LV2 Atoms: A Data Model for Real-Time Audio Plugins

David E. Robillard
School of Computer Science, Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6
Canada
d@drobilla.net

Abstract

This paper introduces the LV2 *Atom* extension, a simple yet powerful data model designed for advanced control of audio plugins or other real-time applications. At the most basic level, an atom is a standard header followed by a sequence of bytes. A standard type model can be used for representing structured data which is meaningful across projects. Atoms are currently used by several projects for various applications including state persistence, time synchronisation, and network-transparent plugin control. Atoms are intended to form the basis of future standard protocols to increase the power of host:plugin, plugin:plugin, and UI:plugin interfaces.

Keywords

LV2, plugin, data, state, protocol

1 Introduction

LV2 is a portable plugin standard for audio systems, similar in scope to LADSPA, VST, AU, and others. It defines a C API for code and a format for data files which collectively describe a plugin. The *core* LV2 API is similar in power to LADSPA, but *extensions* can support more advanced functionality. This allows the interface to evolve and accommodate the needs of real software as they arise.

LV2 is supported by many applications, including Digital Audio Workstations like Ardour [Davis and others, 2014], hardware effects processors like MOD [Ceccolini and Germani, 2013], and signal processing languages like Faust [Gräf, 2013].

One key piece of functionality LV2 adds to LADSPA is the ability to transmit events. This is most commonly used to communicate via MIDI [MID, 1983] for playing notes, selecting programs, etc. MIDI is nearly ubiquitous in musical equipment, but has significant limitations [Moore, 1988]. Many applications require a more powerful model to both express and manipulate state. To take a simple yet common example, MIDI has no standard way to express "load sample `/media/bonk.wav`". Other protocols like OSC [Wright, 1997] have attempted to address the limitations of MIDI, but are still designed around flat *commands*, which limits their ability to express structured data.

This paper introduces the LV2 *Atom* extension, a simple yet powerful data model designed for advanced control of LV2 plugins or other real-time applications. Atoms serve both as a model for representing state, and a protocol for accessing or manipulating it. This includes simple values such as numeric controllers or sample file names, but the model-based approach allows developers to work with more sophisticated data as well.

The key distinction between atoms and MIDI or OSC messages is that atoms are not just commands, but a general data format. This paper aims to show that building on the foundation of a solid data model is more elegant and powerful than command-based protocols like MIDI or OSC. The idea is conceptually similar to the popular use of JSON [Crawford, 2006] in the web community: define a simple data model for representing information, then construct messages *within* that data model.

However, atoms are not introduced to the exclusion of other protocols. In fact, MIDI messages are transmitted to and from plugins as a particular type of atom. At the lowest level, atoms are simply a binary blob with a standard header. On top of this, a type model is defined which allows complex structures to be built from a few standard primitive and container types. This model has several advantages, including extensibility, support for round-trip portable serialisation, and natural expression in plugin data files.

There are two aspects to the LV2 atom specification: the low-level mechanics (Section 2) define the binary format of atoms and how they may be used, while the high-level semantics (Section 3) define a type model built upon this simple binary format. Using this model, projects can communicate meaningful structures at a conceptually high level, while the actual mechanics involved are simply the copying of small binary blobs. This approach to plugin control has many applications (Section 4) in current projects, which typically use the provided convenience APIs for reading and writing atoms (Section 5) with ease. Ultimately, atoms are intended to form the basis of future work (Section 6) designing standard protocols for advanced plugin control.

2 Mechanics

2.1 Atom Definition

An LV2 atom is a 64-bit aligned chunk of memory that begins with a 32-bit size and type:

```
typedef struct {
        uint32_t size;
        uint32_t type;
} LV2_Atom;
```

This *atom header* is immediately followed by the *body* which is `size` bytes long. Atoms are, by definition, Plain Old Data (POD): contiguous chunks of memory that can safely be copied bytewise.[1] At the most basic level, this is all there is to atoms.

Types are assigned dynamically and not restricted to any fixed set. Developers can define new atom types, though all types are required to be POD. Any atom can thus be copied or stored, even by an implementation which does not understand its type. Among other advantages, this makes it possible for hosts to transmit atoms between plugins without explicitly supporting each type used. Developers are free to send complex data between plugins, or between UIs and plugins, without being held back by lacking standards or host support. Section 3.3 explains in detail how this decentralised extensibility is achieved.

Note, however, that atoms are only POD by definition, not necessarily portable: atoms may contain architecture-specific data like integers with native endianness. The atom specification includes a set of standard types which should be used when persistence or interoperability are important (see Section 3.1).

2.2 Communication via Ports

Plugins can send or receive atoms via an `AtomPort` which (like any LV2 port) is either an input or an output. An `AtomPort` is connected directly to an `LV2_Atom` (just as a standard LADSPA or LV2 control port is connected directly to a `float`).

An `AtomPort` can be used with any atom type. Plugins specify which types are supported using the `atom:bufferType` property in their data files. Several types may be supported by a single port.

[1] Type 0 has been reserved for a special reference type, in case a need for non-POD communication arises in the future.

Input logistics are straightforward: the host connects the input to an atom before calling the plugin's `process()` method.

Outputs are slightly trickier since the plugin must know how much space is available for writing, but atom types may have variable size. To resolve this, the host initialises the `size` field in the output buffer to the amount of available space before calling `process()`. Plugins read this value, then write a complete atom (including `size` and `type`) to the buffer before returning.

Thus far, atoms have been described without referring to specific types. An `AtomPort` can be connected to any value, but since plugins process signals over a block of time, it is usually more useful for ports to contain many time-stamped atoms, or *events*. To achieve this, ports are connected to a `Sequence` atom. This is the mechanism commonly used by LV2 plugins to process streams of sample-accurate MIDI messages alongside audio. The following section describes the set of standard atom types, which includes simple types like `Int` and containers like `Sequence`.

3 Semantics

3.1 Atom Types

The structure of the atom type model is similar to JSON values or ERLANG terms [Virding et al., 1996]: a few primitive types, and collections which can be used to build larger structures. The hierarchy of standard types defined in the atom extension[2] is shown in Figure 1.

Primitives represent a single value, and do not contain other atoms. The simplest types are primitives with fixed size, like `Int`. These types have a corresponding C struct defined in `atom.h` which completely describes their binary format, for example:

```
typedef struct {
    LV2_Atom atom;
    int32_t  body;
} LV2_Atom_Int;
```

A `URID` is a URI which has been mapped to a 32-bit integer by the host. This facility allows URIs to be used conceptually, but with the performance of fixed-size integers (Section 3.3 explains the purpose of URIDs in more detail).

Other primitive types have variable size. `String` and `Literal` represent raw strings and string literals with a datatype or language, respectively. A `Chunk` contains an opaque binary blob of data.

[2] There is also a standard type for MIDI messages defined in the separate LV2 MIDI extension.

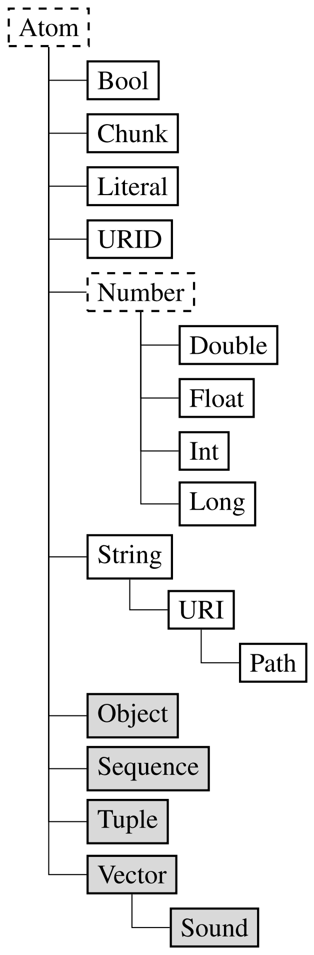

Figure 1: The Atom type hierarchy. Abstract types are dashed, and collections are grey.

Larger structures can be built from these primitives using collections. The simplest collection type is `Tuple`, a series of atoms of any type. An `Object` is a set of *properties*, each with a URID *key* and a *value* of any type. `Tuple` and `Object` are analogous to arrays and objects in JSON, or tuples and dictionaries in Python [van Rossum, 2010], respectively: universal containers that can express almost any structured data.

The remaining collection types are essentially optimisations for audio applications. A `Sequence` is, like `Tuple`, a series of atoms, but each is preceded with a time stamp.[3] A `Vector` is a series of fixed-length atoms with the same type and no headers, making the vector body an unadorned C array. `Sound` is a descriptive type, identical in format to a `Vector` of `float`, but explicitly representing a sample of audio.

[3] Currently time stamps are always in samples, though other units are possible.

3.2 Portable Serialisation

In addition to a binary format, each atom type has a portable serialisation. This allows implementations (typically hosts) to convert atoms to and from text for portable storage, network transmission, or human readability. This format is used to describe atoms in the following sections, but it is important to keep in mind that plugins work with atoms in their native binary form.

Most primitive types are associated with XSD [W3C, 2004b] datatypes which define their textual format. Table 1 shows this mapping along with an example string. URID is omitted since the portable serialisation of a URID is a URI.

Atom	XSD	Example
Bool	boolean	true
Chunk	base64Binary	vu/erQ==
Double	double	2.99e8
Float	float	0.6180
Int	int	-42
Long	long	4294967296
String	string	hello
URI	anyURI	http://lv2plug.in/
Path	anyURI	/home/drobilla/

Table 1: Text serialisation for primitives.

Containers and `Literal` have an abstract RDF [W3C, 2004a] serialisation which can technically be written in many formats. Here, the syntax of choice is Turtle [Beckett and Berners-Lee, 2011], which is used in LV2 data files.

All containers have a portable serialisation, but this paper focuses on the use of `Object`. The format for the other containers is omitted for brevity, but can be found in the LV2 atom specification.

An object in Turtle begins with its ID, followed by properties separated with semicolons. A "." terminates the description. For example, an `Object` named `eg:control` with three properties can be written as:

```
eg:control
    lv2:minimum 0.0 ;
    lv2:maximum 1.0 ;
    lv2:default 0.5 .
```

The ID and properties shown here are abbreviated URIs, for example, `lv2:minimum` is actually the URI `http://lv2plug.in/ns/lv2core#minimum`. A full Turtle document has prefix directives to define these precisely.

Numbers are shown unquoted, which is valid but does not precisely map to Atom types (e.g. 1.0 could be a `Float` or a `Double`). To preserve type in a machine serialisation, explicitly typed literals like "1.0"^^xsd:float are used instead.

3.2.1 Serialisation in Practice

A text-based format for describing atoms facilitates discussion, but is also useful in practice. The Sratom [Robillard, 2012b] library provides a simple C API for lossless round-trip serialisation of any atom built from the standard types. This is used in several different scenarios:

- Saving plugin state in sessions, which is supported by many hosts.
- Jalv [Robillard, 2012a], a single-plugin host for Jack [Davis, 2001], can log all communication between plugin and UI to the console. This is particularly useful for debugging.
- Ingen [Robillard, 2014], a modular plugin host and plugin itself, has a UI that communicates to the engine exclusively via atoms. When running as a plugin, binary atoms are sent via AtomPort, but the UI can also run remotely by communicating over a TCP socket in Turtle. This way, UIs on a different architecture can control the engine, including those written in non-C languages like Python or Javascript.

3.3 URIs and Extensibility

Types and properties are identified by URI. The benefit of URIs is that anyone can define new terms without needing to worry about clashes or centralised coordination.

In the context of LV2 atoms, this allows developers to invent new types and properties without requiring "approval". This freedom is particularly useful while developing new ideas, be they experimental, for internal use only, or intended for eventual standardisation.

For example, the previous sections use the lv2:minimum property, but suppose a plugin developer additionally needs to describe a "sweet spot" for controls. There is no standard LV2 property for this concept, so the developer can define their own (e.g. http://drobilla.net/ns/sweetSpot), use it in their data files, implement host support if necessary, send it between plugins or between plugin and UI, and so on. The implementation can be tested and released to the public without any binary compatibility issues. This flexibility allows LV2 to evolve to meet real developer needs with minimal friction.

Note that URIs here are simply serving as global identifiers, and are not required to actually resolve on the Internet. However, developers should use URIs in domains where they *could* host pages, since this avoids potential conflicts.[4] There is no need to own an entire domain, for example many plugins use URIs at popular project hosting sites.

URI schemes other than HTTP may be used, but are not recommended. One advantage of HTTP is the ability to have URIs resolve to useful resources, particularly documentation. All standard LV2 URIs work this way, so documentation is often just a click away (follow the above lv2:minimum URI for an example). The LV2 distribution includes a tool, lv2specgen, which generates documentation for types and properties which are defined in Turtle.

4 Applications

4.1 Time

The most common use of objects to communicate between plugins and hosts is transport synchronisation. To keep plugins updated with tempo information, hosts send an object with properties describing the current time and tempo, whenever changes occur.

Most hosts send updates that roughly correspond to Jack transport information, but with floating point beats instead of PPQN ticks, and a single floating point speed instead of only "rolling" or "stopped". For example:

```
[]
    a                    time:Position ;
    time:frame           88200 ;
    time:speed           0.0 ;
    time:bar             1 ;
    time:barBeat         0.0 ;
    time:beatUnit        4 ;
    time:beatsPerBar     4.0 ;
    time:beatsPerMinute  120.0 .
```

4.2 UI Communication

Atoms are also useful for communicating with components other than the host. The most common of these in practice is communication between a plugin and a custom UI (which, in LV2, happens via ports). Many UIs need to perform more advanced operations than is possible via float control ports. For example, a plugin may include an envelope with an arbitrary number of points, which a UI could control with messages like

```
[]
    a          eg:EnvelopeSegment ;
    eg:endX    1.6 ;
    eg:endY    0.5 ;
    eg:shape   eg:linear .
```

[4] Inventing URIs under other domains without permission is inappropriate!

Several projects have made use of such messages for sophisticated plugin control from UIs. While host-transparent (and thus automatable) control is preferable, full control of some plugins requires messages that are not currently standardised (e.g. LV2 presently has no concept of envelope segments, or multi-dimensional controls in general). However, though the message does not have standardised semantics, it is built from standard atom types so that hosts can make some sense of it. In particular, hosts can serialise such messages for controlling a plugin running on a remote computer or embedded device. This is a good example of how an extensible model allows developers to achieve their goals without being held back by lagging standardisation or host support.

This method of plugin control is relatively new, and remains an experimental area. In the future, standardised message types will allow plugins and UIs to communicate with each other in sophisticated ways, and hosts to support friendly interfaces and automation for message-based plugin control.

4.3 Plugin State

LV2 has a *state* extension which allows plugins to save and restore state beyond simple control port values. The state extension does not directly depend on the atom extension, but has a property-based API that meshes naturally with `Object`. Plugins use host-provided callbacks to save/restore a URID key, `void*` value, and URID type.

The fact that plugin state and `Object` are both based on properties suggests an elegant approach to plugin design: a set of properties can serve both as plugin state and real-time control protocol. This means plugin developers do not need to design both a state model and protocol, but simply define a set of properties that describes their plugin's state.

For example, the sampler example plugin included with LV2 can play any `.wav` file, and the sample can be loaded by sending a message like:

```
[]
    a              patch:Set ;
    patch:property eg:sample ;
    patch:value    </media/bonk.wav> .
```

The `patch:Set` type and properties used here are defined in the LV2 *patch* extension, which defines several message types for getting and setting property values.

4.4 Properties

Developers can invent new property URIs and use them in code without defining anything. However, it can be useful to define properties for documentation purposes, and in some cases host support.

Properties are defined in Turtle, so they can be included alongside plugin descriptions. For example:

```
eg:sweetSpot
    a            rdf:Property ;
    rdfs:domain  lv2:ControlPort ;
    rdfs:range   xsd:float ;
    rdfs:label   "sweet spot" ;
    rdfs:comment "The nicest value." .
```

Defining properties in this machine-readable format is mainly useful for generating documentation (all standard LV2 properties are defined in this way), but this information can be used by hosts as well.

This is still an area of exploration, but for example, Jalv will show a file selector in its host-generated UI for plugins that support properties with `Path` values. Setting the property is achieved by sending a `patch:Set` message like the example shown in the previous section.

4.5 Presets and Default State

Plugin descriptions can include a set of default state properties which should be loaded initially. A preset has a similar structure to a plugin description, and can also include state. This means that presets can not only set port values, but restore arbitrary internal plugin state like loaded samples. The benefit of using standard atom types to describe state is that developers can write default state in plugin data files, and hosts can serialise state/presets in the same format. For example, a preset for a sampler can specify a sample to load like so:

```
eg:somePreset
    lv2:appliesTo eg:sampler ;
    # ...
    state:state [
        eg:sample <click.wav>
    ] .
```

5 Reading and Writing Atoms

It's convenient to think of atoms in high level terms, and describe objects in human-readable Turtle, but plugins are typically written in C and must work with binary atoms. For simple primitives types like `Int` this is trivial: the appropriate structs can be created, copied, and read in the usual way.

Collections are more complex, since their bodies have variable size and possibly an irregular and/or nested structure. To make reading collections simpler, iterators for each collection type are provided in a utility header.

For objects, iteration works, but the typical case of getting a few property values is cumbersome and verbose to implement using iteration. For this reason, a simple accessor for object properties is provided. For example, the following simple object describes a 2D point:

```
[]
    a    eg:Point ;
    eg:x 1.0 ;
    eg:y 2.0 .
```

If `obj` points to this object, and `ids.eg_x` and `ids.eg_y` are mapped to the appropriate URIs, the `eg:x` and `eg:y` properties can be accessed like so:

```
const LV2_Atom* x = NULL;
const LV2_Atom* y = NULL;
lv2_atom_object_get(obj,
                    ids.eg_x, &x,
                    ids.eg_y, &y,
                    0);
```

Here, `x` and `y` point to the corresponding values within `obj`. There is no dynamic allocation, so this code is real-time safe and does not require the user to clean up `min` and `max`. If the object does not have a matching property, the result will be `NULL`. Note that this code will continue to work correctly even if additional properties are added to the point object in the future.

Writing collections can be trickier, since they may have irregular or nested structure. For example, an `Object` property may have a `Tuple` or another `Object` as a value. Atoms can be constructed in-place by appending to a buffer, but correctly maintaining container `size` fields and padding requirements can be a delicate task. To make this simple, a *forge* API is provided which allows arbitrarily complex atoms to be constructed in a target buffer. The forge has a method for each atom type: for primitives it simply appends the given value, and for containers it appends the atom header and returns a *frame* which must be popped when the object is finished. Container sizes are updated automatically as atoms are written using this stack of frames. The forge is safe to use in real-time code, and can be used by plugins to write objects directly to `AtomPort` outputs in their `process()` method. For example, the same 2D point object can be written like so:

```
// Begin an eg:Point object (w/ no URI)
LV2_Atom_Forge_Frame frame;
lv2_atom_forge_object(
    forge, &frame, 0, ids.eg_Point);

// eg:x 1.0
lv2_atom_forge_key(forge, ids.eg_x);
lv2_atom_forge_float(forge, 1.0);

// eg:y 2.0
lv2_atom_forge_key(forge, ids.eg_y);
lv2_atom_forge_float(forge, 2.0);

// Finish object
lv2_atom_forge_pop(forge, &frame);
```

6 Conclusions and Future Work

The LV2 Atom specification defines a simple binary format for any type of data, and an expressive type model for representing structured data within that format. This model has proven effective for representing plugin state, host to plugin communication such as tempo synchronisation, and custom control protocols such as between a plugin and its UI.

This work has layed the foundation for more powerful control of plugins and other real-time applications. There are two main areas of future work: additional convenience APIs and tools to make working with atoms as simple as possible, and building more advanced control protocols and other functionality using the atom model.

For convenience, the existing APIs described in Section 5 do a relatively good job of making atom construction and destruction simple in C. However, some developers have found the forge confusing. It is difficult to make a fully capable writing API much simpler given the constraints of C and hard real-time, but one idea is to make a writing counterpart to `lv2_atom_object_get()` which works only for non-nested objects. Using C++, a similar, but more elegant and type-safe interface would be possible, which could work even for nested containers. LV2 is defined in C, but a significant portion of the developer community uses C++, so a C++ convenience wrapper (including idiomatic iterators) would be a welcome improvement. Other minor improvements could ease the mechanics, but since several developers have successfully made use of atoms, focusing on this area may not be an effective use of time.

The other, more interesting, area for future work is building on the foundation of atoms to create more powerful control protocols. One of the biggest limitations of LV2 is the `ControlPort` inherited from LADSPA. Control ports can only hold a single float value, and tie the control rate to how often `process()` is called. This can be problematic for certain types of plugins. The lack of a mechanism for adding and removing ports also means that the set of controls is fixed, which prevents many possibilities such as the multi-point envelope example in Section 4.2. Using events for control instead of control ports can solve all of these problems. Events are much more powerful than a low-rate control signal, and allow a sample-accurate stream of changes to be sent to a plugin for an entire `process()` call. This will be achieved via the current `AtomPort` + `Sequence` mechanism, but the structure of events required is yet to be determined. `Object` can certainly suffice, but the benefits of extensibility need to be weighed against the slight overhead for high-rate controls. There are many possibilities opened up by moving to events, including ramped/smoothed controls, gestures, precise voice control, and note-specific modulation/articulation. This is one of the most exciting frontiers of LV2 development; a powerful event-based control scheme will open up new possibilities beyond what is currently possible with host-agnostic plugins.

References

David Beckett and Tim Berners-Lee. 2011. Turtle - Terse RDF Triple Language. `http://www.w3.org/TeamSubmission/turtle/`. W3C Team Submission.

Gianfranco Ceccolini and Leonardo Germani. 2013. MOD - an LV2 host and processor at your feet. In *Linux Audio Conference 2013 Proceedings*, LAC 2013, pages 157–161. Institute of Electronic Music and Acoustics (IEM).

Douglas Crawford. 2006. The application/json media type for JavaScript Object Notation (JSON). `http://www.ietf.org/rfc/rfc4627`. RFC 4627.

Paul Davis et al. 2014. Ardour Digital Audio Workstation. `http://ardour.org/`.

Paul Davis. 2001. JACK Audio Connection Kit. `http://jackaudio.org/`.

Albert Gräf. 2013. Creating LV2 plugins with Faust. In *Linux Audio Conference 2013 Proceedings*, LAC 2013, pages 145–152. Institute of Electronic Music and Acoustics (IEM).

1983. *Musical Instrument Digital Interface Specification 1.0*. International MIDI Association. `http://www.midi.org/techspecs/`.

F Richard Moore. 1988. The dysfunctions of MIDI. *Computer Music Journal*, 12(1):19–28.

David E. Robillard. 2012a. Jalv. `http://drobilla.net/software/jalv/`.

David E. Robillard. 2012b. Sratom. `http://drobilla.net/software/sratom/`.

David E. Robillard. 2014. Ingen.

Guido van Rossum. 2010. *The Python Language Reference*. Python Software Foundation. `http://docs.python.org/reference/`.

Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG*. Prentice Hall, second edition. `http://www.erlang.org/`.

W3C. 2004a. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`. W3C Recommendation.

W3C. 2004b. XML Schema Part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2/`. W3C Recommendation.

Matthew Wright. 1997. Open Sound Control - a new protocol for communicatiang with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, pages 101–104.