Amtiri

AMTIRI SCRIPT

Patterns from real plants

Generated on November 25, 2026

This chapter collects the idioms that recur in working plants' configurations, with their codes replaced by invented ones, and the functions that complete them. Plant formulas need few functions: across more than a thousand of them, eleven cover everything — if , evl, to_double, or, to_int, format, not, scale, toLocale, now and clamp.

The examples run against the sample variables. Formulas that read a plant's own variables are shown as plain text.

A signed register in engineering units

A Modbus master gives each register as a whole number from 0 to 65535. A device that sends a signed 16-bit number, such as a temperature below zero, sends a negative one as a value above 32767. Subtract 65536 from those:

```
REGISTERS[2] # 65535
if(REGISTERS[2] > 32767, REGISTERS[2] - 65536, REGISTERS[2]) # -1.0
if(REGISTERS[0] > 32767, REGISTERS[0] - 65536, REGISTERS[0]) # 16256
if(REGISTERS[2] > 32767, REGISTERS[2] - 65536, REGISTERS[2]) / 10 # -0.1
```

scale then maps the raw range onto engineering units: `scale(value, rawMin, rawMax, unitMin, unitMax)`. For a transmitter whose register runs from 0 to 1000 over 0 to 14 bar:

```
scale(500, 0, 1000, 0, 14) # 7.0
scale(if(REGISTERS[2] > 32767, REGISTERS[2] - 65536, REGISTERS[2]), 0, 1000, 0, 14) # -0.014
scalein(1500, 0, 1000, 0, 14) # 14.0
```

The plant form reads one register of an array collector's list, here `AI_REGS`:

```
scale(if(AI_REGS[4] > 32767, AI_REGS[4] - 65536, AI_REGS[4]), 0, 1000, 0, 14)
```

`scale` does not hold its result within the unit range: a reading past the raw range gives a value past the unit range. scalein holds it within. For values that span two registers, see Modbus decoding.

Guarding a reading that may not exist yet

After a start, a variable that is neither persistent nor given an initial value exists only once something has written it. Until a Modbus poll has answered, reading its variable fails, and so does every formula that reads it: see Never-written variables. evl gives a default in place of the failure; nvl does not help, because the read fails before `nvl` sees a value.

The plant form converts at the same time: `evl(to_double(PT_101), 0.0)` gives the reading as a decimal, and `0.0` while there is none, or when it holds text that is not a number.

```
evl(to_double(LEVEL), 0.0) # 42.5
evl(to_double(PT_101), 0.0) # 0.0
nvl(PT_101, 0.0) # error
evl(to_double(TAG), 0.0) # 0.0
evl(PT_101 > 4.5, false) # false
evl(LEVEL != null, false) # true
evl(PT_101 != null, false) # false
```

- **"Is there a reading?"** is `evl(PT_101 != null, false)`: `true` only when the variable exists and holds a value. It gives a widget a grey "no data" state; see Multi-state indicators.
- **The default is a value like any other.** A pressure shown as `0.0` because nothing has arrived looks like a real zero. Where the difference matters, show "no data" on its own.
- **Keep the guarded expression small.** `evl` hides every error, a misspelt code included, which would then give the default for ever.

Formatting a value for a label

format lays a value out with Java's format specifiers: `%.1f` for a decimal with one digit after the point, `%d` for a whole number, `%s` for any value. The plant form guards the reading too:

```
"%.1f bar".format(evl(to_double(PT_101), 0.0))
```

- **The decimal separator of `%f`** follows the instance's language, `application.locale`: a point with `en`, the default, and a comma with `es`. See the Configuration reference. A style needs a point whatever the language, so give a style numbers, not formatted text.
- **`%d` refuses a decimal and `%f` a whole number.** Arithmetic always gives a decimal, so convert with to_int or to_double first.
- **`%s` and `+` write a number as the language holds it**, always with a point, and without rounding.
- **`+` joins text only when its left side is text.** Start with the text, or convert the number with to_string.

```
"%d".format(COUNT) # "3"
"%s: %d pumps".format(TAG, COUNT) # "P-101: 3 pumps"
"%d".format(COUNT + 1) # error
"%d".format(to_int(COUNT + 1)) # "4"
"%.1f".format(COUNT) # error
"%s m".format(LEVEL) # "42.5 m"
"Level " + LEVEL + " m" # "Level 42.5 m"
to_string(LEVEL) + " m" # "42.5 m"
LEVEL + " m" # error
"%tT".format(START) # "08:00:00"
"%tF %<tR".format(START) # "2026-09-29 08:00"
```

`%tT` writes a time as hours, minutes and seconds, `%tF` a date, and `%tR` hours and minutes; `%<` reuses the previous argument. To show a time in the viewer's zone, convert it in the widget: `"%tT".format(toLocale(TIME))`.

A clock for dashboards

A widget's formula runs again when a variable it names changes. `now()` is not a variable, so a label that reads only `now()` does not move by itself: it changes only when something else redraws the view. The plants keep the time in a variable instead, and let a Scheduler rewrite it:

Variable	Formula	Why
`TIME`	`now()`	The current time in UTC, rewritten on every tick

Scheduler lists	Cron	Runs
`TIME`	`* * * * * ?`	Every second. `0 * * * * ?`, every minute, is enough for a clock without seconds

A label then shows it in the viewer's time zone:

```
"%tT".format(toLocale(TIME))
```

Every view that shows `TIME` redraws on each tick, and a redraw evaluates every formula of the view. Tick no more often than the display needs.

Colour by state or threshold

A widget's `styles` formula gives a record, and each of its keys becomes a style property, with `_` written as `-`: `stroke_width` is `stroke-width`. Styling widgets has the rules; the plant patterns are these:

- **One `if` per key.** Each key chooses its own value from the state.
- **Every key in every state.** Give each key a value in every state, so that no state depends on what the previous one left.
- **Colours as text.** A style takes a colour as text, such as `"#00AA44"`. The colour functions give a number, which a style cannot use: write it as text with `"#%06X".format(…)`.
- **Sizes and opacities as numbers**, such as `stroke_width: 2` or `opacity: 0.5`, not as formatted text.

```
{fill: if(PUMP_ON, "#00AA44", "#BBBBBB"), stroke: if(PUMP_ON, "#006622", "#888888"), stroke_width: 2} # {fill:
if(LEVEL > SETPOINTS.high, "#DD2222", if(LEVEL < SETPOINTS.low, "#FFAA00", "#00AA44")) # "#00AA44"
rgb(1, 0, 0) # 16711680
"#%06X".format(rgb(1, 0, 0)) # "#FF0000"
"#%06X".format(rgb(0, 0.5, 0)) # "#008000"
```

A threshold on a plant reading, with its guard, and a colour that follows the level through one of the engine's gradients:

```
{fill: if(evl(to_double(LIT_301), 0.0) > 4.2, "#DD2222", "#C4CCD8"), stroke: "#52606D", stroke_width: 1}
{fill: "#%06X".format(gradient(red2green, clamp(LIT_301_PCT / 100, 0.0, 1.0)))}
```

gradient takes the bare name of a gradient, `white2black`, `rainbow`, `black2green`, `black2red`, `red2gray2green`, `red2green` or `red2white2green`, and a value from 0 to 1. A name the engine does not provide fails when the formula runs, with `Unknown gradient 'NAME'`.

Multi-state indicators

For several boolean signals, nest the `if`s with the most urgent state first. A signal that may not exist yet goes through `evl`:

```
if(evl(P_101_TRIP, false), "#DD2222", if(PUMP_ON, "#00AA44", "#C4CCD8")) # "#00AA44"
or(PUMP_ON, COUNT > 5) # true
bit(REGISTERS[3], 0) # true
bit(REGISTERS[3], 1) # false
```

The plant form of a pump symbol adds a grey state for "no data" in front of the others:

```
{fill: if(not(evl(P_101_RUN != null, false)), "#D9DEE4", if(evl(P_101_TRIP, false), "#D62828", if(P_101_RUN, "#
```

A packed status word from a device is split with bit, counting from bit 0, the least significant: `{fill: if(bit(P_101_STATUS, 3), "#DD2222", "#C4CCD8")}`.

For a numeric state code, case is a lookup table: the code, then pairs of candidate and result, then a default. It matches only a value of the same kind, and a whole number never matches a decimal. A code read from a Modbus register is a whole number, but one that a formula computes with arithmetic, or that a dashboard's number field writes, is a decimal: convert it with `to_int`.

```
case(COUNT, 0, "#9AA5B1", 1, "#00AA44", 2, "#FFAA00", 3, "#DD2222", "#D9DEE4") # "#DD2222"
case(COUNT + 0, 3, "#DD2222", "#D9DEE4") # "#D9DEE4"
case(to_int(COUNT + 0), 3, "#DD2222", "#D9DEE4") # "#DD2222"
```

Decimals and whole numbers

A number written without a point is a whole number, and every result of `+`, `-`, `*`, `/` and `^` is a decimal (see Arithmetic). Most functions take either. Three places care:

Where	What happens	Write
`format`	`%d` refuses a decimal, `%f` a whole number	`to_int` or `to_double` first
`case`	A whole number never matches a decimal	`to_int` on the code
`clamp`, `max`, `min`	They give back the chosen argument as it was, so a whole-number limit comes back whole	Limits written as decimals

```
COUNT + 1 # 4.0
type_of(COUNT + 1) # "Double"
clamp(LEVEL, 0, 40) # 40
clamp(LEVEL, 0.0, 40.0) # 40.0
clamp(LEVEL, 0.0, 100.0) # 42.5
clamp(evl(to_double(PT_101), 0.0), 0.0, 10.0) # 0.0
```

The plant form writes every argument as a decimal: `clamp(evl(to_double(PT_101), 0.0), 0.0, 10.0)`. The result is then a decimal whichever limit applies, and a label that formats it with `%.1f` always works. A variable whose Type is `double` also stores a whole-number result as a decimal.

Names

The names the engine and the dashboards use for themselves, `ORCHESTER`, `E`, `PI`, `DEBUG`, `_VALUE_`, `_VIEW_` and `_CALLER_`, are listed in Names to avoid, with the names loops bind. Beyond those:

- **A code a formula can read** is made of the letters A to Z, digits and underscores, and does not start with a digit. `P-101` reads as `P` minus `101`.
- **`true`, `false` and `null` are lower case.** Written in capitals, `TRUE` is a name, and reads a variable of that code.
- **In a panel, parameters are replaced as text.** A parameter named like a style key, such as `FILL` or `STROKE`, is replaced inside the panel's style records too, and a parameter named `E` or `PI` hides the constant. Give parameters names that nothing else in the panel uses; see Regions and panels.

```
PI # 3.141592653589793
TRUE # error
P-101 # error
```

Spaces and line breaks

Spaces around operators are a matter of taste: a formula compiles the same, and as quickly, with or without them. A formula may also break across lines anywhere between two names, numbers or symbols, which keeps a long style record readable:

```
LEVEL+1 # 43.5
LEVEL + 1 # 43.5
LEVEL   *   2 # 85.0
1 + 2 * 3 # 7.0
```

```
{
  fill: if(PUMP_ON, "#00AA44", "#BBBBBB"),
  stroke: if(PUMP_ON, "#006622", "#888888"),
  stroke_width: 2
}
```

Next steps

- Data Orchester functions
- Errors and missing values
- Styling widgets
- Function index

This page describes Amtiri Script 5.4.1 as shipped with Data Orchester engine 6.12.1.