Amtiri

AMTIRI SCRIPT

Data Orchester functions

Generated on November 25, 2026

Data Orchester adds 27 functions to the language. The engine brings 23 of them: history, time series, Modbus decoding, control loops and delayed writes. The Site brings four more, for dashboards and coordinates. This chapter puts them to work in plant terms. Each function also has its own entry, linked below, with its signature and tested examples.

Most examples on this page are plain text, because they need a running plant: a logger with history, a Modbus master or a dashboard. The few that use only the core language run against the sample variables, like the rest of this manual.

Group	Functions	What they are for	Where they work
History	history, loggerValueAt, loggerValuesAt, psplit	Reading what a LOGGER recorded; cutting a window into periods	Any formula. A history formula needs a Scheduler to stay current
Time series	tsAverage, tsMax, tsMin, tsFirst, tsLast, tsCenter, tsSplit, tsMerge, tsFilter, tsDistribution	Summing up a series: averages, extremes, buckets, time per state	Any formula
Modbus decode	mbInt, mbIntHL, mbFloat, mbFloatHL, mbDouble	32-bit and 64-bit values out of a list of registers	Any formula; usually a variable fed by a Modbus master
Control	ctrlPID, ctrlPWM, ctrlKalman	Loops that carry their own state from one step to the next	Variable formulas, where the state is kept
Events	delayedSet	Writing a value into a variable once a delay has passed	Variable formulas
Site	toUTM, openPanel, closePanel, openDashboard	Coordinates in metres; opening panels and dashboards	`toUTM` in any formula of a Site; the other three only in an Action button's script

The Site registers its four functions when it starts. An engine that runs without the Site does not know them, and a formula that calls one does not compile there.

History from a logger

A Logger module keeps the history of the variables it lists. Each time one of them changes to a value that is not `null`, the logger adds a sample, stamped with the time in UTC, to its own database. The history functions read that database, and find it by the logger module's **code**.

Reading a window

history gives every sample of one variable over a window of time:

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

- `PLANT_LOG` **is the code of a LOGGER module**, written bare, without quotes. The variable's code is text, in quotes. Case does not matter in either. Written as a method, `PLANT_LOG.history("LIT_301", from, to)` is the same call.
- **The window** is two date-times in UTC: `from`, then `to`.
- **The result is a time series**: a list of `[date-time, value]` pairs, in UTC, oldest first. The values come back as they were stored: a number stays a number, a boolean a boolean.
- **A logger code that no LOGGER module carries** makes the formula fail when it runs.

Part of the series	Pair
First	`[from, the value in force at from]`, when the logger holds a sample at or before `from`
Middle	Every sample logged after `from` and before `to`, oldest first
Last	`[to, the last value]`

A sample logged exactly at `to` is left out. When nothing was logged at or before `from`, the series starts at the first sample inside the window. When the logger holds no sample of the variable before `to`, the result is the empty list, `[]`.

The first and last pairs make the series cover the whole window, so a time-weighted average over it counts every second from `from` to `to`, and not only the stretch between the first and the last change.

Keeping a history formula current

A formula is recalculated when a variable it names changes (see Triggers). A history formula names none that changes on its own. The logger's code counts as one of its names, and the log reports it when the configuration loads, but no variable carries it. A new sample in the logger triggers nothing, and neither does the passing of time.

So give a history formula a Scheduler: a SCHEDULER module recalculates the variables it lists, on a cron schedule.

Variable	Formula	Why
`LIT_301_MAX_24H`	`tsMax(history(PLANT_LOG, "LIT_301", addHours(now(), -24), now()))`	The highest level of the last 24 hours

Scheduler lists	Cron	Runs
`LIT_301_MAX_24H`	`0 */10 * * * ?`	Every ten minutes, on the minute

Read history in a variable, and show that variable on a dashboard. A widget formula runs every time its view redraws, which happens whenever a variable the view shows changes, and each run would read the logger's database again.

The value at an instant

loggerValueAt gives the value one variable had at one instant, and loggerValuesAt the values of several:

```
loggerValueAt("PLANT_LOG", "FQ_501", DAY_END)
loggerValuesAt("PLANT_LOG", ["FQ_501", "FQ_502"], DAY_END)
```

- **The logger's code is in quotes here**, unlike in `history`. The variable codes are text; case does not matter.
- **The value in force at the instant** is the last sample at or before it, as it was stored: a number stays a number. When the logger holds no sample of the variable at or before the instant, the value is `null`.
- `loggerValuesAt` **gives a record** with one field per code: read one with `.FQ_501`.

A pair of day boundaries serves every daily figure. Two variables hold them, and a Scheduler ticks them:

Variable	Formula	Why
`DAY_END`	`date_of(now())`	Midnight UTC at the start of today
`DAY_START`	`addDays(date_of(now()), -1)`	Midnight UTC at the start of yesterday
`FQ_501_YESTERDAY`	`loggerValueAt("PLANT_LOG", "FQ_501", DAY_END) - loggerValueAt("PLANT_LOG", "FQ_501", DAY_START)`	The volume through the meter yesterday, from its totaliser

Scheduler lists	Cron	Runs
`DAY_START`, `DAY_END`	`0 * * * * ?`	Every minute

The two boundaries are recalculated every minute, but their values change only once a day, at midnight UTC. Only a change reaches the formulas that read them, so `FQ_501_YESTERDAY` and every other daily figure below is recalculated only at midnight, without a Scheduler of its own. Until the logger holds a sample from before yesterday, one of the two reads is `null`, and the difference is `null` or fails; evl gives it a default.

The core date functions build these boundaries:

```
date_of(START) # 2026-09-29T00:00
addDays(date_of(START), -1) # 2026-09-28T00:00
addHours(START, -24) # 2026-09-28T08:00
START.addMinutes(90) # 2026-09-29T09:30
```

These are UTC days. Types and conversions explains the time zones, and why conversions to local time belong in widget properties rather than in variable formulas.

Time-series recipes

A time series is a list of `[date-time, value]` pairs, oldest first, in UTC. `history` gives one, and the time-series functions take any list of that shape.

Function	Gives	Notes
`tsAverage(s)`	The time-weighted mean: each value counts for as long as it was in force	Booleans count as 1 and 0, so the result is the fraction of time spent `true`. An interval that starts with a `null` value is left out. `null` with fewer than two samples
`tsMax(s)`, `tsMin(s)`	The largest and the smallest value	Not weighted by time. `null` values are skipped; `null` for an empty series
`tsFirst(s)`, `tsLast(s)`	The first and the last **pair**, `[date-time, value]`	Read the value with `[1]`. `null` for an empty series
`tsCenter(s)`	The value in force halfway between the first and the last sample's time	Numbers only
`tsSplit(s, segments)`	One series per segment	Each starts with the value in force at the segment's start and ends with the value in force at its end
`tsMerge({a: s1, b: s2})`	One series of `[date-time, {a: …, b: …}]`	An entry for every instant of any series, holding the last known value of each. A series that has not started yet is absent from the record
`tsFilter(s, x, condition)`	The samples for which the condition holds	`x` is bound to each pair in turn. Where a run of rejected samples begins, a `[date-time, null]` break is inserted
`tsDistribution(s)`	A record: the time spent at each value, in days	Each interval counts for the value in force during it. The keys are the values written as text
`psplit(from, to, n)`	`n` equal `[start, end]` segments covering the window	`n` must be at least 1. Filed under History

A daily average over the last 24 hours

Variable	Formula	Why
`LIT_301_AVG_24H`	`tsAverage(history(PLANT_LOG, "LIT_301", addHours(now(), -24), now()))`	The time-weighted average level over the last 24 hours

Scheduler lists	Cron	Runs
`LIT_301_AVG_24H`	`0 */10 * * * ?`	Every ten minutes

A level that stood at 2.0 m for 20 hours and at 4.0 m for 4 hours averages 2.33 m, however many samples each stretch produced. For the average of a fixed day, read the window from `DAY_START` to `DAY_END` instead, and the formula needs no Scheduler of its own.

A day in hourly buckets

psplit cuts a window into equal segments, and tsSplit cuts a series along them. forEach then sums up each bucket:

Variable	Formula	Why
`FIT_501_HOURLY`	`forEach(tsSplit(history(PLANT_LOG, "FIT_501", DAY_START, DAY_END), psplit(DAY_START, DAY_END, 24)), bucket, tsAverage(bucket))`	Yesterday's flow as 24 hourly averages, oldest first

- The result is a list of 24 decimals. An hour that lies before the logger's first sample of the variable gives `null`.
- `psplit` makes equal segments of UTC time. The last one ends exactly at `to`.
- `bucket` is a name that `forEach` binds while it runs. Choose one that no variable uses: see Loop and binding names.

- `tsMax(bucket)` gives the hourly peaks instead, and `psplit(DAY_START, DAY_END, 96)` quarter-hours.

Run hours per state

For a boolean, such as a pump's running signal, the time-weighted average is the fraction of the window spent `true`. Times 24, it is yesterday's run hours:

```
tsAverage(history(PLANT_LOG, "P_101_RUN", DAY_START, DAY_END)) * 24
```

For a state held as text, tsDistribution gives the time spent in each state, in days:

```
tsDistribution(history(PLANT_LOG, "P_101_MODE", DAY_START, DAY_END))
evl(tsDistribution(history(PLANT_LOG, "P_101_MODE", DAY_START, DAY_END)).AUTO, 0.0) * 24
```

The first gives a record such as `{AUTO: 0.75, MANUAL: 0.25}`: 18 hours in `AUTO`, 6 in `MANUAL`. The second reads the hours in `AUTO`. A state that did not occur has no field, so `evl` gives it `0.0`.

The keys are the values written as text. A key that is not a name, such as `2` or `true`, cannot be read with a dot. For a numeric state code, turn each sample into a boolean and average that instead:

```
tsAverage(forEach(history(PLANT_LOG, "P_101_STATE", DAY_START, DAY_END), smp, [smp[0], smp[1] == 2])) * 24
```

That gives the hours spent in state 2.

The latest value

tsLast gives the last **pair** of a series, `[date-time, value]`, and tsFirst the first. Read the value with `[1]`:

```
tsLast(history(PLANT_LOG, "LIT_301", DAY_START, DAY_END))[1]
tsLast(history(PLANT_LOG, "LIT_301", DAY_START, DAY_END))[1] - tsFirst(history(PLANT_LOG, "LIT_301", DAY_START,
```

The first is the level at the end of yesterday; the second, how much the level rose or fell over the day. An empty series gives `null`, and `null[1]` fails: wrap the read in evl where the logger may hold nothing yet. For one instant, `loggerValueAt` reads the value with a single query, and also counts a sample logged exactly at that instant.

Bad samples and several series

tsFilter drops the samples a condition rejects, such as the value a transmitter sends while it is faulty:

```
tsAverage(tsFilter(history(PLANT_LOG, "TT_201", DAY_START, DAY_END), smp, smp[1] > -50))
```

Where a run of rejected samples begins, `tsFilter` inserts a `[date-time, null]` break, and `tsAverage` leaves out the time from that break to the next accepted sample. The average therefore covers only the time the transmitter was sound.

tsMerge puts several series on one timeline, for a table or an export:

```
tsMerge({flow: history(PLANT_LOG, "FIT_501", DAY_START, DAY_END), level: history(PLANT_LOG, "LIT_301", DAY_STAR
```

Each entry is `[date-time, {flow: …, level: …}]`, with the last known value of each series at that instant.

Modbus decoding

A Modbus register holds 16 bits. A Modbus master reads registers as whole numbers from 0 to 65535, and an array collector writes a list of them into one variable. A value wider than 16 bits spans several registers, and devices disagree on the order of its words and bytes. The decoding functions read two registers (four for `mbDouble`) from such a list, starting at an offset:

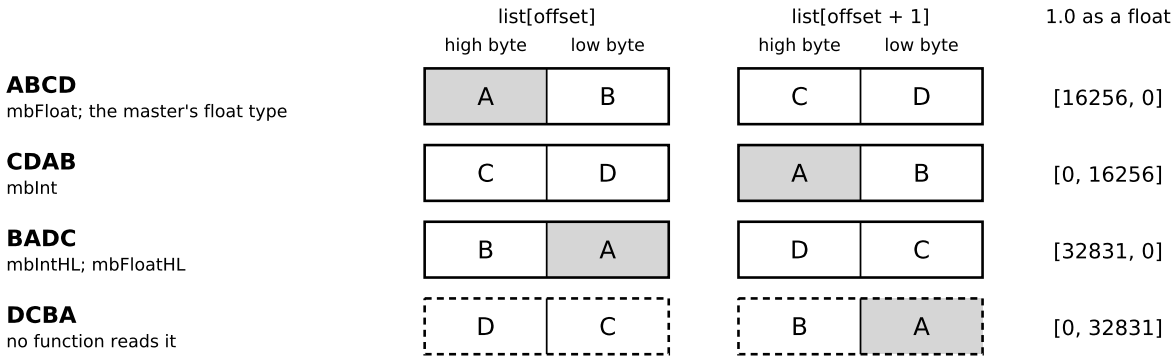
```
mbFloat(PM_101_REGS, 0)
```

- **The list** is a variable written by an array collector, of type `FC03 - Holding registers (int16 array)` or `FC04 - Input registers (int16 array)`.
- **The offset** is the index of the first register, counting from 0. It may be any whole number, including one a formula computes, such as `2 * 3`.

Word and byte orders

Name the four bytes of a 32-bit value A, B, C and D, A being the most significant. Each register holds two bytes, high byte first.

Where the bytes of a 32-bit value sit, A being the most significant



mbDouble reads four registers in the first order: A B, C D, E F, G H. The shaded byte is A.

Function	Registers	Order	The first register holds	Gives
mbFloat	2	ABCD	The high word	A 32-bit decimal
mbFloatHL	2	BADC	The high word, its two bytes swapped	A 32-bit decimal
mbInt	2	CDAB	The low word	A signed 32-bit whole number
mbIntHL	2	BADC	The high word, its two bytes swapped	A signed 32-bit whole number
mbDouble	4	ABCDEFGH	The highest word	A 64-bit decimal

The `HL` functions swap the two bytes inside each register as well as putting the high word first. A collector of type `FC03 - Holding registers (float32)` or `FC04 - Input registers (float32)` reads its two registers in the same order as `mbFloat`, ABCD.

```
mbFloat([16256, 0], 0)           # 1.0
mbFloatHL([32831, 0], 0)         # 1.0
mbInt([34464, 1], 0)             # 100000
mbIntHL([256, 41094], 0)         # 100000
mbDouble([16368, 0, 0, 0], 0)    # 1.0
mbFloat([0, 16256], 0)           # 2.278E-41
```

The last line reads a CDAB float with `mbFloat`. A tiny value such as `2.278E-41`, or a huge one, usually means the words are the other way round.

A 32-bit decimal holds about seven significant digits, and arithmetic shows its exact binary value: two registers that encode `0.1` read as `0.1`, but add `0` and the result is `0.10000000149011612`. A variable whose Type is `double` stores that longer form too.

A worked example

A meter's five holding registers are read by one array collector into the variable `PM_101_REGS`:

Index	Register	Holds
0	17533	Pressure, a 32-bit decimal, ABCD
1	20480	
2	34464	Volume counter, a 32-bit whole number, low word first
3	1	
4	65436	Temperature in tenths of a degree, a signed 16-bit number

Variable	Formula	Why
`PT_101`	`mbFloat(PM_101_REGS, 0)`	`1013.25`: registers 0 and 1, high word first
`FQ_101`	`mbInt(PM_101_REGS, 2)`	`100000`: registers 2 and 3, low word first, so 1 × 65536 + 34464
`TT_101`	`if(PM_101_REGS[4] > 32767, PM_101_REGS[4] - 65536, PM_101_REGS[4]) / 10`	`-10.0`: a register above 32767 holds a negative number

The three formulas recalculate whenever the collector reads a list that differs from the previous one.

Orders no function reads

No function decodes a whole number in ABCD or DCBA order, or a decimal in CDAB or DCBA order. When the device lets you choose its order, choose one the functions read. A whole number in ABCD order can be computed with arithmetic: the first register times 65536, plus the second, and minus 4294967296 when the result is above 2147483647. With the sample variable `REGISTERS`:

```
REGISTERS[2] # 65535
if(REGISTERS[2] > 32767, REGISTERS[2] - 65536, REGISTERS[2]) # -1.0
REGISTERS[3] * 65536 + REGISTERS[2] # 131071.0
with(REGISTERS[2] * 65536 + REGISTERS[3], raw, if(raw > 2147483647, raw - 4294967296.0, raw)) # -65535.0
```

The second line is the signed reading of one 16-bit register. The third reads registers 2 and 3 low word first, as `mbInt (REGISTERS, 2)` does, and gives the same `131071`, as a decimal. The last reads them high word first, as a signed number. The result of arithmetic is always a decimal; to_int turns it back into a whole number where one is needed. Write `4294967296.0` with its point: a whole-number literal must fit in 32 bits.

Control loops

> **Warning**
>
> Deprecated since engine 6.15.0. The PID, PWM and Kalman filter modules do the same work on their own clock, with plain outputs and no state record, and without the defects listed below. The functions still run, and each formula that calls one is named in a warning when the configuration loads. The formula palette no longer offers them; they are listed outside the palette . They will be removed in the next major version.

The control functions compute one step of a loop each time their formula runs: a PID controller, a pulse-width modulator and a Kalman filter. None of them keeps anything in memory between steps. Each returns its whole state as a record, and takes the previous state back in on the next step.

The state variable

The state lives in the variable whose formula makes the call. That formula passes its own code as the state: a formula's own code reads the variable's previous value, and does not make the variable a trigger of itself. A second variable reads the output field:

Variable	Formula	Why
`TIC_201_PID`	`ctrlPID({p: 2, i: 0.1, d: 0, mino: 0, maxo: 100}, TIC_201_PID, TIC_201_SP - TT_201)`	The controller's state; runs when the setpoint or the temperature changes
`TIC_201_OUT`	`TIC_201_PID.o`	The output, from 0 to 100 %, for a forwarder or a dashboard

- **Never inline.** A formula such as `ctrlPID({…}, SOMETHING, TIC_201_SP - TT_201).o` gets no previous state, so it starts again at every step and its output stays at `0`.
- **On the first step**, while the state variable has never been written, each function starts a fresh state.
- **Settings keys are lower case**, exactly as shown. A key in other capitals counts as missing, and a missing key, other than the PWM's optional `mintoggle`, makes the formula fail.
- **The clock.** Time comes from the engine's own clock, in nanoseconds. The fields `ts`, `cs` and `lt` hold it as whole numbers.
- **The Type.** A state variable may leave its Type empty. If it declares one, give the clock fields the type `long`: `{a: double, e: double, o: double, ts: long}` for a PID, `{cs: long, lt: long, o: boolean}` for a PWM and `{o: double, p: double, ts: long}` for a Kalman filter. `DateTime` and `int` do not fit the clock, and every step would be refused.

PID

ctrlPID takes its settings, the previous state and the error, the distance from the setpoint.

Setting	Meaning
`p`	Proportional gain
`i`	Integral gain, per second
`d`	Derivative gain, in seconds
`mino`, `maxo`	The output's limits. The integral term is held within them too, so it cannot wind up

Each step measures the time since the previous one. The integral term `a` grows by `i × error × time`, and the output `o` is `p × error + a + d × (change in error) / time`, held between `mino` and `maxo`. The state is `{a, e, o, ts}`: the integral term, the error, the output and the clock.

- **The sign of the error** sets the direction. `SP - PV` raises the output while the measurement is below the setpoint, as heating needs; for a process that the output drives down, such as cooling, pass `PV - SP`.
- **The first step** gives an output of `0`, whatever `mino` says, and starts the clock.
- **A missing error**, a `null` or a reading that fails, keeps the state as it was.
- **The step is event-driven.** The controller steps only when its formula runs: here, when the setpoint or the temperature changes. A Scheduler tick, such as every second (`* * * * * ?`), gives it regular steps as well.

PWM

ctrlPWM turns a demand into an on/off output that is on for a share of each period, for a heater or a dosing pump:

Variable	Formula	Why
`TIC_201_PWM`	`ctrlPWM({period: 60, minin: 0, maxin: 100, mintoggle: 5}, TIC_201_PWM, TIC_201_OUT)`	The modulator's state
`HTR_201_CMD`	`TIC_201_PWM.o`	`true` or `false`, for a coil forwarder

Scheduler lists	Cron	Runs
`TIC_201_PWM`	`* * * * * ?`	Every second

Setting	Meaning
`period`	The cycle, in seconds
`minin`, `maxin`	The demand that means always off, and the one that means always on
`mintoggle`	Optional: the shortest time between two switchings, in seconds. Without it, 0

The share of time on is `(demand - minin) / (maxin - minin)`, held between 0 and 1; when `minin` equals `maxin` it is 0. Each period starts with the output on for that share of the period, then off. The state is `{cs, lt, o}`: when the cycle started, when the output last switched, and the output itself, a boolean. The first step starts a cycle with the output off, and a missing demand keeps the state as it was.

A PWM needs a Scheduler. The output can switch only when the formula runs, and a steady demand triggers nothing. The tick is the resolution of the pulse: with a tick every second, the output switches only on a tick, so a 60-second period has 60 steps.

Kalman filter

ctrlKalman smooths a noisy measurement:

Variable	Formula	Why
`FT_301_KF`	`ctrlKalman({q: 0.01, r: 4}, FT_301_KF, FT_301)`	The filter's state; runs on every new reading
`FT_301_FILT`	`FT_301_KF.o`	The smoothed flow

Setting	Meaning
`q`	How fast the true value may drift: its variance per second
`r`	How noisy the measurement is: its variance

The state is `{o, p, ts}`: the estimate, its variance and the clock. The first reading becomes the first estimate. A missing reading keeps the state as it was; if the very first reading is missing, the estimate starts from `0` and moves towards the readings that follow.

Let the reading trigger the filter, and give it no Scheduler: each step takes in the current reading once, so a tick would count an unchanged reading again.

After a restart

The clock in `ts`, `cs` and `lt` counts nanoseconds from a starting point of its own, which can change from one run of the engine to the next. A state variable that is **Persistent** is saved at every step, since its clock changes each time, and after a restart it brings back the previous run's clock:

- **PID.** The first step measures its time against the old clock. A time of zero or less is taken as one nanosecond, and with a derivative gain the output then jumps to one of its limits for that step. A very long time makes the integral term jump, up to a limit.
- **PWM.** When the saved cycle start lies ahead of the new clock, no new cycle can begin, and the output can stay on, whatever the demand, until the new clock catches up with the saved one.
- **Kalman filter.** Only the weight given to the first reading after the restart changes.

Leave control state variables not persistent. After a restart, the first step then starts a fresh state: a PID's output at `0`, a PWM's output off, and a Kalman filter's estimate from the first reading.

Delayed writes

delayedSet writes a value into a variable once a delay has passed. The write waits in the engine's event queue, and the formula carries on at once:

Variable	Formula	Why
`P_102_QUEUED`	`delayedSet("P_102_CMD", P_101_CMD, 5000)`	Starts pump 2 five seconds after pump 1, from one command

- **One write per evaluation.** The formula runs on every change of a variable it reads, and once at start, and each run queues one write. Here every change of `P_101_CMD` reaches `P_102_CMD` five seconds later, in order.
- **The value is taken when the formula runs.** A later change of `P_101_CMD` does not alter a write already waiting.
- **The result** is `true` when the write was queued. It is `false` when the variable does not exist, when the delay is not from 0 to 604800000 milliseconds (seven days), when the queue is full or when the engine is stopped. The variable holding the formula keeps that result, so a dashboard can show it.
- **The target is named as text.** The Program diagram draws no link to it, and renaming the variable does not rewrite the text.
- **A stop or a restart drops** every write still waiting. Nothing is kept across a restart.
- **Never delay a write into a variable the formula reads.** Each write changes that variable, so the formula runs again and queues another write, for ever. The engine does not detect it. What finally stops it is the queue's limit of 100,000 waiting events.

Dashboards and panels

Three Site functions navigate a dashboard. They act only in an **Action button's script**, which runs when the button is clicked. Anywhere else, in a variable formula or a widget property, they do nothing and give `null`.

Function	Does
openPanel `(id, code, dX, dY [, bindings])`	Shows the panel `code` over the view, centred `dX` and `dY` canvas units from the centre of the button; a negative `dY` places it above
closePanel `(id)`	Takes away the panel showing under `id`, wherever it was opened from
openDashboard `(code[, bindings])`	Replaces what the view shows with the dashboard `code`. From a button inside a panel, it replaces the panel's content

```
openPanel("detail", "PUMP_DETAIL", 0, -120, {PUMP: "P_101_RUN", SPEED: "P_101_SPEED"})
(closePanel("detail"), openDashboard("STATION_2"))
openDashboard(if(ALARM_ZONE == 2, "STATION_2", "STATION_1"))
```

- **One panel per id.** Asking again for an id that shows the same panel in the same place changes nothing, and asking for another panel replaces it. A click elsewhere does not close a panel opened by `openPanel`; `closePanel` does, and so does leaving the dashboard.
- **Bindings name variables in quotes.** `{PUMP: "P_101_RUN"}` binds the panel's parameter `PUMP` to the variable `P_101_RUN`: when the panel opens, `PUMP` in its formulas is replaced by that code. Written bare, `{PUMP: P_101_RUN}` would paste the variable's current value, such as `true`, into the formulas instead.
- **Only declared parameters are bound.** A binding for a parameter the panel does not declare is dropped. A parameter left unbound is read as a variable of its own name.
- **Several calls make a sequence**, `(a, b)`: each runs in turn.
- **The code can be computed.** The third line picks a dashboard from a variable when the button is clicked.
- **A button does not write variables.** The language has no assignment. A dashboard writes through its input widgets: switches, buttons, fields and sliders bound to a variable.

See Buttons for the Action button and its script, and Regions and panels for panels, their parameters and the Region widget, which opens a panel without a script.

Coordinates

toUTM turns a latitude and a longitude, in decimal degrees on WGS84, into Universal Transverse Mercator coordinates: a record with `easting` and `northing` in metres, and `zone`, the zone number and latitude band as text.

```
toUTM(-33.45, -70.66)    # {easting: 345713.051560063, northing: 6297591.954234887, zone: "19H"}
```

Metres make distances a matter of subtraction, for two points in the same zone. With a remote unit that reports its position:

Variable	Formula	Why
`RTU_7_UTM`	`toUTM(RTU_7_LAT, RTU_7_LON)`	The unit's position in metres
`RTU_7_OFFSET_M`	`sqrt((RTU_7_UTM.easting - 345713.0) ^ 2 + (RTU_7_UTM.northing - 6297592.0) ^ 2)`	How far it is, in metres, from where it was installed

Next steps

- Patterns from real plants
- Errors and missing values
- Logger
- Scheduler
- Modbus
- Function index

This page describes Amtiri Script 5.4.1 as shipped with Data Orchester engine 6.15.0.