support for two-way synchronisation**: State changes can originate both from the client-side logic and within

NIME'23, 31 May–2 June, 2023, Mexico City, Mexico.

the Live GUI. To support this fluidity, it is important for *AbletonOSC* to be able to stay in sync with changes that happen on both sides: changes in the GUI must be listened for and reflected within the client, and client-side changes should propagate to the Live GUI.

- **Scalable to complex sets**: For larger-scale projects, the API should be able to support large and complex Live sets, containing hundreds of tracks and thousands of clips.
- **Cross-platform**: Should offer support for both Windows and macOS versions of Ableton Live.

3. IMPLEMENTATION

AbletonOSC is implemented by harnessing Live's Remote Script software interface, an internal API that enables the creators of third-party hardware interfaces to interact with arbitrary components and properties of the LOM. This is achieved by writing scripts using the Python programming language, which are placed in the system's extensions directory and loaded by Live at boot time.

Although this is not a publicly-documented API, there have been extensive third-party efforts to decompile and document the Remote Script functionality[1]. In fact, Remote Scripts effectively provide comprehensive access to the Live Object Model, in just the same manner as Max For Live.

The Remote Script API has existed in Ableton circa Live 6 (2006), steadily evolving to what is today a mature and stable state. In Live 11 (2021), the internals were updated from Python 2 to 3. However, the underlying structures and APIs have remained consistent, typically with incremental changes as new types of device are added. This suggests a reasonable expectation of longevity for projects that rely on Remote Script APIs.

In *AbletonOSC*, Open Sound Control parsing and generation is achieved with the `python-osc`[7] library, a native Python implementation of the OSC 1.0 specification.

4. THE API

AbletonOSC automatically begins listening for OSC messages when Live starts up. This binds to UDP port `11000` by default, and can be configured by changing `OSC_LISTEN_PORT` in `constants.py`.

4.1 Namespace design

All OSC addresses in *AbletonOSC* begin with `/live`, to disambiguate from other reserved prefixes such as `/_cs`[5].

The second path component corresponds to the class of entity being queried/modified, with top-level entities for `song`, `track`, `clip` and other key objects (listed in Table 1).

Each entity has two classes of attribute attached to it:

- **Properties** (4.2), which may be read-only or read-write, and offer `get`, `set`, or `start_listen`/`stop_listen` actions; and
- **Methods** (4.3), called on an entity with zero or more arguments to perform a class-specific activity

[1]For example, by Julien Bayle: `https://structure-void.com/PythonLiveAPI_documentation/Live11.0.xml`

Table 1: Namespace: Top-level entity classes

Address	Description
`/live/application`	Represents the global Live application, generating an event on startup and with getters to query the application state and version.
`/live/song`	The top-level song object, containing global properties (tempo, quantization, metronome state), and a list of `Tracks` and `Scenes`.
`/live/track`	A MIDI or audio track, containing a list of `ClipSlots`, and zero or more `Devices`.
`/live/clip_slot`	A container inside a `Track` that may house a `Clip`
`/live/clip`	Can contain either MIDI notes or an audio sample.
`/live/device`	Represents an instrument or effect, which can be a Live builtin or a VST/Audio Unit. Contains multiple `parameters`, each with a name, value, range and control curve.

4.2 Properties

In the Live Object Model, an object may have one or more properties, which are defined within the LOM, and may be read-only or read-write. For example, a Track has the properties listed in Table 2.

Three main types of operation are defined within *AbletonOSC*:

- `get`: Query the value of a parameter. May return a float, int, boolean, or string, based on the parameter's type. Responds to the OSC client by sending an OSC message with the same address as the `get` query, plus an additional argument containing the parameter's value (see Section 4.2.1).
- `set`: Set the value of a read-write parameter. May accept a float, int, boolean, or string type.
- `listen`: To dynamically track the status of a parameter, a user of the API can call `start_listen` with the parameter's name. This installs a listener for parameter changes, and notifies the OSC client upon any change events by responding to the same address as a `get` query. The listener can later be removed by calling `stop_listen`.

When constructing an OSC message, each of these "verbs" are appended to the item being queried, with entity IDs (and, for setters, the new parameter value) as OSC arguments.

For example, to set the name of track 3 clip 1:

```
/live/clip/set/name 3 1 foo
```

4.2.1 *Responses*

The OSC 1.0 specification is a one-way protocol with connectionless messaging, and does not explicitly specify an approach for handling replies to messages. *AbletonOSC* adopts a convention of always sending OSC responses to the same IP address as the query's originating host, to a default

Table 2: Properties of a Track

read-only	read-write
can_be_armed	arm
fired_slot_index	color
has_audio_input	color_index
has_audio_output	current_monitoring_state
has_midi_input	fold_state
has_midi_output	mute
is_foldable	solo
is_grouped	name
is_visible	
playing_slot_index	

Table 3: Methods of a Track

method	parameters
delete_device	device_index (int)
stop_all_clips	

UDP port of `11001` (configurable with `OSC_RESPONSE_PORT` in `constants.py`).

Responses include the full path and identifier of the entity, with a verb of `get`.

4.2.2 *Example flow*

```
-> /live/song/get/tempo
<- /live/song/get/tempo 120.0
-> /live/song/set/tempo 110.0
-> /live/song/get/tempo
<- /live/song/get/tempo 110.0
```

4.2.3 *Wildcards*

AbletonOSC supports the usage of Open Sound Control wildcard patterns, which allow the API consumer to specify multiple entities or properties by substituting an asterisk `*` symbol where a name would normally occur. For example, to retrieve every property of clip 4 on track 3, an API user can query `/live/clip/get/* 3 4`. The API will respond with individual `/live/clip/get` responses for each property:

```
-> /live/clip/get/* 3 4
<- /live/clip/get/name 3 4 "My Clip"
<- /live/clip/get/length 3 4 8.0
<- /live/clip/get/is_midi_clip 3 4 True
<- /live/clip/get/notes 3 4 60 0.0 1.0 100 False
...
```

Note that the track and clip ID are included in each response. This is critical in a connectionless protocol such as OSC in which the sequence of delivery is not guaranteed, as it allows the client to infer the target clip of each response, even if the responses arrive out-of-order.

Wildcards can also be used to begin listening for multiple properties. To listen for changes to every property of track 3:

```
/live/track/start_listen/* 3
```

4.3 Methods

Methods are class-specific actions that are performed on a given object. For example, the `Song` object has multiple methods associated with playback:

```
start_playing
stop_playing
continue_playing
jump_by [beats (int)]
jump_to_prev_cue
jump_to_next_cue
```

Calling a method is typically a one-shot action, and does not generate a response, except in the case of errors.

4.4 Error handling

Errors are reported with an OSC response of `/live/error`, with a single string parameter containing the error summary.

Additionally, errors are logged to `logs/abletonosc.log`, within the top-level `AbletonOSC` directory. In the case of internal errors, this is accompanied by a stack trace for more detailed investigation.

5. USAGE PATTERNS

5.1 Dynamic state synchronisation

For many applications, it is vital to be able to dynamically reflect Ableton's complete state space in the client-side representation. For example, a touch-screen display to control track levels and sends will initially need to send a query to obtain the number of tracks available, their names, and each track's current gain and send levels. If a track gain is modified within the Live GUI, the touch-screen should update to reflect this change.

This dynamic state synchronization is typically achieved in a two-step process: (i) Request a snapshot of the initial state; then (ii) Create listeners to track future state changes.

5.1.1 *Request a snapshot of the initial state*

To initialise the client's state, a batch query must be made to return the full set of parameters of interest. This can be accomplished using a large number of parameter queries, or by making queries for wildcard patterns. However, both of these methods introduce an additional messaging overhead, as one reply is sent per parameter per entity, which can result in hundreds or thousands of OSC reply messages. For larger sets, this can be prohibitively slow and inefficient.

AbletonOSC provides a more efficient way to obtain a state snapshot in the form of the `/live/song/get/track_data` API endpoint. This endpoint lets the client specify the specific subset of properties that it requires, across a specified range of tracks.

For example, specifying the parameters `0 12 track.name clip.name clip.length` will return a single OSC message with arguments corresponding to the respective track names, clip names, and clip lengths for every clip within tracks 0 to 11, in the below format:

```
[track_0_name, clip_0_0_name, clip_0_1_name, ...,
               clip_0_0_length, clip_0_1_length, ...,
 track_1_name, clip_1_0_name, clip_1_1_name, ...,
               clip_1_0_length, clip_1_1_length, ...,
               ...
               clip_11_0_length, clip_11_1_length...]
```

This is a substantially more compact representation than one query per parameter, and allows the client to receive a snapshot of large quantities of set data in a single query.

5.1.2 Create listeners to track future state changes

After the client has mirrored the initial set state, it can stay in sync by registering listeners to be notified of any future state. This can be achieved using wildcards; for example:

```
/live/track/start_listen/* *
/live/clip/start_listen/* * *
```

5.2 Following the beat

It is often useful for external processes to be able to carry out operations in synchrony with the development of the musical structure; for example, to re-generate the contents of a clip's melody lines once every 4 bars.

To address this, *AbletonOSC* features a beat listener, which sends a trigger message to the client once per beat, including the current beat number:

```
/live/song/get/beat 0
/live/song/get/beat 1
/live/song/get/beat 2
...
```

Note that, due to the limited time resolution of Ableton's internal message handling, this may be subject to latency of up to 100ms, and so should not be relied on for low-latency time-critical events.

6. EXAMPLES OF NEW INTERACTIONS

Below are a few examples of classes of interaction with Ableton Live that are enabled by *AbletonOSC*.

6.1 Creation of flexible GUIs

A popular emerging application area for *AbletonOSC* is in the creation of custom GUIs that allow a musician to shape their interface to their needs, using third-party applications such as TouchOSC [9] and Open Stage Control [6]. These applications allow users to visually lay out controls such as triggers, toggles and faders, which can then send real-time parameter updates to Live via *AbletonOSC*.

By doing so, a musician can foreground the specific set of controls that are relevant to their live interactions, eliminating superfluous visual complexity to streamline the interaction experience, enabling context-dependent UI elements, and taking advantage of the unique affordances of multi-touch surfaces [12].

6.2 Exploring device parameter space

Via the `/live/device/get/parameters` API, it is possible to query the entire set of names, ranges and control curves of every parameter supported by an Ableton Live Instrument or Audio Effect. For example, all parameters of a device can be randomised by calling `/live/device/set/parameters/value` with a list of values that fall within the `min` and `max` range, as obtained by querying `/live/device/get/parameters/{min,max}`.

This provides the basis for exploring musical parameter space using techniques such as neural networks and genetic algorithms [17].

Note that, for external VSTs and Audio Units, it is necessary to expose parameters to Live using the "Configure" dialogue before they can be accessed via *AbletonOSC*. This is an inherent limitation of the Live Object Model as of the date of writing, and applies in the same way to Max For Live devices.

6.3 Generating melodic sequences

By exposing `/live/clip/{get,add}/notes`, *AbletonOSC* allows clients to query and create the MIDI note events that are sequenced within clips. This is a powerful strategy for adding generative elements to a set, by applying algorithmic compositional processes to populate clips' contents.

This also opens the doors to third-party client applications that may want to generate and manipulate clip contents on-the-fly; for example, it would be possible to create a client for live coding practice[3] that uses Ableton Live as its engine.

6.4 Algorithmic music and sonification

Programmatic control over Ableton Live means that it can more broadly be used as a platform for algorithmically-generated music and data-driven compositional practices such as sonification [8], in which generative patterns or external data sources are used to create and reshape musical structures on the fly.

An applied example is *Living Symphonies* [11], a sound installation that models the dynamics of a forest ecosystem. *AbletonOSC* is used to trigger, filter and arrange thousands of clips across hundreds of tracks, in response to an ecological simulation and sensor array capturing moment-to-moment changes in weather conditions. In this context, *AbletonOSC* bridges between studio-grade compositional practices and generative systems, allowing composed fragments to be arranged and sequenced dynamically, producing an output that sounds orchestrated yet is fundamentally nonlinear.

An alternative approach with a heavier weighting towards pre-composed sequences is nonlinear navigation of the horizontal Arrangement View. *AbletonOSC* exposes access to a song's cue points, with the ability to query and modify the positions of cues and clips within the Arrangement View, which allows control over sequential compositions: cue points can be triggered, created, skipped and looped, allowing for the introduction of nonlinearity into traditionally-composed compositions, installations or performances.

6.5 Client libraries

Finally, the *AbletonOSC* API invites the development of external client libraries that can activate and control elements of Live from standalone client applications. An example is pylive [10], a Python library that encapsulates key entities (Song, Track, Clip, etc) in Python classes, providing an abstraction layer that eliminates the need for the user to craft OSC messages. The user can thereby interact with Live using a handful of lines of code, without needing to care about the underlying communication interface.

7. PERFORMANCE AND LIMITATIONS

7.1 Time resolution

The time resolution of *AbletonOSC* is bounded by the internals of Ableton Live's message handling, which operates on a 100ms tick. This means that processing of queries may take as long as 100ms, meaning that low-latency interactions are not possible.

More generally, the Live Object Model interface does not expose any mechanism for triggering note-down events or other types of event trigger. For these reasons, clients should still use MIDI for relaying note triggers and other time-sensitive events to Live.

7.2 Maximum event rate

The performance of *AbletonOSC* was evaluated by sending clip trigger and parameter control messages to the system at increasing rates. On an Apple MacBook Pro M1 (2021), the system was able to process over 100 queries/second without noticeable latency. Beyond 200 queries/second, audible latency began to occur in clip trigger events. This is likely due to the processing time overshooting the 100ms available per tick.

As scheduling is constrained by the time resolution described above, with no support for multi-threading in Live's internal Python implementation, it will be challenging to improve significantly on this limitation.

However, 100 queries/sec is likely to be an order of magnitude more than required in most real-time scenarios, and MIDI messaging can be used simultaneously for high-density control applications.

8. FUTURE WORK

This initial release of *AbletonOSC* exposes support for all of the key primitives of Ableton Live (Song, Track, Clip, Clip Slot, and Device). There are many other secondary structures documented within the LOM API [4] that are still awaiting integration. These include Browser, Groove, DrumPad, Chain, RackDevice, DeviceIO, and other specific device subclasses (CompressorDevice, Eq8Device, SimplerDevice, WavetableDevice, etc).

The library does not currently include support for OSC bundles or time tags, both part of the OSC 1.0 specification [15]. These would help to reduce network communication overhead, and improve event synchronisation in the face of severe latency and jitter.

Surveying the future landscape of computer music, Wright et al [16] propose a vision in which the complete state of a complex musical software system can be serialised and recreated entirely using OSC messages. Although further work needs to be done to address the library's current limitations, attempting to realise this vision would be a useful litmus test to evaluate the scope and maturity of *AbletonOSC*.

9. ACKNOWLEDGEMENTS

The author would like to thank all of the contributors who have provided valuable code and feedback to the initial release of *AbletonOSC*; in particular, Jörn Lengwenings, Mark Marijnissen, and Marco Buongiorno Nardelli. Thanks also to Stu Fisher (and other authors) for *LiveOSC*, the predecessor to this work, and Julien Bayle, NSUSpray, and Hanz Petrov for their work in decompiling and documenting the LOM API.

10. MORE INFORMATION

Complete API documentation and download of *AbletonOSC* are available at:

`https://github.com/ideoforms/abletonosc`

11. REFERENCES

[1] Ableton. Ableton Connection Kit. https://www.ableton.com/en/packs/connection-kit/, 2016.

[2] B. Boettcher, J. Malloch, J. Wang, and M. M. Wanderley. Mapper4Live: Using Control Structures to Embed Complex Mapping Tools into Ableton Live. In *NIME 2022*, 2022.

[3] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(3):321–330, Dec. 2003.

[4] Cycling '74. LOM: The Live Object Model (Max 8 Documentation). https://docs.cycling74.com/max8/vignettes/live_object_model, 2023.

[5] R. B. Dannenberg and Z. Chi. O2: Rethinking Open Sound Control. In *Proceedings of the International Computer Music Conference*, 2016.

[6] J.-E. Doucet. Open Stage Control. http://openstagecontrol.ammd.net/, 2014.

[7] T. Faudot. python-osc. https://pypi.org/project/python-osc/, 2013.

[8] T. Hermann, A. Hunt, and J. G. Neuhoff. *The sonification handbook*, volume 1. Logos Verlag Berlin, 2011.

[9] Hexler. TouchOSC. https://hexler.net/touchosc, 2008.

[10] D. Jones. pylive. https://github.com/ideoforms/pylive, 2012.

[11] D. Jones and J. Bulley. Living Symphonies. https://www.livingsymphonies.com/, 2014.

[12] P. McGlynn, V. Lazzarini, G. Delap, and X. Chen. Recontextualizing the Multi-touch Surface. In *Proceedings of NIME 2012*, 2012.

[13] D. Wessel and M. Wright. Problems and prospects for intimate musical control of computers. In *Proceedings of the CHI'01 Workshop on New Interfaces for Musical Expression*, 2001.

[14] R. Wreglesworth. Which DAW Do Most Producers Use? https://musicianshq.com/most-popular-daw-software-which-daw-do-most-producers-use/, 2023.

[15] M. Wright. Open Sound Control Spec 1.0. https://opensoundcontrol.stanford.edu/spec-1_0.html, 2002.

[16] M. Wright. OpenSound Control: State of the Art 2003. In *Proceedings of NIME 2003*, volume 3. Springer International Publishing, 2003.

[17] M. J. Yee-King, L. Fedden, and M. d'Inverno. Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):150–159, Apr. 2018.