Specifications

- Mnemonics Database
- Metadata File
- Mnemonic Limit Definitions
- Expected Values Definition

Mnemonics Database

The telemetry database defines named telemetry values (mnemonics): which packets contain them, how to decode their raw values, how to convert those values to engineering and scientific units, and how to display, monitor, and mine them. Each row defines one mnemonic.

This page describes the tab-delimited input accepted by our ground software. It covers the shared format and generic decoders and conversions. Mission plugins can define additional decoder and equation types with their own field meanings. Files ending in `.json` use a separate, mission-specific packet-definition format; the column specification below does not describe that format.

File format

- Use a plain-text file with actual **tab characters** between cells and one row per line. A spreadsheet must be exported as tab-delimited text; `.xls` and `.xlsx` workbooks are not read directly. Use a filename such as `tm_database.txt` or `tm_database.tsv`.
- The first line is the header. It must contain every column marked **Required** below, even if a particular mnemonic does not use that column. Optional columns can be omitted entirely. Columns can be reordered; header matching is case-insensitive. Use the exact canonical names in this reference, without extra spaces or duplicate columns.
- Keep a cell for every header column in every data row, including trailing empty cells. Empty cells require adjacent tabs. Extra, unrecognized columns are ignored.
- Empty lines are ignored. A comment line starts with `#` in its first character position. The header must come before any comments.
- Cells cannot contain literal tabs or line breaks. This is not a CSV quoting format: double quotes do not protect a tab, and doubled quotes are not unescaped. Ordinary text cells may have one pair of surrounding double quotes removed; leave decoder specifications and JSON cells unquoted.
- Use ASCII or Latin-1 characters for mnemonic names and other ordinary text fields. Do not depend on arbitrary Unicode text being preserved.

Value notation and defaults

Integer fields generally accept decimal and `0x`-prefixed hexadecimal notation. A leading zero may select octal notation, so write decimal `12`, not `012`. **Use positive decimal byte counts in `DECODE_ID2`**, or an explicit colon-separated byte-order specification. Coefficients, thresholds, and time intervals use decimal floating-point notation, including scientific notation such as `1.5e-3`.

Boolean cells use `1` or `Y` for true and `0` or `N` for false. The first character determines the value: `y` is also true, but `true` is false. Do not put spaces before the value.

The defaults in the field tables apply to a **new row** when the cell is blank or an optional column is absent. Most numbers default to zero, strings to empty, and Booleans to false. A valid decoding definition still needs explicit values; default `DECODE_TYPE=0` does not select a generic decoder. Invalid numeric text is not consistently rejected and may become zero. Validate values before loading.

Blank cells in a later definition of an existing `DATA_ID` generally retain the previous value. Decoding-layout fields need special care; see Database patches.

Field reference

Identification and packet selection

Column	Header	Value / new-row default	Meaning
`SUB_SYSTEM`	Required	Text / empty	Namespace for the mnemonic. Together with `NAME`, forms the lookup key `SUB_SYSTEM:NAME`. An empty subsystem produces a key such as `:TEMPERATURE`.
`NAME`	Required	Text / empty	Mnemonic name within the subsystem. Use a unique combined key and preserve spelling and case in references.
`DESCRIPTION`	Required	Text / empty	Human-readable description displayed with the mnemonic.
`DATA_ID`	Required	Positive integer; no valid default	Stable numeric identifier, also called an HKID. Use a unique ID from `1` through `2147483647`. Zero and invalid IDs are skipped. A negative ID is a deletion directive, described below.
`PACKET_TYPE`	Required	Unsigned integer / `0`	Packet type containing the value, commonly a CCSDS APID. Must match the application's packet reader and packet-type definitions. This is not the CCSDS primary-header type bit.
`PACKET_SUBTYPE`	Optional	Unsigned integer / `0`	Additional packet selector. `0` accepts any subtype; a nonzero value requires an exact subtype match.
`PACKET_NAME`	Optional	Text / empty	Descriptive packet name. Packet selection uses `PACKET_TYPE` and `PACKET_SUBTYPE`, not this name.
`APPLY_WHEN`	Optional	Condition expression / empty	Decode only when all referenced raw values meet their conditions. Empty means unconditional. See Conditional decoding.
`ADC_#`	Optional	Unsigned integer / `0`	ADC channel metadata. Does not select a byte offset or change a generic conversion.
`MUX_CH`	Optional	Unsigned integer / `0`	Multiplexer channel metadata. Does not filter packets or select samples in the generic decoders.

Keep `DATA_ID` and the combined mnemonic key stable when revising a definition. Reusing an ID updates the same mnemonic rather than adding an independent row. Renaming a key for an existing ID can leave the previous key usable as an alias; avoid relying on this as a renaming mechanism.

Raw decoding and sample layout

Column	Header	Value / new-row default	Meaning
`DECODE_TYPE`	Required	Decoder ID / `0`	Selects the decoder. Generic IDs are listed below; mission-specific IDs require the corresponding plugin.
`DECODE_ID1`	Required	Packet byte offset or offset list / `0`	For generic packet-field decoders, zero-based offset from the beginning of the complete packet, **including packet headers**. Also supports multiple sample offsets and sample-time offsets.
`DECODE_ID2`	Required	Byte count or byte-order list / `0`	For numeric fields, sets the number and significance of bytes. A positive decimal count is big-endian; an explicit list supports other orders. String and byte-field meanings differ, as described below.
`DECODE_ID3`	Required	Unsigned mask up to 64 bits / `0`	Integer-field bit mask applied after assembling the bytes. `0` means no masking. Use a full-width mask for signed fields. Ignored by generic float and double decoders.
`DECODE_ID4`	Required	Nonnegative integer / `0`	Right shift applied after masking integer fields. Ignored by generic float and double decoders.
`ARRAY_LENGTH`	Optional	Positive integer / `1`	Number of consecutive samples when `DECODE_ID1` is a single offset. `0` is treated as `1`. Explicit offset lists define their own sample count.
`SAMPLE_PERIOD`	Optional	Seconds / `0`	With a single starting offset, a positive period adds `i × SAMPLE_PERIOD` to sample `i`'s packet timestamp. Zero leaves all samples at the packet time. Explicit time offsets take precedence.
`CLOCK`	Optional	Clock name / no clock	Selects a named clock from `clocks.txt` for supported time decoders. Names are case-sensitive. See Clock definitions.

Byte order, masks, and shifts

For a numeric field, `DECODE_ID2=2` means two bytes in big-endian order and is equivalent to `2:1`. In an explicit list, each entry gives the significance of the corresponding packet byte: `1` is the least-significant byte, `2` the next, and so on.

Numeric-field `DECODE_ID2`	Interpretation
`2` or `2:1`	Two-byte big-endian value.
`1:2`	Two-byte little-endian value.
`4` or `4:3:2:1`	Four-byte big-endian value.
`1:2:3:4`	Four-byte little-endian value.
`2:1:4:3`	Four-byte value with swapped two-byte words.

For integer values, the result is `(assembled_bytes & mask) >> shift`, except that a zero mask leaves the assembled value unmasked. Use at most eight bytes, valid byte significances, and shifts below 64. For decoder `4`, use a contiguous mask whose lowest selected bit is moved to bit zero by `DECODE_ID4`; the number of selected mask bits determines the signed width.

For example, a three-bit field in bits 15–13 of a two-byte big-endian word uses `DECODE_TYPE=3`, `DECODE_ID2=2`, `DECODE_ID3=0xE000`, and `DECODE_ID4=13`. Packet bytes `A0 00` decode to raw value `5`. A signed 16-bit field uses decoder `4`, mask `0xFFFF`, and shift `0`; bytes `FF FE` decode to `-2`.

Multiple samples

Generic numeric decoders `3`–`6` support these layouts:

Layout	Configuration	Result
Consecutive samples	`DECODE_ID1=20`, `DECODE_ID2=2`, `ARRAY_LENGTH=3`, `SAMPLE_PERIOD=0.1`	Values at byte offsets `20`, `22`, `24`, timestamped at packet time plus `0`, `0.1`, `0.2` seconds.
Explicit offsets	`DECODE_ID1=20:28:36`, `DECODE_ID2=2`	Three values from nonconsecutive locations, all at the packet time.
Explicit offsets and times	`DECODE_ID1=20,0:28,0.1:36,0.2`, `DECODE_ID2=2`	Three values with the specified time offsets in seconds. Negative time offsets are also allowed.

Each explicit entry is `byte_offset` or `byte_offset,time_offset`; entries are separated by colons. If one sample has an explicit time offset, provide one for every sample. Do not mix timed and untimed entries. The packet offset identifies the start of a sample; its bytes must be consecutive even when the samples themselves are not.

Generic decoder IDs

ID	Decoder	Configuration
`1`	Legacy simple unsigned	Deprecated. Use `3` for new definitions.
`2`	Legacy little-endian unsigned	Deprecated. Use `3` with explicit little-endian byte order.
`3`	Unsigned integer	Byte order, mask, shift, and sample layout as above.
`4`	Two's-complement integer	Same layout as `3`, with sign extension based on mask width.
`5`	IEEE 754 float	Exactly four bytes. Byte order and sample layout apply; mask and shift do not.
`6`	IEEE 754 double	Exactly eight bytes. Byte order and sample layout apply; mask and shift do not.

ID	Decoder	Configuration
`7`	String	Uses the first sample offset. See string rules below.
`8`	Time40	Four big-endian bytes of seconds.
`9`	Time42	Four big-endian bytes of seconds plus two of subseconds; default `65536` subsecond ticks per second.
`10`	Time44	Four big-endian bytes of seconds plus four of subseconds; default `1000000` ticks per second.
`12`	Time12	One byte of seconds plus two big-endian bytes of subseconds; default `65536` ticks per second.
`14`	Time41	Four big-endian bytes of seconds plus one of subseconds; default `256` ticks per second.
`15`	Time43	Four big-endian bytes of seconds plus three of subseconds; default `16777216` ticks per second.
`21`	TimeNS	Eight big-endian bytes containing a nanosecond count.
`28`	RTime40	Four **little-endian** bytes of seconds; supports the same clock correction as Time40.
`51`–`56`	History statistics	Maximum, average, minimum, rate, sum, and sum of squares respectively. Require `HISTORY` and a history-aware consumer.
`61`	Bytes	Uses the first sample offset. Reads up to `DECODE_ID2` bytes or the packet end, whichever comes first; zero bytes are preserved. Use a positive decimal maximum length.

The listed time decoders use fixed layouts and the first sample offset; numeric byte-order permutations and arrays do not change their time format. Their layouts are big-endian except for `RTime40`. Set `DECODE_ID2` to the corresponding field length and set unused mask and shift fields to zero. These time decoders use `CLOCK` when available; without it they return the decoded seconds count and fractional seconds without epoch or leap-second correction.

IDs `8`–`50` are reserved for time decoding, but an enum value alone does not ensure that a decoder is available. Other time IDs and mission-specific IDs must be checked against the plugins loaded by the target application. An unavailable decoder produces no decoded value.

For strings, an explicit `DECODE_ID2=1:2:3:4` reads characters in their packet order. A bare count `4` expands to `4:3:2:1` and therefore reads those four characters in reverse order. A zero count reads until a NUL byte or the packet end. A negative decimal count such as `-2` reads until a NUL byte or two bytes before the packet end. Fixed-length strings also stop at the first NUL. Use `RAW_FMT=%s`; numeric conversion equations are not meaningful for string or byte values.

Conversion and presentation

Column	Header	Value / new-row default	Meaning

`RAW_FMT`	Required	printf-style format / empty	Display format for the decoded raw value; examples: `%u`, `%04X`, `%.3f`, `%s`. Match the format to the decoded type.
`ENG_FMT`	Required	printf-style format / empty	Display format for engineering values, normally floating-point, such as `%.4f`.
`ENG_EQ_TYPE`	Required	Equation ID / `0`	Conversion from raw to engineering value. `0` is identity.
`ENG_K1`, `ENG_K2`	Required, both columns	Floating-point / `0` each	Engineering conversion coefficients. Only two coefficients are available at this stage.
`SCI_UNITS`	Required	Text / empty	Scientific unit label, such as `degC`, `V`, or `Pa`. The generic raw and engineering unit labels are `Raw` and `Value`.
`SCI_FMT`	Required	printf-style format / empty	Display format for scientific values, normally floating-point, such as `%.2f`.
`SCI_EQ_TYPE`	Required	Equation ID / `0`	Scientific conversion. Its input is controlled by `SCI_FROM_ENG`.
`SCI_FROM_ENG`	Required	Boolean / `0`	`0`: scientific conversion consumes the raw value. `1`: it consumes the engineering value.
`SCI_K1`–`SCI_K8`	Required, all eight columns	Floating-point / `0` each	Scientific conversion coefficients. Fill unused coefficients with zero or leave their cells blank for new rows.
`EXTRA_1`	Optional	Text / empty	Additional decoder/conversion configuration. Generic lookup conversion `12` uses it as a lookup-table filename; mission plugins may assign other meanings.
`DISCRETE_LABELS`	Optional	Label expression / empty	Named states or ranges, for example `0=OFF;1=ON;2..4=FAULT`. Labels annotate values; they do not replace numeric conversions.

Let `raw` be the decoded numeric value, `E` the engineering conversion, and `S` the scientific conversion:

Requested value	Result
Raw	`raw`
Engineering	`E(raw)`
Scientific, `SCI_FROM_ENG=0`	`S(raw)`
Scientific, `SCI_FROM_ENG=1`	`S(E(raw))`

Formats control presentation, not stored numeric precision. Discrete label values and inclusive ranges use decimal numbers; use decimal points for fractional values, such as `0.5..1.5=NOMINAL`. Avoid overlapping label ranges and semicolons or equals signs inside label text. Which value mode receives a label depends on the viewer; check it in the target display.

Generic equation IDs and coefficients

In this table, `x` is the input to the selected stage and `K1`, `K2`, etc. refer to that stage's coefficients. Engineering has only `K1` and `K2`; equations requiring more must be used at the scientific stage. Mission-specific equations require their own plugin definitions.

ID	Equation	Coefficient meanings / result
`0`	Identity	`y=x`; coefficients ignored.
`1`	Polynomial	`y=K1 + K2*x + K3*x² + ... + K8*x⁷`. Engineering is linear: `K1 + K2*x`.
`2`	ADC to volts	`y=K1*K2*(x-32768)/65536`.
`3`	ADC to ohms	`y=K2*(x-K1)/(65536-x)`.
`4`	ADC to thermistor temperature	`R=K2*(x-K1)/(65536-x)`; `y=1/(K3+K4*ln(R)+K5*ln(R)³)-273.13`. Requires five coefficients.
`5`	Resistance simulator temperature	`R=K1/(1/x-1)`; `y=K2+K3*R+K4*R²`. Requires four coefficients.
`6`	PRT temperature	Solves `x=K1+K2*y+K3*y²`; selects the root closest to the linear inverse, or smallest magnitude when `K2=0`. Requires three coefficients.
`7`	Pyro / sense-wire temperature	`a=K6-x`; `y=K1+K2*a+K3*a²+K4*a³+K5*a⁴`. Requires six coefficients.
`8`	Thermistor temperature from volts	`R=K2*(x+K1)/(K1-x)`; then the same temperature expression as type `4`. Requires five coefficients.
`9`	Raw bits to float, then polynomial	Reinterprets the low 32-bit unsigned value as IEEE 754 float, then applies the type `1` polynomial to that float. For new float fields, decoder `5` is usually clearer.
`10`	Natural logarithm	`y=K1+K2*ln(x)`; requires positive `x`.
`11`	Signed fixed-point	Interpret the input as a two's-complement integer of `K1` total bits, then divide by `2^K2`. `K2` is the fractional-bit count. Input is limited to 32 bits.
`12`	Lookup table	Uses `EXTRA_1`; coefficients ignored. See lookup-table rules below.
`13`	Logarithm with configurable base	`y=K1+K2*log(x)/log(K3)`. Requires three coefficients, positive `x`, and positive base other than `1`; returns zero for invalid input/base.

The temperature offset `273.13` above is the value used by these conversions. Select coefficients and input modes for the actual calibration; avoid singular denominators and invalid logarithm or square-root domains. An unavailable equation logs a warning and uses identity conversion, so a displayed value alone does not prove that its calibration was loaded.

For lookup conversion `12`, set `EXTRA_1` to a filename such as `temperature.csv`. Place it under `lookupTables/` within the telemetry-definition directory supplied to the application. This base directory is distinct from the database file's directory. The lookup file contains comma-separated `x,y` numeric pairs, without a header. Empty lines and lines beginning with `#` are ignored. Provide distinct input values in consistently ascending or descending order. Conversion interpolates linearly; supply a table covering the expected input range. A missing or empty table leaves the input unchanged.

Limit monitoring

Column	Header	Value / new-row default	Meaning
`LIMIT_TYPE_1`, `LIMIT_TYPE_2`	Optional, independently	Encoded integer / `0`	Up to two legacy limit checks. Zero disables that check. The code selects both value mode and comparison.
`YELLOW_LIMIT_1`, `YELLOW_LIMIT_2`	Optional, independently	Floating-point / unset	Yellow threshold for the corresponding legacy check. An empty cell means no threshold on a new row; numeric `0` is an active zero threshold.
`RED_LIMIT_1`, `RED_LIMIT_2`	Optional, independently	Floating-point / unset	Red threshold for the corresponding legacy check. Same unset-versus-zero distinction.
`LIMITS`	Optional	Single-line JSON object / empty	Scientific-value limits, optionally selected by another mnemonic's scientific value. This is a separate limit format from the legacy columns.

Legacy limit codes

Comparison	Raw	Engineering	Scientific
Less than threshold	`1`	`2`	`3`
Greater than threshold	`5`	`6`	`7`
Equal to threshold	`9`	`10`	`11`
Not equal to threshold	`13`	`14`	`15`

Each check compares against its yellow and red thresholds independently. Legacy less-than and greater-than comparisons are **strict**. Define the low check with type `3`, and the high check with type `7`, for a scientific interval. For example, type `3` with yellow `-10` and red `-20`, plus type `7` with yellow `50` and red `60`, checks both ends of a temperature range. Use these codes without higher-bit persistence flags; legacy checks do not provide dependable configurable persistence. Use JSON `ec` when an excursion count is needed.

JSON `LIMITS`

Put compact JSON directly in a single cell, for example `{"limits":[{"rl":-20,"yl":-10,"yh":50,"rh":60,"ec":2}]}`. Do not wrap the whole cell in quotes or escape its internal double quotes.

JSON member	Location	Meaning
`limits`	Root	Array of limit definitions.

JSON member	Location	Meaning
`cm`	Root, optional	Context mnemonic: numeric `DATA_ID`, or a lookup-key string when the consumer supports name resolution. Numeric IDs work across more consumers.
`rl`, `yl`, `yh`, `rh`	Limit definition, optional	Red low, yellow low, yellow high, red high thresholds in scientific units. Omit a threshold to leave that bound unchecked.
`ec`	Limit definition, optional	Positive excursion count before entering an alarm state; default `2`. `1` responds to the first violation. Returning to the normal state resets the count.
`cr`	Limit definition, optional	Inclusive context range, written as a number for an exact value or a string such as `"1..3"`. A definition without `cr` is the default.

JSON low thresholds trigger on `value <= threshold`; high thresholds on `value >= threshold`. These include the boundary, unlike legacy comparisons. Red thresholds are checked before yellow thresholds.

A contextual example is `{"cm":1001,"limits":[{"cr":"1..3","yh":50,"rh":60,"ec":2},{"yh":70,"rh":80,"ec":2}]}`. When mnemonic `1001` has a scientific value in `1..3`, the first definition applies; otherwise the default applies. Use nonoverlapping context ranges and at most one default. A contextual check needs an available context value; the GUI monitor defers checking while that value is unavailable.

The GUI limit monitor uses a nonempty `LIMITS` object in preference to legacy checks. Avoid configuring both formats for the same mnemonic unless the target consumer explicitly requires it. To clear a JSON limit definition in a patch, use `{}`; a blank cell retains it.

History statistics

Column	Header	Value / new-row default	Meaning
`HISTORY`	Optional	Single-line JSON object / empty	Source mnemonic and retention bounds for history decoder IDs `51`–`56`.

Example cell: `{"mnemonic":"TEMPERATURE","samples":100,"window":60}`.

JSON member	Meaning
`mnemonic`	Source mnemonic name, **without** a leading colon. The current history consumer looks it up as `:NAME`, so the source must have an empty `SUB_SYSTEM`.
`samples`	Maximum retained sample count. Zero means no count bound.
`window`	Retention duration in seconds relative to the latest sample. Zero means no time bound.

Set at least one positive retention bound; with both zero, no samples are retained. If both are positive, both bounds apply. Statistics consume the source's **scientific values**. Rate is the least-squares slope per second, not simply the difference between the last two samples. Use identity conversions on a history row if the statistic already has the desired units.

History values require registration and updates by a history-aware current-value consumer; they cannot be reconstructed from one packet in isolation. Choose a compatible `PACKET_TYPE` for the source and history row, and retain normal required header columns even where byte decoding is unused.

XINA mining and grouping

Column	Header	Value / new-row default	Meaning
`XINA_MINE_TYPE`	Optional	Mining ID or supported lowercase name / `0`	Selects the mining policy. When this column is present, give every active row an explicit valid value; a blank cell is not a valid mining policy.
`XINA_MINE_RAW`	Optional	Boolean / `0`	Requests raw-value mining in addition to scientific values in consumers that support it.
`XINA_MINE_ENG`	Optional	Boolean / `0`	Requests engineering-value mining in addition to scientific values in consumers that support it.
`BUNDLE_ID`	Optional	Signed 64-bit integer / `0`	Bundle/group metadata for supporting consumers and database tooling. Does not alter raw decoding or itself select bundle mining.

ID	Accepted text	Policy
`0`	Use numeric `0`	No mining. Some missions also use it to mark deprecated mnemonics.
`1`	`bin`	Binned summary data.
`2`	`delta`	Change-based data in addition to bins.
`3`	`all`	All samples in addition to bins.
`4`	`blob`	Blob policy; verify support in the target mining consumer.
`5`	Use numeric `5`	Bundle policy; verify the target consumer's bundle configuration.

Text names are case-sensitive. `none` and `bundle` are not accepted text forms; use `0` and `5`. Mining output is application-dependent and may also be controlled by command-line options. In particular, declaring a blob or bundle policy does not ensure that a consumer implements that output.

If the application is configured to ignore mining type `0`, those rows are excluded during loading. When the entire `XINA_MINE_TYPE` column is absent, rows pass that loading filter, but a new row's stored policy still defaults to `0`. Set `3` explicitly when you intend normal full mining.

Conditional decoding

`APPLY_WHEN` uses `DATA_ID=min..max` conditions separated by semicolons. Bounds and referenced IDs use decimal integers, and ranges are inclusive. Example: `1001=1..1;1002=5..10` requires raw value `1` for mnemonic `1001` and raw value `5` through `10` for mnemonic `1002`.

All conditions must pass. Referenced values are decoded from the **same packet**, using their raw values; a missing or undecodable reference fails the condition. Select references that are available in that packet and avoid circular dependencies. This condition gates availability, not calibration or limit selection. Consumers can explicitly bypass apply-when checking, so verify it in the target workflow.

Clock definitions

The loader reads `clocks.txt` from the directory of the **first database file** in the configured load list. Despite its extension, it contains JSON with a root `clocks` array. Each entry can contain:

Member	Meaning
`name`	Clock name matched by the row's `CLOCK` cell.
`epoch`	UTC epoch as an ISO date-time string, for example `"1970-01-01T00:00:00Z"`.
`sub_seconds_per_second`	Ticks per second. Zero or absence uses the decoder's default.
`leap_seconds_offset`	Initial seconds offset subtracted when converting the epoch-relative count to Unix time; default zero.
`leap_seconds`	Optional array of entries with UTC `datetime` and integer `seconds` correction values. Use the mission's clock definition.

A minimal no-leap-correction definition is `{"clocks":[{"name":"unix","epoch":"1970-01-01T00:00:00Z","sub_seconds_per_second":65536,"leap_seconds_offset":0}]}`. A row using this clock sets `CLOCK=unix`. An absent clock file or unresolved name leaves a new row without clock correction; verify that the clock actually loaded before interpreting epoch-based timestamps.

The same first database directory may contain `constants.json` for custom expression decoders. It is a JSON array of objects with `name`, `type` (`int`, `double`, or `string`), and a string-valued `value`, such as `[{"name":"gain","type":"double","value":"1.25"}]`. Constants do not substitute into generic coefficient cells; they are available to supporting custom decoders.

Database patches

Database files load in their configured order. Each has its own header, with all required columns. A later row with an existing positive `DATA_ID` updates that definition; a new positive ID adds a definition. To delete a row, put the negative of its ID in `DATA_ID`, for example `-1001`. A deletion row needs no other values, but keep the file's tabular shape.

For updates:

- Blank cells generally preserve existing values. Write explicit `0` to reset numeric or Boolean fields. A cell containing `""` clears ordinary string fields such as `DESCRIPTION`; it does not clear JSON cells.
- **Always repeat the intended `DECODE_ID1` and `DECODE_ID2` in an update row.** Layout is reinterpreted on each update; blank decoder cells are not a reliable way to preserve it. Retain or explicitly update the applicable `ARRAY_LENGTH` and `SAMPLE_PERIOD` as well.
- Repeat a valid `XINA_MINE_TYPE` if that column is present. A blank mining cell can reject the row rather than preserve its policy.
- A blank legacy threshold retains the prior threshold. Set its `LIMIT_TYPE_n=0` to disable that check. Use `{}` to clear `LIMITS` or `HISTORY` JSON objects.

- To rebuild a definition completely from defaults, delete its ID and then add a full definition with the positive ID. This also removes old name aliases.

Missing required header columns stop loading with an error. Float/double widths and mismatched explicit time/packet-offset counts are also rejected. Other problems may only produce warnings or skipped values: zero or invalid IDs are skipped, malformed `LIMITS` / `HISTORY` JSON is not applied, and unavailable decoders produce no value. Check application logs and expected mnemonic values rather than treating a successful application startup as full file validation.

Database text exported by the current stream writer does not preserve every accepted field: it omits `BUNDLE_ID`, `XINA_MINE_RAW`, and `XINA_MINE_ENG`, and represents array layouts as expanded offsets rather than retaining `ARRAY_LENGTH` and `SAMPLE_PERIOD` columns. Keep the original input file as the authoritative source when these fields matter.

Usage example

tmdatabase-example.tsv is a complete tab-delimited file with all required headers and three unsigned numeric mnemonics. It requires the generic decoder and conversion plugins. Its IDs and APID are illustrative; replace them with the target application's assignments and packet layout.

Key / ID	Definition	Expected result
`demo:TEMPERATURE` / `1001`	APID `100`, two-byte big-endian value at offset `12`; scientific polynomial `-50 + 0.1*raw`; scientific units `degC`.	Packet bytes `02 EE` give raw `750`, engineering `750`, scientific `25.0 degC`. Legacy low/high checks use strict scientific thresholds.
`demo:MODE` / `1002`	Same APID, one-byte unsigned value at offset `14`; identity conversions; labels `0=OFF;1=ON;2=FAULT`.	Byte `01` gives value `1`, labeled `ON` in a supporting display.
`demo:SAMPLES` / `1003`	Same APID, three two-byte big-endian values beginning at offset `16`; period `0.1` seconds; identity conversions.	Bytes `00 0A 00 14 00 1E` give values `10`, `20`, `30` at packet time plus `0`, `0.1`, `0.2` seconds.

The packet must include its headers and extend through byte offset `21` for all three rows to decode. Configure the application's packet reader separately so these packets are recognized as type `100`. Add the TSV path to the application's database-file configuration and supply the telemetry-definition directory if lookup tables or mission plugins require it.

For a new mnemonic, select a unique key and ID, identify its packet type and complete-packet offset, choose its decoder and byte layout, then configure conversion, units, and formats. Add conditions, limits, history, or mining policy only as required. Verify the raw value against known packet bytes first, then check conversions and any conditional or monitoring behavior.

Related source references

- Database loader and field mapping and text field parsing.
- Generic decoders and decoder/equation identifiers.
- Generic conversion definitions and conversion equations.
- Legacy limit checks, JSON limit definitions, and history retention.

Metadata File

Mnemonic Limit Definitions

This page defines the format of the limit definitions used by the Data Viewer application.

A Limit Definition is a JSON object that describes limit thresholds for a single mnemonic telemetry value. The operator/user will be notified when a limit is triggered.

The JSON object is provided in the `LIMITS` column of the mnemonic definition text file (e.g. `DRAMS_TM_Database.txt`).

The top level object supports the following fields:

Field	Required	Value Format	Meaning
`limits`	Yes	JSON Array	Array of 1 or more limit objects. Each object defines a set of limit thresholds. If a Limit Definition does not define a context mnemonic, the array should contain a single object.
`cm`	No	String or integer	Context Mnemonic. The optional mnemonic that is used to determine if a limit should be checked. For example, you may want to only check a current monitor mnemonic's limit when a temperature is between 25 and 30 celsius. In that case, the temperature mnemonic would be the context mnemonic. Can be provided as the mnemonic name or numeric ID e.g. `MMC.HK:WRP_CURRENT` or `5204`.

Each limit object supports the following fields:

Field	Required	Value Format	Meaning
`cr`	No	Single number or range syntax	Context range. Only used if `cm` is provided at the top level. If the context mnemonic equals this value or falls within this range, the limit will be checked. If `cm` is provided but `cr` is not, the limit object will be used as the "default" limit. Single number example: `1.5` the limit will be used when the context mnemonic's value equals `1.5.`. Range example: `"5.2..10.3"` the limit will be used when the context mnemonic's value falls between 5.2 and 10.3, inclusive.

Field	Required	Value Format	Meaning
`ec`	No	Integer	Excursion count. Number of consecutive telemetry values that must exceed a threshold before the limit is triggered. Defaults to 2 if not provided.
`yh`	No	Number	Yellow high threshold. Triggers a yellow high alarm when the telemetry value is greater than or equal to this value for the required excursion count.
`yl`	No	Number	Yellow low threshold. Triggers a yellow low alarm when the telemetry value is less than or equal to this value for the required excursion count.
`rh`	No	Number	Red high threshold. Triggers a red high alarm when the telemetry value is greater than or equal to this value for the required excursion count.
`rl`	No	Number	Red low threshold. Triggers a red low alarm when the telemetry value is less than or equal to this value for the required excursion count.

Notes:

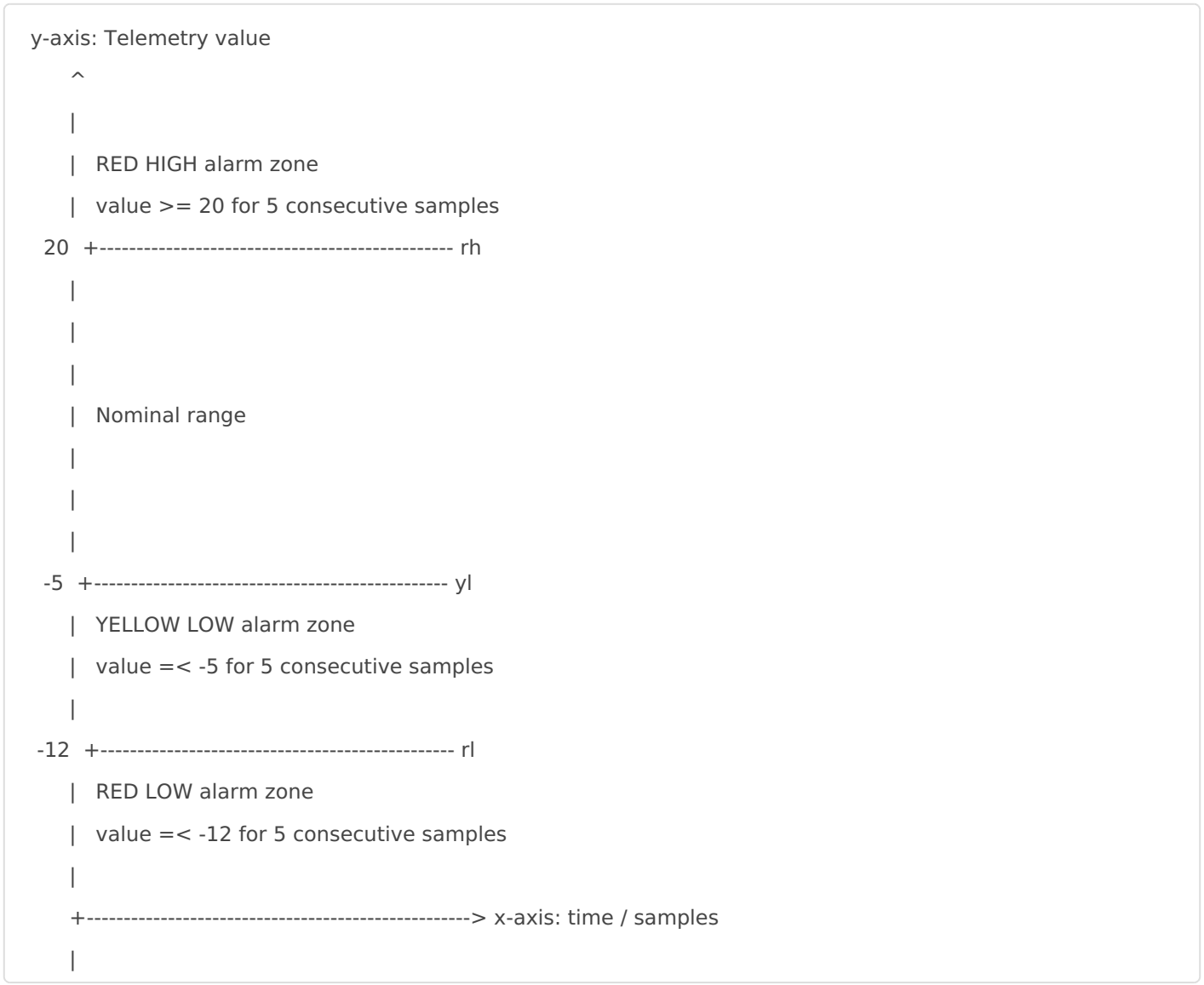
- At least 1 of the threshold values must be defined
- Threshold will be compared against the telemetry value's converted/science value

Example A:

- Limit definition for a `BOARD_TEMP` mnemonic
- Single limit with no context mnemonic
- Yellow high undefined
- Excursion count of 5 means the value must exceed the threshold for 5 consecutive values

```
{
  "limits": [
    {
      "ec": 5,
      "rh": 20,
      "yl": -5,
      "rl": -12
    }
  ]
}
```

The below visualizes how the limit thresholds are evaluated:



Example B:

- Limit definition for a `CURRENT_MONITOR` mnemonic
- `BOARD_TEMP` context mnemonic is used to determine which limit should be checked
- Defines a "default" limit that is used when none of the context ranges are satisfied
- Defines a limit with a context range using the range syntax. If the `BOARD_TEMP` is [0, 25], then this limit will be used.

```
{
 "cm": "BOARD_TEMP",
 "limits": [
  {
   "rh": 2.5,
   "yh": 2.0
  },
  {
   "cr": "0..25",
```

```
    "ec": 3,
    "rh": 2.3,
    "yh": 1.8
   }
  ]
}
```

Example C:

- Limit definition for a `VOLTAGE_MONITOR` mnemonic
- `MODE` context mnemonic is used to determine which limit should be checked
- If `MODE`'s value does not match any of the configured limits' context ranges, no limit will be checked. Note how the context range value is provided as a number.

```
{
 "cm": "MODE",
 "limits": [
   {
    "cr": 0,
    "rh": 1.5,
    "yh": 1.0
   },
   {
    "cr": 1,
    "ec": 2,
    "rh": 5.0,
    "yh": 4.5
   },
   {
    "cr": 2,
    "ec": 4,
    "rh": 10.0,
    "yh": 9.5
   }
 ]
}
```

Expected Values Definition

The Expected Values Definition file is a CSV file that is used by the c_expectedvaluechk Python tool.