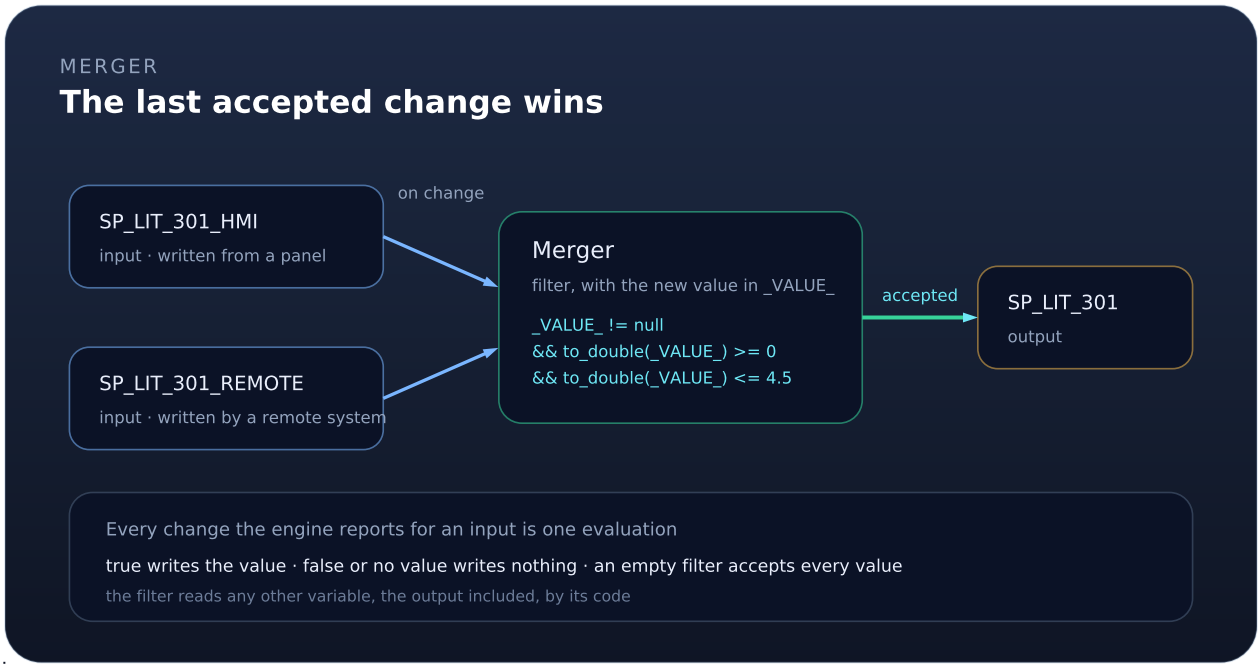
Amtiri

REFERENCE MANUAL

Logic Modules

Generated on November 25, 2026



Some modules never talk to the outside world. They work on the variables inside an instance, doing what a formula cannot. This guide covers them: what each one is for, how to configure it, and what to watch out for.

Module	What it does	Typical use
Merger	Writes to one output the value of whichever input changed last, when a filter accepts it	A setpoint written from several places; redundant meters; the latest of several messages
Delay	Writes every change of a source variable into a destination variable, a fixed time later	A condition acted on only once it has lasted; one command starting two pumps a few seconds apart; a value compared with what it was a minute ago
FIFO	Keeps the recent changes of a variable as a list of rows, each with its time	The last state changes of a pump on its dashboard; a day of module errors
PID	Steers an input to its setpoint, writing a bounded output every period	A room or a tank held at a temperature; a level held by a pump's speed
PWM	Turns a demand into on/off pulses, on for a share of every cycle	A boiler relay or a dosing pump driven by a PID's output
Kalman filter	Smooths a noisy input into an estimate, every period	A temperature from a noisy analogue input, before a PID

Merger

A formula recomputes from all of the variables it reads whenever one of them changes, but it never learns which one changed. "Take the value of whichever input changed last" therefore cannot be written as a formula. The `MERGER` module does exactly that: the engine tells it which of its inputs changed, it asks its filter about the new value, and when the filter accepts it, the value is written to the output.

Key	Setting	Type	Default	Meaning
`inputs`	Inputs	variable codes, separated by commas	empty	The variables whose changes the merger takes
`filter`	Filter	expression	empty	An expression that decides whether a change is taken. Empty means every change is taken
`output`	Output	variable code	empty	The variable that receives the accepted values

A merger also has the settings every module has: whether it is enabled, its name, and its status and message variables. They are described once, in Settings every module shares.

The filter

While the filter runs, the new value of the input that changed is available as `_VALUE_`. Every other variable is read by its own code, the output included, so a filter can compare the new value with the current output or depend on a mode variable.

The filter returns	What happens
`true`	The value is written to the output
`false`, or no value	Nothing is written
Anything else	Nothing is written, and a warning is logged
An error	Nothing is written, and a warning is logged

A filter that reads a variable which has never held a value fails, as a formula would. Give such a variable an initial value.

Rules

- **Every change is evaluated.** Each change the engine reports for an input runs the filter once. A Modbus slave that rewrites a register with the value it already held is reported too; writing that same value to the output changes nothing further.
- **The last accepted change wins.** Changes are handled on the thread that reports them. Two inputs that change at the same instant from different sources, for example a panel and a Modbus poll, may reach the output in either order.
- **Nothing is written at start.** The output keeps its value (persistent or initial) until the first accepted change.
- **The output keeps its type.** An accepted value is converted to the output's declared type. A value that cannot be converted is not written, and a warning names it.
- **Only the inputs run the merger.** A change to a variable the filter reads, but which is not an input, does not run it.

Messages

A merger reports its problems the way every module does: through its status variable and its three message variables — see Status and messages. A problem is written once and rewritten only when its text changes, and the next notification handled cleanly clears it. All texts are in English; the full list is in Module Messages.

When	Level	Text
Start: the merger does not run	`ERROR`	`No inputs`
	`ERROR`	`No output`
	`ERROR`	`Input {input} is not a variable`
	`ERROR`	`Output {output} is not a variable`
	`ERROR`	`Filter does not compile: {reason}`
Start: the merger runs	`WARNING`	`Variable {name} has the name of the bound value, so the filter cannot read it`
	`WARNING`	`Input {input} ({from}) cannot be assigned to output {output} ({to})`
While running	`WARNING`	`Filter failed for {input}: {reason}`
	`WARNING`	`Filter returned {type}, not a boolean`
	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Writing output {output} failed: {reason}`
	`WARNING`	`Notification failed: processing {input}: {reason}`

`{name}` is always `_VALUE_`, the name the filter reads the new value by: the warning means a variable of that name exists. `{from}` and `{to}` are the input's and the output's types.

A merger that cannot run never stops the engine: the other modules start as usual.

> **Warning**
>
> A merger has no loop protection. A loop that writes the same value back ends by itself, because an unchanged value goes no further. A loop that changes the value on the way, such as a formula `SP + 1` feeding one of the merger's own inputs, never ends. Check that the output does not lead back to an input.

Also keep in mind:

- **Renaming.** When an import renames variables, the inputs, the output and the names the filter reads follow the rename. A code written inside quotes in the filter, such as `"SP_LIT_301_HMI"`, is text and is not renamed.
- **List inputs.** Use inputs that hold single values. A Modbus slave's register bank is one list that the slave changes in place, so once written to the output, the output holds that same list. Read the register you need through a formula, such as `HR[0]`, and use the formula as the input.

Example. A level setpoint written from a panel and by a remote system, taken only when it is within the tank's limits:

```
Inputs   SP_LIT_301_HMI, SP_LIT_301_REMOTE
Filter   _VALUE_ != null && to_double(_VALUE_) >= 0 && to_double(_VALUE_) <= 4.5
Output   SP_LIT_301
```

Writing 2.0 from the panel sets `SP_LIT_301` to 2.0. A remote 5.0 is out of range and changes nothing; a remote 3.5 then sets it to 3.5.

Example. Two redundant flow meters, keeping only plausible readings:

```
Inputs   FIT_501_A, FIT_501_B
Filter   _VALUE_ != null && to_double(_VALUE_) >= 0 && to_double(_VALUE_) < 500
Output   FIT_501
```

Example. The latest error from several modules, ignoring a message that was cleared:

```
Inputs   PLC1_ERROR, OPCUA_ERROR, PUSH_ERROR
Filter   _VALUE_ != null && _VALUE_ != ""
Output   LAST_MODULE_ERROR
```

Example. A pump command from a local switch or from the SCADA, through the OPC UA server, where the last one given wins:

```
Inputs   CMD_LOCAL, CMD_SCADA
Filter   (empty)
Output   CMD_PUMP_1
```

Delay

A formula has no memory and no clock: it cannot say "this value, 30 seconds later". The `DELAY` module does. Each time its source changes, it takes the new value and writes it to its destination `delay` milliseconds later. The destination replays the source shifted in time: two changes 100 ms apart reach it 100 ms apart, both of them, in order.

Key	Setting	Type	Default	Meaning
`src`	Source	variable code	empty	The variable whose changes are delayed
`dst`	Destination	variable code	empty	The variable that receives each value, `delay` milliseconds later
`delay`	Delay	whole number, in milliseconds	`1000`	From 1 to 604800000 (seven days). The editor also shows it in seconds, minutes and hours

A delay also has the settings every module has, described in Settings every module shares.

Rules

- **Every change is replayed.** Each change of the source is one delayed write. The delay does not wait for the source to settle: a value that changes back and forth reaches the destination back and forth, later.
- **The value is taken when the source changes.** A later change does not alter a write already waiting. A list that a module changes in place, such as a Modbus slave's register bank, is copied as it was at that moment.
- **Nothing is written at start.** The destination keeps its value (persistent or initial) until the first delayed write.
- **The destination keeps its type.** A value its declared type cannot take is not written, and a warning names it.
- **A stop, a restart or a reload drops** every change still waiting. Changing the delay takes a reload, which starts with nothing waiting.
- **For a single delayed write from a formula**, without a module, use delayedSet.

Messages

When	Level	Text
Start: the delay does not run	`ERROR`	`No source`
	`ERROR`	`No destination`
	`ERROR`	`{field} {code} is not a variable`
	`ERROR`	`Source and destination are the same variable`
	`ERROR`	`Delay must be an integer from 1 to 604800000 ms, not {value}`
Start: the delay runs	`WARNING`	`Source {src} ({from}) cannot be assigned to destination {dst} ({to})`
While running	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Event queue full: {count} events pending, {dropped} dropped`

`{field}` is `Source` or `Destination`. The event queue is full when the engine already holds 100,000 waiting writes: the change `{dropped}` names, such as `change of LIT_301_LOW`, is lost.

Example. Act on a low level only when it was already low 30 seconds earlier, with the formula `LOW_CONFIRMED = LIT_301_LOW && LIT_301_LOW_30S`:

```
Source        LIT_301_LOW
Destination   LIT_301_LOW_30S
Delay         30000
```

Example. Start pump 2 five seconds after pump 1, from one command:

```
Source        CMD_P_101
Destination   CMD_P_102
Delay         5000
```

Example. Compare a flow with its value one minute ago, with the formula `FIT_501 - FIT_501_1MIN_AGO`:

```
Source        FIT_501
Destination   FIT_501_1MIN_AGO
Delay         60000
```

FIFO

A FIFO (first in, first out) keeps the recent changes of a variable. Each time its source changes, it adds a row with the time and the new value to the list its destination holds, and drops the oldest rows by count and by age. A dashboard table can show that list as it is.

Key	Setting	Type	Default	Meaning
`src`	Source	variable code	empty	The variable whose changes are recorded
`dst`	Destination	variable code	empty	The variable that holds the list of rows
`maxLines`	Max lines	whole number	`20`	The most rows kept, from 1 to 10000. The oldest rows are dropped first
`timeToLive`	Time to live	whole number	`3600`	The age at which a row is dropped, in `timeUnit`. `0` means no age limit
`timeUnit`	Unit	`ms`, `s` or `d`	`s`	Milliseconds, seconds or days. An hour is 3600 s

A FIFO also has the settings every module has, described in Settings every module shares.

The list

The destination holds a list of rows, the oldest first and the newest last:

```
[
  { "time": "2026-09-23T08:14:02.118Z", "value": 1 },
  { "time": "2026-09-23T08:31:47.905Z", "value": 0 },
  { "time": "2026-09-23T09:02:10.440Z", "value": 1 }
]
```

- `time` is when the change arrived, in UTC.
- `value` is what the source held then. A list is copied as it was at that moment.
- Declare the destination's type as `[{time:ZonedDateTime, value:T}]`, where `T` is the source's type, or `any`. The editor shows the type the destination needs, and a destination created from the editor gets it. An untyped destination works too.

Rules

- **One change, one row.** Each change the engine reports for the source adds a row. A write that leaves the value as it was is not a change, and adds nothing.
- **The destination changes only when the source does.** There is no timer: rows past their age stay until the next change of the source removes them, so a quiet source keeps showing older rows. A panel can hide them by comparing `time` with the current time.
- **The history survives a restart when the destination is persistent.** At start, the FIFO takes back the rows its destination holds. Anything that is not a list of rows starts an empty list, with no warning.
- **The limits apply at start too**, but nothing is written until the source next changes.
- **A persistent destination is saved on every change.** Above 1,000 rows that is a large file, and the FIFO warns about it.

Messages

When	Level	Text
Start: the FIFO does not run	`ERROR`	`No source`
	`ERROR`	`No destination`
	`ERROR`	`{field} {code} is not a variable`
	`ERROR`	`Source and destination are the same variable`
	`ERROR`	`Max lines must be an integer of at least 1, not {value}`
	`ERROR`	`Max lines must be at most 10000, not {value}`
	`ERROR`	`Time to live must be an integer of at least 0, not {value}`
	`ERROR`	`Time unit must be ms, s or d, not {value}`
Start: the FIFO runs	`WARNING`	`Destination {dst} ({type}) cannot hold a list of rows with a time and a value`
	`WARNING`	`Destination {dst} is persistent, so its entity file is rewritten with up to {n} rows on every change`
While running	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Event queue full: {count} events pending, {dropped} dropped`

Example. The last 20 state changes of a pump, for its dashboard:

```
Source          P_101_STATE
Destination     P_101_STATE_HISTORY
Max lines       20
Time to live    3600
Unit            s
```

Example. A day of module errors, however many there are, fed by the merger of the third example above:

```
Source          LAST_MODULE_ERROR
Destination     MODULE_ERROR_LOG
Max lines       500
Time to live    1
Unit            d
```

Control modules

> **Warning**
>
> A control module writes to the plant on its own clock. Data Orchester is not a safety system: keep interlocks and limits in the device, and test a loop before it drives live equipment. See Safety and responsibility.

The PID, PWM and Kalman filter modules run control loops inside the engine. They replace the formula functions `ctrlPID`, `ctrlPWM` and `ctrlKalman`, deprecated since engine 6.15.0, which kept their state in a record and stepped whenever something in their formula changed. What the three modules share:

- **Each runs on its own clock.** It steps every `period` milliseconds, and only then. A change of an input runs nothing, so a setpoint edit cannot run an extra step.
- **Inputs are functions, read when the step runs.** A PID's input and setpoint, a PWM's demand and a Kalman filter's input are written as a formula is. Most often an input is one variable's name, but it can compute its value, such as `TEMP_RAW / 10`, and a setpoint can be a plain number, such as `21.5`.
- **Outputs are plain values**: a number, or `true` and `false`. There is no state record and no `.o`.
- **An output is written only when its value changes**, so a steady loop does not fill a logger.
- **The newest output wins.** When the writes to an output fall behind, for example behind a slow link to a device, the module's newer value replaces the older one still waiting to be written, so the device always gets the current output and never a backlog of old ones.
- **An input that cannot be read**, because it fails or gives no number, skips the step, keeps the outputs as they are and is reported as a warning. An input fails, for example, while a variable it reads has no value yet. The next step that reads it clears the warning.
- **A PID's gains and limits and a Kalman filter's noises are functions too**, read at every step as the inputs are. A setting that cannot be read skips the step as an input does. A PWM's other settings are numbers.
- **An input cannot read the module's own output.** A loop that feeds on what it writes stops the module with an error at start.
- **After a restart** a PID and a Kalman filter start from the value their output holds, so a persistent output resumes the loop where it was. A PWM starts a new cycle.

Moving a formula to a module

Formula today	Module
`S = ctrlKalman(K, S, TEMP)`, read as `S.o`	Kalman filter: `input` `TEMP`, `dst` a new `double` variable, `processNoise` from `K.q`, `measurementNoise` from `K.r`, and `period` the interval the formula ran at
`S = ctrlPID(P, S, SP - PV)`, read as `S.o`	PID: `input` `PV`, `setpoint` `SP`, `action` `reverse`, the gains and limits copied from `P` (`p` to `kp`, `i` to `ki`, `d` to `kd`, `mino` to `outputMin`, `maxo` to `outputMax`)
`S = ctrlPID(P, S, PV - SP)`	The same, with `action` `direct`
`S = ctrlPID(P, S, PV - SP)` with every gain negative	`action` `reverse`, with each gain written without its minus sign: the arithmetic is the same
`S = ctrlPWM(W, S, DEMAND)`, read as `S.o`	PWM: `demand` `DEMAND`, `inputMin` and `inputMax` from `W.minin` and `W.maxin`; `period` and `minToggle` from `W.period` and `W.mintoggle`, times 1000

Then point every reference to `S.o`, in formulas, dashboards and loggers, at the module's output, and delete `S`. Three things behave differently after the move: a setpoint change gives no derivative kick, a PWM keeps its duty for the whole cycle, and a PWM's output goes off when the module closes.

PID

A PID loop reads an input and a setpoint every `period`, and writes an output that steers the input towards the setpoint. The output is the sum of three terms, held between `outputMin` and `outputMax`: a proportional term, `kp` times the error; an integral term, which grows by `ki` times the error every second; and a derivative term, `kd` times the rate at which the input changes, against it.

Key	Setting	Type	Default	Meaning
`input`	Input	expression	empty	The measured value, the process value: most often a variable's name
`setpoint`	Setpoint	expression	empty	The value the loop steers the input to: a variable's name, or a number
`dst`	Output	variable code	empty	Receives the output, held between `outputMin` and `outputMax`
`kp`	Proportional gain	expression	`1`	Output per unit of error
`ki`	Integral gain	expression, per second	`0`	How fast the integral term grows, per unit of error
`kd`	Derivative gain	expression, in seconds	`0`	Output per unit of input change per second
`outputMin`	Output minimum	expression	`0`	The lowest output, and the lowest the integral term goes
`outputMax`	Output maximum	expression	`100`	The highest output, and the highest the integral term goes
`action`	Action	`reverse` or `direct`	`reverse`	`reverse`: the error is setpoint minus input, so a low input raises the output, as heating needs. `direct`: input minus setpoint, as cooling needs
`period`	Period	whole number, in milliseconds	`1000`	From 100 to 3600000 (one hour)
`dstP`	P part	variable code	none	Optional: receives the proportional term
`dstI`	I part	variable code	none	Optional: receives the integral term
`dstD`	D part	variable code	none	Optional: receives the derivative term

A PID also has the settings every module has, described in Settings every module shares.

Settings that are functions

`input`, `setpoint`, `kp`, `ki`, `kd`, `outputMin` and `outputMax` are functions, written as a formula is. The PID compiles them when it opens and reads them at every step, so the input can be computed, the setpoint can be a plain number, and a gain or a limit can follow a variable. A plain number is the simplest function, and a variable's name the simplest input.

For example, with `input` set to `TEMP_RAW / 10`, the PID controls a reading scaled from a raw count, and with `setpoint` set to `21.5`, it needs no setpoint variable. With `outputMax` set to `BOILER_LIMIT`, an operator caps the heating from a dashboard, and with `kp` set to `if(NIGHT_MODE, 150, 300)`, the loop is gentler while `NIGHT_MODE` is true.

The variables the settings read are inputs of the PID: the Program diagram draws them, and an import renames them as it renames the variables a formula reads.

Rules

- **The derivative is taken on the input**, not on the error. A setpoint change moves the proportional and integral terms only. The first step, and the first one after a skipped step, have no derivative.
- **The integral is held inside the output limits.** A saturated loop comes off its limit as soon as the error changes sign.
- **A restart is bumpless.** At start, the integral takes the value the output holds, held inside the limits, or 0 when it holds none. With a persistent output, a restart resumes where the loop was.

- **A new `ki` does not bump the output.** The integral keeps the term it has built, and the new `ki` only changes how fast it grows from there. New limits hold the integral from the step that brings them.
- **A setting that cannot be read skips the step**: the outputs hold and a warning names the setting. That is a setting that fails, such as an input whose variable has no value yet, gives no number, gives a gain below 0, or gives an `outputMin` that is not below `outputMax`. The next good step clears the warning.
- **The input and the setpoint cannot read an output of the PID**, `dst` or a term output. That stops the PID with an error at start.
- **A constant is checked at start.** A setting that reads no variable is checked as a number was: a gain below 0, or limits with no room between them, stop the PID with an error.
- **Editing a setting takes a reload**, as for every module. A setting that reads a variable follows it at every step without one.

Term outputs

The three term outputs show what the output is made of, before the limits: `dstP` receives the proportional term, `dstI` the integral term and `dstD` the derivative term. Their sum is the output before the limits, and `dst` is that sum held between `outputMin` and `outputMax`, so when the loop saturates the sum shows by how much. Each one is written only when it names a variable, and only when its value changes.

Log them with a Logger while tuning a loop. A `dstI` that drifts to a limit is winding up against it. A `dstD` that jumps from step to step follows a noisy input, which a Kalman filter in front of the PID smooths.

Messages

When	Level	Text
Start: the PID does not run	`ERROR`	`No destination`
	`ERROR`	`{field} {code} is not a variable`
	`ERROR`	`{part} part {code} must be a variable of its own`
	`ERROR`	`No {setting}`
	`ERROR`	`{setting} does not compile: {reason}`
	`ERROR`	`{setting} cannot read the output {code}`
	`ERROR`	`{gain} must be a number of at least 0, not {value}`
	`ERROR`	`Output minimum {min} must be below output maximum {max}`
	`ERROR`	`Action must be reverse or direct, not {value}`
	`ERROR`	`Period must be an integer from {min} to 3600000 ms, not {value}`
Start: the PID runs	`WARNING`	`Destination {dst} ({type}) cannot hold a number`
While running	`WARNING`	`{setting} failed: {reason}`
	`WARNING`	`{setting} failed: {value} is not a number`
	`WARNING`	`{gain} gave {value}, below 0`
	`WARNING`	`Output minimum {min} is not below output maximum {max}`
	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Event queue full: {count} events pending, {dropped} dropped`

`{field}` is `Destination` or a part, such as `P part`. `{setting}` is `input`, `setpoint`, `kp`, `ki`, `kd`, `outputMin` or `outputMax`; in the message about reading an output it is `input` or `setpoint`. `{gain}` is `kp`, `ki` or `kd`. In the period message, `{min}` is 100 for a PID.

Example. A room held at its setpoint by a boiler, through a PWM:

```
Input           TEMP_FILTERED
Setpoint        TEMP_OBJ
Output          HEAT_DEMAND
kp              300
ki              0.5
kd              10000
Output limits   0 to 100
Action          reverse
Period          3000
```

PWM

A PWM (pulse-width modulation) turns a demand into an output that is on for a share of every cycle, for a relay that can only be on or off: a boiler, a heater, a dosing pump. At the start of each cycle it reads the demand and switches the output on, then off once that share of the cycle has passed.

Key	Setting	Type	Default	Meaning
`demand`	Demand	expression	empty	The demand, often a PID's output: most often a variable's name
`dst`	Output	variable code	empty	The on/off output, `true` or `false`
`inputMin`	Demand for 0 %	number	`0.0`	The demand that keeps the output off for the whole cycle
`inputMax`	Demand for 100 %	number	`100.0`	The demand that keeps it on for the whole cycle
`period`	Period	whole number, in milliseconds	`60000`	The cycle, from 1000 to 3600000 (one hour)
`minToggle`	Minimum toggle	whole number, in milliseconds	`0`	The shortest on or off time, below half the period

A PWM also has the settings every module has, described in Settings every module shares.

Rules

- **The share is the demand's place between its limits**, `(demand - inputMin) / (inputMax - inputMin)`, held between 0 and 1. An `inputMin` above `inputMax` inverts the demand.
- **The demand is a function, read once per cycle.** It is written as a formula is, so `max(HEAT_DEMAND, FROST_DEMAND)` drives one relay from two loops. A change waits for the next cycle, so the on time follows the demand and not the moment it changed.
- **Edges fall on time**, however slowly the demand changes.
- **No pulse and no gap is shorter than `minToggle`.** A cycle whose on time or off time would be shorter is all off or all on instead.
- **The output is written only when it changes.** At 0 % and at 100 % it is written once, not every cycle.
- **A demand that cannot be read**, because it fails or gives no number, makes the cycle an off cycle, and is reported.
- **The demand cannot read the PWM's own output.** That stops the PWM with an error at start.
- **The output goes off when the PWM closes**: when it is disabled or reloaded, or when the engine stops. The PWM writes `false` before the module that carries its output to the device closes, so the device receives it. A Stop waits for that write.

Messages

When	Level	Text
Start: the PWM does not run	`ERROR`	`No destination`
	`ERROR`	`{field} {code} is not a variable`
	`ERROR`	`No {setting}`
	`ERROR`	`{setting} does not compile: {reason}`
	`ERROR`	`{setting} cannot read the output {code}`
	`ERROR`	`{field} must be a number, not {value}`
	`ERROR`	`Input minimum and maximum must differ, both are {value}`
	`ERROR`	`Period must be an integer from {min} to 3600000 ms, not {value}`
	`ERROR`	`Minimum toggle must be from 0 to below half the period, not {value}`
Start: the PWM runs	`WARNING`	`Destination {dst} ({type}) cannot hold on/off`
While running	`WARNING`	`{setting} failed: {reason}`
	`WARNING`	`{setting} failed: {value} is not a number`
	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Event queue full: {count} events pending, {dropped} dropped`

`{setting}` is `demand`. `{field}` is `Destination` in the message about a variable, and `inputMin` or `inputMax` in the message about a number. `{min}` is 1000 for a PWM.

Example. The boiler relay of the PID example above, in 30-second cycles, never switching for less than a second:

```
Demand            HEAT_DEMAND
Output            BOILER_RELAY
Demand for 0 %    0
Demand for 100 %  100
Period            30000
Minimum toggle    1000
```

Kalman filter

A Kalman filter smooths a noisy input. Every `period` it reads the input and moves its estimate towards it, by a share that depends on how far the true value may have drifted since the last step and on how noisy the input is.

Key	Setting	Type	Default	Meaning
`input`	Input	expression	empty	The noisy value: most often a variable's name
`dst`	Output	variable code	empty	Receives the estimate
`processNoise`	Process noise	expression, per second, above 0	`0.01`	How fast the true value may drift: its variance per second
`measurementNoise`	Measurement noise	expression, above 0	`1`	How noisy the input is: its variance
`period`	Period	whole number, in milliseconds	`1000`	From 100 to 3600000 (one hour)

A Kalman filter also has the settings every module has, described in Settings every module shares.

Settings that are functions

`input`, `processNoise` and `measurementNoise` are functions, written as a formula is. The filter compiles them when it opens and reads them at every step, so the input can be computed and a noise can follow a variable. A plain number is the simplest function, and a variable's name the simplest input.

For example, with `input` set to `LEVEL_RAW * 0.01`, the filter smooths a reading scaled from a raw count, and with `measurementNoise` set to `if(PUMP_ON, 2000, 500)`, it trusts a level reading less while the pump stirs the tank.

The variables the settings read are inputs of the filter: the Program diagram draws them, and an import renames them as it renames the variables a formula reads.

Rules

- **It steps on its clock, not on a change.** A sensor polled twice that reads the same value has still been measured twice, and the estimate moves towards a steady reading at every step. Set the period to the sensor's polling interval.
- **The first input seeds the estimate**, unless the output already holds a number at start, as a persistent output does after a restart. Then that number is the seed. Either way, the estimate's variance starts at the first step's `measurementNoise`.
- **A setting that cannot be read skips the step**: the estimate holds and a warning names the setting. That is an input or a noise that fails, such as an input whose variable has no value yet, or gives no number, or a noise that gives a value not above 0. The skipped step's time still counts in the next step.
- **The input cannot read the filter's own output.** That stops the filter with an error at start.
- **A constant is checked at start.** A noise that reads no variable and is not above 0 stops the filter with an error.
- **Editing a noise takes a reload**, as for every module. A noise that reads a variable follows it at every step without one.
- **Tuning.** A larger `measurementNoise` smooths more; a larger `processNoise` follows the input faster.

Messages

When	Level	Text
Start: the filter does not run	`ERROR`	`No destination`
	`ERROR`	`{field} {code} is not a variable`
	`ERROR`	`No {setting}`
	`ERROR`	`{setting} does not compile: {reason}`
	`ERROR`	`{setting} cannot read the output {code}`
	`ERROR`	`{field} must be a number above 0, not {value}`
	`ERROR`	`Period must be an integer from {min} to 3600000 ms, not {value}`
Start: the filter runs	`WARNING`	`Destination {dst} ({type}) cannot hold a number`
While running	`WARNING`	`{setting} failed: {reason}`
	`WARNING`	`{setting} failed: {value} is not a number`
	`WARNING`	`{field} gave {value}, not above 0`
	`WARNING`	`Output {output} refused {value}: {reason}`
	`WARNING`	`Event queue full: {count} events pending, {dropped} dropped`

`{setting}` is `input`, `processNoise` or `measurementNoise`; in the message about reading an output it is `input`. `{field}` is `Destination` in the message about a variable, and `processNoise` or `measurementNoise` in the two messages about a noise's value. `{min}` is 100 for a Kalman filter.

Example. A room temperature from an analogue input polled every 3 seconds, filtered before the PID above:

```
Input              TEMP_C
Output             TEMP_FILTERED
Process noise      0.1
Measurement noise  500
Period             3000
```

Next steps

- Settings every module shares
- Module messages
- Module catalogue
- Connectivity
- Configuration reference

This page describes Data Orchester engine 6.18.0.