Structured Data Standards

XINA standard data structures terms and organizing principles.

Introduction

Although XINA is very flexible and can be configured to meet almost any data organization requirements, we have defined standard organization principles for common use cases with pre-built front end tooling. These are not hard limitations, just recommendations based on past experience, performance benchmarks, and cost/benefit analysis. Additionally, by adhering to these standards projects can quickly leverage built-in XINA front end tools and data processing pipelines, as well as first class API actions for interacting with data in complex ways. We call this collection of standards **structured data standards**, or **structs**.

Data Models

The primary organizational concept of the struct system is the **data model**. Abstractly, a data model (or simply **model**) is defined as having a set of **synchronously relevant data**. For example, a project might have a flight model, ETU model, etc. Models store data in independent databases, and multiple models may be importing data in parallel.

Broadly we use **time** as the primary method to organize and synchronize data within a model. In XINA this is represented as an 8-byte unsigned integer Unix time with microsecond precision. We use Unix time because it is:

- Widely and consistently supported
- Time zone independent
- Efficiently converted

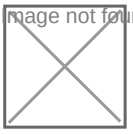
Other time formats may be available for data export depending on project requirements.

Project Organization

Data models must employ certain organizational requirements in XINA to ensure they are interpretted correctly by struct API calls and front end tools. These apply to both structures within model groups, as well as the organization of model groups themselves.

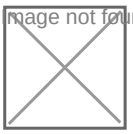
Projects / Categories

A **project** should be defined by a single XINA group at the top level. Each model is then defined by a single XINA group, which contain all groups and databases associated exclusively with the model. These should either be defined in the project group, or may be subdivided into **category** groups.



A project may use a mix of both approaches or additional levels of subcategories if required, but it is recommended to either use a flat structure or single level of category groups to avoid confusion. Models may be referred to by the path relative to their project group (in the above example, model_a would be referenced as `model_a` or `category_a.model_a`, respectively).

Project and category groups may also include additional groups and databases of data or resources which are not model specific, such as journals or definitions databases. In most cases with standard structures, models will default to databases or groups within the model, but search for them up the tree if not found. A complete project group might look like:



Project Configuration

A group is defined as a project by the `xs_struct_project` key. The value is a JSON object with the following definition:

Key	Value	Default
`def_mn`	relative path to mnemonic definitions database	`def.mn`
`def_prof`	path to profile definitions database	`def.prof`
`def_plot`	path to plot definitions database	`def.plot`

A group is defined as a category by the `xs_struct_category` key. The value is a JSON object extending the definition of the `xs_struct_project` key, automatically inheriting any unset values from the project configuration.

All models are required to provide an `mn_def`, `prof_def`, and `plot_def` database. It is **strongly recommended** that these be shared by the entire project, and that all models use the same temporal precision, to maximize

intercompatibility between models. Sharing definitions databases does not preclude identifying particular definitions as relevant only to specific models.

Model Organization

Data within a model falls into four primary classifications:

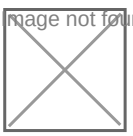
- **Telemetry**
  - source data file(s) from data collection point
  - typically stored in a raw (sometimes binary) format
  - storage cost is cheap
  - accessing data means downloading files or most likely requires custom XINA tools
  - may be divided into multiple **data sources** (see below)
- **Viewable Data**
  - extracted from telemetry into XINA database(s)
  - telemetry is the single source of truth for this data, not intended to be user editable
    - (except under controlled circumstances with struct API calls)
  - data is either **mnemonic**, **instant**, or **interval** (see below)
  - can be accessed and analyzed with built-in XINA tools
  - storage is expensive
  - optimizations may be needing depending on project requirements, data volumes
- **User Metadata**
  - additional data added by users, often directly through the XINA interface
  - XINA likely the primary repoository for this data
  - for example, a journal
- **Definitions / References**
  - may be user entered or defined outside XINA
  - may exist at model level or above (category/project level)
  - more formal and restricted than user metadata

Model Configuration

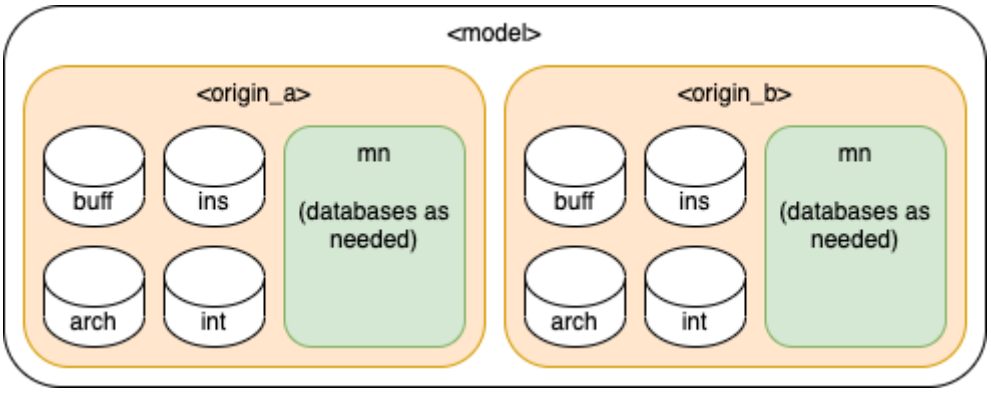
A group is recognized as a model if the `xs_struct_model` key is set in the group objects. The value is a JSON object extending the definition of the `xs_struct_project` key, automatically inheriting any unset values from the parent project or category configuration.

Origin

Abstractly, a **data origin** (or simply **origin**) is a single point of data import to a model. In many cases, a model will only have a single data origin; for example, if all data is provided directly from a single instrument, or multiple components are merged into a single data stream through FEDS before import into XINA. In these cases delineation by origin is not required in model organzation, and should use this pattern:



However, in environments with multiple import points running in parallel, databases must be designed with multiple origins.



In this example each source file would need to specify either `origin_a` or `origin_b`. Additionally, each origin has distinct databases for instant, interval, and mnemonic data. This would be required if each data source provided all three data types. As requirements for instants and intervals are less stringent than mnemonics, in some circumstances instants and intervals could be considered a single source and populated independently:

Image not found or type unknown

Mnemonics

A **mnemonic** defines a single field of **numeric** data in a XINA model. A **datapoint** is a single logical piece of data, consisting of:

- time (Unix microseconds)
- mnemonic identifier
- value (numeric)

In other words, **the value of a single mnemonic at a moment in time**.

A model has one or more mnemonic databases, containing all of the datapoints associated with the model.

Mnemonic Definitions

All mnemonics must be defined in a **mnemonic definitions** database. Again, it is **strongly recommended** to use a single definitions database for an entire project to faciliate comparison of data between models.

A core challenge of working with mnemonics is synchronizing mnemonic definitions from XINA to the point of data collection. Especially in early test environments, fields may be frequently added or removed on the the fly and labels may change, but must be consistently associated with a single mnemonic definition. Broadly there are two approaches to manage this challenge.

The first is user maintained mnemonic definitions. This is recommended for environments without frequent changes, and ideally one data source. The end user is responsible for ensuring that imported data has matching `mn_id` values to mnemonics present in the definitions database. This will typically result in faster imports and support complex or custom data pipeline solutions.

The second solution is allowing XINA to manage mnemonic definitions. With this approach, data can be imported with plain text labels and automatically associated with mnemonic definitions if available, or new definitions can be created on the fly.

Both approaches can be accomplished with the `model_mn_import` API action, documented here. The details of the required approach will depend on project requirements.

Standard Fields

field	type	description
`mn_id`	`int(4)`	unique mnemonic ID
`name`	`utf8vstring(128)`	unique mnemonic name
`desc`	`utf8text`	plain text mnemonic description
`meas`	`utf8vstring(32)`	measurement label (for example, `voltage`, `current`)
`unit`	`utf8vstring(32)`	measurement unit (for example, `V`, `mA`)
`state`	`model_mn_state`	current state of mnemonic (`active`, `inactive`, `archived`, `deprecated`)

field	type	description
`origins`	`jsonobject`	map of model(s) to associated origin(s)
`full`	`asciivstring(32)`	the primary database for the mnemonic, default `f8` (may be `null`)
`bin`	`set(asciivstring(32))`	the opt-in bin database(s) to include the mnemonic in
`format`	`asciivstring(32)`	printf-style format to render values
`enum`	`jsonobject`	mapping of permitted text values to numeric values
`labels`	`list(jsonobject)`	mapping of numeric values or ranges to labels
`aliases`	`set(asciivstring(128))`	set of additional names associated with the mnemonic
`meta`	`jsonobject`	additional metadata as needed
`query`	`asciivstring(32)`	query name for meta-mnemonics (may be `null`)
`conf`	`jsonobject`	configuration for meta-mnemonics (may be `null`)

Mnemonic names are **case insensitive** and **normalized** with any leading/trailing whitespace removed, with any internal whitespace represented by a single underscore character. For example, `"v_mon"` = `"V  Mon"` = `" V MON "`. Although not required, XINA tools will interpret the period character (`.`) to indicate a tree structure of mnemonic relationships, and brackets (`[]`) to indicate an array of values. Although the mnemonic name is intended to be unique, insertion of a mnemonic with the same name but different unit will create a new mnemonic definition. This is intended to avoid interruption of data flow, but should be corrected with the Mnemonic Management tool when possible. The `model` and `origin` are populated automatically for auto-generated mnemonic definitions.

The mnemonic `state` affects how the mnemonic will be displayed and populated. An `inactive` mnemonic indicates data is no longer relevant or actively populated and will be hidden by default. A `deprecated` mnemonic extends this concept but will throw errors if additional data points for the mnemonic are imported.

If `enum` is provided a mnemonic will apply labels to enumerated numeric values, as provided in `values`. For example, a 0|1 on|off state could be represented by `{"0":"OFF", "1":"ON"}`. Values in this map may also be used to parse imported data.

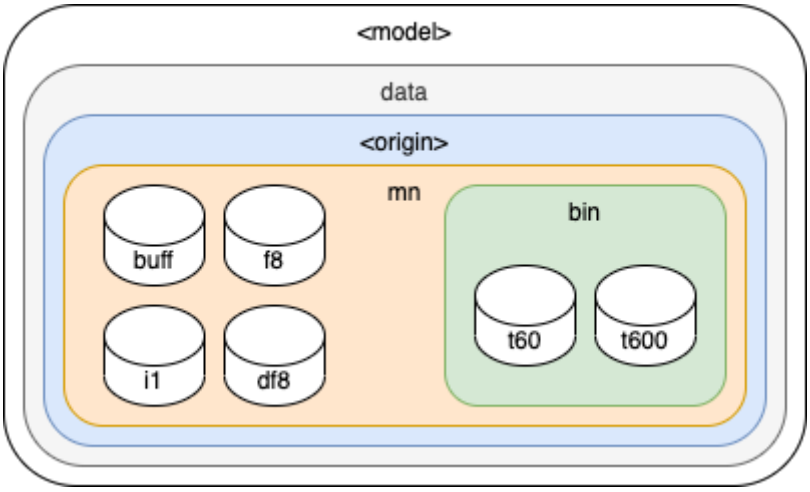
A mnemonic may specify one or more `aliases` to indicate additional names that should be included in the single mnemonic definition. If present, the aliases are referenced at a **higher priority** than the mnemonic name during import lookup. For example, a given mnemonic `a` is erroneously labeled `b` in some imported data, which creates a new separate mnemonic definition for `b`. To correct this, `b` could be added as an alias for `a`, and the `b` mnemonic could be deprecated. All `a` and `b` data from the source telemetry would then correctly be merged into the `a` mnemonic.

`name`, `unit`, `state`, `enum`, `models`, and `aliases` may be used during the data import process to validate and interpret data. Full details of how each field is used is documented with the associated API action.

Mnemonic Databases

Within a model, each data source must have a set of one or more mnemonic databases. Each set should be contained by a group, which can be configured to define any relationships between the databases. This will

typically include a **full** database, containing all or **delta** optimized data (see below for additional information), and one or more types of **bin** databases, depending on requirements.



While each data source must have its own mnemonic database(s), it may be beneficial for a single data source to further subdivide mnemonics into different types of databases for optimization purposes. For example, a model with a large number of mnemonics that only require single byte precision would see significant performance gains from separate databases using the `int(1)` type. In practice this could look like:

Full Database

In most cases, there will be a single primary database containing **full mnemonic** data (all points from original telemetry), **delta mnemonic** data (an optimization option, see below), or a mix of both. Data is stored with a single data point per row.

Standard Fields

field	type	description
`t`	`instant(us)`	time
`mn_id`	`int(4)`	unique mnemonic ID
`v`	`float(8)`	data point value (may be `null`)
`n`	`int(4)`	number of data points
`ref_id`	`int(4)`	mnemonic ID on insert

A value of `null` may be used for `v` to indicate a gap in data, otherwise data will appear visually connected by default in XINA charts. `null` may also be appropriate to represent `NaN` or `Inf` values, as these cannot be stored in the database, but the preference to include these as `null` or omit them altogether may depend on an individual project.

For large data sets with infrequent value changes, it may be beneficial to employ a **delta mnemonic** optimization. This requires the `n` field listed above. In this case, a point is only included in the database at the moment the value for a given mnemonic changes, and the number of points is stored in `n`. For example, given the set of points:

t	v
0	0
1	0

t	v
2	0
3	1
4	1
5	1
6	1
7	2
8	2
9	2

Delta optimization would condense the data to:

t	v	n
0	0	2
2	0	1
3	1	3
6	1	1
7	2	2
9	2	1

Note the final data point of a data set is always included.

Bin Database(s)

The most common data optimization employed with mnemonics is **binning**, combining multiple data points over a fixed time range into a single data point with a min, max, avg, and standard deviation. A model may define one or more bin databases depending on performance requirements, but four types are supported by default. The time range of bins is interpretted as `[start, end)`.

Time Binning

Bins are applied on a **fixed time interval** for all points in the database (for example, 1 minute or 1 hour).

Standard Fields

field	type	description	required
`t`	`instant` (matching model standard)	start time of the bin	yes
`t_min`	`instant` (matching model standard)	time of first data point in bin	yes
`t_max`	`instant` (matching model standard)	time of last data point in bin	yes
`mn_id`	`int(4)`	unique mnemonic ID	yes
`n`	`int(4)`	number of data points in bin	yes

field	type	description	required
avg	float(8)	average of points in bin	yes
min	float(8)	min of points in bin	yes
max	float(8)	max of points in bin	yes
med	float(8)	median of points in bin	no
var	float(8)	variance of points in bin	no
std	float(8)	standard deviation of points in bin	no

Interval Binning

Bins are based on explicitly defined **intervals**.

Standard Fields

field	type	description	required
t_start	instant(us)	start time of the bin	yes
t_end	instant(us)	end time of the bin	yes
dur	duration(us)	duration	yes
t_min	instant(us)	time of first data point in bin	yes
t_max	instant(us)	time of last data point in bin	yes
u_id	UUID	UUID of associated interval	yes
p_id	int(8)	primary ID of associated interval	yes
s_id	int(4)	secondary ID of associated interval	yes
mn_id	int(4)	unique mnemonic ID	yes
n	int(4)	number of data points in bin	yes
avg	float(8)	average of points in bin	yes
min	float(8)	min of points in bin	yes
max	float(8)	max of points in bin	yes
med	float(8)	median of points in bin	no
var	float(8)	variance of points in bin	no
std	float(8)	standard deviation of points in bin	no

Events

To organize time based data in XINA, we employ **events**, which come in two forms: **instants**, referring to a **single moment in time**, and **intervals**, referring to a **range of time**. The goal of events is to make it easy to find, compare, and trend data. Each has their own databases and include fields for:

- **type** (indicates how the event should be viewed and interpreted)
- **UUID** (universally unique identifier, generated at the creation of the event)
- numeric **event ID** (meaning can depend on type)
- plain text **label** (up to 128 bytes)
- plain text, HTML, or JSON **content**
- optional JSON object **metadata**

The UUID uniquely identifies an event, and is the only way to permanently, globally specify it. It should be applied at the time of creation to ensure consistency even if data is reprocessed. The event ID is optional, and can be used as needed (when not provided it will be zero by default). Its much faster and more reliable to query numbers than text, so this is the best way to indicate events having commmon meaning.

Event Database

Default Location

`<model>.event`

`<model>.eventf` (single file per event)

`<model>.eventfs` (multi file per event)

Required Fields

field	type	description
`uuid`	`uuid`	UUID
`e_id`	`int(8)`	event ID
`t_start`	`instant(us)`	start time (inclusive)
`t_end`	`instant(us)`	end time (exclusive)
`dur`	`duration(us)`	`t_end` - `t_start`
`interval`	`boolean`	`true` if event is an interval, `false` if event is an instant
`open`	`boolean`	`true` if event is an open interval, `false` otherwise
`type`	`struct_event_type`	event type (see below)
`level`	`struct_event_level`	event level (see below)
`label`	`utf8vstring(128)`	plain text label
`content`	`utf8text`	extended text / CSV / HTML / JSON

field	type	description
`meta`	`jsonobject`	additional metadata as needed
`conf`	`jsonobject`	additional information specific to `type`

Note that `duration`, `interval`, and `open` are **computed automatically** from `t_start` and `t_end` and **cannot be provided manually**.

Event Types

XINA defines a fixed set of standard event types, each with an associated numeric code. The type is stored as the code in the database for performance reasons; for practical purposes most actions can use the type name directly, unless interacting directly with the API.

Standard Types

code	name	ins	int	description
`0`	`message`	?	?	Basic event, IDs optional, no implicit ID interpretation
`1`	`marker`	?	?	Organized event, IDs imply related events
`2`	`alert`	?	?	Organized event, level (severity) required, IDs imply related events
`2000`	`test`		?	Discrete test period, may not overlap other tests, IDs optional, unique if used
`2001`	`activity`		?	Discrete activity period, may not overlap other activities, IDs optional, unique if used
`2002`	`phase`		?	Discrete phase period, may not overlap other phases, IDs optional, unique if used
`3000`	`data`	?	?	General purpose data set
`3001`	`spectrum`	?	?	General purpose spectrum data

Additional types will be added in the future as needed, with codes based on this chart:

Standard Type Code Ranges

code	ins	int	description
`0-999`	?	?	General types for instants and intervals
`1000-1999`	?		General types for instants only
`2000-2999`		?	General types for intervals only

code	ins	int	description
`3000-3999`	?	?	Data set types for instants and intervals
`4000-4999`	?		Data set types for instants only
`5000-5999`		?	Data set types for intervals only

Data Format

The `data` event type indicates a basic data set. This is typically used with the single file per event database structure, in which case the file will contain the data set. For event databases without files, the data is expected to be stored in the `content` field. This is only recommended for small datasets (less than 1MB).

Files must be either ASCII or UTF-8 encoded. New lines will be interpretted from either `\n` or `\r\n`. The `conf` object may define other customization of the format:

Conf Definition

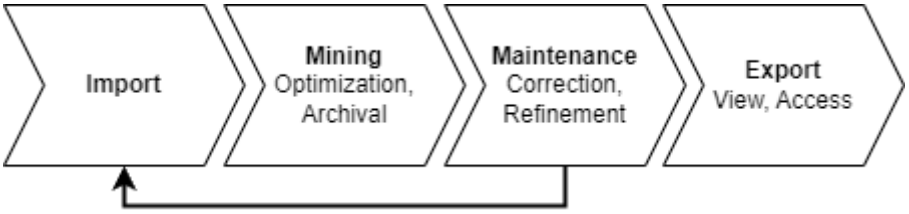
Key	Value	Default	Description
`delimiter`	`string`	auto detect (`','`, `'\t'`, `';'`)	value delimiter
`quoteChar`	`character`	`"` (double quote character)	value quote character
`ignoreLines`	`number`	`0`	number of lines to skip before the header
`invalid`	`null`, `'NaN'`, `number`	`null`	preferred interpretation of invalid literal
`nan`	`null`, `'NaN'`, `number`	`null`	preferred interpretation of `'Nan'` literal
`pInfinity`	`null`, `'Inf'`, `number`	`null`	preferred interpretation of positive `'Infinity'` literal
`nInfinity`	`null`, `'Inf'`, `number`	`null`	preferred interpretation of negative `'Infinity'` literal
`utc`	`boolean`	`false`	if `true`, interpret all unzoned timestamps as UTC

Starting after the number provided for `ignoreLines`, the content must include a header for each column, with a name and optional unit in parentheses. Special standard unit names may be used to indicate time types, which will apply different processing to the column:

Unit	Description
`ts`	text timestamp, interpretted in local browser timezone (absent explicit zone)
`ts_utc`	text timestamp, interpretted as UTC timezone (absent explicit zone)
`unix_s`	Unix time in seconds
`unix_ms`	Unix time in milliseconds
`unix_us`	Unix time in microseconds

Structs Data Lifecycle

The XINA structs mnemonic data lifecycle involves four primary phases:



Source Files

Each origin maintains a set of **source files**, containing all data imported into XINA for that origin.

The primary type of source files are **archive source files**. Archive files are considered the **definitive record of source data for a range of time for a single origin**. These are stored in the XINA xbin binary file format. These are imported directly with the STRUCT ARCHIVE IMPORT action. Archive files are **mined** through the XINA Structs Mine task into XINA databases in order to be viewed in the XINA client, and are used to generate export packages.

Alternatively, an origin may use **buffer source files**. Buffer files may be imported in a variety of data formats and are not subject to the same strict requirements as archive files. These may are imported directly with the STRUCT BUFFER IMPORT action. Mnemonic data from buffer files is loaded into a temporary buffer database for immediate viewing in the XINA client. Buffer files are **archived** (merged and converted into archive files) through the XINA Structs Archive task, which can be run manually or configured to run in regular intervals. *This is the recommended approach for importing mnemonic data when getting started with XINA Structs.*

Data Flow

In general, there are three supported approaches for origin data flow: **buffer** import, **variable time archive** import, and **fixed time archive** import. While a single origin can only support one workflow, a model may combine multiple workflows using multiple origins.

Buffer Import

The buffer import workflow is the most flexible mnemonic import method. Buffer files do not need to adhere to strict requirements (aside from conforming to standard accepted file formats). Buffer files for a given origin may have duplicated data, overlapping data, and can introduce new mnemonic definitions on demand.

Buffer files are imported with the STRUCT BUFFER IMPORT action. This invokes three effects:

- the raw buffer file is parsed, validated, and stored in the model origin **mnemonic buffer file database**
- new **mnemonic definitions** are created for any unrecognized mnemonic labels

- data is added to the **mnemonic buffer database** for the associated origin

No additional data processing occurs as part of this step. XINA models utilizing buffer source files must implement routine execution of the `STRUCT_BUFFER_ARCHIVE` asynchronous task (typically every hour) to merge the files into archive files in a fixed-time archive format, which can then be processed by `STRUCT_ARCHIVE_MINE` tasks to fully process data into model standard databases.

Pros

- minimal client side configuration required to get started
- allows smaller, faster file uploads to view data close to real-time
- flexible and responsive to changing environments, mnemonics, requirements

Cons

- performance is worse than client side aggregation
- not recommended above 1k total data points per second

Struct Archive Task

The XINA Struct Archive task merges and compresses buffer files into archive files. This step is required to resolve any data discrepancies and ensure data is preserved in accordance with the requirements of archive files. The task performs the following steps:

- load all unprocessed files from the buffer file database
- for each time ranges affected by unprocessed files
  - process each file into processed format
  - load any existing processed files in those time ranges
  - merge data from all processed files for time range into single archive file
  - upload newly processed buffer files
  - delete unprocessed buffer files
  - upload merged archive file
  - run mining task on merged archive file
  - delete any mnemonic data already present for time range
  - import mnemonic data generated by mining task

Direct Archive Import

Archive files are imported directly with the STRUCT ARCHIVE IMPORT action.

Pros

- much higher performance ceiling than server side aggregation
- stringent validation ensures data conforms to standard

Cons

- more complex initial setup
- mnemonic definitions must be pre-defined and cannot be added on-the-fly
- mnemonic definitions need coordination between client and server
- changes are more complex and likely involve human interaction

Fixed-Time Archive Import

With fixed-time archive import each archive has a fixed time range. This is a recommended solution for projects which generate a persistent data stream (for example, data sources piped through a FEDS server).

Variable-Time Archive Import

With variable-time archive import each archive specifies a custom time range. This is a recommended solution for projects which generate their own archival equivalent (for example, outputting a discrete data set after running a script). Because the time ranges are determined by the source data, it is recommended to generate interval events matching each file as a time range reference.

Source File Formats

Currently there are two natively supported general purpose formats, one using the codes `csv` / `tsv` (full documentation here), and a binary format using the code `xbin` (full documentation here) Additional formats will be added in the future, and custom project-specific formats may be added as needed.

Assumptions and Limitations

Each archive source file is considered the **single source of truth for all mnemonics, instants, and intervals for it's associated origin for its time range**. This has the following implications:

Archive files with the same origin cannot contain overlapping time ranges. If an import operation is performed with a file violating this constraint the operation will fail and return an error.

Within a single model, each mnemonic may only come from a single origin. Because mnemonics are not necessarily strictly associated with models, and the source may vary between models, this cannot be verified on import and must be verified on the client prior to importing data.

Structs CSV / TSV Format Reference

The XINA Structs CSV / TSV formats provide a standard delimited text file format for mnemonic data.

Source File Format

Files must be either ASCII or UTF-8 encoded. New lines will be interpretted from either `\n` or `\r\n`. The `conf` object may define other customization of the format:

Conf Definition

Key	Value	Default	Description
delimiter	string	auto detect (`','`, `'\t'`, `';'`)	value delimiter
quote_char	character	`"` (double quote character)	value quote character
ignore_lines	number	`0`	lines to ignore after UUID and before header
mode	`"row"` or `"col"`	auto-detect	mnemonic mode (see below)
t	`"auto"`, `"iso8601"`, `"s"`, `"ms"`, or `"us"`	`"auto"`	time format (see below)
zone	string	time zone to use if not provided	

The first line must contain an appropriately generated 128-bit UUID in the standard 36 character format.

If the `mode` property is `"row"`, the file must contain three columns:

Name	Description	Alternate Names
t	Unix time or ISO8601 zoned timestamp	time, timestamp
mn	mnemonic name or ID	mnemonic, n, name
v	value (numeric, empty, or `null`)	val, value

The header is used to determine the order of the columns.

For example (whitespace added for clarity, not required):

```
123e4567-e89b-12d3-a456-426614174000
t , mn    , v
0 , v_mon , 1
0 , i_mon , 5
1 , t_mon , 100
```

```
2 , v_mon , 1.1
2 , i_mon , 4
3 , t_mon ,
4 , v_mon , 1.2
4 , i_mon , 3
5 , t_mon , 101
```

If `mode` is `"col"`, the file must first contain a time column, followed by a column for each mnemonic. The column headers must specify the mnemonic name or ID for each column. Unlike `row`, `null` values must be spelled out explicitly, as empty values will **not** create a point in the database.

For example, the following is equivalent to the above example (whitespace added for clarity, not required):

```
123e4567-e89b-12d3-a456-426614174000
t      , v_mon , i_mon , t_mon
0      , 1     , 5     ,
1      ,       ,       , 100
2      , 1.1   , 4     ,
3      ,       ,       , null
4      , 1.2   , 3     ,
5      ,       ,       , 101
```

If the `mode` property is not specified, the mode will be determined by the number of columns in the file. If there are exactly 3 columns with names matching the required columns for the `"row"` mode, that mode is used; otherwise the file is assumed to use the column mode.

Time Parsing

The mode of time processing is determined by the value for `t` in `conf`. The `auto` mode attempts to interpret the most likely formatting for the timestamp. If the value is an integer or floating point format, it will be interpretted as a Unix timestamp, with precision based on these rules:

- t > `1e16`: error, value above typical range
- t > `1e14`: microseconds
- t > `1e11`: milliseconds
- t > `1e8`: seconds
- t <= `1e8`: error, value below typical range

Otherwise it will be interpretted as a zoned ISO8601 timestamp. If `t` is set explicitly in the configuration the time will always be interpretted in that context. The ISO timestamp may use the standard format: `2023-05-31T17:55:07.000` or condensed `20230531T175507.000`. If the `zone` property provided in the configuration, the timestamps do not require a zone. Otherwise they must include an explicit zone.

XBin Format Reference

The XBin (XINA Binary) format provides a XINA standard binary format for time based data files. It uses the file extension `xbin`.

The xbin format organizes **key-value** data by **time**. The data content is a series of **rows** in ascending time order, with each row having a single microsecond precision Unix time, unique within the file.

Segment Format

XBin data is often encoded in **segments**, which are defined by an initial 1, 2, or 4 byte unsigned integer length, then that number of bytes. These are referred to in this document as:

- **seg1** (up to 255 bytes)
- **seg2** (up to 65,535 bytes)
- **seg4** (up to 2,147,483,647 bytes)

If the length value of a segment is zero there is no following data and the value is considered **empty**.

Examples

The string `"foo"` has a 3 byte UTF-8 encoding: `0x66`, `0x6f`, `0x6f`.

As a seg1, this is encoded with a total of 4 bytes (the initial byte containing the length, 3):

`0x03` `0x66` `0x6f` `0x6f`

As a seg2, 5 bytes:

`0x00` `0x03` `0x66` `0x6f` `0x6f`

And as a seg4, 7 bytes:

`0x00` `0x00` `0x00` `0x03` `0x66` `0x6f` `0x6f`

Value Format

Each value starts with a 1 byte unsigned integer indicating the value type, followed by additional byte(s) containing the value itself, as applicable.

Value Type Definition

Code	Value	Length (bytes)	Description
`0`	`null`	0	literal `null` / empty string
`1`	ref dict index	1	index 0 to 255 (see below)
`2`	ref dict index	2	index 256 to 65,535

Code	Value	Length (bytes)	Description
3	ref dict index	4	index 65,536 to 2,147,483,647
4	true	0	boolean literal
5	false	0	boolean literal
6	int1	1	1 byte signed integer
7	int2	2	2 byte signed integer
8	int4	4	4 byte signed integer
9	int8	8	8 byte signed integer
10	float4	4	4 byte floating point
11	float8	8	8 byte floating point
12	string1	variable	seg1 UTF-8 encoded string
13	string2	variable	seg2 UTF-8 encoded string
14	string4	variable	seg4 UTF-8 encoded string
15	json1	variable	seg1 UTF-8 encoded JSON
16	json2	variable	seg2 UTF-8 encoded JSON
17	json4	variable	seg4 UTF-8 encoded JSON
18	jsonarray1	variable	seg1 UTF-8 encoded JSON array
19	jsonarray2	variable	seg2 UTF-8 encoded JSON array
20	jsonarray4	variable	seg4 UTF-8 encoded JSON array
21	jsonobject1	variable	seg1 UTF-8 encoded JSON object
22	jsonobject2	variable	seg2 UTF-8 encoded JSON object
23	jsonobject4	variable	seg4 UTF-8 encoded JSON object
24	bytes1	variable	seg1 raw byte array
25	bytes2	variable	seg2 raw byte array
26	bytes4	variable	seg4 raw byte array
27	xstring1	variable	seg1 xstring
28	xstring2	variable	seg2 xstring
29	xstring4	variable	seg4 xstring
30	xjsonarray1	variable	seg1 xjson array
31	xjsonarray2	variable	seg2 xjson array
32	xjsonarray4	variable	seg4 xjson array
33	xjsonobject1	variable	seg1 xjson object
34	xjsonobject2	variable	seg2 xjson object

Code	Value	Length (bytes)	Description
`35`	xjsonobject4	variable	seg4 xjson object
`36` - `255`	unusued, reserved		

XString Format

The **xstring** value type allows chaining mutliple encoded values to be interpretted as a string. The xstring segment length must be the total number of bytes of all encoded values in the string.

Note that although any data type may be included in an xstring, the exact string representation of certain values may vary depending on the decoding environment (specifically, the formatting of floating point values) and thus it is not recommended to include them in xstring values. JSON values will be converted to their minimal string representation. Byte arrays will be converted to a hex string. Null values will be treated as an empty string.

XJSON Array Format

The **xjsonarray** value type allows chaining mutliple encoded values to be interpretted as a JSON array. The xjsonarray segment length must be the total number of bytes of all encoded values in the array.

XJSON Object Format

The **xjsonobject** value type allows chaining mutliple encoded values to be interpretted as a JSON object. Each pair of values in the list is interpretted as a key-value pair. The xjsonobject segment length must be the total number of bytes of all encoded key-value pairs in the object. Note that key values must resolve to a string, xstring, number, boolean, or null (which will be interpretted as an empty string key).

Examples

Null Value:

Code	Content (0 bytes)
`0x00`	

300 (as 2 byte integer):

Code	Content (2 bytes)
`0x07`	`0x01` `0x2c`

0.24 (as 8 byte float):

Code	Content (8 bytes)
`0x0b`	`0x3f` `0xce` `0xb8` `0x51` `0xEB` `0x85` `0x1E` `0xb8`

"foo" (as string1):

Code	Content (4 bytes)
`0x0c`	`0x03` `0x66` `0x6f` `0x6f`

{"foo":"bar"} (as json1):

Code	Content (14 bytes)
`0x0f`	`0x0d` `0x7b` `0x22` `0x66` `0x6f` `0x6f` `0x22` `0x3a` `0x22` `0x62` `0x61` `0x72` `0x22` `0x7d`

"foo123" (as xstring1, split as string1 "foo" and int1 123):

Code	Content (7 bytes)
`0x1b`	[`0x06`](total length) [`0x03` `0x66` `0x6f` `0x6f`]("foo") [`0x04` `0x7b`](123)

Reference Dictionary

The xbin format provides user-managed compression through the reference dictionary. It can contain up to the 4 byte signed integer index space (2,147,483,647). The order of values affects the compression ratio; index 0-255 can be represented with a single byte, 256-65,535 with 2 bytes, and above requires 4 bytes.

Binary File Format

UUID

The file starts with a 16 byte binary encoded UUID. This is intended to uniquely identify the file, but the exact implementation and usage beyond this is not explicitly defined as part of the format definition. For XINA purposes two xbin files with the same UUID would be expected to be identical.

Header

A value which must either be `null` or a `jsonobject1`, `jsonobject2`, or `jsonobject4`. This is currently a placeholder with no defined parameters.

Reference Dict

A seg4 containing 0 to 2,147,483,647 encoded values, which may be referenced by zero based index with the reference dict index value types.

Rows

Each row contains:

- 8 byte signed integer containing Unix time with microsecond precision
- seg4 of row data, containing
  - header, single value which must either be `null` or a `jsonobject1`, `jsonobject2`, or `jsonobject4`
  - one or more key,value pairs

The row header is currently a placeholder with no defined parameters.

Example File

Given a data set with UUID 9462ef87-f232-4694-922c-12b93c95e27c:

t	voltage	current	label
0	5	10	"foo"
1			"bar"
2	5	null	

A corresponding xbin file containing the same data would be:

UUID (16 bytes)

`0x94` `0x62` `0xef` `0x87` `0xf2` `0x32` `0x46` `0x94` `0x92` `0x2c` `0x12` `0xb9` `0x3c` `0x95` `0xe2` `0x7c`

Header (1 byte)

`0x00` (null, 1 byte)

Reference Dict, three values, "voltage", "current", "label" (29 bytes)

`0x00` `0x00` `0x00` `0x19` (seg4 length, 25)

`0x0a` `0x07` `0x76` `0x6f` `0x6c` `0x74` `0x61` `0x67` `0x65` ("voltage", 9 bytes)

`0x0a` `0x07` `0x63` `0x75` `0x72` `0x72` `0x65` `0x6e` `0x74` ("current", 9 bytes)

`0x0a` `0x05` `0x6c` `0x61` `0x62` `0x65` `0x6c` ("label", 7 bytes)

Row t0 (22 bytes)

`0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x00` (time, 0, 8 bytes)

`0x00` `0x00` `0x00` `0x0e` (row length, 15, 4 bytes)

`0x00` (header, null, 1 byte)

`0x01` `0x00` (reference to index 0, "voltage", 2 bytes)

`0xff` (type code reference to index 0, 5, 1 byte)

`0x01` `0x01` (reference to index 1, "current", 2 bytes)

`0x04` `0x0a` (integer value 10, 2 bytes)

`0x01` `0x02` (reference to index 2, "label", 2 bytes)

`0x0a` `0x03` `0x66` `0x6f` `0x6f` (string "foo", 5 bytes)

Row t1 (20 bytes)

`0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x01` (time, 1, 8 bytes)

`0x00` `0x00` `0x00` `0x08` (row length, 8, 4 bytes)

`0x00` (header, null, 1 byte)

`0x01` `0x02` (reference to index 2, "label", 2 bytes)

`0x0a` `0x03` `0x62` `0x61` `0x72` (string "bar", 5 bytes)

Row t2 (19 bytes)

`0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x00` `0x02` (time, 2, 8 bytes)

`0x00` `0x00` `0x00` `0x0e` (row length, 15, 4 bytes)

`0x00` (header, null, 1 byte)

`0x01` `0x00` (reference to index 0, "voltage", 2 bytes)

`0x00` (type code reference to index 0, 5, 1 byte)

`0x01` `0x01` (reference to index 1, "current", 2 bytes)

`0x00` (null, 1 byte)

Struct Definitions Reference

Groups

Project

Top level struct group. All struct groups and databases must be decendants of a project to be recognized. Name and label are customizable.

Parameter	Value
type	project
version	1.0.0

Category

Mid-level struct group for organization. Must be child of project or category. Name and label are customizable.

Parameter	Value
type	category
version	1.0.0

Model

Group for which all data is locally co-relevant. Must be child of either project or category. Name and label are customizable.

Parameter	Value
type	model
version	1.0.0

Origin

Group for all data from a single data origin. Must be the child of a model. Name and label are customizable.

Parameter	Value
type	origin
version	1.0.0

Definitions

Group containing definitions databases.

Task

Mnemonic

Mnemonic Bin

Databases

Definitions

Event Def

Mnemonic Def

Holds mnemonic definitions, specifying how they are displayed, interpretted and processed. Must be direct child of definitions group:

`<project>.def.mn` or `...<category>.def.mn` or `...<model>.def.mn`

Parameter	Value
type	def_mn
version	1.0.0
name	mn
label	Mnemonic

Fields

Name	Type	Req	Description
mn_id	`int(4)`	?	unique mnemonic ID
name	`utf8vstring(128)`	?	unique mnemonic name
desc	`utf8text`		plain text mnemonic description
unit	`utf8vstring(32)`		measurement unit (for example, `"V"`, `"mA"`)
state	`struct_mn_state`	?	current state of mnemonic
origins	`jsonobject`	?	map of model(s) to associated origin(s)
full	`asciivstring(32)`		the primary database for the mnemonic, default `f8`
bin	`set(asciivstring(32))`		the opt-in bin database(s) to include the mnemonic in
format	`asciivstring(32)`		printf-style format to render values

Name	Type	Req	Description
enums	`jsonobject`		mapping of permitted text values to numeric values
labels	`list(jsonobject)`		mapping of numeric values or ranges to labels
aliases	`set(asciivstring(128))`		set of additional names associated with the mnemonic
meta	`jsonobject`		additional metadata as needed
query	`asciivstring(32)`		query name for meta-mnemonics
conf	`jsonobject`		configuration for meta-mnemonics

Changelog

1.0.0

- `enum` changed to `enums` since "enum" is often a reserved keyword
- `meas` field removed (measure now assumed from `unit`)

Nominal Def

Plot Def

Profile Def

Events

Event databases come in three forms, simple events, single file per event, and multiple files per event.

Event

Each record is a single event. May be a direct child of either a model or origin:

`...<model>.event` or `...<origin>.event`

Parameter	Value
type	event
version	1.0.1
name	event
label	Event

Fields

*Note that **virtual** fields are calculated from other fields and cannot be populated manually.*

Name	Type	Req	Description

uuid	`uuid`	?	event UUID
e_id	`int(8)`	?	event ID (default to `0` if not provided)
t_start	`instant(us)`	?	start time
t_end	`instant(us)`		end time (if `null`, event is an open interval)
dur	`duration(us)`	?	**virtual** duration in microseconds (`null` if open)
interval	`boolean`	?	**virtual** `t_start` != `t_end`
open	`boolean`	?	**virtual** `t_end` is `null`
type	`struct_event_type`	?	event type (default to `message` if not provided)
level	`struct_event_level`	?	event level (default to `none` if not provided)
name	`utf8vstring(128)`		event name (if associated with event definition)
label	`utf8vstring(128)`	?	plain text label
content	`utf8text`		extended event content
meta	`jsonobject`		additional metadata as needed
conf	`jsonobject`		configuration for specific event types

Changelog

1.0.1

- corrected `name` as not required

1.0.0

- `pid` (primary ID) changed to `e_id` (event ID) to avoid confusion
- `sid` removed (additional IDs may be added as needed)
- `int` changed to `interval` (`int` is commonly reserved keyword)
- `dur`, `interval`, and `open` are now derived fields from `t_start` and `t_end`
- added `struct_event_type` and `struct_event_level` data types
- added `name` as event definition association

Event File

Uses same structure as event database, with one additional field.

Name	Type	Req	Description
file_name	`utf8filename`	?	safe file name

Event Files

Mnemonics

Mn Full

Mn Buffer

Mn Delta

Mn Bin Time

Mn Bin Interval

Mn File Archive

Contains all mnemonic archive files for an origin. Parent must be an origin group:

`...<origin>.archive`

Parameter	Value
type	archive
version	1.0.0
name	archive
label	Archive

Fields

Name	Type	Req	Description
uuid	`uuid`	?	file UUID
t_start	`instant(us)`	?	start time
t_end	`instant(us)`	?	end time
dur	`duration(us)`	?	**virtual** duration in microseconds
t_min	`instant(us)`	?	time of first data in file
t_max	`instant(us)`	?	time of last data in file
file_name	`utf8filename`	?	archive file name
format	`asciivstring(32)`		file format (default `"xbin"`)
meta	`jsonobject`		additional metadata as needed
conf	`jsonobject`		configuration for format as needed

Mn File Buffer

Contains all mnemonic buffer files for an origin. Parent must be an origin group:

`...<origin>.buffer`

Parameter	Value
type	archive
version	1.0.0
name	archive
label	Archive

Fields

Name	Type	Req	Description
uuid	`uuid`	?	file UUID
file_name	`utf8filename`	?	buffer file name
t_min	`instant(us)`	?	time of first data in file
t_max	`instant(us)`	?	time of last data in file
dur	`duration(us)`	?	**virtual** duration in microseconds
state	`struct_buffer_state`	?	buffer file state
flag	`struct_buffer_flag`		buffer file flag
format	`asciivstring(32)`		buffer file format (default `"csv"`)
conf	`jsonobject`		configuration for format as needed

The state field may be one of four values:

- `PENDING` - the file data is present in the mnemonic buffer database but has not been processed further
- `PROCESSED` - the file has been converted into a standard xbin file format
- `ARCHIVED` - the file contents have been distributed to the appropriate archive file(s)
- `DEPRECATED` - the file is preserved but no longer included in archive files

The flag field may be one of two values:

- `DEPRECATE` - the file is queued for deprecation
- `DELETE` - the file is queued for deletion

Tasks

Archive Task

Mine Task

Spectra

The spectra definition is a property for event databases.

Property	Value	Req	Description
tabs	array of tab conf(s)		custom tabs for UI
presearch	array of presearch confs		custom pre-search components for UI
filters	array of filter confs		
grouping	array of field name(s)		
charts	charts conf	?	
tables	array of table conf		
query	query conf		
labels	labels conf		

Spectra Tab Conf

Configuration for a spectra search tab. This may be a `string`, referencing the name of a custom tab implementation, or an object with a `"type"` property specifying a tab type and additional properties applicable for that type. Currently there are no custom tab types, but they may be added in the future.

Spectra Database Tab

Under Construction

The database tab employs a record search for a separate target database of any type, and a solution for converting a selection from the target database to the spectra database.

Property	Value	Req	Description
type	`"database"`	?	tab type name
database	database specifier	?	target database specifier
map	see below	?	solution to map target selection to spectra selection

The `"map"` property may be a `string`, `array` of `strings`, or `object`.

If a `string`, the value must be the name of a custom selection function (none currently exist, they may be added in the future).

Spectra Presearch Conf

Specifies a set of components to display before the main spectra search component.

Spectra Field Presearch

Specifies a standalone component to search a particular field.

Property	Value	Req	Description
type	`"field"`	?	presearch type name
field	field specifier	?	

Property	Value	Req	Description
options	see below		options for search dropdown

Spectra Filters Conf

Specifies filters / badges for spectra search.

Property	Value	Req	Description
name	`string`	?	system name for filter
label	`string`		display label (uses name if absent)
badge	`string`		badge label (uses name if absent)
desc	`string`		description for badge / filter tooltip
color	`string`		color code or CSS class
e	`expression`	?	expression to apply for filter

Spectra Charts Conf

Specifies options for each spectra chart.

Property	Value	Req	Description
summary	spectra chart conf	?	summary chart conf
spectra	spectra chart conf	?	spectra chart conf

Spectra Chart Conf

Specifies options for a single spectra chart.

Property	Value	Req	Description
x	`string[]`	?	x axis options
y	`string[]`	?	y axis options
tooltip	`string`		record format string

Spectra Tables Conf

Under Construction

Spectra Query Conf

Under Construction

Spectra Labels Conf

Labels are specified as an `object` mapping standard label values to custom values. These will be defined as needed.

Units Reference

WIP: Struct Extract Interface

For projects that use telemetry data files (files of packets), XINA Mining and Export functionality delegates the decoding and conversion of mnemonic data to mission specific tools. These mission specific tools should implement the defined interface to work seamlessly with XINA.

Input Config

TODO:

- tm.meta
- Verify that filter state can be computed from the CVT

```
{
  file_path: <file_path>,
  meta_path: <meta_file_path>,
  out: <the dir where files should be output to>,
  cvt_path: <path to cvt file>,
  filter_path: <path to filter definitions file>,
  model: <the mission's model>,
  timeslice_id: <int>, // needed?
  time_source: <pkt_or_grt>,
  raw: [],
  eng: [],
  sci: [],
}
```

Output

The output of the mission specific tool should be a xbin file, which XINA's tools will then process to generate the mining and export products.