NWB Enhancement Proposal
NWBEP001: Events

Version: 0.4.0
Last modified: November 24, 2024

NWBEP moderators/leads: Ryan Ly <rly@lbl.gov>, Oliver Rübel <oruebel@lbl.gov>
Point of Contact: Ryan Ly <rly@lbl.gov>
Contributors: NWB Technical Advisory Board, NWBEP001 Review Working Group (Josh Siegle, Kay Robbins, Chris Rodgers, Alessandra Trapani)

Summary: Event data are extremely common in neurophysiology experiments; however, there are no dedicated neurodata types for storing event data in NWB. Currently, data curators use overly generic or overly complex neurodata types to store event data and can choose from multiple approaches. Both data curators and data users would benefit from a standardized, unified, and simple approach for storing event data in NWB.

1. Table of Contents

2. Definitions

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119.

Neurodata type names are written in the font `Courier New`.

3. Rationale

3.1. Background

What application problem and use cases does the enhancement address? What is the target audience for this enhancement? How would the broader neurophysiology community benefit from the enhancement?

Most neurophysiology experiments require the storage of event markers, e.g., to mark the time of a task event, such as the onset of a visual stimulus or an instantaneous lick, nose-poke, or reward delivery. Events can also be used to annotate the times of unusual or interesting activity, such as the times of a specific behavior detected from a video or the time of a hippocampal ripple. These event markers are often used to align neural activity to particular task events or behaviors of interest. They can also be used to synchronize data acquisition streams, trigger an action or state change in the experimental control computer, and more.

Neurophysiology experiment control software and hardware commonly signal events using TTL (Transistor-Transistor Logic) pulses. TTL pulses can be sent to different channels of a data acquisition system to indicate different event types. They are often differentiated by the ID of the channel, or combination of channels, that a pulse was received on, sometimes referred to as the pulse value. For example, a TTL value of 1 could signal the start of a trial, a TTL value of 2 could signal the start of a stimulus presentation, etc. See Appendix section **8.1. TTL Pulses** for more information about TTL pulses.

The target audience is all experimenters, data analysts, and developers of data acquisition systems who want to store event markers.

3.2. Current implementation

How is the data currently being stored? What are the advantages and disadvantages of the current implementation? Are there existing standards or conventions relevant to the data?

As far as we know, as of August 2024, based on data published on DANDI and experience working with users directly, most NWB users store event data using one of the four approaches described below. Most public datasets that we could identify as storing event data used one of these approaches to store lick times, reward times, times of task events, times of subject actions, times of specific behaviors, or times of specific neural activity.

This proposal proposes creating a new approach to store event data that is similar to the ndx-events 0.2.0 `AnnotatedEvents` type described below and deprecating the other approaches.

Previous approaches for storing event data are primarily:

(1) A generic `TimeSeries`, or a subtype of `TimeSeries`, where the "data" dataset contains all 1 or True values (e.g., only the timestamps matter), one object per event type, and the "timestamps" dataset contains the timestamps of the events. Users might specify the "continuity" attribute of the "data" dataset to the value "instantaneous". These `TimeSeries` objects are sometimes placed within a `BehavioralEvents` or `BehavioralTimeSeries` container with other related `TimeSeries` objects.

This approach is not ideal because:

1. The `TimeSeries` type is generic, so downstream users and tools cannot easily detect that a `TimeSeries` object contains event data as opposed to other time series data
2. The `TimeSeries` type contains extraneous fields and complexity for storing event data. In the `TimeSeries` type, the dataset named "data" is required, so users often populate it with a value (e.g., an integer 1 or a boolean true) for every timestamp. The specific value may carry additional meaning information about the event. The datasets named "control", and "control_description" are not relevant for storing event data, and the "comments" attribute is also unnecessary. The subgroup "sync" is a legacy field in `TimeSeries`; all timestamps should be aligned before storing in NWB.
3. The `TimeSeries` type allows users to store the time axis as either an explicit 1D array of timestamps or a starting time and sampling rate when data are sampled at a regular frequency. Events rarely occur at a regular frequency, so this second approach to storing the time axis is not useful unless events have duration (see below).
4. Explicit timestamps are stored in the "timestamps" dataset of `TimeSeries` as float64 values (in seconds), but as some systems do not output event timestamps to that precision, it may be better to relax the minimum precision requirement from 64-bit to 32-bit. It may still be useful to allow the storage of the sampling rate (resolution) of the data acquisition system and the earliest possible event.

In some cases, events with duration are stored in NWB as `TimeSeries` objects, either:

- Continuously sampled where the "data" dataset contains a value (e.g., an integer 1 or a boolean true) for every timestamp that the event is ongoing and a different value (e.g., an integer 0 or a boolean false) for every other timestamp. Users might specify the

"continuity" attribute of the "data" dataset to the value "continuous". This is sometimes referred to as Digital I/O.
- Irregularly sampled where the "data" dataset contains a specific value (e.g., an integer 1 or a boolean true) for the onset of the event and a different value (e.g., an integer 0 or a boolean false) for the offset of the event. Users might specify the "continuity" attribute of the "data" dataset to the value "step". This usage is uncommon.

(2) An `AnnotationSeries`, which is a subtype of `TimeSeries`, where the "data" dataset contains custom text labels representing different event types and the "timestamps" dataset contains the time of each event.

This approach is better than (1) for storing event data, but it is still not ideal, because the `AnnotationSeries` type, which is a subtype of `TimeSeries`, contains extraneous fields, does not allow for parameterization of event types or non-string labels, and cannot capture events with durations.

(3) A generic `DynamicTable` with custom columns, where event times are stored one per row, similar to the `Units` table. Users might add additional custom columns with metadata about each event or event type.

This approach is not ideal because `DynamicTable` is generic, so downstream users and tools cannot easily detect that a `DynamicTable` object contains event data, and the column names are not standardized, which makes it difficult for software tools and people to interpret the data.

In some cases, events with duration are stored as a combination of onset and offset times, e.g., for stimulus presentation. These are most often stored as custom columns within a Trials `DynamicTable`. The names of these columns are variable.

Events with duration could also be represented using the "start_time" and "stop_time" columns of a `TimeIntervals` table neurodata type, but that type is intended to be used to store longer time periods, such as epochs or trials, and would require a duration for all events in the table.

(4) One of the `Events`, `LabeledEvents`, and `AnnotatedEventsTable` from ndx-events 0.2.0.

Ndx-events version 0.2.0 was released in October 2020. It primarily contained four new neurodata types with different levels of complexity and expressivity:
- An `Events` type for storing simple events. It has a description and a 1D array of timestamps.
- A `LabeledEvents` type that inherits from `Events` for storing events that can take on multiple values. Each timestamp is associated with an unsigned integer value in the "label_keys" dataset. Each unique value of the "label_keys" dataset is associated with a text label in the "labels" dataset.

- A `TTLs` type that is an alias for the `LabeledEvents` type, that should be used only for TTL data. To our knowledge, no public dataset uses this type.
- An `AnnotatedEvents` type that inherits from `DynamicTable`. Each row corresponds to a different event type and has an ID, a text label, a text description, and a 1D ragged array of timestamps.

While these types improve on previous approaches, the approach of having these four new types is not ideal because it was unclear to data curators which was the best type to use, and it was unclear to downstream users and tools which type to look for to find event data. A data analysis or visualization tool had to accommodate all four types. Experimenters who convert data to NWB, researchers who reuse NWB data, and tools that interpret NWB data would all benefit from standardized, simple, dedicated neurodata types for storing event and TTL data.

While the `LabeledEvents` type could be solely used to describe most types of events, it does not easily allow users to add custom, structured metadata to each event in the way that an `AnnotatedEvents` type does with its custom columns.

These four approaches are the most commonly observed ways to store event data currently in NWB.

3.3. Scope

What aspects are within the scope of this proposal? What aspects are outside the scope of this proposal?

In scope: Instantaneous events, including TTL pulses, as described above. Events can also have a duration.

Out of scope: Mapping of event metadata values stored in a file to more descriptive meanings. For example, the value “left” in the column “nosepoke” could be mapped to the meaning “the infrared beam was broken in the left-most port, signaling a likely nosepoke”. While this mapping is related to the current proposal, it can be implemented in many ways and has broader relevance to all terms in NWB. An experimental implementation is presented in this proposal.

4. Implementation

4.1. Proposed changes to schema and software

How should the schema of the extension be structured?
What changes to software should be made to support the extensions?

Are there any best practices or conventions that are not enforced by the proposed enhancements to the schema (if any) that should be implemented in the NWB Inspector when encountering relevant data?
Please provide examples.
Use the NDX template to implement the extension proposal.

Based on feedback from the NWB Technical Advisory Board, review of the usage of ndx-events 0.2.0, and feedback from the NWBEP001 Review Working Group, we propose a new, streamlined set of NWB neurodata types:

1. A `TimestampVectorData` type that extends `VectorData` and stores a 1D array of *timestamps* (float32) in seconds
    a. Values are in seconds from session start time (like all other timestamps in NWB)
    b. It has a scalar string attribute named "unit". The value of the attribute is fixed to "seconds".
    c. It has an optional scalar float attribute named "resolution" that represents the smallest possible difference between two timestamps. This is usually 1 divided by the sampling rate for timestamps of the data acquisition system. (Alternatively, the event sampling rate could be stored.)
    d. This type can be used to represent a column of timestamps in any `DynamicTable`, such as the NWB `Units` table and the new `EventsTable` described below.
2. A `DurationVectorData` type that extends `VectorData` and stores a 1D array of *durations* (float32) in seconds. It is otherwise identical to the `TimestampVectorData` type.
    a. If this is used in a table where some events have a duration and some do not (or it is not known yet), then a value of NaN can be used for events without a duration or with a duration that is not yet specified. If the latter, the mapping should be documented in the description of the `DurationVectorData`.
3. A `CategoricalVectorData` type that extends `VectorData` and stores the mappings of data values (of any type) to meanings. This is an *experimental* type to evaluate one possible way of storing the meanings (longer descriptions) associated with different categorical values stored in a table column. This can be used to add categorical metadata values to an `EventsTable`. This type will be marked as *experimental* while the NWB team evaluates possible alternate solutions to annotating the values of a dataset, such as LinkML-based term sets, non-table based approaches, and external mapping files.
    a. The type contains an object reference to a `MeaningsTable` named "meanings". See below. Unfortunately, because `CategoricalVectorData` is a dataset, it cannot contain a `MeaningsTable` within it, so the MeaningsTable is placed in the parent `EventsTable` and referenced by the `CategoricalVectorData`.
    b. It may also contain an optional 1D attribute named "filter_values" to define missing and invalid values within a data field to be filtered out during analysis, e.g., the dataset may contain one or more of: "undefined" or "None" to signal that

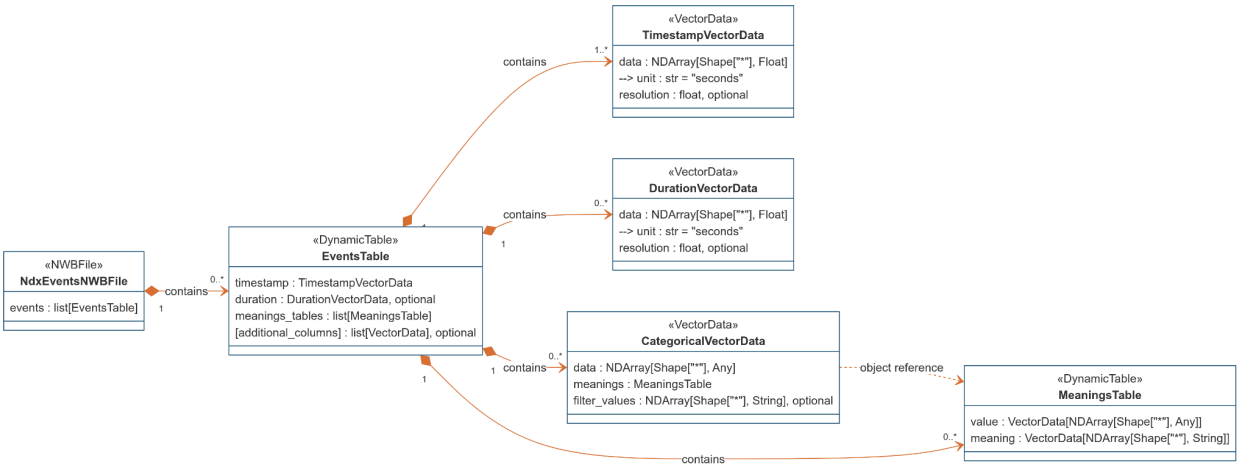
those values in the `CategoricalVectorData` dataset are missing or invalid. Due to constraints of NWB/HDMF attributes, attributes must have a dtype, so currently, only string values (not -1 or NaN) are allowed.

c. This type is similar to an `EnumData`, which is a `VectorData` of an enumerated type, except that the values stored in the column are strings that are short-hand representations of the concept, as opposed to integers. Storing strings is slightly less efficient than storing integers, but for these use cases, these tables will rarely be large and storing strings directly is more intuitive and accessible to users.

4. A `MeaningsTable` type that extends `DynamicTable` with two required columns:
    a. A “value” column that contains all the possible values that could be stored in the parent `CategoricalVectorData` object. For example, if the `CategoricalVectorData` stores the port in which the subject performed a nose poke, the possible values might be “left”, “center”, and “right”. All possible values must be listed, even if not all values are observed, e.g., if the subject does not poke in the “center” port, “center” should still be listed to signal that it was a possible option.
    b. A “meaning” column with string dtype that contains a longer description of the concept. For example, for the value “left”, the meaning might be “The subject performed a nosepoke in the left-most port, from the viewpoint of the subject. This is signaled by detection of the port’s infrared beam being broken.”
    c. Users can add custom, user-defined columns to provide additional information about the possible values, such as HED (Hierarchical Event Descriptor) tags. For HED tags, users may consider using the `HedTags` type, a subtype of `VectorData`, in the ndx-hed extension.
    d. As described in `CategoricalVectorData`, this arrangement will be marked as experimental.
5. An `EventsTable` type for storing a collection of event times that have the same parameterizations/properties/metadata (i.e., they are the same type of event, such as licks, image presentations, or reward deliveries)
    a. It inherits from `DynamicTable` and stores metadata related to each event time / instance, one per row.
    b. It has a “timestamp” column of type `TimestampVectorData` is required.
    c. It has a “duration” column of type `DurationVectorData` is optional.
    d. Because this inherits from `DynamicTable`, users can add additional custom columns to store other metadata, such as parameterizations of an event, e.g., reward value in uL, image category, or tone frequency.
    e. The “description” of this table should include information about how the event times were computed, especially if the times are the result of processing or filtering raw data. For example, if the experimenter is encoding different types of events using a “strobed” or “N-bit” encoding (see 8.1. TTL Pulses in the Appendix), then the “description” value should describe which channels were used and how the event time is computed, e.g., as the rise time of the first bit.

f. It contains a collection of `MeaningsTable` objects referenced by any `CategoricalVectorData` columns. These columns are placed in a subgroup of the `EventsTable` named “meanings”. Alternatively, these `MeaningsTable` objects could be placed under the root `NWBFile`, but it is probably more useful to keep them close to the objects that they describe. As described in `CategoricalVectorData`, this arrangement will be marked as experimental.

Here is a graphical representation of the new data types:



The PyNWB and MatNWB APIs would provide functions to create these tables. For example, in PyNWB:

```python
stimulus_presentation_events = EventsTable(name=“stimulus_presentation_events”)
stimulus_presentation_events.add_column(“stimulus_type”,
col_cls=CategoricalVectorData)
stimulus_presentation_events.add_row(timestamp=1.0, stimulus_type="circle")
stimulus_presentation_events.add_row(timestamp=4.5, stimulus_type="square")
nwbfile.add_events_table(stimulus_presentation_events)
```

The APIs would also provide the following interfaces:

1. `nwbfile.events_tables` returns a dictionary of `EventsTable` objects, similar to `nwbfile.acquisition`
    a. Use `nwbfile.events_tables[“stimulus_presentation_events”]` to access an `EventsTable` by name
2. `nwbfile.merge_events_tables(tables: list[EventsTable])`, which merges a selection of `EventsTable` objects into a read-only table, sorted by timestamp
3. `nwbfile.get_all_events()`, which merges all the `EventsTable` objects into one read-only table, sorted by timestamp

A working NWB extension can be found here: https://github.com/rly/ndx-events

The extension works with PyNWB (Python) and MatNWB (MATLAB) on all operating systems. The extension has no dependencies besides the core NWB software.

Within ndx-events, there is an additional neurodata type, `NdxEventsNWBFile`, that extends `NWBFile` and adds an untyped group named "events" at the root of the file (at the same level as "acquisition" and "processing"). This group can contain any number of `EventsTable` objects. This is similar to how currently in NWB, the root group has a subgroup named "intervals" that holds `TimeIntervals` table objects. After integration of this proposal with the core schema, the `NWBFile` schema should be updated to include this added group named "events".

After integration of this proposal with the core NWB schema, we propose that:

- Currently, a `BehavioralEvents` object contains any number of `TimeSeries` objects that represent behavioral event times of different types. This hierarchy is unnecessary when the `EventsTable` and `EventTypesTable` can hold multiple event types. Deprecate the `BehavioralEvents` type (old files with this scheme must still be readable, but new files should not be allowed). Recommend the usage of `EventsTable` instead.
- Discourage or deprecate the `AnnotationSeries` type in favor of `EventsTable`.
- Discourage the use of the "continuity" attribute on the "data" dataset of `TimeSeries`.
- Consider updating the "spike_times" dataset in the `Units` table with a `TimestampVectorData`. Similarly for the "start_time" and "stop_time" datasets of the `TimeIntervals` table.
- If data contain an ndx-events neurodata type (version 0.2.0), then the PyNWB API should read these into classes representing the core versions of these data types, and reorganize the data as necessary.
- The NWB Inspector and best practices should be updated to detect and discourage all other known forms of storing event data in NWB in favor of the ones proposed here.

4.2. Examples

4.2.1. Example 1 - Licks from different spouts

`Path: "/events/licks" (type: EventsTable)`

- `description: Lick times.`

timestamp (type: TimestampVectorData with dtype float)	lick_spout (type: VectorData with dtype string)
2.0	left

5.5	right
5.6	left

The “lick_spout” column (`path: "/events/licks/lick_spout"`) might have the property:

- `description: Location of the spout that was licked: “Left” or “Right”.`

4.2.3. Example 2 - Licks from different spouts with meanings table

`Path: "/events/licks" (type: EventsTable)`

- `description: Lick times.`

timestamp (type: TimestampVectorData with dtype float)	lick_spout (type: CategoricalVectorData with dtype string)
2.0	left
5.5	right
5.6	left

In contrast to Example 1, the “lick_spout” column is of type `CategoricalVectorData` instead of `VectorData`. It might have the properties:

- `description: Location of the spout that was licked: “Left” or “Right”.`
- `meanings (type: MeaningsTable)`: an object reference to a `MeaningsTable` in the parent EventsTable at the path `"/events/licks/meanings/lick_spout_meanings"`

value (type: VectorData with dtype string)	meaning (type: VectorData with dtype string)
left	A lick occurred in the left spout from the subject’s point of view
right	A lick occurred in the right spout from the subject’s point of view

Compared to Example 1, this encoding provides a more explicit, standardized description of the possible values in the column “lick_spout” and describes their meanings.

4.2.3. Example 3 - Adding HED tags

In Example 2, the “meanings” table can be annotated with Hierarchical Event Descriptor (HED) tags to provide additional richness and functionality.

- `meanings (type: MeaningsTable):`

A

value (type: VectorData with dtype string)	meaning (type: VectorData with dtype string)	HED (type: HedTags)
left	A lick occurred in the left spout from the subject’s point of view	(Action/Lick,(Left, Object/Pipe))
right	A lick occurred in the right spout from the subject’s point of view	(Action/Lick, (Right, Object/Pipe))

column with type: `HedTags` can also be added to the `EventsTable` for event-specific tags.

4.2.4. Example 4 - Multiple types of events

If an experiment has four types of events - stimulus presentation, nosepoke, reward, and fixation - each with a different set of parameterizations, then the NWB file would contain an `EventsTable` for each.

Event Type 1: Stimulus Presentations

`Path: "/events/stimulus_presentations" (type: EventsTable)`
`description: Information about the visual stimulus presented to the subject at different times.`

timestamp (type: TimestampVectorData with dtype float)	stimulus_type (type: CategoricalVectorData with dtype string)	color (type: CategoricalVectorData with dtype string)	area_in_pixels_2 (type: VectorData with dtype: float)
1.0	circle	red	100
4.5	square	white	50

The “stimulus_type” column (`path "/events/stimulus_presentation_events/stimulus_type"`) might have the properties:

- `description: The stimulus that was presented at the center of the screen: "circle" or "square".`
- `meanings (type: MeaningsTable)`: an object reference to a `MeaningsTable` in the parent `EventsTable` at the path `"/events/stimulus_presentation_events/meanings/lick_spout_meanings"`

This

value (type: VectorData with dtype string)	meaning (type: VectorData with dtype string)	HED (type: HedTags)
circle	A circle presented at the center of the screen	Sensory-event, Visual-presentation, (Circle, {color}, {area_in_pixels_2})
square	A square presented at the center of the screen	Sensory-event, Visual-presentation, (Square, {color}, {area_in_pixels_2})

example uses HED tags that contain templates (in curly braces) that can be filled in by HED tools reading from other rows of the `EventsTable`. Note: the HED template notation is currently supported only for BIDS sidecars, but the HED team plans to add support to the HED specification and HED python tools.

The “color” column (`path "/events/stimulus_presentation_events/color"`) might have the properties:

- `description: The color of the stimulus: "red", "white", or "green".`
- `meanings (type: MeaningsTable)`: an object reference to a `MeaningsTable` in the parent `EventsTable` at the path `"/events/stimulus_presentation_events/meanings/color_meanings"`

value (type: VectorData with dtype string)	meaning (type: VectorData with dtype string)	HED (type: HedTags)

red	RGB (255, 0, 0)	Red
white	RGB (255, 255, 255)	White
green	RGB (0, 255, 0)	Green

The “area_in_pixels_2” column could have a `HED tag: “Area/# px^2”` metadata field to indicate that this HED tag applies to all values in the column (this is usually applicable to numeric columns). Applying a HED tag to a VectorData object is not yet supported by the ndx-hed extension but this feature is planned.

Event Type 2: Nosepokes

`Path: "/events/nosepokes" (type: EventsTable)`

timestamp (type: TimestampVectorData with dtype float)	port_number (type: CategoricalVectorData with dtype int)
2.0	3
5.5	1
5.6	2

The “port_number” column (`path “/events/nosepokes/port_number”`) might have the properties:

- `description: The number of the port the animal poked its nose in, counting from the left: 1, 2, or 3.`
- `meanings (type: MeaningsTable)`: an object reference to a `MeaningsTable` in the parent `EventsTable` at the path `"/events/nosepokes/meanings/port_number_meanings"`

value (type: VectorData with dtype int)	meaning (type: VectorData with dtype string)
1	The IR beam in the left-most port from the subject’s point of view was broken
2	The IR beam in the middle port from the subject’s point of view was broken

3	The IR beam in the right-most port from the subject's point of view was broken

Event Type 3: Rewards

Path: "/events/rewards" (type: EventsTable)

timestamp (type: TimestampVectorData with dtype float)	reward_in_ml (type: VectorData with dtype float)
3.5	0.12
12.2	0.21

The "reward_in_ml" column (path "/events/stimulus_presentation_events/reward_in_ml") might have the properties:

- description: The amount of water given to the animal at the end of a trial in mL.

A mapping is not appropriate for continuous values, so the column type is VectorData instead of CategoricalVectorData. However, if there are only 3 possible volumes of reward, the data curator may consider using a CategoricalVectorData instead.

Event Type 4: Fixation Events

In processing raw eye-tracking data, the experimenter might identify the object that the subject is looking at.

Path: "/events/fixation_events" (type: EventsTable)

timestamp (type: TimestampVectorData with dtype float)	duration (type: DurationVectorData)	mean_position_in_pixels (type: VectorData with dtype float, shape (2,) for (x, y))	fixated_object (type: CategoricalVectorData with dtype string)
3.6	0.4	(300,400)	car
12.0	0.7	(500,320)	face

The "mean_position_in_pixels" and "fixated_object" columns may have a description. The "fixated_object" column may also have a "meanings" table.

The result of `nwbfile.get_all_events()` would be a read-only table that merges the above four EventsTable objects, re-orders the events by timestamp, and uses “n/a” or nan to represent undefined values:

timestamp (type: TimestampVectorData with dtype float)	source_events_table (type: DynamicTableRegion)	stimulus_type (type: CategoricalVectorData with dtype string)	color (type: CategoricalVectorData with dtype string)	area_in_pixels (type: VectorData with dtype float)	port_number (type: CategoricalVectorData with dtype int)	reward_in_ml (type: VectorData with dtype float)
1.0	stimulus_presentation_events	circle	red	100	n/a	nan
2.0	nosepoke_events	n/a	n/a	nan	3	nan
3.5	reward_events	n/a	n/a	nan	n/a	0.1
3.6	fixation_events	n/a	n/a	nan	n/a	nan

For space, the columns from the `fixation_events` table are not shown in the above example.

Each `CategoricalVectorData` column can be inspected to get the column's description and the meanings for its possible values.

4.2.5. Example 5 - Multiple types of events in one table

If an experiment has two types of events, the data curator may choose to place these into a single `EventsTable` for ease of access and interpretation, at the cost of extra space to store undefined values when some columns are relevant to one type of event and not another.

```
Path: "/events/stimulus_presentations" (type: EventsTable)
description: Information about the visual stimulus presented to the
subject at different times.
```

timestamp	stimulus_modality (type:	stimulus_category (type:	visual_contrast (type: VectorData

(type: TimestampVectorData with dtype float)	CategoricalVectorData with dtype string)	CategoricalVectorData with dtype string)	with dtype: float)
1.0	visual	high	100
4.5	auditory	low	nan
6.0	visual	low	20
9.1	auditory	high	nan

4.3. Discussion

Does the implementation consider and meet the NWBEP quality metrics and evaluation criteria?
What alternatives for implementing were considered?
What are the advantages and disadvantages of the alternative implementations?
Are there reasons why this enhancement should NOT be made?

Although the structure of having two linked tables (`EventTypesTable`, `EventsTable`) is more complex than the simple `Events` type defined in ndx-events 0.2.0, based on community feedback, we realized it would be beneficial to downstream users and tools to have a *single* way of representing event data that supports all likely use cases, rather than several different ways that offer slight differences in storage efficiency and simplicity. Convenience functions can be created at the API level to reduce complexity. All data from ndx-events 0.2.0 can be converted to the proposed neurodata types in this NWBEP.

This structure is also similar to the BIDS schema for storing task events, where 1) an events.tsv file contains a table of events with columns for onset time, duration, trial type, and other fields, and 2) an events.json contains information about each column, including for trial types (event types) and a description for each trial type.

An earlier version of the proposal included a `TtlTypesTable` and a `TtlsTable` type specifically for TTL events, but it was deemed to be unnecessary by the review working group. An earlier version of the proposal also included a `EventTypesTable` to allow for user-defined metadata to be associated with each event type, but in practice, this adds complexity and there is no known use case for adding user-defined metadata to event types.

Advantages:

- Single method for storing events data, with a specialized subtype for TTL data.

- Allows custom metadata to be attached to each event type or event time.

Disadvantages:

- `DynamicTable` data are more complicated than single 1D arrays and can be difficult to inspect outside of the API. For simple use cases, such as a single array of lick times, this two-table structure may be overly complex. These issues could be resolved partially by adding convenience functions in the APIs.

Events often occur in the context of a behavioral task. We have developed a separate extension, ndx-structured-behavior, that loosely links the neurodata types for recorded events and event types that are defined in this proposal with neurodata types for a task program and an enhanced trials table.

Recorded events and event types may be annotated with Hierarchical Event Descriptors (HED) using the ndx-hed extension.

We cannot think of any reasons why this enhancement (or a similar enhancement that solves this problem) should not be made.

5. Public Examples

Example datasets using the proposed extension?
Example use cases?

Aside from the usages of the earlier 0.2.0 version of ndx-events described in Appendix Section 8.2., there are no known datasets using the latest proposed data types.

An example NWB file using ndx-events 0.4.0 can be downloaded from GitHub:
https://github.com/rly/ndx-events/blob/main/test_events.nwb

6. Changelog

For each main version of the enhancement proposal, briefly describe the main changes made. If the NWBEP is accompanied by a Neurodata Extension (NDX) then include a link to the corresponding changelog here.

6.1. Version 0.4.0 (Nov 11, 2024)

- Removed `EventTypesTable`, `TTLTypesTable`, `TTLsTable`, `Task` from version 0.3.0
- Added new data types `CategoricalVectorData` and `MeaningsTable`. Moved the EventsTable to be stored under the path “/events” in the NWB File.

6.2. Version 0.3.0

- Removed all data types from version 0.2.0.
- Added new data types `TimestampVectorData`, `DurationVectorData`, `EventTypesTable`, `EventsTable`, `TTLTypesTable`, `TTLsTable`, `Task`.

6.3. Version 0.2.0

Created new data types for `Events`, `LabeledEvents`, and `TTLs` for acquired data. (nwb-schema #302)

- An `Events` type for storing simple events. It has a description and a 1D array of timestamps (unit: 'seconds')
- A `LabeledEvents` type that inherits from `Events` for storing events that can take on multiple values. Each timestamp is associated with a uint value in the 'label_keys' dataset. Each unique value of the 'label_keys' dataset is associated with a text label in the 'labels' dataset.
- The `TTLs` type would inherit from `LabeledEvents`. The TTL pulse value would be used as the label key. This type groups all TTL values together in a single dataset, which is often how TTL data come in. If desired, users could separate the TTL events by pulse value and store those data in separate `Events` types in `/processing/events`.

These types would live in `/acquisition`.

Created an `AnnotatedEvents` type that inherits from `DynamicTable`. Each row corresponds to a different type of event and has an ID, a text label, a text description, and a 1D array of timestamps. The timestamps dataset would be a ragged array. Users could separate TTL events by pulse value and store those values in this new table. Users might want to store only the TTL events and event times that are relevant to their analyses here. This table might live in `/processing/events`.

Pros:

- The format of the acquired event times is easy to understand and the data would be easy to inspect.
- The format of the annotated events allows for the addition of user-defined metadata for each event type.

Cons:

- Event times are duplicated between acquisition events and annotated events, but the latter is organized, annotated, and could be filtered, so I don't think this is a big deal.
- `DynamicTable` data are more complicated and difficult to inspect outside of the API.
- The code for `DynamicTable` is bulky and has historically required hacks to make them work for particular use cases.

7. References

8. Appendix

8.1. TTL Pulses

In brief, a TTL pulse is a brief change in the state of a digital line that signals the occurrence of an experimental event, such as the presentation of a visual stimulus. Hardware TTL lines typically flip between 0 Volts (low) and 5 Volts (high) states, but TTL pulses can also be represented in software, where they result in no actual voltage change. In addition to indicating experimental events, TTL pulses may indicate the start and end of experimental epochs, or they may directly control physical devices, such as a camera or LED.

In more detail: Standard 5V TTL circuits have a "low" state that is encoded by a voltage of 0 to 0.8 volts and a "high" state that is encoded by a voltage of 2 to 5 volts. Depending on the configuration, data acquisition systems may encode transitions from the "low" state to the "high" state (rising edge) or from the “high” state to the “low state” (falling edge) as TTL pulse events. The widths (durations) of TTL pulses vary but typically last under 10 ms. The widths are typically not used (however, in rare cases, if the pulse width is too long, two TTL pulses close in time may not be resolvable, so it may be useful to store the pulse width if available).

TTL pulses can be sent to different channels of a data acquisition system to indicate different event types. This is one of the most common uses of TTL pulses in neurophysiology. A pulse on each channel could represent a different event type. This strategy is sometimes referred to as individual input encoding. If there are 8 TTL input channels and the data acquisition system assigns the TTL pulse a value based on the channel it was received on, then there would be 8 possible values for TTL pulses, 1 to 8.

Alternatively, some data acquisition systems can be configured to represent simultaneous TTL pulses across multiple channels as particular values. This strategy is sometimes referred to as strobed word encoding or N-bit encoding. For example, if an 8-bit strobed word encoding is used, then TTL pulses are received on one or more of 8 channels simultaneously (e.g., through a parallel port on a computer), and their combination encodes an 8-bit word, which can range from a value of 1 to 255.

In rare cases, the meanings of TTL pulses change during the course of a session, e.g., if multiple tasks are run during a single session. In rare cases, users may want to store metadata with each pulse rather than each pulse value, e.g., to mark particular events as invalid that should be ignored in analysis. These cases are not considered in this proposal.

See also discussions here:

- https://github.com/NeurodataWithoutBorders/nwb-schema/issues/301
- https://github.com/NeurodataWithoutBorders/nwb-schema/pull/302
- https://github.com/NeurodataWithoutBorders/nwb-schema/issues/306

Table 1. References on how different acquisition systems represent event and TTL data

Plexon	https://plexon.com/wp-content/uploads/2017/06/OmniPlex-Digital-Input-Guide.pdf (Chapter 9 Digital Inputs)
Neuralynx	https://neuralynx.com/documents/Digital%20Lynx%20SX%20User%20Manual.pdf https://neuralynx.com/software/NeuralynxDataFileFormats.pdf https://neuralynx.com/documents/CheetahReferenceGuide.pdf (Section 4.7 Digital I/O Overview)
Blackrock	https://www.blackrockmicro.com/wp-content/ifu/LB-0028-15.00-Cerebus-Instructions-for-Use.pdf
Bpod	https://sites.google.com/site/bpoddocumentation/user-guide/hardware-overview https://sites.google.com/site/bpoddocumentation/bpod-user-guide/overview
Bonsai	https://bonsai-rx.org/docs/tutorials/state-machines.html
Open Ephys	https://open-ephys.github.io/acq-board-docs/User-Manual/Peripheral-devices.html https://open-ephys.github.io/gui-docs/User-Manual/Recording-data/Binary-format.html#events
SpikeGLX	https://github.com/billkarsh/SpikeGLX/blob/master/Markdown/UserManual.md#synchronization https://github.com/billkarsh/SpikeGLX/blob/master/Markdown/ColorTTL_Help.md
MonkeyLogic	https://monkeylogic.nimh.nih.gov/docs_MainMenu.html
Inquisit	https://www.millisecond.com/support/docs/current/html/howto/howtosendport.htm
Tobii	https://connect.tobii.com/s/article/How-To-Send-and-Receive-Triggers?language=en_US
PsychToolbox	https://github.com/Psychtoolbox-3/Psychtoolbox-3/wiki/FAQ%3A-TTL-Triggers-in-Windows
Sending TTL pulses using USB	https://link.springer.com/article/10.3758/s13428-021-01571-z https://stefanappelhoff.com/usb-to-ttl/
ndx-structured-behavior	https://github.com/rly/ndx-structured-behavior

8.2. Examples of events via current pynwb core

The following examples show how events are encoded **without** the use of ndx-events proposed here, which is proposed to unify the representation of event data.

Lick times

1. DANDI:000409 uses a DynamicTable with a single non-id column “lick_time” to store lick times, one per row
2. DANDI:000017 uses a TimeSeries with boolean data (all True) and explicit timestamps to store instantaneous lick times. The TimeSeries is placed within a BehavioralEvents object.
3. DANDI:000231 uses two TimeSeries objects with int64 data (all 1s) and explicit timestamps to store instantaneous lick times on the left and on the right. The lick times are sampled at a resolution of 1 ms and may have a 60 ms refractory period. The TimeSeries are placed within a BehavioralEvents object.
4. DANDI:000054 uses a TimeSeries with float64 data (0, 1, 2, 3, 4) and regular sampling at 15.4 Hz to store the number of licks at each time. There is a separate TimeSeries that stores the smoothed version of this data as the lick rate. These TimeSeries objects are placed within a BehavioralTimeSeries object. This form of storing event data can be considered a continuous time series and may be easier for users to work with in this form than as a list of timestamps, given that other behavioral time series data and the neural responses share the same timestamps.

Reward times

1. DANDI:000231 uses two TimeSeries objects to store instantaneous reward times on the left and on the right, in the same way as it stores lick times (int64 data (all 1s) and explicit timestamps; see 1c. above).
2. DANDI:000054 uses a TimeSeries object with float64 data (0 or 1) and regular sampling at 15.4 Hz to store the times of collisions with objects. The TimeSeries are placed within a BehavioralTimeSeries object in the same way as it stores lick times (see 1d. above).
3. DANDI:000061 uses two AnnotationSeries objects with string data values “Left Reward” and “Right Reward” to store reward times on the left and right.

Other events

1. DANDI:000004, as described by the data descriptor paper https://www.nature.com/articles/s41597-020-0415-9, contains TTL data.
   The TTL pulse values are stored as string representations of floats, e.g., “31.0” which have different meanings, e.g., “31.0” means the user responded (new image, confident). All pulse values and timestamps are stored in a single AnnotationSeries object. The data also includes a TimeSeries that has matching timestamps and float data values indicating which experiment block each pulse occurred in (a TimeIntervals object may be better to indicate this).
2. DANDI:000015 uses three TimeSeries objects with float64 data (all 1s) and explicit timestamps to store times for go, sample, and delay. The TimeSeries are placed within a BehavioralEvents type.

3. DANDI:000061 uses AnnotationSeries objects named “puf” and “rip” to store string data values “Airpuff” and {“Ripple start 20”, “Ripple peak 20”, “Ripple stop 20”} respectively, in addition to the AnnotationSeries objects for reward events.
4. DANDI:000410 uses a TimeSeries with uint8 data (1 or 0) and explicit timestamps to store start and stop times of nose-pokes, respectively. The TimeSeries are placed within a BehavioralEvents type.

8.3. Examples of events via ndx-events

10 dandisets have published NWB files using neurodata types from ndx-events 0.2.0.

Events

1. DANDI:000053
    a. ndx-events.Events “licks”
    b. ndx-events.Events “rewards”
    c. These Events objects and other objects (e.g., EyeTracking, Position, TimeSeries) are stored together in a ProcessingModule named “behavior”.
2. DANDI:000055
    a. ndx-events.Events “ReachEvents”
        i. Contralateral wrist movement events. Quantitative neural and behavioral features for each event are in the “reaches” TimeIntervals table with columns for each feature. However, the timestamps don’t match between the ReachEvents and the “reaches” TimeIntervals table… They are offset by 30682.41 seconds.
3. DANDI:000251
    a. ndx-events.Events “lick_times”

LabeledEvents

4. DANDI:000060
    a. ndx-events.LabeledEvents “LabeledEvents”
        i. Labels: ["sample-start chirp_start_times", "sample-start chirp_stop_times", "go_start_times", "go_stop_times", …]
5. DANDI:000230
    a. ndx-events.LabeledEvents “task_events”
        i. Labels: ["PhotoStim_start_times", "PhotoStim_stop_times"]
6. DANDI:000552
    a. ndx-events.LabeledEvents “RewardEventsEightMazeTrack”
        i. Labels: ["right_reward", "left_reward"]
7. DANDI:000559
    a. ndx-events.LabeledEvents “BehavioralSyllableOffline”
        i. Labels: ['Paw lick/scrunch', 'Pause', 'Pause, low rear', 'Pause', …]

8. DANDI:000568
    a. ndx-events.LabeledEvents “RewardEventsLinearTrack”
        i. Labels: ["right_reward", "left_reward"]

AnnotatedEventsTable

9. DANDI:000350
    a. ndx-events.AnnotatedEventsTable “BurstEvents”
        i. Events of classified bursting activity.
10. DANDI:000351
    a. ndx-events.AnnotatedEventsTable “eventidx_table”
        i. Four event types: [session end, Lick3 onset, Lick3 offset, Background solenoid] with codes/indexes [0, 5, 6, 7], used with a custom extension for showing when certain events occurred. The extension has three parallel arrays for event code, event time, and non-solenoid flag.

8.4. Appendix References

https://www.allaboutcircuits.com/textbook/digital/chpt-3/logic-signal-voltage-levels/
https://learn.sparkfun.com/tutorials/logic-levels/ttl-logic-levels