

Logger

Generated on October 1, 2026

A **LOGGER** keeps the history of variables. Each time one of its variables changes, it writes a row with the new value and the time, in a database file of its own. Formulas and dashboards read that history back through the history functions of Amtiri Script, and a [Logger Replicator](#) copies it to another instance.

Module	What it does	Typical use
Logger (<code>LOGGER</code>)	Writes a row each time one of its variables changes	Trends on a dashboard, daily averages and totals, the history of a module's status

The module also has the settings every module shares: whether it is enabled, its name, its status and message variables and their thresholds. They are described in [Settings every module shares](#).

Logger

Settings

Key	Setting	Type	Default	Meaning
<code>code</code>	Code	text	none	Names the logger: its file, and the name the history functions and replicators use. Required
<code>variables</code>	Variables	variable codes, separated by commas	none	The variables whose changes are written. Empty writes nothing

The code

- **It names the file:** `loggers/<code>.sqlite` in the instance home, with the code in lower case. `PLANT_LOG` writes into `loggers/plant_log.sqlite`.
- **It is required.** A `LOGGER` without one does not run. When the configuration loads, its status reads `Configuration invalid: A LOGGER module needs a code: it names the database file.` A Start loads the configuration and then opens the modules at once, so on a Start that message is replaced straight away by the opening error, which begins `Database unavailable: .`
- **It must make a file name.** A code holding `/`, `\` or `..` is refused the same way. Use letters, digits and `_`.
- **Case does not count.** `PLANT_LOG` and `plant_log` are the same logger and the same file, so give each `LOGGER` a code of its own.
- **The file outlives the module.** Deleting a `LOGGER` leaves its file, and a `LOGGER` created again with the same code carries on with the same history. Changing a logger's code starts an empty file and leaves the old one in `loggers/`.

What is written

Each row holds three things:

Column	Holds
variable	The variable's code, in capital letters
timestamp	When the logger was told of the change, in UTC, written <code>yyyy-MM-dd HH:mm:ss.SSS</code> , such as <code>2026-09-29 14:05:07.250</code>
value	The value, as text in the engine's own notation; the history functions turn it back into a number, a text, a list or whatever it was

- **One row per change.** A row is written each time the engine reports that one of the variables changed. A write that leaves the value as it was is not a change, and writes no row. A value that does not change for a week costs nothing for a week.
- **No value, no row.** A variable that changes to no value writes nothing, so the history holds the value it had before. That is why a module's status variable belongs in the list beside its messages: see [Keeping a history of status and messages](#).
- **An announcement is logged too.** When a Modbus master, OPC UA client, S7 or EtherNet/IP client opens, it announces its forwarded variables as changed, and a LOGGER that logs one of them writes a row with the same value; see [Forwarders when a module opens](#).
- **Logging never waits.** Rows go into a queue of the logger's own and are written in order, so a slow disk never holds up the formula or the module that changed the value.

The file is a SQLite database with one table, `log`, holding those three columns and indexed by variable and time. While the logger is open, SQLite keeps its latest rows in `.sqlite-wal` beside the file, so copy a logger's files with the engine stopped. A [configuration transfer](#) can carry a logger's history from one instance to another.

Warning

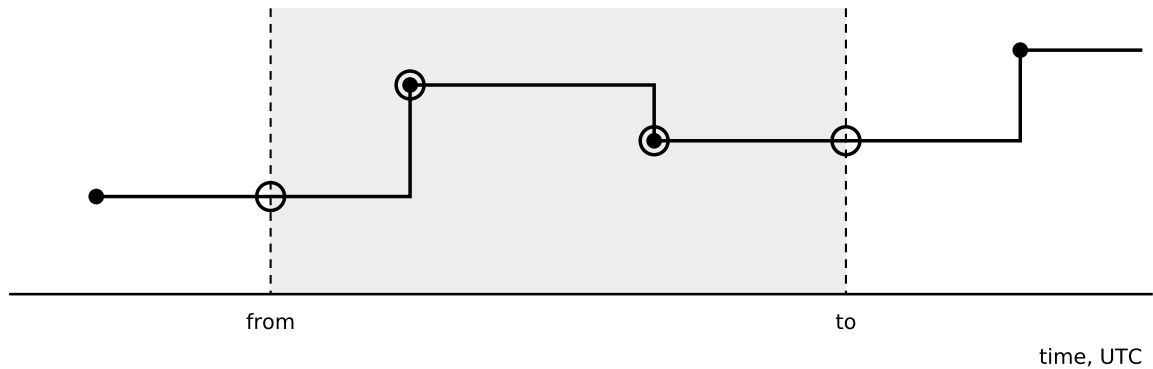
Rows are never deleted. The file grows with every change for as long as the logger runs: a variable that changes every second adds 86 400 rows a day. Log what you will read back, and keep fast-moving values out of a logger unless their history matters. Put the instance home on a disk with room to grow.

When the database fails

- **At the start.** A file that cannot be opened, such as one on a read-only disk, keeps the module from opening, with its reason as the error. The logger stays in error until the engine is started again.
- **While running.** A row the database refuses is dropped, not kept for later, and counted: `Database unavailable: {n} rows dropped`, rewritten at most once a second. The next row tries the database again, so logging resumes by itself when the database is back, but the error stays until the next Start.
- **A long queue.** When 5 000 rows are waiting to be written, the module warns `Logging queue backlog: {n} row(s) waiting`, until the queue is down to 1 000.

Reading the history back

- a row: the value changed, and holds until the next row
- what `history(..., from, to)` returns



Amtiri Script reads a logger by its code:

- `history(CODE, "VARIABLE", from, to)` gives the samples of one variable over a window, oldest first: the value in force at `from`, every row inside the window, and the last value again at `to`. A value holds from its row until the next, which is what the time-series functions, such as `tsAverage`, weigh it by. The logger's code is written bare, not in quotes; the variable is text, in any case.
- `loggerValueAt` and `loggerValuesAt` read the value in force at one instant.
- Times are UTC, like the rows: `now()` gives the current UTC time, and `addHours(now(), -24)` the same time a day earlier.

A formula that reads history names the variable as text, so no change of that variable recalculates it. Give it a [Scheduler](#) definition to recalculate it on a clock.

Receiving replicated history

A **LOGGER** can also be the target of a [Logger Replicator](#) on another instance. The rows arrive with their own UTC timestamps and go straight into the file, a whole batch at a time; rows the file already holds are dropped. The target must be enabled, and it should log nothing of its own: see [A dedicated target logger](#).

Messages

The complete list is in [Module messages](#).

When	Level	Text
Loading: the code is missing or cannot name a file, or the file cannot be created. A Start replaces it at once with the opening error below	ERROR	Configuration invalid: {reason}
Opening: the database opened	INFO	Database {file} opened
Opening: the database cannot be opened. The module does not run	ERROR	Database unavailable: {reason}
Running: rows could not be written and were dropped. The count is rewritten at most once a second	ERROR	Database unavailable: {n} rows dropped
5 000 rows are waiting to be written	WARNING	Logging queue backlog: {n} row(s) waiting

Example

A logger for a tank's level, a flow and a pump, and the status of the PLC module that reads them:

```
Code      PLANT_LOG
Name      Plant history
Variables LIT_301, FIT_501, P101_RUN, PLC1_STATUS
```

The rows go into `loggers/plant_log.sqlite`, and the module's info message reads `Database plant_log.sqlite opened`. A variable `LIT_301_AVG_24H` with the formula

```
tsAverage(history(PLANT_LOG, "LIT_301", addHours(now(), -24), now()))
```

and a Scheduler definition `0 0/15 * * * ?` holds the level's time-weighted average over the last 24 hours, recalculated every quarter of an hour.

Next steps

- [Scheduler](#)
- [Orchester links](#)
- [Settings every module shares](#)
- [Module messages](#)
- [Module catalogue](#)

This page describes Data Orchester engine 6.12.0.