Amtiri

REFERENCE MANUAL

Settings every module shares

Generated on November 25, 2026

A few settings mean the same thing in every module, whatever its type: whether it runs, its name, the variables it reports its status into, and when a warning becomes an error. The engine also treats every module by the same rules when it builds, opens, notifies and stops it. This page describes both once; each module page describes only the settings of its own type.

The shared settings

Every module's editor shows **Enable** and **Name**, and a **Status and messages** section, or tab, for the rest.

Key	Setting	Type	Default	Meaning
`enabled`	Enable	boolean	`false`	Whether the module runs. A new module is disabled
`name`	Name	text	`no name`	The module's label in the editor, on the diagram and in the log. The editor also builds the codes it suggests for the status variables from it
`statusField`	Status variable	variable code	none	Receives the status as a number: -1 error, 0 off, 1 warning, 2 on
`messageInfoField`	Info message	variable code	none	Receives the latest information message
`messageWarningField`	Warning message	variable code	none	Receives the latest warning
`messageErrorField`	Error message	variable code	none	Receives the latest error
`errorAfterFailures`	Error after failures	int	`3`	How many failed cycles in a row turn a warning into an error. At least 1
`errorAfterSeconds`	Error after seconds	int	`60`	How long a lost connection stays a warning before it becomes an error. At least 1
`required`	Required to start the engine	boolean	`false`	Whether the engine may run without this module. When on, a module that cannot open stops the whole start

A module record also holds `type`, `id` and `moduleType`, which the editor writes and nobody edits: `moduleType` is the code the module catalogue lists. The editor leaves out of the record a status field left empty, a threshold at its default and `required` when it is off, so a module whose status section was never touched keeps its file unchanged.

A **disabled** module is still built, and still counts towards the licence's module limit, but it is not opened and is not told about changes. Its status is 0 and its info message reads `Module closed`.

Status and messages

Every module can report what it is doing into four variables of your own. They are ordinary variables: a dashboard can show them, a formula can react to them, a peer can be sent them, and a LOGGER can keep their history.

Setting	Variable type	Holds
Status variable	`int`	-1 error, 0 off, 1 warning, 2 on
Info message	`string`	What the module is doing when nothing is wrong
Warning message	`string`	What is wrong but not stopping it
Error message	`string`	What is stopping it

Each field's pencil opens the variable picker, and the picker offers to **create** the variable it suggests: the module's name in capital letters, with every character that cannot be in a code turned into `_`, followed by `_STATUS`, `_INFO`, `_WARNING` or `_ERROR`. A variable that already has that code, is declared with that type and has no formula is chosen instead of a second one being made.

Every field is optional, and a module with none says nothing. The status variables count towards the edition's variable limit like any other variable. Message texts are English in every language, as the module messages list them.

What turns a warning into an error

What happened	Status	The message reads
A cycle failed: a poll, a send, a request	Warning, then error once `errorAfterFailures` cycles have failed in a row	`… (1 of 3)`, `… (2 of 3)`, then `… after 3 attempts`
Something that retrying cannot fix: an unknown peer, a refused credential, a bad setting	Error at once	The message alone
A connection was lost	Warning at once, error once it has stayed lost for `errorAfterSeconds`	The connection's own message
Part of a cycle failed, or something lasts without being a failure: one register of many, a backlog	Warning	The message alone

- **Counting settles.** From the threshold on, the message stays `… after 3 attempts`, so a module that fails every second writes its variable once, not once a second.
- **Success clears.** A cycle that works clears the failures, and a connection that comes back clears the warning or error it caused.
- **A warning about one item holds.** When a device refuses one Modbus register, S7 address or EtherNet/IP tag, or an IEC 104 station reports one point invalid or of another type, the warning stays while that item stays bad, whatever the other items do, and ends when that item is read again. With several bad, it names the latest still bad. Before engine 6.18.8 the rest of the same cycle cleared it, so it showed only for a moment.
- **Recovery is announced.** A module that returns to on after something failed writes `Recovered after {n} failure(s) in {duration}` into its info message, with durations such as `45 s`, `2 min 10 s`, `1 h 5 min` or `3 d 2 h`. A module that only warned returns quietly.
- **Some types use the thresholds for more.** The Modbus slaves also measure their own time windows with them; their page says how.

How the variables are written

- **The status changes:** all four are rewritten together. The message that caused the change goes into its slot, and the slots with nothing to say are cleared.
- **The status stays the same:** only the slot the new message belongs to is rewritten, and only when its text differs from what the variable already holds.
- **A condition clears:** its slot falls back to whatever else is still true at that level, or to nothing.
- **At every Start** all four are rewritten, so a value left over from the last run, or kept by a persistent variable, cannot go on showing a status nobody has earned.

A cleared message is written as **no value**, not as an empty string. Before a text reaches a variable, anything that looks like a credential is taken out: the user and password in an address such as `opc.tcp://user:secret@host` are removed, and values written as `password=…`, `secret: …` or `token=…` become `***`. The text is also made one line, and cut at 200 characters.

When the configuration loads, the engine writes a warning to its log for a status variable that is not defined, has a formula, is persistent, is read by its own module, is also written by another module, is used for two slots, or is declared with a type that cannot hold what the slot writes.

On the diagram

Each module node lights its own icon: the icon sits on a rounded tile coloured by the status, **green** for on, **amber** for a warning, **red** for an error and **grey** for off. Hover it for the status in words and the messages behind it. The tile is not part of an exported diagram, which would otherwise claim something about the plant at the moment you pressed Export.

A tile with a **dashed** edge means the configuration on screen and the running engine disagree: you have switched the module off here, and it keeps running until the engine is started again.

Keeping a history of status and messages

There is nothing special to switch on: add the four variables to a LOGGER's `variables` list, exactly as you would a temperature. The logger writes a row whenever one of them changes, and the status, a whole number, can be charted on a dashboard through `history(...)` like any other logged value. Every change also shows in **Console** as it happens, as a line `CODE <- value`.

> **Important**
>
> Always include the status variable in the list. Clearing a message writes no value, and a logger does not record that, so the status row is what marks the end of a condition.

> **Caution**
>
> Never add a LOGGER's own status variables to its own `variables` list. It would be told about its own messages, and a logger whose database is failing would feed itself. A logger's status belongs in a different logger, or in none.

> **Warning**
>
> Logger files are never purged. Log the status alone unless you need the message texts.

What the engine does with every module

The licence

While the instance holds a licence in force, including its grace period, a module is built only when the licence admits its type and the module limit has room for it. Disabled modules count towards the limit. A module the licence refuses is not built at all, and its status says `The licence does not admit this module`. Without a licence in force every module is built, and the engine stops two hours after a Start; see How licensing works.

Building and opening

- **A record the engine cannot build**, such as one holding a setting of the wrong kind, and **a record that is built but cannot be prepared**, such as a LOGGER without a code, give `Configuration invalid: {reason}` as the configuration loads, which every Start does first. The module is left out of the plant: it is never opened, it is not told about changes, and the message stays. It does not run until the record is fixed and the engine started again. Before engine 6.18.8 a record that could not be prepared stayed in the plant, the Start opened it, and the open's own error replaced the message: `Database unavailable: {reason}` for that LOGGER.
- **A module that cannot open** goes to error with its own message, or with `Open failed: {reason}` when it has none of its own. Whatever it had opened on the way is closed again, and **the rest of the plant starts**.
- **A required module** is the exception. When a module with **Required to start the engine** on cannot open, the start is abandoned: every module is stopped and says `Engine stopped`, except the one that failed, which keeps its error. A disabled module never blocks a start, required or not.
- **Open order.** `MB-TCP-Slave` and `OPCUA-Server` open before every other module, so they listen before the modules that feed their variables start. The others open after them, in no set order.

Trying to open again

Some types fail to open because of somebody else's state: a port another program still holds, a server or a PLC that is not answering yet. When one of these fails to open, the engine tries again on its own: first after 5 seconds, then after 10, 20, 40, 80 and 160 seconds, and from then on every 5 minutes, until it opens. Stop cancels the attempts.

Tries again	Stays in error until the next Start
`MB-TCP-Slave`, `MB-SER-Slave`, `OPCUA-Client`, `OPCUA-Server`, `S7-Client`, `EIP-Client`, `IEC104-Client`, `IEC104-Server`, `CAN-Raw`, `CAN-Signals`, `NMEA-2000`, `NMEA-0183-Listener`, `NMEA-0183-Talker`	Every other type

A module that opened and then loses its device or its peer is a different case: it keeps running, reports the lost connection, and reconnects by itself. Each module page says how.

Changes and notifications

A module that reads variables is told each time one of them changes, and each module is told on its own: one that fails does not stop the others from hearing about the change. A module whose handling of a change fails shows the warning `Notification failed: {reason}` until it next handles one cleanly.

Stopping

Stop closes every module and cancels any pending attempt to open one. Each module's status becomes 0, off, and its info message `Engine stopped`.

Values a module writes

Every value a module writes into a variable goes through the engine's one write path, the same one formulas, dashboards and peers use:

- **The declared type decides.** The value is converted to the variable's declared type. One that cannot be converted is refused, the variable keeps what it held, and the log gets a `[TYPE]` line. With `orchester.typing.strict=false` in the configuration file, such a value is logged and written anyway. A variable declared with no type takes any value. `OPCUA-Client`, `OPCUA-Server` and `IEC104-Server` also warn `Variable {variable} refused the value: {reason}`, and the two servers tell the client that sent the value that it was refused.
- **Only a change is a change.** A value equal to the one the variable holds changes nothing, and nothing that depends on the variable runs.
- **Persistent variables are saved** each time the value changes.

Forwarders when a module opens

`MB-TCP-Master`, `MB-SER-Master`, `OPCUA-Client`, `S7-Client` and `EIP-Client` write variables out to a device through their forwarders, one write each time a forwarded variable changes. When one of these modules opens, it announces a change of every variable it forwards, so each one's current value is written to the device once at start. The announcement reaches everything else that watches those variables too: formulas that read them recalculate, and a LOGGER that logs one writes a row with the same value. `IEC104-Client` works differently: once per opening, when its link to the station first comes up, it sends the current value of each variable it forwards, skipping any that holds no value, and announces nothing to the rest of the plant. A link that drops and comes back sends nothing again.

The bus modules - `CAN-Raw`, `CAN-Signals`, `NMEA-2000` and `NMEA-0183-Talker` - announce nothing and send nothing because they opened. A forwarder goes out when its variable changes and, when a row gives an interval, on that clock, which starts when the module opens. A tick that finds the bus not yet up, or no address claimed yet on an NMEA 2000 backbone, sends nothing; the next one sends.

Schedules

Modules that act on a clock take a Quartz **CRON** expression: collectors of the Modbus masters, polled OPC UA rows, S7 and EtherNet/IP collectors, the IEC 104 client's interrogation and clock synchronisation, DOPULL, DBREPLICATOR and SCHEDULER. The editor shows the expression in words, and when it would run next.

An expression has six fields, **seconds first**, and an optional seventh:

```
seconds  minutes  hours  day-of-month  month  day-of-week  [year]
```

Character	Means	Example
`*`	Every value	`*` in hours: every hour
`?`	No value, in day-of-month or day-of-week	`?` in day-of-week when day-of-month is set
`-`	A range	`MON-FRI`
`,`	A list	`0,30` in minutes
`/`	Every so many, from a start	`0/15` in minutes: 0, 15, 30 and 45
`L`	The last one	`L` in day-of-month: the last day of the month
`#`	The nth weekday of the month	`6#3`: the third Friday (1 is Sunday)

Exactly one of day-of-month and day-of-week must be `?`. `0 0 6 * * *`, with a `*` in both, is refused, and so is the five-field form of other schedulers, `0 6 * * *`.

Expression	Runs
`*/10 * * * * ?`	Every 10 seconds
`0 * * * * ?`	Every minute, at second 0
`0 0/5 * * * ?`	Every 5 minutes
`0 0/15 * * * ?`	Every quarter of an hour
`0 0 * * * ?`	Every hour, on the hour
`0 5 0 * * ?`	Every day at 00:05
`0 30 7 ? * MON-FRI`	At 07:30, Monday to Friday
`0 0 0 1 * ?`	At midnight on the first day of each month
`0 0 12 L * ?`	At noon on the last day of each month

- **Time zone.** Schedules follow the plant's time zone, `application.timezone` in the configuration file: the zone formulas use, and the one the editor shows the next run in. When it is empty, they follow the Java process's zone: `server.timezone`, or the machine's own zone when that is empty too. Before engine 6.18.9 schedules always followed the Java process's zone, so where the two keys differed, hourly and daily schedules ran at another hour than the editor showed.
- **No catching up.** A schedule runs only while the engine runs: a time that passed while it was stopped is not made up later.
- **A schedule that cannot be read** stops its module from opening, with `Invalid schedule '{cron}': {reason}`. An empty one reads `No schedule given`.

Secrets from the environment

Four settings take a secret either as it is or as `env:NAME`, which names an environment variable of the service:

Where	Setting
Peer	`secret`
OPC UA Client	`password`
OPC UA Server	each user's `password`
Mail Sender	`smtpPassword`

With `env:`, the entity file, and so every export and transfer of the configuration, carries the variable's name and never its value. An environment variable the service does not have gives no value: an enabled peer whose secret is missing is loaded but cannot be used, takes no place among the peers the licence admits, and every module that uses it says `Peer '{peer}' is disabled or has no secret` (before engine 6.18.8 it was not loaded, and they said `Peer '{peer}' is not defined`); a Mail Sender signs in with an empty password, which the mail server refuses. Every other module setting is taken as it is written, `env:` included. HTTPS without a reverse proxy shows how to give the service an environment variable on each platform; the service reads its environment when it starts.

History

The engine keeps no history of its own. To keep one of any variable, a module's status included, add it to a LOGGER's `variables`: the logger writes a row at each change, and the history functions of Amtiri Script read those rows back.

Messages every module can report

These come from the engine rather than from the module. The complete list is in Module messages.

When	Level	Text
The module opened, and said nothing more specific	`INFO`	`Module opened`
The module is disabled	`INFO`	`Module closed`
The engine stopped	`INFO`	`Engine stopped`
The module is back to on after failures	`INFO`	`Recovered after {n} failure(s) in {duration}`
The module could not open, and gave no message of its own	`ERROR`	`Open failed: {reason}`
The record could not be built or prepared. The module is left out	`ERROR`	`Configuration invalid: {reason}`
The licence does not admit the module	`ERROR`	`The licence does not admit this module`
A schedule cannot be read	`ERROR`	`Invalid schedule '{cron}': {reason}`
Handling a change failed	`WARNING`	`Notification failed: {reason}`

Next steps

- Module catalogue
- Module messages
- Logger
- Configuration reference

This page describes Data Orchester engine 6.20.0.