Amtiri

AMTIRI SCRIPT

Amtiri Script

Generated on November 25, 2026

Amtiri Script is the formula language of Data Orchester. A calculated variable, a Merger's filter, the condition that sends a mail and every changing property of a dashboard are written in it. This chapter says where scripts run, what each place offers and how a result becomes a variable's value. The chapters after it describe the language itself.

What a script is

A script is **one expression**. It reads values, combines them with operators and functions, and gives back one result:

```
LEVEL * 2 # 85.0
if(PUMP_ON, "Running", "Stopped") # "Running"
SETPOINTS.low < LEVEL < SETPOINTS.high # true
```

- **No statements.** There is no `;` and there are no keywords. `if`, loops and lookups are functions, and `(a, b)` evaluates each expression in turn and gives the last one's result.
- **Names are variable codes.** `LEVEL` reads the current value of the variable whose code is `LEVEL`. Names are not case-sensitive: Data Orchester upper-cases every name, so `level` reads `LEVEL` too.
- **A script never writes a variable.** The engine writes a variable formula's result; modules and the dashboards' input widgets write the rest. See why there is no assignment.
- **The same functions everywhere.** One environment holds every function, wherever a script runs. Only the special names differ from place to place, and three dashboard functions act only in an Action button's script.

Where scripts run

Place	Runs when	Special names	What the result does
Variable formula	A variable it names changes, or a Scheduler ticks it	Its own code: its previous value	Becomes the variable's value
Initial value	Once at each start, for a variable that is not persistent or has no saved value	—	Becomes the variable's first value
Merger filter	An input of the Merger changes	`_VALUE_`: the input's new value	`true` writes that value to the output
Mailer enabler	The trigger variable changes	—	`true`, or an empty enabler, sends the mail
WebService Out enabler, URL and body	The trigger variable changes	—	`true`, or an empty enabler, sends; the URL and the body are text
Dashboard widget property	A variable it names changes	—	The property: a label, the visibility, an offset, the styles
Formula inside a panel	As a widget property	The panel's parameters, replaced by the bound codes	As a widget property
Action button script	The button is clicked	`_VIEW_`, `_CALLER_`	Shown in a dialog when the button's "Display result" is on
Peer `script.execute`	A peer asks for it	—	Returned to the peer

What each place does with a script that fails is in Errors and missing values.

Variable formulas

The **Formula** of a variable in the Program. The engine evaluates it whenever one of the variables it names changes, and writes the result as the variable's value.

- **Its own code is its previous value.** A formula may read the variable it calculates, and doing so does not make it a trigger of itself. Until the variable has been written once, that read fails: guard it with evl.
- **One at a time.** Formulas run one after another on the engine's formula thread, never two at once.
- **Types.** A variable that declares a **Type** converts the result to it. A result that cannot be converted is not written, and a warning with the variable's code goes to the log.

Initial values

The **Initial value** of a variable that is not persistent is evaluated once at each start, and becomes the variable's first value. A persistent variable starts from its saved value instead, and from its initial value only when it has none saved, as at its first start; that value is then saved. A persistent variable saved as `null` restarts at its initial value too, since the saved file cannot tell `null` from nothing. Before engine 6.18.8 a persistent variable with nothing saved started with no value. The engine does not evaluate every formula at start: it puts the saved and initial values in place, and then recalculates the formulas that read them.

Initial values are evaluated in no set order. An initial value that reads another variable may find it not yet written, so keep initial values to constants, such as `0`, `false` or `"Stopped"`.

Module expressions

- **Merger filter.** While the filter runs, `_VALUE_` holds the new value of the input that changed. Every other variable is read by its own code. See Logic modules.
- **Mailer enabler.** Evaluated when the trigger variable changes; the mail goes when it gives `true`. The subject and the body are not formulas: they are text, with `{CODE}` placeholders that the Mailer fills with the variables' values. See Mailer.
- **WebService Out.** The enabler, the URL and the body are all expressions. A fixed URL is written in quotes, and a body is built by joining text and values. See WebService Out.

```
"{\"tag\": \"" + TAG + "\", \"level\": " + LEVEL + "}" # "{\"tag\": \"P-101\", \"level\": 42.5}"
```

Dashboards

- **Widget properties.** A label, the visibility, the position and size offsets, the styles, a chart's values: each property that holds a formula is evaluated again when a variable it names changes. There is no timer, so a clock is shown by a variable that a Scheduler recalculates, not by a label that reads `now()`.
- **Panels.** A panel declares parameters. When it opens, every parameter name in its formulas is replaced, as text, by the code of the variable bound to it. A binding therefore names a variable in quotes, `{PUMP: "P101_RUN"}`; written bare, `{PUMP: P101_RUN}`, it would paste that variable's current value into the panel's formulas.
- **Action buttons.** The script runs on every click, with `_VIEW_` holding the view and `_CALLER_` the button. It is the only place where openPanel, openDashboard and closePanel act; anywhere else they do nothing and give `null`. Several calls are written as a sequence:

```
(closePanel("detail"), openPanel("detail", "PUMP_DETAIL", 0, -120, {PUMP: "P101_RUN"}))
```

An Action button cannot write a variable. A dashboard writes through its input widgets: the switches, buttons, fields and sliders bound to a variable.

Peers

A peer may ask this instance to evaluate a script, with the `script.execute` action, only when the peer has the **scripting** permission here. The script runs in the same environment as the formulas, and its result goes back to the peer. See Connectivity.

How a result becomes the value

A variable formula's result takes the same path as every other write in the engine:

1. **It is converted** to the variable's declared type, when it has one. A result that does not fit is refused, and a warning names the variable.
2. **It is compared** with the current value. An unchanged value stops here.
3. **It is stored**: saved when the variable is persistent, and logged.
4. **Everything that depends on it hears of it.** The formulas that name the variable are recalculated, the modules that read it (loggers, forwarders, the Mailer, the Merger) are told, and the dashboards showing it redraw.

A formula that gives `null` sets the variable to `null`. A formula that fails leaves the value as it was.

Triggers and the Scheduler

A variable formula is recalculated when a **trigger** changes. By default the triggers are the names the formula contains, except its own code.

- **Every name counts.** A field name after a dot and a name that a loop binds are names too: `STATE.o` names `STATE` and `O`. A name that is not a variable changes nothing, but the log reports it when the configuration loads, as `Trigger 'O' on variable 'OUT' not defined.`
- **Nothing else triggers.** The passing of time does not: `now()` gives a new value only when the formula is recalculated for another reason. A new sample in a logger does not either.
- **The Triggers setting** of a variable lists the variables whose changes recalculate its formula. When it is set, it replaces the names found in the formula.
- **A Scheduler** recalculates the formulas of the variables it lists, on a cron schedule. It hands them no value: each formula runs as if a trigger had changed. This is how a formula that reads the time or a logger's history is kept up to date. See Scheduler.

The formula editor

Every field that takes a formula opens the same editor: the variable's formula and initial value, the module expressions and the dashboard properties.

- **Text and blocks.** The editor shows the formula as text or as blocks, and switches between the two. Both modes hold the same formula: the block mode keeps `//` comments and reads `^` as the engine does.
- **The palette** stays open in both modes. It lists the installation's modules and variables, then the language by section, in the order of the function reference. Selecting an entry, or a block, shows its description, its signature and its parameters.
- **The status line** says whether the formula compiles. It checks the syntax, the function names and the number of arguments. It does not check variable names: the engine reports those when the configuration loads.
- **Apply keeps what it is given.** A formula that does not compile can still be applied: read the status line first.
- **Not in the palette.** There is no assignment block. `exec`, `functions()` and `variables()` exist but are not offered; see Flow.

The sample variables

Every example on these pages is written as `expression # result` and runs against these variables:

Variable	Value	Kind
`LEVEL`	`42.5`	A decimal
`COUNT`	`3`	A whole number
`PUMP_ON`	`true`	A boolean
`TAG`	`"P-101"`	Text
`REGISTERS`	`[16256, 0, 65535, 1]`	A list of whole numbers, as a Modbus master's array collector gives
`SETPOINTS`	`{low: 10, high: 90}`	A record
`START`	`2026-09-29T08:00`	A date and time

Results are written the way the product holds them. A decimal always shows its point, so `85.0` is a decimal and `85` a whole number. Text is in quotes, a record's fields are listed in alphabetical order, and a date and time is written `2026-09-29T08:00`. `error` means the evaluation fails, `syntax error` that the script does not compile, and `*` that any value is right, as for `now()`.

The examples are tested against the release this page describes. Examples that need a running plant, such as a logger's history, are shown as plain text and link to their function's entry, where they are tested too.

How this manual is organised

Part	Pages
Language	This overview; Syntax; Types and conversions; Errors and missing values; Data Orchester functions, which puts the history, time-series, Modbus and control functions to work; and Patterns from real plants
Functions	An index from A to Z; Operators, literals and structures; and a page per palette section: Math, Logic, Values and types, Strings, Dates, Colours, Collections, Flow, History, Time series, Modbus decode, Events and Site

Next steps

- Syntax
- Types and conversions
- Errors and missing values
- Logic modules

This page describes Amtiri Script 5.4.3 as shipped with Data Orchester engine 6.18.8.