

Structured Data Standards

XINA standard data structures terms and organizing principles.

- [Introduction](#)
- [Data Organization](#)
- [Data Lifecycle](#)
- [Mnemonics](#)
- [Events](#)
- [Structs Data Format](#)
- [Structs DSV Format](#)
- [Structs Ids List Format](#)
- [XBin Format](#)
- [Struct Extract Interface](#)
- [Struct Definitions Reference](#)
- [Name Conventions Reference](#)
- [Units Reference](#)

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. 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**.

These are not hard limitations of the overall XINA system, but serve as our recommended entry point into using XINA based on past experience, performance benchmarks, and cost/benefit analysis.

Data Organization

Structs XINA groups employ certain organizational requirements to ensure they are interpreted correctly by structs API calls and front end tools.

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 import data in parallel.

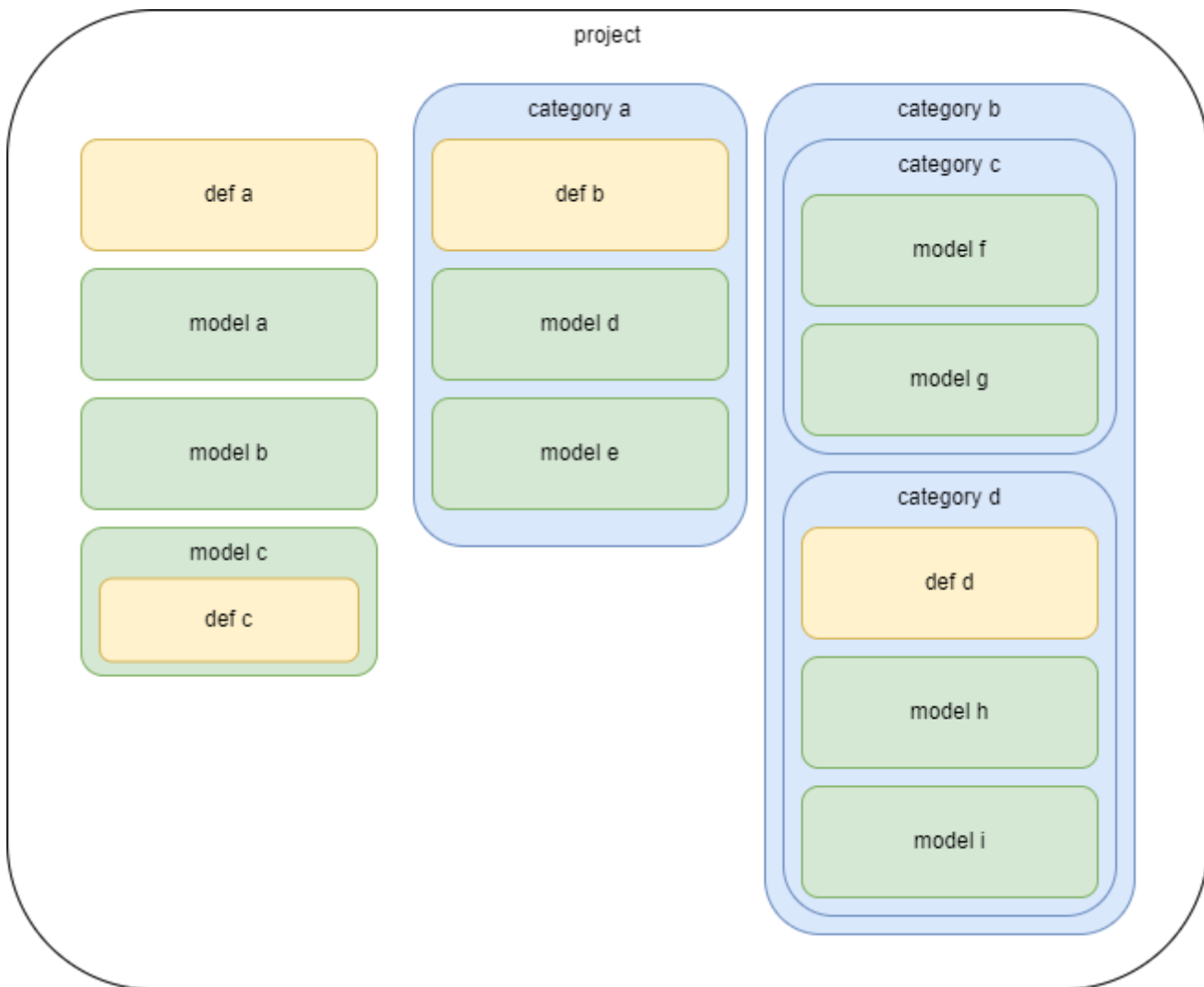
Broadly we use **time** as the primary method to organize and synchronize data within a model. In XINA this is represented as an 8-byte unsigned integer Unix time with microsecond precision. We use Unix time because it is:

- Widely and consistently supported
- Time zone independent
- Efficiently convertible to other formats and time systems

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

Projects / Categories

Projects and categories provide organization for multiple models. A project is a top-level group containing multiple models and/or categories, with each category containing multiple models or further categories. Project and category groups may also include additional groups and databases of data or resources which are not model specific, such as notebooks 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:



Note the definitions groups, which provide context for the data in models. A model is associated with the definitions defined by its closest ancestor. In this case, "definitions a" apply to models a, b, f, and g, "definitions b" to models d and e, "definitions c" to model c, and "definitions d" to models h and i.

In practice it is strongly recommended to use a single definitions group as broadly as possible to facilitate comparing data. Tools are designed to efficiently compare data (even across models) with the same set of definitions.

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 **pipes** (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 needed depending on project requirements, data volumes
- **User Metadata**
 - additional data added by users, often directly through the XINA interface
 - XINA likely the primary repository for this data
 - for example, a notebook
- **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

Pipes

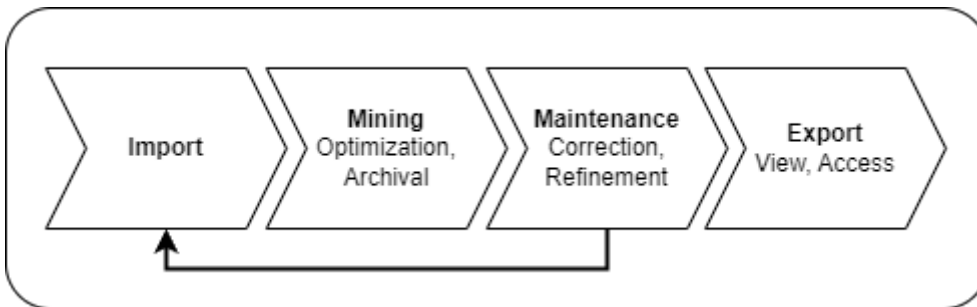
Abstractly, a **data pipe** (or simply **pipe**) is a single point of data import to a model. In many cases, a model will only have a single pipe; for example, if all data is provided directly from a single instrument, or multiple components are merged into a single data stream through FEDS before import into XINA.

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:

Data Lifecycle

The XINA structs mnemonic data lifecycle involves four primary phases:



Source Files

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

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 pipe**. 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 pipe 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 pipe data flow: **buffer** import, **variable time archive** import, and **fixed time archive** import. While a single pipe can only support one workflow, a model may combine multiple workflows using multiple pipe.

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 pipe 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 pipe **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 pipe

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 pipe for its time range**. This has the following implications:

Archive files with the same pipe 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 pipe. 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.

Mnemonics

A **mnemonic** defines a single field of **numeric** data in a XINA model. A **datapoint** is a single record of a mnemonic, 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 are defined in a **mnemonic definitions** database. It is **strongly recommended** to use a single definitions database for an entire project to facilitate 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 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.

Fields

Mnemonic ID

Unique numeric ID, assigned by XINA.

Name

Mnemonic name. Conforms to [structs naming conventions](#). Must be unique in combination with subname and unit.

Subname

Essentially a name suffix.

Description

Optional plain text mnemonic description.

Unit

Optional (but strongly recommended) measurement unit (for example, `V`, `mA`, etc).

Standard Fields

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

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.

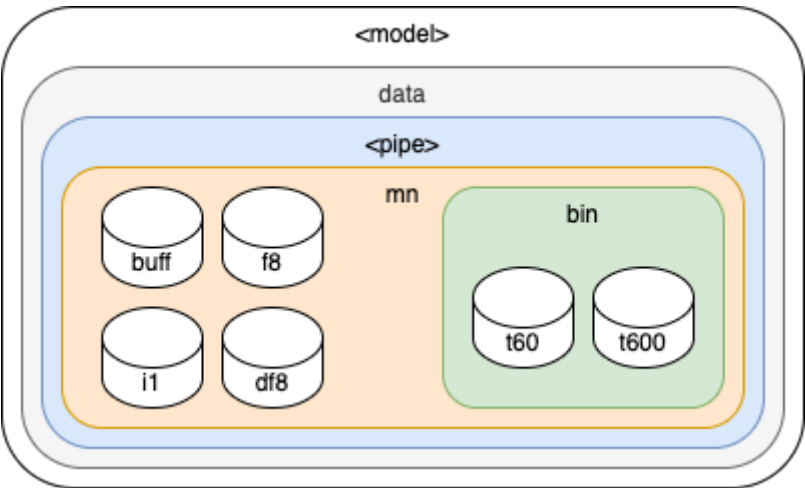
If `enum` is provided a mnemonic will apply labels to enumerated numeric values, as provided in `values` . For example, a 0|1 on|off state could be represented by `{"0":"OFF", "1":"ON"}` . Values in this map may also be used to parse imported data.

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.

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 mnemonic.

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 |
| 2 | 0 |
| 3 | 1 |
| 4 | 1 |

| t | v |
|---|---|
| 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 interpreted 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 |
| 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 |

| field | type | description | required |
|-------|----------|-------------------------------------|----------|
| 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

Events are the primary means of organizing structs data. They have 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.

Fields

Unlike most structs databases, event databases may include as many custom fields as required, so long as they do not conflict with the required standard fields:

UEID

Universally unique event identifier (UUID). Intended to permanently, globally specify each event. Should be generated at the creation of the event to ensure consistency even if data is reprocessed.

Event ID

Optional numeric reference to an [event definition](#) (also see below). If not provided, defaults to .

Type

Indicates how the event should be viewed and interpreted. The options are defined by XINA.

Level

Indicates how the event should be viewed and interpreted. The options are defined by XINA.

Label

Required plain text description of the event. Limited to to 128 bytes for indexing.

Content

Optional plain text, HTML, or JSON of unlimited length.

Meta

Optional JSON object of arbitrary additional content.

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, ID optional |
| 1 | marker | ? | ? | Organized event, ID required |
| 2 | alert | ? | ? | Organized event, ID required, level (severity) required |
| 2000 | test | | ? | Discrete test period, may not overlap other tests, ID optional |
| 2001 | activity | | ? | Discrete activity period, may not overlap other activities, ID optional |
| 2002 | phase | | ? | Discrete phase period, may not overlap other phases, ID optional |
| 2010 | filter | | ? | Filter state |
| 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 |
| 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 |

Definitions

As with mnemonics, events may be identified with [event definitions](#). However, unlike mnemonics, not every event requires a definition. The event ID field associates an event with an event definition. Each event ID is associated with a unique name, describing the definition. These work similarly to mnemonic names for purposes for definition creation. If an event is inserted with an unrecognized name, a new definition will be created for that name and assigned a new event ID.

Context

An event database may either be a child of a model or pipe group. A model event database is essentially like any other XINA database, but with support for the **STRUCT EVENT** API actions.

Pipe event databases are more restrictive. The events must be embedded in the data set of the pipe, and cannot be inserted manually. Each event is associated with the archive in which it starts. Events may cross archive boundaries by initially being inserted as `open` events (having a start time but no end time) and later using the `$event.close` operation to assign an end time. Additionally, pipe event databases have an associated event change database to track manually applied updates outside of the source data set. This allows changes to be preserved if data is remined from the archive files. The intention is that these accurately reflect the archived data, so fields must opt-in to being editable by this operation.

Data Formats

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 interpreted from either `\n` or `\r\n`. The `conf` object may define other customization of the format:

Conf Definition

| Key | Value | Default | Description |
|--------------------------|--|---|--|
| <code>delimiter</code> | <code>string</code> | auto detect (<code>'</code> , <code>'\t'</code> , <code>','</code>) | value delimiter |
| <code>quoteChar</code> | <code>character</code> | <code>"</code> (double quote character) | value quote character |
| <code>ignoreLines</code> | <code>number</code> | <code>0</code> | number of lines to skip before the header |
| <code>invalid</code> | <code>null</code> , <code>'NaN'</code> , <code>number</code> | <code>null</code> | preferred interpretation of invalid literal |
| <code>nan</code> | <code>null</code> , <code>'NaN'</code> , <code>number</code> | <code>null</code> | preferred interpretation of <code>'Nan'</code> literal |
| <code>pInfinity</code> | <code>null</code> , <code>'Inf'</code> , <code>number</code> | <code>null</code> | preferred interpretation of positive <code>'Infinity'</code> literal |
| <code>nInfinity</code> | <code>null</code> , <code>'Inf'</code> , <code>number</code> | <code>null</code> | preferred interpretation of negative <code>'Infinity'</code> literal |
| <code>utc</code> | <code>boolean</code> | <code>false</code> | if <code>true</code> , 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 |
|---------------------|--|
| <code>ts</code> | text timestamp, interpreted in local browser timezone (absent explicit zone) |
| <code>ts_utc</code> | text timestamp, interpreted as UTC timezone (absent explicit zone) |
| <code>unix_s</code> | Unix time in seconds |

| Unit | Description |
|---------|---------------------------|
| unix_ms | Unix time in milliseconds |
| unix_us | Unix time in microseconds |

Structs Data Format

Structs data files can contain mnemonic and/or event data. They have two basic forms, **archive** files, which must use the [XBin format](#), and **buffer** files, which may either use the [structs DSV format](#) or the XBin format. In either format, they essentially provide a set of time-key-value mappings. By default keys are interpreted as mnemonics, unless they start with the dollar sign (\$) character, in which case they are interpreted according to the rules of this document. Currently this is only used to embed event data, but additional features may be added in the future.

Note that buffer files are only intended to be imported with the [STRUCT BUFFER IMPORT](#) API action.

Mnemonics

Mnemonic data may either be denoted by the numeric [mnemonic ID](#), or text [mnemonic name](#). The value will be interpreted as a mnemonic ID if it is a [numeric XBin type](#), or if it is a string containing only digit characters. If the ID is used, it must already be associated with an existing [mnemonic definition](#), or an error will be thrown. Otherwise the text will be parsed as follows:

```
mnemonic  = name [ ';' subname ] [ ( ':' unit-enums ) | ( '(' unit-enums ')' ) ] [ '#' description ]
unit-enums = unit [ ';' enums ]
enums      = enum | ( enums [ '|' enum ] )
enum       = [ integer '=' ] label
```

The parsed mnemonic has five elements: the **name**, **subname**, **unit**, **enum map**, and **description**. Only the name is required. The combination of name, subname, and unit are used to lookup an existing mnemonic. The comparison is case insensitive and any whitespace is treated as a single underscore. If a matching mnemonic is not found, a new mnemonic is created automatically. Note that the overall name cannot contain the colon (:), semicolon (;), dollar sign (\$), or number sign (#) characters.

The enum map specifies a one-to-one map of integer values to text labels. This is optional and unlike the other parameters, not interpreted as a key component of the mnemonic. They will be included in the generated mnemonic definition if it doesn't exist.

Events

Event operations may also be embedded in structs data files. These are indicated by keys which start with `$event`. The supported operations are inserting instant events, opening interval events, and closing interval events. Events may specify a textual `"name"` instead of `"e_id"`, which can be used to lookup a corresponding event definition ID, or create a new event definition if one does not exist with the provided name. From a user's perspective, a `name` can be easier to remember and more descriptive than a numeric `e_id`. The event's `a_id_start` (Start Archive ID) and `a_id_end` (End Archive ID) will be automatically populated with the Archive ID of the file being processed.

For the purposes of the following examples, assume a model "m" contains a pipe "p" with 3 event databases, "e", "ef" (single file per event), and "efs" (multi-file per event).

See also the [Events](#) and [Event Definitions](#) references for the supported fields.

Insert Event Operation

```
$event.insert.<database name>
```

Creates a single [event](#) which may be an instant or a completed interval. The database name must correspond to an event database in the associated pipe. The value must be a JSON object in the standard [record format](#), with fields appropriate to the specified [Events database](#). The event must omit both the `"t_start"` and `"t_end"` values which will be populated by the time value from the corresponding row in the buffer file. The `a_id_start` and `a_id_end` will be populated from the file's `a_id`.

Open Interval Event Operation

```
$event.open.<database name>
```

Creates a single open event (interval with a start time and no end time). Uses the key `$event.open.<database name>`. The value must be a single JSON object, defining the content of the event record. The event must omit both the `"t_start"` and `"t_end"` values (`"t_start"` is populated by the row time, and `"t_end"` is `null` for an open interval). The `a_id_start` will be populated from the file's `a_id`.

Close Interval Event Operation

```
$event.close.<database name>
```

Closes an existing open event. The value must be a single JSON object, which may be used to update the values of any fields of the event. The object must omit both the `"t_start"` and `"t_end"` values (`"t_start"` cannot be edited, and `"t_end"` is populated by the row time). The `a_id_end` will be populated from the file's `a_id`.

Structs DSV Format

The XINA Structs DSV (delimiter separated values) format provides a standard delimited text data file format. This is recommended for data files attached to events, and forms the basis for the [structs buffer file format](#).

Files have certain standard requirements:

- Must be UTF-8 encoded
- New lines will be interpreted from either `\n` or `\r\n`
- Blank lines will be ignored
- Lines starting with the `#` character are treated as comments and ignored

The `conf` object may define other customization of the format:

| Key | Value | Default | Description |
|--------------|-------------|---|---|
| delimiter | string | <code>,</code> (comma character) | value delimiter |
| quote_char | character | <code>"</code> (double quote character) | value quote character |
| ignore_lines | number | <code>0</code> | lines to ignore at the start of the file |
| zone | string | UTC | time zone to use if not provided |
| values | JSON object | | preferred interpretation of string literals (see below) |

It is strongly recommended to include a unique [appropriately generated 128-bit UUID in the standard 36 character format](#) as a comment in the first processed line of each file. (If `ignore_lines > 0`, this would be the first line after that number of lines.)

There are two format modes for DSV files: the **row** format, in which each line contains a time, label, and value, and the **column** format, in which each row contains a time and one or more values. The first processed uncommented line will be interpreted as the column header, which is used to determine the file format. The file will be treated as row mode if it contains exactly three columns, with each having one of the reserved column names in the table below.

| Name | Description | Alternate Names |
|----------|---|---|
| t | Unix time or ISO8601 zoned timestamp | ts, time, timestamp, datetime, unix_time, unix, utc |
| k | key | key, m, m_id, mn, mn_id, mnemonic, mnemonic_id, n, name |
| v | value (numeric, empty, or <code>null</code>) | 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 , k      , 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 , null
4 , v_mon , 1.2
4 , i_mon , 3
5 , t_mon , 101
```

Otherwise, the file will be interpreted as the column format, where the first must be the time column, followed by a column for each mnemonic. The column headers must specify the mnemonic name or ID for each column.

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
```

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 interpreted 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 interpreted as a zoned ISO8601 timestamp. If `t` is set explicitly in the configuration the time will always be interpreted in that context. The ISO timestamp may use the standard format:

```
2023-05-31T17:55:07.000
```

Or condensed format:

20230531T175507.000

If the `zone` property provided in the configuration, the timestamps do not require a zone. Otherwise they must include an explicit zone.

Non-Numeric Values

Values which are non-numeric may be ignored, treated as `null`, or mapped to explicit values using the `values` property of the `conf` object. Ignored values are treated by XINA as though they do not exist in the file. `null` values are stored as an actual data point in the XINA database, but with the value `null` instead of a numeric value. (This is primarily useful to create a visual gap in plots.)

The following values are **ignored** by default (note, case-insensitive and whitespace agnostic):

- `""` (empty string)
- `"nv"` (no value, ITOS)
- `"na"`
- `"n/a"`

The following values are interpreted as `null` by default (note, case-insensitive and whitespace agnostic):

- `"null"`
- `"nil"`
- `"none"`
- `"nan"`
- `"inf"`
- `"+inf"`
- `"-inf"`
- `"infinity"`
- `"+infinity"`
- `"-infinity"`

Custom value interpretations may be specified in the `values` object as either `"ignore"`, `null`, or a numeric value. For example:

```
{
  "?": "ignore",
  "notta": null,
  "onetwothree": 123
}
```

Any text value which does not include a custom or default mapping will cause an error. The defaults may be extended in the future.

Structs Ids List Format

WORK IN PROGRESS

The Struct Ids List format is a comma delimited string format for storing references to mnemonics. It is used in various Definitions such as the [Profile Definition](#) to store which mnemonics should be included in the Export Package.

The string format should be parsed as:

- Comma delimited, whitespace ignored
- For each item in list:
 - unroll each instance of `#[]`
 - instance can only contain numeric values or numeric ranges
 - characters before `#` and after `[]` should be prefixed and appended to each unrolled item, respectively
- For each unrolled item:
 - if starts with `@`, treat as `ext_id`
 - if starts with a digit
 - if contains `-`, treat as a numeric range
 - else, treat as single `mn_id`
 - otherwise treat as `name[:subname]` (aka mn key)

A numeric range is shorthand syntax for a range of numbers. For example `3-6` should be expanded to `3,4,5,6`.

Example:

```
@#[1,2,3]sci,name;subname,@100sci,@#[1,5-6,13,15-17]raw,15,20-22,#[40,50-52],30,name2
```

Will be parsed into:

```
mn_ids: [ 15, 20, 21, 22, 40, 50, 51, 52, 30 ]
```

```
ext_ids: [ "1sci", "2sci", "3sci", "100sci", "1raw", "5raw", "6raw", "13raw", "15raw", "16raw", "17raw" ]
```

```
mn keys: [ "name;subname", "name2" ]
```

XBin Format

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) |

| Code | Value | Length (bytes) | Description |
|------|----------------|----------------|--------------------------------|
| 2 | ref dict index | 2 | index 256 to 65,535 |
| 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 |

| Code | Value | Length (bytes) | Description |
|----------|------------------|----------------|-------------------|
| 34 | xjsonobject2 | variable | seg2 xjson object |
| 35 | xjsonobject4 | variable | seg4 xjson object |
| 36 - 255 | unused, reserved | | |

XString Format

The **xstring** value type allows chaining mutiple encoded values to be interpreted 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 mutiple encoded values to be interpreted 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 mutiple encoded values to be interpreted as a JSON object. Each pair of values in the list is interpreted 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 interpreted 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 | 0x0d0x7b0x220x660x6f0x6f0x220x3a0x220x620x610x720x220x7d |

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

| Code | Content (7 bytes) |
|------|--|
| 0x1b | [0x06](total length) [0x030x660x6f0x6f]("foo") [0x040x7b](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 Extract Interface

Warning: This page is a Work In Progress

For projects that use packet file archives, XINA Mining ([struct_mining](#)) and Export ([struct_export](#)) delegates the decoding and conversion of mnemonic data to a mission specific tool, which we will reference as `struct_extract`. This tool should implement the following interface for integration with XINA.

Environment

- The app should be runnable on Ubuntu 22.04.
- Any required environment setup will be performed on a per project basis.

Input

`struct_extract` should accept a single argument which is the path to a JSON file. The JSON file defines the parameters needed to extract the requested mnemonic data. Project specific keys can be added as needed. The following table defines the standard parameters:

Note: If `raw`, `eng`, or `sci` are not provided, all mnemonics should be extracted as "science".

| Key | Required | Description |
|---------|----------|---|
| dir | ? | The path to the dir containing the archive file and any ancillary files needed for processing |
| dest | ? | The path to the dir that all output (e.g. <code>xbin</code> file) should be placed |
| mission | ? | TBC: The mission which may be needed to determine how the archive file is processed |
| model | ? | TBC: The mission's model which may be needed to determine how the archive file is processed |
| raw | | The mnemonic IDs of the mnemonics that should be extracted and output, unconverted |
| eng | | The mnemonic IDs of the mnemonics that should be extracted and output with the engineering conversion applied |
| sci | | The mnemonic IDs of the mnemonics that should be extracted and output with the science conversion applied |

| Key | Required | Description |
|-----------|----------|---|
| time_mode | | For projects that support it, defines which time source should be used when timestamping the mnemonic data. Either "pkt" or "grt" (ground receipt time). If not provided, it should default to packet time. |

Example JSON configuration file:

```
{
}
```

Output

The output of the tool should be a [xbin file](#). XINA's tools will then process the `xbin` file to produce the required mine or export products.

For mission specific data, `struct_extract` should output using one of the following options [TBD]:

- Self generate the data file and corresponding XINA JSON import file. XINA will take care of importing the data.
- Output the data into the xbin file. XINA will take care of the rest.

Return Codes

Return codes are used by the `struct_extract` app to return final execution status to the XINA processing.

| Name | Code | Description |
|------------------------|------|--|
| Success | 0 | Execution was successful |
| Error | 1 | Generic error code for unsuccessful execution. A more specific error code should be preferred over this one. |
| Finished with warnings | 3 | Execution finished but there were warnings. The log file should be examined for more info |

Struct Definitions Reference

This page provides a reference for the purpose and structure of all structs standard groups and databases.

Structs groups and databases are marked with a JSON object using the key `xs_struct`. This contains three standard parameters:

- `"type"` - the name of the type of struct element as a string
- `"v"` - the current version this instance of the specified type as a string
- `"conf"` - optional JSON object, format depends on type

Versioning is tied to the version number of the XINA server. Typically server updates will automatically apply needed changes to all structs schema elements, incrementing their `"v"` property to the latest version. In the event an upgrade cannot be performed the version will not be changed.

Groups

Note that group versioning is used to manage databases required within groups.

Project

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

Created with the [STRUCT CREATE PROJECT](#) action.

Struct Parameters

| Parameter | Value |
|-----------|------------------------|
| type | <code>"project"</code> |

Group Parameters

| Parameter | Value |
|-----------|-------|
| name | * |
| label | * |

Category

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

Created with the [STRUCT CREATE CATEGORY](#) action.

Struct Parameters

| Parameter | Value |
|-----------|------------|
| type | "category" |

Group Parameters

| Parameter | Value |
|-----------|-------|
| name | * |
| label | * |

Model

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

Created with the [STRUCT CREATE MODEL](#) action.

Struct Parameters

| Parameter | Value |
|-----------|---------|
| type | "model" |

Group Parameters

| Parameter | Value |
|-----------|-------|
| name | * |
| label | * |

Pipe

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

Created with the [STRUCT CREATE PIPE](#) action.

Struct Parameters

| Parameter | Value | Default |
|-----------|--------|---------|
| type | "pipe" | |

Conf Parameters

| Parameter | Value | Default |
|-----------|---------|---------|
| discrete | boolean | false |
| buffer | boolean | false |
| variable | boolean | false |
| condense | boolean | false |

| Parameter | Value | Default |
|-----------|------------------------------|---------|
| duration | archive length in minutes | 60 |
| partition | { "from": <y0>, "to": <y1> } | |

Group Parameters

| Parameter | Value |
|-----------|-------|
| name | * |
| label | * |

Notes

If `discrete` is `true`, mnemonic data is not considered persistent between archives, and open/close interval operations are not supported.

If `buffer` is `true`, the `mn_buffer` database will be generated, and the pipe will be included in automated archive tasks. Otherwise, the [STRUCT BUFFER IMPORT](#) action will not be supported.

If `variable` is `true`, mnemonic and event databases in the pipe will include the archive ID field (`a_id`). If `buffer` is `true`, `variable` must be `false` (since buffer-generated archives cannot be variable).

`duration` specifies the archive length in minutes, if `variable` is false. This cannot be changed. The default is 60 minutes (1 hour). A shorter window may be appropriate for very high data volumes. The maximum is 1440 (24 hours), and the value must be evenly divisible into 1440.

Changelog

11.0.0 (planned)

- renamed from `origin` to `pipe`

Definitions

Group containing definitions databases.

Struct Parameters

| Parameter | Value |
|-----------|-------|
| type | "def" |

Group Parameters

| Parameter | Value |
|-----------|---------------|
| name | "def" |
| label | "Definitions" |

Changelog

11.0.0 (planned)

- add filter definitions database

Task

Group containing task tracking databases. Must be a child of a pipe group.

Struct Parameters

| Parameter | Value |
|-----------|--------|
| type | "task" |

Group Parameters

| Parameter | Value |
|-----------|--------|
| name | "task" |
| label | "Task" |

Mnemonic

Group containing mnemonic data databases. Must be a child of a pipe group.

Struct Parameters

| Parameter | Value |
|-----------|-------|
| type | "mn" |

Group Parameters

| Parameter | Value |
|-----------|------------|
| name | "mn" |
| label | "Mnemonic" |

Mnemonic Bin

Group containing binned mnemonic data databases. Must be a child of a mnemonic group.

Struct Parameters

| Parameter | Value |
|-----------|----------|
| type | "mn_bin" |

Group Parameters

| Parameter | Value |
|-----------|-------|
| name | "bin" |

| Parameter | Value |
|-----------|-------|
| label | "Bin" |

Databases

Structs databases typically specify a set of required fields, and may permit the inclusion of additional custom fields. Changes to the spec involving fields will usually be treated as minor version changes, though they may require manual user correction if an added field conflicts with a custom field already present in a particular database instance.

Note that fields marked as **virtual** are calculated from the values of other field(s) and cannot be populated or edited manually.

Definitions

All definitions databases must be direct children of a [definitions](#) group, and all definitions groups must contain one of each definition database.

Diagram Definitions

Holds diagram definitions. The diagram itself is in an attached SVG file.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | "def_diagram" |

Database Parameters

| Parameter | Value |
|-----------|----------------|
| name | "diagram" |
| label | "Diagram" |
| format | "{name}" |
| order | (name , desc) |
| singular | "diagram" |

Fields

| Name | Type | Req | Description |
|-----------|------------------|-----|------------------------|
| name | utf8vstring(128) | ? | unique conf name |
| desc | utf8text | | plain text description |
| file_name | utf8filename | ? | file name |
| conf | jsonobject | | diagram configuration |

| Name | Type | Req | Description |
|------|------------|-----|-------------------------------|
| meta | jsonobject | | additional metadata as needed |

Changelog

11.0.0

- order changed to (name, asc)
- added desc field

Event Definitions

Holds event definitions, specifying how they are displayed, interpreted and processed. Filter Definition fields are defined in the conf field.

Struct Parameters

| Parameter | Value |
|-----------|-------------|
| type | "def_event" |

Database Parameters

| Parameter | Value |
|-----------|--------------------|
| name | "event" |
| label | "Event" |
| format | "{name}" |
| order | (name, asc) |
| singular | "event definition" |

Fields

| Name | Type | Req | Description |
|---------|------------------|-----|---------------------------------|
| e_id | int(4) | ? | unique ID |
| name | utf8vstring(128) | ? | unique name |
| desc | utf8text | | plain text description |
| meta | jsonobject | | additional arbitrary metadata |
| conf | jsonobject | | configuration for pseudo-events |
| aliases | set(utf8string) | | alternative name(s) |
| ext_id | asciivstring(64) | | external ID |

Filter conf jsonobject

| Name | Type | Req | Description |
|------|------------------|-----|-------------------------------|
| type | utf8vstring(128) | ? | Must have a value of "filter" |

| Name | Type | Req | Description |
|----------------|------------------|-----|---------------------------------------|
| condition | utf8text | ? | filter condition expression |
| t_start_offset | duration(us) | | start time offset (0 if not provided) |
| t_end_offset | duration(us) | | end time offset (0 if not provided) |
| models | set(asciistring) | | Models that the filter will apply to |

Changelog

11.2.0

- added Filter conf specification (no structural change)

11.0.0

- added aliases field
- added ext_id field
- removed type field
- changed e_id type from int(8) to int(4)

Mnemonic Definitions

Holds mnemonic definitions, specifying how they are displayed, interpreted and processed.

Struct Parameters

| Parameter | Value |
|-----------|--------|
| type | def_mn |

Database Parameters

| Parameter | Value |
|-----------|---------------------|
| name | mn |
| label | Mnemonic |
| format | {name} ({unit}) |
| order | (name , asc) |
| singular | mnemonic definition |

Fields

| Name | Type | Req | Description |
|---------|------------------|-----|----------------------|
| mn_id | int(4) | ? | unique mnemonic ID |
| name | utf8vstring(128) | ? | unique mnemonic name |
| subname | utf8vstring(32) | | mnemonic sub-name |

| Name | Type | Req | Description |
|---------|------------------|-----|---|
| desc | utf8text | | plain text mnemonic description |
| unit | utf8vstring(32) | | measurement unit (for example, "V", "mA") |
| state | struct_mn_state | ? | current state of mnemonic |
| pipes | jsonobject | ? | map of model(s) to associated pipe(s) |
| full | asciivstring(32) | | the primary database for the mnemonic, default f8 |
| bin | set(asciistring) | | the opt-in bin database(s) to include the mnemonic in |
| format | asciivstring(32) | | printf-style format to render values |
| enums | jsonobject | | mapping of permitted text values to numeric values |
| labels | list(jsonobject) | | mapping of numeric values or ranges to labels |
| aliases | set(utf8string) | | set of additional names associated with the mnemonic |
| meta | jsonobject | | additional metadata as needed |
| query | asciivstring(32) | | query name for pseudo-mnemonics |
| conf | jsonobject | | configuration for pseudo-mnemonics |
| ext_id | asciistring | | external ID |

Changelog

11.0.0

- added subname field
- added ext_id field
- rename origins field to pipes

1.0.0

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

Mnemonic Tracking

Used for tracking mnemonic selection activity. Although this is not strictly a definitions database, it is tightly coupled to the mnemonic definitions database, and is thus defined in the definitions context.

Struct Parameters

| Parameter | Value |
|-----------|----------------|
| type | "def_mn_track" |

Database Parameters

| Parameter | Value |
|-----------|-----------------------|
| name | "mn_track" |
| label | "Mnemonic Tracking" |
| format | "{t} {mn_id} {user}" |
| order | (name , asc) |
| singular | "mnemonic definition" |

Fields

| Name | Type | Req | Description |
|--------|------------------|-----|-------------------------------|
| t | instant(us) | ? | time of selection |
| mn_id | int(4) | ? | mnemonic ID selected |
| user | user_id | ? | user taking action |
| mns | set(int(4)) | | other mnemonic ID(s) selected |
| models | set(asciistring) | | model(s) in current context |

Changelog

11.0.0 (planned)

- rename `mn_ids` to `mns`
- order changed to (`t`, desc)
- format changed to "{t} {mn_id} {user}"

Nominal Definitions

Holds mnemonic nominal range definitions.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | "def_nominal" |

Database Parameters

| Parameter | Value |
|-----------|--|
| name | "nominal" |
| label | "Nominal" |
| format | "{mn_id} {color} ({min}, {max}) {label}" |
| order | (mn_id , asc), (label , asc) |
| singular | "nominal definition" |

Fields

| Name | Type | Req | Description |
|--------|----------------------|-----|---|
| unid | uuid | ? | unique nominal range ID |
| mn_id | int(4) | ? | unique mnemonic ID |
| label | utf8vstring(128) | ? | nominal range label |
| desc | utf8text | | plain text nominal range description |
| color | struct_nominal_color | | range color indicator |
| min | float(8) | | min value for the range |
| max | float(8) | | max value for the range |
| models | set(asciistring) | | models for which this range should apply (all if null) |
| meta | jsonobject | | additional metadata as needed |

Notes

The struct_nominal_color type is an enum of **green** (0), **yellow** (1), and **red** (2).

Changelog

11.2.0

- added "meta" field

11.0.0

- renamed "nominal_id" to "unid"

Plot Configuration Definitions

Holds mnemonic plot configuration definitions.

Struct Parameters

| Parameter | Value |
|-----------|------------|
| type | "def_plot" |

Database Parameters

| Parameter | Value |
|-----------|----------------------|
| name | "plot" |
| label | "Plot Conf" |
| format | "{name}" |
| order | (name , asc) |
| singular | "plot configuration" |

Fields

| Name | Type | Req | Description |
|-----------|------------------|-----|---|
| name | utf8vstring(128) | ? | unique conf name |
| desc | utf8text | | plain text conf description |
| plot_conf | struct_plot_conf | ? | configuration |
| models | set(asciistring) | | models for which this conf should apply (any if <code>null</code>) |

Changelog

11.0.0

- renamed field `conf` to `plot_conf` (to match profile definition)
- changed type of `plot_conf` from `jsonobject` to `struct_plot_conf`

Profile Definitions

Holds mnemonic profile definitions.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | "def_profile" |

Database Parameters

| Parameter | Value |
|-----------|---------------|
| name | "profile" |
| label | "Profile" |
| format | "{name}" |
| order | (name , asc) |
| singular | "profile" |

Fields

| Name | Type | Req | Description |
|-----------|------------------|-----|--|
| name | utf8vstring(128) | ? | unique profile name |
| desc | utf8text | | plain text profile description |
| models | set(asciistring) | | models for which this conf should apply (all if <code>null</code>) |
| data_conf | struct_data_conf | ? | profile data configuration |
| plot_conf | struct_plot_conf | | profile plot configuration. If not provided, defaults to 1 mnemonic per plot, 1 plot per page. |

| Name | Type | Req | Description |
|------------|------------------------|-----|--------------------------|
| auto_confs | list(struct_auto_conf) | | automation configuration |

struct_data_conf

jsonobject

| Name | Type | Req | Description |
|---------|----------------------------------|-----|---|
| ids | utf8text | | CSV range list of <code>mn_id</code> s e.g. "1,2-10,100" or <code>ext_id</code> formatted string e.g. "@[1,4-8,12,100-200]sci;@[2,3]raw" |
| filter | jsonarray of filter jsonobject s | | List of filters to apply. Each filter object may reference an existing filter by name using the <code>filter</code> key, or directly provide a filter definition (the <code>type</code> and <code>models</code> fields are ignored). An <code>ids</code> key can be used to define which mnemonics the filter should apply to. If <code>ids</code> is not provided, then the filter will be applied to every mnemonic. Only one filter per mnemonic is currently supported. |
| limit | boolean | | If <code>True</code> , generate the Limit Report. Defaults to <code>False</code> . |
| pkt | boolean | | If <code>True</code> , export using Packet Time instead of Ground Receipt Time. Defaults to <code>False</code> . Ignored if the archives only contain 1 time source. |
| join | boolean | | If <code>True</code> , the data file will be formatted with 1 unique time per row, and 1 mnemonic per column. Defaults to <code>False</code> . |
| fill | boolean | | If <code>True</code> and <code>join</code> is <code>True</code> , empty cells will be populated with the most recent value. Defaults to <code>False</code> . |
| dis | boolean | | If <code>True</code> , each mnemonic will also be exported using the Discrete conversion, if available. Defaults to <code>False</code> . |
| columns | jsonobject | | Defines which columns are included in the data file. |

columns

jsonobject

| Name | Type | Req | Description |
|-----------------|---------|-----|-------------|
| date_utc | boolean | | |
| ts_utc_iso | boolean | | |
| ts_utc_excel | boolean | | |
| ts_utc_excel_ms | boolean | | |

| Name | Type | Req | Description |
|---------------|---------|-----|-------------|
| ts_utc_doy | boolean | | |
| t_utc_unix_s | boolean | | |
| t_utc_unix_ms | boolean | | |
| t_utc_unix_us | boolean | | |
| date_tai | boolean | | |
| ts_tai_iso | boolean | | |
| ts_tai_doy | boolean | | |
| t_tai_unix_s | boolean | | |
| t_tai_unix_ms | boolean | | |
| t_tai_unix_us | boolean | | |
| t_tai_tai_s | boolean | | |
| t_tai_tai_ms | boolean | | |
| t_tai_tai_us | boolean | | |
| t_rel_s | boolean | | |
| t_rel_ms | boolean | | |
| t_rel_us | boolean | | |
| name | boolean | | |
| unit | boolean | | |

plot_conf jsonobject

See [Export Plot Format](#)

struct_auto_conf jsonobject

Note: The `auto_confs` field is a list of these described JSON objects.

| Name | Type | Req | Description |
|-------|-----------------------|-----|--|
| daily | boolean | | If <code>True</code> , the Profile will be exported once per day. Defaults to <code>False</code> . |
| mine | boolean | | If <code>True</code> , the Profile will be exported during the Mining Task when any of the defined <code>intervals</code> are processed. |
| users | set(utf8vstring(128)) | | The list of NASA AUIDs to notify via email when the Daily Profile Export is generated |

Notes

Requires review before use.

Changelog

11.0.0

- added `auto_conf`
- renamed field `data` to `data_conf`
- renamed field `plot` to `plot_conf`
- changed type of `plot_conf` from `jsonobject` to `struct_plot_conf`
- changed type of `data_conf` from `jsonobject` to `struct_data_conf`

Trend Definitions

Holds mnemonic trend definitions.

Struct Parameters

| Parameter | Value |
|-----------|--------------------------|
| type | <code>"def_trend"</code> |

Database Parameters

| Parameter | Value |
|-----------|---------------------------------|
| name | <code>"trend"</code> |
| format | <code>"Trend"</code> |
| order | <code>(name , asc)</code> |
| singular | <code>"trend definition"</code> |

Fields

| Name | Type | Req | Description |
|------------|--------------------------------|-----|---|
| name | <code>utf8vstring(128)</code> | ? | unique trend name |
| desc | <code>utf8text</code> | | plain text trend description |
| profiles | <code>set(utf8string)</code> | ? | profile name(s) to include in trend |
| models | <code>set(asciistring)</code> | | models for which this trend should apply (any if <code>null</code>) |
| trend_conf | <code>struct_trend_conf</code> | | trend configuration |
| plot_conf | <code>jsonobject</code> | | Mapping of profile name to <code>struct_plot_conf</code> objects, allowing you to override a Profile's plot conf. |
| auto_conf | <code>struct_auto_conf</code> | | automated generation configuration |

`struct_trend_conf` `jsonobject`

| Name | Type | Req | Description |
|------|------|-----|-------------|
|------|------|-----|-------------|

| | | | |
|----------------|----------------------------------|--|---|
| bin_size | int(4) | | The bin size in minutes to use when trending time ranges |
| bin_count | int(4) | | Multiplier of the bin_size to use when trending time ranges. The actual trended bin size in minutes is bin_size * bin_count . |
| t | array of time range JSON objects | | List of JSON objects describing time ranges to trend e.g. { "start": "2021-06-30T00:00:00Z", "end": "2021-07-21T00:00:00Z" } |
| intervals | array of interval JSON objects | | List of JSON objects describing Event Intervals to trend |
| disable_filter | boolean | | If True , do not use any filtered data. Defaults to False . |

plot_conf jsonobject

Allows the Trend Definition to override a Profile's plot configuration. In the below example, FLT_CRIT_TEMPS is a profile name. The plot configuration is the same format as the Profile's plot configuration.

```
{
  "FLT_CRIT_TEMPS": {
    "pages": [
      {
        "plots": [
          {
            "title": "Chamber Pressure",
            "mnemonics": [
              "oci.gse.ocl_pc.Pressure.hk.Chamber"
            ]
          }
        ]
      }
    ],
    ...
  }
}
```

struct_auto_conf jsonobject

| Name | Type | Req | Description |
|-------|---------|-----|---|
| daily | boolean | | If True , the Trend will be generated once per day. Defaults to False . |
| mine | boolean | | If True , the Trend will be generated during the Mining Task when either the Time Range or Interval condition is satisfied. |

| Name | Type | Req | Description |
|-------|-----------------------|-----|--|
| users | set(utf8vstring(128)) | | The list of NASA AUIDs to notify via email when the Daily Trend is generated |

Changelog

11.2.0

- added `models` field
- added `trend_conf` field
- added `auto_conf` field
- removed `intervals` field

11.0.0

- initial release

Files

Archive

Contains all archive files for a pipe. Must be a child of a pipe group.

Struct Parameters

| Parameter | Value |
|-----------|----------------|
| type | "file_archive" |

Database Parameters

| Parameter | Value |
|-----------|---------------------|
| name | "archive" |
| label | "Archive" |
| format | "{t_start} {t_end}" |
| singular | "archive file" |

Fields

| Name | Type | Req | Description |
|---------|-------------|-----|--|
| a_id | int(4) | ? | archive ID |
| ufid | uuid | ? | file UUID |
| t_start | instant(us) | ? | start time of the time range that the file covers |
| t_end | instant(us) | ? | end time of the time range that the file covers, not inclusive |

| Name | Type | Req | Description |
|-----------|------------------|-----|--|
| dur | duration(us) | ? | virtual <code>t_end</code> - <code>t_start</code> |
| 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 |
| meta | jsonobject | | additional metadata as needed |
| format | asciivstring(32) | | file format: <code>xbin</code> or <code>xpf</code> , where <code>xpf</code> is a zipped dir (default <code>xbin</code>) |
| conf | jsonobject | | configuration for format as needed |

Changelog

11.0.0

- changed type from `"mn_file_archive"` to `"file_archive"`
- renamed `uuid` to `ufid`

Buffer

Contains all buffer files for a pipe. Must be a child of a pipe group.

Struct Parameters

| Parameter | Value |
|-----------|----------------------------|
| type | <code>"file_buffer"</code> |

Database Parameters

| Parameter | Value |
|-----------|----------------------------|
| name | <code>"buffer"</code> |
| label | <code>"Buffer"</code> |
| format | <code>"{file_name}"</code> |
| singular | <code>"buffer file"</code> |

Fields

| Name | Type | Req | Description |
|-----------|--------------|-----|--|
| ufid | 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 <code>t_max</code> - <code>t_min</code> |

| Name | Type | Req | Description |
|--------|---------------------|-----|------------------------------------|
| state | struct_buffer_state | ? | buffer file state |
| flag | struct_buffer_flag | | buffer file flag |
| meta | jsonobject | | additional metadata as needed |
| format | asciivstring(32) | | buffer file format (default "csv") |
| conf | jsonobject | | configuration for format as needed |

Notes

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
- 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
- RESTORE - the file is queued for restoration
- DELETE - the file is queued for deletion

Changelog

11.0.0

- changed type from "mn_file_buffer" to "file_buffer"
- renamed uuid to ufid
- removed PROCESSED state

CFT

Current filter table for a pipe. Stores a file for each archive containing the state of each filter at the end of the archive. Only created in pipes where discrete is false.

Struct Parameters

| Parameter | Value |
|-----------|------------|
| type | "file_cft" |

Database Parameters

| Parameter | Value |
|-----------|---------------|
| name | "cft" |
| label | "CFT" |
| format | "{file_name}" |

| Parameter | Value |
|-----------|------------|
| singular | "CFT file" |

Fields

| Name | Type | Req | Description |
|-----------|------------------|-----|------------------------------------|
| a_id | int(4) | ? | archive ID |
| ufid | uuid | ? | file UUID |
| file_name | utf8filename | ? | CVT file name |
| format | asciivstring(32) | | CVT file format (default "csv") |
| conf | jsonobject | | configuration for format as needed |

Changelog

11.0.0

- initial release

CVT

Current value table for a pipe. Stores a file for each archive containing the values for each mnemonic at the end of the archive. Only created in pipes where discrete is false.

Struct Parameters

| Parameter | Value |
|-----------|------------|
| type | "file_cvt" |

Database Parameters

| Parameter | Value |
|-----------|---------------|
| name | "cvt" |
| label | "CVT" |
| format | "{file_name}" |
| singular | "CVT file" |

Fields

| Name | Type | Req | Description |
|-----------|--------------|-----|-------------------------------|
| a_id | int(4) | ? | archive ID |
| ufid | uuid | ? | file UUID |
| file_name | utf8filename | ? | CVT file name |
| meta | jsonobject | | additional metadata as needed |

| Name | Type | Req | Description |
|--------|------------------|-----|------------------------------------|
| format | asciivstring(32) | | CVT file format (default "csv") |
| conf | jsonobject | | configuration for format as needed |

Changelog

11.0.0

- initial release

Package

Stores generated export packages. Child of a model group.

Struct Parameters

| Parameter | Value |
|-----------|----------------|
| type | "file_package" |

Database Parameters

| Parameter | Value |
|-----------|----------------|
| name | "package" |
| label | "Package" |
| format | "{file_name}" |
| singular | "package file" |

Fields

| Name | Type | Req | Description |
|-----------------|------------------|-----|---|
| t_start | instant(us) | ? | start time |
| t_end | instant(us) | ? | end time |
| a_id | int(4) | | archive ID (if generated automatically) |
| ueid | UUID | | UEID, if time range from event |
| label | utf8vstring(128) | ? | package label |
| file_name | utf8filename | ? | package file name |
| data_conf | struct_data_conf | ? | data configuration |
| plot_conf | struct_plot_conf | | plot configuration |
| auto_conf | struct_auto_conf | | automation configuration |
| profile | utf8vstring(128) | | profile name, if profile used |
| profile_version | int(4) | | profile version, if profile used |

| Name | Type | Req | Description |
|------|-------------------------|-----|-------------------------------|
| meta | <code>jsonobject</code> | | additional metadata as needed |

Changelog

11.0.0

- initial release

Trend Package

Stores generated Trend Export packages. Child of a model group.

Struct Parameters

| Parameter | Value |
|-----------|---------------------------|
| type | <code>"file_trend"</code> |

Database Parameters

| Parameter | Value |
|-----------|----------------------------|
| name | <code>"trend"</code> |
| label | <code>"Trend"</code> |
| format | <code>"{file_name}"</code> |
| singular | <code>"trend file"</code> |

Fields

| Name | Type | Req | Description |
|------------|--------------------------------|-----|--|
| name | <code>utf8vstring(128)</code> | ? | Name of Trend Definition used to generate the Trend Package |
| desc | <code>utf8text</code> | ? | Description of Trend Definition used to generate the Trend Package |
| profiles | <code>set(utf8string)</code> | ? | Profiles included in the Trend Package |
| trend_conf | <code>struct_trend_conf</code> | ? | Trend Definition's trend conf used to generate the Trend Package |
| plot_conf | <code>struct_plot_conf</code> | ? | Trend Definition's plot conf used to generate the Trend Package |
| file_name | <code>utf8filename</code> | ? | The filename of the Trend Package zip file |
| meta | <code>jsonobject</code> | ? | Metadata |
| t_min | <code>instant(us)</code> | ? | Time of the earliest bin point used in the trend |

| Name | Type | Req | Description |
|--------------|--------------------------|-----|--|
| t_max | <code>instant(us)</code> | ? | Time of the latest bin point used in the trend |
| generated_at | <code>instant(us)</code> | ? | Datetime when the Trend Package was generated |
| generated_by | <code>user_id</code> | ? | ID of user that generated the Trend Package |

Changelog

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 child of either a model or pipe group.

Struct Parameters

| Parameter | Value |
|-----------|----------------------|
| type | <code>"event"</code> |

Database Parameters

| Parameter | Value |
|-----------|--|
| name | * (default <code>"event"</code>) |
| label | * (default <code>"Event"</code>) |
| format | * (default <code>"{t_start} {event_id} {label}"</code>) |
| order | * (default <code>(t_start , desc), (event_id , asc)</code>) |
| singular | * (default <code>"event"</code>) |

Fields

| Name | Type | Req | Virtual | Description |
|---------|--------------------------|-----|---------|--|
| ueid | <code>uuid</code> | ? | | event UUID |
| e_id | <code>int(4)</code> | ? | | event ID (default to <code>0</code> if not provided) |
| a_id | <code>int(4)</code> | ? | | archive ID (only present if child of a pipe group) |
| t_start | <code>instant(us)</code> | ? | | start time |
| t_end | <code>instant(us)</code> | | | end time, not inclusive (if <code>null</code> , event is an open interval) |

| Name | Type | Req | Virtual | Description |
|----------|--------------------|-----|---------|---|
| dur | duration(us) | ? | ? | duration in microseconds (null if open) |
| interval | boolean | ? | ? | t_start != t_end |
| open | boolean | ? | ? | 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) |
| label | utf8vstring(128) | ? | | plain text label |
| content | utf8text | | | extended event content |
| meta | jsonobject | | | additional metadata as needed |
| conf | jsonobject | | | configuration for specific event types |

Notes

Virtual fields are calculated from other fields and cannot be populated manually.

Changelog

11.0.0

- changed uuid to uuid
- changed e_id type from int(8) to int(4)
- removed name field
- added a_id field (when child of pipe group)

1.0.2

- corrected t_end and dur as not required

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.

Database Parameters

| Parameter | Value |
|-----------|--|
| name | * (default <code>"eventf"</code>) |
| label | * (default <code>"Event File"</code>) |
| singular | * (default <code>"event file"</code>) |

| Name | Type | Req | Description |
|-----------|---------------------------|-----|-------------|
| file_name | <code>utf8filename</code> | ? | file name |

Event Files

Uses same structure as event database, but with a child file database, allowing each event to contain zero or more files.

Database Parameters

| Parameter | Value |
|-----------|---|
| name | * (default <code>"eventfs"</code>) |
| label | * (default <code>"Event Files"</code>) |
| singular | * (default <code>"event files"</code>) |

Event Update

Captures updates to events as records.

Struct Parameters

| Parameter | Value |
|-----------|-----------------------------|
| type | <code>"event_update"</code> |

Database Parameters

| Parameter | Value |
|-----------|---|
| name | <code>"{name}_update"</code> |
| label | <code>"{label} Update"</code> |
| format | <code>"{t_start} {ueid} {label}"</code> |
| order | <code>(t, desc)</code> |
| singular | <code>"event change"</code> |

Fields

| Name | Type | Req | Description |
|------|--------------------------|-----|-------------------|
| t | <code>instant(us)</code> | ? | event change time |

| Name | Type | Req | Description |
|--------|-------------------------|-----|--------------------|
| uuid | <code>uuid</code> | ? | event UUID |
| update | <code>jsonobject</code> | ? | field(s) to update |

Changelog

11.0.0

- initial release

Mnemonics

Databases containing mnemonic data. Unless otherwise indicated, mnemonic databases can be configured with a partitioning system which subdivides the tables internally into UTC calendar months. This is beneficial for selective mnemonic data repopulation, since each month can be instantly erased before being regenerated, and doesn't require downtime of the entire dataset. At creation time a start and end year must be specified, and partitions will be created for each month in that range. Data with timestamps outside the range will be stored in either a pre-range or post-range partition as applicable.

Mn Buffer

Each record is a single mnemonic data point. Holds data imported through the buffer pipeline. Unlike other mnemonic databases, may contain duplicate data points.

This database does not hold data indefinitely. The automated archive pipeline will remove data as it is archived and mined into the primary mnemonic databases.

Struct Parameters

| Parameter | Value |
|-----------|--------------------------|
| type | <code>"mn_buffer"</code> |

Database Parameters

| Parameter | Value |
|-----------|--|
| name | <code>"buffer"</code> |
| label | <code>"Buffer"</code> |
| format | <code>"{t} {mn_id} {v}"</code> |
| singular | <code>"mnemonic buffer datapoint"</code> |

Fields

| Name | Type | Req | Description |
|-------|--------------------------|-----|--------------------|
| t | <code>instant(us)</code> | ? | time |
| mn_id | <code>int(4)</code> | ? | unique mnemonic ID |
| v | <code>{conf.type}</code> | | value |

Mn Full

Each record is a single mnemonic data point. The data type for mnemonic values is configurable, and determines the database name. By default the mnemonic data group will contain a full float(8) database.

Struct Parameters

| Parameter | Value |
|-----------|-----------|
| type | "mn_full" |

Conf Parameters

| Parameter | Value | Default |
|-----------|---|------------|
| type | "int(1)", "int(2)", "int(4)", "int(8)", "float(4)", or "float(8)" | "float(8)" |

Database Parameters

| Parameter | Value |
|-----------|---------------------------------------|
| name | "i1", "i2", "i4", "i8", "f4", or "f8" |
| label | "Full <conf.type>" |
| format | "{t} {mn_id} {v}" |
| singular | "mnemonic datapoint" |

Fields

| Name | Type | Req | Description |
|-------|-------------|-----|--------------------|
| a_id | int(4) | ? | archive ID |
| t | instant(us) | ? | time |
| mn_id | int(4) | ? | unique mnemonic ID |
| v | {conf.type} | | value |

Changelog

11.0.0

- added a_id

Mn Delta

An optimized mnemonic storage solution with each record representing one or more mnemonic data points, by only including points where the mnemonic value actually changes. The value data type is customizable, as with the Mn Full database. By default the mnemonic data group will contain a delta float(8) database.

Struct Parameters

| Parameter | Value |
|-----------|-------|
|-----------|-------|

| | |
|------|------------|
| type | "mn_delta" |
|------|------------|

Conf Parameters

| Parameter | Value | Default |
|-----------|---|------------|
| type | "int(1)", "int(2)", "int(4)", "int(8)", "float(4)", or "float(8)" | "float(8)" |

Database Parameters

| Parameter | Value |
|-----------|---|
| name | "di1", "di2", "di4", "di8", "df4", or "df8" |
| label | "Delta {conf.type}" |
| format | "{t} {mn_id} {v} ({n})" |
| singular | "mnemonic delta datapoint" |

Fields

| Name | Type | Req | Description |
|-------|-------------|-----|---|
| a_id | int(4) | ? | archive ID |
| t | instant(us) | ? | time |
| mn_id | int(4) | ? | unique mnemonic ID |
| v | {conf.type} | | value |
| n | int(4) | ? | number of datapoints included in this point |

Changelog

11.0.0

- added a_id

Mn Bin Time

Contains mnemonic data binned on fixed time intervals. By default these will be created for 1 minute ("t60") and 10 minute ("t600") bin sizes.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | "mn_bin_time" |

Conf Parameters

| Parameter | Value |
|-----------|---------------------|
| t | bin size in seconds |

Database Parameters

| Parameter | Value |
|-----------|------------------------------------|
| name | "t<conf.t>" |
| label | "Time (<conf.t>s)" |
| format | "{t} {mn_id} {avg} ({min}, {max})" |
| singular | "mnemonic bin" |

Fields

| Name | Type | Req | Description |
|-------|-------------|-----|-----------------------------|
| a_id | int(4) | ? | archive ID |
| t | instant(us) | ? | start time of the bin |
| mn_id | int(4) | ? | unique mnemonic ID |
| t_min | instant(us) | ? | time of first datapoint |
| t_max | instant(us) | ? | time of last datapoint |
| n | int(4) | ? | number of datapoints in bin |
| avg | float(8) | ? | average |
| min | float(8) | ? | min |
| max | float(8) | ? | max |
| std | float(8) | ? | sample standard deviation |

Changelog

11.0.0

- added `a_id` field
- removed `med` and `var` fields

Mn Bin Interval

Contains mnemonic data binned by interval events.

Struct Parameters

| Parameter | Value |
|-----------|-------------------|
| type | "mn_bin_interval" |

Database Parameters

| Parameter | Value |
|-----------|------------|
| name | "interval" |
| label | "Interval" |

| Parameter | Value |
|-----------|---|
| format | "{t} {e_id} {mn_id} {avg} ({min}, {max})" |
| singular | "mnemonic bin" |

Fields

| Name | Type | Req | Description |
|---------|-------------|-----|-----------------------------|
| a_id | int(4) | ? | archive ID |
| ueid | uuid | ? | |
| e_id | int(8) | ? | |
| t_start | instant(us) | ? | start time |
| t_end | instant(us) | ? | end time |
| mn_id | int(4) | ? | unique mnemonic ID |
| t_min | instant(us) | ? | time of first datapoint |
| t_max | instant(us) | ? | time of last datapoint |
| n | int(4) | ? | number of datapoints in bin |
| avg | float(8) | ? | average |
| min | float(8) | ? | min |
| max | float(8) | ? | max |
| std | float(8) | | sample standard deviation |

Changelog

11.0.0

- added `a_id` field
- renamed field `uuid` to `ueid`
- removed `med` and `var` fields

Mn Bin Edge

Contains mnemonic data binned on archive boundaries and cross-archive interval event boundaries. Used to generate interval bins efficiently for cross-archive interval events. Only created in pipes where `discrete` is `false`.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | "mn_bin_edge" |

Database Parameters

| Parameter | Value |
|-----------|--------|
| name | "edge" |

| Parameter | Value |
|-----------|--|
| label | "Edge" |
| format | "{t_start} - {t_end} {mn_id} {avg} ({min}, {max})" |
| singular | "mnemonic bin" |

Fields

| Name | Type | Req | Description |
|---------|-------------|-----|-----------------------------|
| a_id | int(4) | ? | archive ID |
| t_start | instant(us) | ? | start time |
| t_end | instant(us) | ? | end time |
| mn_id | int(4) | ? | unique mnemonic ID |
| t_min | instant(us) | ? | time of first datapoint |
| t_max | instant(us) | ? | time of last datapoint |
| n | int(4) | ? | number of datapoints in bin |
| avg | float(8) | ? | average |
| min | float(8) | ? | min |
| max | float(8) | ? | max |
| std | float(8) | ? | sample standard deviation |

Changelog

11.0.0

- initial release

Tasks

Store logs and associated files for data processing tasks.

Condense

Each record logs a single execution of a condense task.

Struct Parameters

| Parameter | Value |
|-----------|---------------|
| type | task_condense |

Database Parameters

| Parameter | Value |
|-----------|------------|
| name | "condense" |

| Parameter | Value |
|-----------|-----------------|
| label | "Condense" |
| format | "{task_id} {t}" |
| singular | "condense task" |

Fields

| Name | Type | Req | Description |
|-----------|------------------|-----|-------------------------------|
| task_id | task_id | ? | unique task ID |
| t | instant(us) | ? | time when task submitted |
| meta | jsonobject | | additional metadata as needed |
| conf | jsonobject | | task configuration |
| condensed | list(jsonobject) | | buffer file(s) condensed |

Mine

Each record logs a single execution of a mine task.

Struct Parameters

| Parameter | Value |
|-----------|-----------|
| type | task_mine |

Database Parameters

| Parameter | Value |
|-----------|-----------------------|
| name | "mine" |
| label | "Mine" |
| format | "{task_id} {t_start}" |
| singular | "mine task" |

Fields

| Name | Type | Req | Description |
|---------|-------------|-----|--------------------------------|
| task_id | task_id | ? | unique task ID |
| t | instant(us) | ? | time when task submitted |
| ufid | uuid | ? | source archive file UUID |
| t_start | instant(us) | ? | source archive file start time |
| t_end | instant(us) | ? | source archive file end time |
| meta | jsonobject | | additional metadata as needed |
| conf | jsonobject | | task configuration |

Changelog

11.0.0

- renamed `uuid` to `ufid`

Archive

Each record logs a single execution of an archive task.

Struct Parameters

| Parameter | Value |
|-----------|--------------|
| type | task_archive |

Database Parameters

| Parameter | Value |
|-----------|-----------------|
| name | "archive" |
| label | "Archive" |
| format | "{task_id} {t}" |
| singular | "archive task" |

Fields

| Name | Type | Req | Description |
|------------|------------------|-----|-------------------------------|
| task_id | task_id | ? | unique task ID |
| t | instant(us) | ? | time when task submitted |
| meta | jsonobject | | additional metadata as needed |
| conf | jsonobject | | task configuration |
| archives | list(jsonobject) | | archives updated |
| archived | list(jsonobject) | | buffer file(s) archived |
| restored | list(jsonobject) | | buffer file(s) restored |
| deprecated | list(jsonobject) | | buffer file(s) deprecated |
| deleted | list(jsonobject) | | buffer file(s) deleted |

Spectra

The spectra definition is a property for event databases.

| Property | Value | Req | Description |
|----------|----------------------|-----|--------------------|
| tabs | array of tab conf(s) | | custom tabs for UI |

| Property | Value | Req | Description |
|-----------|--------------------------|-----|-------------------------------------|
| 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 | <code>"database"</code> | ? | 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 | <code>"field"</code> | ? | presearch type name |
| field | field specifier | ? | |
| 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.

Name Conventions Reference

Structs definition names have certain requirements and optional conventions, unless otherwise indicated.

Names are limited to **128 characters** and may not include the following reserved characters:

- `&` (ampersand)
- `!` (exclamation point)
- `?` (question mark)
- `$` (dollar sign)
- `:` (colon)
- `;` (semicolon)
- `#` (number symbol)
- `*` (asteriks)
- `@` (at symbol)
- `,` (comma)
- `(` (open parentheses)
- `)` (close parentheses)
- `{` (open brace)
- `}` (close brace)

In cases where names are used by API actions to lookup definitions, the `@` character may be used to indicate an external ID instead of a plain name.

For matching purposes names are **case insensitive** and **normalized** with any leading/trailing whitespace removed and any internal whitespace represented by a single underscore character. For example:

```
"v_mon" = "V Mon" = " V MON "
```

XINA tools will interpret the period character (`.`) to indicate a tree structure relationship, and brackets (`[]`) to indicate an array of values. This is entirely presentational, not functional.

For example, the set of names:

```
foo.bar
foo.baz.bit
foo.arr[0]
foo.arr[1]
foo.arr[2]
```

Would be displayed as:

- `foo`
 - `bar`
 - `baz`
 - `bit`
 - `arr[]`

- `arr[0]`
- `arr[1]`
- `arr[2]`

This is not required but highly recommended to improve organization in large namespaces.

Units Reference